MPI: A Message-Passing Interface Standard

Version 2.2

Message Passing Interface Forum

2009年9月4日

【日本語訳】

日本語訳にあたって

本書は“MPI: A Message-Passing Interface Standard Version 2.2”の日本語訳である．MPI Forumの意思により，基本的に英語版の規格書が唯一の公式版であるため，本書は“unofficial version”という位置付けになっている．もし日本語訳と英語原著との間で齟齬があった場合は，原著版が常に正しいことになる．

翻訳にあたり，内容的におかしいと思われた部分に関しては，翻訳時点で最新のMPI 3.1とつきあわせ，MPI 3.1で修正されている部分に関しては脚注にその旨を明記した．また，原文の意味や解釈に疑問が生じた部分に関しては全て MPI Forumに確認をとっている．同時に，翻訳作業中に見つけた原文の（LaTeXの使い方を含む）おかしな部分や表現の問題に関してはMPI Forumに報告済みである．

本翻訳以前にMPI 1.3の日本語訳が存在しており，本書の翻訳にあたり，MPI 1.3の翻訳を参考にした．それ以外にMPI 1.3の翻訳と本書の翻訳において，翻訳者の重複を含め，関係はない．

翻訳にあたり，MPI特有の用語に関しては基本的にカタカナ表記することとした．このため，「ゲット」や「プット」などいくつかの用語は聴き馴染みのないものとなっている．議論となったのは “null” の扱いである．英語の発音をカタカナ表記にすると「ナル」とすべきであるが，日本では「ヌル」が広く使われている．このため「ナル」という表記はかえって混乱を招くと考え，あえて英語表記のまま“null”とした．また，例えば「ユーザ」と「ユーザー」などの日本語表記の揺れは，基本的にインターネット検索数の多いものを採用することとした．

オリジナルは`pdflatex`を使っている関係から，CJKパッケージ（http://cjk.ffii.org）を用いて日本語化した．このため日本語特有の禁則処理など一部に問題が生じている．その点に関してはご容赦を願いたい．

本翻訳は，PCクラスタコンソーシアム (http://www.pccluster.org/) の支援により，国立研究開発法人理化学研究所計算科学研究機構の有志ボランティアが携わった．以下，翻訳に携わった名前を記す．

石川 裕　　小倉 崇浩　亀山 豊久　高木 将通
畑中 正行　堀 敦史　　山口 訓央　（50音順）

日本語訳に関する不備等に関しては `mpi-trans@pccluster.org` 宛に報告されたい．

2015年12月　　MPI 2.2 日本語訳初版

次ページ以降の全てのページは“MPI: A Message-Passing Interface Standard Version 2.2”の翻訳である．例外として，MPI 2.2 の“Errata” に関しては読者の便宜を優先し，本文に反映させてある．

本書では，Message-Passing Interface（MPI：メッセージ通信インターフェイス）標準バージョン2.2について説明する．MPI標準には1対1メッセージ通信，集団的通信，グループおよびコミュニケータの概念，プロセストポロジー，環境管理，プロセスの作成と管理，片方向通信，拡張集団通信，外部インターフェイス，入出力，それ以外のトピック，プロファイリングインターフェイスが含まれる．C言語とFortran言語の呼び出し形式も定義される．

技術的にはこのバージョンの標準は「MPI：メッセージ通信インターフェイス標準第2.1版2008年6月23日」に基づいている．MPI Forumはこの標準に7つの新しいルーチンを追加し，多数の機能強化と明確化を行った．

時系列で見ると，標準の発展はMPI-1.0（1994年6月）からMPI-1.1（1995年6月12日），MPI-1.2（1997年7月18日）へと進む段階でいくつかの点に関する明確化と追加が行われ，MPI-2文書の一部として公開された．MPI-2.0（1997年7月18日）では新機能が追加され，MPI-1.3（2008年5月30日）では歴史的経緯により文書1.1および 1.2と訂正された文書が1つの文書に統合され，MPI-2.1（2008年6月23 日）ではそれまでの文書が統合された．バージョンMPI-2.2（2009年9 月）はMPI-2.1をベースに明確化が進められ，誤りが修正され，いくつかの機能強化が行われている．

©1993, 1994, 1995, 1996, 1997, 2008, 2009 University of Tennessee, Knoxville, Tennessee. University of Tennesseeの著作権表示と本書のタイトルが記載され， University of Tennesseeの許可を得てコピーしている旨が記載されている限り，無料で本書のすべてまたは一部をコピーすることができる．

Version 2.2：2009年9月4日　この文書に含まれるのは，大部分がMPI-2.1文書を修正および明確化した内容である．一部，機能強化も行われているが，MPI-2.1の正しいプログラムはすべてそのままMPI 2.2の正しいプログラムとなっている．新機能として採用されているのは，ユーザにとって必須であり，オープンソースの実装で，既存の MPI実装への影響の小さい物のみである．

Version 2.1：2008年6月23日　この文書はそれまでの文書MPI-1.3（2008年5月30日）とMPI-2.0（1997年7月18日）を統合したものである．MPI-2.0の第4章「他章に該当しない事項」や第7章「拡張集合操作」などの章はMPI-1.3 にも組み込まれている．この文書には，MPI Forumにより収集された修正内容や明確化された内容も反映されている．

Version 1.3：2008年5月30日　この文書はそれまでの文書MPI-1.1（1995年6月12日）とMPI-1.2の章をMPI-2（1997年7月18日）に統合したものである．この文書には，MPI Forumにより収集されたMPI-1.1およびMPI-1.2の修正内容も反映されている．

Version 2.0：1997年7月18日　MPI-1.1のリリース後に，MPI Forumは再び会合を開き，間違いの修正と機能強化を検討した．MPI-2の焦点は，プロセスの作成と管理，片方向通信，拡張集団的通信，外部インターフェイス，並列入出力に置かれるようになった．「他章に該当しない事項」では別の場所に適合しない項目，特に言語の相互運用性について検討される．

Version 1.2：1997年7月18日　MPI-2 Forumは1997年7月18日に，標準の第3章“MPI-2: Extensions to the Message-Passing Interface”としてMPI-1.2を導入した．このセクションではMPI標準第1.1版の不明瞭な点が明らかにされ，細かい間違いが訂正されている．MPI-1.2の唯一の新機能は，MPI標準のどのバージョンにしたがって実装を行うかを特定するためのものである．MPI-1とMPI-1.1の違いはわずかである．MPI-1.1とMPI-1.2の違いは非常にわずかであるが，MPI-1.2とMPI-2には大きな違いがある．

Version 1.1：1995年6月　1995年5月から，Message-Passing Interface Forumは再び会合を開くようになった．その目的は1995年5月5日付けのMPI文書，すなわち下に記す第1.0版として言及されているものの間違いを訂正し，不明瞭な点を明らかにすることであった．そこでの議論の成果が本文書，第1.1版である．第1.0版からの変更点は比較的小さなものである．本文書の，すべての変更点に印をつけた版も入手できる．この段落は，変更箇所の一例である．

Version 1.0：1994年5月　Message-Passing Interface Forum（MPIF）は40を越える団体の参加を受けて 1993年1月から会合を繰り返し，メッセージ通信のためのライブラリインターフェイスの標準に関する議論と定義を行った．MPIFはいかなる公式標準制定組織からの認可も援助も受けていない．

Message-Passing Interfaceの目標は，簡単に言うと，メッセージ通信を行プログラム書くための，広く一般に使われる標準を作り出すことである．この目標を実現するには，MPIは実際的で，可搬であり，効率が良く，かつ融通の利くメッセージ通信の標準を確立しなければならない．

本文書はMessage-Passing Interface Forumの最終報告第1.0版である．本文書にはMPIインターフェイスのために提案されたすべての技術的項目が盛り込まれている．この草稿の版組はLaTeXにより1994年5月5日に行った．

MPIに関する意見がある場合，`mpi-comments@mpi-forum.org` まで送付いただきたい．送付された意見はMPI Forum委員会のメンバーに転送され，検討される．

目次

図目次

表目次

謝辞

本書はMPI Forumの委員を務めた多数の人の貢献により生まれたものである．この会議には世界各地から多数が参加した．MPI標準はこのグループの熱心で献身的な作業の結晶である．

技術的な開発はサブグループによって行われ，その内容は委員会全体で再検討された．Message-Passing Interface（MPI）の開発期間中，この尽力を多数の人々が支援した．

以下にMPI-1.0およびMPI-1.1の取りまとめの中心的な役割を果たした人員を示す．

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication
- Al Geist, Marc Snir, Steve Otto, Collective Communication
- Steve Otto, Editor
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset

上述した人以外で，MPI-1.0およびMPI-1.1の活動に積極的に参加した人々の一部を以下に示す．

Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz
Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster
Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower
Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison
Leslie Hart	Tom Haupt	Don Heller	Tom Henderson
Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga
James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe
Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley
Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer
Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison
Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson
Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith

Tennessee大学とOak Ridge国立研究所は，anonymous FTPメールサーバによるドラフトの公開を行い，当文書の配布の手段を提供した.

MPI-1標準の作業は部分的にARPAおよびNSF の助成ASC-9310330，アメリカ国立科学財団（NSF）の科学技術センター協力合意No. CCR-8809615，欧州共同体委員会のEspritプロジェクトP6643（PPPE）の支援を受けた.

MPI-1.2 および MPI-2.0:

以下にMPI-1.2およびMPI-2.0の取りまとめの中心的な役割を果たした人員を示す.

- Ewing Lusk, Convener and Meeting Chair
- Steve Huss-Lederman, Editor
- Ewing Lusk, Miscellany
- Bill Saphir, Process Creation and Management
- Marc Snir, One-Sided Communications
- Bill Gropp and Anthony Skjellum, Extended Collective Operations
- Steve Huss-Lederman, External Interfaces
- Bill Nitzberg, I/O
- Andrew Lumsdaine, Bill Saphir, and Jeff Squyres, Language Bindings
- Anthony Skjellum and Arkady Kanevsky, Real-Time

上述した人以外で，MPI-2 Forumの会議に積極的に参加した人々の一部を示す.

Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar
Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz
Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen
Ying Chen	Albert Cheng	Yong Cho	Joel Clark
Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie
Zhenqian Cui	Suresh Damodaran-Kamal		Raja Daoud
Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra
Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon
Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman
Ian Foster	Hubertus Franke	Richard Frost	Al Geist
Robert George	David Greenberg	John Hagedorn	Kei Harada
Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson
Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones
Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica
Steve Landherr	Mario Lauria	Mark Law	Juan Leon
Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams
John May	Oliver McBryan	Brian McCandless	Tyce McLarty
Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha
Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco
Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce
Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov
James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee
Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha
Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman
Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale
David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur
Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon
David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong
Dave Wright			

MPI Forumは，電子メールを通して，または個人的に貴重な意見をいただいた人々にも謝意を表明する.

以下の機関は，上記の人々がMPI-2の活動に参加できるよう，時間や旅費の面で支援した.

Argonne National Laboratory
Bolt, Beranek, and Newman
California Institute of Technology
Center for Computing Sciences

Convex Computer Corporation
Cray Research
Digital Equipment Corporation
Dolphin Interconnect Solutions, Inc.
Edinburgh Parallel Computing Centre
General Electric Company
German National Research Center for Information Technology
Hewlett-Packard
Hitachi
Hughes Aircraft Company
Intel Corporation
International Business Machines
Khoral Research
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
MPI Software Techology, Inc.
Mississippi State University
NEC Corporation
National Aeronautics and Space Administration
National Energy Research Scientific Computing Center
National Institute of Standards and Technology
National Oceanic and Atmospheric Adminstration
Oak Ridge National Laboratory
Ohio State University
PALLAS GmbH
Pacific Northwest National Laboratory
Pratt & Whitney
San Diego Supercomputer Center
Sanders, A Lockheed-Martin Company
Sandia National Laboratories
Schlumberger
Scientific Computing Associates, Inc.
Silicon Graphics Incorporated
Sky Computers
Sun Microsystems Computer Corporation
Syracuse University
The MITRE Corporation
Thinking Machines Corporation
United States Navy
University of Colorado

University of Denver
University of Houston
University of Illinois
University of Maryland
University of Notre Dame
University of San Fransisco
University of Stuttgart Computing Center
University of Wisconsin

MPI-2は非常に厳しい予算の下で運営された（実際，最初の会合の案内が出された時点では予算はまったくなかった）．多くの機関が，MPI Forumのメンバーの活動や旅費を援助する形で，MPI-2の活動を支援した．直接的な支援は，NSFおよびDARPAの米国学術関係者の旅費に関するNSF契約CDA-9115428 と，Espritのヨーロッパ参加者のためのプロジェクトHPC標準（21111）により行われた．

MPI-1.3 および MPI-2.1:

統合された文書の編集とまとめ作業は以下の人員が行った．

- Richard Graham, Convener and Meeting Chair
- Jack Dongarra, Steering Committee
- Al Geist, Steering Committee
- Bill Gropp, Steering Committee
- Rainer Keller, Merge of MPI-1.3
- Andrew Lumsdaine, Steering Committee
- Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)
- Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)

MPI-2.1の整合性を保つため，全章が再検討された．必要な修正は以下の人員が行った．

- Bill Gropp, Frontmatter, Introduction, and Bibliography
- Richard Graham, Point-to-Point Communication
- Adam Moody, Collective Communication

- Richard Treumann, Groups, Contexts, and Communicators
- Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communications
- George Bosilca, Environmental Management
- David Solt, Process Creation and Management
- Bronis R. de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O
- Jeffrey M. Squyres, Language Bindings and MPI 2.1 Secretary
- Rolf Rabenseifner, Deprecated Functions and Annex Change-Log
- Alexander Supalov and Denis Nagorny, Annex Language Bindings

上述した人以外で，MPI-2 Forumの会議に積極的に参加した人々，電子メールによる修正項目の協議に参加した人々の一部を示す.

Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
Richard Barrett	Christian Bell	Robert Blackmore
Gil Bloch	Ron Brightwell	Jeffrey Brown
Darius Buntinas	Jonathan Carter	Nathan DeBardeleben
Terry Dontje	Gabor Dozsa	Edric Ellis
Karl Feind	Edgar Gabriel	Patrick Geoffray
David Gingold	Dave Goodell	Erez Haba
Robert Harrison	Thomas Herault	Steve Hodson
Torsten Hoefler	Joshua Hursey	Yann Kalemkarian
Matthew Koop	Quincey Koziol	Sameer Kumar
Miron Livny	Kannan Narasimhan	Mark Pagel
Avneesh Pant	Steve Poole	Howard Pritchard
Craig Rasmussen	Hubert Ritzdorf	Rob Ross
Tony Skjellum	Brian Smith	Vinod Tipparaju
Jesper Larsson Träff	Keith Underwood	

MPI Forumは，電子メールを通して，または個人的に貴重な意見をいただいた人々にも謝意を表明する.

以下の機関は，上記の人々がMPI-2の活動に参加できるよう，時間や旅費の面で支援した.

Argonne National Laboratory
Bull

Cisco Systems, Inc.
Cray Inc.
The HDF Group
Hewlett-Packard
IBM T.J. Watson Research
Indiana University
Institut National de Recherche en Informatique et Automatique (INRIA)
Intel Corporation
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
Mathworks
Mellanox Technologies
Microsoft
Myricom
NEC Laboratories Europe, NEC Europe Ltd.
Oak Ridge National Laboratory
Ohio State University
Pacific Northwest National Laboratory
QLogic Corporation
Sandia National Laboratories
SiCortex
Silicon Graphics Incorporated
Sun Microsystems, Inc.
University of Alabama at Birmingham
University of Houston
University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Wisconsin

MPI Forumの会合の資金の一部は，アメリカ国立科学財団の賞金 #CCF-0816909によって支援された.
た，HDFグループが米国の研究者の旅費を援助した.

MPI-2.2:

MPI-2.2の整合性を保つため，全章が再検討された．必要な修正は以下の人員が行った

- William Gropp, Frontmatter, Introduction, and Bibliography; MPI 2.2 chair.

- Richard Graham, Point-to-Point Communication and Datatypes
- Adam Moody, Collective Communication
- Torsten Hoefler, Collective Communication and Process Topologies
- Richard Treumann, Groups, Contexts, and Communicators
- Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications
- George Bosilca, Datatypes and Environmental Management
- David Solt, Process Creation and Management
- Bronis R. de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O
- Jeffrey M. Squyres, Language Bindings and MPI 2.2 Secretary
- Rolf Rabenseifner, Deprecated Functions, Annex Change-Log, and Annex Language Bindings
- Alexander Supalov, Annex Language Bindings

上述した人以外で，MPI-2 Forumの会議に積極的に参加した人々，電子メールによる修正項目の協議に参加した人々の一部を示す.

Pavan Balaji	Purushotham V. Bangalore	Brian Barrett
Richard Barrett	Christian Bell	Robert Blackmore
Gil Bloch	Ron Brightwell	Greg Bronevetsky
Jeff Brown	Darius Buntinas	Jonathan Carter
Nathan DeBardeleben	Terry Dontje	Gabor Dozsa
Edric Ellis	Karl Feind	Edgar Gabriel
Patrick Geoffray	Johann George	David Gingold
David Goodell	Erez Haba	Robert Harrison
Thomas Herault	Marc-André Hermanns	Steve Hodson
Joshua Hursey	Yutaka Ishikawa	Bin Jia
Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian
Ranier Keller	Matthew Koop	Quincey Koziol
Manojkumar Krishnan	Sameer Kumar	Miron Livny
Andrew Lumsdaine	Miao Luo	Ewing Lusk
Timothy I. Mattox	Kannan Narasimhan	Mark Pagel
Avneesh Pant	Steve Poole	Howard Pritchard
Craig Rasmussen	Hubert Ritzdorf	Rob Ross
Martin Schulz	Pavel Shamis	Galen Shipman
Christian Siebert	Anthony Skjellum	Brian Smith
Naoki Sueyasu	Vinod Tipparaju	Keith Underwood
Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu

MPI Forumは，電子メールを通して，または個人的に貴重な意見をいただいた人々にも謝意を表明する．

以下の機関は，上記の人々がMPI-2.2の活動に参加できるよう，時間や旅費の面で支援した．

Argonne National Laboratory
Auburn University
Bull
Cisco Systems, Inc.
Cray Inc.
Forschungszentrum Jülich
Fujitsu
The HDF Group
Hewlett-Packard
International Business Machines
Indiana University
Institut National de Recherche en Informatique et Automatique (INRIA)
Institute for Advanced Science & Engineering Corporation

Intel Corporation
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
Mathworks
Mellanox Technologies
Microsoft
Myricom
NEC Corporation
Oak Ridge National Laboratory
Ohio State University
Pacific Northwest National Laboratory
QLogic Corporation
RunTime Computing Solutions, LLC
Sandia National Laboratories
SiCortex, Inc.
Silicon Graphics Inc.
Sun Microsystems, Inc.
Tokyo Institute of Technology
University of Alabama at Birmingham
University of Houston
University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Tokyo
University of Wisconsin

MPI Forumの会合の資金の一部は，アメリカ国立科学財団の賞金 #CCF-0816909によって支援された.
また，HDFグループが米国の研究者の旅費を援助した.

第1章

MPIの概要

1.1 概要と目標

メッセージ通信インターフェイス（MPI：Message-Passing Interface）とは，メッセージ通信ライブラリインターフェイスの仕様である．この定義の全ての部分が重要である．第一に，MPIアドレスはメッセージ通信の並列プログラミングモデルで，データは各プロセスの動作が共同して1つのプロセスのアドレス空間から別のプロセスのアドレス空間に移動する．（従来のメッセージ通信モデルに集団操作，リモートメモリアクセス操作，動的プロセス作成，並列入出力が追加されている．）MPIは実装ではなく仕様であるため，MPIには多数の実装方法がある．この仕様はライブラリインターフェイスのためのものであるため，MPIは言語ではなく，全てのMPI操作は（C言語，C++言語，Fortran 77言語，Fortran 95言語の）MPI標準の一部である適切な言語のビルドに従って関数，サブルーチン，メソッドとして表現される．この標準は並列計算ベンダー，コンピュータ科学者，アプリケーション開発者のコミュニティによるオープンなプロセスを通して定義された．次のいくつかのセクションでMPI開発の歴史を概説する．

メッセージ通信の標準を確立することの主な利点は，可搬性が向上し，使いやすくなることである．下位レベルのメッセージ通信ルーチンの上に上位レベルの ルーチンやアブストラクションが構築される分散メモリ通信環境では，標準化の利点が特に明確となる．さらに，ここに提案するようなメッセージ通信の標準の定義は，効率的な実装が可能な，明確に定義された ルーチンの基本セットをベンダーに提供し，場合によってはハードウェアのサポートを提供するため，スケーラビリティを向上させることができる．

簡潔に記述されたMPIの目標は，メッセージ通信プログラムを記述するための広く使用される標準を開発することである．そのため，インターフェイスではメッセージ通信のための実践的で，効率的で，可搬性と柔軟性に優れた標準を確立する必要がある．

全ての目標のリストを以下に示す．

- アプリケーションプログラムのインターフェイスを設計する．（コンパイラやシステム 実装ライブラリ用とは限らない）

- 効率のよい通信を可能にする．メモリ間コピーを避け，計算と通信を並行して進めることができ，通信コプロセッサがあれば処理の一部をそれに任せるようなもの．
- 異種間環境でも使用できる処理系に備える．
- C言語，C++言語，Fortran 77言語，Fortran 95言語のための使い易い呼び出し形式を可能にする．
- 信頼できる通信インターフェイスを想定する．ユーザは通信障害に対処する必要がない．そのような障害は下位の通信サブシステムが処理する．
- 多くのベンダーのプラットフォーム上に実装でき，その際，下位の通信およびシステムソフトウェアに大きな変更を加えずに済むインターフェイスを定義する．
- このインターフェイスの意味は言語に依存しないようにする．
- このインターフェイスは，マルチスレッド環境への対応が可能なように設計する．

1.2 MPI-1.0の背景

MPI の設計に際しては，いくつかの既存のメッセージ通信システムが持つもっとも魅力的な機能の数々を採りいれるよう努めた．既存のシステムの中から1 つを選択し，それを標準として採用するというやり方は採らなかった．MPI は， IBMのT. J. Watson Research Center文献[1, 2]での研究や，IntelのNX/2J[38]，Express文献[12]，nCUBEの Vertex[34]，p4[7, 8]， PARMACS[5, 9]から多大な影響を受けてきた．他に，Zipcode[40, 41]，Chimp[16, 17]， PVM[4, 14]，Chameleon[25]， PICL[24]などからも重要な貢献があった．

MPI の標準化の事業には，主に米国，ヨーロッパの40の組織から約60人が参加した．並列コンピュータの主要なベンダーのほとんどがMPI に参加し，大学，国立研究所，そして企業からの研究者もこれに加わった．標準化の作業が始まったのは，「分散メモリ環境におけるメッセージ通信のための標準に関するワークショップ[48]Workshop on Standards for Message-Passing in a Distributed Memory Environment）」においてである．このワークショップは1992年4月20日～30日，米Virginia州 Williamsburgで「並列コンピューティング研究センター」（Center for Research on Parallel Computing）の主催により開催された．このワークショップにおいて，標準的なメッセージ通信インターフェイスの中心となる基本機能が討議され，標準化のプロセスを継続するため，ワーキンググループが設立された．

1992年11月に，Dongarra，Hempel，Hey，Walkerにより，**MPI1**として知られる予備的な草案が提示され，修正版が1993年4月に完成した[15]．MPI1には，Williamsburgのワークショップにおいて，メッセージ通信標準に必要と指摘された主要な機能が具体化されていた．MPI1はもともと，論議を促進するためのたたき台だったので，主に1対1通信に焦点を当てている．MPI1は，標準化における多くの重要な課題を明らかにしたが，こ

れには集団通信ルーチンは含まれておらず，またマルチスレッド環境には対応していなかった．

1992年11月に，MPI ワーキンググループのミーティングがMinneapolisで開かれ，そこで標準化をより形式に則ったプロセスにし，High Performance Fortran Forumの手順と組織構成をほぼそのまま採用することが決定された．標準の主要な構成分野ごとに小委員会が作られ，それぞれに電子メール討議サービスが設立された．さらに，1993年秋までにMPI標準の草案を作成するという目標が設定された．この目標を達成するため，MPIワーキンググループは， 1993年の最初の9カ月間，6週間ごとに2日間の会合を開き，1993年11月の会議でMPI標準の草案を発表するに至った．これらの会合と電子メールによる議論がMPIフォーラムを構成したものであり，その会員資格はハイパフォーマンスコンピューティングに関わる全ての人に対して常に開放されていた．

1.3 MPI-1.1, MPI-1.2, および MPI-2.0の背景

1995年3月から，MPIフォーラムは最初のMPI標準の文書に対する修正と機能強化を協議するための会合を開始した [21]．これらの協議の最初の成果がMPI仕様のバージョン1.1となり，1995年6月にリリースされた[22]（正式なMPI 文書のリリースについては`http://www.mpi-forum.org`を参照）．その時点で，次の5つの点が活動の中心となった．

1. MPI-1.1 文書のためのさらなる修正と明確化
2. 機能のタイプに大きな変化のないMPI-1.1 への追加事項（新しいデータ型のコンストラクタ，言語の相互運用性など）
3. 「MPI-2の機能」と考えられるまったく新しいタイプの機能（動的プロセス，単方向通信，並列入出力など）
4. Fortran 90言語およびC++言語のための呼び出し形式．MPI-2では MPI-1 およびMPI-2 機能のためのC++言語の呼び出し形式が指定され， Fortran 90言語の問題に対処するための，MPI-1 およびMPI-2 機能のためのFortran 77言語の呼び出し形式の機能強化が指定されている．
5. MPI プロセスおよびフレームワークが有益であると考えられるが，標準化の前にさらなる協議と経験が必要な領域の協議（共有メモリマシンのゼロコピーの意味，リアルタイムの仕様など）

修正点と明確化された内容（上記のリストの1に該当する項目）は，MPI-2文書の第3章「MPI第1.2版」に記載されている．この章には，バージョン番号を特定するための関数も記載されている．MPI-1.1 への追加事項（上記のリストの2，3，4に該当する項目）はMPI-2 文書の残りの章に記載され，MPI-2の仕様を構成している．上記のリストの5に該当する項目は別の文書“MPI Journal of Development” JOD）に移されており，MPI-2標準の記載からは外されている．

この構成により，利用者や実装者は特定の実装において準拠すべきMPI のレベルを理解しやすくなっている．

- MPI-1の準拠とは，MPI-1.3への準拠を意味する．これは準拠の有益なレベルである．これは，MPI-2文書の第3章に記載されたMPI-1.1関数の動作の明確化に準拠して実装することを意味する．実装の一部で， MPI-2.1 に準拠するよう変更する必要がある場合がある．
- MPI-2の準拠とは，MPI-2.1 の全てへの準拠を意味する．
- “MPI Journal of Development”はMPI標準から独立したものである．

前方互換性が保たれていることは強調されなければならない．つまり，正しい MPI-1.1 プログラムは，MPI-1.3 でも MPI-2.1 でも正しく， MPI-1.3　プログラムは MPI-2.1 でも正しいプログラムである．

1.4　MPI-1.3および MPI-2.1の背景

MPI-2.0のリリース後，MPIフォーラムは両方の標準の文書（MPI-1.1およびMPI-2.0）に対する修正と明確化の作業を継続した．1998年10月12日に短い文書“Errata for MPI-2.0”がリリースされた．2001年7月5日にMPI-2.0の修正と明確化の最初の項目一覧が公表され，2番目の項目一覧のための投票は2002年5月22日に行われた．どちらの投票も電子による方法で行われた．2002年5月15日に両方の項目一覧が1つの文書“Errata for MPI-2”にまとめられた．この修正プロセスはその後中断されたが，フォーラムと電子メールでの参加者は明確化の新しい要求に対して活動を継続した．

MPIフォーラムの定期的な活動はBonnでのEuroPVM/MPI’06，Parisでの EuroPVM/MPI’07，RenoでのSC’07の3つの会合で再開された．2007年12月，運営委員会により新しいMPIフォーラムの会合の組織化が開始され，8週間ごとに定期的に開かれることになった．2008年1月14〜16日にChicagoで開かれた会合で，MPIフォーラムは各バージョンのMPI標準について，既存のMPI文書と今後のMPI文書を統合して1つの文書にすることを決定した．　技術的／歴史的経緯により，このシリーズはMPI-1.3 から開始されることになった．追加の項目一覧3および4では，1995年に始まった修正リストの古い問題から，昨年の修正リストの最新の問題までを解決している．全ての文書（MPI-1.1，MPI-2，Errata for MPI-1.1（1998年10月12日）， MPI-2.1 Ballots 1-4）が1つの草稿に統合された後，各章について，章の執筆者とレビューチームが定義された．　彼らはMPI-2.1 文書の整合性が保証されるよう文書を完成させた．最終的なMPI-2.1標準の文書は2008年6月に完成され，EuroPVM/MPI’08の直前に， 2008年9月のDublinでの会合において2回目の投票によりリリースされた．MPIフォーラムの現在の主な活動内容はMPI-3の準備である．

1.5 MPI-2.2の背景

MPI-2.2はMPI-2.1標準へのマイナーアップデートである．このバージョンでは，MPI-2.1標準で未修正となっていた誤りやあいまいな点への対応のほか，以下の基準を満たすMPI-2.1への少数の機能強化が行われた．

- 適切なMPI-2.1のプログラムは適切なMPI-2.2のプログラムでもある．
- 機能強化は利用者にとって大きな利点をもたらすものでなければならない．
- 機能強化は実装のために労力を要するものであってはならない．そのため，このような変更は全てオープンソースの実装によって行われなければならない．

MPI-2.2の協議はMPI-3の協議と並行して行われ，MPI-2.2のために提案された機能強化が後からMPI-3に移行した場合もある．

1.6 この標準を利用する人々

この標準が意図する利用者は，可搬なメッセージ通信プログラムをFortran言語，C言語，C++言語で記述しようとする全ての人である．この中には，個々のアプリケーションのプログラマや，並列マシンで動作するよう設計されたソフトウェアの開発者，環境やツールの作成者を含む．この標準が，これらの広範囲に渡る人々にとって魅力的なものであるためには，初歩のユーザには簡単で使いやすいインターフェイスを提供する一方で，先進的マシンに備わる高性能のメッセージ通信操作を意味的に妨げないものでなければならない．

1.7 この標準の実現対象となるプラットフォーム

メッセージ通信パラダイムの魅力は，少なくとも部分的には，その広い可搬性から生じている．この方法で表現されたプログラムは，分散メモリのマルチプロセッサ，ワークステーションのネットワーク，そしてこれらの組み合わせの上でも動作しうる．さらに，マルチコアプロセッサ用，ハイブリッドアーキテクチャ用などの共有メモリによる実現も可能である．このパラダイムは，共有メモリと分散メモリの観点を組み合わせたアーキテクチャが出現しても，またネットワークが高速化しても時代遅れにはならない．したがって，この標準を実装することが可能であり有用であるマシンの範囲は非常に広いはずであり，そこには通信ネットワークによって接続された他の（並列機か逐次機かを問わない）マシンの集まりで構成されるような「マシン群」まで含まれる．

このインターフェイスは，まったく一般的なMIMDプログラムでの使用にも，より制限されたSPMDのスタイルで書かれたプログラムでの使用にも同じように適している．MPIには，専用のプロセッサ間通信ハードウェアを備えるスケーラブルな並列コンピュータでの性能を高めることを狙いとした多くの機能がある．したがって，このようなマシンには高性能な専用MPI処理系が提供されることが期待される．また一方では，

Unixプロセッサ間通信プロトコルを用いるMPI処理系は，ワークステーションクラスタやワークステーションの異機種間ネットワークの間での可搬性を提供する.

1.8 標準に含まれるもの

標準は以下の事項を含む.

- 1対1通信
- データ型
- 集団操作
- プロセスグループ
- 通信コンテクスト
- プロセストポロジー
- 環境管理と問い合わせ
- Infoオブジェクト
- プロセスの作成と管理
- 片方向通信
- 外部インターフェイス
- 並列ファイル入出力
- Fortran言語，C言語，C++言語の呼び出し形式
- プロファイリングインターフェイス

1.9 標準に含まれないもの

標準は以下の事項を指定しない.

- 現在標準となっているもの以上のオペレーティングシステムのサポートを必要とする操作. 例えば割込駆動の受信，リモート実行，アクティブメッセージなど.
- プログラム構築ツール
- デバッグ機能

考慮された結果，この標準には含まれなかった機能が多数ある．これはさまざまな理由によるが，そのうちの1つは，標準の仕上げに際してつきまとう時間的制約によるものである．含まれていない機能は，特定の実装により拡張機能としていつでも提供され得る．おそらくMPIの将来のバージョンでは，これらの事項のうちいくつかについて扱うことになるであろう．

1.10 この文書の構成

以下は，この文書の残りの章についてのリストで，それぞれに簡単な説明を付けた．

- 第2章，**MPIの用語と規則**．この章では，このMPI文書全体で使用される表記上の用語と規則について説明する．
- 第3章，**1対1通信**．この章では，MPIの基本的な2者間通信に関するサブセットについて定義する．*Send*と*receive*はここで扱う．加えて，基本的な通信を強力で効率のよいものにするよう設計された，多くの関連する関数についても述べる．
- 第4章，**データ型**．この章では，データのレイアウトを記述する方法を定義する．例えば，メッセージの送信または受信バッファとして使用可能な，メモリ内の構造体配列などがある．
- 第5章，**集団的通信**．この章では，プロセスグループ内での集団的通信動作を定義する．良く知られている例としては，あるプロセスのグループ（必ずしも全プロセスではない）内でのバリア同期およびブロードキャストがある．MPI-2では，集団通信の意味はグループ間コミュニケータも含むよう拡張された．2つの新しい集団操作も追加されている．
- 第6章，**グループ，コンテクスト，コミュニケータ，キャッシング**．この章では，プロセスのグループを形成して，それらを操作する方法，固有の通信コンテクストを取得する方法，その2つを結合してコミュニケータを作る方法を説明する．
- 第7章，**プロセストポロジー**．この章では，プロセスグループ（線形順序集合）を多次元グリッドのような，より複雑なトポロジー構造に対応づけるときに役に立つよう用意された一群のユーティリティ関数について説明する．
- 第8章，**MPI環境管理**．この章では，プログラマが現在のMPI環境について管理，問い合わせをおこなう方法について説明する．これらをおこなう関数は，正しい頑健なプログラムの記述に必要であり，とりわけ，非常にポータブルなメッセージ通信プログラムを構築するために重要である．
- 第9章，**Infoオブジェクト**．この章では，いくつかのMPIルーチンの入力として使用される不可視オブジェクトを定義する．

- 第10章，**プロセスの作成と管理**．この章では，プロセスの作成が可能なルーチンを定義する．
- 第11章，**片方向通信**．この章では，1つのプロセスで完了させることができる通信ルーチンを定義する．ここには，共有メモリ操作（プット/ゲット）とリモートアキュムレート操作が含まれる．
- 第12章，**外部インターフェイス**．この章では，開発者がMPI 上で使用できるよう設計されたルーチンを定義する．ここには，汎用の要求， MPI不可視オブジェクトのデコード用ルーチン，スレッドが含まれる．
- 第13章，**入出力**．この章では，並列入出力のためのMPIのサポートを定義する．
- 第14章，**プロファイリングインターフェイス**．この章では，MPI処理系がサポートしなければならない，簡単な，名前のシフトの規則について説明する．このシフトの狙いの1つは，性能のプロファイリングのための呼び出しを， MPIのソースコードに手を加える必要なしにMPIに入れることができるようにすることである．名前シフトは，単なるインターフェイスに過ぎず，実際にプロファイリングで何をどうするかということについては何も指示しない．実際，名前シフトを他の目的で有効に使用することも可能である．
- 第15章，**廃止された関数**．この章では，参照用に保存されたルーチンを説明する．ただし，これらの関数は今後のバージョンの標準から削除される可能性があるため，使用はお勧めできない．
- 第16章，**言語別の呼び出し形式**．この章では，C++言語の呼び出し形式を説明し，Fortran言語の問題点を示し，C言語，C++言語， Fortran言語の間の相互運用性について説明する．

付録は以下のとおりである．

- 付録A，**言語呼び出し形式要約**．ここでは，C言語， C++言語，Fortran言語における具体的な構文を，全てのMPI関数，定数，型について示す．
- 付録B，**変更履歴**．ここでは，前のバージョンの標準からの主な変更点の概要を示す．
- **索引のページ**．例，定数，定義済みのハンドル，コールバックルーチンのプロトタイプ，全てのMPI関数の場所を示す．

MPIには，それぞれのMPI処理系の相互運用性を促進するための各種インターフェイスが用意されている．この中には，入出力用，また MPI_PACK_EXTERNALおよびMPI_UNPACK_EXTERNAL用の標準的なデータ表現がある．相互運用性を可能にするこれらのインターフェイスの実際の呼び出し形式の定義は，本書の対象範囲外である．

MPIフォーラムで協議された内容は別の文書に含まれ，MPI標準には含まれない．これらは，有益な協議内容を保存し，さらなる活動の開始点とするため，“Journal of Development” (JOD) の一部に組み込まれている．JODの章の内容を以下に示す．

- 第2章，**独立した生成プロセス（Spawning Independent Processes）**．この章では，動的プロセス管理の一部の要素，特に，生成プロセスが通信の対象としないプロセスの管理について説明している．これは，フォーラムで詳細に協議されながら最終的にはMPI標準に組み込まないことが決定された機能である．
- 第3章，**スレッドとMPI（Threads and MPI）**．この章では，マルチスレッド環境においてMPI処理系とスレッドライブラリの間で予想される相互作用の一部を説明している．
- 第4章，**コミュニケータID（Communicator ID）**．この章では，コミュニケータ識別子を提供するためのアプローチについて説明している．
- 第5章，**他章に該当しない事項（Miscellany）**．この章では，MPI JOD のその他のトピック，特に共有メモリ環境で使用できるシングルコピー ルーチンと新しいデータ型コンストラクタについて説明している．
- 第6章，**完全なFortran 90言語のインターフェイスに向けて（Toward a Full Fortran 90 Interface）**．この章では，より詳細なFortran 90言語のインターフェイスを提供するためのアプローチについて説明している．
- 第7章，**スプリット集団通信（Split Collective Communication）**．この章では，ノンブロッキング集団操作の仕様について説明している．
- 第8章，**リアルタイムMPI（Real-Time MPI）**．この章では，リアルタイム処理のためのMPIサポートについて説明している．

第2章

MPIの用語と規則

この章では，MPI文書全体で使用される表記上の用語と規則，行われた選択のいくつか，およびそれらの選択の背後にある根拠について説明する．基本的にはMPI-1の用途と規則の章と同様だが，大小いくつかの違いがある．大きな違いとしては，命名規則，一部の意味の定義，ファイルオブジェクト，Fortran 90言語とFortran 77言語，C++言語，プロセス，シグナルとの相互作用がある．

2.1 文書の表記

根拠 インターフェイス仕様の中でおこなわれた設計上の選択の根拠は，この文書全体を通して，この形式で提示される．読者によっては，これらの節をとばすこともできるし，インターフェイスの設計に興味のある読者は念入りに読むこともできる． （**根拠の終わり**）

ユーザへのアドバイス ユーザに訴えたいことや使用法の説明については，この文書全体を通して，この形式で提示される．読者によっては，これらの節はとばすこともできるし，MPIのプログラミングに興味のある読者は念入りに読むこともできる． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 主に実装者に対する注釈となることについては，この文書全体を通して，この形式で提示される．読者によっては，これらの節はとばすこともできるし， MPIの実装に興味のある読者は念入りに読むこともできる． （**実装者へのアドバイス終わり**）

2.2 命名規則

多くの場合，C言語関数のMPIの名前はMPI_Class_action_subset いう形式で示される．この規則はMPI-1に由来するものである．MPI-2以降， MPI関数の名前の標準化は以下の規則に従って進められている．C++言語の 呼び出し形式が特にこの規則に従っている（22ページの第2.6.4節を参照）．

1. C言語では，特定のタイプのMPIオブジェクトに関連するルーチンはすべてMPI_Class_action_subsetという形式にする必要があり，サブセットがない場合はMPI_Class_actionという形式にする必要がある．Fortran言語では，特定のタイプのMPIオブジェクトに関連するルーチンはすべてMPI_CLASS_ACTION_SUBSETという形式にする必要があり，サブセットがない場合はMPI_CLASS_ACTIONという形式にする必要がある．C言語およびFortran言語では，Classを定義するためにC++言語の用語を使用する．C++言語では，ルーチンはクラス上のメソッドであり，名前はMPI::Class::Action_subsetとなる．ルーチンが特定のクラスに関連しているが，オブジェクトメソッドとしては意味をなさない場合，クラスの静的メンバ関数となる．

2. ルーチンがクラスに関連していない場合，C++言語では名前をMPI_Action_subsetという形式にする必要があり，Fortran言語ではMPI_ACTION_SUBSETという形式にする必要がある．C++言語では，MPIの名前空間MPI::Action_subsetで範囲指定する必要がある．

3. 特定のアクションの名前が標準化されている．特に，**Create**は新しいオブジェクトを作成し，**Get**はオブジェクトに関する情報を取得し，**Set**はこの情報を設定し，**Delete**は情報を削除し，**Is**はオブジェクトに特定の属性があるかどうかを問い合わせる．

C言語およびFortran言語のMPI関数の名前（MPI-1プロセスで定義された）の中に，これらの規則に違反しているものがある．一般的な例外はルーチンでの クラスの名前の欠落と，想定される場所での**アクション**の欠落である．

MPIの識別子は30文字までに限定される（プロファイリングインターフェイスでは31文字）．この制限は，一部のコンパイルシステムでの制限の超過を避けるためのものである．

2.3 手続きの仕様

MPI手続きの仕様は，言語に依存しない表記を使用して記述する．手続き呼び出しの引数には，IN，OUT，INOUTの区別を示す．これらの意味は以下の通りである．

- IN: 呼び出しにより使用されるが，更新されることはない．
- OUT: 呼び出しにより更新されることがある．
- INOUT: 呼び出しにより使用，更新の両方が行われる．

特別な場合が1つある．引数が不可視オブジェクトのハンドル（これらの用語については第2.5.1節で定義）であって，そのオブジェクトがその手続き呼び出しで更新される場合，その引数はINOUTまたはOUTと示す．ハンドル自身は修正されないにも関わらずこのように示す．つまり，INOUTまたはOUT属性は，ハンドルの参照するものが更新され

るということを示すために使うことがある．そのため，C++言語では，IN引数は通常，constオブジェクトの参照またはポインタとなる．

> **根拠** MPIの定義は，INOUT引数の使用をできる限り避けている．このような使い方は，特にスカラ引数の場合，間違いの元になるからである． （**根拠の終わり**）

MPIでのIN，OUT，INOUTの使用は，ユーザに引数の使用方法を示すことを目的としているが，全ての言語に直接変換できるほど厳密に分類されていない（Fortran 90言語のINTENTや，C言語のconstなど）．例えば，通常，「定数の」MPI_BOTTOMは出力バッファ引数に渡すことができる．同様に，MPI_STATUS_IGNOREはOUTステータス引数として渡すことができる．

MPIの関数では，ある引数が1つのプロセスではINとして使用され，別のプロセスではOUTとして使用されることがよくある．このような引数は，文法的にはINOUT引数であり，INOUTと示す．しかし，意味的には，1つの呼び出しで入力と出力の両方のために使用されることはない．

また，ある引数の値が一群のプロセスのうちの幾つかにとってのみ必要な場合もよくある．あるプロセスにおいてある引数が意味をもたない場合は，任意の値を引数として渡して構わない．

特に指定しない限り，OUTまたはINOUTの引数について，同じMPI手続きに渡す他の引数を用いて別名をつけてはならない．以下に示すのはC言語において引数に別名をつけた例である．次のようなC言語の手続きを定義すると，

```
void copyIntBuffer( int *pin, int *pout, int len )
{   int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}
```

この手続きの以下のコードにおける呼び出しは，別名つきの引数を持つ．

```
int a[10];
copyIntBuffer( a, a+3, 7);
```

C言語はこれを許可しているが，MPI手続きのこのような使い方は特に指定しない限り，禁止する．なおFortran言語は，引数に別名をつけることを禁止している．

全てのMPI関数の仕様は，最初に言語に依存しない表記によって示す．そのすぐ下に，ISO C言語版の関数を，その下にFortran言語およびC++言語における同じ関数を表示する．この文書ではFortran言語はFortran 90言語を指す．第2.6節を参照すること．

2.4 意味に関する用語

MPI手続きについて説明する際，以下に示す，意味に関する用語を使用する．

ノンブロッキング 操作が完了する前で，なおかつ呼び出しにおいて指定された（バッファなどの）リソースをユーザが再使用できるようになる前に，手続きから戻ることがありうる場合．ノンブロッキング要求は，MPI_ISENDなどの呼び出しによって

開始される．完了という用語は操作，要求，通信に関して使用される．操作は，ユーザがリソースを再使用できるようになり，出力バッファが更新された時点，つまりMPI_TESTの呼び出しがflag = trueを返した時点で**完了**する．ウェイトの呼び出しでflag = trueが返された時点，あるいはテストまたはステータス取得呼び出しでflag = trueが返された時点で**要求は完了**する．この完了の呼び出しは2つの効果がある．ひとつは，要求からステータスが取り出される．もうひとつは，テストとウェイトによる場合，要求が永続的でなければ解放され，永続的であれば非アクティブになる．参加している全ての操作が完了した時点で**通信が完了**する．

ブロッキング 手続きから戻り次第，呼び出しにおいて指定された リソースをユーザが再使用してよい場合．

ローカル 手続きの完了がそれを実行しているプロセスのみに依存する場合．

非ローカル 操作を完了するために，別のプロセスでのなんらかのMPI手続きの実行が必要となることもある場合．このような操作は，別のユーザプロセスとの通信を必要とすることもある．

集団 あるプロセスグループ中の全てのプロセスが，その手続きを呼び出す必要がある場合．集団的呼び出しは，同期化される場合もあり，されない場合もある．同じコミュニケータ上の集団的呼び出しは，プロセスグループの全てのメンバーが同じ順序で実行する必要がある．

定義済み 名前（MPI_INT，MPI_FLOAT_INT，MPI_UBなど）が定義された（定数）データ型，またはMPI_TYPE_CREATE_F90_INTEGER，MPI_TYPE_CREATE_F90_REAL，MPI_TYPE_CREATE_F90_COMPLEXにより構成されたデータ型．前者を名前付き，後者を名前無しという．

派生 定義されていないデータ型．

可搬 定義済みのデータ型の場合，またはタイプコンストラクタ MPI_TYPE_CONTIGUOUS，MPI_TYPE_VECTOR，MPI_TYPE_INDEXED，MPI_TYPE_CREATE_INDEXED_BLOCK，MPI_TYPE_CREATE_SUBARRAY,，MPI_TYPE_DUP，MPI_TYPE_CREATE_DARRAYのみを使用して可搬なdatatypeから派生する場合．このようなデータ型は，データ型の全ての変位が1つの定義済みデータ型の範囲内にあるため，可搬である．そのため，このようなデータ型が1つのメモリ内のデータのレイアウトに適合する場合，同じ宣言が使用されていれば，2つのシステムが異なるアーキテクチャを使用している場合でも，別のメモリ内の対応するデータのレイアウトに適合する．それに対して，データ型がMPI_TYPE_CREATE_HINDEXED，MPI_TYPE_CREATE_HVECTORまたはMPI_TYPE_CREATE_STRUCTを使用して構成されている場合，データ型には明確なバイトの変位（配列の制約に対応するための埋め草の付加など）が適用される．これらの変位は，1つのメモリ上のデータのレイアウトに適合する場合には適

さないと思われるが，別のアーキテクチャによりプロセッサ上で動作する別のプロセスにおけるデータのレイアウトのために使用される．

等価 2つのデータ型が同じ呼び出し（および引数）のシーケンスにより作成されていて，同じ型マップを持っていると考えられる場合．2つの等価なデータ型が必ずしも同じキャッシュ属性や同じ名前を持っているとは限らない．

2.5 データ型

2.5.1 不可視オブジェクト

MPIは，**システムメモリ**を管理する．システムメモリとは，ここでは，メッセージのバッファリングや，さまざまなMPIオブジェクト（グループ，コミュニケータ，データ型など）の内部表現の格納に使われるメモリを指す．このメモリにはユーザが直接アクセスすることができず，したがってここに格納されるオブジェクトは**不可視**である．つまり，そのサイズと形状はユーザには見えない．不可視オブジェクトは，ユーザ領域にある**ハンドル**を通してアクセスする．不可視オブジェクトを操作するMPI手続きには，そのオブジェクトにアクセスするためのハンドル引数を渡す．ハンドルは，MPI呼び出しでオブジェクトのアクセスに使うだけでなく，そのまま代入したり比較したりすることもできる．

Fortran言語の場合，全てのハンドルは整数型を持つ．C言語およびC++言語の場合，オブジェクトの種類ごとに，異なるハンドル型が定義される．また，C++言語ではハンドルそのものが明確なオブジェクトである．C言語およびC++言語の型は，代入および等価演算子をサポートする必要がある．

> **実装者へのアドバイス** Fortran言語の場合，ハンドルは，システムテーブル中の不可視オブジェクトのテーブルに対する添字として実現できる．C言語の場合は，同様の添字にするか，あるいはオブジェクトへのポインタにすることができる．C++言語のハンドルはテーブルの添え字またはポインタを単純に「包む」ことができる．（**実装者へのアドバイス終わり**）

不可視オブジェクトの割り当てと解放は各オブジェクト型に固有の呼び出しによって行う．これらは各オブジェクトについて述べた節に挙がっている．これらの呼び出しは，対応する型のハンドル引数を受け取る．割り当て呼び出しの場合，この引数はオブジェクトへの有効な参照値を返す出力引数である．解放のための呼び出しでは，この引数は「無効なハンドル」を値として戻る入出力引数である．MPIは「無効なハンドル」定数を各オブジェクト型ごとに1つ設ける．この定数と比較することにより，ハンドルが有効かどうかを調べることができる．

解放処理の呼び出しは，ハンドルを無効にし，オブジェクトには，解放するべきものであることを示す印をつける．そのオブジェクトは，この呼び出しの後，ユーザからはアクセスできなくなる．しかしながらMPIはそのオブジェクトをすぐに解放する必要は

ない．このオブジェクトに関わる（その解放処理の時点で）保留中の操作は，正常に終了し，オブジェクトはその後に解放される．

不可視オブジェクトとそのハンドルは，オブジェクトを生成したプロセスでのみ意味を持ち，別のプロセスに転送することはできない．

MPIでは，いくつかのあらかじめ定義された不可視オブジェクト，およびこれらのオブジェクトに対するあらかじめ定義された静的ハンドルが用意されている．このようなオブジェクトは消去してはならない．C++言語では，これはあらかじめ定義されたこれらのオブジェクトへのハンドルを`static const`として宣言することにより実行される．

根拠 この設計では，MPIデータ構造のために使う内部表現を隠してあるので，C言語，C++言語，Fortran言語のいずれでも同じような呼び出し形式を使うことができる．またこれらの言語における型に関する規則との矛盾も回避されており，将来の機能拡張も簡単に行えるようになっている．ここで使用されている不透明オブジェクトのメカニズムは，POSIXのFortran言語用呼び出し形式の標準にゆるやかに沿ったものになっている．

ユーザ領域でハンドル，システム領域でオブジェクトという具合に明示的に区別すると，メモリ領域を回収する解放呼び出しを，ユーザプログラムの適切な場所で行えるようになる．仮に，不可視オブジェクトがユーザ領域にあったならば，そのオブジェクトを必要とする保留中の動作が完了する前に，そのユーザ領域が有効である範囲の外に出ないよう非常に気を付けなければならない．ここに示した設計では，オブジェクトに解放すべきものとして印をつけることができ，ユーザプログラムが上述の範囲外に出ることも可能で，それでもオブジェクト自体は，保留中の動作が完了するまで解放されることはない．

ハンドルが代入や比較をサポートしなければならないと定めたのは，そのような処理がよく行われるからである．これにより，可能な実現方法の幅は狭くなった．これに替えて，ハンドルが任意の不透明な型であることを許すという選択もあり得た．この場合，代入と比較を行うためのルーチン導入しなければならなくなり，より複雑になるので，この規定は採用されなかった． （**根拠の終わり**）

ユーザへのアドバイス 参照先のない参照値が間違ってできることがある．これは，ユーザがあるハンドルに別のハンドルの値を代入し，そのあとこれらのハンドルに対応するオブジェクトを解放することによりできる．逆に，あるハンドル変数を，それに対応するオブジェクトを解放する前に，解放してしまうと，そのオブジェクトにはアクセスできなくなる（これは，例えばハンドルが，あるサブルーチン内のローカル変数であって，関連するオブジェクトが解放される前にサブルーチンが終了する場合に起きる）．不可視オブジェクトへの参照値の追加や削除は，そのようなオブジェクトの割当，解放をおこなうMPI呼び出しに伴う場合を除き，ユーザの責任で避けなければならない． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 不可視オブジェクトの定義が意図するところでは，各不可視オブジェクトは 1つ1つ別のオブジェクトであり，このようなオブジェクトを割

り当てる呼び出しは，そのオブジェクトに必要な全ての情報をコピーする．実装は，必要以上のコピーを避けるため，コピーを参照で置き換えてもよい．例えば，ユーザ定義データ型には，その構成要素のコピーを持たせる代わりに，その構成要素への参照値を持たせてもよい．MPI_COMM_GROUPの呼び出しは，コミュニケータに対応するグループのコピーの代わりに，そのグループに対する参照値を返してもよい．そのような場合，実装は，参照カウントを管理する必要があり，またオブジェクトの割り当てと解放を，見かけ上オブジェクトがコピーされているように見えるように行わなければならない． （**実装者へのアドバイス終わり**）

2.5.2 配列引数

MPI呼び出しで，不可視オブジェクトの配列またはハンドルの配列の引数が必要になる場合もある．ハンドルの配列は普通の配列であって，そのエントリは同じ型のオブジェクトに対するハンドルであり，その配列の中の連続した位置に並んでいる．このような配列を使う場合は常に，有効なエントリの数を示すため長さを示すlen引数を添える必要がある（この数値が他から得られる場合は除く）．有効なエントリは配列の前方寄りにあり，lenはその個数を示すが，これは，配列全体のサイズと同じでなくてもよい．他の配列引数でも同じアプローチがとられる．場合によっては，NULLハンドルが有効なエントリとみなされる．ステータスの配列に対してNULL引数が望ましい場合，MPI_STATUSES_IGNOREを使用する．

2.5.3 ステート型

MPI手続きでは，さまざまな場所で**ステート型**の引数を使用する．このようなデータ型の値は全て名前で識別され，それについての操作は定義されていない．例えばMPI_TYPE_CREATE_SUBARRAYルーチンステート型の引数 orderを取り，これはMPI_ORDER_C，MPI_ORDER_FORTRANなどの値を持つ．

2.5.4 名前付き定数

MPI手続きはときに基本的な型の引数の特別な値に，特別な意味を割り当てている．例えばタグは1対1通信操作の整数値を取る引数であるが，これには特別なワイルドカード値としてMPI_ANY_TAGがある．このような引数は，ある値域を通常値としてとる．この値域は対応する基本データ型の値域の一部分である．特殊な値（例えばMPI_ANY_TAG）は，この通常値域の外になる．タグなどの通常値域は，環境問い合わせ関数を使用して問い合わせすることができる（MPI-1文書の第7章）．送信元などの他の値域は，他のMPIルーチン与えられる値によって決まる（送信元の場合，コミュニケータのサイズ）．

MPIはまた，MPI_COMM_WORLDなどの定義済みの名前付き定数のハンドルも提供する．

全ての名前付き定数は，下記のFortran言語の場合を除いて初期化式および代入の中で使用することができるが，必ずしも配列の宣言の中で，またはC言語やC++言語の`switch`文やFortran言語の`select/case`文のラベルとして使用する必要はない．つまり，

名前付き定数はリンク時の定数であって，必ずしもコンパイル時の定数ではない．以下に示す名前付き定数は，C言語やC++言語とFortran言語語の両方においてコンパイル時の定数でなければならない．これらの定数は実行時に値が変わることはない．定数ハンドルによってアクセスされる不可視オブジェクトはMPIの初期化（MPI_INIT）からMPI終了処理（MPI_FINALIZE）までの間存在し，その間，値が変わることはない．ハンドル自体は定数で，初期化式または代入で使用することもできる．

コンパイル時の定数とする必要がある（したがって，配列長の宣言のほか，C言語やC++言語の`switch`文やFortran言語の`case`/`select`文のラベルに使用できる）定数を以下に示す．

MPI_MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)

このほか，これに対応するC++言語の定数のうち，該当するものが含まれる． Fortran言語で初期化式や代入に使用できない定数を以下に示す．

`MPI_BOTTOM`
`MPI_STATUS_IGNORE`
`MPI_STATUSES_IGNORE`
`MPI_ERRCODES_IGNORE`
`MPI_IN_PLACE`
`MPI_ARGV_NULL`
`MPI_ARGVS_NULL`
`MPI_UNWEIGHTED`

実装者へのアドバイス Fortran言語では，これらの特別な定数を実装するために，Fortran言語の標準以外の言語構造を使用する必要がある場合がある．実装ではこれらの値と規定のデータの区別ができないため，定数に特別な値を使用する（例えば，`PARAMETER`文で定義することにより）ことはできない．データは対象となるコンパイラでアドレスによって渡されるため，通常，これらの定数は定義済みの静的変数（MPIで宣言された`COMMON`ブロックの変数など）として実装される．サブルーチン内では，このアドレスをFortran言語の標準以外のメカニズム（Fortran言語の拡張機能，またはC言語の関数の実装）によって取得することができる． （**実装者へのアドバイス終わり**）

2.5.5 選択型

MPI関数はときに**選択型**（またはunion）データ型の引数を使用する．いくつかの異なる呼び出しが，同じルーチンに対するものであるにも関わらず，参照渡しによって，それぞれ違う型の実引数を渡すことがある．このような引数を与えるためのメカニズムは，言語によって異なる．Fortran言語の場合，本文書では<type>を使用して選択型変数を表現し，C言語およびC++言語の場合は，void *を使用する．

2.5.6 アドレス型

いくつかのMPI手続きは，呼び出し側のプログラム内の絶対アドレスを示す**アドレス型**引数を使用する．このような引数のデータ型は，C言語では，C++言語では，Fortran言語では`INTEGER(KIND=MPI_ADDRESS_KIND)`となる．これらの型は，1つの言語のアドレス値を変換なしで別の言語に直接渡せるように，同じ幅とエンコードアドレス値を持っていなければならない．アドレス範囲の開始を示すためのMPI定数MPI_BOTTOMがある．

2.5.7 ファイルのオフセット

入出力用に，ファイルにサイズ，変位，オフセットを指定する必要がある．これらの数字はFortran言語の整数のデフォルトサイズに設定可能な32ビットを簡単に超えてしまう場合がある．これに対応するため，Fortran言語ではこれらの数字が`INTEGER (KIND=MPI_OFFSET_KIND)`として宣言される．C言語ではMPI_Offsetを使用し，C++言語ではMPI::Offsetを使用する．これらの型は，1つの言語のオフセット値を変換なしで別の言語に直接渡せるように，同じ幅とエンコードアドレス値を持っていなければならない．

2.6 言語の呼び出し形式

この節では，MPIにおける言語の呼び出し形式一般についての規則と，Fortran言語，ISO C言語，C++言語についての個別の規則を定義する（ANSI C言語は ISO C言語に変更されている）．C++言語の呼び出し形式は廃止されている．ここではさまざまなオブジェクトの表現，およびこの標準の表現に使用する命名規則を定義する．実際の呼び出し方法については別の箇所で定義する．

MPIの呼び出し形式はFortran 90言語用のものだが，Fortran 77言語環境でも使用できるよう設計されている．

`PARAMETER`という言葉は，Fortran言語ではキーワードなので，ここでは「引数（“argument”）」という言葉を使ってサブルーチンに渡す引数を表現する．C言語およびC++言語では普通これらはパラメータと呼ばれるが，C言語およびC++言語のプログラマは「引数」という言葉（C言語およびC++言語ではこれには特定の意味はない）を理解できるであろうから，それをあてにして，Fortran言語のプログラマを不必要に混乱させるのは避けることにした．

Fortran言語では大文字と小文字が区別されるため，リンカではFortran言語の名前の解決のために小文字または大文字を使用することができる．大文字と小文字が区別される言語を使用する場合は，“mpi_”や“pmpi_”という接頭辞は使用しないようにする必要がある．

2.6.1 廃止された名前と関数

多くの章で，廃止または置換されたMPI-1の構成物に言及している．これらは，第15章で述べるように現在もMPI標準の一部となっている構成物であるが，MPI-2でこれより優れた解決法が用意されているため，今後は使用をやめることをお勧めする．例えば，アドレス引数を持つMPI-1の関数用のFortran言語の呼び出し形式では整数型を使用する．これはC言語の呼び出し形式とは整合性がなく，32ビット整数型と64ビットアドレスを使用するマシン上で問題を起こす．MPI-2では，これらの関数にはアドレス引数のための新しい呼び出し形式を備えた新しい名前が与えられている．古い関数の使用は廃止されている．整合性のため，ここと他のいくつかのケースで，古い関数と同等の内容ではあるが，C言語の新しい関数も用意されている．古い名前は廃止されている．MPI-1で定義済みのデータ型MPI_UBおよびMPI_LBもこの一例である．これらは，扱いにくく，間違いやすいため，廃止されている．MPI-2の関数MPI_TYPE_CREATE_RESIZEDは同じ効果を得るための便利なメカニズムを備えている．

表2.1に，廃止された全ての構成物の一覧を示す．MPI_LBおよびMPI_UB定数は関数MPI_TYPE_CREATE_RESIZEDに置換されているが，これは主にサイズ変更されたデータ型を作成する目的でMPI_TYPE_STRUCTへの入力のデータ型として使用されていたためである．また，この一覧に一部のC言語の`typedef`とFortran言語のサブルーチンの名前が含まれているが，これらはコールバック関数の型である．

2.6.2 Fortran言語の呼び出し形式に関する事項

もともと，MPI-1.1ではFortran 77言語のための呼び出し形式が用意されていた．これらの呼び出し形式は残されているが，現在はFortran 90言語の標準に従って解釈される．下記のように，MPIでは現在も大部分のFortran 77言語のコンパイラを使用することができる．Fortran言語という用語を使用した場合，Fortran 90言語言語のことを指す．

全てのMPI名は`MPI_`という接頭辞を持ち，また全ての文字は大文字である．プログラムは，この接頭辞`MPI_`で始まる名前を宣言してはならない．例えば，変数，サブルーチン，関数，パラメタ，派生データ型，抽象インターフェイス，またはモジュールに対しそうした名前を宣言してはならない． プロファイリングインターフェイスとの衝突を避けるため，プログラムでは`PMPI_`という接頭辞を持つ関数も避ける必要がある．この制約は名前の衝突を避けるために必要である．

全てのMPIFortran言語のサブルーチンには，最後の引数に戻り値を付ける．いくつかのMPI操作は関数であり，これらは戻り値引数を持たない．成功した場合の戻り値はMPI_SUCCESSになる．その他の場合のエラーコードは実装によって異なる．第8章および付録Aのエラーコードを参照すること．

廃止	MPI-2での置換
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_LB	MPI_TYPE_CREATE_RESIZED
MPI_UB	MPI_TYPE_CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function	MPI_Comm_errhandler_function
MPI_KEYVAL_CREATE	MPI_COMM_CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI_COMM_FREE_KEYVAL
MPI_DUP_FN	MPI_COMM_DUP_FN
MPI_NULL_COPY_FN	MPI_COMM_NULL_COPY_FN
MPI_NULL_DELETE_FN	MPI_COMM_NULL_DELETE_FN
MPI_Copy_function	MPI_Comm_copy_attr_function
COPY_FUNCTION	COMM_COPY_ATTR_FN
MPI_Delete_function	MPI_Comm_delete_attr_function
DELETE_FUNCTION	COMM_DELETE_ATTR_FN
MPI_ATTR_DELETE	MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI_COMM_SET_ATTR

表 2.1: 廃止された構成物

文字列の最大長を表す定数は，第16.3.9節に示すように， Fortran言語ではC言語およびC++言語よりも1小さい．

Fortran言語では，ハンドルは整数型で表現される．二値変数は，論理型を持つ．

配列引数の添字は1から始まる．

MPIFortran言語の呼び出し形式はいくつかの点でFortran 90言語の標準との整合性がない．これらの不整合にはレジスタの最適化の問題などがあり，ユーザコードとの関連性がある．これについては第16.2.2節で詳しく説明する．これらについても，Fortran 77言語との整合性はない．

2.6.3 C言語の呼び出し形式に関する事項

ISO C言語の宣言形式を使用する．全てのMPI名は`MPI_`という接頭辞を持ち，あらかじめ定義された定数は全て大文字からなる名前を持ち，あらかじめ定義された型や関数では，接頭辞のあとの文字が1文字だけ大文字になる． プログラムは接頭辞 `MPI_` で始まる名前（識別子）を宣言してはならない．ここでいう名前とは例えば，変数，関数，

定数，型，またはマクロに対する名前である． プロファイリングインターフェイスをサポートするため，名前がPMPI_という接頭辞で始まる関数をプログラムで宣言しないようにする必要がある．

名前付き定数の定義，関数プロトタイプ，型定義は，mpi.hという名のヘッダーファイルの中にいれて提供しなければならない．

ほとんど全てのC言語関数はエラーコードを返す．成功した場合の戻り値はMPI_SUCCESSになるが，エラーが起こった場合の戻り値は，実装によって異なる．

不可視オブジェクトの各種類に対応するハンドルに対しては，それぞれ別の型宣言が用意される

配列引数の添字は0から始まる．

論理フラグは整数で，値が0の場合「偽」，0以外の場合「真」を意味する．

選択型引数は，void *型のポインタである．

アドレス引数は，MPIで定義する型MPI_Aintである．ファイルの変位は型MPI_Offsetである． MPI_Aintは，対象とするアーキテクチャ上の任意の有効なアドレスを保持するのに必要なサイズを持つ整数値として定義する． MPI_Offsetは，対象とするアーキテクチャ上の任意の有効なファイルサイズを保持するのに必要なサイズを持つ整数値として定義する．

2.6.4 C++言語の呼び出し形式に関する事項

C++言語の呼び出し形式は廃止されている．標準には，C++言語ではなく，C言語用の規則が規定されている．このような場合，適宜，C言語の規則をC++言語に適用する必要がある．特に，規定されている定数の値はC言語およびFortran言語用のものである．これらとC++言語の名前の相互参照を付録Aに示す．

使用する宣言の形式はISO C++言語のものである．MPIの全ての名前はMPIという名前空間の範囲内で宣言されているため，MPI::という接頭辞を付けて参照される．定義された定数は全て大文字であり，クラス名，定義された型，関数は最初の文字だけが大文字である． プログラムはMPI 名前空間の中で名前を宣言してはならない．例えば，変数，関数，定数，型，またはマクロに対する名前である．この制約は名前の衝突を避けるために必要である．

名前付き定数の定義，関数プロトタイプ，型定義は，mpi.hという名のヘッダーファイルの中にいれて提供しなければならない．

> **実装者へのアドバイス** mpi.hファイルにはC言語とC++言語の両方の定義を記述することができる．通常，定義された値を使用する（一般的に__cplusplusだが，必須ではない）だけで，C++言語の使用中にC++言語の定義を保護しているかどうかを確認することができる．C言語のコンパイラにより，この方法で保護されたソースを正規のC言語のコードとすることが求められる可能性がある．この場合，全てのC++言語の定義を別のヘッダーファイルに記述し，“#include”指示文を使用して必要なC++言語の定義をmpi.hファイルに組み込むことができる． （**実装者へのアドバイス終わり**）

通常，オブジェクトの作成，または情報を返すC++言語の関数は，オブジェクトまたは情報を戻り値に格納する．MPI関数の言語非依存のプロトタイプはC++言語の戻り値を出力パラメータとして組み込むため，MPI関数の意味的記述ではそのパラメータ名によってC++言語の返却値を参照する．それ以外のC++言語の関数は`void`を返す．

場合によっては，MPIで値を返さないよう指定することもできる．例えば，ステータスが格納されないよう指定することができる．C言語とFortran言語では特別な入力値によってこれが行われるが，C++言語の場合は2つの呼び出し形式を使用して行われる．一方の呼び出し形式にはオプションの引数があり，他方にはない．

C++言語の関数はエラーコードを返さない．デフォルトのエラーハンドラがMPI::ERRORS_THROW_EXCEPTIONSに設定されている場合，C++言語の例外メカニズムに基づいてMPI::Exceptionオブジェクトが投げられ，エラーのシグナルが発行される． 特定の型ではデフォルトのエラーハンドラ（MPI::ERRORS_ARE_FATAL）は変更されていない．ユーザのエラーハンドラも使用できる．MPI::ERRORS_RETURN呼び出し元の関数に制御を戻すだけで，ユーザがエラーコードを取得するという規定もない．

整数のエラーコードを返すユーザのコールバック関数は例外を投げられないようにする必要があり，返されたエラーは適切なエラーハンドラを呼び出すことにより，MPI実装によって処理される．

> **ユーザへのアドバイス** C++言語でMPIのエラー処理を自分で行いたい場合，C言語でその目的で使用するMPI::ERRORS_RETURNではなく，MPI::ERRORS_THROW_EXCEPTIONSエラーハンドラを使用する必要がある．複数の言語が混在している状況で例外を使用する場合，注意が必要となる． （**ユーザへのアドバイス終わり**）

不可視オブジェクトのハンドルはそれ自体がオブジェクトでなければならず，C言語およびFortran言語と意味的に同様の機能を実行するには代入と等価演算子をオーバーライドする必要がる．

配列引数の添字は0から始まる．

論理フラグの型は`bool`である．

選択型型引数は，`void *`型のポインタである．

アドレス引数は，MPIで定義する整数型MPI::Aintである．この型は，対象とするアーキテクチャ上の任意の有効なアドレスを保持するのに必要なサイズを持つ整数値として定義する．同様に，MPI::Offsetはファイルのオフセットを保持するための整数値である．

大部分のMPIの関数はMPIC++言語のクラスのメソッドである．MPIのクラス名を言語非依存のMPIの型から生成するには，接頭辞`MPI_`を削除し，`MPI`の名前空間内で型を範囲指定する．例えば， MPI_DATATYPEは`MPI::Datatype`となる．

一般的に，MPIの関数名は所定の命名規則に従う．場合によっては， MPIの関数はMPI-1用に定義されている関数に関連し，命名規則に従っていない名前を持っていることがある．この場合，言語非依存の名前は，命名規則に違反するMPI-2の名前が与えられるとしても，MPIの名前と同じになる．この場合，C言語とFortran言語の名前は言語非

依存の名前と同じになる．しかし，C++言語の名前にはこの命名規則が反映され，C言語およびFortran言語の名前と異なる可能性がある．そのため，MPIの名前と同じC++言語の名前が言語非依存の名前と異なる場合がある．その結果，C++言語の名前が言語非依存の名前と異なることになる．言語非依存の名前MPI_FINALIZEDとC++言語の名前MPI::Is_finalizedはその一例である． C++言語では，関数typedefは適切なクラス内で公開で作成される．しかし，これらの宣言は以下のように多少煩雑になる．

{typedef MPI::Grequest::Query_function(); （廃止された呼び出し形式．第15.2節を参照）
}

は次のようになる．

```
namespace MPI {
  class Request {
    // ...
  };

  class Grequest : public MPI::Request {
    // ...
    typedef Query_function(void* extra_state, MPI::Status& status);
  };
};
```

C++言語のtypedefを宣言するときにこの長い手続きの代わりに，縮約された形式を使用する．特に，関数のtypedefに対して，クラスと名前空間の有効範囲を明確に示す．そのため，上記の例は以下のようなテキストで示される．

```
typedef int MPI::Grequest::Query_function(void* extra_state,
                                          MPI::Status& status)
```

付録A.4と本書全体に示したC++言語の呼び出し形式は，単純な一連の名前生成規則をMPIの関数仕様に当てはめることにより作成された．これらのガイドラインは大部分の場合の要件を満たしていても，すべての状況に適しているとは限らない．曖昧さが生じる場合や，特別な意味的な記述が必要な場合，状況に応じてこれらのガイドラインを変更することができる．

1. 全ての関数，型，定数は，MPIというnamespaceの有効範囲内で宣言される．

2. MPIのハンドルの配列は常に配列リスト内に残される（IN引数かOUT引数かに関係なく）．

3. MPIの関数の引数リストにスカラのINハンドルが含まれ，関数をそのハンドルに対応するオブジェクトのメソッドとして定義することに意味がある場合，関数では対応するMPIのクラスのメンバ関数が作成される．メンバ関数は対応するMPIの関数名に従って命名されるが，接頭辞MPI_ とオブジェクト名の接頭辞（ある場合）は削除される．また，以下が適用される．

   (a) スカラのINハンドルは引数リストから除外され，これは除外引数に対応する．

   (b) 関数はconstとして宣言される．

4. クラスに属し，かつ，クラスにユニークなスカラのINまたはINOUTパラメータを持たない場合，MPIの関数は，そのクラスの（static）関数となる．

5. 引数リストにMPI_STATUS型（または配列）でない1つのOUT引数が含まれる場合，その引数はリストから除外され，関数はその値を返す．

   例 **2.1** MPI_COMM_SIZEのC++言語呼び出し形式は次のようになる．
   int MPI::Comm::Get_size(void) const.

6. 引数リストに複数のOUT引数がある場合，1つが戻り値として選択され，リストから除外される．

7. 引数リストにOUT引数が含まれていない場合，関数の戻り値は`void`となる．

   例 **2.2** MPI_REQUEST_FREEのC++言語呼び出し形式は次のようになる． void MPI::Request::Free(void)

8. 上記の規則が適用されないMPIの関数はクラスのメンバではなく，`MPI`の名前空間内で定義される．

   例 **2.3** MPI_BUFFER_ATTACHのC++言語呼び出し形式は次のようになる． void MPI::Attach_buffer(void* buffer, int size).

9. 全てのクラス名，定義された型，関数名は最初の文字だけが大文字となる．定義された定数は全て大文字となる．

10. INポインタ，参照，または配列の引数は`const`として宣言する必要がある．

11. ハンドルは参照により渡される．

12. 配列引数は，意味的な正確さを表すため，ポインタではなく，角括弧（`[]`）で示される．

2.6.5 関数とマクロ

C言語のマクロとして実装することができるのは，MPI_WTIME，MPI_WTICK，PMPI_WTIME，PMPI_WTIME，PMPI_WTICK，第16.3.4節のハンドル変換関数（MPI_Group_f2cなど）である．

実装者へのアドバイス 実装者はマクロとして実装するルーチンを文書化する必要がある． （**実装者へのアドバイス終わり**）

ユーザへのアドバイス ルーチンがマクロとして実装されている場合，MPIのプロファイリングインターフェイスと一緒に使用することはできない． （**ユーザへのアドバイス終わり**）

2.7 プロセス

MPIプログラムは，自律的なプロセスで構成される．これらのプロセスは独自のコードをMIMDスタイルで実行する．各プロセスによって実行されるコードは，同一である必要はない．これらのプロセスは，MPIの通信プリミティブを呼び出すことにより通信する．通常，各プロセスは独自のアドレス空間で実行されるが，共有メモリに基づくMPIの実装も考えられる．

この文書で規定する，並列プログラムの動作は，MPI呼び出しのみが通信に使用されることを前提としている．MPIプログラムと他の通信手段，入出力，プロセス管理との相互作用については規定していない．標準の仕様に特に記載されていない限り，MPIは同様または同等の機能を提供する外部メカニズムとの相互作用の結果に関して要件を定めていない．ここに含まれる外部メカニズムとの相互作用として対象となるのはプロセス制御，共有およびリモートメモリアクセス，ファイルシステムのアクセスおよび制御，プロセス間通信，プロセスのシグナル発行，端末入出力だが，これに限定されない．質の高い実装をおこなうためには，このような相互作用がユーザの使い勝手の向上に反映されるよう努め，必要に応じて制約事項を文書化する必要がある．

> **実装者へのアドバイス** MPI内でサポートされる機能のためのこうした追加メカニズムをサポートする実装では，これらとMPIとの相互作用について文書化することが求められる．（**実装者へのアドバイス終わり**）

MPIとスレッドの相互作用については，第12.4節で定義する．

2.8 エラー処理

MPIは，信頼性のあるメッセージ転送をユーザに提供する．送られるメッセージは，常に正しく受け取られるため，ユーザが転送エラー，ランタイムその他のエラー状態についてチェックをおこなう必要はない．言い換えれば，MPIには，通信システム内で発生するエラーを扱うメカニズムはない．MPI の実装を信頼性のない下位のメカニズムの上に構築する場合，ユーザをその信頼性のなさから隔離したり，回復不能なエラーを障害として通知したりするのは，そのMPIサブシステムの実装者の仕事である．可能な限り，このような障害は関連する通信呼び出しでエラーとして通知される．同様にMPIそれ自身にはプロセッサの障害を処理するメカニズムはない．

とはいえ，MPIプログラムが間違っている可能性は当然ある．**プログラムエラー**は，MPI呼び出しの引数が正しくない（送信操作のときに送信先が存在しない，受信操作のときにバッファが小さすぎるなど）ときに発生しうる．この種のエラーは，どのような実装においても起きると考えられる．また，**リソースエラー**は，使用可能なシステムリソース（保留中のメッセージ数，システムバッファ数など）の量をプログラムが超えたときに発生しうる．この種のエラーが起きるかどうかは，システム内の使用可能なリソースの量およびリソース割当てに使うメカニズムに依存し，システムごとに異なる可能

性がある．質の高い実装は，重要なリソースについては十分ゆとりのある上限を提供して，これに代表される移植性の問題を軽減するのに役立つ．

C言語およびFortran言語では，ほとんど全てのMPI呼び出しでは，操作が正しく終了したことを示すコードを返す．エラーが呼び出しの最中に発生した場合，可能な限り，MPI呼び出しはエラーコードを返す．デフォルトでは，MPIライブラリの実行中にエラーが見つかった場合，ファイル操作を除いて，その並列計算は異常終了する．しかしMPIは，ユーザがこのデフォルト時の振舞いを変更して，回復可能なエラーを処理するためのメカニズムを提供する．ユーザは，エラーは全て致命的でないと指定して，MPI呼び出しが返すエラーコードをユーザ自身で処理することができる．またユーザはユーザ独自のエラー処理ルーチンを用意することもできる．このルーチンは，MPI呼び出しが異常な状態で戻ってくる場合には常に呼び出される．MPIエラー処理機能については，第8.3節で説明している．C++言語の関数の戻り値はエラーコードではない．デフォルトのエラーハンドラがMPI::ERRORS_THROW_EXCEPTIONSに設定されている場合，C++言語の例外メカニズムに基づいてMPI::Exceptionオブジェクトが投げられ，エラーのシグナル発行が行われる．497ページの第16.1.8節も参照すること．

エラーが発生したとき，MPI呼び出しが意味のあるエラーコードを返す機能は，いくつかの要因によって制限される．MPIである種のエラーが検出できないことがある．また他の種のエラーについては，通常の実行モードで検出するにはコストがかかりすぎることもある．あるいは，ある種のエラーでは「壊滅的」に被害が大きいため，MPIが制御を整合のとれた状態で呼び出し側に返すことができないこともある．

その他の微妙な問題として，非同期通信の性質によって発生するものがある．つまり，MPI呼び出しは，その呼び出しが戻った後にも非同期的に継続する処理を起動することがある．このため，その操作は正しく終了したことを示すコードを返すが，そのあとでエラー例外が発生する，という可能性がある．同じ操作に関連する呼び出し（例えば，非同期操作が終了したことを確認する呼び出し）がそのあとにある場合，この呼び出しに対応するエラー引数をエラーの性質を示すために使用する．場合によっては，操作に関連する全ての呼び出しが終了したあとにエラーが発生することがあり，そのためエラーの性質を示すために使えるエラー値がないこともある（例えば，レディモードによる送信での受信側のエラー）．このようなエラーは，致命的なものとして扱わなければならない．その状態からユーザが復旧するための情報を返すことができないからである．

この文書では，MPI呼び出しでエラーが発生したあとの計算処理の状態を規定していない．望ましい操作は，しかるべきエラーコードが返され，エラーの影響が可能な限り狭い範囲に留められることである．例えば，エラーを起こした受信呼び出しが，メッセージ受信用に指定された領域を越える，受信側のメモリのいかなる部分をも書き換えることがない，という動作は非常に望ましい．

この文書では，MPI呼び出しでエラーが発生したあとの計算処理の状態を規定していない．望ましい操作は，しかるべきエラーコードが返され，エラーの影響が可能な限り狭い範囲に留められることである．例えば，エラーを起こした受信呼び出しが，メッセージ受信用に指定された領域を越える，受信側のメモリのいかなる部分をも書き換えることがない，という動作は非常に望ましい．

MPIではユーザが新しいエラーコードを作成するための方法が定義されている．これを第8.5節に示す．

2.9 実装に関する事項

いくつかの領域において，MPIの実装が動作環境やシステムとやりとりすることがある．MPIはいかなるサービス（入出力やシグナル処理など）の提供も義務づけてはいないが，これらのサービスが使用できるのであれば，このような動作を提供するよう強く提唱している．これは，同じサービスを提供するプラットフォーム間の可搬性を実現する上で，非常に重要な事項である．

2.9.1 基本ランタイムルーチン独立性

MPIプログラムでは，以下のことが必要となる．それは，基本的な言語環境の一部であるライブラリルーチン（例えばFortran言語の`write`， ISO C言語の`printf`や`malloc`）が，MPI_INITとMPI_FINALIZEの間に実行される場合，それらがそれぞれ独立に動作すること，およびそれらが完了するかどうかは同じMPIプログラム内の他のプロセスの動作には依存しないことである．

これは，操作が集団的な並列サービスを提供するライブラリルーチンの作成を妨げるものではまったくない．しかしながら以下のプログラムは，MPI_COMM_WORLDのサイズに関係なく，ISO C言語環境で最後まで動作するものと期待されている（実行ノードで`printf`が使用できることが前提）．

```
int rank;
MPI_Init((void *)0, (void *)0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) printf("Starting program\n");
MPI_Finalize();
```

同様のFortran言語およびC++言語プログラムも最後まで動作するものと期待される．

逆に，要求されないものの例としては，複数のタスクから呼び出されるとき，これらのルーチンが特定の順序で動作すること，がある．例えばMPIでは，以下のプログラムからの出力がどうなるべきかもどうなるのが望ましいかも定めない（ここでも実行ノードで入出力が行えることが前提）．

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Output from task rank %d\n", rank);
```

さらに，リソースの枯渇その他のエラーによって失敗した呼び出しは，ここで要求している事項に違反するとは見なされない（ただしこれらは，正常終了しなくてもよいだけで，終了する必要はある）．

2.9.2 シグナルとの相互作用

MPIはシグナルを使用したプロセス間のやりとりについては指定していないし，MPIがシグナルセーフである必要もない．実装ではシグナルをそれ自身の使用のために予約することができる．実装では使用するシグナルを文書化しておく必要があり，SIGALRM，SIGFPE，SIGIOを使用しないことを強く推奨する．また，実装ではシグナルハンドラ内からのMPI呼び出しの使用を禁止することもできる．

マルチスレッド環境では，MPI呼び出しを実行しないスレッドでのみシグナルを捕捉することにより，シグナルとライブラリMPIの間の衝突を避けることができる．質の高い単一スレッド実装はシグナルセーフで，シグナルによって中断されたMPI呼び出しはシグナル処理の完了後に再開され，正常に終了する．

2.10 プログラム例

この文書中のプログラム例は，解説のみを目的としている．標準を規定することを意図したものではない．またプログラム例については，注意深いチェックや検証がおこなわれているわけではない．

第3章

1対1通信

3.1 概要

プロセスによるメッセージの送信と受信が基本的なMPI通信メカニズムである．1対1信の基本的な操作は，**送信(send)**と**受信(receive)** である．その使用法を以下の例で説明する．

```
#include "mpi.h"
int main( int argc, char **argv )
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  if (myrank == 0)    /* code for process zero */
  {
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1)  /* code for process one */
  {
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received :%s:\n", message);
  }
  MPI_Finalize();
}
```

この例では，プロセス0(myrank = 0)は**送信**関数MPI_SENDを使用して，プロセス1にメッセージを送っている．この関数では，メッセージデータを取出すための送信側のメモリ内の**送信バッファ**を指定する．上の例の場合，送信バッファは変数messageを含む，プロセス0のメモリ上の記憶領域である．送信バッファの位置，サイズ，型は，送信関数の先頭3つの引数により指定される．送信されるメッセージは，変数messageの13文字である．またこの送信関数は，エンベロープをメッセージに付加する．エンベロープはメッセージ送信先を指定し，その中には**受信**関数が特定のメッセージを選択するときに使用できる区別情報がある．送信関数の最後の3つの引数によって，送信されるメッセージのエンベロープが送信側のランクと共に特定される．プロセス1(myrank = 1)が，受信関

数MPI_RECVを使用して，このメッセージを受信する．エンベロープの値に従って受信されるメッセージが選択され，メッセージデータが**受信バッファ**に格納される．上の例では，受信バッファは，プロセス1のメモリ上の変数messageを含む記憶領域から構成される．受信関数の先頭3つの引数により受信バッファの位置，サイズ，型が指定される．次の3つの引数は，受信メッセージの選択に使用される．最後の引数は，受け取ったメッセージの情報を返すために使用される．

次の節では，ブロッキング送信関数および受信関数について説明する． 送信，受信，ブロッキング通信の意味，型一致の条件，異機種 環境における型変換，より一般的な通信モードについて説明する． ノンブロッキング通信については次に説明し，そのあとにチャネル式の構成物，送受信関数などを続け[1]，最後に，「ダミー」プロセスMPI_PROC_NULLについて説明する．

3.2 ブロッキング送信関数および受信関数

3.2.1 ブロッキング送信

ブロッキング送信関数の構文は，以下のようになる．

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファ内の要素数（非負の整数型）
IN	datatype	送信バッファの各要素のデータ型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
{void MPI::Comm::Send(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}
```

この呼び出しのブロッキングの意味については，第3.4節で説明する．

3.2.2 メッセージデータ

MPI_SEND関数によって指定される送信バッファは，datatypeで示される型の，アドレスbufから始まる，連続したcount個のエントリによって構成される．メッセージの長さは，**バイト数**ではなく**要素数**で指定することに注意すること．後者は機種非依存になっており，アプリケーションレベルに近くなる．

[1]訳者註：この部分，原文では重複しているが， MPI3.1 では正しく修正されている

メッセージのデータ部分は，datatypeで示されるそれぞれの型の連続したcount個の値で構成される．countの値は0でも良く，その場合，メッセージのデータ部分が空になる．メッセージのデータ値に指定できる基本的なデータ型は，ホスト言語の基本的なデータ型に対応している．Fortran言語で利用可能なこの引数の型および対応するFortran言語のデータ型を表3.1に示す．

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

表 3.1: Fortran言語のデータ型に対応する定義済みのMPIのデータ型

C言語で利用可能なこの引数の型および対応するC言語のデータ型を表3.2に示す．

データ型MPI_BYTEとMPI_PACKEDは，Fortran言語やC言語のデータ型に対応するものがない．データ型MPI_BYTEは，バイト（8桁の2進数）で構成される．バイトは解釈されていないもので，文字とは異なるものである．異なるマシンでは異なる文字表現になることがあり，また文字を表現するときに複数バイトを使用することもある．一方，バイトは全てのマシンで同じ2進値を持つ．データ型MPI_PACKEDの使用については，第4.2節で説明する．

MPIは，Fortran言語とISO C言語の基本データ型と対応するデータ型 のサポートを要求する．ホスト言語がその他のデータ型を持つ場合， MPIデータ型にも追加のものが必要になる．つまりDOUBLE COMPLEX型として宣言されたFortran言語の倍精度複素数に対するMPI_DOUBLE_COMPLEX や，REAL*2，REAL*4，REAL*8型としてそれぞれ宣言されたFortran言語の実数に対するMPI_REAL2，MPI_REAL4，MPI_REAL8や，INTEGER*1，INTEGER*2，INTEGER*4としてそれぞれ宣言されたFortran言語の整数に対するMPI_INTEGER1，MPI_INTEGER2，MPI_INTEGER4などがそれにあたる．

根拠 設計における目標の1つは，追加的なプリプロセスやコンパイルをおこなうことなく，MPIをライブラリとして実装できるようにすることである．そのため，通信呼び出しが通信バッファに変数のデータ型についての情報を持つことを前提とすることはできない．この情報は，明示的な引数によって提供しなければならない．このようなデータ型の情報の必要性は，第3.3.2節で説明する． （**根拠の終わり**）

根拠 データ型MPI_C_BOOL，MPI_INT8_T，MPI_INT16_T，MPI_INT32_T，MPI_UINT8_T，MPI_UINT16_T，MPI_UINT32_T，MPI_C_COMPLEX，MPI_C_FLOAT_COMPLEX，MPI_C_DOUBLE_COMPLEX，

MPI datatype	C datatype
MPI_CHAR	char （印字可能文字として扱う）
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG（同義語として）	signed long long int
MPI_SIGNED_CHAR	signed char （整数値として扱う）
MPI_UNSIGNED_CHAR	unsigned char （整数値として扱う）
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t （<stddef.h>の中で定義されている） （印字可能文字として扱う）
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX（同義語として）	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

表 3.2: C言語のデータ型に対応する定義済みのMPIのデータ型

MPI_C_LONG_DOUBLE_COMPLEXには対応するC++言語呼び出し形式グのデータ型がない．これは，C言語のプリプロセッサおよび C++言語の名前空間における名前との衝突を避けるために意図的におこなわれている．C++言語のアプリケーションでは，機能を損なうことなく C言語の呼び出し形式を使用することができる． （根拠の終わり）

MPI datatype	C datatype	Fortran datatype
MPI_AINT	MPI_Aint	INTEGER (KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	MPI_Offset	INTEGER (KIND=MPI_OFFSET_KIND)

表 3.3: C言語およびFortran言語のデータ型に対応する定義済みのMPIのデータ型

データ型MPI_AINTおよびMPI_OFFSETは，MPIで定義された C言語の型MPI_AintおよびMPI_Offset，Fortran言語の INTEGER (KIND=MPI_ADDRESS_KIND)および INTEGER (KIND=MPI_OFFSET_KIND)に対応する．これを 表3.3に示す．これらの型による言語間通信についての詳細は，第16.3.10節を参照すること．

MPI datatype	C++ datatype
MPI_CXX_BOOL	bool
MPI_CXX_FLOAT_COMPLEX	std::complex<float>
MPI_CXX_DOUBLE_COMPLEX	std::complex<double>
MPI_CXX_LONG_DOUBLE_COMPLEX	std::complex<long double>

表 3.4: C++言語のデータ型に対応する定義済みのMPIのデータ型

C++言語コンパイラがある場合，表3.4の中の データ型もまた，C言語およびFortran言語でサポートされる．

3.2.3 メッセージエンベロープ

データ部分の他に，メッセージは，メッセージを区別したり選択的に受信したりするときに使用できる情報を伝える．この情報は，一定の数のフィールドによって構成されるもので，集合的に**メッセージエンベロープ**と呼ばれる．これらのフィールドは次のようになっている．

送信元
送信先
タグ
コミュニケータ

メッセージの送信元は，メッセージの送信側のIDによって暗黙的に決定される．他のフィールドは，送信関数の引数によって指定される．

メッセージの送信先は，dest引数で指定される．

メッセージタグは整数値であり，tag引数で指定される．この整数は，プログラムがメッセージのタイプを区別するために使用される．有効なタグの値の範囲は，0,...,UBであるが，ここで言うUBは実装依存である．この値は，表8章で述べているように，属性MPI_TAG_UBの値を問い合わせることで得ることが出来る．MPIでは，UBの値を32767以上にする必要がある．

comm引数により，送信関数で使用される**コミュニケータ**が指定される．コミュニケー

タについては第6章で説明するが，以下にその使用法を簡単にまとめる．

コミュニケータは，通信操作に対応する通信コンテクストを指定する．それぞれの通信コンテクストに応じて，別々の「通信世界」が提供され，メッセージは常にそれらが送られるコンテクストの中で受信されて，異なるコンテクストで送られたメッセージが干渉することはない．

コミュニケータはまた，この通信コンテクストを共有するプロセスの集合を特定する．この**プロセスグループ**内ではプロセスは順序付けられ，グループ内でのランクで識別される．このため，destの有効な値の範囲は， 0, ... , n-1となる．ここでnはグループ内のプロセスの数である．（コミュニケータがグループ間コミュニケータの場合，送信先はリモートグループ内のランクで識別される．第6節参照のこと．）

MPIは定義済みのコミュニケータMPI_COMM_WORLDを提供する．これを使用すると，MPI初期化の後にアクセスできる全てのプロセスとの通信が可能になり，プロセスは，MPI_COMM_WORLD内のランクで識別される．

> **ユーザへのアドバイス** ユーザが，既存のほとんどの通信ライブラリで提供されているような，フラットな名前空間の概念および1つの通信コンテクストで満足していれば，comm引数として使用するのは定義済みの変数MPI_COMM_WORLDで十分である．これを使用することによって，すべてのプロセスとの通信が初期化時に使用できるようになる．
>
> ユーザは，第6章で説明するように新しいコミュニケータを定義することができる．コミュニケータは，ライブラリとモジュールのための有効なカプセル化メカニズムを提供する．これらを使用することにより，モジュールは，独立の通信世界と独自のプロセス番号体系を持つことができる． （**ユーザへのアドバイス終わり**）

> **実装者へのアドバイス** メッセージエンベロープは通常固定長のメッセージヘッダとして符号化される．しかしながら，実際の符号化は，実装依存である．いくつかの情報（例えば送信元または送信先）は暗黙的にもすることができ，メッセージによって明示的に転送される必要はない．またプロセスは，相対的なランクや，絶対的なIDなどにより識別できる． （**実装者へのアドバイス終わり**）

3.2.4 ブロッキング受信

ブロッキング受信関数の構文は，以下のようになる．

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	受信バッファの先頭アドレス（選択型）
IN	count	受信バッファ内の要素数（非負の整数型）
IN	datatype	受信バッファの各要素のデータ型（ハンドル）
IN	source	送信元のランクまたはMPI_ANY_SOURCE（整数型）
IN	tag	メッセージタグまたはMPI_ANY_TAG（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	status	ステータス型オブジェクト（ステータス型）

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
{void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
             int source, int tag, MPI::Status& status) const （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
             int source, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}
```

この呼び出しのブロッキングの意味論については，第3.4節節で説明する．

受信バッファは，datatypeで指定される型の連続したcount個の要素を含む記憶領域によって構成される．この記憶領域の開始アドレスはbufである．受信されるメッセージの長さは，受信バッファの長さ以下でなければならない．オーバーフローエラーは，送られてくるデータが切り詰めなくては受信バッファに収まらない場合に発生する．

受信バッファより短いメッセージが到着した場合，（短い）メッセージに対応する範囲内の領域のみで更新が行われる．

ユーザへのアドバイス 第3.8節で説明しているMPI_PROBE関数を使用すると，長さの不明なメッセージを受信することができる． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス プログラムエラーの場合の特定の動作がMPIによって規定されていない場合でも，オーバーフローの処理方法として推奨される方法は，入ってくるメッセージの送信元とタグについての情報をstatusに入れて戻ることである．その受信関数はエラーコードを返す．高品質な実装ではまた，受信バッファの範囲外のメモリは上書きされない事を保証する．

メッセージが受信バッファより短い場合，MPIは非常に厳格に他の範囲の更新を禁止する．ある種の最適化のために，もっと寛大な処置は可能であろうが，これは許されていない．実装では，たとえそれが奇数アドレスでも，正確に受信バッファの終了地点で，受信側のメモリに対するコピーを終了させる準備をしておかなくてはならない． （**実装者へのアドバイス終わり**）

受信関数は，メッセージエンベロープの値によってメッセージを選択する．そのエンベロープが，受信関数によって指定されたsource，tag，commのそれぞれの値と一致する場合にメッセージは受信関数によって受信される．受信側は，sourceの値としてワイルドカードMPI_ANY_SOURCE，またtagの値としてワイルドカードMPI_ANY_TAGを指定することができ，これによって任意の送信元やタグが受け入れ可能であることを示す．commの値としてはワイルドカードを指定することはできない．このように，メッセージが受信プロセスに送られ， コミュニケータが一致し，送信元が一致（source=MPI_ANY_SOURCEパターンになっていない場合），タグが一致（tag=MPI_ANY_TAGパターンになっていない場合）する場合に限り，メッセージは受信関数で受信される．

メッセージタグは，受信関数のtag引数によって指定される．引数sourceがMPI_ANY_SOURCEと異なる場合は，同じコミュニケータに付随するプロセスグループ（グループ間コミュニケータの場合はリモートプロセスグループ）内のランクとして指定される．そのため，source引数に対応する有効な値の範囲は，{0,...,n-1}∪{MPI_ANY_SOURCE}になる（ここでnはこのグループ内のプロセス数）．

送信関数と受信関数の間の次の違いに注意しなければならない．受信関数では，任意の送信元からメッセージを受けつけるが，送信関数では，特定の受信側を指定しなければならない．このことは，データ転送が送信側によってなされる「プッシュ型」通信メカニズムに対応していることを示す（データ転送が受信側によってなされる「プル型」メカニズムとは異なる）．

送信元＝送信先は許される，つまりプロセスがメッセージを自分自身に送ることは可能である（ただし，上記で説明したブロッキング送受信関数の場合，デッドロックを引き起こす可能性があるため安全ではない．第3.5節を参照のこと）．

> **実装者へのアドバイス** メッセージコンテクストや，コミュニケータのその他の情報を，追加のタグフィールドとして実装することができる．このフィールドではワイルドカード一致が許可されていないことや，このフィールドの値の設定がコミュニケータ操作関数で制御されるなどの点で通常のメッセージタグとは異なっている． （実装者へのアドバイス終わり）

3.2.5 リターンステータス型

受信関数でワイルドカード値が使用された場合，受信したメッセージの送信元やタグがわからないことがある．また一つのMPI関数で複数のリクエストが実行された場合（第3.7.5節を参照），それぞれのリクエストごとに，別々のエラーコードを返さなければならない場合もあり得る．これらの情報はMPI_RECV関数のstatus引数で返される．statusの型は，MPIで定義される．ステータス型変数は，ユーザによって明示的に割り当てられなければならない．つまりこれらはシステムオブジェクトではない．

C言語の場合，statusはMPI_SOURCE，MPI_TAG，MPI_ERRORという名前の3つのフィールドを含む構造体になる．その構造体には追加されたフィールドが含まれることもある．そのため，status.MPI_SOURCE，status.MPI_TAG， status.MPI_ERRORには，それぞれ受信メッセージの送信元，タグ，エラーコードが含まれる．

Fortran言語の場合，statusは，MPI_STATUS_SIZEのサイズを持つINTEGERの配列になる．3つの定数MPI_SOURCE，MPI_TAG，MPI_ERRORはそれぞれ送信元，タグ，エラーのフィールドが格納されているエントリを示す添字である．そのため，status(MPI_SOURCE)，status(MPI_TAG)，status(MPI_ERROR)には，それぞれ受信メッセージの送信元，タグ，エラーコードが含まれる．

C++言語では，statusオブジェクトは以下のメソッドを使用して処理される．

{int MPI::Status::Get_source() const （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::Status::Set_source(int source) （廃止された呼び出し形式, 第15.2節を参照）}
{int MPI::Status::Get_tag() const （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::Status::Set_tag(int tag) （廃止された呼び出し形式, 第15.2節を参照）}
{int MPI::Status::Get_error() const （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::Status::Set_error(int error) （廃止された呼び出し形式, 第15.2節を参照）}

一般的に，メッセージ通信の呼び出しは，ステータス型変数のエラーコードフィールドを変更しない．このフィールドは，第3.7.5節で説明する複数のステータス型を返す関数でのみ更新される可能性がある．このフィールドは，それらの関数がMPI_ERR_IN_STATUSというエラーコードを返す場合のみ更新される．

> **根拠** ステータス型のエラーフィールドは，MPI_WAITのように，1つのステータス型のみを返す呼び出しの場合必要ない．この場合は関数自体から返された情報と同一だからである．現在のMPIの設計では，このような場合にエラーフィールドをセットするという余分なオーバーヘッドを避けている．このフィールドは，複数のステータス型を返す呼び出しでのみ必要になる．それぞれのリクエストで異なるエラーが発生する可能性があるためである． （**根拠の終わり**）

ステータス型引数はまた，受信したメッセージの長さについての情報を返す．ただし，この情報はステータス型変数のフィールドとして直接得られるわけではなく，この情報を「解読」するためにMPI_GET_COUNTの呼び出しが必要になる．

MPI_GET_COUNT(status, datatype, count)

IN	status	受信関数の返り値（ステータス型）
IN	datatype	受信バッファの各エントリのデータ型（ハンドル）
OUT	count	受信されたエントリの数（整数型）

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

{int MPI::Status::Get_count(const MPI::Datatype& datatype) const （廃止された呼び出し形式, 第15.2節を参照）}

この関数は受信したエントリの数を返す（ここでも**バイト**ではなく，それぞれの**datatype**が示す型での**エントリ**の数を数える）．datatype引数は，status変数をセットした受信呼び出しによって渡された引数と一致しなければならない（後程，第4.1.11節で，MPI_GET_COUNTが特定の状況でMPI_UNDEFINED値を返す場合を考察する）．

根拠 メッセージ通信ライブラリによっては，INOUTのcount，tag，source引数を使用するものがある．つまりこれらの引数を，入ってくるメッセージの選択基準の指定と，受信メッセージの実際のエンベロープの戻り値の両方で使用する．ステータス型を示す異なった引数を使用することにより，INOUT引数を用いる場合に発生することの多いエラーを回避することができる（例えば，MPI_ANY_TAG定数を受信側のタグとして使用する場合など）．ライブラリによっては，暗黙的に「最後の受信メッセージ」を用いる呼び出しを使用するものがあるが，これはスレッドセーフではない．

性能を向上させるために，datatype引数がMPI_GET_COUNTに渡される．メッセージ中に含まれる要素の数を勘定せずにメッセージを受信することもあり，要素数の値は必要でないことが多い．また，こうすることにより，同じ関数をMPI_PROBEまたはMPI_IPROBEの呼び出し後に使用することができる．MPI_PROBEまたはMPI_IPROBEから返されるステータス型を使用して，MPI_RECVの呼び出しに同じデータ型を指定してこのメッセージを受信することができる． （**根拠の終わり**）

0バイトが転送された長さ0のデータ型ではMPI_GET_COUNTのcount引数として返される値は0となる．転送されたバイト数が0より大きい場合，MPI_UNDEFINEDが返される．

根拠 長さが0のデータ型は多くの場合に作成される可能性がある．重要なのはMPI_TYPE_CREATE_DARRAYの場合で，特定の分散配列の定義により一部のMPIプロセスで空のブロックが生成されることになる．SPMDスタイルで記述されたプログラムはこの特別なケースをチェックしないため，ステータス型をチェックするにはMPI_GET_COUNTを使用する． （**根拠の終わり**）

ユーザへのアドバイス 受信に必要なバッファサイズはデータ変換と受信データ型のストライドの影響を受ける可能性がある．ほとんどの場合，最も安全な方法はMPI_GET_COUNTと受信で同じデータ型を使用することである． （**ユーザへのアドバイス終わり**）

全ての送受信関数で，buf，count，datatype，source，dest，tag，comm，status引数が，この節で説明したブロッキング関数MPI_SEND，MPI_RECVと同様に使用される．

3.2.6 Status用のMPI_STATUS_IGNOREの受け渡し

全てのMPI_RECVの呼び出しではstatus引数が使用され，これによってシステムは受信したメッセージに関する詳細情報を返すことができる．これ以外にも，statusが返されるMPI呼び出しが多数ある．MPI_STATUS型のオブジェクトはMPI不可視オブジェクトではなく，構造体が`mpi.h`および`mpif.h`で宣言され，ユーザのプログラム内に存在する．多くの場合，アプリケーションプログラムは`status`フィールドを検査する必要がないように構成されている．このような場合，ユーザがステータス型オブジェクトを割り当て

ることは無駄なことで，MPI実装でこのオブジェクトのフィールドを記入することは特に無駄なことである．

この問題に対処するため，2つの定義済みの構造体MPI_STATUS_IGNOREおよびMPI_STATUSES_IGNOREが用意されている．これらを受信，ウェイト，テストの関数に渡すと，ステータス型フィールドに記入しないことが処理系に通知される．ただし，MPI_STATUS_IGNOREはMPI_STATUSオブジェクトの特別な型ではなく，その引数のための特別な値である．C言語ではこれが特別なMPI_STATUSのアドレスではなく，NULLになると考えることができる．

MPI_STATUS_IGNOREと配列バージョンのMPI_STATUSES_IGNOREは，受信，ウェイト，テストの関数にstatus引数が渡される場合には使用することができる．MPI_STATUS_IGNOREはstatusがIN引数の場合には使用できない．ただし，Fortran言語ではMPI_STATUS_IGNOREとMPI_STATUSES_IGNOREはMPI_BOTTOMのようなオブジェクトである（初期化や代入には使用できない）．第2.5.4節を参照すること．

一般的に，この最適化はstatusまたはstatus配列がOUT引数である全ての関数に適用することができる．ただし，ここではstatusがINOUT引数に変換される．MPI_STATUS_IGNOREを渡すことができる関数は全て，MPI_RECV，MPI_TEST，MPI_WAIT，MPI_REQUEST_GET_STATUSの各種形態である．配列が渡されると，MPI_{TEST|WAIT}{ALL|SOME}関数と同様に，配列引数用に別の定数MPI_STATUSES_IGNOREが渡される．MPI関数は，MPI_STATUS_IGNOREまたはMPI_STATUSES_IGNOREがその関数に渡されている場合でも，MPI_ERR_IN_STATUSを返すことができる．

MPI_STATUS_IGNOREとMPI_STATUSES_IGNOREは，C言語とFortran言語で同じ値を持つ必要はない．

MPI_{TEST|WAIT}{ALL|SOME}関数で status配列のステータス型の一部をMPI_STATUS_IGNOREに設定することはできず， MPI_STATUSES_IGNOREを使用して呼び出しで**全ての**ステータスを無視するよう指定するか，status配列の全ての位置で正常なステータスを渡すことにより**1つも**無視しないように指定する．

MPI_STATUS_IGNOREまたはMPI_STATUSES_IGNOREのためのC++言語の呼び出し形式はない．OUTまたはINOUTの`MPI::Status`引数を無視できるようにするため，OUTまたはINOUTの`MPI::Status`パラメータを持つ全てのMPIC++言語の呼び出し形式に，OUTまたはINOUT`MPI::Status`パラメータを削除した第2のバージョンのものがオーバーロードされる．

例 3.1 MPI_PROBEのためのC++言語の呼び出し形式は以下のようになる．

```
void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
void MPI::Comm::Probe(int source, int tag) const
```

3.3 データ型の一致とデータ変換

3.3.1 型一致規則

メッセージ転送を，次の3つのフェーズで構成されるものと考えることができる．

1. データが送信バッファから引き出され，メッセージが組み立てられる．
2. メッセージが送信側から受信側へ転送される．
3. 入ってくるメッセージからデータが取りだされ，分解されて受信バッファに入れられる．

この3つのフェーズで型一致を守らなければならない．送信バッファのそれぞれの変数の型が，送信関数でのエントリで指定された型と一致しなければならない．また送信関数で指定された型は，受信関数で指定された型と一致しなければならない．さらに，受信バッファのそれぞれの変数の型も，受信関数のエントリで指定された型と一致しなければならない．これらの3つの規則が守られていないプログラムは，誤りである．

型一致をもっと正確に定義するためには，ホスト言語の型と通信関数で指定した型との一致，および送信側と受信側の型の一致という，2つの問題を扱う必要がある．

送信と受信の型は，両方の関数の型の名前が同一であれば一致する（フェーズ 2）．つまりMPI_INTEGERはMPI_INTEGERと一致し，MPI_REALはMPI_REALと一致するということである．この規則には1つだけ例外があり，第4.2節で説明したMPI_PACKED型は他のどんな型とも一致させることができる．

ホストプログラムの変数の型は，通信関数で使用されているデータ型名が，ホストプログラム変数の基本型と対応する場合，その関数で指定した型と一致する．例えば，型名MPI_INTEGERを持つエントリは，`INTEGER`型のFortran言語変数と一致する．Fortran言語とC言語におけるこの対応を示す表が第3.2.2節にある．この規則にも例外が2つある．型名MPI_BYTEやMPI_PACKEDは，データ記憶領域の任意のバイトと対応するために使用されるものであり（バイトによるアドレス指定可能マシンの場合），このバイトを含む変数のデータ型とは関係ない．第4.2節に示すように，MPI_PACKED型は，明示的にパックするデータを送信したり，明示的にアンパックされるデータを受信したりするときに使用される．MPI_BYTE型は，メモリ内の任意のバイトのバイナリ値を変更なしに転送するときに使用される．

以上をまとめると，型一致規則は以下の3つのカテゴリに分類される．

- 型の指定された値の通信（すなわちMPI_BYTEデータ型以外を使用する場合）では，送信側のプログラム，送信呼び出し，受信呼び出し，受信側のプログラムでの対応するエントリのデータ型は，全て一致しなければならない．
- 型の指定されない値の通信（すなわちMPI_BYTEデータ型を使用する場合），すなわち送信側と受信側の両方がMPI_BYTEデータ型を使用する場合では，送信側プログラムと受信側プログラムでの，対応するエントリの型になんらの要件もなく，それらが同じである必要もない．

- MPI_PACKEDが使用されている，パックされたデータを含む通信.

以下の例で，最初の2つの場合を説明する.

例 **3.2** 送信側と受信側が一致した型を指定する場合

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

aとbの両方が，10以上の大きさの実数型配列の場合，このプログラムは正しい.
（Fortran言語の場合，例えばa(1)が10個の実数要素を持つ配列に記憶領域をEQUIVALENCEさせることができる場合など，aとbのいずれかのサイズが10より小さくても，このプログラムを使用することは正しい時がある.）

例 **3.3** 送信側と受信側の型が一致しない場合

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

このプログラムは，送信側と受信側が一致するデータ型引数を指定しないため， 誤りである.

例 **3.4** 送信側と受信側が型指定されない値での通信をした場合

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

このプログラムは，aとbの型とサイズにかかわらず正しい（ それぞれの変数に割り当てられたメモリ領域の境界を超えたメモリアクセスが生じない場合）.

ユーザへのアドバイス MPI_BYTE型のバッファがMPI_SEND関数の引数として渡された場合，MPIはbuf引数によって示されたアドレスから始まる，連続した領域に格納されたデータを送る．これを行うと，通常のユーザが期待するものとデータ配置が違っている場合，予期しない結果になることがある．例えば，Fortran言語コンパイラの中には，CHARACTER型の変数を，文字の長さと，実際の文字列に対するポインタを含む構造体として実装するものがある．このような環境では，MPI_BYTE型を使用して，Fortran言語のCHARACTER型の変数を送受信すると，文字列を転送したときの予想結果と違う結果になることがある．このためユーザに対しては，可能な限り型指定の通信を使用するように奨める． （**ユーザへのアドバイス終わり**）

MPI_CHARACTER型

MPI_CHARACTER型は，CHARACTER型のFortran言語変数に格納された文字列全体ではなく，その変数の中の1文字と対応する．CHARACTER型のFortran言語の変数または部分文字列は，文字の配列であるかのように転送される．これについては以下の例で説明する．

例 **3.5** Fortran言語のCHARACTERの転送

```
CHARACTER*10 a
CHARACTER*10 b

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF
```

プロセス1における文字列bの最後の5文字が，プロセス0における文字列aの最初の5文字で置換される．

根拠 もう1つの選択肢は，MPI_CHARACTERを任意の長さの文字列と一致させることである．こうした場合には問題が発生する．

Fortran言語の文字変数は，一定の長さの文字列で，特殊な終端記号がない．文字を表現するときの決まった規則はなく，その長さを格納する方法に関する規則もない．コンパイラによっては，文字引数を一対の引数としてルーチンに渡すものもある．この場合，1つには文字列のアドレスが含まれ，もう1つには文字列の長さが含まれる．派生データ型で定義された型（第4.1節）を含む通信バッファが，あるMPI通信呼び出しに渡される場合を想定してみよう．この通信バッファにCHARACTER型が含まれている場合，これらの文字列の長さの情報がMPIルーチンに渡されることはない．

この問題があるために，MPI呼び出しでは，文字列の長さに関する明示的な情報を提供しなければならない．MPI_CHARACTER型に長さのパラメータを追加することはできるが，これはあまり便利ではなく，適切な派生データ型を定義することによって同じ機能を実現することができる． （**根拠の終わり**）

実装者へのアドバイス コンパイラによっては，Fortran言語のCHARACTER引数を，長さと実際の文字列を示すポインタを持つ構造体として渡すものもある．このような環境では，文字列に至るために，MPI呼び出しでポインタの参照解決をおこなう必要がある． （**実装者へのアドバイス終わり**）

3.3.2 データ変換

MPIの目標の1つは，異機種環境で並列計算をサポートすることである．異機種環境で通信を行うときは，データ変換が必要になる場合がある．ここでは以下の用語を使用する．

型変換 値のデータ型を変更すること．例えば，REALを丸めてINTEGERに変換すること．

表現変換 値の2進表現を変更すること．例えば，16進浮動小数点数からIEEE浮動小数点数に変換すること．

型一致規則により，MPI通信では型変換は伴わない．それに対して，ある型の値の表現形式が異なるような環境の間でこの型の値が転送される場合には，MPI通信での表現変換が必要になる．MPIでは表現変換の規則を規定していない．そのような変換では，整数，論理，または文字としての値を維持することと， 浮動小数点数としての値を，相手側のシステムで表現できる最も近い値に変換することが必要である．

浮動小数点数変換の場合，オーバーフローやアンダーフローの例外が発生することもある．整数や文字の変換でも，あるシステムで表現できる値が別のシステムで表現できないとき，例外が発生することがある．表現変換のときに発生する例外は，結果的に通信時のエラーになる．エラーは，送信操作と受信操作のいずれか，またはその両方で発生する．

メッセージで送信される値が，型の指定されない値の場合（つまりMPI_BYTE型），受信側に格納されるそのバイトの2進表現は，送信側でロードされるバイトの2進表現と同一である．送信側と受信側が同じ環境で動作していても，あるいは違う環境で動作していても，これは真になる．表現変換は必要ない（MPI_CHARACTER型やMPI_CHAR型の値が転送されるときは，表現変換が発生し得ることに注意，例えばEBCDIC符号からASCII符号への変換）．

全てのプロセスが同じ環境で動作するような，同機種環境でMPIプログラムが実行されるとき，変換を行う必要はない．

3.2から3.4までの3つの例題を考えてみる．aとbが10以上のサイズを持つREALの配列である場合，最初のプログラムは正しい．送信側と受信側が異なる環境で実行する場合，送信バッファから取り出される10個の実数値が，受信バッファに格納される前に，受信側の実数表現に変換される．送信バッファから取り出される実数の要素の数が，受信バッファに格納される実数の要素の数と等しくても，格納されるバイト数はロードされるバイト数と同じである必要はない．例えば，送信側が実数に4バイトの表現を使用し，受信側が8バイトの表現を使用することもある．

2番目のプログラムは誤りであり，その動作は保証されない．

3番目のプログラムは正しい．送信側と受信側が異なる環境で動作しても，送信バッファからロードされたのとまったく同じ40バイトの列が受信バッファに格納される．送られるメッセージは，受信されるメッセージとまったく同じ長さ（バイトで）と同じ2進表現を持っている．aおよびbが異なる型の場合，または同じ型であっても異なるデータ表現が使用されている場合，受信バッファ に格納されたビット列は，送信バッファで符号化された値とは異なる値を符号化することがある．

データの表現変換は，メッセージのエンベロープにも適用される．送信元，送信先，タグは全て整数であり，変換が必要な場合がある．

実装者へのアドバイス 現在の定義では，メッセージにデータ型の情報を持つことを要求していない．送信側と受信側の両方が，データ型に関する完全な情報を与え

る．異機種環境では，XDRのような機種非依存のコード変換を使用するか，受信側で送信側の表現を独自の表現に変換するか，さらには送信側で変換を行うかのいずれかになる．システムで送信側と受信側のデータ型の間の不一致を検出するようにするため，その他の型情報をメッセージに追加することができる．これは，遅いがより安全という意味で，デバッグ時に特に便利である可能性がある． （**実装者へのアドバイス終わり**）

MPIは言語間の通信をサポートする必要がある．つまり，C言語またはC++言語 のプロセスによって送信されたメッセージをFortran言語のプロセスが受信したり，この逆方向で送受信したりするのをサポートする必要がある．この動作については519ページの第16.3節で定義している．

3.4 通信モード

第3.2.1節で説明した送信呼び出しは，**ブロッキング**である．つまり，メッセージデータとエンベロープが安全に格納されて，送信側が送信バッファを自由に修正できるようになるまでは，この呼び出しは戻らない．メッセージは，対応する受信バッファに直接コピーされるか，もしくは，一時的なシステムバッファにコピーされる．

メッセージのバッファリングは，送信操作と受信操作を独立なものとする．ブロッキング送信は，マッチする受信が受信側で実行されていなくても，メッセージがバッファリングされればすぐに完了することができる．一方で，メッセージバッファリングは，新たなメモリ間コピーを伴い，また，バッファリングのためのメモリ割り当てを必要とするのでコストが高くなる可能性がある． MPIはいくつかの通信モードの選択肢を提供しており，これにより，通信プロトコルの選択を制御することができる．

第3.2.1節で説明した送信呼び出しは，**標準**送信モードを使用する．このモードでは，送信メッセージがバッファリングされるかどうかを決定するのはMPIに任せている．MPIは，送信メッセージをバッファリングすることもできる．このような場合には，送信呼び出しは，マッチする受信が起動される前に完了することができる．一方で，バッファ領域が使用できない場合もあるし，MPIが性能上の理由から，送信メッセージをバッファリングしないことを選択する場合もある．この場合には，送信呼び出しは，マッチする受信が発行され，データが受信側に移動してしまうまでは完了しない．

このように，標準モードでの送信は，マッチする受信が発行されているかどうかにかかわらず，開始することができる．これは，マッチする受信が発行される前に完了することもある．標準モード送信は**非ローカル**である．つまり，送信操作が成功完了するかどうかは，マッチする受信が発生するかどうかに依存する場合がある．

根拠 MPIにおいて，標準送信がバッファリングするかどうかを指定しないのは，可搬なプログラムを作るという要求に起因している．メッセージサイズが増えるに従って，どのシステムもバッファ資源を使い尽くすため，また，ある実装では，バッファリングをほとんど提供しないために，MPIは正しい（したがって可搬な）プログラムは標準モードでシステムバッファリングに依存しないという立場をとる．

バッファリングは正しいプログラムの性能を向上させる場合もある．しかし，プログラムの結果には影響を与えない．もし，ユーザがある量のバッファリングを保証したければ，バッファモードの送信と共に，第3.6節のユーザ提供バッファシステムを使用するべきである． （**根拠の終わり**）

他に，3つの追加的な通信モードが存在する．

バッファモード送信操作は，マッチする受信が発行されているかどうかにかかわらず，開始することができる．これは，マッチする受信が発行される前に完了する可能性もある．しかしながら，標準送信とは異なり，この操作は**ローカル**である．そして，その完了は，マッチする受信が発生するかどうかに依存しない．したがって，送信が実行され，マッチする受信が発行されない場合には，MPIは送信呼び出しが完了できるように送信メッセージをバッファリングしなければならない．バッファ領域が不十分な場合には，エラーが発生する．利用可能なバッファ領域の量はユーザが制御する（第3.6節参照）．バッファモードが有効に働くためには，ユーザによるバッファ割り当てが必要になる．

同期モードを使用する送信は，マッチする受信が発行されているかどうかにかかわらず，開始することができる．しかしながら，送信は，マッチする受信が発行され，そして，受信操作が同期送信によって送信されたメッセージを受信開始した場合にのみ成功完了する．このように，同期送信の完了は，送信バッファ が再利用できることを示すだけでなく，受信側が実行におけるあるポイントに到達したこと，つまり，マッチする受信の実行が開始されたことを示す．送信および受信の両操作がブロッキング操作の場合，同期モードの使用は同期通信を意味する．つまり通信は，両方のプロセスが通信で出会う前にはどちら側も完了しない．このモードで実行される送信は，非ローカルである．

レディ通信モードを使用する送信は，マッチする受信がすでに発行されている場合に**のみ**，開始することができる．それ以外の場合は，この操作は誤りであり，その結果は保証されない．システムによっては，レディ通信は，他の通信で必要なハンドシェイク操作をなくすことが可能となり，性能改善をもたらす． 送信操作の完了は，マッチする受信の状態には依存せず，単に送信バッファが再使用できることを意味する．レディモードを使用する送信操作は，標準送信操作あるいは同期送信操作と意味的には同じである．単に送信側が（マッチする受信が既に発行されているという）追加情報をシステムに提供し，ある種のオーバーヘッドを抑えるということにすぎない．したがって，正しいプログラムにおいては，性能を除くプログラムの動作に影響を与えることなく，レディ送信を標準送信で置き換えることができる．

これら3つの追加的な通信モードに対して，3つの追加的な送信関数が提供されている．通信モードは1文字の接頭語で表される．Bはバッファ，Sは同期，そしてRはレディである．

MPI_BSEND (buf, count, datatype, dest, tag, comm)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファ内の要素数（非負の整数型）
IN	datatype	各送信バッファ要素のデータ型（ハンドル）
IN	dest	送信先先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
{void MPI::Comm::Bsend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

バッファモードの送信.

MPI_SSEND (buf, count, datatype, dest, tag, comm)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファ内の要素数（非負の整数型）
IN	datatype	各送信バッファ要素のデータ型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
{void MPI::Comm::Ssend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

同期モードの送信.

MPI_RSEND (buf, count, datatype, dest, tag, comm)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファ内の要素数（非負の整数型）
IN	datatype	各送信バッファ要素のデータ型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
{void MPI::Comm::Rsend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

レディモードの送信.

受信操作は一つだけであり，どの送信モードにもマッチすることができる．前節で記述された受信操作は**ブロッキング**である．つまり，受信バッファが新しく受信したメッセージを取り込んだ後にのみ戻る．受信は，マッチする送信が完了する前に，完了することができる．（当然，それはマッチする送信が開始した後にのみ戻ることができる．）

MPIのマルチスレッド実装では，システムは，送信や受信操作でブロッキングされたスレッドを待機させ，同じアドレス空間の別のスレッドをスケジュールすることができる．このような場合には，通信が完了するまでユーザの責任において通信バッファを変更しないようにする必要がある．そうしないと，計算の結果は保証されない.

実装者へのアドバイス 同期送信は，マッチする受信が発行される前には完了できないので，そのような操作により送信されたメッセージのバッファリングは通常行わない.

標準送信に対しては，できるだけ，送信側をブロッキングするよりも，バッファリングを選択することが推奨される．プログラマは，同期送信モードを使用することによって，マッチする受信が発生するまで送信側をブロッキングしたい意向を明示することができる.

さまざまな通信モードで使用可能な通信プロトコルを以下に概説する.

レディ送信：メッセージはできるだけ早く送られる.

同期送信：送信側は送信要求メッセージを送信する．受信側はこの要求を保存する．マッチする受信が発行されたときに，受信側は送信許可メッセージを送り返し，送信側がメッセージを送信する.

標準送信：短いメッセージに対しては第1の送信プロトコルが使用され，長いメッセージに対しては第2のプロトコルが使用される.

バッファ送信：送信側はメッセージをバッファにコピーし，それを（標準送信と同じプロトコルを使用して）ノンブロッキング送信により送信する.

フロー制御とエラー回復のためには，追加的な制御メッセージが必要になると考えられる．当然，その他にも可能なプロトコルは多数ある.

レディ送信は，標準送信として実装可能である．この場合には，レディ送信を使用することによる性能上の利点（あるいは欠点）はない.

標準送信は，同期送信として実装可能である．この場合には，データバッファリングは必要ない．しかし，ユーザはバッファリングを利用することもできる.

マルチスレッド環境では，ブロッキング通信の実行は，スレッドスケジューラが実行スレッドを待機させ，別のスレッドの実行のスケジュールを許可することで，実行スレッドのみをブロッキングする必要がある．

（実装者へのアドバイス終わり）

3.5 1対1通信の意味論

正しいMPI実装は，1対1通信のある種の一般的な性質を保証しており，この節ではこれについて説明する．

順序 メッセージは**追越し禁止**である：送信側が2つのメッセージを続けて同じ送信先に送信し，両方の送信が同じ受信とマッチする場合には，第1のメッセージがまだ保留中の状態であれば，この受信操作は第2のメッセージを受信することができない．受信側が2つの受信を続けて発行し，両方が同じメッセージにマッチする場合には，第1の受信がまだ保留中の状態であれば，第2の受信操作はこのメッセージに対応できない．この要件により，送信と受信のマッチが容易になる．このことは，プロセスが単一スレッドでしかも受信側においてワイルドカードMPI_ANY_SOURCEが使用されていない場合に，メッセージ通信コードの決定性を保証する．（後で述べる，MPI_CANCELやMPI_WAITANYのようなある種の呼び出しは，追加的な非決定性のもととなる．）

プロセスが単一スレッドの実行の場合には，このプロセスによって実行されるどの2つの通信も順序が決まっている．一方，プロセスがマルチスレッドの場合には，スレッド実行が意味するところでは，2つの違ったスレッドにより実行された2つの送信操作の間の相対的な順序は不定となる．たとえ，1つのスレッドが物理的に他よりも先行していたとしても，それらの操作は論理的には並行している．このような場合には，送信された2つのメッセージは任意の順序で受信されうる．同様に，論理的には並行している2つの受信操作が連続で送信された2つのメッセージを受信する場合には，これらの2つのメッセージは2つの受信といずれの順序でもマッチされうる．

例 3.6 追越し禁止メッセージの例

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

第1の送信により送信されたメッセージは，第1の受信により受信されなければならない，また，第2の送信により送信されたメッセージは，第2の受信により受信されなければならない．

プログレス マッチする送信と受信の1対が2つのプロセスで起動された場合には，システムの他の動作とは独立に，これら2つの操作のうち少なくとも1つは完了する．つまり，受信が別のメッセージによって完了する，ということが起こらない場合には，送信操作が完了し，送信されたメッセージが同じ受信先で発行されたマッチする別の受信により消費される，ということが起こらない場合には，受信操作が完了する．

例 3.7 交差してマッチする2つの対の例

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
    CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

両プロセスが第1の通信呼び出しを起動する．プロセス0の第1の送信はバッファモードを使用しているため，プロセス1の状態とは無関係に完了しなければならない．マッチする受信が発行されないので，メッセージはバッファ領域にコピーされる．（充分なバッファ領域がない場合にはこのプログラムはエラーとなる．）次に，第2の送信が起動される．この時点において，マッチする1対の送信操作と 受信操作が開始可能となり，両操作は完了しなければならない．次にプロセス1は，第2の受信呼び出しを起動し，これはバッファリングされたメッセージとマッチする．プロセス1が，送信されたのとは逆順にメッセージを受信したことに注意すること．

公平性 MPIは通信の取り扱いの**公平性**をなんら保証しない．送信が発行された場合を考えてみよう．この時，送信先のプロセスが，この送信にマッチする受信を繰り返し発行するのだが，その度に別の送信元から送信された別のメッセージにより追い越されるために，このメッセージが受信されないといったことが起こりうる．同様に，受信がマルチスレッドプロセスによって発行された場合を考えてみよう．この受信にマッチするメッセージが繰り返し受信されるのだが，このノード（で実行中の他のスレッド）で発行された他の受信により追い越されるために，この受信が完了しないといったことも起こりうる．このような状況を回避するのは，プログラマの責任である．

資源制限 保留中の通信操作は限りあるシステム資源を消費している．資源の不足により， MPI呼び出しの実行ができない場合には，エラーが起こりうる．高品質な実装では，それぞれのレディモードや同期モードでの保留中の送信及び保留中の 受信に対して，（小さな）一定量の資源を使用する．しかし，バッファ領域は， マッチする受信がない場合に標準モードで送信されたメッセージを格納するために消費されうるし，バッファモードで送られてきたメッセージを格納するためには必ず消費される．多くのシステムにおいては，バッファリングに利用できる領域量は，プログラムデータメモリに比べて非常に小さくなっている．そのため，利用可能バッファ領域を使い尽くすプログラムになりやすい．

MPIでは，バッファモードで送られるメッセージのためのバッファメモリをユーザが用意することができる．さらにMPIは，このバッファ使用に対する詳細な操作モデルを指定している．MPIの実装では，このモデルで示される以上の事が要求される．これにより，ユーザはバッファ送信の使用時にバッファオーバーフローを避けることができる．バッファの割り当てと使用法は第3.6節で説明する．

バッファ領域の不足により完了できないバッファ送信は誤りである．その様な状況が検知されれば，プログラムが異常終了する可能性があるというエラーが出される．一方，バッファ領域の不足により完了できない標準送信操作は，バッファ領域が利用可能になるか，マッチする受信が発行されるのを待ちながら，単にブロッキングする．この動作は，多くの状況において望ましいものである．生産者側が，繰り返し新しい値を作成して，それらを消費者側に送信している状況を考えてみよう．生産者側が，消費者側の消費できる新しい値をより速く作成すると仮定する．バッファ送信が使用された場合には，バッファのオーバーフローが生じると考えられる．これが発生するのを防ぐためには，他の同期をプログラムに加える必要がある．標準送信が使用される場合には，バッファ領域が使用できなければ，送信操作がブロッキングし，生産者側は自動的に速度調整される．

状況によっては，バッファ領域の不足はデッドロック状況を招く．これについては，以下の例で説明する．

例 3.8 メッセージの交換

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

このプログラムは，データのためのバッファ領域が使用できなくても成功する．この例では，標準送信操作は同期送信で置き換えることができる．

例 3.9 誤った メッセージ交換の試み

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

第1のプロセスの受信操作はその送信の前に完了しなければならず，これは第2のプロセスのマッチする送信が実行された場合にのみ完了することができる．第2のプロセスの受信操作はその送信の前に完了しなければならず，これは第1のプロセスのマッチする送信が実行された場合にのみ完了することができる．このプログラムは常にデッドロックする．他の送信モードに対しても，同じ事が起こる．

例 3.10 バッファリングに依存した交換

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

各プロセスにより送信されるメッセージは，送信操作が戻り，また，受信操作が開始する前にコピーアウトされなければならない．プログラムが完了するためには，送信される2つのメッセージのうち少なくとも1つはバッファリングされることが必要である．したがって，このプログラムは，通信システムが少なくともcount個のワードデータをバッファリングできる場合にのみ成功する．

ユーザへのアドバイス 標準送信が使用されるときには，バッファ領域が使用できないために両方のプロセスがブロッキングされる場合に，デッドロック状況が発生する．同期モードが使用される場合にも，同様のことが必ず起こる．バッファモードが使用され，十分なバッファ領域が利用できない場合には，プログラムは完了しない．しかしながら，デッドロック状況は起こらず，バッファオーバーフローエラーになる．

プログラムが完了するために，メッセージのバッファリングが必要ない場合には，プログラムは「安全」である．そのようなプログラムでは全ての送信を同期送信に置き換えることができ，しかもプログラムは正しく実行される．この保守的なプログラミングスタイルは，最良の可搬性を提供する．これは，プログラムの完了が，使用可能なバッファ領域の量や，使用される通信プロトコルに依存しないからである．

多くのプログラマは高い自由度を好み，例3.10に示されるような，「危険な」プログラミングスタイルを使用しようとする．そのような場合には，標準送信を使用することにより性能と頑健さの最良の妥協点を見出すことができる．つまり，高品質な実装では「通常の」プログラムがデッドロックしないように，十分なバッファリングが提供される．バッファ送信モードは，より多くのバッファリングを必要とするプログラムに対して，またプログラマが制御を強化したい状況において使用できる．このモードは，バッファオーバーフロー状況の方がデッドロック状況よりも診断しやすいため，デバッギングの目的のためにも使用されうる．

第3.7節で説明するように，送信メッセージのバッファリングの必要性を避けるために，ノンブロッキングメッセージ通信操作を使用することができる．このことにより，バッファ領域の不足によるデッドロックを回避でき，計算と通信との重ね合わせを可能にし，また，バッファの割り当てとバッファへのメッセージのコピーによるオーバーヘッドを回避することにより，性能を改善する． （**ユーザへのアドバイス終わり**）

3.6 バッファの割り当てと使用法

ユーザは，バッファモードで送られるメッセージのバッファリングに使用されるバッファを指定することができる．バッファリングは送信側で行われる．

MPI_BUFFER_ATTACH(buffer, size)

IN	buffer	バッファの先頭アドレス（選択型）
IN	size	バッファサイズ（バイト）（非負の整数型）

```
int MPI_Buffer_attach(void* buffer, int size)
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
{void MPI::Attach_buffer(void* buffer, int size)} （廃止された呼び出し形式, 第15.2節を参照）
```

送信メッセージのバッファリングに使用するユーザメモリ内のバッファをMPI に与える．このバッファは，バッファモードで送られるメッセージによってのみ使用される．同時には，ただ1つのバッファのみが1つのプロセスに結びつけられる．

MPI_BUFFER_DETACH(buffer_addr, size)

OUT	buffer_addr	バッファの先頭アドレス（選択型）
OUT	size	バッファサイズ（バイト）（非負の整数型）

```
int MPI_Buffer_detach(void* buffer_addr, int* size)
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    <type> BUFFER_ADDR(*)
    INTEGER SIZE, IERROR
{int MPI::Detach_buffer(void*& buffer)} （廃止された呼び出し形式, 第15.2節を参照）
```

現在MPIに関連付けれらたバッファを切り離す．この呼び出しは，切り離されたバッファのアドレスとサイズを返す．現在バッファにある全てのメッセージが転送されるまで，この操作はブロッキングする．この関数が戻った時点で，ユーザはこのバッファにより占められた領域を再利用したり，解放したりできる．

例 3.11 バッファの結びつけおよび切り離しのための呼び出し

```
#define BUFFSIZE 10000
int size;
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach( &buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */
```

ユーザへのアドバイス C言語関数MPI_Buffer_attachとMPI_Buffer_detachは両方とも第1引数にvoid*型を持つが，これらの引数は異なった方法で使われる．つまり，MPI_Buffer_attachへはバッファを指すポインタが渡され， MPI_Buffer_detachへは，この呼び出しがポインタ値を返せるように，ポインタのアドレスが渡される．（**ユーザへのアドバイス終わり**）

根拠 複雑な型変換を避けるため，両引数はvoid*型として定義されている（それぞれがvoid*とvoid**となるのではなく）．例えば，最後の例では，char**型をもつ&buffは，型変換される事なく引数としてMPI_Buffer_detach に渡される．正式な引数がvoid**型を持つ場合には，呼び出しの前後で型変換を行う必要がある．（**根拠の終わり**）

この節では，バッファモード送信におけるMPIの動作について説明する．現在バッファが付随していない場合，MPIはプロセスにサイズ0のバッファが付随しているかのように動作する．

MPIは，送信メッセージデータが，送信プロセスにより指定されたバッファ領域へ巡回的な連続領域割り当ての原則に従ってバッファリングされるかのように，送信メッセージに必要な量のバッファを，送信メッセージに対して，提供しなければならない．以下では，この原則を定義するモデル実装について概説する．MPIは，より多くのバッファリングを提供することもあり，以下に記述されるものより優れたバッファ割り当てアルゴリズムを使用することもある．一方，MPIは以下に記述した単純バッファアロケータが領域を使いきった時にはいつでもエラーシグナルを返す可能性がある．特に，プロセスに明示的に付随したバッファがない時には，バッファ送信はエラーを引き起こす可能性がある．

MPIは，標準モード送信で実行されるバッファリングについて，問い合わせや制御のためのメカニズムを用意していない．そのような情報は，ベンダーが実装ごとに提供することが望ましい．

根拠 バッファ通信の可能な実装は，多岐にわたる．バッファリングは，送信側，受信側あるいは両方で行うことができるし，バッファを1つの送信/受信のペア専用にすることも，全ての通信で共有することもできるし，バッファリングを実メモリかあるいは仮想メモリで行うこともできるし，専用のメモリ を使ったり，他のプロセスとの共有メモリを使ったりすることもできるし，バッファを静的に割り当てたり，動的に変更したりすることもできる． これら全ての選択に互換性をもつような，バッファリングについての問い合わせや制御を行い，なおかつ有益な情報を提供するような可搬なメカニズムを提供することは容易ではない． （**根拠の終わり**）

3.6.1 バッファモードのモデル実装

モデル実装は，第4.2節で説明するパッキング関数とアンパッキング関数，そして第3.7節で説明するノンブロッキング通信関数を使用する．

保留中のメッセージエントリ（pending message entries＝PME）の循環キューが管理されると仮定する．各々のエントリは，保留中のノンブロッキング送信を識別する通信リクエストハンドル，次のエントリへのポインタおよびパックされたメッセージデータなどを含んでいる．エントリは，バッファ内の連続した位置に格納される．キューの末尾とキューの先頭の間には，空き領域があってもよい．

バッファ送信呼び出しは，以下のコードのように実行されることになる．

- 先頭から末尾に向かって，まだ完了していないリクエストの最初のエントリに達するまで，すでに完了した通信の全てのエントリを消去しながら，PMEキューを順に移動する．つまり，そのエントリを指すようにキューの先頭を更新する．
- 新しいメッセージのエントリを格納するために必要なバイト数nを計算する．上限は，以下のようにして計算できる．MPI_BSEND呼び出しで使用されるcount，datatype，comm引数を持つMPI_PACK_SIZE(count, datatype, comm, size)関数の呼び出しは，メッセージデータのバッファリングに必要な領域の上限を返す（第4.2節参照）．MPI定数MPI_BSEND_OVERHEADは，このエントリ（例えば，ポインタやエンベロープ情報）で消費される追加領域の上限を与える．
- バッファ内（キューの末尾に続く領域，あるいは，キューの末尾がバッファの最後に近い場合にはバッファの最初の領域）で次の連続するnバイトの空き領域を探す．領域が見つからない場合には，バッファオーバーフローエラーを出す．
- PMEキューの最後に，リクエストハンドル，次のポインタ，パックされたメッセージデータを含む新しいエントリを連続した領域で追加する．データのパックには，MPI_PACKが使用される．
- パックされたデータに対し，ノンブロッキング送信（標準モード）を発行する．
- 戻る．

3.7 ノンブロッキング通信

多くのシステムにおいて，通信と計算を重ね合わせることによって性能を向上させることができる．通信が賢い通信コントローラによって自動的に実行できるようなシステムにこのことは特にあてはまる．軽量スレッドはこのような重ね合わせを達成するための一つの機構である．これより高い性能が実現されることの多いもう一つの機構は，**ノンブロッキング**通信を用いる方法である．ノンブロッキング**送信開始**が呼び出されると，送信操作が起動されるが，完了はしない．送信バッファからメッセージがコピーされる前に送信開始の呼び出しは戻ってくることができる．通信を完了させるため，すなわちデータが送信バッファからコピーされたことを確認するためには，別の**送信完了**の呼び出しが必要となる．適切なハードウェアを使えば，送信が初期化されたが完了はしていない時点で，送信側での計算と並行させて送信側のメモリからのデータの転送を行うことができる．同様に，ノンブロッキング**受信開始呼び出し**は 受信演算子を起動するが完

了はしない．この呼び出しはメッセージが受信バッファ に格納される前に戻る．受信操作を完了させ，データが受信バッファに受け取られたことを確認するためには，別の**受信完了**の呼び出しが必要である．適切なハードウェアを使えば，受信が起動され終了する以前に，受信側のメモリへのデータの転送を計算と並行させて行える．ノンブロッキング受信を使用すると， 受信バッファの位置に関する情報は早い時点で提供されているので，システムでのバッファリングや メモリからメモリへのコピーを避けることもできる．

ノンブロッキング送信開始の呼び出しは，ブロッキング送信と同じ**標準**，**バッファ**，**同期**，**レディ**の4つのモードを使用することができる．これらはブロッキング通信のときと同じ意味を持っている．レディ以外の全てのモードでは，マッチする受信が発行されたかどうかにかかわらず送信を開始することができる．ノンブロッキングのレディモードでの送信は，マッチする受信が発行されたときにのみ開始することができる．全ての場合において，送信開始呼び出しはローカルであり，他のプロセスの状態に依らず直ちに戻ってくる．その呼び出しがシステムのいずれかの資源を使い果たしてしまったときは，失敗となりエラーコードが返される．MPIの高品質な実装においては，このようなことは「病的」な場合にのみ起こるようになっていなければならない．すなわちMPIの実装は，多くの保留中のノンブロッキング操作をサポートすることができなければならない． 送信完了の呼び出しは，データが送信バッファからのコピーが終わったときに戻ってくる．このことは送信モードによっては付加的な意味をもつことがある．

送信モードが**同期**なら，送信はマッチする受信が開始したとき，すなわち受信が発行され送信とマッチされたときにのみ完了することができる．この場合には送信完了呼び出しは非ローカルである．ただし，同期ノンブロッキング送信は，ノンブロッキング受信とマッチがとられれば，受信完了呼び出しが発生する前に完了することができる．（送信側が転送が完了することを「知れ」ば，受信側が転送が完了することを「知る」前であってもただちに完了する．）

送信モードが**バッファ**のときは，もし保留中の受信がなければメッセージはバッファリングされなければならない．この場合には送信完了呼び出しはローカルであり，マッチする受信の状態に関係なく成功しなければならない．

送信モードが**標準**のときは，もしメッセージがバッファリングされているなら，送信完了呼び出しはマッチする受信が起こる前に戻ることがある．一方，対応する受信が発生し，メッセージが受信バッファにコピーされるまで送信完了は終了しないこともありうる．

ノンブロッキング送信はブロッキング受信とマッチさせることができ，逆も可能である．

> **ユーザへのアドバイス** 標準モードに対しては送信操作の完了はマッチする受信が発行されるまで遅延されることがある．一方で同期モードに対しては送信操作の完了はマッチする受信が発行されるまで必ず遅延される．この2つの場合においては，ノンブロッキング送信を使用することにより，送信側は受信側より先に進むことができ，2つのプロセスの速度の揺らぎがあっても，計算は影響を受けにくい．

バッファモードとレディモードにおけるノンブロッキング送信はより限定された効果しか持たない．例えば，ブロッキング版のバッファ送信はマッチする受信呼び出しがいつ行われたかに関係なく，完了することができる．しかし，これらの送信の開始と完了を分離することにより，MPIライブラリ内部の最適化のための機会が得られる．例えば，バッファ送信を開始することにより，メッセージをバッファリングするかどうか，またバッファリングの方法を決定する際の実装の柔軟性が高まる．データのコピーと計算を並行して行うことができる場合，ノンブロッキングバッファとレディモードの両方に利点がある．

メッセージ通信モデルは，通信は送信側によって起動されるということを意味している．もし送信側が通信を起動したときに受信がすでに発行されているならば，通信は一般的により小さなオーバーヘッドを持つ．（データを受信側のバッファに直接移動させることができ，保留中の送信リクエストをキューに入れる必要がない．）しかしながら，受信操作はマッチする送信が起こった後にのみ完了することができる． ノンブロッキング受信を使用することにより，送信を待っている間も受信側をブロッキングすることなく，より低い通信オーバーヘッドが実現できる．（**ユーザへのアドバイス終わり**）

3.7.1 通信リクエストオブジェクト

ノンブロッキング通信は，通信操作を識別するために不可視の**リクエスト**オブジェクトを使用し，通信を起動する操作と終了させる操作を対応させる．これらはハンドルを通してアクセスされるシステムオブジェクトである．リクエストオブジェクトは，送信モード，それに対応した通信バッファ，そのコンテクスト，送信のために使用されるタグや送信先引数，受信のために使用されるタグや送信元引数などのさまざまな通信操作の特性を識別する．さらに，このオブジェクトには保留中の通信操作の状態についての情報が格納される．

3.7.2 通信の起動

ブロッキング通信に対するのと同じ命名規則を使用する．すなわちバッファ，同期，レディモードに対しB，S，Rという接頭辞を使う．さらにI（即時：immediate）という接頭辞が付くと呼び出し がノンブロッキングであることを意味する．

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファの中の要素の数（非負の整数型）
IN	datatype	各送信バッファの要素のデータ型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{MPI::Request MPI::Comm::Isend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
              び出し形式, 第15.2節を参照）}
```

標準モードのノンブロッキング送信を開始する.

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファの中の要素の数（非負の整数型）
IN	datatype	各送信バッファの要素のデータ型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{MPI::Request MPI::Comm::Ibsend(const void* buf, int count, const
              MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
              び出し形式, 第15.2節を参照）}
```

バッファモードのノンブロッキング送信を開始する.

MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファの中の要素の数（非負の整数型）
IN	datatype	各送信バッファの要素のデータ型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{MPI::Request MPI::Comm::Issend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

同期モードのノンブロッキング送信を開始する.

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信バッファの中の要素の数（非負の整数型）
IN	datatype	各送信バッファの要素のデータ型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{MPI::Request MPI::Comm::Irsend(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

レディモードのノンブロッキング送信を開始する.

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

OUT	buf	受信バッファの先頭アドレス（先頭アドレス）
IN	count	受信バッファの中の要素の数（非負の整数型）
IN	datatype	各受信バッファの要素のデータ型（ハンドル）
IN	source	送信元のランクまたはMPI_ANY_SOURCE（整数型）
IN	tag	メッセージタグまたはMPI_ANY_TAG（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
{MPI::Request MPI::Comm::Irecv(void* buf, int count, const
              MPI::Datatype& datatype, int source, int tag) const （廃止された
              呼び出し形式, 第15.2節を参照）}
```

ノンブロッキング受信を開始する.

これらの呼び出しは通信リクエストオブジェクトを割り当て，リクエストハンドル（requestという引数）と対応付ける．このリクエストは後で通信の状態を問い合わせたり，その完了を待ち合わせたりするために使用することができる.

ノンブロッキング送信呼び出しは，システムが送信バッファからデータをコピーすることを開始してもよいことを意味する．送信側は，ノンブロッキング送信関数が呼ばれた後は，送信が完了するまで送信バッファのどの部分を変更することも許されない.

ノンブロッキング受信呼び出しは，システムが受信バッファにデータを書き込むことを始めてもよいことを意味する．受信側は，ノンブロッキング受信関数が呼ばれた後は，受信が完了するまで受信バッファのどの部分にアクセスすることも許されない.

> **ユーザへのアドバイス** Fortran言語のコンパイラによって行われる引数のコピーとレジスタの最適化に関する問題を回避するため，第16.2.2節の「データのコピーおよび連続領域配置による問題」（504ページ）と「レジスタの最適化での問題」（507ページ）のヒントに注意すること． （**ユーザへのアドバイス終わり**）

3.7.3 通信の完了

関数MPI_WAITとMPI_TESTはノンブロッキング通信を完了させるために使われる．送信操作の完了は送信側がその時点で自由に送信バッファの記憶領域を更新できることを意味する.（送信操作自体は送信バッファの内容を変更しない.）送信の完了はメッセージが受信されたことを意味するのではなく，通信サブシステムによって既にバッファリングされた可能性があることを意味している．しかし，**同期モード**の送信が使用されたときは，送信操作の完了はマッチする受信が起動され，このマッチする受信によってメッセージが最終的に受信されたことを意味する.

受信操作の完了は，受信バッファが受信メッセージを取り込んでおり，受信側はそれに自由にアクセスでき，状態オブジェクトが設定されているということを意味する．マッチする送信操作が完了したことを意味するわけではない（当然，送信が起動されたことは意味する）．

以下のような用語を使うことにする．**null**ハンドルとはMPI_REQUEST_NULLという値をもったハンドルである．持続的リクエストとそれに対するハンドルは，そのリクエストが現在実行中の通信と関連していなければ**非アクティブ状態**である（第3.9節参照）．ハンドルはnullでも非アクティブ状態でもなければ**アクティブ状態**である．**空**のステータスとは，tag= MPI_ANY_TAG，source = MPI_ANY_SOURCE，error = MPI_SUCCESSを返すように設定されたステータスであり，MPI_GET_COUNTやMPI_GET_ELEMENTSの呼び出しに対してcount = 0を返し，MPI_TEST_CANCELLEDの呼び出しに対してfalseを返すように内部的に設定されるステータスでもある．返される値が重要でない時には，状態変数に空を設定する．古くなってしまった情報をアクセスすることによるエラーを防ぐために，このような方法でステータスが設定される．

MPI_WAIT，MPI_TEST，またはその他の派生関数(MPI_{TEST|WAIT}{ALL|SOME|ANY})の呼び出しによって返されるstatusオブジェクト内のフィールド（requestは送信呼び出しに対応）は定義されていない．ただし，2つの例外があり，ウェイトまたはテストの呼び出しがMPI_ERR_IN_STATUSを返す場合はエラーステータスフィールドに有効な情報が格納され，戻り値はMPI_TEST_CANCELLEDの呼び出しにより照会することができる．

エラークラスMPI_ERR_IN_STATUSに属するエラーコードを返すのは，配列MPI_STATUSを取るMPI完了関数のみとする必要がある．1つのMPI_STATUS値のみを返す関数MPI_TEST，MPI_TESTANY，MPI_WAIT，MPI_WAITANYの場合は，（MPI_STATUS引数のMPI_ERRORフィールドではなく）通常のMPIのエラーを返すプロセスを使用する必要がある．

MPI_WAIT(request, status)

INOUT	request	リクエスト（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::Request::Wait(MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::Request::Wait() （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_WAITの呼び出しはrequestによって識別される操作が完了したとき戻ってくる．このリクエストに対応する通信オブジェクトがノンブロッキング送信やノンブロッキング受信の呼び出しによって作成された場合には，MPI_WAITを呼び出すことによってオブジェクトは解放され，リクエストハンドルはMPI_REQUEST_NULLに設定される．MPI_WAITは非ローカルな操作である．

呼び出しは完了した操作についての情報をstatusに返す．受信操作に対するステータス

オブジェクトの内容には，第3.2.5節に記述したような方法でアクセスすることができる．送信操作に対するステータスオブジェクトはMPI_TEST_CANCELLEDを呼び出すことによって照会することができる（第3.8節参照）．

nullや非アクティブ状態のrequest引数でMPI_WAITを呼び出すことが許されている．この場合には操作はただちに空のstatusを返す．

ユーザへのアドバイス MPI_IBSENDの後，MPI_WAITが成功して戻ってきたということは，ユーザが送信バッファを再使用することができるということを意味している．すなわち，データはすでにMPI_BUFFER_ATTACHによって貼り付けられたバッファにコピーされたか，外部に送られたということである．この時点では送信をキャンセルすることはできないため，注意が必要である（第3.8節参照）． マッチする受信が発行されなければ，バッファを解放することはできない．これは定められたMPI_CANCELの目的（通信サブシステムに渡されたプログラム領域を常に解放することができる）に少々反する． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス マルチスレッド環境では，MPI_WAITの呼び出しは呼び出されたスレッドのみをブロッキングするようにしなければならず，スレッドスケジューラが他のスレッドの実行をスケジュールできるようにしなければならない．（**実装者へのアドバイス終わり**）

MPI_TEST(request, flag, status)

INOUT	request	通信リクエスト（ハンドル）
OUT	flag	操作が完了すれば真（論理型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
{bool MPI::Request::Test(MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{bool MPI::Request::Test() （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_TESTの呼び出しでは，requestによって識別される操作が終了していればflag = trueを返す．このような場合，ステータスオブジェクトは完了した操作についての情報を格納するように設定される．通信オブジェクトがノンブロッキング送信やノンブロッキング受信によって生成された場合には，このオブジェクトは解放され，リクエストハンドルにはMPI_REQUEST_NULLが設定される．それ以外の場合は，呼び出しはflag = falseを返す．この場合にはステータスオブジェクトの値は不定となる．MPI_TESTはローカルな操作である．

受信操作に対して返されるステータスオブジェクトは，第3.2.5節で説明したような方法でアクセスできる情報を持っている．送信操作に対するステータスオブジェクト

は MPI_TEST_CANCELLEDを呼び出すことによってアクセスできる情報を持っている（第3.8節参照）.

MPI_TESTは，nullあるいは非アクティブ状態のrequest引数によって呼び出すことができる．このような場合には，flag = trueと空のstatusを返す.

関数MPI_WAITとMPI_TESTは受信と送信の両者を完了させるために使うことができる.

ユーザへのアドバイス ノンブロッキングMPI_TEST呼び出しを利用することにより，ユーザは単一の実行スレッド内で別の作業をスケジューリングすることができる．MPI_TESTを定期的に呼び出すことによってイベント駆動型のスレッドスケジューラをエミュレートすることができる． （**ユーザへのアドバイス終わり**）

例 3.12 ノンブロッキング操作とMPI_WAITの簡単な使用法

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** do some computation to mask latency ****
    CALL MPI_WAIT(request, status, ierr)
END IF
```

以下の操作を使用すれば，対応する通信が完了するのを待たずにリクエストオブジェクトを解放することができる.

MPI_REQUEST_FREE(request)

INOUT	request	通信リクエスト（ハンドル）

```
int MPI_Request_free(MPI_Request *request)
MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
{void MPI::Request::Free() （廃止された呼び出し形式, 第15.2節を参照）}
```

解放するリクエストオブジェクトにマークを付け，requestをMPI_REQUEST_NULLに設定する．このリクエストに対応する継続中の通信は完了することができる．これが完了した後にのみリクエストは解放される.

根拠 MPI_REQUEST_FREEという機構は，送信側の性能向上と便利さのために用意されている． （**根拠の終わり**）

ユーザへのアドバイス MPI_REQUEST_FREEを呼び出すことによりリクエストが解放されると，MPI_WAITやMPI_TESTを呼び出して関連する通信が正常に完了した

ことをチェックすることはできない．また，もし通信中にエラーが起こると，エラーコードをユーザに返すことができなくなる．このようなエラーは致命的なものとして取り扱うべきである．受信側は受信が完了し受信バッファが再利用可能であることを確認する手だてがなくなるので，アクティブな受信リクエストを解放すべきではない． （ユーザへのアドバイス終わり）

例 **3.13** MPI_REQUEST_FREEを使用した例

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank.EQ.0) THEN
    DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
    END DO
ELSE IF (rank.EQ.1) THEN
    CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
    DO I=1, n-1
       CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
       CALL MPI_REQUEST_FREE(req, ierr)
       CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
       CALL MPI_WAIT(req, status, ierr)
    END DO
    CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
END IF
```

3.7.4 ノンブロッキング通信の意味論

ノンブロッキング通信の意味論は，第3.5節の定義をノンブロッキングの場合に適した形に拡張することによって定義される．

順序 ノンブロッキング通信操作は通信を起動する呼び出しの実行順に従って順序づけられる．第3.5節の追越し禁止要求は，順序についてのこの定義によってノンブロッキング通信へ拡張される．

例 **3.14** ノンブロッキング操作に対するメッセージの順序

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
      CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
      CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1, status, ierr)
CALL MPI_WAIT(r2, status, ierr)
```

プロセス1がいずれかの受信を実行する前に両方のメッセージが送られていたとしても，プロセス0の最初の送信は，プロセス1の最初の受信にマッチする．

プログレス 受信を完了させるMPI_WAITの呼び出しは，マッチする送信が開始したら終了して戻る．ただし，その送信が他の受信にマッチした場合はその限りではない．特にマッチする送信がノンブロッキングである時には，送信を完了させるための呼び出しが送信側によって実行されるかどうかにかかわらず，受信は完了しなければならない．同様に，送信を完了させるMPI_WAITは，マッチする受信が開始されたら終了して戻る．ただし，その受信が他の送信にマッチした場合はその限りではない．そして，それは受信の完了の呼び出しが実行されるかどうかに関わらない．

例 3.15 プログレスの意味論の実例

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
      CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
      CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
      CALL MPI_WAIT(r, status, ierr)
END IF
```

正しいMPIの実装においては，このコードはデッドロックを起こしてはならない．プロセス0の最初の同期送信は，プロセス1がまだウェイトの呼び出しの終了に到達していなくても，プロセス1がマッチする（ノンブロッキング）受信を発行した後には完了しなければならない．そうして，プロセス0は継続して2番目の送信を実行し，プロセス1は実行を終了する．

受信を完了させるMPI_TESTが繰り返し同じ引数で呼ばれ，マッチする送信が開始されている場合，この呼び出しは最終的にflag = trueを返す．ただし，その送信が他の受信にマッチした場合はこの限りではない．送信を完了させるMPI_TESTが繰り返し同じ引数で呼ばれ，マッチする受信が開始されている場合，この呼び出しは最終的にflag = trueを返す．ただし，その受信が他の送信にマッチした場合はこの限りではない．

3.7.5 多重完了

特定のメッセージを待つのではなく，リストの中の全ての，あるいはいくつかの，あるいは任意の操作の完了を待つことができれば便利である．いくつかの操作のうちの1つの完了を待つために，MPI_WAITANYあるいはMPI_TESTANYの呼び出しを使用することができる．MPI_WAITALLやMPI_TESTALLの呼び出しを，リストの中の全ての保留中の操作を待つために使用することができる．MPI_WAITSOMEやMPI_TESTSOMEの呼び出しは，リストの中の全ての開始可能な操作を完了させるために使用することができる．

MPI_WAITANY (count, array_of_requests, index, status)

IN	count	リストの長さ（非負の整数型）
INOUT	array_of_requests	リクエストの配列（ハンドルの配列）
OUT	index	完了した操作に対するハンドルの添字（整数型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
MPI_Status *status)
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
{static int MPI::Request::Waitany(int count,
             MPI::Request array_of_requests[], MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{static int MPI::Request::Waitany(int count,
             MPI::Request array_of_requests[]) （廃止された呼び出し形式, 第15.2節を参照）}
```

配列の中のアクティブなリクエストに対応する操作の1つが完了するまでブロッキングする．1つ以上の操作が開始可能で終了できるならば，任意の1つが選ばれる．配列の中のそのリクエストの添字がindexに返され，完了した通信の状態がstatusに返される．（配列はC言語においては0から，Fortran言語においては1から添字が付けられる．）ノンブロッキング通信操作によって割り当てられたリクエストならば，それは解放されリクエストハンドルにはMPI_REQUEST_NULLが設定される．

リストarray_of_requestsはnullハンドルや非アクティブハンドルを含むことができる．リストがアクティブなハンドルを含んでいなければ（リストの長さがゼロか全ての要素がnullか非アクティブ），呼び出しはindex = MPI_UNDEFINEDと空のstatusを返す．

MPI_WAITANY(count, array_of_requests, index, status)の実行はMPI_WAIT(&array_of_requests[i], status)の実行と同じ結果になる．ここでiは（indexの値がMPI_UNDEFINEDでなければ）indexによって返される値である．1つのアクティブな要素を含む配列を持つMPI_WAITANYはMPI_WAITと同等である．

MPI_TESTANY(count, array_of_requests, index, flag, status)

IN	count	リストの長さ（整数型）
INOUT	array_of_requests	リクエストの配列（ハンドルの配列）
OUT	index	完了した操作の添字，何も完了しなかった時はMPI_UNDEFINED（整数型）
OUT	flag	操作のうちの 1 つが完了したときは真（論理型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
int *flag, MPI_Status *status)
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
```

{static bool MPI::Request::Testany(int count,
MPI::Request array_of_requests[], int& index,
MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{static bool MPI::Request::Testany(int count,
MPI::Request array_of_requests[], int& index) （廃止された呼び出し形式, 第15.2節を参照）}

アクティブなハンドルに対応する操作の1つが完了しているか，あるいはどれも完了していないかのテスト．前者の場合には，flag = trueが返され，配列の中のこのリクエストの添字がindexに返され，その操作の状態がstatusに返される．もしリクエストがノンブロッキング通信の呼び出しによって割り当てられたものであれば，そのリクエストは解放され，ハンドルにはMPI_REQUEST_NULLが設定される（配列はC言語の場合は0から，Fortran言語の場合は1から添字が始まる）．後者の場合（完了した操作がない場合），flag = falseが返され，MPI_UNDEFINEDという値がindexに返され，statusは不定となる．

配列はnullハンドルあるいは非アクティブなハンドルを含むことができる．配列がアクティブなハンドルを含んでいなければ，呼び出しはflag = true，index = MPI_UNDEFINEDとなり，statusは空となる．

リクエストの配列がアクティブなハンドルを含んでいれば，MPI_TESTANY(count, array_of_requests, index, status)の実行は，任意のある順序のi=0, 1 ,..., count-1に対して，1つの呼び出しがflag = trueを返すか，全てが失敗するまでMPI_TEST(&array_of_requests[i], flag, status)が実行された場合と同じ効果を持つ．前者の場合にはindexに最後のiの値が，後者の場合にはMPI_UNDEFINEDが設定される．1つのアクティブな要素を含む配列を持ったMPI_TESTANYはMPI_TESTと同等である．

MPI_WAITALL(count, array_of_requests, array_of_statuses)

IN	count	リストの長さ（非負の整数型）
INOUT	array_of_requests	リクエストの配列（ハンドルの配列）
OUT	array_of_statuses	ステータスオブジェクトの配列（ステータス型の配列）

```
int MPI_Waitall(int count, MPI_Request *array_of_requests,
MPI_Status *array_of_statuses)
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*)
    INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

{static void MPI::Request::Waitall(int count,
MPI::Request array_of_requests[],
MPI::Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}
{static void MPI::Request::Waitall(int count,
MPI::Request array_of_requests[]) （廃止された呼び出し形式, 第15.2節を参照）}

リストの中のアクティブなハンドルに対応する全ての通信操作が完了するまでブロッキングし，これら全ての操作のステータスを返す（リストの中のどのハンドルもアクティブでない場合も含む）．両方の配列は同じ数の有効なエントリーを持つ．

array_of_statusesのi番目のエントリーにはi番目の操作のステータスの戻り値が設定される．ノンブロッキング通信操作によって生成されたリクエストは解放され，配列の中の対応するハンドルにはMPI_REQUEST_NULLが設定される．リストはnullや非アクティブなハンドルを含むこともできる．そのようなエントリーは，この呼び出しによってそれぞれ空に設定される．

MPI_WAITALL(count, array_of_requests, array_of_statuses)が誤りなく実行されたときは，任意のある順序のi=0 ,..., count-1に対してMPI_WAIT(&array_of_request[i], &array_of_statuses[i])が実行されたときと同じ効果を待つ．長さ1の配列を持つMPI_WAITALLはMPI_WAITと同等である．

MPI_WAITALLの呼び出しによって完了した1つあるいはそれ以上の通信が失敗した時には，それぞれの通信についての特定の情報を返すことが望ましい．このような場合，関数MPI_WAITALLはエラーコードMPI_ERR_IN_STATUSを返し，それぞれのステータスのエラーフィールドに特定のエラーコードを返す．特定の通信が完了した場合はこのコードはMPI_SUCCESSであり，失敗した場合は他の特定のエラーコードである．あるいは，失敗でも完了でもない場合はMPI_ERR_PENDINGとなる．関数MPI_WAITALLはどのリクエストにもエラーがなかったときはMPI_SUCCESSを返し，（不正な引数等の）他の理由で失敗したときは，他のエラーコードを返す．このような場合，ステータスのエラーフィールドは更新されない．

根拠 これは，アプリケーションにおける誤りの取扱いを簡素化してくれる．アプリケーションのコードは，エラーが発生したかどうか判定するために（1つの）関数の結果をテストするだけでよい．エラーが発生したときにのみ個々のステータスを確かめる必要がある． （**根拠の終わり**）

MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN	count	リストの長さ（非負の整数型）
INOUT	array_of_requests	リクエストの配列（ハンドルの配列）
OUT	flag	（論理型）
OUT	array_of_statuses	ステータスオブジェクトの配列（ステータス型の配列）

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
MPI_Status *array_of_statuses)
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
{static bool MPI::Request::Testall(int count,
             MPI::Request array_of_requests[],
             MPI::Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}
{static bool MPI::Request::Testall(int count,
             MPI::Request array_of_requests[]) （廃止された呼び出し形式, 第15.2節を参照）}
```

配列の中のアクティブなハンドルに対応する全ての通信が完了したときにflag = trueを返す（リストの中のハンドルがどれもアクティブでない場合を含む）．この場合，アクティブなハンドルのリクエストに対応するステータスの各エントリには対応する通信のステータスが設定される．もしノンブロッキング通信の呼び出しによってリクエストが割り当てられていたときは，リクエストの割当は解放され，ハンドルにはMPI_REQUEST_NULLが設定される．nullあるいは非アクティブなハンドルに対応するステータスの各エントリには空が設定される．

それ以外の場合には，flag = falseが返され，どのリクエストも変更されず，ステータスのエントリの値は不定となる．これはローカルな操作である．

MPI_TESTALLの実行中に起こったエラーはMPI_WAITALLの中のエラーとして処理される．

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN	incount	array_of_requestsの長さ（非負の整数型）
INOUT	array_of_requests	リクエストの配列（ハンドルの配列）
OUT	outcount	完了したリクエストの数（整数型）
OUT	array_of_indices	完了した操作の添字の配列（整数型の配列）
OUT	array_of_statuses	完了した操作に対するステータスオブジェクトの配列（ステータス型の配列）

```
int MPI_Waitsome(int incount, MPI_Request *array_of_requests,
int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
{static int MPI::Request::Waitsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[],
             MPI::Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}
{static int MPI::Request::Waitsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[]) （廃止された呼び出し形式, 第15.2節を参照）}
```

リストの中のアクティブなハンドルに対応する操作のうち，少なくとも1つが終了するまで待つ．リストarray_of_requestsからのリクエストのうち完了したものの数をoutcountに返す．配列array_of_indicesの最初のoutcount個の位置にこれらの操作の添字を返す．（配列array_of_requests内の添字．配列はC言語の場合は0から，Fortran言語の場合は1から添字が始まる．）配列array_of_statusの最初のoutcount個の位置に，完了したこれら操作に対するステータスを返す．完了したリクエストがノンブロッキング通信の呼び出しによって割り当てられた場合は，そのリクエストは解放され，対応するハンドルにはMPI_REQUEST_NULLが設定される．

リストがアクティブなハンドルを含まない場合は，呼び出しはただちにoutcount =

MPI_UNDEFINEDを返す.

MPI_WAITSOMEによって完了した1つあるいはそれ以上の通信が失敗した時は，それぞれの通信についての特定の情報を返すことが望ましい．引数outcount, array_of_indices，array_of_statusesは成功，あるいは失敗した全ての通信の完了を示すように調節される．呼び出しはエラーコードMPI_ERR_IN_STATUSを返し，それぞれのステータスの返ってきたエラーフィールドは成功を示すように設定されるか，発生した特定のエラーを示すように設定される．この呼び出しは，失敗に終わったリクエストがなかったときはMPI_SUCCESSを返し，他の理由（不正な引数など）によって失敗したときは他のエラーコードを返す．このような場合には，ステータスのエラーフィールドは更新されない.

MPI_TESTSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN	incount	array_of_requestsの長さ（非負の整数型）
INOUT	array_of_request	リクエストの配列（ハンドルの配列）
OUT	outcount	完了したリクエストの数（整数型）
OUT	array_of_indices	完了した操作の添字の配列（整数型の配列）
OUT	array_of_statuses	完了した操作に対するステータスオブジェクトの配列（ステータス型の配列）

```
int MPI_Testsome(int incount, MPI_Request *array_of_requests,
int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
{static int MPI::Request::Testsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[],
             MPI::Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}
{static int MPI::Request::Testsome(int incount,
             MPI::Request array_of_requests[], int array_of_indices[]) （廃止された呼び出し形式, 第15.2節を参照）}
```

ただちに戻るという点を除いて，動作はMPI_WAITSOMEと同様である．操作が1つも完了していないときはoutcount = 0を返す．リストの中にアクティブなハンドルが無いときはoutcount = MPI_UNDEFINEDを返す.

MPI_TESTSOMEはローカルな操作であり，MPI_WAITSOMEが通信が完了するまでブロッキングされていたのに対し，少なくとも1つのアクティブなハンドルを含むリストを渡されると，ただちに戻ってくる．この呼び出しはどちらも**公平性**の要求を満たす．すなわち，MPI_WAITSOMEあるいはMPI_TESTSOMEに渡されたリクエストのリストに受信リクエストが繰り返し現れ，マッチする送信が発行されると，その送信に他の受信が応じなければ受信は最終的には成功する．送信リクエストについても同様である.

MPI_TESTSOMEの実行中に起こったエラーは，MPI_WAITSOMEと同様に扱われる.

ユーザへのアドバイス MPI_TESTSOMEの使用はMPI_TESTANYの使用よりも有効であると考えられる．前者は完了した全ての通信についての情報を返すが，後者では完了したそれぞれの通信について新たな呼び出しが必要となる．

複数のクライアントを持つサーバでは，クライアントの要求がいつまでも受け付けられないという状態にならないようにMPI_WAITSOMEを利用することができる．クライアントはサーバにサービス要求のメッセージを送る．サーバはクライアントごとに受信リクエストを付けてMPI_WAITSOMEを呼び出し，完了した全ての受信を取り扱う．代わりにMPI_WAITANYの呼び出しを使うと，他のクライアントからの要求が常に最初に割り込んでしまい，あるクライアントの要求がいつまでも受け付けられないということが起こりうる． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス MPI_TESTSOMEは保留中の通信を可能な限りたくさん完了させなければならない． （**実装者へのアドバイス終わり**）

例 3.16 クライアントサーバのコード（スタベーションが起こりうる）

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN         ! client code
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE         ! rank=0 -- server code
       DO i=1, size-1
          CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                    comm, request_list(i), ierr)
       END DO
       DO WHILE(.TRUE.)
          CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
          CALL DO_SERVICE(a(1,index))  ! handle one message
          CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag,
                     comm, request_list(index), ierr)
       END DO
END IF
```

例 3.17 MPI_WAITSOMEを用いた同様のコード．

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN         ! client code
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
ELSE         ! rank=0 -- server code
    DO i=1, size-1
```

```
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag,
                      comm, request_list(i), ierr)
    END DO
    DO WHILE(.TRUE.)
       CALL MPI_WAITSOME(size, request_list, numdone,
                         indices, statuses, ierr)
       DO i=1, numdone
          CALL DO_SERVICE(a(1, indices(i)))
          CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
                         comm, request_list(indices(i)), ierr)
       END DO
    END DO
END IF
```

3.7.6 statusの非破壊なテスト

この呼び出しは，リクエストを解放しないでリクエストに関連する情報にアクセスする場合に有益である（後からユーザがアクセスすることが想定される場合）．複数のソフトウェアの層から完了した同じリクエストにアクセスし，そこからステータス情報を取得することができるため，ライブラリをより便利に多層構造にすることができる．

MPI_REQUEST_GET_STATUS(request, flag, status)

IN	request	リクエスト（ハンドル）
OUT	flag	MPI_TESTと同様の論理型のフラグ（論理型）
OUT	status	MPI_STATUSオブジェクト（フラグが真の場合）（ステータス型）

```
int MPI_Request_get_status(MPI_Request request, int *flag,
MPI_Status *status)
MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
{bool MPI::Request::Get_status(MPI::Status& status) const （廃止された呼び出し形式, 第15.2節を参照）}
{bool MPI::Request::Get_status() const （廃止された呼び出し形式, 第15.2節を参照）}
```

操作が完了した場合はflag=trueを設定し，この場合，statusにリクエストステータスを返す．ただし，テストまたはウェイトとは異なり，リクエストの解放や非アクティブ化は行わないため，その後，テストやウェイトや解放の呼び出しをそのリクエストを使用して実行する必要がある．操作が完了しない場合，flag=falseを設定する．

MPI_REQUEST_GET_STATUSの呼び出しには，nullまたは非アクティブなリクエストの引数を使用することができる．このような場合，操作はflag=trueと空のステータスを返す．

3.8 プローブおよびキャンセル

MPI_PROBEおよびMPI_IPROBE操作により，実際にメッセージを受け取ることなく，送信したメッセージが確認できる．ユーザは，プローブ操作により得られた情報に基づいて，メッセージを受け取る方法を決定することができる（基本的には，この情報はstatusにより返されるものである）．特に，あらかじめ確認されたメッセージの長さに応じて，受信バッファの記憶領域を割り当てることができる．

MPI_CANCEL操作により，保留中の通信をキャンセルできる．この操作は，クリーンアップのために必要である．送信または受信の発行は，ユーザリソース（送受信バッファ）の消費に直接関連している．キャンセルは，これらのリソースを悪影響なく解放するために必要とされるものであることができる．

MPI_IPROBE(source, tag, comm, flag, status)

IN	source	送信元のランクまたはMPI_ANY_SOURCE（整数型）
IN	tag	メッセージタグまたはMPI_ANY_TAG（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	flag	（論理型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
MPI_Status *status)
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
{bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const （廃
              止された呼び出し形式, 第15.2節を参照）}
{bool MPI::Comm::Iprobe(int source, int tag) const （廃止された呼び出し形式,
              第15.2節を参照）}
```

MPI_IPROBE (source, tag, comm, flag, status)は，受信可能で，引数source, tag，commにより指定されたパターンにマッチするメッセージがあった場合，flag = trueを返す．この呼び出しによるメッセージは，プログラムの中で同じ時点で実行されるMPI_RECV(..., source, tag, comm, status)によって受信されるメッセージと一致し，MPI_RECV()によって返されるのと同じ値をstatusに返す．そうでない場合，この呼び出しはflag = falseを返し，statusは未定義のままとなる．

第3.2.5節で説明しているように，MPI_IPROBEがflag = trueを返した場合，プローブされたメッセージの送信元やタグ，長さを見つけるため，ステータスオブジェクトの内容が続いてアクセスされうる．

MPI_IPROBEで返されたstatusに入っていた，同じコミュニケータ，ソース，タグを用いた後から行われた受信は，プローブでマッチされたメッセージを受信する．ただし，プローブの後に割り込むような他の受信がなく，また受信の前にその送信のキャンセルが成功していない場合に限る．受信プロセスがマルチスレッドの場合，この条件が満た

されるようにすることはユーザの責任である．

MPI_PROBEのsource引数は，MPI_ANY_SOURCEとする設定が可能であり，tag引数にも同様にMPI_ANY_TAGが設定可能である．そのため，任意の送信元やタグによって，複数のメッセージについてプローブすることができる．ただし，comm引数に対して特定の通信コンテクストが与えられなければならない．

メッセージをプローブした後では，すぐにメッセージを受信する必要はなく，また受信する以前に何度でも同じメッセージをプローブすることが可能である．

MPI_PROBE(source, tag, comm, status)

IN	source	送信元のランクまたはMPI_ANY_SOURCE（整数型）
IN	tag	メッセージタグまたはMPI_ANY_TAG（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const （廃
            止された呼び出し形式, 第15.2節を参照）}
{void MPI::Comm::Probe(int source, int tag) const （廃止された呼び出し形式,
            第15.2節を参照）}
```

MPI_PROBEは，マッチするメッセージを発見した後でのみ戻るブロッキング呼び出しである点を除いて，MPI_IPROBEと同様に動作する．

MPIにおけるMPI_PROBEやMPI_IPROBEの実装では，プログレスを保証する必要がある．あるプロセスによりMPI_PROBEの呼び出しが実行され，プローブにマッチした送信がいずれかのプロセスによって起動されている場合，そのメッセージが，同時に動作する別の受信操作（プローブのプロセスにおいて別のスレッドにより実行されるもの）によって受信されない場合に限り，MPI_PROBEの呼び出しがいつかは戻る．同様に，プロセスがMPI_IPROBEでビジーウェイトし，マッチするメッセージが発行された場合には，そのメッセージが同時に動作する別の受信操作によって受信されない場合に限り，MPI_IPROBEの呼び出しは最終的にflag = trueを返す．

例 3.18 送信されたメッセージを待つためのブロッキングプローブの使用

```
        CALL MPI_COMM_RANK(comm, rank, ierr)
        IF (rank.EQ.0) THEN
            CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
        ELSE IF (rank.EQ.1) THEN
            CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
        ELSE IF (rank.EQ.2) THEN
            DO i=1, 2
               CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                              comm, status, ierr)
               IF (status(MPI_SOURCE) .EQ. 0) THEN
100                CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
```

```
               ELSE
200                CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
               END IF
            END DO
         END IF
```

それぞれのメッセージは正しい型で受信される.

例 3.19 前の例と同様のプログラム（ただしこの例では問題がある）

```
         CALL MPI_COMM_RANK(comm, rank, ierr)
         IF (rank.EQ.0) THEN
              CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
         ELSE IF (rank.EQ.1) THEN
              CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
         ELSE IF (rank.EQ.2) THEN
             DO i=1, 2
               CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                              comm, status, ierr)
               IF (status(MPI_SOURCE) .EQ. 0) THEN
100                 CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                                  0, comm, status, ierr)
               ELSE
200                 CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                                  0, comm, status, ierr)
               END IF
             END DO
         END IF
```

100および200というラベルを付けた2つの受信呼び出しでsource引数としてMPI_ANY_SOURCEを使用し，例3.18に少し修正を加えている．このプログラムはこのままでは，正しく動作しない．すなわち，これらの受信操作が，あらかじめMPI_PROBEの呼び出しによってプローブされたメッセージとは異なるメッセージを受け取る可能性がある.

> **実装者へのアドバイス** MPI_PROBE(source, tag, comm, status)の呼び出しは，同じ地点で実行するMPI_RECV(..., source, tag, comm, status)の呼び出しによって受け取られるメッセージと一致する．このメッセージが，送信元s，タグt，コミュニケータcを持つとする．プローブ呼び出しにおけるタグ引数が値MPI_ANY_TAGをもつ場合には，プローブされるメッセージは，任意のタグとコミュニケータcを持つ，送信元sからの最初の保留中のメッセージである．いかなる場合でも，プローブされたメッセージは，タグtとコミュニケータcを持つ送信元sからの最初の保留中のメッセージである（これは，メッセージの順序を保持するように，受信したメッセージである）．このメッセージは，受信されるまでタグtとコミュニケータcを持つ送信元sからの最初の保留中のメッセージとして保持される．プローブ後の受信操作は，プローブと同じコミュニケータを使用し，プローブによって返されたタグと送信元の値を使用する場合，このメッセージが別の受信操作によってすでに受信されているのでなければ，受信する必要がある． （**実装者へのアドバイス終わり**）

MPI_CANCEL(request)

IN	request	通信リクエスト（ハンドル）

```
int MPI_Cancel(MPI_Request *request)
MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
{void MPI::Request::Cancel() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_CANCELの呼び出しは，キャンセルを行う保留中のノンブロッキング通信関数（送信または受信）をマークする．キャンセルの呼び出しは，ローカルに実行される．この呼び出しは即座に戻り，通信が実際にキャンセルされる以前である可能性もある． キャンセルをマークした通信はそれでも，MPI_REQUEST_FREE，MPI_WAITまたはMPI_TEST （または任意の派生関数）の呼び出しを使って完了する必要がある．

ある通信がキャンセルの対象とされた場合，他のプロセスの状態に関わらず，通信に対するMPI_WAIT呼び出しが戻ることが保証される（したがって，MPI_WAITはローカル関数として動作することになる）．同様に，ビジーウェイトループにおいて，キャンセルの対象とされた通信に対してMPI_TESTを繰り返し呼び出した場合，この呼び出しは，最終的には成功することになる．

MPI_CANCELは，非持続的なリクエストを用いる場合と同様な方法で，持続的なリクエスト（第3.9節参照）を用いた通信をキャンセルするためにも利用することができる．キャンセル操作の成功は，動作中の通信をキャンセルすることになるが，リクエスト自体をキャンセルするわけではない．MPI_CANCELが呼び出され，続いてMPI_WAITやMPI_TESTが呼び出された場合，リクエストは非アクティブになり，新しい通信でアクティブにすることができる．

バッファ送信のキャンセルが成功すると，保留中のメッセージによって確保されていたバッファ領域は解放される．

通信に対しては，キャンセルが成功するか，動作が成功するかのどちらかであり，いずれもが成功することはない．ある送信がキャンセルの対象としてマークされた場合，送信が正常に完了する（このときは，送信されたメッセージは送信先のプロセスに受信される）か，送信のキャンセルが成功する（このときは，送信先においてメッセージの全てが受信されない）かのいずれかでなければならない．したがって，キャンセルされた送信にマッチする受信に対しては，別の送信がマッチする必要がある．また，ある受信がキャンセルの対象としてマークされた場合，送信が正常に完了するか，受信のキャンセルが成功する（このときは，受信バッファの全てが更新されない）かのいずれかにならなければならない．したがって，キャンセルされた受信にマッチする送信に対しては，別の受信がマッチする必要がある．

操作がキャンセルされた場合，通信を完了させる関数のstatus引数に対して，キャンセルの結果についての情報が返される．

根拠 INリクエストハンドルパラメータは参照により渡される必要がないことになっているが，MPI-1.0以来，C言語の呼び出し形式では引数の型が MPI_Request*として列記されている．そのため，この関数のシグネチャは既存のMPIアプリケーシ

ョンに影響を及ぼさずには変更することはできない. （根拠の終わり）

MPI_TEST_CANCELLED(status, flag)

IN	status	ステータスオブジェクト（ステータス型）
OUT	flag	（論理型）

```
int MPI_Test_cancelled(MPI_Status *status, int *flag)
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

{bool MPI::Status::Is_cancelled() const （廃止された呼び出し形式, 第15.2節を参照）}

ステータスオブジェクトに関連する通信のキャンセルが成功した場合には，flag = trueが返される．この場合，status中の他の全てのフィールド（count，tagなど）は，不定になる．そうでない場合には，flag = falseが返される．受信操作がキャンセルされた可能性がある場合，ステータスの戻り値の他の項目の情報について確認する前に，受信操作がキャンセルされたかどうかを調べるために，まずはじめにMPI_TEST_CANCELLEDを呼び出す必要がある．

ユーザへのアドバイス キャンセルは，コストのかかる操作になる可能性があるので，例外的な場合にのみ利用すべきである. （ユーザへのアドバイス終わり）

実装者へのアドバイス 送信操作で“eager”プロトコル（マッチする受信が発行される前に，受信側に対してデータを転送する）を用いた場合，この送信のキャンセルを行うには，割り当てられていたバッファの領域を解放するために，対象とされる受信側との通信が必要になる場合がある．また，システムによっては，この通信が，対象とされる受信側に対して割り込みを行う必要がある．なお，実装に当たり，以下に示す点を注意する必要がある. MPI_CANCELのために通信が必要となる可能性があるものの，MPI_CANCEL自体はローカルな操作となることに注意せねばならない．その操作が他のプロセスにより実行されたコードに依存していないためである. もし，他のプロセスに対する処理が必要な場合，その処理はアプリケーションからは見えないようになっている必要がある（そのため，割り込みや割り込みハンドラが必要とされる）. （実装者へのアドバイス終わり）

3.9 持続的通信リクエスト

並列計算のループの内部において，同一の引数リストを持つ通信が，しばしば繰り返し実行される. このような場合，通信での引数リストを**持続的**な通信リクエストと一度結びつけて，このリクエストを用いてメッセージの起動と完了を繰り返し行うことによって，これらの通信の効率を最適化することができる．つまり，このようにして生成された持続的な通信リクエストは，通信ポートまたは「ハーフチャネル」とみなすことができる．これらの通信では，送信ポートと受信ポートとの間に呼び出し形式が完成して

いないため，標準的なチャネルのような完全な機能が提供されているわけではない．このような構成を行うことにより，プロセスと通信コントローラとの間で行われる通信に関して，オーバーヘッドを軽減できる．一方，ある通信コントローラと他の通信コントローラとの通信に関しては，このようなオーバーヘッドの軽減はできない．持続的な通信リクエストにより送信されるメッセージは，必ずしも，持続的なリクエストによる受信操作によって受信される必要はなく，また，逆の状況であった場合も同様である．

持続的な通信リクエストは，以下の5つの呼び出しのうちの1つを利用して生成される．なお，これらの呼び出しでは，通信は実行されない．

MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信される要素の数（非負の整数型）
IN	datatype	各要素の型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{MPI::Prequest MPI::Comm::Send_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}
```

標準モードでの送信操作のための持続的な通信リクエストを生成し，送信操作での全ての引数をそれに結びつける．

MPI_BSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信される要素の数（非負の整数型）
IN	datatype	各要素の型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

```
{MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

バッファモードでの送信のための持続的な通信リクエストを生成する.

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信される要素の数（非負の整数型）
IN	datatype	各要素の型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{MPI::Prequest MPI::Comm::Ssend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

同期モードでの送信操作のための持続的な通信オブジェクトを生成する.

MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN	buf	送信バッファの先頭アドレス（選択型）
IN	count	送信される要素の数（非負の整数型）
IN	datatype	各要素の型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	tag	メッセージタグ（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
{MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count, const
             MPI::Datatype& datatype, int dest, int tag) const （廃止された呼
             び出し形式, 第15.2節を参照）}
```

レディモードでの送信操作のための持続的な通信オブジェクトを生成する.

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

OUT	buf	受信バッファの先頭アドレス（選択型）
IN	count	受信された要素の数（非負の整数型）
IN	datatype	各要素の型（ハンドル）
IN	source	送信元のランクまたはMPI_ANY_SOURCE（整数型）
IN	tag	メッセージタグまたはMPI_ANY_TAG（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	request	通信リクエスト（ハンドル）

```
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
{MPI::Prequest MPI::Comm::Recv_init(void* buf, int count, const
             MPI::Datatype& datatype, int source, int tag) const （廃止された
             呼び出し形式, 第15.2節を参照）}
```

受信操作のための持続的通信リクエストを生成する．ユーザがMPI_RECV_INITに対して引数を渡すことにより受信バッファに書き込み許可を与えるため，引数bufはOUTとなる．

持続的な通信リクエストは，それが生成された後では非アクティブである．つまり，アクティブな通信が持続的な通信リクエストに結びつけられることはない．

持続的な通信リクエストを用いた通信（送信または受信）はMPI_START関数により起動される．

MPI_START(request)

INOUT	request	通信リクエスト（ハンドル）

```
int MPI_Start(MPI_Request *request)
MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
{void MPI::Prequest::Start() （廃止された呼び出し形式, 第15.2節を参照）}
```

ここでの引数requestは，上記の5種類の呼び出しのいずれかにより返されるハンドルである．対応するリクエストは，この状態では非アクティブである．ただし，一度呼び出しが実行された後では，リクエストはアクティブになる．

このリクエストがレディモードでの送信であった場合，呼び出しが実行される前に，マッチする受信が発行されている必要がある．呼び出しの後，この操作が完了するまでは，通信バッファを変更してはならない．

この呼び出しはローカルで，第3.7節で説明したノンブロッキング通信関数と同様の意味を持つ．つまり，MPI_SEND_INITによって生成されたリクエストを使用したMPI_STARTの呼び出しは，MPI_ISENDの呼び出しと同様な方法によって通信を開始する．また，MPI_BSEND_INITによって生成されたリクエストを使用したMPI_STARTの

呼び出しは，MPI_IBSENDの呼び出しと同様の方法によって通信を開始する，などである．

MPI_STARTALL(count, array_of_requests)

IN	count	リストの長さ（非負の整数型）
INOUT	array_of_requests	リクエストの配列（ハンドルの配列）

```
int MPI_Startall(int count, MPI_Request *array_of_requests)
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
{static void MPI::Prequest::Startall(int count,
             MPI::Prequest array_of_requests[])
```
（廃止された呼び出し形式, 第15.2節を参照）}

array_of_requestsのリクエストに対応する全ての通信を開始する．

MPI_STARTALL(count, array_of_requests)の呼び出しは，任意の順序でi=0 ,..., count-1に対して繰り返し実行されるMPI_START (&array_of_requests[i])の呼び出しと同じ機能を持っている．

MPI_STARTやMPI_STARTALLの呼び出しにより開始された通信は，MPI_WAITや MPI_TESTの呼び出し，または第3.7.5節で説明した派生関数の呼び出しによって完了する．このリクエストは，上記の呼び出しが完了した後で非アクティブになる． このリクエストへの領域の割り当てが解放されることはない．そして，MPI_STARTやMPI_STARTALLの呼び出しによって新たにアクティブにすることができる．

持続的なリクエストは，MPI_REQUEST_FREEの呼び出し（第3.7.3節）によって領域の割り当てが解放される．

MPI_REQUEST_FREEの呼び出しは，持続的なリクエストが生成された後，プログラムの任意の時点で実行することが可能である．ただし，このリクエストは非アクティブになった後でのみ割当が解放される．アクティブな受信リクエストは解放してはいけない． そうしてしまうと，受信が完了したかどうかを確認することはできなくなる．一般的に，これらの受信リクエストは非アクティブになった状態で解放することが望ましい．この規則に従っている場合には，この節で説明した関数は，以下に記述された順序で起動される．

Create (Start Complete)* Free

ここで*は0回または1回以上の繰り返しを示す．同じ通信オブジェクトが複数の同時実行されるスレッド内で使用される場合，正しい順序に従うようにするために呼び出しを調整するのは，ユーザの責任である．

MPI_STARTにより起動された送信操作は，任意の受信操作とマッチすることが可能であり，同様に，MPI_STARTにより起動された受信操作は，任意の送信操作によって生成されるメッセージを受信することが可能である．

ユーザへのアドバイス Fortran言語のコンパイラによって行われる引数のコピーとレジスタの最適化に関する問題を回避するため，第16.2.2節の「データのコピーおよび連続領域配置による問題」（504ページ）と「レジスタの最適化での問題」（507ページ）のヒントに注意すること．（**ユーザへのアドバイス終わり**）

3.10 送受信

送受信操作は，特定の送信先に対するメッセージの送信動作と，他のプロセスからの別のメッセージの受信動作を1つの呼び出しとして組み合わせたものである．送信元と送信先は，同一にすることもできる．この送受信操作は， 連鎖的に繋がれたプロセスに沿ってシフト操作を実行するときに大変有効である．ブロッキング送信および受信がこのようなシフト操作に利用される場合，デッドロックを引き起こすような巡回依存性を避けるためにも，送信と受信の順序を正しく保つ必要がある（例えば，偶数のプロセスは送信をしてから受信を行い，奇数のプロセスであればはじめに受信を行ってから送信を行う）．このような送受信操作が利用される場合，これらの問題は通信サブシステムにより調節される．さまざまな論理的なトポロジー上でシフト操作を実行するため，第7章で説明するような関数と結合する形式で送受信操作を利用することができる．また，送受信操作は，リモートプロシージャ呼び出しを実装する場合においても有効である．

送受信操作により送信されたメッセージは，通常の受信操作によって受信したり，プローブ操作によってプローブしたりすることができる．送受信操作では，通常の送信操作によって送信されたメッセージを受信することもできる．

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcount	送信バッファ内の要素数（非負の整数型）
IN	sendtype	送信バッファ内の要素の型(ハンドル)
IN	dest	送信先のランク（整数型）
IN	sendtag	送信タグ（整数型）
OUT	recvbuf	受信バッファの先頭アドレス（選択型）
IN	recvcount	受信バッファ内の要素数（非負の整数型）
IN	recvtype	受信バッファ内の要素の型（ハンドル）
IN	source	送信元のランクまたはMPI_ANY_SOURCE（整数型）
IN	recvtag	受信タグまたはMPI_ANY_TAG（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype,
int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
             MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
             int recvcount, const MPI::Datatype& recvtype, int source,
             int recvtag, MPI::Status& status) const （廃止された呼び出し形式,
             第15.2節を参照）}
{void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
             MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
             int recvcount, const MPI::Datatype& recvtype, int source,
             int recvtag) const （廃止された呼び出し形式, 第15.2節を参照）}
```

ブロッキング送受信操作を実行する．送信と受信の両方で同じコミュニケータが使われているが，タグは異なったものすることが望ましい．送信バッファと受信バッファは独立したものでなければならず，異なった長さとデータ型を持つことができる．

送受信操作の意味論は，呼び出し元が2つの並行スレッドをフォークし，一つは送信を実行し，もう一つは受信を実行し，最後にこれらの2つのスレッドをジョインする動作と同じものである．

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

INOUT	buf	送信兼受信バッファの先頭アドレス（選択型）
IN	count	送信兼受信バッファ内の要素数（非負の整数型）
IN	datatype	送信兼受信バッファ内の要素の型（ハンドル）
IN	dest	送信先のランク（整数型）
IN	sendtag	送信メッセージタグ（整数型）
IN	source	送信元のランクまたはMPI_ANY_SOURCE（整数型）
IN	recvtag	受信メッセージタグまたはMPI_ANY_TAG（整数型）
IN	comm	コミュニケータ（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::Comm::Sendrecv_replace(void* buf, int count, const
             MPI::Datatype& datatype, int dest, int sendtag, int source,
             int recvtag, MPI::Status& status) const （廃止された呼び出し形式,
             第15.2節を参照）}
```

{void MPI::Comm::Sendrecv_replace(void* buf, int count, const
MPI::Datatype& datatype, int dest, int sendtag, int source,
int recvtag) const （廃止された呼び出し形式, 第15.2節を参照）}

ブロッキング送受信を実行する．送信操作と受信操作の両方に対して同じバッファが使用されるため，送信されたメッセージは受信されたメッセージに置換されてしまう．

実装者へのアドバイス “replace”が付いた呼び出しに対しては中間バッファリングの追加が必要になる． （**実装者へのアドバイス終わり**）

3.11 nullプロセス

多くの場合，通信を行う場合に，「ダミー」の送信元と送信先を指定することは有用である．これによって，非循環型のシフト操作を送受信操作の呼び出しで実行する場合などにおいて，境界を取り扱う必要のあるコードを単純にすることが出来る．

呼び出しにおいて，送信元と送信先の引数が必要な場合にはランクの代わりに特別な値MPI_PROC_NULLを使用することができる．プロセスMPI_PROC_NULLによる通信では何も実行されない．この場合，MPI_PROC_NULLへの送信は成功し，ただちに戻る．また，MPI_PROC_NULLからの受信も成功し，受信バッファに対する修正が行われることなく，ただちに戻る．source = MPI_PROC_NULLとして 受信が実行される場合には，ステータスオブジェクトはsource = MPI_PROC_NULL，tag = MPI_ANY_TAG，count = 0を返す．

第4章

データ型

基本のデータ型については，第3.2.2節のメッセージデータ（32ページ）と第3.3節のデータ型の一致とデータ変換（42ページ）で紹介した．この章では，これをさらに拡大してデータの配置を説明する．異機種間の不連続なデータを効率的に転送するための汎用データ型についても検討する．最後に，メッセージの明示的なパック／アンパックの呼び出しについて説明する．

4.1　派生データ型

ここまで，すべての1対1通信は，同一の基本データ型の並びを含むバッファしか扱ってこなかった．このことは2つの大きな制限を含んでいた．1つは，異なるデータ型をもつ複数の値を含むメッセージを渡したい場合である（例えば，整数型の個数と，その後に実数列が続くような場合）．もう1つは，不連続なデータを送信したい場合である（例えば，行列のサブブロック）．1つの解決策は，送信側で，不連続なデータを連続なバッファにパックし，そして受信側でアンパックすることである．しかし，これは通信サブシステムがスキャッタ–ギャザーの機能を持っている場合でさえ，余分なメモリコピー操作が送信側と受信側の両方で必要になるという欠点を持つ．代わりに，MPI は，より一般化され，データ型が混在し，かつ不連続な通信バッファを明確に定義するための機構を提供する．データを転送する前に連続なバッファにまずパックする必要があるのか，あるいはデータが置かれた場所から直接集めるのかは実装によって決まる．

ここで提供される一般化された機構によって，コピーなしに，様々な形やサイズを持つオブジェクトを転送することができる．しかし，MPIライブラリがホスト言語で宣言されたオブジェクトを認識できることを前提としていない．したがって，構造体や配列の一部を転送しようとすると，その構造体や配列の一部の定義を模倣した通信バッファの定義をMPIに与える必要が生じてしまう．これらの手段はライブラリの設計者が，ホスト言語で定義されたオブジェクトを転送できるような通信関数を定義するために使うことができる．例えば，シンボルテーブルやドープベクターで利用可能な定義をデコードすることにより，そうした通信関数を定義できる．このような高レベルの通信関数はMPI仕様の一部でない．

これまで説明で用いられてきた基本データ型の代わりに，派生データ型で置換えることにより，より一般化された通信バッファを明確に定義できる．派生データ型は，この章で説明するコンストラクタに使って，基本データ型から構成される．また，派生データ型を構成するメソッドは再帰的に適用することができる．

汎用データ型は次の2つのことを規定する不可視オブジェクトである．

- 基本データ型の並び
- 整数(バイト)変位の並び

変位の並びの中のアイテムは，正であるとか，お互い重複がないとか，昇順であるとかの要件はない．したがって，アイテム間の順序は，格納の順序と一致する必要がないし，あるアイテムがその中で複数回現れてもよい．このような並びの対（あるいは対の並び）を**型マップ**と呼ぶ．基本データ型の並び（変位は考慮しない）は，そのデータ型の**型シグネチャ**である．

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

このような型マップで，$type_i$は基本データ型で$disp_i$は変位である．

$$Typesig = \{type_0, ..., type_{n-1}\}$$

これは上記の型マップに関連付けられた型シグネチャである．この型マップは，基底アドレス**buf**と共に用いて，通信バッファを指定する．この例では n個のエントリを持ち，i番目のエントリのアドレスは$buf + disp_i$で型は $type_i$になる．このような通信バッファから組み立てられたメッセージは，n個の値からなり，各値は$Typesig$で定義される型をもつ．

ほとんどのデータ型コンストラクタは，反復回数やブロック長の引数を持つ．これらの引数に対し許される値は非負整数である．ただし，これらの値が0の場合，型マップの中には要素を何も生じさせないし，データ型の上下限や範囲に対し何の影響も与えない．

汎用データ型のハンドルは基本データ型の代わりに送受信操作の引数として用いることができる．操作MPI_SEND(buf, 1, datatype,...)は基底アドレスbufとdatatypeで表される汎用データ型とで定義される送信バッファを使う．このときdatatype引数で決まる 型シグネチャを持つメッセージが生成される． MPI_RECV(buf, 1, datatype,...)は基底アドレスbufと datatypeで表される汎用データ型とで定義される受信バッファを使う．

汎用データ型は全ての送受信操作で利用できる． 第4.1.11節で第2引数countが1より大きな場合について論じる．

第3.2.2節に示した基本データ型は汎用データ型の特殊なもので定義済みである．つまり，MPI_INTは1つのint型のエントリを持ちその変位が0であるような，型マップ$\{(\mathsf{int}, 0)\}$を持つ定義済みのデータ型のハンドルである．他の基本データ型についても同様である．

データ型の**範囲**はこのデータ型のエントリによって占められる最初のバイトからなる最後のバイトまでの区間によって定義され，アライメントを満たすように丸められる．

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

の場合

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{4.1}$$

となる．$type_i$がk_iの倍数バイト アドレスのアライメントを必要とする場合，ϵ は$extent(Typemap)$ を一番近い$max_i k_i$の倍数バイトに丸めるのに必要な非負の値となる．**範囲**の正確な定義は107ページに示す．

例 4.1 $Type = \{(\text{double}, 0), (\text{char}, 8)\}$(変位0でdoubleが 1つ，続けて変位8でcharがある)を仮定する．さらにdoubleは8の倍数の アライメントが必要であるとする．すると，このデータ型の範囲は16になる（9が次の8の倍数（16）に切り上げられる）．文字の直後にdoubleが続くようなデータ型の 範囲も16である．

> **根拠** 範囲の定義は，構造体の配列の各構造体の最後に付加される パディングの量がアライメントの制約を満たすのに必要な最小限の量であることを前提としている．より厳密な範囲の管理は第4.1.6節に示す．このような厳密な管理は，例えばunion型が用いられるような，この前提が成り立たない場合に必要となる． （**根拠の終わり**）

4.1.1 明確なアドレスを持つ型コンストラクタ

Fortran言語の関数MPI_TYPE_CREATE_HVECTOR，MPI_TYPE_CREATE_HINDEXED，MPI_TYPE_CREATE_STRUCT，およびMPI_GET_ADDRESSでは，C言語とC++言語でそれぞれ引数の型 MPI_AintとMPI::Aintが使用される場合に，`INTEGER(KIND=MPI_ADDRESS_KIND)`型の引数を取る．ただし，Fortran 90言語のKINDの概念をサポートしていないFortran 77言語では，デフォルトのINTEGERが32ビットであるのにアドレスが64ビットである場合に，これらの引数はINTEGER*8型の引数をとる．

4.1.2 データ型コンストラクタ

Contiguous もっとも単純なデータ型コンストラクタはMPI_TYPE_CONTIGUOUSである．このコンストラクタは，連続な領域へのデータ型の反復を可能にする．

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	反復回数（非負の整数型）
IN	oldtype	旧データ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
{MPI::Datatype MPI::Datatype::Create_contiguous(int count) const （廃止された
              呼び出し形式, 第15.2節を参照）}
```

newtypeは，oldtypeのcount個のコピーを連結することによって得られるデータ型である．連結(concatenation)は，**範囲**を使って，連結されたコピーのサイズとして定義される．

例 4.2 oldtypeの型マップが $\{(\text{double}, 0), (\text{char}, 8)\}$, で，範囲が16，そしてcount $= 3$であるとする．このときnewtypeの型マップは次のようになる．

$$\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40)\};$$

つまり，doubleとcharが交互に，それぞれ $0, 8, 16, 24, 32, 40.$ の変位で現れる．

一般的に，oldtypeの型マップが次のようなものであると仮定する．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

さらに，範囲がexであるとする．そのとき，newtypeはcount · n個のエントリの，次のような型マップを持つ．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex),$$

$$..., (type_0, disp_0 + ex \cdot (\text{count} - 1)), ..., (type_{n-1}, disp_{n-1} + ex \cdot (\text{count} - 1))\}.$$

Vector　関数MPI_TYPE_VECTORはより汎用的なコンストラクタであり，等間隔のブロックからなる領域へのデータ型の反復を可能にする．各ブロックは，元のデータ型の同じ数のコピーを連結することによって得られる．また，ブロック間の間隔は，元のデータ型の範囲の倍数である．

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	ブロック数（非負の整数型）
IN	blocklength	個々のブロックの要素数（非負の整数型）
IN	stride	個々のブロックの先頭の間隔の要素数（整数型）
IN	oldtype	旧データ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
{MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength,
             int stride) const （廃止された呼び出し形式, 第15.2節を参照）}
```

例 4.3 再び型マップが $\{(\text{double}, 0), (\text{char}, 8)\}$, で，範囲が16であるようなoldtypeを仮定する． MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype)の呼び出しは，次のような型マップを持つデータ型を作る．

$$\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40),$$

$$(\text{double}, 64), (\text{char}, 72), (\text{double}, 80), (\text{char}, 88), (\text{double}, 96), (\text{char}, 104)\}.$$

つまり，元のデータ型の3 つのコピーからなる2つのブロックが，要素4つ分の間隔（$4 \cdot 16$バイト）で並ぶ．

例 4.4 MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype)の呼び出しは次のようなデータ型を作る．

$$\{(\text{double}, 0), (\text{char}, 8), (\text{double}, -32), (\text{char}, -24), (\text{double}, -64), (\text{char}, -56)\}.$$

一般的に，oldtypeの型マップが次のようなものであると仮定する．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

さらに，範囲がexであり，かつblocklength引数がblであるとする．新たに生成されるデータ型はcount $\cdot$ bl $\cdot\, n$個のエントリの，次のような型マップを持つ．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$

$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$

$$(type_0, disp_0 + (\text{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\text{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \text{stride} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \text{stride} \cdot ex), ...,$$

$$(type_0, disp_0 + (\text{stride} + \text{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\text{stride} + \text{bl} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \text{stride} \cdot (\text{count} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \text{stride} \cdot (\text{count} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + (\text{stride} \cdot (\text{count} - 1) + \text{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + (\text{stride} \cdot (\text{count} - 1) + \text{bl} - 1) \cdot ex)\}.$$

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)は MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype)あるいは MPI_TYPE_VECTOR(1, count, n, oldtype, newtype)（nは任意）と等価である．

Hvector　MPI_TYPE_CREATE_HVECTOR は，strideが要素数ではなくバイト単位であることを除けば，MPI_TYPE_VECTORと同じである．両方のベクトル型コンストラクタの使い方は第4.1.14節で説明する．（Hは “heterogeneous”（異機種）を意味する）

MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	ブロック数（非負の整数型）
IN	blocklength	個々のブロックの要素数（非負の整数型）
IN	stride	個々のブロックの先頭の間隔のバイト数（整数型）
IN	oldtype	旧データ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
{MPI::Datatype MPI::Datatype::Create_hvector(int count, int blocklength,
             MPI::Aint stride) const （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は廃止されたMPI_TYPE_HVECTORに代わるものである．第15章も参照すること．

oldtypeの型マップが次のようであると仮定する．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

さらに，範囲がexであり，かつblocklength引数がblであるとする．新たに生成されるデータ型はcount · bl · n個のエントリの，次のような型マップを持つ．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1}),$$

$$(type_0, disp_0 + ex), ..., (type_{n-1}, disp_{n-1} + ex), ...,$$

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \mathsf{stride}), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride}), ...,$$

$$(type_0, disp_0 + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1)), ..., (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1)), ...,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex)\}.$$

Indexed MPI_TYPE_INDEXEDは，ブロックの並びへの，元のデータ型の反復を可能にする（個々のブロックは元のデータ型の連結である）．ただし，個々のブロックは異なる数のコピーを持つことができ，また異なる変位を持つことができる．全てのブロックの変位は元の型の範囲の倍数である．

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	ブロック数，または array_of_displacements と array_of_blocklengthsのエントリ数（非負の整数型）
IN	array_of_blocklengths	ブロックごとの要素数（非負の整数型の配列）
IN	array_of_displacements	個々のブロックの変位，oldtypeの 範囲の倍数（整数型の配列）
IN	oldtype	旧データ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
{MPI::Datatype MPI::Datatype::Create_indexed(int count,
              const int array_of_blocklengths[],
              const int array_of_displacements[]) const （廃止された呼び出し形式，
              第15.2節を参照）}
```

例 4.5 型マップが $\{(\mathsf{double}, 0), (\mathsf{char}, 8)\}$, で，範囲が16であるようなoldtypeを仮定する．B = (3, 1)， D = (4, 0)としたときの MPI_TYPE_INDEXED(2, B, D, oldtype, newtype)は次のような型マップを持つデータ型を返す．

$$\{(\mathsf{double}, 64), (\mathsf{char}, 72), (\mathsf{double}, 80), (\mathsf{char}, 88), (\mathsf{double}, 96), (\mathsf{char}, 104),$$

$(\mathsf{double}, 0), (\mathsf{char}, 8)\}$.

つまり，元のデータの3つのコピーが変位64から，1つのコピーが変位0から始まる．

一般的に，oldtypeの型マップが次のようなものであると仮定する．

$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$,

さらに，範囲がexであり，array_of_blocklength引数がBであり，かつarray_of_displacements引数がDであるとする．新たに生成される データ型は $n \cdot \sum_{i=0}^{\mathsf{count}-1}$ B[i] 個のエントリを持ち，次のようである．

$\{(type_0, disp_0 + \mathsf{D[0]} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[0]} \cdot ex), ...,$

$(type_0, disp_0 + (\mathsf{D[0]} + \mathsf{B[0]} - 1) \cdot ex), ..., (type_{n-1}, disp_{n-1} + (\mathsf{D[0]} + \mathsf{B[0]} - 1) \cdot ex), ...,$

$(type_0, disp_0 + \mathsf{D[count\text{-}1]} \cdot ex), ..., (type_{n-1}, disp_{n-1} + \mathsf{D[count\text{-}1]} \cdot ex), ...,$

$(type_0, disp_0 + (\mathsf{D[count\text{-}1]} + \mathsf{B[count\text{-}1]} - 1) \cdot ex), ...,$

$(type_{n-1}, disp_{n-1} + (\mathsf{D[count\text{-}1]} + \mathsf{B[count\text{-}1]} - 1) \cdot ex)\}$.

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)はMPI_TYPE_INDEXED(count, B, D, oldtype, newtype)と等価である．ただし，

$\mathsf{D[j]} = j \cdot \mathsf{stride},\ j = 0, ..., \mathsf{count} - 1,$

かつ

$\mathsf{B[j]} = \mathsf{blocklength},\ j = 0, ..., \mathsf{count} - 1.$

である．

Hindexed MPI_TYPE_CREATE_HINDEXED は，array_of_displacementsでのブロックの変位の指定がoldtypeの範囲の倍数ではなく，バイト数であることを除いて，MPI_TYPE_INDEXEDと同じである．

MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	ブロック数，またはarray_of_displacementsとarray_of_blocklengthsのエントリ数（非負の整数型）
IN	array_of_blocklengths	ブロックごとの要素数（非負の整数型の配列）
IN	array_of_displacements	個々のブロックのバイト単位の変位（整数型の配列）
IN	oldtype	旧データ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_create_hindexed(int count, int array_of_blocklengths[],
MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
MPI_Datatype *newtype)
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
{MPI::Datatype MPI::Datatype::Create_hindexed(int count,
              const int array_of_blocklengths[],
              const MPI::Aint array_of_displacements[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は廃止されたMPI_TYPE_HINDEXEDに代わるものである． 第15章も参照すること．

oldtypeの型マップが次のようであると仮定する．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

さらに，範囲がexであり，array_of_blocklength引数がBであり，かつarray_of_displacements引数がDであるとする．新たに生成される データ型は$n \cdot \sum_{i=0}^{\text{count}-1}$ B[i]個のエントリの，次のような型マップを持つ．

$$\{(type_0, disp_0 + \text{D[0]}), ..., (type_{n-1}, disp_{n-1} + \text{D[0]}), ...,$$

$$(type_0, disp_0 + \text{D[0]} + (\text{B[0]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \text{D[0]} + (\text{B[0]} - 1) \cdot ex), ...,$$

$$(type_0, disp_0 + \text{D[count-1]}), ..., (type_{n-1}, disp_{n-1} + \text{D[count-1]}), ...,$$

$$(type_0, disp_0 + \text{D[count-1]} + (\text{B[count-1]} - 1) \cdot ex), ...,$$

$$(type_{n-1}, disp_{n-1} + \text{D[count-1]} + (\text{B[count-1]} - 1) \cdot ex)\}.$$

Indexed_block　この関数は，全てのブロックのブロック長が同じであることを除いて，MPI_TYPE_INDEXEDと同じである．ブロックサイズが常に 1である非構造格子から生じる間接アドレシングを使用するコード（ギャザー／スキャッタ）が多数ある．以下の便利な関数では一定のブロックサイズと任意の変位が使用できる．

MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype, newtype)

IN	count	変位の配列の長さ（非負の整数型）
IN	blocklength	ブロックのサイズ（非負の整数型）
IN	array_of_displacements	変位の配列（整数型の配列）
IN	oldtype	旧データ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_create_indexed_block(int count, int blocklength,
int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
    NEWTYPE, IERROR
{MPI::Datatype MPI::Datatype::Create_indexed_block(int count,
             int blocklength, const int array_of_displacements[]) const
             （廃止された呼び出し形式, 第15.2節を参照）}
```

Struct MPI_TYPE_CREATE_STRUCT [1] は最も汎用的なコンストラクタである．これまでの MPI_TYPE_CREATE_HINDEXEDを更に一般化して，個々のブロックを異なるデータ型の反復で構成することができる．

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	ブロック数（非負の整数型），または配列 array_of_types, array_of_displacements, array_of_blocklengthsのエントリ数
IN	array_of_blocklength	ブロックごとの要素数（非負の整数型の配列）
IN	array_of_displacements	個々のブロックのバイト単位の変位（整数型の配列）
IN	array_of_types	個々のブロックの要素の型（データ型オブジェクトのハンドルの配列）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[],
MPI_Datatype *newtype)
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
{static MPI::Datatype MPI::Datatype::Create_struct(int count,
             const int array_of_blocklengths[], const MPI::Aint
             array_of_displacements[],
```

[1] 訳者註：MPI-2.2の原文は“MPI_TYPE_STRUCT”であるが，“MPI_TYPE_CREATE_STRUCT”の誤り．MPI-3では修正済．

const MPI::Datatype array_of_types[]) （廃止された呼び出し形式, 第15.2節を参照）}

この関数は廃止されたMPI_TYPE_STRUCTに代わるものである． 第15章も参照すること．

例 **4.6** type1の型マップが次のようで，

$$\{(\text{double}, 0), (\text{char}, 8)\},$$

範囲が16であり，B = (2, 1, 3)，D = (0, 16, 26)， T = (MPI_FLOAT, type1, MPI_CHAR)であるとき， MPI_TYPE_STRUCT(3, B, D, T, newtype)は次のような型マップを持つ データ型を返す．

$$\{(\text{float}, 0), (\text{float}, 4), (\text{double}, 16), (\text{char}, 24), (\text{char}, 26), (\text{char}, 27), (\text{char}, 28)\}.$$

つまり，2つのMPI_FLOATのコピーが変位0から始まり，続いて1つの type1のコピーが変位16から始まり，3つのMPI_CHARのコピーが変位26から始まる(floatは4バイトを占めるものとする)．

一般に，array_of_types引数がTであるとし，ここでT[i]が次のような型マップを指すハンドルであるとする．

$$typemap_i = \{(type^i_0, disp^i_0), ..., (type^i_{n_i-1}, disp^i_{n_i-1})\},$$

さらに，範囲がex_iであり，array_of_blocklength引数がBであり，array_of_displacements引数がDであり，かつcount引数がcであるとする．そのとき，新たに生成されるデータ型は$\sum_{i=0}^{c-1}$ B[i] $\cdot\, n_i$個のエントリの，次のような型マップを持つ．

$$\{(type^0_0, disp^0_0 + \text{D[0]}), ..., (type^0_{n_0}, disp^0_{n_0} + \text{D[0]}), ...,$$

$$(type^0_0, disp^0_0 + \text{D[0]} + (\text{B[0]} - 1) \cdot ex_0), ..., (type^0_{n_0}, disp^0_{n_0} + \text{D[0]} + (\text{B[0]-1}) \cdot ex_0), ...,$$

$$(type^{c-1}_0, disp^{c-1}_0 + \text{D[c-1]}), ..., (type^{c-1}_{n_{c-1}-1}, disp^{c-1}_{n_{c-1}-1} + \text{D[c-1]}), ...,$$

$$(type^{c-1}_0, disp^{c-1}_0 + \text{D[c-1]} + (\text{B[c-1]} - 1) \cdot ex_{c-1}), ...,$$

$$(type^{c-1}_{n_{c-1}-1}, disp^{c-1}_{n_{c-1}-1} + \text{D[c-1]} + (\text{B[c-1]-1}) \cdot ex_{c-1})\}.$$

MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype)は MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype)と等価である． ただし， Tの各エントリはoldtypeと等しいとする．

4.1.3 サブ配列データ型コンストラクタ

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN	ndims	配列の次元数（正の整数型）
IN	array_of_sizes	配列全体の各次元でのデータ型oldtypeの要素数（正の整数型の配列）
IN	array_of_subsizes	サブ配列の各次元でのデータ型oldtypeの要素数（正の整数型の配列）
IN	array_of_starts	各次元でのサブ配列の開始座標（非負の整数型の配列）
IN	order	配列の格納順序フラグ（ステート型）
IN	oldtype	配列要素のデータ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_create_subarray(int ndims, int array_of_sizes[],
int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype
oldtype, MPI_Datatype *newtype)
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
    ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
{MPI::Datatype MPI::Datatype::Create_subarray(int ndims,
              const int array_of_sizes[], const int array_of_subsizes[],
              const int array_of_starts[], int order) const （廃止された呼び出し形式, 第15.2節を参照）}
```

サブ配列データ型コンストラクタは，n次元の配列のn次元のサブ配列を記述したMPIデータ型を作成する．サブ配列は全体の配列の中なら任意の場所に置いてよく，サブ配列はまた，全体の配列の内側に収まる限り，それを上限として，任意のサイズ（ただし非ゼロ）のサブ配列であってよい．この型のコンストラクタは，グローバルな配列を含んだ単一ファイルに対し，複数のプロセス間で分散されたサブ配列にアクセスするためのファイル型を生成するのに役立つ．MPIの入出力，特に405ページの 第13.1.1節を参照すること．

このデータ型コンストラクタは，任意の次元数の配列を扱うことができ，C言語およびFortran言語のどちらの行列の並び順[2]（言い換えると，行優先か列優先か）に対しても動作する．もちろん，C言語のプログラムがFortran言語の順序を使用してもよいし，Fortran言語のプログラムがC言語の順序を使用してもよいことに注意すること．

ndimsパラメータはデータ配列全体の次元数を指定するもので，このndimsは，array_of_sizes，array_of_subsizes，およびarray_of_startsの各配列の中の要素数を伝える．

n次元配列および要求されたサブ配列において，各次元でのデータ型oldtypeの要素数はそれぞれ，array_of_sizesおよびarray_of_subsizesで指定される．array_of_subsizesのど

[2]訳者註: 原文 ordered matrix

の次元に対しても（仮に次元iとして），array_of_subsizes[i] < 1 や array_of_subsizes[i] $>$ array_of_sizes[i] のような値を指定することは誤りである．

array_of_startsはそのサブ配列の各次元の開始座標を含む．配列には0から添字が振られる．array_of_startsのどの次元に対しても（仮に次元iとして），array_of_starts[i] < 0 や array_of_starts[i] $>$ (array_of_sizes[i] $-$ array_of_subsizes[i]) のような値を指定することは誤りである．

> **ユーザへのアドバイス** 配列の添字が1から振られたFortran言語のプログラムでは，サブ配列の特定の次元の開始座標がnの場合，その次元のarray_of_startsのエントリは n-1となる． （**ユーザへのアドバイス終わり**）

order引数では，配列全体とサブ配列の格納順序を指定する． 以下のうちの1つを設定しなければならない．

MPI_ORDER_C C言語の配列で使用される順序（つまり，行優先）

MPI_ORDER_FORTRAN Fortran言語の配列で使用される順序（つまり，列優先）

特別なパディングのないndims次元のサブ配列(newtype)を，関数Subarray()によって次のように定義できる．

$$\begin{aligned}\text{newtype} \;=\; & \text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\},\\ & \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\\ & \{start_0, start_1, \ldots, start_{ndims-1}\}, \text{oldtype})\end{aligned}$$

oldtypeの型マップが次のような形式であるとする．

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

$type_i$が定義済みのMPIデータ型であり，exがoldtypeの範囲であるとき，次の3つの式を使用してSubarray()関数を再帰的に定義する． 式4.2では基本手順を定義する． 式4.3では order = MPI_ORDER_FORTRANの場合の再帰手順を定義し，式4.4ではorder = MPI_ORDER_Cの場合の再帰手順を定義する．

$$\begin{aligned}&\text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\},\\ &\qquad \{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\})\\ =\;& \{(\text{MPI\_LB}, 0),\\ &(type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex),\\ &(type_0, disp_0 + (start_0 + 1) \times ex), \ldots, (type_{n-1},\\ &\qquad disp_{n-1} + (start_0 + 1) \times ex), \ldots\\ &(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots,\end{aligned} \tag{4.2}$$

$$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$$
$$(\mathsf{MPI\_UB}, size_0 \times ex)\}$$

$$\begin{aligned}
&\mathrm{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \mathrm{Subarray}(ndims - 1, \{size_1, size_2, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_1, subsize_2, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_1, start_2, \ldots, start_{ndims-1}\}, \\
&\qquad\qquad \mathrm{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \mathsf{oldtype}))
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&\mathrm{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \\
&= \mathrm{Subarray}(ndims - 1, \{size_0, size_1, \ldots, size_{ndims-2}\}, \\
&\qquad \{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\}, \\
&\qquad \{start_0, start_1, \ldots, start_{ndims-2}\}, \\
&\qquad\qquad \mathrm{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \mathsf{oldtype}))
\end{aligned} \tag{4.4}$$

入出力のコンテクストにおけるMPI_TYPE_CREATE_SUBARRAYの使用例は 第13.9.2節を参照すること.

4.1.4 分散配列データ型コンストラクタ

分散配列データ型コンストラクタはHPF形式[30]のデータの分散した配置をサポートしている. しかしHPFの場合と異なり, C言語またはFortran言語の配列の格納順序を指定することができる.

ユーザへのアドバイス この型コンストラクタを使用して, 以下のようにHPF形式のファイルビューを作成することができる. グループの全てのプロセスから同一引数（rankは例外で, 個別に設定する必要がある）でこのコンストラクタを呼び出すことにより, 補助ファイル型が作成される. その後, これらのファイル型は（同じdispおよび etypeと一緒に）ビューを定義する（MPI_FILE_SET_VIEWにより）のに使用される. MPI入出力, 特に405ページの 第13.1.1節と419ページの 第13.3節を参照すること. このビューを使用して, 集団データアクセス操作により（同一オフセットを使用して）HPF形式の分散した配置パターンが生成される. （**ユーザへのアドバイス終わり**）

MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs, array_of_psizes, order, oldtype, newtype)

IN	size	プロセスグループのサイズ（正の整数型）
IN	rank	プロセスグループのランク（非負の整数型）
IN	ndims	配列の次元とプロセスグリッドの次元の数（正の整数型）
IN	array_of_gsizes	グローバル配列の各次元での型oldtypeの要素数（正の整数型の配列）
IN	array_of_distribs	各次元での配列の分散した配置（ステート型の配列）
IN	array_of_dargs	各次元での分散した配置の引数（正の整数型の配列）
IN	array_of_psizes	各次元でのプロセスグリッドのサイズ（正の整数型の配列）
IN	order	配列の格納順序フラグ（ステート型）
IN	oldtype	旧データ型（ハンドル）
OUT	newtype	新データ型（ハンドル）

```
int MPI_Type_create_darray(int size, int rank, int ndims,
int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int
array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, OLDTYPE,
NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
    ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
{MPI::Datatype MPI::Datatype::Create_darray(int size, int rank, int ndims,
              const int array_of_gsizes[], const int array_of_distribs[],
              const int array_of_dargs[], const int array_of_psizes[],
              int order) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_TYPE_CREATE_DARRAYはoldtype要素のndims次元の分散した配置に対応するデータ型をndims次元の論理プロセスのグリッドに生成するのに使用できる．array_of_psizesの使用しない次元は1に設定する必要がある（104ページの例4.7を参照）．MPI_TYPE_CREATE_DARRAYを正しく呼び出すには，式 $\prod_{i=0}^{ndims-1} array\_of\_psizes[i] = size$ を満たす必要がある．プロセスグリッド内のプロセスの順序は，仮想カルテシアンプロセストポロジーの場合と同様，行優先が前提となる．

ユーザへのアドバイス Fortran言語とC言語の両方の配列で，プロセスグリッド内のプロセスの順序は行優先が前提となる．これは，MPIの仮想カルテシアンプロセストポロジーで使用される順序に整合する．このような仮想プロセストポロジーを作成したり，またはプロセスグリッド内のプロセスの座標を確認したりするために，該当するプロセストポロジー関数を使用することができる．255ページの第7章を参照すること．（**ユーザへのアドバイス終わり**）

配列の各次元は，次のいずれかの方法で分散させることができる．

- MPI_DISTRIBUTE_BLOCK - ブロックの分散した配置

- MPI_DISTRIBUTE_CYCLIC - サイクリックの分散した配置
- MPI_DISTRIBUTE_NONE - 次元の分散なし

定数MPI_DISTRIBUTE_DFLT_DARGはデフォルトの分散した配置引数を指定する．分散した配置を行わない次元の分散した配置の引数は無視される．次元の分散した配置がMPI_DISTRIBUTE_BLOCKであるどの次元に対しても（仮に次元iとして），array_of_dargs[i] $*$ array_of_psizes[i] $<$ array_of_gsizes[i] のような値を指定することは誤りである．

例えば，HPF配置ARRAY(CYCLIC(15))は分散した配置の引数が15のMPI_DISTRIBUTE_CYCLICに対応し，HPF配置ARRAY(BLOCK)は 分散した配置の引数がMPI_DISTRIBUTE_DFLT_DARGの MPI_DISTRIBUTE_BLOCKに対応する．

order引数は，MPI_TYPE_CREATE_SUBARRAYの場合と同様，格納順序を指定するのに使用される．そのため，この型コンストラクタで記述される配列は Fortran言語（列優先）またはC言語（行優先）の順序で格納することができる．orderに使用できる値は，MPI_ORDER_FORTRANと MPI_ORDER_Cである．

このルーチンは，“cyclic()”という関数で定義された型マップを使用して新しいMPI データ型を作成する（以下を参照）．

このルーチンは，MPI_DISTRIBUTE_DFLT_DARGを使わない MPI_DISTRIBUTE_CYCLICの場合に，汎用性を失うことなく，型マップを定義するのに十分である．

MPI_DISTRIBUTE_BLOCKおよびMPI_DISTRIBUTE_NONEは，以下のように次元iに対してMPI_DISTRIBUTE_CYCLICに変換することができる．

array_of_dargs[i]がMPI_DISTRIBUTE_DFLT_DARGと等しい MPI_DISTRIBUTE_BLOCKはarray_of_dargs[i]が次のように設定されたMPI_DISTRIBUTE_CYCLICと等価である．

$$(\text{array\_of\_gsizes[i]} + \text{array\_of\_psizes[i]} - 1)/\text{array\_of\_psizes[i]}.$$

array_of_dargs[i]がMPI_DISTRIBUTE_DFLT_DARGでない場合，MPI_DISTRIBUTE_BLOCKとMPI_DISTRIBUTE_CYCLICは 等価である．

MPI_DISTRIBUTE_NONEはarray_of_dargs[i]がarray_of_gsizes[i]に設定されたMPI_DISTRIBUTE_CYCLICと等価である．

最後に，array_of_dargs[i]がMPI_DISTRIBUTE_DFLT_DARGに等しいMPI_DISTRIBUTE_CYCLICは，array_of_dargs[i]が1に設定された MPI_DISTRIBUTE_CYCLICと等価である．

MPI_ORDER_FORTRANの場合，ndims次元の分散配列（newtype）は以下の部分コードによって定義される．

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i+1] = cyclic(array_of_dargs[i],
                         array_of_gsizes[i],
                         r[i],
                         array_of_psizes[i],
                         oldtype[i]);
}
newtype = oldtype[ndims];
```

MPI_ORDER_Cの場合，コードは以下のようになる．

```
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
   oldtype[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
                           array_of_gsizes[ndims - i - 1],
                           r[ndims - i - 1],
                           array_of_psizes[ndims - i - 1],
                           oldtype[i]);
}
newtype = oldtype[ndims];
```

ここで，$r[i]$は次元iでのプロセスグリッド内のプロセスの位置（ランクrankによる）である．$r[i]$の値は以下の部分コードによって与えられる．

```
t_rank = rank;
t_size = 1;
for (i = 0; i < ndims; i++)
        t_size *= array_of_psizes[i];
for (i = 0; i < ndims; i++) {
    t_size = t_size / array_of_psizes[i];
    r[i] = t_rank / t_size;
    t_rank = t_rank % t_size;
}
```

oldtypeの型マップの形式が次のとおりであるとする．

$$\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$$

ここで，$type_i$は定義済みのMPI データ型であり，exがoldtypeの範囲であるとする．
この場合，関数cyclic()は次のように定義される．

$$\begin{aligned}
&\text{cyclic}(darg, gsize, r, psize, \text{oldtype}) \\
&= \{(\text{MPI_LB}, 0), \\
&\quad (type_0, disp_0 + r \times darg \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex), \\
&\quad (type_0, disp_0 + (r \times darg + 1) \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex), \\
&\quad \ldots \\
&\quad (type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex), \\
&\\
&\quad (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex), \\
&\quad (type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \ldots,
\end{aligned}$$

$$
\begin{aligned}
&(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),\\
&\ldots\\
&(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),\\
&\qquad \vdots\\
&(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),\\
&(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex\\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)),\\
&\ldots\\
&(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex\\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)), \ldots,\\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex\\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)),\\
&(\mathsf{MPI\_UB}, gsize * ex)\}
\end{aligned}
$$

ここで，$count$は以下のコードによって定義される．

```
nblocks = (gsize + (darg - 1)) / darg;
count = nblocks / psize;
left_over = nblocks - count * psize;
if (r < left_over)
    count = count + 1;
```

ここで，$nblocks$はプロセッサー間で分散させる必要のあるブロックの数である．また，$darg_{last}$は以下の部分コードによって定義される．

```
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
     darg_last = darg;
else
     darg_last = num_in_last_cyclic - darg * r;
     if (darg_last > darg)
            darg_last = darg;
     if (darg_last <= 0)
            darg_last = darg;
```

例 4.7 HPFの分散した配置に対応するファイル型の生成を検討する．

```
      <oldtype> FILEARRAY(100, 200, 300)
!HPF$ PROCESSORS PROCESSES(2, 3)
!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

これは，実行にアタッチされたプロセスが6個あると仮定して，以下のFortran言語のコードで実現できる．

```
ndims = 3
array_of_gsizes(1) = 100
array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
array_of_dargs(1) = 10
array_of_gsizes(2) = 200
array_of_distribs(2) = MPI_DISTRIBUTE_NONE
array_of_dargs(2) = 0
array_of_gsizes(3) = 300
array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
array_of_psizes(1) = 2
array_of_psizes(2) = 1
array_of_psizes(3) = 3
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
     array_of_distribs, array_of_dargs, array_of_psizes,        &
     MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
```

4.1.5 アドレス関数とサイズ関数

汎用データ型の変位は，あるバッファの先頭アドレスからの相対的なものである．**絶対アドレス**をこれらの変位の代わりに用いることができる．絶対アドレスはアドレス空間の始点である「アドレス0」からの相対的な 変位として扱う．この初期アドレス0は定数MPI_BOTTOMで表される． したがって，buf引数にはMPI_BOTTOMを渡すことで，データ型の定義に通信バッファ内のエントリとして絶対アドレスを指定できる．

メモリ内での位置に対するアドレスは関数MPI_GET_ADDRESSを呼び出すことによって得られる．

MPI_GET_ADDRESS(location, address)

IN	location	呼び出し元メモリ内の位置(選択型)
OUT	address	位置のアドレス(整数型)

```
int MPI_Get_address(void *location, MPI_Aint *address)
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
{MPI::Aint MPI::Get_address(void* location)  （廃止された呼び出し形式, 第15.2節を参照）
              }
```

この関数は廃止されたMPI_ADDRESSに代わるものである． 第15章も参照すること．

locationの(バイト)アドレスを返す．

ユーザへのアドバイス 現行のFortran言語のMPIコードはこのバージョンのMPI実装で，修正なしで動作可能であるし，任意のシステムに移植可能である．しかし，

プログラム内で $2^{32}-1$ より大きいアドレスを使用すると動作しない可能性がある．新しく作成するコードは，これらの新しい関数を使うようにして書かれるべきである．これにより，C言語/C++言語との互換性が得られ，64ビットアーキテクチャ上の誤動作を回避することができる．しかし，そうした新しく作成するコードは，KIND宣言をサポートしない古いFortran 77言語環境に移植する場合に，（多少の）書き直しが必要となることもある． （**ユーザへのアドバイス終わり**）

例 4.8 MPI_GET_ADDRESSを配列に使う

```
   REAL A(100,100)
   INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
   CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
   CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
   DIFF = I2 - I1
! DIFFの値は909*sizeofrealである．I1とI2の値は
! \implementationによって決まる．
```

ユーザへのアドバイス C言語のユーザはMPI_GET_ADDRESSの使用を避け，アドレス演算子&を使用したいという場合がある．しかし，& **変換式**はアドレスではなくポインタであることに注意しなければならない．ISO C言語ではポインタ（あるいはポインタを変換したint）がオブジェクトの指し示す絶対アドレスを示すことを要求していないが，これが一般的ではある．さらに，セグメント化されたアドレス空間を持つマシン上では，参照に対して，唯一の定義が存在しない場合もある．MPI_GET_ADDRESSをC言語の変数への「参照」に利用することで，このようなマシン上での可搬性が保証される． （**ユーザへのアドバイス終わり**）

ユーザへのアドバイス ユーザへのアドバイス Fortran言語のコンパイラによって実行される引数のコピーとレジスタの最適化に関する問題を回避するため，第16.2.2節の「データのコピーとシーケンスの対応付けによる問題」（504ページ）と「レジスタの最適化に関する問題」（507ページ）のヒントに注意すること． （**ユーザへのアドバイス終わり**）

以下の補助関数は派生データ型に対する有用な情報を与えるものである．

MPI_TYPE_SIZE(datatype, size)

IN	datatype	データ型（ハンドル）
OUT	size	データ型のサイズ（整数型）

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
{int MPI::Datatype::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_TYPE_SIZEは，datatypeに関連付けられた型シグネチャの中の全エントリの合計サイズをバイト単位で返す．つまり，このデータ型で作成されるメッセージデータの合計サイズである． データ型内で複数回現れるエントリは複数回数えられる．

4.1.6 下限マーカと上限マーカ

型マップの上限と下限を明示的に指定し，107ページの定義と置き換えると便利な場合がある．これによって，最初や最後に「穴」を持つデータ型や，上限の後ろや下限の前に拡張されたエントリを持つようなデータ型を定義することができる．このような使い方の例を第4.1.14節に示す． また，ユーザが上限や範囲の計算に利用されるアライメントの規則を変更することができる．例えば，あるC言語のコンパイラではユーザがプログラム中の構造体のいくつかのデフォルトのアライメントを変更することが可能なこともある．このような場合，ユーザはこれらの構造体に合致するようにデータ型の上下限を明示的に指定する必要がある．

これを実現するために，MPI_LBおよびMPI_UBの，2つの「疑似データ型」が加えられた．これらはそれぞれ，データ型に対し下限と上限をマークするのに使うことができる．これらの疑似データ型は場所を占有しない ($extent(\mathsf{MPI\_LB}) = extent(\mathsf{MPI\_UB}) = 0$)．これらの疑似データ型は，マークされたデータ型のサイズや回数(count)に影響を与えないし，そのデータ型単独で生成されるメッセージの内容にも影響を与えない．しかし，データ型の範囲の定義には影響するので，データ型コンストラクタによるこのデータ型の反復結果には影響する．

例 4.9 D = (-3, 0, 6)，T = (MPI_LB, MPI_INT, MPI_UB)，かつ B = (1, 1, 1)とする．このとき，MPI_TYPE_STRUCT(3, B, D, T, type1)は範囲が9（-3から5まで（5を含む））で，変位0に整数値を持つようなデータ型を作る．これは，{(lb, -3), (int, 0), (ub, 6)} のような並びで表すことができる．MPI_TYPE_CONTIGUOUS(2, type1, type2) の呼び出しによって，このデータ型が2回反復されたら，新たに生成されるデータ型は {(lb, -3), (int, 0), (int,9), (ub, 15)} で表すことができる．（型ubのエントリは，より大きな変位を持つ型ubの別のエントリがあれば削除することができ，型lbのエントリは，より小さな変位を持つ型lbの別のエントリがあれば削除することができる．）

一般的に，

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

の場合，$Typemap$の下限は

$$lb(Typemap) = \begin{cases} \min_j disp_j & \text{型lbを持つエントリがない場合} \\ \min_j\{disp_j \text{ such that } type_j = \mathsf{lb}\} & \text{そうでない場合} \end{cases}$$

のように定義される．

同様に，$Typemap$の上限は

$$ub(Typemap) = \begin{cases} \max_j disp_j + sizeof(type_j) + \epsilon & \text{型ubを持つエントリがない場合} \\ \max_j\{disp_j \text{ such that } type_j = \mathsf{ub}\} & \text{そうでない場合} \end{cases}$$

のように定義される．

したがって，範囲は，

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

となる． $type_i$がk_iの倍数のバイトアドレスのアライメントを必要とする場合，ϵは$extent(Typemap)$ を$\max_i k_i$の次の倍数に丸めるために必要な非負の最小の値となる．

この範囲の定義の修正で，さまざまなデータ型コンストラクタの正式な定義がなされた．

4.1.7 データ型の範囲と上下限

以下の関数は，MPI_TYPE_UB，MPI_TYPE_LB，MPI_TYPE_EXTENTの3つの関数に代わるものである．Fortran言語呼び出し形式では，アドレスサイズの整数も返す．MPI_TYPE_UB，MPI_TYPE_LB，MPI_TYPE_EXTENTの使用は廃止された．

MPI_TYPE_GET_EXTENT(datatype, lb, extent)

IN	datatype	情報取得のためのデータ型（ハンドル）
OUT	lb	データ型の下限（整数型）
OUT	extent	データ型の範囲（整数型）

```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
MPI_Aint *extent)
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
{void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent) const （廃
            止された呼び出し形式, 第15.2節を参照）}
```

datatypeの下限と範囲を返す（107ページの 第4.1.6節で定義）．

MPIでは，下限マーカと上限マーカ（MPI_LBとMPI_UB）を使用して， データ型の範囲を変更することができる．これにより，連続するデータ型のストライドを制御することができるので，有用である．連続するデータ型は，データ型コンストラクタによる反復や，送信または受信呼び出しのcount引数による反復で生じる．しかし，ストライドの制御を実現するための現行の方法は使い勝手が悪く，制約も多い．MPI_LBとMPI_UBは「厄介」で，データ型で一度使用されると，上書きができない（例えば，新しいMPI_UBマーカを追加することにより上限を引き上げることができるが，既存のMPI_UBマーカよりも引き下げることはできない）．この変更を容易に行うため，新しいタイプのコンストラクタが用意されている． MPI_LBとMPI_UBの使用は廃止された．

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)

IN	oldtype	入力データ型（ハンドル）
IN	lb	データ型の新しい下限（整数型）
IN	extent	データ型の新しい範囲（整数型）
OUT	newtype	出力データ型（ハンドル）

```
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
extent, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
{MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb,
              const MPI::Aint extent) const （廃止された呼び出し形式, 第15.2節を参照）
              }
```

newtypeにoldtypeと同じである新しいデータ型のハンドルを返す．ただし，この新しいデータ型の下限はlbに設定され，上限はlb + extentに設定される． 古い**lb**マーカと**ub**マーカは消去され，下限マーカと上限マーカの新しいペアが，lb引数とextent引数によって示される位置に設定される．これは，count > 1 引数の通信操作の中で使用された場合や，新しい派生データ型のコンストラクションで使用された場合に，データ型の動作に影響を及ぼす．

ユーザへのアドバイス データ型の下限，上限，および範囲に設定したりアクセスしたりする場合，古いMPI-1関数ではなく，これら2つの新しい関数を使用することをユーザに強く推奨する．（**ユーザへのアドバイス終わり**）

4.1.8 データ型の正しい範囲

1対1ルーチンの上に実装されたスパニングツリーとしてギャザー（149ページの第5.5節も参照）を実装するとする．受信バッファはルートプロセスでしか有効でないため，中間ノードでデータを受信するための一時的空間を割り当てる必要がある．しかし，ユーザがMPI_UBとMPI_LBの値を使用して範囲を修正している場合，データ型の範囲に基づいて割り当てるのに必要な空間の量を見積もることができない．そのため，データ型の正しい範囲を返す関数を用意している．

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

IN	datatype	情報の取得が必要なデータ型（ハンドル）
OUT	true_lb	データ型の正しい下限（整数型）
OUT	true_extent	データ型の正しいサイズ（整数型）

```
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
MPI_Aint *true_extent)
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
{void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb,
              MPI::Aint& true_extent) const （廃止された呼び出し形式, 第15.2節を参照）
              }
```

true_lbは，指定されたデータ型によって示された，最低の格納単位のオフセットを返す．つまり，MPI_LBマーカを無視して，対応する型マップの下限を返す．true_extentは，指定された データ型の正しいサイズを返す．つまり，MPI_LBマーカおよびMPI_UBマーカを無視し，かつアライメントのための丸めを計算しないで，対応する型マップの範囲

を返す．datatypeに関連付けられた型マップが次のような場合，

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

true_lbとtrue_ubは以下のようになる．

$$true\_lb(Typemap) = min_j\{disp_j \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

$$true\_ub(Typemap) = max_j\{disp_j + sizeof(type_j) \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

またtrue_extentは次のようになる．

$$true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$$

(これを，107ページの第4.1.6節および108ページの第4.1.7節の関数
MPI_TYPE_GET_EXTENTの定義と比較すること．)

true_extentは，非圧縮のデータ型を保持するのに必要なメモリの最小バイト数である．

4.1.9 コミットと解放

データ型オブジェクトは通信で利用される前に**コミットされて**いなければならない．コミットされていないデータ型もコミットされたデータ型も，データ型コンストラクタの引数として用いることができる．基本データ型は「あらかじめコミットされている」ので，コミットする必要はない．

MPI_TYPE_COMMIT(datatype)

INOUT	datatype	コミットするデータ型（ハンドル）

```
int MPI_Type_commit(MPI_Datatype *datatype)
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

{`void MPI::Datatype::Commit()` （廃止された呼び出し形式, 第15.2節を参照）}

コミット操作は指定されたデータ型をコミットする．つまり通信バッファの内容ではなく，通信バッファの形式的記述をコミットする．したがって，ひとたびデータ型をコミットすれば，バッファの内容の変更や，実際には異なる開始アドレスによる異なるバッファの内容を通信するために，データ型を繰り返し再利用することができる．

実装者へのアドバイス システムは，通信を容易にするためにコミット時にデータ型を内部表現へ「コンパイル」してもよい．例えば，圧縮された表現からデータ型のフラットな表現に変え，最も有利な転送機能を選ぶことができる． （**実装者へのアドバイス終わり**）

MPI_TYPE_COMMIT はコミット済みの データ型に対しても実行可能である．この場合，無操作と等価である．

例 **4.10** 以下のコードはMPI_TYPE_COMMITの使い方の例である.

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
              ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used for communication
type2 = type1
              ! type2 can be used for communication
              ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
              ! new uncommitted type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used anew for communication
```

MPI_TYPE_FREE(datatype)

INOUT	datatype	解放されるデータ型（ハンドル）

```
int MPI_Type_free(MPI_Datatype *datatype)
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
{void MPI::Datatype::Free() （廃止された呼び出し形式, 第15.2節を参照）}
```

指定されたdatatypeに関連付けられたデータ型オブジェクトに，解放のためのマークを付け，datatypeをMPI_DATATYPE_NULLに設定する．このデータ型をその時点で使っている通信は，正常完了する．データ型を解放しても，解放されたデータ型から作成された他のデータ型には影響しない．派生データ型を生成するときのへの入力データ型引数があたかも，値渡しされたかのように，システムは振舞う.

実装者へのアドバイス 実装は，データ型を解放する時期を決定するために，そのデータ型を使っている通信の参照カウンタを保持してもよい．またある実装では，派生データ型コンストラクタへのデータ型引数ををコピーする代わりに，そのポインタを保持するようにしてもよい．この場合，データ型オブジェクトを解放する時期を知るために，有効なデータ型定義への参照を追跡する必要がある．（**実装者へのアドバイス終わり**）

4.1.10 データ型の複製

MPI_TYPE_DUP(type, newtype)

IN	type	データ型（ハンドル）
OUT	newtype	typeのコピー（ハンドル）

```
int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)
MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
    INTEGER TYPE, NEWTYPE, IERROR
```

{MPI::Datatype MPI::Datatype::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

MPI_TYPE_DUPは，データ型コンストラクタであり，関連付けられたキー値を含む，既存のtypeを複製する． 各キー値に対して，それぞれのコピーコールバック関数は新しいデータ型[3]において，このキー値に関連付けられた属性値を決定する．ただし，コピーコールバックがとってよいアクションの一つに，新しいデータ型から属性を削除することも含む．newtypeには，typeと完全に同じプロパティと，コピーされたキャッシュ情報のすべてをもつ，新しいデータ型が返される．データ型のキャッシュについては，244ページの第6.7.4節を参照すること．新しいデータ型は，同じ上限および下限を持つとともに，第4.1.13節の関数で完全にデコードした場合，実質同じ結果が得られる．newtypeは古いtypeと同様にコミットされた状態を持つ．

4.1.11 通信時の汎用データ型の利用

派生データ型のハンドルは，どこであれデータ型引数が要求される通信呼び出しに渡すことができる．MPI_SEND(buf, count, datatype , ...)形式の呼び出しでcount $>$ 1の場合，あたかもdatatypeがcount回連結された新たなデータ型が渡されたかのように考えることができる．そのため，MPI_SEND(buf, count, datatype, dest, tag, comm)は以下と等価である．

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm).
```

countとdatatypeを引数に持つ他の通信関数全てに同様のことがいえる．

datatypeが次のような型マップを持ち，範囲が$extent$であるような 送信操作 MPI_SEND(buf, count, datatype, dest, tag, comm)を実行すると仮定する．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

（空のエントリ，MPI_UBおよびMPI_LB の「疑似データ型」はこの型マップ中には列挙されないが，$extent$の値には影響を与える．）この送信操作は $n \cdot$ count 個のエントリを送信する．このとき，$i \cdot n + j$ 番目の エントリが $addr_{i,j} =$ buf $+ extent \cdot i + disp_j$ に配置され，型$type_j$をもつ．ただし，$i = 0, ...,$ count $- 1$ および $j = 0, ..., n - 1$ である．これらのエントリは連続である必要もなければ別々である必要もない．順序も任意である．

呼び出し側のプログラム中でアドレス$addr_{i,j}$に格納されている変数は$type_j$にマッチする型でなければならない．型のマッチングについては 第3.3.1節で定義している．送信されるメッセージは $n \cdot$ count 個のエントリからなり， $i \cdot n + j$ 番目のエントリは型$type_j$をもつ．

同様に，datatypeが次のような型マップを持ち，範囲が$extent$であるような受信操作MPI_RECV(buf, count, datatype, source, tag, comm, status) を実行すると仮定する．

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

[3]訳者註: 原文 communicator ; MPI Forum に確認予定.

（ここでも，空のエントリ，MPI_UBおよびMPI_LBの「疑似データ型」は型マップ中には列挙されないが，$extent$ の値には影響を与える.）この受信操作は$n \cdot$ count 個の エントリを受信する．このとき，$i \cdot n + j$ 番目のエントリが buf $+ extent \cdot i + disp_j$ に配置され，型$type_j$をもつ．届いたメッセージがk個の要素を持っている場合， k は$k \leq n \cdot$ count でなければならず，メッセージの$i \cdot n + j$ 番目の要素は$type_j$とマッチする型でなければならない.

型のマッチングは対応するデータ型の型シグネチャ，すなわち，基本データ型で表される構成要素の並びに従って定義される．型のマッチングはデータ型定義のいくつかの面，例えば変位（メモリ中の配置）や使われた中間型など，には依存しない.

例 4.11 この例は，型のマッチングが派生データ型の構成要素である基本データ型によって定義されることを示す.

```
...
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
...
CALL MPI_SEND( a, 4, MPI_REAL, ...)
CALL MPI_SEND( a, 2, type2, ...)
CALL MPI_SEND( a, 1, type22, ...)
CALL MPI_SEND( a, 1, type4, ...)
...
CALL MPI_RECV( a, 4, MPI_REAL, ...)
CALL MPI_RECV( a, 2, type2, ...)
CALL MPI_RECV( a, 1, type22, ...)
CALL MPI_RECV( a, 1, type4, ...)
```

個々の送信は，どの受信とでも適合する.

データ型は互いにデータの上書きのあるエントリを指定してもよい．しかし，このようなデータ型を受信に使うのは誤りである（これは実際に受信されたメッセージが，どのエントリも上書きしない位に短くても誤りである）.

datatypeが以下のような型マップを持つ，MPI_RECV(buf, count, datatype, dest, tag, comm, status)を実行すると仮定する.

$$\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}.$$

受信されたメッセージは受信バッファを全て満たす必要はなく，またnの倍数個の領域を満たす必要もない．任意の数k個の基本要素を受信可能である．ただし，$0 \leq k \leq$ count$\cdot n$ である．受信することができる基本要素の数は，問い合わせ関数 MPI_GET_ELEMENTSを使ってstatusから取り出すことができる.

MPI_GET_ELEMENTS(status, datatype, count)

IN	status	受信操作の戻りステータス（ステータス）
IN	datatype	受信操作に使ったデータ型（ハンドル）
OUT	count	受信された基本要素数（整数型）

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
{int MPI::Status::Get_elements(const MPI::Datatype& datatype) const （廃止された呼び出し形式, 第15.2節を参照）}
```

すでに定義した関数，MPI_GET_COUNT（第3.2.5節）はこれとは異なる動作をする．この関数は受信された「トップレベルのエントリ数」，すなわち datatype型の「コピー数」を返す．前の例で，MPI_GET_COUNTは $0 \leq k \leq$ count となるkを返す．MPI_GET_COUNTがkを返す場合，受信された基本要素数（MPI_GET_ELEMENTSが返す値）は$n \cdot k$である．受信された基本要素数が nの倍数ではない場合，すなわち，その受信操作がdatatypeの整数個の「コピー」を受信しなかった場合，MPI_GET_COUNTはMPI_UNDEFINEDを返す． datatype引数は，status変数を設定した受信呼び出しによって渡された引数とマッチしなければならない．

例 4.12 MPI_GET_COUNTとMPI_GET_ELEMENTSの使い方

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)
...
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
      CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=1
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)     ! returns i=MPI_UNDEFINED
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)  ! returns i=3
END IF
```

関数MPI_GET_ELEMENTSはまた，プローブされたメッセージの中の要素数を知るために，プローブ後にも使うことができる．これら2つの関数，MPI_GET_COUNTとMPI_GET_ELEMENTSは，基本データ型を用いた場合同じ値を返す．

> **根拠** MPI_GET_COUNTの定義に対する拡張は自然なものである．ある場合には，受信バッファが埋められているときに，この関数がcount引数を返すことを期待するかもしれない．別の場合，時としてdatatypeは，転送したいデータの基本単位，例えば，レコード（構造体）配列中の1個のレコード，を表現することを期待するかもしれない．そうした場合，いくつの構成要素を受信したかを，わざわざ各構成要素の中の要素の数で割り算することなく，知ることができるべきである．しかし，他の場合に，datatypeが 受信側のメモリ中の複雑なデータの配置を定義するために用いられているために，転送の基本単位を表すことができない場合がある．このような場合には関数MPI_GET_ELEMENTSを使用する必要がある． （**根拠の終わり**）

実装者へのアドバイス この定義は，受信によって，通信バッファを構成するために定義されたエントリの外側の領域を変更できないことを暗に意味する．特に，構造体中のパディング領域は，このような構造体があるプロセスから別のプロセスへコピーされる時に変更されてはならないことを示している．このため，パディング領域を含めて1つの連続なブロックと見立てる，単純な構造体のコピーの最適化は行なうことができない．実装の際には計算結果に影響のない範囲でこの最適化を自由に行なってよい．ユーザはパディングをメッセージの一部として明示的に加えることで，この最適化を「強制的に行う」ことができる． （**実装者へのアドバイス終わり**）

4.1.12 アドレスの正しい利用

C言語またはFortran言語で続けて宣言された変数が連続な領域に配置されるとは限らない．したがって，変位がある変数から他の変数へとまたがらないように注意して用いなければならない．同様に，セグメント化されたアドレス空間を持つ計算機の場合は，アドレスは一意ではなく，アドレスの計算には特別な方法がある．そのため，**アドレス**，つまり開始アドレスMPI_BOTTOMに対する相対変位の利用は制限する必要がある．

変数どうしが同じ**連続記憶領域**に属するとは，次のような場合である．つまり，それらの変数どうしが，同じ配列や，Fortran言語での同じCOMMONブロックや，C言語での同じ構造体に属する場合である．有効なアドレスは次のように再帰的に定義される．

1. 呼び出し側プログラムの変数を引数として渡した時，関数 MPI_GET_ADDRESSは有効なアドレスを返す．
2. 呼び出し側プログラムの変数を引数として渡した時，通信関数は引数buf を有効なアドレスとして評価する．
3. vが有効なアドレスであり，iが整数であるとき， vとv+iが同じ連続記憶領域にあれば，v+iは有効なアドレスである．
4. vが有効なアドレスであれば，MPI_BOTTOM + v は有効なアドレスである．

正しいプログラムは，通信バッファ内のエントリの位置を特定するために有効なアドレスのみを利用する．さらに，uとvが有効なアドレスである時，（整数の）差u - vはuとvが同じ連続した領域にある場合に限り計算できる．アドレスに関してこれ以外の算術演算は意味を持たない．

上記の規則は，派生データ型が同じ連続記憶領域内に全て含まれる通信バッファを定義するために用いられる限り，特に制約はない．しかし，同じ連続記憶領域内にない変数を含む通信バッファを作る場合には一定の制約に従う必要がある．なぜなら，基本的にそのような通信バッファを，通信呼び出しで使うことができるのは，次のような呼び出しに限られるからである．つまり，buf = MPI_BOTTOMおよびcount = 1の引数を指定し，かつすべての変位が有効な（絶対）アドレスであるようなdatatype引数を使う呼び出しに制約される．

ユーザへのアドバイス MPI呼び出しでホストプログラム中の配列やレコードの範囲を知ることができないこともあるため，ユーザのアドレス空間のオーバーフロー以外は， MPI実装によって「範囲外」の変位のエラーを検出できない可能性がある．（**ユーザへのアドバイス終わり**）

実装者へのアドバイス 連続したアドレス空間を持つ計算機上では（絶対）アドレスと，（相対）変位を区別する必要はない．MPI_BOTTOMは0で，アドレスと 変位の双方が整数となる．区別を必要とする計算機上で，アドレスは MPI_BOTTOMを含む式とみなすことができる． （**実装者へのアドバイス終わり**）

4.1.13 データ型のデコード

MPIのデータ型オブジェクトを使用すると，ユーザはメモリ内で任意のデータの配置を明確に定義することができる．不可視なデータ型オブジェクト内の配置情報にアクセスすることが有益である場合がいくつかある．不可視なデータ型オブジェクトはMPI以外でもさまざまに使用されている．さらに，多数のツールが，データ型に関する内部情報が表示しようとする．このために，データ型デコード機能が用意されている．この節で説明する2つの関数は共に，データ型の初期定義で使用された呼び出しシーケンスを再現するために，そのデータ型をデコードするのに使われる．これらは，ユーザがデータ型の型マップと型シグネチャを判別するために使用できる．

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, combiner)

IN	datatype	アクセスするデータ型（ハンドル）
OUT	num_integers	combinerを構成する呼び出しで使用される入力整数の数（非負の整数型）
OUT	num_addresses	combinerを構成する呼び出しで使用される入力アドレスの数（非負の整数型）
OUT	num_datatypes	combinerを構成する呼び出しで使用される入力データ型の数（非負の整数型）
OUT	combiner	結合子（ステート型）

```
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
int *num_addresses, int *num_datatypes, int *combiner)
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
{void MPI::Datatype::Get_envelope(int& num_integers, int& num_addresses,
             int& num_datatypes, int& combiner) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_TYPE_GET_ENVELOPEは，指定されたdatatypeに対し，そのdatatypeを作成した呼び出しで使った，入力引数の数および型に関する情報を返す．num_*引数に返される値は，デコードルーチンMPI_TYPE_GET_CONTENTSで十分な大きさの配列を用意する

ために使用できる．この呼び出しと返される値の意味を以下で説明する．combinerには，datatypeの作成で使用された MPIデータ型コンストラクタ呼び出しが反映される．

根拠

combinerにdatatypeの作成時に使用されたコンストラクタが反映されるよう規定されているため，デコードされた情報を使用して，オリジナルの作成に使用された呼び出しシーケンスを効率的に再現することができる．複数のコンストラクタ呼び出しのいずれの呼び出しでも，MPI_TYPE_GET_CONTENTSから得られる情報が同じ場合，そのいずれのコンストラクタ呼び出しも実質的に同じであると言える．例えば，C言語の呼び出しMPI_Type_hindexedとMPI_Type_create_hindexedとは常に実質的に同じであるが，Fortran言語の呼び出しMPI_TYPE_HINDEXEDは，MPI実装に依っては，これらのいずれの呼び出しとも異なることがある．これは最も有益な情報であり，内部表現が異なっていて，オリジナルのコンストラクタの呼び出しシーケンスを記憶しておくという実装の制約があったとしても，十分な妥当性がある．

デコードされた情報ではデータ型の複製も管理される．このことは，定義済みのデータ型と定義済みのデータ型の複製を区別する必要がある場合に重要である．前者は解放できない定数オブジェクトであり，後者は解放できる派生データ型である． （根拠の終わり）

以下の表では，combinerに返される値を左側に，それに対応する呼び出しを右側に示す．

combinerがMPI_COMBINER_NAMEDの場合，datatypeは 定義済みの名前付きデータ型となる．

アドレス引数を持つ廃止された呼び出しの場合，呼び出しで整数の引数を使用したか，アドレスサイズの引数を使用したかを区別する必要がでてくる．例えば，hvectorには2つの結合子，MPI_COMBINER_HVECTOR_INTEGERとMPI_COMBINER_HVECTORがある．前者はFortran言語からのMPI-1呼び出しの場合に使用され，後者はC言語またはC++言語からのMPI-1呼び出しの場合に使用される．しかし，MPI_ADDRESS_KIND = MPI_INTEGER_KIND（つまり，整数の引数とアドレスサイズの引数が同じである場合）となるシステムでは，Fortran言語からのMPI_TYPE_HVECTORの呼び出しによって構成されたデータ型に対し結合子MPI_COMBINER_HVECTORが返されることがある．同様に，Fortran言語からのMPI_TYPE_HINDEXEDの呼び出しによって構成されたデータ型に対しMPI_COMBINER_HINDEXEDが返されることがあり，Fortran言語からのMPI_TYPE_STRUCTの呼び出しによって構成されたデータ型に対し MPI_COMBINER_STRUCTが返されることがある． このようなシステムでは，アドレスサイズの引数をとるコンストラクタと整数の引数をとるコンストラクタは同じであるため，区別する必要はない．望ましい呼び出しは全て，アドレスサイズの引数を使用するため，この場合は2つも結合子は必要ない．

MPI_COMBINER_NAMED	定義済みの名前付きデータ型
MPI_COMBINER_DUP	MPI_TYPE_DUP
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR
MPI_COMBINER_HVECTOR_INTEGER	MPI_TYPE_HVECTOR （Fortran言語から）
MPI_COMBINER_HVECTOR	MPI_TYPE_HVECTOR （C言語/C++言語，Fortran言語の一部から）または MPI_TYPE_CREATE_HVECTOR
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED
MPI_COMBINER_HINDEXED_INTEGER	MPI_TYPE_HINDEXED （Fortran言語から）
MPI_COMBINER_HINDEXED	MPI_TYPE_HINDEXED （C言語/C++言語，Fortran言語の一部から）または MPI_TYPE_CREATE_HINDEXED
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK
MPI_COMBINER_STRUCT_INTEGER	MPI_TYPE_STRUCT （Fortran言語から）
MPI_COMBINER_STRUCT	MPI_TYPE_STRUCT （C言語/C++言語，Fortran言語の一部から）または MPI_TYPE_CREATE_STRUCT
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED

表 4.1: MPI_TYPE_GET_ENVELOPEから返されるcombiner値

根拠 オリジナルの呼び出しの作成時にアドレス情報が切り詰められた可能性があるかどうかを知ることが重要である．いくつかのルーチンではFortran言語からの廃止された呼び出しが，デフォルトのINTEGERのサイズがアドレスのサイズよりも小さい場合に，切り詰めの対象となることがあった． （**根拠の終わり**）

datatypeの作成の呼び出しで使用された実際の引数は，以下の呼び出しから取得することができる．

MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes)

IN	datatype	アクセスするデータ型（ハンドル）
IN	max_integers	array_of_integers内の要素数（非負の整数型）
IN	max_addresses	array_of_addresses内の要素数（非負の整数型）
IN	max_datatypes	array_of_datatypes内の要素数（非負の整数型）
OUT	array_of_integers	datatypeの構成に使用された整数引数を含む（整数型の配列）
OUT	array_of_addresses	datatypeの構成に使用されたアドレス引数を含む（整数型の配列）
OUT	array_of_datatypes	datatypeの構成に使用されたデータ型引数を含む（ハンドルの配列）

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
int max_addresses, int max_datatypes, int array_of_integers[],
MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[])
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES, IERROR)
    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
    ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
{void MPI::Datatype::Get_contents(int max_integers, int max_addresses,
             int max_datatypes, int array_of_integers[],
             MPI::Aint array_of_addresses[],
             MPI::Datatype array_of_datatypes[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

datatypeは定義済みの名前なしのデータ型，または派生データ型でなければならない．datatypeが定義済みの名前付きのデータ型であったならば，その呼び出しは誤りである．

max_integers，max_addresses，およびmax_datatypesに渡される値はそれぞれ，同じdatatype引数に対する MPI_TYPE_GET_ENVELOPE呼び出しで num_integers，num_addresses，およびnum_datatypesに返される値と少なくとも同じ大きさでなければならない．

根拠 引数max_integers，max_addresses，max_datatypesは，呼び出しでエラーチェックに使用できる． （**根拠の終わり**）

array_of_datatypesに返されるデータ型は，オリジナルのコンストラクタ呼び出しで使用されたデータ型と等価なデータ型オブジェクトのハンドルである．これらが派生データ型である場合，返されるデータ型は新しいデータ型オブジェクトで，ユーザが責任を持ってMPI_TYPE_FREEを使用してこれらのデータ型を解放する必要がある．これらが定義済みのデータ型である場合，返されるデータ型はその（定数の）定義済みのデータ型で，解放することはできない．

返される派生データ型のコミット状態は未定義である．つまり，データ型がコミットされていることもあれば，されていないこともある．さらに，返されるデータ型の属性の内容も未定義である．

MPI_TYPE_GET_CONTENTSはMPI_TYPE_CREATE_F90_REAL，

MPI_TYPE_CREATE_F90_INTEGER，またはMPI_TYPE_CREATE_F90_COMPLEX（定義済みの名前なしデータ型）を使用して構成されたdatatype引数を使用して呼び出すことができる．この場合，空のarray_of_datatypesが返される．

根拠 データ型の同等性の定義では，等価な定義済みデータ型どうしが等しいことを暗に意味する．同じ名前付き定義済みデータ型間で同じハンドルを使用するよう規定することで，使用されているデータ型を判断するために==または.EQ.比較演算子を使用することができる． （**根拠の終わり**）

実装者へのアドバイス array_of_datatypes で返されるデータ型は，ユーザからみて，あたかも各データ型が型コンストラクタの呼び出しで使用されたデータ型の等価なコピーとみなせる必要がある． array_of_datatypes用のデータ型に対し，新しくデータ型を生成するのか，それとも参照カウント機能などの別の機能を介して提供するのか，意味論が保持されている限り，実装に任されている．（**実装者へのアドバイス終わり**）

根拠 返されるデータ型のコミット状態と属性は，意図的にあいまいなままにされている．オリジナルの構成で使用されてたデータ型は，コンストラクタ呼び出しで使用されてから以降に変更されている可能性がある．属性が追加されたり，削除されたり，または修正されたりした可能性や，そのデータ型がコミットされた可能性もある．意味論上，これらの変更を追跡しない参照カウント実装が可能である．（**根拠の終わり**）

廃止されたデータ型コンストラクタの呼び出しで，Fortran言語のアドレス引数はINTEGER型である．望ましい呼び出しでは，アドレス引数は，INTEGER(KIND=MPI_ADDRESS_KIND)型の引数である．呼び出しMPI_TYPE_GET_CONTENTSはINTEGER(KIND=MPI_ADDRESS_KIND)型の引数に全て，アドレスを返す．このことは，廃止された呼び出しを使用した場合でも該当する．そのため，返される値の位置はC言語の呼び出し形式によって返されるものと同様に考えることができる．アドレスに関与する廃止された呼び出しのデータ型コンストラクタに対応する望ましい呼び出しを検査することによって確認することもできる．

根拠 全てのアドレス引数をarray_of_addresses引数に返すことにより，C言語およびFortran言語でのdatatypeのデコードの結果を同じ引数に渡すことができる．INTEGER(KIND=MPI_ADDRESS_KIND)型の整数が，古いMPI-1呼び出しにおけるデータ型構成で使用されたINTEGER引数と少なくとも同じ大きさであることが前提であれば，情報の損失は発生しない．（**根拠の終わり**）

以下では，datatypeに使われたデータ型コンストラクタに依存して，返される配列の各エントリにどんな値が置かれるかを定義する．また，以下ではそれらの配列の必要となるサイズも規定する．そのサイズは実際には，MPI_TYPE_GET_ENVELOPEによって返される値である．Fortran言語で，次の呼び出しがなされたと仮定し，説明を行う．

```
      PARAMETER (LARGE = 1000)
      INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE)
!     CONSTRUCT DATATYPE TYPE (NOT SHOWN)
      CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
      IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
        WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
        " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
        CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR)
      ENDIF
      CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
```

C言語でこれに類似する呼び出しは以下のとおりである.

```
#define LARGE 1000
int ni, na, nd, combiner, i[LARGE];
MPI_Aint a[LARGE];
MPI_Datatype type, d[LARGE];
/* construct datatype type (not shown) */
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
  fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
  fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
          LARGE);
  MPI_Abort(MPI_COMM_WORLD, 99);
};
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
```

C++言語のコードは上記のC言語のコードとに類似しており，同じ値が返される.

以下の説明では，小文字の名前の引数を使用する.

結合子がMPI_COMBINER_NAMEDの場合， MPI_TYPE_GET_CONTENTS呼び出しは誤りである.

結合子がMPI_COMBINER_DUPの場合,

Constructor argument	C & C++ location	Fortran location
oldtype	d[0]	D(1)

となり，ni = 0，na = 0，nd = 1となる.

結合子がMPI_COMBINER_CONTIGUOUSの場合,

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
oldtype	d[0]	D(1)

となり，ni = 1，na = 0，nd = 1となる.

結合子がMPI_COMBINER_VECTORの場合,

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	i[2]	I(3)
oldtype	d[0]	D(1)

となり，ni = 3，na = 0，nd = 1となる.

結合子がMPI_COMBINER_HVECTOR_INTEGERまたは MPI_COMBINER_HVECTORの場合，

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
stride	a[0]	A(1)
oldtype	d[0]	D(1)

となり，ni = 2，na = 1，nd = 1となる．

結合子がMPI_COMBINER_INDEXEDの場合，

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
oldtype	d[0]	D(1)

となり，ni = 2*count+1，na = 0，nd = 1となる．

結合子がMPI_COMBINER_HINDEXED_INTEGERまたは MPI_COMBINER_HINDEXEDの場合，

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
oldtype	d[0]	D(1)

となり，ni = count+1，na = count，nd = 1となる．

結合子がMPI_COMBINER_INDEXED_BLOCKの場合，

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
blocklength	i[1]	I(2)
array_of_displacements	i[2] to i[i[0]+1]	I(3) to I(I(1)+2)
oldtype	d[0]	D(1)

となり，ni = count+2，na = 0，nd = 1となる．

結合子がMPI_COMBINER_STRUCT_INTEGERまたは MPI_COMBINER_STRUCTの場合，

Constructor argument	C & C++ location	Fortran location
count	i[0]	I(1)
array_of_blocklengths	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_displacements	a[0] to a[i[0]-1]	A(1) to A(I(1))
array_of_types	d[0] to d[i[0]-1]	D(1) to D(I(1))

となり，ni = count+1，na = count，nd = countとなる．

結合子がMPI_COMBINER_SUBARRAYの場合，

Constructor argument	C & C++ location	Fortran location
ndims	i[0]	I(1)
array_of_sizes	i[1] to i[i[0]]	I(2) to I(I(1)+1)
array_of_subsizes	i[i[0]+1] to i[2*i[0]]	I(I(1)+2) to I(2*I(1)+1)
array_of_starts	i[2*i[0]+1] to i[3*i[0]]	I(2*I(1)+2) to I(3*I(1)+1)
order	i[3*i[0]+1]	I(3*I(1)+2]
oldtype	d[0]	D(1)

となり，ni = 3*ndims+2，na = 0，nd = 1となる．

結合子がMPI_COMBINER_DARRAYの場合，

Constructor argument	C & C++ location	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
array_of_gsizes	i[3] to i[i[2]+2]	I(4) to I(I(3)+3)
array_of_distribs	i[i[2]+3] to i[2*i[2]+2]	I(I(3)+4) to I(2*I(3)+3)
array_of_dargs	i[2*i[2]+3] to i[3*i[2]+2]	I(2*I(3)+4) to I(3*I(3)+3)
array_of_psizes	i[3*i[2]+3] to i[4*i[2]+2]	I(3*I(3)+4) to I(4*I(3)+3)
order	i[4*i[2]+3]	I(4*I(3)+4)
oldtype	d[0]	D(1)

となり，ni = 4*ndims+4，na = 0，nd = 1となる．

結合子がMPI_COMBINER_F90_REALの場合，

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

となり，ni = 2，na = 0，nd = 0となる．

結合子がMPI_COMBINER_F90_COMPLEXの場合，

Constructor argument	C & C++ location	Fortran location
p	i[0]	I(1)
r	i[1]	I(2)

となり，ni = 2，na = 0，nd = 0となる．

結合子がMPI_COMBINER_F90_INTEGERの場合，

Constructor argument	C & C++ location	Fortran location
r	i[0]	I(1)

となり，ni = 1，na = 0，nd = 0となる．

結合子がMPI_COMBINER_RESIZEDの場合，

Constructor argument	C & C++ location	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	D(1)

となり，ni = 0，na = 2，nd = 1となる.

4.1.14 例

以下の例で派生データ型の使い方を示す.

例 4.13 3次元配列の一部を送受信する.

```
      REAL a(100,100,100), e(9,9,9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C      extract the section a(1:17:2, 3:11, 2:10)
C      and store it in e(:,:,:).

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for a 1D section
      CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)

C     create datatype for a 2D section
      CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)

C     create datatype for the entire section
      CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice,
                             threeslice, ierr)

      CALL MPI_TYPE_COMMIT( threeslice, ierr)
      CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9,
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 4.14 行列の下三角部分を(厳密に)コピーする.

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     copy lower triangular part of array a
C     onto lower triangular part of array b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     compute start and size of each column
      DO i=1, 100
        disp(i) = 100*(i-1) + i
        blocklen(i) = 100-i
      END DO
```

```
C     create datatype for lower triangular part
      CALL MPI_TYPE_INDEXED( 100, blocklen, disp, MPI_REAL, ltype, ierr)

      CALL MPI_TYPE_COMMIT(ltype, ierr)
      CALL MPI_SENDRECV( a, 1, ltype, myrank, 0, b, 1,
                         ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 **4.15** 行列を転置する.

```
      REAL a(100,100), b(100,100)
      INTEGER row, xpose, sizeofreal, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     transpose matrix a onto b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for matrix in row-major order
      CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)

      CALL MPI_TYPE_COMMIT( xpose, ierr)

C     send matrix in row-major order and receive in column major order
      CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
                MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 **4.16** 転置の問題への別のアプローチ：

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(2), blocklen(2), type(2), row, row1, sizeofreal
      INTEGER  myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     transpose matrix a onto b

      CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)

C     create datatype for one row
      CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)

C     create datatype for one row, with the extent of one real number
      disp(1) = 0
      disp(2) = sizeofreal
      type(1)  = row
      type(2)  = MPI_UB
      blocklen(1)  = 1
```

```
      blocklen(2)  = 1
      CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)

      CALL MPI_TYPE_COMMIT( row1, ierr)

C     send 100 rows and receive in column major order
      CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                 MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

例 **4.17** 構造体の配列を操作する.

```
struct Partstruct
   {
      int    class;  /* particle class */
      double d[6];   /* particle coordinates */
      char   b[7];   /* some additional information */
   };

struct Partstruct    particle[1000];

int          i, dest, rank, tag;
MPI_Comm     comm;


/* build datatype describing structure */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
MPI_Aint     disp[3];
MPI_Aint     base;


/* compute displacements of structure components */

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i < 3; i++) disp[i] -= base;

MPI_Type_struct( 3, blocklen, disp, type, &Particletype);

   /* If compiler does padding in mysterious ways,
   the following may be safer */

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int          blocklen1[4] = {1, 6, 7, 1};
MPI_Aint     disp1[4];

/* compute displacements of structure components */

MPI_Address( particle, disp1);
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
```

```
for (i=0; i < 4; i++) disp1[i] -= base;

/* build datatype describing structure */

MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);


             /* 4.1:
        send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);


             /* 4.2:
        send only the entries of class zero particles,
        preceded by the number of such entries */

MPI_Datatype Zparticles;   /* datatype describing all particles
                              with class zero (needs to be recomputed
                              if classes change) */
MPI_Datatype Ztype;

MPI_Aint     zdisp[1000];
int          zblock[1000], j, k;
int          zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
   if (particle[i].class == 0)
      {
        zdisp[j] = i;
        zblock[j] = 1;
        j++;
      }

/* create datatype for class zero particles  */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Address(&j, zzdisp);
MPI_Address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);


       /* A probably more efficient way of defining Zparticles */

/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
   if (particle[i].index == 0)
```

```
      {
         for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
         zdisp[j] = i;
         zblock[j] = k-i;
         j++;
         i = k;
      }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);


               /* 4.3:
         send the first two coordinates of all entries */

MPI_Datatype Allpairs;      /* datatype for all pairs of coordinates */

MPI_Aint sizeofentry;

MPI_Type_extent( Particletype, &sizeofentry);

     /* sizeofentry can also be computed by subtracting the address
        of particle[0] from the address of particle[1] */

MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

      /* an alternative solution to 4.3 */

MPI_Datatype Onepair;   /* datatype for one pair of coordinates, with
                           the extent of one particle entry */
MPI_Aint disp2[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};

MPI_Address( particle, disp2);
MPI_Address( particle[0].d, disp2+1);
MPI_Address( particle+1, disp2+2);
base = disp2[0];
for (i=0; i<2; i++) disp2[i] -= base;

MPI_Type_struct( 3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit( &Onepair);
MPI_Send( particle[0].d, 1000, Onepair, dest, tag, comm);
```

例 4.18 前の例と同じ操作．ただし，データ型に絶対アドレスを使用する．

```
struct Partstruct
   {
      int class;
      double d[6];
      char b[7];
   };

struct Partstruct particle[1000];

           /* build datatype describing first array entry */
```

```
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          block[3] = {1, 6, 7};
MPI_Aint     disp[3];

MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
MPI_Type_struct( 3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute
   addresses */

                /* 5.1:
           send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);


                /* 5.2:
         send the entries of class zero,
         preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

MPI_Aint     zdisp[1000];
int          zblock[1000], i, j, k;
int          zzblock[2] = {1,1};
MPI_Datatype zztype[2];
MPI_Aint     zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
   if (particle[i].index == 0)
      {
         for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
         zdisp[j] = i;
         zblock[j] = k-i;
         j++;
         i = k;
      }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
   their absolute addresses*/

/* prepend particle count */
MPI_Address(&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

例 **4.19** 共用体の処理

```
union {
   int     ival;
   float   fval;
      } u[1000];

int     utype;

/* All entries of u have identical type; variable
   utype keeps track of their current type */

MPI_Datatype   type[2];
int            blocklen[2] = {1,1};
MPI_Aint       disp[2];
MPI_Datatype   mpi_utype[2];
MPI_Aint       i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;

type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

例 **4.20** この例では，データ型のデコード方法を示す．ルーチンprintdatatypeは データ型の要素を表示する．定義済みでないデータ型については， MPI_Type_freeを使用すること．

```
/*
  Example of decoding a datatype.

  Returns 0 if the datatype is predefined, 1 otherwise
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int printdatatype( MPI_Datatype datatype )
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
```

```
    int i;

    MPI_Type_get_envelope( datatype,
                           &num_ints, &num_adds, &num_dtypes, &combiner );
    switch (combiner) {
    case MPI_COMBINER_NAMED:
        printf( "Datatype is named:" );
        /* To print the specific type, we can match against the
           predefined forms. We can NOT use a switch statement here
           We could also use MPI_TYPE_GET_NAME if we prefered to use
           names that the user may have changed.
         */
        if      (datatype == MPI_INT)    printf( "MPI_INT\n" );
        else if (datatype == MPI_DOUBLE) printf( "MPI_DOUBLE\n" );
        ... else test for other types ...
        return 0;
        break;
    case MPI_COMBINER_STRUCT:
    case MPI_COMBINER_STRUCT_INTEGER:
        printf( "Datatype is struct containing" );
        array_of_ints   = (int *)malloc( num_ints * sizeof(int) );
        array_of_adds   =
                    (MPI_Aint *) malloc( num_adds * sizeof(MPI_Aint) );
        array_of_dtypes = (MPI_Datatype *)
            malloc( num_dtypes * sizeof(MPI_Datatype) );
        MPI_Type_get_contents( datatype, num_ints, num_adds, num_dtypes,
                               array_of_ints, array_of_adds, array_of_dtypes );
        printf( " %d datatypes:\n", array_of_ints[0] );
        for (i=0; i<array_of_ints[0]; i++) {
            printf( "blocklength %d, displacement %ld, type:\n",
                    array_of_ints[i+1], array_of_adds[i] );
            if (printdatatype( array_of_dtypes[i] )) {
                /* Note that we free the type ONLY if it
                   is not predefined */
                MPI_Type_free( &array_of_dtypes[i] );
            }
        }
        free( array_of_ints );
        free( array_of_adds );
        free( array_of_dtypes );
        break;
        ... other combiner values ...
    default:
        printf( "Unrecognized combiner type\n" );
    }
    return 1;
}
```

4.2 パックとアンパック

既存のいくつかの通信ライブラリは不連続なデータを送信するためにパック／アンパック関数を提供している．この場合，ユーザは送信前に連続なバッファに明示的にパックし，受信後に連続なバッファからアンパックする．利用者は第4.1節で説明したような派生データ型を，ほとんどの場合，明示的にパック，アンパックしなくてよい． ユーザ

は送信または受信するデータの配置を指定し，通信ライブラリが直接不連続なバッファにアクセスする．パック／アンパックルーチンは既存のライブラリとの互換性のために提供されている．また，MPIでは実現できないようないくつかの機能を提供する．例えば，受信内容が既に受信された部分に依存するような，複数に分かれたメッセージを受信することができる．他には，送出するメッセージをユーザが明示的に指定した場所にバッファリングすることで，システムのバッファリング規則を置き換えることができる．つまり，パック，アンパック操作を利用することでMPI上に別の通信ライブラリを容易に開発できる．

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

IN	inbuf	入力バッファ始点（選択型）
IN	incount	入力データ項目数（非負の整数型）
IN	datatype	個々の入力データ項目のデータ型（ハンドル）
OUT	outbuf	出力バッファ始点（選択型）
IN	outsize	出力バッファバイト長（非負の整数型）
INOUT	position	バッファ中での現在のバイト位置（整数型）
IN	comm	パックされたメッセージのコミュニケータ（ハンドル）

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
int outsize, int *position, MPI_Comm comm)
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
{void MPI::Datatype::Pack(const void* inbuf, int incount, void *outbuf,
              int outsize, int& position, const MPI::Comm &comm) const （廃止
              された呼び出し形式, 第15.2節を参照）}
```

inbuf，incount，およびdatatypeで指定される送信バッファ内のメッセージをoutbufおよびoutsizeで指定されるバッファへパックする．入力バッファには MPI_SENDで使うことができるバッファ指定できる．出力バッファは，アドレスoutbufから始まるoutsizeバイトの領域を含む，連続な領域である（長さは，MPI_PACKED型メッセージの通信バッファであるかのように，要素数ではなく，バイトで表される）．

positionの入力値は出力バッファ中の，パッキングに使われるべき書込み開始位置である．positionはパックされたメッセージのサイズ分インクリメントされる．positionの出力値は，出力バッファ中の次の書込み開始位置，つまりパックされたメッセージが占める領域の直後の位置である．comm引数はパックされたメッセージを送信するために後に利用するコミュニケータである．

MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

IN	inbuf	入力バッファ始点（選択型）
IN	insize	入力バッファバイト長（非負の整数型）
INOUT	position	現在のバイト位置（整数型）
OUT	outbuf	出力バッファ始点（選択型）
IN	outcount	アンパックする項目数（整数型）
IN	datatype	個々の出力データ項目のデータ型（ハンドル）
IN	comm	パックされるメッセージのコミュニケータ（ハンドル）

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
int outcount, MPI_Datatype datatype, MPI_Comm comm)
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
{void MPI::Datatype::Unpack(const void* inbuf, int insize, void *outbuf,
             int outcount, int& position, const MPI::Comm& comm) const （廃
             止された呼び出し形式, 第15.2節を参照）}
```

outbuf，outcount，およびdatatypeで指定される受信バッファへ inbufおよびinsizeで指定されるバッファ空間からデータをアンパックする．出力バッファにはMPI_RECVで利用できる，どの通信バッファでも指定できる．入力バッファは，アドレスinbufで始まるinsizeバイトの領域を含む連続な記憶領域である．positionの入力値は入力バッファ中の，パックされた当該メッセージが占める領域のうちの読み出し先頭位置である．positionはパックされたメッセージのサイズ分インクリメントされる．そのため，positionの出力値は，入力バッファ中の次の読み出し先頭位置，つまりアンパックされたメッセージが占めていた領域の直後の位置である．commはパックされたメッセージを受信するのに使ったコミュニケータである．

ユーザへのアドバイス ユーザへのアドバイス MPI_RECVとMPI_UNPACKの違いに注意すること．MPI_RECVではcount引数は受信可能な最大項目数を指定する．また，実際に受信される項目数は到着メッセージの長さで決まる．一方で，MPI_UNPACKではcount引数は，実際にアンパックされる項目数である．つまり，対応するメッセージの「サイズ」はpositionの増分に一致する．この違いの理由は2つあり，1つは，ユーザがアンパックすべき量を決定するので，「到着メッセージのサイズ」が事前に決まらないこと，もう1つは，項目数からアンパックされるべき「メッセージサイズ」を決定するのが容易ではないことが，その理由である．実際，異機種混在のシステムではこの数を前もって決めることはできない．（**ユーザへのアドバイス終わり**）

パックとアンパックの動作を理解するには，メッセージのデータ部分を，そのメッセージで送信される連続する値を連結して得られるデータの並びとして考えるとわかりやすい．パック操作は，あたかもメッセージをバッファに送信するかのようにこの並びを

バッファ空間に格納する．また，アンパック操作は，あたかもメッセージがバッファから受信されるかのようにこの並びをバッファ空間から取り出す（Fortran言語の内部ファイルや，C言語のsscanf等の同様の関数を考えるとわかりやすい）．

複数のメッセージが，1個の**パッキング単位**の中に連続してパックされてもよい．これは，MPI_PACKに対する，複数の連続した**同族の**呼び出しによってもたらされる．この場合，最初の呼び出しはposition = 0で行なわれ，続くそれぞれの呼び出しは直前のpositionの出力値を入力値として与える．ただし，その各呼び出しでoutbuf，outcount，およびcommには同じ値を使う．このパッキング単位は，個々の送信バッファを「連結」した送信バッファを使用して1回の送信操作によって，メッセージの中に格納されたのと等価の情報を含んでいる．

パッキング単位はMPI_PACKED型を使用して送信できる．任意の1対1通信，または集団通信関数を使用して，そのパッキング単位を形成するバイトの並びを，あるプロセスから他のプロセスへ，移動することができる．現時点で，このパッキング単位は任意のデータ型を用いて，どのような受信操作でも受信できる．型マッチング規則は，メッセージがMPI_PACKED型で送信される場合，緩和される．

どんなデータ型（MPI_PACKEDを含む）で送信されたメッセージでも，MPI_PACKED型で受信することができる．このようなメッセージは MPI_UNPACK呼び出しによってアンパックできる．

1つのパッキング単位を，複数の連続したメッセージにアンパックすることができる（あるいは，正規の「型が決まった」送信で生成されたメッセージでも可能）．これは，MPI_UNPACKに対する，複数の連続した同族の呼び出しによってもたらされる．最初の呼び出しはposition = 0で行なわれ，続くそれぞれの呼び出しは直前のpositionの出力値を入力値として与える．ただし，その各呼び出しでinbuf，insize，およびcommには同じ値を使う．

2つのパッキング単位の連結は必ずしも1つのパッキング単位になるとは限らない．またパッキング単位の一部分が1つのパッキング単位になるとは限らない．したがって，2つのパッキング単位を連結して，それから1つのパッキング単位としてその連結結果をアンパックすることはできない．また，パッキング単位の一部分を，独立した1つのパッキング単位としてアンパックすることはできない．一連の同族のパック呼び出し（あるいは正規の送信）によって作られた各パッキング単位は，一連の同族のアンパック呼び出しによって，1つの単位として，アンパックされなければならない．

> **根拠** パッキング単位における「アトミックな」パックとアンパックに対するこの制約は，実装が，パッキング単位の先頭に，付加情報を挿入することを可能にする．例えば，送り手のアーキテクチャの記述などの情報を付加することができる（異機種環境での型変換に使用するために）．（**根拠の終わり**）

以下の呼び出しによって，ユーザはメッセージをパックするのに必要な領域を知ることができる．つまり，バッファ領域の割り当てを管理することができる．

MPI_PACK_SIZE(incount, datatype, comm, size)

IN	incount	パック呼び出しに渡すcount引数（非負の整数型）
IN	datatype	パック呼び出しに渡すdatatype引数（ハンドル）
IN	comm	パック呼び出しに渡すcommunicator引数（ハンドル）
OUT	size	パックされたメッセージの上限値（バイト単位）（整数型）

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
int *size)
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
{int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const （廃
            止された呼び出し形式, 第15.2節を参照）}
```

MPI_PACK_SIZE(incount, datatype, comm, size)の呼び出しは，MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm) の呼び出しの結果であるposition引数の増加分の上限をsizeに返す.

根拠 この呼び出しは，厳密な値ではなく，上限を返す．なぜなら，メッセージをパックするのに要する領域の大きさは状況に依存する可能性があるからである（例えば，パッキング単位にパックされた最初のメッセージはより多くの領域を占める可能性がある）．（**根拠の終わり**）

例 **4.21** MPI_PACKを使用する例

```
int         position, i, j, a[2];
char        buff[1000];

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
   /* SENDER CODE */

   position = 0;
   MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
   MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
   MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else  /* RECEIVER CODE */
   MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

例 **4.22** 詳細な例

```
int   position, i;
float a[1000];
char  buff[1000];

MPI_Comm_rank(MPI_Comm_world, &myrank);
if (myrank == 0)
{
```

```
  /* SENDER CODE */

  int len[2];
  MPI_Aint disp[2];
  MPI_Datatype type[2], newtype;

  /* build datatype for i followed by a[0]...a[i-1] */

  len[0] = 1;
  len[1] = i;
  MPI_Address( &i, disp);
  MPI_Address( a, disp+1);
  type[0] = MPI_INT;
  type[1] = MPI_FLOAT;
  MPI_Type_struct( 2, len, disp, type, &newtype);
  MPI_Type_commit( &newtype);

  /* Pack i followed by a[0]...a[i-1]*/

  position = 0;
  MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);

  /* Send */

  MPI_Send( buff, position, MPI_PACKED, 1, 0,
            MPI_COMM_WORLD);

/* *****
   One can replace the last three lines with
   MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   ***** */
}
else if (myrank == 1)
{
   /* RECEIVER CODE */

  MPI_Status status;

  /* Receive */

  MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);

  /* Unpack i */

  position = 0;
  MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);

  /* Unpack a[0]...a[i-1] */
  MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

例 4.23 各プロセスからルートにcountとcount個の文字が送信され，ルートで全ての文字が1つの文字列に連結される.

```
int  count, gsize, counts[64], totalcount, k1, k2, k,
     displs[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
```

```
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

      /* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);

      /* pack count, followed by count characters */
position = 0;
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);

if (myrank != root) {
      /* gather at root sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, NULL, 0,
            MPI_DATATYPE_NULL, root, comm);

      /* gather at root packed messages */
   MPI_Gatherv( lbuf, position, MPI_PACKED, NULL,
            NULL, NULL, NULL, root, comm);

} else {   /* root code */
      /* gather sizes of all packed messages */
   MPI_Gather( &position, 1, MPI_INT, counts, 1,
            MPI_INT, root, comm);

      /* gather all packed messages */
   displs[0] = 0;
   for (i=1; i < gsize; i++)
     displs[i] = displs[i-1] + counts[i-1];
   totalcount = displs[gsize-1] + counts[gsize-1];
   rbuf = (char *)malloc(totalcount);
   cbuf = (char *)malloc(totalcount);
   MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf,
            counts, displs, MPI_PACKED, root, comm);

       /* unpack all messages and concatenate strings */
   concat_pos = 0;
   for (i=0; i < gsize; i++) {
      position = 0;
      MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, &count, 1, MPI_INT, comm);
      MPI_Unpack( rbuf+displs[i], totalcount-displs[i],
            &position, cbuf+concat_pos, count, MPI_CHAR, comm);
      concat_pos += count;
   }
   cbuf[concat_pos] = '\0';
}
```

4.3 標準のMPI_PACKおよびMPI_UNPACK

これらの関数は第13.5.2節で指定された“external32”データ形式でバッファとの間でデ

ータの読み取り／書き込みを行い，パックに必要なサイズを計算する．将来の拡張性のために第1引数にデータ形式を指定できるようになっているが，現在，datarep引数に使用できる値は“external32”のみである．

ユーザへのアドバイス これらの関数を使用して，例えば，あるMPI実装から別のMPI 実装にポータブルな形式で型が決まったデータを送信することができる．（**ユーザへのアドバイス終わり**）

バッファには，ヘッダなしで，パックされたデータのみが含まれる．MPI_PACK_EXTERNALによってパックされたデータの送信と受信には，MPI_BYTEが使用されなくてはならない．

根拠 MPI_PACK_EXTERNALにはメッセージにヘッダがないことを指定し，データの形式も指定する．MPI_PACKではヘッダを使用することができる（使用されることもある）ため，MPI_PACK_EXTERNALでパックされたデータに対して データ型MPI_PACKEDを使用することはできない． （**根拠の終わり**）

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)

IN	datarep	データ表現（文字列）
IN	inbuf	入力バッファ始点（選択型）
IN	incount	入力データ項目数（整数型）
IN	datatype	個々の入力データ項目のデータ型（ハンドル）
OUT	outbuf	出力バッファ始点（選択型）
IN	outsize	出力バッファバイト長（整数型）
INOUT	position	バッファ中での現在のバイト位置（整数型）

```
int MPI_Pack_external(char *datarep, void *inbuf, int incount,
MPI_Datatype datatype, void *outbuf, MPI_Aint outsize, MPI_Aint *position)
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
POSITION, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
{void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
             int incount, void* outbuf, MPI::Aint outsize,
             MPI::Aint& position) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, position)

IN	datarep	データ表現（文字列）
IN	inbuf	入力バッファ始点（選択型）
IN	insize	入力バッファバイト長（整数型）
INOUT	position	バッファ中での現在のバイト位置（整数型）
OUT	outbuf	出力バッファ始点（選択型）
IN	outcount	出力データ項目数（整数型）
IN	datatype	出力データ項目のデータ型（ハンドル）

```
int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
MPI_Aint *position, void *outbuf, int outcount, MPI_Datatype datatype)
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
DATATYPE, IERROR)
    INTEGER OUTCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
{void MPI::Datatype::Unpack_external(const char* datarep,
             const void* inbuf, MPI::Aint insize, MPI::Aint& position,
             void* outbuf, int outcount) const   （廃止された呼び出し形式, 第15.2節
             を参照）}
```

MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)

IN	datarep	データ表現（文字列）
IN	incount	入力データ項目数（整数型）
IN	datatype	個々の入力データのデータ型（ハンドル）
OUT	size	出力バッファバイト長（整数型）

```
int MPI_Pack_external_size(char *datarep, int incount,
MPI_Datatype datatype, MPI_Aint *size)
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    CHARACTER*(*) DATAREP
{MPI::Aint MPI::Datatype::Pack_external_size(const char* datarep,
             int incount) const （廃止された呼び出し形式, 第15.2節を参照）}
```

第5章

集団的通信

5.1 概論と概要

集団的通信は，複数のプロセスから成る1つまたは複数のグループに関連した通信として定義される．MPIで規定するこのタイプの機能としては次のものがある．

- MPI_BARRIER: グループ内の全メンバ にまたがるバリア同期．（第5.3節）
- MPI_BCAST: グループ内のあるメンバから全メンバへのブロードキャスト（第5.4節）．これは図5.1に “broadcast” として示されている．
- MPI_GATHER, MPI_GATHERV: グループ内の全メンバ からあるメンバへのデータのギャザー（第5.5節）．これは図5.1に“gather” として示されている．
- MPI_SCATTER, MPI_SCATTERV: グループ内のあるメンバから全メンバへのデータのスキャッタ（第5.6節）．これは図5.1に“scatter” として示されている．
- MPI_ALLGATHER, MPI_ALLGATHERV: グループ内の全メンバがの結果を受信するギャザーのバリエーション（第5.7節）．これは図5.1で “allgather” として示されている．
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW: グループ内の全メンバから全メンバへのデータのスキャッタ／ギャザー（完全交換）とも呼ばれる）（第5.8節）．これは図5.1で“complete exchange” として示される．
- MPI_ALLREDUCE, MPI_REDUCE: sum，max，min，またはユーザ定義関数などの結果をグループ内の全メンバへ戻したり，あるメンバだけに戻したりするバリエーションをもつ大域的なリデュース操作（第5.9節）．
- MPI_REDUCE_SCATTER: リデュースおよびスキャッタ操作の組み合わせ（第5.10節）．
- MPI_SCAN, MPI_EXSCAN: グループ内の全メンバにまたがるスキャン（プリフィックスとも呼ばれる）（第5.11節）．

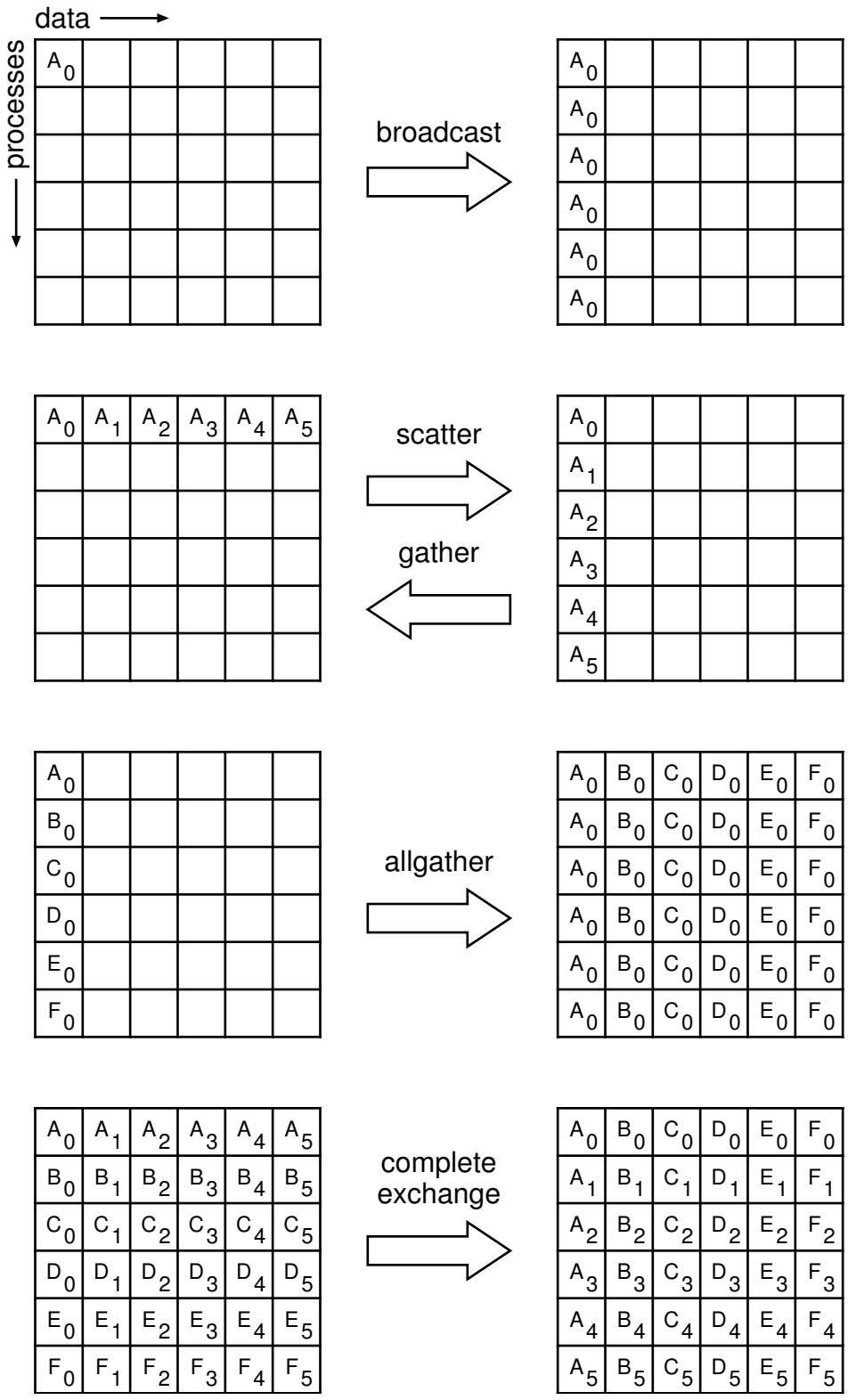

図 5.1: 6プロセスのグループの集団移動関数の図解．それぞれの場合において，各行はあるプロセスの中のデータ位置を表している．すなわち，ブロードキャストでは，はじめに第1プロセスだけがデータA_0を持っているが，ブロードキャスト後，全プロセスがそのデータA_0を持つ．

集団的ルーチンの呼び出しでの 重要な引数の1つは，参加するプロセスの1つ または複数の グループを定義し，操作のためのコンテクストを提供するコミュニケータである． これについては，第5.2節で詳しく説明する．集団操作の構文および意味は，1対1操作の構文および意味と整合するように定義される．そのため，一般的なデータ型が使用でき，これは 第4章 で説明したように送信プロセスと受信プロセスの間で一致していなければならない． ブロードキャストやギャザーといったいくつかの集団的ルーチンでは，1つのプロセスだけがメッセージを発信したり受信したりする．そのプロセスは**ルート**と呼ばれる． 集団関数の引数で，「ルートでのみ意味を持つ」指定されるものは，ルートを除く全参加プロセスでは無視される．通信バッファ，一般のデータ型，および型一致規則の詳細については第4章を，グループの定義方法およびコミュニケータの作成方法については第6章を参照すること．

集団操作の型一致条件は，1対1通信における送信側と受信側との間の対応条件よりも厳しい．つまり，集団操作では，送信されるデータ量は受信側が指定するデータ量と厳密に一致していなければならない．しかし送信側と受信側とで型マップ（メモリ内のレイアウト，第4.1節を参照）が違っていても許される．

集団的ルーチンの呼び出しは集団操作への関与が完了するとすぐに戻ることができる（しかし，戻ることを強制するわけではない）．呼び出しの完了は，呼び出しプロセスが自由に通信バッファの領域を変更できることを示す．しかし，それはグループ内の他のプロセスが操作を完了，あるいは開始していることを示すものではない（操作の説明で特記している場合を除く）．すなわち，集団的通信の呼び出しは，全呼び出しプロセスの同期の効果をもたらす場合もあれば，そうでない場合もある．当然，この文の記述はバリア同期関数には適用されない．

集団的通信の呼び出しでは，1対1通信と同じコミュニケータを使用することができる．MPIでは，集団的通信の呼び出しで作られたメッセージが1対1通信で作られたメッセージと混同されないことを保証している．集団的ルーチンの正しい使用法についての詳細は，第5.12節に記載している．

根拠 データ一致制限（型の対応で）によって，送信データの量を確認するためにMPI_RECVの引数statusと類似の機能を設けるという煩わしいことを避けるようにした．集団的ルーチンのいくつかは，ステータス値の配列を必要とする．

様々な集団関数の実装が可能となるべく，同期に関して記述されている．

集団操作は，メッセージタグの引数を受け付けない． MPIが将来に改訂されたとき，ノンブロッキングな集団関数の定義において，多重な，あるいは，保留された集団的通信における曖昧さを取り除くためにタグ（または類似のメカニズム）の追加が必要になるであろう． （**根拠の終わり**）

ユーザへのアドバイス 集団操作の副次作用である同期に頼ることは，正しいプログラムにとって危険である．例えば，一部の実装で同期という副次作用をブロードキャストルーチンにもたらす場合があるとしても，標準ではこの作用を要求していないし，これに依存するプログラムは可搬でない．

一方，正しくて移植性のあるプログラムは，集団呼び出しでは同期をとっているかもしれないということを考慮しなければならない．同期による副次作用に頼ることはできないが，その副次作用を考慮してプログラムしなければならない．こうした問題点について第5.12節で詳述する． （ユーザへのアドバイス終わり）

実装者へのアドバイス ベンダーは自社のアーキテクチャにあった最適な集団的ルーチンを書くことができるし，MPIの1対1通信関数や，いくつかの補助関数を使用するだけで完璧な集団のライブラリを書くことができる．その時実行中の1対1通信のと処理の干渉を回避するために，1対1通信の上位での実装で，集団操作用にユーザからは見えない隠れた専用のコミュニケータを作ることも可能である．この点についてはさらに第5.12節で詳述する． （実装者へのアドバイス終わり）

集団的ルーチンの多くの説明で，ブロッキングMPI の1対1ルーチンの図を示している．これは，どのプロセスでどのようなデータが送受信されるかを示すためのものである．これらの多くの例は正しいMPIのプログラムではなく，単純化するため，バッファの制約がないという前提になっていることが多い．

5.2 コミュニケータ引数

集団関数の重要なコンセプトは，操作に参加するプロセスの1つまたは複数のグループを持つということである．ルーチンは，明示的な引数としてグループの識別子を持たない．その代わりに，コミュニケータ引数がある．グループとコミュニケータについては第6節で詳しく説明する．この章では，コミュニケータにはグループ内コミュニケータとグループ間コミュニケータの2種類があることを理解しておけば十分である． グループ内コミュニケータは，コンテクストと結びついた1つのプロセスグループのための識別子と考えることができる．グループ間コミュニケータは，コンテクストと結びついた 2つのプロセスグループを識別する．

5.2.1 グループ内コミュニケータ集団操作の仕様

グループ内コミュニケータで識別されるグループ内の全てのプロセスは 集団的ルーチンを呼び出す必要がある．

多くの場合，集団的通信はグループ内コミュニケータのために「インプレイス」で実行することができ，出力バッファを入力バッファと同じとすることができる．これは，実行する操作に応じて，送信バッファまたは受信バッファ引数の代わりに特別な引数の値MPI_IN_PLACEを使用して指定する．

根拠 「インプレイス」操作は MPI実装とユーザの両方による不必要なメモリ動作を低減するために用意されている．送信バッファと受信バッファが同じアドレスを持っているかどうかをテストするための単純なチェックは一部の場合（MPI_ALLREDUCEなど）には機能するが，それ以外の場合（MPI_GATHERなど，ルートが0でない場合）には適切でない．また，Fortran言語では引数のエイリアス

を明示的に禁止しており，特別な値の使用により「インプレイス」操作を示すアプローチにより問題を軽減することができる． （根拠の終わり）

ユーザへのアドバイス 「インプレイス」オプションを使用することにより，多くの集団呼び出しの受信バッファを送受信バッファにすることができる．そのため，INTENT を含むFortran言語の呼び出し形式ではこれらをOUTではなく，INOUTとしてマークする必要がある．

MPI_IN_PLACEは特別な値で，使用するにはMPI_BOTTOMと同じ制約がある．

一部のグループ間コミュニケータ集団操作では「インプレイス」オプションがサポートされていない（MPI_ALLTOALLVなど）． （ユーザへのアドバイス終わり）

5.2.2 グループ間コミュニケータへの集団操作の適用

集団操作がグループ間コミュニケータに適用される方法を理解するため，大部分のMPIグループ内コミュニケータ集団操作を以下のいずれかのカテゴリに分類して考えることができる（例えば， 文献[43]を参照）．

全対全 全てのプロセスが結果に関与する．全てのプロセスが結果を受信する．

- MPI_ALLGATHER, MPI_ALLGATHERV
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
- MPI_ALLREDUCE, MPI_REDUCE_SCATTER
- MPI_BARRIER

全対1 全てのプロセスが結果に関与する．1つのプロセスが結果を受信する．

- MPI_GATHER, MPI_GATHERV
- MPI_REDUCE

1対全 1つのプロセスが結果に関与する．全てのプロセスが結果を受信する．

- MPI_BCAST
- MPI_SCATTER, MPI_SCATTERV

その他 上記のカテゴリに適合しない集団操作．

- MPI_SCAN, MPI_EXSCAN

MPI_SCANとMPI_EXSCANのデータ移動パターンはこの分類法に適合しない．

グループ間コミュニケータへの集団的通信の適用は，2つのグループで適切に説明される．例えば，全対全のMPI_ALLGATHER操作は1つのグループの全てのメンバからデータをギャザーし，結果を他のグループの全てのメンバに提示する操作として説明することができる（図5.2を参照）．もう1つの例では，1対全のMPI_BCAST操作は，

1つのグループの1つのメンバから他のグループの全てのメンバにデータを送信する．MPI_REDUCE_SCATTERなどの計算付集団的操作も同様の解釈ができる（図5.3を参照）．グループ内コミュニケータの場合，これらの2つのグループは同じである． グループ間コミュニケータの場合，これらの2つのグループは異なる．全対全操作の場合，このような各操作は対称な全二重動作が可能なように，2つのフェーズで記述される．

以下の集団操作もグループ間コミュニケータに適用される．

- MPI_BARRIER,
- MPI_BCAST,
- MPI_GATHER, MPI_GATHERV,
- MPI_SCATTER, MPI_SCATTERV,
- MPI_ALLGATHER, MPI_ALLGATHERV,
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW,
- MPI_ALLREDUCE, MPI_REDUCE,
- MPI_REDUCE_SCATTER.

C++言語では，これらの関数の呼び出し形式は`MPI::Comm`クラスとなる．しかし，C++言語の`MPI::Comm`では集団操作が意味を持たないため（グループ間コミュニケータでもグループ内コミュニケータでもないため），関数は全て純粋仮想である．

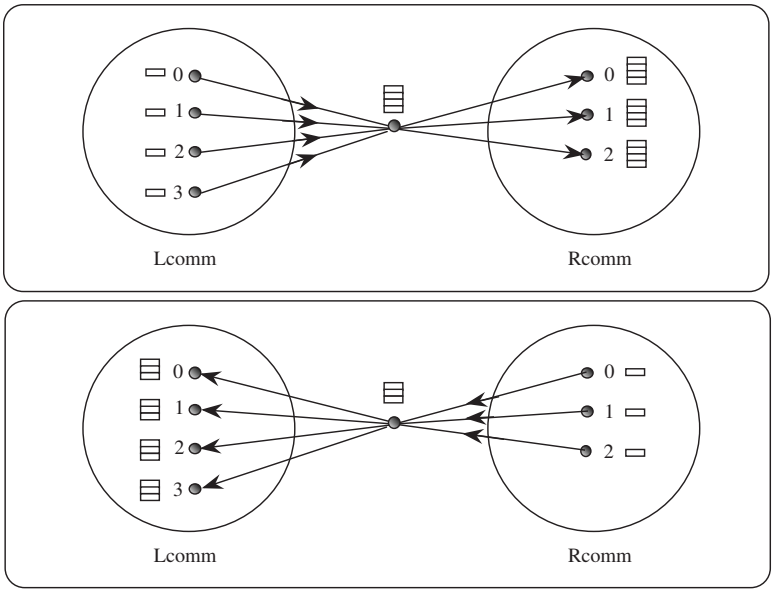

図 5.2: グループ間コミュニケータのオールギャザー． 仮想的な 1 つのプロセスにデータを集める方法を例示しているが，アルゴリズムを示しているわけではない． 2つのフェーズで双方向のオールギャザーが行われる．

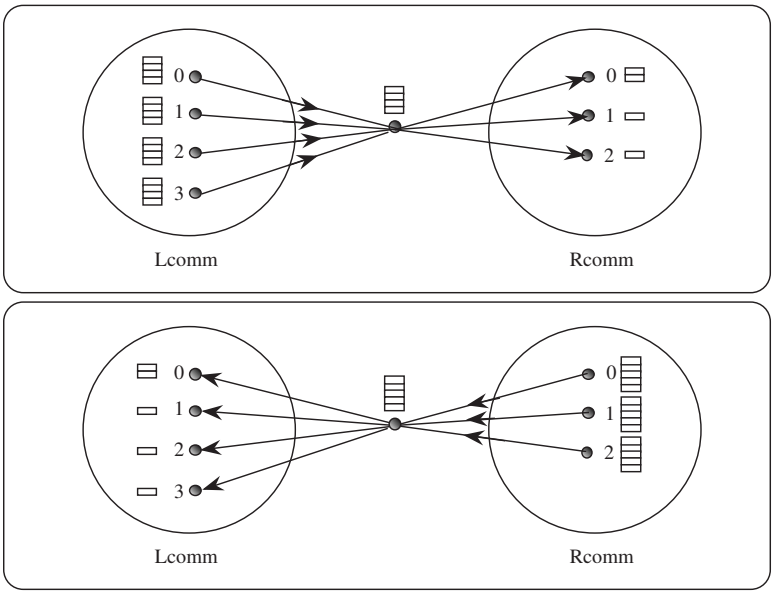

図 5.3: グループ間コミュニケータのリデューススキャッタ． 仮想的な 1 つのプロセスにデータを集める方法を例示しているが，アルゴリズムを示しているわけではない． 2つのフェーズで双方向のリデューススキャッタが行われる．

5.2.3 グループ間コミュニケータ集団操作の仕様

グループ間コミュニケータによって識別される両方のグループの全てのプロセスは集団的ルーチンを呼び出す必要がある．

グループ間コミュニケータの場合は，あるプロセスからそのプロセス自身への通信が行われないため，グループ間コミュニケータの「インプレイス」オプションは グループ間コミュニケータに適用されない．

グループ間コミュニケータ集団的通信の場合，操作が全対1または1対全のカテゴリに含まれるのであれば，送信は単方向となる．送信方向はルート引数の特別な値で示される． この場合，ルートプロセスを含むグループでは，グループ内の全てのプロセスはそのルートの特別な引数を使用してルーチンを呼び出す必要がある．このため，ルートプロセスは特別なルート値MPI_ROOTを使用し，同じグループ内の他の全てのプロセスはルートとしてMPI_PROC_NULLを使用する．他のグループの全てのプロセス（ルートプロセスに対してリモートグループであるグループ）は集団的ルーチンを呼び出してルートのランクを提供する必要がある．操作が全対全のカテゴリに含まれる場合，送信は双方向となる．

根拠 1対1および1対全のカテゴリに含まれる操作は本来単方向で，方向を指定するための明確な方法がある．全対全のカテゴリに含まれる操作は交換の一部として発生することがあり，この場合は一度に双方向で通信することに意味がある． （**根拠の終わり**）

5.3 バリア同期

MPI_BARRIER(comm)

IN	comm	コミュニケータ（ハンドル）

```
int MPI_Barrier(MPI_Comm comm)
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
{void MPI::Comm::Barrier() const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
```

comm がグループ内コミュニケータの場合，MPI_BARRIERは全グループメンバが呼び出すまでブロックする．この呼び出しがプロセスに戻るのは，全てのグループメンバがその呼び出しを開始した後である．

commがグループ間コミュニケータの場合，MPI_BARRIERには 2つのグループが含まれる．呼び出しがグループ間コミュニケータの1つの グループ（グループA）内のプロセスに戻るのは，他のグループ（グループB）の全てのメンバがその呼び出しを開始した後（逆も同様）である．プロセスは同じグループ内の全てのプロセスが呼び出しを開始する前に呼び出しから戻ることがある．

5.4 ブロードキャスト

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（非負の 整数型）
IN	datatype	バッファのデータ型（ハンドル）
IN	root	ブロードキャストルートのランク（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm )
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
{void MPI::Comm::Bcast(void* buffer, int count,
              const MPI::Datatype& datatype, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
```

commがグループ内コミュニケータの場合，MPI_BCASTは，ランクが rootであるプロセスからそのプロセスを含むグループ内の全プロセスへメッセージをブロードキャストする．comm およびrootとして同じ引数を使ってグループ内の全メンバにより呼び出される．戻った時点で，rootのバッファの内容が全プロセスへコピーされている．

一般的に，datatypeでは派生データ型が使用できる．任意のプロセスの count，datatypeの型シグネチャはルートのcount，datatypeの型シグネチャと同じでなけ

ればならない．このことは，各プロセスとルートの間で，送信データの量と受け取るデータの量が等しくなければならないということを意味している．MPI_BCAST および他の全てのデータ移動集団的ルーチンによりこの制約が課される．ただし送信側と受信側とで型マップの違いだけは許容される．

「インプレイス」オプションはここでは無効である，

commがグループ間コミュニケータの場合，呼び出しにはグループ間コミュニケータの全てのプロセスが含まれるが， 1つのグループ（グループA）によりルートプロセスが定義される．他方のグループ（グループB）の全てのプロセスは，引数(root)にグループAのルートのランクである同じ値を渡す．ルートは rootに値MPI_ROOTを渡す． グループAのその他の全てのプロセスはrootに値 MPI_PROC_NULLを渡す．データはルートからグループBの全てのプロセスにブロードキャストされる． グループBのプロセスのバッファ引数はルートのバッファ引数と整合していなければならない．

5.4.1 MPI_BCASTの使用例

ここではグループ内コミュニケータを使用した例を示す．

例 **5.1** プロセス0からグループ内の全てのプロセスに100個のintをブロードキャストする．

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root, comm);
```

ここで取りあげる一部分のコードの多くは，変数（上記のcommなど）に適切な値が代入されているものと仮定している．

5.5 ギャザー

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcount	送信バッファの要素の数（非負の 整数型）
IN	sendtype	送信バッファの要素のデータ型（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型，ルートでのみ意味を持つ）
IN	recvcount	1回の受信の要素数（非負の 整数型，ルートでのみ意味を持つ）
IN	recvtype	受信バッファの要素のデータ型(ルートでのみ意味を持つ)（ハンドル）
IN	root	受信プロセスのランク（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
```

```
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
{void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype, int root) const = 0 （廃止された呼
             び出し形式, 第15.2節を参照）}
```

commがグループ間コミュニケータの場合，各プロセス（ルートプロセスを含む）は，送信バッファの内容をルートプロセスへ送信する．ルートプロセスはメッセージを受信し，ランクの順番に格納する． あたかもグループの中のn個のプロセス（ルートプロセスを含む）が以下のルーチンの呼び出しを実行し，

$$\texttt{MPI\_Send}(\texttt{sendbuf}, \texttt{sendcount}, \texttt{sendtype}, \texttt{root}, ...),$$

さらに次の呼び出しをn回実行した結果と同じである．

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{i} \cdot \texttt{recvcount} \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...),$$

ここで，extent(recvtype)はMPI_Type_get_extent()の呼び出しで得ることができる型の範囲である．

言い換えると，グループ内のプロセスが送信した n個のメッセージをランク順に連結し，あたかも得られたメッセージをMPI_RECV(recvbuf, recvcount·n, recvtype, ...) の呼び出しによって受け取ったかのようにルートが受信する．

ルート以外の全てのプロセスについては受信バッファは無視される．

通常，sendtypeとrecvtypeにはデータ型が使用できる，各プロセスのsendcount，sendtypeの型シグネチャは，ルートのrecvcount，recvtypeの型シグネチャと等しくなければならない．このことは，各プロセスとルートの間で，送信データの量と受け取るデータの量が等しくなければならないということを意味する．ただし送信側と受信側とで型マップの違いは許容される．

rootのプロセスでは関数への全ての引数は意味を持つが，他のプロセスでは引数sendbuf，sendcount，sendtype，root，commのみが意味を持つ．引数rootおよびcommは全てのプロセスで同じ値でなければならない．

個数と型の指定は，ルートプロセス上の同じ位置に複数回書き込まれることがあってはならない．そのような呼び出しは誤りである．

ルートプロセスの引数recvcountは各プロセスから受信する項目数を示しているのであって，受信する項目の総数を示しているのではないことに注意すること．

ルートでsendbufの値としてMPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．このような場合，sendcountとsendtypeは無視され， ギャザーされたベクトルに対してルートの寄与分がすでに受信バッファ内の正しい場所にあると仮定される．

commがグループ間コミュニケータの場合，呼び出しにはグループ間コミュニケータの全てのプロセスが含まれるが，1つのグループ（グループA）によりルートプロセスが定

義される．他方のグループ（グループB）の全てのプロセスは，引数rootにグループAのルートのランクである同じ値を渡す．ルートは rootに値MPI_ROOTを渡す．グループAのその他の全てのプロセスはrootに値MPI_PROC_NULLを渡す．データはグループBの全てのプロセスからルートにギャザーされる． グループBのプロセスの送信バッファ引数はルートの受信バッファ引数と整合していなければならない．

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcount	送信バッファの要素の数（非負の 整数型）
IN	sendtype	送信バッファの要素のデータ型（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型，ルートでのみ意味を持つ）
IN	recvcounts	各プロセスから受信した要素の数を持つ 非負の 整数型の（グループサイズの長さの）配列（ルートでのみ意味を持つ）
IN	displs	（グループサイズの長さの）整数配列． エントリiはプロセスiから送られてくるデータを置く位置の recvbufからの相対変位を指定する（ルートでのみ意味を持つ）.
IN	recvtype	受信バッファの要素のデータ型（ルートでのみ意味を持つ）（ハンドル）
IN	root	受信プロセスのランク（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,
int root, MPI_Comm comm)
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR
{void MPI::Comm::Gatherv(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int displs[],
             const MPI::Datatype& recvtype, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GATHERVはMPI_GATHERの機能を拡張したもので， recvcountsが配列になっており，各プロセスから可変個のデータを受け取れるようになっている．さらに，新しい引数としてdisplsを提供することにより，ルート上のデータの配置に関して自由度が増している．

commがグループ間コミュニケータの場合，結果としては，あたかもルートプロセスを含む各プロセスがメッセージをルートへ送信し，

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

ルートプロセスが受信をn回繰り返した場合と同じである．

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{displs[j]} \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts[j]}, \texttt{recvtype}, \texttt{i}, ...).$$

プロセスj から送られたデータは，オフセットdispls[j] 要素（recvtypeの表現で）から始まるrootプロセスのrecvbufに配置される．

ルート以外の全てのプロセスでは，受信バッファは無視される．

プロセスi の sendcount，sendtypeで示される型シグネチャは，ルートの recvcounts[i]，recvtype で示される型シグネチャと等しくなければならない．このことは，各プロセスとルートとの間でペアごとに，送信データの量が受け取るデータの量と等しくなければならないということを意味する．ただし，例5.6 に示されているように，送信側と受信側とで型マップの違いは許容される．

rootプロセスでは，関数の全ての引数が意味を持つが，それ以外のプロセスでは引数sendbuf，sendcount，sendtype，root，commのみが意味をもつ．引数rootおよびcommは全てのプロセスで値が同一でなければならない．

個数，型，変位の指定により，ルートプロセス上の同じ位置に複数回書き込まれることがあってはならない．そのような呼び出しはエラーとなる．

ルートでsendbufの値としてMPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．このような場合，sendcountとsendtypeは無視され， ギャザーされたベクトルに対してルートの関与分がすでに受信バッファ内の正しい場所にあると仮定される．

commがグループ間コミュニケータの場合，呼び出しにはグループ間コミュニケータの全てのプロセスが含まれるが， 1つのグループ（グループA）によりルートプロセスが定義される．他方のグループ（グループB）の全てのプロセスは，引数rootに グループAのルートのランクである同じ値を渡す．ルートは rootに値MPI_ROOTを渡す． グループAのその他の全てのプロセスはrootに値 MPI_PROC_NULLを渡す．データはグループBの全てのプロセスからルートにギャザーされる．グループBのプロセスの 送信バッファ引数はルートの受信バッファ引数と整合していなければならない．

5.5.1 MPI_GATHER, MPI_GATHERVの使用例

ここではグループ内コミュニケータを使用した例を示す．

例 5.2 グループ内の全てのプロセスからルートへ100個のintをギャザーする． 図5.4を参照すること．

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

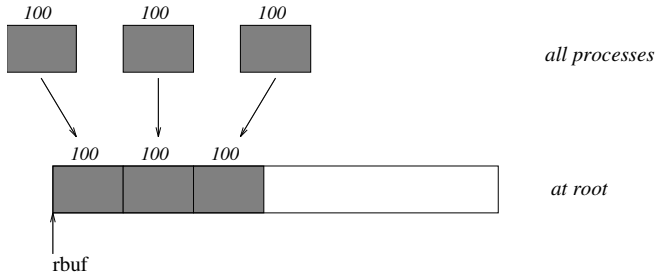

図 5.4: ルートプロセスがグループ内の各プロセスから100 個のintをギャザーする.

例 5.3 前の例の変更 – ルートだけが受信バッファ用のメモリを割り当てる.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank( comm, &myrank);
if ( myrank == root) {
   MPI_Comm_size( comm, &gsize);
   rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

例 5.4 前の例と同じことをしているが，派生データ型を使用している．ギャザーではルートプロセスと各プロセスとの間で型対応が行われているので，その型は gsize*100個の整数の集まりと一致しない.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

例 5.5 各プロセスはルートへ100個のintを送信するが，それぞれを受信側でstride個のint分だけの間隔をおいて配置する．MPI_GATHERVおよびdispls引数を使用してこの効果を得ることができる．$stride \geq 100$と仮定する．図5.5を参照すること.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;

...
```

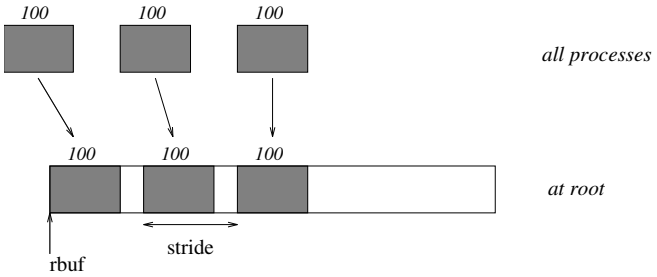

図 5.5: ルートプロセスは，グループ内の各プロセスから100個のintをギャザーし，それぞれをstrideの int個分の間隔をおいて配置する．

```
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                          root, comm);
```

ただし，$stride < 100$ の場合に，このプログラムはエラーは誤りである．

例 5.6 受信側についての例5.5と同じ．ただし，C言語における 100×150のint配列の第0列から100個の整数を送信する． 図5.6を参照すること．

```
MPI_Comm comm;
int gsize,sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
/* Create datatype for 1 column of array
 */
MPI_Type_vector( 100, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                      root, comm);
```

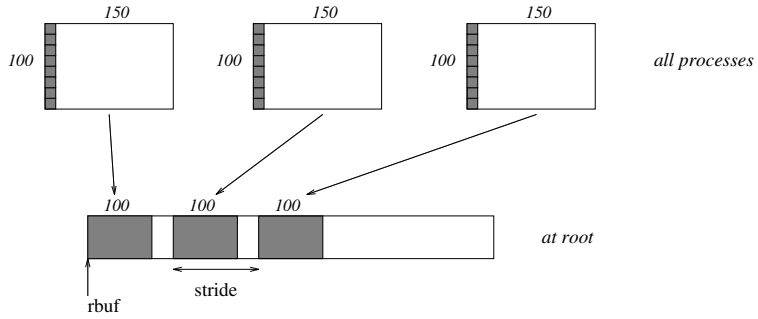

図 5.6: ルートプロセスはC言語における 100×150の配列の第0列をギャザーし，それぞれをstride個のint個分だけの間隔をおいて配置する．

例 5.7 プロセスiはC言語における100 × 150のint配列の第i列から(100-i)個のintを送信する．上記2つの例と同様に，stride間隔でバッファの中へ読み込む．図5.7を参照すること．

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i;     /* note change from previous example */
}
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

ただし，各プロセスから受信するデータの量は異なる．

例 5.8 例5.7と同じ．ただし，送信側では別の方法で行われている．送信側で正しいストライドとなるようなデータ型を生成し，C言語における配列から1列読み込む．第4.1.14節の例4.16で行ったのと同じである．

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
```

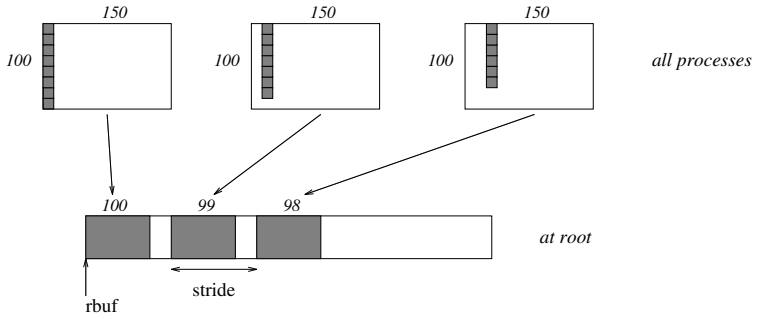

図 5.7: ルートプロセスはC言語における100×150の配列の第i列から100-i個のintをギャザーし，それぞれをstride個のint分の間隔をおいて配置する.

```
    MPI_Datatype stype,type[2];
    int *displs,i,*rcounts;

    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );
    rbuf = (int *)malloc(gsize*stride*sizeof(int));
    displs = (int *)malloc(gsize*sizeof(int));
    rcounts = (int *)malloc(gsize*sizeof(int));
    for (i=0; i<gsize; ++i) {
        displs[i] = i*stride;
        rcounts[i] = 100-i;
    }
    /* Create datatype for one int, with extent of entire row
     */
    disp[0] = 0;       disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;   blocklen[1] = 1;
    MPI_Type_create_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                                                          root, comm);
```

例 5.9 送信側は例5.7と同じ．ただし，受信側では受信したブロックの間のストライドはブロック毎に異なる． 図5.8を参照すること.

```
    MPI_Comm comm;
    int gsize,sendarray[100][150],*sptr;
    int root, *rbuf, *stride, myrank, bufsize;
    MPI_Datatype stype;
    int *displs,i,*rcounts,offset;

    ...

    MPI_Comm_size( comm, &gsize);
    MPI_Comm_rank( comm, &myrank );

    stride = (int *)malloc(gsize*sizeof(int));
```

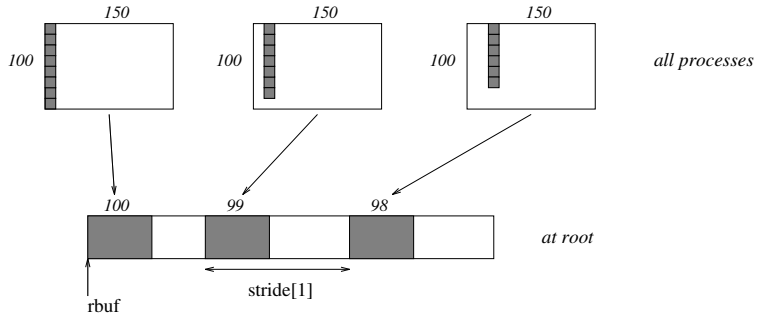

図 5.8: ルートプロセスはC言語における100×150の配列の第i 列から 100-i intをギャザーし，それぞれをstride int[i]個分だけの間隔をおいて配置する（可変ストライド）.

```
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 */

/* set up displs and rcounts vectors first
 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained
 */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Create datatype for the column we are sending
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
                                                    root, comm);
```

例 5.10 プロセスi は，C言語における100 × 150のint配列の第i列からnum個のintを送信する．しかしrootではnumにどのような値が設定されているか分からないので，まずギャザーを行ってこの値を獲得する．データは受信側で連続した領域に配置される.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, myrank, disp[2], blocklen[2];
MPI_Datatype stype,type[2];
int *displs,i,*rcounts,num;

...

MPI_Comm_size( comm, &gsize);
```

```
    MPI_Comm_rank( comm, &myrank );

    /* First, gather nums to root
     */
    rcounts = (int *)malloc(gsize*sizeof(int));
    MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
    /* root now has correct rcounts, using these we set displs[] so
     * that data is placed contiguously (or concatenated) at receive end
     */
    displs = (int *)malloc(gsize*sizeof(int));
    displs[0] = 0;
    for (i=1; i<gsize; ++i) {
        displs[i] = displs[i-1]+rcounts[i-1];
    }
    /* And, create receive buffer
     */
    rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
                                                          *sizeof(int));
    /* Create datatype for one int, with extent of entire row
     */
    disp[0] = 0;       disp[1] = 150*sizeof(int);
    type[0] = MPI_INT; type[1] = MPI_UB;
    blocklen[0] = 1;   blocklen[1] = 1;
    MPI_Type_create_struct( 2, blocklen, disp, type, &stype );
    MPI_Type_commit( &stype );
    sptr = &sendarray[0][myrank];
    MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
                                                        root, comm);
```

5.6 スキャッタ

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	送信バッファのアドレス（選択型，ルートでのみ意味を持つ）
IN	sendcount	各プロセスに送られる要素数(非負の 整数型，ルートでのみ意味を持つ)
IN	sendtype	送信バッファ要素のデータ型（ルートでのみ意味を持つ）（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型）
IN	recvcount	受信バッファ内の要素数（非負の 整数型）
IN	recvtype	受信バッファ要素のデータ型（ハンドル）
IN	root	送信プロセスのランク（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)
```

```
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
{void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const
              MPI::Datatype& sendtype, void* recvbuf, int recvcount,
              const MPI::Datatype& recvtype, int root) const = 0 （廃止された呼
              び出し形式, 第15.2節を参照）}
```

MPI_SCATTER はMPI_GATHERの逆操作である．

commがグループ内コミュニケータの場合，あたかもルートが以下の送信操作

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{i} \cdot \texttt{sendcount} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, ...),$$

をn回繰返し，各プロセスが以下の受信操作を行った結果と同じである．

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

ルートがMPI_Send(sendbuf, sendcount·n, sendtype, ...)によってメッセージを送ることであるとも言い換られる．このメッセージはn個の等しいセグメントに分割されてi番目のセグメントがグループ内のi 番目のプロセスに送られる．各プロセスはこのメッセージを上で述べた通りに受け取る．

全ての非ルートプロセスにおいて送信バッファは無視される．

ルートプロセスのsendcount，sendtypeで示される型シグネチャは，全てのプロセスのrecvcount，recvtypeで示される型シグネチャと等しくなければならない（しかし，型マップは異なっていてもよい）．このことは．各プロセスとルートとの間でペアごとに，送信データの量が受け取るデータの量と等しくなければならないということを意味する．ただし，送信側と受信側とで型マップの違いは許容される．

rootプロセスでは，関数の全ての引数が意味を持つが，それ以外のプロセスでは引数recvbuf，recvcount，recvtype，root，commのみが意味をもつ．引数rootおよびcommは全てのプロセスで値が同一でなければならない．

個数，型の指定により，ルートプロセス上の同じ位置で複数回読み取られることがあってはならない．

> **根拠** 必要ではないが，MPI_GATHERとの対称性を達成するために上述の制約が課されている．対応する制約（多重書き込み制約）は必要である． （**根拠の終わり**）

ルートでrecvbufの値としてMPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．このような場合，recvcountとrecvtypeは無視され，ルートはそれ自体へのデータの「送信」を行わない． スキャッタされたベクトルはn個のセグメントを持っていると仮定される．ここで，nはグループのサイズで，ルートが「それ自体に送信」するべき$root$番目のセグメントは移動しない．

commがグループ間コミュニケータの場合，呼び出しにはグループ間コミュニケータの全てのプロセスが含まれるが， 1つのグループ（グループA）によりルートプロセスが定義される．他方のグループ（グループB）の全てのプロセスは，引数rootにグループAのルートのランクである同じ値を渡す．ルートは rootに値MPI_ROOTを渡す． グループAのその他の全てのプロセスはrootに値 MPI_PROC_NULLを渡す．データはルートか

らグループBの全てのプロセスにスキャッタされる．グループBのプロセスの受信バッファ引数はルートの送信バッファ引数と整合していなければならない．

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	送信バッファのアドレス（選択型，ルートでのみ意味を持つ）
IN	sendcounts	各 ランク[1]に送る要素の数を指定する（グループサイズの長さ）非負の整数配列
IN	displs	（グループサイズの長さの）整数配列．エントリiはプロセスi に送られるデータを取得するsendbufへの相対変位として指定する．
IN	sendtype	送信バッファの要素のデータ型（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型）
IN	recvcount	受信バッファの要素数（非負の 整数型）
IN	recvtype	受信バッファの要素のデータ型（ハンドル）
IN	root	送信側プロセスのランク（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR
{void MPI::Comm::Scatterv(const void* sendbuf, const int sendcounts[],
             const int displs[], const MPI::Datatype& sendtype,
             void* recvbuf, int recvcount, const MPI::Datatype& recvtype,
             int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_SCATTERVはMPI_GATHERVの逆操作である．

MPI_SCATTERVはMPI_SCATTERの機能を拡張したもので， sendcountsが配列になっており，各プロセスに可変個のデータを 送信できるようになっている．さらに，追加の引数としてdisplsを提供することにより，ルート上のデータの取得場所に関して自由度が増している．

commがグループ内コミュニケータの場合，あたかもルートが以下の送信操作を

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{displs[i]} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts[i]}, \texttt{sendtype}, \texttt{i}, ...),$$

n回繰り返し，各プロセスが以下の受信を行った結果と同じである．

$$\texttt{MPI\_Recv}(\texttt{recvbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

ルート以外の全てのプロセスでは，送信バッファは無視される．

[1]訳者註：MPI-2.2の原文は “processor”であるが， “rank”の誤り．MPI-3では訂正されている．

ルートのrecvcounts[i]，recvtypeで示される型シグネチャはプロセスi のsendcount，sendtypeで示される型シグネチャと等しくなければならない（しかし，型マップは異なっていてもよい）．このことは，各プロセスとルートとの間でペアごとの，送信データの量が受け取るデータの量と等しくなければならないということを意味する．ただし，送信側と受信側とで型マップの違いは許容される．

rootプロセスでは，関数の全ての引数が意味を持つが，それ以外のプロセスでは引数recvbuf，recvcount，recvtype，root，commのみが意味をもつ．引数rootおよびcommは全てのプロセスで値が同一でなければならない．

個数，型，変位の指定により，ルートプロセス上の同じ位置で複数回読み取られることがあってはならない．

ルートでrecvbufの値としてMPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．このような場合，recvcountとrecvtypeは無視され，ルートはそれ自体へのデータの「送信」を行わない． スキャッタされたベクトルはn個のセグメントを持っていると仮定される．ここで，nはグループのサイズで，ルートが「それ自体に送信」するべき$root$番目のセグメントは移動しない．

commがグループ間コミュニケータの場合，呼び出しにはグループ間コミュニケータの全てのプロセスが含まれるが，1つのグループ（グループA）によりルートプロセスが定義される．他方のグループ（グループB）の全てのプロセスは，引数rootにグループAのルートのランクである同じ値を渡す．ルートはrootに値MPI_ROOTを渡す．グループAのその他の全てのプロセスは rootに値MPI_PROC_NULLを渡す．データはルートからグループBの全てのプロセスにスキャッタされる． グループBのプロセスの受信バッファ引数はルートの送信バッファ引数と整合していなければならない．

5.6.1 MPI_SCATTER, MPI_SCATTERVの使用例

ここではグループ間コミュニケータを使用した例を示す．

例 5.11 例5.2の逆の例．100個のintをルートプロセスから グループ内の各プロセスにスキャッタする．図5.9.を参照すること．

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

例 5.12 例5.5の逆の例．ルートプロセスは100個のintを他のプロセスにスキャッタするが，intは送信バッファにstride個のintの間隔で置かれている．このような時はMPI_SCATTERVの使用が求められる．$stride \geq 100$を仮定している．図5.10を参照すること．

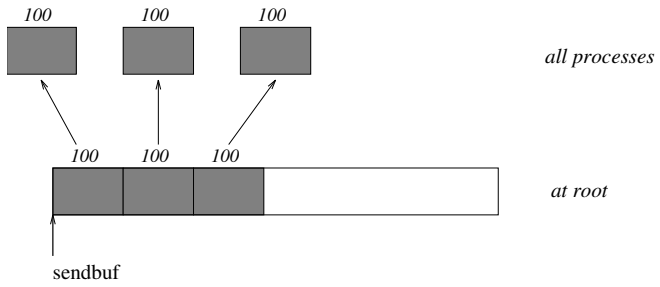

図 5.9: 100個のintをルートプロセスからグループ内の各プロセスにスキャッタする.

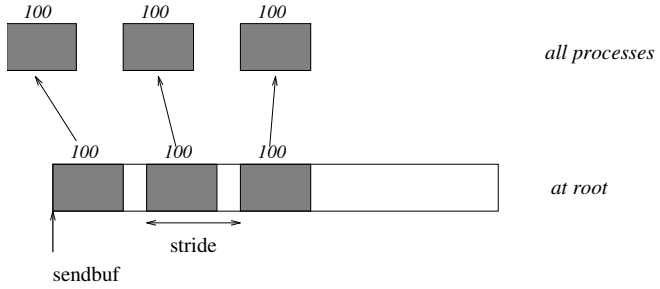

図 5.10: ルートプロセスがstride個のintの間隔の空いた100個のintをスキャッタ送信する.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;

...

MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
                                                        root, comm);
```

例 5.13 例5.9の逆の例. 送信側（ルートプロセス）ではストライドを変化させながらブロック間でデータが散在しており，受信側では100×150 のC言語配列の第i列に受けとる. 図5.11を参照すること.

```
MPI_Comm comm;
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
```

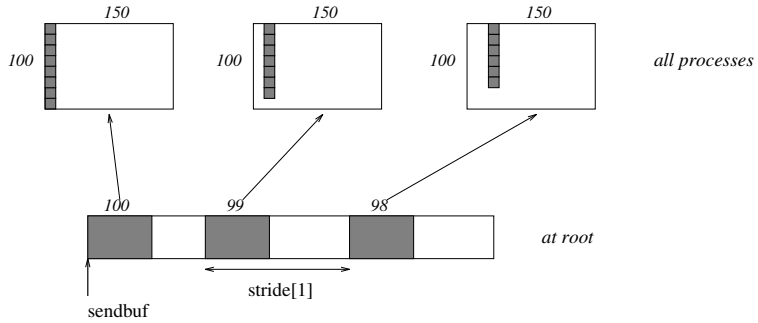

図 5.11: ルートは，100-i個のintブロックを100×150のC言語配列の第i列へスキャッタさせる．送信側では，ブロックがそれぞれstride[i]個の間隔で散在している

```
...
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving
 */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
                                                        root, comm);
```

5.7 全プロセスへのギャザー

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcount	送信バッファ内の要素の数（非負の 整数型）
IN	sendtype	送信バッファの要素のデータ型（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型）
IN	recvcount	任意のプロセスから受信する要素の数（非負の 整数型）
IN	recvtype	受信バッファの要素のデータ型（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
{void MPI::Comm::Allgather(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0 （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_ALLGATHERは，ルートプロセスだけでなく，全てのプロセスが結果を受けとるMPI_GATHERと考えることができる．第j 番目のプロセスから送信されるデータのブロックが全てのプロセスで受信され，バッファrecvbufの第j番目のブロックとして格納される．

プロセスのsendcount，sendtypeで示される型シグネチャはその他の任意のプロセスのrecvcount，recvtypeで示される型シグネチャと等しくなければならない．

commがグループ内コミュニケータの場合，MPI_ALLGATHER(...)の呼び出しは，あたかも全てのプロセスが以下の操作を，

```
MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
                                              recvtype,root,comm)
```

root = 0 , ..., n-1までのn回の呼び出しを実行し結果と同じである．MPI_ALLGATHERの正しい使用規則はMPI_GATHERの対応する規則から容易に見い出せる．

全てのプロセスで引数sendbufに値MPI_IN_PLACEを渡すことにより．グループ内コミュニケータの「インプレイス」オプションが指定される． sendcountとsendtypeは無視される．そして，各プロセスの入力データが入る領域は，そのプロセスがそれ自体の寄与分を受信バッファに受信するための領域であると仮定される．

commがグループ間コミュニケータの場合，一方のグループ（グループA）の各プロセスはsendcount個のデータ項目に寄与し，これらのデータが連結されて結果が他方のグループ（グループB）内の各プロセスに格納される．逆に，グループBの各プロセスの寄与

分の連結がグループAの各プロセスに格納される．グループAの送信バッファ引数がグループBの受信バッファ引数に整合していて，逆も同様になっている必要がある．

ユーザへのアドバイス グループ間通信上で実行されるMPI_ALLGATHERの通信パターンは対称である必要はない．グループAのプロセスによって送信される項目の数（グループAの引数sendcount，sendtype，およびグループB の引数recvcount，recvtypeによって指定）はグループBのプロセスによって送信される項目の数（グループBの引数sendcount，sendtype，グループAの引数recvcount，recvtypeによって指定）と同じでなくてもよい．特に，逆方向の通信の場合，sendcount = 0 を指定することにより，データを一方向にのみ移動させることができる．（**ユーザへのアドバイス終わり**）

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcount	送信バッファ内の要素の数（非負の整数型）
IN	sendtype	送信バッファの要素のデータ型（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型）
IN	recvcounts	各プロセスから受信した要素の数を持つ 非負の整数型の（グループサイズの長さの）配列
IN	displs	（グループサイズの長さの）整数配列．エントリiはプロセスiから送られてくるデータを置く位置のrecvbufからの相対変位を指定する（ルートでのみ意味を持つ）．
IN	recvtype	受信バッファの要素のデータ型（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,
MPI_Comm comm)
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
{void MPI::Comm::Allgatherv(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf,
             const int recvcounts[], const int displs[],
             const MPI::Datatype& recvtype) const = 0 （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_ALLGATHERVは，ルートプロセスだけでなく，全てのプロセスが結果を受けとるMPI_GATHERVと考えることができる．第j 番目のプロセスから送信されるデータのブロックが全てのプロセスで受信され，バッファrecvbufの第j番目のブロックとして格納される．ブロックは全て同じサイズである必要はない．

プロセスjのsendcount，sendtypeで示される型シグネチャはその他の任意のプロセスのrecvcounts[j]，recvtypeで示される型シグネチャと等しくなければならない．

commがグループ内コミュニケータの場合，あたかも全てのプロセスが以下に示す，

```
MPI_GATHERV(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,
                                                    recvtype,root,comm),
```

root = 0 , ...，n-1に対して実行した結果を同じである．MPI_ALLGATHERVの正しい使用規則はMPI_GATHERVの対応する規則から容易に見い出せる．

全てのプロセスで引数sendbufに値MPI_IN_PLACE を渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，sendcountとsendtypeは無視され，各プロセスの入力データが入る領域は，そのプロセスがそれ自体の寄与分を 受信バッファに受信するための領域であると仮定される．

commがグループ間コミュニケータの場合，一方のグループ（グループA）の各プロセスはsendcount個のデータ項目に寄与し，これらのデータが連結されて結果が他方のグループ（グループB）内の各プロセスに格納される．逆に．グループBの各プロセスの寄与分の連結がグループAの各プロセスに格納される． グループAの送信バッファ引数がグループBの受信バッファ引数に整合していて，逆も同様になっている必要がある．

5.7.1 MPI_ALLGATHERの使用例

ここではグループ内コミュニケータを使用した例を示す．

例 5.14 例5.2のオールギャザー版．MPI_ALLGATHERを使って， グループ中の全てのプロセスから各プロセスに100個のintをギャザーする．

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

呼び出しの完了後，全てのプロセスはグループ全体の連結されたデータを保持している．

5.8 全対全スキャッタ／ギャザー

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcount	各プロセスに送信される要素の数（非負の 整数型）
IN	sendtype	送信バッファの要素のデータ型（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型）
IN	recvcount	各プロセスから受信した要素の数（非負の 整数型）
IN	recvtype	受信バッファの要素のデータ型（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
{void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const
             MPI::Datatype& sendtype, void* recvbuf, int recvcount,
             const MPI::Datatype& recvtype) const = 0 （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_ALLTOALLはMPI_ALLGATHER を拡張したもので，各プロセスから各受信側に異なるデータが送信される．プロセスiから送信されたj番目のブロックがプロセスjによって受信され，recvbufのi番目のブロックに格納される．

あるプロセスのsendcount，sendtypeで示される型シグネチャはその他の任意のプロセスのrecvcount，recvtypeで示される型シグネチャと等しくなければならない．このことは，プロセスの各ペアの間で，送信データの量が受け取るデータの量と等しくなければならないということを意味する．ここでも，型マップの違いは許容される．

commがグループ内コミュニケータの場合，あたかも呼び出しで各プロセス（それ自体を含む）への送信を実行し，

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{i} \cdot \texttt{sendcount} \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcount}, \texttt{sendtype}, \texttt{i}, ...),$$

以下の呼び出しで他の全てのプロセスからの受信を実行した場合と同じ結果である．

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{i} \cdot \texttt{recvcount} \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcount}, \texttt{recvtype}, \texttt{i}, ...).$$

全てのプロセスにおいて引数は全て意味を持つ．引数commは全てのプロセスで同じ値でなければならない．

全てのプロセスで引数sendbufに値MPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，sendcountとsendtypeは無視される． 送信されるデータがrecvbufから取得され， 受信したデータによって置換される．送信データおよび受信データは， recvcountおよびrecvtypeで指定したのと同じ型マップを持っていなければならない．

根拠 MPI_ALLTOALLインスタンスの値を大きくすると，送信バッファと 受信バッファの両方の割り当てで使用されるメモリ容量が大きくなりすぎる可能性がある．「インプレイス」オプションではアプリケーションのメモリ使用量を効果的に半減させることができ，送信されるデータがMPI_ALLTOALLの交換後に送信プロセスで使用されない場合（並列高速フーリエ変換など）には有益である． （**根拠の終わり**）

実装者へのアドバイス ユーザはメモリを節約するために「インプレイス」オプションを使用することもできる．そのため，質の高いMPI実装のためにはシステムのバッファリングを最小限に抑えるようにする必要がある． （**実装者へのアドバイス終わり**）

commがグループ間コミュニケータの場合，あたかもグループAの各プロセスがグループBの各プロセスにメッセージを送信したのと同じであり，逆もまた同様である，グループAのプロセスiの j番目の送信バッファがグループBのプロセスjのi番目の受信バッファと整合していて，逆も同様になっている必要がある．

ユーザへのアドバイス グループ間コミュニケータドメインで完全な交換を実行する場合，グループAのプロセスからグループBのプロセスに送信されるデータ項目の数は逆方向に送信される項目の数と同じでなくてもよい．特に，逆方向でsendcount = 0を指定することにより，単方向の通信を行うことができる． （**ユーザへのアドバイス終わり**）

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcounts	各ランク[2] に送信する要素の数を指定する（グループサイズの長さの）非負の整数配列
IN	sdispls	（グループサイズの長さの）整数配列．エントリjはプロセスjに対して送られて来るデータを取得する場所を sendbufからの相対変位として指定する
IN	sendtype	送信バッファの要素のデータ型（ハンドル）
OUT	recvbuf	受信バッファのアドレス（選択型）
IN	recvcounts	各ランク[2]から受信可能な要素の数を指定する（グループサイズの長さの） 非負の整数配列
IN	rdispls	（グループサイズの長さの）整数配列． エントリiはプロセスi から送られてくるデータを置く位置の recvbufからの相対変位を指定する．
IN	recvtype	受信バッファの要素のデータ型（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls,
MPI_Datatype recvtype, MPI_Comm comm)
```

[2]訳者註：MPI-2.2の原文は“processor”であるが， “rank”の誤り．MPI-3では修正済．

```
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
{void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype& sendtype,
             void* recvbuf, const int recvcounts[], const int rdispls[],
             const MPI::Datatype& recvtype) const = 0 （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_ALLTOALLVは，送信されるデータの位置をsdisplsによって指定し，受信側のデータを格納する位置をrdispls によって指定する自由度を付加したMPI_ALLTOALLである．

commがグループ内コミュニケータの場合，プロセスi から送られるデータのj番目のブロックはプロセスjにより受信され，recvbufのi番目のブロックに格納される．これらのブロック全てが同じサイズである必要はない．

プロセスiのsendcounts[j]，sendtypeで示される型シグネチャはプロセスjのrecvcounts[i]，recvtypeで示される型シグネチャと等しくなければならない，このことは，プロセスの各ペアの間で，送信データの量が受け取るデータの量と等しくなければならないということを意味する．ここでも， 送信側と受信側の型マップの違いは許容される．

あたかも以下の呼び出しにより各プロセスから他の各プロセスにメッセージが送信され，

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{sdispls}[\texttt{i}] \cdot \texttt{extent}(\texttt{sendtype}), \texttt{sendcounts}[\texttt{i}], \texttt{sendtype}, \texttt{i}, ...),$$

以下の呼び出しにより他の全てのプロセスからメッセージを受信した結果と同じになる．

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{rdispls}[\texttt{i}] \cdot \texttt{extent}(\texttt{recvtype}), \texttt{recvcounts}[\texttt{i}], \texttt{recvtype}, \texttt{i}, ...).$$

全てのプロセスにおいて，引数は全て意味を持つ．引数commは全てのプロセスで同じ値でなければならない．

全てのプロセスで引数sendbufに値MPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される． この場合， sendcounts，sdispls，sendtypeは無視される．送信されるデータがrecvbufから取得され，受信したデータによって置換される． 送信データおよび受信データは，recvcounts配列およびrecvtypeで指定したのと同じ型マップを持っていなければならなず，rdisplsで指定された受信バッファ内の場所から取得される．

ユーザへのアドバイス 「インプレイス」オプションを指定する（全てのプロセスに対して行う必要がある）ことは，コミュニケータのグループ内の2つのプロセス間で同じ量および型のデータが送受信されることを意味する．異なるプロセスのペアで異なる両のデータを交換することができる．プロセスiのrecvcounts[j] およびrecvtypeがプロセスjのrecvcounts[i]およびrecvtypeと一致するようにする必要がある．この対称の交換は， 送信されるデータがMPI_ALLTOALLVの交換後に送信プロ

セスで使用されないような場合に有益でありうる．（ユーザへのアドバイス終わり）

commがグループ間コミュニケータの場合，あたかもグループAの各プロセスがグループBの各プロセスにメッセージを送信した結果と同じであり，逆もまた同様である．グループAのプロセスiのj番目の送信バッファがグループBのプロセスjのi番目の受信バッファと整合していて，逆も同様になっている必要がある．

根拠 MPI_ALLTOALLとMPI_ALLTOALLVの定義は，1対1通信をn回独立に指定して行なうという柔軟性を示しているが，2点の例外がある．全てのメッセージが同じ型を持ち，スキャッタ（あるいはギャザー）されるデータの格納先（元）が逐次的である点である．（根拠の終わり）

実装者へのアドバイス 集団的通信を1対1通信の観点から見れば，メッセージは 送信側から受信側に直接送信する形をとるが，通信経路に木構造を持ったものを実装してもよい．効率が良くなるのであれば，分岐（スキャッタの場合）または合流（ギャザーの場合）するノードを中継してメッセージを転送することができる．（実装者へのアドバイス終わり）

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
IN	sendcounts	各ランク[3]に送信する要素の数を指定する（グループサイズの長さ） 非負の整数の配列
IN	sdispls	（グループサイズの長さの）整数配列．エントリiはプロセスjに対して送られて来るデータを取得する場所をsendbufからの相対変位としてバイトでに指定する（整数配列）
IN	sendtypes	（グループサイズの長さの）データ型の配列． エントリjはプロセスjに送信するデータ型を指定する（ハンドルの配列）
OUT	recvbuf	受信バッファのアドレス（選択型）
IN	recvcounts	各ランク[3]から受信可能な要素の数を指定する（グループサイズの長さの） 非負の整数配列
IN	rdispls	（グループサイズの長さの）整数配列．エントリiはプロセスi から送られてくるデータを置く位置のrecvbufからの相対変位を指定する（整数配列）
IN	recvtypes	（グループサイズの長さの）データ型の配列． エントリiはプロセスiから受信したデータ型を指定する（ハンドルの配列）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[], int rdispls[],
MPI_Datatype recvtypes[], MPI_Comm comm)
```

[3]訳者註：MPI-2.2の原文は“processor”であるが， “rank”の誤り．MPI-3では修正済.

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, IERROR
{void MPI::Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
             const int sdispls[], const MPI::Datatype sendtypes[], void*
             recvbuf, const int recvcounts[], const int rdispls[], const
             MPI::Datatype recvtypes[]) const = 0 （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_ALLTOALLWは完全な交換のもっとも一般的な形式である．MPI_TYPE_CREATE_STRUCTと同様，もっとも一般的な型であるコンストラクタMPI_ALLTOALLWでは，カウント，変位， データ型を個別に指定できる．また，最大の自由度を得るため，送信バッファと受信バッファ内のブロックの変位がバイト単位で指定される．

commがグループ内コミュニケータの場合，プロセスiから送信される j番目のブロックはプロセスjによって受信され． recvbufのi番目のブロックに格納される．これらのブロックは全てが同じサイズである必要はない．

プロセスiのsendcounts[j]，sendtypes[j]で示される型シグネチャはプロセスjのrecvcounts[i]， recvtypes[i]で示される型シグネチャと等しくなければならない．このことは，プロセスの各ペアの間で，送信データの量が受け取るデータの量と等しくなければならないということを意味する．ここでも，送信側と受信側とで型マップの違いは許容される．

あたかも以下の呼び出しにより各プロセスから他の各プロセスにメッセージが送信され，

$$\texttt{MPI\_Send}(\texttt{sendbuf} + \texttt{sdispls}[\texttt{i}], \texttt{sendcounts}[\texttt{i}], \texttt{sendtypes}[\texttt{i}], \texttt{i}, ...),$$

以下の呼び出しにより他の全てのプロセスからメッセージを受信した結果と同じになる．

$$\texttt{MPI\_Recv}(\texttt{recvbuf} + \texttt{rdispls}[\texttt{i}], \texttt{recvcounts}[\texttt{i}], \texttt{recvtypes}[\texttt{i}], \texttt{i}, ...).$$

全てのプロセスにおいて，引数は全て意味を持つ．引数commには全てのプロセスで同じコミュニケータを指定する必要がある．

MPI_ALLTOALLVと同様，全てのプロセスで引数sendbuf にMPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，sendcounts， sdispls，sendtypeは無視される．送信されるデータがrecvbufから取得され，受信したデータによって置換される． 送信データおよび受信データは，recvcounts配列および recvtypes配列で指定したのと同じ型マップを持っていなければならなず，rdisplsで指定された受信バッファ内の場所から取得される．

commがグループ間コミュニケータの場合，あたかもグループAの各プロセスがグループBの各プロセスにメッセージを送信した結果と同じであり，逆もまた同様である．グループAのプロセスiのj番目の送信バッファがグループBのプロセスjのi番目の受信バッファと整合していて，逆も同様になっている必要がある．

根拠 MPI_ALLTOALLW関数は入力引数を慎重に選択することにより，いくつかのMPI関数を汎用化している．例えば，1つを除いた全てのプロセスを`sendcounts[i] = 0`とすることにより`MPI_SCATTERW`関数を実現する． （**根拠の終わり**）

5.9 大域的なリデュース操作

このセクションの関数は，グループの全メンバに対し大域的なリデュース演算（sum, max，論理など）を実行する．リデュース演算は，定義済み演算の一覧，またはユーザ定義演算のいずれかとすることができる．大域的なリデュース関数にはいくつかバリエーションがある．1つのグループの1つのメンバにリデュースの結果を返すリデュース，1つのグループの全メンバにこの結果を返すオールリデュース，および2つのスキャン（並列プリフィックス） 操作の3つである．さらに，リデューススキャッタ操作はリデュース操作の機能とスキャッタ操作の機能を組み合わせたものである．

5.9.1 リデュース

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	送信バッファのアドレス（選択型）
OUT	recvbuf	受信バッファのアドレス（選択型，ルートでのみ意味を持つ）
IN	count	送信バッファ内の要素の数（非負の 整数型）
IN	datatype	送信バッファの要素のデータ型（ハンドル）
IN	op	リデュース演算（ハンドル）
IN	root	ルートプロセスのランク（整数型）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
{void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op, int root)
             const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
```

commがグループ内コミュニケータの場合，MPI_REDUCE は演算opを使用してグループ内の各プロセスの入力バッファで与えられる要素に対してリダクションを行ない，その結果をランクrootのプロセスの出力バッファに返す．入力バッファは，引数sendbuf，count，datatypeで定義される．出力バッファは，引数recvbuf，count，datatypeで定義される．両者とも同じ型の要素を同じ個数だけ持つ．このルーチンは，count，datatype，op，root， commについて同じ引数を使用して全てのグループメンバから呼び出される．したがって全てのプロセスは同じ型の要素を持つ同じ長さの入力バッファと出

力バッファを用意する．各プロセスは，1つの要素，または要素の列を与えることができ，要素の列の場合には，列の各エントリの要素ごとにリダクションを行う．例えば，MPI_MAXという演算で，送信バッファに浮動小数点数の2つの要素（count = 2, datatype= MPI_FLOAT）が入っている場合，recvbuf(1) = global max(sendbuf(1)) およびrecvbuf(2) = global max(sendbuf(2)) となる．

第5.9.2節に，MPIで用意している定義済み演算の一覧を掲載した．さらに， 演算を適用できるデータ型もまとめた．そのほかにユーザは複数のデータ型についての操作で基本型または派生型のいずれでもオーバーロードできるユーザ用の演算も定義できる．これについては第5.9.5節で詳述する．

演算opは常に結合的であると仮定する．定義済み演算は全て可換であると仮定する．ユーザは結合的であるが可換ではないと仮定された演算を定義することもできる．リデュースの「標準的」評価順序はグループ内のプロセスのランクによって決まる．しかし，実装では結合性，または結合性と可換性を利用して，評価順序を変更することもできる．浮動小数点数加算など厳密には結合的または可換でない演算についてはこの演算によってリデュースの結果が変わることもある．

実装者へのアドバイス MPI_REDUCEを実装する場合には，同じ順序で現れる同じ引数で関数を適用する際には，必ず同じ結果が得られるようにすることが強く求められる．このためランク[4]の物理的配置を利用する最適化はできない場合があることに注意すること． （**実装者へのアドバイス終わり**）

ユーザへのアドバイス アプリケーションが，浮動小数点操作の非結合的を無視できない場合や，特別なリデュースの順序が必要で結合的なものとして扱えないユーザ定義の演算を使用する（第5.9.5節を参照）．このようなアプリケーションでは評価順序を明確に規定する必要がある．例えば，左から右（または右から左）の評価順序が厳密に求められる操作の場合，1つのプロセスで全てのオペランドをギャザー（MPI_GATHERなどにより）し，リデュース操作を望ましい順序で適用（MPI_REDUCE_LOCALなどにより）することによってこれを実現できる．必要に応じて，他のプロセスへの結果のブロードキャストまたはスキャッタ（MPI_BCASTなどにより）を行う． （**ユーザへのアドバイス終わり**）

MPI_REDUCEのdatatype引数はopと互換性がなければならない．定義済み演算子は第5.9.2節と第5.9.4節で挙げられているMPIの型でしか機能しない．また，定義済みの演算子用のdatatypeとopはすべてのプロセスで同じでなければならない．

ユーザは各プロセスでMPI_REDUCEに異なるユーザ定義演算子を与えることができてしまう． この場合，MPIではどのオペランドにどの演算が使用されるかは定義されない．ユーザ定義演算子は一般的な派生データ型でも機能する． この時，リデュース演算が適用される各引数はそのようなデータ型によって記述される1つの要素であり，それは複数の基本値を含んでいてもよい．これについてはさらに，第5.9.5節で詳述する．

[4]訳者註：MPI-2.2の原文は“processor”であるが， “rank”の誤り．MPI-3では修正済.

ユーザへのアドバイス ユーザはMPI_REDUCEの実装方法についての仮定は控える必要がある．最も安全な方法は，各プロセスによってMPI_REDUCEに同じ関数が渡されるようにすることである． （**ユーザへのアドバイス終わり**）

「送信」バッファでのデータ型のオーバーラップは許可されている．「受信」バッファでのデータ型のオーバーラップを行うとエラーとなり，予測不能の結果が生じることがある．

ルートで引数sendbufに値MPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，ルートで受信バッファから入力データが取得され，ここでは入力データが出力データによって置換される．

commがグループ間コミュニケータの場合，呼び出しにはグループ間コミュニケータの全てのプロセスが含まれるが， 1つのグループ（グループA）によりルートプロセスが定義される．他方のグループ（グループB）の全てのプロセスは，引数rootに グループAのルートのランクである同じ値を渡す．ルートはrootに値MPI_ROOTを渡す．グループAのその他の全てのプロセスはrootに値MPI_PROC_NULLを渡す． グループBでは送信バッファ引数のみが意味を持ち，ルートでは受信バッファ引数のみが意味を持つ．

5.9.2 定義済みリデュース演算

以下の定義済み演算はMPI_REDUCEのために用意されていて，関数MPI_ALLREDUCE, MPI_REDUCE_SCATTER，MPI_SCAN，MPI_EXSCANと関連している．これらの操作は，opに次の値を入れることで呼び出される．

名前	意味
MPI_MAX	最大値
MPI_MIN	最小値
MPI_SUM	和
MPI_PROD	積
MPI_LAND	論理積
MPI_BAND	ビット演算の積
MPI_LOR	論理和
MPI_BOR	ビット演算の和
MPI_LXOR	排他的論理和
MPI_BXOR	ビット演算の排他的論理和
MPI_MAXLOC	最大値と位置
MPI_MINLOC	最小値と位置

2つの演算，MPI_MINLOCおよびMPI_MAXLOCについては第5.9.4節で別に説明する．他の定義済み演算については，引数opとdatatypeの組み合わせのうち，使用できるものを以下にリストアップする．最初に，次のようにMPI基本データ型のグループを定義する．

C言語整数型：	MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG（同義）, MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR, MPI_INT8_T, MPI_INT16_T, MPI_INT32_T, MPI_INT64_T, MPI_UINT8_T, MPI_UINT16_T, MPI_UINT32_T, MPI_UINT64_T
Fortran言語整数型：	MPI_INTEGER, MPI_AINT, MPI_OFFSET, および，以下から返されたハンドル： MPI_TYPE_CREATE_F90_INTEGER, および，利用できる場合：MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_INTEGER8, MPI_INTEGER16
実数型：	MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION MPI_LONG_DOUBLE および，以下から返されたハンドル MPI_TYPE_CREATE_F90_REAL, および，利用できる場合：MPI_REAL2, MPI_REAL4, MPI_REAL8, MPI_REAL16
論理型：	MPI_LOGICAL, MPI_C_BOOL,
複素数型：	MPI_COMPLEX,MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, MPI_CXX_LONG_DOUBLE_COMPLEX, および，以下から返されたハンドル MPI_TYPE_CREATE_F90_COMPLEX, および，利用できる場合： MPI_DOUBLE_COMPLEX, MPI_COMPLEX4, MPI_COMPLEX8, MPI_COMPLEX16, MPI_COMPLEX32
バイト型：	MPI_BYTE

各演算に対し，有効なデータ型を以下に示す.

Op	使用できるデータ型
MPI_MAX, MPI_MIN	C言語整数型，Fortran言語整数型，実数型
MPI_SUM, MPI_PROD	C言語整数型，Fortran言語整数型，実数型，複素数型
MPI_LAND, MPI_LOR, MPI_LXOR	C言語整数型，論理型
MPI_BAND, MPI_BOR, MPI_BXOR	C言語整数型，Fortran言語整数型，バイト型

以下に，グループ内コミュニケータを使用した例を示す.

例 5.15 あるグループ中のプロセスに分配される2つのベクトルの内積を計算し，ノード0に結果を返すルーチン.

```
SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
REAL a(m), b(m)       ! local slice of array
REAL c                ! result (at node zero)
REAL sum
INTEGER m, comm, i, ierr

! local sum
sum = 0.0
DO i = 1, m
   sum = sum + a(i)*b(i)
END DO

! global sum
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
RETURN
```

例 5.16 あるグループの中のプロセスに分配されるベクトルと配列の積を計算し，ノード0に結果を返すルーチン.

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
```

```
! return result at node zero (and garbage at the other nodes)
RETURN
```

5.9.3 符号付き文字とリデュース

データ型MPI_SIGNED_CHARおよびMPI_UNSIGNED_CHARはリデュース演算で使用できる．MPI_CHAR，MPI_WCHAR，MPI_CHARACTER（印字可能可能文字を表す）はリデュース演算で使用できない．異機種環境では，MPI_CHAR，MPI_WCHAR，MPI_CHARACTERは印字可能文字が保持されるように変換されるが，MPI_SIGNED_CHARとMPI_UNSIGNED_CHARは整数値が保持されるように変換される．

> **ユーザへのアドバイス** データ型MPI_CHAR，MPI_WCHARおよびMPI_CHARACTERは文字用で，文字コードの異なるマシン間で送信する場合に，整数値ではなく，印字可能可能文字が保持されるように変換される．データ型MPI_SIGNED_CHARおよびMPI_UNSIGNED_CHARは，整数値を保持する必要がある場合にC言語で使用される． （**ユーザへのアドバイス終わり**）

5.9.4 MINLOCとMAXLOC

演算子MPI_MINLOCは，大域的な最小値とこの最小値に位置するインデックスを計算する．MPI_MAXLOCもこれと同様に，大域的な最大値とそのインデックスを計算する場合に使用する．これの応用の1つとして，大域的な最小値（最大値）とこの値を持つプロセスのランクを計算する例をとりあげる．

MPI_MAXLOCを定義する演算は次のとおりである．

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

ここで

$$w = \max(u, v)$$

さらに

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI_MINLOCも同様に定義する．

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$

ここで

$$w = \min(u, v)$$

さらに

$$k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

この演算は両者とも結合的であり可換である． $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$のペアの列に対してMPI_MAXLOCのリデュースを適用した場合，戻り値は(u, r)となることに注意されたい．ここで， $u = \max_i u_i$ とrはこの列の中の最初の大域的な最大値のインデックスである．したがって，各プロセスが値とそのグループ内におけるランクを与えるとすると，op = MPI_MAXLOCとしたリデュース操作は最大値とその値を持つ最初のプロセスのランクを返す．同様に，MPI_MINLOCを使用して最小値とそのインデックスを返すことができる．より一般的には，MPI_MINLOCは**辞書順での最小値**を求めるということであり，要素が各ペアの最初の成分に応じて順序付けられ2番目の成分により決定される．

リデュース演算は，値とインデックスのペアからなる引数に対する演算と定義される．Fortran言語とC言語とでは，型はこのペアを記述するように与えられる．Fortran言語では，引数に複数の型が混在していた場合は問題となる．Fortran言語のこの問題は，インデックスを値と同じ型に強制的に変換し，値と同じ型を持つペアから成る型をMPIが提供することで回避されている．C言語では， MPIの提供する型は別々の型のペアであり，インデックスは`int`型である．

リデュース操作の中でMPI_MINLOCおよびMPI_MAXLOCを使用するためには，ペア（値とインデックス）を表すdatatype引数を与えなければならない．MPIはこのような定義済みデータ型を9個用意している．MPI_MAXLOCとMPI_MINLOCの操作を次のデータ型とともに使用することができる．

Fortran言語:

名前	説明
MPI_2REAL	`REAL`型のペア
MPI_2DOUBLE_PRECISION	`DOUBLE PRECISION` 型変数のペア
MPI_2INTEGER	`INTEGER`型のペア

C言語:

名前	説明
MPI_FLOAT_INT	`float` と `int`
MPI_DOUBLE_INT	`double` と `int`
MPI_LONG_INT	`long` と `int`
MPI_2INT	`int` のペア
MPI_SHORT_INT	`short` と `int`

MPI_LONG_DOUBLE_INT	long double と int

データ型MPI_2REALは，あたかも次のように定義された結果と同様である（第4.1節を参照）.

```
MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)
```

MPI_2INTEGER，MPI_2DOUBLE_PRECISION， MPI_2INTについても同様のことがいえる．データ型MPI_FLOAT_INTは，あたかも次の命令群によって定義された結果と同じである.

```
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_CREATE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

MPI_LONG_INTおよびMPI_DOUBLE_INT についても同様のことがいえる.
以下に，グループ内コミュニケータを使用した例を示す.

例 5.17 各プロセスはC言語における30個のdoubleの配列を持つ．30個のそれぞれの場所について，最大値を持つその値とプロセスのランクを計算する.

```
...
/* each process has an array of 30 double: ain[30]
 */
double ain[30], aout[30];
int  ind[30];
struct {
    double val;
    int   rank;
} in[30], out[30];
int i, myrank, root;

MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}
MPI_Reduce( in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm );
/* At this point, the answer resides on process root
 */
if (myrank == root) {
    /* read ranks out
     */
    for (i=0; i<30; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
    }
}
```

例 5.18 Fortran言語での同様の例

```
    ...
    ! each process has an array of 30 double: ain(30)

    DOUBLE PRECISION ain(30), aout(30)
    INTEGER ind(30)
    DOUBLE PRECISION in(2,30), out(2,30)
    INTEGER i, myrank, root, ierr

    CALL MPI_COMM_RANK(comm, myrank, ierr)
    DO I=1, 30
        in(1,i) = ain(i)
        in(2,i) = myrank    ! myrank is coerced to a double
    END DO

    CALL MPI_REDUCE( in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,
                                                                  comm, ierr )
    ! At this point, the answer resides on process root

    IF (myrank .EQ. root) THEN
        ! read ranks out
        DO I= 1, 30
            aout(i) = out(1,i)
            ind(i) = out(2,i)  ! rank is coerced back to an integer
        END DO
    END IF
```

例 **5.19** 各プロセスは値の空でない配列を持つ．大域的な最小値，その値を保持するプロセスのランク，このプロセス上でのインデックスを見つける．

```
#define  LEN   1000

float val[LEN];        /* local array of values */
int count;             /* local number of values */
int myrank, minrank, minindex;
float minval;

struct {
    float value;
    int   index;
} in, out;

    /* local minloc */
in.value = val[0];
in.index = 0;
for (i=1; i < count; i++)
    if (in.value > val[i]) {
        in.value = val[i];
        in.index = i;
    }

    /* global minloc */
MPI_Comm_rank(comm, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce( &in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
    /* At this point, the answer resides on process root
     */
if (myrank == root) {
```

```
    /* read answer out
     */
    minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}
```

根拠 ここで与えたMPI_MINLOCおよびMPI_MAXLOCの定義には，これら2つの演算を特別に扱う必要がないという利点がある．他のリデュース演算のように処理すれば良い．プログラマはMPI_MAXLOCおよびMPI_MINLOCの独自の定義を必要に応じて与えることができる．欠点としては，値とインデックスを最初にインタリーブしなければならないという点と，更にFortran言語の場合，インデックスと値を同じ型に強制変換しなければならないという点が挙げられる． （**根拠の終わり**）

5.9.5 ユーザ定義リデュース演算

MPI_OP_CREATE(function, commute, op)

IN	function	ユーザ定義関数（関数）
IN	commute	可換の場合はtrue それ以外はfalse
OUT	op	演算（ハンドル）

```
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR
{void MPI::Op::Init(MPI::User_function* function, bool commute) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_OP_CREATEは，ユーザ定義のリデュース演算をopハンドルにバインドし，その後，これをMPI_REDUCE，MPI_ALLREDUCE，MPI_REDUCE_SCATTER，MPI_SCAN，MPI_EXSCANで使用することができる．ユーザ定義演算は結合的であると仮定される．commute = trueであれば，この演算は可換かつ結合的でなければならない．commute = falseであれば，操作順序は固定され，プロセスランクの昇順でプロセス0 から始まると定義される．操作の結合性から評価の実行順序を変更することが可能である．また，commute = trueの場合は，さらに可換性を利用して評価の実行順序を変更することが可能である．

引数functionはユーザ定義関数であり， invec，inoutvec，len，datatypeの4つの引数を持たなければならない．

この関数のISO C言語プロトタイプは次のとおりである．

```
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
MPI_Datatype *datatype);
```

ユーザ定義関数のFortran言語での宣言は次のとおりである．

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
```

```
    <type> INVEC(LEN), INOUTVEC(LEN)
    INTEGER LEN, TYPE
```

ユーザ定義関数のC++言語での宣言は次のとおりである．

```
{typedef void MPI::User_function(const void* invec, void *inoutvec, int
             len, const Datatype& datatype); （廃止された呼び出し形式．第15.2節を参
             照）}
```

datatype引数は，MPI_REDUCEの呼び出しに渡されるデータ型のハンドルである．ユーザのリデュース関数を作成する場合には，次のことが成立するようにしなければならない．u[0], ... , u[len-1]をこの関数を呼び出すときの引数invec．len，datatypeで記述される通信バッファの中のlen個の要素とする．v[0], ... , v[len-1] をこの関数が呼び出されるときの引数inoutvec，len，datatypeで記述される通信バッファの中のlen個の要素とする．w[0], ... ,w[len-1]をこの関数から戻るときの引数inoutvec，len， datatypeで記述される通信バッファの中のlen個の要素とする．i=0 , ... , len-1についてw[i] = u[i]∘v[i]である．ここで， ∘は関数が計算するリデュース演算である．

invec，inoutvecをfunctionがリダクションするlen個の要素の配列と考えることもできる．リダクションの結果をinoutvecに上書きする，これが名前の由来である．この関数の呼び出しにより．len個の各要素についてリデュース演算の評価が要素毎に行われる．つまり，関数は$i = 0, ..., count - 1$についてinoutvec[i]に値invec[i] ∘ inoutvec[i]を返す．ここで，∘ はこの関数が演算する結合操作である．

> **根拠** len引数を使用することで，MPI_REDUCEが入力バッファ中の要素ごとに関数を呼び出さずに済むようになる．さらに言えば，システム側で入力データに対する関数の適用方法を選べるということである．C言語では，Fortran言語との互換性から参照渡しとしている．
>
> 既知の大域的なハンドルに対し，datatype引数の値を内部で調べることで，1つのユーザ定義関数をオーバロードし，複数の異なるデータ型に対し使用することが可能である． （**根拠の終わり**）

一般データ型をユーザ関数に渡すことができる．ただし，連続していないデータ型を使用すると，効率が低下するおそれがある．

ユーザ定義関数内部でMPI通信関数を呼び出すことはできないが，エラーが発生した場合に関数の内部でMPI_ABORTを呼び出すことができる．

> **ユーザへのアドバイス** オーバロードされるユーザ定義リデュース関数のライブラリを定義するとする．datatype引数は，オペランド型に応じて各呼び出しの正しい実行パスを選択するのに利用される．ユーザ定義リデュース関数は渡されるdatatype引数を解読することはできず，またデータ型ハンドルとそれが表すデータ型との対応関係を識別することもできない．この対応関係は，データ型を生成した時点で確定している．ライブラリを使用する前に，ライブラリの初期化プリアンブルを実行しなければならない．このプリアンブルコードによって，ライブラリが

使用するデータ型を定義し，これらのデータ型のハンドルをユーザコードとライブラリコードで共有するスタティックな大域変数に格納する．

MPI_REDUCEのFortran言語バージョンは，Fortran言語の呼び出し規約を使用してユーザ定義リデュース関数を呼び出し，Fortran言語のデータ型引数を渡す．C言語バージョンではC言語の呼び出し規約でデータ型ハンドルのC言語の表現を使用する．両方の言語を同時に使用することを予定しているユーザの場合には，それぞれに応じたリデュース関数の定義を行わなければならない． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 「インプレイス」オプションを使用しない，MPI_REDUCEの単純なミスによる不効率な実装を以下に示す．

```
MPI_Comm_size(comm, &groupsize);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1,...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root
 */
if (rank == root) {
    MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
}
if (rank == groupsize-1) {
    MPI_Send(sendbuf, count, datatype, root, ...);
}
if (rank == root) {
    MPI_Wait(&req, &status);
}
```

リデュース演算はプロセス0からプロセス groupsize-1まで順番に進行する．この順序は，User_reduce() 関数によって定義される演算子が 可換でない場合を考慮して順序を選択する．結合性を利用し，さらにツリーを用いて対数オーダーでリダクションを行うことにより効率のよい実装を行うことができる．MPI_OP_CREATEへのcommute引数が真の場合， 結合性を利用できる．さらに，サイズ len < count の単位で要素を転送，リデュースすることにより必要な一時バッファの大きさをリデュースし，通信を計算によりパイプライン化することができる．

定義済みリデュース操作はユーザ定義操作のライブラリとして実装できる．しかし，MPI_REDUCEでこれらの関数を特別な場合として処理すれば性能が向上する可能性がある． （**実装者へのアドバイス終わり**）

MPI_OP_FREE(op)

INOUT	op	演算（ハンドル）

```
int MPI_op_free( MPI_Op *op)
MPI_OP_FREE( OP, IERROR)
    INTEGER OP, IERROR
{void MPI::Op::Free() （廃止された呼び出し形式, 第15.2節を参照）}
```

解放するユーザ定義リダクション演算をマークし，op をMPI_OP_NULL に設定する．

ユーザ定義リデュースの例

ここでユーザ定義リデュース演算の例を取りあげる．ここではグループ内コミュニケータを使用した例を示す．

例 5.20 C言語で複素数型配列の積を計算する．

```
typedef struct {
    double real,imag;
} Complex;

/* the user-defined function
 */
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr )
{
    int i;
    Complex  c;

    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real -
                    inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                    inout->imag*in->real;
        *inout = c;
        in++; inout++;
    }
}

/* and, to call it...
 */
...

    /* each process has an array of 100 Complexes
     */
    Complex a[100], answer[100];
    MPI_Op myOp;
    MPI_Datatype ctype;

    /* explain to MPI how type Complex is defined
     */
    MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
    MPI_Type_commit( &ctype );
    /* create the complex-product user-op
     */
    MPI_Op_create( myProd, 1, &myOp );
```

```
MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );

/* At this point, the answer, which consists of 100 Complexes,
 * resides on process root
 */
```

5.9.6 オールリデュース

MPIでは，演算結果がグループの全てのプロセスへ返されるリデュース操作のバリエーションが用意されている．MPIでは，これらの操作に関わる同じグループ内の全てのプロセスが同一の結果を受け取ることが要求されている．

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
OUT	recvbuf	受信バッファの先頭アドレス（選択型）
IN	count	送信バッファ内の要素の数（非負の 整数型）
IN	datatype	送信バッファの要素のデータ型（ハンドル）
IN	op	演算（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
{void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const = 0
  （廃止された呼び出し形式, 第15.2節を参照）}
```

commがグループ内コミュニケータの場合，MPI_ALLREDUCEの動作は結果がグループの全メンバの受信バッファに現れる点を除き，MPI_REDUCE と同じである．

> **実装者へのアドバイス** オールリデュース操作は，リデュースの後にブロードキャストが続くものとして実装することができる．しかし，直接実装したほうが効率がよいと思われる． （**実装者へのアドバイス終わり**）

全てのプロセスの引数sendbufに値MPI_IN_PLACEを渡すことにより，グループ間コミュニケータの「インプレイス」オプションが指定される．この場合，各プロセスで受信バッファから入力データが取得され，ここでは入力データが出力データによって置換される．

commがグループ間コミュニケータの場合，グループAのプロセスによって提供されるデータのリデュースの結果がグループBの各プロセスに格納され，同様に逆のことも行われる．両方のグループで，同じ型仕様を指定するcountおよびdatatype引数を使用する必要がある．

ここではグループ内コミュニケータを使用した例を示す．

例 5.21 ベクトルと，プロセスのグループに分散した配列の積を計算し，全ノードに結果を返すルーチン（例5.16も参照）

```
SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n)    ! local slice of array
REAL c(n)            ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr

! local sum
DO j= 1, n
  sum(j) = 0.0
  DO i = 1, m
    sum(j) = sum(j) + a(i)*b(i,j)
  END DO
END DO

! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

! return result at all nodes
RETURN
```

5.9.7 プロセスローカルなリデュース

このセクションで説明する関数は，標準のMPI操作では対応が難しい特別なリデュースパターンを実装しようとするライブラリの実装者にとって重要である．

以下の関数では，リデュース演算子がローカル引数に適用される．

MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op)

IN	inbuf	入力バッファ（選択型）
INOUT	inoutbuf	入力バッファと出力バッファの組み合わせ（選択型）
IN	count	inbufおよびinoutbufバッファ内の要素の数（非負の整数型）
IN	datatype	inbufおよびinoutbufバッファの要素のデータ型（ハンドル）
IN	op	演算（ハンドル）

```
int MPI_Reduce_local(void* inbuf, void* inoutbuf, int count,
MPI_Datatype datatype, MPI_Op op)
MPI_REDUCE_LOCAL(INBUF, INOUBUF, COUNT, DATATYPE, OP, IERROR)
    <type> INBUF(*), INOUTBUF(*)
    INTEGER COUNT, DATATYPE, OP, IERROR
{void MPI::Op::Reduce_local(const void* inbuf, void* inoutbuf, int count,
             const MPI::Datatype& datatype) const （廃止された呼び出し形式,
             第15.2節を参照）}
```

この関数は，第5.9.5節のユーザ定義演算で説明したように，opで指定される演算をinbufおよびinoutbufの要素に適用し，結果をinoutbufに要素ごとに格納する．inbufと

inoutbuf（入力と結果）の両方が，countで指定されたのと同じ要素数を持ち，datatypeで指定されたのと同じデータ型を持つ．MPI_IN_PLACE オプションは利用できない．

リデュース演算はその可換性の問い合わせを行うことができる．

MPI_OP_COMMUTATIVE(op, commute)

IN	op	演算（ハンドル）
OUT	commute	opが可換の場合はtrue，それ以外の場合はfalse（論理型）

```
int MPI_Op_commutative(MPI_Op op, int *commute)
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
    LOGICAL COMMUTE
    INTEGER OP, IERROR
{bool MPI::Op::Is_commutative() const （廃止された呼び出し形式, 第15.2節を参照）}
```

5.10 リデューススキャッタ

MPIでは，リデュース操作のバリエーションが用意されている．そこでは，戻る際に演算結果がグループ内の全プロセスにスキャッタされる．全てのプロセスに等しいサイズのブロックをスキャッタする操作もあれば，プロセスごとに異なるサイズのブロックをスキャッタする操作もある．

5.10.1 MPI_REDUCE_SCATTER_BLOCK

MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
OUT	recvbuf	受信バッファの先頭アドレス（選択型）
IN	recvcount	ブロックごとの要素数（非負の整数型）
IN	datatype	送信バッファおよび受信バッファの要素のデータ型（ハンドル）
IN	op	演算（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Reduce_scatter_block(void* sendbuf, void* recvbuf, int recvcount,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
{void MPI::Comm::Reduce_scatter_block(const void* sendbuf, void* recvbuf,
             int recvcount, const MPI::Datatype& datatype,
             const MPI::Op& op) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
```

commがグループ内コミュニケータの場合，MPI_REDUCE_SCATTER_BLOCKはまず，演算opを使用して，sendbuf，count，datatypeで定義される送信バッファで

count = n*recvcount個の要素（n はcommのグループ内のプロセスの数）のベクトルに対して大域的な要素ごとのリデュースを行う．ルーチンは，recvcount，datatype，op，commの同じ引数を使用して，グループの全メンバによって呼び出される．生成されるベクトルは，グループの各プロセスにスキャッタされるrecvcount個の要素の連続するn個のブロックとして扱われる．i番目のブロックがプロセスi に送信され，recvbuf，recvcount．datatypeで定義される受信バッファに格納される．

> **実装者へのアドバイス** MPI_REDUCE_SCATTER_BLOCKルーチンは機能的に，countがrecvcount*nと等しいMPI_REDUCEギャザー操作を実行した後に sendcountとrecvcountが等しいMPI_SCATTER を実行するのと同等である．ただし，直接実装した方が高速になる場合がある．（**実装者へのアドバイス終わり**）

全てのプロセスで引数sendbufに値MPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，入力データは受信バッファから取得される．

commがグループ間コミュニケータの場合，一方のグループ（グループA）のプロセスによって提供されるデータのリデュースの結果が他方のグループ（グループ B）のプロセス間でスキャッタされ，同様に逆のことが行われる．各グループ内で，全てのプロセスがrecvcount引数に同じ値を渡し，送信バッファに格納されたcount = n*recvcount個の要素（nはグループのサイズ）の入力ベクトルを渡す．要素の数countは2つのグループで同じでなければならない．他方のグループから得られるベクトルはグループのプロセス間でrecvcount個の要素のブロックでスキャッタされる．

> **根拠** 一方のグループの送信バッファの長さが他方のグループのローカルrecvcount引数によって決定できるようにするため，最後の制約が必要となる．これがないと，要素のリデュースの数を割り出すために通信が必要となる．（**根拠の終わり**）

5.10.2 MPI_REDUCE_SCATTER

MPI_REDUCE_SCATTERは，ブロックのスキャッタのサイズを可変とできるようにMPI_REDUCE_SCATTER_BLOCKの機能を拡張したものである．ブロックサイズは，i番目のブロックにrecvcounts[i] 個の要素が入るようにrecvcounts配列によって決まる．

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
OUT	recvbuf	受信バッファの先頭アドレス（選択型）
IN	recvcounts	（グループサイズの長さの）非負の整数配列
IN	datatype	送信バッファおよび受信バッファの要素のデータ型（ハンドル）
IN	op	演算（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
{void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
             int recvcounts[], const MPI::Datatype& datatype,
             const MPI::Op& op) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
```

commがグループ内コミュニケータの場合，MPI_REDUCE_SCATTERはまず，演算opを使用して，sendbuf，count，datatypeで定義される送信バッファで count $= \sum_{i=0}^{n-1}$ recvcounts[i]個の要素（n はcommのグループ内のプロセスの数）のベクトルに対して大域的な要素ごとのリデュースを行う．ルーチンは，recvcounts，datatype，op， commの同じ引数を使用して，グループの全メンバによって呼び出される．生成されるベクトルは，i番目のブロックの要素数がrecvcounts[i]であるn個の連続するブロックとして扱われる．ブロックは，グループのプロセスにスキャッタされる．i番目のブロックがプロセスi に送信され，recvbuf，recvcounts[i]， datatypeで定義される受信バッファに格納される．

> **実装者へのアドバイス** MPI_REDUCE_SCATTERルーチンは機能的に，count がrecvcounts[i]の合計と等しいMPI_REDUCEリデュース操作を実行した後にsendcountとrecvcountが等しいMPI_SCATTERVを実行するのと同等である．ただし，直接実装した方が高速になる場合がある． （**実装者へのアドバイス終わり**）

引数sendbufに値MPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，入力データは受信バッファから取得される．recvcounts[i]==0のプロセスには受信バッファが割り当てられないことがあるため，全てのプロセスで「インプレイス」オプションを指定する必要はない．

commがグループ間コミュニケータの場合，一方のグループ（グループA）のプロセスによって提供されるデータのリデュースの結果が他方のグループ（グループB）のプロセス間でスキャッタされ，同様に逆のことが行われる．各グループ内で，全てのプロセスが同じrecvcounts引数を渡し，送信バッファに格納されたcount $= \sum_{i=0}^{n-1}$ recvcounts[i]個の要素（n はグループのサイズ）の入力ベクトルを渡す．他方のグループから得られるベクトルはグループのプロセス間でrecvcounts[i]個の要素のブロックでスキャッタされる．要素の数countは2つのグループで同じでなければならない．

> **根拠** 送信バッファの長さがローカルrecvcountsエントリの合計によって決定できるようにするために最後の制約が必要となる．これがないと，リデュースされる要素数を割り出すために通信が必要となる． （**根拠の終わり**）

5.11 スキャン

5.11.1 包括的スキャン

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
OUT	recvbuf	受信バッファの先頭アドレス（選択型）
IN	count	入力バッファ内の要素の数（非負の 整数型）
IN	datatype	入力バッファの要素のデータ型（ハンドル）
IN	op	演算（ハンドル）
IN	comm	コミュニケータ（ハンドル）

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
{void MPI::Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const （廃止
             された呼び出し形式, 第15.2節を参照）}
```

commがグループ内コミュニケータの場合，MPI_SCANはグループ中に分散したデータに対しプリフィックスリデュースを実行する場合に使用する．この演算は，ランクi のプロセスの受信バッファの中に，ランク0,...,iまでのプロセスの送信バッファの中の値のリデュースを返す．サポートされている演算の種類や意味，および送信，受信バッファに対する制約はMPI_REDUCEの場合と同様である．

引数sendbufにMPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，入力データは受信バッファから取得され，出力データによって置換される．

この演算はグループ間コミュニケータに対しては無効である．

5.11.2 排他的スキャン

MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	sendbuf	送信バッファの先頭アドレス（選択型）
OUT	recvbuf	受信バッファの先頭アドレス（選択型）
IN	count	入力バッファ内の要素の数（非負の整数型）
IN	datatype	入力バッファの要素のデータ型（ハンドル）
IN	op	演算（ハンドル）
IN	comm	グループ内コミュニケータ(ハンドル)

```
int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
{void MPI::Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count,
             const MPI::Datatype& datatype, const MPI::Op& op) const （廃止
             された呼び出し形式, 第15.2節を参照）}
```

commがグループ内コミュニケータの場合，MPI_EXSCANはグループ中に分散したデータのプリフィックスリデュースを実行する場合に使用する．ランク0のプロセスのrecvbufの値は未定義で，recvbufはプロセス0では意味を持たない．ランク1のプロセスのrecvbufの値はランク0のプロセスのsendbufの値として定義されている．ランク$i > 1$のプロセスの場合，この演算は，ランクiのプロセスの受信バッファの中に，ランク$0, \ldots, i-1$（端の数を含む）までのプロセスの送信バッファの中の値のリデュースを返す．サポートされている演算の種類や意味，および送信，受信バッファに対する制約はMPI_REDUCEの場合と同様である．

引数sendbufにMPI_IN_PLACEを渡すことにより，グループ内コミュニケータの「インプレイス」オプションが指定される．この場合，入力データは受信バッファから取得され，出力データによって置換される．ランク0の受信バッファはこの演算によって変更されない．

この操作はグループ間コミュニケータに対しては無効である．

> **根拠** 排他的スキャンは包括的スキャンよりも一般的である．包括的スキャン操作は，排他的スキャンを使用し，これにローカル寄与分をローカルに組み合わせることにより，実現することができる．MPI_MAXのような反転できない操作の場合，包括的スキャンにより排他的スキャンの計算をすることはできない． （**根拠の終わり**）

5.11.3 MPI_SCANの使用例

ここではグループ内コミュニケータを使用した例を示す．

例 5.22 この例では，ユーザ定義操作を使用して**部分スキャン**を行う．部分スキャンは入力として値の集合および論理値の集合をとり，その論理値によりスキャンの様々なセグメントを区分けする．例えば，

値	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
論理	0	0	1	1	1	0	0	1
結果	v_1	$v_1 + v_2$	v_3	$v_3 + v_4$	$v_3 + v_4 + v_5$	v_6	$v_6 + v_7$	v_8

この結果をもたらす演算子は次のとおりである．

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ j \end{pmatrix},$$

ここで，

$$w = \begin{cases} u + v & \text{if } i = j \\ v & \text{if } i \neq j \end{cases} .$$

これが非可換演算子であることに注意する．これを実装するC言語のコードは次のとおりである．

```
typedef struct {
    double val;
    int log;
} SegScanPair;

/* the user-defined function
 */
void segScan( SegScanPair *in, SegScanPair *inout, int *len,
                                                    MPI_Datatype *dptr )
{
    int i;
    SegScanPair c;

    for (i=0; i< *len; ++i) {
        if ( in->log == inout->log )
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
```

ユーザ定義関数のinout引数は演算子の右辺に対応する．この演算子を使用する場合には，次のように非可換であることを指定するのを忘れないようにする必要がある．

```
    int i,base;
    SegScanPair  a, answer;
    MPI_Op       myOp;
    MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
    MPI_Aint     disp[2];
    int          blocklen[2] = { 1, 1};
    MPI_Datatype sspair;

    /* explain to MPI how type SegScanPair is defined
     */
    MPI_Get_address( a, disp);
    MPI_Get_address( a.log, disp+1);
    base = disp[0];
    for (i=0; i<2; ++i) disp[i] -= base;
    MPI_Type_create_struct( 2, blocklen, disp, type, &sspair );
    MPI_Type_commit( &sspair );
    /* create the segmented-scan user-op
     */
    MPI_Op_create( segScan, 0, &myOp );
    ...
    MPI_Scan( &a, &answer, 1, sspair, myOp, comm );
```

5.12 正当性

正しくかつ可搬なプログラムは，集団的通信を同期的に行うか否かにかかわらず，デッドロックが発生しないように集団的通信を呼び出さなければならない．次の例は，グループ内コミュニケータでの集団的ルーチンの危険な使用例である．

例 5.23 以下の例は誤りである．

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;
    case 1:
        MPI_Bcast(buf2, count, type, 1, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

commのグループが{0,1}であると仮定する．2つのプロセスは逆順で2つのブロードキャスト操作を実行する．操作が同期していると，デッドロックが発生する．

集団操作は通信を行うグループの全てのメンバで同じ順序で実行されなければならない．

例 5.24 以下の例は誤りである．

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
    case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
}
```

comm0のグループが{0,1}，comm1のグループが{1, 2}， comm2のグループが{2,0}であると仮定する．ブロードキャストが同期操作であれば，巡回依存性がある．つまり，comm2のブロードキャストはcomm0のブロードキャスト後でしか完了せず，comm0のブロードキャストはcomm1のブロードキャストの後でしか完了せず， comm1のブロードキャストはcomm2のブロードキャストの後でしか完了しない．このように，このコードではデッドロックが発生することになる．

集団操作は．巡回依存性が発生しない順序で実行しなければならない．

例 5.25 以下の例は誤りである．

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

プロセス0は，ブロードキャストを実行し，その後ブロッキング送信操作を行う．プロセス1はまず送信と一致するブロッキング受信を実行し，その後プロセス0のブロードキャストと一致するブロードキャスト呼び出しを実行する．このプログラムでもデッドロックが発生する**可能性がある**．プロセス0でのブロードキャスト呼び出しは，プロセス1が一致するブロードキャスト呼び出しを実行するまでブロックする場合があるので，送信は実行されない．プロセス 1は明らかに受信でブロックされるので，この場合ブロードキャストを実行することはない．

集団操作と1対1操作の実行の相対的な順序はこのようにすべきである．それは集団操作と1対1通信とが同期しても，デッドロックを引き起こさないようにするためである．

例 5.26 安全でない非決定性プログラム．

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

3つのプロセスは全てブロードキャストに参加している．プロセス0はブロードキャスト後にメッセージをプロセス1に送信し，プロセス2はブロードキャストの前にメッセージをプロセス1に送信する．プロセス1は引数にワイルドカードで送信元を指定して，ブロードキャストの前後に受信する．

このプログラムには送受信の一致方法の違いで2通りの実行の仕方が考えられる．これを図5.12に示す．ここで， 2番目の実行は，ブロードキャスト後の送信が他のノードにおいてブロードキャスト前に受信されるという，特殊な結果になる．この例は，特定の同期効果を持つ集団的通信関数に依存すべきではないという事実を証明している．1番目の実行が起こる時のみ（ブロードキャストが同期をとる時のみ）正常に動作するプログラムは誤りである．

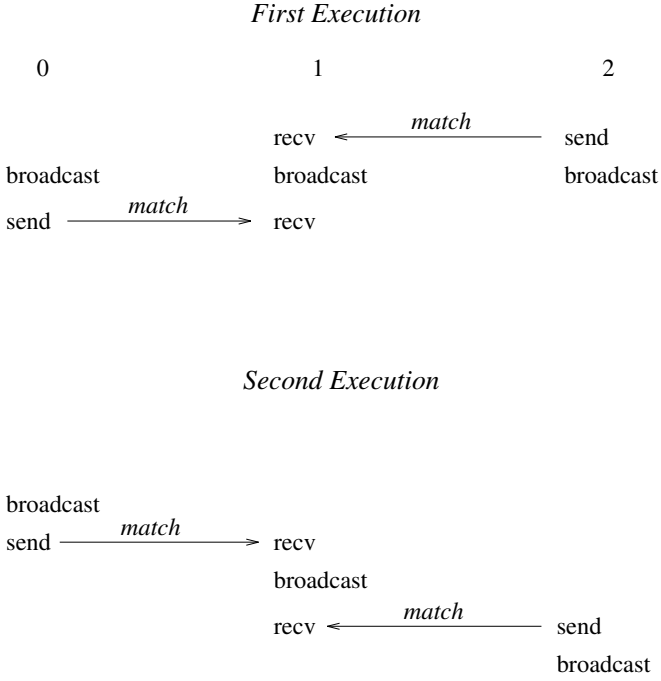

図 5.12: 競合状態が発生すると送信と受信の間に非決定性による一致が生じる．プログラムを決定性にするためにブロードキャストの同期に依存することはできない．

最後に，マルチスレッド実装では，1つのプロセスで同時に実行される複数の集団的通信呼び出しが発生する可能性がある．このような状況ではユーザ側の責任で同じプロセスで2つの異なる集団的通信呼び出しが同時に同じコミュニケータを使用しないようにする必要がある．

実装者へのアドバイス 1対1MPI通信を使用してブロードキャストを実装していると仮定する．この時は，以下2つの規則に従わなければならない．

1. 全ての受信でその送信元を明示する（ワイルドカードを使用しない）．
2. 各プロセスは後の集団呼び出しに関連するメッセージを送信する前に，1つの集団呼び出しに関連する全てのメッセージを送信する．

これで，1対1メッセージの順序が保存されているので，後続のブロードキャストに属するメッセージと混同することはありえない．1対1通信のメッセージと集団的通信のメッセージを混同しないようにするのは実装者の責任である．これを達成するための1つの方法は，コミュニケータを生成するたびに集団的通信用の「隠されたコミュニケータ」を生成することである．また，例えば隠されたタグまたはコンテクストビットを使用してコミュニケータが1対1通信用か集団的通信用かを指定することにより，同様の効果をより安価に実現できる． （**実装者へのアドバイス終わり**）

第6章

グループ，コンテクスト，コミュニケータ，キャッシング

6.1 はじめに

この章では，並列ライブラリの開発を支援するMPIの諸機能を紹介する．並列ライブラリは主要アルゴリズムの並列化における複雑さを緩和するために必要とされる．並列ライブラリの利用により，並列化の過程における一貫した正当性を確保でき，MPI自体で実現できる以上の「高いレベル」の可搬性を実現できる．同様に，ライブラリを利用することで，プログラマは（行列演算などの）主要アルゴリズムを実装するにあたりデータ構造，データ配置，演算手順をそのたびに定義する作業を繰り返さずに済む．良い並列ライブラリには（システムや問題の規模あるいは浮動小数点数の型に応じてデータ配置や戦略の異なる）複数のバリエーションがあるので，これもまたユーザから隠す必要がある．

本章で説明する機能を使用し，ライブラリを記述する際の詳しい情報については文献[42]および文献[3]を参照すること．

6.1.1 ライブラリをサポートするのに必要な機能

堅牢な並列ライブラリを作成するため次のような機能が必要となる．

- ライブラリ外の通信によって干渉される事のない通信を保証する安全な通信空間，
- ライブラリと関与していない（おそらく無関連のコードを実行している）プロセスとの不必要な同期を避けるための集団操作のスコープ，
- ライブラリ独自のデータ構造およびアルゴリズムに適した用語での通信の記述を可能にするプロセス命名の抽象化，
- 通信プロセス群に対する，新たな集団操作などの新しいユーザ定義属性による「修飾」．このメカニズムによってユーザやライブラリ作成者はメッセージ通信の記法を拡張することができる．

また，通信コンテクスト，通信プロセスのグループを簡潔に表し，プロセス命名の抽象化を扱い，修飾を格納するため統一されたメカニズムまたはオブジェクトが必要である．

6.1.2 MPIのライブラリサポート

堅牢なライブラリをサポートするためにMPI が提供する概念には次のものがある．

- 通信のコンテクスト，
- プロセスのグループ，
- 仮想トポロジー，
- 属性キャッシング，
- コミュニケータ．

コミュニケータ（文献[19, 40, 45]を参照）は上記の概念を全て含み，MPIにおける全ての通信操作の適切なスコープを定める．コミュニケータは， 単一のプロセスグループ内の操作を行うグループ内コミュニケータと，2つのプロセスグループ間で1対1通信を行うグループ間コミュニケータの2種類に分けられる．

キャッシング． コミュニケータ（下記参照）は新しい属性をコミュニケータに付与するための「キャッシング」メカニズムを提供する．この新しい属性はMPIに組み込まれた属性と同等に扱われる．この機能は，上級ユーザがコミュニケータをさらに修飾する場合や，MPI自身がコミュニケータの機能を実装するために使用する．例えば，第7章で説明する仮想トポロジー関数はこの方法でサポートされる．

グループ． グループは，順序付されたプロセスの一群であり，各プロセスはそれぞれランクを与えられる．プロセス間通信における低レベルの名前はグループによって定義される（ランクが送受信に使用される）．このように，グループは1対1通信におけるプロセス名のスコープを定義する．さらに，グループは集団操作のスコープも定義する．MPIにおいてグループはコミュニケータと別個に扱われる場合があるが，通信操作にはコミュニケータしか使用できない．

グループ内コミュニケータ． MPIのメッセージ通信においてもっとも一般的に使用される 手段はグループ内コミュニケータによるものである．グループ内コミュニケータはグループのインスタンス，1対1通信および集団的通信のコンテクスト，仮想トポロジーおよびそのほかの属性を含む．これらの機能は次のように働く．

- MPIのコンテクストは，メッセージ通信が独立した安全な「空間」で実行されることを可能とするものである．コンテクストは，メッセージを区別する付加タグに類似している．メッセージはシステムによって区別される．異なるライブラリ（また

は異なるライブラリ呼び出し）で別々の通信 コンテクストを使用することにより，ライブラリ実行の内部通信を外部通信から隔離する．これによって，「他の」コミュニケータ上に通信中であってもライブラリ呼び出しが可能となり，ライブラリコードの実行の前後で同期を取る必要がない．また，処理中の1対1通信は単一のコミュニケータ内の集団的通信を妨害しないことが保証されている．

- **グループ**は，1つのコミュニケータの通信（上記参照）への参加者を定義する．
- **仮想トポロジー**は，単一のグループ内のランクとトポロジーからあるいはトポロジーへの特別なマッピングを定義する．この機能を実現するためのコミュニケータ用の特別なコンストラクタを 第7章に定義する．本章で説明しているグループ内コミュニケータはトポロジーを持たない．
- **属性**は，ユーザやライブラリが後の参照のために，コミュニケータに追加したローカル情報を定義する．

ユーザへのアドバイス 通信ライブラリの多くの実装では， 並列プログラムを起動したときに利用可能な全てのプロセスを含む唯一の定義済み通信空間が用意され，これらのプロセスには連続したランクが割り当てられている．1対1通信への参加者はそのランクで識別され，（ブロードキャストなどの）集団的通信は常に全てのプロセスに関与する．MPIでは，定義済みコミュニケータMPI_COMM_WORLDを使用することでこの実装にしたがうことができる． **この実装に満足しているユーザは，コミュニケータ引数が必要な場所にMPI_COMM_WORLDを設定すればよく，本章の残り部分は飛ばしても構わない．** （ユーザへのアドバイス終わり）

グループ間コミュニケータ これまでの説明では**グループ内通信**について述べてきた．MPIはさらに2つの重ならない**グループ間通信**もサポートする．いくつかの並列モジュールで構成されるアプリケーションを作成する場合，あるモジュールが他のモジュールと当該モジュール内のローカルランクを使用して通信を行えるようにすると便利である．これは特に，クライアントまたはサーバのいずれかが並列動作するクライアント／サーバパラダイムで重宝するものである．グループ間通信によって，全てのプロセスが初期化時に割り当てられていない動的モデルにMPIを拡張するためのメカニズムが提供される．このような場合には，「空間」をまたぐ通信が必要になる．グループ間通信は，**グループ間コミュニケータ**と呼ぶオブジェクトによりサポートされる．これらのオブジェクトは2つのグループをこの両グループで共有される通信コンテクストと結合する．グループ間コミュニケータでは，これらの機能は次のように作用する．

- **コンテクスト**は，2つのグループの間でメッセージ通信の独立した安全な「空間」を持つことを可能とする．ローカルグループにおける送信は常にリモートグループにおける受信であり，またその逆も同様である．メッセージはシステムによって区別される．異なるライブラリ（または異なるライブラリ呼び出し）で別々の通信コンテクストを使用することにより，ライブラリ実行の内部通信を外部通信から隔離

する．これによって，「他の」コミュニケータ上で通信中であってもライブラリ呼び出しが可能となり，ライブラリコードの実行の前後で同期を取る必要がない．

- ローカルグループおよびリモートグループは，グループ間コミュニケータの受信者と送信先を指定する．
- 仮想トポロジーはグループ間コミュニケータに関しては定義されない．
- 前述のように，属性キャッシュは，ユーザまたはライブラリが後の参照のためにコミュニケータに追加したローカル情報を定義する．

MPIは，グループ間コミュニケータを生成し操作するためのメカニズムを提供する．これは，グループ内コミュニケータと同様に1対1通信および集団的通信に使用される．グループ間通信を必要としないユーザはこの拡張機能を無視してもよい．重なり合うグループ間でグループ間通信を必要とするユーザは，当該機能をMPI上に実装しなければならない．

6.2 基本概念

この節では，前節で紹介した概念に対して形式的な定義を与える．

6.2.1 グループ

グループとは，順序付けされたプロセス識別子（以下プロセスと呼ぶ）の集合のことである．プロセスは実装依存オブジェクトである．グループ内の各プロセスは整数**ランク**が付加されている．ランクは連続で，0から始まる．グループは不可視な**グループオブジェクト**なので，プロセスからプロセスへ直接引き渡す事はできない．コミュニケータにおいてグループは通信「空間」への参加者を記述しまた参加者の順序付けのために使用される（そこで，参加者には通信「空間」の中で固有の名前が与えられることになる）．

MPI_GROUP_EMPTYは，メンバを持たない特別なグループとして定義されている．定義済み定数MPI_GROUP_NULLは，無効なグループハンドルの値として使用される．

ユーザへのアドバイス MPI_GROUP_EMPTYは空のグループへの有効なハンドルであり，無効なハンドルであるMPI_GROUP_NULLと混同してはならない．前者はグループ操作の引数として使用されるが，後者はグループを解放するときに返されるもので，引数として使用できない． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス グループは仮想-実プロセスアドレス変換テーブルで表すことができる．各コミュニケータオブジェクト（下記参照）はこのようなテーブルを指すポインタを持つ．

MPIの単純な実装では，グループをテーブル形式に列挙する．しかし多数のプロセスがある場合，スケーラビリティとメモリ利用度を高めるために高度なデータ構造

が意味を持つ．MPIではこのような実装が可能である． （実装者へのアドバイス終わり）

6.2.2 コンテクスト

コンテクストとは，通信空間の分割を行う（次に定義する）コミュニケータの特性である．あるコンテクストで送信されるメッセージは他のコンテクストでは受信できない．さらに，集団操作が可能な場合では集団操作は 処理中の1対1通信操作と無関係に行える．コンテクストは明示的なMPIオブジェクトではなく，コミュニケータの実現の一部である（以下参照）．

実装者へのアドバイス 同じプロセス内の異なるコミュニケータは異なるコンテクストを持つ．コンテクストは，コミュニケータを1対1通信およびMPIで定義する集団的通信に対し安全なものとするために必要なシステム管理タグである．ここで安全という意味は，1つのコミュニケータ内の集団的通信および1対1通信が干渉しない，かつ，異なるコミュニケータ上の通信が互いに干渉しないことである．

コンテクストの1つの実装方法としては，送信時メッセージへ付加し，合致するメッセージを受信する補助タグを使用する方法がある．各グループ内コミュニケータは，2つのタグの値（1対1通信に1つ，集団的通信に1つ）を持つ．コミュニケータ生成関数は新しいグループ固有のコンテクストに関し，各プロセス間で合意を得るために集団的通信を使用する．

同様に，グループ間通信（必ず1対1通信である）では，コミュニケータに2つのコンテクストタグが格納される．1つはグループAが送信にグループBが受信に使用し，他の1つはグループBが送信にグループAが受信に使用するものである．

コンテクストは明示されたオブジェクトではないので，他の実装も可能である．（実装者へのアドバイス終わり）

6.2.3 グループ内コミュニケータ

グループ内コミュニケータはグループとコンテクストの概念からなる．実装固有の最適化およびアプリケーショントポロジー（次の第7章で定義する）をサポートするためにコミュニケータは追加情報を「キャッシュ」することができる（第6.7節を参照）．MPIの通信操作は1対1通信および集団操作のスコープと「通信空間」を決定するためにコミュニケータを参照する．

各コミュニケータは有効な参加者からなるグループを持ち，このグループにはローカルプロセスを含んでいる．メッセージの送信元と送信先はそのグループにおけるプロセスのランクにより指定される．

集団的通信の場合，グループ内コミュニケータは集団操作に加わるプロセスの集合を（必要であれば，その順序も）指定する．したがって，コミュニケータは通信の「空間的」スコープを制限し，ランクによって機種非依存なプロセス指定方法を提供する．

グループ内コミュニケータは，不可視な**グループ内コミュニケータオブジェクト**によって表され，したがってプロセスからプロセスへ直接には転送できない．

6.2.4 定義済みグループ内コミュニケータ

MPI_INITまたはMPI_INIT_THREADを呼び出すと（自分自身を含め）初期化後に通信できる全てのプロセスを含む初期グループ内コミュニケータであるMPI_COMM_WORLDが定義される．さらに，そのプロセスのみが含まれるコミュニケータMPI_COMM_SELFが提供される．

定義済み定数MPI_COMM_NULLは無効なコミュニケータハンドルに使用する値である．

MPIの静的なプロセスモデルの実装では，MPIの初期化後に計算に関係する全てのプロセスが利用可能となる．この場合，MPI_COMM_WORLDはその計算に使用可能な全てのプロセスのコミュニケータであり，全てのプロセスで同じ値を持つ．プロセスが動的にMPIの実行に参加できる実装においては，他の全てのプロセスにアクセスする以前に計算を開始する場合があり得る．この状況において，MPI_COMM_WORLDは参加プロセスが即座に通信できるプロセス全てを含むコミュニケータである．したがって，MPI_COMM_WORLDは同時に異なるプロセスにおいて異なる値を持つことがある．

全てのMPIの実装はMPI_COMM_WORLDコミュニケータを提供することが要求される．このコミュニケータはプロセスの実行中に解放することはできない．このコミュニケータに対応するグループは定義済み定数とはなっていないが，MPI_COMM_GROUPを使用してアクセスされる場合がある（下記参照）．MPIはMPI_COMM_WORLDのプロセスのランクとその（機種依存）絶対アドレスとの間の対応関係を指定しない．また，MPIはホストプロセスの関数を指定することもない．他の実装依存定義済みコミュニケータも提供される可能性もある．

6.3 グループ管理

この節では，MPIにおけるプロセスグループの操作について説明する．これらの操作はローカルなものであり，その実行にはプロセス間通信を必要としない．

6.3.1 グループアクセサ

MPI_GROUP_SIZE(group, size)

IN	group	グループ（ハンドル）
OUT	size	グループ内のプロセス数（整数型）

```
int MPI_Group_size(MPI_Group group, int *size)
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
{int MPI::Group::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GROUP_RANK(group, rank)

IN	group	グループ（ハンドル）
OUT	rank	呼び出しプロセスのグループ内のランク，もしくはプロセスがグループのメンバーでない場合はMPI_UNDEFINED （整数型）

```
int MPI_Group_rank(MPI_Group group, int *rank)
MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
{int MPI::Group::Get_rank() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)

IN	group1	グループ1（ハンドル）
IN	n	配列ranks1とranks2のランク数（整数型）
IN	ranks1	グループ1中の0以上の有効なランクの配列
IN	group2	グループ2（ハンドル）
OUT	ranks2	グループ2 において対応するランクの配列，対応するものがない場合はMPI_UNDEFINED

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
MPI_Group group2, int *ranks2)
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
{static void MPI::Group::Translate_ranks (const MPI::Group& group1, int n,
             const int ranks1[], const MPI::Group& group2, int ranks2[])
             （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は，2つの異なるグループに所属する同一のプロセスの順序付けの対応を決定するために使用する．例えば，MPI_COMM_WORLDのグループ中のプロセスのランクを知っている場合に，その部分集合のグループにおける同じプロセスのランクを知りたい場合がある．

MPI_PROC_NULLはMPI_GROUP_TRANSLATE_RANKSへの入力のための有効なランクである．MPI_PROC_NULLは変換されたランクとして返される．

MPI_GROUP_COMPARE(group1, group2, result)

IN	group1	第1グループ（ハンドル）
IN	group2	第2グループ（ハンドル）
OUT	result	結果（整数型）

```
int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
{static int MPI::Group::Compare(const MPI::Group& group1,
            const MPI::Group& group2) （廃止された呼び出し形式, 第15.2節を参照）}
```

グループのメンバと順序が2つのグループで同一ならば結果はMPI_IDENTとなる．これは，例えばgroup1およびgroup2が同じハンドルの場合に起こる．グループメンバは同じで

あるが順序が異なる場合にはMPI_SIMILARとなる．それ以外の場合にはMPI_UNEQUALとなる．

6.3.2 グループコンストラクタ

グループコンストラクタは，既存のグループのサブセットやスーパーセットを生成する場合に使用する．これらのコンストラクタは，既存のグループから新規グループを生成する．これらはローカルな操作であり，異なるグループを異なるプロセスで定義する場合がある．またプロセスはそれ自身を含まないグループを定義する場合もある．コミュニケータを作る関数の引数としてグループを使用する場合にはグループ内の各プロセスが同一のグループ定義を持つことが必要とされる．MPIは新規にグループを構築するためのメカニズムを提供せず，他のすでに定義されているグループからのみグループを作ることができる．初期コミュニケータMPI_COMM_WORLDに付随する（関数MPI_COMM_GROUPによりアクセス可能な）グループが他の全てのグループ定義のベースとなる．

根拠 後述するMPI_COMM_DUPに類似したグループ複製関数はない．グループが一度作成されると，ハンドルのコピーを作成することでそのグループへの複数の参照を持つことができるのでグループデュプリケータは必要ない本節に示すコンストラクタは既存のグループのサブセットおよびスーパーセットが必要な場合に対処するものである． （**根拠の終わり**）

実装者へのアドバイス 各グループコンストラクタは新しいグループオブジェクトを返すのと同様に動作する．この新規グループが既存グループのコピーの場合には，参照回数を管理することによって新しいオブジェクトを作成しないようにすることができる． （**実装者へのアドバイス終わり**）

MPI_COMM_GROUP(comm, group)

IN	comm		コミュニケータ（ハンドル）
OUT	group		commに対応するグループ（ハンドル）

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
{MPI::Group MPI::Comm::Get_group() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_COMM_GROUPはgroupにcommのグループのハンドルを返す．

MPI_GROUP_UNION(group1, group2, newgroup)

IN	group1		第1グループ（ハンドル）
IN	group2		第2グループ（ハンドル）
OUT	newgroup		和集合グループ（ハンドル）

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
{static MPI::Group MPI::Group::Union(const MPI::Group& group1,
             const MPI::Group& group2) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN	group1	第1グループ（ハンドル）
IN	group2	第2グループ（ハンドル）
OUT	newgroup	積集合グループ（ハンドル）

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
{static MPI::Group MPI::Group::Intersect(const MPI::Group& group1,
             const MPI::Group& group2) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN	group1	第1グループ（ハンドル）
IN	group2	第2グループ（ハンドル）
OUT	newgroup	差集合グループ（ハンドル）

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
{static MPI::Group MPI::Group::Difference(const MPI::Group& group1,
             const MPI::Group& group2) （廃止された呼び出し形式, 第15.2節を参照）}
```

集合演算に似た演算を次のように定義する.

和 第1グループ(group1)の全ての要素の後に，第1グループにない第2グループ(group2)の全ての要素を続ける.

積 第1グループの要素で第2グループの要素でもある全ての要素．順序は第1グループのとおりとする.

差 第2グループにない第1グループの要素の全ての要素で，順序は第1グループのとおりとする.

これらの演算では，出力グループのプロセス順序は可能ならば主として第1グループの順序で，ついで必要であれば第2グループの順序で決定される．和も積も可換ではないが，両方とも結合的である.

新しいグループは空の場合があり，MPI_GROUP_EMPTYに等しくなる.

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN	group	グループ（ハンドル）
IN	n	配列ranksの要素数（およびnewgroupのサイズ）（整数型）
IN	ranks	newgroupに出力されるgroup中のプロセスのランクの配列（整数配列）
OUT	newgroup	上記から生成された新しいグループ．順序はranksによって決まる．（ハンドル）

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
{MPI::Group MPI::Group::Incl(int n, const int ranks[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

関数MPI_GROUP_INCLは，グループgroup中のランクrank[0],..., rank[n-1]のn個のプロセスからなるグループnewgroupを生成する．newgroupの中のランクiのプロセスはgroupの中のランクranks[i]を持つプロセスである．ranksのnの要素のそれぞれはgroupの中の有効なランクでなければならず，全ての要素は異なっていなければならない．そうでない場合には，プログラムは誤りである．n = 0の場合，newgroupはMPI_GROUP_EMPTYである．例えば，この関数はグループの要素の順序を変更する場合に使用できる．MPI_GROUP_COMPAREも参照すること．

MPI_GROUP_EXCL(group, n, ranks, newgroup)

IN	group	グループ（ハンドル）
IN	n	配列ranksの要素数（整数型）
IN	ranks	newgroupに出力されないgroup中の整数のランクの配列（整数配列）
OUT	newgroup	上記から生成された新しいグループ．順序はgroupによって決まる．（ハンドル）

```
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
{MPI::Group MPI::Group::Excl(int n, const int ranks[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

関数MPI_GROUP_EXCLは，groupからランクranks[0] ,... ranks[n-1]のプロセスを削除することで得られるプロセスのグループnewgroupを作成する．newgroupの中のプロセスの順序付けはgroupにおける順序付けと同じである．ranksのn個の要素のそれぞれはgroup内で有効なランクでなければならず，全ての要素は異なっていなければならない．そうでない場合には，プログラムは誤りである．n = 0であれば，newgroupはgroupと同一である．

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

IN	group	グループ（ハンドル）
IN	n	配列rangesの3つ組みの数（整数型）
IN	ranges	newgroupに含まれるべきgroup中のプロセスのランクを示す（最初のランク，最後のランク，ストライド）の3つ組みの整数の 1次元配列
OUT	newgroup	上記から生成される新しいグループ．プロセスの順序は配列rangesによって決まる．（ハンドル）

```
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
MPI_Group *newgroup)
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
{MPI::Group MPI::Group::Range_incl(int n, const int ranges[][3]) const （廃
            止された呼び出し形式, 第15.2節を参照）}
```

rangesが次のような3つ組みだとすると，

$$(first_1, last_1, stride_1), ..., (first_n, last_n, stride_n)$$

newgroupは次のようなランクのgroup内のプロセスの列である．

$$first_1, first_1 + stride_1, ..., first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, ...$$

$$first_n, first_n + stride_n, ..., first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$$

計算で求めたそれぞれのランクはgroup内で有効なランクでなければならず，計算で求めた全てのランクは異なっていなければならない．そうでない場合，プログラムは誤りである．$first_i > last_i$や，$stride_i$が負の場合もあるが，0にはならないことに注意すること．

このルーチンの動作は，配列rangesをその配列によって指定されたランクを含む単一の配列へ拡張し，結果の配列ranksおよびその他の引数をMPI_GROUP_INCLに渡した動作と同等である．MPI_GROUP_INCLの呼び出しは，ranksの中の各ランクiを引数rangesの中の3つ組み(i,i,1)で置き換えたMPI_GROUP_RANGE_INCLの呼び出しと同等である．

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

IN	group	グループ（ハンドル）
IN	n	配列rangesの要素数（整数型）
IN	ranges	出力グループnewgroupから排除されるgroup中のプロセスのランクを示す（最初のランク，最後のランク，ストライド）の3つ組みの整数の1次元配列
OUT	newgroup	上記から生成される新しいグループ．groupの順序は保たれる．（ハンドル）

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
MPI_Group *newgroup)
```

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
{MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3]) const （廃
            止された呼び出し形式, 第15.2節を参照）}
```

計算で求めたそれぞれのランクはgroupの中で有効なランクでなければならず，計算で求めたランクは全て異なっていなければならない．そうでない場合，プログラムは誤りである．

このルーチンの動作は，配列rangesを除外されているランクを含む単一の配列へ拡張し，結果の配列ranksおよびその他の引数をMPI_GROUP_EXCLに渡した動作と同等である．MPI_GROUP_EXCLの呼び出しは，ranksの中の各ランクiを引数rangesの中の3つ組み(i,i,1)で置き換えたMPI_GROUP_RANGE_EXCLの呼び出しと同等である．

ユーザへのアドバイス 範囲操作はランクを明示的に列挙しないので，効率よく実装されていればよりスケーラブルである．高品質な実装ではこの利点を得られるのでMPIプログラマはできるかぎり範囲操作を使用するよう推奨する． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス （時間と空間の）よりよいスケーラビリティを得るため，できればグループメンバを列挙しない形で実装すべきである． （**実装者へのアドバイス終わり**）

6.3.3 グループデストラクタ

MPI_GROUP_FREE(group)

INOUT	group	グループ（ハンドル）

```
int MPI_Group_free(MPI_Group *group)
MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
{void MPI::Group::Free() （廃止された呼び出し形式, 第15.2節を参照）}
```

この操作は，グループオブジェクトに解放マークをつける．ハンドルgroupは，呼び出しによりMPI_GROUP_NULLに設定される．このグループを使用する実行中の操作は全て正常に完了する．

実装者へのアドバイス MPI_COMM_GROUP，MPI_COMM_CREATE，MPI_COMM_DUPを呼び出すごとにインクリメントされ，MPI_GROUP_FREEあるいはMPI_COMM_FREEを呼び出すごとにデクリメントされる参照カウントを持つことができる．グループオブジェクトは，参照カウントが0になると最終的に解放される． （**実装者へのアドバイス終わり**）

6.4 コミュニケータ管理

この節では，MPIにおけるコミュニケータの操作について説明する．コミュニケータを参照する操作はローカルであり，実行のためにプロセス間通信を必要としない．コミュニケータを生成する操作は集団的であり，プロセス間通信を必要とする場合がある．

実装者へのアドバイス 高品質な実装では，複数回にわたる呼び出しによる（同じグループ，またはそのサブセットの）コミュニケータの生成に関連するオーバーヘッドを，1つの集団的通信で複数のコンテクストを割り当てることにより低減しなければならない． （**実装者へのアドバイス終わり**）

6.4.1 コミュニケータアクセサ

次のものは全てローカル通信操作である．

MPI_COMM_SIZE(comm, size)

IN	comm	コミュニケータ（ハンドル）
OUT	size	commのグループのプロセス数（整数型）

```
int MPI_Comm_size(MPI_Comm comm, int *size)
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
{int MPI::Comm::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}
```

根拠 この関数は，MPI_COMM_GROUP でコミュニケータのグループにアクセスし（上記参照），MPI_GROUP_SIZEを使用してサイズを計算し，MPI_GROUP_FREEによって一時的なグループを解放する操作と等価である．しかし，この関数は非常によく使用されるのでショートカットが導入された． （**根拠の終わり**）

ユーザへのアドバイス この関数は，コミュニケータに関わるプロセスの個数を返す．MPI_COMM_WORLDの場合，これは利用可能なプロセスの総数を示している（MPIの本バージョンでは，初期化後にプロセスの個数を変更する標準的な手段は用意されていない）．

この関数は，特定のライブラリやプログラムで利用可能な並列度を決定するために次のMPI_COMM_RANKとともに使用されることが多い．関数MPI_COMM_RANKは，0...size−1の範囲で呼び出したプロセスのランクを示す．ただし，sizeはMPI_COMM_SIZEの戻り値である． （**ユーザへのアドバイス終わり**）

MPI_COMM_RANK(comm, rank)

IN	comm	コミュニケータ（ハンドル）
OUT	size	commのグループにおける呼び出しプロセスのランク（整数型）

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
{int MPI::Comm::Get_rank() const （廃止された呼び出し形式, 第15.2節を参照）}
```

根拠 この関数は，MPI_COMM_GROUPでコミュニケータのグループにアクセスし（上記参照），MPI_GROUP_RANKを使用してランクを計算し，MPI_GROUP_FREEによって一時的なグループを解放する操作と等価である．しかし，この関数は非常によく使用されるのでショートカットが導入された． （**根拠の終わり**）

ユーザへのアドバイス この関数は，特定のコミュニケータグループの中の自プロセスのランクを与える．上述のように，MPI_COMM_SIZEとともに使用すると便利である．

次のような動作をする多くのプログラムは，マスタースレーブモデルに基づいて作成される．つまり，1つのプロセス（ランク0のプロセスなど）の管理下に，他のプロセス群が計算ノードとしてサービスするような場合である．このモデルでは，前出の2つの呼び出しがコミュニケータの個々のプロセスの役割を決定するために使用される． （**ユーザへのアドバイス終わり**）

MPI_COMM_COMPARE(comm1, comm2, result)

IN	comm1	第1コミュニケータ（ハンドル）
IN	comm2	第2コミュニケータ（ハンドル）
OUT	result	結果（整数型）

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR
{static int MPI::Comm::Compare(const MPI::Comm& comm1,
              const MPI::Comm& comm2) （廃止された呼び出し形式, 第15.2節を参照）}
```

comm1とcomm2が同じオブジェクト（同一グループ，同一コンテクスト）のハンドルである場合，かつその場合にかぎりMPI_IDENTが返される．対応するグループのメンバおよびランク順序が同一の場合にはMPI_CONGRUENTが返される． この場合2つのコミュニケータはコンテクストのみが異なる．両方のコミュニケータのグループのメンバは同一であるが，ランク順が異なる場合にMPI_SIMILARとなる．これら以外の場合は，MPI_UNEQUALとなる．

6.4.2 コミュニケータコンストラクタ

次に示す関数は，コミュニケータcommに付随するグループまたは複数のグループ中の全てのプロセスにより呼び出される集団関数である．

> **根拠** MPIでは新しいコミュニケータを生成する場合にコミュニケータが必要とされるという点で，鶏が先か卵が先かという議論があり得ることに注意されたい．全てのMPIコミュニケータを生成するための基本コミュニケータはMPIの外であらかじめMPI_COMM_WORLDとして定義されている．このモデルは，多くの議論の後にMPIで書いたプログラムの「安全性」を高めるために選択された． （**根拠の終わり**）

MPIインターフェイスにはグループ内コミュニケータとグループ間コミュニケータの両方に適用される 4つのコミュニケータ構成ルーチンが用意されている．構成ルーチンMPI_INTERCOMM_CREATE（後述）はグループ間コミュニケータにのみ適用される．

グループ内コミュニケータは1つのグループにのみ関与するが，グループ間コミュニケータは2つのグループに関与する．以下のグループ間コミュニケータの説明では，グループ間コミュニケータの2つのグループを「**左**」グループおよび「**右**」グループと呼ぶ．グループ間コミュニケータのプロセスは左グループまたは右グループのメンバである．そのプロセスの観点から，プロセスが属するグループを「**ローカル**」グループと呼び，他方のグループを（そのプロセスとの対比で）「**リモート**」グループと呼ぶ．左グループおよび右グループというラベルにより，特定のプロセスに関係しないグループ間コミュニケータ内の 2つのグループ（ローカルグループおよびリモートグループ）を説明することができる．

MPI_COMM_DUP(comm, newcomm)

IN	comm	コミュニケータ（ハンドル）
OUT	newcomm	commのコピー（ハンドル）

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
{MPI::Intracomm MPI::Intracomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）
              }
{MPI::Intercomm MPI::Intercomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）
              }
{MPI::Cartcomm MPI::Cartcomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Graphcomm MPI::Graphcomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）
              }
{MPI::Distgraphcomm MPI::Distgraphcomm::Dup() const （廃止された呼び出し形式,
              第15.2節を参照）}
{MPI::Comm& MPI::Comm::Clone() const = 0 （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Intracomm& MPI::Intracomm::Clone() const （廃止された呼び出し形式, 第15.2節を
              参照）}
```

{MPI::Intercomm& MPI::Intercomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Cartcomm& MPI::Cartcomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Graphcomm& MPI::Graphcomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Distgraphcomm& MPI::Distgraphcomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}

MPI_COMM_DUPは，付随するキーとともに既存のコミュニケータcommを複製する．付随する各々のキーに対応するコピーコールバックは新しいコミュニケータに付随する属性値を決定する．コピーコールバック関数の利用方法として新規コミュニケータから属性を削除する操作が挙げられる．newcommには 同じグループ（または同じ複数グループ），コピーされたキャッシュ情報，かつ新しいコンテクストをもつ新規コミュニケータが返される（第6.7.1節を参照）．C++言語のDup()およびClone()の呼び出し形式については， 495ページの第16.1.7節を参照すること．

ユーザへのアドバイス この操作は，元のコミュニケータと同じ特性を持つ複製された通信空間を使用する並列ライブラリ関数を提供するためにある．この特性には，属性（下記参照）とトポロジー（第7章参照）が含まれる．この呼び出しは，コミュニケータcommに処理中の1対1通信がある場合でも有効である．典型的な呼び出しでは，並列関数のはじめにMPI_COMM_DUPが呼ばれ，それによって複製されたコミュニケータは関数の終わりでMPI_COMM_FREEによって解放される．他のコミュニケータ管理の方法も可能である．

この呼び出しは，グループ内コミュニケータとグループ間コミュニケータの両方に適用される． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 実際にはグループ情報をコピーする必要はなく，新しい参照を追加し，参照カウントをインクリメントすればよい．コピーオンライトの手法をキャッシュ情報に使用することもできる． （**実装者へのアドバイス終わり**）

MPI_COMM_CREATE(comm, group, newcomm)

IN	comm	コミュニケータ（ハンドル）
IN	group	commのグループのサブセットである グループ（ハンドル）
IN	comm	新しいコミュニケータ（ハンドル）

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

{MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const （廃止された呼び出し形式, 第15.2節を参照）}

commがグループ内コミュニケータの場合，この関数はgroup引数によって定義された通信グループを持つ新規コミュニケータnewcommを返す．キャッシュ情報はcommからnewcommに伝播されない．各プロセスはcommに関連するgroupのサブグループであるgroup引数により呼び出す必要があり，これはMPI_GROUP_EMPTYとすることもできる．プロセスごとにgroup引数に異なる値を指定することもできる．プロセスが空でないgroupにより呼び出された場合，そのgroupの全てのプロセスは引数と同じgroupにより関数を呼び出す必要がある．つまり，同じプロセスを同じ順序で呼び出す必要がある．そうでない場合，呼び出しは誤りである．このことは，複数プロセス間で指定される一連のグループがそれぞれ異なっていなければならないことを示している．呼び出しプロセスがgroup引数として渡されたグループのメンバである場合，newcommは関連グループとしてgroupを持つコミュニケータである．プロセスがMPI_GROUP_EMPTYなどの所属外のgroupにより呼び出しを行う場合，newcommとしてMPI_COMM_NULLが返される．関数は集団的であり，commのグループ内の全てのプロセスによって呼び出される必要がある．

根拠 このインターフェイスでは，全てのcommのプロセス内で同じgroupが必要とされる，元のMPI-1.1のメカニズムをサポートしている．重ならないサブグループのメンバシップに関する情報をユーザが持っている場合にMPI_COMM_SPLITで生じる不必要な通信を削減するための実装が行えるように，MPI-2.2では重ならないサブグループが使用できるよう拡張された． （**根拠の終わり**）

根拠 commのグループ全体が呼び出しを行うという要求条件は次の考察から生じた．

- 通常の集団的通信を使用してMPI_COMM_CREATEを実装できるようにする．
- 特に一部重なり合うグループ同士が新規コミュニケータを生成する場合を含め，さらなる安全性を提供する．
- コンテクスト生成時に，可能な場合には通信を避ける実装を許容する．

（**根拠の終わり**）

ユーザへのアドバイス MPI_COMM_CREATEは，別の通信空間で別のMIMD計算を行うためにプロセスグループのサブセットを提供するための手段を提供する．MPI_COMM_CREATEで得られたコミュニケータnewcommにさらにMPI_COMM_CREATE（または他のコミュニケータコンストラクタ）を適用し，計算を並列化された副計算に再分割することができる．より一般的なサービスは後述のMPI_COMM_SPLITが提供する． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス MPI_COMM_DUPの呼び出しでは，全てのプロセスで同じgroup（コミュニケータに関連するgroup）が使用される．MPI_COMM_CREATEの呼び出しでは，プロセスは同じgroupまたは重ならないサブグループを用意する．

この両方の呼び出しにおいて，理論的には通信なしでグループ全体でユニークなコンテクストに合意することが可能である． しかし，これらの関数をローカルに実行するにはより大きなコンテクスト名前空間の使用が必要となりまたエラー検査の機会を減らすことにもなる．実装に際してはこれらの相反する目標に対して，1つの集団操作で複数のコンテクストを一括して割り当てるなど，個々に妥協することができる．

重要：関与するプロセスと同期せずに新しいコミュニケータを生成する場合，通信システムは受信側プロセスでまだ割り当てられていないコンテクストに到達するメッセージに対処できなければならない． （**実装者へのアドバイス終わり**）

commがグループ間コミュニケータの場合，出力コミュニケータはローカルグループがgroupに含まれるプロセスのみで構成されるグループ間コミュニケータでもある（図6.1を参照）．group引数に含まれるプロセスは，newcommの一部である入力グループ間コミュニケータのローカルグループ内のプロセスのみとする必要がある．commの同じローカルグループ内の全てのプロセスはgroupに同じ値を指定する必要がある．groupにグループ間コミュニケータのローカルグループ内のプロセスが 1つも指定されていない，または呼び出しプロセスがgroupに含まれていない場合，MPI_COMM_NULLが返される．

根拠 左グループまたは右グループが空の場合，空のグループの側でMPI_COMM_NULLを返す必要があるため，MPI_GROUP_EMPTYのグループ間コミュニケータではなく，nullコミュニケータが返される． （**根拠の終わり**）

例 6.1 以下の例で，グループ間コミュニケータの左側の最初のノードがグループ間コミュニケータの右側の全てのメンバに加わって新しいコミュニケータを形成する方法を示す．

```
MPI_Comm  inter_comm, new_inter_comm;
MPI_Group local_group, group;
int       rank = 0; /* rank on left side to include in
                       new inter-comm */

/* Construct the original intercommunicator: "inter_comm" */
...

/* Construct the group of processes to be in new
   intercommunicator */
if (/* I'm on the left side of the intercommunicator */) {
  MPI_Comm_group ( inter_comm, &local_group );
  MPI_Group_incl ( local_group, 1, &rank, &group );
  MPI_Group_free ( &local_group );
}
else
  MPI_Comm_group ( inter_comm, &group );

MPI_Comm_create ( inter_comm, group, &new_inter_comm );
MPI_Group_free( &group );
```

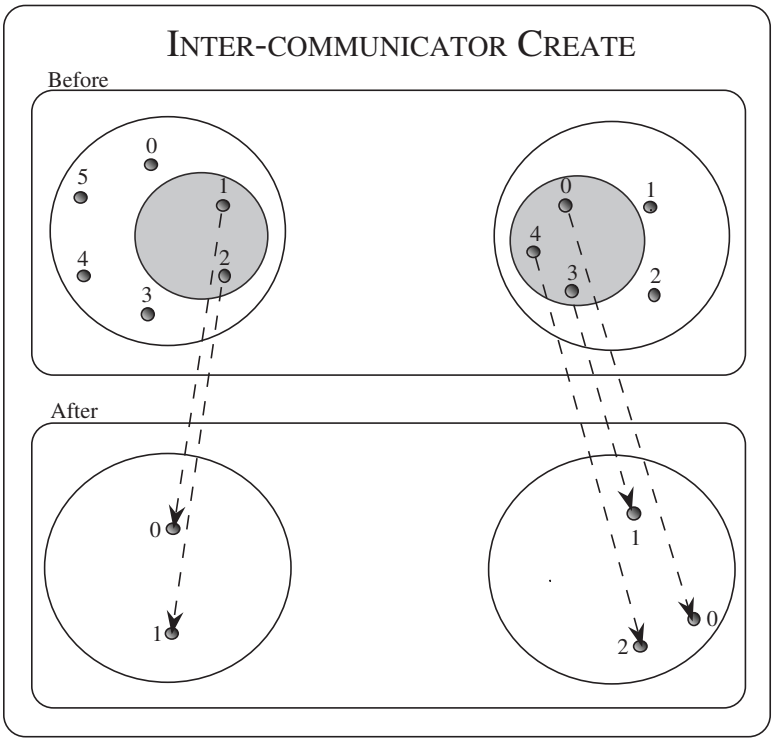

図 6.1: グループ間コミュニケータ向けに拡張したMPI_COMM_CREATEを使用したグループ間コミュニケータの作成．入力グループは灰色の円．

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	comm	コミュニケータ（ハンドル）
IN	color	サブセット割り当ての制御（整数型）
IN	key	ランク割り当ての制御（整数型）
OUT	newcomm	新規コミュニケータ（ハンドル）

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
{MPI::Intercomm MPI::Intercomm::Split(int color, int key) const （廃止された呼
            び出し形式, 第15.2節を参照）}
{MPI::Intracomm MPI::Intracomm::Split(int color, int key) const （廃止された呼
            び出し形式, 第15.2節を参照）}
```

この関数は，commに付随するグループをcolorの値に対応してそれぞれ重ならないサブグループに分割する．各サブグループは同一colorのプロセスを全て含む．各サブグループ内で，プロセスは引数keyの値の順序でランクが付けられ，同じkeyの値をもつプロセスは旧グループのランク順に従ってランクが決められる．各サブグループに対応した新しいコミュニケータが生成され，newcommに返される．プロセスはcolorの値としてMPI_UNDEFINEDを供給することができ，その場合，newcommにはMPI_COMM_NULLが返る．この関数は，集団呼び出しであるが，各プロセスは異なるcolorとkeyで呼び出すことができる．

グループ内コミュニケータcommでは，group引数のメンバであるプロセスが「color = groupの数」（重ならない全てのグループの独自の採番に基づく）と「key = group内でのランク」を渡し，group引数のメンバでない全てのプロセスが「color = MPI_UNDEFINED」を渡すとすれば，MPI_COMM_SPLIT(comm, color, key, newcomm)の呼び出しは，MPI_COMM_SPLIT(comm, color, key, newcomm)の呼び出しと同等である．

colorの値は非負でなければならない．

ユーザへのアドバイス これはプロセスグループをk個のサブグループに分割するためのきわめて強力なメカニズムである．ここでkは，プロセス全体に対して設定したカラーの個数で決まり，ユーザが暗黙のうちに選択した値である．生成したコミュニケータは互いに重なり合うことはない．このような分割は，マルチグリッド法や，線形代数などの計算処理の階層構造を定義する際に有用であろう． グループ内コミュニケータの場合，MPI_COMM_SPLITはMPI_COMM_CREATEと同様の機能を持ち，通信グループを重ならないサブグループに分割する．

MPI_COMM_SPLITは，一部のプロセスがグループ内の他のメンバに関する完全な情報を持っていないが，全てのプロセスが所属グループ（のcolor）を知っている場合に有益である．この場合，MPI実装は通信によって他のグループメンバを検出する．MPI_COMM_CREATEは全てのプロセスがグループ内のメンバに関する完全な情報を持っている場合に有益である．この場合，MPIではグループのメンバシップを検出するための余分な通信を回避することができる．

MPI_COMM_SPLITで生成されるコミュニケータ同士は重なり合わない（プロセスは，1回の呼び出しに付き1つのcolorのみを持つ）．MPI_COMM_SPLITを複数回呼び出すことでこの制約を解消できる．このように分割するには，colorおよびkeyの使用方法を工夫することが推奨される．

colorが固定の場合，keyは一意である必要はない． MPI_COMM_SPLITにより，このkeyに従ってプロセスが昇順にソートされ，keyの値が同じ場合は一貫性のある方法で順序付けされる．全てのkeyが同様に指定される場合，指定されたcolorの全てのプロセスは親グループとの相対のランク順となる．

基本的に，指定されたcolorの全てのプロセスに対してkeyを0にした場合，新しいコミュニケータでプロセスのランク順を意識する必要はない． （**ユーザへのアドバイス終わり**）

根拠 MPI_UNDEFINEDに割り当てられた値との衝突を避けるため，colorの値は非負に制限されている． （**根拠の終わり**）

グループ間コミュニケータに対するMPI_COMM_SPLITの結果として，右側のプロセスと同じcolorを持つ左側のプロセスが組み合わされて新しいグループ間コミュニケータが生成される．key引数はグループ間コミュニケータの両側のプロセスの相対ランクを示す

（図6.2を参照）．colorがグループ間コミュニケータの一方でしか指定されていない場合，MPI_COMM_NULLが返される．colorとしてMPI_UNDEFINEDが指定されたプロセスについても，MPI_COMM_NULLが返される．

ユーザへのアドバイス グループ間コミュニケータの場合，MPI_COMM_CREATEよりMPI_COMM_SPLITの方が一般的である．MPI_COMM_SPLITの1回の呼び出しで重なりのない一連のグループ間コミュニケータを生成することができ，MPI_COMM_CREATEの呼び出しでは1つのグループ間コミュニケータの生成のみができる．（**ユーザへのアドバイス終わり**）

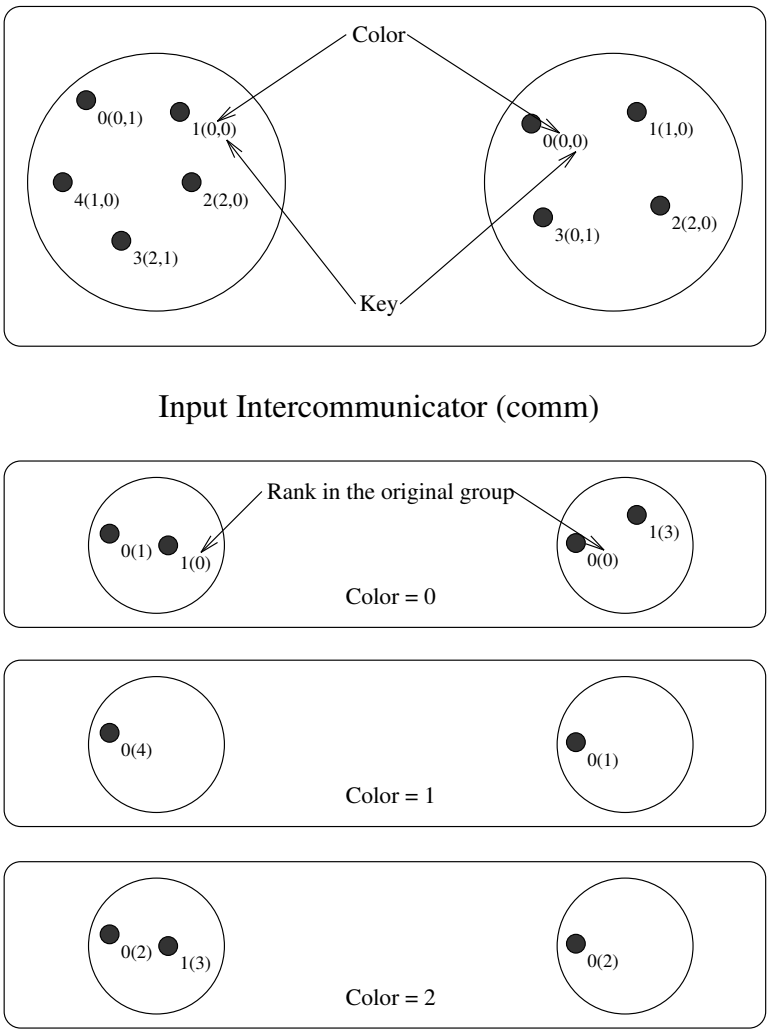

図 6.2: グループ間コミュニケータをMPI_COMM_SPLITで分割した例.

例 6.2 （並列クライアント／サーバ）．以下のクライアントのコードに，グループ間コ

ミュニケータの左側のクライアントをグループ間コミュニケータの右側のサーバプール中の1台のサーバに割り当てる方法を示す.

```
/* Client code */
MPI_Comm  multiple_server_comm;
MPI_Comm  single_server_comm;
int       color, rank, num_servers;

/* Create intercommunicator with clients and servers:
   multiple_server_comm */
...

/* Find out the number of servers available */
MPI_Comm_remote_size ( multiple_server_comm, &num_servers );

/* Determine my color */
MPI_Comm_rank ( multiple_server_comm, &rank );
color = rank % num_servers;

/* Split the intercommunicator */
MPI_Comm_split ( multiple_server_comm, color, rank,
                 &single_server_comm );
```

対応するサーバのコードを以下に示す.

```
/* Server code */
MPI_Comm  multiple_client_comm;
MPI_Comm  single_server_comm;
int       rank;

/* Create intercommunicator with clients and servers:
   multiple_client_comm */
...

/* Split the intercommunicator for a single server per group
   of clients */
MPI_Comm_rank ( multiple_client_comm, &rank );
MPI_Comm_split ( multiple_client_comm, rank, 0,
                 &single_server_comm );
```

6.4.3 コミュニケータデストラクタ

MPI_COMM_FREE(comm)

INOUT	comm	破壊されるコミュニケータ（ハンドル）

```
int MPI_Comm_free(MPI_Comm *comm)
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

{void MPI::Comm::Free() （廃止された呼び出し形式, 第15.2節を参照）}

この集団操作は，通信オブジェクトに解放のマークを付ける．ハンドルは MPI_COMM_NULLにセットされる．このコミュニケータを使用中の未終了の操作は全て

正常終了する．なぜなら，オブジェクトは，アクティブな参照がほかにない場合にかぎり実際に解放されるからである．この呼び出しはグループ内コミュニケータとグループ間コミュニケータに適用される．全てのキャッシュ属性に対する削除コールバック関数（第6.7節を参照）は任意の順序で呼び出される．

> **実装者へのアドバイス** MPI_COMM_DUPの呼び出しによってインクリメントされ，MPI_COMM_FREEの呼び出しによってデクリメントされるような参照カウントメカニズムを使用することができる．オブジェクトはカウントが0になると最終的に解放される．
>
> この関数は集団的であるが，通常ローカルに実装されることが期待される．ただし，MPIライブラリのデバッグバージョンにおいては同期操作としての実装を選択することができる．（**実装者へのアドバイス終わり**）

6.5 例題

6.5.1 一般的な慣例#1

例#1a:

```
int main(int argc, char **argv)
{
  int me, size;
  ...
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  (void)printf ("Process %d size %d\n", me, size);
  ...
  MPI_Finalize();
}
```

例#1aは「何もしない」プログラムで，プログラム自身を正しく初期化し，「全プロセスを含む」コミュニケータを参照し，メッセージを出力する．また，このプログラムは正しく終了している．この例は，MPIがprintf形式の通信自体をサポートしていることを意味していない．

例#1b（sizeが偶数であると仮定している）:

```
int main(int argc, char **argv)
{
   int me, size;
   int SOME_TAG = 0;
   ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

   if((me % 2) == 0)
   {
```

```
        /* send unless highest-numbered process */
        if((me + 1) < size)
           MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
     }
     else
        MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);

     ...
     MPI_Finalize();
  }
```

例#1bは，「全プロセスを含む」コミュニケータにおける「偶数」プロセスと「奇数」プロセスとの間のメッセージ交換を概略的に示している．

6.5.2 一般的な慣例#2

```
int main(int argc, char **argv)
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
      /* get input, create buffer ``data'' */
      ...
  }

  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

  ...

  MPI_Finalize();
}
```

この例は集団的通信の使用法を説明している．

6.5.3 （おおむね）一般的な慣習#3

```
int main(int argc, char **argv)
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group MPI_GROUP_WORLD, grprem;
  MPI_Comm commslave;
  static int ranks[] = {0};
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* local */

  MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);  /* local */
  MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

  if(me != 0)
```

```
  {
    /* compute on slave */
    ...
    MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1, commslave);
    ...
    MPI_Comm_free(&commslave);
  }
  /* zero falls through immediately to this reduce, others do later... */
  MPI_Reduce(send_buf2, recv_buff2, count2,
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Group_free(&grprem);
  MPI_Finalize();
}
```

この例は，0番目のプロセス以外の全てのプロセスからなるグループを（「全プロセスからなる」グループから）どのように生成するか，その新しいグループに対応したコミュニケータ(commslave)をどのように形成するかを示している．新しいコミュニケータは集団呼び出しで使用され，全てのプロセスはMPI_COMM_WORLDコンテクストで集団呼び出しを実行する．この例は，それぞれ異なるコンテクストを持つ2つのコミュニケータがどのように通信を保護しているかを示している．つまり，MPI_COMM_WORLDの通信はcommslaveの通信から隔離されるし，またその逆もあるということである．

ここで示したように，どのプロセスにおいても異なるコミュニケータ内のコンテクストは互いに隔離されるので「グループ安全性」はコミュニケータを介して実現される．

6.5.4 例#4

次の例は，1対1通信と集団的通信との間の「安全性」を示すものである．MPIは，1つのコミュニケータが安全な1対1通信および集団的通信を実行できることを保証している．

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT       50

int main(int argc, char **argv)
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group MPI_GROUP_WORLD, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

  MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
  MPI_Group_rank(subgroup, &me);     /* local */

  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);

  if(me != MPI_UNDEFINED)
```

```
    {
        MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                          the_comm, request);
        MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                          the_comm, request+1);
        for(i = 0; i < SOME_COUNT, i++)
          MPI_Reduce(..., the_comm);
        MPI_Waitall(2, request, status);

        MPI_Comm_free(&the_comm);
    }

    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Group_free(&subgroup);
    MPI_Finalize();
}
```

6.5.5 ライブラリの例#1

メインプログラム:

```
int main(int argc, char **argv)
{
  int done = 0;
  user_lib_t *libh_a, *libh_b;
  void *dataset1, *dataset2;
  ...
  MPI_Init(&argc, &argv);
  ...
  init_user_lib(MPI_COMM_WORLD, &libh_a);
  init_user_lib(MPI_COMM_WORLD, &libh_b);
  ...
  user_start_op(libh_a, dataset1);
  user_start_op(libh_b, dataset2);
  ...
  while(!done)
  {
     /* work */
     ...
     MPI_Reduce(..., MPI_COMM_WORLD);
     ...
     /* see if done */
     ...
  }
  user_end_op(libh_a);
  user_end_op(libh_b);

  uninit_user_lib(libh_a);
  uninit_user_lib(libh_b);
  MPI_Finalize();
}
```

ユーザライブラリの初期化コード:

```
void init_user_lib(MPI_Comm comm, user_lib_t **handle)
{
  user_lib_t *save;

```

```
     user_lib_initsave(&save); /* local */
     MPI_Comm_dup(comm, &(save -> comm));

     /* other inits */
     ...

     *handle = save;
   }
```

ユーザスタートアップコード:

```
   void user_start_op(user_lib_t *handle, void *data)
   {
     MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
     MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
   }
```

ユーザ通信クリーンアップコード:

```
   void user_end_op(user_lib_t *handle)
   {
     MPI_Status status;
     MPI_Wait(handle -> isend_handle, &status);
     MPI_Wait(handle -> irecv_handle, &status);
   }
```

ユーザオブジェクトクリーンアップコード:

```
   void uninit_user_lib(user_lib_t *handle)
   {
     MPI_Comm_free(&(handle -> comm));
     free(handle);
   }
```

6.5.6 ライブラリの例#2

メインプログラム:

```
   int main(int argc, char **argv)
   {
     int ma, mb;
     MPI_Group MPI_GROUP_WORLD, group_a, group_b;
     MPI_Comm comm_a, comm_b;

     static int list_a[] = {0, 1};
#if  defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
     static int list_b[] = {0, 2 ,3};
#else/* EXAMPLE_2A */
     static int list_b[] = {0, 2};
#endif
     int size_list_a = sizeof(list_a)/sizeof(int);
     int size_list_b = sizeof(list_b)/sizeof(int);

     ...
     MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
```

```
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);

     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
     if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);

     if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);

     if(comm_b != MPI_COMM_NULL)
     {
       lib_call(comm_b);
       lib_call(comm_b);
     }

     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
     MPI_Group_free(&MPI_GROUP_WORLD);
     MPI_Finalize();
   }
```

ライブラリ:

```
   void lib_call(MPI_Comm comm)
   {
     int me, done = 0;
     MPI_Status status;
     MPI_Comm_rank(comm, &me);
     if(me == 0)
        while(!done)
        {
           MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
           ...
        }
     else
     {
       /* work */
       MPI_Send(..., 0, ARBITRARY_TAG, comm);
       ....
     }
#ifdef EXAMPLE_2C
     /* include (resp, exclude) for safety (resp, no safety): */
     MPI_Barrier(comm);
#endif
   }
```

上記の例はlist_bにランク3を含むかどうか，そして同期がlib_callに含まれるかどうかにより，実際には3つの例となっている．この例は，コンテクストを使用したとしても，同じコンテクストで複数回のlib_callの呼び出しを行うとこれらの呼び出しは必ずしも互

いに安全な状態にないということを示している（いわゆる，「バックマスキング」現象が発生し得る）．MPI_Barrierが追加された場合に安全な通信が実現する．この例は，コンテクストを使用したとしてもライブラリは慎重に作成しなければならないということを示している．ランク3がない場合には，バックマスキングを避けて安全性を確保するための同期は必要ない．

「リデュース」や「オールリデュース」などのアルゴリズムでは，MPIの基本機能によって送信元を安全に選択できるので，本質的にバックマスキングの問題は発生しない．同じ根または異なる根の一般的なツリー型ブロードキャストアルゴリズムを複数回呼び出す場合（文献[45]を参照のこと）も同様である．ここで，同じコンテクストにおけるプロセス間メッセージの2者間での順序保証，および送信元を安全に選択できるというMPIの2つの性質が前提となる．いずれかの特徴を削除すると，バックマスキングが発生しないという保証がなくなる．

決定性が失われたアルゴリズム送り先の順番が決まらないブロードキャストを行おうとするアルゴリズム順序が非決定的 順序が非決定的なブロードキャストや，ワイルドカード操作を含む他の呼び出しを実行しようとするアルゴリズムは，「リデュース」，「オールリデュース」，「ブロードキャスト」の順序が決定的な実装が持つ，良い特性を一般には持たない．このようなアルゴリズムにおいて，正しい処理を行うためには（コミュニケータのスコープ内で）単調増加するタグを利用しなくてはならないこともある．

前述の議論は全て1対1通信で実装された「集団呼び出し」を仮定している．MPIでは集団呼び出しは1対1通信を使用して実装してもいいし，しなくてもよい．これらのアルゴリズムは，MPIでの集団呼び出しの実装方法とは関係なく，正確さと安全性の議論のために例示された．第6.9節も参照のこと．

6.6 グループ間通信

本節では，グループ間通信の概念を導入し，それをサポートするMPIの仕様について説明を加える．さらに，利用者レベルのサーバを含むプログラムを作成するためのサポートについても述べる．

これまでに述べてきた1対1通信は，同じグループのメンバであるプロセス間の通信のみを考慮の対象としてきた．本章の前の部分で述べたように，このようなタイプの通信は「グループ内通信」と呼ばれ，そのような通信で使用されるコミュニケータは「グループ内コミュニケータ」と呼ばれる．

モジュール化され，複数分野にまたがるアプリケーションでは，異なるプロセスグループがそれぞれ違ったモジュールを実行したり，異なるモジュールに含まれるプロセスが互いにパイプライン的な，あるいはより一般的なグラフ状の通信を行なったりする．このようなアプリケーションでは，あるプロセスが通信の相手プロセスを特定する最も自然な方法は，相手グループ中での相手プロセスのランクを指定することである．一方，内部に利用者レベルのサーバを含むようなアプリケーションでは，各サーバが複数のクライアントに対してサービスを提供するようなプロセスグループであったり，各クライアントが複数のサーバからのサービスを受けるプロセスグループであるような状況が生

ずる．このようなアプリケーションにおいても，相手プロセスを相手グループにおけるランクを用いて指定することがきわめて自然だと考えられる．先にも述べたように，このようなタイプの通信は「グループ間通信」と呼ばれ，ここで使われるコミュニケータは「グループ間コミュニケータ」と呼ばれる．

グループ間通信は，異なるグループに属するプロセス間での1対1通信である．グループ間通信を開始するプロセスを含むグループは「ローカルグループ」と呼ばれる．送信操作における送信プロセス，あるいは受信操作における受信プロセスがこれに相当する．一方，通信の相手プロセスを含むグループは「リモートグループ」と呼ばれる．送信操作における受信プロセス，受信操作における送信プロセスがこれに相当する． グループ内コミュニケータの場合と同様，相手プロセスは(communicator, rank)のペアを用いて指定される．しかし，グループ内通信と異なり，ランクはリモートグループに従って定義される．

グループ間コミュニケータに関する全てのコンストラクタの動作はブロッキングである．ローカルグループとリモートグループは重なりを持ってはならない．

> **ユーザへのアドバイス** いくつかの理由により，グループは重なりがないことが重要となる．まず，グループ間コミュニケータの目的は，重なりのないグループ間の通信のためのコミュニケータを提供することである．これはMPI_INTERCOMM_MERGEの定義にも反映され，これを使用してユーザは生成されたグループ内コミュニケータ内のプロセスのランク付けを制御することができ，このランク付けは，グループに重なりがあるとほとんど意味がない．また，集団操作をグループ間コミュニケータに自然に拡張することは，グループに重なりがない場合に高い効果が得られる． （**ユーザへのアドバイス終わり**）

以下に，グループ間通信とグループ間コミュニケータの特性をまとめる．

- 1対1通信および集団的通信のシンタックスは，グループ間通信，グループ内双方の通信に関して同様である．同じコミュニケータを送信操作と受信操作の双方に用いることができる．
- 相手プロセスは，送信と受信の双方に関して，リモートグループでのランクによって指定される．
- グループ間コミュニケータを用いた通信は，異なるコミュニケータを用いた他のいかなる通信とも干渉しないことが保証される．
- 1つのコミュニケータはグループ内，グループ間通信のいずれかの通信にのみ使用でき，両方に用いることはできない．

あるコミュニケータがグループ内コミュニケータか，グループ間コミュニケータのいずれであるかを知るためには，関数MPI_COMM_TEST_INTERを用いることができる．グループ間コミュニケータは，コミュニケータの情報を参照するためのいくつかの関数に対して，引数として与えることができる．一方，グループ内コミュニケータのためのコ

ンストラクタ関数のうちいくつかは，グループ間コミュニケータを入力引数としてとることができないものがある（例えばMPI_CART_CREATE）.

実装者へのアドバイス 1対1通信を実現するために，コミュニケータは各プロセスにおいて以下のような項目の組によって表現することができる.

group

send_context

receive_context

source

グループ間コミュニケータに対しては，**group**はリモートグループを表わし，**source**はローカルグループでのランクを表わす．グループ内コミュニケータに対しては，**group**はコミュニケータグループ（リモートグループ=ローカルグループ）を，**source**はそのグループにおけるプロセスのランクを表わす．送信コンテクストと受信コンテクストは同一である．**group**は，ランクからプロセスの絶対位置への変換テーブルとして表現される.

グループ間コミュニケータを論ずるには，ローカルグループとリモートグループそれぞれにおける 2つのプロセスを考える必要がある．いま，グループ$\mathcal{P}$ におけるプロセス **P** がグループ間コミュニケータ$\mathbf{C}_{\mathcal{P}}$を持ち，グループ$\mathcal{Q}$ におけるプロセス **Q** がグループ間コミュニケータ$\mathbf{C}_{\mathcal{Q}}$を持つとする．このとき，

- $\mathbf{C}_{\mathcal{P}}$**.group**はグループ$\mathcal{Q}$ を表わし，$\mathbf{C}_{\mathcal{Q}}$**.group**はグループ$\mathcal{P}$ を表わす.
- $\mathbf{C}_{\mathcal{P}}$**.send_context** = $\mathrm{C}_{\mathcal{Q}}$**.receive_context** であり，そのコンテクストはグループ$\mathcal{Q}$ に関して一意である．同様に，
  $\mathbf{C}_{\mathcal{P}}$**.receive_context** = $\mathbf{C}_{\mathcal{Q}}$**.send_context** であり，そのコンテクストはグループ$\mathcal{P}$に対して一意である.
- $\mathbf{C}_{\mathcal{P}}$**.source**はグループ$\mathcal{P}$ におけるプロセス**P**のランクであり， $\mathbf{C}_{\mathcal{Q}}$**.source**はグループ$\mathcal{Q}$ におけるプロセス**Q**のランクである.

プロセス**P**がグループ間コミュニケータを用いてプロセス**Q**にメッセージを送る場合を考える．このとき，**P**は**group**テーブルを用いて**Q**の絶対位置を見出す．**source**と**send_context**はメッセージに付加される.

プロセス**Q**が**source**を明示的に指定してグループ間コミュニケータを用いて受信をポストする場合を考える．このとき，**Q**は**receive_context**をメッセージの**context**と照合し，**soruce**引数をメッセージの**source**と照合する.

同様なアルゴリズムは，グループ内コミュニケータに関しても適用できる.

グループ間コミュニケータのアクセサとコンストラクタをサポートするためには，上に述べたモデルにさらに構造体を追加することが必要である．この構造体は，ロ

ーカル通信グループと付加的な安全性保証のためのコンテクストに関する情報を格納するために用いる．

（実装者へのアドバイス終わり）

6.6.1 グループ間コミュニケータのアクセサ

MPI_COMM_TEST_INTER(comm, flag)

IN	comm	コミュニケータ（ハンドル）
OUT	flag	（論理型）

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG
{bool MPI::Comm::Is_inter() const （廃止された呼び出し形式, 第15.2節を参照）}
```

このローカル関数によって，呼び出し元プロセスは，あるコミュニケータがグループ間コミュニケータかグループ内コミュニケータかを知ることができる．本関数は，前者であればtrueを，後者であればfalseを返す．

これまでに挙げられたグループ内コミュニケータに関するアクセサに対し，入力引数としてグループ間コミュニケータを与えた場合，それらの関数は次の表に示すように動作する．

MPI_COMM_SIZE	ローカルグループのサイズを返す．
MPI_COMM_GROUP	ローカルグループを返す．
MPI_COMM_RANK	ローカルグループ内でのランクを返す．

表 6.1: （グループ間コミュニケータモードでの）MPI_COMM_*関数の振る舞い

さらに，MPI_COMM_COMPARE操作はグループ間コミュニケータに関しても用いることができる．双方のコミュニケータはグループ内コミュニケータかグループ間コミュニケータのいずれかである必要があり，そうでない場合にはMPI_UNEQUALが返される．MPI_CONGRUENTとMPI_SIMILARの結果を得るためには，対応するローカルグループとリモートグループがそれぞれ正しく比較できなくてはならない．特に，ローカルグループ，リモートグループの一方でも類似してはいても完全に同じではないという場合，MPI_SIMILARが返される可能性がある．

次に挙げるアクセサは，つねにグループ間コミュニケータのリモートグループに関するアクセス手段を提供する．

これらは全てローカル操作である．

MPI_COMM_REMOTE_SIZE(comm, size)

IN	comm	グループ間コミュニケータ（ハンドル）
OUT	size	commのリモートグループ内のプロセス数（整数型）

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
{int MPI::Intercomm::Get_remote_size() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_COMM_REMOTE_GROUP(comm, group)

IN	comm	グループ間コミュニケータ（ハンドル）
OUT	size	commに対応するリモートグループ（ハンドル）

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
{MPI::Group MPI::Intercomm::Get_remote_group() const （廃止された呼び出し形式, 第15.2節を参照）}
```

根拠 グループ間コミュニケータを構成するローカルグループとリモートグループに対称な方法でアクセスできることは重要である．そのために，本関数およびMPI_COMM_REMOTE_SIZE が提供されている． （**根拠の終わり**）

6.6.2 グループ間コミュニケータの操作

本節では，グループ間コミュニケータに関する4つのブロッキング操作を紹介する．MPI_INTERCOMM_CREATEは2つのグループ内コミュニケータを結合してグループ間コミュニケータとするのに用いられるMPI_INTERCOMM_MERGE関数はグループ間コミュニケータを構成するローカルグループとリモートグループを結合して1つのグループ内コミュニケータを生成する．以前にも述べたMPI_COMM_DUPとMPI_COMM_FREEはそれぞれ，グループ間コミュニケータを複製，解放する．

グループ間コミュニケータの構成要素となるローカルグループとリモートグループとの間の重複は認められない．重複がある場合，そのプログラムは誤りであり，デッドロックに陥る可能性が高い．（プロセスがマルチスレッド構成であり，MPIの関数呼び出しがプロセス全体ではなく1つのスレッドしかブロックしないのであれば，プロセスが重複してグループに属することも可能である．その場合には，利用者の責任において，そのプロセスの2つの「役割」がそれぞれ別のスレッドによって実行されることを保証する必要がある．）

MPI_INTERCOMM_CREATE関数は，次に述べる状況下で，2つの既存のグループ内コミュニケータからグループ間コミュニケータを生成するのに用いられる．まず，それぞれのグループ中で，選ばれた少なくとも1つのメンバ（「グループリーダ」）が他方のグループのグループリーダと通信できなくてはならない．すなわち，2つのグループリーダが

属する「ピア」コミュニケータが存在し，かつ各リーダは他方のリーダの仲介コミュニケータ内でのランクを知っている必要がある．さらに，各グループのメンバは，それぞれのリーダのランクを知っていなくてはならない．

2つのグループ内コミュニケータからのグループ間コミュニケータの生成時には，ローカルグループとリモートグループでの独立した集団操作と，ローカルグループのプロセスとリモートグループのプロセスの間での1対1通信が必要となる．

標準的なMPIの実装（初期化時に静的にプロセスが割り当てられる）では，MPI_COMM_WORLDコミュニケータ（より好ましくは，ピア専用のMPI_COMM_WORLDの複製）がピアコミュニケータとなりうる．スポーンやジョインを使ったアプリケーションについては， ピアとして使われるべきグループ間コミュニケータを最初に生成する必要があるかもしれない．

第7章で述べるアプリケーショントポロジーに関する関数は，グループ間コミュニケータに対しては適用することはできない．このような機能が必要な利用者は，MPI_INTERCOMM_MERGEを用いてグループ内コミュニケータを生成し，それに対してグラフあるいはカルテシアントポロジーを当てはめて適当なトポロジー属性を持たせるのが良い．あるいは，一般性を失わないような独自のアプリケーショントポロジーメカニズムを考案することも可能である．

MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)

IN	local_comm	ローカルグループ内コミュニケータ（ハンドル）
IN	local_leader	local_comm内のローカルグループリーダのランク（整数型）
IN	peer_comm	「仲介」コミュニケータ．local_leaderでのみ意味を持つ（ハンドル）
IN	remote_leader	peer_comm内のリモートグループリーダのランク．local_leaderでのみ意味を持つ．（整数型）
IN	tag	「安全」タグ（整数型）
OUT	newintercomm	新規グループ間コミュニケータ（ハンドル）

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
TAG, NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR
{MPI::Intercomm MPI::Intracomm::Create_intercomm(int local_leader, const
             MPI::Comm& peer_comm, int remote_leader, int tag) const （廃止
             された呼び出し形式, 第15.2節を参照）}
```

本関数はグループ間コミュニケータを生成する．動作は，ローカルグループとリモ

ートグループの和集合について集団的である．プロセスは各グループ内で同一の引数local_commとlocal_leaderを指定する必要がある．remote_leader，local_leader，およびtagに関してワイルドカードを用いることはできない．

本関数はリーダ間の1対1通信に関して，コミュニケータpeer_commとタグtagを用いる．したがって，peer_comm上でこの通信と干渉する可能性のある未完了の通信が存在しないように注意を払う必要がある．

ユーザへのアドバイス ピアコミュニケータの使用にともなう問題を避けるために，例えばMPI_COMM_WORLDの複製のような専用のピアコミュニケータを利用することを推奨する．（**ユーザへのアドバイス終わり**）

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

IN	intercomm	グループ間コミュニケータ（ハンドル）
IN	high	（論理型）
OUT	newintracomm	新規コミュニケータ（ハンドル）

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
MPI_Comm *newintracomm)
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH
{MPI::Intracomm MPI::Intercomm::Merge(bool high) const （廃止された呼び出し形式，
                第15.2節を参照）}
```

本関数は，intercommを構成する2つのグループの和集合から 1つのグループ内コミュニケータを生成する．全てのプロセスは，その2つのグループのそれぞれで同一のhighの値を設定しなくてはならない．一方のグループに属するプロセスがhigh = falseを設定し，他方のグループに属するプロセスがhigh = trueを設定した場合，和集合内でのランクは“low”のグループが“high”のグループより前になるように決定される．全てのプロセスが同じhighの値を設定したならば，和集合内でのランクは任意となる．この関数は2つのグループの和集合に関して集団的であり，かつブロッキングである．

各プロセスの新規グループ間コミュニケータのエラーハンドラは，ローカルグループに寄与するコミュニケータから継承される．このため，同じコミュニケータ内のプロセスごとにエラーハンドラが異なるという状態が発生する可能性がある．

実装者へのアドバイス MPI_INTERCOMM_MERGE，MPI_COMM_FREE，MPI_COMM_DUPの実装はMPI_INTERCOMM_CREATEの実装と似ている．ただし前3者では，グループリーダ間の通信において，仲介コミュニケータのコンテクストではなく入力のグループ間コミュニケータ固有のコンテクストを用いる．（**実装者へのアドバイス終わり**）

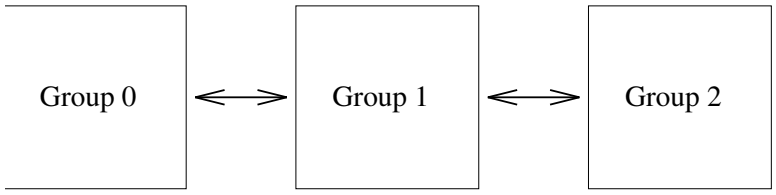

図 6.3: 3グループでのパイプライン.

6.6.3 グループ間コミュニケータの使用例

例1: 3グループでの「パイプライン」

グループ0と1とが通信を行ない，グループ1と2とが通信を行なう．したがって，グループ0は1つのグループ間コミュニケータを，グループ1は2つのグループ間コミュニケータを，グループ2は1つのグループ間コミュニケータを必要とする.

```
int main(int argc, char **argv)
{
  MPI_Comm   myComm;       /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm;  /* inter-communicator */
  MPI_Comm   mySecondComm; /* second inter-communicator (group 1 only) */
  int membershipKey;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* User code must generate membershipKey in the range [0, 1, 2] */
  membershipKey = rank % 3;

  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

  /* Build inter-communicators.  Tags are hard-coded. */
  if (membershipKey == 0)
  {                     /* Group 0 communicates with group 1. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
  }
  else if (membershipKey == 1)
  {              /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
  }
  else if (membershipKey == 2)
  {                    /* Group 2 communicates with group 1. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &myFirstComm);
  }

  /* Do work ... */

  switch(membershipKey)  /* free communicators appropriately */
  {
```

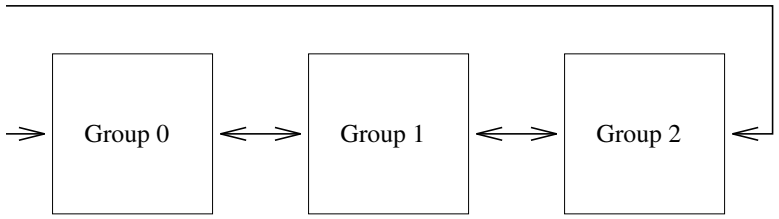

図 6.4: 3グループでのリング.

```
  case 1:
     MPI_Comm_free(&mySecondComm);
  case 0:
  case 2:
     MPI_Comm_free(&myFirstComm);
     break;
  }

  MPI_Finalize();
}
```

例2: 3グループでの「リング」

グループ0と1，グループ1と2，グループ2と0がそれぞれ通信を行なう．したがって，それぞれが2つのグループ間コミュニケータを必要とする．

```
int main(int argc, char **argv)
{
  MPI_Comm   myComm;      /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm; /* inter-communicators */
  MPI_Comm   mySecondComm;
  MPI_Status status;
  int membershipKey;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...

  /* User code must generate membershipKey in the range [0, 1, 2] */
  membershipKey = rank % 3;

  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

  /* Build inter-communicators.  Tags are hard-coded. */
  if (membershipKey == 0)
  {             /* Group 0 communicates with groups 1 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          2, &mySecondComm);
  }
  else if (membershipKey == 1)
  {         /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
```

```
                            1, &myFirstComm);
      MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                            12, &mySecondComm);
    }
    else if (membershipKey == 2)
    {       /* Group 2 communicates with groups 0 and 1. */
      MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                            2, &myFirstComm);
      MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                            12, &mySecondComm);
    }

    /* Do some work ... */

    /* Then free communicators before terminating... */
    MPI_Comm_free(&myFirstComm);
    MPI_Comm_free(&mySecondComm);
    MPI_Comm_free(&myComm);
    MPI_Finalize();
  }
```

6.7 キャッシング

MPIは，アプリケーションで3種類のMPIオブジェクトであるコミュニケータ，ウィンドウ，データ型に**属性**と呼ばれる任意の情報を付加するため，「キャッシング」という機能を用意している．より正確に言えば，キャッシング機能を用いれば，次のような機能を可搬なライブラリとして実装できるようになる．

- MPIのグループ内コミュニケータ，グループ間コミュニケータ，ウィンドウ，またはデータ型と情報を関連付けることにより呼び出し間で情報を伝え，
- その情報を高速に検索し，
- 古くなった情報が再利用されないことを保証する．これはオブジェクトが解放され，その後，MPIによって そのオブジェクトが使用していたのと同じハンドルが再利用された場合にも保証される．

集団的通信やアプリケーショントポロジーを扱うようなMPIの組み込みルーチンでキャッシングの能力が必要とされる．これらの機能へのインターフェイスをMPIの標準の一部として定義することには重要である．それは集団的通信とアプリケーショントポロジーを扱うようなルーチンが可搬になるためである．そしてまた，ユーザが記述したルーチンで標準のMPIの呼び出し手順を用いることにより拡張性がより向上するためである．

ユーザへのアドバイス コミュニケータMPI_COMM_SELFは，この属性キャッシングメカニズムを通してプロセスにローカルな属性を設定するのに適している． （**ユーザへのアドバイス終わり**）

根拠 極端な例では，全ての不可視なハンドルでキャッシングを行うことができる．あるいは，コミュニケータでのみ行うこともできる．キャッシングはコストが伴うため，明らかに必要で，コスト増加が過度でない場合だけ使用するように制限する必要がある．そのため，他のハンドルではなく，ウィンドウとデータ型が追加されている． （**根拠の終わり**）

Fortran言語の整数型とC言語のポインタのサイズの違いが問題となることがある．コミュニケータでの属性のキャッシングによりこの問題に対処するため，このケースのための関数も用意されている．データ型とウィンドウでキャッシングを行うための関数もこの問題に対応している．アドレスサイズの問題については，第16.3.6節を参照すること．

実装者へのアドバイス 高品質な実装では，MPI_XXX_CREATE_KEYVALの呼び出しにより生成されたkeyvalがMPI_YYY_GET_ATTR，MPI_YYY_SET_ATTR，MPI_YYY_DELETE_ATTR，MPI_YYY_FREE_KEYVALの呼び出し時に型の違ったオブジェクトと一緒に使用された場合に，エラーを発生させる必要がある．そのためには，関連するユーザ関数の型に関する情報を各keyvalで管理する必要がある．（**実装者へのアドバイス終わり**）

6.7.1 機能説明

コミュニケータ，ウィンドウ，データ型に属性を結び付けることができる． 属性はプロセスに対してローカルであり，またそれらが結び付けられたコミュニケータに固有である．属性は，MPI_COMM_DUPによって複製された場合を除いてコミュニケータからコミュニケータへと伝播することはない（そして，アプリケーションは属性をコピーするためにコールバック関数に許可を与える必要がある）．

ユーザへのアドバイス C言語の属性はvoid *型である．通常，このような属性は詳しい情報が記述された構造体のポインタまたはMPIオブジェクトのハンドルである．Fortran言語では属性は整数型(INTEGER)である．このような属性は，MPIオブジェクトのハンドル，または単なる整数型の属性とすることができる． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 属性はC言語のポインタのサイズと同じ大きさかあるいはそれより大きなサイズのスカラ値である．属性は，MPIのハンドルを常に保持することができる． （**実装者へのアドバイス終わり**）

ここで定義されるキャッシングインターフェイスでは，MPIによるコミュニケータ，ウィンドウ，データ型の内部での属性の格納が不可視的でなければならない．アクセサ関数は次のものを含んでいる．

- （属性を識別するための）キー値を得る．コミュニケータが破壊あるいはコピーされた時にMPIがそのアプリケーションを知らせるためのコールバック関数を指定する．

- 属性の値の保持と取得を行う．

実装者へのアドバイス キャッシングとコールバック関数は，アプリケーションの明示的なリクエストに対して，常に同期して呼び出される．これによりユーザ領域とシステム領域間で繰り返される交錯の問題を避けることができる（この同期呼び出し規則は，MPIの一般的な特性である）．

キー値の選択はMPIに一任されている．これは，MPIが属性の組の実装を最適に行うためである．また，これにより同一のコミュニケータを使用する独立したモジュールでのキャッシング情報のコンフリクト衝突を避けることができる．

コールバック機能だけからなるかなり小さなインターフェイスでは，可搬なコードで完全なキャッシング機能を実装することが可能かもしれない．しかし，こうした最小のインターフェイスでは， 任意のコミュニケータを処理するには，ある種の表探索が暗に必要になる．一方ここではより完全なインターフェイスを定義している．これは属性への高速なアクセスを可能にするためである． コミュニケータの中で（対応する属性テーブルを見つけるための）ポインタの使用，および（個々の属性を計算するための）賢く選択されたキー値の使用を通して 属性への高速なアクセスを達成する．最小のインターフェイスの持つ固有の効果的な点を考慮しても，ここで 定義されるより完全なインターフェイスの方が優れているように思われる． （**実装者へのアドバイス終わり**）

MPIはキャッシングに関して次のようなサービスを提供する．これらは全てプロセスでローカルである．

6.7.2 コミュニケータ

コミュニケータでキャッシングを行うための関数を以下に示す．

MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, extra_state)

IN	comm_copy_attr_fn	comm_keyvalのためのコピーコールバック関数（関数）
IN	comm_delete_attr_fn	comm_keyvalのための削除コールバック関数（関数）
OUT	comm_keyval	今後のアクセスのためのキー値（整数型）
IN	extra_state	コールバック関数のためのその他の状態

```
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval,
void *extra_state)
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

{static int MPI::Comm::Create_keyval(MPI::Comm::Copy_attr_function*
comm_copy_attr_fn,
MPI::Comm::Delete_attr_function* comm_delete_attr_fn,
void* extra_state) （廃止された呼び出し形式, 第15.2節を参照）}

この関数は新規の属性キーを生成する．キーはプロセスにおいてローカルで一意であり，ユーザからは不可視であり，明示的に整数型として格納される．割り当てが完了すると，キー値を使用して属性を関連付け，ローカルに定義されたあらゆるコミュニケータ上で属性にアクセスすることができる．

この関数は，廃止されたMPI_KEYVAL_CREATEに代わるものである．C言語の呼び出し形式も同じである．Fortran言語の呼び出し形式は，extra_stateがアドレスサイズの整数型である点が異なる．また，コピーと削除のコールバック関数は，アドレスサイズの属性と整合性のあるFortran言語の呼び出し形式を備えている．

C言語のコールバック関数は

```
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
void *extra_state, void *attribute_val_in, void *attribute_val_out,
int *flag);
```

および

```
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
void *attribute_val, void *extra_state);
```

である．これはMPI-1.1の呼び出しと同じであるが，名前が新しくなっている．古い名前は廃止された．

Fortran言語のコールバック関数は

```
SUBROUTINE COMM_COPY_ATTR_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

および以下である．

```
SUBROUTINE COMM_DELETE_ATTR_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
EXTRA_STATE, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

C++言語のコールバック関数は

{typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,
int comm_keyval, void* extra_state, void* attribute_val_in,
void* attribute_val_out, bool& flag); （廃止された呼び出し形式．
第15.2節を参照）}

および以下である．

{typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm,
int comm_keyval, void* attribute_val, void* extra_state); （廃止された呼び出し形式．第15.2節を参照）}

comm_copy_attr_fn関数はコミュニケータがMPI_COMM_DUPによって複製されたときに呼び出される．comm_copy_attr_fnはMPI_Comm_copy_attr_function型でなければならない． コピーコールバック関数は，oldcomm中のそれぞれのキー値に対して任意の順に呼び出される．それぞれのコピーコールバック関数の呼び出しは，キー値とそ

れに一致する属性に対して行なわれる．flag = 0が返された場合には，複製されたコミュニケータ中の属性は削除されている．そうでない場合(flag = 1)には新しい属性値がattribute_val_outを通じてセットされる．この関数は成功時にはMPI_SUCCESSを返し，エラー時にはエラーコードを返す（エラー時には，MPI_COMM_DUPは失敗する）．

C言語，C++言語，Fortran言語において，引数comm_copy_attr_fnをMPI_COMM_NULL_COPY_FNまたはMPI_COMM_DUP_FNとして指定することができる．MPI_COMM_NULL_COPY_FNはflag = 0をセットし，MPI_SUCCESSを返すだけの関数である．単純なコピー関数としてMPI_COMM_DUP_FNが用意されている．この関数はflag = 1をセットし，attribute_val_outにattribute_val_inの値を返し，MPI_SUCCESSを返却値とする． これら関数は廃止されたMPI-1の定義済みコールバックMPI_NULL_COPY_FNおよびMPI_DUP_FNに代わるものである．

ユーザへのアドバイス attribute_val_inとattribute_val_outは両方ともvoid *型であるが，その使い方は異なっている．C言語のコピー関数は，attribute_val_in中の属性の**値**をMPIを通じでコピーし，attribute_val_outに属性の**アドレス**を入れる．これはこの関数が，（新しい）属性値を返すことができるようにするためである．void *型を利用するのは，煩雑なキャストを避けるためである．

有効なコピー関数には2種類ある．1つは属性を含んだデータ構造まるごとをコピーすることにより情報を完全に複製するものである．もう1つは，他のデータ構造への参照をリファレンスカウント方式で持つだけの方法である．後者の方法では属性のその他の型はまったくコピーされない（それらはoldcommだけで指定されている可能性がある）． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス C言語のインターフェイスはC言語で作成されたキー値によるコピーと削除の関数を仮定していて，Fortran言語の呼び出しインターフェイスはFortran言語で作成されたキー値を仮定している． （**実装者へのアドバイス終わり**）

comm_copy_attr_fnと類似しているのは次のように定義される削除コールバック関数である．comm_delete_attr_fn関数は，コミュニケータがMPI_COMM_FREEによって削除された場合か，明示的にMPI_COMM_DELETE_ATTRが呼び出された場合に呼び出される．comm_delete_attr_fnはMPI_Comm_delete_attr_function型でなくてはならない．

この関数は，MPI_COMM_FREE，MPI_COMM_DELETE_ATTR，MPI_COMM_SET_ATTRにより属性が削除されなくてはならなくなった時には常に呼び出される．この関数は成功時にはMPI_SUCCESSを返し，失敗時にはエラーコードを返す（この時には，MPI_COMM_FREEは失敗する）．

C言語，C++言語，Fortran言語において，引数comm_delete_attr_fnをMPI_COMM_NULL_DELETE_FNとして指定することができる．MPI_COMM_NULL_DELETE_FNはMPI_SUCCESSを返すだけの関数である．MPI_COMM_NULL_DELETE_FNは廃止されたMPI_NULL_DELETE_FNに代わるものである．

属性コピー関数または属性削除関数がMPI_SUCCESS以外の値を返す場合，これを呼び出した関数呼び出し（MPI_COMM_FREEなど）は誤りである．

特別なキー値MPI_KEYVAL_INVALIDがMPI_KEYVAL_CREATEによって返されることはない．そのため，キー値の静的な初期化のために使用することができる．

実装者へのアドバイス C++言語のルーチンMPI::Comm::Create_keyvalの呼び出しで，comm_copy_attr_fn引数として定義済みのC言語の関数MPI_COMM_NULL_COPY_FNまたはMPI_COMM_DUP_FNを使用したり，comm_delete_attr_fn引数として定義済みのC言語の関数MPI_COMM_NULL_DELETE_FNを使用したりできるようにするために，第1入力引数，第2入力引数，または両方の入力引数としてC言語の関数が使用できる（C++言語のプロトタイプに一致する引数の代わりに）3つのルーチンでこのルーチン(MPI::Comm::Create_keyval)をオーバーロードしてもよい． （**実装者へのアドバイス終わり**）

ユーザへのアドバイス 内部的にMPI::Comm::Create_keyvalを呼び出す「ラッパー」ルーチンを記述しようとしていて，comm_copy_attr_fnやcomm_delete_attr_fnがこのラッパールーチンの引数である場合，またユーザ定義のC++言語のコピーおよび削除関数と定義済みのC言語関数の両方でこのラッパールーチンをコールバックする必要がある場合，上記の「実装者へのアドバイス」で説明したのと同じオーバーロードが必要となる場合もある． （**ユーザへのアドバイス終わり**）

MPI_COMM_FREE_KEYVAL(comm_keyval)

INOUT	comm_keyval	キー値（整数型）

```
int MPI_Comm_free_keyval(int *comm_keyval)
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
    INTEGER COMM_KEYVAL, IERROR
{static void MPI::Comm::Free_keyval(int& comm_keyval) （廃止された呼び出し形式,
              第15.2節を参照）}
```

現存の属性キーを解放する．この関数はkeyvalの値をMPI_KEYVAL_INVALIDにセットする．使用中の属性キーを解放することは誤りではないことに注意．実際の解放は，（そのプロセスでの他のコミュニケータによる）そのキーへの全ての参照が解放されるまで行われないからである．これらの参照は，MPI_COMM_DELETE_ATTRを呼び出して1つの属性のインスタンスを解放するか，あるいはMPI_COMM_FREEを呼び出して解放済みのコミュニケータに関係する全ての属性を解放することにより，プログラムで明示的に解放する必要がある．

この呼び出しはMPI-1の呼び出しMPI_KEYVAL_FREEと同じだが，新しいコミュニケータ固有の作成関数と対応させる必要がある．MPI_KEYVAL_FREEは廃止された．

MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

INOUT	comm	属性を結びつけるコミュニケータ（ハンドル）
IN	comm_keyval	キー値（整数型）
IN	attribute_val	属性値

```
int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
{void MPI::Comm::Set_attr(int comm_keyval, const void* attribute_val) const
             （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は，指定された属性値attribute_valを格納する．この属性値は後にMPI_COMM_GET_ATTRにより取り出される．値が既に存在していた場合，その結果はMPI_COMM_DELETE_ATTRが呼び出されて以前の値が削除された（そしてコールバック関数comm_delete_attr_fnが実行された）後に新しい値が格納されたのと同様となる． keyvalの値に対応するキーがない場合，呼び出しは誤りである．特にMPI_KEYVAL_INVALIDは誤ったキー値である．comm_delete_attr_fn関数がMPI_SUCCESSと異なるエラーコードを返した時は，本関数は失敗する．

この関数は廃止されたMPI_ATTR_PUTに代わるものである．C言語の呼び出し形式は同じである．Fortran言語の呼び出し形式は，attribute_valがアドレスサイズの整数型である点が異なる．

MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)

IN	comm	属性が結びつけられているコミュニケータ（ハンドル）
IN	comm_keyval	キー値（整数型）
OUT	attribute_val	属性値（flag = falseでない場合）
OUT	flag	キーと関連する属性がない場合，false（論理型）

```
int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
int *flag)
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
{bool MPI::Comm::Get_attr(int comm_keyval, void* attribute_val) const （廃止
             された呼び出し形式, 第15.2節を参照）}
```

キー値をもとに属性を検索する．keyvalの値に対応するキーがない場合，呼び出しは誤りである．一方，キー値は存在するが，commに属性が存在しない場合にはその呼び出しは正しい．そのような場合には，呼び出しによってflag = falseが返される．特にMPI_KEYVAL_INVALIDは誤ったキー値である．

ユーザへのアドバイス MPI_Comm_set_attrの呼び出しでは，attribute_valに属性の値が渡される．MPI_Comm_get_attrでは，attribute_valに，属性値の返されるべ

きアドレスが渡される．したがって，属性値それ自体がvoid*型のポインタである場合，MPI_Comm_set_attrの実際のattribute_valパラメータはvoid*型であり，MPI_Comm_get_attrの実際のattribute_valパラメータはvoid**型である．（**ユーザへのアドバイス終わり**）

根拠 正式なパラメータattribute_valまたはvoid*型（void**ではなく）を使用することにより，属性値がvoid*以外の型で宣言される場合に必要な煩雑なキャストを行わずに済む．（**根拠の終わり**）

この関数は廃止されたMPI_ATTR_GETに代わるものである．C言語の呼び出し形式は同じである．Fortran言語の呼び出し形式は，attribute_valがアドレスサイズの整数型である点が異なる．

MPI_COMM_DELETE_ATTR(comm, comm_keyval)

INOUT	comm	属性の削除の対象となるコミュニケータ（ハンドル）
IN	comm_keyval	キー値（整数型）

```
int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
{void MPI::Comm::Delete_attr(int comm_keyval) （廃止された呼び出し形式, 第15.2節を参照）}
```

キーに基づいてキャッシュから属性を削除する．この関数は，keyvalの作成時に指定された属性削除関数comm_delete_attr_fnを呼び出す．comm_delete_attr_fn関数がMPI_SUCCESS以外のエラーコードを返した場合は，本関数は失敗する．

コミュニケータが関数MPI_COMM_DUPを用いて複製される時は必ず，現在設定されている属性の全てのコールバックコピー関数が（任意の順で）呼び出される．コミュニケータが関数MPI_COMM_FREEを用いて削除された時には必ず，現在設定されている属性の全てのコールバック削除関数が呼び出される．

この関数はMPI_ATTR_DELETEと同じだが，新しいコミュニケータ固有の関数と対応させる必要がある．MPI_ATTR_DELETEは廃止された．

6.7.3 ウィンドウ

ウィンドウのキャッシング用の新しい関数を以下に示す．

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

IN	win_copy_attr_fn	win_keyvalのためのコピーコールバック関数（関数）
IN	win_delete_attr_fn	win_keyvalのための削除コールバック関数（関数）
OUT	win_keyval	今後のアクセスのためのキー値（整数型）
IN	extra_state	コールバック関数のためのその他の状態

```
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
MPI_Win_delete_attr_function *win_delete_attr_fn, int *win_keyval,
void *extra_state)
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
{static int MPI::Win::Create_keyval(MPI::Win::Copy_attr_function*
             win_copy_attr_fn,
             MPI::Win::Delete_attr_function* win_delete_attr_fn,
             void* extra_state)} （廃止された呼び出し形式, 第15.2節を参照）
```

C言語，C++言語，Fortran言語において，引数win_copy_attr_fnをMPI_WIN_NULL_COPY_FNまたはMPI_WIN_DUP_FNとして指定することができる．MPI_WIN_NULL_COPY_FNは flag = 0をセットし，MPI_SUCCESSを返すだけの関数である．単純なコピー関数としてMPI_WIN_DUP_FNが用意されている．この関数はflag = 1をセットし，attribute_val_outにattribute_val_inの値を返し，MPI_SUCCESSを返却値とする．

C言語，C++言語，Fortran言語において，引数win_delete_attr_fnをMPI_WIN_NULL_DELETE_FNとして指定することができる．MPI_WIN_NULL_DELETE_FNはMPI_SUCCESSを返すだけの関数である．

C言語のコールバック関数は

```
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
void *extra_state, void *attribute_val_in, void *attribute_val_out,
int *flag);
```

および以下である．

```
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
void *attribute_val, void *extra_state);
```

Fortran言語のコールバック関数は

```
SUBROUTINE WIN_COPY_ATTR_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

および以下である．

```
SUBROUTINE WIN_DELETE_ATTR_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

C++言語のコールバックは
{typedef int MPI::Win::Copy_attr_function(const MPI::Win& oldwin,
int win_keyval, void* extra_state, void* attribute_val_in,
void* attribute_val_out, bool& flag); （廃止された呼び出し形式. 第15.2節を参照）}

および以下である.
{typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval,
void* attribute_val, void* extra_state); （廃止された呼び出し形式. 第15.2節を参照）}

属性コピー関数または属性削除関数がMPI_SUCCESS以外の値を返す場合，これを呼び出した関数呼び出し（MPI_WIN_FREEなど）は誤りである.

MPI_WIN_FREE_KEYVAL(win_keyval)

INOUT	win_keyval	キー値（整数型)

```
int MPI_Win_free_keyval(int *win_keyval)
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR
{static void MPI::Win::Free_keyval(int& win_keyval) （廃止された呼び出し形式,
              第15.2節を参照）}
```

MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)

INOUT	win	属性を結びつけるウィンドウ（ハンドル）
IN	win_keyval	キー値（整数型）
IN	attribute_val	属性値

```
int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
{void MPI::Win::Set_attr(int win_keyval, const void* attribute_val) （廃止さ
              れた呼び出し形式, 第15.2節を参照）}
```

MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)

IN	win	属性が結びつけられているウィンドウ（ハンドル）
IN	win_keyval	キー値（整数型）
OUT	attribute_val	属性値（flag = falseでない場合）
OUT	flag	キーと関連する属性がない場合，false（論理型）

```
int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
int *flag)
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
{bool MPI::Win::Get_attr(int win_keyval, void* attribute_val) const （廃止さ
              れた呼び出し形式, 第15.2節を参照）}
```

MPI_WIN_DELETE_ATTR(win, win_keyval)

INOUT	win	属性の削除の対象となるウィンドウ（ハンドル）
IN	win_keyval	キー値（整数型）

```
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
{void MPI::Win::Delete_attr(int win_keyval) （廃止された呼び出し形式, 第15.2節を参照）
             }
```

6.7.4 データ型

データ型のキャッシングのための新しい関数を以下に示す.

MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)

IN	type_copy_attr_fn	type_keyvalのためのコピーコールバック関数（関数）
IN	type_delete_attr_fn	type_keyvalのための削除コールバック関数（関数）
OUT	type_keyval	今後のアクセスのためのキー値（整数型）
IN	extra_state	コールバック関数のためのその他の状態

```
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
MPI_Type_delete_attr_function *type_delete_attr_fn, int *type_keyval,
void *extra_state)
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
EXTRA_STATE, IERROR)
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
{static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function*
             type_copy_attr_fn, MPI::Datatype::Delete_attr_function*
             type_delete_attr_fn, void* extra_state) （廃止された呼び出し形式,
             第15.2節を参照）}
```

C言語，C++言語，Fortran言語において，引数type_copy_attr_fnをMPI_TYPE_NULL_COPY_FNまたはMPI_TYPE_DUP_FNとして指定することができる. MPI_TYPE_NULL_COPY_FNはflag = 0をセットし，MPI_SUCCESSを返すだけの関数である．単純なコピー関数としてMPI_TYPE_DUP_FNが用意されている．この関数はflag = 1をセットし，attribute_val_outにattribute_val_inの値を返し，MPI_SUCCESSを返却値とする.

C言語，C++言語，Fortran言語において，引数type_delete_attr_fnをMPI_TYPE_NULL_DELETE_FNとして指定することができる. MPI_TYPE_NULL_DELETE_FNはMPI_SUCCESSを返すだけの関数である.

C言語のコールバック関数は

```
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
int type_keyval, void *extra_state, void *attribute_val_in,
void *attribute_val_out, int *flag);
```

および以下である.

```
typedef int MPI_Type_delete_attr_function(MPI_Datatype type,
int type_keyval, void *attribute_val, void *extra_state);
```

Fortran言語のコールバック関数は

```
SUBROUTINE TYPE_COPY_ATTR_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
        ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

および以下である.

```
SUBROUTINE TYPE_DELETE_ATTR_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
EXTRA_STATE, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

C++言語のコールバックは

```
{typedef int
            MPI::Datatype::Copy_attr_function(const MPI::Datatype& oldtype,
            int type_keyval, void* extra_state,
            const void* attribute_val_in, void* attribute_val_out,
            bool& flag); （廃止された呼び出し形式. 第15.2節を参照）}
```

および以下である.

```
{typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& type,
            int type_keyval, void* attribute_val, void* extra_state); （廃
            止された呼び出し形式. 第15.2節を参照）}
```

属性コピー関数または属性削除関数がMPI_SUCCESS以外の値を返す場合，これを呼び出した関数呼び出し（MPI_TYPE_FREEなど）は誤りである.

MPI_TYPE_FREE_KEYVAL(type_keyval)

INOUT	type_keyval	キー値（整数型）

```
int MPI_Type_free_keyval(int *type_keyval)
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR
{static void MPI::Datatype::Free_keyval(int& type_keyval) （廃止された呼び出し形
            式, 第15.2節を参照）}
```

MPI_TYPE_SET_ATTR(type, type_keyval, attribute_val)

INOUT	type	属性を結びつけるデータ型（ハンドル）
IN	type_keyval	キー値（整数型）
IN	attribute_val	属性値

```
int MPI_Type_set_attr(MPI_Datatype type, int type_keyval,
void *attribute_val)
MPI_TYPE_SET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

{void MPI::Datatype::Set_attr(int type_keyval, const void* attribute_val)
（廃止された呼び出し形式, 第15.2節を参照）}

MPI_TYPE_GET_ATTR(type, type_keyval, attribute_val, flag)

IN	type	属性が結びつけられているデータ型（ハンドル）
IN	type_keyval	キー値（整数型）
OUT	attribute_val	属性値（flag = falseでない場合）
OUT	flag	キーと関連する属性がない場合，false（論理型）

```
int MPI_Type_get_attr(MPI_Datatype type, int type_keyval, void
*attribute_val, int *flag)
MPI_TYPE_GET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
{bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const
```
（廃止された呼び出し形式, 第15.2節を参照）}

MPI_TYPE_DELETE_ATTR(type, type_keyval)

INOUT	type	属性の削除の対象となるデータ型（ハンドル）
IN	type_keyval	キー値（整数型）

```
int MPI_Type_delete_attr(MPI_Datatype type, int type_keyval)
MPI_TYPE_DELETE_ATTR(TYPE, TYPE_KEYVAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
{void MPI::Datatype::Delete_attr(int type_keyval) （廃止された呼び出し形式,
```
第15.2節を参照）}

6.7.5 無効なKeyvalに対するエラークラス

属性のキー値はMPI_{TYPE,COMM,WIN}_CREATE_KEYVALによりシステムで割り当てられる．このような値のみを，キー値を入力引数として使用する関数に渡すことができる．これらの関数のいずれかに誤ったキー値が渡されたことを警告するため，MPIの新しいエラークラスMPI_ERR_KEYVALが用意されている．これを返すのは， MPI_ATTR_PUT，MPI_ATTR_GET，MPI_ATTR_DELETE，MPI_KEYVAL_FREE，MPI_{TYPE,COMM,WIN}_DELETE_ATTR，MPI_{TYPE,COMM,WIN}_SET_ATTR，MPI_{TYPE,COMM,WIN}_GET_ATTR，MPI_{TYPE,COMM,WIN}_FREE_KEYVAL，MPI_COMM_DUP，MPI_COMM_DISCONNECT，MPI_COMM_FREEである．最後の3つがここに加わっているのは，keyvalが属性のコピー関数と削除関数の引数であるためである．

6.7.6 属性の例

ユーザへのアドバイス この例では2度目以降の集団操作が効果的にキャッシングを利用するにはどのように書けばよいかを示している．コーディングスタイルは，

MPI関数の結果がエラーステータスのみを返すことを想定している．（ユーザへのアドバイス終わり）

```
/* key for this module's stuff: */
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct
{
   int ref_count;          /* reference count */
   /* other stuff, whatever else we want */
} gop_stuff_type;

Efficient_Collective_Op (comm, ...)
MPI_Comm comm;
{
  gop_stuff_type *gop_stuff;
  MPI_Group       group;
  int             foundflag;

  MPI_Comm_group(comm, &group);

  if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
  {
    if ( ! MPI_Comm_create_keyval( gop_stuff_copier,
                                   gop_stuff_destructor,
                                   &gop_key, (void *)0));
    /* get the key while assigning its copy and delete callback
       behavior. */

    MPI_Abort (comm, 99);
  }

  MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
  if (foundflag)
  { /* This module has executed in this group before.
       We will use the cached information */
  }
  else
  { /* This is a group that we have not yet cached anything in.
       We will now do so.
    */

    /* First, allocate storage for the stuff we want,
       and initialize the reference count */

    gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }

    gop_stuff -> ref_count = 1;

    /* Second, fill in *gop_stuff with whatever we want.
       This part isn't shown here */

    /* Third, store gop_stuff as the attribute value */
    MPI_Comm_set_attr ( comm, gop_key, gop_stuff);
  }
  /* Then, in any case, use contents of *gop_stuff
     to do the global op ... */
}
```

```
   /* The following routine is called by MPI when a group is freed */

   gop_stuff_destructor (comm, keyval, gop_stuff, extra)
   MPI_Comm comm;
   int keyval;
   gop_stuff_type *gop_stuff;
   void *extra;
   {
     if (keyval != gop_key) { /* abort -- programming error */ }

     /* The group's being freed removes one reference to gop_stuff */
     gop_stuff -> ref_count -= 1;

     /* If no references remain, then free the storage */
     if (gop_stuff -> ref_count == 0) {
       free((void *)gop_stuff);
     }
   }

   /* The following routine is called by MPI when a group is copied */
   gop_stuff_copier (comm, keyval, extra, gop_stuff_in, gop_stuff_out, flag)
   MPI_Comm comm;
   int keyval;
   gop_stuff_type *gop_stuff_in, *gop_stuff_out;
   void *extra;
   {
     if (keyval != gop_key) { /* abort -- programming error */ }

     /* The new group adds one reference to this gop_stuff */
     gop_stuff -> ref_count += 1;
     gop_stuff_out = gop_stuff_in;
   }
```

6.8 命名オブジェクト

エラーレポート，デバッグ，プロファイリングなどの際に，MPIのコミュニケータ，ウィンドウ，またはデータ型と，印字可能な識別子とを関係づけることができれば便利な場合がよくある．不可視なオブジェクトに結びつけられた名前は，オブジェクトがMPIルーチンによって複製またはコピーされたときに伝播されない．コミュニケータについては，以下の2つの関数を使用してこれに対応することができる．

MPI_COMM_SET_NAME (comm, comm_name)

INOUT	comm	識別子をセットする対象となるコミュニケータ（ハンドル）
IN	comm_name	名前として記憶される文字列（文字列）

```
int MPI_Comm_set_name(MPI_Comm comm, char *comm_name)
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) COMM_NAME
```

{void MPI::Comm::Set_name(const char* comm_name) （廃止された呼び出し形式, 第15.2節を参照）}

MPI_COMM_SET_NAMEを使用すると，コミュニケータと名前の文字列を関連付けることができる．MPI_COMM_SET_NAMEに渡される文字列は MPIライブラリ内部に保存される（そのため，呼び出しの直後に呼び出し側が解放するか，スタックに割り当てることができる）．nameの先頭のスペースは意味を持つが，末尾のスペースは意味を持たない．

MPI_COMM_SET_NAMEはローカルな操作（かつ非集団操作）で，MPI_COMM_SET_NAMEの呼び出しを行ったプロセス内で認識されるコミュニケータの名前のみに影響する．必ずしも，全てのプロセス内のコミュニケータに同じ（あるいは何らかの）名前を割り当てる必要はない．

ユーザへのアドバイス MPI_COMM_SET_NAMEはコードのデバッグ支援のために用意されているので，混乱を避けるため，全プロセスで同一コミュニケータに同じ名前を割り当てることは意味がある． （**ユーザへのアドバイス終わり**）

格納可能な名前の長さは，Fortran言語ではMPI_MAX_OBJECT_NAMEの値までに制限され，C言語およびC++言語ではnullターミネータを考慮してMPI_MAX_OBJECT_NAME-1までに制限されている．これより長い名前をつけるようとすると，名前の切り捨てが発生する．MPI_MAX_OBJECT_NAMEの値は少なくとも64でなければならない．

ユーザへのアドバイス メモリ不足の場合， どんな長さの名前をつけようとしても失敗する可能性が残るため， MPI_MAX_OBJECT_NAMEの値について，この長さより短い名前が必ず設定できるという保証ではなく，ただ名前の長さの厳格な上限としてのみ考える必要がある． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 名前用に固定サイズの領域をあらかじめ割り当てる実装では，その割り当てたサイズをMPI_MAX_OBJECT_NAMEの値として使用する必要がある．ヒープ領域から名前の領域を割り当てる実装では，MPI_COMM_GET_NAMEの呼び出し時に最大でこのサイズの文字列用の領域をユーザが割り当てる必要があるため，MPI_MAX_OBJECT_NAMEとして比較的小さい値を定義する必要がある．（**実装者へのアドバイス終わり**）

MPI_COMM_GET_NAME (comm, comm_name, resultlen)

IN	comm	名前を返す対象となるコミュニケータ（ハンドル）
OUT	comm_name	前にコミュニケータに格納されていた名前，あるいはそのような名前がない場合は空の文字列（文字列）
OUT	resultlen	返された名前の長さ（整数型）

```
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
```

```
    INTEGER COMM, RESULTLEN, IERROR
    CHARACTER*(*) COMM_NAME
{void MPI::Comm::Get_name(char* comm_name, int& resultlen) const （廃止された
              呼び出し形式, 第15.2節を参照）}
```

MPI_COMM_GET_NAMEは，指定のコミュニケータと関連付けられていた最後の名前を返す．名前の設定と取得は，任意の言語について行える．使用する言語に関係なく，同じ名前が返される．nameはMPI_MAX_OBJECT_NAME文字の長さの作成済み文字列を格納できるように割り当てる必要がある．MPI_COMM_GET_NAMEは設定された名前のコピーをnameに返す．

C言語ではさらに，name[resultlen]の位置にnull文字が格納される．resultlenはMPI_MAX_OBJECT_NAME-1を超えることはない．Fortran言語では，名前の右側が空白文字で埋められる．resultlenがMPI_MAX_OBJECT_NAMEを超えないようにする必要がある．

ユーザが名前をコミュニケータと関連付けていない場合，またはエラーが発生した場合，MPI_COMM_GET_NAMEは空の文字列を返す（Fortran言語の場合は全てが空白，C言語およびC++言語の場合は""）．定義済みの3つのコミュニケータでは，定義済みの名前が関連付けられている．そのため，MPI_COMM_WORLD，MPI_COMM_SELFの名前と，MPI_COMM_GET_PARENTによって返されたコミュニケータの名前（MPI_COMM_NULLでない場合）は，それぞれデフォルトがMPI_COMM_WORLD，MPI_COMM_SELF，MPI_COMM_PARENTとなる．システムがコミュニケータにデフォルト名を使用するよう設定した場合でも，同じコミュニケータに対してユーザが名前を設定することができる．この場合，古い名前が削除され，新しい名前が割り当てられる．

根拠 単に定義済みの属性キーを提供する代わりに，コミュニケータの名前を設定および取得するための独立した関数を用意している．理由は以下のとおりである．

- 一般的に，Fortran言語では文字列を属性として格納することができない．
- ヒープ領域から割り当てられていることが分かっている場合を除いて，文字列の属性のための削除関数をセットアップするのが容易ではない．
- 属性キーを有益にするには`strdup`呼び出しの追加のコードが必要となる．これが標準化されていない場合，ユーザが記述する必要がある．これは非常に無駄な作業で，容易に削減することができる．
- Fortran言語の呼び出し形式は記述が単純ではなく（Fortran言語のコンパイルシステムの詳細によって異なる），可搬でもない．そのため，ユーザコードではなく，ライブラリで管理する必要がある．

（**根拠の終わり**）

ユーザへのアドバイス 上記の定義は，MPI_COMM_GET_NAMEによって返された文字列は名前がない場合でも常に有効な文字列であるため，そのまま出力しても安全であることを意味している．

名前とコミュニケータの関連付けは，MPIプログラムの意味には影響を及ぼさないが，名前の保存のためにプログラムの格納要件が（必ず）増加する．そのため，名前とコミュニケータの関連付けのために必ずこれらの関数を使用しなければならないというわけではない．

しかし，名前とコミュニケータを関連付けておけば，デバッガやプロファイラで表示される情報の内容がわかりやすくなるため，MPIアプリケーションのデバッグやプロファイリングがしやすくなる． （**ユーザへのアドバイス終わり**）

データ型の名前の設定と取得に使用される関数を以下に示す．

MPI_TYPE_SET_NAME (type, type_name)

INOUT	type	識別子を設定する対象となるデータ型（ハンドル）
IN	type_name	名前として記憶される文字列（文字列）

```
int MPI_Type_set_name(MPI_Datatype type, char *type_name)
MPI_TYPE_SET_NAME(TYPE, TYPE_NAME, IERROR)
    INTEGER TYPE, IERROR
    CHARACTER*(*) TYPE_NAME
{void MPI::Datatype::Set_name(const char* type_name) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_TYPE_GET_NAME (type, type_name, resultlen)

IN	type	名前を返す対象となるデータ型（ハンドル）
OUT	type_name	前にデータ型に格納されていた名前，あるいはそのような名前がない場合は空の文字列（文字列）
OUT	resultlen	返された名前の長さ（整数型）

```
int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen)
MPI_TYPE_GET_NAME(TYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER TYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME
{void MPI::Datatype::Get_name(char* type_name, int& resultlen) const （廃止された呼び出し形式, 第15.2節を参照）}
```

定義済みの名前付きデータ型にはデータ型の名前のデフォルト名がある．例えば，MPI_WCHARにはMPI_WCHARというデフォルト名がある．

ウィンドウの名前の設定と取得に使用される関数を以下に示す．

MPI_WIN_SET_NAME (win, win_name)

INOUT	win	識別子を設定する対象となるウィンドウ（ハンドル）
IN	win_name	名前として記憶される文字列（文字列）

```
int MPI_Win_set_name(MPI_Win win, char *win_name)
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
```

```
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
{void MPI::Win::Set_name(const char* win_name) （廃止された呼び出し形式, 第15.2節を
              参照）}
```

MPI_WIN_GET_NAME (win, win_name, resultlen)

IN	win	名前を返す対象となるウィンドウ（ハンドル）
OUT	win_name	前にウィンドウに格納されていた名前，あるいはそのような名前がない場合は空の文字列（文字列）
OUT	resultlen	返された名前の長さ（整数型）

```
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
    CHARACTER*(*) WIN_NAME
{void MPI::Win::Get_name(char* win_name, int& resultlen) const （廃止された呼
              び出し形式, 第15.2節を参照）}
```

6.9 ゆるい同期モデルの形式化

この節では，特にグループ内通信に重点を置いて，ゆるい同期モデルについて詳しく説明する．

6.9.1 基本説明

呼び出し側がコミュニケータ（コンテクストとグループを含む）を呼び出される側に渡す時，サブプログラムの実行によってそのコミュニケータが副作用の影響を受けることのないようにしなくてはならない．つまり，プロセスを必要とする可能性のあるコミュニケータ上ではアクティブな操作があってはならない．このモデルに従ってライブラリを記述すれば，「安全」な動作が実現される．このように指定されたライブラリにより，呼び出される側は，コミュニケータを介する通信をしていればどのような通信も他の通信に邪魔されることはない．したがって，（コミュニケータ上にあらかじめ割り当てられたコンテクストによるような場合でも）同期なしで新しくコミュニケータを作成するような高度な実装が可能になり，これによって重大なオーバーヘッドが課されなくなる．

このような形態によってもたらされる安全は，例えば，ライブラリルーチンへのディスクリプタの配列の受け渡しのような，一般の計算機科学で利用されているものと同様である．ライブラリルーチンは，正当でかつ変更可能なディスクリプタのようなものとして考えられている．

6.9.2 実行モデル

ゆるい同期モデルでは，実行中のプロセスがそれぞれの手続きを呼び出し，**並列手続き**へ制御をうつすことで効率をあげることができる．この呼び出しは集団操作である．

それは実行グループの全てのプロセスで実行され，呼び出しは全てのプロセスでほぼ同じ順に実行される．しかし，呼び出しは同期させる必要がない．

並列手続きがあるプロセス中で**アクティブ**であるとは，そのプロセスが 手続きを集団的に実行しているグループに属しており，グループのメンバのいくつかが現在手続きのコードを実行中であるということである．並列手続きがあるプロセス中でアクティブならば，たとえ現在，この手続きのコードを実行していない場合でも，このプロセスはこの手続きに関するメッセージ受けとることができる．

コミュニケータの静的割り当て

任意の時点において，どのプロセスにおいてもアクティブな並列手続きが1つまでであり，その手続きを実行しているプロセスのグループが固定されているような場合には，コミュニケータを静的にアロケートすることができる．例えば，並列手続きの全ての呼び出しに全プロセスが関与し，プロセスがみなシングルスレッドであり，かつ再帰的な並列手続きの呼び出しがないような場合が挙げられる．

そのような場合には，コミュニケータはそれぞれの手続きへ静的に割り当てることができる．つまり静的な割り当てが，初期化のコードの部分であらかじめ可能になる． 各ライブラリの中の一つの手続きしか各プロセッサ上で並行にアクティブにならないよう，並列の手続きをライブラリにまとめることができるなら，一つのライブラリあたり一つのコミュニケータを割り当てれば十分である．

コミュニケータの動的割り当て

新しい並列手続きが同一の並列手続きを実行するグループの一部から常に呼び出される場合には，並列手続きの呼び出しは深くネストしたものとなる．したがって，同一の並列手続きを実行するプロセスは，同一の実行スタックを持つ．

そのような場合には，それぞれの新しい並列手続きごとに，新しいコミュニケータが動的に割り当てられなければならない．この割り当ては呼び出し側で行われる．新しいコミュニケータは，呼び出される側の実行グループが呼び出し側の実行グループと同じである場合はMPI_COMM_DUPを呼び出すことで作成され，呼び出し側の実行グループが個々の並列ルーチンを実行するいくつかのサブグループに分解される場合には，MPI_COMM_SPLITを呼び出すことで作成される．新しいコミュニケータは，呼び出されるルーチンへ引数として渡される．

それぞれの呼び出しでは，新しいコミュニケータの作成を減らすか，ある場合には避けることもできる．例えば，もしも実行グループが分割されない場合には，コミュニケータのスタックをあらかじめ割り当てておき，それを再帰呼び出しのスタックとみなし，再利用するようにスタックを管理することができる．

呼び出し側と呼び出される側で同一のコミュニケータを利用した場合でも，通信が順序付けられているという特性を利用することで，呼び出し側と呼び出される側の通信の混乱を避けることができる．そのような場合には次の2つの規則を守る必要がある．

- 手続き呼び出しの前（あるいは手続きから戻る前）に送ったメッセージは，その手続きの呼び出し（あるいはリターン）の前に受信側で受信を終了すること．
- メッセージは常に送信元によって選択されること（MPI_ANY_SOURCEによって作成されたものは使用しない）．

一般的な場合

一般に，同一グループ中で同じ並列手続きを複数並行してアクティブに呼び出す場合があり，ここでは呼び出しのネストは少なくなる．このとき，新しいコミュニケータをそれぞれの呼び出しで作成する必要がある．重なりのあるプロセスの集合上で2つの異った並列手続きが並行して呼び出された場合にコミュニケータの作成を適切に調整することは，ユーザの責任である．

第7章

プロセストポロジー

7.1 はじめに

本章ではMPIのトポロジーのメカニズムについて解説する．トポロジーとはグループ内コミュニケータに与えることができる特別なオプション属性であり，グループ間コミュニケータには追加できない．トポロジーは（コミュニケータ内で）グループをなすプロセス群に便利な命名メカニズムを提供し，さらに，ランタイムシステムがプロセス群をハードウェアにマッピングするのを補助することもある．

第6章で説明したように，MPIのプロセスグループをn個のプロセスの集合とすると，グループ内の各プロセスには0からn-1までのランクが割り当てられる．多くの並列アプリケーションでは，プロセス群への線形なランク付けは，プロセスの論理的通信パターン（これは通常，もとになる問題の幾何学的配置と利用する数値アルゴリズムによって決定される）を十分には反映していない．多くの場合，プロセス群は2次元あるいは3次元格子のようなトポロジー的なパターンに配置される．より一般的には，論理的プロセス配置はグラフで表現される．この章では，この論理的プロセス配置を「仮想トポロジー」と呼ぶことにする．

仮想プロセストポロジーは，もとになる物理的ハードウェアのトポロジーとはまったく異なる．与えられたマシン上での通信性能を改善させるような場合には，システムは仮想トポロジーを利用して，プロセス群を物理的プロセッサに割り当ててもよい．しかし，このマッピングをどのように行うかということについては，MPIの対象外である．他方で，仮想トポロジーを記述することは，アプリケーションにのみ依存し，マシンとは独立している．この章で記載されている関数とは，マシンとは独立したマッピングのみを扱う関数である．

> **根拠** 物理的なマッピングについては取り上げないが，ランタイムシステムは仮想トポロジーに関する情報があれば，これはアドバイスとなり得る．格子／トーラス構造をハイパーキューブや格子などのハードウェアトポロジーにマッピングする手法は既知である．より複雑なグラフ構造については，適切な発見的手法により近最適解が得られることがある[32]．他方，ユーザが論理的プロセス配置を「仮想トポロジー」として指定する方法がなければ，ランダムマッピングとなる可能性が高

い．これは，一部のマシンの相互接続ネットワーク内で不必要な競合を引き起こす．最近のワームホールルーティングアーキテクチャ上での適切なプロセス－プロセッサ間マッピングから得られた，性能改善の予測値および測定値に関する詳細が文献[10, 11]に示されている．

性能面の利点の他に，仮想トポロジーは便利なプロセス名前構造として機能し，その結果，メッセージ通信プログラムの読み易さや記述力に多くの利点をもたらすことができる． （**根拠の終わり**）

7.2 仮想トポロジー

プロセス集合における通信パターンはグラフで表現できる．ノードでプロセスを表し，相互に通信するプロセスをエッジで接続する．MPIはグループ内のプロセスの任意の対でのメッセージ通信を提供する．チャネルを明示的に開く必要はない．したがって，ユーザが定義するプロセスグラフに「ミッシングリンク」があっても，対応するプロセス間でメッセージを交換できないということはない．これはむしろ，この接続が仮想トポロジーでは無視されるということを意味する．この戦略は，このトポロジーではこの通信の経路を指定する有効な方法がないということを示唆している．他にも，自動マッピングツールが（これがランタイ環境に存在する場合）マッピングの際にこのエッジを考慮しないということも考えられる．

どのようなアプリケーションでも，仮想トポロジーをグラフによって指定するだけで十分である．しかし，多くのアプリケーションではグラフ構造は規則的であり，グラフを詳細にセットアップすることはユーザにとっては不便なもので，実行時に効率を落とすことも考えられる．並列アプリケーションの大部分はリング，2次あるいは高次元の格子，またはトーラスのようなプロセストポロジーを使用する．これらは各座標方向におけるプロセス数と次元数により完全に定義される．また一般的に，格子とトーラスのマッピングは一般グラフのマッピングよりは簡単な問題である．そこでこれらには系統的に対処するのが望ましい．

カルテシアン構造におけるプロセス座標には0から番号が振られる．カルテシアン構造を持つプロセス群には常に行優先番号付けを使用する．このことは，例えば(2×2)格子における4つのプロセスのグループランクと座標には次の関係があることを意味する．

```
coord (0,0):  rank 0
coord (0,1):  rank 1
coord (1,0):  rank 2
coord (1,1):  rank 3
```

7.3 MPIへの埋め込み

この章で定義されている仮想トポロジーを支援する機能は，MPIの他の部分と整合がとれており，また可能な限り，他の箇所で定義された関数を利用している．トポロジー

情報はコミュニケータに付加される．この情報は第6章で説明したキャッシングメカニズムを用いてコミュニケータに追加される．

7.4 関数の概要

関数MPI_GRAPH_CREATE， MPI_DIST_GRAPH_CREATE_ADJACENT，MPI_DIST_GRAPH_CREATE，MPI_CART_CREATEは，一般的な（グラフ）仮想トポロジーおよびカルテシアントポロジーを生成するのに使う．これらのトポロジー生成関数は集団的である．他の集団呼び出しと同様，呼び出しが同期するしないに関わらず，正しく動作するようにプログラムを書かなければならない．

トポロジー生成関数は入力として既存のコミュニケータcomm_oldをとる．これはトポロジーをマッピングするプロセスの集合を定義している．MPI_GRAPH_CREATEおよびMPI_CART_CREATEの場合，全ての入力引数はcomm_oldのグループの全てのプロセスの値と同じでなければならない．MPI_DIST_GRAPH_CREATE_ADJACENTおよびMPI_DIST_GRAPH_CREATEの場合，入力された通信グラフは呼び出しプロセスに分配される．そのため，各プロセスは引数にそれぞれ異なるグラフを示す値を渡す．しかし，全てのプロセスはreorderおよびinfo引数には同じ値を渡す必要がある．いずれの場合も，トポロジー構造をキャッシュ情報に持つ新規コミュニケータcomm_topolが生成される（第6章を参照）．関数MPI_COMM_CREATEの場合と同様に，comm_oldから伝播されたキャッシュ情報はcomm_topolへは伝播されない．

MPI_CART_CREATEを使用すると，任意の次元のカルテシアン構造が記述できる．この関数では，各座標方向についてプロセス構造が周期的か否かを指定する．n次元ハイパーキューブは各座標方向につき2個のプロセスを持つn次元トーラスであることに注意すること．したがって，ハイパーキューブ構造に対する特別なサポートは不要である．ローカルな補助関数MPI_DIMS_CREATEを使用すると，与えられた次元数に対して，バランスのとれたプロセス配置を計算することができる．

> **根拠** EXPRESS[12]とPARMACSにも同様の関数が含まれている． （**根拠の終わり**）

関数MPI_TOPO_TESTを使用すると，コミュニケータに付加されたトポロジーについて問い合わせできる．トポロジー情報は，一般グラフについては関数MPI_GRAPHDIMS_GETおよびMPI_GRAPH_GETを使用して，カルテシアントポロジーについてはMPI_CARTDIM_GETおよびMPI_CART_GETを使用して，コミュニケータから抽出できる．カルテシアントポロジーを操作するために，いくつかの追加関数が用意されている．関数MPI_CART_RANKおよびMPI_CART_COORDSは，カルテシアン座標をグループランクへ，逆にグループランクをカルテシアン座標へ変換する．関数MPI_CART_SUBを使用すると，（MPI_COMM_SPLITと同様に）部分カルテシアン領域を抽出できる．関数MPI_CART_SHIFTはプロセスが1つのカルテシアン次元におけるその隣接と通信するために必要な情報を提供する．2つの関数MPI_GRAPH_NEIGHBORS_COUNTおよびMPI_GRAPH_NEIGHBORSを使用すると，グ

ラフの中のノードの隣接を抽出できる．分散グラフの場合，関数MPI_DIST_NEIGHBORS_COUNTおよびMPI_DIST_NEIGHBORSを使用して呼び出しノードの隣接を抽出できる．関数MPI_CART_SUBは，入力コミュニケータのグループ上で集団的である．他の全ての関数はローカルである．

2つの追加関数MPI_GRAPH_MAPおよびMPI_CART_MAPについては最後の節で紹介する．一般にこれらの関数はユーザが直接呼び出す関数ではない．しかし，第6章で説明したコミュニケータ操作関数と一緒に使用すれば，他の全てのトポロジー関数の実装には十分である．このような実装については第7.5.8節で概説する．

7.5 トポロジーコンストラクタ

7.5.1 カルテシアンコンストラクタ

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN	comm_old	入力コミュニケータ（ハンドル）
IN	ndims	カルテシアン格子の次元数 (整数型)
IN	dims	各次元毎のプロセス数を指定するndims個の要素を持つ整数配列
IN	periods	ndimsのサイズを持つ論理型配列．この配列はそれぞれの次元が周期的（true）か非周期的（false）を示す．
IN	reorder	ランク番号を変更してよい (true) か否か（false）（論理型）
OUT	comm_cart	新しいカルテシアントポロジーを持つコミュニケータ（ハンドル）

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
int reorder, MPI_Comm *comm_cart)
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
{MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],
             const bool periods[], bool reorder) const （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_CART_CREATEは，カルテシアントポロジー情報を付加した新しいコミュニケータのハンドルを返す．reorder = falseであれば，新規グループの中の各プロセスのランクは旧グループでのそれと同じである．そうでない場合は，（仮想トポロジーを物理的マシンへ有効に埋め込むことを選択できるように）この関数はプロセスを並べ替える場合がある．カルテシアン格子の総サイズがグループcomm_old[1]のサイズよりも小さければ，MPI_COMM_SPLITの場合と同様に，いくつかのプロセスにはMPI_COMM_NULLを返す．ndimsが0の場合，0次元のカルテシアントポロジーが生成される．この関数がグループサ

[1]訳者註：MPI-2.2ではcommとなっているが，これはcomm_oldの誤りであり，MPI-3では修正されている．

イズよりも大きな格子を指定した場合，またはndimsが負の値の場合，この呼び出しは誤りとなる．

7.5.2 カルテシアン支援関数: MPI_DIMS_CREATE

カルテシアントポロジーでは関数MPI_DIMS_CREATEを使用して，バランスをとるべきグループ内のプロセスの個数と，ユーザが指定できる任意の制約に応じて，座標軸毎にバランスのとれたプロセス配置を選ぶことができる．これには（グループMPI_COMM_WORLDのサイズの）全てのプロセスをn次元トポロジーに分割するという用途がある．

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN	nnodes	格子内のノード数（整数型）
IN	ndims	カルテシアントポロジーの次元数（整数）
INOUT	dims	各次元に対してそのノード数を指定したサイズ ndimsの整数配列

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
{void MPI::Compute_dims(int nnodes, int ndims, int dims[]) （廃止された呼び出し
                形式, 第15.2節を参照）}
```

配列dimsのエントリは，ndims次元でかつ全部でnnodes個のノードのカルテシアン格子を記述するように設定される．次元は適切な分割アルゴリズムを使用して，できるだけ互いに近い値になるように設定される．呼び出し側はさらに配列dimsの要素を指定することで，このルーチンの操作を制限することができる．dims[i]が正数に設定されている場合，このルーチンは次元iのノードの個数を変更しない．dims[i] = 0となっているエントリのみが，このルーチンの呼び出しにより変更される．

dims[i]の入力値が負の場合は誤りとなる．nnodesが $\prod_{i,dims[i]\neq 0} dims[i]$ の倍数でない場合も誤りとなる．

呼び出しにより設定されたdims[i]は非増加順に並べられる．配列dimsはルーチンMPI_CART_CREATEへの入力として使用するのに適している．MPI_DIMS_CREATEはローカルである．

例 7.1

呼び出し前の dims	関数呼び出し	戻った後の dims
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	呼び出しエラー

7.5.3 一般（グラフ）コンストラクタ

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

IN	comm_old	入力コミュニケータ（ハンドル）
IN	nnodes	グラフにおけるノードの個数（整数型）
IN	index	ノードの次数を表す整数配列（下記参照）
IN	edges	グラフのエッジを表わす整数配列（下記参照）
IN	reorder	ランク番号を変更してよい（true）か否か（false)（論理型）
OUT	comm_graph	グラフトポロジーを付け加えたコミュニケータ（ハンドル）

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
int reorder, MPI_Comm *comm_graph)
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER
{MPI::Graphcomm MPI::Intracomm::Create_graph(int nnodes, const int index[],
             const int edges[], bool reorder) const （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_GRAPH_CREATEは，グラフトポロジー情報が付加された新しいコミュニケータのハンドルを返す．reorder = falseであれば，新規グループの中の各プロセスのランクは旧グループでのそれと同一である．そうでない場合には，この関数はプロセスを並べ替える場合がある．グラフのサイズnnodesがcomm_old[2]のグループのサイズよりも小さければ，MPI_CART_CREATEおよびMPI_COMM_SPLITと同様に，いくつかのプロセスにはMPI_COMM_NULLを返す．グラフが空，つまりnnodes == 0の場合，全てのプロセスでMPI_COMM_NULLが返される．この呼び出しを入力コミュニケータのグループサイズよりも大きなグラフを指定して呼び出すと，誤りとなる．

3つのパラメータnnodes，index，edgesでグラフ構造を定義する．nnodesはグラフのノードの個数である．ノードには0からnnodes-1までの番号が付けられる．配列index のi番目のエントリには，最初のi個のグラフのノードの隣接の総数が格納される．ノード0，1，...，nnodes-1の隣接のリストは，配列edgesの中の連続した位置に格納される．配列edgesはエッジリストを平坦化した表現である．indexのエントリの総数はnnodesで，edgesのエントリの総数はグラフのエッジの本数に等しい．

引数nnodes，index，edgesの定義については，以下の簡単な例で説明する．

例 7.2 以下の隣接行列を持つ4個のプロセス0，1，2，3があるとする．

[2]訳者註：MPI-2.2ではcommとなっているが，これはcomm_oldの誤りであり，MPI-3では修正されている．

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

この場合，入力引数は次の通りである．

nnodes = 4
index = 2, 3, 4, 6
edges = 1, 3, 0, 3, 0, 2

したがってC言語では，index[0]はノード0の次数であり，index[i] - index[i-1]はノードi，i=1，...，nnodes-1の次数である．ノード0の隣接リストは0 ≤ j ≤ index[0]−1についてedges[j]に格納され，ノードiの隣接リストはindex[i−1] ≤ j ≤ index[i]−1について edges[j]に格納される．

Fortran言語では，index(1)はノード0の次数であり，index(i+1) - index(i)はノードi，i=1，...，nnodes-1の次数である．ノード0 の隣接リストは1 ≤ j ≤ index(1)についてedges(j)に格納され，ノードi，i > 0の隣接リストはindex(i) + 1 ≤ j ≤ index(i + 1)について edges(j)に格納される．

プロセスの隣接リスト内で1つのプロセスを複数回定義することができる（つまり，2つのプロセスの間に複数のエッジが存在する可能性がある）．また，プロセスはそれ自体の隣接とすることもできる（つまり，グラフ内の自己参照ループ）．隣接行列は非対称とすることができる．

ユーザへのアドバイス 複数のエッジまたは非対称の隣接行列を使用した場合の性能の詳細は定義されていない．ノード隣接エッジの定義では通信の方向は示されていない． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 以下のトポロジー情報がコミュニケータに格納されるであろう．

- トポロジーのタイプ（カルテシアン／グラフ），
- カルテシアントポロジーの場合
  1. ndims （次元数）
  2. dims （各座標軸におけるプロセスの個数）
  3. periods （周期情報）
  4. own_position （格子内での自位置，rankとdimsから計算で求めることも可能）
- グラフトポロジーの場合
  1. index,
  2. edges,

これらはグラフ構造を定義するベクトルである．

グラフ構造については，ノードの個数はグループ内のプロセスの個数に等しい．したがって，ノードの個数は明示的に格納しなくてもよい．配列 indexの最初に0を挿入すると，トポロジー情報へのアクセスが簡単になる． （実装者へのアドバイス終わり）

7.5.4 分散（グラフ）コンストラクタ

一般的なグラフコンストラクタでは，各プロセスが十分な（大域的な）通信グラフを呼び出しに渡すことが前提となる．これによりこのコンストラクタのスケーラビリティが制限される．分散グラフインターフェイスを使用することにより，十分に分散した形で通信グラフが指定される．各プロセスで指定されるのは，認識している通信グラフの一部のみである．通常，これで考えられるのは，プロセスが最終的にデータを受信または取得する元となるプロセスの集合，あるいはプロセスがデータを送信または配置する先となるプロセスの集合，あるいはこのようなエッジの組み合わせである．分散グラフトポロジーを作成するために，2種類のインターフェイスを使用することができる．MPI_DIST_GRAPH_CREATE_ADJACENTでは，各プロセスの論理通信グラフ内で全ての入力および出力（隣接）エッジを指定して分散グラフコミュニケータを作成するため，作成中に最小限の通信が必要となる．MPI_DIST_GRAPH_CREATEは柔軟性に優れていて，プロセスは通信が他のプロセス対で発生することを示すことができる．

MPIライブラリによる最適化の可能性を促進するため，分散グラフコンストラクタでは重み付け通信エッジを使用したり，プロセスの並べ替えやMPIライブラリによる最適化にさらに影響を及ぼしうるinfo引数を使用したりすることができる．例えば，エッジの重み付けの解釈方法，並べ替えの質，グラフ処理のためにMPIライブラリに許容される時間などに関するヒントが得られる．

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

IN	comm_old	入力コミュニケータ（ハンドル）
IN	indegree	sourcesおよびsourceweights配列のサイズ（非負の整数型）
IN	sources	呼び出しプロセスを送信先とするプロセスのランク（非負の整数の配列）
IN	sourceweights	呼び出しプロセスへのエッジの重み付け（非負の整数の配列）
IN	outdegree	destinationsおよびdestweights配列のサイズ（非負の整数型）
IN	destinations	呼び出しプロセスを送信元とするプロセスのランク（非負の整数の配列）
IN	destweights	呼び出しプロセスからのエッジの重み付け（非負の整数の配列）
IN	info	重み付けの最適化と解釈のヒント（ハンドル）
IN	reorder	ランク付けを変更してよい(true)か否か (false)（論理型））
OUT	comm_dist_graph	分散グラフトポロジーとのコミュニケータ（ハンドル）

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
int sources[], int sourceweights[], int outdegree, int destinations[], int
destweights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, COMM_DIST_GRAPH,
IERROR)
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create_adjacent(int
             indegree, const int sources[], const int sourceweights[],
             int outdegree, const int destinations[],
             const int destweights[], const MPI::Info& info, bool reorder)
             const （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Distgraphcomm
             MPI::Intracomm::Dist_graph_create_adjacent(int indegree,
             const int sources[], int outdegree, const int destinations[],
             const MPI::Info& info, bool reorder) const （廃止された呼び出し形
             式, 第15.2節を参照）}
```

MPI_DIST_GRAPH_CREATE_ADJACENTは分散グラフトポロジー情報が付加された新規コミュニケータへのハンドルを返す．各プロセスはエッジに関する全ての情報を仮想分散グラフトポロジー内の隣接に渡す．呼び出しプロセスでは，送信元プロセスと送信先プロセスで同じ重み付けによりグラフの各エッジが記述されるようにする必要がある．指定の(source,dest)ペアに対して複数のエッジがある場合，これらのエッジの重み付けの順序は問題にならない．完全な通信トポロジーはcomm_oldの全てのプロセスのsources配列で示される全てのエッジの組み合わせであり，これはdestinations配列で示される全てのエッジの組み合わせと同じでなければならない．送信元と送信先のランク

はcomm_oldのプロセスのランクでなければならない．これにより，通信グラフの指定を完全に分散することができる．孤立プロセス（出力または入力エッジのないプロセス，つまりindegreeおよびoutdegreeが0として指定されているため，グラフ指定で送信元または送信先のランクとして機能しないプロセス）とすることもできる．

この呼び出しでは，トポロジー情報が付加されている分散グラフトポロジーの型の新規コミュニケータcomm_dist_graphが生成される．comm_dist_graph内のプロセスの数はcomm_old内のプロセスの数と同じである．MPI_DIST_GRAPH_CREATE_ADJACENTの呼び出しは集団的である．

重み付けは非負の整数型として指定され，プロセスの再マッピング戦略やその他のMPI内部の最適化に影響を及ぼすために使用できる．例えば，特定のエッジと一緒にその後の通信呼び出しの概数の引数をエッジの重み付けとして使用することができる．同様に，エッジの多重度を利用することで，プロセスのペア間の通信を強化するすることができる．しかし，エッジの重み付けの正確な意味はMPI標準では規定されておらず，実装に任されている．C言語または Fortran言語では，全てのエッジが同じ重みを持つ（事実上は重み付けなしと同じ）ことを示す重み付けの配列用の特別な値MPI_UNWEIGHTEDをアプリケーションで供給することができる．C++言語ではこの定数は存在せず，重み付けの引数を引数リストから除外することができる．comm_oldの全部ではなく一部のプロセスでMPI_UNWEIGHTEDを供給したり， C++言語で重み付けの配列を除外したりすると，誤りとなる． MPI_UNWEIGHTEDは特別な重み付けの値ではなく，配列全体の引数のための特別な値であることに注意すること．C言語では，NULLと同等となる．Fortran言語では，MPI_UNWEIGHTEDはMPI_BOTTOMのようなオブジェクト（初期化や割り当てには使用できない）である． 第2.5.4節を参照すること．

infoおよびreorder引数の意味については，以下のルーチンの説明で定義する．

MPI_DIST_GRAPH_CREATE(comm_old, n, sources, degrees, destinations, weights, info, reorder, comm_dist_graph)

IN	comm_old	入力コミュニケータ（ハンドル）
IN	n	このプロセスがエッジを指定する対象となる送信元ノードの数（非負の整数型）
IN	sources	このプロセスがエッジを指定する対象となるn個の送信元ノードが記述された配列（非負の整数の配列）
IN	degrees	送信元ノード配列内の各送信元ノードのための送信先の数を指定する配列（非負の整数の配列）
IN	destinations	送信元ノード配列内の送信元ノードのための送信先ノード（非負の整数の配列）
IN	weights	送信先エッジに対する送信元のための重み付け（非負の整数の配列）
IN	info	重み付けの最適化と解釈に関するヒント（ハンドル）
IN	reorder	プロセス順序を変更してよい (true) か否か(false)（論理型）
OUT	comm_dist_graph	分散グラフトポロジーが付加されたコミュニケータ（ハンドル）

```
int MPI_Dist_graph_create(MPI_Comm comm_old, int n, int sources[],
int degrees[], int destinations[], int weights[], MPI_Info info,
int reorder, MPI_Comm *comm_dist_graph)
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
              const int sources[], const int degrees[], const int
              destinations[], const int weights[], const MPI::Info& info,
              bool reorder) const （廃止された呼び出し形式, 第15.2節を参照）}
{MPI::Distgraphcomm MPI::Intracomm::Dist_graph_create(int n,
              const int sources[], const int degrees[],
              const int destinations[], const MPI::Info& info, bool reorder)
              const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_DIST_GRAPH_CREATEは分散グラフトポロジー情報が付加された新規コミュニケータへのハンドルを返す．具体的には，以下のように各プロセスが有向(source,destination)通信エッジの集合を指定して コンストラクタを呼び出す．各プロセスはsources配列内のn個の送信元ノードの配列を渡す．各送信元ノードについて，送信先ノードの非負の数がdegrees配列で指定される．送信先ノードはdestinations配列の該当する連続のセグメントに格納される．正確に言うと，sources内のi番目のノードがsの場合，destinations[degrees[0]+...+degrees[i-1]+j]に格納されたj番目のエッジのdにより，degrees[i]のエッジ(s,d)が指定される．このエッジの重み付けはweights[degrees[0]+...+degrees[i-1]+j]に格納される．sourcesおよびdestinationsの両方の配列には同じノードを複数回含むことができ，送信先または送信元のノードはどの順序で

記述しても影響しない．同様に，異なるプロセスで同じ送信元および送信先ノードによりエッジを指定することもできる．送信元および送信先ノードはcomm_oldのプロセスランクでなければならない．プロセスごとに，指定する送信元ノードおよび送信先ノードの数を変えることもできるし，送信元／送信先のエッジを変えることもできる．これにより，通信グラフの指定を完全に分散させることができる．孤立プロセス（出力または入力エッジのないプロセス，つまり，グラフ指定で送信元または送信先ノードとして機能しないプロセス）とすることもできる．

この呼び出しは，トポロジー情報が付加されている分散グラフトポロジーの型の新規コミュニケータcomm_dist_graphを生成する．comm_dist_graph内のプロセスの数はcomm_old内のプロセスの数と同じである．MPI_Dist_graph_createの呼び出しは集団的である．

reorder = falseの場合，comm_dist_graph内の全てのプロセスのランクはcomm_old内のものと同じになる．reorder = trueの場合，通信グラフのエッジでの通信を向上させるため，MPIライブラリを他の（comm_oldの）プロセスに自由に再マッピングすることができる．各エッジに関連付けられた重み付けはそのエッジでの通信の量と強弱に関するMPIライブラリのヒントで，最適な並べ替えを計算するのに使用できる．

重み付けは非負の整数型として指定され，プロセスの再マッピング戦略やその他のMPI内部の最適化に影響を及ぼすために使用できる．例えば，特定のエッジと一緒にその後の通信呼び出しの概数の引数をエッジの重み付けとして使用することができる．同様に，エッジの多重度を利用することで，プロセスのペア間の通信を強化するすることができる．しかし，エッジの重み付けの正確な意味はMPI標準では規定されておらず，実装に任されている．C言語またはFortran言語では，全てのエッジが同じ重みを持つ（事実上は重み付けなしと同じ）ことを示す重み付けの配列用の特別な値MPI_UNWEIGHTEDをアプリケーションで供給することができる．C++言語ではこの定数は存在せず，重み付けの引数を引数リストから除外することができる．comm_old の全部ではなく一部のプロセスでMPI_UNWEIGHTEDを供給したり，C++言語で重み付けの配列を除外したりすると，誤りとなる． MPI_UNWEIGHTEDは特別な重み付けの値ではなく，配列全体の引数のための特別な値であることに注意すること．C++言語では，nullと同等となる．Fortran言語では，MPI_UNWEIGHTEDはMPI_BOTTOMのようなオブジェクト（初期化や割り当てには使用できない）である．第2.5.4節を参照すること．

weights引数の意味はinfo引数の影響を受ける可能性がある．Info引数はマッピングをガイドするのに使用できる．例えば，異なる SMPノード上のプロセス間でエッジの最大数を最小限に抑えたり，このようなエッジの合計数を最小限に抑えたりできる．MPI実装は特定のヒントに従う必要はなく，並べ替えを行わないMPI実装において有効である．MPI実装では，より多くのinfoのキー値のペアを指定することができる．すべてのプロセスで，同じinfoのキーと値のペアの集合を指定する必要がある．

> **実装者へのアドバイス** MPI実装では，追加でサポートするinfoのキーと値のペアを明文化する必要がある．MPI_INFO_NULL は常に有効でデフォルトが分散グラフトポロジーであるとMPIライブラリに示すこともある．

実装では明示的に分散したトポロジーの部品から1つのトポロジーを構成する必要はない．しかし，全てのプロセスが分散の指定を受けた完全なトポロジーを構成し，これをMPI_GRAPH_CREATEの呼び出しで使用してトポロジーを生成することができる．これは機能の参照実装として機能させることができ，小さなコミュニケータで使用することができる．しかし，質の高いスケーラブルな実装ではトポロジーグラフが分散されて保存される．（**実装者へのアドバイス終わり**）

例 7.3 例7.2と同様に，以下の隣接行列とユニットのエッジの重み付けを持つ4個のプロセス0，1，2，3があるとする．

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

MPI_DIST_GRAPH_CREATEを使用して，このグラフをさまざまな方法で構成することができる．例えば，各プロセスで出力エッジを指定することができる．プロセスごとの引数は次のようになる．

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

プロセス0でグラフ全体を渡すこともできる．この場合，プロセスごとの引数は次のようになる．

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	1,3,0,3,0,2	1,1,1,1,1,1
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

上記のどちらの場合も，アプリケーションで明示的に同じ重み付けを指定する代わりに，MPI_UNWEIGHTEDを供給することができる．

このグラフはMPI_DIST_GRAPH_CREATE_ADJACENTに次に示す引数を渡すことで作ることができる．

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

例 7.4 次元と対角線のエッジを使用して全てのプロセスが通信する2次元の $P \times Q$ トーラスの例．カルテシアントポロジーによってモデル化することはできないが，以下のコードに示すようにMPI_DIST_GRAPH_CREATEを使用して容易に取得することができる．この例では，次元を使用した通信の重みは対角線を使用した通信の2倍となる．

```
/*
Input:     dimensions P, Q
Condition: number of processes equal to P*Q; otherwise only
           ranks smaller than P*Q participate
*/
int rank, x, y;
int sources[1], degrees[1];
int destinations[8], weights[8];

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* get x and y dimension */
y=rank/P; x=rank%P;

/* get my communication partners along x dimension */
destinations[0] = P*y+(x+1)%P; weights[0] = 2;
destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;

/* get my communication partners along y dimension */
destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;

/* get my communication partners along diagonals */
destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;

sources[0] = rank;
degrees[0] = 8;
MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
                      weights, MPI_INFO_NULL, 1, comm_dist_graph)
```

7.5.5 トポロジー問い合わせ関数

上記関数のいずれか1つを用いてトポロジーを定義している場合，問い合わせ関数を使用してトポロジー情報を調べることができる．これらは全てローカルな呼び出しである．

MPI_TOPO_TEST(comm, status)

IN	comm	コミュニケータ（ハンドル）
OUT	status	コミュニケータcommの トポロジー型（ステート型）

```
int MPI_Topo_test(MPI_Comm comm, int *status)
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
{int MPI::Comm::Get_topology() const （廃止された呼び出し形式, 第15.2節を参照）}
```

関数MPI_TOPO_TESTはコミュニケータに付加されたトポロジーの型を返す.

出力値statusは次のうちのいずれかである.

MPI_GRAPH	グラフトポロジー
MPI_CART	カルテシアントポロジー
MPI_DIST_GRAPH	分散グラフトポロジー
MPI_UNDEFINED	トポロジーなし

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN	comm	グラフ構造を持つコミュニケータ（ハンドル）
OUT	nnodes	グラフのノードの個数（整数型）（グラフの中のプロセスの数と等しい）
OUT	nedges	グラフのエッジの個数（整数型）

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
{void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const （廃止された
        呼び出し形式, 第15.2節を参照）}
```

関数MPI_GRAPHDIMS_GETおよびMPI_GRAPH_GET はMPI_GRAPH_CREATEによってコミュニケータに付加されたグラフトポロジー情報を検索する関数である.

MPI_GRAPHDIMS_GETから得られる情報を使用すると，MPI_GRAPH_GETの以下の呼び出しで，ベクトルindexおよび edgesのサイズを正確に決めることができる.

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)

IN	comm	グラフ構造を持つコミュニケータ（ハンドル）
IN	maxindex	呼び出し側プログラムのベクトルindex のサイズ（整数型）
IN	maxedges	呼び出し側プログラムのベクトルedges のサイズ（整数型）
OUT	index	グラフ構造を格納した整数配列（詳細はMPI_GRAPH_CREATEの定義を参照）
OUT	edges	グラフ構造を格納した整数配列

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
int *edges)
```

```
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
{void MPI::Graphcomm::Get_topo(int maxindex, int maxedges, int index[],
             int edges[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_CARTDIM_GET(comm, ndims)

IN	comm	カルテシアン構造を持つコミュニケータ（ハンドル）
OUT	ndims	カルテシアン構造の次元数（整数型）

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
{int MPI::Cartcomm::Get_dim() const （廃止された呼び出し形式, 第15.2節を参照）}
```

関数MPI_CARTDIM_GETおよびMPI_CART_GET はMPI_CART_CREATEによってコミュニケータに付加されたカルテシアントポロジー情報を返す．commに0次元のカルテシアントポロジーが付加されている場合，MPI_CARTDIM_GET はndims=0を返し，MPI_CART_GETは全ての出力引数を変更しないで保持する．

MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN	comm	カルテシアン構造を持つコミュニケータ（ハンドル）
IN	maxdims	呼び出し側プログラムのベクトルdims，periods，coordsのサイズ（整数型）
OUT	dims	カルテシアン次元ごとのプロセスの数（整数配列）
OUT	periods	カルテシアン次元のそれぞれについて周期的か否か（true/false）（論理型配列）
OUT	coords	カルテシアン構造中の呼び出しプロセスの座標（整数配列）

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
int *coords)
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
{void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
             int coords[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_CART_RANK(comm, coords, rank)

IN	comm	カルテシアン構造を持つコミュニケータ（ハンドル）
IN	coords	プロセスのカルテシアン座標を指定した（サイズndimsの）配列（整数配列）
OUT	rank	指定したプロセスのランク（整数型）

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR
{int MPI::Cartcomm::Get_cart_rank(const int coords[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

関数MPI_CART_RANKは，カルテシアン構造を持つプロセスグループに対して，論理的プロセス座標を1対1ルーチンが使用するプロセスランクに変換する．

periods(i) = trueとなる次元iについて，座標coords(i) が範囲外である，つまりcoords(i) $<$ 0またはcoords(i) $\geq$ dims(i)であれば，自動的に区間0 $\leq$ coords(i) $<$ dims(i)へシフトされる．非周期的次元の場合には，範囲外の座標を指定すると誤りになる．

commに0次元のカルテシアントポロジーが付加されている場合，coordsは意味を持たず，rankに0が返される．

MPI_CART_COORDS(comm, rank, maxdims, coords)

IN	comm	カルテシアン構造を持つコミュニケータ（ハンドル）
IN	rank	グループcommの中でのプロセスのランク（整数型）
IN	maxdims	呼び出し側プログラムのベクトルcoordsのサイズ（整数型）
OUT	coords	指定したプロセスのカルテシアン座標を格納する（サイズndimsの）配列（整数配列）

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
{void MPI::Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const
（廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_CART_COORDSは，逆のマッピングであるランクから座標への変換を行う．

commに0次元のカルテシアントポロジーが付加されている場合，coordsは変更されない．

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN	comm	グラフトポロジーを持つコミュニケータ（ハンドル）
IN	rank	グループcommの中でのプロセスのランク（整数型）
OUT	nneighbors	指定したプロセスの隣接にあたるプロセスのランク（整数配列）

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
{int MPI::Graphcomm::Get_neighbors_count(int rank) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN	comm	グラフトポロジーを持つコミュニケータ（ハンドル）
IN	rank	グループcommの中でのプロセスのランク（整数型）
IN	maxneighbors	配列neighbors のサイズ（整数型）
OUT	neighbors	指定したプロセスの隣接にあたるプロセスのランク（整数配列）

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
int *neighbors)
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
{void MPI::Graphcomm::Get_neighbors(int rank, int maxneighbors, int
             neighbors[]) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GRAPH_NEIGHBORS_COUNTおよびMPI_GRAPH_NEIGHBORSは一般的なグラフトポロジーの隣接情報を提供する．問い合わされたランクに対して返される隣接の数と配列では，全ての隣接が対象とされ，MPI_GRAPH_CREATEの最初の呼び出しで指定されたのと同じエッジの順序が反映される．特に，MPI_GRAPH_NEIGHBORS_COUNT とMPI_GRAPH_NEIGHBORSは，MPI_GRAPH_CREATEに渡された最初のindexおよびedges配列に基づいて（index[-1]が実質的に0であると仮定して）値を返す．

- MPI_GRAPH_NEIGHBORS_COUNTから返されるcountは(index[rank]- index[rank-1])となる．
- MPI_GRAPH_NEIGHBORSから返されるneighbors配列はedges[index[rank-1]]からedges[index[rank]-1]までとなる．

例 7.5 以下の隣接行列を持つ4個のプロセス0，1，2，3があるとする （一部の隣接は2回以上示される）．

process	neighbors
0	1, 1, 3
1	0, 0
2	3
3	0, 2, 2

そのため，MPI_GRAPH_CREATEの入力引数は以下のようになる．

nnodes = 4
index = 3, 5, 6, 9
edges = 1, 1, 3, 0, 0, 3, 0, 2, 2

したがって，4つの各プロセスに対する呼び出しMPI_GRAPH_NEIGHBORS_COUNTおよびMPI_GRAPH_NEIGHBORSは以下を返す．

Input rank	Count	Neighbors
0	3	1, 1, 3
1	2	0, 0
2	1	3
3	3	0, 2, 2

例 7.6 commはシャッフルエクスチェンジを持つコミュニケータとする．このグループは2^n個のメンバを持つ．各プロセス には$a_1, \ldots, a_n$，ここで$a_i \in \{0,1\}$，というラベルが付けられており，それぞれ3個の隣接を持つ．エクスチェンジとはexchange$(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ $(\bar{a} = 1 - a)$，シャッフルとは shuffle$(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$，アンシャッフルとはunshuffle$(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$であるとする．$n = 3$ではグラフの隣接リストは以下の通りである．

node	exchange neighbors(1)	shuffle neighbors(2)	unshuffle neighbors(3)
0 (000)	1	0	0
1 (001)	0	2	4
2 (010)	3	4	1
3 (011)	2	6	5
4 (100)	5	1	2
5 (101)	4	3	6
6 (110)	7	5	3
7 (111)	6	7	7

コミュニケータcommには上記トポロジーが付加されているとする．以下の部分コードはこの3種類の隣接を順繰りに巡り，それぞれに適切な置換を実行する．

```
C  assume: each process has stored a real number A.
C  extract neighborhood information
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
C  perform exchange permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0,
     +     neighbors(1), 0, comm, status, ierr)
C  perform shuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0,
     +     neighbors(3), 0, comm, status, ierr)
C  perform unshuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0,
     +     neighbors(2), 0, comm, status, ierr)
```

MPI_DIST_GRAPH_NEIGHBORS_COUNTおよびMPI_DIST_GRAPH_NEIGHBORSは分散グラフトポロジーの隣接情報を提供する．

MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)

IN	comm	分散グラフトポロジーを持つコミュニケータ（ハンドル）
OUT	indegree	このプロセスへのエッジの数（非負の整数型）
OUT	outdegree	このプロセスからのエッジの数（非負の整数型）
OUT	weighted	生成中にMPI_UNWEIGHTEDが渡される場合はfalse, それ以外の場合はtrue（論理型）

```
int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
int *outdegree, int *weighted)
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
    INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
    LOGICAL WEIGHTED
{void MPI::Distgraphcomm::Get_dist_neighbors_count(int& indegree,
             int& outdegree, bool& weighted) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_DIST_GRAPH_NEIGHBORS(comm, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights)

IN	comm	分散グラフトポロジーを持つコミュニケータ（ハンドル）
IN	maxindegree	sourcesおよびsourceweights配列のサイズ（非負の整数型）
OUT	sources	呼び出しプロセスが送信先となるプロセス（非負の整数から成る配列）
OUT	sourceweights	呼び出しプロセスへのエッジの重み付け（非負の 整数から成る配列）
IN	maxoutdegree	destinationsおよびdestweights配列のサイズ（非負の整数型）
OUT	destinations	呼び出しプロセスが送信元となるプロセス（非負の整数からなる配列）
OUT	destweights	呼び出しプロセスからのエッジの重み付け（非負の整数から成る配列）

```
int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
int sourceweights[], int maxoutdegree, int destinations[],
int destweights[])
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
    INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), IERROR
{void MPI::Distgraphcomm::Get_dist_neighbors(int maxindegree,
             int sources[], int sourceweights[], int maxoutdegree,
             int destinations[], int destweights[]) （廃止された呼び出し形式, 第15.2節を参照）}
```

これらの呼び出しはローカルである．MPI_DIST_GRAPH_NEIGHBORS_COUNTによって返されるプロセスとの間でやりとりされるエッジの数は，MPI_DIST_GRAPH_CREATE_ADJACENTまたは MPI_DIST_GRAPH_CREATEの呼び出し

で（MPI_DIST_GRAPH_CREATEの場合はおそらく呼び出しプロセス以外のプロセスによって）渡されるこのようなエッジの総数である．複数回定義されているエッジは MPI_DIST_GRAPH_NEIGHBORSによって全てカウントされ，ある順序で返される．sourceweightsまたはdestweightsあるいは両方に対しMPI_UNWEIGHTEDが渡されていた場合，あるいはグラフ作成時にMPI_UNWEIGHTEDが指定されていた場合，それに対応する重みの情報は何も返されない．sourcesとdestinationsの値の順序に関する要件は，同じ入力引数commを使用したルーチンの2つの呼び出しがエッジを同じ順序で返すことのみである． maxindegreeまたはmaxoutdegreeがMPI_DIST_GRAPH_NEIGHBOR_COUNTによって返された数字よりも小さい場合，リスト全体のうちの最初の部分のみが返される．返されるエッジの順序は， MPI_DIST_GRAPH_CREATE_ADJACENTを使用した場合にcommの作成時に与えられた順序と同じでなくても構わない．

実装者へのアドバイス 問い合わせ呼び出しはローカルとして定義されるため，各プロセスは入力エッジと出力エッジにより隣接のリストを格納する必要がある．分散グラフ指定から各プロセスの隣接リストを計算するため，MPI_DIST_GRAPH_CREATEの集団呼び出しで通信が必要となる． （**実装者へのアドバイス終わり**）

7.5.6 カルテシアン座標のシフト

プロセストポロジーがカルテシアン構造であれば，座標方向に沿ったMPI_SENDRECV操作を使用して，データのシフトを実行したい場合がある．MPI_SENDRECVは入力として，受信については送信元プロセスのランクを，送信については送信先プロセスのランクをとる．カルテシアンプロセスグループに対して関数MPI_CART_SHIFTが呼ばれると，呼び出しプロセスに上記の識別子を与える．これはMPI_SENDRECVに渡すことができる．ユーザは，座標方向と（正または負の）シフトするステップ数を指定する．この関数はローカルである．

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN	comm	カルテシアン構造を持つコミュニケータ（ハンドル）
IN	direction	シフトを行なう座標の次元（整数型）
IN	disp	変位（> 0：上方へのシフト < 0：下方へのシフト）（整数型）
OUT	rank_source	送信元プロセスのランク（整数）
OUT	rank_dest	送信先プロセスのランク（整数）

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
int *rank_source, int *rank_dest)
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
{void MPI::Cartcomm::Shift(int direction, int disp, int& rank_source,
             int& rank_dest) const （廃止された呼び出し形式, 第15.2節を参照）}
```

引数directionはシフトの次元，つまりシフトで値が変更される座標を示している．ndimsを次元数とすると，座標には 0からndims-1までの番号が付けられる．

MPI_CART_SHIFTは，指定された座標方向におけるカルテシアングループが周期的か否かに応じて，循環シフトかまたは非循環シフトの識別子を提供する．非循環シフトの場合，値MPI_PROC_NULL をrank_sourceまたはrank_destに返すことがある．これは，シフトの送信元または送信先が範囲外であることを示している．

負の値またはカルテシアンコミュニケータの次元数以上の値を指定してMPI_CART_SHIFTを呼び出すと，誤りになる．このことは，0次元のカルテシアントポロジーが付加されたcommを使用してMPI_CART_SHIFTを呼び出した場合も誤りになることを示している．

例 7.7 コミュニケータcommには2次元周期的カルテシアントポロジーが付加されているとする．また，REAL型の2次元配列1プロセスにつき1要素という形で変数Aに格納されているとする．この配列を列iがiステップだけシフトするように（垂直に，つまり列に沿って）変形したい．

```
....
C find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr)
C find Cartesian coordinates
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
     +                          status, ierr)
```

ユーザへのアドバイス Fortran言語において，DIRECTION = iで示される次元はDIMS(i+1)個のノードを持つ．ここでDIMSは格子を生成するのに使用した配列である．C言語においては，direction = i で示される次元はdims[i]で指定される次元である．（ユーザへのアドバイス終わり）

7.5.7 カルテシアン構造の分割

MPI_CART_SUB(comm, remain_dims, newcomm)

IN	comm	カルテシアン構造を持つコミュニケータ（ハンドル）
IN	remain_dims	remain_dimsのi番目のエントリがtrueであれば部分格子に残り，false であれば部分格子に残らない，という指定をおこなう（論理型ベクトル）
OUT	newcomm	呼び出しプロセスを含む部分格子を持つコミュニケータ（ハンドル）

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
```

```
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
{MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) const （廃止された
            呼び出し形式, 第15.2節を参照）}
```

カルテシアントポロジーをMPI_CART_CREATEで生成した場合には，関数MPI_CART_SUBにより，より低い次元を持つカルテシアン部分格子を形成するサブグループにコミュニケータを分割し，各サブグループごとに部分格子 となるようなコミュニケータを生成することができる．remain_dimsの全てのエントリがfalseか，またはcommにすでに0次元のカルテシアントポロジーが付加されている場合，newcommに0次元のカルテシアントポロジーが付加される．（この関数は，MPI_COMM_SPLITと密接に関係している．）

例 **7.8** MPI_CART_CREATE(..., comm)によって$(2 \times 3 \times 4)$格子が定義されていると仮定する．remain_dims = (true, false, true) とすると，次の関数呼び出しにより，

```
MPI_CART_SUB(comm, remain_dims, comm_new),
```

2×4カルテシアントポロジーを成す8個のプロセスを持つ3個のコミュニケータが生成される．remain_dims = (false, false, true)とすると，MPI_CART_SUB(comm, remain_dims, comm_new)の呼び出しで，1次元カルテシアントポロジーを成す4個のプロセスを持つ，重なり合わない6個のコミュニケータが生成される．

7.5.8 低レベルトポロジー関数

この節で紹介する2つの追加関数を使用すると，他の全てのトポロジー関数を実装することができる．一般に，これらはMPIが提供している機能に加えて仮想トポロジー機能を作成する場合でないかぎり，ユーザが直接呼び出すような関数ではない．

MPI_CART_MAP(comm, ndims, dims, periods, newrank)

IN	comm	入力コミュニケータ（ハンドル）
IN	ndims	カルテシアン構造の次元数（整数型）
IN	dims	各座標方向におけるプロセス数を指定したサイズndimsの整数配列
IN	periods	各座標方向に対しその格子が周期的か否かを指定したサイズndimsの論理型配列
OUT	newrank	呼び出しプロセスの再順序付けされたランク．呼び出しプロセスが格子に属さなければMPI_UNDEFINEDとなる．（整数型）

```
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
int *newrank)
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)
{int MPI::Cartcomm::Map(int ndims, const int dims[], const bool periods[])
            const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_CART_MAPは呼び出しプロセスの物理マシン上での「最適な」配置を計算する．この関数の実装としては，常に呼び出しプロセスのランクを返す，つまり順序を変更しないものが考えられる．

実装者へのアドバイス 関数MPI_CART_CREATE(comm, ndims, dims, periods, reorder, comm_cart) は，`reorder = true` の場合には， MPI_CART_MAP(comm, ndims, dims, periods, newrank)を呼び出し，次に`newrank` $\neq$ `MPI_UNDEFINED`であれば `color = 0`とし，それ以外の場合には`color = MPI_UNDEFINED`，そして`key = newrank`としてMPI_COMM_SPLIT(comm, color, key, comm_cart)を呼び出すことで実装できる．

関数MPI_CART_SUB(comm, remain_dims, comm_new)は，破棄する次元をまとめて1つの数にエンコーディングして`color`とし，残す次元をまとめて1つの数にエンコーディングしたものを用いて，MPI_COMM_SPLIT(comm, color, key, comm_new)を呼び出すことで実装できる．

他の全てのカルテシアントポロジー関数は，コミュニケータにキャッシュされているトポロジー情報を使用し，ローカルに実装できる． （**実装者へのアドバイス終わり**）

上の関数に対応する，一般的なグラフ構造のための新しい関数は次の通りである．

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

IN	comm	入力コミュニケータ（ハンドル）
IN	nnodes	グラフのノードの個数（整数）
IN	index	グラフ構造を指定する整数型配列 MPI_GRAPH_CREATEを参照
IN	edges	グラフ構造を指定する整数配列
OUT	newrank	呼び出しプロセスの再順序付けられたランク．呼び出しプロセスがグラフに属さなければMPI_UNDEFINEDとなる（整数）

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
int *newrank)
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
{int MPI::Graphcomm::Map(int nnodes, const int index[], const int edges[])
             const （廃止された呼び出し形式, 第15.2節を参照）}
```

実装者へのアドバイス 関数MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph) は，`reorder = true`の場合には，MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)を呼び出し，次に`newrank` $\neq$ `MPI_UNDEFINED`であれば`color = 0`とし，それ以外の場合には`color = MPI_UNDEFINED`，そして`key = newrank`としてMPI_COMM_SPLIT(comm, color, key, comm_graph)を呼び出すことで実装できる．

他の全てのグラフトポロジー関数は，コミュニケータにキャッシュされているトポロジー情報を使用し，ローカルに実装できる． （**実装者へのアドバイス終わり**）

7.6 アプリケーション例

例 7.9 図7.1の例は，格子定義と問い合わせ関数をアプリケーションプログラムでどのように使用するかを示している．偏微分方程式，例えばポアソン方程式を矩形領域について解いてみる．まず，プロセス群を2次元構造に組織する．各プロセスは4方向（上下左右）の隣接のランクを問い合わせる．数値問題は反復法で解く．詳細はサブルーチン`relax`の中に隠されている．

各緩和ステップで，各プロセスは自身が所有している全ての点について格子点に対する解関数の値を計算する．次に，プロセス間の境界の値を隣接プロセスと交換しなければならない．例えばサブルーチン exchangeに，更新された値を左側の隣接`(i-1,j)`に送るためのMPI_SEND(...,neigh_rank(1),...)のような呼び出しを含むことができる．

```
      integer ndims, num_neigh
      logical reorder
      parameter (ndims=2, num_neigh=4, reorder=.true.)
      integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
      integer neigh_rank(num_neigh), own_position(ndims), i, j
      logical periods(ndims)
      real*8 u(0:101,0:101), f(0:101,0:101)
      data dims / ndims * 0 /
      comm = MPI_COMM_WORLD
C     Set process grid size and periodicity
      call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
      periods(1) = .TRUE.
      periods(2) = .TRUE.
C     Create a grid structure in WORLD group and inquire about own position
      call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
      call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
C     Look up the ranks for the neighbors.  Own process coordinates are (i,j).
C     Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
      i = own_position(1)
      j = own_position(2)
      neigh_def(1) = i-1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
      neigh_def(1) = i+1
      neigh_def(2) = j
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
      neigh_def(1) = i
      neigh_def(2) = j-1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
      neigh_def(1) = i
      neigh_def(2) = j+1
      call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
C     Initialize the grid functions and start the iteration
      call init (u, f)
      do 10 it=1,100
        call relax (u, f)
C     Exchange data with neighbor processes
        call exchange (u, comm_cart, neigh_rank, num_neigh)
10    continue
      call output (u)
      end
```

図 7.1: 2次元並列ポアソンソルバーのプロセス構造のセットアップ

第8章

MPI環境管理

本章では，MPIの実装と実行環境（エラー処理など）に関連する各種パラメータの取得および設定（設定ついては適当な場合）を行うルーチンについて論じる．MPI実行環境へ入退場するための手続きについてもここで説明する．

8.1 実装情報

8.1.1 バージョンの問い合わせ

MPI標準の変更に対処するため，使っている環境において，MPI標準のどのバージョンを使用中であるかを確認するための，コンパイル時および実行時の両方の手段がある．

「バージョン」は，バージョンとサブバージョンを示す2つの独立した整数によって表現される．C言語およびC++言語では以下のようになる．

```
#define MPI_VERSION    2
#define MPI_SUBVERSION 2
```

Fortran言語では以下のようになる．

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 2)
PARAMETER (MPI_SUBVERSION = 2)
```

実行時に知る方法は以下のとおりである．

MPI_GET_VERSION(version, subversion)

OUT	version	バージョン番号（整数型）
OUT	subversion	サブバージョン番号（整数型）

```
int MPI_Get_version(int *version, int *subversion)
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
{void MPI::Get_version(int& version, int& subversion) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_GET_VERSIONはMPI_INITの前とMPI_FINALIZEの後に呼び出すことができる数少ない関数の1つである．MPI標準のこのバージョンと以前のバージョンで有効な(MPI_VERSION, MPI_SUBVERSION)のペアは (2,2)，(2,1)，(2,0)，および(1,2)である．

8.1.2 環境の問い合わせ

MPIが初期化される時，実行環境を記述する属性の集合がコミュニケータMPI_COMM_WORLDに結びつけられる．これらの属性の値を問い合わせるには，第6章で説明した関数MPI_COMM_GET_ATTRを使用する．これらの属性の削除，キーの解放，および値の変更をするのは誤りである．

定義済み属性キーとしては次のものがある．

MPI_TAG_UB タグ値の上限．

MPI_HOST ホストプロセスが存在していればそのランク，そうでなければMPI_PROC_NULL．

MPI_IO 正規の入出力機能を持つノードのランク（場合によってはmyrank）．同じコミュニケータでもノードごとにこのパラメータに対して異なる値を返す場合がある．

MPI_WTIME_IS_GLOBAL クロックを同期させるかどうかを示す論理型変数．

ベンダーは実装依存のパラメータ（ノード番号，実メモリサイズ，仮想メモリサイズ，など）を追加してよい．

これらの定義済み属性の値はMPI初期化(MPI_INIT)からMPI終了 (MPI_FINALIZE)まで変化せず，またユーザが更新したり削除したりすることはできない．

> **ユーザへのアドバイス** C言語の呼び出し形式では，これらの属性の返す値は要求された値を格納する変数（整数型）へのポインタである． （**ユーザへのアドバイス終わり**）

必要なパラメータの値については以下で詳しく説明する．

タグ値

タグ値は0からMPI_TAG_UBの返す値（MPI_TAG_UBを含む）までの範囲の値である．これらの値はMPIプログラムの実行中に変化しないことが保証されている．さらに，タグ上限値は32767以上でなければならない．MPIの実装では，MPI_TAG_UBの値はこの値よりも大きくすることができる．例えば，値$2^{30} - 1$はMPI_TAG_UBの妥当な値である．

属性MPI_TAG_UBはMPI_COMM_WORLDに属す全てのプロセスで同じ値を持つ．

ホストランク

MPI_HOSTの返す値は，コミュニケータMPI_COMM_WORLDに関連付けられたグループ中のホストプロセス（もしあれば）のランクである．ホストがなければMPI_PROC_NULLが返される．MPIでは，プロセスがホストであるということの意味を指定しないし，またホストの存在さえ要求しない．

属性MPI_HOSTはMPI_COMM_WORLDに属す全てのプロセスで同じ値を持つ．

入出力ランク

MPI_IOの返す値は，言語標準の入出力機能を持つことができるプロセッサのランクである． Fortran言語では，Fortran言語の入出力操作の全てがサポートされていることを意味する（`OPEN`，`REWIND`，`WRITE` など）．C言語およびC++言語では，ISO C言語およびC++言語の入出力操作の全てがサポートされていることを意味する（`fopen`，`fprintf`，`lseek`など）．

全てのプロセスが言語標準の入出力機能を持つことができれば，値MPI_ANY_SOURCEが返される．そうでない場合，呼び出しプロセスが言語標準の入出力機能を持つことができれば，それ自体のランクが返される．そうでない場合には，言語標準の入出力機能を提供できるプロセスがあれば，そのようなプロセスのうち1つのプロセスのランクが返される．全てのプロセスが同じ値を返す必要はない．どのプロセスも言語標準の入出力機能を提供できない場合には，値MPI_PROC_NULLが返される．

> **ユーザへのアドバイス** 入力は集団的ではなく，また，この属性はどのプロセスが入力を提供できるか，あるいは入力を提供しているのかを示してはいないことに注意すること． （**ユーザへのアドバイス終わり**）

クロック同期

MPI_WTIME_IS_GLOBALの返す値は，MPI_COMM_WORLDに属す全てのプロセスのクロックが同期していれば1であり，そうでなければ0である．同期させるための明示的な操作が行われている場合に，クロックの集団は同期していると考える．MPI_WTIMEの呼び出しで測定される時間の変化は長さ0のMPIメッセージが往復するのに要する時間の半分未満となることが期待される．あるプロセスの送信直前と，別のプロセスのマッチする受信直後に時間を測定した場合，受信側での時間は送信側での時間よりも常に後の時間を示さなければならない．

属性MPI_WTIME_IS_GLOBALは，クロックが同期していない場合には存在する必要はない（ただし，属性キーMPI_WTIME_IS_GLOBALは常に有効である）．この属性はMPI_COMM_WORLD以外のコミュニケータに関連付けることができる．

属性MPI_WTIME_IS_GLOBALはMPI_COMM_WORLDに属す全てのプロセスで同じ値を持つ．

MPI_GET_PROCESSOR_NAME(name, resultlen)

OUT	name	（仮想ノードではなく）実ノードに対する唯一の指定子
OUT	resultlen	nameに返される結果の（印字可能文字の）長さ

```
int MPI_Get_processor_name(char *name, int *resultlen)
MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR
{void MPI::Get_processor_name(char* name, int& resultlen) （廃止された呼び出し形式, 第15.2節を参照）}
```

このルーチンは，ルーチンを呼び出した時点のプロセッサの名前を返す．この名前は長さに制限のない文字列である．この値から，ハードウェアの特定の一部分が識別できなければならない．可能な値としては，”processor 9 in rack 4 of mpp.cs.org”や”231”（231は稼働中の同一機種システムの実際のプロセッサ番号である）がある．引数nameは，少なくともMPI_MAX_PROCESSOR_NAME文字の記憶域がなければならない．MPI_GET_PROCESSOR_NAMEはこの文字数までnameの中に書き込める．

実際に書き込まれた文字数は出力引数resultlenに返される．C言語では加えて，name[resultlen]にnull文字が保存される．resultlenは，MPI_MAX_PROCESSOR_NAME-1よりも大きくならない．Fortran言語では，nameの右にスペースがパディングされる．resultlenは，MPI_MAX_PROCESSOR_NAMEよりも大きくならない．

根拠 この関数では，プロセスマイグレーションが生じて，移動した先のプロセッサを返すような，MPI実装が可能である．ただし，プロセスマイグレーションの**必要性**や定義はMPIに含まれない．MPI_GET_PROCESSOR_NAMEのこの定義により，そのような実装が可能となるだけである． （**根拠の終わり**）

ユーザへのアドバイス ユーザは少なくともMPI_MAX_PROCESSOR_NAMEの大きさの領域をプロセッサ名を書き込むために用意しなければならない．プロセッサ名はこの長さまで許される．ユーザは名前の実際の長さを調べるために，出力引数resultlenを調べる必要がある． （**ユーザへのアドバイス終わり**）

定数MPI_BSEND_OVERHEADは，MPI_BSENDの呼び出しによりバッファリングされたメッセージごとの固定オーバーヘッドの上限を定める（第3.6.1節を参照）．

8.2 メモリ割り当て

一部のシステムでは，特別に割り当てられたメモリ（例えば，SMP上で通信するグループのプロセス間で共有されるメモリ）にアクセスすると，メッセージ通信およびリモートメモリアクセス (RMA) 操作の動作が速くなる．MPIにはこのような特別なメモリの割当と解放を行うためのメカニズムが用意されている．メッセージ通信やRMAのためにこのようなメモリを使用することは必須ではなく，このメモリは動的に割り当てられた他のメモリと同様に制限なしで使用できる．しかし，実装ではMPI_WIN_LOCKおよ

びMPI_WIN_UNLOCK関数の使用が，このようなメモリに割り当てられたウィンドウに制限される場合がある（第11.4.3節を参照）.

MPI_ALLOC_MEM(size, info, baseptr)

IN	size	メモリセグメントのサイズ（バイト単位）（非負の整数型）
IN	info	info引数（ハンドル）
OUT	baseptr	割り当てられたメモリセグメントの先頭を指すポインタ

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
{void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info) （廃止された呼び出し形式, 第15.2節を参照）}
```

info引数を使用して，割り当てられたメモリの望ましい場所を制御するための指示を与えることができる．このような指示によって呼び出しの意味が変わることはない．有効なinfo値は実装依存である．nullの指示値info = MPI_INFO_NULLは常に有効である.

関数MPI_ALLOC_MEMは，メモリ不足のために失敗したことを示すクラスMPI_ERR_NO_MEMのエラーコードを返す場合がある.

MPI_FREE_MEM(base)

IN	base	MPI_ALLOC_MEMによって割り当てされたメモリセグメントの先頭アドレス（選択型）

```
int MPI_Free_mem(void *base)
MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR
{void MPI::Free_mem(void *base) （廃止された呼び出し形式, 第15.2節を参照）}
```

関数MPI_FREE_MEMは，無効なbase引数を示すクラスMPI_ERR_BASEのエラーコードを返す場合がある.

根拠 MPI_ALLOC_MEMとMPI_FREE_MEMのC言語およびC++言語の呼び出し形式はC言語ライブラリ呼び出しの`malloc`および`free`の呼び出し形式に似ており，MPI_Alloc_mem(..., &base)の呼び出しはMPI_Free_mem(base)の呼び出し（間接参照が1段浅い）と対となる必要がある．型のキャストを容易にするため，両方の引数は同じvoid*型として宣言される．Fortran言語の呼び出し形式はC言語およびC++言語の呼び出し形式と一貫しており，Fortran言語のMPI_ALLOC_MEM呼び出しは割り当てられたメモリの（整数型の）アドレスをbaseptrに返す.
MPI_FREE_MEMのbase引数は選択型の引数で，その場所に格納された変数（の参照）を渡す． （**根拠の終わり**）

実装者へのアドバイス MPI_ALLOC_MEMで特別なメモリを割り当てる場合，セグメントを解放するときにメモリセグメントのサイズがわかるように，C言語のmallocおよびfree 関数の設計と同様の設計を使用する必要がある．特別なメモリを使用しない場合，MPI_ALLOC_MEMは単にmallocを呼び出し，MPI_FREE_MEMはfreeを呼び出す．

共有メモリシステムでは，共有メモリセグメントのメモリを割り当てるために，MPI_ALLOC_MEMの呼び出しを使うことができる． （**実装者へのアドバイス終わり**）

例 8.1 ポインタをサポートするFortran言語でのMPI_ALLOC_MEMの使用例． 4バイトのREALとし，そのポインタはアドレスサイズであるとする．

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
CALL MPI_ALLOC_MEM(4*100*100, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

標準のFortran言語は（C言語のような）ポインタをサポートしていないため，これはFortran 77言語のコードでもFortran 90言語のコードでもない．一部のコンパイラ（特に，本書の執筆時点ではg77およびIntelのFortranコンパイラ）はこのコードに対応していない．

例 8.2 同じ例（C言語）

```
float  (* f)[100][100] ;
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

8.3 エラー処理

MPIの実装では，MPI呼び出し中に発生するある種のエラーを処理できなかったり，処理しない選択をしたりすることがある．このようなエラーには，浮動小数点数エラーやアクセス違反など，例外やトラップを発生させるエラーがある．MPIで処理されるエラーの集合は実装依存である．これらのエラーのみ**MPI 例外**が発生する．

上記の文は，本書の中のエラー処理に関する記述よりも優先される．特に，エラー処理**される**という記述は，エラー処理**される可能性がある**と読み替える必要がある．

ユーザはエラーハンドラを3種類のオブジェクト，つまりコミュニケータ，ウィンドウ，ファイルに関連付けることができる．このそれぞれのオブジェクトのためのMPI呼び出し中に起こったどのようなMPI例外に対しても，その指定されたエラー処理ルーチンが使用される．どのオブジェクトとも関連しないMPI呼び出しはコミュニケータMPI_COMM_WORLDに結びつけられていると考える．オブジェクトにエラーハンドラを結びつけることは完全にローカルであり，異なるプロセスでは異なるエラーハンドラを同じオブジェクトに結びつけることができる．

MPIではいくつかの定義済みエラーハンドラが利用できる．

MPI_ERRORS_ARE_FATAL このハンドラは，呼び出されると，実行中の全てのプロセスを異常停止させる．これはこのハンドラを呼び出したプロセスでMPI_ABORTが呼ばれたのと同じ効果を持つ．

MPI_ERRORS_RETURN このハンドラはエラーコードをユーザに返すだけである．

実装では，さらに定義済みエラーハンドラを提供することもできるし，またプログラマが独自のエラーハンドラをコーディングすることもできる．

デフォルトでは初期化後にエラーハンドラMPI_ERRORS_ARE_FATALが， MPI_COMM_WORLDに関連付けられる．したがって，ユーザがエラー処理を制御しないようにしている場合，MPIが処理するエラーは全て致命的エラーとして取り扱われる．（ほとんど）全てのMPI呼び出しはエラーコードを生成するので，ユーザはメインプログラムでエラーを処理することを選択してよい．例えば，メインプログラムは，MPI呼び出しの戻り値を調べ，呼び出しが失敗したなら，適切な復旧プログラムを実行することで，処理することができる．この場合，エラーハンドラMPI_ERRORS_RETURNが使用される．通常は，MPI呼び出しごとにエラーのテストはせずに，そのエラーを（自明でない）MPIエラーハンドラで適切に処理したほうが都合が良いし，また効率もよい．

エラーが検出された後，MPIの状態は未定義である．つまり，ユーザ定義エラーハンドラ，またはMPI_ERRORS_RETURNを使用しても，必ずしも，エラー検出後にユーザがMPIを使用し続けられる**とはかぎらない**．これらのエラーハンドラの目的は，プログラムが終了する前に，ユーザがユーザ定義エラーメッセージを発行し，MPIに関連しない処理（入出力バッファのフラッシュなど）を実行できるようにすることである．エラーの後MPI処理が続行できるようにMPIを実装してもよいが，そうする必要はない．

> **実装者へのアドバイス** 質の高い実装では，可能な限り，エラーの影響を抑え，エラーハンドラが呼び出された後も通常処理が続行できるようにすることが望まれる．その実装解説書にはエラーの各クラスの起こりうる影響について情報を提供することが望まれる． （**実装者へのアドバイス終わり**）

MPIエラーハンドラはハンドルによってアクセスされる不可視のオブジェクトである．新しいエラーハンドラを作成するためのMPI呼び出し，オブジェクトにエラーハンドラを関連付けるためのMPI呼び出し，およびどのエラーハンドラがオブジェクトに関連付けられているかをテストするためのMPI呼び出しが用意されている．C言語およ

びC++言語には，コミュニケータ，ファイル，およびウィンドウ引数をとるユーザ定義エラー処理コールバック関数のためのtypedefが用意されている． Fortran言語には，3種類のユーザルーチンがある．

MPI_XXX_CREATE_ERRHANDLER(function, errhandler)の呼び出しにより， エラーハンドラオブジェクトが生成される．ここで，XXXはCOMM，WIN，またはFILEに対応する．

MPI_XXX_SET_ERRHANDLERの呼び出しにより，コミュニケータ，ウィンドウ，またはファイルにエラーハンドラが結びつけられる． エラーハンドラは定義済みのエラーハンドラ，または MPI_XXX_CREATE_ERRHANDLER（該当するXXXを持つもの）を呼び出すことにより生成されたエラーハンドラとする必要がある． 定義済みのエラーハンドラMPI_ERRORS_RETURNおよび MPI_ERRORS_ARE_FATALをコミュニケータ，ウィンドウ，ファイルに 結びつけることができる． C++言語では，定義済みのエラーハンドラMPI::ERRORS_THROW_EXCEPTIONSを コミュニケータ，ウィンドウ，ファイルに結びつけることもできる．

現在コミュニケータ，ウィンドウ，ファイルに関連付けられているエラーハンドラは，MPI_XXX_GET_ERRHANDLERを呼び出すことにより取得することができる．

MPI_XXX_CREATE_ERRHANDLERを呼び出して生成されたエラーハンドラは， MPI関数MPI_ERRHANDLER_FREEを使用して解放することができる．

MPI_{COMM,WIN,FILE}_GET_ERRHANDLERの動作は，新しいエラーハンドラオブジェクトが生成されるのと同様である．つまり，エラーハンドラが不要となった時点で， MPI_ERRHANDLER_GETまたはMPI_{COMM,WIN,FILE}_GET_ERRHANDLER から返されたエラーハンドラを使用してMPI_ERRHANDLER_FREE を呼び出し，エラーハンドラを解放のためにマークする必要がある．この動作は，MPI_COMM_GROUPおよびMPI_GROUP_FREEと同様である．

> **実装者へのアドバイス** 高品質な実装では， MPI_XXX_CREATE_ERRHANDLERを呼び出して生成されたエラーハンドラが MPI_YYY_SET_ERRHANDLERの呼び出しにより間違った型のオブジェクトに 結びつけられた場合，エラーを発生させる必要がある．そのために，各エラーハンドラで，関連付けられたユーザ関数のtypedefに関する情報を管理する必要がある． （**実装者へのアドバイス終わり**）

これらの呼び出しの構文を以下に示す．

8.3.1 コミュニケータ用のエラーハンドラ

MPI_COMM_CREATE_ERRHANDLER(function, errhandler)

IN	function	ユーザ定義のエラー処理手続き（関数）
OUT	errhandler	MPIエラーハンドラ（ハンドル）

```
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *function,
MPI_Errhandler *errhandler)
```

```
MPI_COMM_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
{static MPI::Errhandler
            MPI::Comm::Create_errhandler(MPI::Comm::Errhandler_function*
            function)（廃止された呼び出し形式, 第15.2節を参照）}
```

コミュニケータに結びつけることができるエラーハンドラを作成する．この関数は，廃止されたMPI_ERRHANDLER_CREATEと同じである．

C言語では，ユーザルーチンはMPI_Comm_errhandler_function型の関数でなければならない．これは次のように定義される．

```
typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
```

最初の引数は使用するコミュニケータである．2番目の引数は，エラーを起こしたMPIルーチンが返すエラーコードである．このルーチンの返却値がMPI_ERR_IN_STATUSである場合，これはエラーハンドラが呼び出される原因となった要求が生じたステータス中で返されるエラーコードである．残りの引数は“stdargs”引数であり，その数と意味は実装依存である．実装の際はこれらの引数について明確に文書化しなければならなず，Fortran言語でハンドラを書けるようにアドレスを使用する．このtypedefは，廃止されたMPI_Handler_functionに代わるものである．

Fortran言語では，ユーザルーチンは以下の形式でなければならない．

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
    INTEGER COMM, ERROR_CODE
```

C++言語では，ユーザルーチンは以下の形式でなければならない．

```
{typedef void MPI::Comm::Errhandler_function(MPI::Comm &, int *, ...);（廃
            止された呼び出し形式．第15.2節を参照）}
```

根拠 可変引数リストを用いたのは，エラーハンドラに付加情報を提供するためのISO標準のフックを利用するためであるが，ISO C言語ではこのフックを使用しないで引数を追加することは禁止されている．（**根拠の終わり**）

ユーザへのアドバイス 新しく生成されたコミュニケータは，「親」コミュニケータに関連付けられたエラーハンドラを継承する．特に，初期化の直後にコミュニケータMPI_COMM_WORLDにハンドラを関連付けることにより，全てのコミュニケータ用の「グローバル」エラーハンドラを指定することができる．（**ユーザへのアドバイス終わり**）

MPI_COMM_SET_ERRHANDLER(comm, errhandler)

INOUT	comm	コミュニケータ（ハンドル）
IN	errhandler	コミュニケータ用の新しいエラーハンドラ（ハンドル）

```
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

{void MPI::Comm::Set_errhandler(const MPI::Errhandler& errhandler) （廃止された呼び出し形式, 第15.2節を参照）}

コミュニケータに新しいエラーハンドラを結びつける．エラーハンドラは定義済みのエラーハンドラ，またはMPI_COMM_CREATE_ERRHANDLERを呼び出して生成されるエラーハンドラとする必要がある．この呼び出しは，廃止されたMPI_ERRHANDLER_SETと同じである．

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

IN	comm	コミュニケータ（ハンドル）
OUT	errhandler	コミュニケータに現在関連付けられているエラーハンドラ（ハンドル）

```
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

{MPI::Errhandler MPI::Comm::Get_errhandler() const （廃止された呼び出し形式, 第15.2節を参照）}

コミュニケータに現在関連付けられているエラーハンドラを取得する．この呼び出しは，廃止されたMPI_ERRHANDLER_GETと同じである．

例：ライブラリ関数はその 入口で，コミュニケータに現在登録されている エラーハンドラを退避しておき，このコミュニケータについてそれ自体のプライベートな エラーハンドラを設定し， 終了前に元のエラーハンドラに復元することができる．

8.3.2 ウィンドウ用のエラーハンドラ

MPI_WIN_CREATE_ERRHANDLER(function, errhandler)

IN	function	ユーザ定義のエラー処理手続き（関数）
OUT	errhandler	MPIエラーハンドラ（ハンドル）

```
int MPI_Win_create_errhandler(MPI_Win_errhandler_function *function,
MPI_Errhandler *errhandler)
MPI_WIN_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

{static MPI::Errhandler
MPI::Win::Create_errhandler(MPI::Win::Errhandler_function*
function) （廃止された呼び出し形式, 第15.2節を参照）}

ウィンドウオブジェクトに結びつけることができるエラーハンドラを作成する． C言語では，ユーザルーチンはMPI_Win_errhandler_function型の関数でなければならない．これは次のように定義される．

```
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
```

最初の引数は使用するウィンドウである． 2番目の引数は返されるエラーコードである． Fortran言語では，ユーザルーチンは以下の形式でなければならない．

```
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
    INTEGER WIN, ERROR_CODE
```

C++言語では，ユーザルーチンは以下の形式でなければならない．

```
{typedef void MPI::Win::Errhandler_function(MPI::Win &, int *, ...); （廃止された呼び出し形式．第15.2節を参照）}
```

MPI_WIN_SET_ERRHANDLER(win, errhandler)

INOUT	win	ウィンドウ（ハンドル）
IN	errhandler	ウィンドウ用の新しいエラーハンドラ（ハンドル）

```
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
{void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler) （廃止された呼び出し形式, 第15.2節を参照）}
```

ウィンドウに新しいエラーハンドラを結びつける． エラーハンドラは定義済みのエラーハンドラ，または MPI_WIN_CREATE_ERRHANDLERを呼び出して生成されるエラーハンドラとする必要がある．

MPI_WIN_GET_ERRHANDLER(win, errhandler)

IN	win	ウィンドウ（ハンドル）
OUT	errhandler	ウィンドウに現在関連付けられているエラーハンドラ（ハンドル）

```
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
{MPI::Errhandler MPI::Win::Get_errhandler() const （廃止された呼び出し形式, 第15.2節を参照）}
```

ウィンドウに現在関連付けられているエラーハンドラを取得する．

8.3.3 ファイル用のエラーハンドラ

MPI_FILE_CREATE_ERRHANDLER(function, errhandler)

IN	function	ユーザ定義のエラー処理手続き（関数）
OUT	errhandler	MPIエラーハンドラ（ハンドル）

```
int MPI_File_create_errhandler(MPI_File_errhandler_function *function,
MPI_Errhandler *errhandler)
MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
{static MPI::Errhandler
             MPI::File::Create_errhandler(MPI::File::Errhandler_function*
             function) （廃止された呼び出し形式, 第15.2節を参照）}
```

ファイルオブジェクトに結びつけることができるエラーハンドラを作成する．C言語では，ユーザルーチンはMPI_File_errhandler_function型の関数でなければならない．これは次のように定義される．

```
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);
```

最初の引数は使用するファイルである．2番目の引数は返されるエラーコードである．

Fortran言語では，ユーザルーチンは以下の形式でなければならない．

```
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
    INTEGER FILE, ERROR_CODE
```

C++言語では，ユーザルーチンは以下の形式でなければならない．

```
{typedef void MPI::File::Errhandler_function(MPI::File &, int *, ...); （廃
            止された呼び出し形式．第15.2節を参照）}
```

MPI_FILE_SET_ERRHANDLER(file, errhandler)

INOUT	file	ファイル（ハンドル）
IN	errhandler	ファイル用の新しいエラーハンドラ（ハンドル）

```
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
{void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler) （廃止され
            た呼び出し形式, 第15.2節を参照）}
```

ファイルに新しいエラーハンドラを結びつける．エラーハンドラは定義済みのエラーハンドラ，またはMPI_FILE_CREATE_ERRHANDLERを呼び出して生成されるエラーハンドラとする必要がある．

MPI_FILE_GET_ERRHANDLER(file, errhandler)

IN	file	ファイル（ハンドル）
OUT	errhandler	ファイルに現在関連付けられているエラーハンドラ（ハンドル）

```
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
{MPI::Errhandler MPI::File::Get_errhandler() const （廃止された呼び出し形式,
            第15.2節を参照）}
```

ファイルに現在関連付けられているエラーハンドラを取得する．

8.3.4 エラーハンドラの解放とエラー文字列の取得

MPI_ERRHANDLER_FREE(errhandler)

INOUT	errhandler	MPIエラーハンドラ（ハンドル）

```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
```

```
    INTEGER ERRHANDLER, IERROR
{void MPI::Errhandler::Free()} （廃止された呼び出し形式, 第15.2節を参照）
```

このルーチンはerrhandlerに関連したエラーハンドラを解放のためにマークし，errhandlerをMPI_ERRHANDLER_NULLに設定する．このエラーハンドラは，これを関連付けられた全てのオブジェクト（コミュニケータ，ウィンドウ，ファイル）の解放後に解放される．

MPI_ERROR_STRING(errorcode, string, resultlen)

IN	errorcode	MPIルーチンによって返されるエラーコード
OUT	string	errorcodeに対応する文
OUT	resultlen	stringに返される結果の（印字可能文字の）長さ

```
int MPI_Error_string(int errorcode, char *string, int *resultlen)
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING
{void MPI::Get_error_string(int errorcode, char* name, int& resultlen)} （廃
            止された呼び出し形式, 第15.2節を参照）
```

このルーチンはエラーコードやエラークラスと関連付けられたエラー文字列を返す．引数stringは，少なくともMPI_MAX_ERROR_STRING文字の記憶域がなければならない．

実際に書き込まれた文字数は，出力引数resultlenに返される．

根拠 この関数の形式は，Fortran言語とC言語の呼び出し形式が同様なものとなるように選択された．文字列へのポインタを返すバージョンは難しい点が2点ある．第1に，返却文字列は静的に割り当てられていなければならず，（連続するMPI_ERROR_STRINGの呼び出しによって返されるポインタが正しいメッセージを指すようにするには）エラーメッセージごとに領域が異なっていなければならないということである．第2に，Fortran言語では，CHARACTER*(*)を返すように宣言された関数は，例えばPRINT文などの文中で参照することができないということである． （**根拠の終わり**）

8.4 エラーコードおよびクラス

MPIで返されるエラーコードは，（MPI_SUCCESSを除いて）完全に実装に委ねられている．これは，実装時に（MPI_ERROR_STRINGで使われる） エラーコードにできるだけ多くの情報を入れることができるようにするためである．

アプリケーションがエラーコードを解釈できるように，ルーチンMPI_ERROR_CLASSはエラーコードを**エラークラス**と呼ばれる標準エラーコードの小さな集合のうちの1つに変換する．有効なエラークラスを表8.1および表8.2に示す．

エラークラスはエラーコードの部分集合である．MPI関数はエラークラス番号を返すことができ，関数MPI_ERROR_STRINGを使用してエラークラスと関連付けられたエラ

MPI_SUCCESS	エラーなし
MPI_ERR_BUFFER	無効なバッファポインタ
MPI_ERR_COUNT	無効なカウント引数
MPI_ERR_TYPE	無効なデータ型引数
MPI_ERR_TAG	無効なタグ引数
MPI_ERR_COMM	無効なコミュニケータ
MPI_ERR_RANK	無効なランク
MPI_ERR_REQUEST	無効な要求（ハンドル）
MPI_ERR_ROOT	無効なルート
MPI_ERR_GROUP	無効なグループ
MPI_ERR_OP	無効な操作
MPI_ERR_TOPOLOGY	無効なトポロジー
MPI_ERR_DIMS	無効な次元引数
MPI_ERR_ARG	その他の無効な引数
MPI_ERR_UNKNOWN	未知のエラー
MPI_ERR_TRUNCATE	受信時にメッセージが切り詰められた
MPI_ERR_OTHER	このリストの中に載っていない既知のエラー
MPI_ERR_INTERN	MPI内部（実装）エラー
MPI_ERR_IN_STATUS	ステータス中のエラーコード
MPI_ERR_PENDING	保留要求
MPI_ERR_KEYVAL	無効なkeyvalが渡された
MPI_ERR_NO_MEM	メモリ不足のためMPI_ALLOC_MEMに失敗した
MPI_ERR_BASE	MPI_FREE_MEMに無効なbase引数が渡された
MPI_ERR_INFO_KEY	キーがMPI_MAX_INFO_KEYより長い
MPI_ERR_INFO_VALUE	値がMPI_MAX_INFO_VALより長い
MPI_ERR_INFO_NOKEY	MPI_INFO_DELETEに無効なキーが渡された
MPI_ERR_SPAWN	プロセス生成に失敗
MPI_ERR_PORT	MPI_COMM_CONNECTに無効なポート名が渡された
MPI_ERR_SERVICE	MPI_UNPUBLISH_NAMEに無効なサービス名が渡された
MPI_ERR_NAME	MPI_LOOKUP_NAMEに無効なサービス名が渡された
MPI_ERR_WIN	無効な win 引数
MPI_ERR_SIZE	無効な size 引数
MPI_ERR_DISP	無効な disp 引数
MPI_ERR_INFO	無効な info 引数
MPI_ERR_LOCKTYPE	無効な locktype 引数
MPI_ERR_ASSERT	無効な assert 引数
MPI_ERR_RMA_CONFLICT	ウィンドウへのアクセスの衝突
MPI_ERR_RMA_SYNC	RMA呼び出しの同期エラー

表 8.1: エラークラス（第1部）

MPI_ERR_FILE	無効なファイルハンドル
MPI_ERR_NOT_SAME	全てのプロセスで同一でない集団的引数，またはプロセスごとに異なる順序で呼び出される集団的ルーチン
MPI_ERR_AMODE	MPI_FILE_OPENに渡されたamodeに関するエラー
MPI_ERR_UNSUPPORTED_DATAREP	MPI_FILE_SET_VIEWにサポートされていないdatarepが渡された
MPI_ERR_UNSUPPORTED_OPERATION	サポートされていない操作．例えば，逐次アクセスのみがサポートされるファイルのシークなど
MPI_ERR_NO_SUCH_FILE	ファイルが存在しない
MPI_ERR_FILE_EXISTS	ファイルが存在する
MPI_ERR_BAD_FILE	無効なファイル名（長すぎるパス名など）
MPI_ERR_ACCESS	操作権限なし
MPI_ERR_NO_SPACE	領域不足
MPI_ERR_QUOTA	割り当て量を超過した
MPI_ERR_READ_ONLY	読み取り専用のファイルまたはファイルシステム
MPI_ERR_FILE_IN_USE	ファイルがなんらかのプロセスで現在開かれているため，ファイル操作が完了しなかった
MPI_ERR_DUP_DATAREP	定義済みのデータ表現識別子がMPI_REGISTER_DATAREPに渡されたため，変換関数が登録できなかった
MPI_ERR_CONVERSION	ユーザ提供のデータ変換関数でエラーが発生した
MPI_ERR_IO	その他の入出力エラー
MPI_ERR_LASTCODE	最後のエラーコード

表 8.2: エラークラス（第2部）

ー文字列を作ることができる．MPIエラークラスは有効なMPIエラーコードである．特に，MPIエラークラス用に定義された値は有効なMPIエラーコードである．

エラーコードは以下の式を満たす．

$$0 = \text{MPI\_SUCCESS} < \text{MPI\_ERR\_...} \leq \text{MPI\_ERR\_LASTCODE}.$$

根拠 MPI_ERR_UNKNOWNとMPI_ERR_OTHERとの違いは， MPI_ERR_OTHERに関してはMPI_ERROR_STRINGが有用な情報を返すことができるという点にある．

C言語での慣例と一致させるためにはMPI_SUCCESS $= 0$ が必要である．エラークラスとエラーコードを分けることで，上のようにエラークラスを定義することができる．既知のLASTCODEがあることは，多くの場合正確な判断検査になる． （**根拠の終わり**）

MPI_ERROR_CLASS(errorcode, errorclass)

IN	errorcode	MPIルーチンによって返されるエラーコード
OUT	errorclass	errorcodeに関連付けられたエラークラス

```
int MPI_Error_class(int errorcode, int *errorclass)
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
{int MPI::Get_error_class(int errorcode) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_ERROR_CLASS関数は標準エラーコード（エラークラス）をそれ自体にマッピングする.

8.5 エラークラス，エラーコード，エラーハンドラ

既存のMPI実装の上に，独自のエラーコードおよびエラークラスの集合を持つことができる階層ライブラリを作成したい場合がある．このようなライブラリの例は，MPIに基づく入出力ライブラリである（405ページの第13章を参照）．このため，以下の機能が必要である.

1. 既知のMPI実装のエラークラスに新しいエラークラスを追加する.
2. MPI_ERROR_CLASSが有効になるように，このエラークラスにエラーコードを関連付ける.
3. MPI_ERROR_STRINGが有効になるように，これらのエラーコードに文字列を関連付ける.
4. コミュニケータ，ウィンドウ，オブジェクトに関連付けられたエラーハンドラを呼び出す.

このための関数がいくつか用意されている．これらは全てローカルである． エラークラスまたはエラーコードは，アプリケーションで大量に生成されることはないと考えられるため，解放するための関数は用意されていない.

MPI_ADD_ERROR_CLASS(errorclass)

OUT	errorclass	新しいエラークラスの値（整数型）

```
int MPI_Add_error_class(int *errorclass)
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR
{int MPI::Add_error_class() （廃止された呼び出し形式, 第15.2節を参照）}
```

新しいエラークラスを作成し，その値を返す.

根拠 既存のエラーコードおよびエラークラスとのコンフリクトを避けるため，値の設定はユーザではなく，実装によって行われる． （**根拠の終わり**）

実装者へのアドバイス 高品質な実装では，全てのプロセスで 同じ決定論的な方法により新しいerrorclassの値が返される． （**実装者へのアドバイス終わり**）

ユーザへのアドバイス MPI_ADD_ERROR_CLASSの呼び出しはローカルであるため，この呼び出しを行った全てのプロセスで同じerrorclassが返されない場合がある．そのため，プロセスの集合で同時に新しいエラーを登録すれば全てのプロセスで同じerrorclassが生成されると考えるのは安全ではない．しかし，実装で決定論的な方法により新しいerrorclassが返され， それらが，同じプロセスの集合（例えば，全てのプロセス）で常に同じ順序で生成される場合，値は同じになる．ただし，決定論的アルゴリズムを使用した場合でも，値がプロセスごとに異なる可能性がある．これは例えば，異なるが重なり合っているプロセスのグループにより一連の呼び出しが行われる場合に発生しうる．こうした問題のため，複数のプロセスで「同じ」エラーが発生しても，同じ値のエラーコードが生成されない場合がある．（**ユーザへのアドバイス終わり**）

MPI_ERR_LASTCODEの値は定数値で，新しいユーザ定義のエラーコードおよびエラークラスの影響を受けない．その代わり，定義済みの属性キーMPI_LASTUSEDCODEがMPI_COMM_WORLDに関連付けられる．このキーに対応する属性値は，ユーザ定義のエラークラスを含めて現在のエラークラスの最大値となる．これはローカル値で，プロセスごとに異なる可能性がある．このキーによって返される値は常にMPI_ERR_LASTCODE以上となる．

ユーザへのアドバイス キーMPI_LASTUSEDCODEによって返される値は，ユーザが関数を呼び出して明示的にエラークラス／コードを追加するまで変更されない．マルチスレッド環境では，この値が変更されていないという前提に特に注意を払う必要がある．エラーコードとエラークラスは必ずしも密集しているわけではない．MPI_LASTUSEDCODE未満の各エラークラスが有効であると想定することはできない． （**ユーザへのアドバイス終わり**）

MPI_ADD_ERROR_CODE(errorclass, errorcode)

IN	errorclass	エラークラス（整数型）
OUT	errorcode	errorclassに関連付ける新しいエラーコード（整数型）

```
int MPI_Add_error_code(int errorclass, int *errorcode)
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR
{int MPI::Add_error_code(int errorclass) （廃止された呼び出し形式, 第15.2節を参照）}
```

errorclassに関連付ける新しいエラーコードを生成し，errorcodeにその値を返す．

根拠 既存のエラーコードおよびエラークラスとのコンフリクトを避けるため，新しいエラーコードの値の設定はユーザではなく，実装によって行われる． （**根拠の終わり**）

実装者へのアドバイス 高品質な実装では，全てのプロセスで同じ決定論的な方法により新しいerrorcodeの値が返される． （**実装者へのアドバイス終わり**）

MPI_ADD_ERROR_STRING(errorcode, string)

IN	errorcode	エラーコードまたはエラークラス（整数型）
IN	string	errorcodeに対応する文（文字列）

```
int MPI_Add_error_string(int errorcode, char *string)
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) STRING
{void MPI::Add_error_string(int errorcode, const char* string) （廃止された呼び出し形式, 第15.2節を参照）}
```

エラー文字列をエラーコードまたはエラークラスに関連付ける．文字列の長さはMPI_MAX_ERROR_STRINGを超えてはならない．文字列の長さは呼び出し言語内で定義されている．C言語またはC++言語では，null終端文字は文字列の長さに含まれない．Fortran言語では，末尾の空白は除外される．すでに文字列が格納されているerrorcodeに対してMPI_ADD_ERROR_STRINGを呼び出すと，古い文字列が新しい文字列で置換される．値がMPI_ERR_LASTCODE以下のエラーコードまたはエラークラスに対してMPI_ADD_ERROR_STRINGを呼び出すのは誤りである．

文字列が設定されていない状態でMPI_ERROR_STRINGを呼び出すと，空の文字列（Fortran言語では全てが空白文字，C言語およびC++言語では""）が返される．

286ページの第8.3節で，エラーハンドラを生成し，コミュニケータ，ファイル，ウィンドウに関連付ける方法を説明する．

MPI_COMM_CALL_ERRHANDLER (comm, errorcode)

IN	comm	エラーハンドラを持つコミュニケータ（ハンドル）
IN	errorcode	エラーコード（整数型）

```
int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
{void MPI::Comm::Call_errhandler(int errorcode) const （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は，エラーコードが設定されてコミュニケータに割当てられたエラーハンドラを呼び出す．この関数は，エラーハンドラが正常に呼び出された場合，（そのプロセスを異常停止せず，エラーハンドラが戻ると仮定すると）C言語およびC++言語ではMPI_SUCCESSを返し，IERRORに同じ値を返す．

ユーザへのアドバイス デフォルトのエラーハンドラはMPI_ERRORS_ARE_FATALである．そのため，このコミュニケータまたは，コミュニケータが生成される前の親コミュニケータに対してデフォルトのエラーハンドラが変更されていない場合，

MPI_COMM_CALL_ERRHANDLER 呼び出しはcommのプロセスを異常停止する．（ユーザへのアドバイス終わり）

MPI_WIN_CALL_ERRHANDLER (win, errorcode)

IN	win	エラーハンドラを持つウィンドウ（ハンドル）
IN	errorcode	エラーコード（整数型）

```
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR
{void MPI::Win::Call_errhandler(int errorcode) const （廃止された呼び出し形式,
              第15.2節を参照）}
```

この関数は，エラーコードが設定されてウィンドウに割当てられたエラーハンドラを呼び出す．この関数は，エラーハンドラが正常に呼び出された場合，（そのプロセスを異常停止せず，エラーハンドラが戻ると仮定すると）C言語およびC++言語ではMPI_SUCCESSを返し，IERRORに同じ値を返す．

ユーザへのアドバイス コミュニケータの場合と同様，ウィンドウのデフォルトのエラーハンドラも MPI_ERRORS_ARE_FATALである． （ユーザへのアドバイス終わり）

MPI_FILE_CALL_ERRHANDLER (fh, errorcode)

IN	fh	エラーハンドラを持つファイル（ハンドル）
IN	errorcode	エラーコード（整数型）

```
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
    INTEGER FH, ERRORCODE, IERROR
{void MPI::File::Call_errhandler(int errorcode) const （廃止された呼び出し形式,
              第15.2節を参照）}
```

この関数は，エラーコードが設定されてファイルに割当てられたエラーハンドラを呼び出す．この関数は，エラーハンドラが正常に呼び出された場合，（そのプロセスを異常停止せず，エラーハンドラが戻ると仮定すると）C言語およびC++言語ではMPI_SUCCESSを返し，IERRORに同じ値を返す．

ユーザへのアドバイス コミュニケータやウィンドウのエラーとは異なり，ファイルのデフォルトの動作は MPI_ERRORS_RETURNを持つ． （ユーザへのアドバイス終わり）

ユーザへのアドバイス ユーザは，MPI_COMM_CALL_ERRHANDLER，MPI_FILE_CALL_ERRHANDLER，またはMPI_WIN_CALL_ERRHANDLERにより再帰的にハンドラを呼び出さないよう注意する必要がある．再帰的な呼び出しを

行うと，再帰が無限に繰り返される可能性がある．この状況は，MPI_COMM_CALL_ERRHANDLER，MPI_FILE_CALL_ERRHANDLER，またはMPI_WIN_CALL_ERRHANDLERをエラーハンドラ内で呼び出した場合に発生する可能性がある．

エラーコードとエラークラスはプロセスに関連付けられる．そのため，これらを任意のエラーハンドラ内で使用することができる． エラーハンドラは，与えられた任意のエラーコードに対応できるように準備する必要がある．さらに，適切なエラーコードだけを使用してエラーハンドラを呼び出すことをお勧めする．例えば，通常はファイルエラーはファイルエラーハンドラに送信される． （**ユーザへのアドバイス終わり**）

8.6 時刻関数と同期

MPIは時刻関数を定義する．時刻関数そのものは「メッセージ通信」の範疇ではないが，MPIではこれを規格化する．なぜなら，「性能デバッギング」では並列プログラムの時間を計ることが重要であり，かつ（POSIX 1003.1-1988と1003.4D 14.1やFortran 90言語中の）既存の時刻関数は不便であったり，高精度な時刻へ適切にアクセスできる関数として提供されていないからである． 25ページの第2.6.5節も参照すること．

MPI_WTIME()

```
double MPI_Wtime(void)
DOUBLE PRECISION MPI_WTIME()
{double MPI::Wtime() （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_WTIMEは，ある過去の時刻からの経過時間を表す秒数を浮動小数点数で返す．

「過去の時刻」は，プロセスの存続期間中変更されないことが保証されている．ユーザは，大きな秒数を他の単位に変換したい場合には自分で変換を行わなければならない．

この関数は可搬であり（「時計の刻み数」ではなく秒数を返す），高精度であることも可能であり，不要な処理も必要ない．次のように使用する．

```
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n",endtime-starttime);
}
```

返される時刻はそれを呼び出したノードにローカルなものである．異なるノードが「同じ時刻」を返す必要はない（MPI_WTIME_IS_GLOBALの説明も参照すること）．

MPI_WTICK()

```
double MPI_Wtick(void)
DOUBLE PRECISION MPI_WTICK()
{double MPI::Wtick() （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_WTICKは，MPI_WTIMEの精度を秒数で返す．すなわち，時計の刻み幅の秒数を倍精度値として返す．例えば，時計が1ミリ秒ごとに増加するカウンターとしてハードウェアにより実装されている場合，MPI_WTICKが返す値は 10^{-3} でなければならない．

8.7 起動

MPIの1つの目標は**ソースコードの可搬性**を実現することである．これは，MPIを使って適切な言語標準に準拠して作成されたプログラムはそのままで 可搬であり，あるシステムから別のシステムへ移したときにソースコードの変更を必要としてはいけないということを意味する．このことは，コマンドラインからMPIプログラムを起動する方法やMPIプログラムの実行環境をセットアップするためにユーザがしなければならないことについては何も明確に**述べてはいない**．しかし，実装では他のMPIルーチンが呼ばれる前に何らかのセットアップを実行する必要がある場合もある．この場合に備えて，MPIには初期化ルーチンMPI_INITがある．

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)
MPI_INIT(IERROR)
    INTEGER IERROR
{void MPI::Init(int& argc, char**& argv) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::Init() （廃止された呼び出し形式, 第15.2節を参照）}
```

全てのMPIプログラムでは，初期化ルーチンMPI_INITまたはMPI_INIT_THREADを1回だけ呼び出す必要がある．初期化ルーチンを2回以上呼び出すのは誤りである．MPI初期化ルーチンを呼び出す前に呼び出すことができるMPI関数は，MPI_GET_VERSION, MPI_INITIALIZED， MPI_FINALIZEDのみである．ISO C言語の場合は，mainへの引数により渡されるargcとargv，またはNULLが使用できる．

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program    */

    MPI_Finalize();     /* see below */
}
```

Fortran言語の場合は，IERRORのみが使用できる．

C言語およびC++言語のアプリケーションでmainのargcおよびargv引数にNULLを渡せるようにするには，適合するMPIの実装が必要となる．C++言語では，これらの引数をまったくとらないMPI::Initのための代替の 呼び出し形式がある．

根拠 一部のアプリケーションでは，ライブラリからMPI_Initが呼び出され，mainのargcとargvにアクセスしない場合がある．環境に関する特別な情報やmpiexecによって提供される情報を必要とするアプリケーションでは，環境変数から情報を取得できると考えられる． （**根拠の終わり**）

MPI_FINALIZE()

```
int MPI_Finalize(void)
MPI_FINALIZE(IERROR)
    INTEGER IERROR
{void MPI::Finalize() （廃止された呼び出し形式, 第15.2節を参照）}
```

このルーチンは全てのMPI状態を終了させる．各プロセスは，終了の前にMPI_FINALIZEを呼び出す必要がある．MPI_ABORTが呼び出されていない場合，各プロセスではMPI_FINALIZEを呼び出す前に保留中のノンブロッキング通信が（ローカルに）完了していなければならない．また，最後のプロセスがMPI_FINALIZEを呼び出した時点で，保留中の全ての送信が対になる受信とマッチされ，保留中の全ての受信が対になる送信とマッチされなければならない．

例えば，以下のプログラムは正しく動作する．

```
Process 0               Process 1
---------               ---------
MPI_Init();             MPI_Init();
MPI_Send(dest=1);       MPI_Recv(src=0);
MPI_Finalize();         MPI_Finalize();
```

マッチする受信がない場合，プログラムは誤りである．

```
Process 0               Process 1
-----------             -----------
MPI_Init();             MPI_Init();
MPI_Send (dest=1);
MPI_Finalize();         MPI_Finalize();
```

ブロッキング通信操作や，MPI_WAITまたはMPI_TESTから正常に戻った場合，バッファの再使用が可能になり，ユーザによる通信が完了していることを示しているが，ローカルプロセスの処理が残っていないことを保証するものではない．MPI_ISENDによって生成された要求ハンドルを使用したMPI_REQUEST_FREEから正常に戻った場合，ハンドルが無効化されるが，操作の完了を保証するわけではない．MPI_ISENDが完了するのは，何らかの方法によってマッチする受信が完了していることが確認できたときのみである．MPI_FINALIZEは，戻る前に，ユーザが完了した通信によって必要とされるローカルな全ての処理が実際に実行されることを保証する．

MPI_FINALIZEは，完了していない保留中の通信については何も保証しない（完了はMPI_WAIT，MPI_TEST，MPI_REQUEST_FREEと他の完了の確認の組み合わせによってのみ保証される）．

例 8.3 このプログラムは正しく動作する．

```
rank 0                              rank 1
=====================================================
...                                 ...
MPI_Isend();                        MPI_Recv();
MPI_Request_free();                 MPI_Barrier();
MPI_Barrier();                      MPI_Finalize();
MPI_Finalize();                     exit();
exit();
```

例 **8.4** このプログラムは誤りであり，動作は未定義である．

```
rank 0                              rank 1
=====================================================
...                                 ...
MPI_Isend();                        MPI_Recv();
MPI_Request_free();                 MPI_Finalize();
MPI_Finalize();                     exit();
exit();
```

MPI_BUFFER_DETACHがMPI_BSEND（またはバッファリング付き送信）とMPI_FINALIZEの間に実行されない場合，MPI_FINALIZEにより黙示的にMPI_BUFFER_DETACHが実行される．

例 **8.5** このプログラムは正しく動作し，MPI_Finalizeの実行後にはバッファの切り離しが行われている．

```
rank 0                              rank 1
=====================================================
...                                 ...
buffer = malloc(1000000);           MPI_Recv();
MPI_Buffer_attach();                MPI_Finalize();
MPI_Bsend();                        exit();
MPI_Finalize();
free(buffer);
exit();
```

例 **8.6** この例では，MPI_Iprobe()はFALSEフラグを返す必要がある．MPI_Test_cancelled()は，プロセス0 のMPI_Cancel()とプロセス1のMPI_Finalize()の実行の相対順序に関係なく，TRUEフラグを返す必要がある．

MPI_Iprobe()呼び出しは，ユーザが送信先に“tag1”メッセージが存在することをユーザに認識させずに，処理系には認識させておくためのものである．

```
rank 0                              rank 1
=====================================================
MPI_Init();                         MPI_Init();
MPI_Isend(tag1);
MPI_Barrier();                      MPI_Barrier();
```

```
                                    MPI_Iprobe(tag2);
MPI_Barrier();                      MPI_Barrier();
                                    MPI_Finalize();
                                    exit();
MPI_Cancel();
MPI_Wait();
MPI_Test_cancelled();
MPI_Finalize();
exit();
```

実装者へのアドバイス 実装では，後から発生する可能性のあるメッセージの取り消しの処理が全て完了するまで，MPI_FINALIZE からの戻りを遅らせる必要がある場合もある．考えられる1つの解決策は，MPI_FINALIZEの内部にバリアを設置することである． （**実装者へのアドバイス終わり**）

MPI_FINALIZEが戻った後では，MPI_GET_VERSION，MPI_INITIALIZED，MPI_FINALIZED以外のMPIルーチンは（MPI_INITでさえ）呼び出すことはできない．各プロセスはMPI_FINALIZEを呼び出す前に開始した保留中の通信を全て完了する必要がある．呼び出しが戻れば，各プロセスはローカルの処理を続行することも，その後の他のプロセスとのMPI通信に参加せずに終了することもできる．MPI_FINALIZEは接続されている全てのプロセスに対して集団的である．スポーン（spawn），アクセプト（accept），またはコネクト（connect）が行われたプロセスがない場合，これはMPI_COMM_WORLDが対象となる．それ以外の場合，かつて接続していたまたは現在も接続中である全てのプロセスの集合に対して集団的である．これらの事項は，343ページの第10.5.4節で説明している．

実装者へのアドバイス プロセスが開始した全ての通信を完了した場合でも，このような通信は下位のMPIシステムから見ると完了していない可能性がある．例えば，ブロッキング送信は，データが送信側にバッファリングされた状態でも完了している可能性がある．MPI実装では，MPI_FINALIZEが戻る前にプロセスが全てのMPI通信への関与を完了していることを保証する必要がある．そのため，MPI_FINALIZEの呼び出し後にプロセスが終了した場合，継続中の通信のエラーが発生することはない． （**実装者へのアドバイス終わり**）

全てのプロセスがMPI_FINALIZEから戻る必要はないが，計算のMPI部分が完了していることがユーザにわかるように，少なくともMPI_COMM_WORLDのプロセス0は戻る必要がある．また，POSIX環境では，MPI_FINALIZEから戻る各プロセスに対して終了コードを記述することが望ましい場合もある．

例 8.7 以下に，少なくとも1つのプロセスが戻り，戻ったプロセスの1つがプロセス0であることが確認できることを必要とする使用法を示す．以下のようなコードを使用すれば，どれだけ多数のプロセスが戻っても正しく動作する．

```
    ...
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    ...
    MPI_Finalize();
    if (myrank == 0) {
        resultfile = fopen("outfile","w");
        dump_results(resultfile);
        fclose(resultfile);
    }
    exit(0);
```

MPI_INITIALIZED(flag)

OUT	flag	MPI_INITが呼び出された場合にフラグはtrueとなり，それ以外はfalseとなる．

```
int MPI_Initialized(int *flag)
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

{`bool MPI::Is_initialized()` （廃止された呼び出し形式, 第15.2節を参照）}

このルーチンは，MPI_INITの呼び出しが完了しているかどうかの確認のために使用することができる．MPI_INITIALIZEDは，呼び出しプロセスですでにMPI_INITが呼び出されている場合にtrueを返す．MPI_FINALIZEが呼び出されているかどうかは，MPI_INITIALIZEDの動作に影響しない．これはMPI_INITが呼び出される前に呼び出すことのできる数少ないルーチンの1つである．

MPI_ABORT(comm, errorcode)

IN	comm	異常停止すべきタスクのコミュニケータ
IN	errorcode	呼び出し環境に返すエラーコード

```
int MPI_Abort(MPI_Comm comm, int errorcode)
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

{`void MPI::Comm::Abort(int errorcode)` （廃止された呼び出し形式, 第15.2節を参照）}

このルーチンは，commのグループ中の全てのタスクを「できるかぎり」異常停止しようとする．この関数は呼び出し環境でエラーコードに応じた処理を要求しない．しかし，UnixやPOSIX環境では，これをメインプログラムからの`return``errorcode`として処理する必要がある．

これがプロセスの部分集合である場合， MPI実装が commによって表わされるプロセスだけを異常停止する ことができない場合もある．この場合， MPI実装はコネクトされた全てのプロセスを異常停止すべきであるが，コネクトされていないどんなプロセスも異常停止すべきでない ．スポーン（spawn），アクセプト（accept），またはコネクト（connect）が行われたプロセスがない場合，MPI_COMM_WORLDに関連付けられた全てのプロセスが異常停止される．

根拠 今後，環境へのMPIの拡張として動的なプロセス管理などが行えるように，コミュニケータの引数が用意されている．特に，これを使用してMPI実装でのMPI_COMM_WORLDの部分集合の異常停止を行うことができるが，必須ではない．（**根拠の終わり**）

ユーザへのアドバイス エラーコードが実行可能ファイルから返されるかMPIプロセスの起動メカニズム（mpiexecなど）から返されるかは，MPIライブラリの品質に関する事柄であり，必須事項ではない． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 高品質な実装では，可能なかぎり，MPIプロセスの起動メカニズム（mpiexec，シングルトンinitなど）からエラーコードを返そうとする．（**実装者へのアドバイス終わり**）

8.7.1 プロセス終了時のユーザ関数の実行

MPIプロセスの終了時に処理を実行できれば便利な場合がある．例えば，MPIジョブ（または，動的に生成されたプロセスの場合に終了させるジョブのその部分）の完了まで有益である初期化をルーチンで実行することができる．MPIでこれを実現するには， コールバック関数を使用してMPI_COMM_SELFに属性を結びつける． MPI_FINALIZEが呼び出されると，まずMPI_COMM_SELFで MPI_COMM_FREEと同等の機能が実行される．これにより，MPI_COMM_SELFに関連付けられた全てのキーで，MPI_COMM_SELFで設定されたのと逆の順序で削除コールバック関数が実行される．MPI_COMM_SELFにキーが結びつけられていない場合コールバックは呼び出されない．MPI_COMM_SELFの「解放」は，MPIの他の部分が影響を受ける前に行われる．そのため，例えばこれらのコールバック関数でMPI_FINALIZEDを呼び出すとfalseが返される．MPI_COMM_SELFについて処理が完了すると，MPI_FINALIZEが行う残りの処理と順序は規定されていない．

実装者へのアドバイス 属性はサポートされる任意の言語から追加することができるため， MPI実装では正しいコールバックが実行されるように，生成時の言語を記憶しておく必要がある．内部でMPI_COMM_SELFで属性削除コールバックを使用する実装では，ライブラリまたはアプリケーションにアプリケーションレベルのコールバックの実行前にシャットダウンされるMPI実装の部分がないことを保証するため，MPI_INIT / MPI_INIT_THREADから戻る前に内部コールバックを登録する必要がある． （**実装者へのアドバイス終わり**）

8.7.2 MPIが終了しているかどうかの確認

MPIの目標の1つは，階層ライブラリを利用できるようにすることであった．ライブラリでこれをすっきりとした形で行うために，MPIがアクティブであるかどうかを知る必要がある．MPIでは，MPIが初期化されているかどうかを確認するための関数MPI_INITIALIZEDが用意された． 問題は，MPIが終了済であるかどうかを知ろうとする

ときに発生する．それは，一度MPIが終了してしまうと，もはやアクティブでなくなり，再始動することも不可能となるからである． ライブラリが適切に機能するには，この状態を確認できなければならない．そのため，以下の関数が必要となる．

MPI_FINALIZED(flag)

OUT	flag	MPIが終了した場合にtrue（論理型）

```
int MPI_Finalized(int *flag)
MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
{bool MPI::Is_finalized()（廃止された呼び出し形式, 第15.2節を参照）}
```

このルーチンは，MPI_FINALIZEが完了しているとtrueを返す．MPI_FINALIZEDの呼び出しは，MPI_INITの前やMPI_FINALIZEの後に行っても問題ない．

> **ユーザへのアドバイス** MPI_INITが完了していてMPI_FINALIZEが完了していない場合，MPIは「アクティブ」であるため，MPI関数を呼び出しても安全である．ライブラリが，MPIがアクティブかどうかを知るための他の方法を提供していない場合，MPI_INITIALIZEDおよびMPI_FINALIZEDを使用してこれを確認することができる．例えば，MPIはMPI_FINALIZEの実行中に呼び出されたコールバック関数内で「アクティブ」である． （**ユーザへのアドバイス終わり**）

8.8 可搬なMPIプロセスの起動

MPIの多数の実装において，MPIプログラム用の以下の形式の起動コマンドが用意されている．

```
mpirun <mpirun arguments> <program> <program arguments>
```

プログラムの起動用のコマンドをプログラム自体と分離することにより，特にネットワークや異機種の実装での自由度が高くなっている．例えば，起動スクリプトはMPIプログラム自体を実行しているマシン上で実行する必要はない．

標準の起動メカニズムを備えているため，これらを管理するコマンドラインやスクリプトにまでMPIプログラムの可搬性が拡張される．例えば，何百というプログラムを実行する検証スイートスクリプトは，このような標準の起動メカニズムを使用して記述すると，可搬なスクリプトにすることができる．「標準の」コマンドと，標準でなく実装間で可搬でない既存の実行方法とを混同しないために， `mpirun` の代わりにMPIは`mpiexec`を規定する．

標準の起動メカニズムはMPIの使いやすさの向上に役立つが，環境の範囲が非常に広い（例えば，コマンドラインインターフェイスさえない場合もある）ため，MPIでこのようなメカニズムの使用を必須と定めることはできない．その代わり，MPIでは`mpiexec`起動コマンドを規定し，これを推奨しているが，必須とはしていない．（実装者へのアドバ

イス）．しかし，mpiexecというコマンドが用意されている実装の場合，以下の形式を使用する必要がある．

以下の形式

```
mpiexec -n <numprocs> <program>
```

を，初期MPI_COMM_WORLDを使用し，グループに<numprocs>個のプロセスが含まれる<program>を起動する少なくとも1つの方法として提案する．mpiexecのその他の引数は 実装依存である．

実装者へのアドバイス MPIプログラム用の特別な起動コマンドを用意する場合，以下の形式を使用することをお勧めする． この構文は,mpiexecをMPI_COMM_SPAWNのコマンドラインバージョンとして見なせるように選択されている（第10.3.4節を参照）．

MPI_COMM_SPAWNと同様,

```
mpiexec -n    <maxprocs>
       -soft  <        >
       -host  <        >
       -arch  <        >
       -wdir  <        >
       -path  <        >
       -file  <        >
        ...
       <command line>
```

アプリケーションプログラム用の1つのコマンドラインとその引数で十分な場合，上記のようになる．これらの引数の意味については，第10.3.4節を参照すること．MPI_COMM_SPAWN_MULTIPLEに対応するケースでは，2通りの形式が使用できる．

形式A：

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

MPI_COMM_SPAWNと同様，引数は全てオプションである（-n x 引数もオプションで，デフォルトは実装依存である．1となることも，環境変数から取得されることも，コンパイル時に指定されることもある）．引数の名前と意味は，MPI_COMM_SPAWNのinfo引数のキーから取得される．このほか，実装依存の引数を使用することもできる．

形式Aは，入力するには便利であるが，コロンをプログラムの引数として使用することはできない．そのため，代わりにファイルベースの形式が利用できる．

形式B：

```
mpiexec -configfile <filename>
```

ここで，<filename>の各行は形式Aでコロンで区切られた部分に相当する．‘#’ で始まる行はコメントである．行が複数にまたがる場合は各行の最後に‘\’を付けて続けることができる．

例 8.8 現在のマシンまたはデフォルトのマシンでmyprogという16個のインスタンスを起動する．

```
mpiexec -n 16 myprog
```

例 8.9 ferrariというマシン上で10個のプロセスを起動する．

```
mpiexec -n 10 -host ferrari myprog
```

例 8.10 同じプログラムの3個のコピーを，それぞれ異なるコマンドラインの引数を使用して起動する．

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

例 8.11 5台のSunでoceanプログラムを起動し，10台のRS/6000でatmosプログラムを起動する．

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

この場合の実装では，適切なタイプのホストを選択するための方法があることが前提となる．そのランクは指定した順序となる．

例 8.12 5台のSunでoceanプログラムを起動し，10台のRS/6000でatmosプログラムを起動する（形式B）．

```
mpiexec -configfile myfile
```

ここで，myfileの内容は次のとおりである．

```
-n 5  -arch sun    ocean
-n 10 -arch rs6000 atmos
```

（実装者へのアドバイス終わり）

第9章

Infoオブジェクト

MPIの多くのルーチンは引数infoを取る．infoは， C言語ではMPI_Info型のハンドル，C++言語ではMPI::Info型のハンドル， Fortran言語ではINTEGER型のハンドルを持つ不可視オブジェクトである． このオブジェクトは順序不定の (key,value)ペアの集合（keyもvalueも文字列）を保管する． 1つのキーは1つの値しか持つことができない． MPIには複数のキーが予約されていて，予約されたキーを 実装が使う場合，そのキーで決められた機能を提供する必要がある． 実装では必ずしもこれらのキーをサポートする必要はなく， MPIで予約されていないキーをサポートすることもできる．

実装では，キーを認識できるかどうかに関係なく，infoオブジェクトを任意の(key, value)ペアのキャッシュとしてサポートする必要がある．認識できないキーを無視するため， MPI_Infoの形式でヒントを取得する各関数を用意する必要がある． infoオブジェクトのこの記述は，実装がキーを認識できるが関連付けられた値を認識できない場合に，特定の関数が対処する方法を定義するものではない． MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY， MPI_INFO_GET_VALUELEN，MPI_INFO_GETは，階層機能でも Infoオブジェクトを使用できるように，すべての(key,value)ペアを保持する必要がある．

キーには実装で定義される最大長MPI_MAX_INFO_KEYがあり，この範囲は32〜255である．値には実装で定義される最大長 MPI_MAX_INFO_VALがある． Fortran言語では，先頭および末尾の両方の空白文字が除外される． 戻り値は，これらの最大長を超えることはない．keyもvalueも大文字と小文字が区別される．

根拠 キーには最大長が決められている．それは，既知のキーの集合は常に有限であって 実装で把握されており，またキーを複雑にする理由がないためである． このように最大サイズは小さいので，アプリケーションでは サイズMPI_MAX_INFO_KEYのキーを宣言することができる．値のサイズの制限については，実装で任意の長さの文字列を扱う必要がないようにするために設ける必要がある． （**根拠の終わり**）

ユーザへのアドバイス MPI_MAX_INFO_VALは非常に大きくなることがあるため，そのサイズの文字列を宣言することは賢明でない場合がある． （**ユーザへのアドバ**

イス終わり）

ノンブロッキングルーチンの引数である場合，infoはそのルーチンが戻る前に解析されるため，戻った直後に変更または解放ができるようになっている．

記述がキーまたは値を論理型，整数型，またはリストとして参照する場合，これらの型の文字列表現を意味する． 実装ではinfo値の文字列を他の型に変換する方法に関する独自の規則を定義することができるが，可搬性を保証するため，すべての実装で以下の表現をサポートする必要がある．論理型の正規の値には文字列 “true”および“false”（すべて小文字）が含まれていなければならない． 整数型の場合，正規の値には，プログラムの標準の整数型の範囲内である整数の 10進値の文字列表現が含まれていなければならない．（しかし，正規の整数が必ずしも指定のキーに対する正規の値とならない可能性がある．）正の数の場合，+符号はオプションである．+または−の符号と数字の先頭の桁の間に空白文字を入れてはならない．カンマ区切りのリストの場合，文字列はカンマで区切られた正規の要素を含んでいなければならない．上記のinfo値の型やカンマ区切りのリストの各要素では，先頭および末尾の空白文字は自動的に除外される．これらの規則はこれらの型のすべてのinfo値に適用される． 実装では，他のinfoキーの値に対して異なる解釈を自由に指定できる．

MPI_INFO_CREATE(info)

OUT	info	生成されるinfoオブジェクト（ハンドル）

```
int MPI_Info_create(MPI_Info *info)
MPI_INFO_CREATE(INFO, IERROR)
    INTEGER INFO, IERROR
{static MPI::Info MPI::Info::Create()} （廃止された呼び出し形式, 第15.2節を参照）
```

MPI_INFO_CREATEは新しいinfoオブジェクトを生成する．新しく生成されたオブジェクトにはキー／値ペアは含まれていない．

MPI_INFO_SET(info, key, value)

INOUT	info	infoオブジェクト（ハンドル）
IN	key	キー（文字列）
IN	value	値（文字列）

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY, VALUE
{void MPI::Info::Set(const char* key, const char* value)} （廃止された呼び出し形式, 第15.2節を参照）
```

MPI_INFO_SETは(key,value)ペアをinfoに追加し，同じキーの値がすでに設定されている場合，値を上書きする． C言語ではkeyとvalueはnullで終わる文字列である． Fortran言語では， keyとvalueの先頭および末尾の空白文字は除外される． keyまたはvalueが許

容最大サイズを超える場合，それぞれエラー MPI_ERR_INFO_KEYまたは MPI_ERR_INFO_VALUEが発生する．

MPI_INFO_DELETE(info, key)

INOUT	info	infoオブジェクト（ハンドル）
IN	key	キー（文字列）

```
int MPI_Info_delete(MPI_Info info, char *key)
MPI_INFO_DELETE(INFO, KEY, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY
{void MPI::Info::Delete(const char* key) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_INFO_DELETEはinfoから(key,value)ペアを削除する． infoにkeyが定義されていない場合，この呼び出しで クラスMPI_ERR_INFO_NOKEYのエラーが発生する．

MPI_INFO_GET(info, key, valuelen, value, flag)

IN	info	infoオブジェクト（ハンドル）
IN	key	キー（文字列）
IN	valuelen	value引数の長さ（整数型）
OUT	value	値（文字列）
OUT	flag	keyが定義されている場合は true，定義されていない場合はfalse（論理型）

```
int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value,
int *flag)
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    CHARACTER*(*) KEY, VALUE
    LOGICAL FLAG
{bool MPI::Info::Get(const char* key, int valuelen, char* value) const （廃
            止された呼び出し形式, 第15.2節を参照）}
```

この関数は，その前のMPI_INFO_SETの呼び出しでキーに関連付けられた値を取得する．このような値が存在する場合，flagにtrueを設定し， valueに値を返す．値が存在しない場合，flagにfalseを設定し，valueは変更しない．valuelenは valueで使用可能な文字数を示す．これが値の実際のサイズよりも小さい場合，収まらない分が値から切り詰められる．C言語では，null終端文字を格納するため， valuelenは割り当てた領域のサイズより1文字少なくすべきである．

keyがMPI_MAX_INFO_KEYのサイズより大きい場合， 呼び出しは誤りである．

MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)

IN	info	infoオブジェクト（ハンドル）
IN	key	キー（文字列）
OUT	valuelen	value引数の長さ（整数型）
OUT	flag	keyが定義されている場合はtrue，定義されていない場合はfalse（論理型）

```
int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen,
int *flag)
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    LOGICAL FLAG
    CHARACTER*(*) KEY
{bool MPI::Info::Get_valuelen(const char* key, int& valuelen) const （廃止された呼び出し形式, 第15.2節を参照）}
```

keyに関連付けられたvalueの長さを取得する． keyが定義されている場合，valuelenにはその関連付けられた値の長さが設定され，flagにtrueが設定される． keyが定義されていない場合，valuelenは変更されず， flagにfalseが設定される．C言語またはC++言語で返される長さには，文字列終端文字は含まれない．

keyがMPI_MAX_INFO_KEYのサイズより大きい場合， 呼び出しは誤りである．

MPI_INFO_GET_NKEYS(info, nkeys)

IN	info	infoオブジェクト（ハンドル）
OUT	nkeys	定義されたキーの数（整数型）

```
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
    INTEGER INFO, NKEYS, IERROR
{int MPI::Info::Get_nkeys() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_INFO_GET_NKEYSは infoに現在定義されているキーの数を返す．

MPI_INFO_GET_NTHKEY(info, n, key)

IN	info	infoオブジェクト（ハンドル）
IN	n	キー番号（整数型）
OUT	key	キー（文字列）

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
    INTEGER INFO, N, IERROR
    CHARACTER*(*) KEY
{void MPI::Info::Get_nthkey(int n, char* key) const （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は， infoに定義されたn番目のキーを返す．キーは $0 \ldots N-1$ の番号が与えられており， NはMPI_INFO_GET_NKEYSによって返された値である．0から$N-1$のすべてのキーは，定義されていることが保証されている． 与えられたキー番号

はMPI_INFO_SETまたは MPI_INFO_DELETEによってinfoが変更されないかぎり，変化しない.

MPI_INFO_DUP(info, newinfo)

IN	info	infoオブジェクト（ハンドル）
OUT	newinfo	infoオブジェクト（ハンドル）

```
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
MPI_INFO_DUP(INFO, NEWINFO, IERROR)
    INTEGER INFO, NEWINFO, IERROR
{MPI::Info MPI::Info::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_INFO_DUPは既存のinfoオブジェクトを複製し， 同じ(key,value)ペア，および同じキー順序を持つ 新しいオブジェクトを生成する.

MPI_INFO_FREE(info)

INOUT	info	infoオブジェクト（ハンドル）

```
int MPI_Info_free(MPI_Info *info)
MPI_INFO_FREE(INFO, IERROR)
    INTEGER INFO, IERROR
{void MPI::Info::Free() （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数はinfoを解放し，MPI_INFO_NULLを設定する． info引数の値は，infoがルーチンに渡されるたびに解釈される． ルーチンから戻った後にinfoが変更された場合，その解釈には反映されない.

第10章

プロセスの生成と管理

10.1 はじめに

MPI は主に，プロセスやリソース管理よりも通信が主題である．しかし，通信のための有益な枠組を定義するには，ある程度これらの事柄について考えておく必要がある．この章では，実行環境に最小限の制限を課すものの，プロセスの管理へのさまざまなアプローチが可能となる一群のMPI インターフェイスを示す．

プロセスの生成のためのMPI モデルでは，共通のMPI_COMM_WORLDのメンバシップにより関連付けられる一群の初期プロセスの生成と，MPI アプリケーションの起動後のプロセスの生成と管理が可能となる．この後半のプロセス生成の大きな推進力は，PVM文献[23]の研究によりもたらされた．この研究は，利点と落とし穴を併せ持つプロセス管理とリソース制御における豊富な経験を提供してきた．

MPI フォーラムは，既存の，また今後開発されるリソースおよびプロセスのコントローラの広範囲に適した可搬なインターフェースを設計できないと判断し，リソース制御を対象としないことを決定した．リソース管理では，仮想並列マシンでのノードの追加／削除， リソースの予約／スケジューリング，MPPの計算パーティションの管理，利用可能なリソースに関する情報の返却など，幅広い能力が求められる．ここでは，リソース管理が，密に結合されたシステムの場合はコンピュータベンダーによって，または環境がワークステーションのクラスタの場合はサードパーティのソフトウェアパッケージによって，外部的に提供されるものと仮定している．

MPI にプロセス管理を含める理由としては，技術的な面と実践的な面の両方がある．ある種の重要なメッセージ通信アプリケーションはプロセス管理を必要とする．ここには，タスク並列，並列モジュールを利用したシリアルアプリケーション，起動が必要なプロセスの数とタイプを実行時に評価する必要がある問題など，が含まれる．実践的な面では，PVMからMPI への移行を行うワークステーションクラスタの利用者は，プロセスとリソースの管理のためにPVMの機能を使用することに慣れている場合がある．このような機能が欠落していれば，実際に移行に障害をきたすと考えられる．

以下の目標がMPIのプロセス管理の設計の中心となっている．

- MPI のプロセスモデルが現在の並列環境の大部分に適用されなければならない．こ

こには，緊密に統合されたMPPからワークステーションの異機種ネットワークまでの全てが含まれる．

- MPI がオペレーティングシステムの責任を引き受けてはならない．その代わり，アプリケーションとシステムソフトウェアの間の有益なインターフェイスを提供する必要がある．
- MPIは動的プロセスによる通信の非決定性を保証する必要がある．つまり，動的プロセス管理により不可避の競合状態が発生することがあってはならない．
- MPI は性能を犠牲にするような機能を備えていてはならない．

プロセス管理モデルは2つの方法でこのような問題に対処する．まず，MPI は主に通信ライブラリとしての機能を維持する．並列プログラムを実行する並列環境の管理は行わず，アプリケーションおよび外部リソースとプロセスマネージャの間の最小限のインターフェイスを提供する．

つぎに，MPI はメンバの出現方法に関係なく， コミュニケータという一貫した概念を維持する．コミュニケータは生成後に変化することはなく，常に一意に決まる集団操作によって生成される．

10.2 動的プロセスモデル

動的プロセスモデルでは，MPIアプリケーションの起動後にプロセスを生成し協調的に終了することができる．このモデルは，新しく生成されたプロセスと既存のMPIアプリケーションの間の通信を確立するためのメカニズムを備えている．また，一方が他方によって「起動」されたのではない場合でも，2 つの既存のMPIアプリケーションの間の通信を確立するためのメカニズムを備えている．

10.2.1 プロセスの起動

MPI アプリケーションは，外部プロセスマネージャへのインターフェイスを通して新しいプロセスを起動することもできる．

MPI_COMM_SPAWNはMPIプロセスを起動してこれらのプロセスとの通信を確立し，グループ間コミュニケータを返す． MPI_COMM_SPAWN_MULTIPLEは複数の異なるバイナリファイル（または異なる引数を持つ同じバイナリファイル）を起動し，同じMPI_COMM_WORLDに配置し，グループ間コミュニケータを返す．

MPIは既存のグループの抽象化を使用してプロセスを表現する．プロセスは (group, rank)のペアによって識別される．

10.2.2 ランタイム環境

MPI_COMM_SPAWNおよびMPI_COMM_SPAWN_MULTIPLEルーチンはMPIとMPIアプリケーションのランタイム環境とのインターフェイスとなる．問題は，非常に広範囲の

ランタイム環境とアプリケーションの要件であり，MPIが特定の対象に合わせて調整するわけにはいかないことにある．このような環境の例としては以下のものがある．

- **バッチキューイングシステムによるMPPの管理．**通常，バッチキューイングシステムはアプリケーションが開始される前にリソースを割り当て，リソースの使用（CPU時間，メモリの使用など）に制限を課し，ジョブの開始後のリソースの割当変更を許可しない．また，多くのMPPには特別な制限または拡張として，1つのプロセッサ上で動作可能なプロセスの数の制限，並列アプリケーションのギャングスケジュールプロセスの能力などがある．

- **PVMによるワークステーションのネットワーク．** PVMParallel Virtual Machine）を使用すると，ワークステーションのネットワークから「仮想マシン」を生成することができる．アプリケーションでは，PVMライブラリを通して仮想マシンを拡張したり，プロセスを管理（生成，終了，出力のリダイレクトなど）したりすることができる．マシンやプロセスの管理要求は，外部リソースマネージャによって捕捉または処理することができる．

- **負荷分散システムによるワークステーションのネットワーク管理．**負荷分散システムは，平均負荷などの動的な量に基づいて，スポーンされたプロセスの場所を選択することができる．リソースが利用不能になった場合，プロセスをマシンから別のマシンに透過的に移行することができる．

- **Unixによる大規模なSMP．**アプリケーションはユーザによって直接実行され，オペレーティングシステムによって下位レベルでスケジュールされる．プロセスは特別なスケジュール特性（ギャングスケジューリング，プロセッサアフィニティ，デッドラインスケジューリング，プロセッサのロックなど）を持つことができ，OSのリソースの制限（プロセス数，メモリ容量など）の対象となる．

MPIでは，アプリケーションが動作する環境が存在することを黙示的に仮定している．動作しているプロセスの問い合わせ，任意のプロセスの終了， ランタイム環境の属性（プロセッサの個数，メモリの容量など）の検索といった一般的な機能などの「オペレーティングシステム」のサービスは提供していない．

MPIアプリケーションとランタイム環境との複雑なやりとりは環境固有のAPIを通して実行する必要がある．このようなAPIの例はPVMタスクとマシン管理ルーチン（`pvm_addhosts`，`pvm_config`，`pvm_tasks`など）で，おそらくこれは可能であればMPIの(group,rank)を返すように変更されている．他に，CondorやPBS APIを利用する方法もある．

下位レベルでは確かにMPIがランタイムシステムとやり取りできなければならないが，このやりとりはアプリケーションレベルでは意識されず，やりとりの詳細はMPI標準では規定されていない．

多くの場合，MPIの機能を大幅に犠牲にすることなく環境固有の情報をMPIインターフェイスから切り離しておくことはできない．アプリケーションで環境固有の機能を利

用できるようにするため，多くの MPIルーチンがアプリケーションで環境固有の情報を指定可能とするためのinfo引数を採用している．機能と可搬性の間にトレードオフがある，infoを使用するアプリケーションは可搬でない．

MPIは下位に「仮想マシン」モデルが存在する必要があると規定していない．このモデルはMPIアプリケーションと黙示的な「オペレーティングシステム」によるリソースとプロセスの管理の一貫したグローバルなビューを備えたものである．そのため，例えば，あるタスクによってスポーンされたプロセスが別のタスクから認識できなかったり，あるプロセスによってランタイム環境に追加されたホストが別のプロセスから認識できなかったり，複数の異なるプロセスによって生成されたタスクが利用可能なリソース上で自動的に分散されなかったりする．

MPIとランタイム環境の間のやりとりは以下の領域に制限される．

- プロセスはMPI_COMM_SPAWNおよびMPI_COMM_SPAWN_MULTIPLEを使用して新しいプロセスを起動することができる．

  プロセスが子プロセスをスポーンすると，info引数（オプション）を使用してプロセスの起動場所や起動方法をランタイム環境に通知することができる．この追加情報はMPIに対して不可視とすることができる．

- MPI_COMM_WORLDの属性MPI_UNIVERSE_SIZEは，初期ランタイム環境の「大きさ」，すなわち，全体でいくつのプロセスを有効に起動することができるかをプログラムに伝える．この値からMPI_COMM_WORLDのサイズを差し引いて，すでに動作しているプロセスの他にいくつのプロセスを有効に起動させることが可能かを算出することができる．

10.3 プロセスマネージャのインターフェイス

10.3.1 MPIのプロセス

プロセスはMPIでは(group, rank)のペアによって表現される．(group, rank) のペアは唯一のプロセスを指すが，プロセスは複数のグループに属することがあるため，特定の(group, rank)のペアは決定されない．

10.3.2 プロセスの起動と通信の確立

以下のルーチンは多数のMPIプロセスを起動し，これらのプロセスを使用して通信を確立し，グループ間コミュニケータを返す．

> **ユーザへのアドバイス** MPIでは静的SPMDまたはMPMDアプリケーションを起動するために，最初に1つのプロセスを起動し，MPI_COMM_SPAWNを使用してそのプロセスから兄弟プロセスを起動することができる．この方法は，主に性能上の理由から推奨できない．できれば，全てのプロセスを1つのMPIアプリケーションとして一度に起動するのが望ましい． （**ユーザへのアドバイス終わり**）

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes)

IN	command	スポーンされるプログラムの名前（文字列，ルートでのみ意味を持つ）
IN	argv	commandの引数（文字列配列，ルートでのみ意味を持つ）
IN	maxprocs	プロセスの起動の最大数（整数型，ルートでのみ意味を持つ）
IN	info	ランタイムシステムにプロセスの起動場所と起動方法を通知するkey-valueのペアの集合（ハンドル，ルートでのみ意味を持つ）
IN	root	前の引数を検査するプロセスのランク（整数型）
IN	comm	スポーンプロセスのグループが含まれるグループ内コミュニケータ（ハンドル）
OUT	intercomm	元のグループと新しくスポーンされたグループの間のグループ間コミュニケータ（ハンドル）
OUT	array_of_errcodes	プロセスごとの1つのコード（整数配列）

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info
info, int root, MPI_Comm comm, MPI_Comm *intercomm,
int array_of_errcodes[])
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERROR
{MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
              const char* argv[], int maxprocs, const MPI::Info& info,
              int root, int array_of_errcodes[]) const （廃止された呼び出し形式,
              第15.2節を参照）}
{MPI::Intercomm MPI::Intracomm::Spawn(const char* command,
              const char* argv[], int maxprocs, const MPI::Info& info,
              int root) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_COMM_SPAWNはcommandで指定されたMPIプログラムと同じmaxprocs個のコピーを起動し，これらとの通信を確立し，グループ間コミュニケータを返そうとする．生成されたプロセスは子と呼ばれる．子は親のものとは分離された独自のMPI_COMM_WORLDを持つ．MPI_COMM_SPAWNはcommに対して集団的であり，子でMPI_INITが呼び出されるまで戻らない場合もある．同様に，子のMPI_INITは全ての親がMPI_COMM_SPAWNを呼び出すまで戻らないことがある．この意味で，親のMPI_COMM_SPAWNと子のMPI_INITは，親プロセスと子プロセスのKreplace結合和集合に対して 集団操作を形成する．MPI_COMM_SPAWNによって返されるグループ間コミュニケータはローカルグループの親プロセスとリモートグループの子プロセスを持つ．ローカルグループでのプロセスの順序およびリモートグループでのプロセスの順序は，それぞれ親のcommのグループの順序および子のMPI_COMM_WORLDの順序と同じである．このグループ間コミュニケータは関数MPI_COMM_GET_PARENTを通して子で取得することができる．

ユーザへのアドバイス 実装によってはは，子がMPI_INITを呼び出す前に自動的に通信を確立するかもしれない．そのため，親でMPI_COMM_SPAWNが完了したとしても，必ずしも子でMPI_INITの呼び出しが完了しているということにはならない（ただし，返されたグループ間コミュニケータはすぐに使用できる）． （**ユーザへのアドバイス終わり**）

command引数 command引数はスポーンするプログラムの名前が格納された文字列である． C言語の場合，文字列はnullで終わる．Fortran言語では，先頭および末尾の空白文字は除外される．MPIでは，実行形式の検索方法や作業ディレクトリの規定方法は指定していない．これらの規則は実装内容によって決まり，ランタイム環境に対して適切でなければならない．

実装者へのアドバイス 実装では，実行可能ファイルの検索と作業ディレクトリの確認のために自然な規則に従う必要がある．例えば，グローバルファイルシステムを持つ同一機種システムではスポーンプロセスの作業ディレクトリを最初に検索するか，またはUnixのシェルと同様にPATH環境変数のディレクトリを検索しうる．PVMの上位への実装では，実行可能ファイル（通常は`$HOME/pvm3/bin/$PVM_ARCH`）を検索するためのPVMの規則を使用する．IBM SP上のPOEで動作するMPI実装では，POEの実行可能ファイル検索方法に従う．実装では実行可能ファイルの検索と作業ディレクトリの確認のための規則を文書化する必要があり，質の高い実装ではこれらの規則の一部の管理をユーザに任せるようにする必要がある． （**実装者へのアドバイス終わり**）

commandで名前を指定したプログラムがMPI_INITを呼び出さず，その代わりにフォークしたプロセスでMPI_INITを呼び出す場合，その結果は未定義となる．実装でこのような場合が許可されることもあるが，必ずしもこのようにする必要はない．

ユーザへのアドバイス MPIでは，起動したプログラムがシェルスクリプトで，そのシェルスクリプトによって起動されるプログラムでMPI_INITが呼び出される場合の動作については規定していない．実装によってはこの操作が許可されている場合もあるが，シェルスクリプトに渡される引数をプログラムに渡すこと，環境の特定の部分を変更しないこと，などの要件が課されることもある． （**ユーザへのアドバイス終わり**）

argv引数 argvはプログラムに渡される引数が格納された文字列の配列である． argvの最初の要素はcommandに渡される最初の引数であり，一部で慣習となっているようなコマンドそのものではない．引数リストはC言語およびC++言語の場合はnullで終了し，Fortran言語の場合は空の文字列で終了する．Fortran言語では，先頭および末尾の空白文字は除外されるので，全てが空白文字で構成される文字列は空の文字列とみなされる．空の引数リストを示すため，C言語，C++言語，およびFortran言語で定数MPI_ARGV_NULLを使用することができる．C言語およびC++言語では，この定数はnullと同じになる．

例 10.1 C言語およびFortran言語でのargvの例
C言語でプログラム“ocean” を引数“-gridfile”および “ocean1.grd”を使用して実行する方法：

```
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Comm_spawn(command, argv, ...);
```

または，コンパイル時に不明な項目がある場合の方法：

```
char *command;
char **argv;
command = "ocean";
argv=(char **)malloc(3 * sizeof(char *));
argv[0] = "-gridfile";
argv[1] = "ocean1.grd";
argv[2] = NULL;
MPI_Comm_spawn(command, argv, ...);
```

Fortran言語の場合：

```
CHARACTER*25 command, argv(3)
command = ' ocean '
argv(1) = ' -gridfile '
argv(2) = ' ocean1.grd'
argv(3) = ' '
call MPI_COMM_SPAWN(command, argv, ...)
```

引数は，オペレーティングシステムによって許されている場合，プログラムに渡される．C言語では，MPI_COMM_SPAWNのargv引数は2つの点でmainのargv引数と異なる．第1 に，要素が1つ分シフトする．特に，mainのargv[0]は実装によって渡され，慣習的に（commandによって渡された）プログラムの名前が格納される．例えば，mainのargv[1]はMPI_COMM_SPAWNのargv[0]に対応し，mainのargv[2]はMPI_COMM_SPAWNのargv[1]に対応する，などである．第2に，MPI_COMM_SPAWNのargvは，その長さを確定しうるにはnullで終了していなければならない． MPI_ARGV_NULLのargvをMPI_COMM_SPAWNに渡すと，mainが受け取るargcは1，argvは要素0が（慣習的に）プログラム名となる．

Fortran言語の実装で，プログラムにより引数が取得できるルーチンが提供されている場合，そのメカニズムを通して引数を入手できる場合がある．C言語では，オペレーティングシステムがmain()のargvの引数を(わたさない)サポートしていない場合，MPI実装でargvに引数を追加し，これをMPI_INITに渡すことができる．

MPI_INITに渡すことができるように MPI実装で追加することができる.

maxprocs引数 MPIはmaxprocs個のプロセスのスポーンを試みる．maxprocs 個のプロセスをスポーンできない場合，エラークラスMPI_ERR_SPAWN が発生する．

実装では，実装でmaxprocs個の 全てのプロセスがスポーンできない場合にエラーを発生させる代わりに指定より少ない数のプロセスを生成できるように， info引数により

デフォルト動作を変更することができる．原則的に， info引数はスポーンされたプロセスの数に対して許容される値の 任意の集合$\{m_i : 0 \leq m_i \leq \text{maxprocs}\}$ を指定することができる．集合$\{m_i\}$は必ずしも値maxprocsを含まない．実装でこれらの許容されるプロセス数のうちの1つがスポーンできる場合， MPI_COMM_SPAWNは正常に戻り，スポーンされたプロセスの数mがintercommのリモートグループのサイズによって渡される．mがmaxprocより小さい場合，下記のように，他のプロセスがスポーンされなかった理由がarray_of_errcodesに与えられる．許容される数のプロセスのうちの1つがスポーンできない場合，MPI_COMM_SPAWN によりエラークラスMPI_ERR_SPAWNが発生する．

デフォルト動作を持つスポーン呼び出しは **hard**と呼ばれる．maxprocs個より少ないプロセスが返される可能性のあるスポーン呼び出しはsoftと呼ばれる．infoのsoftキーについての詳細は， 328ページの第10.3.4節を参照すること．

> **ユーザへのアドバイス** デフォルトでは，要求はhardであり，MPIエラーは致命的エラーとなる．つまり，デフォルトでは，要求された全てのプロセスがMPIによってスポーンできない場合は致命的エラーとなる．「最大をN個とし，できるだけ多くのプロセスをスポーンする」という動作が求められる場合，ソフトスポーンを実行する必要がある．ここでは，許容される値$\{m_i\}$の集合は$\{0 \ldots N\}$となる．しかし，実装においてsoft生成をサポートすることは必須でないため，これは完全に可搬というわけではない． （**ユーザへのアドバイス終わり**）

info引数 この章の全てのルーチンのinfo引数は，C言語の場合はMPI_Info型，C++言語の場合はMPI::Info型，Fortran言語の場合は**整数型**の不可視ハンドルである．これはユーザ指定の多数の(key,value)のペアのコンテナである．keyとvalueは文字列（C言語の場合はnullで終了する`char*`，Fortran言語の場合は`character*(*)`）である． info引数の生成と操作を行うルーチンの詳細は， 311ページの第9節を参照すること．

SPAWN呼び出しの場合，infoはMPIとランタイムシステムに対して，プロセスの起動方法に関する追加の（そして，実装内容によって決まる可能性のある）指示を提供する．C言語またはFortran言語では，アプリケーションはMPI_INFO_NULLを渡すことができる．プロセスの場所に対する詳細な制御を必要としない可搬なプログラムは，MPI_INFO_NULLを使用する必要がある．

MPIはinfo引数の内容を規定していない．特別なkey値の数を予約しているだけである（328ページの第10.3.4節を参照）．info引数は非常に自由度が高く，例えば，実行可能ファイルとそのコマンド行の引数を指定するためにも使用できる．この場合，MPI_COMM_SPAWN のcommand引数は空にすることができる．これが可能なのは，MPIでは実行可能ファイルの検索方法が規定されておらず，またinfo引数はランタイムシステムに実行可能ファイル“”（空の文字列）の検索場所を伝えることができるためである．当然，これを行うプログラムは複数のMPI実装間で可搬ではない．

root引数 root引数の前の全ての引数は，comm のランクがrootであるプロセスでのみ検査される．他のプロセスでこれらの引数の値は無視される．

array_of_errcodes引数 array_of_errcodesは，MPIが起動要求を出された各プロセスのステータスをMPIがレポートするときの長さmaxprocsの配列である． maxprocs個の全てのプロセスがスポーンされた場合，array_of_errcodesに値MPI_SUCCESSが格納される．m個（$0 \leq m < \text{maxprocs}$）のプロセスのみがスポーンされた場合，$m$個のエントリにはMPI_SUCCESSが格納され，それ以外にはMPIでプロセスを起動できなかった理由を示す実装固有のエラーコードが格納される．MPIでは，どのエントリがエラーとなったプロセスに対応するかについては指定されない．実装では， 例えば，詳細仕様と1対1対応のエラーコードをinfo引数に格納することができる．引数リストにエラーがない場合，これらのエラーコードは全てエラークラスMPI_ERR_SPAWNに属する． C言語またはFortran言語では，エラーコードが不要な場合はアプリケーションでMPI_ERRCODES_IGNOREを渡すことができる．C++言語では，この定数は存在せず，array_of_errcodes引数を引数リストから省略することができる．

実装者へのアドバイス Fortran言語ではMPI_ERRCODES_IGNOREは，MPI_BOTTOMのような特別な定数の型である．17ページの第2.5.4節を参照すること． （**実装者へのアドバイス終わり**）

MPI_COMM_GET_PARENT(parent)

OUT	parent		親コミュニケータ（ハンドル）

```
int MPI_Comm_get_parent(MPI_Comm *parent)
MPI_COMM_GET_PARENT(PARENT, IERROR)
    INTEGER PARENT, IERROR
{static MPI::Intercomm MPI::Comm::Get_parent() （廃止された呼び出し形式, 第15.2節を
              参照）}
```

プロセスがMPI_COMM_SPAWNまたはMPI_COMM_SPAWN_MULTIPLEにより起動された場合，MPI_COMM_GET_PARENTは現在のプロセスの「親」 グループ間コミュニケータを返す．このグループ間コミュニケータはMPI_INIT 内で黙示的に生成され，親のSPAWNによって返されるのと同じグループ間コミュニケータである．

プロセスがSPAWNによってスポーンされたものでなかった場合，MPI_COMM_GET_PARENTはMPI_COMM_NULLを返す．

コミュニケータが解放または切断されると，MPI_COMM_GET_PARENTはMPI_COMM_NULLを返す．

ユーザへのアドバイス MPI_COMM_GET_PARENTは1つのグループ間コミュニケータのハンドルを返す．MPI_COMM_GET_PARENTの2回目の呼び出しでは，同じグループ間コミュニケータのハンドルが返される． MPI_COMM_DISCONNECTまたはMPI_COMM_FREE によりハンドルを解放すると，そのグループ間コミュニケータへの他の参照は無効（参照先なし）になる．親コミュニケータでのMPI_COMM_FREEの呼び出しは有益ではないことに注意すること． （**ユーザへのアドバイス終わり**）

根拠 フォーラムの狙いは，MPI_COMM_WORLDと同様の定数MPI_COMM_PARENTを生成することだった．ところが，明確に許可されているにもかかわらず，このような定数はMPI_COMM_DISCONNECTの引数として（構文的に）使用できない．（根拠の終わり）

10.3.3 複数の実行可能ファイルの起動と通信の確立

MPI_COMM_SPAWNはほとんどの場合に対応しているが，複数のバイナリファイル，または複数の引数の集合を持つ同じバイナリファイルをスポーンすることはできない．以下のルーチンは，複数のバイナリファイル，または複数の引数の集合を持つ同じバイナリファイルをスポーンし，これらとの通信を確立し，これらを同じMPI_COMM_WORLDに配置する．

MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes)

IN	count	コマンドの数（正の整数型，MPIのルートでのみ意味を持つ— ユーザへのアドバイスを参照）
IN	array_of_commands	実行するプログラム（文字列配列，ルートでのみ意味を持つ）
IN	array_of_argv	commandsの引数（文字列の配列，ルートでのみ意味を持つ）
IN	array_of_maxprocs	各コマンド用に起動するプロセスの最大数（整数配列，ルートでのみ意味を持つ）
IN	array_of_info	プロセスの起動場所と起動方法をランタイムシステムに伝えるinfoオブジェクト（ハンドルの配列，ルートでのみ意味を持つ）
IN	root	前の引数を検査する際のプロセスのランク（整数型）
IN	comm	スポーンするプロセスのグループのためのグループ内コミュニケータ（ハンドル）
OUT	intercomm	元のグループと新しくスポーンされたグループの間のグループ間コミュニケータ（ハンドル）
OUT	array_of_errcodes	プロセスごとに1つのエラーコード（整数配列）

```
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[],
int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES,
IERROR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
{MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
             const char* array_of_commands[], const char** array_of_argv[],
             const int array_of_maxprocs[],
             const MPI::Info array_of_info[], int root,
             int array_of_errcodes[]) （廃止された呼び出し形式, 第15.2節を参照）}
```

{MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
const char* array_of_commands[], const char** array_of_argv[],
const int array_of_maxprocs[],
const MPI::Info array_of_info[], int root) （廃止された呼び出し形式, 第15.2節を参照）}

MPI_COMM_SPAWN_MULTIPLEは，複数の実行可能ファイルが指定できる点を除いて，MPI_COMM_SPAWNと同じである．第1引数countには指定の数を渡す．それに続く4つの各引数は単にMPI_COMM_SPAWN内の対応する引数の配列である． Fortran言語用のarray_of_argvでは，要素array_of_argv(i,j)はコマンド番号 iのj番目の引数である．

根拠 Fortran言語の列優先順に慣れたFortran言語のプログラマには，これは反対に見えるかもしれない．しかし，MPI_COMM_SPAWNの引数をまとめるにはこのようにする必要がある． array_of_argvの先頭の次元はcountと同じでなければならないことに注意すること． （根拠の終わり）

ユーザへのアドバイス array_of_argvと同様に，引数countはルートでのみMPIによって解釈される．array_of_argvの先頭の次元はcountであるため，ルート以外のノードでのcountが正の値でない場合，array_of_argvがサブルーチンで無視されるとしても，理論上はランタイム境界チェックエラーが発生する．これが発生する場合，ルート以外のノードでcountに明示的に妥当な値を渡す必要がある． （ユーザへのアドバイス終わり）

どの言語でも，アプリケーションで定数MPI_ARGVS_NULL（C言語では(char***)0のようになる）を使用することにより，引数をコマンドに渡さないよう指定することができる．array_of_argvの個々の要素にMPI_ARGV_NULLを設定した場合の影響は定義されない．一部の特定のコマンドの引数の指定では，引数なしのコマンドは対応するargvの最初の要素をnull（C言語では(char *)0，Fortran言語では空の文字列）とする必要がある．

スポーンされた全てのプロセスは同じMPI_COMM_WORLDを持つ．MPI_COMM_WORLD内のランクは，コマンドがMPI_COMM_SPAWN_MULTIPLEで指定された順序に直接対応する．例えば，m_1個のプロセスが最初のコマンドで生成され，m_2個のプロセスが2番目のコマンドで生成されるとする．最初のコマンドに対応するプロセスはランク $0, 1, \ldots, m_1 - 1$を持つ． 2番目のコマンドのプロセスはランク$m_1, m_1+1, \ldots, m_1+m_2-1$を持つ． 3番目のコマンドのプロセスはランク$m_1+m_2, m_1+m_2+1, \ldots, m_1+m_2+m_3-1$を持つ．以下も同様となる．

MPI_COMM_SPAWNを複数回呼び出すと異なるMPI_COMM_WORLDを持つ多数の子の集合が生成されるが， MPI_COMM_SPAWN_MULTIPLEを呼び出すと1つのMPI_COMM_WORLDを持つ子が複数生成されるため，この2つのメソッドは完全に等価ではない．複数の実行可能ファイルをスポーンする必要がある場合に，MPI_COMM_SPAWNを複数回呼び出す代わりにMPI_COMM_SPAWN_MULTIPLEを使用した方が良いという点については，2つの性能上の理由もある．第1に，複数の子を一度に生成すれば1つずつ順に生成するよりも速く処理ができる．第2に，いくつかの実装で

は同時に生成されたプロセス間の通信は個別に生成されたプロセス間の通信よりも速くできる．

array_of_errcodes引数はサイズ$\sum_{i=1}^{count} n_i$の1次元の配列で，n_iはarray_of_maxprocsのi番目の要素である．コマンド番号iはこの配列の要素$\sum_{j=1}^{i-1} n_j$から$\left[\sum_{j=1}^{i} n_j\right] - 1$の$n_i$個の連続するスロットに対応する．エラーコードはMPI_COMM_SPAWNと同様に扱われる．

例 10.2 C言語およびFortran言語でのarray_of_argvの例

C言語でプログラム“ocean”を引数“-gridfile”および“ocean1.grd”を使用して実行し，プログラム“atmos”を引数“atmos.grd”を使用して実行する方法：

```
char *array_of_commands[2] = {"ocean", "atmos"};
char **array_of_argv[2];
char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
char *argv1[] = {"atmos.grd", (char *)0};
array_of_argv[0] = argv0;
array_of_argv[1] = argv1;
MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
```

Fortran言語での実行方法：

```
CHARACTER*25 commands(2), array_of_argv(2, 3)
commands(1) = ' ocean '
array_of_argv(1, 1) = ' -gridfile '
array_of_argv(1, 2) = ' ocean1.grd'
array_of_argv(1, 3) = ' '

commands(2) = ' atmos '
array_of_argv(2, 1) = ' atmos.grd '
array_of_argv(2, 2) = ' '

call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
```

10.3.4 予約されたキー

以下のキーが予約されている．これらのキーを実装が解釈することは必須ではないが，実装によってキーを解釈する場合，下記の機能を提供しなければならない．

host 値はホスト名．ホスト名の形式は実装によって決定される．

arch 値はアーキテクチャ名．有効なアーキテクチャ名とその意味は実装によって決定される．

wdir 値はスポーンされたプロセスが実行されるマシン上のディレクトリの名前．このディレクトリが実行中のプロセスの作業ディレクトリとなる．ディレクトリ名の形式は実装によって決定される．

path 値は実装によって実行可能ファイルが検索されるディレクトリまたはディレクトリの集合．パスの形式は実装によって決定される．

file 値は追加情報が指定されるファイルの名前．ファイル名の形式とファイルの内部形式は実装によって決定される．

soft 値はMPI_COMM_SPAWN（など）が生成可能なプロセスの数の許容値である数字の集合を指す，値の形式はFortran 90言語の三つ組みのカンマ区切りの一覧で，それぞれは整数の集合であり，これらの集合が結合されて集合を構成する．この集合の負の値とmaxprocsより大きな値は無視される．MPIは，集合内の数字と一致する，可能な限り最大に近い数のプロセスをスポーンする．三つ組みの順序は重要ではない．

Fortran 90言語の三つ組みの意味は以下のとおりである．

1. aはaを意味する．
2. a:bは$a, a+1, a+2, \ldots, b$を意味する．
3. a:b:cは$a, a+c, a+2c, \ldots, a+ck$を意味する．ここで，$c > 0$の場合，$k$は$a+ck \leq b$を満たす最大の整数．$c < 0$の場合，$a + ck \geq b$を満たす最大の整数． $b > a$の場合，cは正の数でなければならない．$b < a$の場合，cは負の数でなければならない．

例：

1. a:bはaからbまでの範囲となる．
2. 0:Nは完全な「ソフト」の機能を示す．
3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096では，2のべき乗のプロセス数が生成できる．
4. 2:10000:2では偶数のプロセスが生成できる．
5. 2:10:2,7では2，4，6，7，8，または10個のプロセスが生成できる．

10.3.5 スポーンの例

MPI_COMM_SPAWN を使用したManager-workerの例

```
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
   int world_size, universe_size, *universe_sizep, flag;
   MPI_Comm everyone;           /* intercommunicator */
   char worker_program[100];

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   if (world_size != 1)    error("Top heavy with management");

   MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
                     &universe_sizep, &flag);
```

```
   if (!flag) {
        printf("This MPI does not support UNIVERSE_SIZE. How many\n\
processes total?");
        scanf("%d", &universe_size);
   } else universe_size = *universe_sizep;
   if (universe_size == 1) error("No room to start workers");

   /*
    * Now spawn the workers. Note that there is a run-time determination
    * of what type of worker to spawn, and presumably this calculation must
    * be done at run time and cannot be calculated before starting
    * the program. If everything is known when the application is
    * first started, it is generally better to start them all at once
    * in a single MPI_COMM_WORLD.
    */

   choose_worker_program(worker_program);
   MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
             MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
             MPI_ERRCODES_IGNORE);
   /*
    * Parallel code here. The communicator "everyone" can be used
    * to communicate with the spawned processes, which have ranks 0,..
    * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
    * "everyone".
    */

   MPI_Finalize();
   return 0;
}



/* worker */

#include "mpi.h"
int main(int argc, char *argv[])
{
   int size;
   MPI_Comm parent;
   MPI_Init(&argc, &argv);
   MPI_Comm_get_parent(&parent);
   if (parent == MPI_COMM_NULL) error("No parent!");
   MPI_Comm_remote_size(parent, &size);
   if (size != 1) error("Something's wrong with the parent");

   /*
    * Parallel code here.
    * The manager is represented as the process with rank 0 in (the remote
    * group of) the parent communicator.  If the workers need to communicate
    * among themselves, they can use MPI_COMM_WORLD.
    */

   MPI_Finalize();
   return 0;
}
```

10.4 通信の確立

ここでは，コミュニケータを共有しないMPIプロセスの2つの集合の間で通信を確立するための関数を示す．

これらの関数は，以下のような状況で有益である．

1. 別々に起動されたアプリケーションの2つの部分が通信する必要がある場合．
2. 可視化ツールを実行中のプロセスに結びつけたい場合．
3. サーバが複数のクライアントからの接続を受け付けたい場合．クライアントとサーバは両方とも並列プログラムとすることができる．

これらの各状況で，MPIはそれまでに存在しなかった，親／子関係のない通信経路を確立する必要がある．この節で説明するルーチンは， MPIグループ間コミュニケータを生成することにより，プロセスの2つの集合間の通信を確立する．ここで，グループ間コミュニケータの2つのグループは元のプロセスの集合である．

既存のコミュニケータを共有しない2つのプロセスグループ間のコンタクトを確立することは，集団であるが，非対称のプロセスである．1つのプロセスグループは他のプロセスグループからの接続を受け付ける．クライアント／サーバ型のアプリケーションでない場合でも，このグループのことを（並列）**サーバ**と呼ぶ．他方のグループはサーバへの接続を行う．このグループのことを**クライアント**と呼ぶ．

> **ユーザへのアドバイス** この節の全体を通して**クライアント**および**サーバ**という名称を使用するが， MPIではクライアント／サーバシステムの従来の堅牢性を保証しない．この節で説明する機能は，同じアプリケーションの協調する2つの部分が互いに通信できるようにするためのものである．例えば，クライアントでセグメンテーション違反が発生して終了したり， クライアントが計算付集団的操作に参加しなかったりした場合， サーバがクラッシュまたはハングすることがある． （**ユーザへのアドバイス終わり**）

10.4.1 名前，アドレス，ポート，それら全て

MPIのクライアント／サーバルーチンの複雑さの大部分は「クライアントがサーバへのコンタクト方法をどのように見つけるか」という問題である，当然ながら，難しいのは，クライアントとサーバの間の既存の通信経路がなく，通信を確立するための待ち合わせ場所にある程度合意しなければならないことである．

待ち合わせ場所の合意には常に第三者が介在する．第三者はそれ自体が 待ち合わせ場所となることもあれば，サーバからクライアントへの待ち合わせ情報の伝達することもある．厄介なのは，クライアントはどのサーバにコンタクトすべきかを知りたいのではなく，その要求を扱うことができるサーバにコンタクトしたい，という点である．

MPIで幅広いランタイムシステムに対応しながら，シンプルで可搬なコードを記述する能力を維持できることが理想である．以下の項目はMPIと互換性を備えていなければならない．

- サーバは既知のインターネットアドレスhost:portに常駐する.
- サーバは端末にアドレスを出力し，ユーザがこのアドレスをクライアントプロセスに渡す.
- サーバはネームサーバに関するアドレス情報を配置する．これは合意した名前により取得することができる.
- クライアントが接続するサーバは実際にはブローカーであり，クライアントと実際のサーバとの仲介役として機能する.

MPIはネームサーバを必要としないため，必ずしも全ての実装が上記の全てのシナリオをサポートできるわけではない．しかし，MPIにはオプションのネームサーバインターフェイスが用意されており，外部のネームサーバとの互換性がある.

port_nameは，サーバにコンタクト可能な下位レベルのネットワークアドレスをエンコードするシステム提供文字列である．通常，これはIPアドレスとポート番号であるが，実装では自由に任意のプロトコルを使用できる．サーバはMPI_OPEN_PORTルーチンを使用してport_nameを確立する．次に，MPI_COMM_ACCEPTを使用して，指定されたポートとの接続を受け付ける．クライアントはport_nameを使用してサーバへの接続を行う.

port_nameのメカニズムはそれ自体では完全に可搬だが，port_nameでクライアントとの通信を接続する必要があるため， 使用が難しい場合がある．サーバがアプリケーションが提供するservice_nameを知っていて，クライアントがport_name を知らなくてもそのservice_nameに接続できれば，使い勝手が良くなる.

MPI実装ではサーバがMPI_PUBLISH_NAMEを使用して (port_name，service_name)のペアを公開し，クライアントがMPI_LOOKUP_NAMEを使用してサービス名からポート番号を取得することができる．これにより，3つのレベルの可搬性が実現され，機能のレベルを向上させることができる.

1. 名前を公開する機能に依存しないアプリケーションが最も可搬性が高い．通常，port_nameは「手動」でサーバからクライアントに転送する必要がある.
2. MPI_PUBLISH_NAMEメカニズムを使用するアプリケーションは，このサービスを備えた実装の間で完全に可搬である． 全ての実装間で可搬であるためには，名前が公開されていない場合に使用できるフォールバックメカニズムをこれらのアプリケーションが備えている必要がある.
3. アプリケーションはMPIの名前の公開機能を無視し，名前を公開するための独自の（可能であればシステム提供の）メカニズムを使用することができる．これにより自由度が高まるが，可搬ではない.

10.4.2 サーバルーチン

サーバは2つのルーチンにより，それ自体を利用できるようにする．まず，MPI_OPEN_PORTを呼び出して接触先となりうるportを確立する必要がある．次に，MPI_COMM_ACCEPTを呼び出してクライアントからのコンタクトを受け付ける必要がある．

MPI_OPEN_PORT(info, port_name)

IN	info	アドレスの確立方法に関する実装固有の情報（ハンドル）
OUT	port_name	新しく確立されたポート（文字列）

```
int MPI_Open_port(MPI_Info info, char *port_name)
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERROR
{void MPI::Open_port(const MPI::Info& info, char* port_name) （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は，サーバがクライアントからの接続を受け付けることができる，port_name文字列にエンコードされたネットワークアドレスを確立する． port_nameは可能であればinfo引数の情報を使用して，システムによって渡される．

MPIはシステム提供のポート名をport_nameにコピーする．port_nameは新しくオープンされたポートを識別し，サーバへのコンタクトのためにクライアントが使用できる．システムが提供可能な文字列の最大サイズはMPI_MAX_PORT_NAMEである．

> **ユーザへのアドバイス** ステムによりポート名がport_nameにコピーされる．アプリケーションはこの値を保持するのに十分なサイズのバッファを渡す必要がある．（**ユーザへのアドバイス終わり**）

port_nameは基本的にはネットワークアドレスである．これは属している通信世界の中で一意であり（実装内容によって決まる），その通信世界内の任意のクライアントが使用できる．例えば，インターネット(host:port)アドレスの場合，インターネット上で一意となる．IBM SPの下位レベルのスイッチアドレスの場合，そのSPで一意となる．

> **実装者へのアドバイス** これらの例は実装を制約するためのものではない．port_nameには，例えば，適切に定義された通信ドメイン内で一意である限り，ユーザ名またはバッチジョブの名前を格納することができる．通信ドメインの規模が大きくなるほど，MPIのクライアント／サーバ機能は有益になる． （**実装者へのアドバイス終わり**）

アドレスの適切な形式は実装で定義される．例えば，インターネットアドレスとしては，ホスト名またはIPアドレス，あるいは実装によりIPアドレスにデコードできる任意のものを使用することができる．ポート名は，MPI_CLOSE_PORTを使用し，システムによる解放が行われた後で再使用することができる．

実装者へのアドバイス port_nameはユーザが手入力することができるよう，読むことが可能で，空白を含まない形式を選択すると有益である． （**実装者へのアドバイス終わり**）

infoはアドレスの確立方法を実装に伝えるのに使用することができる．実装のデフォルトにするにはMPI_INFO_NULLとすることができ，通常はこのように設定される．

MPI_CLOSE_PORT(port_name)

IN	port_name	ポート（文字列）

```
int MPI_Close_port(char *port_name)
MPI_CLOSE_PORT(PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER IERROR
{void MPI::Close_port(const char* port_name) （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数はport_nameで表されるネットワークアドレスを解放する．

MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)

IN	port_name	ポート名（文字列，rootでのみ使用）
IN	info	実装依存の情報（ハンドル， rootでのみ使用）
IN	root	ルートノードのcomm のランク（整数型）
IN	comm	呼び出しが集団であるグループ内コミュニケータ（ハンドル）
OUT	newcomm	リモートグループとしてのクライアントとの グループ間コミュニケータ（ハンドル）

```
int MPI_Comm_accept(char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm)
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
{MPI::Intercomm MPI::Intracomm::Accept(const char* port_name,
            const MPI::Info& info, int root) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_COMM_ACCEPTはクライアントとの通信を確立する．呼び出しコミュニケータにおいて集団的である．クライアントとの通信を可能とするグループ間コミュニケータを返す．

port_nameはMPI_OPEN_PORTの呼び出しにより確立されていなければならない．

infoはACCEPT呼び出しの微調整を可能にするための， 実装で定義される文字列である．

10.4.3 クライアントルーチン

クライアント側のルーチンは1つのみである．

MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)

IN	port_name	ネットワークアドレス（文字列， rootでのみ使用）)
IN	info	実装によって決まる情報（ハンドル， rootでのみ使用）
IN	root	ルートノードのcommのランク（整数型）
IN	comm	呼び出しが集団であるグループ内コミュニケータ（ハンドル）
OUT	newcomm	リモートグループとしてのサーバとのグループ間コミュニケータ（ハンドル）

```
int MPI_Comm_connect(char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm)
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
{MPI::Intercomm MPI::Intracomm::Connect(const char* port_name,
             const MPI::Info& info, int root) const （廃止された呼び出し形式,
             第15.2節を参照）}
```

このルーチンはport_nameによって指定されたサーバとの通信を確立する．呼び出しコミュニケータに対して集団で，リモートグループが MPI_COMM_ACCEPTに参加するグループ間コミュニケータを返す．

名前付きポートがない（あるいはクローズされている）場合，MPI_COMM_CONNECTによりエラークラスMPI_ERR_PORTが発生する．

ポートが存在し，保留中のMPI_COMM_ACCEPTがない場合，実装で定義された時間の経過後に接続がタイムアウトするか，サーバがMPI_COMM_ACCEPTを呼び出して成功する．タイムアウトが発生した場合，MPI_COMM_CONNECTによりエラークラスMPI_ERR_PORTが発生する．

> **実装者へのアドバイス** タイムアウト時間は，短くも長くもできる．しかし，質の高い実装ではサーバが複数のクライアントからの同時の要求を処理できるよう，接続要求がキューイングされる．質の高い実装では，MPI_OPEN_PORT，MPI_COMM_ACCEPT，MPI_COMM_CONNECT のinfo引数を通して，ユーザがタイムアウトとキューイング動作を制御するためのメカニズムを提供することもできる． （**実装者へのアドバイス終わり**）

MPIは接続要求に対するサービスにおける公平さを保証しない．つまり，接続要求は必ずしも開始された順序になるわけではなく，他の接続要求との競合により特定の接続要求が 通らなくなることもある．

port_nameはサーバのアドレスである．これはサーバでMPI_OPEN_PORT によって返される名前と同じでなければならない．ここではある程度の自由が認められる．port_nameの形式が同等であれば，実装ではこれらも受け付けることができる．例えば，port_nameが(hostname:port)の場合，実装では (ip_address:port)も受け付ける．

10.4.4 名前の公開

この節で示すルーチンは名前を公開するためのメカニズムを提供している．(service_name，port_name)のペアはサーバによって公開され，クライアントで取得するにはservice_nameのみを使用すればよい．MPI実装ではservice_name の**有効範囲**，つまりservice_nameが取得できるドメインを定義する．ドメインが空のセットの場合，つまり，クライアントが情報を取得できない場合，名前の公開がサポートされていないことになる．実装では有効範囲の規定方法を文書化する必要がある．質の高い実装では名前公開関数のinfo引数を通して，ユーザがある程度の制御を行えるようにする．例は個々の関数の説明の中で示す．

MPI_PUBLISH_NAME(service_name, info, port_name)

IN	service_name	ポートに付加するサービス名（文字列）
IN	info	実装の情報（ハンドル）
IN	port_name	ポート名（文字列）

```
int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
{void MPI::Publish_name(const char* service_name, const MPI::Info& info,
            const char* port_name) （廃止された呼び出し形式, 第15.2節を参照）}
```

このルーチンは，アプリケーションが既知のservice_nameを使用してシステム提供のport_nameを取得できるように，(port_name, service_name)のペアを公開する．

実装では公開されたサービス名の**有効範囲**，つまりサービス名が一意であるドメイン，逆に言えば(port name, service name) のペアが取得できるドメインを定義する必要がある．例えば，サービス名はジョブ（ジョブは分散オペレーティングシステムまたはバッチスケジューラによって定義される）で一意，マシンで一意，またはケルベロスのレルムで一意とすることができる．有効範囲はMPI_PUBLISH_NAMEのinfo引数によって異なることがある．

MPIでは，1つのport_nameに対して複数のservice_nameを公開することができる．逆に，infoによって規定された有効範囲内でservice_nameがすでに公開されている場合，MPI_PUBLISH_NAMEの動作は未定義である．MPI実装は，MPI_PUBLISH_NAMEのinfo引数のメカニズムを通して，複数のサーバに対して同じ有効範囲内で同じサービスを持つことができる方法を提供する．この場合，実装により定義されるポリシーによって，MPI_LOOKUP_NAMEが複数のポート名のうちのどれを返すかが決まる．

service_nameは有効範囲が制限されていても，実装次第により， port_nameは実装によって使用される，通信世界内で常にグローバルな有効範囲（つまり，グローバルに一意）を持っていることに注意すること．

port_nameはMPI_OPEN_PORTによって確立されていて， MPI_CLOSE_PORTによって削除されていないポートの名前とする必要がある．そうでない場合の結果は未定義である．

実装者へのアドバイス MPI実装は，ユーザが直接アクセスできるネームサービスを使用する場合もある．この場合，MPIによって公開された名前がユーザによって公開された名前と容易に衝突する可能性がある．このような衝突を避けるため，MPI実装では同じサービスを使用するユーザコードと衝突しないようにサービス名を修飾する必要がある．当然，このような名前の修飾はユーザにとっては完全にトランスペアレントである．

実装でネームサーバを使用しようとする場合，次のような状況が問題となるが，避けることは難しい．1台のマシンで複数の“ocean”インスタンスが動作しているとする．サービス名の有効範囲が1つのジョブに限定されている場合，複数のoceanが共存する可能性がある．しかし，実装がサイト全体の有効範囲効範囲を持つ場合，2回目以降のMPI_PUBLISH_NAME呼び出しは全て失敗するため，複数のインスタンスを共存させることはできない．これに対する普遍的な解決策はない．

このような状況に対応するために，質の高い実装では名前を公開するドメインを制限できるようにする必要がある．（**実装者へのアドバイス終わり**）

MPI_UNPUBLISH_NAME(service_name, info, port_name)

IN	service_name	サービス名（文字列）
IN	info	実装固有の情報（ハンドル）
IN	port_name	ポート名（文字列）

```
int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
{void MPI::Unpublish_name(const char* service_name, const MPI::Info& info,
             const char* port_name) （廃止された呼び出し形式, 第15.2節を参照）}
```

このルーチンは，すでに公開されているサービス名を非公開にする． 公開されていないサービス名，またはすでに非公開にされているサービス名を非公開にしようとするとエラーとなり， エラークラスMPI_ERR_SERVICEにより示される．

公開されているサービス名は全て，対応するポートがクローズされる前，および公開プロセスが終了する前に非公開にする必要がある．プロセスがそれ自体の公開していないサービス名を非公開にしようとした場合のMPI_UNPUBLISH_NAMEの動作は，実装によって決まる．

MPI_PUBLISH_NAMEのinfo引数を使用してサービス名の公開方法を実装に伝えた場合， サービス名を非公開にする方法を実装に伝えるための情報がMPI_UNPUBLISH_NAMEに渡されたinfoに含まれていることが実装によって求められることがある．

MPI_LOOKUP_NAME(service_name, info, port_name)

IN	service_name	サービス名（文字列）
IN	info	実装固有の情報（ハンドル）
OUT	port_name	ポート名（文字列）

```
int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
{void MPI::Lookup_name(const char* service_name, const MPI::Info& info,
              char* port_name) （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数は，service_nameを使用してMPI_PUBLISH_NAMEによって公開されたport_nameを取得する．service_nameが公開されていない場合，エラークラスMPI_ERR_NAMEのエラーが発生する．アプリケーションは，使用可能な最大サイズのポート名を保持できるだけの十分なport_nameのバッファを用意する必要がある（上記MPI_OPEN_PORTの説明を参照）．

実装で同じ有効範囲内で同じservice_nameを持つ複数のエントリ持つことができる場合，その実装で規定された方法で特定のport_nameが選択される．

MPI_PUBLISH_NAMEのinfo引数を使用して実装にサービス名の公開方法を伝える場合，MPI_LOOKUP_NAMEでも同じinfo引数が必要になることがある．

10.4.5 予約されたキー値

以下のキー値が予約されている．実装ではこれらのキー値を解釈する必要はないが，キー値を解釈する場合，以下の機能を備えていなければならない．

ip_port portを確立するIPポート番号を含む値（MPI_OPEN_PORT専用）．

ip_address portを確立するIPアドレスを含む値．このアドレスがMPI_OPEN_PORT呼び出しの行われるホストの有効なIPアドレスでない場合，結果は未定義である（MPI_OPEN_PORT専用）．

10.4.6 クライアント／サーバの例

最もシンプルな例 — 完全に可搬

以下の例に，クライアント／サーバインターフェイスを使用する最もシンプルな方法を示す．ここではサービス名はまったく使用しない．

サーバ側：

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
```

```
    MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
    /* do something with intercomm */
```

サーバが端末にポート名を出力する．ユーザはクライアントの起動時にこれを入力する必要がある（これが機能するようにMPI実装でstdin がサポートされているとする）．
クライアント側：

```
    MPI_Comm intercomm;
    char name[MPI_MAX_PORT_NAME];
    printf("enter port name: ");
    gets(name);
    MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
```

Ocean/Atmosphere - 名前の公開に基づく

この例は，“ocean”アプリケーションは海洋大気結合気候モデルの「サーバ」側である．ここでは，MPI実装が名前を公開しているとする．

```
    MPI_Open_port(MPI_INFO_NULL, port_name);
    MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);

    MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
    /* do something with intercomm */
    MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
```

クライアント側：

```
    MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
    MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
                      &intercomm);
```

シンプルなクライアント／サーバの例

これはシンプルな例であり， サーバは一度に1つの接続のみを受け付け，クライアントが切断を要求するまでその接続に対するサービスを行う．サーバは1つのプロセスである．

以下にサーバを示す．これは1つの接続を受け付け，タグが1のメッセージを 受信するまでデータを処理する．タグが0のメッセージを受信するとサーバは終了する．

```
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Comm client;
    MPI_Status status;
    char port_name[MPI_MAX_PORT_NAME];
    double buf[MAX_DATA];
    int    size, again;
```

```
    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size != 1) error(FATAL, "Server too big");
    MPI_Open_port(MPI_INFO_NULL, port_name);
    printf("server available at %s\n",port_name);
    while (1) {
        MPI_Comm_accept( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                         &client );
        again = 1;
        while (again) {
            MPI_Recv( buf, MAX_DATA, MPI_DOUBLE,
                      MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status );
            switch (status.MPI_TAG) {
                case 0: MPI_Comm_free( &client );
                        MPI_Close_port(port_name);
                        MPI_Finalize();
                        return 0;
                case 1: MPI_Comm_disconnect( &client );
                        again = 0;
                        break;
                case 2: /* do something */
                ...
                default:
                        /* Unexpected message type */
                        MPI_Abort( MPI_COMM_WORLD, 1 );
                }
            }
        }
}
```

以下にクライアントを示す.

```
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Comm server;
    double buf[MAX_DATA];
    char port_name[MPI_MAX_PORT_NAME];

    MPI_Init( &argc, &argv );
    strcpy(port_name, argv[1] );/* assume server's name is cmd-line arg */

    MPI_Comm_connect( port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                      &server );

    while (!done) {
        tag = 2; /* Action to perform */
        MPI_Send( buf, n, MPI_DOUBLE, 0, tag, server );
        /* etc */
        }
    MPI_Send( buf, 0, MPI_DOUBLE, 0, 1, server );
    MPI_Comm_disconnect( &server );
    MPI_Finalize();
    return 0;
}
```

10.5 その他の機能

10.5.1 ユニバースサイズ

多くの「動的」なMPIアプリケーションが，アプリケーションの実行前にリソースが割り当てされている静的なランタイム環境に存在すると考えられる．ユーザ（または，恐らくはバッチシステム）がこのような擬似静的アプリケーションを使用する場合，通常は起動する多数のプロセスと想定されるプロセスの総数を指定する．アプリケーションが知る必要のあるのは，存在するスロットの数，つまり，スポーンすべきプロセスの数である．

MPIには，可搬な方法でアプリケーションがこの情報を取得するための MPI_COMM_WORLD，MPI_UNIVERSE_SIZEの属性が用意されている．この属性は，想定されるプロセスの総数を示す．Fortran言語では，属性は整数値である．C言語では，属性は整数値へのポインタである．通常，アプリケーションはMPI_UNIVERSE_SIZEからMPI_COMM_WORLDのサイズを差し引いて，スポーンすべきプロセスの数を算出する．MPI_UNIVERSE_SIZEはMPI_INITで初期化され，MPIによって変更されない．定義されると，MPI_COMM_WORLD の全てのプロセスで同じ値となる．MPI_UNIVERSE_SIZEはMPIで規定されない方法により，アプリケーション起動メカニズムによって決定される（MPI_COMM_WORLDのサイズはこのようなパラメータの別の例である）．

MPI_UNIVERSE_SIZEは，例えば以下の方法によって設定可能である．

- MPIプロセスを起動する`-universe_size`プログラムの引数
- アプリケーションに割り当てられているプロセッサの数を知るための，バッチスケジューラとの自動的なやりとり
- ユーザが設定した環境変数
- info引数によりMPI_COMM_SPAWNに渡される追加情報

実装ではMPI_UNIVERSE_SIZEの設定方法を文書化する必要がある．実装で MPI_UNIVERSE_SIZEを設定する機能がサポートされていない場合がある．この場合，属性MPI_UNIVERSE_SIZEは設定されない．

MPI_UNIVERSE_SIZEは必ずしも厳格な制限ではなく，推奨値である． 例えば，ある実装では，要求すればアプリケーションでプロセッサごとに50のプロセスをスポーンすることができる．しかし，プロセッサごとに1つのプロセスのみを生成したいという場合もある．

MPI_UNIVERSE_SIZEはアプリケーションの起動時に指定されていると仮定され，基本的にユーザが（`mpiexec`などの，MPIプロセス起動メカニズムを通して）アプリケーションに重要なランタイム情報を渡すことができる可搬なメカニズムである．ランタイム環境とのやりとりは必要ないことに注意すること．アプリケーションの実行中にランタイム環境でサイズが変更された場合でも，MPI_UNIVERSE_SIZEは更新されず，アプリケーションはランタイムシステムと直接やりとりすることにより変更を知る必要がある．

10.5.2 シングルトンMPI_INIT

質の高い実装では，MPI_INITを呼び出すことにより，任意のプロセス（「並列アプリケーション」メカニズムによって起動されていないものも含む）をMPIプロセスにすることができる．このようなプロセスはMPI_COMM_ACCEPTおよびMPI_COMM_CONNECTルーチンを使用して他のMPIプロセスと接続したり，他のMPIプロセスをスポーンしたりすることができる．MPIではこの動作を必須としていないが，技術的に実現可能であれば強く推奨している．

実装者へのアドバイス 同じMPI_COMM_WORLDに属しているMPIプロセスを起動するには，特別な調整が必要となる．例えば，各プロセスを「同時に」起動する必要があり，各プロセスの通信を確立するためのメカニズムが必要である．ユーザまたはオペレーティングシステムが，単にプロセスを起動する以上の特別な手順を実行する必要がある．

アプリケーションでMPI_INITが開始されたら，これらの特別な手順が実行されたかどうかをアプリケーションで明確に確認できなければならない．プロセスでMPI_INITが開始され，特別な手順が実行されていない（つまり，他のプロセスによりMPI_COMM_WORLDを生成するための情報が渡されていない）ことが確認された場合，正常に，シングルトンMPIプログラム，つまりMPI_COMM_WORLDがサイズ1であるプログラム，を生成する．

実装によっては，MPIが「MPI環境」なしでは機能できない場合もある．例えば，MPIがデーモンの動作を必要としたり，MPIがMPPのフロントエンドでは全く動作しなかったりすることがある．この場合，MPI実装は次のいずれかとなる．

1. 環境を作成する（デーモンの起動など）．
2. 環境を生成できず，環境が予め起動されていない場合，エラーが発生する．

質の高い実装ではシングルトンMPIプロセスの生成を試み，エラーは発生しない．（**実装者へのアドバイス終わり**）

10.5.3 MPI_APPNUM

MPI_COMM_WORLDには定義済みの属性MPI_APPNUMがある． Fortran言語では，属性は整数値である．C言語では，属性は整数値へのポインタである．プロセスがMPI_COMM_SPAWN_MULTIPLEによってスポーンされた場合，MPI_APPNUMは現在のプロセスを生成したコマンドの番号である．番号は0から始まる．プロセスがMPI_COMM_SPAWNにより生成された場合，MPI_APPNUM は0となる．

また，プロセスがスポーン呼び出しによってではなく，複数のプロセス指定の処理が可能な実装固有の起動メカニズムによって起動された場合， MPI_APPNUMには該当するプロセス指定の番号が設定されなければならない．特に，以下により起動された場合，

```
mpiexec spec0 [: spec1 : spec2 : ...]
```

MPI_APPNUMには該当する指定の番号が設定されなければならない．

アプリケーションがMPI_COMM_SPAWNまたはMPI_COMM_SPAWN_MULTIPLEによってスポーンされておらず， MPI_APPNUMが実装固有の起動メカニズムにおいて意味を持たない場合，MPI_APPNUMは設定されない．

MPI実装はオプションにより，info引数を通してMPI_APPNUMの値を上書きするためのメカニズムを提供することができる．MPIは全てのSPAWN呼び出し用に以下のキーを予約している．

appnum 値には，子のMPI_APPNUMのデフォルト値を上書きする整数が格納される．

根拠 1つのアプリケーションが起動されたとき，MPI_COMM_WORLDのサイズを調べることにより，いくつのプロセスが存在するかを確認することができる．複数のSPMDサブアプリケーションで構成されるアプリケーションには，いくつのサブアプリケーションが存在するか，プロセスがどのサブアプリケーションに属しているかを確認する方法はない．特別な場合にはこれを確認する方法があるが，一般的なメカニズムはない．MPI_APPNUMはこのための一般的なメカニズムを提供する．（**根拠の終わり**）

10.5.4 接続の解放

接続される前のクライアントとサーバは，はそれぞれ独立したMPIアプリケーションである．一方のエラーが他方に影響することはない．MPI_COMM_CONNECTとMPI_COMM_ACCEPTを使用して接続が確立されると，一方のエラーが他方に影響を及ぼすことがある．一方のエラーが他方に影響しないように．クライアントとサーバが切断できることが望ましい．同様に，子のエラーが親に影響したり，親のエラーが子に影響したりすることがないように，親と子が切断できることが望ましい場合がある．

- 2つのプロセスは，間に（直接または間接の）通信パスがある場合に**接続される**．より正確には次のようになる．
  1. 2つのプロセスは以下のいずれかの場合に接続される．
     (a) 同じコミュニケータに属している場合（MPI_COMM_WORLDを含むグループ間またはグループ内コミュニケータ），**または**
     (b) MPI_COMM_DISCONNECTではなくMPI_COMM_FREEによって解放されたコミュニケータに属していた場合，**または**
     (c) 同じウィンドウまたはファイルハンドルのグループに属している場合
  2. AがBに接続され，BがCに接続されている場合，AはCに接続されている．
- 2つのプロセスは，接続されていない場合は**切断されている**（**独立した状態**でもある）．
- 上記の定義により接続は過渡的な属性であり， MPIプロセスの世界を切断された（独立した）プロセスの集合（同等のクラス）に分割する．

- 接続されているが同じMPI_COMM_WORLDを共有していないプロセスは，間の通信パスがMPI_COMM_DISCONNECTによって遮断された場合は切断された（独立した）状態になることがある．

他の章でのMPIルーチンに以下の追加規則が適用される．

- MPI_FINALIZEは接続されたプロセスの集合に対して集団的である．
- MPI_ABORTは独立したプロセスを中断させない．（comm引数を無視して）呼び出し元のMPI_COMM_WORLDの 全てのプロセスを中断させる可能性はある．また，commのプロセスのみを中断させるよう「最大限要求する」が，接続されたプロセスも中断させることがある．
- MPI_FINALIZEを呼び出さずにプロセスを終了した場合，独立したプロセスは影響を受けないが，接続されたプロセスへの影響は未定義である．

MPI_COMM_DISCONNECT(comm)

INOUT	comm	コミュニケータ（ハンドル）

```
int MPI_Comm_disconnect(MPI_Comm *comm)
MPI_COMM_DISCONNECT(COMM, IERROR)
    INTEGER COMM, IERROR
{void MPI::Comm::Disconnect()  （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数はcommの保留中の全ての通信が内部的に完了するのを待ち合わせ，コミュニケータのオブジェクトを解放し，ハンドルをMPI_COMM_NULLに設定する．これは集団操作である．

コミュニケータMPI_COMM_WORLDまたはMPI_COMM_SELFを使用して呼び出してはいけない．

バッファリングされたデータが送信先に送信されるように，MPI_COMM_DISCONNECTの呼び出しは通信が完了し，一致している場合のみ行うことができる．この要件はMPI_FINALIZEの場合と同様である．

MPI_COMM_DISCONNECTの動作はMPI_COMM_FREEと同じだが，保留中の通信が内部的に完了するのを待ち合わせ，切断されたプロセスの動作に関する保証を有効にする点が異なる．

ユーザへのアドバイス 2つのプロセスを切断するには，MPI_COMM_DISCONNECT， MPI_WIN_FREE，およびMPI_FILE_CLOSEを呼び出して 2つのプロセスの間の全ての通信パスを削除する必要がある場合がある．2つのプロセスを完全に独立した状態にする前に，複数のコミュニケータを切断する（あるいは複数のウィンドウまたはファイルを解放する）必要がある場合もある．（**ユーザへのアドバイス終わり**）

根拠 代わりにMPI_COMM_FREEを使用することもできるのは便利だが，この関数は明示的に保留中の通信の完了の待ち合わせを行わない． （**根拠の終わり**）

10.5.5 MPI通信のもう1つの確立方法

MPI_COMM_JOIN(fd, intercomm)

IN	fd	ソケットファイル記述子
OUT	intercomm	新規グループ間コミュニケータ（ハンドル）

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
    INTEGER FD, INTERCOMM, IERROR
{static MPI::Intercomm MPI::Comm::Join(const int fd) （廃止された呼び出し形式,
              第15.2節を参照）}
```

MPI_COMM_JOINは，Berkeley Socketインターフェイス文献[33, 37]をサポートする環境に存在するMPI実装を対象としている．Berkeley Socketをサポートしない環境に存在する実装では，MPI_COMM_JOINのエントリポイントを用意し，MPI_COMM_NULLを返す必要がある．

この呼び出しは，ソケットによって接続される2つのMPIプロセスの和集合からグループ間コミュニケータを生成する．ローカルおよびリモートプロセスが実装内容によって決まる同じMPI通信世界にアクセスできる場合，MPI_COMM_JOINは正常に実行される．

> **ユーザへのアドバイス** MPI実装は，共有メモリセグメントや特別なスイッチなど，MPI通信用の特定の通信媒体を必要とすることがある．この場合，これらを接続するソケットがあり，これらが同じMPI実装を使用している場合でも，2つのプロセスが正常に参加できない場合がある．（**ユーザへのアドバイス終わり**）

> **実装者へのアドバイス** 質の高い実装では，望ましい媒体が利用できない場合，低速の媒体を利用して通信を確立しようとする．実装でこれが行われない場合，ソケット（特にソケットがTCP接続の場合）で使用される媒体によりMPI通信が行えない理由を明文化する必要がある．（**実装者へのアドバイス終わり**）

fdはSOCK_STREAM（信頼性の高い双方向のバイトストリーム接続）型のソケットを表わすファイル記述子である．ソケットに対しては，SIGIOによるノンブロッキング入出力および非同期通知を有効にしてはならない．ソケットは接続状態でなければならない．ソケットは， MPI_COMM_JOINの呼び出し時に静止している必要がある（下記を参照）．それは標準のソケットAPI呼び出しを使用してソケットを生成するアプリケーションの責任である．

MPI_COMM_JOINは，ソケットの両端のプロセスによって呼び出されなければならない．MPI_COMM_JOINは，両方のプロセスでの呼び出されるまで戻らない．2つのプロセスはローカルプロセス/リモートプロセスと呼ばれる．

MPIはソケットを使用してそれ自体のためのグループ間コミュニケータの生成をブートストラップ方式で行う．MPI_COMM_JOINから戻ると，ファイル記述子がオープンされ，静止状態となる（下記を参照）．

MPIがグループ間コミュニケータを生成できないが，ソケットを最初の状態のままにしておくことができ，保留中の通信がないのであれば，処理は成功で， intercommにMPI_COMM_NULLが設定される．

MPI_COMM_JOINが呼び出される前とMPI_COMM_JOINが戻った後は，ソケットが静止していなければならない．より具体的に言うと， MPI_COMM_JOINの開始時にソケットのreadは，リモートプロセスがMPI_COMM_JOINを呼び出す前にソケットに書き込まれたデータの読み取りを行わない．MPI_COMM_JOINの終了時に read はリモートプロセスがMPI_COMM_JOINから戻る前にソケットに書き込んだデータの読み取りを行わない．最初の状態を保証するのはアプリケーションの責任で，2番目の状態を保証するのはMPI実装の責任である．マルチスレッドアプリケーションでは，あるスレッドがMPI_COMM_JOINを呼び出している間に別のスレッドがソケットにアクセスしないこと，つまりMPI_COMM_JOINの同時呼び出しが行われないことを保証する必要がある．

> **実装者へのアドバイス** MPIは新しいコミュニケータ内のMPIメッセージのために，利用可能な任意の通信パスを自由に使用でき，ソケットは最初のハンドシェイキイングにのみ使用される． （**実装者へのアドバイス終わり**）

MPI_COMM_JOINはこのために非MPI通信を使用する．非MPI 通信と保留中のMPI通信とのやりとりの結果は未定義となる．そのため，接続された2つのプロセスでMPI_COMM_JOINを呼び出した結果も未定義となる（接続されたプロセスの定義については， 343ページの第10.5.4節を参照）．

返されたコミュニケータは，通常のMPIコミュニケータ作成メカニズムを通して，追加のプロセスとのMPI 通信を確立するために使用できる．

第11章

片方向通信

11.1 はじめに

リモートメモリアクセス(RMA)は，1つのプロセスが，送信側と受信側の両方の全ての通信パラメータを指定可能にすることによって，MPIの通信メカニズムを拡張する．この通信モードは，プロセス間のデータの分布が固定されている（またはゆっくりとしか変化しない），しかしデータのアクセス箇所を動的に変えるアプリケーションにおいて，そのコーディングを容易にする．このような場合，各プロセスが，他のプロセスに置かれた，アクセスまたは更新する必要のあるデータが何かを計算することは可能である．しかしプロセスには，自身のメモリ上のどこのデータがリモートプロセスによってアクセスまたは更新されるかわからないかもしれないし，アクセスまたは更新してくるリモートプロセスの身元さえわからないかもしれない．したがって，その転送パラメータは片側でのみ，全て利用可能である．正規の送受信では送信側と受信側で操作をマッチさせる必要がある．マッチする操作を発行するには，アプリケーションが送受信の転送パラメータを配布する必要がある．このためには，全てのプロセスが時間のかかるグローバル計算に参加したり，全てのプロセスが受信すべき潜在的な通信要求を定期的にポーリングしその通信要求に従って動作したりする必要があるかもしれない． RMA通信メカニズムを利用すると，グローバル計算や明示的なポーリングの必要性を避けることができる．この一般的な例は`A = B(map)`という形式の代入の実行である．ここで，`map`は置換ベクトルであり，`A`，`B`，および`map`は同一の方法で分散されている場合である．

メッセージ通信では， 送信側から受信側へのデータの**通信**， 送信側と受信側の**同期**という2つの効果を達成する．RMA設計はこれらの2つの関数を分離する．通信呼び出しとして，MPI_PUT（リモート書き込み），MPI_GET（リモート読み取り），およびMPI_ACCUMULATE（リモート更新）という3種類が用意されている．同期のタイプに応じて，多数の同期呼び出しも用意されている．設計は疎結合メモリシステムのものと同様で，ユーザが同期呼び出しを使用してメモリアクセスの正しい順序を指定する必要があり，効率化のため，同期呼び出しが発生するまで実装は通信操作を遅延させることができる．

RMA関数の設計を利用することにより，一貫性のある共有メモリもしくは一貫性の

ない共有メモリ，DMAエンジン，ハードウェアでサポートされるプット／ゲット操作，通信コプロセッサなど，多くの点で実装者は各種プラットフォームで提供される高速通信メカニズムを利用することができる．最もよく使用されるRMA通信メカニズムをメッセージ通信の上位に階層化することができる．しかし，分散メモリ環境において，一部のRMA関数では非同期通信エージェント（ハンドラ，スレッドなど）をサポートする必要がある．

ここでは呼び出しを行うプロセスのことを**オリジン**と呼び，メモリにアクセスするプロセスのことを**ターゲット**と呼ぶ．そのため，プット操作では，送信元=オリジン，送信先=ターゲットとなり，ゲット操作では，送信元=ターゲット，送信先=オリジンとなる．

11.2 初期化

11.2.1 ウィンドウの生成

この初期化操作を使用すると，グループ内コミュニケータのグループ内の各プロセスが，リモートプロセスによるアクセスを許可したメモリの「ウィンドウ」を指定することができる．この初期化操作は集団操作である．ある関連呼び出しでは，ウィンドウの集合を所有しかつそれにアクセスするプロセスのグループを表現する不可視オブジェクトを返す．別の呼び出しは，初期化呼び出しで指定されたのと同じ各ウィンドウの属性を返す．

MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)

IN	base	ウィンドウの先頭アドレス（選択型）
IN	size	ウィンドウのサイズ（バイト単位）（非負の整数型）
IN	disp_unit	変位のローカルユニットサイズ（バイト単位）（正の整数型）
IN	info	info引数（ハンドル）
IN	comm	コミュニケータ（ハンドル）
OUT	win	呼び出しによって返されるウィンドウオブジェクト（ハンドル）

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, MPI_Win *win)
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
{static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int
             disp_unit, const MPI::Info& info, const MPI::Intracomm& comm)
             (廃止された呼び出し形式, 第15.2節を参照）}
```

これはcommのグループ内の全てのプロセスによって実行される集団呼び出しである．RMA操作を実行するためにこれらのプロセスで使用できるウィンドウオブジェクトを返

す．各プロセスは，commのグループ内のプロセスによるRMAアクセス用にエクスポーズする既存メモリのウィンドウを指定する．ウィンドウはアドレスbaseで始まるsizeバイトで構成される．size = 0を指定することにより，プロセスはメモリをエクスポーズしないことを選択してよい．

変位ユニット引数はRMA操作でのアドレス演算を容易にするために用意されている．RMA操作のターゲットの変位引数はdisp_unit倍される．ここでdisp_unitは，ターゲットプロセスによってウィンドウ生成時に指定された係数disp_unitである．

根拠 ウィンドウのサイズは，4GBを超えるアドレス空間のウィンドウが格納できるように，アドレスサイズの整数型を使用して指定する（物理的なメモリサイズが4GB未満の場合でもアドレスが連続していない場合，アドレスの範囲は4GBを超えることがある）．（**根拠の終わり**）

ユーザへのアドバイス 型typeが要素の配列で構成されるウィンドウの場合，disp_unitに対するよくみられる選択肢は1（スケーリングなし）と，（C言語の構文で）`sizeof(type)`とである．後者の選択肢の場合，異機種環境においても，RMA呼び出しで配列の添字を使用し，バイトの変位に正しくスケールさせることができる．（**ユーザへのアドバイス終わり**）

info引数は，そのウィンドウの期待される利用パターンについての最適化のヒントを，ランタイムに対し提供する．以下のinfoキーが定義済みである．

no_locks — trueに設定すると，ローカルウィンドウが（MPI_WIN_LOCKの呼び出しを使って）ロックされることはないと，実装は仮定してよい．これは次のことを意味する．つまりこのウィンドウは3者間通信に使われることはなく，非同期エージェントの動作なしに（あるいは動作を削減して），RMAを実装することができる．

commグループの様々なプロセスで，領域，サイズ，変位ユニット，およびinfo引数のまったく異なる対象ウィンドウを指定することができる．特定のプロセスへの全てのゲット，プット，およびアキュムレートアクセスが特定の対象ウィンドウに適合している限り，問題はない．メモリ内の同じ領域がそれぞれ異なるウィンドウオブジェクトに関連付けられ，複数のウィンドウに提示されることがある．しかし，別々であるが，重複するウィンドウへの同時並行的な通信は誤った結果を導くことがある．

ユーザへのアドバイス ウィンドウはプロセスメモリの任意の部分で生成することができる．しかし，システムによっては，MPI_ALLOC_MEM（284ページの第8.2節を参照）によって割り当てた方がメモリ内のウィンドウの性能が高くなる．また，システムによっては，ウィンドウの境界を「自然な」境界（ワード，ダブルワード，キャッシュライン，ページフレームなど）に合わせると性能が向上する．（**ユーザへのアドバイス終わり**）

実装者へのアドバイス RMA操作でメモリ領域ごとに異なるメカニズムを使用する場合（例えば，共有メモリセグメントにはロード／ストア，プライベート

メモリには非同期ハンドラのように)， MPI_WIN_CREATE呼び出しではウィンドウで使用するメモリのタイプを特定する必要がある．このために，MPIは，MPI_ALLOC_MEMまたは他の実装固有のメカニズムによって割り当てられるメモリセグメントのリストを，割り当てられたメモリセグメントのタイプに関する情報と一緒に内部管理する．MPI_WIN_CREATEの呼び出し時に，MPIは，どのメモリセグメントがそれぞれのウィンドウを包含しているかを検査し，それに基づいてRMA操作に使用すべきメカニズムを決定する．

ベンダーは，片方向通信での使用に適したメモリ領域を割り当てたり，特定したりするための実装固有のメカニズムを追加してよい．特に，このようなメカニズムは，静的変数をこのような望ましい領域に配置するのに使用できる．

実装者はウィンドウのアライメントに関して性能上の影響を明文化する必要がある． （**実装者へのアドバイス終わり**）

MPI_WIN_FREE(win)

INOUT	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_free(MPI_Win *win)
MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR
{void MPI::Win::Free() （廃止された呼び出し形式, 第15.2節を参照）}
```

ウィンドウオブジェクトwinを解放し，nullハンドル（MPI_WIN_NULLと等しい）を返す．これは，winに関連付けられたグループ内の全てのプロセスによって実行される集団呼び出しである．プロセスがウィンドウwin上のRMA通信への関与を完了した後でしか，MPI_WIN_FREE(win)を呼び出すことはできない．つまり，プロセスがMPI_WIN_FENCEを呼び出したか，前のMPI_WIN_POSTの呼び出しにマッチするMPI_WIN_WAITを呼び出したか，前のMPI_WIN_STARTの呼び出しにマッチするMPI_WIN_COMPLETEを呼び出したか，前のMPI_WIN_LOCKの呼び出しにマッチするMPI_WIN_UNLOCKを呼び出した後となる．呼び出しが戻った時点で，ウィンドウメモリを解放することができる．

実装者へのアドバイス MPI_WIN_FREEはバリア同期を要求する．つまり，winのグループ内の全てのプロセスが解放を呼び出すまで，どのプロセスも解放から復帰しない可能性がある．これにより，ウィンドウの解放後にプロセスがリモートウィンドウに（例えば，lock/unlockにより）アクセスしないようにすることができる．（**実装者へのアドバイス終わり**）

11.2.2 ウィンドウの属性

ウィンドウの生成時に，ウィンドウと一緒に以下の3つの属性がキャッシングされる．

MPI_WIN_BASE	ウィンドウのベースアドレス
MPI_WIN_SIZE	ウィンドウのサイズ（バイト単位）
MPI_WIN_DISP_UNIT	ウィンドウに関連付けられている変位ユニット

C言語では，
MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag)，
MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag)，および
MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag)
の呼び出しでそれぞれ，baseにウィンドウwinの開始点のポインタが返され，sizeおよびdisp_unitのポインタの指す場所に，そのウィンドウのサイズと変位ユニットが返される．C++言語でも同様である．

Fortran言語では，
MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror)，
MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror)，および
MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror)
の呼び出しでbase，size，およびdisp_unitにそれぞれウィンドウwinのベースアドレス（の整数型表現），サイズ，および変位ユニットが返される．（ウィンドウ属性のアクセス関数については，241ページの第6.7.3節で定義する．）

その他の「ウィンドウ属性」，つまりウィンドウに結びつけられたプロセスのグループは以下の呼び出しを使用して取得することができる．

MPI_WIN_GET_GROUP(win, group)

IN	win	ウィンドウオブジェクト（ハンドル）
OUT	group	ウィンドウへのアクセスを共有するプロセスのグループ（ハンドル）

```
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
    INTEGER WIN, GROUP, IERROR
{MPI::Group MPI::Win::Get_group() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_WIN_GET_GROUPは，winに関連付けられたウィンドウを生成するのに使用されるコミュニケータのグループの複製を返す．グループはgroupに返される．

11.3 通信呼び出し

MPIは3種類のRMA通信呼び出しをサポートしている．MPI_PUTはデータを呼び出し元のメモリ（オリジン）からターゲットのメモリに転送し，MPI_GETはターゲットのメモリから呼び出し元のメモリにデータを転送し，MPI_ACCUMULATEは，呼び出し元のメモリから送信された値をこれらの領域に追加するなど，ターゲットのメモリ内の領域の更新を行う．これらの操作は**ノンブロッキング**で，呼び出しは転送を開始するが，呼び出しが戻った後も転送が継続されることがある．転送は，関与するウィンドウオブジェクトに対して呼び出し元が後続の**同期**呼び出しを発行した時点で，オリジンとターゲ

ットの両方で完了する．これらの同期呼び出しについては，359ページの第11.4節で説明する．

RMA呼び出しのローカル通信バッファは更新しないようにする必要があり，ゲット呼び出しのローカル通信バッファはRMA呼び出しの後，後続の同期呼び出しが完了するまではアクセスしないようにする必要がある．

ウィンドウ内の同じメモリ領域への同時並行的な競合するアクセスは誤りである．例えば，ある領域がプットまたはアキュムレート操作により更新された場合，ロードまたは別のRMA操作によるこの領域へのアクセスは，ターゲットで更新操作が完了するまではできない．この規則には1つだけ例外があり，同じ領域は複数の同時並行的なアキュムレート呼び出しによって更新できる．この場合の結果はまるで，ある順序でこれらの更新が起こったかのようになる．また，ウィンドウはプットまたはアキュムレート操作と，ローカルなストア操作により同時並行的に更新することはできない．これは，これらの2つの更新がウィンドウ内の別の領域にアクセスする場合でも同様である．この最後の制限により，多くのシステムでより効率的なRMA操作の実装が可能になる．これらの制限については，377ページの第11.7節で詳しく説明する．

これらの呼び出しでは，オリジンとターゲットで通信バッファを指定するのに一般的なデータ型引数を使用する．そのため，転送操作は送信元でデータをギャザーし，送信先でスキャッタすることもできる．しかし，両方の通信バッファを指定する全ての引数は呼び出し元で渡される．

これらの3つの呼び出し全てで，ターゲットプロセスはオリジンプロセスと同じであってよい．つまり，そのプロセスのメモリ上のデータの移動にRMA操作を使ってよい．

> **根拠** サポートする「自己通信」の選択は，メッセージ通信の場合と同様である．これによりコードをシンプルにすることができ，アキュムレート操作で，ローカル変数に対するアトミックな更新を可能にするのに非常に便利である． （**根拠の終わり**）

MPI_PROC_NULLはMPI RMA呼び出しのMPI_ACCUMULATE，MPI_GET，およびMPI_PUTにおける有効なターゲットランクである．効果はMPIの1対1通信のMPI_PROC_NULLの場合と同様である．ランクMPI_PROC_NULLをもつどんなRMA操作の後にも，それでもなお，そのアクセスエポックを開始した同期メソッドでアクセスエポックを終わらせる必要がある．

11.3.1 プット

プット操作の実行は，オリジンプロセスで送信を実行し，ターゲットプロセスでそれにマッチする受信を実行するのに似ている．明らかな違いは，全ての引数が1つの呼び出しで渡され，その呼び出しはオリジンプロセスによって実行される点である．

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

IN	origin_addr	オリジンバッファの先頭アドレス（選択型）
IN	origin_count	オリジンバッファのエントリの数（非負の整数型）
IN	origin_datatype	オリジンバッファの各エントリのデータ型（ハンドル）
IN	target_rank	ターゲットのランク（非負の整数型）
IN	target_disp	ウィンドウの先頭からターゲットバッファまでの変位（非負の整数型）
IN	target_count	ターゲットバッファのエントリの数（非負の整数型）
IN	target_datatype	ターゲットバッファの各エントリのデータ型（ハンドル）
IN	win	通信に使用するウィンドウオブジェクト（ハンドル）

```
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
{void MPI::Win::Put(const void* origin_addr, int origin_count, const
             MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
             target_disp, int target_count, const MPI::Datatype&
             target_datatype) const （廃止された呼び出し形式, 第15.2節を参照）}
```

オリジンプロセス[1]のアドレスorigin_addrで始まる，origin_datatypeで指定された型のorigin_count個の連続するエントリをwin，target_rankのペアで指定されたターゲットプロセス[2]に転送する．データはターゲットバッファのアドレスtarget_addr = window_base + target_disp×disp_unitに書き込まれる．ここで，window_baseとdisp_unitはウィンドウの初期化時にターゲットプロセスによって指定されたベースアドレスとウィンドウの変位ユニットである．

ターゲットバッファは引数target_countとtarget_datatypeによって指定される．

データ転送は，オリジンプロセスが引数 origin_addr，origin_count，origin_datatype，target_rank，tag，およびcommにより送信操作を実行し，ターゲットプロセスが引数target_addr，target_count，target_datatype，source，tag，commにより受信操作を実行した場合に行われる処理と同じである．target_addrは上記のように算出したターゲットバッファのアドレスで，commはwinのグループのコミュニケータである．

この通信についても，類似のメッセージ通信と同じ制限に従う必要がある．target_datatypeではターゲットバッファの重複するエントリを指定できない．送信される

[1]訳者註：原文 origin node
[2]訳者註：原文 target node

メッセージは，切り詰めることなくターゲットバッファに収まらなければならない．さらに，ターゲットバッファは対象ウィンドウにも適合する必要がある．

target_datatype引数は，オリジンプロセスで定義されたデータ型オブジェクトのハンドルである．しかし，このオブジェクトはターゲットプロセスで解釈され，結果は，オリジンプロセスで定義に使用した呼び出しと同じ順序により，ターゲットのデータ型オブジェクトがターゲットプロセスで定義されたのと同様になる．ターゲットのデータ型に格納されるのは，絶対アドレスではなく，相対変位のみでなければならない．ゲットおよびアキュムレート操作の場合も，これと同様である．

ユーザへのアドバイス target_datatype引数はターゲットプロセスのメモリ内のデータのレイアウトを定義するものではあるが，オリジンプロセスで定義されたデータ型オブジェクトのハンドルである．可搬なデータ型のみを使用すれば（可搬なデータ型については，13ページの第2.4節で定義），同一機種環境でも異機種環境でも問題は起こらない．

プット転送の性能は，ウィンドウの領域や，オリジンバッファおよびターゲットバッファの形状と領域の選択により，システムによっては大きな影響を受けることがある．例えば，MPI_ALLOC_MEMによって割り当てられたメモリ内の対象ウィンドウへの転送は共有メモリシステムの方が速かったり，連続バッファからの転送が（全部ではないにしても）大部分のシステムで速かったり，通信バッファのアライメントが性能に影響したりすることもある． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 高品質な実装では，そのプロセスによってエクスポーズされたウィンドウの外側のメモリへのリモートアクセスを防ごうとする．これはデバッグ面，およびRMAを使用するクライアント／サーバのコードに対する保護面の両方の目的をもつ．このような高品質な実装では，できるだけ各RMA呼び出し時にウィンドウの境界をチェックし，境界違反が見つかった場合，呼び出し元でMPI例外を発生させる．つまり，オリジンでこの例外条件をチェックすることが可能であることに注意すること．もちろん，こうしたチェックで達成される安全性は，そのチェックのための追加コストと比較検討されなければならない． （**実装者へのアドバイス終わり**）

11.3.2 ゲット

MPI_GET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

OUT	origin_addr	オリジンバッファの先頭アドレス（選択型）
IN	origin_count	オリジンバッファのエントリの数（非負の整数型）
IN	origin_datatype	オリジンバッファの各エントリのデータ型（ハンドル）
IN	target_rank	ターゲットのランク（非負の整数型）
IN	target_disp	ウィンドウの先頭からターゲットバッファの先頭までの変位（非負の整数型）
IN	target_count	ターゲットバッファのエントリの数（非負の整数型）
IN	target_datatype	ターゲットバッファの各エントリのデータ型（ハンドル）
IN	win	通信に使用するウィンドウオブジェクト（ハンドル）

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)
MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
{void MPI::Win::Get(void *origin_addr, int origin_count, const
             MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
             target_disp, int target_count, const MPI::Datatype&
             target_datatype) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_PUTと同様であるが，データ転送の方向が逆である．データはターゲットのメモリからオリジンのメモリにコピーされる．origin_datatypeではオリジンバッファの重複するエントリを指定できない．ターゲットバッファは対象ウィンドウ内からはみ出してはならないし，コピーされるデータは切り詰めることなくオリジンバッファに収まらなければならない．

11.3.3 例

例 11.1 一般的な間接的な代入A = B(map)の実装方法を示す．ここで，A，B，およびmapは同一方法で分散され，mapは置換である．簡単化のため，ここでは等サイズのブロックによるブロック分割を仮定する．

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
```

```
1
2   INTEGER otype(p), oindex(m),   & ! used to construct origin datatypes
3        ttype(p), tindex(m),      & ! used to construct target datatypes
4        count(p), total(p),       &
5        win, ierr
    INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal
6
7   ! This part does the work that depends on the locations of B.
8   ! Can be reused while this does not change
9
10  CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
11  CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,   &
                        comm, win, ierr)
12
13  ! This part does the work that depends on the value of map and
14  ! the locations of the arrays.
15  ! Can be reused while these do not change
16
17  ! Compute number of entries to be received from each process
18
19  DO i=1,p
20    count(i) = 0
21  END DO
22  DO i=1,m
23    j = map(i)/m+1
24    count(j) = count(j)+1
25  END DO
26
27  total(1) = 0
28  DO i=2,p
29    total(i) = total(i-1) + count(i-1)
30  END DO
31
32  DO i=1,p
33    count(i) = 0
34  END DO
35
36  ! compute origin and target indices of entries.
37  ! entry i at current process is received from location
38  ! k at process (j-1), where map(i) = (j-1)*m + (k-1),
39  ! j = 1..p and k = 1..m
40
41  DO i=1,m
42    j = map(i)/m+1
43    k = MOD(map(i),m)+1
44    count(j) = count(j)+1
45    oindex(total(j) + count(j)) = i
46    tindex(total(j) + count(j)) = k
47  END DO
48
    ! create origin and target datatypes for each get operation
    DO i=1,p
      CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, oindex(total(i)+1),   &
                                         MPI_REAL, otype(i), ierr)
      CALL MPI_TYPE_COMMIT(otype(i), ierr)
      CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, tindex(total(i)+1),   &
                                         MPI_REAL, ttype(i), ierr)
      CALL MPI_TYPE_COMMIT(ttype(i), ierr)
    END DO
```

```
! this part does the assignment itself
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,p
  CALL MPI_GET(A, 1, otype(i), i-1, 0, 1, ttype(i), win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
DO i=1,p
  CALL MPI_TYPE_FREE(otype(i), ierr)
  CALL MPI_TYPE_FREE(ttype(i), ierr)
END DO
RETURN
END
```

例 11.2 よりシンプルな記述例では，ターゲットバッファ用にデータ型を作成する必要はない．しかし，以下のように，エントリごとに別のゲット呼び出しが必要となる．このコードは非常にシンプルであるが，通常は大きな配列ではあまり効率はよくない．

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i),m)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

11.3.4 アキュムレート関数

プット操作において，そこにあったデータを置き換えるのではなく，ターゲットプロセスに存在するデータと，ターゲットプロセスに転送するデータを計算して1つにまとめた方が便利な場合がしばしばある．例えば，あるプロセスのメモリ内のsum変数に，各プロセスの分担分を加算することにより，関与する全てのプロセスの累算(accumulation of a sum)を可能にする．

MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)

IN	origin_addr	バッファの先頭アドレス（選択型）
IN	origin_count	バッファのエントリの数（非負の整数型）
IN	origin_datatype	バッファの各エントリのデータ型（ハンドル）
IN	target_rank	ターゲットのランク（非負の整数型）
IN	target_disp	ウィンドウの先頭からターゲットバッファの先頭までの変位（非負の整数型）
IN	target_count	ターゲットバッファのエントリの数（非負の整数型）
IN	target_datatype	ターゲットバッファの各エントリのデータ型（ハンドル）
IN	op	リデュース操作（ハンドル）
IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Accumulate(void *origin_addr, int origin_count,
MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR
{void MPI::Win::Accumulate(const void* origin_addr, int origin_count, const
             MPI::Datatype& origin_datatype, int target_rank, MPI::Aint
             target_disp, int target_count, const MPI::Datatype&
             target_datatype, const MPI::Op& op) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_ACCUMULATEは，操作opを使って，オリジンバッファの内容をターゲットバッファにアキュムレートする．ここで，オリジンバッファはorigin_addr，origin_count，およびorigin_datatypeによって定義されるバッファであり，ターゲットバッファは，target_rankおよびwinで指定される対象ウィンドウの中のオフセットtarget_dispの位置にある，引数target_countおよびtarget_datatypeで指定されるバッファである．
MPI_ACCUMULATEは MPI_PUTに似ているが，転送したデータでターゲット領域を上書きする代わりに，ターゲット領域と転送データとを計算してターゲット領域に1つにまとめる点が異なる．

この関数は，MPI_REDUCEで使えるどの定義済みの操作も使うことができる． ユーザ定義の関数は使用できない．例えば，opがMPI_SUMの場合，オリジンバッファの各要素がターゲットバッファの対応する要素に加算され，ターゲットの以前の値は置換される．

origin_datatypeおよびtarget_datatype引数は定義済みのデータ型または派生データ型でなければならない．また，派生データ型の全ての基本となるデータ型は同じ定義済みのデータ型でなければならない．両方のdatatype引数は，同じ定義済みのデータ型から構成する必要がある．操作opはその定義済みの型の要素に適用される．target_datatypeには

重複するエントリを指定してはならず，ターゲットバッファは対象ウィンドウに収まらなければならない．

新しい定義済みの操作MPI_REPLACEが定義される．これは結合的な関数 $f(a,b) = b$ に対応する．つまり，ターゲットメモリ内の現在の値はオリジンによって指定された値に置換される．

MPI_REPLACEはMPI_ACCUMULATEでのみ使用でき，MPI_REDUCEなどの集団リダクション操作では使用できない．

> **ユーザへのアドバイス** MPI_PUTは操作MPI_REPLACEを指定したMPI_ACCUMULATEの特別なケースと考えることができる．しかし，MPI_PUTとMPI_ACCUMULATEとでは同時並行的な更新に関して異なる制約があることに注意すること． （**ユーザへのアドバイス終わり**）

例 11.3 $\mathtt{B(j)} = \sum_{\mathtt{map(i)=j}} \mathtt{A(i)}$を計算する．配列A，B，およびmapは同一方法で分散されているとする．ここではシンプルなコードを記述する．

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, win, ierr
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  k = MOD(map(i),m)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL,   &
                      MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

このコードは357ページの例11.2のコードと基本的に同じであるが，ここではゲットの呼び出しがアキュムレートの呼び出しに置換されている．（map が1対1の場合，コードは $\mathtt{B} = \mathtt{A}(\mathtt{map}^{-1})$ を計算する．これは前の例での計算とは逆の代入である．）同様に，355ページの例11.1のゲットの呼び出しをアキュムレートの呼び出しに置換することにより，2つのプロセス間で通信を1回行うだけで計算を行うことができるようになる．

11.4 同期呼び出し

RMA通信は以下の2つのカテゴリに分類される．

- **アクティブターゲット**通信：データがあるプロセスのメモリから別のプロセスのメモリに移動され，両方が明示的に通信に関与する．この通信のパターンはメッセージ通信と同様であるが，ここではデータ転送引数は全て一方のプロセスによって渡され，他方のプロセスは同期に参加するのみである．

- **パッシブターゲット**通信：データがあるプロセスのメモリから別のプロセスのメモリに移動され，オリジンプロセスのみが明確に転送に関与する．そのため，2つのオリジンプロセスが対象ウィンドウ内の同じ領域にアクセスすることにより，通信を行うことができる．対象ウィンドウを所有するプロセスは2つの通信プロセスとは別であってもよく，この場合は明確に通信に参加しない．この通信パラダイムは共有メモリモデルに最も近く，共有データは領域に関係なく全てのプロセスからアクセスすることができる．

引数winを持つRMA通信呼び出しは，winが**アクセスエポック**区間にあるプロセスでのみ行われなければならない．このようなアクセスエポックはwinでのRMA同期呼び出しで開始され，winでの0個以上のRMA通信呼び出し（MPI_PUT，MPI_GET，またはMPI_ACCUMULATE）が行われ，winでの別の同期呼び出しで完了する．これにより，複数のデータ転送によって同期のオーバヘッドを1回に抑えることができる．実装者はより高い自由度でRMA操作の実装を行うことができる．

同じプロセスのwinに対する個々のアクセスエポックは互いに重なり合ってはならない．それに対して，異なるwin引数に関するアクセスエポックは重なり合ってもよい．アクセスエポックの間，ローカル操作や他のMPI呼び出しも行ってよい．

アクティブターゲット通信では，RMA操作による対象ウィンドウへのアクセスは，**エクスポージャーエポック**の間のみ行える．このようなエクスポージャーエポックは，ターゲットプロセスによって実行されるRMA同期呼び出しによって開始および完了する．同じウィンドウのプロセスでの個々のエクスポージャーエポックは分割する必要があるが，このようなエクスポージャーエポックは別のウィンドウのエクスポージャーエポック，あるいは同じまたは別のwin引数用のアクセスエポックと重複していてもよい．オリジンプロセスのアクセスエポックとターゲットプロセスのエクスポージャーエポックの間には1対1の対応があり，対象ウィンドウに対してオリジンプロセスによって発行されたRMA操作は，同じアクセスエポックに発行された場合は（その場合に限り），同じエクスポージャーエポックにその対象ウィンドウにアクセスする．

パッシブターゲット通信では，ターゲットプロセスはRMA同期呼び出しを実行せず，エクスポージャーエポックという概念はない．

MPIでは以下の3つの同期メカニズムが用意されている．

1. MPI_WIN_FENCE集団的同期呼び出しは並列計算でよく使用されるシンプルな同期パターンをサポートする．つまり緩い同期モデルをサポートしており，ここではグローバルな計算フェーズがグローバルな通信フェーズと交互に発生する．このメカニズムは，通信プロセスのグラフが非常に頻繁に変化したり，各プロセスが他の多くのプロセスと通信したりする，緩い同期アルゴリズムにおいて最も便利である．

この呼び出しはアクティブターゲット通信に使用される．オリジンプロセスのアクセスエポックとターゲットプロセスのエクスポージャーエポックは，MPI_WIN_FENCEの呼び出しにより開始および終了する．プロセスはアクセスエポックにwinのグループの全てのプロセスのウィンドウにアクセスすることができ，ローカルウィンドウには，エクスポージャーエポックにwinのグループの全てのプロセスがアクセスできる．

2. 4つの関数MPI_WIN_START，MPI_WIN_COMPLETE，MPI_WIN_POST，およびMPI_WIN_WAITは，次のような場合，同期を最小限に抑えるのに使用できる．つまり，通信するプロセスのペアしか同期しない場合で，かつこれらのペアが同期するのが，そのウィンドウに対するローカルアクセスと，同じウィンドウへのRMAアクセスとを正確に順序づけする目的に限られる場合である．このメカニズムは，各プロセスがごく限られた（論理的）近傍プロセスとしか通信せず，通信グラフが固定かあまり変化しない場合に効率が向上する可能性がある．

   これらの呼び出しはアクティブターゲット通信に使用される．アクセスエポックはオリジンプロセスでMPI_WIN_STARTの呼び出しにより開始され，MPI_WIN_COMPLETEの呼び出しにより終了する．スタート呼び出しは，そのアクセスエポックのターゲットプロセスのグループを指定するgroup引数を備えている．エクスポージャーエポックはターゲットプロセスでMPI_WIN_POSTの呼び出しにより開始され，MPI_WIN_WAITの呼び出しにより終了する．ポスト呼び出しは，そのエクスポージャーエポックのオリジンプロセスの集合を指定するgroup引数を備えている．

3. 最後に，共有ロックおよび排他ロックが，2つの関数MPI_WIN_LOCKおよびMPI_WIN_UNLOCKで用意されている．ロック同期はMPI呼び出しを介し共有メモリモデルを模倣するMPIアプリケーションで便利である．例えば，“billboard”モデルの場合である．その場合，プロセスは不定回，billboardの異なる部分にアクセスしたり更新したりすることができる．

   これらの2つの呼び出しはパッシブターゲット通信を備えている．アクセスエポックはMPI_WIN_LOCKの呼び出しにより開始され，MPI_WIN_UNLOCKの呼び出しにより終了する．winのそのアクセスエポックの間，1つの対象ウィンドウしかアクセスできない．

図11.1に，アクティブターゲット通信の一般的な同期パターンを示す．

postとstartとの同期により，ターゲットプロセスが（post呼び出しにより）ウィンドウをエクスポーズするまで，オリジンプロセスのプット呼び出しが開始しないことが保証される．また， ターゲットプロセスでのウィンドウのエクスポーズがウィンドウへのその前のローカルアクセスの完了後にのみ行われるよう保証される．completeとwaitとの同期により，ウィンドウが（wait呼び出しにより）アンエクスポーズされる前にオリジンプロセスのプット呼び出しが完了するよう保証される．ターゲッ

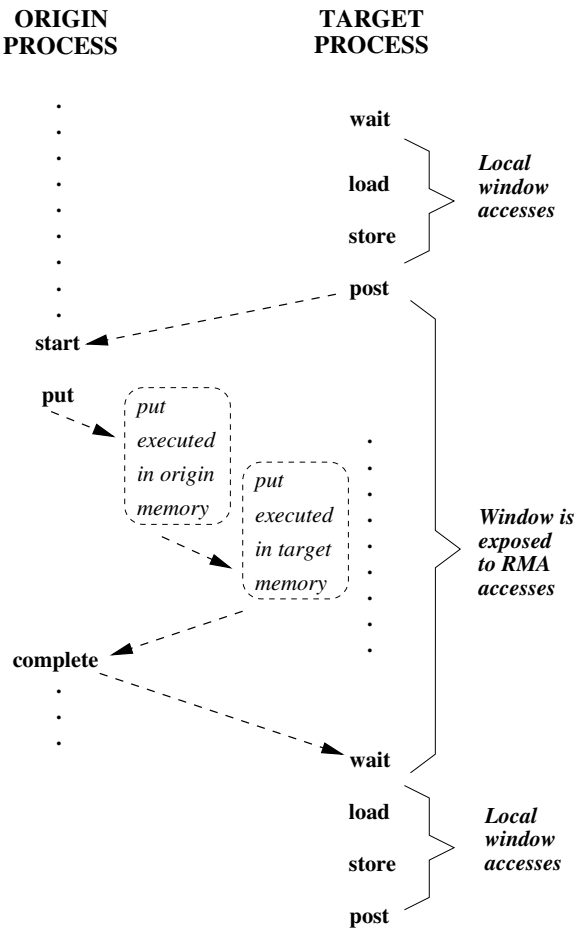

図 11.1: アクティブターゲット通信：点線の矢印は同期を示す（イベントの順序）

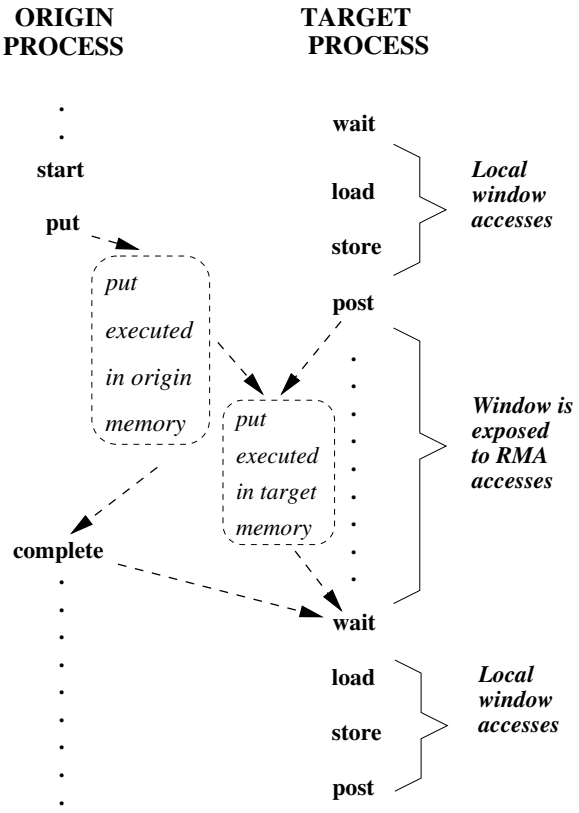

図 11.2: 弱い同期のアクティブターゲット通信：点線の矢印は同期を示す（イベントの順序）

トプロセスによる対象ウィンドウへの次のローカルアクセスは，waitが戻った後にのみ行われる．

図11.1は，同期に基づく自然な時間的順序で発生する操作を示している．postはマッチするstartの前に実行され，completeはマッチするwaitの前に実行される．しかし，このような**強い同期**はウィンドウの正しいアクセス順序のための要件より必要以上に厳しいので，MPI呼び出しの意味論では図11.2に示すような**弱い同期**も許している．

対象ウィンドウへのアクセスはpostの後，ウィンドウがエクスポーズされるまで遅延される．ただし，プットされたデータがバッファリングされるよう実装されている場合，startが早く完了し，putやcompleteも早く終了することがある．同期呼び出しはウィンドウのアクセス順序を正しくするが，必ずしも他の操作を同期はしない．この弱い同期により，より効率的な実装が可能になる．

図11.3にパッシブターゲット通信の一般的な同期パターンを示す．最初のオリジンプロセスがターゲットプロセスのメモリを通して2番目のオリジンプロセスにデータを通信する．しかし，ターゲットプロセスは明示的にこの通信には関与しない．

lockおよびunlock呼び出しにより，2つのRMAアクセスが同時並行的に発生しないことが保証される．しかし，オリジン1によるputがオリジン2によるgetより先に行われるとは限らない．

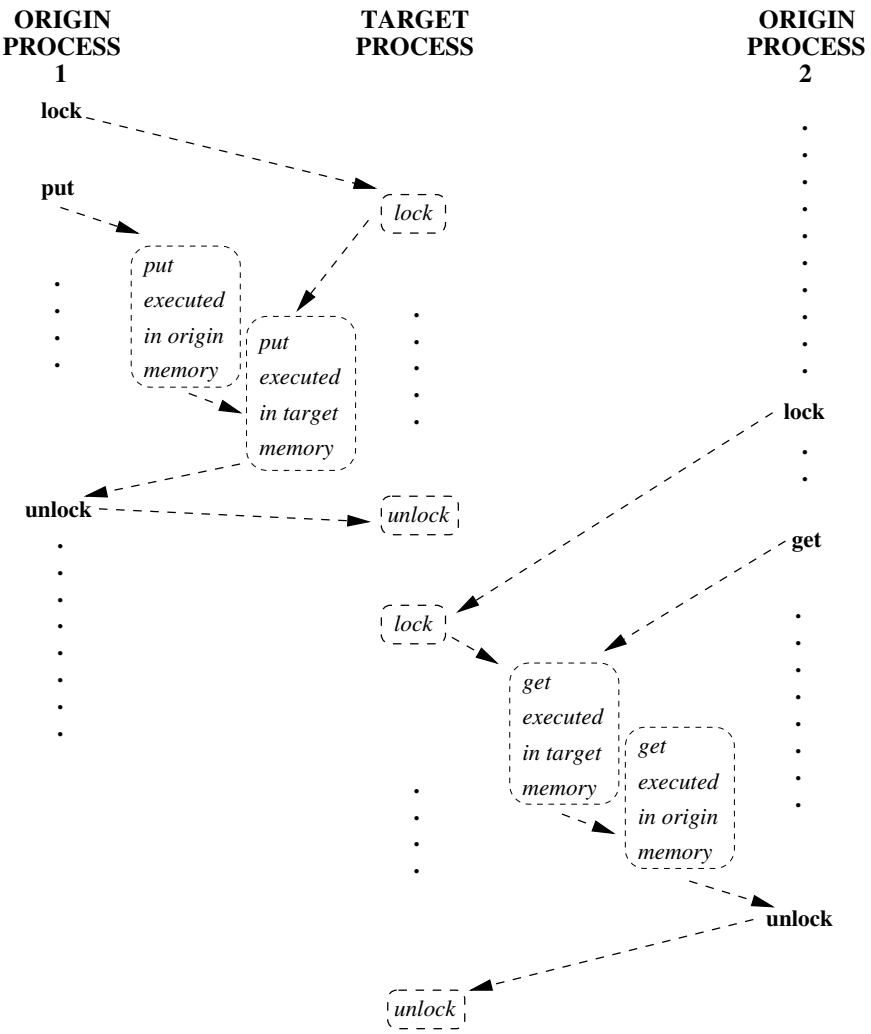

図 11.3: パッシブターゲット通信：点線の矢印は同期を示す（イベントの順序）

11.4.1 フェンス

MPI_WIN_FENCE(assert, win)

IN	assert	プログラムアサーション（整数型）
IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_fence(int assert, MPI_Win win)
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
{void MPI::Win::Fence(int assert) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI呼び出しMPI_WIN_FENCE(assert, win)はwinに基づいてRMA呼び出しと同期する．この呼び出しはwinのグループに対して集団的である．あるプロセスでフェンス呼び出しの前に呼び出したwinに対する全てのRMA操作は，そのフェンス呼び出しが復帰するまでに，そのプロセスで必ず完了する．これらの操作は，ターゲットでフェンス呼び出しが戻る前にターゲットで完了する．フェンス呼び出しが戻った後でプロセスによって開始されたwin上のRMA操作は，MPI_WIN_FENCEがターゲットプロセスによって呼び出された後でないと対象ウィンドウにアクセスできない．

この呼び出しによりRMAのアクセスエポックが完了するのは，別のフェンス呼び出しが先に行われていて，これらの2つの呼び出しの間にローカルプロセスがwin上でRMA通信呼び出しを発行した場合である．この呼び出しによりRMAのエクスポージャーエポックが完了するのは，別のフェンス呼び出しが先に行われていて，ローカルウィンドウがこれらの2つの呼び出しの間のRMAアクセスのターゲットであった場合である．この呼び出しによりRMAのアクセスエポックが開始するのは，別のフェンス呼び出しと，これらの2回のフェンス呼び出しの間に発行されたRMA通信呼び出しがこの後に行われた場合である．この呼び出しによりエクスポージャーエポックが開始するのは，別のフェンス呼び出しがこの後に行われ，ローカルウィンドウがこれらの2回のフェンス呼び出しの間のRMAアクセスのターゲットである場合である．そのため，このフェンス呼び出しはポスト，スタート，コンプリート，およびウェイトのサブセットの呼び出しと等価である．

通常，フェンス呼び出しではバリア同期が必要となり，プロセスでMPI_WIN_FENCEの呼び出しが完了するのは，グループ内の他の全てのプロセスがマッチする呼び出しを開始した後である．しかし，終わらせるべきいかなるエポックもないことがわかっているMPI_WIN_FENCEの呼び出し（特に，assert = MPI_MODE_NOPRECEDEの場合の呼び出し）は，必ずしもバリア同期の機能を果たさない．

assert引数は呼び出しのコンテクスト上で，さまざまな最適化に使用してよいアサーションを渡すのに供される．これについては，第11.4.4節で説明する．assert = 0という値は常に有効である．

ユーザへのアドバイス MPI_WIN_FENCEの呼び出しは，フェンス呼び出しと同期されたプット，ゲット，またはアキュムレートの呼び出しの前後の両方に実行する

必要がある．（ユーザへのアドバイス終わり）

11.4.2 一般的なアクティブターゲット同期

MPI_WIN_START(group, assert, win)

IN	group	ターゲットプロセスのグループ（ハンドル）
IN	assert	プログラムのアサーション（整数型）
IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
{void MPI::Win::Start(const MPI::Group& group, int assert) const （廃止された
            呼び出し形式, 第15.2節を参照）}
```

winに対しRMAアクセスエポックを開始する．このアクセスエポック中にwin上で発行されたRMA呼び出しはgroup内のプロセスのウィンドウにしかアクセスしてはならない．group内の各プロセスはマッチするMPI_WIN_POSTの呼び出しを発行する必要がある．マッチするMPI_WIN_POSTの呼び出しがターゲットプロセスによって実行されるまで，必要に応じて各対象ウィンドウへのRMAのアクセスは遅延される．対応するMPI_WIN_POSTの呼び出しが実行されるまでMPI_WIN_STARTをブロッキングすることができるが，これは必須ではない．

assert引数は呼び出しのコンテクスト上で，さまざまな最適化に使用してよいアサーションを渡すのに供される．これについては，第11.4.4節で説明する．assert = 0という値は常に有効である．

MPI_WIN_COMPLETE(win)

IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_complete(MPI_Win win)
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
{void MPI::Win::Complete() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_WIN_STARTの呼び出しによって開始されたwin上のRMAのアクセスエポックを完了する．このアクセスエポック中にwin上で発行された全てのRMA通信呼び出しは，呼び出しが戻った時点でオリジンで完了している．

MPI_WIN_COMPLETEはオリジンでは，その前のRMA呼び出しを完了させるが，ターゲットでは完了を強制しない．オリジンでプットまたはアキュムレート呼び出しが完了していても，ターゲットでは完了していない場合がある．

以下の例で呼び出しのシーケンスを考えてみる．

例 11.4

```
MPI_Win_start(group, flag, win);
MPI_Put(...,win);
MPI_Win_complete(win);
```

MPI_WIN_COMPLETEの呼び出しは，プット呼び出しがオリジンで完了するまで戻らず，対象ウィンドウはMPI_WIN_STARTの呼び出しとターゲットプロセスによるMPI_WIN_POSTの呼び出しのマッチが完了するまでは，プット操作によりアクセスされない．そのため，実装者は多くの選択が可能となる．全てのターゲットプロセスでマッチするMPI_WIN_POSTの呼び出しが行われるまで，MPI_WIN_STARTの呼び出しをブロッキングすることができる．他の実装方法として，MPI_WIN_STARTの呼び出しはノンブロッキングだが，MPI_PUTの呼び出しはマッチするMPI_WIN_POSTの呼び出しが行われるまでブロッキングされるようにすることもできる．別の実装として，最初の2つの呼び出しはノンブロッキングだが，MPI_WIN_COMPLETE の呼び出しはMPI_WIN_POSTの呼び出しが行われるまでブロッキングされるようにすることもできる．さらに別の実装として，いずれかのターゲットプロセスでMPI_WIN_POSTが呼び出される前に3つの全ての呼び出しが完了するようにすることもできる． この最後のケースでは，ターゲットでの完了より早くオリジンでプットが完了することが可能なように，プットされたデータがバッファされなければならない．しかし，MPI_WIN_POSTの発行後は，それ以上の依存性なしで上記のシーケンスは必ず完了する．

MPI_WIN_POST(group, assert, win)

IN	group	オリジンプロセスのグループ（ハンドル）
IN	assert	プログラムのアサーション（整数型）
IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
{void MPI::Win::Post(const MPI::Group& group, int assert) const （廃止された呼び出し形式, 第15.2節を参照）}
```

winに関連付けられたローカルウィンドウのRMAのエクスポージャーエポックを開始する．エクスポージャーエポック中のウィンドウwinに対し，group内のプロセスしか，RMA呼び出しでアクセスすべきでない．group内の各プロセスはマッチするMPI_WIN_STARTの呼び出しを発行しなければならない．MPI_WIN_POSTはブロッキングを行わない．

MPI_WIN_WAIT(win)

IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_wait(MPI_Win win)
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
{void MPI::Win::Wait() const （廃止された呼び出し形式, 第15.2節を参照）}
```

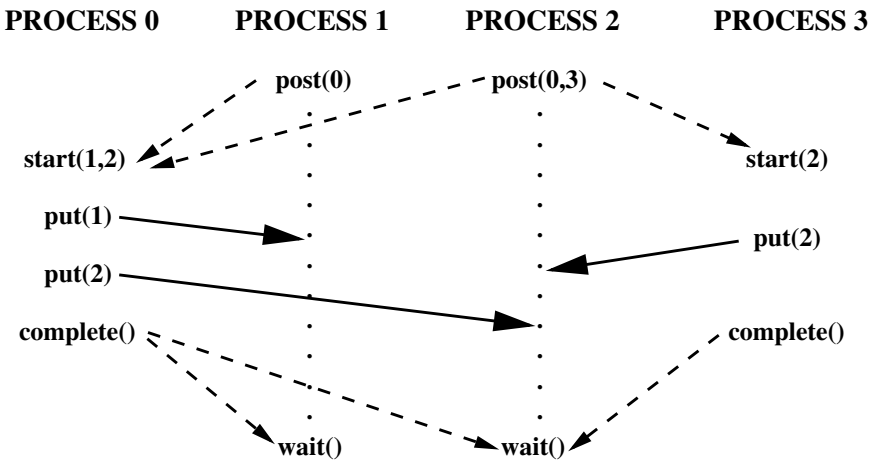

図 11.4: アクティブターゲット通信：点線の矢印は同期を示し，実線の矢印はデータ転送を示す

win上でMPI_WIN_POSTの呼び出しにより開始されたRMAのエクスポージャーエポックを完了する．この呼び出しは，このエクスポージャーエポック中にウィンドウへのアクセスが許可されていた各オリジンプロセスにより発行されたMPI_WIN_COMPLETE(win)の呼び出しにマッチする．MPI_WIN_WAITの呼び出しは，マッチする全てのMPI_WIN_COMPLETEの呼び出しが行われるまでブロッキングされる．これにより，これらの全てのオリジンプロセスがローカルウィンドウへのRMAアクセスを完了していることが保証される．呼び出しが戻った時点で，対象ウィンドウでのこれらの全てのRMAアクセスは完了している．

図11.4 はこれらの4つの関数の使用法を示している．

プロセス0はデータをプロセス1および2のウィンドウにプットし，プロセス3はデータをプロセス2のウィンドウにプットする．各スタート呼び出しは，アクセスされるウィンドウのプロセスのランクの一覧を指定し，各ポスト呼び出しは，ローカルウィンドウにアクセスするプロセスのランクの一覧を指定する．この図は強い同期を前提とした，各種イベントに対するありうるタイミングを説明している． 弱い同期では，スタート，プット，コンプリートの呼び出しが マッチするポスト呼び出しより前に発生することがある．

MPI_WIN_TEST(win, flag)

IN	win	ウィンドウオブジェクト（ハンドル）
OUT	flag	成功フラグ（論理型）

```
int MPI_Win_test(MPI_Win win, int *flag)
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
{bool MPI::Win::Test() const （廃止された呼び出し形式, 第15.2節を参照）}
```

これはノンブロッキングバージョンのMPI_WIN_WAITである．マッチする

MPI_WIN_COMPLETE呼び出しによって完了が通知され，対応するMPI_WIN_POST呼び出しでエクスポーズする対象とされたグループによるローカルウィンドウへの全てのアクセスが完了した場合，この関数はflag = trueを返す．それ以外の場合はflag = falseを返す．前者の場合，MPI_WIN_WAITの呼び出しは，ただちに復帰するはずである．MPI_WIN_TESTがflag = trueを返した場合はMPI_WIN_WAITが戻ったのと同じ作用である．flag = falseが返された場合，呼び出しによる表立った作用はない．

MPI_WIN_TESTの呼び出しは，MPI_WIN_WAITの呼び出しが可能な場合のみ行う必要がある．MPI_WIN_TEST呼び出しでflag = trueが一度返されたら，ウィンドウに対し再度ポストが呼ばれるまでMPI_WIN_TESTを呼び出してはならない．

ウィンドウwinに，winのプロセスによって通信に使用される「隠れた」コミュニケータwincommが関連付けられているとする．ポスト呼び出しとスタート呼び出しとの間のマッチング規則，およびコンプリート呼び出しとウェイト呼び出しとの間のマッチング規則は，以下の（部分）モデル実装を検討してみることで，送信と受信のマッチング規則から導き出すことができる．

MPI_WIN_POST(group,0,win) wincommを使用し，group内の各プロセスに対するタグtag0付きのノンブロッキングの送信を開始する．これらの送信の完了を待つ必要はない．

MPI_WIN_START(group,0,win) wincommを使用し，group内の各プロセスからのタグtag0付きのノンブロッキングの受信を開始する．ターゲットプロセス i 内のウィンドウへのRMAアクセスは，プロセス i からの受信が完了するまで遅延する．

MPI_WIN_COMPLETE(win) 前のスタート呼び出しのグループ内の各プロセスに対する，タグtag1付きのノンブロッキングの送信を開始する．これらの送信の完了を待つ必要はない．

MPI_WIN_WAIT(win) 前のポスト呼び出しのグループ内の各プロセスに対する，タグtag1付きのノンブロッキングの受信を開始する．全ての受信の完了を待つ．

正しいプログラムでは競合は発生しない．なぜなら，各送信は一意の受信とマッチし，逆も同様だからである．

> **根拠** 一般的なアクティブターゲット同期の設計では，ユーザが通信リンクの両端で通信パターンの詳細情報を提供する必要があるため，各オリジンでターゲットの一覧を指定し，各ターゲットでオリジンの一覧を指定する．これにより実装者の自由度が最大限に向上する（そのため，効率も向上する）．一方が他方の識別子を「知っている」ため，どちらからでも同期を開始することができる．これにより，発生しうる競合に対しても最大限の保護を行うことができる．しかし，一般的に，RMAで求められるよりも多くの情報が必要となる．一般的にオリジンではターゲットのランクを知るだけで十分であるが，ターゲットではオリジンのランクを知るだけでは十分でない．「匿名」通信を利用するには，フェンスまたはロックのメカニズムが必要となる． （**根拠の終わり**）

ユーザへのアドバイス 有向グラフ $G = < V, E >$ で表現される通信パターンがあるとする．ここで，$V = \{0, \ldots, n-1\}$ および $ij \in E$ とし，オリジンプロセス i がターゲットプロセス j のウィンドウにアクセスするとする．そして，それぞれのプロセス i がMPI_WIN_POST($ingroup_i$, ...)の呼び出しを発行し，続いてMPI_WIN_START($outgroup_i$,...)の呼び出しを発行する．ここで，$outgroup_i = \{j : ij \in E\}$ および $ingroup_i = \{j : ji \in E\}$ である．group引数が空の場合，呼び出しは無操作であり，省略することができる．通信呼び出しの後，スタートを発行した各プロセスはコンプリートを発行する．最後に，ポストを発行した各プロセスはウェイトを発行する．

各プロセスは，別のメンバを持つgroup引数を使用して呼び出すことができることに注意すること．（**ユーザへのアドバイス終わり**）

11.4.3 ロック

MPI_WIN_LOCK(lock_type, rank, assert, win)

IN	lock_type	MPI_LOCK_EXCLUSIVE または MPI_LOCK_SHARED（ステート型）
IN	rank	ロックされたウィンドウのランク（非負の整数型）
IN	assert	プログラムのアサーション（整数型）
IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
{void MPI::Win::Lock(int lock_type, int rank, int assert) const （廃止された呼び出し形式, 第15.2節を参照）}
```

RMAのアクセスエポックを開始する．ランクがrankのプロセスのウィンドウのみが，このアクセスエポック中にwin上のRMA操作によりアクセスできる．

MPI_WIN_UNLOCK(rank, win)

IN	rank	ウィンドウのランク（非負の整数型）
IN	win	ウィンドウオブジェクト（ハンドル）

```
int MPI_Win_unlock(int rank, MPI_Win win)
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
{void MPI::Win::Unlock(int rank) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_WIN_LOCK(...,win)の呼び出しにより開始されたRMAのアクセスエポックを完了する．このアクセスエポック中に発行されたRMA操作は，この呼び出しが戻った時点でオリジンとターゲットの両方で完了している．

ロックは，lock呼び出しとunlock呼び出しの間に発行されるRMA呼び出しの影響を受けるロック済みの対象ウィンドウへのアクセスを保護するため，かつlock呼び出し

とunlock呼び出しの間に実行されるロック済みのローカルウィンドウへのローカルなロード／ストアアクセスを保護するために使用される．排他ロックによって保護されるアクセスはこのウィンドウのサイトにおいて，ロック保護された同じウィンドウへの他のアクセスと同時並行的にはならない．また，共有ロックによって保護されるアクセスはこのウィンドウのサイトにおいて，同じウィンドウへの排他ロックによって保護されるアクセスと，同時並行的にはならない．

ウィンドウを同時並行的にロック済かつエクスポーズ済（エクスポージャーエポック状態）にすることは誤りである．つまり，ターゲットプロセスがMPI_WIN_POSTを呼び出し済であり，かつMPI_WIN_WAITをまだ呼び出していない場合，プロセスはその対象ウィンドウをロックするためにMPI_WIN_LOCKを呼び出してはならない．また，ローカルウィンドウがロックされた状態で，MPI_WIN_POSTを呼び出すことは誤りである．

根拠 代替案として，MPIがエクスポージャーエポックとロック期間との間で相互排他を強制する方法もある．しかし，この代替案は，ロック同期およびアクティブターゲット同期の2つの同期メカニズムの間で干渉しないとき，このような滅多におこらない相互干渉サポートのための，余分なオーバーヘッドを生じてしまう．ここで推奨するプログラミングスタイルは，1組のウィンドウを使う場合，ある時点では1つの同期メカニズムしか使わず，ある同期メカニズムから別のメカニズムへのシフトでは，それ自体滅多に起こらないようにしつつ，グローバルな同期を伴うようにしながらシフトする，というものである． （**根拠の終わり**）

ユーザへのアドバイス ウィンドウでロック期間とエクスポージャーエポックとの間で相互排他を実行するには，明示的な同期コードを使用する必要がある． （**ユーザへのアドバイス終わり**）

実装者は，lock呼び出しによって同期されるRMA通信の使用を，MPI_ALLOC_MEMで割り当てたメモリ内のウィンドウに制限することができる（284ページの第8.2節）．ロックは，このようなメモリでしか可搬に使えない．

根拠 メモリが共有されていない状態でパッシブターゲット通信の実装を行うには，非同期エージェントが必要となる．特別に割り当てられたメモリだけに制限すれば，このようなエージェントをより容易に実装し，より高い性能を得ることができる．さらに共有メモリを使用する場合，このようなエージェントを完全に避けることが可能である．そのため，共有メモリマシン間の第3者通信[3]に対し共有メモリを使わせるという制約を課すことは自然であると考えられる．

その反面，パッシブターゲット通信は，非標準のFortran言語の機能，つまりC言語のポインタのような機能を利用しないと利用できないことになるが，これらは一部のFortran言語のコンパイラ（本書の執筆時点ではg77およびWindows/NTコンパイラ）ではサポートされていない．また，COMMONブロックや，他の静的に宣言されたFortran言語の配列に対し，パッシブターゲット通信は可搬な対象とはなりえない． （**根拠の終わり**）

[3]訳者註：MPIプロセス以外の通信

以下の例で呼び出しのシーケンスを考えてみる.

例 11.5

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win)
MPI_Put(..., rank, ..., win)
MPI_Win_unlock(rank, win)
```

MPI_WIN_UNLOCKの呼び出しは，オリジンとターゲットでプット転送が完了するまで戻らない．これにより実装者の自由度が大幅に向上する．MPI_WIN_LOCKの呼び出しはウィンドウの排他ロックが得られるまでブロッキングされることがある．または，MPI_PUTの呼び出しはロックが得られるまでブロッキングされるのに対してMPI_WIN_LOCKの呼び出しはブロッキングされないことがある．あるいは，MPI_WIN_UNLOCKはロックが得られるまでブロッキングされるのに対して，最初の2つの呼び出しはブロッキングされないことがある．つまり対象ウィンドウの更新はMPI_WIN_UNLOCKの呼び出しが行われるまで延期される．しかし，MPI_WIN_LOCKの呼び出しを使用してローカルウィンドウのロックを行う場合，lock呼び出しが戻った後に発行されるウィンドウへのローカルなロード／ストアアクセスがロックによって保護されることがあるため，ロックが得られるまで呼び出しをブロッキングする必要がある.

11.4.4 アサーション

MPI_WIN_POST，MPI_WIN_START，MPI_WIN_FENCE，およびMPI_WIN_LOCK呼び出しにおいて，assert引数は，呼び出しのコンテクスト上でアサーションを渡すのに使用される．このアサーションは性能を最適化するために使うことができる．assert引数は，プログラムの正しい情報を渡す場合，プログラムの意味論を変更しない．不正な情報を渡すことは誤りである．プログラムが特に何も約束しないような，普通のケースでは常にassert = 0を渡してかまわない.

> **ユーザへのアドバイス** assertの情報を利用していない実装も多く，一部の情報は一貫性のない共有メモリマシンにのみ関係している．各システムで有益な情報については，実装マニュアルを参照する必要がある．逆に，適用可能なときにつねに，正しいアサーションを渡すアプリケーションは可搬であるし，利用可能なときにはアサーション固有の最適化を生かすことができる． （**ユーザへのアドバイス終わり**）
>
> **実装者へのアドバイス** 実装では常にassert引数を無視することができる．実装者は，実装で重要なassertの値を明文化する必要がある． （**実装者へのアドバイス終わり**）

assertは整数型定数MPI_MODE_NOCHECK，MPI_MODE_NOSTORE，MPI_MODE_NOPUT，MPI_MODE_NOPRECEDE，およびMPI_MODE_NOSUCCEEDの0個以上のビットベクトル ORである．各呼び出しの重要なオプションを以下に示す.

ユーザへのアドバイス C言語／C++言語ユーザは，次の定数を組み合わせるのに，ビットベクトル OR (|) を使うことができる．Fortran 90言語のユーザは固有のビットベクトルIORを使用することができる．Fortran 77言語のユーザは，サポートするシステムでビットベクトルIORを（非可搬に）使用することができる．または，Fortran言語のユーザは定数のOR演算のために整数加算を可搬に使用できる（ただし，同一の定数を複数回加算することはできない）．（**ユーザへのアドバイス終わり**）

MPI_WIN_START:

MPI_MODE_NOCHECK — MPI_WIN_STARTの呼び出しの実行時に，マッチするMPI_WIN_POSTの呼び出しは，全てのターゲットプロセスですでに完了している．スタート呼び出しでnocheckオプションを指定できるのは，マッチする各ポスト呼び出しで指定されている場合のみである．これは，ハンドシェイクがコードで暗に示されている場合にハンドシェイクを省くことができる「準備完了送信(ready-send)」の最適化と同様である（しかし，スタートもポストも両方ともnocheckオプションを指定しなければならないのに対して，準備完了送信は正規の受信とマッチする）．

MPI_WIN_POST:

MPI_MODE_NOCHECK — MPI_WIN_POSTの呼び出しの実行時に，対応するMPI_WIN_STARTの呼び出しがどのオリジンプロセスでも行われていない．ポスト呼び出しでnocheckオプションを指定できるのは，マッチするスタート呼び出しで指定されている場合のみである．

MPI_MODE_NOSTORE — ローカルウィンドウが前回の同期以来，ローカルなストア（またはローカルなゲットまたは受信呼び出し）によって更新されていなかった．これにより，ポスト呼び出しでのキャッシュ同期が不要になる場合がある．

MPI_MODE_NOPUT — ローカルウィンドウは，後続の（ウェイト）同期まで，ポスト呼び出しの後のプットまたはアキュムレート呼び出しによって更新されない．これにより，ウェイト呼び出しでのキャッシュ同期が不要になる場合がある．

MPI_WIN_FENCE:

MPI_MODE_NOSTORE — ローカルウィンドウが前回の同期以来，ローカルなストア（またはローカルなゲットまたは受信呼び出し）によって更新されていなかった．

MPI_MODE_NOPUT — ローカルウィンドウは，次の（フェンス）同期まで，フェンス呼び出しの後のプットまたはアキュムレート呼び出しによって更新されない．

MPI_MODE_NOPRECEDE — フェンスはローカルに発行されたRMA呼び出しのシーケンスを完了しない．このアサーションがウィンドウグループ内のいずれかのプロセスによって与えられる場合，グループ内の全てのプロセスによって与えられなければならない．

MPI_MODE_NOSUCCEED — フェンスはローカルに発行されたRMA呼び出しのシーケンスを開始しない．このアサーションがウィンドウグループ内のいずれかのプロセスによって与えられる場合，グループ内の全てのプロセスによって与えられなければならない．

MPI_WIN_LOCK:

MPI_MODE_NOCHECK — 呼び出し元がウィンドウロックを保持している間，他のプロセスは衝突するロックを保持しない，または取得しようとしない．これは，他の手段によって相互排他が達成されているが，lockおよびunlock呼び出しに付随する一貫性のある操作がなお必要な場合に有用である．

ユーザへのアドバイス nostoreおよびnoprecedeフラグは呼び出しの前に行われていた処理に関する情報を提供し，noputおよびnosucceedフラグは呼び出しの後に行われる処理に関する情報を提供する．（**ユーザへのアドバイス終わり**）

11.4.5 その他の説明

RMAルーチンが完了すれば，そのルーチンに引数として渡された不可視オブジェクトを解放しても安全である．例えば，MPI_PUT呼び出しのdatatype引数は，通信が完了していない可能性があっても，呼び出しが戻ればすぐに解放することができる．

メッセージ通信の場合と同様に，データ型をRMA通信で使う前にコミットする必要がある．

11.5 例

例 11.6 下の例で，フェンス同期を使用した一般的な緩い同期の反復コードを示す．各プロセスのウィンドウは配列Aで構成され，この配列にはプット呼び出しのオリジンバッファとターゲットバッファが含まれる．

```
...
while(!converged(A)){
  update(A);
  MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                         todisp[i], 1, totype[i], win);
  MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
  }
```

同じコードは，プットの代わりにゲットを使用して記述することができる．通信フェーズの間，各ウィンドウは同時並行的に読み取り（プットのオリジンバッファとして）

と書き込み（プットのターゲットバッファとして）を行う．プットのターゲットバッファと他の通信バッファの間で重なりがない限り，これは問題ない．

例 11.7 同様に一般的な例だが，計算／通信の重複が増える．ここでは更新フェーズを2つのサブフェーズに分割するとする．1つ目は通信に関係する“boundary”が更新される場合で，2つ目は通信データを使用も提供もしない“core”が更新される場合である．

```
...
while(!converged(A)){
  update_boundary(A);
  MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                    fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
  }
```

ゲット通信とコアの更新は，同じ領域にアクセスしないため，同時並行的に行うことができる．また，ゲット呼び出しによるオリジンバッファのローカル更新はupdate_core呼び出しによるcoreのローカル更新と同時並行的に行うことができる．プット通信と同様の重複を得るには，core用とboundary用に別のウィンドウを使用する必要がある．これが必要となるのは，同じまたは重なるウィンドウでローカルなストアとプットを同時並行的に行うことが許可されていないためである．

例 11.8 例11.6と同じコードだが，ポストースタートーコンプリートーウェイトを使用して書き直している．

```
...
while(!converged(A)){
  update(A);
  MPI_Win_post(fromgroup, 0, win);
  MPI_Win_start(togroup, 0, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                        todisp[i], 1, totype[i], win);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
  }
```

例 11.9 例11.7と同じ例だが，フェーズを分割している．

```
...
while(!converged(A)){
  update_boundary(A);
  MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
  MPI_Win_start(fromgroup, 0, win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                    fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
  }
```

例 11.10 チェッカーボード，またはダブルバッファ通信パターンで，計算／通信の重複の拡大が可能．配列A1の値を使用して配列A0が更新され，同様に逆の更新も行われる．通信が対称的であると考え，プロセスAがプロセスBからデータを取得する場合，プロセスBもプロセスAからデータを取得する．ウィンドウwiniは配列Aiで構成される．

```
...
if (!converged(A0,A1))
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Barrier(comm0);
/* the barrier is needed because the start call inside the
loop uses the nocheck option */
while(!converged(A0, A1)){
  /* communication on A0 and computation on A1 */
  update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
  for(i=0; i < neighbors; i++)
    MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
                fromdisp0[i], 1, fromtype0[i], win0);
  update1(A1); /* local update of A1 that is
                  concurrent with communication that updates A0 */
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
  MPI_Win_complete(win0);
  MPI_Win_wait(win0);

  /* communication on A1 and computation on A0 */
  update2(A0, A1); /* local update of A0 that depends on A1 (and A0)*/
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
  for(i=0; i < neighbors; i++)
    MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                fromdisp1[i], 1, fromtype1[i], win1);
  update1(A0); /* local update of A0 that depends on A0 only,
                  concurrent with communication that updates A1 */
  if (!converged(A0,A1))
    MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
  MPI_Win_complete(win1);
  MPI_Win_wait(win1);
  }
```

プロセスがwin1に関連付けられたリモートウィンドウへのRMAアクセスが完了する前に，win0に関連付けられたローカルウィンドウをポストする．wait(win1)呼び出しが復帰した時点で，呼び出しプロセスの全ての隣接プロセスはwin0に関連付けられたウィンドウをポスト済である．逆に，プロセスがwait(win0)呼び出しが復帰した時点で，呼び出しプロセスの全ての隣接プロセスはwin1に関連付けられた ウィンドウをポスト済である．そのため，MPI_WIN_STARTの呼び出しと一緒にnocheckオプションを使用することができる．

update(A1, A0)（またはupdate(A0, A1)）呼び出しによって使用される配列A0（またはA1）の領域がRMA通信によって変更される領域と重ならない場合，ゲット呼び出しの代わりにプット呼び出しを使用することができる．システムによっては，プット呼び出しは一方向でしか情報交換を必要としないため，ゲット呼び出しよりもプット呼び出しの方が効率的な場合がある．

11.6 エラー処理

11.6.1 エラーハンドラ

MPI_WIN_CREATE(...,comm,...)の呼び出し中にエラーが発生すると，commに現在関連付けられているエラーハンドラが呼び出される．その他の全てのRMA呼び出しは入力win 引数を備えている．このような呼び出しの途中でエラーが発生すると，winに現在関連付けられているエラーハンドラが呼び出される．

winに関連付けられているデフォルトのエラーハンドラはMPI_ERRORS_ARE_FATALである．winに明示的に新しいエラーハンドラを関連付けることにより，このデフォルトを変更することができる（286ページの第8.3節を参照）．

11.6.2 エラークラス

片方向通信用に以下のエラークラスが定義されている．

MPI_ERR_WIN	不正なwin引数
MPI_ERR_BASE	不正なbase引数
MPI_ERR_SIZE	不正なsize引数
MPI_ERR_DISP	不正なdisp引数
MPI_ERR_LOCKTYPE	不正なlocktype引数
MPI_ERR_ASSERT	不正なassert引数
MPI_ERR_RMA_CONFLICT	ウィンドウへの衝突するアクセス
MPI_ERR_RMA_SYNC	RMA呼び出しの誤った同期

表 11.1: 片方向通信ルーチンのエラークラス

11.7 意味論と正しさ

RMA操作の意味論をよく理解するため，次の仮定をおく．まず，システムはウィンドウごとに，別々の**公開**コピーを管理しているとする．さらに，プロセスの元のメモリ領域（**非公開**ウィンドウコピー）を管理しているとする． プロセスのメモリ上には各変数のインスタンスが1つしかないが，そのインスタンスを含むウィンドウごとに，その変数の別な公開コピーがある． ロードはプロセスメモリ内のインスタンスにアクセスする（ここにはMPI sendが含まれる）．ストアはプロセスメモリ内のインスタンスにアクセスし，更新する（ここにはMPI receiveが含まれる）が，更新は同じ領域の別の**公開**コピーに影響することがある．ウィンドウでのゲットはそのウィンドウの公開コピーにアクセスする．ウィンドウでのプットまたはアキュムレートはそのウィンドウの公開コピーにアクセスし，これを更新するが，プロセスメモリ内の同じ領域の非公開コピーと，重複する他のウィンドウの公開コピーに影響することがある．これについて，図11.5に示す．

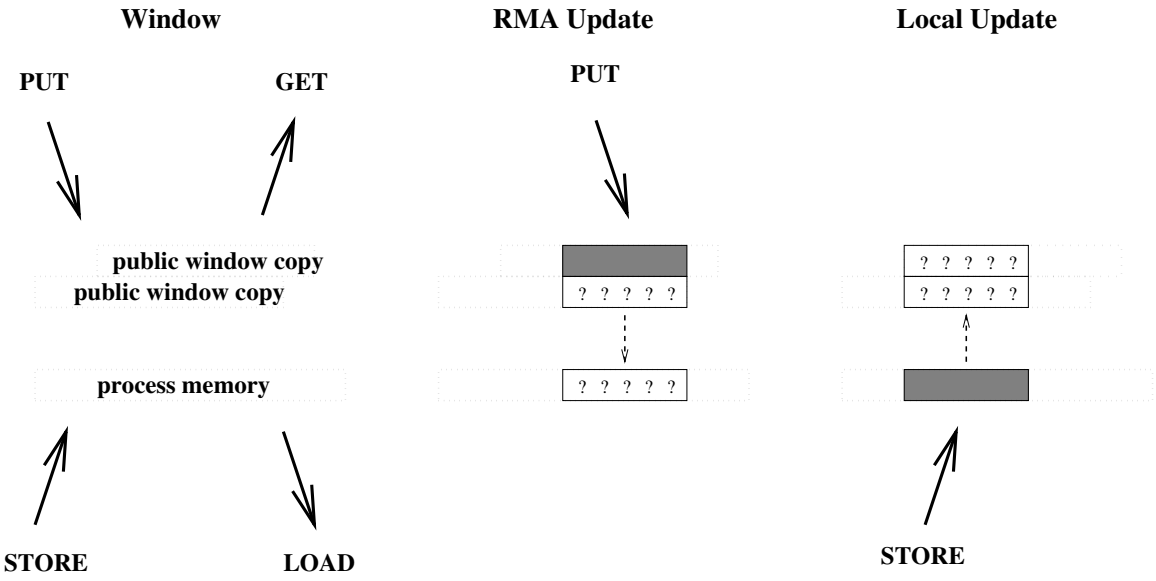

図 11.5: ウィンドウの概要図

以下の規則により，操作がオリジンまたはターゲットで完了していなければならない最終時点を規定する．オリジンのプロセスメモリ内でゲット呼び出しによって実行される更新はオリジンでゲット操作が完了した時点で（あるいはそれより早く）検知可能になり，対象ウィンドウの公開コピー内でプットまたはアキュムレート呼び出しによって実行される更新はターゲットでプットまたはアキュムレートが完了した時点で（あるいはそれより早く）検知可能になる．この規則では，1つのウィンドウコピーの更新が別の重なりのあるコピーで検知可能になる最終時点も規定される．

1. RMA操作は，オリジンでこのアクセスを同期させる後続のMPI_WIN_COMPLETE，MPI_WIN_FENCE，またはMPI_WIN_UNLOCK呼び出しにより，オリジンで完了する．

2. RMA操作がMPI_WIN_FENCEの呼び出しによりオリジンで完了する場合，その操作はターゲットプロセスによるマッチするMPI_WIN_FENCEの呼び出しにより，ターゲットで完了する．

3. RMA操作がMPI_WIN_COMPLETEの呼び出しによりオリジンで完了する場合，その操作はターゲットプロセスによるマッチするMPI_WIN_WAITの呼び出しにより，ターゲットで完了する．

4. RMA操作がMPI_WIN_UNLOCKの呼び出しによりオリジンで完了する場合，その操作は同じMPI_WIN_UNLOCKの呼び出しによりターゲットで完了する．

5. プロセスメモリ内の非公開ウィンドウコピーの領域の更新内容は，少なくともウィンドウの所有者によりそのウィンドウでその後にMPI_WIN_POST，MPI_WIN_FENCE，またはMPI_WIN_UNLOCKが実行されたときに，公開ウィンドウコピーで検知可能になる．

6. プットまたはアキュムレート呼び出しによる公開ウィンドウコピーに対する更新の内容は，少なくともウィンドウの所有者によりそのウィンドウでその後にMPI_WIN_WAIT，MPI_WIN_FENCE，またはMPI_WIN_LOCKが実行されたときに，プロセスメモリ内の非公開コピーで検知可能になる．

公開コピーから非公開コピーへの転送を完了させるMPI_WIN_FENCEまたはMPI_WIN_WAIT呼び出し(6)は，ウィンドウコピー(2，3) でプットまたはアキュムレート操作を完了させるのと同じ呼び出しである．プットまたはアキュムレートアクセスがロックと同期されている場合，公開ウィンドウコピーの更新は更新プロセスがMPI_WIN_UNLOCKを実行するとすぐに完了する．逆に，プロセスメモリ内での非公開コピーの更新は，ターゲットプロセスがそのウィンドウで同期呼び出しを実行するまで遅延されることがある(6)．そのため，プロセスメモリの更新は常に，プロセスが適切な同期呼び出しを実行するまで遅延される可能性がある．また，フェンスまたはポスト－スタート－コンプリート－ウェイト同期が使用されている場合，公開ウィンドウコピーの更新は，ウィンドウの所有者が同期呼び出しを実行するまで遅延される可能性がある．ロック同期を使用する場合のみ，ウィンドウの所有者が関連する同期呼び出しを実行しない場合でも公開ウィンドウコピーの更新が必要となる．

上記の規則により，公開ウィンドウコピーの更新が別の重複する公開ウィンドウコピーで検知可能になるタイミングも暗に定義される．例えば，重複する2つのウィンドウwin1およびwin2を考えてみる．ウィンドウの所有者がMPI_WIN_FENCE(0, win1)を呼び出すと，プロセスメモリ内で前の更新がリモートプロセスによるウィンドウwin1から検知可能になる．その後のMPI_WIN_FENCE(0, win2)の呼び出しでは，これらの更新がwin2の公開コピーで検知可能になる．

正しいプログラムは以下の規則に従う必要がある．

1. ウィンドウ内の領域は，その領域の更新が開始されたら，その更新がプロセスメモリ内の非公開ウィンドウコピーで検知可能になるまで，ローカルにアクセスしてはならない．

2. ウィンドウ内の領域は，その領域の更新が開始されたら，その更新が公開ウィンドウコピーで検知可能になるまで，RMA操作のターゲットとしてアクセスしてはならない．この規則には1つ例外がある．同じ変数が，同じウィンドウで，同じ定義済みのデータ型を使用して，同じ操作を使用する2つの同時並行的なアキュムレートにより更新される場合である．

3. プットまたはアキュムレートは，別の（重複する）対象ウィンドウに対するプットまたはアキュムレート更新，またはローカルな更新が対象ウィンドウ内の領域で開始されたら，その更新がウィンドウの公開コピーで検知可能になるまで，対象ウィンドウにアクセスしてはならない．逆に，ウィンドウ内の領域へのプロセスメモリのローカルな更新は，その対象ウィンドウへのプットまたはアキュムレート更新が開始されたら，プットまたはアキュムレート更新がプロセスメモリで検知可能にな

るまで，開始してはならない．どちらの場合も，ウィンドウの重ならない領域にアクセスする場合であっても，操作にこの制限が適用される．

これらの規則に違反するプログラムは誤りである．

根拠 正しいRMAアクセスに関する最後の制限は，ウィンドウ内の重複しない領域への同時並行的なアクセスも禁止しているため，過度な制限に感じられるかもしれない．これを制限する理由は，アーキテクチャによっては，明示的な一貫性の回復操作が同期ポイントで必要となることがあるためである．ストアによってローカルに更新された領域と，プットまたはアキュムレート操作によってリモートに更新された領域とでは，異なる操作が必要となることがある．この制限がないと，MPIライブラリにより，ウィンドウ内のどの領域がプットまたはアキュムレート呼び出しによって更新されたかを正確に追跡しなければならなくなる．このような情報を管理するための過剰なオーバーヘッドは許容できないであろう． （**根拠の終わり**）

ユーザへのアドバイス 以下の規則に従うことにより，正しいプログラムを記述することができる．

フェンス： フェンス呼び出しの間の各期間中に，プットやアキュムレート呼び出し，またはローカルなストアいずれかによって，各ウィンドウを更新してよいが，両方を行ってはならない．プットまたはアキュムレート呼び出しによって更新された領域は同じ期間中にアクセスしてはならない（ただし，複数のアキュムレート呼び出しにより同じ領域を同時並行的に更新する場合は例外）．ゲット呼び出しによってアクセスされる領域は，同じ期間中に更新してはならない．

ポストースタートーコンプリートーウェイト： ウィンドウがポストされている間，そのウィンドウに対しプットまたはアキュムレート呼び出しによる更新が行われている場合は，そのウィンドウはローカルに更新すべきでない．また，ウィンドウがポストされている間，プットまたはアキュムレート呼び出しにより更新された領域は，アクセスされるべきでない（ただし，複数のアキュムレート呼び出しにより同じ領域を同時並行的に更新する場合は例外）．また，ウィンドウがポストされている間，ゲット呼び出しによってアクセスされる領域は，更新されるべきでない．

ポスト－スタート同期を使用すると，ターゲットプロセスはそのウィンドウがRMAアクセスの可能な状態であることをオリジンプロセスに伝えることができ，コンプリート－ウェイト同期を使用すると，オリジンプロセスはそのウィンドウに対するRMAアクセスが完了したことをターゲットプロセスに伝えることができる．

ロック： 衝突がありうる場合，ウィンドウの更新が排他ロックによって保護される．衝突のないアクセス（読み取り専用アクセス，アキュムレートアクセスなど）は，ローカルアクセスとRMAアクセスの両方について，共有ロックによって保護される．

ウィンドウまたは同期モードの変更： プロセスメモリとウィンドウコピーの値が同じであることが保証されているときに，同期モードを変更したり，重複する2つのウィンドウに属する領域へのアクセスに使用するウィンドウを変更したりすることができる．このことは，ウィンドウへのRMAアクセスがフェンスで同期されている場合はローカルなMPI_WIN_FENCEの呼び出しの後にも当てはまり，アクセスがポスト－スタート－コンプリート－ウェイトで同期されている場合はローカルなMPI_WIN_WAITの呼び出しの後にも当てはまり，アクセスがロックで同期されている場合はオリジン（ローカルまたはリモート）でのMPI_WIN_UNLOCKの呼び出しの後にも当てはまる．

また，プロセスはゲット操作が完了するまではゲット操作のローカルバッファにアクセスしてはならず，プットまたはアキュムレート操作が完了するまではプットまたはアキュムレート操作のローカルバッファを更新してはならない．

RMA同期操作は，更新が公開ウィンドウおよび非公開ウィンドウで検知可能になるのを保証するタイミングを定義する．更新がそれより早く検知可能になる場合があるが，このような動作は実装依存である．（**ユーザへのアドバイス終わり**）

意味論を以下の例で説明する．

例 **11.11** 規則5:

```
Process A:                  Process B:
                            window location X

                            MPI_Win_lock(EXCLUSIVE,B)
                            store X /* local update to private copy of B */
                            MPI_Win_unlock(B)
                            /* now visible in public window copy */

MPI_Barrier                 MPI_Barrier

MPI_Win_lock(EXCLUSIVE,B)
MPI_Get(X) /* ok, read from public window */
MPI_Win_unlock(B)
```

例 **11.12** 規則6:

```
Process A:                  Process B:
                            window location X

MPI_Win_lock(EXCLUSIVE,B)
MPI_Put(X) /* update to public window */
MPI_Win_unlock(B)

MPI_Barrier                 MPI_Barrier

                            MPI_Win_lock(EXCLUSIVE,B)
                            /* now visible in private copy of B */
                            load X
                            MPI_Win_unlock(B)
```

Xの非公開コピーはバリアの後で必ずしも更新されていないため，プロセスBでlock/unlockを省略するとロードで更新前の値が返されることがある．

例 11.13 規則では，以下のシーケンスおいて，Bがロック前にローカルなストアによって更新したXの値が，プロセスAで検知可能であることを保証していない．

```
Process A:                  Process B:
                            window location X

                            store X /* update to private copy of B */
                            MPI_Win_lock(SHARED,B)
MPI_Barrier                 MPI_Barrier

MPI_Win_lock(SHARED,B)
MPI_Get(X) /* X may not be in public window copy */
MPI_Win_unlock(B)
                            MPI_Win_unlock(B)
                            /* update on X now visible in public window */
```

例 11.14 以下のシーケンスで

```
Process A:                  Process B:
window location X
window location Y

store Y
MPI_Win_post(A,B) /* Y visible in public window */
MPI_Win_start(A)            MPI_Win_start(A)

store X /* update to private window */

MPI_Win_complete            MPI_Win_complete
MPI_Win_wait
/* update on X may not yet visible in public window */

MPI_Barrier                 MPI_Barrier

                            MPI_Win_lock(EXCLUSIVE,A)
                            MPI_Get(X) /* may return an obsolete value */
                            MPI_Get(Y)
                            MPI_Win_unlock(A)
```

プロセスAによるMPI_WIN_WAIT呼び出しもMPI_WIN_COMPLETE呼び出しも公開ウィンドウコピーでの検知可能性について保証していないため，プロセスBがXの値をプロセスAによるローカルな更新に従って読み取ることを保証していない．BがAによって格納されたXの値を読み取れるようにするには，ローカルなストアをローカルなMPI_PUTで置換し，公開ウィンドウコピーを更新する必要がある．この置換により，XがAのプロセスメモリ内の非公開コピーで検知可能になるのは，プロセスAでMPI_WIN_WAIT呼び出しの後のみであることに注意すること． MPI_WIN_POST呼び出しの前に行われたYの更新は，MPI_WIN_POST呼び出しの後に公開ウィンドウで検知可能になり，プロセスBで正しく取得できるようになる．MPI_GET(Y)呼び出し

はMPI_WIN_START操作によって開始されたアクセスエポックに移すことができ，その場合もプロセスBはAによって格納された値を取得できる．

例 11.15 最後に，以下のシーケンスでは

```
Process A:                  Process B:
                            window location X

MPI_Win_lock(EXCLUSIVE,B)
MPI_Put(X) /* update to public window */
MPI_Win_unlock(B)

MPI_Barrier                 MPI_Barrier

                            MPI_Win_post(B)
                            MPI_Win_start(B)

                            load X /* access to private window */
                                   /* may return an obsolete value */

                            MPI_Win_complete
                            MPI_Win_wait
```

規則(5，6)は，BのXの非公開コピーの更新処理がロードの前に完了していることを保証していない．プロセスAによってプットされた値を読み取れるようにするには，ローカルなロードをローカルなMPI_GET操作に置換するか，またはMPI_WIN_WAITの呼び出し後に配置する必要がある．

11.7.1 アトミック性

同じ領域に対して，同じ操作と定義済みのデータ型を使用して同時並行的にアキュムレートを実行した結果は，ある順序で逐次にその領域に対しアキュムレートを実行したのと同じになる．逆に，2つの領域が両方とも2つのアキュムレート呼び出しによって更新された場合，2つの領域で逆の順番で更新が行われる可能性もある．そのため，MPI_ACCUMULATEの呼び出し全体がアトミックに実行されるという保証はない．アトミック性の欠如の影響は次のように限定的である．前の正当性の条件が暗に意味するとおり，（ターゲットで）MPI_ACCUMULATE呼び出しが完了するまでは，MPI_ACCUMULATEの呼び出しによって更新される領域をロードやアキュムレート以外のRMA呼び出しによってアクセスできない程度である．また，異なる計算順序は異なる結果を導く可能性があるが，コンピュータでの演算が真に結合的でも可換でもない場合のみである．

11.7.2 プログレス

片方向通信は1対1通信と同じプログレスの要件を有する．つまり，一度通信が可能になれば，完了することが保証される．このため，他のRMA呼び出しとの同期のために必要な場合を除いて，RMA呼び出しはローカルな意味論を持つ必要がある．

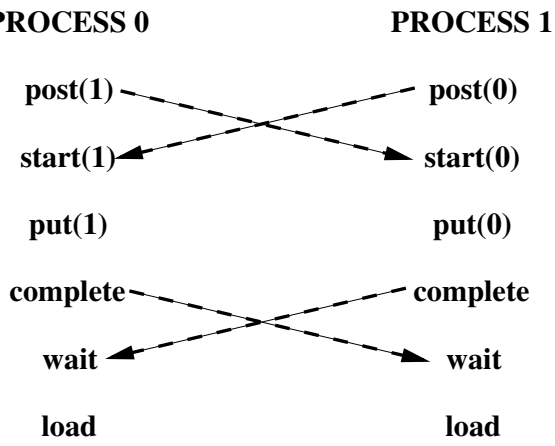

図 11.6: 対称な通信

RMA通信が可能になるタイミングの定義については，あいまいさがある．このあいまいさにより，実装者にとっては1対1通信の場合よりも自由度が高まる．対応する同期（MPI_WIN_FENCE，MPI_WIN_POSTなど）の実行が完了すると，対象ウィンドウへのアクセスが可能になる．オリジンプロセスでは，対応するプット，ゲット，またはアキュムレート呼び出しが実行されるとすぐ，あるいは後続の同期呼び出しの発行を待ってから，RMA通信が可能になる．オリジンとターゲットの両方で通信が可能になったら，通信を完了させる必要がある．

366ページの例11.4の部分コードを考えてみる．対象ウィンドウがポストされていない場合，一部の呼び出しはブロッキングされることがある．しかし，対象ウィンドウがポストされている場合，この部分コードは必ず完了する．データ転送はプット呼び出しが行われるとすぐに開始されることもあるが，後続のコンプリート呼び出しが行われるまで遅延することもある．

372ページの例11.5の部分コードを考えてみる．別のプロセスのロックとの衝突がある場合，一部の呼び出しがブロッキングされることがある．しかし，ロックの競合がない場合，当該部分コードは必ず完了する．

図11.6に示すコードを考えてみる．

各プロセスはプット操作を使用して他のプロセスのウィンドウを更新し，それ自体のウィンドウにアクセスする．ポスト呼び出しはノンブロッキングであり，必ず完了する．ポスト呼び出しが行われたら，そのウィンドウへのRMAアクセスが可能になり，各プロセスでスタート－プット－コンプリートの呼び出しのシーケンスを完了させる必要がある．これらが完了したら，両方のプロセスでウェイト呼び出しを完了させる必要がある．そのため，この通信はデータの転送量に関係なく，必然的にデッドロックは起きない．

最後の例で，各プロセスのポスト呼び出しとスタートの呼び出しの順序が逆であるとする．このとき，各プロセスはスタート呼び出しでブロッキングされ，マッチするポスト呼び出しが行われるのを待つため，コードがデッドロックすることがある．同様に，各プロセスのコンプリート呼び出しとウェイト呼び出しの順序が逆であるとすると，プログラムがデッドロックする．

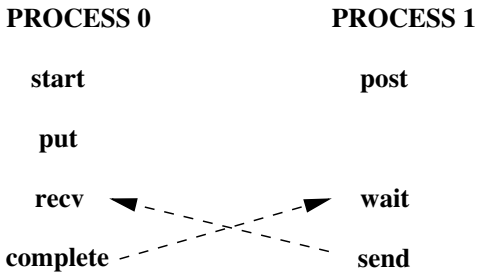

図 11.7: デッドロックする場面

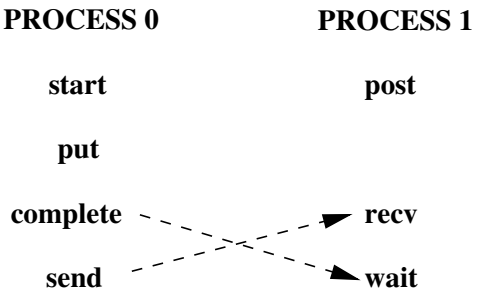

図 11.8: デッドロックなし

以下の2つの例は，コンプリートとウェイトの同期が対称でないことを示している．つまり，ウェイト呼び出しはコンプリートが実行されるまでブロッキングされるが，この逆の状況にはならない．図11.7のコードを考えてみる．

このコードはデッドロックする．プロセス1のウェイトはプロセス0がコンプリートを呼び出すまでブロッキングされ，プロセス0の受信はプロセス1が送信を呼び出すまでブロッキングされるためである．それに対して，図11.8のコードを考えてみる．

このコードはデッドロックしない．プロセス1がポストを呼び出すと，プロセス0でスタート，プット，コンプリートのシーケンスを進行させ，完了させることができる．プロセス0は送信呼び出しに進み，プロセス1の受信呼び出しを完了させることができる．

根拠 MPI実装では，MPI呼び出しでブロッキングされている間も，参加している有効な全ての通信でプロセスがプログレスするよう保証する必要がある．このことは，送受信でも同様であり，RMA通信にも適用される．そのため，図11.8の例では，プロセス0のプットおよびコンプリート呼び出しはプロセス1が受信呼び出しでブロッキングされている間でも必ず完了する．このために，受信呼び出しでブロッキングされている間にプットされたデータを転送するなど，プロセス1の関与が必要となることがある．

同様の問題として，プロセスの計算がビジー状態の場合や，MPI以外の呼び出しでブロッキングされている場合にこのような進行を促す必要があるかどうかというものがある．最後の例で，送受信(send/receive)のペアをソケットへの書き込み／ソケットからの読み取り(write-to-socket/read-from-socket)のペアに置き換えるとする．このとき，MPIではデッドロックを避けるかどうかを規定しない．プロセス1のブ

ロッキング受信呼び出しを非常に長い計算のループに置き換えるとする．ここで，MPI標準の1つの解釈に従って，プロセス1がこの期間内にMPI呼び出しに達していないとしても，プロセス0は一定の遅延時間の後にコンプリート呼び出しから戻らなければならない．別の解釈に従えば，プロセス1がウェイト呼び出しに達するか，別のMPI呼び出しに達するまで，コンプリート呼び出しをブロッキングさせることもできる．どちらの解釈においても，プロセスが計算の無限ループに入るのでなければ，質的な動作は同じで，差は大きな問題とならないこともある．しかし，定量的に見ると違いがある．異なるMPI実装にはこうした解釈の違いが反映される．このあいまいさは残念なことではあるが，多くの実際のコードには影響しないように見える．MPIフォーラムでは，この問題には異論も多く，決定が実装者には大きく影響してもユーザへの影響は小さいため，標準のどの解釈が正しいかという決定をしないことにした． （根拠の終わり）

11.7.3 レジスタとコンパイラの最適化

ユーザへのアドバイス この節の内容は全てユーザへのアドバイスである． （ユーザへのアドバイス終わり）

レジスタに保存される変数とこれらの変数のメモリ値の間には一貫性の問題がある．RMA呼び出しはメモリ（またはキャッシュ）内の変数にアクセスするが，この変数の最新の値はレジスタ内にある．ゲットは変数の最新の値を返さず，レジスタの内容がメモリに格納されるときにプットが上書きされることがある．

この問題を以下のコードで示す:

Source of Process 1	**Source of Process 2**	**Executed in Process 2**
`bbbb = 777`	`buff = 999`	`reg_A:=999`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	
`call MPI_PUT(bbbb`		`stop appl. thread`
`into buff of process 2)`		`buff:=777 in PUT handler`
		`continue appl. thread`
`call MPI_WIN_FENCE`	`call MPI_WIN_FENCE`	
	`ccc = buff`	`ccc:=reg_A`

この例では，変数`buff`はレジスタ`reg_A`によって割り当てられるため，`ccc`には新しい値777ではなく，`buff`の古い値が格納される．

この問題は，いくつかのケースで送受信でも起きる問題であり，第16.2.2節でより詳しく論じる．

MPI実装により，標準に準拠したC言語のプログラムではこの問題が回避される．多くのFortran言語のコンパイラでは，コンパイラの最適化を無効にすることなく，この問題が回避される．しかし，完全に可搬な方法でレジスタの一貫性の問題を回避するには，RMAウィンドウの使用を`COMMON`ブロックに格納された変数か，`VOLATILE`と宣言された変

数（`VOLATILE`はFortran言語の標準の宣言ではないが，多くのFortran言語のコンパイラでサポートされている）に制限する必要がある．詳細と追加の解決策については，507ページの第16.2.2節「レジスタ最適化での問題」で説明する．Fortran言語のそれ以外の問題については，504ページの「データのコピーおよび連続領域配置による問題」も参照すること．

第12章

外部インターフェイス

12.1 はじめに

この章では，まずはじめに，**汎用リクエスト**を生成するのに使用できる呼び出しについて説明する．汎用リクエストを使用すると，ユーザはMPIのものと同様のインターフェースを使用して新しいノンブロッキング操作を生成することができる．これを使用することにより，MPIの上位に新しい機能を階層化することができる．次に，12.3節ではstatusの情報の設定について説明する．これは汎用リクエストのために必要である．

その後の第12.4節では，MPIでのスレッドの扱いについて説明する．スレッドの準拠は必要ではないが，標準にはスレッド機能が提供されている場合の動作について規定する．

12.2 汎用リクエスト

汎用リクエストの目的は，ユーザが新しいノンブロッキング操作を定義できるようにすることである．このような高度なノンブロッキング操作は（汎用化した）リクエストによって表現される．ノンブロッキング操作の基本的な特性は，この操作が非同期に完了に向かう，つまり通常のプログラム実行と同時に進行することである．通常，例えば，独立したスレッドまたはシグナルハンドラで，ユーザコードの実行と同時にコードが実行される必要がある．オペレーティングシステムでは同時実行をサポートするさまざまなメカニズムが用意されている．MPIは標準化を推進するものでもこれらのメカニズムに代わる機能を提供するものでもなく，新しい非同期操作を定義しようとするプログラマが基本となるオペレーティングシステムによって提供されるメカニズムを使用することを想定している．そのため，この章で示す呼び出しは，汎用リクエストに適用した場合に，MPI_WAIT，MPI_CANCELなどのMPI呼び出しの効果を定義するための手段と，汎用操作の完了をMPIに通知するための手段を提供するだけである．

> **根拠** ユーザ定義のノンブロッキング操作の同時実行を達成するためのMPI標準メカニズムを定義することも魅力的である．しかし，オペレーティングシステムで使

用される特定のメカニズムを考慮することなく，このようなメカニズムを定義することは非常に難しい．フォーラムでは，平行処理メカニズムは基本となるオペレーティングシステム管轄であり，MPIにより標準化すべきではなく，MPI標準ではこのようなメカニズムとMPIとの相互作用のみを扱うべきだと考えている． （**根拠の終わり**）

通常のリクエストでは，リクエストに関連する操作はMPI実装により実行され，操作はアプリケーションの介入なしで完了する．汎用リクエストでは，リクエストに関連する操作はアプリケーションによって実行されるため，アプリケーションは操作が完了した時点でMPIに通知する必要がある．これは，MPI_GREQUEST_COMPLETEを呼び出すことにより行われる．MPIは汎用リクエストの「完了」ステータスを管理する．それ以外のリクエストの状態は，ユーザが管理する必要がある．

新しい汎用リクエストは，以下で開始される．

MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)

IN	query_fn	リクエストステータスの問い合わせを行うときに呼び出されるコールバック関数（関数）
IN	free_fn	リクエストを解放するときに呼び出されるコールバック関数（関数）
IN	cancel_fn	リクエストを取り消すときに呼び出されるコールバック関数（関数）
IN	extra_state	その他の状態
OUT	request	汎用リクエスト（ハンドル）

```
int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function
*cancel_fn, void *extra_state, MPI_Request *request)
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
IERROR)
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
{static MPI::Grequest
             MPI::Grequest::Start(const MPI::Grequest::Query_function*
             query_fn, const MPI::Grequest::Free_function* free_fn,
             const MPI::Grequest::Cancel_function* cancel_fn,
             void *extra_state) （廃止された呼び出し形式, 第15.2節を参照）}
```

ユーザへのアドバイス C++言語では汎用リクエストはクラスMPI::Grequestに属し，これは MPI::Requestの派生クラスであることに注意すること．これは，C言語とFortran言語の通常のリクエストと同じ型である． （**ユーザへのアドバイス終わり**）

この呼び出しは汎用リクエストを開始し，requestにそのハンドルを返す．

コールバック関数の構文と意味を以下に示す．全てのコールバック関数は，MPI_GREQUEST_START呼び出しを開始することにより，リクエストに付加された

extra_state引数を渡される．これは，ユーザ定義のリクエストの状態を管理するのに使用できる．

C言語では，問い合わせ関数は以下のようになる．

```
typedef int MPI_Grequest_query_function(void *extra_state,
MPI_Status *status);
```

Fortran言語では以下のようになり，

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

C++言語では以下のようになる．

```
{typedef int MPI::Grequest::Query_function(void* extra_state,
              MPI::Status& status); （廃止された呼び出し形式．第15.2節を参照）}
```

query_fn関数は汎用リクエスト用に返す必要のあるステータスを計算する．ステータスにはリクエストの取消しの成功／失敗に関する情報も含まれる（MPI_TEST_CANCELLEDによって返される結果）．

query_fnコールバックは，このコールバックに関連する汎用リクエストを完了させたMPI_{WAIT|TEST}{ANY|SOME|ALL} 呼び出しによって呼び出される．コールバック関数は，リクエストが呼び出しの発生時に完了している場合，MPI_REQUEST_GET_STATUSの呼び出しによっても呼び出される．どちらの場合も，コールバックはユーザによってMPI 呼び出しに渡された対応するステータス変数への参照を渡され，コールバック関数によって設定されたステータスがMPI呼び出しによって返される．query_fnを呼び出す原因となったMPI関数にユーザがMPI_STATUS_IGNOREまたはMPI_STATUSES_IGNOREを渡した場合，MPIはquery_fnに有効なステータスオブジェクトを渡し，このステータスはコールバック関数が戻ると無視される．query_fn はリクエストに対してMPI_GREQUEST_COMPLETEが呼び出された後にのみ呼び出され，例えばこのリクエストに対してユーザがMPI_REQUEST_GET_STATUSを複数回呼び出した場合，同じ汎用リクエストに対して複数回呼び出されることがある．また，MPI_{WAIT|TEST}{SOME|ALL} の呼び出しにより，MPI呼び出しで汎用リクエストが完了するたびに，query_fnコールバック関数が複数回呼び出されることがある．これらの呼び出しの順序はMPIでは指定されない．

C言語では，解放関数は以下のようになる．

```
typedef int MPI_Grequest_free_function(void *extra_state);
```

Fortran言語では以下のようになる．

```
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

C++言語では以下のようになる．

```
{typedef int MPI::Grequest::Free_function(void* extra_state); （廃止された呼び
              出し形式．第15.2節を参照）}
```

free_fn関数は，ユーザが割り当てたリソースをクリーンアップするために，汎用リクエストの解放時に呼び出される．

free_fnコールバックは，このコールバックに関連する汎用リクエストを完了させたMPI_{WAIT|TEST}{ANY|SOME|ALL} 呼び出しによって呼び出される．free_fnは同じリクエストに対するquery_fnの呼び出しの後に呼び出される．しかし，MPIリクエストで複数の汎用リクエストが完了した場合，free_fnコールバック関数が呼び出される順序はMPIでは指定されない．

free_fnコールバックはMPI_REQUEST_FREEの呼び出しによって解放される汎用リクエストのためにも呼び出される（このようなリクエストに対して WAIT_{WAIT|TEST}{ANY|SOME|ALL} は呼び出されない）．この場合，コールバック関数はMPI呼び出しMPI_REQUEST_FREE(request)またはMPI呼び出し MPI_GREQUEST_COMPLETE(request)のうち，後に発生した方で呼び出される．つまり，この場合，実際の解放コードはMPI_REQUEST_FREEとMPI_GREQUEST_COMPLETEの両方の呼び出しが行われた後すぐに実行される．requestはfree_fnが完了する後まで解放されない．free_fnは，正しいプログラムによりリクエストごとに1回だけ呼び出される．

> **ユーザへのアドバイス** MPI_REQUEST_FREE(request)を呼び出すとrequestハンドルがMPI_REQUEST_NULLに設定される．これにより，汎用リクエストへのこのハンドルは無効になる．しかし，MPIではfree_fnの完了後までオブジェクトを解放しないため，このハンドルのユーザコピーはその時点までは有効である．MPI_GREQUEST_COMPLETEの後までfree_fnは呼び出されないため，この呼び出しのためにこのハンドルのユーザコピーを使用することができる．MPIによるオブジェクトの解放はfree_fnの実行後に行われることに注意する必要がある．この時点で，requestハンドルのユーザコピーは有効なリクエストを指さなくなる．この場合，MPIはユーザコピーをMPI_REQUEST_NULLに設定しないため，ユーザの責任でこの古いハンドルにアクセスしないようにする必要がある．これは，MPIがオブジェクトの解放を後の時点まで保留し，ユーザもそれを知っている特別なケースである．（**ユーザへのアドバイス終わり**）

C言語では，取消し関数は以下のようになる．

```
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
```

Fortran言語では以下のようになる．

```
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    LOGICAL COMPLETE
```

C++言語では以下のようになる．

```
{typedef int MPI::Grequest::Cancel_function(void* extra_state,
              bool complete); （廃止された呼び出し形式．第15.2節を参照）}
```

cancel_fn関数は汎用リクエストの取消しを開始するために呼び出される．これはMPI_CANCEL(request)によって呼び出される．リクエストで

MPI_GREQUEST_COMPLETEがすでに呼び出されている場合はMPIはコールバック関数にcomplete=trueを渡し，それ以外の場合はcomplete=falseを渡す．

全てのコールバック関数はエラーコードを返す．コールバック関数を呼び出したMPI関数により，返されたコードは適切なエラーコードとして扱われる．たとえば，エラーコードが返される場合，コールバック関数によって返されるエラーコードは，そのコールバック関数を呼び出したMPI関数によって返される．query_fnとfree_fnの両方を呼び出すMPI_{WAIT|TEST}{ANY} 呼び出しの場合，MPI呼び出しは最後のコールバック，つまりfree_fnによって返されるエラーコードを返す．MPI_{WAIT|TEST}{SOME|ALL}の呼び出しの1つ以上のリクエストがエラーとなった場合，MPI呼び出しはMPI_ERR_IN_STATUSを返す．このような場合，MPI呼び出しがステータスの配列を渡されたとすると，MPIは完了した汎用リクエストに対応する各ステータスに，対応するfree_fnコールバック関数の呼び出しによって返されたエラーコードを返す．しかし，MPI関数がMPI_STATUSES_IGNOREを渡された場合，各コールバック関数によって返された個々のエラーコードは消失する．

> **ユーザへのアドバイス** status query_fnはMPI_WAITまたはMPI_TESTによって呼び出されることがあり，この場合はstatusのエラーフィールドを変更することは**できない**ため，query_fnではstatusのエラーフィールドを設定してはならない．MPIライブラリはquery_fnが呼び出される「コンテクスト」を知っており，返されるエラーコードをstatusのエラーフィールドに設定するタイミングを正しく決定することができる． （**ユーザへのアドバイス終わり**）

MPI_GREQUEST_COMPLETE(request)

INOUT	request	汎用リクエスト（ハンドル）

```
int MPI_Grequest_complete(MPI_Request request)
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
{void MPI::Grequest::Complete()  （廃止された呼び出し形式, 第15.2節を参照）}
```

この呼び出しは，汎用リクエストrequestによって示される操作が完了したことをMPI に通知する（第2.4節の定義を参照）．MPI_WAIT(request,status)の呼び出しが戻り，MPI_TEST(request, flag, status)の呼び出しがflag=trueを返すのは，MPI_GREQUEST_COMPLETEの呼び出しによりこれらの操作が完了したと宣言された後のみである．

MPIはコールバック関数によって実行されるコードに制限を課さない．しかし，新しいノンブロッキング操作は，MPI_TEST，MPI_REQUEST_FREE，MPI_CANCELといったMPI呼び出しに関する一般的な意味規則が守られるように定義される必要がある．例えば，これらの呼び出しはすべてローカルであり，ノンブロッキングであると考えられる．そのため，コールバック関数query_fn，free_fn，またはcancel_fnによるブロッキングMPI通信呼び出しを呼び出すのは，これらの呼び出しが有限の時間内に戻ることが保証されるようなコンテクストに制限する必要がある．MPI_CANCELが呼び出されたら，

他のプロセスの状態に関係なく（操作は「ローカル」の意味を獲得している），取消し操作は一定の時間内に完了しなければならない．成功の場合も，失敗の場合も，副作用が発生してはならない．ユーザは新しく定義した操作に対して，これらの同じ特性を保証する必要がある．

実装者へのアドバイス MPI_GREQUEST_COMPLETEを呼び出すと，ブロッキングされているユーザプロセス/スレッドのブロッキング状態が解除されることがある．MPIライブラリは，ブロッキングされていたユーザの計算が再開されるよう保証する必要がある．（実装者へのアドバイス終わり）

12.2.1 例

例 12.1 この例では，バイナリツリーを使用してintでのユーザ定義のリデュース操作を行うためのコードを示す．ルート以外の各ノードが2つのメッセージを受信し，これらを合計し，上位ノードに送信する．ステータスが返されないものとし，操作は取り消せないものとする．

```
typedef struct {
   MPI_Comm comm;
   int tag;
   int root;
   int valin;
   int *valout;
   MPI_Request request;
   } ARGS;


int myreduce(MPI_Comm comm, int tag, int root,
              int valin, int *valout, MPI_Request *request)
{
   ARGS *args;
   pthread_t thread;

   /* start request */
   MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);

   args = (ARGS*)malloc(sizeof(ARGS));
   args->comm = comm;
   args->tag = tag;
   args->root = root;
   args->valin = valin;
   args->valout = valout;
   args->request = *request;

   /* spawn thread to handle request */
   /* The availability of the pthread_create call is system dependent */
   pthread_create(&thread, NULL, reduce_thread, args);

   return MPI_SUCCESS;
}

/* thread code */
void* reduce_thread(void *ptr)
```

```
{
   int lchild, rchild, parent, lval, rval, val;
   MPI_Request req[2];
   ARGS *args;

   args = (ARGS*)ptr;

   /* compute left,right child and parent in tree; set
      to MPI_PROC_NULL if does not exist  */
   /* code not shown */
   ...

   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
   val = lval + args->valin + rval;
   MPI_Send( &val, 1, MPI_INT, parent, args->tag, args->comm );
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
   MPI_Grequest_complete((args->request));
   free(ptr);
   return(NULL);
}

int query_fn(void *extra_state, MPI_Status *status)
{
   /* always send just one int */
   MPI_Status_set_elements(status, MPI_INT, 1);
   /* can never cancel so always true */
   MPI_Status_set_cancelled(status, 0);
   /* choose not to return a value for this */
   status->MPI_SOURCE = MPI_UNDEFINED;
   /* tag has no meaning for this generalized request */
   status->MPI_TAG = MPI_UNDEFINED;
   /* this generalized request never fails */
   return MPI_SUCCESS;
}


int free_fn(void *extra_state)
{
   /* this generalized request does not need to do any freeing */
   /* as a result it never fails here */
   return MPI_SUCCESS;
}


int cancel_fn(void *extra_state, int complete)
{
   /* This generalized request does not support cancelling.
      Abort if not already done.  If done then treat as if cancel failed.*/
   if (!complete) {
     fprintf(stderr,
             "Cannot cancel generalized request - aborting program\n");
     MPI_Abort(MPI_COMM_WORLD, 99);
     }
   return MPI_SUCCESS;
}
```

12.3 情報とステータスの関連付け

MPIは，1対1操作以外にも何種類かのリクエストをサポートしている．ここには入出力用のMPI呼び出しから汎用リクエストまで，タイプの異なるものが含まれる．これらの呼び出しでは同じリクエストメカニズムを使用できるのが望ましい．そうすることにより，タイプの異なるリクエストで待ち合わせやテストを行うことができる．一方，MPI_{TEST|WAIT}{ANY|SOME|ALL} はリクエストに関する情報を持ったステータスを返す．リクエストの汎用化においては，ステータスオブジェクトに返す情報を定義する必要がある．

各MPI呼び出しではステータスオブジェクトの適切なフィールドが埋められる．使われないフィールドの値は未定義となる．MPI_{TEST|WAIT}{ANY|SOME|ALL} を呼び出すと，ステータスオブジェクトの任意の（訳者註：1つ以上の）フィールドを変更することができる．特に，未定義のフィールドを変更することができる．指定のリクエストのための意味のある値を持つフィールドは，新しいリクエストの節で定義される．

汎用リクエストにはこれ以外にも検討すべき事項がある．ここで，ユーザはリクエストに対応するための関数を用意する．他のMPI呼び出しと違い，ユーザはstatusに返される情報を提供する必要がある．status引数は，ステータスを設定する必要のあるコールバック関数に直接渡される．ユーザは5つのステータス値のうちの3つを直接設定することができる．countおよびcancelフィールドは不可視である．これに対処するため，以下の関数が用意されている．

MPI_STATUS_SET_ELEMENTS(status, datatype, count)

INOUT	status	countを関連付けるステータス（ステータス）
IN	datatype	countと関連付けるデータ型（ハンドル）
IN	count	statusと関連付ける要素の数（整数型）

```
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
int count)
MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
{void MPI::Status::Set_elements(const MPI::Datatype& datatype, int count)
              （廃止された呼び出し形式, 第15.2節を参照）}
```

この呼び出しは，MPI_GET_ELEMENTSの呼び出しがcount を返すように，statusの不可視部分を変更する．MPI_GET_COUNTは互換性のある値を返す．

> **根拠** 要素の数は，整数でないデータ型の個数も扱えるため，カウントの代わりに設定される． （**根拠の終わり**）

後続のMPI_GET_COUNT(status, datatype, count)またはMPI_GET_ELEMENTS(status, datatype, count)の呼び出しでは，MPI_STATUS_SET_ELEMENTSの呼び出しで使用したdatatype引数と同じ型仕様を持つdatatype引数を使用する必要がある．

根拠 これは，countが受信操作によって設定される場合に課される制限と同様である．その場合，MPI_GET_COUNTおよびMPI_GET_ELEMENTSの呼び出しでは，受信呼び出しで使用されるデータ型と同じ型仕様を持つdatatypeを使用する必要がある．（**根拠の終わり**）

MPI_STATUS_SET_CANCELLED(status, flag)

INOUT	status	取消しフラグを関連付けるステータス（ステータス）
IN	flag	リクエストが取り消された場合，true（論理型）

```
int MPI_Status_set_cancelled(MPI_Status *status, int flag)
MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
{void MPI::Status::Set_cancelled(bool flag)  （廃止された呼び出し形式, 第15.2節を参照）
                }
```

flagフラグがtrueに設定されている場合，後続の MPI_TEST_CANCELLED(status, flag)の呼び出しもflag = trueを返す．それ以外の場合，falseを返す．

ユーザへのアドバイス 再利用が意図されていない値のステータスフィールドを再利用することはお勧めできない．再利用すると，ステータスオブジェクトの使用時に予期しない結果が生じることがある．例えば，値が範囲外の場合やこのようなエラーを検出できない場合，MPI_GET_ELEMENTSを呼び出すとエラーが発生することがある．汎用リクエストを備えるextra_state引数は，ステータスに論理的に属していない情報を返すのに使用できる．また，MPIによって内部的に設定されるステータスの値（MPI_RECVなど）を変更すると，予期しない結果が生じることがあるため，変更しないことを強くアドバイスする．（**ユーザへのアドバイス終わり**）

12.4 MPIとスレッド

ここでは，MPI呼び出しとスレッドの相互作用について規定する．**スレッド対応**のMPI実装の最小限の要件を示し，スレッド環境の初期化に使用できる関数を定義する．MPIは，スレッドがサポートされていないか，十分に機能しない環境でも実装できる．そのため，すべてのMPI実装でこの章に示すすべての要件を満たす必要はない．

この章では一般的にPOSIXスレッドと類似のスレッドパッケージを想定しているが[29]，スレッドの呼び出しの構文と意味はここには示さない．これらは本書の対象範囲ではない．

12.4.1 全般

スレッド対応の実装では，MPIプロセスはマルチスレッド環境となることがある．各スレッドはMPI呼び出しを発行することができるが，スレッドを個別に指定することは

できず，送信または受信呼び出しのランクはスレッドではなく，プロセスを識別する．プロセスに送信されたメッセージはこのプロセス内の任意のスレッドで受信できる．

> **根拠** このモデルはプロセス間通信のPOSIXモデルに対応しており，プロセスは単一スレッドではなく，マルチスレッド環境となるが，このことはこのプロセスの外部インターフェースには影響しない．MPI「プロセス」が単一POSIX プロセス内のPOSIXスレッドであるようなMPI実装は，この定義によればスレッド対応ではない（実際にはその「プロセス」は単一スレッドである）．（**根拠の終わり**）
>
> **ユーザへのアドバイス** 同一アプリケーション内のスレッドが競合する通信呼び出しを発行する場合，競合を回避するのはユーザの責任である．ユーザは，スレッドごとに異なるコミュニケータを使用することにより，同一プロセス内の2つのスレッドが競合する通信呼び出しを発行しないようにすることができる．（**ユーザへのアドバイス終わり**）

スレッド対応の実装の2つの主な要件を以下に示す．

1. MPI呼び出しはすべて「スレッドセーフ」である．つまり，並行に実行される2つのスレッドはMPI呼び出しを行うことができ，その実行がインタリーブされた場合でも，順番に呼び出しが行われたのと同じ結果となる．

2. ブロッキングしうるMPI呼び出しがブロックした場合，呼び出し側のスレッドのみがブロッキングされ，別のスレッドがあれば実行することができる．呼び出し側のスレッドは，待ち合わせているイベントが発生するまでブロッキングされる．ブロッキングされていた通信が可能になり，開始できるようになると，有限時間内に呼び出しが完了し，スレッドは実行可能としてマークされる．あるスレッドがブロッキングされていても，同じプロセスの他の実行可能なスレッドの進行を妨げることはなく，MPI呼び出しの実行を妨げることもない．

例 12.2 プロセス0は2つのスレッドで構成される．1つ目のスレッドはブロッキング送信呼び出しMPI_Send(buff1, count, type, 0, 0, comm)を実行し，2つめのスレッドはブロッキング受信呼び出しMPI_Recv(buff2, count, type, 0, 0, comm, &status)を実行する．つまり，1つ目のスレッドがメッセージを送信し，2つめのスレッドがそれを受信する．この通信は常に成功する．最初の要件に従い，この実行は2つの呼び出しのインターリービングに対応する．2つめの要件に従い，呼び出しは呼び出し側のスレッドのみをブロッキングすることができ，他方のスレッドの進行を妨げることはない．送信呼び出しが受信呼び出しよりも先に進んだ場合，送信スレッドがブロッキングされることがあるが，これによって受信スレッドの実行が妨げられることはない．そのため，受信呼び出しが行われる．両方の呼び出しが行われると，通信が可能になり，両方の呼び出しが完了する．それに対して，送信が発行され，その後で対応する受信が発行される単一スレッドのプロセスではデッドロックが発生することがある．マルチスレッド環境の実装では，ブロッキングされた呼び出しが他のスレッドの進行を妨げる可能性がないため，その進行にはより厳しい要件が課される．

実装者へのアドバイス MPI呼び出しは，プロセスにグローバルな1つのロックを使用してMPIコードを保護するなどの方法で1つずつ順に実行することにより，スレッドセーフにすることができる．しかし，ブロッキングされた操作は，プロセス内の別のスレッドの進行を妨げることがあるため，ロックを保持することはできない．ロックは，送信の発行，送信の完了などのアトミックなでローカルに完了するサブ操作の期間中のみ保持され，この間に解放される．精密なロックは同時並行性を高めることができるが，ロックのオーバーヘッドも高くなる．同時並行性は，MPIプロトコルの一部を別のサーバースレッドで実行することでも実現できる．（**実装者へのアドバイス終わり**）

12.4.2 明確化

初期化と完了 MPI_FINALIZEの呼び出しは，MPIを初期化したのと同じスレッドで行う必要がある．このスレッドのことを**メインスレッド**と呼ぶ．この呼び出しは，すべてのプロセススレッドでMPI呼び出しが完了し，保留中の通信または入出力操作がない状態で行う必要がある．

> **根拠** この制約により実装を簡素化できる． （**根拠の終わり**）

マルチスレッドでの同じリクエストの完了 2つのスレッドがブロッキングされ，同じリクエストで待ち合わせているプログラムはエラーになる．同様に，同じリクエストを含むリクエスト配列に対しMPI_{WAIT|TEST}{ANY|SOME|ALL} を同時に呼び出すことはできない．MPIでは，1つのリクエストは1回のみ完了させることができる．この規則に違反するwaitまたはtestの組み合わせはエラーとなる．

> **根拠** これは，マルチスレッド実行の結果によりMPI呼び出しがインターリーブされる，という見方と整合する．単一スレッドの実装では，リクエストでwaitが発行されると，そのリクエストハンドルは，同じハンドルで2回目のwaitが発行可能になる前に無効化される．スレッドを使用して，リクエストを無効化することなくをブロッキングすることができるため，ユーザの責任により，別のスレッドのMPI_WAIT{ANY|SOME|ALL} で同じリクエストを使用しないようにすること．この制限により，通信や入出力イベントで1つのスレッドのみがブロッキングされるため，実装を簡素化することができる． （**根拠の終わり**）

先行するMPI_PROBEまたはMPI_IPROBEの呼び出しによって返されたsourceおよびtag値を使用する受信呼び出しは，プローブの後かつその受信の前に他の一致する受信がない場合のみ，プローブ呼び出しと一致するメッセージを受信する．マルチスレッド環境では，適切な相互排他論理を使用してこの条件を適用するのはユーザの責任である．これを適用するには，各コミュニケータがプロセスごとに1つのスレッドによってのみ使用されていることを確認する．

集団呼び出し コミュニケータ，ウィンドウ，またはファイルハンドルでの集団呼び出しの対応付けは，各プロセスで呼び出しが発行される順序に従って行われる．並行するスレッドが同じコミュニケータ，ウィンドウ，またはファイルハンドル上でこのような呼び出しを発行する場合，スレッド間同期を使用して呼び出しが正しい順序で行われることを確認するのはユーザの責任である．

> **ユーザへのアドバイス** コミュニケータcommの各MPIプロセスがそれぞれ，3つの並行するスレッドを使用して，各MPIプロセスのスレッドAがcommで集団操作を呼び出し，スレッドBが前にcommでオープンされた既存のファイルハンドルでファイル操作を呼び出し，スレッドCが同様に前にcommで生成された既存のウィンドウハンドルで片方向操作を呼び出すことができる． （**ユーザへのアドバイス終わり**）
>
> **根拠** MPI_FILE_OPENおよびMPI_WIN_CREATEですでに規定されているように，ファイルハンドルとウィンドウハンドルは，コミュニケータそのものではなく，基本となるコミュニケータのプロセスのグループのみを継承する．コミュニケータ，ウィンドウハンドル，およびファイルハンドルへのアクセスは相互に影響を及ぼしてはならない． （**根拠の終わり**）
>
> **実装者へのアドバイス** ファイルまたはウィンドウ操作の実装で内部的にMPI通信が使用される場合，複製されたコミュニケータがファイルまたはウィンドウブジェクトにキャッシュされることがある． （**実装者へのアドバイス終わり**）

例外ハンドラ 例外ハンドラは必ずしも例外が発生したMPI呼び出しのスレッドコンテクストで実行されるわけではなく，エラーコードを返すスレッドとは異なるスレッドによって実行されることもある．

> **根拠** MPI実装をマルチスレッド化することで，MPI呼び出しを行ったスレッドとは異なるスレッドで通信プロトコルの一部が実行することができる．この設計により，例外が発生したスレッドで例外ハンドラを実行することが可能になる． （**根拠の終わり**）

シグナルと取消しの相互作用 MPI呼び出しを実行するスレッドが（別のスレッドによって）取り消された場合，またはスレッドがMPI呼び出しの実行中にシグナルを捕捉した場合，結果は未定義となる．MPI呼び出しを実行していない場合は，MPIプロセスのスレッドが終了したり，シグナルを補足したり．別のスレッドによって取り消さたりすることがある．

> **根拠** シグナルセーフのC言語ライブラリ関数はほとんどなく，多くは取消しポイント（それを実行するスレッドの取消しが行えるポイント）を持つ．この制限により実装が簡素化される（MPIライブラリが「非同期キャンセルセーフ」または「非同期シグナルセーフ」である必要はない）． （**根拠の終わり**）

ユーザへのアドバイス ユーザは独立した非MPIスレッドでシグナルを捕捉することができる（MPI 呼び出しスレッドでシグナルをマスクし，1つ以上の非MPIスレッドでシグナルのマスク解除をすることにより）．優れたプログラミングのためには，sigwaitの呼び出しで，発生が予測されるシグナルごとに，異なるスレッドをブロッキングする必要がある．ユーザはMPI実装によって使用されるシグナルを捕捉してはならない．各MPI実装では内部で使用するシグナルを文書化する必要があるため，ユーザはこれらのシグナルの使用を避けることができる． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス マルチスレッドで実行する場合，MPIライブラリではスレッドセーフでないライブラリ呼び出しを行ってはならない． （**実装者へのアドバイス終わり**）

12.4.3 初期化

MPIの初期化と，MPIスレッド環境の初期化のために，MPI_INITの代わりに以下の関数を使用することができる．

MPI_INIT_THREAD(required, provided)

IN	required	望ましいスレッドサポートのレベル（整数型）
OUT	provided	提供されるスレッドサポートのレベル（整数型）

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
int *provided)
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
    INTEGER REQUIRED, PROVIDED, IERROR
{int MPI::Init_thread(int& argc, char**& argv, int required)  （廃止された呼び出し形式, 第15.2節を参照）}
{int MPI::Init_thread(int required)  （廃止された呼び出し形式, 第15.2節を参照）}
```

ユーザへのアドバイス C言語およびC++言語では，argcとargvの受渡しは任意である．C言語では，これは適切なポインタを渡すことによって達成される．C++言語では，これらの2つの場合に対応する2つの呼び出し形式を使用して達成される．これは，第8.7節で説明したMPI_INITを使用するのと同様である． （**ユーザへのアドバイス終わり**）

この呼び出しは，MPI_INITを呼び出した場合と同様にMPIを初期化する．また，スレッド環境も初期化する．引数requiredは望ましいスレッドサポートのレベルを指定するのに使用する．使用できる値をスレッドサポートの昇順で示す．

MPI_THREAD_SINGLE 1つのスレッドのみを実行する．

MPI_THREAD_FUNNELED プロセスはマルチスレッドとすることができるが，アプリケーションではメインスレッドでのみMPI呼び出しが行われるようにする必要があ

る（メインスレッドの定義については，403ページのMPI_IS_THREAD_MAINを参照）．

MPI_THREAD_SERIALIZED プロセスはマルチスレッドにすることができ，複数のスレッドでMPI呼び出しを行うことができるが，1度に1つしか呼び出しを実行できないため，2つの異なるスレッドから同時にMPI呼び出しを行うことはない（MPI呼び出しはすべて「逐次化される」）．

MPI_THREAD_MULTIPLE 複数のスレッドで制限なくMPIを呼び出すことができる．

これらの値は単調増加，つまりMPI_THREAD_SINGLE $<$ MPI_THREAD_FUNNELED $<$ MPI_THREAD_SERIALIZED $<$ MPI_THREAD_MULTIPLEである．

MPI_COMM_WORLD内のプロセスによっては，必要となるスレッドサポートのレベルが異なることがある．

この呼び出しは，MPIによって提供される実際のスレッドサポートのレベルに関する情報をprovidedに返す．これは上記の4つの値のいずれかとすることができる．MPI_INIT_THREADによって提供されうるスレッドサポートのレベルは実装で決まり，プログラムの実行の開始前にユーザが（mpiexecの引数などで）指定した情報に基づいて決めることができる．可能であれば，この呼び出しはprovided = requiredを返す．これに失敗した場合，この呼び出しはprovided > requiredとなるように（ユーザが必要とするよりも強いサポートレベルが得られるように）最低サポートレベルを返す．最後に，ユーザの要件を満たすことができない場合，この呼び出しはprovidedに最高サポートレベルを返す．

スレッド対応のMPI実装はprovided = MPI_THREAD_MULTIPLEを返すことができる．このような実装は，requiredの値に関係なく，常にprovided = MPI_THREAD_MULTIPLEを返すことができる．それとは正反対に，スレッド対応でないMPIライブラリは，requiredの値に関係なく，常にprovided = MPI_THREAD_SINGLEを返すことができる．

MPI_INITの呼び出しはrequired = MPI_THREAD_SINGLEを使用してMPI_INIT_THREADを呼び出したのと同じ効果となる．

ベンダーは，mpiexecの引数など，MPIプログラムの開始時に利用可能なスレッドサポートのレベルを指定するための（実装依存の）手段を提供することができる．これはMPI_INIT やMPI_INIT_THREADの呼び出しの結果に影響する．例えば，MPI_THREAD_MULTIPLEのみが利用できるようにMPIプログラムが起動されているとする．この場合，MPI_INIT_THREAD はrequiredの値に関係なくprovided = MPI_THREAD_MULTIPLEを返し，MPI_INITの呼び出しはMPIのスレッドサポートのレベルをMPI_THREAD_MULTIPLEに初期化する．逆に，4つのすべてのスレッドサポートのレベルが利用できるようにMPIプログラムが起動されているとする．この場合，MPI_INIT_THREADの呼び出しはprovided = requiredを返し，MPI_INITの呼び出しはMPIのスレッドサポートのレベルをMPI_THREAD_SINGLEに初期化する．

根拠 MPIコードが単一スレッド環境で実行される場合，または複数のスレッドがあっても同時に実行されることがない場合は，さまざまな最適化が可能で，相互

排他コードを省略することができる．また，1つのスレッドだけを実行する場合，MPIライブラリではユーザスレッドとの衝突を起こすことなく，スレッドセーフでないライブラリ関数を使用することができる．また，1つの通信スレッド，複数の計算スレッドのモデルは多くのアプリケーションに適合する．例えばプロセスのコードがSMPのクラスタ内のSMPノードで実行するためにコンパイラによって並列処理されているMPI呼び出しを使用する逐次処理として書かれたFortran言語/C言語/C++言語のプログラムである場合，プロセスの計算はマルチスレッドになるが，MPI呼び出しは単一のスレッドで実行されるだろう．

この設計では，ライブラリの静的な呼び出し形式を必要とする環境のため，また現在のマルチスレッドMPIコードの互換性のため，スレッドサポートのレベルの静的指定が可能となっている． （**根拠の終わり**）

実装者へのアドバイス providedがMPI_THREAD_SINGLEでない場合，MPIライブラリではスレッドセーフでないC言語/C++言語/Fortran言語のライブラリ呼び出しを行ってはならない．例えば，`malloc`がスレッドセーフでない環境では，MPIライブラリで`malloc`を使用しないようにする必要がある．

実装者によっては，スレッドサポートのレベルごとに異なるMPIライブラリを使用したいという場合もある．この場合，動的リンクを使用し，MPI_INIT_THREADの呼び出し時にリンクされるライブラリを選択する．これが可能でないときは，必要なスレッドサポートのレベルがリンク時に指定される場合のみ，それより下のレベルのスレッドサポートの最適化が行われる． （**実装者へのアドバイス終わり**）

現在のスレッドサポートのレベルの問い合わせには，以下の関数を使用することができる．

MPI_QUERY_THREAD(provided)

OUT	provided	提供されるスレッドサポートのレベル（整数型）

```
int MPI_Query_thread(int *provided)
MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR
{int MPI::Query_thread() （廃止された呼び出し形式, 第15.2節を参照）}
```

この呼び出しは現在のスレッドサポートのレベルをprovidedに返す．これは，MPIがMPI_INIT_THREAD()の呼び出しによって初期化されている場合は，MPI_INIT_THREADによってprovidedに返される値である．

MPI_IS_THREAD_MAIN(flag)

OUT	flag	呼び出しスレッドがメインスレッドの場合はtrue，それ以外の場合はfalse（論理型）

```
int MPI_Is_thread_main(int *flag)
MPI_IS_THREAD_MAIN(FLAG, IERROR)
```

```
    LOGICAL FLAG
    INTEGER IERROR
{bool MPI::Is_thread_main()} （廃止された呼び出し形式, 第15.2節を参照）
```

これがメインスレッド（MPI_INITまたはMPI_INIT_THREADを呼び出したスレッド）であるかどうかを調べるため，スレッドでこの関数を呼び出すことができる．

この章で示したルーチンはすべて，すべてのMPI実装でサポートされていなければならない．

根拠 MPIライブラリは，これらの関数の呼び出しが記述されたポータブルなコードが正しくリンクできるように，スレッドがサポートされていない場合でも，これらの呼び出しを備えていなければならない．MPI_INITは，現在のMPIコードとの互換性を保証するため，引き続きサポートされている． （**根拠の終わり**）

ユーザへのアドバイス スレッドの生成はMPIの初期化の前に行うことができるが，MPI_INITIALIZED以外のMPI呼び出しは，1つのスレッド（結果として，メインスレッドとなる）によってMPI_INIT_THREADが呼び出されるまで実行してはならない．これによりマルチスレッドのプロセスからMPI実行を開始することができる．

提供されるスレッドサポートのレベルは，そのプロセスでMPIが初期化されるとき（またはその前）に1回だけ指定できるMPIプロセスのグローバルな属性である．サードパーティの可搬なライブラリは，提供されるスレッドサポートのレベルに対応するように記述されていなければならない．対応していない場合，その使用が特定のスレッドサポートのレベルに制限される．このようなライブラリが特定のスレッドサポートのレベル，例えばMPI_THREAD_MULTIPLEでのみ実行可能な場合，MPI_QUERY_THREADを使用してユーザが正しいスレッドサポートのレベルにMPIを初期化しているかどうかを確認し，正しく初期化されていない場合は例外を発生させることができる． （**ユーザへのアドバイス終わり**）

第13章

入出力

13.1 はじめに

POSIXは広い範囲で可搬なファイルシステムのモデルを備えているが，POSIXインターフェイスでは並列入出力で求められる可搬性と最適化が得られない．

効率化のための重要な最適化（グルーピング[35]，集団バッファリング[6, 13, 36, 39, 46]，ディスク指向入出力[31]など）のためには並列入出力システムが2種類のinterfaceを持つ必要がある． 1番目は，プロセス間でのファイルデータの分割をサポートする上位レベルのインターフェイスであり， 2番目は，複数のプロセスメモリから複数のファイルへ，またはその逆へ，グローバルなデータ構造を完全に転送できる集団インターフェイスである．また，さらなる効率向上は，非同期入出力，ストライドアクセス，ストレージデバイス（ディスク）上での物理的ファイル配置の制御により得られる．この章で説明する入出力環境はこれらの機能を提供する．

共有ファイルへのアクセス用の共通パターン（ブロードキャスト，リデュース，スキャッタ，ギャザー）を表現する入出力アクセスモードを定義するのではなく，派生データ型を使用してデータの分割を表現するというアプローチを取る． 定義済みの限られたアクセスパターンの集合に比べて，このアプローチには自由度と表現力を増すという利点がある．

13.1.1 定義

ファイル MPIファイルは型付けされたデータ項目の順序集合である． MPIでは，これらの項目の必要な部分集合に対してランダムアクセスまたは逐次アクセスが行える．ファイルはプロセスのグループによって集団的に開かれる．ファイルでのすべての集団入出力呼び出しは，このグループに対して集団的である．

変位 ファイルの**変位**とは，ファイルの先頭に対する絶対バイト位置のことである．変位により，**ビュー**の開始領域が定義される．「ファイル変位」と「ファイルの「型マップ変位」は異なるため，注意する必要がある．

etype etype（**要素**データ型）はデータアクセスと位置付けの単位である．これはMPIの

任意の定義済みまたは派生データ型とすることができる．生成されるすべての型マップ変位が非負で単調非減少である限り，任意のMPIデータ型コンストラクタルーチンを使用して派生etypeを構成することができる．データアクセスでは，etype単位でetype型のデータ項目全体の読み取りまたは書き込みが行われる．オフセットはetypeのカウントとして表現され，ファイルポインタはetypeの先頭を指す．コンテクストによって，“etype”という用語は要素データ型の3つの側面のいずれか，つまり特定のMPIの型，その型のデータ項目，またはその型の範囲を記述するのに使用される．

ファイル型 ファイル型はプロセス間でファイルを分割するための基本で，ファイルへのアクセスのための雛型を定義する．ファイル型は，単一のetype，または同じetypeの複数のインスタンスから構成される派生MPIデータ型である．また，ファイル型の穴の範囲はetypeの範囲の倍数とする必要がある．ファイル型の型マップ内の変位は個々に異なっている必要はないが，非負で単調非減少でなければならない．

ビュー ビューは，順序が規定されたetypeの集合として，開かれたファイルから検知可能でまたアクセス可能な現在のデータの集合を定義する．各プロセスは，変位，etype，ファイル型という3つの量で定義されたファイルの独自のビューを備えている．ファイル型によって示されるパターンは，変位を先頭として，反復してビューを定義する．反復のパターンは， MPI_TYPE_CONTIGUOUSにファイル型と任意の大きさのカウントを渡した場合に生成されるのと同じパターンとして定義される．図13.1に，敷き詰めの方法を示す．この例のファイル型は，最初の穴と最後の穴がビューで反復されるように，明示的に下限と上限を設定する必要があることに注意すること．ビューはプログラムの実行中にユーザが変更できる．デフォルトのビューは直線のバイトストリーム（変位は0で，etypeとファイル型はMPI_BYTE）である．

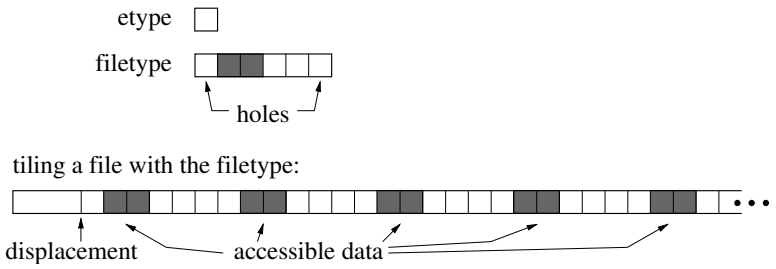

図 13.1: etypesとファイル型

プロセスのグループは，互いに重ならないビューを使用することで，スキャッタ／ギャザーパターンなどのグローバルデータ分配が行える（図13.2を参照）．

オフセット オフセットとは，etypeのカウントとして表現される，現在のビューに対するファイルの相対位置のことである．この位置の計算では，ビューのファ

図 13.2: 並列プロセス間のファイルの分割

イル型の穴はスキップされる．オフセット0はビュー内で（ビュー内の変位と最初の穴をスキップした後で）最初に検知可能であるetypeの位置である．例えば，図13.2のプロセス1のオフセット2は，ファイル内で変位の後の8番目のetypeの位置である．「明示的なオフセット」とは，明示的なデータアクセスルーチンで仮引数として使用されるオフセットである．

ファイルサイズとファイル終端 MPIファイルの**サイズ**はファイルの先頭からバイト単位で測定される．新しく生成されたファイルのサイズは0バイトとなる．サイズを絶対変位として使用することにより，ファイル内の最後のバイトの直後のバイトの位置が求められる．指定のビューに対する**ファイル終端**は，ファイル内の最終バイトの後に始まる現在のビューでアクセス可能な最初のetypeのオフセットである．

ファイルポインタ **ファイルポインタ**とは，MPIによって暗黙に管理されるオフセットである．「個別ファイルポインタ」とは，そのファイルを開いたプロセスごとのローカルなファイルポインタである．「共有ファイルポインタ」とは，そのファイルを開いたプロセスのグループによって共有されるファイルポインタである．

ファイルハンドル **ファイルハンドル**とは，MPI_FILE_OPENによって生成され，MPI_FILE_CLOSEによって解放される不可視オブジェクトである．開かれたファイルに対するすべての操作は，ファイルハンドルを通してそのファイルを参照する．

13.2 ファイル操作

13.2.1 ファイルを開く

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN	comm	コミュニケータ（ハンドル）
IN	filename	開くファイルの名前（文字列）
IN	amode	ファイルのアクセスモード（整数型）
IN	info	infoオブジェクト（ハンドル）
OUT	fh	新しいファイルハンドル（ハンドル）

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
MPI_File *fh)
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER COMM, AMODE, INFO, FH, IERROR
{static MPI::File MPI::File::Open(const MPI::Intracomm& comm,
             const char* filename, int amode, const MPI::Info& info) （廃止
             された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_OPENは，commコミュニケータグループ内のすべてのプロセスでファイル名filenameによって識別されるファイルを開く．MPI_FILE_OPENは集団的ルーチンで，すべてのプロセスはamodeに対して同じ値を渡す必要があり，すべてのプロセスは同じファイルを参照するfilenameを渡す必要がある．（infoの値は違っても構わない．）commはグループ内コミュニケータでなければならず，MPI_FILE_OPENにグループ間コミュニケータを渡すのは誤りである．MPI_FILE_OPENのエラーはデフォルトのファイルエラーハンドラを使用して処理される（465ページの第13.7節を参照）．プロセスは，他のプロセスとは独立して，MPI_COMM_SELFコミュニケータを使用してファイルを開くことができる．返されるファイルハンドルfhは，その後，MPI_FILE_CLOSEを使用してファイルが閉じられるまでファイルにアクセスするために使用できる．MPI_FINALIZEを呼び出す前に，ユーザはMPI_FILE_OPENを使用して開いたすべてのファイルを（MPI_FILE_CLOSEにより）閉じる必要がある．コミュニケータcommはMPI_FILE_OPENの影響を受けず，すべてのMPIルーチン（MPI_SENDなど）で引き続き使用できる．また，commの使用により入出力の動作が妨げられることはない．

filename引数でファイル名を指定するための形式は実装依存であり，文書化する必要がある．

実装者へのアドバイス 実装では，filenameにファイルに関する追加情報を指定する1つ以上の文字列を格納する必要がある場合がある．例えば，ファイルシステムのタイプ（ufs:などのプリフィックス），リモートホスト名（machine.univ.edu:などのプリフィックス），ファイルのパスワード（/PASSWORD=SECRETのサフィックスなど）などを格納する．（**実装者へのアドバイス終わり**）

ユーザへのアドバイス MPIの実装で，ファイルの名前空間はすべてのアプリケーションのすべてのプロセスで同じでない可能性がある．例えば，“/tmp/foo”がプロセスごとに異なるファイルを指していたり，1つのファイルがプロセスの場所によって，異なる名前を持っていたりすることがある．実装ではこのような名前空間のエラーを検出することができない場合があるため，ユーザの責任において，filename引数によって1つのファイルが参照されるようにしなければならない．（**ユーザへのアドバイス終わり**）

最初，すべてのプロセスはファイルを線形のバイトストリームとして参照し，各プロセスはデータをそれ自体のネイティブな表現で（データ表現の変換は行われない）参照する．（POSIXファイルはネイティブな表現の線形のバイトストリーム．）ファイルのビューはMPI_FILE_SET_VIEWルーチンによって変更することができる．

以下のアクセスモードがサポートされている（amodeに以下の整数定数のビットベクトルのORを設定することで指定される）．

- MPI_MODE_RDONLY — 読み取り専用，
- MPI_MODE_RDWR — 読み取りおよび書き込み，
- MPI_MODE_WRONLY — 書き込み専用，
- MPI_MODE_CREATE — ファイルが存在しない場合は生成する，
- MPI_MODE_EXCL — すでに存在するファイルを生成しようとした場合にエラー，
- MPI_MODE_DELETE_ON_CLOSE — 閉じるときにファイルを削除，
- MPI_MODE_UNIQUE_OPEN — ファイルは別の場所で同時に開けない，
- MPI_MODE_SEQUENTIAL — ファイルは逐次にのみアクセス可能，
- MPI_MODE_APPEND — すべてのファイルポインタの初期位置をファイル終端に設定．

ユーザへのアドバイス C言語/C++言語のユーザはビットベクトルOR(|)を使用してこれらの定数を組み合わせることができる． Fortran 90言語のユーザはビットベクトルのIORイントリンシックを使用することができる． Fortran 77言語のユーザは，システムでサポートされていればビットベクトルIORを（可搬ではないが）使用することができる．または，Fortran言語のユーザは定数のOR演算のために整数加算を使用でき，これは可搬である． （それぞれの定数はたかだか1回しか加算に現れてはならない）． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス これらの定数の値は，ビット単位のORと，これらの定数を要素とする重複要素を持たない集合の要素の合計値が等しくなるように定義する必要がある． （**実装者へのアドバイス終わり**）

モードMPI_MODE_RDONLY，MPI_MODE_RDWR，MPI_MODE_WRONLY，MPI_MODE_CREATE，およびMPI_MODE_EXCLの意味はPOSIXの対応するモードと同じである [29]．MPI_MODE_RDONLY，MPI_MODE_RDWR，またはMPI_MODE_WRONLYのいずれか1つを指定する必要がある．MPI_MODE_CREATEまたはMPI_MODE_EXCLをMPI_MODE_RDONLYと一緒に指定するのは誤りである．MPI_MODE_SEQUENTIALをMPI_MODE_RDWRと一緒に指定するのは誤りである．

MPI_MODE_DELETE_ON_CLOSEモードを使用すると，ファイルを閉じたときにファイルが削除される（MPI_FILE_DELETEを実行するのと同じ）．

MPI_MODE_UNIQUE_OPENモードを使用すると，ファイルのロックのオーバーヘッドが除去され，実装でのアクセスを最適化できる．ファイルが別の場所で同時に開かれている場合，このモードでファイルを開くのは誤りである．

> **ユーザへのアドバイス** MPI_MODE_UNIQUE_OPENの場合，**別の場所で開かれていない**という表現にはMPI環境の内部と外部の両方が含まれる．特に，外部のイベント（自動バックアップ機能など）によりファイルが開かれる可能性があることを意識する必要がある．MPI_MODE_UNIQUE_OPENを指定する場合，ユーザの責任により，このような外部のイベントが発生しないようにする必要がある．（**ユーザへのアドバイス終わり**）

MPI_MODE_SEQUENTIALモードを使用すると，実装での一部の逐次デバイス（テープおよびネットワークストリーム）へのアクセスを最適化することができる．このモードで開かれているファイルに対して，逐次でないアクセスを試みるのは誤りである．

MPI_MODE_APPENDを指定した場合，保証されるのは，すべての共有および個々のファイルポインタがMPI_FILE_OPENが戻るときに最初のファイル終端に位置づけられることだけである．それ以降のファイルポインタの位置付けはアプリケーションによって決まる．特に，すべての書き込みが追記されることを実装が保証するわけではない．

アクセスモードに関連するエラーはクラスMPI_ERR_AMODEで報告される．

info引数を使用して，ファイルのアクセスパターンとファイルシステムに特有の情報を渡すことができる（415ページの第13.2.8節を参照）．定数MPI_INFO_NULLはinfoを指定する必要がない場合に使用できる．

> **ユーザへのアドバイス** 一部のファイル属性は本質的に実装内容によって決まる（ファイルのパーミッションなど）．これらの属性を設定するには，info引数，またはMPIの有効範囲外の機能を使用する必要がある．（**ユーザへのアドバイス終わり**）

ファイルはデフォルトではノンアトミックモードのファイル一貫性の意味論に基づいて開かれる（455ページの第13.6.1節を参照）．より厳格なアトミックモードの整合性の意味論は，コンフリクトのあるアクセスのアトミック性の保証のために必要となるが，これを設定するにはMPI_FILE_SET_ATOMICITYを使用する必要がある．

13.2.2 ファイルを閉じる

MPI_FILE_CLOSE(fh)

INOUT	fh		ファイルハンドル (ハンドル)

```
int MPI_File_close(MPI_File *fh)
MPI_FILE_CLOSE(FH, IERROR)
    INTEGER FH, IERROR
{void MPI::File::Close() （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_CLOSEは最初にファイル状態の同期を取り（MPI_FILE_SYNCを実行するのと同じ），次にfhのファイルを閉じる．アクセスモードMPI_MODE_DELETE_ON_CLOSEを使用してファイルが開かれていた場合，ファイルが削除される（MPI_FILE_DELETEを実行するのと同じ)．MPI_FILE_CLOSEは集団的ルーチンである．

> **ユーザへのアドバイス** ファイルを閉じるときに削除する場合で，そのファイルに現在アクセスしている他のプロセスがある場合，ファイルのステータスとこれらのプロセスによるその後のアクセスの動作は，実装依存である． （**ユーザへのアドバイス終わり**）

ユーザの責任において，すべての未完了のノンブロッキング要求と，プロセスによって実行されたfhに関連付けれらた分割された集団操作が，そのプロセスによるMPI_FILE_CLOSEの呼び出しの前に完了していることを保証する必要がある．

MPI_FILE_CLOSEroutineはファイルハンドルオブジェクトを解放し，fhにMPI_FILE_NULLを設定する．

13.2.3 ファイルの削除

MPI_FILE_DELETE(filename, info)

IN	filename	削除するファイルの名前（文字列）
IN	info	infoオブジェクト（ハンドル）

```
int MPI_File_delete(char *filename, MPI_Info info)
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR
{static void MPI::File::Delete(const char* filename, const MPI::Info& info)
            （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_DELETEはファイル名filenameによって識別されるファイルを削除する．ファイルが存在しない場合，MPI_FILE_DELETEはクラスMPI_ERR_NO_SUCH_FILEでエラーを報告する．

info引数を使用して，ファイルシステムに特有の情報を渡すことができる（415ページの第13.2.8節13.2.8節を参照)．定数MPI_INFO_NULLはnullのinfoを指し，infoを指定する必要がない場合に使用できる．

プロセスで現在ファイルが開かれている場合，そのファイルへのアクセスの動作（および未完了のアクセスの動作）は実装依存である．また，開かれているファイルを削除するかどうかも実装依存である．ファイルが削除されない場合，クラスMPI_ERR_FILE_IN_USEまたはMPI_ERR_ACCESSでエラーが報告される．エラーの処理は，デフォルトのエラーハンドラを使用して行われる（465ページの第13.7節を参照）．

13.2.4 ファイルのサイズ変更

MPI_FILE_SET_SIZE(fh, size)

INOUT	fh	ファイルハンドル（ハンドル）
IN	size	ファイルの切り詰めまたは拡張サイズ（整数型）

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
{void MPI::File::Set_size(MPI::Offset size)  （廃止された呼び出し形式, 第15.2節を参照）
             }
```

MPI_FILE_SET_SIZEはファイルハンドルfhのファイルのサイズを変更する．sizeはファイルの先頭からバイト単位で測定される．MPI_FILE_SET_SIZEは集団的で，グループ内のすべてのプロセスはsizeに同じ値を渡す必要がある．

sizeが現在のファイルのサイズより小さい場合，sizeによって定義された位置でファイルが切り詰められる．実装では，この位置を超えた場所にあるファイルブロックを自由に開放することができる．

sizeが現在のファイルのサイズより大きい場合，ファイルのサイズはsizeとなる．前に書き込まれたファイルの領域は影響を受けない．ファイルの新しい領域（古いファイルサイズとsizeの間の変位の領域）内のデータの値は未定義となる．MPI_FILE_SET_SIZEルーチンでファイル領域を割り当てるかどうかは実装依存である．ファイル領域を強制的に確保するには，MPI_FILE_PREALLOCATEを使用する．

MPI_FILE_SET_SIZEは個々のファイルポインタまたは共有ファイルポインタに影響を及ぼさない．ファイルを開いたときにMPI_MODE_SEQUENTIALモードが指定されている場合，このルーチンを呼び出すのは誤りである．

> **ユーザへのアドバイス** MPI_FILE_SET_SIZE操作によってファイルの切り詰めが行われた後で，ファイルポインタがファイル終端を超えた先を指すことができる．これは正しい動作で，現在のファイル終端を超えた先をシークするのと同じである．（**ユーザへのアドバイス終わり**）

fhに対するノンブロッキング要求とスプリット集団操作はすべて，MPI_FILE_SET_SIZEを呼び出す前に完了していなければならない．そうでない場合，MPI_FILE_SET_SIZEの呼び出しは誤りである． 一貫性の意味論の視点で見ると，MPI_FILE_SET_SIZEは書込み操作であり，古いファイルサイズと新しいファイルサ

イズの中間の変位にアクセスする操作との間でコンフリクトが生じる（455ページの第13.6.1節を参照）.

13.2.5 ファイルの領域の事前アロケート

MPI_FILE_PREALLOCATE(fh, size)

INOUT	fh	ファイルハンドル（ハンドル）
IN	size	ファイルの事前アロケートサイズ（整数型）

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
{void MPI::File::Preallocate(MPI::Offset size) （廃止された呼び出し形式, 第15.2節を
              参照）}
```

MPI_FILE_PREALLOCATEは，fhのファイルの最初のsizeバイトについてストレージスペースの割り当てを保証する．MPI_FILE_PREALLOCATEは集団的で，グループ内のすべてのプロセスはsizeに同じ値を渡す必要がある．前に書き込まれたファイルの領域は影響を受けない．新しく割り当てられたファイルの領域については，MPI_FILE_PREALLOCATEの結果は未定義のデータを書き込むのと同じになる．sizeが現在のファイルのサイズより大きい場合，ファイルのサイズはsizeに増える．sizeが現在のファイルのサイズ以下の場合，ファイルのサイズは変更されない．

ファイルポインタ，保留中のノンブロッキングアクセス，ファイルの一貫性の扱いについては，MPI_FILE_SET_SIZEの場合と同様である．ファイルを開いたときにMPI_MODE_SEQUENTIALモードが指定されている場合，このルーチンを呼び出すのは誤りである．

ユーザへのアドバイス 実装によっては，ファイルの事前割当はコストが大きいことがある．（**ユーザへのアドバイス終わり**）

13.2.6 ファイルのサイズの問い合わせ

MPI_FILE_GET_SIZE(fh, size)

IN	fh	ファイルハンドル（ハンドル）
OUT	size	ファイルのサイズ（バイト単位）（整数型）

```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
{MPI::Offset MPI::File::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_SIZEはファイルハンドルfhのファイルの現在のバイト単位のサイズをsizeに返す．一貫性の意味論に関する限り，MPI_FILE_GET_SIZEはデータアクセス操作である（437ページの13.6.1節を参照）．

13.2.7 ファイルパラメータの問い合わせ

MPI_FILE_GET_GROUP(fh, group)

IN	fh	ファイルハンドル（ハンドル）
OUT	group	ファイルを開いたグループ（ハンドル）

```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
    INTEGER FH, GROUP, IERROR
{MPI::Group MPI::File::Get_group() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_GROUPはfhのファイルを開くのに使用したコミュニケータのグループの複製を返す．グループはgroupに返される．groupを解放するのはユーザの責任である．

MPI_FILE_GET_AMODE(fh, amode)

IN	fh	ファイルハンドル（ハンドル）
OUT	amode	ファイルを開くのに使用したファイルアクセスモード（整数型）

```
int MPI_File_get_amode(MPI_File fh, int *amode)
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR
{int MPI::File::Get_amode() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_AMODEはfhのファイルのアクセスモードをamodeに返す．

例 13.1 Fortran 77言語では，amodeビットベクトルをデコードするには以下のようなルーチンが必要となる．

```
      SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
!
!   TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
!   IF SET, RETURN 1 IN BIT_FOUND, 0 OTHERWISE
!
      INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
      BIT_FOUND = 0
      CP_AMODE = AMODE
 100  CONTINUE
      LBIT = 0
      HIFOUND = 0
      DO 20 L = MAX_BIT, 0, -1
         MATCHER = 2**L
         IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
            HIFOUND = 1
```

```
            LBIT = MATCHER
            CP_AMODE = CP_AMODE - MATCHER
         END IF
 20   CONTINUE
      IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
      IF (BIT_FOUND .EQ. 0 .AND. HIFOUND .EQ. 1 .AND. &
          CP_AMODE .GT. 0) GO TO 100
      END
```

amodeを1ビットずつデコードするため，このルーチンを連続して呼び出すことができる．例えば，以下の部分コードではMPI_MODE_RDONLYのチェックが行われる．

```
      CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
      IF (BIT_FOUND .EQ. 1) THEN
         PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
      ELSE
         PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
      END IF
```

13.2.8 ファイルのinfo

infoを通して指定されるヒント（311ページの第9節を参照）を使用して，ユーザはファイルのアクセスパターンの情報やファイルシステムに特有の情報などを与えて最適化を指示することができる．ヒントを提示することで，実装で入出力の性能を向上させたり，システムリソースの使用を最小限に抑えたりすることができる．しかし，ヒントが入出力インターフェースの意味論を変えることはない．つまり，実装ではすべてのヒントを無視することも自由である．ヒントは不可視のinfoオブジェクトを通して，MPI_FILE_OPEN，MPI_FILE_DELETE，MPI_FILE_SET_VIEW，およびMPI_FILE_SET_INFOでファイルごとに指定される．有効なヒントのサブセットを指定したinfoオブジェクトをMPI_FILE_SET_VIEWまたはMPI_FILE_SET_INFOに渡す場合，このinfoで指定しない設定済みのヒントやデフォルトのヒントには影響しない．

> **実装者へのアドバイス** あるシステム用にヒントをつけてプログラムを作成し，これらのヒントをサポートしていない別のシステムで後から実行する場合もある．一般的に，サポートされていないヒントは単に無視されなければならない．言うまでもなく，ヒントは必須ではありえない．しかし，特定の実装で使用されるヒントについては，ユーザがこのヒントの値を指定しない場合にデフォルト値を渡す必要がある．（**実装者へのアドバイス終わり**）

MPI_FILE_SET_INFO(fh, info)

INOUT	fh	ファイルハンドル（ハンドル）
IN	info	infoオブジェクト（ハンドル）

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
MPI_FILE_SET_INFO(FH, INFO, IERROR)
```

```
    INTEGER FH, INFO, IERROR
{void MPI::File::Set_info(const MPI::Info& info) （廃止された呼び出し形式, 第15.2節
              を参照）}
```

MPI_FILE_SET_INFOはfhのファイルのヒントのための新しい値を設定する．MPI_FILE_SET_INFOは集団的ルーチンである．infoオブジェクトはプロセスごとに違っていても構わないが，実装によりすべてのプロセスで同じであることが求められるinfoエントリは各プロセスのinfoオブジェクトで同じ値にならなければならない．

ユーザへのアドバイス ファイルを生成するとき，または開くときに実装で使用できる多数のinfo項目は，ファイルが生成されたり，開かれたりした後では容易に変更できない．そのため，実装は，オープン呼び出しであれば受け付けたであろうヒントをこの呼び出しでは無視することができる． （**ユーザへのアドバイス終わり**）

MPI_FILE_GET_INFO(fh, info_used)

IN	fh	ファイルハンドル（ハンドル）
OUT	info_used	新しいinfoオブジェクト（ハンドル）

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR
{MPI::Info MPI::File::Get_info() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_INFOはfhのファイルのヒントが格納された新しいinfoオブジェクトを返す．この開かれたファイルに関連する，システムによって実際に使用されるすべてのヒントの現在の設定は，info_usedに返される．このようなヒントがない場合，新しく生成され，キー値のペアが格納されていないinfoオブジェクトへのハンドルが返される．MPI_INFO_FREEによりinfo_usedを解放するのはユーザの責任である．

ユーザへのアドバイス info_usedに値が返されるinfoオブジェクトには，このファイルのための現在アクティブなすべてのヒントが格納される．システムではユーザによって設定されたヒントが識別できず，ユーザが設定した以外のヒントを識別する可能性があるため，このヒントの集合はMPI_FILE_OPEN，MPI_FILE_SET_VIEW，MPI_FILE_SET_INFOに渡されたヒントの集合よりも大きくなったり小さくなったりする場合がある． （**ユーザへのアドバイス終わり**）

予約されたファイルのヒント

有益と思われるいくつかのヒント（infoキー値）を以下に紹介する．以下に示すキー値が予約されている．実装ではこれらのキー値を解釈する必要はないが，キー値を解釈する場合，記述された機能を提供する必要がある（「info」の詳細は，311ページの第9節を参照）．

これらのヒントは主に並列入出力デバイスでのアクセスパターンとデータの配置に影響する．紹介する各ヒントの名前について，そのヒントの目的とヒント値の型を説明する．“[**SAME**]”の注釈には参加するすべてのプロセスによって渡されるヒント値が同一でなければならないことを指定する．そうしない場合プログラムは誤りである．また，一部のヒントは実行のコンテクストに応じて決まり，特定のタイミングでのみ実装によって使用される（例えば，file_permはファイルの生成中のみ有効）．

access_style (**カンマ区切りの文字列のリスト**)： このヒントは，ファイルが閉じられるまで，またはaccess_styleキー値が変更されるまでのファイルへのアクセス方法を指定する．ヒント値はカンマ区切りのリストで，以下である． read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random.

collective_buffering (**論理型**) [**SAME**]： このヒントはアプリケーションが集団バッファリングを利用するかどうかを指定する．集団バッファリングとは集団的アクセスで行われる最適化のことである．グループ内のすべてのプロセスがファイルへアクセスする代わりに一定数のターゲットノードによって行われる．これらのターゲットノードにより複数の小さな要求が大きなディスクアクセスに統合される．このキーの値としてはtrueとfalseが使用できる．追加のヒントにより，集団バッファリングパラメータcb_block_size，cb_buffer_size，cb_nodesがさらに指示される．

cb_block_size (**整数型**) [**SAME**]： このヒントは，集団バッファリングのファイルアクセスに使用するブロックサイズを指定する．**ターゲットノード**はこのサイズのチャンクでデータにアクセスする．チャンクはラウンドロビン（CYCLIC）パターンでターゲットノード間で分配される．

cb_buffer_size (**整数型**) [**SAME**]： このヒントは各ターゲットノードで集団バッファリングのために使用できる総バッファ領域で，通常はcb_block_sizeの倍数で指定する．

cb_nodes (**整数型**) [**SAME**]： このヒントは集団バッファリングに使用されるターゲットノードの数を指定する．

chunked (**カンマ区切りの整数のリスト**) [**SAME**]： このヒントは，多次元配列としてファイルが構成されることを指定する．この配列は部分配列単位でアクセスされることが多い．このヒントの値はカンマ区切りの配列の次元のリストで，最も重要なものが最初に指定される（C言語の場合のように行優先の順序で格納された配列の場合，最も重要な次元が最初の次元のであり，Fortran言語の場合のように列優先の順序で格納された配列の場合は，最も重要な次元が最後の次元であり，配列の次元を逆にする必要がある）．

chunked_item (**カンマ区切りの整数のリスト**) [**SAME**]： このヒントは配列のエントリのサイズをバイト単位で指定する．

chunked_size (カンマ区切りの整数のリスト) [**SAME**]： このヒントは部分配列の次元を指定する．これはカンマ区切りの配列の次元のリストで，最も重要なものが最初に指定される．

filename (文字列)： このヒントは，ファイルが開かれた時に使用されたファイル名を指定する．開かれたファイルのファイル名を実装で返すことができる場合，このキーを使用してMPI_FILE_GET_INFOにより返される．このキーは，MPI_FILE_OPEN，MPI_FILE_SET_VIEW，MPI_FILE_SET_INFO，MPI_FILE_DELETEに渡された場合は無視される．

file_perm (文字列) [**SAME**]： このヒントはファイルの生成用に使用するファイルのパーミッションを指定する．このヒントの設定は，MPI_MODE_CREATEが格納されたamodeをMPI_FILE_OPENに渡した場合のみ有用である．このキーに対する正しい値の集合は実装依存である．

io_node_list (カンマ区切りの文字列のリスト) [**SAME**]： このヒントは，ファイルを格納するために使用する必要のある入出力デバイスのリストを指定する．このヒントに最も関連するのはファイルの生成時である．

nb_proc (整数型) [**SAME**]： このヒントは，このファイルにアクセスするプログラムを実行するために通常割り当てられる並列プロセスの数を指定する．このヒントに最も関連するのはファイルの生成時である．

num_io_nodes (整数型) [**SAME**]： このヒントはシステム内の入出力デバイスの数を指定する．このヒントに最も関連するのはファイルの生成時である．

striping_factor (整数型) [**SAME**]： このヒントはファイルをストライピングすべき入出力デバイス数を指定する．このヒントが有効なのはファイルの生成時である．

striping_unit (整数型) [**SAME**]： このヒントはこのファイル用に使用される，提案のストライピング単位を指定する．ストライピング単位とは，多数のデバイスでのストライピングを行う場合に，次のデバイスに進む前に1台の入出力デバイスに割り当てる連続データの量である．これはバイト単位で表現される．このヒントに関連するのはファイルの生成時のみである．

13.3 ファイルのビュー

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

INOUT	fh	ファイルハンドル（ハンドル）
IN	disp	変位（整数型）
IN	etype	要素データ型（ハンドル）
IN	filetype	ファイル型（ハンドル）
IN	datarep	データ表現（文字列）
IN	info	infoオブジェクト（ハンドル）

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
MPI_Datatype filetype, char *datarep, MPI_Info info)
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
{void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
             const MPI::Datatype& filetype, const char* datarep,
             const MPI::Info& info) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_SET_VIEWルーチンはファイル内のデータのプロセスビューを変更する．ビューの起点はdispに設定され，データの型はetypeに設定され，プロセスへのデータの分配はfiletypeに設定され，ファイル内でのデータの表現はdatarepに設定される．また，MPI_FILE_SET_VIEWは個々のファイルポインタと共有ファイルポインタを0にリセットする．MPI_FILE_SET_VIEWは集団的で，ファイルデータ表現内でのdatarepの値とetypeの範囲はグループ内のすべてのプロセスで同じでなければならず，disp，filetype，infoの値は異なっていてもよい．etypeとfiletypeに渡すデータ型はコミットされている必要がある．

etypeは常にファイル内のデータ配置を指定する．etypeが可搬なデータ型である場合（13ページの第2.4節を参照），etypeの範囲はファイルのデータ表現に合わせてデータ型の変位をスケールさせることにより計算される．etypeが可搬なデータ型でない場合，etypeの範囲の計算時にスケーリングは行われない．異機種環境で可搬でないetypeを使用する場合は，注意が必要である．詳細は447ページの第13.5.1節を参照すること．

ファイルが開かれたときにMPI_MODE_SEQUENTIALモードが指定されている場合，dispに特別な変位MPI_DISPLACEMENT_CURRENTを渡す必要がある．これにより変位が共有ファイルポインタの現在の位置に設定される．ファイルのamodeがMPI_MODE_SEQUENTIALに設定されている場合を除いて，MPI_DISPLACEMENT_CURRENTは無効である．

根拠 磁気テープやストリーミングネットワーク接続に対応するものなどの逐次ファイルの場合，**変位**が意味を持たないこともある．MPI_DISPLACEMENT_CURRENTを使用すると，そのようなタイプのファイルに対してビューを変更することができる． （**根拠の終わり**）

実装者へのアドバイス 多くの場合に，MPI_FILE_OPENの直後にMPI_FILE_SET_VIEWの呼び出しが行われると考えられる．高品質な実装はこのような動作の効率をよくするであろう．（**実装者へのアドバイス終わり**）

disp変位引数は，ビューが開始される位置（ファイルの先頭からのバイト単位の絶対オフセット）を指定する．

ユーザへのアドバイス dispは，ヘッダをスキップするために，または異なるパターンでアクセスされるデータセグメントのシーケンスがファイルに含まれる場合に使用される（図13.3を参照）．それぞれ異なる変位とファイル型を使用する個々のビューを，各セグメントにアクセスするのに使用できる．

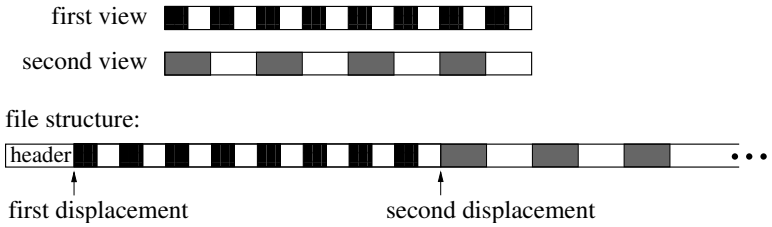

図 13.3: 変位

（**ユーザへのアドバイス終わり**）

etype（**要素**（elementary）データ型）はデータアクセスと位置付けの単位である．これはMPIの任意の定義済みまたは派生データ型とすることができる．生成されるすべての型マップ変位が非負で単調非減少である限り，任意のMPIデータ型コンストラクタルーチンを使用して派生etypeを構成することができる．データアクセスはetype単位で行われ，etype型のデータ要素の全てが読み書きされる．オフセットはetypeのカウントとして表現され，ファイルポインタはetypeの先頭を指す．

ユーザへのアドバイス 異機種環境での相互運用性を保証するため，etypeの作成時には追加の制限に従わなければならない（444ページの第13.5節を参照）．（**ユーザへのアドバイス終わり**）

ファイル型は1つのetype，または同じetypeの複数のインスタンスから構成される派生MPIデータ型となる．また，ファイル型内の穴の範囲はetypeの範囲の倍数でなければならない．これらの変位は個々に異なっている必要はないが，非負でかつ，単調非減少でなければならない．

このファイルが書込み用に開かれている場合，etypeにもfiletypeにも重複する領域を格納することはできない．この制限は通信での「受信で使用するデータ型には重複する領域を指定できない」という制限と同等である．しかし，異なるプロセスのfiletypeは相互に重複することがある．

filetypeに穴がある場合，穴の中のデータには呼び出しプロセスからアクセスできない．

しかし，その後，disp，etype，filetype引数を変更し，MPI_FILE_SET_VIEWを呼び出すことにより，ファイルの別の部分にアクセスすることができる．

etypeおよびfiletypeの作成時に絶対アドレスを使用するのは誤りである．

ファイルのアクセスパターンの情報やファイルシステム特有の情報を与えて最適化を指示するためにinfo引数を使用することができる（415ページの第13.2.8節を参照）．定数MPI_INFO_NULLはのinfoを指し，infoを指定する必要がない場合に使用できる．

datarep引数はファイル内のデータ表現を指定する文字列である．詳しい説明と有効な値の論考については，ファイルの相互運用性の節（444ページの第13.5節）を参照すること．

ユーザは責任を持って，fhに対するすべてのノンブロッキング要求とスプリット集団操作をMPI_FILE_SET_VIEWの呼び出しの前に完了させる必要がある．そうしないと，MPI_FILE_SET_VIEWの呼び出しは誤りとなる．

MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

IN	fh	ファイルハンドル（ハンドル）
OUT	disp	変位（整数型）
OUT	etype	要素データ型（ハンドル）
OUT	filetype	ファイル型（ハンドル）
OUT	datarep	データ表現（文字列）

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
MPI_Datatype *filetype, char *datarep)
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
{void MPI::File::Get_view(MPI::Offset& disp, MPI::Datatype& etype,
             MPI::Datatype& filetype, char* datarep) const （廃止された呼び出し
             形式, 第15.2節を参照）}
```

MPI_FILE_GET_VIEWはファイル内のデータのプロセスビューを返す．変位の現在の値はdispに返される．etypeおよびfiletypeは，それぞれ現在のetypeおよびファイル型の型マップに等しい型マップを持つ新しいデータ型である．

データ表現はdatarepに返される．datarepが，返されるデータ表現文字列を格納するのに十分な大きさを持つようにするのはユーザの責任である．データ表現文字列の長さはMPI_MAX_DATAREP_STRINGの値に制限される．

また，現在のビューを設定するために可搬なデータ型が使用された場合，MPI_FILE_GET_VIEWによって返されるこれに対応するデータ型も可搬なデータ型となる．etypeまたはfiletypeが派生データ型である場合，これらを解放するのはユーザの責任である．返されるetypeとfiletypeは，両方ともコミットされた状態である．

13.4 データアクセス

13.4.1 データアクセスルーチン

データは，読み込みおよび書き込み呼び出しを発行することにより，ファイルとプロセスの間で移動される．データアクセスには，位置付け（明示的なオフセットと暗黙的なファイルポインタ），同期（ブロッキングとノンブロッキングおよびスプリット集団操作），連携（非集団操作と集団操作）という3つの直交する側面がある．2種類のファイルポインタ（個別と共有）を含むこれらのデータアクセスルーチンの以下の組み合わせを表13.1に示す．

位置付け	同期	連携	
		非集団操作	集団操作
明示的オフセット	ブロッキング	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	ノンブロッキング & スプリット集団操作	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
個別ファイルポインタ	ブロッキング	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	ノンブロッキング & スプリット集団操作	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
共有ファイルポインタ	ブロッキング	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	ノンブロッキング & スプリット集団操作	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

表 13.1: データアクセスルーチン

POSIX read()/fread()およびwrite()/fwrite()はブロッキングの非集団操作で，個別ファイルポインタを使用する．これに相当するMPIはMPI_FILE_READとMPI_FILE_WRITEである．

データアクセスルーチンの実装は，性能向上のため，データをバッファリングすることがある．読み込み操作の完了後はデータは常にユーザのバッファ内で利用できるため，これは読み込みに影響しない．しかし，書き込みについては，MPI_FILE_SYNCルーチンのみがデータがストレージデバイスに転送されていることを保証する．

位置付け

MPIでは，データアクセスルーチンのための位置付けの方法として，明示的なオフセット，個別ファイルポインタ，共有ファイルポインタの3種類が用意されている．異なる

位置付け方法を，相互に影響を及ぼさないように同じプログラム内で組み合わせることができる．

明示的なオフセットを受け付けるデータアクセスルーチンはその名前に_ATを含む（MPI_FILE_WRITE_ATなど）．明示的なオフセットを使用した操作は引数として直接渡されたファイル位置でデータアクセスを実行する．ファイルポインタは使用されず，更新もされない．これは，“seek”が発行されないため，アトミックなseek-and-readまたはseek-and-write操作とは異なることに注意すること．明示的なオフセットを使用した操作については，425ページの第13.4.2節で説明する．

個別ファイルポインタルーチンの名前は位置の修飾子を含まない（MPI_FILE_WRITEなど）．個別ファイルポインタを使用した操作については，428ページの第13.4.3節で説明する．共有ファイルポインタを使用するデータアクセスルーチンはその名前に_SHAREDまたは_ORDEREDを含む（MPI_FILE_WRITE_SHAREDなど）．共有ファイルポインタを使用した操作については，433ページの第13.4.4節で説明する．

MPIで管理されるファイルポインタに関する主な意味論的な問題は，これらが入出力操作によって更新される方法とタイミングである．一般的に，各入出力操作では，ファイル操作によってアクセスされた最後のデータ項目の次のデータ項目にファイルポインタが置かれる．ノンブロッキングまたはスプリット集団操作では，入出力を開始した呼び出しによって，アクセスが完了する前にポインタが更新される可能性がある．

より正式には，

$$new\_file\_offset = old\_file\_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

ここで，$count$はアクセスされる$datatype$の項目数，$elements(X)$はXの型マップ内の定義済みデータ型の数，old_file_offsetは呼び出し前の暗黙的なオフセット値である．ファイルの位置new_file_offsetは，現在のビューでの相対位置としての，etypeの個数として測られる．

同期

MPIはブロッキングおよびノンブロッキング入出力ルーチンをサポートしている．

ブロッキング入出力呼び出しは，入出力要求が完了するまで戻らない．

ノンブロッキング入出力呼び出しは，入出力操作を開始するが，完了の待ち合わせを行わない．適切なハードウェアがあれば，ユーザのバッファを用いたデータの入出力を，計算の処理を同時に進めることができる．ノンブロッキング入出力呼び出しの**要求完了**呼び出し（MPI_WAIT，MPI_TEST，またはその変型）は入出力要求を完了させるために必要になる．これにより，データの読み取りまたは書き込みが完了し，ユーザがバッファを安全に再使用できるようになったことを確認できる．ノンブロッキング型のルーチンはMPI_FILE_IXXXという名前で，Iは「即時（immediate）」の意味である．

操作の開始と完了の間に，ノンブロッキングデータアクセス操作のローカルバッファにアクセスしたり，そのバッファを他の通信の送信元または送信先として使用したりするのは誤りである．

スプリット集団ルーチンは，集団データアクセスの「ノンブロッキング」操作を制限付きでサポートしている（439ページの第13.4.5節を参照）．

協調

非集団的データアクセスルーチンMPI_FILE_XXXには，それぞれと対になる集団的ルーチンがある．大部分のルーチンで，対になるルーチンはMPI_FILE_XXX_ALL，またはMPI_FILE_XXX_BEGINとMPI_FILE_XXX_ENDのペアとなる．MPI_FILE_XXX_SHAREDルーチンと対になるルーチンはMPI_FILE_XXX_ORDEREDである．

非集団呼び出しの完了は，呼び出しプロセスの動作のみに依存する．しかし，集団呼び出し（プロセスグループのすべてのメンバによって呼び出されなければならない）の完了は集団呼び出しに参加する他のプロセスの動作に依存することがある．集団呼び出しの意味論に関する規則は，459ページの第13.6.4節を参照すること．

グローバルデータアクセスは自動最適化の大きな可能性を秘めているため，集団操作では非集団操作よりもはるかに高い性能が見込めるはずである．

データアクセスのルール

データは，読み込みおよび書き込みルーチンを呼び出すことにより，ファイルとプロセスの間で移動される．読み込みルーチンはデータをファイルからメモリに移動する．書き込みルーチンはデータをメモリからファイルに移動する．ファイルはファイルハンドルfhにより指定される．ファイルデータの領域は現在のビューへのオフセットによって指定される．メモリ内のデータはbuf，count，datatypeの3つによって指定される．完了時，呼び出しプロセスによってアクセスされたデータの量がstatusに返される．

オフセットはアクセスのためのファイル内の開始位置を指定する．オフセットは常に，現在のビューでの相対位置として，etypeを単位として測られる．明示的なオフセットを使用したルーチンは引数としてoffsetを渡す（負の値は誤りである）．ファイルポインタルーチンはMPIによって管理される暗黙的なオフセットを使用する．

データアクセスルーチンはユーザのバッファbufとファイルの間でdatatype型のcount個のデータ項目の転送（読み込みまたは書き込み）を試みる．ルーチンに渡されるdatatypeはコミットされたデータ型でなければならない．buf，count，datatypeに対応するメモリ内のデータの配置はMPI通信関数と同様に解釈される．32ページの第3.2.2節と112ページの第4.1.11節を参照すること．データはファイルの，現在のビューによって指定された部分からアクセスされる．（419ページの第13.3節）．datatypeの型シグネチャは現在のビューのetypeのいくつかの連続するコピーの型シグネチャと一致していなければならない．受信の場合と同様，重複する領域（複数回格納されるメモリの領域）を含むdatatypeを読み取り用に指定するのは誤りである．

ノンブロッキングなデータアクセスルーチンはMPIがデータアクセスを開始できること，また要求ハンドルrequestを入出力操作に関連付けられることを示している．ノンブロッキング操作は，MPI_TEST，MPI_WAIT，またはその変型によって完了する．

データアクセス操作は，完了時に，アクセスしたデータの量をstatusに返す．

ユーザへのアドバイス Fortran言語のコンパイラによって実行される引数のコピーとレジスタ最適化に関する問題を回避するため，第16.2.2節の「データのコピーおよび連続領域配置による問題」（504ページ）と「レジスタ最適化での問題」（507ページ）のヒントに注意すること．（**ユーザへのアドバイス終わり**）

ブロッキングルーチンの場合，statusは直接返される．ノンブロッキングルーチンおよびスプリット集団ルーチンの場合，statusは操作の完了時に返される．呼び出しプロセスによってアクセスされる定義済みの要素とdatatypeエントリの数は，それぞれMPI_GET_COUNTとMPI_GET_ELEMENTSを使用してstatusから抽出することができる．MPI_ERRORフィールドの解釈は他の操作の場合と同様で，通常は未定義だが，MPIルーチンがMPI_ERR_IN_STATUSを返す場合は意味がある．この引数の戻り値が不要な場合，（C言語およびFortran言語では）ユーザはstatus引数にMPI_STATUS_IGNOREを渡すことができる．C++言語では，status引数はオプションである．操作が取り消されたかどうかを調べるために，statusをMPI_TEST_CANCELLEDに渡すことができる．statusのその他のすべてのフィールドは未定義である．

読み取り時に，プログラムは読み取ったデータの量が要求された量より少ないことに注意することにより，ファイル終端を検知することができる．ファイル終端を越えて書き込むとファイルのサイズが増大する．エラーが発生した（あるいは読み取りがファイル終端に達した）場合を除いて，アクセスしたデータの量は要求された量となる．

13.4.2 明示的なオフセットを使用したデータアクセス

ファイルを開いたときにMPI_MODE_SEQUENTIALモードが指定されている場合に，この節でで示すルーチンを呼び出すのは誤りである．

MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)

IN	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファ要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された
             呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_at(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_READ_ATはoffsetによって指定された位置からファイルを読み取る.

MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)

IN	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファ要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_READ_AT_ALLはブロッキングMPI_FILE_READ_ATインターフェイスの集団操作バージョンである.

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write_at(MPI::Offset offset, const void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_WRITE_ATはoffsetによって指定された位置からファイルに書き込む.

MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype, MPI::Status& status)
             （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write_at_all(MPI::Offset offset, const void* buf,
             int count, const MPI::Datatype& datatype) （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_FILE_WRITE_AT_ALLはブロッキングMPI_FILE_WRITE_ATインターフェイスの集団操作バージョンである.

MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)

IN	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	request	リクエストオブジェクト（ハンドル）

```
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_IREAD_ATはMPI_FILE_READ_ATインターフェイスのノンブロッキングバージョンである.

MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)

INOUT	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	request	リクエストオブジェクト（ハンドル）

```
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{MPI::Request MPI::File::Iwrite_at(MPI::Offset offset, const void* buf,
              int count, const MPI::Datatype& datatype) （廃止された呼び出し形式,
              第15.2節を参照）}
```

MPI_FILE_IWRITE_ATはMPI_FILE_WRITE_ATインターフェイスのノンブロッキングバージョンである.

13.4.3 個別ファイルポインタを使用したデータアクセス

MPIは各ファイルハンドルに対してプロセスごとに個別のファイルポインタを管理している．このポインタの現在の値は，この節で説明するデータアクセスルーチンのオフセットを暗黙的に指定する．これらのルーチンは，MPIによって管理される個別ファイルポインタのみを使用，更新する．共有ファイルポインタは使用，更新しない.

個別ファイルポインタルーチンは，425ページの第13.4.2節で説明した明示的なオフセットルーチンを使用したデータアクセスと同じ意味論を持つが，以下の点が異なる.

- offsetはMPIで管理される個別ファイルポインタの現在の値として定義される.

個別ファイルポインタ操作の開始後，個別ファイルポインタは，アクセスされる最後のetypeの後の次のetypeを指すよう更新される．ファイルポインタはファイルの現在のビューでの相対位置として更新される.

ファイルが開かれたときにMPI_MODE_SEQUENTIALモードが指定されている場合，この節のルーチンを呼び出すのは誤りであるが，MPI_FILE_GET_BYTE_OFFSETは例外である.

MPI_FILE_READ(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype,
             MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read(void* buf, int count, const MPI::Datatype& datatype)
             （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_READは個別ファイルポインタを使用してファイルを読み取る.

例 13.2 以下のFortran言語の部分コードはファイル終端に達するまでファイルを読み取る例である.

```
!   Read a preexisting input file until all data has been read.
!   Call routine "process_input" if all requested data is read.
!   The Fortran 90 "exit" statement exits the loop.

      integer   bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
      parameter (bufsize=100)
      real      localbuffer(bufsize)

      call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
      call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
                          MPI_INFO_NULL, ierr )
      totprocessed = 0
      do
         call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
                             status, ierr )
         call MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
         call process_input( localbuffer, numread )
         totprocessed = totprocessed + numread
         if ( numread < bufsize ) exit
      enddo

      write(6,1001) numread, bufsize, totprocessed
1001  format( "No more data:  read", I3, "and expected", I3, &
              "Processed total of", I6, "before terminating job." )

      call MPI_FILE_CLOSE( myfh, ierr )
```

MPI_FILE_READ_ALL(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_all(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Read_all(void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された
             呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_all(void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_READ_ALLはブロッキングMPI_FILE_READインターフェイスの集団操作バージョンである.

MPI_FILE_WRITE(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Write(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された
             呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write(const void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_WRITEは個別ファイルポインタを使用してファイルを書き込む.

MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

{void MPI::File::Write_all(const void* buf, int count,
const MPI::Datatype& datatype, MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void MPI::File::Write_all(const void* buf, int count,
const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

MPI_FILE_WRITE_ALLはブロッキングMPI_FILE_WRITEインターフェイスの集団操作バージョンである.

MPI_FILE_IREAD(fh, buf, count, datatype, request)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	request	リクエストオブジェクト（ハンドル）

```
int MPI_File_iread(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
{MPI::Request MPI::File::Iread(void* buf, int count,
              const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_IREADはMPI_FILE_READインターフェイスのノンブロッキングバージョンである.

例 **13.3** 以下のFortran言語の部分コードを用いてファイルポインタの更新の意味論を説明する.

```
!   Read the first twenty real words in a file into two local
!   buffers.  Note that when the first MPI_FILE_IREAD returns,
!   the file pointer has been updated to point to the
!   eleventh real word in the file.

      integer   bufsize, req1, req2
      integer, dimension(MPI_STATUS_SIZE) :: status1, status2
      parameter (bufsize=10)
      real      buf1(bufsize), buf2(bufsize)

      call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
      call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
                         MPI_INFO_NULL, ierr )
      call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
                           req1, ierr )
      call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
                           req2, ierr )

      call MPI_WAIT( req1, status1, ierr )
      call MPI_WAIT( req2, status2, ierr )
```

```
      call MPI_FILE_CLOSE( myfh, ierr )
```

MPI_FILE_IWRITE(fh, buf, count, datatype, request)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	request	リクエストオブジェクト（ハンドル）

```
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
{MPI::Request MPI::File::Iwrite(const void* buf, int count,
              const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_IWRITEはMPI_FILE_WRITEインターフェイスのノンブロッキングバージョンである．

MPI_FILE_SEEK(fh, offset, whence)

INOUT	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
IN	whence	更新モード（ステート型）

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Seek(MPI::Offset offset, int whence) （廃止された呼び出し形式,
              第15.2節を参照）}
```

MPI_FILE_SEEKはwhenceに従って個別ファイルポインタを更新する．whenceで使用できる値は以下のとおりである．

- MPI_SEEK_SET：ポインタがoffsetに設定される
- MPI_SEEK_CUR：ポインタが現在のポインタ位置+offsetに設定される
- MPI_SEEK_END：ポインタがファイル終端+offsetに設定される

offsetには負の値も設定でき，この場合は後方へのシークが行われる．ビューにおける負の位置にシークするのは誤りである．

MPI_FILE_GET_POSITION(fh, offset)

IN	fh	ファイルハンドル（ハンドル）
OUT	offset	個別ポインタのオフセット（整数型）

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{MPI::Offset MPI::File::Get_position() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_POSITIONは，現在のビューに対する個別ファイルポインタの現在の位置をetype単位でoffsetに返す．

ユーザへのアドバイス 返されたoffsetは，後に現在の位置に戻るために，whence = MPI_SEEK_SETとしてMPI_FILE_SEEKを呼び出す際に使用することができる．現在のファイルポインタの位置に変位を設定するには，まずMPI_FILE_GET_BYTE_OFFSETを使用してoffsetを絶対バイト位置に変換してから，得られた変位を使用してMPI_FILE_SET_VIEWを呼び出す． （**ユーザへのアドバイス終わり**）

MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)

IN	fh	ファイルハンドル（ハンドル）
IN	offset	オフセット（整数型）
OUT	disp	オフセットの絶対バイト位置（整数型）

```
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
MPI_Offset *disp)
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
{MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp) const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_BYTE_OFFSETはビューの相対オフセットを絶対バイト位置に変換する．fhの現在のビューに対するoffsetの（ファイル先頭からの）絶対バイト位置がdispに返される．

13.4.4 共有ファイルポインタを使用したデータアクセス

MPIは集団MPI_FILE_OPENごとに正確に1つの共有ファイルポインタを管理する（コミュニケータグループ内のプロセス間で共有される）．このポインタの現在の値は，この節で説明するデータアクセスルーチン内のオフセットを暗黙的に指定する．これらのルーチンはMPIによって管理される共有ファイルポインタのみを使用，更新する．個別ファイルポインタは使用，更新しない．

共有ファイルポインタルーチンは，425ページの第13.4.2節で説明した明示的なオフセットルーチンを使用したデータアクセスと同じ意味論を持つが，以下の点が異なる．

- offsetはMPIで管理される共有ファイルポインタの現在の値として定義される．
- 共有ファイルポインタルーチンを複数回呼び出した場合の結果は，呼び出しがシリアライズされたとした場合の結果と定義される．
- すべてのプロセスで同じファイルビューが使用されていない場合，共有ファイルポインタルーチンを使用するのは誤りである．

非集団共有ファイルポインタルーチンの場合，処理の順序は一意に定まらない．そのため，ユーザが，別の同期方法を用いて処理が特定の順序になるよう強制する必要がある．

共有ファイルポインタ操作の開始後，共有ファイルポインタは，アクセスされる最後のetypeの後の次のetypeを指すよう更新される．ファイルポインタはファイルの現在のビューに対する相対位置として更新される．

非集団操作

MPI_FILE_READ_SHARED(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Read_shared(void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された
             呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_shared(void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_READ_SHAREDは共有ファイルポインタを使用してファイルを読み取る．

MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Write_shared(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された
             呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write_shared(const void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_WRITE_SHAREDは共有ファイルポインタを使用してファイルを書き込む.

MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	request	リクエストオブジェクト（ハンドル）

```
int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
{MPI::Request MPI::File::Iread_shared(void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_IREAD_SHAREDはMPI_FILE_READ_SHAREDインターフェイスのノンブロッキングバージョンである.

MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	request	リクエストオブジェクト（ハンドル）

```
int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
{MPI::Request MPI::File::Iwrite_shared(const void* buf, int count,
              const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_IWRITE_SHAREDはMPI_FILE_WRITE_SHAREDインターフェイスのノンブロッキングバージョンである.

集団操作

共有ファイルポインタを使用した集団アクセスの意味論は，ファイルへのアクセスがグループ内のプロセスのランクで決定された順序で行われるということである．各プロセスについて，データのアクセスを行うファイルの領域は，グループ内のランクがこのプロセスのものより小さいすべてのプロセスがそれぞれのデータにアクセスした後に共有ファイルポインタが配置される位置となる．また，同じプロセスによるその後の共有オフセットアクセス[1]とこの集団アクセスが干渉しないようにするため，グループ内のすべてのプロセスがそれぞれのアクセスを開始した後でないと呼び出しが戻らないようになっている．呼び出しが戻ると，共有ファイルポインタはすべてのプロセスによって使用されたファイルビューに従って，最後に要求されたetypeの後の，アクセス可能な次のetypeを指す.

ユーザへのアドバイス グループ内のすべてのプロセスが共有ファイルポインタを使用してファイルにアクセスする必要があるプログラムがあっても，そのプログラムへのデータアクセスはプロセスのランクの順で行う**必要**がない場合もある． このようなプログラムでは，実装が，共有順序付けルーチン（例えばMPI_FILE_WRITE_SHAREDではなく，MPI_FILE_WRITE_ORDERED）に性能を向上するアクセスの最適化を施すことができる． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス すべてのプロセスで要求されるデータへのアクセスをシリアライズする必要はない．すべてのプロセスで要求が発行されると，すべてのアクセスの対象となるファイル内の領域が計算可能になり，相互に独立して，可能であれば並行してアクセスができるようになる． （**実装者へのアドバイス終わり**）

[1]訳註：原文には“shared offset accesses”とあるが，共有ファイルポインタによるアクセスの誤りである.

MPI_FILE_READ_ORDERED(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Read_ordered(void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された
             呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_ordered(void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_READ_ORDEREDはMPI_FILE_READ_SHAREDインターフェイスの集団操作バージョンである.

MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Write_ordered(const void* buf, int count,
             const MPI::Datatype& datatype, MPI::Status& status) （廃止された
             呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write_ordered(const void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参
             照）}
```

MPI_FILE_WRITE_ORDEREDはMPI_FILE_WRITE_SHAREDインターフェイスの集団操作バージョンである.

シーク

ファイルを開いたときにMPI_MODE_SEQUENTIALモードが指定されている場合，以下の2つのルーチン（MPI_FILE_SEEK_SHAREDおよびMPI_FILE_GET_POSITION_SHARED）を呼び出すのは誤りである.

MPI_FILE_SEEK_SHARED(fh, offset, whence)

INOUT	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
IN	whence	更新モード（ステート型）

```
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Seek_shared(MPI::Offset offset, int whence) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_SEEK_SHAREDはwhenceに従って共有ファイルポインタを更新する．whenceで使用できる値は以下のとおりである．

- MPI_SEEK_SET：ポインタがoffsetに設定される
- MPI_SEEK_CUR：ポインタが現在のポインタ位置+offsetに設定される
- MPI_SEEK_END：ポインタがファイル終端+offsetに設定される

MPI_FILE_SEEK_SHAREDは集団的であり，ファイルハンドルfhに関連付けれらたコミュニケータグループ内のすべてのプロセスはoffsetとwhenceに同じ値を使用してMPI_FILE_SEEK_SHAREDを呼び出す必要がある．

offsetには負の値も設定でき，この場合は後方へのシークが行われる．ビューにおける負の位置にシークするのは誤りである．

MPI_FILE_GET_POSITION_SHARED(fh, offset)

IN	fh	ファイルハンドル（ハンドル）
OUT	offset	共有ポインタのオフセット（整数型）

```
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{MPI::Offset MPI::File::Get_position_shared() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_POSITION_SHAREDは，現在のビューに対する共有ファイルポインタの現在の位置をetype単位でoffsetに返す．

ユーザへのアドバイス 返されたoffsetは，後に現在の位置に戻るために，whence = MPI_SEEK_SETとしてMPI_FILE_SEEK_SHAREDを呼び出す際に使用することができる．現在のファイルポインタの位置に変位を設定するには，まずMPI_FILE_GET_BYTE_OFFSETを使用してoffsetを絶対バイト位置に変換してから，得られた変位を使用してMPI_FILE_SET_VIEWを呼び出す． （**ユーザへのアドバイス終わり**）

13.4.5 スプリット集団データアクセスルーチン

MPIはスプリット集団データアクセスルーチンを使用した，すべてのデータアクセス用の「ノンブロッキング集団」入出力操作を制限付きで提供している．これらのルーチンは，1つの集団操作が開始ルーチンと終了ルーチンの2つに分割されているため，「分割」集団ルーチンと呼ばれる．開始ルーチンは，ノンブロッキングデータアクセス（MPI_FILE_IREADなど）と同様に操作を開始させる．終了ルーチンは，対応するtestやwait（MPI_WAITなど）と同様に操作を完了させる．ノンブロッキングデータアクセス操作と同様，ユーザはルーチンが未完了の状態では開始ルーチンに渡されたバッファを使用してはならず，バッファを安全に開放するには終了ルーチンによって操作を完了させておく必要がある．

ファイルハンドルfhでのスプリット集団データアクセス操作には，以下の意味論に関する規則が適用される．

- どのMPIプロセスにおいても，かつどの時点においても，1つのファイルハンドルに対応するアクティブなスプリット集団操作はたかだか1つでなければならない．
- 開始呼び出しは集団的オープンに参加したプロセスのグループに対して集団的であり，集団呼び出しの順序付け規則に従う．
- 終了呼び出しは集団的オープンに参加したプロセスのグループに対して集団的であり，集団呼び出しの順序付け規則に従う．各終了呼び出しは，同じ集団操作でのその前の開始呼び出しと対応している．「終了」呼び出しを行う場合，同じ操作のための，また対応付けが起きていない「開始」呼び出しが，ただ1個だけ前になければならない．
- 開始呼び出し（MPI_FILE_READ_ALL_BEGINなど）または終了呼び出し（MPI_FILE_READ_ALL_ENDなど）のいずれかを発行する場合，対応するブロッキング集団的ルーチンを使用して，スプリット集団データアクセスルーチンを実装しても良い．開始呼び出しと終了呼び出しは，ユーザとMPI実装で集団操作の最適化を可能にするために用意されている．
- スプリット集団操作は，該当する通常の集団操作とは対応しない．例えば，1つの集団的読み込み操作において，あるプロセスのMPI_FILE_READ_ALLは別のプロセスのMPI_FILE_READ_ALL_BEGIN／MPI_FILE_READ_ALL_ENDのペアには対応しない．
- スプリット集団ルーチンでは，開始ルーチンと終了ルーチンの両方でバッファを指定する必要がある．終了ルーチンでのデータ受信用のバッファを指定することにより，507ページの第16.2.2節「レジスタの最適化での問題」に記載された問題の（すべてではないが）多くを回避することができる．
- ファイルハンドルでの集団入出力操作は，そのファイルハンドルでのスプリット集団アクセスと同時（つまり，アクセスの開始と終了の間）に行うことはできない．

つまり，以下のコードは

```
MPI_File_read_all_begin(fh, ...);
...
MPI_File_read_all(fh, ...);
...
MPI_File_read_all_end(fh, ...);
```

誤りである．

- マルチスレッド実装では，あるプロセスによって呼び出されたスプリット集団開始／終了操作は，同じスレッドから呼び出されなければならない．この制限により，マルチスレッドの場合の実装を簡素化できる．（2つのスレッドが同じファイルハンドルに対しスプリット集団操作を開始することは許されていないため，ファイルハンドルに対して1つのスプリット集団操作しかアクティブにできないことに注意すること．）

これらのルーチンの引数の意味は，これと同等の集団操作の呼び出しの場合と同じである（例えば，MPI_FILE_READ_ALL_BEGINおよびMPI_FILE_READ_ALL_ENDの引数の定義はMPI_FILE_READ_ALLの引数の定義と同等である）．開始ルーチン（MPI_FILE_READ_ALL_BEGINなど）はスプリット集団操作を開始し，対応する終了ルーチン（MPI_FILE_READ_ALL_END）により完了するときに，同等の集団的ルーチン（MPI_FILE_READ_ALL）用に定義されたのと同じ結果が得られる．

一貫性の意味論のため（455ページの第13.6.1節），スプリット集団データアクセス操作の対応するペア（MPI_FILE_READ_ALL_BEGIN，MPI_FILE_READ_ALL_ENDなど）で1つのデータアクセスを構成するものする．

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

IN	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）

```
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype)
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf,
             int count, const MPI::Datatype& datatype) （廃止された呼び出し形式,
             第15.2節を参照）}
```

MPI_FILE_READ_AT_ALL_END(fh, buf, status)

IN	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Read_at_all_end(void* buf, MPI::Status& status) （廃止された
        呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_at_all_end(void* buf) （廃止された呼び出し形式, 第15.2節を参照）
        }
```

MPI_FILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

INOUT	fh	ファイルハンドル（ハンドル）
IN	offset	ファイルのオフセット（整数型）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）

```
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype)
MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
{void MPI::File::Write_at_all_begin(MPI::Offset offset, const void* buf,
        int count, const MPI::Datatype& datatype) （廃止された呼び出し形式,
        第15.2節を参照）}
```

MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Write_at_all_end(const void* buf, MPI::Status& status) （廃
        止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write_at_all_end(const void* buf) （廃止された呼び出し形式,
        第15.2節を参照）}
```

MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）

```
int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
{void MPI::File::Read_all_begin(void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_READ_ALL_END(fh, buf, status)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Read_all_end(void* buf, MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_all_end(void* buf) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）

```
int MPI_File_write_all_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
{void MPI::File::Write_all_begin(const void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_WRITE_ALL_END(fh, buf, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_all_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Write_all_end(const void* buf, MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write_all_end(const void* buf) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）

```
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
{void MPI::File::Read_ordered_begin(void* buf, int count,
              const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_READ_ORDERED_END(fh, buf, status)

INOUT	fh	ファイルハンドル（ハンドル）
OUT	buf	バッファの先頭アドレス（選択型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Read_ordered_end(void* buf, MPI::Status& status) （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Read_ordered_end(void* buf) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
IN	count	バッファ内の要素の数（整数型）
IN	datatype	各バッファの要素のデータ型（ハンドル）

```
int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
{void MPI::File::Write_ordered_begin(const void* buf, int count,
             const MPI::Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_WRITE_ORDERED_END(fh, buf, status)

INOUT	fh	ファイルハンドル（ハンドル）
IN	buf	バッファの先頭アドレス（選択型）
OUT	status	ステータスオブジェクト（ステータス型）

```
int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
{void MPI::File::Write_ordered_end(const void* buf, MPI::Status& status)
             （廃止された呼び出し形式, 第15.2節を参照）}
{void MPI::File::Write_ordered_end(const void* buf) （廃止された呼び出し形式,
             第15.2節を参照）}
```

13.5 ファイルの相互運用性

最も基本的なレベルで，ファイルの相互運用性とは，データのビット列だけでなく，ビット列が表現する実際の情報として，ファイルにすでに書き込まれている情報を読み取る能力を指す．MPIでは1つのMPI環境内での完全な相互運用性を保証しており，外部のデータ表現（448ページの第13.5.2節）およびデータ変換関数（449ページの第13.5.3節）を通して，その環境の外側での相互運用性サポートを強化している．

2つのプロセスを同時に起動し，1つのMPI_COMM_WORLDに常駐させることが可能であれば，1つのMPI環境内の相互運用性（「運用性」と考えることができる）により，一貫性の制限の対象として（455ページの第13.6.1節を参照），1つのMPIプロセスによって書き込まれたファイルデータを他の任意のMPIプロセスによって読み取れることが保証される．また，両方のプロセスはデータが書き込まれたファイル内ですべての絶対バイトオフセットで同じデータ値が見えなければならない．

この単一環境ファイル相互運用性は，プロセスの数に関係なくファイルのデータにアクセスできることを示している．

ファイルの相互運用性には以下の3つの側面がある．

- ビットの転送
- 異なるファイル構造間での変換
- 異なるマシンの表現間での変換

ファイルの相互運用性の最初の2つの側面は，マシンに依存する部分が大きいため，この標準のスコープを超えている．しかし，MPI環境との間でのファイルのビットの転送（ファイルをテープに書き込むなどの方法による）は，すべてのMPI実装によってサポートされていなければならない．特に，実装では，POSIX `cp`，`rm`，`mv`のような一般的な操作がファイルに対して行われうる方法を指定する必要がある．また，提供される機能により絶対バイトオフセット間の対応が維持されると考えられる（例えば，考えられるファイル構造の変換後に，MPI環境のバイトオフセット102のデータビットはMPI環境外のバイトオフセット102となる）．例えば，上記のオフセットの一貫性が維持できるのであれば，ネイティブファイルシステムとMPI環境の間でファイルの転送と変換を行うシンプルなオフライン変換機能で十分である．MPIの高品質な実装では，ネイティブファイルシステムでファイルの操作用に用意されているのと同じまたは類似のツールを使用して，ユーザがMPIファイルを操作することができる．

ファイルの相互運用性の残りの側面である異なるマシンの表現間での変換は，etypeとファイル型で指定された型情報によりサポートされる．この機能を利用すると，MPIを使用するかどうかに関係なく，また動作するマシンのアーキテクチャに関係なく，ファイルの情報を2つのアプリケーションの間で共有することができる．

MPIは“native”，“internal”，“external32”という複数のデータ表現をサポートしている．これ以外のデータ表現がサポートされている場合もある．MPIではユーザ定義のデータ表現もサポートしている（449ページの第13.5.3節を参照）．“native”および“internal”データ表現は実装依存であり，“external32”表現はすべてのMPI実装に共通で，ファイルの相互運用性を容易にする．データ表現はMPI_FILE_SET_VIEWの*datarep*引数で指定される．

> **ユーザへのアドバイス** MPIでは，ファイルの書き込み時に使用されたデータ表現に関する情報を保持することが保証されていない．そのため，ファイルデータを正しく取得するには，MPIアプリケーションで責任を持って，ファイルの生成時に使用されたのと同じデータ表現を指定する必要がある．（**ユーザへのアドバイス終わり**）

“native” この表現のデータは，メモリ内とまったく同じようにファイルに格納される．このデータ表現の利点は，純粋な同一機種環境において，型変換の際にデータの精度と入出力の性能が失われないことである．欠点は，異機種MPI環境内でトランスペアレントな相互運用性が失われることである．

ユーザへのアドバイス このデータ表現を使用するのは，同一機種MPI環境でのみ，またはMPIアプリケーションがそれ自体でデータの型変換を実行できる場合のみに限定する必要がある．（**ユーザへのアドバイス終わり**）

実装者へのアドバイス MPIメッセージ通信の上位に読み込みおよび書き込み操作を実装する場合，メッセージルーチンがデータの型変換を実行しないようにメッセージデータの型をMPI_BYTEにする必要がある．（**実装者へのアドバイス終わり**）

“internal” このデータ表現は同一機種環境または異機種環境での入出力操作に使用することができ，実装では必要に応じて型変換を実行する．実装は選択した任意の形式でデータを格納できるが，任意の1つのファイル内のすべての定義済みのデータ型に対して定数の範囲が維持されるという制限がある．生成されるファイルが再使用できる環境は実装時に定義され，実装により文書化する必要がある．

根拠 このデータ表現を使用すると，ファイルの再使用の方法に関して実装で定義される制限はあるものの，異機種環境で入出力を効率的に実行することができる．（**根拠の終わり**）

実装者へのアドバイス “external32”は“internal”で提供される機能の上位集合であるため，実装の選択により“internal”を“external32”として実装することもできる．（**実装者へのアドバイス終わり**）

“external32” このデータ表現では，読み込み/書き込み操作により，448ページの第13.5.2節で定義された“external32”表現との間ですべてのデータが変換されることが規定されている．通信のためのデータ変換規則はこれらの変換にも適用される（MPI-1文書の25〜27ページの3.3.2節を参照）．ストレージ媒体のデータは常にこの標準の表現となり，メモリ内のデータは常にローカルプロセスのネイティブ表現となる．

このデータ表現にはいくつかの利点がある．第1に，異機種MPI環境でファイルを読み込むすべてのプロセスは自動的にデータをそれぞれのネイティブ表現に変換する．第2に，ファイルをあるMPI環境からエクスポートし，別のMPI環境にインポートして，インポート先の環境でファイル内のすべてのデータを読み込めることを保証できる．

このデータ表現の欠点は，データの型変換の際にデータの精度と入出力の性能が失われる可能性があることである．

実装者へのアドバイス MPIメッセージ通信の上位に読み込みおよび書き込み操作を実装する場合，クライアントがデータをMPI_BYTEとして受信し，“external32”表現へ変換したり，“external32”表現へ変換してMPI_BYTEとしてデータを送信すべきである，これにより，データの二重の型変換やこれに

伴う精度や性能のさらなる喪失が回避できる． （実装者へのアドバイス終わり）

13.5.1 ファイルの相互運用性のためのデータ型

ファイルのデータ表現が“native”以外である場合，etypeとファイル型の構成に注意する必要がある．任意のデータ型コンストラクタ関数を使用することができるが，変位をバイト単位で適用できる関数の場合，使用するファイルのデータ表現に対して，ファイル内の値で変位を指定する必要がある．MPIはこれらのバイト変位をそのまま解釈し，スケールは行わない．関数MPI_FILE_GET_TYPE_EXTENTはファイル内でのデータ型の範囲を計算するのに使用できる．可搬なデータ型であるetypeとファイル型については（13ページの第2.4節を参照），MPIはファイルのデータ表現に合わせてデータ型の変位のスケールを行う．読み込み/書き込みルーチンに引数として渡されるデータ型はメモリ内のデータ配置を指定するため，これらは常にメモリ内の変位に対応する変位を使用して構成する必要がある．

ユーザへのアドバイス ファイルは，論理的には，ファイルサーバーのメモリに格納されているものと考えることができる． etypeとfiletypeは，呼び出しプロセスでの定義に使用されるのと同じ呼び出しのシーケンスにより，このファイルサーバーで定義されているものとして解釈される．データ表現が“native”である場合，この論理ファイルサーバーは呼び出しプロセスと同じアーキテクチャ上で動作するため，これらの型により，呼び出しプロセスのメモリ内で定義されるのと同じデータ配置がファイル上で定義される． etypeとfiletypeが可搬なデータ型である場合，ファイル内で定義されるデータ配置は，呼び出しプロセスのメモリ内でスケーリングファクターに従って定義されるものと同じになる．このスケーリングファクターを計算するためにルーチンMPI_FILE_GET_FILE_EXTENTを使用することができる． そのため，“internal”，“external32”，またはユーザ定義データ表現を持つ異機種環境においても，2つの等価で可搬なデータ型を使えばファイル内で同じデータ配置を定義することができる．あるいは，任意のアーキテクチャで型マップと範囲が同じになるようにetypeとfiletypeを構成する必要がある． このためには，これらが明示的な上限と下限（MPI_LBおよびMPI_UBマーカを使用して，またはMPI_TYPE_CREATE_RESIZEDを使用して定義）を持つようにすればよい．etypeおよびfiletypeを構成するデータ型にも，以下の場合にはこの条件が適用される．このデータ型がMPI_TYPE_CONTIGUOUSの呼び出しによって明示的に，あるいは1より大きいblocklength引数により暗黙的に，連続的に複製される場合である．etypeあるいはfiletypeが可搬でなく，アーキテクチャに依存する型マップあるいは範囲を持っている場合，ファイル上でこれによって指定されるデータ配置は実装依存である．

“native”以外のファイルのデータ表現は，対応するメモリ内のデータ表現とは異なる場合がある．そのため，これらのファイルのデータ表現では，ビューを指定する初期変位を含め，ファイルの位置付けのために，ハードコーディングされたバイト

オフセットを使用しないことが重要である．データアクセス操作で可搬なデータ型を使用する場合（13ページの第2.4節を参照），データ表現に合わせてデータ型の穴がスケールされる．しかし，この技法が有効なのは，ファイルビューを生成したすべてのプロセスが同じ定義済みのデータ型からetypeを作成した場合のみである．例えば，あるプロセスがMPI_INTから作成されたetypeを使用し，別のプロセスがMPI_FLOATから作成されたetypeを使用する場合，これらの型の相対サイズがデータ表現ごとに異なる可能性があるため，生成されるビューは可搬でないことがある．（**ユーザへのアドバイス終わり**）

MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)

IN	fh	ファイルハンドル（ハンドル）
IN	datatype	データ型（ハンドル）
OUT	extent	データ型の範囲（整数型）

```
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
MPI_Aint *extent)
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
{MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype) const
```
（廃止された呼び出し形式, 第15.2節を参照）}

ファイルfhのdatatypeの範囲を返す．この範囲は，ファイルfhにアクセスするすべてのプロセスで同じとなる．現在のビューでユーザ定義のデータ表現を使用する場合（449ページの第13.5.3節を参照），MPIはdtype_file_extent_fnコールバックを使用して範囲を計算する．

実装者へのアドバイス ユーザ定義のデータ表現の場合，派生データ型の範囲は，dtype_file_extent_fnを使用してまずこの派生データ型での定義済みのデータ型の範囲を調べることにより，計算することができる（449ページの第13.5.3節を参照）．（**実装者へのアドバイス終わり**）

13.5.2 外部のデータ表現：“external32”

すべてのMPI実装で，この節で定義するデータ表現をサポートする必要がある．オプションのデータ型（MPI_INTEGER2など）のサポートは必要ない．

浮動小数点数の値はすべて適切なサイズのビッグエンディアンIEEE形式[27]である．浮動小数点数の値は3つのうちのいずれかの IEEE形式で表現される．これらはそれぞれ，4，8，16バイトのストレージを必要とするIEEE “Single”（単精度），“Double”（倍精度），“Double Extended”（四倍精度）形式である．IEEE “Double Extended”形式の場合，MPIは，フォーマット幅16バイト，指数部15 ビット，bias = +16383，仮数部112ビットを指定する．エンコードは “Double”形式と同様である．整数値はすべて2の補数のビッグエンディアン形式となる．ビッグエンディアンでは，最上位バイトが最もアドレス

の小さいバイトに入る．C言語の_Bool， Fortran言語のLOGICAL，C++言語のboolの場合， 0はfalse，0以外はtrueを示す．C言語 のfloat _Complex，double _Complex，long double _Complexと，Fortran言語のCOMPLEXおよびDOUBLE COMPLEXは，実数部と虚数部の浮動小数点数形式の値のペアで表現される．文字はISO 8859-1形式である[28]．（MPI_WCHARの）ワイド文字はUnicode形式である[47]．

符号付き数値（MPI_INT，MPI_REAL）は最上位ビットが符号ビットとなる．MPI_COMPLEXおよびMPI_DOUBLE_COMPLEXは実数部および虚数部の最上位ビットが各部の符号ビットとなる．

IEEEの仕様に従い[27]，“NaN”（非数）はシステムごとに異なる． MPI内で“NaN”以外のものとして解釈されないようにする必要がある．

> **実装者へのアドバイス** MPIでの“NaN”の扱いは，XDRで使用されるアプローチと同様である（ftp://ds.internic.net/rfc/rfc1832.txtを参照）． （**実装者へのアドバイス終わり**）

データはすべて，型に関係なく，バイト単位にアラインされる．データ項目はすべて，連続してファイルに格納される（ファイルビューが連続の場合）．

> **実装者へのアドバイス** LOGICALとboolについては，その値を特定するにはすべてのバイトをチェックせねばならない． （**実装者へのアドバイス終わり**）

> **ユーザへのアドバイス** MPI_PACKED型はバイトとして扱われ，変換されない．MPI_PACKEDはパックバッファの先頭にヘッダを配置するオプションを備えていることを知っておく必要がある． （**ユーザへのアドバイス終わり**）

MPI_TYPE_CREATE_F90_REAL，MPI_TYPE_CREATE_F90_COMPLEX，MPI_TYPE_CREATE_F90_INTEGERから返される定義済みのデータ型のサイズは，515ページの第16.2.5節で定義されている．

> **実装者へのアドバイス** サイズの大きな整数をサイズの小さな整数に変換する場合，下位のバイトのみが移動される．符号ビットの値が欠落しないよう注意する必要がある．これにより，データの範囲がサイズの小さな整数の範囲内にある場合，変換エラーを起こさずに済む． （**実装者へのアドバイス終わり**）

表13.2に，“external32”形式の定義済みのデータ型のサイズを示す．

13.5.3 ユーザ定義のデータ表現

以下の2つの状況には，表現を指定することでは対処できない．

1. ユーザが，実装で識別できない表現によりファイルに書き込もうとしている場合．
2. ユーザが，実装で識別できない表現で記述されたファイルを読み取ろうとしている場合．

Type	Length	Optional Type	Length
MPI_PACKED	1	MPI_INTEGER1	1
MPI_BYTE	1	MPI_INTEGER2	2
MPI_CHAR	1	MPI_INTEGER4	4
MPI_UNSIGNED_CHAR	1	MPI_INTEGER8	8
MPI_SIGNED_CHAR	1	MPI_INTEGER16	16
MPI_WCHAR	2		
MPI_SHORT	2	MPI_REAL2	2
MPI_UNSIGNED_SHORT	2	MPI_REAL4	4
MPI_INT	4	MPI_REAL8	8
MPI_UNSIGNED	4	MPI_REAL16	16
MPI_LONG	4		
MPI_UNSIGNED_LONG	4	MPI_COMPLEX4	2*2
MPI_LONG_LONG_INT	8	MPI_COMPLEX8	2*4
MPI_UNSIGNED_LONG_LONG	8	MPI_COMPLEX16	2*8
MPI_FLOAT	4	MPI_COMPLEX32	2*16
MPI_DOUBLE	8		
MPI_LONG_DOUBLE	16		
MPI_C_BOOL	1		
MPI_INT8_T	1	C++ Types	Length
MPI_INT16_T	2	MPI_CXX_BOOL	1
MPI_INT32_T	4	MPI_CXX_FLOAT_COMPLEX	2*4
MPI_INT64_T	8	MPI_CXX_DOUBLE_COMPLEX	2*8
MPI_UINT8_T	1	MPI_CXX_LONG_DOUBLE_COMPLEX	2*16
MPI_UINT16_T	2		
MPI_UINT32_T	4		
MPI_UINT64_T	8		
MPI_AINT	8		
MPI_OFFSET	8		
MPI_C_COMPLEX	2*4		
MPI_C_FLOAT_COMPLEX	2*4		
MPI_C_DOUBLE_COMPLEX	2*8		
MPI_C_LONG_DOUBLE_COMPLEX	2*16		
MPI_CHARACTER	1		
MPI_LOGICAL	4		
MPI_INTEGER	4		
MPI_REAL	4		
MPI_DOUBLE_PRECISION	8		
MPI_COMPLEX	2*4		
MPI_DOUBLE_COMPLEX	2*8		

表 13.2: “external32”の定義済みのデータ型のサイズ

ユーザ定義のデータ表現を使用する場合，ユーザは入出力ストリームにサードパーティのコンバータを組み込んで，データ表現の変換を行うことができる．

MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn, dtype_file_extent_fn, extra_state)

IN	datarep	データ表現識別子（文字列）
IN	read_conversion_fn	ファイル表現からネイティブ表現に変換するために呼び出される関数（関数）
IN	write_conversion_fn	ネイティブ表現からファイル表現に変換するために呼び出される関数（関数）
IN	dtype_file_extent_fn	ファイル内で表現されたデータ型の範囲を取得するために呼び出される関数（関数）
IN	extra_state	追加ステート

```
int MPI_Register_datarep(char *datarep,
MPI_Datarep_conversion_function *read_conversion_fn,
MPI_Datarep_conversion_function *write_conversion_fn,
MPI_Datarep_extent_function *dtype_file_extent_fn, void *extra_state)
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER IERROR
{void MPI::Register_datarep(const char* datarep,
             MPI::Datarep_conversion_function* read_conversion_fn,
             MPI::Datarep_conversion_function* write_conversion_fn,
             MPI::Datarep_extent_function* dtype_file_extent_fn,
             void* extra_state) （廃止された呼び出し形式, 第15.2節を参照）}
```

この呼び出しはread_conversion_fn，write_conversion_fn，dtype_file_extent_fnとデータ表現識別子datarepを関連付ける．datarepはMPI_FILE_SET_VIEWの引数として使用し，その後のデータアクセス操作により，ファイルのデータ表現とネイティブ表現の間でアクセスされるすべてのデータ項目を変換するための変換関数を呼び出すことができる．MPI_REGISTER_DATAREPはローカル操作で，MPIプロセスを呼び出すためのデータ表現の登録のみを行う．datarepがすでに定義されている場合，デフォルトのファイルエラーハンドラによりエラークラスMPI_ERR_DUP_DATAREPのエラーが報告される（465ページの第13.7節を参照）．データ表現文字列の長さはMPI_MAX_DATAREP_STRINGの値に制限されている．MPI_MAX_DATAREP_STRINGの値は64以上でなければならない．データ表現を削除し，関連するリソースを解放するためのルーチンは用意されていない．アプリケーションにより多数生成されることはないと考えられる．

範囲のコールバック

```
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
MPI_Aint *file_extent, void *extra_state);
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
{typedef void MPI::Datarep_extent_function(const MPI::Datatype& datatype,
              MPI::Aint& file_extent, void* extra_state); （廃止された呼び出し形
              式．第15.2節を参照）}
```

関数dtype_file_extent_fnは，ファイル表現のdatatypeを格納するのに必要なバイト数をfile_extentに返す必要がある．この関数には，MPI_REGISTER_DATAREP呼び出しに渡された引数がextra_stateに渡される．MPIはこのルーチンを呼び出すのは，ユーザが使用した定義済みのデータ型の場合のみである．

Datarep変換関数

```
typedef int MPI_Datarep_conversion_function(void *userbuf,
MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position,
void *extra_state);
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
POSITION, EXTRA_STATE, IERROR)
    <TYPE> USERBUF(*), FILEBUF(*)
    INTEGER COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) POSITION
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
{typedef void MPI::Datarep_conversion_function(void* userbuf,
              MPI::Datatype& datatype, int count, void* filebuf,
              MPI::Offset position, void* extra_state); （廃止された呼び出し形式．
              第15.2節を参照）}
```

関数read_conversion_fnはファイルのデータ表現からネイティブ表現への変換を行わなければならない．このルーチンを呼び出す前に，MPIはfilebufを割り当て，count個の連続するデータ項目をfilebufに格納する． 各データ項目の型は，datatypeの型シグネチャ内の，定義済みデータ型の対応するエントリと一致する．この関数にはMPI_REGISTER_DATAREPの呼び出しに渡された引数がextra_stateに渡される． この関数はcount個のすべてのデータ項目をfilebufからuserbufに，datatypeによって規定された配置でコピーする必要がある．また，この際に各データ項目をファイル表現からネイティブ表現に変換する必要がある．datatypeはユーザが読み込み関数に渡したデータ型と同じになる．datatypeのサイズがcount個のデータ項目のサイズよりも小さい場合，変換関数はdatatypeをuserbuf上に連続して敷き詰められたものとして扱う必要がある．変換関数は，変換されたデータをuserbufに格納する．その開始位置は（敷き詰められた）datatypeの中を指すpositionによって決まる．

ユーザへのアドバイス 変換関数とMPI_PACKおよびMPI_UNPACKには類似点があるが，引数countおよびpositionの使用法の違いに注意する必要がある．変換関数では，countはデータ項目の数（datatypeの型マップのエントリ数）で，positionはこの型マップの中を指すインデックスである．MPI_PACKでは，incountはdatatype全体の数で，positionはバイト数である． （**ユーザへのアドバイス終わり**）

実装者へのアドバイス 変換されたread操作は以下のように実装できる．

1. すべてのデータ項目のファイルの範囲を取得する

2. count個のすべてのデータ項目を格納できる大きさのfilebufを割り当てる
3. ファイルからfilebufにデータを読み込む
4. read_conversion_fnを呼び出し，データを変換してuserbufに格納する
5. filebufを解放する

（実装者へのアドバイス終わり）

読み込み操作から変換するすべてのデータを格納できるだけの大きさのバッファをMPIで割り当てられない場合，同じdatatypeとuserbufを使用して変換関数を繰り返し呼び出し，データチャンクをfilebufに読み込んで変換し，次のデータチャンクに進む，ということを繰り返すことができる．最初の呼び出しでは（またすべての変換対象のデータがfilebufに格納された場合），MPIはpositionに0を設定してこの関数を呼び出す．この呼び出し中に変換されたデータは，datatype内の最初のcount個のデータ項目に従ってuserbufに格納される．そして，その後の変換関数の呼び出しで，前の呼び出しで変換されたcount個の項目の分だけpositionの値を加算する．userbufポインタは変更されない．

根拠 変換関数に位置と転送のための1つのデータ型を渡すことにより，変換関数はデータ型をデコードし，内部表現に変換し，それをキャッシュすることができる．そして，その後の呼び出しで，変換関数でpositionを使用してデータ型内でのその場所をすばやく見つけ出し，前の呼び出しの終了時に中断された変換データの格納を継続することができる． （根拠の終わり）

ユーザへのアドバイス 通常，変換関数はデータ型に内部表現をキャッシュすることができるが，複数の変換操作で同じデータ型が同時に使用されている可能性があるため，処理中の変換操作に固有の状態情報はキャッシュしてはならない． （ユーザへのアドバイス終わり）

関数write_conversion_fnはネイティブ表現からファイルのデータ表現への変換を行わなければならない．このルーチンを呼び出す前に，MPIはcount個の連続するデータ項目を格納できるだけの大きさのfilebufを割り当てる． 各データ項目の型は，datatypeの型シグネチャ内の，定義済みデータ型の対応するエントリと一致する． この関数はcount個のデータ項目をuserbufからdatatypeによって規定された配置で取り出し， filebufに連続した配置でコピーする必要がある．また，この際に各データ項目をネイティブ表現からファイル表現に変換する必要がある． datatypeのサイズがcount個のデータ項目のサイズよりも小さい場合，変換関数はdatatypeをuserbuf上に連続して敷き詰められたものとして扱う必要がある．

この関数は，データをuserbufにコピーするが，その開始位置は（敷き詰められた）datatypeの中を指すpositionによって決まる． datatypeはユーザが書き込み関数に渡したデータ型と同じになる． この関数にはMPI_REGISTER_DATAREPの呼び出しに渡された引数がextra_stateに渡される．

定義済みの定数MPI_CONVERSION_FN_NULLはwrite_conversion_fnまたはread_conversion_fnとして使用することができる．その場合，MPIはそれぞれwrite_conversion_fnまたはread_conversion_fnの呼び出しを行わないが，ネイティブデータ表現を使用して，要求されたデータアクセスを行う．

MPI実装では，アクセスされるすべてのデータが変換されていることを保証するため，要求されたすべてのデータ項目を格納できるだけの大きさのfilebufを使用するか，または同じdatatype引数と適切なpositionの値を使用して繰り返し変換関数を呼び出す必要がある．

この節のコールバックルーチン（read_conversion_fn，write_conversion_fn，dtype_file_extent_fn）が実装により呼び出されるのは，422ページの第13.4節で説明したいずれかの読み込みまたは書き込みルーチン，またはMPI_FILE_GET_TYPE_EXTENTがユーザによって呼び出されたときのみである．dtype_file_extent_fnにはユーザが使用した定義済みのデータ型のみが渡される．変換関数には，ユーザが上記のいずれかのルーチンに渡したものと同等のデータ型のみが渡される．

変換関数は再入可能でなければならない． ユーザ定義のデータ表現ではどの型についてもバイトのアライメントしか使用できない．また，変換関数で集団的ルーチンを呼び出したりdatatypeを解放したりするのは誤りである．

変換関数はエラーコードを返す必要がある．返されたエラーコードの値がMPI_SUCCESS以外の場合，実装ではエラークラスMPI_ERR_CONVERSIONのエラーを報告する．

13.5.4 データ表現の対応付け

ユーザは責任を持って，ファイルからのデータの読み取りに使用するデータ表現と，そのデータをファイルに書き込むのに使用したデータ表現が**互換性がある**ことを保証する必要がある．

一般的に，ファイルの書き込み時と読み取り時に同じデータ表現の名前を使用していても，それだけで表現に互換性があるとは言えない．同様に，2つの異なる実装で異なる表現の名前を使用していても，表現の互換性がある場合もある．

“external32”表現を使用すると，“native”表現を使用した場合と比べて精度が失われ，性能が低下することがあるが，互換性は得られる．以下の少なくとも1つのの条件が満たされている場合，“external32”を使用すると互換性が保証される．

- データアクセスルーチンが448ページの第13.5.2節に示した，入出力に参加しているすべての実装によってサポートされる型を直接使用する．データ項目を書き込むのに使用した定義済みの型を，データ項目の読み取りにも使用する必要がある．

- Fortran 90言語のプログラムの場合，データアクセスに参加するプログラムが，精度や範囲を指定するMPIルーチンを使用して互換性のあるデータ型を取得する（511ページの第16.2.5節を参照）．

- 指定のデータ項目について，データアクセスに参加するプログラムが，データ項目の書き込み/読み取りのために互換性のある定義済みの型を使用する．

ユーザ定義のデータ表現は，別の実装の“native”または“internal”表現との実装の互換性を提供するために使用することもできる．

> **ユーザへのアドバイス** 511ページの第16.2.5節では，対応するデータ型の使用が異機種環境でサポートされているルーチンが定義され，その使用法の例が示されている． （**ユーザへのアドバイス終わり**）

13.6 一貫性と意味論

13.6.1 ファイルの一貫性

一貫性の意味論は，1つのファイルに対して複数のアクセスを行った結果を定義する．MPIでのすべてのファイルアクセスは，集団的オープンで生成された特定のファイルハンドルに関係する．MPIは3つのレベルの一貫性を備えている．つまり，1つのファイルハンドルを使用したすべてのアクセス間での逐次一貫性と，アトミックモードを有効にして1つの集団的オープンにより生成されたファイルハンドルを使用したすべてのアクセス間での逐次一貫性と，上記以外の，ユーザによって課されたアクセス間の一貫性である． 逐次一貫性とは，操作がプログラムの順序と整合したある逐次の順番で行われたかのように一連の操作が振る舞う．また，正確なアクセスの順序が指定されていなくても各アクセスがアトミックに行われることを意味する． ユーザは，プログラム中の操作の順序と複数のMPI_FILE_SYNCの呼び出しを使用して一貫性を強制することができる．

FH_1をファイルFOOの特定の1回の集団的オープンから生成されたファイルハンドルの集合とし，FH_2をFOOの別の集団的オープンから生成されたファイルハンドルの集合とする． FH_1とFH_2には何も制限はないことに注意すること． FH_1とFH_2のサイズは違っていてもよく，それぞれのオープンに使用されたプロセスのグループは共通部分を持っていてもよい．また，FH_1のファイルハンドルはFH_2のファイルハンドルの生成前に破壊されていてもよい．次の3つのケースについて考えてみる．つまり，ファイルハンドルが1つの場合（$fh_1 \in FH_1$など）と，1回の集団的オープンで2つのファイルハンドラが生成された場合（$fh_{1a} \in FH_1$，$FH_{1b} \in FH_1$など）と，異なる集団的オープンで2つのファイルハンドルが生成された場合（$fh_1 \in FH_1$，$fh_2 \in FH_2$など）である．

一貫性の意味論のため，スプリット集団データアクセス操作の対応するペア（MPI_FILE_READ_ALL_BEGIN，MPI_FILE_READ_ALL_ENDなど）（439ページの第13.4.5節）は1つのデータアクセス操作を構成するものとする．同様に，ノンブロッキングデータアクセスルーチン（MPI_FILE_IREADなど）と要求を完了するルーチン（MPI_WAITなど）も，1つのデータアクセス操作を構成するものとする．以下のすべての場合に，これらのデータアクセス操作にはブロッキングデータアクセス操作と同じ制限が適用される．

> **ユーザへのアドバイス** MPI_FILE_IREADとMPI_WAITのペアの場合，操作は

MPI_FILE_IREADが呼び出された時点で開始され，MPI_WAITが戻った時点で終了する．（**ユーザへのアドバイス終わり**）

A_1とA_2が2つのデータアクセス操作であるとする．D_1 (D_2)をA_1 (A_2)でアクセスされるすべてのバイトの絶対バイト変位の集合とする．$D_1 \cap D_2 \neq \emptyset$である場合に，2つのデータアクセスは**オーバーラップ**する．これらがオーバーラップし，少なくとも1つが書き込みアクセスである場合，2つのデータアクセスは**コンフリクト**する．

SEQ_{fh}を1つのファイルハンドル上のファイル操作のシーケンスとし，そのファイルハンドルへのMPI_FILE_SYNCによって区切られているとする．（ファイルを開くときも閉じるときも，暗黙的にMPI_FILE_SYNCが実行される．） シーケンス内に書き込みのデータアクセス操作があるか，またはシーケンス内にファイルの状態を変更する（MPI_FILE_SET_SIZE，MPI_FILE_PREALLOCATEなど）ファイル操作がある場合，SEQ_{fh}は「書き込みシーケンス」とする． SEQ_1とSEQ_2の2つのシーケンスについて，一方のシーケンスが（時間的に）他方のシーケンスの完全に前に来る場合，これらは**同時並行**ではないとする．

特定のファイルに対するすべてのアクセスの間で逐次一貫性を保証するための要件は，以下の3つのケースに分けられる．これらの要件のうち1つでも満たされていない場合，そのファイル内のすべてのデータの値は実装依存となる．

ケース1: $fh_1 \in FH_1$　アトミックモードが設定されている場合，fh_1に対するすべての操作は逐次に整合が取られる．ノンアトミックモードが設定されている場合，fh_1に対するすべての操作は，同時でないかコンフリクトしないとき，またはその両方のときに逐次に整合が取られる．

ケース2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$　A_1がfh_{1a}を使用したデータアクセス操作で，A_2がfh_{1b}を使用したデータアクセス操作であるとする．アクセスA_1については，A_1とコンフリクトするアクセスA_2がない場合，MPIは逐次一貫性を保証する．

しかし，POSIXの意味論と違って，コンフリクトするアクセスに対するデフォルトのMPIの意味論では逐次一貫性が保証されない．A_1とA_2がコンフリクトする場合，逐次一貫性を保証するには，MPI_FILE_SET_ATOMICITYルーチンを使用してアトミックモードを有効にするか，以下のケース3に示す条件を満たす必要がある．

ケース3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$　別の集団的オープンで生成されたファイルハンドルを使用して1つのファイルにアクセスする場合を考える．逐次一貫性を保証するには，MPI_FILE_SYNCを使用する必要がある（ファイルを開くときも閉じるときも，暗黙的にMPI_FILE_SYNCが実行される）．

1つのファイルへの複数のアクセスの間で逐次一貫性が保証されるのは，ファイルへの書き込みシーケンスSEQ_1について，ファイルに対してSEQ_1と**同時並行**であるシーケンスSEQ_2がない場合である． 書き込みシーケンスがある状態で逐次一貫性を保証するには，シーケンスの非同時性を保証するメカニズムと一緒にMPI_FILE_SYNCを使用する必要がある．

これらの一貫性の意味論のうちいくつかについてのより詳しい説明は，461ページの第13.6.10節の例を参照すること．

MPI_FILE_SET_ATOMICITY(fh, flag)

INOUT	fh	ファイルハンドル（ハンドル）
IN	flag	アトミックモードを設定するにはtrue，非アトミックモードを設定するにはfalse（論理型）

```
int MPI_File_set_atomicity(MPI_File fh, int flag)
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
{void MPI::File::Set_atomicity(bool flag) （廃止された呼び出し形式, 第15.2節を参照）}
```

FHを1つの集団的オープンによって生成されたファイルハンドルの集合とする．FHを使用したデータアクセス操作の一貫性の意味論はFHに対してMPI_FILE_SET_ATOMICITYを集団的に呼び出すことにより設定される．MPI_FILE_SET_ATOMICITYは集団的で，グループ内のすべてのプロセスがfhとflagに同じ値を渡す必要がある．flagがtrueの場合はアトミックモードが設定され，flagがfalseの場合は非アトミックモードが設定される．

開かれているファイルに対して一貫性の意味論を変更した場合，影響を受けるのは新しいデータアクセスのみである．完了したデータアクセスについては，その実行中に有効であった一貫性の意味論に従うよう保証される．（MPI_WAITなどにより）完了していないノンブロッキングデータアクセスおよびスプリット集団操作は，非アトミックモードの一貫性の意味論に従うことだけが保証される．

> **実装者へのアドバイス** アトミックモードにより保証される意味論は非アトミックモードにより保証される意味論よりも強いため，実装は未完了の要求について，より厳格なアトミックモードの意味論に従っても良い． （**実装者へのアドバイス終わり**）

MPI_FILE_GET_ATOMICITY(fh, flag)

IN	fh	ファイルハンドル（ハンドル）
OUT	flag	アトミックモードの場合はtrue，非アトミックモードの場合は false （論理型）

```
int MPI_File_get_atomicity(MPI_File fh, int *flag)
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
{bool MPI::File::Get_atomicity() const （廃止された呼び出し形式, 第15.2節を参照）}
```

MPI_FILE_GET_ATOMICITYは，1つの集団的オープンによって生成されたファイルハンドルの集合に対するデータアクセス操作の現在の一貫性の意味論を返す． flagがtrueの

場合はアトミックモードが有効であり，flagがfalseの場合は非アトミックモードが有効である．

MPI_FILE_SYNC(fh)

INOUT	fh		ファイルハンドル（ハンドル）

```
int MPI_File_sync(MPI_File fh)
MPI_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR
{void MPI::File::Sync()（廃止された呼び出し形式, 第15.2節を参照）}
```

fhを使用してMPI_FILE_SYNCを呼び出すと，呼び出し側プロセスによるfhへのそれまでのすべての書き込みがストレージデバイスに転送される．別のプロセスによりストレージデバイスが更新されている場合，その後の呼び出し側プロセスによるfhの読み込みではこのようなすべての更新が検知可能になる．場合によっては，逐次一貫性を保証するために，MPI_FILE_SYNCが必要になる場合もある（上記を参照）．

MPI_FILE_SYNCは集団操作である．

ユーザは責任を持って，fhに対するすべてのノンブロッキング要求とスプリット集団操作を完了しているのを確認してMPI_FILE_SYNCを呼び出す必要がある． そうでない場合，MPI_FILE_SYNCの呼び出しは誤りである．

13.6.2 ランダムアクセスと逐次ファイル

MPIは普通のランダムアクセスファイルと，パイプやテープファイルなどの逐次ストリームファイルを区別する．逐次ストリームファイルはMPI_MODE_SEQUENTIALフラグをamodeに設定して開く必要がある．これらのファイルでは，実行できるデータアクセス操作は共有ファイルポインタの読み込みと書き込みのみである． 穴のあるファイル型とetypeを用いるのは誤りである．また，ポインタの概念は意味を持たないため，MPI_FILE_SEEK_SHAREDおよびMPI_FILE_SEEK_SHAREDの呼び出しは誤りであり，データアクセスルーチンのために規定されたポインタ更新規則は適用されない．データアクセス操作によってアクセスされるデータの量は，ファイル終端に達した場合やエラーが発生した場合を除いて，要求された量となる．

> **根拠** このことからわかるように，パイプでの読み取りは，要求されたデータの量が利用可能になるまで，またはパイプへのプロセスによる書き込みがファイル終端を発行するまで，常に待ち合わせる． （**根拠の終わり**）

最後に，磁気テープやストリーミングネットワーク接続に対応するものなど，一部の逐次ファイルでは，ファイルへの書き込みが問題となることがある．つまり， 書き込みを行った後に，切り詰め（MPI_FILE_SET_SIZEでsizeを現在の位置に設定したのと同じ動作）として動作することがある．

13.6.3 プログレス

MPIのプログレスルールは，ユーザに対する約束でもあり，同時に実装者に課される制約の集合でもある．プログレスルールにより，インターフェイスの仕様のみよりも厳しく実装の選択肢が制限されうる場合，プログレスルールのほうが優先される．

例外的な条件（リソースの不足など）によりエラーが発生する場合を除いて，ブロッキングルーチンはすべて有限の時間内に完了しなければならない．

ノンブロッキングデータアクセスルーチンはノンブロッキング1対1通信から以下のプログレスルールを継承する．つまり，ノンブロッキング書き込みは最終的にマッチする受信が発行されるノンブロッキング送信と同等であり，ノンブロッキング読み込みは最終的にマッチする送信が発行されるノンブロッキング受信と同等である．

最後に，実装では，集団呼び出しに関連するグループ内のすべてのプロセスでルーチンの呼び出しが完了するまで，自由に集団的ルーチンのプログレスを遅延させることができる．グループ内のすべてのプロセスでルーチンの呼び出しが完了したら，同等の非集団的ルーチンのプログレスルールに従う必要がある．

13.6.4 集団的ファイル操作

集団的ファイル操作には，集団通信操作と同じ制限が適用される．詳細は，193ページの第5.12節で規定した意味論を参照すること．

集団的ファイル操作はファイルを開くために使用されたコミュニケータの複製に対して集団的であり，この複製のコミュニケータはファイルハンドラ引数によって暗黙的に指定される．特に規定がない限り，集団的ルーチンの他の引数にプロセスごとに異なる値を渡すことができる．

13.6.5 型の一致

入出力のための型の一致規則は通信のための型の一致規則に倣っている．ただし，例外が1つあり，etypeがMPI_BYTEの場合，これはデータアクセス操作の任意のdatatypeに一致する．一般的に，書き込まれたデータ項目のetypeはその項目を読み込むためのetypeと一致する必要があり，各データアクセス操作では現在のetypeがデータアクセスバッファの型宣言にも一致していなければならない．

> **ユーザへのアドバイス** ほとんどの場合，MPI_BYTEをワイルドカードとして使用すると，MPIのファイルの相互運用性が低下する．ファイルの相互運用性は，アクセスするデータ型が正しく明示的に指定されている場合のみ，異機種のデータ表現間で自動変換を行える． （**ユーザへのアドバイス終わり**）

13.6.6 その他の明確化

入出力ルーチンが完了すれば，そのルーチンに引数として渡された不可視オブジェクトを安全に開放することができる．例えば，MPI_FILE_OPENで使用されるcommとinfo，またはMPI_FILE_SET_VIEWで使用されるetypeとfiletypeは，ファイルへのアクセスに影

響を及ぼすことなく解放することができる．ノンブロッキングルーチンおよびスプリット集団操作の場合，引数として渡されたデータバッファを安全に再使用するには，その前に操作が完了していなければならない．

通信の場合と同様，データ型をファイル操作やデータアクセス操作で使用できるようにするためには，事前にコミットしておく必要がある．例えば，etypeとfiletypeはMPI_FILE_SET_VIEWを呼び出す前にコミットしておく必要があり，datatypeはMPI_FILE_READまたはMPI_FILE_WRITEを呼び出す前にコミットしておく必要がある．

13.6.7 MPI_Offset型

MPI_Offsetは，MPIでサポートされる最大のファイルのサイズ（バイト単位）を表現できる大きさを備えた整数型である．変位とオフセットも，常にMPI_Offset型の値として指定される．

Fortran言語では，対応する整数はmpif.hおよびmpiモジュールで定義された，MPI_OFFSET_KINDKINDの整数型である．

KINDパラメータをサポートしていないFortran 77言語環境では，MPI_Offset引数を適切なサイズの整数型として宣言する必要がある．MPI_Offsetにおける言語間の相互運用性は，アドレスの場合と同様である（519ページの第16.3節を参照）．

13.6.8 論理的なファイル配置と物理的なファイル配置

MPIは，仮想ファイル構造（ビュー）内でデータがどのように配置されるかを指定するが，そのファイル構造が1つ以上のディスクにどのように格納されるかを指定するわけではない．物理的なファイル構造の指定が避けられているのは，ディスクへのファイルのマッピングはシステム固有のもので，ファイルの配置に対する特定の制御はプログラムの可搬性を制限するためである．しかし，ファイルの配置を最適化するのにある情報が必要になりうる場合がある．この情報は，ファイルの生成時に*info*を使用して**ヒント**として渡すことができる（415ページの第13.2.8節を参照）．

13.6.9 ファイルのサイズ

ファイルのサイズは，現在のファイル終端の後に書き込むことにより，増加することがある．サイズは，MPI_FILE_SET_SIZEなどの**MPIサイズを変更する**ルーチンを呼び出すことによっても変更される．サイズを変更するルーチンを呼び出しても，必ずしもファイルのサイズが変更されるとは限らない．例えば，現在のサイズより小さいサイズを指定してMPI_FILE_PREALLOCATEを呼び出した場合，サイズは変更されない．

最近のサイズを変更するルーチンの呼び出しの後，またはこのようなルーチンが呼び出されていない場合はMPI_FILE_OPENの後にファイルに書き込まれたバイトの集合を考えてみる．**最後位バイト**を，その集合内の最大の変位を持つバイトとする．ファイルのサイズは，以下のうちの大きい方となる．

- 最後位バイトの変位+1
- サイズを変更するルーチンまたはMPI_FILE_OPENが戻った直後のサイズ

一貫性の意味論を適用すると，MPI_FILE_SET_SIZEおよびMPI_FILE_PREALLOCATEの呼び出しはファイルへの書き込み（古いファイルサイズと新しいファイルサイズの間にある変位のバイトにアクセスする操作とコンフリクトする）と考えられ，MPI_FILE_GET_SIZEはファイルの読み取り（ファイルへのすべてのアクセスと重複する）と考えられる．

ユーザへのアドバイス 集団的ルーチンMPI_FILE_SET_SIZEおよびMPI_FILE_PREALLOCATEを含む操作のシーケンスは書き込みシーケンスである．そのため，455ページの第13.6.1節に規定された条件が満たされていない場合，非アトミックモードでの逐次一貫性は保証されない． （**ユーザへのアドバイス終わり**）

ファイルポインタ更新の意味論（ファイルポインタは，アクセスされた量だけ更新される）は，ファイルのサイズの変更が逐次に一貫性がある場合にのみ保証される．

ユーザへのアドバイス 以下の例を考えてみる．100バイトのファイルに対して，別々のプロセスから2つの操作を行う．1つはMPI_FILE_READによる10バイトの読み取りで，もう1つはMPI_FILE_SET_SIZEによる0バイトの設定である．ユーザがこれらの2つの操作に対して逐次一貫性を適用しない場合，アクセスされる量が0バイトであっても，要求された量（10バイト）だけファイルポインタが更新されることがある． （**ユーザへのアドバイス終わり**）

13.6.10 例

この節の例は，MPIの一貫性と意味論の保証の，実際の使われ方を説明する． これらの例は以下のアクセスを説明する．

- 1つの集団的オープンで取得された複数のファイルハンドルでのコンフリクトするアクセス
- 2つの別の集団的オープンで取得されたファイルハンドルでのすべてのアクセス

コンフリクトするアクセスに対して一貫性を得るための最も簡単な方法は，アトミックモードを設定することにより，逐次一貫性を得ることである．以下のコードで，プロセス1は0個または10個の整数を読み取る．後者の場合，bのすべての要素は5となる．非アトミックモードが設定されている場合，読み込みの結果は未定義となる．

```
/* Process 0 */
int  i, a[10] ;
int  TRUE = 1;

for ( i=0;i<10;i++)
```

```
   a[i] = 5 ;

MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh0, TRUE ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status) ;
/* MPI_Barrier( MPI_COMM_WORLD ) ; */



/* Process 1 */
int  b[10] ;
int  TRUE = 1;
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh1, TRUE ) ;
/* MPI_Barrier( MPI_COMM_WORLD ) ; */
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status) ;
```

ユーザは，例えばMPI_BARRIERの呼び出しなどで時間的順序を設定することにより，プロセス0の書き込みがプロセス1の読み込みより前に行われることを保証することができる．

> **ユーザへのアドバイス** 時間的順序を設定するために，MPI_BARRIER以外のルーチンを使用することができる．上記の例では，プロセス0でMPI_SENDを使用して0バイトのメッセージを送信し，プロセス1でMPI_RECVを使用して受信することができる．（**ユーザへのアドバイス終わり**）

または，非アトミックモードを設定して一貫性を与えることもできる．

```
/* Process 0 */
int  i, a[10] ;
for ( i=0;i<10;i++)
   a[i] = 5 ;

MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh0 ) ;



/* Process 1 */
int  b[10] ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_sync( fh1 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
```

以下の理由により，“sync-barrier-sync”の構成物が必要となる．

- barrierにより，プロセス0の書き込みがプロセス1の読み込みより前に行われることを保証する．
- 最初のsyncにより，すべてのプロセスによって書き込まれたデータがストレージデバイスに転送されるよう保証する．
- 2番目のsyncにより，ストレージデバイスに転送されたすべてのデータがすべてのプロセスで検知可能になることを保証する（これは，この例のプロセス0には影響しない）

以下のプログラムでは，各プロセスに対して一見不必要と思われる2つめの“sync”を削除し，一貫性を得るための誤った試みを示している．

```
/* ----------------  THIS EXAMPLE IS ERRONEOUS --------------- */
/* Process 0 */
int  i, a[10] ;
for ( i=0;i<10;i++)
   a[i] = 5 ;

MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;



/* Process 1 */
int  b[10] ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;

/* ----------------  THIS EXAMPLE IS ERRONEOUS --------------- */
```

上記のプログラムは 集団操作の順序入れ替えを禁止する MPI規則にも違反し，MPI_FILE_SYNCがブロッキングされる実装でデッドロックが発生する．

ユーザへのアドバイス MPI_FILE_SYNCは一部の実装では時間軸で同期する関数として実装されることがある． このような実装を使用する場合，上記の“sync-barrier-sync”を1つの“sync”に置き換えることができる． MPI_FILE_SYNCで時間軸での同期が行われない実装でこのようなコードを使用した結果は未定義となる． （**ユーザへのアドバイス終わり**）

非同期入出力

非同期入出力操作の動作は，同期入出力操作のための上記の規則を適用することで決まる．

以下の例はすべて既存のファイル“myfile”にアクセスする．myfileの10番目のワードにはもともとは整数の2が入っている．それぞれの例では10番目のワードが書き込まれ，読み取られる．

最初に，以下の部分コードを考えてみる．

```
int a = 4, b, TRUE=1;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ;   Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Waitall(2, reqs, statuses) ;
```

非同期データアクセス操作の場合，MPIはアクセスが非同期データアクセスルーチンの呼び出しと対応する要求完了ルーチンからの戻りの間の任意の時点で行われることを規定している．そのため，書き込みの前の読み込みの実行，または読み込みの前の書き込みの実行はいずれもプログラムの順序と同じである．アトミックモードが設定されている場合，MPIにより逐次一貫性が保証され，プログラムは2または4をbに読み込む．アトミックモードが設定されていない場合，逐次一貫性は保証されず，データアクセスのコンフリクトにより，プログラムで2と4以外の値が読み込まれることがある．

同様に，以下の部分コードではファイルアクセスの順序が規定されない．

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ;   Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_Wait(&reqs[1], &status) ;
```

アトミックモードが設定されている場合，2または4がbに読み込まれる．ここでも，MPIは非アトミックモードでの逐次一貫性を保証しない．

逆に，以下の部分コードは

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
```

以下と同じ順序付けを行う．

```
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status ) ;
MPI_File_read_at(fh,  10, &b, 1, MPI_INT, &status ) ;
```

- 1つのファイルハンドルでの非同時操作は逐次に一貫性が取られる.
- この部分プログラムは操作の順序を指定している.

という理由により，MPIは両方の部分プログラムで値4がbに読み込まれることを保証する．この例ではアトミックモードを設定する必要はない.

同様の考察は以下のような形のコンフリクトするアクセスにも当てはまる.

```
MPI_File_write_all_begin(fh,...) ;
MPI_File_iread(fh,...) ;
MPI_Wait(fh,...) ;
MPI_File_write_all_end(fh,...) ;
```

一貫性と意味論に対する制限は以下に関連しないことを想起されたい.

```
MPI_File_write_all_begin(fh,...) ;
MPI_File_read_all_begin(fh,...) ;
MPI_File_read_all_end(fh,...) ;
MPI_File_write_all_end(fh,...) ;
```

同じファイルハンドルでのスプリット集団操作は重複させてはいけないからである（439ページの第13.4.5節を参照）.

13.7 入出力エラー処理

デフォルトでは通信エラーは致命的なもので，MPI_ERRORS_ARE_FATALはMPI_COMM_WORLDに関連付けれらたデフォルトのエラーハンドラである．通常，入出力エラーは通信エラーと比べると破滅的ではなく（「ファイルが見つからない」など），一般的にはこれらのエラーを捕捉し，処理を続行する．このため，MPIでは入出力のための追加のエラー機能を用意している.

> **ユーザへのアドバイス** MPIでは，誤ったMPI呼び出しが起こった後の計算の状態については規定していない．高品質な実装では入出力エラー処理機能をサポートしているため，入出力のための共通の手法を利用してプログラムを記述することができる．（**ユーザへのアドバイス終わり**）

通信と同様，各ファイルハンドルはそれに関連付けれらたエラーハンドラを備えている．MPIの入出力エラー処理ルーチンについては286ページの第8.3節で定義されている.

MPIが特定のファイルハンドルでのエラーの結果としてユーザ定義のエラーハンドラを呼び出す場合，ファイルエラーハンドラに渡される最初の2つの引数はファイルハンド

ルとエラーコードである．（MPI_FILE_OPEN，MPI_FILE_DELETEなどの）有効なファイルハンドラと関連していない入出力エラーの場合，エラーハンドラに渡される最初の引数はMPI_FILE_NULLである．

入出力エラー処理と通信エラー処理の違いには，もう1つ重要な点がある．デフォルトでは，ファイルハンドル用の定義済みのエラーハンドラはMPI_ERRORS_RETURNである．デフォルトのファイルエラーハンドラには2つの目的がある．1つめとして，新しいファイルハンドルが（MPI_FILE_OPENにより）生成されるとき，新しいファイルハンドル用のエラーハンドラには最初にデフォルトのエラーハンドラが設定される．2つめとして，エラーを処理する有効なファイルハンドルのない入出力ルーチン（MPI_FILE_OPEN，MPI_FILE_DELETEなど）はデフォルトのファイルエラーハンドラを使用する．デフォルトのファイルエラーハンドラを変更するには，MPI_FILE_SET_ERRHANDLERのfh引数としてMPI_FILE_NULLを指定する．デフォルトのエラーハンドラの現在の値を確認するには，MPI_FILE_GET_ERRHANDLERのfh引数としてMPI_FILE_NULLを渡す．

> **根拠** 通信の場合，デフォルトのエラーハンドラはMPI_COMM_WORLDから継承する．入出力では，デフォルトの特性を継承できるこのような「ルート」のファイルハンドルはない．新しいグローバルなファイルハンドルを作成するのではなく，デフォルトのエラーハンドラを加工し，MPI_FILE_NULLに付加されたようにする．（**根拠の終わり**）

13.8 入出力エラークラス

入出力ルーチンによって返される，実装依存のエラーコードを，表13.3に定義するようなエラークラスに変換することができる．

また，この章のルーチンの呼び出しは，MPI_ERR_TYPEなど他のMPIクラスでエラーを報告することがある．

13.9 例

13.9.1 スプリット集団入出力を使用したダブルバッファリング

この例では，計算と出力をオーバーラップさせる方法を示す．計算は関数compute_buffer()によって実行される．

```
/*============================================================================
 *
 * Function:            double_buffer
 *
 * Synopsis:
 *      void double_buffer(
 *              MPI_File fh,                            ** IN
 *              MPI_Datatype buftype,                   ** IN
 *              int bufcount                            ** IN
 *      )
 *
```

MPI_ERR_FILE	無効なファイルハンドル
MPI_ERR_NOT_SAME	全プロセスで同一でない集団引数，またはプロセスごとに違った順序で呼び出された集団的ルーチン
MPI_ERR_AMODE	MPI_FILE_OPENに渡されたamodeに関するエラー
MPI_ERR_UNSUPPORTED_DATAREP	MPI_FILE_SET_VIEWに渡されたサポートされていないdatarep
MPI_ERR_UNSUPPORTED_OPERATION	逐次アクセスのみをサポートするファイルでのシークなど，サポートされていない操作
MPI_ERR_NO_SUCH_FILE	ファイルが存在しない
MPI_ERR_FILE_EXISTS	ファイルが存在する
MPI_ERR_BAD_FILE	無効なファイル名（パス名が長すぎるなど）
MPI_ERR_ACCESS	権限なし
MPI_ERR_NO_SPACE	領域の不足
MPI_ERR_QUOTA	クオータの超過
MPI_ERR_READ_ONLY	読み取り専用のファイルまたはファイルシステム
MPI_ERR_FILE_IN_USE	ファイルがあるプロセスによって開かれているため，ファイル操作が完了できなかった
MPI_ERR_DUP_DATAREP	すでに定義されているデータ表現識別子がMPI_REGISTER_DATAREPに渡されたため，変換関数が登録できなかった
MPI_ERR_CONVERSION	ユーザ提供のデータ変換関数でエラーが発生した
MPI_ERR_IO	その他の入出力エラー

表 13.3: 入出力エラークラス

```
 * Description:
 *      Performs the steps to overlap computation with a collective write
 *      by using a double-buffering technique.
 *
 * Parameters:
 *      fh                  previously opened MPI file handle
 *      buftype             MPI datatype for memory layout
 *                          (Assumes a compatible view has been set on fh)
 *      bufcount            # buftype elements to transfer
 *--------------------------------------------------------------------------*/

/* this macro switches which buffer "x" is pointing to */
#define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))

void double_buffer(  MPI_File fh, MPI_Datatype buftype, int bufcount)
{

   MPI_Status status;         /* status for MPI calls */
   float *buffer1, *buffer2;  /* buffers to hold results */
   float *compute_buf_ptr;    /* destination  buffer */
                              /*   for computing */
   float *write_buf_ptr;      /* source for writing */
   int done;                  /* determines when to quit */

   /* buffer initialization */
   buffer1 = (float *)
                     malloc(bufcount*sizeof(float)) ;
   buffer2 = (float *)
                     malloc(bufcount*sizeof(float)) ;
   compute_buf_ptr = buffer1 ;   /* initially point to buffer1 */
   write_buf_ptr   = buffer1 ;   /* initially point to buffer1 */


   /* DOUBLE-BUFFER prolog:
    *   compute buffer1; then initiate writing buffer1 to disk
    */
   compute_buffer(compute_buf_ptr, bufcount, &done);
   MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);

   /* DOUBLE-BUFFER steady state:
    *  Overlap writing old results from buffer pointed to by write_buf_ptr
    *  with computing new results into buffer pointed to by compute_buf_ptr.
    *
    *  There is always one write-buffer and one compute-buffer in use
    *  during steady state.
    */
   while (!done) {
      TOGGLE_PTR(compute_buf_ptr);
      compute_buffer(compute_buf_ptr, bufcount, &done);
      MPI_File_write_all_end(fh, write_buf_ptr, &status);
      TOGGLE_PTR(write_buf_ptr);
      MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
   }

   /* DOUBLE-BUFFER epilog:
    *   wait for final write to complete.
    */
   MPI_File_write_all_end(fh, write_buf_ptr, &status);

```

```
   /* buffer cleanup */
   free(buffer1);
   free(buffer2);
}
```

13.9.2 部分配列ファイル型コンストラクタ

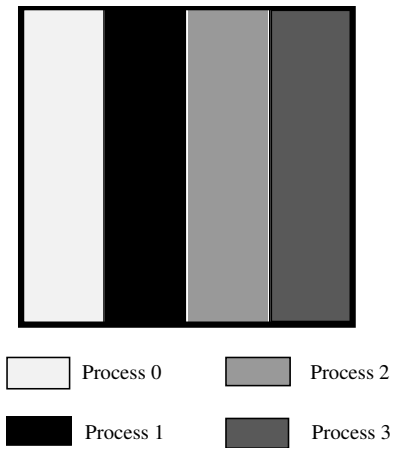

図 13.4: 配列のファイルの配置例

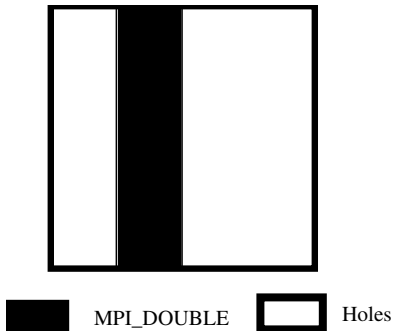

図 13.5: プロセス1のローカル配列のファイル型の例

4プロセスに分散された倍精度浮動小数点数の100×100の2次元配列を書き込むとする．ここで，各プロセスが25列のブロックを持つように分散する（プロセス0が列0～24を持ち，プロセス1が列25～49を持つなど．図13.4を参照）とする． 以下のC言語のプログラムを使用することで各プロセス用のファイル型を生成できる（98ページの第4.1.3節を参照）．

```
   double subarray[100][25];
   MPI_Datatype filetype;
   int sizes[2], subsizes[2], starts[2];
   int rank;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                         MPI_DOUBLE, &filetype);
```

または，Fortran言語の同等のコードは以下のようになる．

```
    double precision subarray(100,25)
    integer filetype, rank, ierror
    integer sizes(2), subsizes(2), starts(2)

    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
    sizes(1)=100
    sizes(2)=100
    subsizes(1)=100
    subsizes(2)=25
    starts(1)=0
    starts(2)=rank*subsizes(2)

    call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
              MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,       &
              filetype, ierror)
```

生成されるファイル型は，ファイルのうちプロセスの部分配列に収まる部分を表現する．また他プロセスが担当する領域を穴として表現する． 図13.5に，プロセス1用に生成されたファイル型を示す．

第14章

プロファイリングインターフェイス

14.1 要求仕様

MPIプロファイリングインターフェイス仕様を満たすために，MPI関数の実装は次のとおりでなければならない.

1. マクロ定義が許可されている関数（第2.6.5節を参照）を除くMPIで定義されている全ての関数に対するメカニズムを用意すること．このために，C言語およびFortran言語では，各MPI関数用のプリフィックスPMPI_を持つ代替のエントリポイント名が必要となる．C++言語のプロファイリングインターフェイスについては第16.1.10節で説明する．マクロとして実装されたルーチンの場合，PMPI_バージョンを用意して想定通りに機能することが求められるが，リンク時にMPI_バージョンをユーザ定義のバージョンに置き換えることはできない.

2. 置き換えないMPI関数があっても，名前衝突を引き起こさずに実行形式のイメージにリンクできることを保証すること.

3. 異なる言語呼び出し形式間で互いに階層関係がある場合には，MPIインターフェイスに対する異なる言語呼び出し形式間の実装を明文化すること．すなわち，それぞれの呼び出し形式に対しプロファイリングインターフェイスを実装しなければいけないのか，または最下層のルーチンに対してのみプロファイリングインターフェイスを実装することで簡略化できるのかを，プロファイラ開発者が分かるようにしなければならない.

4. 階層化アプローチで，異なる言語の呼び出し形式を実装する場合（例えば，Fortran言語の呼び出し形式がC言語の実装を呼び出す「ラッパー」関数の集合である場合），これらのラッパー関数をライブラリの他の部分から分離可能なようにすること.

   これは，分離したプロファイリングライブラリを正しく実装するために必要である．なぜなら（少なくともUnixのリンカにおいては）プロファイリングライブラリが期待通りに動作するためにはラッパー関数を含んでいなければならないからであ

る．この要求仕様により，プロファイリングライブラリを構築する人は，オリジナルのMPIライブラリからこれらの関数を抽出したり，他の不要なコードを記述することなくそれらをプロファイリングライブラリに追加することができる．

5. MPIライブラリに無操作ルーチンMPI_PCONTROLを用意すること．

14.2 考察

MPIプロファイリングインターフェイスの目的は，プロファイリング（および他の類似の）ツールの作成者が，異なるマシン上で各自のコードとMPI実装とのインターフェイスを比較的容易に作成できるようにすることである．

MPIには多くの異なる実装があるがマシンに依存しない標準であるので，MPI用プロファイリングツールの作成者が，ある特定のマシンに実装されたMPIのソースコードを参照すると考えるのは合理的ではない．したがって，そのようなツールの実装者が実装の下層部分を参照することなく，期待される性能情報を集められるようにするメカニズムを用意する必要がある．

このようなインターフェイスは，エンドユーザがMPIに魅力を感じるためには重要であると思われる．さまざまなツールが利用できることは，ユーザがMPI標準に魅力を感じるようになるための大きな要素だからである．

プロファイリングインターフェイスは，インターフェイスにすぎない．使用法については何も示していない．したがって，インターフェイスを通じてどのような情報を集めるのか，また集められた情報をどのように保存するのか，フィルタをかけるのか，表示するのかについては規定していない．

このインターフェイスを開発することになった最初の動機はプロファイリングツールの実装を可能にしたいということであったが，上で規定したようなインターフェイスが他の目的，例えば複数のMPI実装間を「インターネットワーキング」で接続することが有用であると証明される可能性があることは明らかである．定義したものは全てインターフェイスなので，有用であるならばどこに使われても異論はない．

ここで取り扱っている問題は実行形式のイメージを構築する方法と密接に関わっており，それはマシンが異なれば大きく異なる可能性がある．そのため，以下に挙げる例は，全てMPIプロファイリングインターフェイスの目的を実装する手段の1つとして取り扱うべきである．実装に対する実際の要求については，上の「要求仕様」の節で詳述しており，本章の残りの部分ではこれらの要求仕様の正当性と考察について記述する．

以下の例では，Unixシステム上で要求を満たす実装を構築する1つの方法を紹介する（同じように有効なものは他にもあると考えられる）．

14.3 設計の論理

MPI実装が上記の要求仕様を満たしている場合，プロファイリングシステムの実装者はユーザプログラムからのMPI関数呼び出しを全て捕捉することができる．そこで，下

層のMPI実装（名前シフトエントリポイントを通じて）を呼び出す前に必要な情報を集めて，望む結果を得ることができる．

14.3.1 プロファイリングの各種制御

ユーザコードからプロファイラを実行時に動的に制御できるようにする明示的な要求仕様である．これは通常，（少なくとも）次の目的のためである．

- 計算の状態に応じてプロファイリングを許可，禁止する
- 計算処理中の非クリティカルな時点でトレースバッファをフラッシュする
- ユーザイベントをトレースファイルに追加する

これらの要求はMPI_PCONTROLを使用することで満たされる．

MPI_PCONTROL(level, . . .)

IN	level	プロファイリングのレベル

```
int MPI_Pcontrol(const int level, ...)
MPI_PCONTROL(LEVEL)
    INTEGER LEVEL
{void MPI::Pcontrol(const int level, ...)   （廃止された呼び出し形式, 第15.2節を参照）
             }
```

MPIライブラリ自体はこのルーチンを使用せず，ユーザコードへ即座に戻るだけである．しかし，このルーチンの呼び出しが存在していれば，ユーザ側でプロファイリングパッケージを明示的に呼び出すことができる．

MPIはプロファイリングコードの実装についてはなにも規定しないので，MPI_PCONTROL呼び出しの意味を正確に仕様書に明記することはできない．この曖昧さは関数に渡す引数の個数やそのデータ型にも及んでいる．

しかし，異なるプロファイリングライブラリ間でユーザコードの移植性にいくつかのレベルを持たせるために，特定のlevel値に以下の意味を持たせる．

- `level==0` プロファイリングが利用不可
- `level==1` 通常デフォルトの詳細なレベルでプロファイリングが利用可能である
- `level==2` プロファイルバッファをフラッシュする（これはいくつかのプロファイラでは無操作の場合がある）
- 上記以外の`level`の値は， プロファイルライブラリで定義された処理と追加の引数を持つ．

さらに，MPI_INITを呼び出した後のデフォルトの状態が，通常のデフォルトレベルでプロファイリングを利用できるということも必要となる（つまり，MPI_PCONTROLが引数1で呼び出されたのと同様である）．これにより，ユーザはソースコードをまったく修

正する必要なしにプロファイリングライブラリとリンクし，プロファイル出力を得ることができる．

標準MPIライブラリでは無操作のMPI_PCONTROLを規定しているため，そのソースコードを修正してより詳細なプロファイリング情報を取得するようにできるが，それでも全く同じコードを標準MPIライブラリに対してリンクすることができる．

14.4 例

14.4.1 プロファイラの実装

プロファイラを使ってMPI_SEND関数が送信したデータの総量と，この関数が処理に要した総時間を計測するとする．これは次のようにして得られることは明らかである．

```
static int totalBytes = 0;
static double totalTime = 0.0;

int MPI_Send(void* buffer, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
{
   double tstart = MPI_Wtime();       /* Pass on all the arguments */
   int extent;
   int result    = PMPI_Send(buffer,count,datatype,dest,tag,comm);

   MPI_Type_size(datatype, &extent);  /* Compute size */
   totalBytes += count*extent;

   totalTime  += MPI_Wtime() - tstart;         /* and time          */

   return result;
}
```

14.4.2 MPIライブラリの実装

UnixシステムでMPIライブラリがC言語で実装されているものはいろいろなオプションがあり，そこでもっとも明白なもののうち2つをここに紹介する．どちらがよいかはリンカとコンパイラがweakシンボルをサポートしているかどうかによる．

weakシンボルのあるシステム

コンパイラとリンカがweak外部シンボルをサポートしている場合（例えば，Solaris 2.x，その他のsystem V.4マシン），次のように#pragma weakを使用することで，必要なライブラリが1つのみとなる．

```
#pragma weak MPI_Example = PMPI_Example

int PMPI_Example(/* appropriate args */)
{
    /* Useful content */
}
```

この#pragmaの効果として，外部シンボルMPI_Exampleがweak定義として定義される．このことは，リンカでは（例えば，プロファイリングライブラリで）シンボルの定義が他にあってもエラーにならず，他の定義が存在しない場合には，リンカはweak定義を使用するということを意味する．

weakシンボルのないシステム

weakシンボルが存在しない場合には，解決策の1つとして，次のようにC言語のマクロプリプロセッサを使用するという方法が考えられる．

```
#ifdef PROFILELIB
#    ifdef __STDC__
#        define FUNCTION(name) P##name
#    else
#        define FUNCTION(name) P/**/name
#    endif
#else
#    define FUNCTION(name) name
#endif
```

この時，ライブラリ内のユーザに見える関数はそれぞれ，次のように宣言できる．

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
    /* Useful content */
}
```

PROFILELIBマクロシンボルの状態に従って，同じソースファイルをコンパイルして両方のバージョンのライブラリを作成することができる．

MPI関数の組み込みが一度に 1 つ行われるように標準MPIライブラリを構築する必要がある．これは，それぞれの外部関数を別々のファイルからコンパイルしなければならないことを意味しうるため，面倒な要求である．しかし，プロファイリングライブラリの作成者が，捕捉したいMPI関数とそれ以外の標準MPIライブラリを使用する関数への参照のみを定義する必要があるようにするために必要である．そのため，リンク段階ではこのようになる．

```
% cc ... -lmyprof -lpmpi -lmpi
```

ここで，libmyprof.aにはいくつかのMPI関数を捕捉するプロファイラ関数が含まれている．libpmpi.aには，「名前シフト」したMPI関数が含まれており，libmpi.aにはMPI関数の標準の定義が含まれている．

14.4.3 厄介な問題

多重カウント

MPIライブラリの一部は，それ自体より基本的なMPI関数を使用して実装できるので（例えば，1対1通信を使用して実装された集団通信のポータブルな実装），プロファイリング関数から呼ばれたMPI関数の中からプロファイリング関数を呼び出す可能性がある．

このようなことがあると，内側のルーチンで費やされる時間を「二重カウント」することになる可能性もある．この効果は状況によっては実際に有用な場合もあるので（例えば，「集団関数から呼び出されたときに1対1ルーチンでどれだけの時間が費やされるか」という疑問に答えることもできる）MPIライブラリの作成者がこれを解決することを妨げるような制約を強制しないようにした．したがって，プロファイリングライブラリの作成者はこの問題に注意し，作成者自身がこの問題を防ぐようにしなければならない．単一スレッドの世界では，すでにプロファイリングルーチンに入っているかどうかを示す静的変数をプロファイリングコード内で使用して簡単に実現できる．マルチスレッド環境では，これはより複雑なものになる（記録された回数の意味を考えるとよい）．

リンカの奇妙なところ

Unixのリンカは伝統的に1パスで実行する．この結果，関数が必要な場合にライブラリを走査して，ライブラリの関数がイメージに取り込まれるだけである．weakシンボルや同じ関数の複数の定義があった場合，これは奇妙な（そして予期しない）結果を生じる場合がある．

例えば，C言語用の実装の上にラッパー関数をかぶせてFortran言語の呼び出し形式を実現したMPI実装を考えてみる．プロファイルライブラリの作成者はC言語の呼び出し形式のプロファイル関数を用意するだけで十分と考えている．Fortran言語で最終的にこれらの関数が呼び出され，またラッパー関数のコストは大したものでないと考えているためである．しかし，ラッパー関数がプロファイリングライブラリの中にない場合でも，プロファイリングライブラリを呼び出したときにプロファイル用のエントリポイントはどれも未定義にはならない．したがって，プロファイリングコードはどれもイメージの中には取り込まれない．標準MPIライブラリを走査するときに，Fortran言語のラッパー関数は解決され，MPI関数の基本バージョン利用される．全体として，コードは正常にリンクされるが，プロファイルされないという結果になる．

この問題を解決するために，Fortran言語用ラッパー関数をプロファイル用ライブラリの中に取り込むようにしなければならない．これは，ラッパー関数を基本MPIライブラリの残り部分から分離できるように要求することで実現できる．こうすることで，`ar`を使用して基本ライブラリから抽出し，プロファイリングライブラリの中に取り込むことができる．

14.5 複数レベルの捕捉

ここで述べた方法は，各MPI関数について単一の代替名しか用意していないため，プロファイリング関数の入れ子を直接にはサポートしていない．複数レベルの呼び出し捕捉を可能にする実装が検討されたが，以下の欠点を持たない実装を構築することはできなかった．

- 特定の実装言語を仮定する
- プロファイリングを行わないときでもランタイムコストがかかる

MPIの目的の1つは効率がよく低レイテンシの実装を可能にすることであり，特定の実装言語を要求することは標準の役目でないことから，上述の方式を受け入れることに決定した.

しかし，ユーザが呼び出す関数は下層のMPI 関数を呼び出す前に様々な異なったプロファイリング用関数を呼び出すかもしれないので，上記の方式を使用して複数レベルに対応するシステムを実装することは可能である.

残念なことにこのような実装では，上述の単一レベル実装で必要とされる以上に，異なるプロファイリングライブラリ同士で密接に協調することが求められうる.

第15章

廃止された関数

15.1 MPI-2.0から廃止された関数

以下の関数はMPI-2.0で廃止され，MPI_TYPE_CREATE_HVECTORに置き換わっている．廃止された関数の言語非依存の定義とC言語呼び出し形式は，関数名以外は新しい関数のものと同じである．Fortran言語の呼び出し形式のみが異なる．

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)

IN	count	ブロック数（非負の整数型）
IN	blocklength	各ブロック内の要素の数（非負の整数型）
IN	stride	各ブロックの開始点の間のバイト数（整数型）
IN	oldtype	古いデータ型（ハンドル）
OUT	newtype	新しいデータ型（ハンドル）

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_TYPE_CREATE_HINDEXEDに置き換わっている．廃止された関数の言語非依存の定義とC言語呼び出し形式は，関数名以外は新しい関数のものと同じである．Fortran言語の呼び出し形式のみが異なる．

MPI_TYPE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN	count	ブロック数 – また，配列 array_of_displacements および array_of_blocklengths のエントリの数（非負の整数型）
IN	array_of_blocklengths	各ブロックの要素の数（非負の整数型の配列）
IN	array_of_displacements	各ブロックのバイト変位（整数型の配列）
IN	oldtype	古いデータ型（ハンドル）
OUT	newtype	新しいデータ型（ハンドル）

```
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)
MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_TYPE_CREATE_STRUCTに置き換わっている．廃止された関数の言語非依存の定義とC言語呼び出し形式は，関数名以外は新しい関数のものと同じである．Fortran言語の呼び出し形式のみが異なる．

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	ブロック数（整数型）（非負の整数型）– また，配列 array_of_types, array_of_displacements, array_of_blocklengthsのエントリの数
IN	array_of_blocklength	各ブロックの要素の数（非負の整数型の配列）
IN	array_of_displacements	各ブロックのバイト変位（整数型の配列）
IN	array_of_types	各ブロックの要素の型（データ型オブジェクトのハンドルの配列）
OUT	newtype	新しいデータ型（ハンドル）

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
MPI_Datatype *newtype)
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    ARRAY_OF_TYPES(*), NEWTYPE, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_GET_ADDRESSに置き換わっている．廃止された関数の言語非依存の定義とC言語呼び出し形式は，関数名以外は新しい関数のものと同じである．Fortran言語の呼び出し形式のみが異なる．

MPI_ADDRESS(location, address)

IN	location	呼び出し元のメモリ内の領域（選択型）
OUT	address	領域のアドレス（整数型）

```
int MPI_Address(void* location, MPI_Aint *address)
MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_TYPE_GET_EXTENTに置き換わっている．

MPI_TYPE_EXTENT(datatype, extent)

IN	datatype	データ型（ハンドル）
OUT	extent	データ型の範囲（整数型）

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
    INTEGER DATATYPE, EXTENT, IERROR
```

データ型の範囲を返す．データ型の定義については，107ページを参照すること．

以下の2つの関数は，データ型の下限と上限を検出するのに使用できる．

MPI_TYPE_LB(datatype, displacement)

IN	datatype	データ型（ハンドル）
OUT	displacement	オリジンからのバイト単位での下限の変位（整数型）

```
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

MPI_TYPE_UB(datatype, displacement)

IN	datatype	データ型（ハンドル）
OUT	displacement	オリジンからのバイト単位での上限の変位（整数型）

```
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_COMM_CREATE_KEYVALに置き換わっている． 廃止された関数の言語非依存の定義は新しい関数のものと同じであるが，関数名と， C言語/Fortran言語の相互運用時の動作は異なる．529ページの 第16.3.7節を参照すること．この言語の呼び出し形式は変更されている．

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

IN	copy_fn	keyvalのコピーコールバック関数
IN	delete_fn	keyvalの削除コールバック関数
OUT	keyval	将来のアクセス用のキー値（整数型）
IN	extra_state	コールバック関数用の特別な状態

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
*delete_fn, int *keyval, void* extra_state)
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR
```

copy_fn関数はMPI_COMM_DUPによってコミュニケータが複製されるときに呼び出される．copy_fnはMPI_Copy_function型とする必要があり，以下のように定義される．

```
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                              void *extra_state, void *attribute_val_in,
                              void *attribute_val_out, int *flag)
```

この関数のFortran言語での宣言は以下のようになる．

```
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG
```

copy_fnはC言語またはFortran言語でMPI_NULL_COPY_FN またはMPI_DUP_FNとして指定することができ，MPI_NULL_COPY_FNはflag = 0とMPI_SUCCESS を返すだけの関数である．MPI_DUP_FNはflag = 1を設定し，attribute_val_outに値attribute_val_inを返し，MPI_SUCCESSを返す単純なコピー関数である． MPI_NULL_COPY_FNとMPI_DUP_FNも廃止されたため，注意すること．

copy_fnと類似したコールバック削除関数は，以下のように定義される． delete_fn関数の呼び出しは，MPI_COMM_FREEによって コミュニケータが削除される場合，またはMPI_ATTR_DELETEを明示的に呼び出す場合に行われる．delete_fnは MPI_Delete_function型とする必要があり，以下のように定義される．

```
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
void *attribute_val, void *extra_state);
```

この関数のFortran言語での宣言は以下のようになる．

```
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

delete_fnはC言語またはFortran言語でMPI_NULL_DELETE_FNとして指定することができ，MPI_NULL_DELETE_FNはMPI_SUCCESSを返すだけの関数である．MPI_NULL_DELETE_FNも廃止されたため，注意すること．

以下の関数はMPI-2.0で廃止され，MPI_COMM_FREE_KEYVALに置き換わっている．廃止された関数の言語非依存の定義は，関数名以外は新しい関数のものと同じである．この言語の呼び出し形式は変更されている．

MPI_KEYVAL_FREE(keyval)

INOUT	keyval	整数のキー値を解放する（整数型）

```
int MPI_Keyval_free(int *keyval)
MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_COMM_SET_ATTRに置き換わっている．廃止された関数の言語非依存の定義は，関数名以外は新しい関数のものと同じである．この言語の呼び出し形式は変更されている．

MPI_ATTR_PUT(comm, keyval, attribute_val)

INOUT	comm	属性を結びつける先となるコミュニケータ（ハンドル）
IN	keyval	MPI_KEYVAL_CREATEによって返されるキー値（整数型）
IN	attribute_val	属性値

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_COMM_GET_ATTRに置き換わっている．廃止された関数の言語非依存の定義は，関数名以外は新しい関数のものと同じである．この言語の呼び出し形式は変更されている．

MPI_ATTR_GET(comm, keyval, attribute_val, flag)

IN	comm	属性を結びつける先となるコミュニケータ（ハンドル）
IN	keyval	キー値（整数型）
OUT	attribute_val	属性値（flag = falseでない場合）
OUT	flag	属性値が抽出された場合は`true`，キーに属性が関連付けられていない場合は`false`

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG
```

以下の関数はMPI-2.0で廃止され，MPI_COMM_DELETE_ATTRに置き換わっている．廃止された関数の言語非依存の定義は，関数名以外は新しい関数のものと同じである．この言語の呼び出し形式は変更されている．

MPI_ATTR_DELETE(comm, keyval)

INOUT	comm	属性を結びつける先となるコミュニケータ（ハンドル）
IN	keyval	削除される属性のキー値（整数型）

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

以下の関数はMPI-2.0で廃止され，MPI_COMM_CREATE_ERRHANDLERに置き換わっている．廃止された関数の言語非依存の定義は，関数名以外は新しい関数のものと同じである．この言語の呼び出し形式は変更されている．

MPI_ERRHANDLER_CREATE(function, errhandler)

IN	function	ユーザ定義のエラー処理手順
OUT	errhandler	MPIエラーハンドラ（ハンドル）

```
int MPI_Errhandler_create(MPI_Handler_function *function,
MPI_Errhandler *errhandler)
MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
```

MPI例外ハンドラとして使用するユーザルーチンfunctionを登録する．登録された例外ハンドラのハンドルをerrhandlerに返す．

C言語では，このユーザルーチンはMPI_Handler_function型とする必要があり，以下のように定義される．

```
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

第1引数は使用するコミュニケータであり，第2引数は返すべきエラーコードである．

Fortran言語では，このユーザルーチンは以下の形式とする必要がある．

```
SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE)
   INTEGER COMM, ERROR_CODE
```

以下の関数はMPI-2.0で廃止され，MPI_COMM_SET_ERRHANDLERに置き換わっている．廃止された関数の言語非依存の定義は，関数名以外は新しい関数のものと同じである．この言語の呼び出し形式は変更されている．

MPI_ERRHANDLER_SET(comm, errhandler)

INOUT	comm	エラーハンドラを設定する対象となるコミュニケータ（ハンドル）
IN	errhandler	コミュニケータ用の新しいMPIエラーハンドラ（ハンドル）

```
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

呼び出しプロセスで，コミュニケータcommに新しいエラーハンドラerrorhandlerを関連付ける．エラーハンドラは必ずコミュニケータに関連付けられることに注意すること．

以下の関数はMPI-2.0で廃止され，MPI_COMM_GET_ERRHANDLERに置き換わっている．廃止された関数の言語非依存の定義は，関数名以外は新しい関数のものと同じである．この言語の呼び出し形式は変更されている．

MPI_ERRHANDLER_GET(comm, errhandler)

IN	comm	エラーハンドラの取得元となるコミュニケータ（ハンドル）
OUT	errhandler	現在，コミュニケータに関連付けられているMPIエラーハンドラ（ハンドル）

```
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

現在，コミュニケータcommに関連付けられているエラーハンドラ（のハンドル）errhandlerを返す．

15.2 MPI-2.2から廃止された関数

C++言語の呼び出し形式は全体的に廃止された．

根拠 C++言語の呼び出し形式はC言語の呼び出し形式に最小限の機能を付加したものであるが， MPIの仕様に合わせるためのメンテナンスに大きな労力を要する．事実上，C++言語の 呼び出し形式はC言語の呼び出し形式と1対1対応であるため，既存のC++言語のMPIアプリケーションを MPIのC言語の呼び出し形式に変更することは比較的容易である．加えて， C++言語のクラスライブラリの機能を提供する，サードパーティのパッケージも利用可能である（ここでのクラスライブラリの機能とは，MPIのC言語の呼び出し形式の直上に階層化されたC++言語固有の機能を指す）．このような機能は，C++言語のプログラマにとってより表現力が高く，かつ自然であるが，MPI仕様での標準化には適していない．（**根拠の終わり**）

以下の関数のtypedefは廃止されており，新しい名前に置き換わっている．typedefの名前を除き，関数のシグネチャはまったく同じである．名前だけが，他の関数のtypedefの名前のルールに揃えるために更新された．

廃止された名前	新しい名前
MPI_Comm_errhandler_fn	MPI_Comm_errhandler_function
MPI::Comm::Errhandler_fn	MPI::Comm::Errhandler_function
MPI_File_errhandler_fn	MPI_File_errhandler_function
MPI::File::Errhandler_fn	MPI::File::Errhandler_function
MPI_Win_errhandler_fn	MPI_Win_errhandler_function
MPI::Win::Errhandler_fn	MPI::Win:::Errhandler_function

第16章

言語別の呼び出し形式

16.1 C++言語

16.1.1 概要

C++言語の呼び出し形式は廃止された [1].

単なる言語の呼び出し形式の記述だけでなく，インターフェイスの設計で考慮すべきC++言語固有の問題がいくつかある．特に，C++言語では，言語固有の MPIの関数インターフェイスの設計だけでなく，オブジェクトとそのインターフェイスの設計にも関心を払う必要がある．幸運にも，MPIの設計はオブジェクトの概念に基づいて行われたため，自然なクラスの集合はすでにMPIの一部となっている．

MPI-2には関数の仕様の一部としてC++言語の呼び出し形式が含まれている． MPI-2ではMPI-1関数のC言語の呼び出し形式用に新しい名前が用意されている場合もある．この場合，C++言語の呼び出し形式は新しいC言語の名前と一致し， 廃止された名前のための呼び出し形式はない．

16.1.2 設計

MPI用のC++言語のインターフェイスは以下の基準に従って設計されている．

1. C++言語のインターフェイスは，MPIへの軽量の関数インターフェイスを備えたクラスの小さな集合で構成される．クラスは基本的なMPIオブジェクト型に基づいている（コミュニケータ，グループなど）．

2. MPIのC++言語の呼び出し形式はMPIに対する意味的に正しいインターフェイスを備えている．

3. 最大限の規模で， MPI関数用のC++言語の呼び出し形式はMPIクラスのメンバー関数とする．

[1] 訳者注: MPI-3 で削除された．

根拠 基本MPI型に対応する軽量のMPIオブジェクトの集合を用意することは，MPIの暗黙的なオブジェクトベースの設計に最も適していて，これらのオブジェクトのメソッドによりMPI機能を実現することができる．既存のC言語の呼び出し形式をC++言語のプログラムで使用することができるが，C++言語の表現力の多くは失われる．それに対して，包括的なクラスライブラリを使用するとより的確なプログラミングが行えるが，呼び出し形式は指定のMPI機能に対する直接的で明確なマッピングを提供する必要があるため，このようなライブラリはMPI用の言語の呼び出し形式としてはふさわしくない． （**根拠の終わり**）

.

16.1.3 MPI用のC++言語のクラス

MPIのクラス，定数，関数は全てMPI名前空間の有効範囲内で宣言される．そのため，C言語およびFortran言語で使用される “MPI_” プリフィックスの代わりに，MPI 関数では基本的に “MPI::” プリフィックスを使用する．

MPI名前空間のメンバーは，MPIによって暗黙的に使用されるオブジェクトに対応するクラスである．MPI名前空間の簡潔な定義とそのメンバクラスは以下のとおりである．

```
namespace MPI {
  class Comm                              {...};
  class Intracomm : public Comm           {...};
  class Graphcomm : public Intracomm      {...};
  class Distgraphcomm : public Intracomm {...};
  class Cartcomm  : public Intracomm      {...};
  class Intercomm : public Comm           {...};
  class Datatype                          {...};
  class Errhandler                        {...};
  class Exception                         {...};
  class File                              {...};
  class Group                             {...};
  class Info                              {...};
  class Op                                {...};
  class Request                           {...};
  class Prequest  : public Request        {...};
  class Grequest  : public Request        {...};
  class Status                            {...};
  class Win                               {...};
};
```

少数の派生クラスがあり，仮想継承は使用されないこのに注意を要する．

16.1.4 MPIのクラスメンバー関数

MPI用のC++言語の呼び出し形式を構成するメンバー関数のほか，C++言語のインターフェイスには（C++言語で必要となる）追加の関数がある．特に， C++言語のインターフェイスはコンストラクタとデストラクタ，代入演算子，比較演算子を備えていなければならない．

MPI用のC++言語の全ての呼び出し形式を付録A.4に示す．この呼び出し形式では，参照や constなど，C++言語の重要な機能を利用する．コンストラクション，デストラクション，コピー，代入，比較，言語間の運用性のための（全てのMPIメンバークラスに適用される）宣言も示す．

明記された場合を除いて，MPIメンバクラスのstaticでないメンバ関数（コンストラクタと代入演算子用のものを除く）は全て仮想関数である．

根拠 仮想メンバー関数を用意することは，継承のための設計の重要な部分である．仮想関数は実行時にバインディングでき，これによりライブラリのユーザはすでにライブラリに組み込まれているオブジェクトの動作を再定義することができる．これは性能を低下させる要因となるが，必要な処理である（仮想関数は呼び出しの前に見つける必要がある）．ただし，この性能低下が心配な場合は，コンパイル時に関数のバインディングを行うことができる． （**根拠の終わり**）

例 16.1 派生MPIクラスの例を示す．

```
class foo_comm : public MPI::Intracomm {
public:
  void Send(const void* buf, int count, const MPI::Datatype& type,
            int dest, int tag) const
  {
    // Class library functionality
    MPI::Intracomm::Send(buf, count, type, dest, tag);
    // More class library functionality
  }
};
```

実装者へのアドバイス 実装者は，特に階層的な実装の場合に．継承を使用するクラスライブラリで意図しない副作用が発生しないよう注意する必要がある．例えば，MPI_SENDまたはMPI_RECVを繰り返し呼び出すことによりMPI_BCASTを実装する場合， MPI_SENDまたはMPI_RECVを再定義する可能性のある派生コミュニケータクラスによってMPI_BCASTの動作を変更することはできない．MPI_BCASTの実装では基本のMPI::CommクラスのMPI_SEND（またはMPI_RECV）を明示的に使用する必要がある． （**実装者へのアドバイス終わり**）

16.1.5 意味

MPI用のC++言語の呼び出し形式を構成するメンバー関数の意味は，MPI関数の記述自体により規定される．ここでは，C++言語の呼び出し形式の一部でないC++言語のインターフェイスの部分の意味を規定する，本節では関数は各MPIクラスの各関数を列記する代わりに，MPI::⟨CLASS⟩ という型を使用してプロトタイプが規定され，⟨CLASS⟩ という用語は，明記した場合を除いて，有効なMPIクラス名（Groupなど）に置き換えることができる．

コンストラクション／デストラクション デフォルトのコンストラクタ／デストラクタのプロトタイプは以下のように規定される.

{ MPI::<CLASS>() （廃止された呼び出し形式, 第15.2節を参照）}
{ ~MPI::<CLASS>() （廃止された呼び出し形式, 第15.2節を参照）}

コンストラクションとデストラクションについて，不可視MPIユーザレベルオブジェクトはハンドルと同様に振舞う．MPI::Status以外の全てのMPIオブジェクト用デフォルトコンストラクタは対応するMPI::*_NULLハンドルを生成する．つまり，MPIオブジェクトのインスタンスの作成時，対応するMPI::*_NULLオブジェクトとの比較を行うとtrueが返される．デフォルトのコンストラクタは新しいMPI不可視オブジェクトを生成しない．一部のクラスでは，このためのメンバー関数Create()が利用できる．

例 16.2 以下の部分コード例では，テストはtrueを返し，メッセージがcoutに送信される.

```
void foo()
{
  MPI::Intracomm bar;

  if (bar == MPI::COMM_NULL)
    cout << "bar is MPI::COMM_NULL" << endl;
}
```

各MPIユーザレベルオブジェクトのデストラクタは対応するMPI_*_FREE関数を（存在するとしても） 呼び出さない.

根拠 以下の理由により，MPI_*_FREE関数は自動的に呼び出されない.

1. 自動的デストラクションとMPIクラスの浅いコピーの意味が相反する
2. MPIのモデルでは，メモリの割当と解放が実装ではなく，ユーザの責任となっている.
3. 破壊時にMPI_*_FREEを呼び出すと， 集団操作を引き起こすなどの意図しない副作用が発生する可能性がある（これはコピー，代入，コンストラクションの意味にも影響する）. 以下の例では， foo_commでもbar_commでも関数の終了時に自動的に MPI_*_FREEが呼び出されることは望ましくない.

```
void example_function()
{
  MPI::Intracomm foo_comm(MPI::COMM_WORLD), bar_comm;
  bar_comm = MPI::COMM_WORLD.Dup();
  // rest of function
}
```

（根拠の終わり）

コピー／代入 コピーコンストラクタと代入演算子のプロトタイプは以下のように規定される．

{ MPI::<CLASS>(const MPI::<CLASS>& data) （廃止された呼び出し形式, 第15.2節を参照）}
{ MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI::<CLASS>& data) （廃止された呼び出し形式, 第15.2節を参照）}

コピーと代入について，不可視MPIユーザレベルオブジェクトはハンドルと同様に振舞う．コピーコンストラクタはハンドルベースの（狭い）コピーを行う．MPI::Statusオブジェクトはこの規則の例外となる．これらのオブジェクトは代入とコピー構成のための深いコピーを行う．

> **実装者へのアドバイス** 各MPIユーザレベルオブジェクトは，値または参照により，実装固有の状態情報を格納する．MPIオブジェクトハンドルの代入とコピーでは，この値（または参照）を単にコピーするだけの場合がある． （**実装者へのアドバイス終わり**）

例 16.3 この例では，foo_comm用にMPI::Intracomm::Dup()は呼び出されない．オブジェクトfoo_comm は単にMPI::COMM_WORLDのエイリアスである．しかし，bar_commはMPI::Intracomm::Dup()の呼び出しにより生成されるため， foo_commとは別のコミュニケータである（そのため，MPI::COMM_WORLDとも異なる）．baz_commはbar_commのエイリアスとなる．bar_commまたはbaz_commのいずれかがMPI_COMM_FREEにより解放される場合，MPI::COMM_NULLに設定される．その他のハンドルの状態は不定であり，無効であり，必ずしもMPI::COMM_NULLが設定されるとは限らない．

```
MPI::Intracomm foo_comm, bar_comm, baz_comm;

foo_comm = MPI::COMM_WORLD;
bar_comm = MPI::COMM_WORLD.Dup();
baz_comm = bar_comm;
```

比較 比較演算子のプロトタイプは以下のように規定される．

{bool MPI::<CLASS>::operator==(const MPI::<CLASS>& data) const （廃止された呼び出し形式, 第15.2節を参照）}
{bool MPI::<CLASS>::operator!=(const MPI::<CLASS>& data) const （廃止された呼び出し形式, 第15.2節を参照）}

メンバー関数operator==()はハンドルが同じ内部MPIオブジェクトを参照する場合のみtrueを返し，それ以外の場合はfalse を返す．operator!=()はoperator==() の論理補完を返す．しかし，Statusクラスは下層のMPIオブジェクトのハンドルではないため， Statusインスタンスの比較は意味がない．そのため，operator==()およびoperator!=()関数はStatusクラスでは定義されない．

定数 定数はシングルトンオブジェクトで，constとして宣言される．グローバルに定義されたMPIオブジェクトが必ずしも定数とは限らない．例えば， MPI::COMM_WORLDとMPI::COMM_SELFはconstではない．

16.1.6 C++言語のデータ型

表16.1にC++言語の定義済みのMPIデータ型とその対応するC言語およびC++言語のデータ型の一覧を示し，表16.2にFortran言語の定義済みのMPIデータ型とその対応するFortran 77言語のデータ型の一覧を示す．表16.1.6にはその他の全てのMPIデータ型のためのC++言語の名前の一覧を示す．

MPI::BYTEおよびMPI::PACKEDは，それぞれ32ページの第3.2.2節と131ページの第4.2節に示されたMPI_BYTEおよびMPI_PACKEDと同じ制限に従う．

MPIのデータ型	C言語のデータ型	C++言語のデータ型
MPI::CHAR	char	char
MPI::SHORT	signed short	signed short
MPI::INT	signed int	signed int
MPI::LONG	signed long	signed long
MPI::LONG_LONG	signed long long	signed long long
MPI::SIGNED_CHAR	signed char	signed char
MPI::UNSIGNED_CHAR	unsigned char	unsigned char
MPI::UNSIGNED_SHORT	unsigned short	unsigned short
MPI::UNSIGNED	unsigned int	unsigned int
MPI::UNSIGNED_LONG	unsigned long	unsigned long int
MPI::UNSIGNED_LONG_LONG	unsigned long long	unsigned long long
MPI::FLOAT	float	float
MPI::DOUBLE	double	double
MPI::LONG_DOUBLE	long double	long double
MPI_CXX_BOOL		bool
MPI_CXX_COMPLEX		std::complex<float>
MPI_CXX_DOUBLE_COMPLEX		std::complex<double>
MPI_CXX_LONG_DOUBLE_COMPLEX		std::complex<long double>
MPI::WCHAR	wchar_t	wchar_t
MPI::BYTE		
MPI::PACKED		

表 16.1: MPIC言語およびC++言語の定義済みのデータ型のためのC++言語の名前と，それに対応するC言語とC++言語のデータ型

以下の表で，MPIの定義済みのデータ型のグループを定義する．

C言語の整数型:	MPI::INT, MPI::LONG, MPI::SHORT, MPI::UNSIGNED_SHORT, MPI::UNSIGNED, MPI::UNSIGNED_LONG, MPI::LONG_LONG, MPI::UNSIGNED_LONG_LONG, MPI::SIGNED_CHAR, MPI::UNSIGNED_CHAR

MPIのデータ型	Fortran言語のデータ型
MPI::INTEGER	INTEGER
MPI::REAL	REAL
MPI::DOUBLE_PRECISION	DOUBLE PRECISION
MPI::F_COMPLEX	COMPLEX
MPI::LOGICAL	LOGICAL
MPI::CHARACTER	CHARACTER(1)
MPI::BYTE	
MPI::PACKED	

表 16.2: MPI Fortran言語の定義済みのデータ型のためのC++言語の名前と，それに対応するFortran 77言語のデータ型

MPIのデータ型	説明
MPI::FLOAT_INT	C/C++言語のリデュース型
MPI::DOUBLE_INT	C/C++言語のリデュース型
MPI::LONG_INT	C/C++言語のリデュース型
MPI::TWOINT	C/C++言語のリデュース型
MPI::SHORT_INT	C/C++言語のリデュース型
MPI::LONG_DOUBLE_INT	C/C++言語のリデュース型
MPI::TWOREAL	Fortran言語のリデュース型
MPI::TWODOUBLE_PRECISION	Fortran言語のリデュース型
MPI::TWOINTEGER	Fortran言語のリデュース型
MPI::F_DOUBLE_COMPLEX	オプションのFortran言語の型
MPI::INTEGER1	明示的なサイズ型
MPI::INTEGER2	明示的なサイズ型
MPI::INTEGER4	明示的なサイズ型
MPI::INTEGER8	明示的なサイズ型
MPI::INTEGER16	明示的なサイズ型
MPI::REAL2	明示的なサイズ型
MPI::REAL4	明示的なサイズ型
MPI::REAL8	明示的なサイズ型
MPI::REAL16	明示的なサイズ型
MPI::F_COMPLEX4	明示的なサイズ型
MPI::F_COMPLEX8	明示的なサイズ型
MPI::F_COMPLEX16	明示的なサイズ型
MPI::F_COMPLEX32	明示的なサイズ型

表 16.3: その他のMPIのデータ型のためのC++言語の名前．実装でそれ以外のオプションの型を定義することもできる（MPI::INTEGER8など）．

Fortran言語の整数型:	MPI::INTEGER および返されるハンドル MPI::Datatype::Create_f90_integer, および利用可能であれば: MPI::INTEGER1, MPI::INTEGER2, MPI::INTEGER4, MPI::INTEGER8, MPI::INTEGER16
浮動小数点数:	MPI::FLOAT, MPI::DOUBLE, MPI::REAL, MPI::DOUBLE_PRECISION, MPI::LONG_DOUBLE および以下より返されるハンドル MPI::Datatype::Create_f90_real, および利用可能であれば: MPI::REAL2, MPI::REAL4, MPI::REAL8, MPI::REAL16
論理型:	MPI::LOGICAL, MPI::BOOL
複素数型:	MPI::F_COMPLEX, MPI::COMPLEX, MPI::F_DOUBLE_COMPLEX, MPI::DOUBLE_COMPLEX, MPI::LONG_DOUBLE_COMPLEX および以下より返されるハンドル MPI::Datatype::Create_f90_complex, および利用可能であれば: MPI::F_DOUBLE_COMPLEX, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32
バイト型:	MPI::BYTE

上記で定義したグループについて，各リデュース操作のための有効なデータ型を以下に示す．

Op	使用可能な型
MPI::MAX, MPI::MIN	C言語の整数型，Fortran言語の整数型，浮動小数点数型
MPI::SUM, MPI::PROD	C言語の整数型，Fortran言語の整数型，浮動小数点数型，複素数型
MPI::LAND, MPI::LOR, MPI::LXOR	C言語の整数型，論理型
MPI::BAND, MPI::BOR, MPI::BXOR	C言語の整数型，Fortran言語の整数型，バイト型

MPI::MINLOCとMPI::MAXLOCはC言語とFortran言語のそれに対応するものと同様の機能を持つ．177ページの第5.9.4節を参照すること．

16.1.7 コミュニケータ

MPI::Commクラス階層はMPIによって暗黙的に定義された別の種類のコミュニケータを明確化し，強力に型指定することができる．最初のMPIの設計では全ての型のコミュニケータに対して1つの型のハンドルしか定義していないため，C++言語の設計に対して以下の明確化が行われる．

コミュニケータの型 コミュニケータには6種類の型，つまり，MPI::Comm，MPI::Intercomm，MPI::Intracomm，MPI::Cartcomm，MPI::Graphcomm，MPI::Distgraphcommがある．MPI::Commは抽象的な基本コミュニケータクラスで，全ての MPI コミュニケータに共通する機能をカプセル化する．MPI::IntercommとMPI::IntracommはMPI::Commから派生している．MPI::Cartcomm，MPI::Graphcomm，MPI::DistgraphcommはMPI::Intracommから派生している．

> **ユーザへのアドバイス** 基本クラスのインスタンスを使用して派生クラスを初期化することはC++言語では認められていない．例えば，IntracommからCartcommを初期化することは認められていない．また，MPI::Commは抽象的な基本クラスであるため，インスタンスを作成することはできず，クラスMPI::Commのオブジェクトを作成することはできない．ただし，MPI::Commへの参照またはポインタを作成することはできる．
>
> **例 16.4** 以下のコードは誤りである．
>
> ```
> Intracomm intra = MPI::COMM_WORLD.Dup();
> Cartcomm cart(intra);          // This is erroneous
> ```
>
> （ユーザへのアドバイス終わり）

MPI::COMM_NULL MPI::COMM_NULLの特定の型は実装に応じて決まる．MPI::COMM_NULLは全ての型のコミュニケータを使用した比較と初期化で使用できなければならない．MPI::COMM_NULLはまた，パラメータリスト内にコミュニケータ引数があることが想定される関数に渡すことができなければならない（コミュニケータ引数の値にMPI::COMM_NULLが使用できる場合）．

> **根拠** MPI::COMM_NULLにはいくつかの実装方法がありうる．実現方法ではなく，必要な動作を指定することにより，実装者の自由度を最大限にできる．（根拠の終わり）

例 16.5 以下の例で，MPI::COMM_NULLを使用した代入と比較の動作を示す．

```
MPI::Intercomm comm;
comm = MPI::COMM_NULL;            // assign with COMM_NULL
if (comm == MPI::COMM_NULL)       // true
  cout << "comm is NULL" << endl;
if (MPI::COMM_NULL == comm)       // note -- a different function!
  cout << "comm is still NULL" << endl;
```

Dup()はMPI::Commのメンバー関数として定義されず，MPI::Commの派生クラス用に定義される．Dup()は仮想関数ではなく，出力パラメータに値を返す．

MPI::Comm::Clone() MPI用のC++言語のインターフェイスには，新しい関数Clone()が含まれている．MPI::Comm::Clone()は純粋仮想関数である．派生コミュニケータクラスの場合，Clone()の動作はDup()と同様だが，新しいオブジェクトを参照で返す．Clone()関数のプロトタイプは以下のとおりである．

```
Comm& Comm::Clone() const = 0
Intracomm& Intracomm::Clone() const
Intercomm& Intercomm::Clone() const
Cartcomm& Cartcomm::Clone() const
Graphcomm& Graphcomm::Clone() const
Distgraphcomm& Distgraphcomm::Clone() const
```

根拠 Clone()は，C++言語のプログラマとライブラリの作成者によって想定される「仮想複製」機能を備えている．Clone()は新しいオブジェクトを参照で返すため，ユーザは責任を持って最後にオブジェクトを削除する必要がある．Dup()の機能を変更するのではなく，新しい名前が付けられている． （**根拠の終わり**）

実装者へのアドバイス Clone()とDup()のプロトタイプは以下のようなクラスの宣言になる．

```
namespace MPI {
  class Comm {
     virtual Comm& Clone() const = 0;
  };
  class Intracomm : public Comm {
     Intracomm Dup() const { ... };
     virtual Intracomm& Clone() const { ... };
  };
  class Intercomm : public Comm {
     Intercomm Dup() const { ... };
     virtual Intercomm& Clone() const { ... };
  };
  // Cartcomm, Graphcomm,
  // and Distgraphcomm are similarly defined
};
```

（**実装者へのアドバイス終わり**）

16.1.8 例外

MPI用のC++言語インターフェイスには，Set_errhandler()メンバー関数で使用できる定義済みのエラーハンドラMPI::ERRORS_THROW_EXCEPTIONSが用意されている．MPI::ERRORS_THROW_EXCEPTIONSはC++言語の関数でのみ設定または取得できる．C++言語以外のプログラムでC++言語MPI::ERRORS_THROW_EXCEPTIONSエラーハンドラを呼び出すエラーが発生した場合，例外は呼び出しスタックをC++言語のコードで捕捉されるまで遡る．捕捉するC++言語のコードがない場合の動作は未定義である． マルチスレッド環境の場合，またはノンブロッキングMPI呼び出し時のバックグラウンド処理中に例外が発行された場合，その動作は実装次第である．

エラーハンドラMPI::ERRORS_THROW_EXCEPTIONSにより， MPI::SUCCESS以外のMPI結果コードに対して， MPI::Exceptionが発行される．MPI::Exceptionクラスに対する公開インターフェイスは以下のように定義される．

```
namespace MPI {
  class Exception {
  public:

    Exception(int error_code);

    int Get_error_code() const;
    int Get_error_class() const;
    const char *Get_error_string() const;
  };
};
```

実装者へのアドバイス 例外はMPI::ERRORS_THROW_EXCEPTIONSの内部で投げられる．例外が投げられた時点で制御がユーザに戻ることが期待される．エラーが発生し，かつ，MPI_ERRORS_RETURNが指定されている場合，一部のMPI関数はパラメータにある種の情報が返される．同じ種類の情報が例外の発生時に提供される必要がある．

例えば，MPI_WAITALLはステータス配列内に該当するエントリの要求に対してエラーコードを設定し， MPI_ERR_IN_STATUSを返す．
MPI::ERRORS_THROW_EXCEPTIONSを使用する場合，例外が投げられる前に適切にステータス配列内のエラーコードが設定される． （**実装者へのアドバイス終わり**）

16.1.9 言語間の運用性

C++言語インターフェイスでは，異なる言語が混在する環境での運用性のために下記の関数が用意されている．これらの関数では，C言語とC++言語の間のシームレスな移行が可能になっている，<CLASS>に対応するC++言語のクラスに派生クラスがある場合，関数では派生クラスとC言語MPI_<CLASS>の間の変換も行われる．

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)
```

```
MPI::<CLASS>(const MPI_<CLASS>& data)
MPI::<CLASS>::operator MPI_<CLASS>() const
```

これらの関数については，第16.3.4節で説明する．

16.1.10 プロファイリング

ここでは，MPIに対するC++言語プロファイリングインターフェイスの要件について説明する．

> **実装者へのアドバイス** プロファイリングの主な目的はユーザコードから関数呼び出しを捕捉することであるため，関数呼び出しの捕捉とプロファイリングを可能にするために基礎となる実装を階層化する方法を決定するのは実装者である．MPIのC++言語の呼び出し形式の実装が別の言語（C言語など）のMPIの呼び出し形式の上に階層化される場合，またはC++言語の呼び出し形式が別の言語のプロファイリングインターフェイスの上に階層化される場合，下層のMPI実装ですでにMPIプロファイリングインターフェイスの要件が満たされているため，それ以外のプロファイリングインターフェイスは必要ない．
>
> 他のプロファイリングインターフェイスにアクセスできないネイティブ なC++言語MPI実装は，この節で説明する要件を満たしているインターフェイスを実装する必要がある．
>
> 質の高い処理系では，この節で説明されるインターフェイスを実装することにより，可搬なC++言語プロファイリングライブラリを促進することができる．実装者は，C++言語プロファイリングインターフェイスを構築するかどうかのオプションを用意したいと考えることがある．別の言語または別のプロファイリングインターフェイスの呼び出し形式上に階層化されているC++言語の実装は，第3の階層を挿入してC++言語プロファイリングインターフェイス（**実装者へのアドバイス終わり**）

C++言語のMPIプロファイリングインターフェイスの要件を満たすため， MPI関数の実装では以下のことが必要となる．

1. MPI定義の全ての関数が名前シフトによりアクセスできるメカニズムを提供する．そのため，全てのMPI関数（通常はプリフィックスス“MPI::”で始まる）へのアクセスもプリフィックス“PMPI::”を使用して行えなければならない．

2. 置き換えられていないMPI関数が名前の衝突を起こすことなく実行イメージとリンクできるようにすること．

3. 別の言語の上に階層化されている場合，呼び出し形式毎にプロファイリングインターフェイスを実装する必要があるか，最下層のルーチンにのみ実装することで効率化できるか，をプロファイラの開発者が判断できるように，別の言語のMPIインターフェイスの呼び出し形式の実装について文書化すること．

4. 別の言語の呼び出し形式の実装が階層化アプローチを通して実装されている場合（C++言語の呼び出し形式がC言語の実装を呼び出す「ラッパー」関数の集合である場合など），これらのラッパー関数を残りのライブラリから分離できるようにすること．

   （少なくともUnixリンカにより）プロファイリングライブラリを想定通りに実行する場合これらのラッパー関数を組み込む必要があるため，別のプロファイリングライブラリを正しく実装するためにこのことが必要となる．この要件により，プロファイリングライブラリの作成者は元のMPI ライブラリからこれらの関数を抽出し，他の不要なコードを記述することなくプロファイリングライブラリに追加することができる．

5. 無操作ルーチンMPI::PcontrolをMPIライブラリに組み込む．

実装者へのアドバイス C++言語のプロファイリングインターフェイスの実装方法には，（少なくとも）継承とキャッシングの2つがある．継承ベースのアプローチは，コミュニケータクラスの仮想継承実装が必要となる可能性があるため，魅力的とは言えない．そのため，実装者が対応するMPIオブジェクトにPMPIオブジェクトをキャッシングする可能性が高い．キャッシング方式を以下に説明する．

各ルーチンへの「真の」エントリポイントはnamespace PMPI内に用意することができる．そのため，非プロファイリングバージョンはnamespace MPI内に用意することができる．

MPIハンドル内でのPMPIオブジェクトのインスタンスのキャッシングは，プロファイリング方式を実装するのに必要な「所有」関係を提供する．

MPIオブジェクトの各インスタンスは単にPMPIオブジェクトのインスタンスを「包む」．MPIオブジェクトは，内部PMPI オブジェクトで対応する関数を呼び出す前にプロファイリング処理を実行することができる．

プログラムを再リンクするだけでプロファイリングを有効にするための鍵は， 全てのMPI関数を宣言するヘッダファイルを用意することである．関数は別の場所で定義し，ライブラリとしてコンパイルされている必要がある．MPIの定数はMPI名前空間内でextern として宣言する必要がある．例として， mpi.hのサンプルファイルからの抜粋を以下に示す．

例 16.6 mpi.hのサンプルファイル

```
namespace PMPI {
  class Comm {
  public:
    int Get_size() const;
  };
  // etc.
};
```

```
namespace MPI {
public:
  class Comm {
  public:
    int Get_size() const;

  private:
    PMPI::Comm pmpi_comm;
  };
};
```

MPIクラスの全てのコンストラクタ，代入演算子，デストラクタは，必要に応じて内部PMPIオブジェクトの初期化/破壊を行う必要がある．

関数の定義は別々のオブジェクトファイルに格納する必要がある．
PMPIクラスメンバー関数と非プロファイリングバージョンのMPIクラスメンバー関数はlibmpi.aにコンパイルすることができ，プロファイリングバージョンはlibpmpi.aにコンパイルすることができる．libmpi.aと libpmpi.aの両方をリンクする場合MPIクラスメンバー関数の名前の定義が複数存在する状況を避けるため，PMPIクラスメンバー関数とMPI定数をlibmpi.aライブラリの非プロファイリングMPIクラスメンバー関数とは別のオブジェクトファイルに格納する必要があることに注意すること．例えば，次のようになる．

例 **16.7** pmpi.cc：コンパイルしてlibmpi.aに格納

```
int PMPI::Comm::Get_size() const
{
  // Implementation of MPI_COMM_SIZE
}
```

例 **16.8** constants.cc, to be compiled into libmpi.a. constants.cc：コンパイルして libmpi.aに格納

```
const MPI::Intracomm MPI::COMM_WORLD;
```

例 **16.9** mpi_no_profile.cc：コンパイルしてlibmpi.aに格納

```
int MPI::Comm::Get_size() const
{
  return pmpi_comm.Get_size();
}
```

例 **16.10** `mpi_profile.cc`, to be compiled into `libpmpi.a`. `mpi_profile.cc`：コンパイルして`libpmpi.a`に格納

```
int MPI::Comm::Get_size() const
{
  // Do profiling stuff
  int ret = pmpi_comm.Get_size();
  // More profiling stuff
  return ret;
}
```

（実装者へのアドバイス終わり）

16.2 Fortran言語のサポート

16.2.1 概要

Fortran言語のMPI-2の呼び出し形式はFortran 90言語（以降）の標準との互換性が保証されるよう設計されている．これらの呼び出し形式はほとんどの場合，Fortran 77言語と互換性がある暗黙的な形式のインターフェイスである．

根拠 Fortran 90言語には，Fortran 77言語よりも「現代的な」言語になるように設計された多数の機能がある．Fortran 90言語用に調整された一連の呼び出し形式を使用してMPI でこれらの新しい機能を利用できるのが当然と考えられる．MPIでは，多くの技術的な問題のため，（まだ）これらの機能の多くを利用できない．（根拠の終わり）

MPIでは，2つのレベルでFortran言語のサポートを定義している． 第16.2.3節と第16.2.4節を参照すること．この節の残りの部分では，特に明記していない限り，「Fortran言語」と「Fortran 90言語」は「Fortran 90言語」以降を指すものとする．

1. **Fortran言語の基本サポート** このレベルのFortran言語のサポートを実装した場合，MPI-1で規定された元々のFortran言語の呼び出し形式と，第16.2.3節で規定する一部の追加要件が提供される．

2. **Fortran言語の拡張サポート** このレベルのFortran言語のサポートを実装した場合，Fortran言語の基本サポートに，第16.2.4節で説明するFortran 90言語を特別にサポートする追加機能が提供される．

Fortran言語インターフェイスを提供するMPI-2準拠の実装では，対象となるコンパイラがモジュールまたは`KIND`パラメータ型をサポートしていない場合を除いて，Fortran言語の拡張サポートを提供しなければならない．

16.2.2 MPIでのFortran言語の呼び出し形式の問題

ここでは，Fortran言語プログラムでMPIを使用する場合に発生する可能性のある多数の問題について説明する．「ユーザへのアドバイス」に該当するもので，MPIとFortran言語との関連性を明確にする．標準への追加ではなく，標準の明確化を目的としている．

元々のMPI仕様に記載されているように，このインターフェイスはいくつかの点でFortran言語の標準に違反している．そのためにFortran 77言語のプログラムで問題が発生することはほとんどないが，Fortran 90言語のプログラムでは問題が大きくなるため，新しいFortran 90言語の機能を使用する場合は注意が必要となる． 違反したインターフェイスが受け入れられ, 維持されたのは，MPIの使いやすさにとって重要であるからである．この節の残りの部分では，考えられる問題について詳しく説明する．ここでの説明は，元々のMPI仕様でのFortran言語の呼び出し形式（Fortran 77言語ではなく，Fortran 90言語）の説明に優先し，それに取って代わるものである．

以下のMPI機能はFortran 90言語と整合していない．

1. 選択型引数を持つMPIサブルーチンは別の引数型を使用して呼び出すことができる．

2. サイズが仮定された仮引数を持つMPIサブルーチンに スカラー型の実引数を渡すことができる．

3. 多くのMPIルーチンでは，実引数がアドレスによって渡され，引数がサブルーチンへの入り口またはサブルーチンからの出口でコピーされないことが前提となっている．

4. ユーザプログラムが（ノンブロッキング通信で使用される通信バッファなど）のユーザデータの読み取りや変更を行うのと同時に， MPI実装はそのデータの読み取りや変更を行うことがある．

5. いくつかの名前付き「定数」，例えば，MPI_BOTTOM， MPI_IN_PLACE，MPI_STATUS_IGNORE， MPI_STATUSES_IGNORE，MPI_ERRCODES_IGNORE，MPI_UNWEIGHTED，MPI_ARGV_NULL， MPI_ARGVS_NULLは普通のFortran言語定数ではなく，特別な実装が必要となる．詳細は17ページの 第2.5.4節を参照すること．

6. ，メモリ割当ルーチンMPI_ALLOC_MEMは，割り当てされるメモリをFortran言語の変数と関連付けることができる言語拡張がなければ，Fortran言語で有効に使用することができない．

また，MPIは以下のような多くの点でFortran 77言語と整合していない．

- MPIの識別子は6文字を超える．
- MPIの識別子は最初の文字の後に下線を含むことができる．

- MPIにはヘッダーmpif.hが必要である．ヘッダーに対応していないシステムでは，名前付き定数の値を指定する必要がある．
- MPIの多くのルーチンには，アドレス情報が格納されるKIND型パラメータの整数（MPI_ADDRESS_KIND，MPI_OFFSET_KINDなど）が含まれる．Fortran 90言語形式のパラメータ型に対応していないシステムでは，代わりにINTEGER*8またはINTEGERを使用する必要がある．

MPI-1には，アドレスサイズ情報を入力として取り，アドレスサイズ情報を出力として返すいくつかのルーチンが含まれていた．C言語ではこのような引数はMPI_Aint型．Fortran言語ではINTEGER型となっていた． 整数型がアドレス型より小さいマシンでは，これらのルーチンで情報が欠落する可能性がある．MPI-2では，このような関数の使用は推奨されなくなっており，INTEGER型引数KIND=MPI_ADDRESS_KINDを取るルーチンに置き換わっている．多くの新しいMPI-2関数でも非デフォルトのKINDの INTEGER型引数を取っている．詳細は， 19ページの第2.6節と89ページの第4.1.1節を参照すること．

強い型指定による問題

選択型引数を持つ全てのMPI関数は，異なるFortran言語のデータ型の実引数を同じ仮引数と関連付ける．このことはFortran 77言語では許可されておらず，Fortran 90言語ではそれぞれの型に対して関数が異なる関数によりオーバーロードされる場合のみ，技術的に可能となる．C言語では，void*の正式な引数を使用するとこのような問題が回避できる．

以下の部分コード例は技術的には不正で，コンパイル時エラーが発生する可能性がある．

```
integer i(5)
real    x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

特に，Fortran 90言語のコンパイラがエラーを返す心配があるが，コンパイラで警告の発行以上のことが行われるのはまれである．

Fortran言語では，スカラー型の実引数を配列の仮引数に渡すことも技術的に不正である．従って，以下の部分コード例はMPI_SENDのbuf引数はサイズが仮定された配列<type> buf(*)として宣言されているため，エラーとなる可能性がある．

```
integer a
call mpi_send(a, 1, MPI_INTEGER, ...)
```

ユーザへのアドバイス 型のチェックに関する何らかの問題が生じた場合，コンパイラのフラグを使用するか，個別にコンパイルするか，第16.2.4節で説明するFortran言語の拡張サポートを利用したMPI実装を使用することにより，対処する

ことができる．関数またはサブルーチンの引数を使用せず，通常はルーチンにローカルな変数を使用する代替方法は，EQUIVALENCE文を使用して，コンパイラで受け付けられる型を持つ別の変数を生成することである． （ユーザへのアドバイス終わり）

データのコピーおよび連続領域配置による問題

直線的なアドレス空間によりアクセス可能なメモリの連続チャンクというアイディアは，MPIでは暗黙的である．MPI はこのメモリとの間で相互にデータのコピーを行う．MPIプログラムはメモリのアドレスとオフセットを渡すことにより，データの領域を指定する． C言語では，連続領域配置規則とポインタにより必要な全ての下層構造が示される．

Fortran 90言語では，ユーザデータは必ずしも連続領域に格納されない． 例えば部分配列A(1:N:2)にはAの要素のうち，添字が1，3， 5，... のもののみが含まれる．対象がこのような部分配列であるポインタ配列の場合も同様である．大部分のコンパイラでは，仮引数である配列が明示的な形状（B(N)など）で宣言されている場合，またはサイズが仮定されている（B(*) など）場合，メモリの連続領域に格納される．必要に応じて，配列をメモリの連続領域にコピーすることにより，これが行われる．Fortran 77言語とFortran 90言語規格にはこのようなコピーを許すように注意深くかかれているが，Fortran 77言語のほとんどのコンパイラではこれが行われない．[2]

MPIバッファ仮引数はサイズが仮定された配列であるため，ノンブロッキング呼び出しの場合は重大な問題が発生する可能性がある．コンパイラは戻り時に一時配列をコピーするが，MPIはデータが保持されているメモリに継続的にデータのコピーを行う．例えば，以下のコード例を考えてみる．

```
real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

MPI_IRECVの最初の仮引数はサイズが仮定された配列（<type> buf(*)）であるため，部分配列a(1:100:2)はメモリ内の連続領域に格納されるように，MPI_IRECVに渡される前に一時配列にコピーされる．MPI_IRECVはすぐに戻り，一時配列から配列aにデータがコピーされる．その後，MPIは解放された一時配列のアドレスに書き込みを行うことができる．一時配列はデータの送出が完了する前に解放されることがあるため，MPI_ISENDでもコピーの問題が発生する．

大部分のFortran 90言語のコンパイラは，実引数が明確な形状を持っているかサイズの仮定された配列の全体である場合，またはこのような配列の A(1:N)のような「シンプル」な部分配列である場合，コピーを作成しない（「シンプル」の意味は，次の段落で詳しく定義する）．また，多くのコンパイラは，この点について，割当可能な配列を明確な形状を持つ配列と同様に扱う（ただし，一部はこれに該当しないものもある）．しかし，形状が仮定された配列とポインタ配列の場合はこれとは異なり，不連続であるためにコ

[2]Fortran言語の標準は配列データが不連続で格納されることを許すように書かれている．

ピーが行われることもある．このようなコピーにおいて，前の段落で説明したMPIの問題が生じる．

「シンプル」な配列断片の正式な定義は以下のとおりである．

```
name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )
```

つまり，完全に選択される0個以上の次元がある場合，1つの次元はストライドなしで選択され，0個以上の次元はシンプルな添字を使用して選択される．例えば，次のようになる．

```
A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
```

Fortran言語は列優先順で，最初の添字が最も速く変化するため，連続する配列のシンプルな断片も連続となる．[3]

スカラー型の引数でも同様の問題が生じる可能性がある．一部のコンパイラでは，Fortran 77言語の場合でも，呼び出されたプロシージャ内に一部のスカラー型仮引数を作成する．これが問題となることについて，以下の例で示す．

```
call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a

subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end
```

aをコピーする場合，MPI_IRECVは通信の完了時にコピーの方を変更し，a自体は変更しない．

一般的にコピーが行われるのは，重要な式（少なくとも1つの演算子または関数呼び出しを持つもの），親の連続部分（A(1:n:2)など）を選択しない部分配列，ターゲットがこのような部分配列であるポインタ，またはこのような部分配列と（直接的または間接的に）関連付けられた形状の仮定された配列などの引数に対してであることに注意すること．

呼び出し元または呼び出し先のプロシージャで引数のコピーを抑制するコンパイラオプションがある場合，これを使用する必要がある．

明確な形状を持っているかサイズが仮定された配列，このような配列のシンプルな部分配列，またはスカラー型を引数とする呼び出し元プロシージャでコンパイラがコピーを行う場合，またこれを抑制するためのコンパイラオプションがない場合，MPI_GET_ADDRESSを使用するアプリケーションやノンブロッキングMPIルーチン用にコンパイラを使用することはできない．コンパイラが呼び出し先プロシージャでスカラー型引数をコピーし，これを抑制するコンパイラオプションがない場合，このコンパイラは上記の例のようにサブルーチン呼び出し間でメモリ参照を使用するアプリケーションには使用できない．

[3]「シンプル」の定義をシンプルにするため，1つ以外の全ての部分添字を境界なしにするよう定めた．境界なしのコロンを使用すると，コンパイラにとってもプログラムを読む人にとっても，次元の全体が選択されていることが明らかになる．次元全体が1つまたは2つの境界により選択される状態を許可することもできたが，これはプログラムを読む人にとっては配列の宣言や最近の割当を確認する必要があることを意味し，コンパイラにとってはランタイムチェックが必要となることを意味する．

特別な定数

MPIでは，MPI_BOTTOMなど，正式なFortran言語定数として実装できない多数の特別な「定数」が必要となる．詳細なリストは17ページの第2.5.4節を参照すること．C言語では，これらは定数ポインタ，通常はNULLとして実装され，関数プロトタイプが変数自体ではなく，変数のポインタを呼び出す場合に使用される．

Fortran言語では，これらの特別な定数を実装するのに，Fortran言語の非標準の言語構成物を使用する必要がある場合もある．実装では定数用の特別な値と正規のデータとを区別できないため，（parameter 文を使用して定義するなどの方法で）定数用の特別な値を使用することはできない．通常，対象となるコンパイラはデータをアドレスによって渡すことに依拠するため，これらの定数は定義済みのstatic変数（MPIで宣言されたCOMMONブロック内の変数など）として実装される．サブルーチン内では，このアドレスはFortran言語の非標準のメカニズム（Fortran言語拡張，またはC言語関数の実装など）によって抽出することができる．

Fortran 90言語の派生型

MPIでは，Fortran 90言語の派生型を選択型の仮引数に渡すことを明示的にサポートしていない．実際に，mpiモジュールを通して明示的なインターフェイスを提供するMPI実装の場合，コンパイル時にコンパイラにより派生型の実引数が拒否される．明示的なインターフェイスが与えられていない場合でも，Fortran 90言語では派生型または派生型の配列のためのシーケンスの関連付けが保証されないことを知っておく必要がある．例えば，2つの要素で構成される派生型の配列を実装する場合，最初の要素を配列とし，続いて二番目の要素を配列とすることのよって実装できる．この場合，SEQUENCE属性が役に立つ場合がある．

以下の部分コードに，Fortran言語で派生型を送信する 1 つの方法を示す．この例では，全てのデータがアドレスで渡されることを前提としている．

```
type mytype
   integer i
   real x
   double precision d
end type mytype

type(mytype) foo
integer blocklen(3), type(3)
integer(MPI_ADDRESS_KIND) disp(3), base

call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)

base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
disp(3) = disp(3) - base

blocklen(1) = 1
```

```
    blocklen(2) = 1
    blocklen(3) = 1

    type(1) = MPI_INTEGER
    type(2) = MPI_REAL
    type(3) = MPI_DOUBLE_PRECISION

    call MPI_TYPE_CREATE_STRUCT(3, blocklen, disp, type, newtype, ierr)
    call MPI_TYPE_COMMIT(newtype, ierr)

! unpleasant to send foo%i instead of foo, but it works for scalar
! entities of type mytype
    call MPI_SEND(foo%i, 1, newtype, ...)
```

レジスタ最適化での問題

MPIには，ユーザコードから隠されていて，ユーザコードと同じメモリにアクセスしながら同時に実行される操作が用意されている．例えば，MPI_IRECVのデータ転送などがこの例である．コンパイラのオプティマイザは，メモリからのリロードやメモリへの格納なしで変数のコピーをレジスタ内に保存しておける期間を認識できることを前提としている．隠された操作がメモリコピーの読み取りや書き込みを実行している間にユーザコードが変数のレジスタコピーを使用すると，問題が生じる．この節ではレジスタ最適化の落とし穴について説明する．

変数がFortran言語のサブルーチンのローカルである（つまり，モジュールまたはCOMMONブロック内にない）場合，コンパイラでは変数が呼び出しの実引数である場合以外は呼び出し先のサブルーチンによって変更できないことが前提とされる．最も一般的なリンクの変換では，サブルーチンでレジスタの保存とリストアが行われることになっている．そのため，オプティマイザでは，呼び出しの前にこのような有効な変数のコピーを保持していたレジスタが戻り時にも有効なコピーを保持していることが前提とされる．

通常，ユーザがこのことを意識する必要はない．しかし，プログラムでMPI_SEND，MPI_RECVなどのバッファ引数により，関係する実際の変数を隠す名前が使用される場合は，この節に注意する必要がある．MPI_Datatypeに絶対アドレスが含まれるMPI_BOTTOMはその一例である．MPI_GET_ADDRESSを使用することにより，1つの変数をアンカーとして使用して他を取り込むデータ型を生成し，アンカーからのオフセットを取得することは，もう1つの例である．アンカー変数は呼び出しで言及される唯一のものとなる．ユーザのアプリケーションと並行して実行されるMPI操作を使用する場合も注意が必要である．

例16.11に，Fortran言語のコンパイラで行う可能性のある処理の例を示す．

コンパイラは，MPI_RECVによりbufの値が変化することを確認できないため，レジスタの無効になっていることがわからない．bufのアクセスは，MPI_GET_ADDRESSおよびMPI_BOTTOMを使用することにより隠される．

例16.12には，極端ではあるが許容される例を示す．

例 16.11 Fortran 90言語のレジスタ最適化

ソースコード：

```
call MPI_GET_ADDRESS(buf,bufaddr,
                ierror)
call MPI_TYPE_CREATE_STRUCT(1,1,
                bufaddr,
                MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf

call MPI_RECV(MPI_BOTTOM,1,type,...)
val_new = buf
```

コンパイル後のコード：

```
call MPI_GET_ADDRESS(buf,...)

call MPI_TYPE_CREATE_STRUCT(...)


call MPI_TYPE_COMMIT(...)
register = buf
val_old = register
call MPI_RECV(MPI_BOTTOM,...)
val_new = register
```

例 16.12 Fortran 90言語のレジスタ最適化–極端な例

ソースコード：	コンパイル後のコード：	別のコンパイル後のコード：
`call MPI_IRECV(buf,..req)`	`call MPI_IRECV(buf,..req)`	`call MPI_IRECV(buf,..req)`
	`register = buf`	`b1 = buf`
`call MPI_WAIT(req,..)`	`call MPI_WAIT(req,..)`	`call MPI_WAIT(req,..)`
`b1 = buf`	`b1 = register`	

平行スレッド上のMPI_WAITは，bufはMPI_IRECVの呼び出しとMPI_WAITの完了の間で変更される．しかし，コンパイラはMPI_IRECVが戻った後にbufが変更される可能性を知らないため，ソースに書かれているよりも早くbufのロードをスケジュールする可能性がある．MPI_WAITの呼び出しの前後にまたがってbufを保持するためにレジスタの使用を避ける理由はない．右の場合と同様に命令の並び替えが行われることもある．

命令の並び替えやバッファのレジスタへの割当を避けるため，2通りの可搬なFortran言語のコードが考えられる．

- コンパイラは，実引数としてバッファを使用した外部サブルーチンの呼び出しにより呼び出しを囲むことにより， MPIサブルーチンの呼び出しをまたいでバッファの参照が移動することを防止することができる．INTENTが外部サブルーチンで宣言されている場合，OUTまたはINOUTでなければならない．サブルーチン自体は空とすることができるが，コンパイラはこのことがわからず，バッファが変更されている可能性を仮定する必要がある．例えば，上記のMPI_RECVの呼び出しは以下のように置き換え，

  ```
  call DD(buf)
  call MPI_RECV(MPI_BOTTOM,...)
  call DD(buf)
  ```

  以下を別にコンパイルする．（bufは整数とする）

```
subroutine DD(buf)
  integer buf
end
```

コンパイラはまた，MPIサブルーチンの呼び出しをまたいで変数の参照が移動することを防止することもできる．

上記のMPI_WAITの呼び出しのようなノンブロッキング呼び出しの場合，転送の完了が確認できるまで，バッファの参照は許可されない．そのため，この場合は，MPI呼び出しの前の特別な呼び出しは不要である．つまり，この例のMPI_WAITの呼び出しは以下のように置き換えることができる．

```
call MPI_WAIT(req,..)
call DD(buf)
```

- 代替方法は，バッファまたは変数をモジュールまたは共通ブロックに格納して，MPIルーチンの呼び出しで実引数として参照され，定義され，示される各有効範囲内で，USEまたはCOMMON文によりアクセスすることである．コンパイラでMPIプロシージャがモジュールまたは共通ブロックを参照しないことが解析できない場合，コンパイラはMPIプロシージャ（上記の例ではMPI_RECV）がバッファまたは変数を変更する可能性があると仮定する必要がある．

Fortran言語の新しいバージョンで使用可能なVOLATILE属性はバッファまたは変数に必要な特性を与えるが，バッファまたは変数を含むコードの最適化を阻止する可能性もある．

C言語では，引数リストにない変数を変更するサブルーチンはレジスタの最適化の問題を起こさない．これは，&演算子を使用してポインタによりオブジェクトを格納しておいて，後からポインタによってオブジェクトを参照するのがこの言語の重要な部分だからである．一般的に，C言語のコンパイラは問題が発生しないように，暗黙の意味にも対応する．しかし，コンパイラの中には，安全でない可能性のある強力なオプションの最適化レベルを備えているものもある．

16.2.3 Fortran言語の基本サポート

Fortran 90言語は（事実上）Fortran 77言語の上位集合であるため，Fortran 90言語（およびそれ以降）のプログラムでは元々のFortran言語のインターフェイスを使用することができる．以下の要件が追加されている．

1. 実装では，最初のMPI-1の仕様に記載されたように，ファイルmpif.hを用意する必要がある．
2. mpif.hは固定ソース形式と自由ソース形式の両方で有効で同等でなければならない．

実装者へのアドバイス mpif.hを固定ソース形式及び自由ソース形式で両立できるように，継続行なしで mpif.h を構成し，プリプロセッサの自動インクルードを有効にし，固定形式の行の長さの拡張を有効にして作ることを推奨する．これが可能なのは，mpif.hに記述されているのが宣言のみだからであり，共通ブロックの宣言を複数の行に分割することができるからである．Fortran 77言語とFortran 90言語をサポートするには，mpif.hからコメントを全て除去する必要がある． （**実装者へのアドバイス終わり**）

16.2.4 Fortran言語の拡張サポート

Fortran言語の拡張サポートによる実装では，以下を用意する必要がある．

1. mpiモジュール
2. Fortran言語の基本数値型を追加でサポートするための新しい関数の集合．ここに含まれるパラメータ型には，MPI_SIZEOF，MPI_TYPE_MATCH_SIZE，MPI_TYPE_CREATE_F90_INTEGER，MPI_TYPE_CREATE_F90_REAL，MPI_TYPE_CREATE_F90_COMPLEXなどがある．パラメータ型はFortran言語の基本型で，KIND型のパラメータを使用して指定される．これらのルーチンについては，第16.2.5節で詳しく説明する．

また，質の高い処理系では，選択型の引数を持つMPIルーチンで致命的な型の不一致エラーが発生しないようにするためのメカニズムを用意する必要がある．

mpiモジュール

MPI実装では，Fortran 90言語のプログラムでuseできるmpiという名前のモジュールを用意する必要がある．このモジュールでは以下を行う必要がある．

- 全ての名前付きMPI定数の定義
- 値を返すMPI関数の宣言

MPI実装では，標準への準拠を維持しながらMPIの有用性を向上させる他の機能をmpiモジュールに組み込むこともできる．例えば，以下のことができる．

- MPIルーチンの全てまたはサブセットのためのインターフェイスの提供
- これらのインターフェイスブロック内のINTENT情報の提供

実装者へのアドバイス 適切なINTENTはMPI汎用インターフェイスで渡されるものとは異なることがある．実装では，関数がMPI標準に準拠するようにINTENTを選択する必要がある． （**実装者へのアドバイス終わり**）

根拠 MPI汎用インターフェイスで渡されるINTENTは正確に定義されておらず，Fortran言語の正しいINTENTには対応していない場合がある．例えば，絶対アドレスを使用してデータ型により指定されたバッファに受信するには，MPI_BOTTOMのOUT仮引数への関連付けが必要となることがある．また，MPI_BOTTOM やMPI_STATUS_IGNOREなどの「定数」はFortran言語で定義された定数ではないが，標準ではない方法ではあるが「特別なアドレス」を使用する．最後に，MPI-1の汎用 INTENTはいくつかの点がMPI-2で変更されている．例えば，MPI_IN_PLACEではOUTをINOUT に変更している． （**根拠の終わり**）

アプリケーションではmpiモジュールまたはヘッダーmpif.h を使用することができる．実装では型不一致エラーを防止するためのモジュールを使用する必要のある場合がある（以下を参照）．

ユーザへのアドバイス 特定のシステムで型不一致エラーを避けるために必要でない場合でも，mpiモジュールを使用することを推奨する．モジュールの使用は，いくつかの点でヘッダーの使用よりもメリットが大きい． （**ユーザへのアドバイス終わり**）

USE mpiを使ったあるルーチンとINCLUDE mpif.hを使った他のルーチンを一緒にリンクできなければならない．

型一致の問題のない選択型引数を持つサブルーチン

質の高いMPI実装では，MPI選択型引数が型不一致によるコンパイル時またはランタイムの致命的エラーが発生しないことを保証するメカニズムを用意する必要がある．MPI実装では，型一致の問題を防止するため，アプリケーションでmpiモジュールを使用するか，特定のコンパイラフラグを使用してコンパイルすることが必要となる場合がある．

実装者へのアドバイス コンパイラでエラーが発生しない場合，既存のインターフェイスに対して何もする必要はない．コンパイラでエラーが発生する可能性がある場合，一連のオーバーロード関数を使用することができる．M. Hennecke の論文[26]を参照すること．コンパイルでエラーが発生しない場合でも，引数リストのエラーを検出するために，全てのルーチンのための明確なインターフェイスは有益である．また，INTENT 情報を渡す明確なインターフェイスはBUF(*)引数のコピーの量を削減することができる． （**実装者へのアドバイス終わり**）

16.2.5 Fortran言語の基本数値型の追加サポート

この節のルーチンは第16.2.4節で説明したFortran言語の拡張サポートの一部である．

MPIでは，C言語およびFortran言語でサポートされている名前付きの基本型に対応する名前付きデータ型をいくつか用意している．ここには，MPI_INTEGER，MPI_REAL，

MPI_INT，MPI_DOUBLEなどのほか，MPI_REAL4，MPI_REAL8などのオプションの型が含まれる．言語の宣言とMPI の型の間には1対1の対応がある．

Fortran言語（Fortran 90言語以降）では，いわゆるKINDパラメータ型が用意されている．これらの型は，1つ以上の変異形から選択できるオプションの整数型KINDパラメータを持つ基本型（INTEGER，REAL，COMPLEX，LOGICAL，CHARACTERのいずれか）を使用して宣言される．異なるKIND値自体のの特定の意味は実装内容によって決まり，言語によって指定されるものではない．Fortran言語ではREALおよびCOMPLEX型のためのKIND選択関数selected_real_kindとINTEGER 型のためのselected_int_kindが用意されており，ユーザは最小限の精度または桁数により変数を宣言することができる．これらの関数により，Fortran言語でのKINDパラメータのREAL，COMPLEX，およびINTEGER型変数の宣言の可搬性を与える．この方式にはFortran 77言語に対する下位互換がある．Fortran言語のREALおよびINTEGER型変数には，何も指定されない場合のデフォルトKINDがある．Fortran言語のDOUBLEPRECISION型変数はデフォルトKINDのない基本REAL型である．以下の2つの宣言は同等である．

```
double precision x
real(KIND(0.0d0)) x
```

MPIでは数値基本型を使用して通信するための2つの直交する方法が用意されている．最初の方法が利用できるのは変数が可搬に宣言されている場合で，デフォルトをKINDを使用するか，selected_int_kindまたはselected_real_kind関数で取得したKINDパラメータを使用する．この方法では，MPIは自動的に正しいデータサイズ（4または8バイトなど）を選択し，異機種環境での表現の変換を行う．2つ目の方法を使用すると，マシンの表現を公にすることにより，通信を完全に制御することができる．

指定の精度と指数範囲を持つパラメータデータ型

MPIでは，標準のFortran 77言語の数値型のMPI_INTEGER，MPI_COMPLEX，MPI_REAL，MPI_DOUBLE_PRECISION，MPI_DOUBLE_COMPLEXに対応する名前付きデータ型を用意している．MPIは正しいデータサイズを自動的に選択し，異機種環境での表現の変換を行う．この節で説明するメカニズムにより，可搬なパラメータ化数値型をサポートするようこのモデルが拡張される．

可搬なパラメータ型をサポートするためのモデルは次のとおりである．実数変数はKINDパラメータを規定するためにselected_real_kind(p，r)を使用して（おそらく間接的に）宣言される．ここで，pは精度の10進数で，rは指数範囲である．MPI は暗黙的に定義済みのMPIデータ型D(p，r)の2次元配列を管理する．D(p,r)はコンパイラによってサポートされる(p，r)の各値に対して定義され，ここには1つの値が指定されていないペアも含まれる．コンパイラによってサポートされていない添字(p，r)を使用して配列の要素にアクセスしようとするのは誤りである．MPIは暗黙的にCOMPLEXデータ型の同様の配列を管理する．整数型に関しては，selected_int_kindに関連し，要求された

桁数rによってインデックス指定された同様の暗黙的な配列がある．これらの暗黙的な配列に含まれる定義済みのデータ型は名前付きのMPIデータ型MPI_REALなどと同じではなく，新しいセットであることに注意すること．

実装者へのアドバイス 上記は単に説明上のものであり，このような内部配列の実装は期待されない． （**実装者へのアドバイス終わり**）

ユーザへのアドバイス selected_real_kind()は多数の(p,r)のペアを，コンパイラによってサポートされる少数のKINDパラメータにマッピングする．KINDパラメータは言語によって指定されず，可搬ではない．言語から見ると，同じ基本型の固有型とKINDパラメータは同じ型である．異機種環境での相互運用性を保証するため，MPIはより厳格になっている．対応するMPIデータ型が一致するには，同じ(p,r)値（REALおよびCOMPLEX）またはr値（INTEGER）を持っていなければならない．そのため，MPIは基本的な言語の型よりも多くのデータ型を備えている．（**ユーザへのアドバイス終わり**）

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

IN	p	10進数での精度（整数型）
IN	r	10進数の指数範囲（整数型）
OUT	newtype	要求されるMPIデータ型（ハンドル）

```
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
{static MPI::Datatype MPI::Datatype::Create_f90_real(int p, int r) （廃止された呼び出し形式, 第15.2節を参照）}
```

この関数はKIND selected_real_kind(p, r)のREAL変数に一致する定義済みのMPIデータ型を返す．上記のモデルでは，要素D(p, r)のハンドルを返す．selected_real_kind(p, r)の呼び出しではpまたはrを省略することができる（両方を省略することはできない）．同様に，pまたはrの設定をMPI_UNDEFINEDにすることができる．通信では，MPI_TYPE_CREATE_F90_REALによって返されたMPIデータ型Aがデータ型Bに一致するのは，pおよびrに同じ値を使用して呼び出されたMPI_TYPE_CREATE_F90_REALによってBが返された場合，またはBがこのようなデータ型の複製である場合のみである．返されたデータ型を“external32”データ表現により使用する上での制限は515ページに記載されている．

コンパイラによってサポートされていないpおよびrの値を指定するのは誤りである．

MPI_TYPE_CREATE_F90_COMPLEX(p, r, newtype)

IN	p	10進数での精度（整数型）
IN	r	10進数の指数範囲（整数型）
OUT	newtype	要求されるMPIデータ型（ハンドル）

```
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
```

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
{static MPI::Datatype MPI::Datatype::Create_f90_complex(int p, int r) （廃止
              された呼び出し形式, 第15.2節を参照）}
```

この関数はKIND selected_real_kind(p, r)のCOMPLEX 変数に一致する定義済みのMPIデータ型を返す．selected_real_kind(p, r)の呼び出しではpまたはrを省略することができる（両方を省略することはできない）．同様に，pまたは rの設定をMPI_UNDEFINEDにすることができる．この関数によって生成されたデータ型の一致規則は，MPI_TYPE_CREATE_F90_REALによって生成されたデータ型の一致規則と同様である．返されたデータ型を“external32”データ表現により使用する上での制限は515ページに記載されている．

コンパイラによってサポートされていないpおよびrの値を指定するのは誤りである．

MPI_TYPE_CREATE_F90_INTEGER(r, newtype)

IN	r	10進数の指数範囲，つまり10進数の値（整数型）
OUT	newtype	要求されるMPIデータ型（ハンドル）

```
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
    INTEGER R, NEWTYPE, IERROR
{static MPI::Datatype MPI::Datatype::Create_f90_integer(int r) （廃止された呼
              び出し形式, 第15.2節を参照）}
```

この関数はKIND selected_int_kind(r)のINTEGER変数に一致する定義済みのMPIデータ型を返す．この関数によって生成されたデータ型の一致規則は，MPI_TYPE_CREATE_F90_REALによって生成されたデータ型の一致規則と同様である．返されたデータ型を“external32”データ表現により使用する上での制限は515ページに記載されている．

コンパイラによってサポートされていないrの値を指定するのは誤りである．

以下に例を示す．

```
integer       longtype, quadtype
integer, parameter :: long = selected_int_kind(15)
integer(long) ii(10)
real(selected_real_kind(30)) x(10)
call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
...

call MPI_SEND(ii, 10, longtype, ...)
call MPI_SEND(x,  10, quadtype, ...)
```

ユーザへのアドバイス 上記の関数で返されるデータ型は定義済みのデータ型である．これらは解放することができず，コミットする必要がなく，定義済みのリデュース操作で使用することができる，MPIの名前付きの定義済みデータ型と構文的には異なる動作をするが，意味的には同じである状況が2つある．

1. MPI_TYPE_GET_ENVELOPEは，プログラムがpおよびrの値を取得することができる特別な結合子を返す．
2. データ型は名前付きでないため，コンパイル時のイニシャライザとして使用できない．あるいはMPI_TYPE_CREATE_F90_ルーチンのいずれかの呼び出しの前にアクセスできない．

`selected_real_kind()`または`selected_int_kind()`を使用して取得されていない非デフォルトの`KIND`値を指定する変数が宣言されている場合，一致するMPIデータ型を取得するには，次の節で説明するサイズベースのメカニズムを使用する必要がある．（**ユーザへのアドバイス終わり**）

実装者へのアドバイス アプリケーションは同じ (xxxx,p,r)の組み合わせでMPI_TYPE_CREATE_F90_xxxxの呼び出しを繰り返すことがよくある．アプリケーションでは，返された定義済みの名前無しデータ型ハンドルを解放することはできない．非常に多数のハンドルが生成されるのを防止するため，質の高いMPI実装では同じ (REAL/COMPLEX/INTEGER,p,r) の組み合わせに対して同じデータ型ハンドルを返す必要がある．その前のMPI_TYPE_CREATE_F90_xxxxの呼び出しでの(p,r)の組み合わせをチェックし，ハッシュテーブルにより前に生成されたハンドルを検出すると，(xxxx,p,r)の同じ組み合わせで以前に生成されたデータ型を検索する際のオーバーヘッドが制限される．（**実装者へのアドバイス終わり**）

根拠 MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGERインターフェイスでは，入力として，有益でコンパイラ非依存の外部（448ページの第13.5.2節）またはユーザ定義（449ページの第13.5.3節）データ表現を定義するための，また異機種環境で効率的な自動のデータ変換が行えるようにするためのオリジナルの範囲と精度値が必要となる．（**根拠の終わり**）

ここで，448ページの第13.5.2節で説明した“external32”外部データ表現と一緒に使用した場合の，この節で説明するデータ型の動作を規定する．

external32表現は整数型および浮動小数点数の値のデータ形式を指定する．整数値はビッグエンディアンの2の補数形式で表現される．浮動小数点数の値は3つのうちのいずれかのIEEE形式で表現される．これらはそれぞれ，4，8，16バイトのストレージを必要とするIEEE “Single”（単精度），“Double”（倍精度），“Double Extended”（拡張倍精度）形式である．IEEE “DoubleExtended”形式の場合，MPIは16バイトのフォーマット幅を指定し，指数部は15ビット，bias = +10383，仮数ビットは112，“Double”形式と同様のエンコードとなる．

MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGERによって返されるデータ型のexternal32表現は以下の規則によって与えられる．

MPI_TYPE_CREATE_F90_REALの場合は以下のようになる．

```
if      (p > 33) or (r > 4931) then  external32 representation
                                     is undefined
```

```
   else if (p > 15) or (r >  307) then  external32_size = 16
   else if (p >  6) or (r >   37) then  external32_size =  8
   else                                 external32_size =  4
```

MPI_TYPE_CREATE_F90_COMPLEXの場合， MPI_TYPE_CREATE_F90_REALの2倍のサイズとなる．

MPI_TYPE_CREATE_F90_INTEGERの場合は以下のようになる．

```
   if      (r > 38) then  external32 representation is undefined
   else if (r > 18) then  external32_size =  16
   else if (r >  9) then  external32_size =  8
   else if (r >  4) then  external32_size =  4
   else if (r >  2) then  external32_size =  2
   else                   external32_size =  1
```

データ型のexternal32表現が定義されていない場合，external32表現が必要な操作で直接的または間接的に（つまり，別のデータ型の一部として，または複製されたデータ型を通して）データ型を使用した結果は未定義となる．“external32”データ表現を使用する場合，これらの操作にはMPI_PACK_EXTERNAL，MPI_UNPACK_EXTERNALと多くのMPI_FILE関数が含まれる．external32表現が未定義となっている範囲は，将来の標準化のために予約されている．

特定のサイズのMPIデータ型のサポート

MPIには，オプションのFortran 77言語の数値型に対応する，明確なバイト長（MPI_REAL4，MPI_INTEGER8など）の名前付きのデータ型が用意されている．この節では，このモデルを汎用化して全てのFortran言語の数値基本型をサポートするためのメカニズムについて説明する．

各**typeclass**（integer, real, complex）および各ワードサイズに対して，一意のマシン表現があると仮定する．コンパイラでサポートされる全てのペア(**typeclass**，**n**)について， MPIは名前付きのサイズ指定データ型を用意する必要がある．このデータ型の名前はC言語およびFortran言語ではMPI_<TYPE>n，C++言語ではMPI::<TYPE>nとなり，<TYPE> はREAL，INTEGER，COMPLEXのいずれか， **n**はマシン表現のバイト単位での長さとなる．このデータ型はローカルで全ての(**typeclass**, **n**)型の変数と一致する．このような型の名前には以下のようなものがある．

```
MPI_REAL4
MPI_REAL8
MPI_REAL16
MPI_COMPLEX8
MPI_COMPLEX16
MPI_COMPLEX32
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_INTEGER8
MPI_INTEGER16
```

コンパイラによってサポートされる各表現に対して1つのデータ型が必要となる．MPI-1でのこれらの型の解釈との下位互換のため，非標準の宣言REAL*n，INTEGER*nで必

ず表現がサイズ**n**である変数が生成されると仮定する．これらのデータ型は全て定義済みである．

以下の関数を使用すると，Fortran言語基本型のためのサイズを指定したMPIデータ型を取得することができる．

MPI_SIZEOF(x, size)

IN	x	Fortran言語の整数基本型の変数（選択型）
OUT	size	その型のマシン表現のサイズ（整数型）

```
MPI_SIZEOF(X, SIZE, IERROR)
    <type> X
    INTEGER SIZE, IERROR
```

この関数は，渡された変数のマシン表現のサイズをバイト単位で返す．これはFortran言語の汎用ルーチンで，Fortran言語の呼び出し形式しかない．

ユーザへのアドバイス この関数はC言語およびC++言語の*sizeof*演算子と類似しているが，動作は多少異なる．配列引数を渡された場合，配列全体のサイズではなく，基本要素のサイズを返す． （**ユーザへのアドバイス終わり**）

根拠 この関数は他の言語では有益でないため，利用できない． （**根拠の終わり**）

MPI_TYPE_MATCH_SIZE(typeclass, size, type)

IN	typeclass	汎用型指定子（整数型）
IN	size	表現のサイズ（バイト単位）（整数型）
OUT	type	正しい型とサイズを持つデータ型（ハンドル）

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, TYPE, IERROR)
    INTEGER TYPECLASS, SIZE, TYPE, IERROR
{static MPI::Datatype MPI::Datatype::Match_size(int typeclass, int size)
                （廃止された呼び出し形式, 第15.2節を参照）}
```

typeclassは， MPI_TYPECLASS_REAL， MPI_TYPECLASS_INTEGER， MPI_TYPECLASS_COMPLEXのいずれかとなる．この関数は(**typeclass**， **size**)型のローカル変数に一致するMPIデータ型を返す．

この関数は，いずれかの定義済みの名前付きデータ型の，複製ではなく，参照（ハンドル）を返す．この型は解放できない．MPI_TYPE_MATCH_SIZEは，最初に変数のサイズの計算のためにMPI_SIZEOFを呼び出し，次に適切なデータ型を探すためにMPI_TYPE_MATCH_SIZEを呼び出すことにより，Fortran言語の数値基本型に一致するサイズ指定型を取得するのに使用できる．C言語およびC++言語では，MPI_SIZEOFの代わりにC言語の演算子[4]sizeof()を使用することができる．また，デフォルトのKINDの変数の場合，変数のサイズは`typeclass`が分かっていればMPI_TYPE_GET_EXTENTを呼

[4]訳者註：MPI-2.2 の原文は“function”であるが，“operator”の誤り

び出すことにより計算することができる．コンパイラでサポートされていないサイズを指定するのは誤りである．

根拠 利便性の高い関数である．この関数がなければ，正しい名前付きの型を探すのが煩雑な作業になる可能性がある．下記の実装者へのアドバイスを参照すること．（根拠の終わり）

実装者へのアドバイス この関数は一連のテストとして実装することができる．

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *rtype)
{
  switch(typeclass) {
      case MPI_TYPECLASS_REAL: switch(size) {
        case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
        case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
        default: error(...);
      }
      case MPI_TYPECLASS_INTEGER: switch(size) {
         case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
         case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
         default: error(...);       }
     ... etc. ...
   }
}
```

（実装者へのアドバイス終わり）

サイズ指定型との通信

通常の型一致規則はサイズ指定データ型にも適用され，データ型MPI_<TYPE>nを使用して送信された値は別のプロセスの同じデータ型を使用して受信できる．最新型のコンピュータでは，整数に2の補数を使用し，浮動小数点数にIEEE形式を使用している．そのため，これらのサイズ指定データ型を使用した通信では精度の損失や丸め誤差は発生しない．

ユーザへのアドバイス 異機種環境での通信には注意が必要である．以下のコードを検討してみる．

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
    ... initialize x ...
    call MPI_SEND(x, xtype, 100, 1, ...)
else if (myrank .eq. 1) then
    call MPI_RECV(x, xtype, 100, 0, ...)
endif
```

プロセス1とプロセス0でsizeの値が異なる場合，これは異機種環境で正しく動作しないことがある．同一機種環境では問題ない．異機種環境での通信には

少なくとも4つのオプションがある．最初の方法は，デフォルト型の変数を宣言し，これらの型のMPIデータ型を使用する方法である．例えば，REAL型の変数を宣言し，MPI_REALを使用する．2番目の方法は，selected_real_kindまたはselected_int_kindを前の節の関数と一緒に使用する方法である．3番目の方法は，全てのアーキテクチャで同じサイズであると分かっている変数を宣言する方法である（例えば，ほとんど全てのコンパイラでselected_real_kind(12)を使用すると8バイトの表現となる）．4番目の方法は，通信の前に表現のサイズを慎重にチェックする方法である．このために，通信が可能なサイズの変数への明示的な変換と送信側と受信側でサイズを合意するためのハンドシェイクが必要になる．

入出力のために“external32”表現を使用する場合，表現のサイズを明確に意識する必要があることに注意．以下のコードを検討してみる．

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)

if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',                &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,     &
                      MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif

call MPI_BARRIER(MPI_COMM_WORLD, ierror)

if (myrank .eq. 1) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,  &
                  MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
```

プロセス0とプロセス1が別のマシン上にあり，2台のマシンでsizeが異なる場合，このコードは期待通りに動作しないことがある．（**ユーザへのアドバイス終わり**）

16.3 言語の相互運用性

16.3.1 はじめに

ライブラリ開発者はある言語でアプリケーションライブラリを開発するが，そのライブラリが別の言語で記述されたアプリケーションプログラムによって呼び出されることが少なくない．MPIでは現在，ISO（以前はANSI）C言語，C++言語，Fortran言語の呼び出し形式をサポートしている．MPIがサポートしている言語で書かれたアプリケーシ

ョンが別の言語で書かれた MPI に関連する関数を呼び出すことが可能でなければならない.

さらに，MPIを使用してクライアント／サーバコードを開発することができる．このとき，並列クライアントと並列サーバの間でMPI によって通信することができる．サーバとクライアントをそれぞれ異なる言語でコーディングすることができなければならない．そのためには，異なる言語で記述されたアプリケーション間での通信が行えなければならない.

相互運用性を達成するには，いくつかの問題に対処する必要がある.

初期化 全ての言語に対してMPI環境を初期化する方法を指定する必要がある.

言語間でのMPI不可視オブジェクトの受け渡し 言語間でMPIオブジェクトハンドルを受け渡す方法を指定する必要がある．また，別の言語で設定された情報（属性など）を取得するために，ある言語でMPIオブジェクトにアクセスした場合の動作を指定する必要がある.

言語間通信 ある言語で送信されたメッセージを別の言語で受信する方法を指定する必要がある.

MPIの呼び出し形式が新しい言語に対して定義される場合，言語間の相互運用性のソリューションがこれらの言語にも拡張されることが望ましい.

16.3.2 前提

ある言語で記述されたプログラムが別の言語で記述されたルーチンを呼び出すための規則があることを前提とする．これらの規則では，異なる言語の複数のルーチンを1つのプログラムにリンクする方法，異なる言語で関数を呼び出す方法，言語間で引数を渡す方法，異なる言語の基本データ型の間での対応が規定される．一般的に，これらの規則は実装内容に応じて決まる．また，全ての基本データ型が別の言語で一致する型を持っているとは限らない．例えば．C言語／C++言語の文字列はFortran言語の`CHARACTER`の変数と互換性がない場合もある．しかし，Fortran言語の`INTEGER`と（シーケンスが関連付けられた）Fortran言語の`INTEGER`配列はC言語またはC++言語のプログラムに渡すことができる．また，Fortran言語，C言語，C++言語でアドレスサイズの整数があることを前提とする．このことは，デフォルトサイズの整数がデフォルトサイズのポインタと同じサイズであることを意味するのではなく，C言語のアドレスをFortran言語の整数型に保持（して渡す）する方法がある．また，`INTEGER(KIND=MPI_OFFSET_KIND)`をFortran言語からC言語にMPI_Offsetとして渡せることも前提とする.

16.3.3 初期化

いずれかの言語でMPI_INITまたはMPI_INIT_THREADを呼び出すと，全ての言語でMPIが実行できるよう初期化される.

ユーザへのアドバイス 実装では MPI_INITのC言語／C++言語バージョンの（inout） argc，argv引数を使用してargcおよびargvの値を全ての実行プロセスに伝播することがある．MPI_INITのFortran言語バージョンを使用してMPIを初期化するとこの機能が失われることがある． （**ユーザへのアドバイス終わり**）

関数MPI_INITIALIZEDは全ての言語で同じ答えを返す．
関数MPI_FINALIZEは全ての言語のMPI環境を完了させる．
関数MPI_FINALIZEDは全ての言語で同じ答えを返す．
関数MPI_ABORTは，呼び出し元によって，または終了させるプロセスによって使用されている言語に関係なく，プロセスを終了させる．
MPI環境はMPI_INITにより，全ての言語に対して同様に初期化される．例えば，MPI_COMM_WORLDは言語に関係なく同じ情報，例えば同じプロセス，同じ環境属性，同じエラーハンドラを持つ．
ある言語でinfoオブジェクトに追加した情報を，別の言語で取得することができる．

ユーザへのアドバイス 1つのMPIプログラムで複数の言語を使用するには，コンパイル時やリンク時に特別なオプションを使用することが必要になる場合がある．（**ユーザへのアドバイス終わり**）

実装者へのアドバイス 実装では，1つの言語のみを使用するコードでバイナリのサイズが大きくならないように，言語固有のMPIライブラリを必要とするコードにのみこれを選択的にリンクすることができる．MPI初期化コードは，ある言語のライブラリがロードされる場合のみ，その言語の初期化を行う必要がある． （**実装者へのアドバイス終わり**）

16.3.4 ハンドルの転送

ハンドルの受け渡しは，Fortran言語のハンドルからC言語のハンドルに変換するための明示的なC言語のラッパーを使用することにより，Fortran言語とC言語またはC++言語の間で行うできる．Fortran言語でC言語またはC++言語のハンドルに直接アクセスすることはできない．C言語とC++言語の間のハンドルの受け渡しは，C++言語のコードから呼び出されるオーバーロードされたC++言語の演算子を使用して行う．C言語からC++言語のオブジェクトに直接アクセスすることはできない．
Fortran言語のINTEGERと一致するサイズの整数用に，C言語／C++言語では型定義MPI_Fintが用意されている．通常，MPI_Fint はintと同等である．
C言語では，Fortran言語のコミュニケータハンドル（整数型）からC言語のコミュニケータハンドルへの変換，また逆の変換のため，以下の関数が用意されている．25ページの第2.6.5節も参照すること．

```
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
```

commがコミュニケータへの有効なFortran言語のハンドルの場合，MPI_Comm_f2cは同じコミュニケータへの有効なC言語のハンドルを返す．comm= MPI_COMM_NULL

（Fortran言語の値）であれば，MPI_Comm_f2cはnullのC言語ハンドルを返し，commが無効なFortran言語のハンドルであれば，MPI_Comm_f2cは無効なC言語のハンドルを返す．

```
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
```

関数MPI_Comm_c2fは，C言語のコミュニケータハンドルを同じコミュニケータへのFortran言語のハンドルに変換する．nullハンドルはnullハンドルにマッピングされ，無効なハンドルは無効なハンドルにマッピングされる．

他の型の不可視オブジェクト用にも同様の関数が用意されている．

```
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
MPI_Group MPI_Group_f2c(MPI_Fint group)
MPI_Fint MPI_Group_c2f(MPI_Group group)
MPI_Request MPI_Request_f2c(MPI_Fint request)
MPI_Fint MPI_Request_c2f(MPI_Request request)
MPI_File MPI_File_f2c(MPI_Fint file)
MPI_Fint MPI_File_c2f(MPI_File file)
MPI_Win MPI_Win_f2c(MPI_Fint win)
MPI_Fint MPI_Win_c2f(MPI_Win win)
MPI_Op MPI_Op_f2c(MPI_Fint op)
MPI_Fint MPI_Op_c2f(MPI_Op op)
MPI_Info MPI_Info_f2c(MPI_Fint info)
MPI_Fint MPI_Info_c2f(MPI_Info info)
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
```

例 16.13 下の例で，ハンドルの変換を行うC言語のラッパーを使用してC言語のMPI関数MPI_Type_commitをラッピングすることにより，Fortran言語のMPI関数MPI_TYPE_COMMITを実装できる方法を示す．この例では，Fortran言語とC言語のインターフェイスにおいてC言語からの参照時にFortran言語の関数が全て大文字で，引数がアドレスで渡されることが前提となる．

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT( DATATYPE, IERR)
INTEGER DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END



/* C wrapper */

void MPI_X_TYPE_COMMIT( MPI_Fint *f_handle, MPI_Fint *ierr)
{
   MPI_Datatype datatype;

   datatype = MPI_Type_f2c( *f_handle);
   *ierr = (MPI_Fint)MPI_Type_commit( &datatype);
```

```
    *f_handle = MPI_Type_c2f(datatype);
    return;
}
```

その他の全てのMPI関数でも同じアプローチをとることができる．ハンドルがINOUTではなく，OUT（またはIN）である場合，MPI_xxx_f2c（またはMPI_xxx_c2f）の呼び出しを省略することができる．

> **根拠** ここで設計には一般的なケース用，つまりC言語のラッパーを使用してFortran言語のコードでC言語のライブラリを呼び出す，またはC言語のコードでFortran言語のライブラリを呼び出すための便利なソリューションが用意されている．C言語のラッパーが使用される可能性のほうがFortran言語のラッパーが使用される可能性よりも高い．これは，**整数型**の変数がC言語に渡される可能性のほうがC言語のハンドルがFortran言語に渡される可能性よりも高いためである．
>
> 変換された値を引数リストを通してでなく，関数の値として返すと，これらの関数が単純な（恒等関数などに）場合に効率的なインラインコードを生成することができる．ラッパーの変換関数は無効なハンドル引数を捕捉しない．その代わり，無効なハンドルは下位のライブラリ関数に渡され，おそらくそこで入力引数がチェックされる． （**根拠の終わり**）

C言語及びC++言語 C++言語のインターフェイスでは，言語間の相互運用性のために以下の関数が用意されている．明記した場合を除いて，有効なMPI不可視ハンドルの名前（Groupなど）を示すため，以下のトークン<CLASS>が使用される．<CLASS>に対応するC++言語のクラスに派生クラスがある場合のために，派生クラスとC言語のMPI_<CLASS>の間の変換を行うための関数も用意されている．

以下の関数ではC言語のMPIハンドルからC++言語のMPIハンドルへの代入が行える．

```
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)
```

以下のコンストラクタはC言語のMPIハンドルからC++言語のMPIオブジェクトを生成する．これにより，C言語のMPIハンドルからC++言語のMPIハンドルへの変更が自動的に行われる．

```
MPI::<CLASS>::<CLASS>(const MPI_<CLASS>& data)
```

例 16.14 C言語のプログラムでC++言語のライブラリを使用するには，C++言語のライブラリが，下層のC++言語のライブラリを呼び出す前に適切な変換を行うためのC言語のインターフェイスをエクスポートする必要がある．この例では，C言語のコミュニケータを使用してC++言語のライブラリを呼び出すC言語のインターフェイス関数を示す．下層のC++言語関数が呼び出されると，自動的にコミュニケータがC++言語のハンドルに変更される．

```
// C++ library function prototype
void cpp_lib_call(MPI::Comm cpp_comm);
```

```
// Exported C function prototype
extern "C" {
   void c_interface(MPI_Comm c_comm);
}

void c_interface(MPI_Comm c_comm)
{
   // the MPI_Comm (c_comm) is automatically promoted to MPI::Comm
   cpp_lib_call(c_comm);
}
```

以下の関数ではC++言語のオブジェクトからC言語のMPIハンドルへの変換が行える．この場合，この機能を提供するためにキャスト演算子がオーバーロードされる．

```
MPI::<CLASS>::operator MPI_<CLASS>() const
```

例 16.15 C言語のライブラリルーチンがC++言語のプログラムから呼び出される．C言語のライブラリルーチンは，MPI_Commを引数として取るようにプロトタイプが規定されている．

```
// C function prototype
extern "C" {
   void c_lib_call(MPI_Comm c_comm);
}

void cpp_function()
{
   // Create a C++ communicator, and initialize it with a dup of
   //   MPI::COMM_WORLD
   MPI::Intracomm cpp_comm(MPI::COMM_WORLD.Dup());
   c_lib_call(cpp_comm);
}
```

根拠 コンストラクタによるC言語からC++言語への変換，またキャストによるC++言語からC言語への変換を行うと，コンパイラで自動的に変換が行われる．C言語またはFortran言語のインターフェイスをC++言語のライブラリに渡す場合と同様に，C++言語からC言語を呼び出すのが容易になる． （**根拠の終わり**）

ユーザへのアドバイス キャストおよび昇格演算子は 新しいハンドルを値で返すことに注意すること．新しいハンドルをINOUTパラメータとして使用すると内部のMPIオブジェクトには影響を及ぼすが，キャストされた元のハンドルには影響を及ぼさない． （**ユーザへのアドバイス終わり**）

対応するC言語のハンドルを持つC++言語の全てのオブジェクトはアプリケーションによって互換的に使用できることに注意する必要がある．例えば，アプリケーションはMPI_COMM_WORLDに属性をキャッシュしておいて，後でMPI::COMM_WORLDから取得することができる．

16.3.5 ステータス

C言語では，Fortran言語のステータス（整数の配列）からC言語のステータス（構造体）への変換，また逆の変換のため，以下の2つのプロシージャが用意されている．変換は，隠されているものも含めて，ステータス内の全ての情報について行われる．つまり，変換で失われるステータスの情報はない．

```
int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)
```

f_statusが有効なFortran言語のステータスであって，MPI_STATUS_IGNOREまたはMPI_STATUSES_IGNOREのFortran言語の値でない場合，MPI_Status_f2cは同じ内容を持つ有効なC言語のステータスをc_statusに返す．f_statusがMPI_STATUS_IGNOREまたはMPI_STATUSES_IGNOREのFortran言語の値である場合，あるいはf_statusが有効なFortran言語のステータスでない場合，呼び出しは誤りである．

C言語のステータスはFortran言語のステータスと同じソース，タグ，エラーコードの値を持ち，カウント，要素，取消しの問い合わせが行われた場合に同じ答えを返す．変換関数の呼び出しは，未定義のエラーフィールドを持つFortran言語のステータス引数を使用して行うことができるが，この場合，C言語のステータス引数のエラーフィールドの値は未定義となる．

MPI_F_STATUS_IGNORE，およびMPI_F_STATUSES_IGNOREの2つのMPI_Fint*型のグローバル変数はmpi.hで宣言されている．これらはそれぞれ，f_statusがMPI_STATUS_IGNOREまたはMPI_STATUSES_IGNOREのFortran言語の値であるかどうかをC言語でテストするために使用できる．これらはグローバル変数であり，C言語の定数式ではないため，C言語で定数式が必要となる状況で使用することはできない．この値はMPI_INIT呼び出しとMPI_FINALIZE呼び出しの間にのみ定義され，ユーザコードによって変更されることはない．

逆方向の変換を行うために以下の関数が用意されている．

```
int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)
```

この呼び出しはC言語のステータスをFortran言語のステータスに変換するもので．動作はMPI_Status_f2cと同様である．つまり，c_statusの値はMPI_STATUS_IGNOREであってもMPI_STATUSES_IGNOREであってもならない．

> **ユーザへのアドバイス** ステータスの配列については，配列内の各ステータスを単に順に変換していくだけで済むため，配列専用の変換関数は用意されていない．（**ユーザへのアドバイス終わり**）

> **根拠** C言語のラッパーのみを使用してライブラリを階層化するため，MPI_STATUS_IGNOREの処理が必要となる．Fortran言語の呼び出しでMPI_STATUS_IGNOREが渡されている場合，C言語のラッパーでこれを正しく処理する必要がある．この定数は，Fortran言語とC言語で同じ値を持つ必要はない．MPI_Status_f2cでMPI_STATUS_IGNOREを処理することになっていた場合，その結果の型はMPI_Status**でなければならず，これは不十分なソリューションと考えられた．（**根拠の終わり**）

16.3.6 MPIの不可視オブジェクト

特に規定がない限り，不可視オブジェクトは全ての言語で「共通」で，同じ情報が含まれ，両方の言語で意味も同じである．前の節で説明したメカニズムを使用して，言語間でMPIオブジェクトの参照を渡すことができる．ある言語で生成されたオブジェクトは，別の言語でアクセス，変更，解放することができる．

以下で，MPIオブジェクトの各型で発生する問題を詳しく検証する．

データ型

データ型は全ての言語で同じ情報をエンコードする．例えば，MPI_TYPE_GET_EXTENTなどのデータ型アクセサは全ての言語で同じ情報を返す．ある言語で定義されたデータ型が別の言語での通信呼び出しに使用される場合，送信されるメッセージは最初の言語から送信されるメッセージと同じになり，同じ通信バッファがアクセスされ，必要に応じて同じ表現変換が行われる．定義済みの全てのデータ型は任意の言語のデータ型コンストラクタで使用することができる．データ型がコミットされた場合，任意の言語の通信で使用することができる．

関数MPI_GET_ADDRESSは全ての言語で同じ値を返す．ただし，定数MPI_BOTTOMは全ての言語で同じ値を持つ必要はない（533ページの第16.3.9節を参照）．

例 16.16

```
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) AODISP(1)

! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN,AODISP,AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)



/* C code */

void C_ROUTINE(MPI_Fint *ftype)
{
   int count = 5;
   int lens[2] = {1,1};
   MPI_Aint displs[2];
   MPI_Datatype types[2], newtype;

   /* create an absolute datatype for buffer that consists   */
   /*  of count, followed by R(5)                            */

   MPI_Get_address(&count, &displs[0]);
   displs[1] = 0;
   types[0] = MPI_INT;
   types[1] = MPI_Type_f2c(*ftype);
```

```
   MPI_Type_create_struct(2, lens, displs, types, &newtype);
   MPI_Type_commit(&newtype);

   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   /* the message sent contains an int count of 5, followed  */
   /* by the 5 REAL entries of the Fortran array R.          */
}
```

実装者へのアドバイス 以下に示すような実装が可能である．
MPI_GET_ADDRESSによって返されるMPIアドレスは全ての言語で同じ値となる．明らかな設計方針は，MPIアドレスと通常のアドレスを同じにことである．絶対アドレスを持つデータ型が構成されている場合，このアドレスがデータ型に格納される．送信または受信操作が行われると，データ型に格納されたアドレスは全てベースアドレスが加算された変位として解釈される．ベースアドレスはbuf（のアドレス）で，buf = MPI_BOTTOMの場合は0である．そのため，MPI_BOTTOMが0の場合，buf = MPI_BOTTOMの送信または受信呼び出しはのアドレスを持つバッファ引数を持つ呼び出しとして正確に実装され，どちらの場合もベースアドレスはbufとなる．それに対して，MPI_BOTTOM が0でない場合，多少異なる実装が必要となる．buf = MPI_BOTTOMであるかどうかをチェックするためのテストが行われる．真の場合，ベースアドレスは0となり，偽の場合はbufとなる．特に，MPI_BOTTOMの値がFortran言語C言語／C++言語とで異なる場合，buf = MPI_BOTTOMであるかどうかをチェックするための追加テストが少なくとも1つの言語で必要となる．

C言語／C++言語でも，nullポインタと区別するため，MPI_BOTTOMに0以外の値を使用するのが望ましい場合がある．MPI_BOTTOM = cの場合，
MPI_GET_ADDRESSによって返されて絶対データ型に格納されるMPIアドレスとしてMPI_BOTTOMからの変位（通常のアドレス - c）を使用することにより，buf = MPI_BOTTOMのテストを避けることができる． （**実装者へのアドバイス終わり**）

コールバック関数

MPI呼び出しではコールバック関数とMPIオブジェクトを関連付けることができる．例えば，エラーハンドラがコミュニケータおよびファイルと関連付けられ，属性のコピーおよび削除関数が属性キーと関連付けられ，リデュース操作が操作オブジェクトと関連付けられる．多言語環境で，ある言語のMPI呼び出しで渡された関数は別の言語のMPI関数で呼び出すことができる．MPI実装では，このような呼び出しが，関数が記述された言語の呼び出し規則に従うようにする必要がある．

実装者へのアドバイス コールバック関数には言語タグを設定する必要がある．このタグはライブラリ関数によってコールバック関数が渡されるときに設定される(言語毎に異なることが望ましい)．そして，コールバック関数の呼び出し時に正

しい呼び出しシーケンスを実行するために使用される．（実装者へのアドバイス終わり）

エラーハンドラ

実装者へのアドバイス C言語およびC++言語では，エラーハンドラは“stdargs”引数リストを持つ．エラーが発生した言語環境に関する情報をハンドラに渡すのは有益である．（実装者へのアドバイス終わり）

リデュース操作

ユーザへのアドバイス リデュース操作はその引数の1つとして，オペランドのデータ型を受信する．そのため，C言語，C++言語，Fortran言語のデータ型で有効な「多様型」リデュース操作を定義することができる．（ユーザへのアドバイス終わり）

アドレス

一部のデータ型アクセサおよびコンストラクタは，アドレスを保持するための（C言語）または（C++言語）型の引数を持つ．Fortran言語の対応する引数はINTEGER型となる．このため，アドレスが64ビットであるのに対してFortran言語のINTEGERが32ビットである環境において，Fortran言語とC言語／C++言語の間で不整合が生じる．

これは言語間の課題に関係なく，問題である．Fortran言語のプロセスに4GB以上のアドレス空間があるとする．変数のアドレスが2^{32}を超える場合，Fortran言語でMPI_ADDRESSによってどのような値が返されるだろうか．ここに示す設計は，現在のFortran言語のコードとの互換性を維持しつつ，この問題に対処する．

定数MPI_ADDRESS_KINDは，Fortran 90言語でINTEGER(KIND=MPI_ADDRESS_KIND)がアドレスサイズの整数となるように定義されている（通常，INTEGER(KIND=MPI_ADDRESS_KIND)のサイズは32 ビットアドレスマシンでは4，64ビットアドレスマシンでは8だが，必ずしもそうなってはいない）．同様に，定数MPI_INTEGER_KINDはINTEGER(KIND=MPI_INTEGER_KIND)がデフォルトサイズのINTEGERとなるように定義されている．

アドレス引数を持つ関数には7つあり，それぞれMPI_TYPE_HVECTOR，MPI_TYPE_HINDEXED，MPI_TYPE_STRUCT，MPI_ADDRESS，MPI_TYPE_EXTENT，MPI_TYPE_LB，MPI_TYPE_UBである．

このリストの最初の4つの関数を補完するために新しい4つの関数が用意されている，これらの関数については，89ページの第4.1.1節で説明している．残りの3つの関数は，新しい関数MPI_TYPE_GET_EXTENT（同じ節で説明）によって補完される．新しい関数の機能は，C言語／C++言語では古い関数と同じであり，Fortran言語システムでは，デフォルトのINTEGERがアドレスサイズであれば古い関数と同じになる．Fortran言語では，C言語およびC++言語でMPI_AintおよびMPI::Aint型の引数が使用される状況では，INTEGER(KIND=MPI_ADDRESS_KIND)型の引数を使用できる．Fortran 77言語システムのうち，Fortran 90言語のKINDの表記法をサポートしておらず，デフォルトINTEGERが32ビッ

トであるにもかかわらずアドレスが64ビットであるものは，これらの引数は適切な整数型となる．下位互換のため，古い関数も引き続き提供される．しかし，Fortran言語では，アドレスの範囲が2^{32}を超えるシステムでの問題を防止するため，また言語間の互換性を保証するため，ユーザは新しい関数への切り替えが推奨される．

16.3.7 属性

ある言語で割り当てた属性キーを，別の言語で開放することができる．同様に，ある言語で設定した属性値を別の言語でアクセスすることができる．そのために，属性キーは全ての言語で有効な整数範囲で割り当てられる．システムで定義された属性値（MPI_TAG_UB， MPI_WTIME_IS_GLOBALなど）についても同様である．

ある言語で宣言された属性キーは，その言語のコピーおよび削除関数に関連付けられる （MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL呼び出しで提供される関数）．各属性に対してコミュニケータが複製される場合，その関数の言語のための正しい呼び出し規則を使用して対応するコピー関数が呼び出され，同様に削除コールバック関数も呼び出される．

> **実装者へのアドバイス** ここでは，コールバック関数のための正しい呼び出し規則を使用するために，属性が “C”，“C++”，または“Fortran”としてタグ付けされ，言語タグがチェックされなければならない． （**実装者へのアドバイス終わり**）

234ページの第6.7節で説明した属性操作関数は，属性引数をC言語ではvoid*型として，Fortran言語ではINTEGER型として定義する．一部のシステムでは，INTEGERが32ビットであるのに対して，C言語／C++言語のポインタが64ビットとなる．このことは，コミュニケータ属性を使用してFortran言語からC言語／C++言語を呼び出した場合，またはC言語／C++言語からFortran言語を呼び出した場合に問題となる．

MPIは内部的にアドレスサイズの属性を格納するように動作する．Fortran言語のINTEGERが小さい場合，Fortran言語の関数MPI_ATTR_GETは属性ワードの下位32ビットを返し，Fortran言語の関数MPI_ATTR_PUT は属性ワードの下位32ビットを設定する．これはワード全体に符号拡張される（これらの2つの関数はユーザコードによって明示的に呼び出されるか，または属性コピーコールバック関数によって暗黙的に呼び出される）．

アドレスの場合と同様，Fortran言語のアドレスサイズの属性を操作するための新しい関数が用意されており，機能はC言語／C++言語の古い関数と同じである．これらの関数については，234ページの第6.7節で説明している．ユーザはこれらの新しい関数の使用が推奨される．

MPIでは，2つのタイプの属性，つまりアドレス値（ポインタ）属性と整数値属性がサポートされている．C言語およびC++言語の属性関数はアドレス値属性をプット／ゲットする．Fortran言語の属性関数は整数値属性をプット／ゲットする．C言語またはC++言語から整数値属性へのアクセスが行われた場合，MPI_xxx_get_attrは整数値属性の（ポインタの）アドレスを返す．これは，属性がFortran言語の

MPI_xxx_SET_ATTRで格納された場合はMPI_Aintのポインタとなり，廃止されたFortran言語のMPI_ATTR_PUTで格納された場合はintのポインタとなる．Fortran言語からアドレス値属性へのアクセスが行われた場合，MPI_xxx_GET_ATTRはアドレスを整数に変換し，この変換の結果を返す．新しい形式の属性関数を使用した場合，この変換で欠落する情報はなく，MPI_ADDRESS_KINDの整数が返される．廃止された属性関数が使用される場合，変換で切り詰めが発生することがある．C言語では，廃止されたルーチンMPI_Attr_putおよびMPI_Attr_get は，MPI_Comm_set_attrおよびMPI_Comm_get_attrと同様に動作する．

例 16.17 A. C言語での属性値の設定

```
int set_val = 3;
struct foo set_struct;

/* Set a value that is a pointer to an int */

MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
/* Set a value that is a pointer to a struct */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
/* Set an integer value */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
```

B. C言語での属性値の読み取り

```
int flag, *get_val;
struct foo *get_struct;

/* Upon successful return, get_val == &set_val
   (and therefore *get_val == 3) */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
/* Upon successful return, get_struct == &set_struct */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
/* Upon successful return, get_val == (void*) 17 */
/*        i.e., (MPI_Aint) get_val == 17 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);
```

C. （廃止された）Fortran言語のMPI-1呼び出しでの属性値の読み取り

```
LOGICAL FLAG
INTEGER IERR, GET_VAL, GET_STRUCT

! Upon successful return, GET_VAL == &set_val, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
```

D. Fortran言語のMPI-2呼び出しでの属性値の読み取り

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT

! Upon successful return, GET_VAL == &set_val
```

```
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
```

例 **16.18** A. （廃止された）Fortran言語のMPI-1呼び出しでの属性値の設定

```
INTEGER IERR, VAL
VAL = 7
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
```

B. C言語での属性値の読み取り

```
int flag;
int *value;

/* Upon successful return, value points to internal MPI storage and
   *value == (int) 7 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
```

C. （廃止された）Fortran言語のMPI-1呼び出しでの属性値の読み取り

```
LOGICAL FLAG
INTEGER IERR, VALUE

! Upon successful return, VALUE == 7
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
```

D. Fortran言語のMPI-2呼び出しでの属性値の読み取り

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE

! Upon successful return, VALUE == 7 (sign extended)
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
```

例 **16.19** A. Fortran言語のMPI-2読み取りによる属性値の設定

```
INTEGER IERR
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
VALUE1 = 42
VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40

CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
```

B. C言語での属性値の読み取り

```
int flag;
MPI_Aint *value1, *value2;

/* Upon successful return, value1 points to internal MPI storage and
   *value1 == 42 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
/* Upon successful return, value2 points to internal MPI storage and
   *value2 == 2^40 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
```

C. （廃止された）Fortran言語のMPI-1呼び出しでの属性値の読み取り

```
LOGICAL FLAG
INTEGER IERR, VALUE1, VALUE2

! Upon successful return, VALUE1 == 42
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
! Upon successful return, VALUE2 == 2^40, or 0 if truncation
! needed (i.e., the least significant part of the attribute word)
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

D. Fortran言語のMPI-2呼び出しでの属性値の読み取り

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2

! Upon successful return, VALUE1 == 42
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
! Upon successful return, VALUE2 == 2^40
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

定義済みのMPI属性は整数値またはアドレス値となる．MPI_TAG_UBなどの定義済みの整数値属性は，廃止されたFortran言語のルーチンMPI_ATTR_PUTの呼び出しによってプットされたのと同様に動作する．つまり，Fortran言語ではMPI_COMM_GET_ATTR(MPI_COMM_WORLD, MPI_TAG_UB, val, flag, ierr)はタグ値の上限をvalに返し，C言語ではMPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &p, &flag)はタグ値の上限を持つintのポインタをpに返す．

MPI_WIN_BASEなどの定義済みのアドレス値属性は，C言語の呼び出しによってプットされたのと同様に動作する．つまり，Fortran言語ではMPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag, ierror)は整数型に変換されたウィンドウのベースアドレスをvalに返す．C言語ではMPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag)は(void *)にキャストされたウィンドウのベースアドレスのポインタをpに返す．

根拠 この設計は定義済みの属性用に指定された動作と整合し，言語間で属性が受け渡されるときに情報が欠落しないことを保証している．定義済みの属性のための言語の相互運用性はMPI_ATTR_PUTに基づいて定義されているため，ルーチン自体は推奨されなくなっているが，互換性のためにこの定義が残されている． （**根拠の終わり**）

実装者へのアドバイス 実装では，(1)C言語（MPI_Attr_putまたはMPI_Xxx_set_attrを使用），(2)Fortran言語のMPI_XXX_SET_ATTR，(3)廃止されたFortran言語のルーチンMPI_ATTR_PUTのいずれで設定されたかに従って，(1)アドレス属性，(2)INTEGER(KIND=MPI_ADDRESS_KIND)属性，または (3)INTEGER属性として属性のタグ付けを行う必要がある．そのため，属性の取得時に正しい選択が行える．（**実装者へのアドバイス終わり**）

16.3.8 追加ステート

追加ステートはコピーまたは削除コールバック関数によって変更されないようにする必要がある（これはC言語の呼び出し形式では明らかだが，Fortran言語の呼び出し形式では明らかでない）．しかし，これらの関数は追加ステートによって間接的にアクセスされる状態を更新することがある．例えば，C言語では追加ステートを，コピーまたはコールバック関数によって変更されるデータ構造体のポインタとすることができ，Fortran言語では追加ステートを，コピーまたはコールバック関数によって変更されるCOMMON配列のエントリの添字とすることができる．マルチスレッド環境では，異なるスレッドが同じコールバック関数を同時に実行する場合があることを知っておく必要がある．この関数が追加ステートに関係する状態を変更した場合，相互排他コードを使用して，共有される状態の更新やアクセスを保護する必要がある．

16.3.9 定数

MPIの定数は，特に規定がない限り，全ての言語で同じ値を持つ．このことは，定数ハンドル（MPI_INT, MPI_COMM_WORLD, MPI_ERRORS_RETURN,MPI_SUMなど）には適用されない．これらのハンドルは，第16.3.4節で説明するように変換する必要がある．最大文字列長を指定する定数（付録A.1.1のリストを参照）は，Fortran言語ではC言語／C++言語よりも1文字分小さな値となるが，これはC言語／C++言語では長さにnull終了文字が含まれるためである．そのため，これらの定数は，文字列に含まれうる印字可能文字の最大数ではなく，使用する可能性のある最大の文字列を保持するために割り当てる必要のある領域の量を表わす．

ユーザへのアドバイス この定義は，C言語／C++言語で以下のような宣言を使用して文字列の受信バッファを割り当てても安全であることを意味している．

```
char name [MPI_MAX_OBJECT_NAME];
```

（**ユーザへのアドバイス終わり**）

また，MPI_BOTTOMやMPI_STATUS_IGNOREのような定数「アドレス」，つまりハンドルでない参照引数のための特別な値は，言語毎に異なっていても構わない．

根拠 現在のMPI標準では，MPI_BOTTOMをC言語では初期化式に使用できるが，Fortran言語では使用できない．通常，Fortran言語では値による呼び出しがサポー

トされていないため，MPI_BOTTOMはFortran言語では定義済みのstatic変数，例えばMPIで宣言されたCOMMONブロックの変数でなければならない．それに対してC言語では，MPI_BOTTOM = 0とすることが自然である（警告：MPI_BOTTOM = 0と定義すると，nullポインタをMPI_BOTTOMと区別できないことを意味するため，MPI_BOTTOM = 1とした方がよい）．Fortran言語とC言語の値を同じにしようとすると，初期化のプロセスが複雑になる．（**根拠の終わり**）

16.3.10 言語間の通信

MPIでの通信のための型一致規則は変更されておらず，送信される各項目のデータ型の指定は，型シグネチャにおいて，（型の1つがMPI_PACKEDである場合を除いて）この項目を受信するためのデータ型の指定と一致していなければならない．また，型がMPI_BYTEまたはMPI_PACKEDである場合を除いて，メッセージ項目の型が対応する通信バッファ領域の型宣言と一致していなければならない．これらの規則に従っている場合，言語間の通信を行うことができる．

例 16.20 以下の例では，Fortran言語の配列がFortran言語から送信され，C言語で受信される．

```
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, IERR, MYRANK, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) AODISP(1)

! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS( R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN,AODISP,AOTYPE, TYPE, IERR)
CALL MPI_TYPE_COMMIT(TYPE, IERR)

CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
   CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
ELSE
   CALL C_ROUTINE(TYPE)
END IF



/* C code */

void C_ROUTINE(MPI_Fint *fhandle)
{
   MPI_Datatype type;
   MPI_Status status;

   type = MPI_Type_f2c(*fhandle);

   MPI_Recv( MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPIの実装者は，それぞれの型が一致している場合，型一致規則を緩和し，Fortran言語の型を使用してメッセージを送信し，C言語の型を使用して受信する．またはその逆を行うことができる．つまり，Fortran言語の型INTEGERがC言語の型intと同じである場合，MPI実装ではデータ型MPI_INTEGERを使用してデータを送信し，データ型MPI_INTを使用してデータを受信することができる．しかし，このようなコードは可搬ではない．

付録A章

言語呼び出し形式要約

この章では，C言語，Fortran言語そしてC++言語特有の 呼び出し形式を要約する．最初に定数，型宣言，info値そしてキーについて記述する．その後呼び出し形式毎にルーチンプロトタイプを示す.

A.1　定義された値とハンドル

A.1.1　定義された定数

C言語とFortran言語の名前を左の列に列挙し，C++言語の名前を 中央か 右の列に列挙する． `const int`型の定数は，プリプロセッサが整数定数リテラルに置き換えて実装してもよい.

戻り値

C言語型: `const int` (または名前無し`enum`) Fortran言語型: `INTEGER`	C++言語型: `const int` (または名前無し`enum`)
MPI_SUCCESS	MPI::SUCCESS
MPI_ERR_BUFFER	MPI::ERR_BUFFER
MPI_ERR_COUNT	MPI::ERR_COUNT
MPI_ERR_TYPE	MPI::ERR_TYPE
MPI_ERR_TAG	MPI::ERR_TAG
MPI_ERR_COMM	MPI::ERR_COMM
MPI_ERR_RANK	MPI::ERR_RANK
MPI_ERR_REQUEST	MPI::ERR_REQUEST
MPI_ERR_ROOT	MPI::ERR_ROOT
MPI_ERR_GROUP	MPI::ERR_GROUP
MPI_ERR_OP	MPI::ERR_OP
MPI_ERR_TOPOLOGY	MPI::ERR_TOPOLOGY
MPI_ERR_DIMS	MPI::ERR_DIMS
MPI_ERR_ARG	MPI::ERR_ARG
MPI_ERR_UNKNOWN	MPI::ERR_UNKNOWN
MPI_ERR_TRUNCATE	MPI::ERR_TRUNCATE
MPI_ERR_OTHER	MPI::ERR_OTHER
MPI_ERR_INTERN	MPI::ERR_INTERN
MPI_ERR_PENDING	MPI::ERR_PENDING
MPI_ERR_IN_STATUS	MPI::ERR_IN_STATUS
MPI_ERR_ACCESS	MPI::ERR_ACCESS
MPI_ERR_AMODE	MPI::ERR_AMODE
MPI_ERR_ASSERT	MPI::ERR_ASSERT
MPI_ERR_BAD_FILE	MPI::ERR_BAD_FILE
MPI_ERR_BASE	MPI::ERR_BASE
MPI_ERR_CONVERSION	MPI::ERR_CONVERSION
MPI_ERR_DISP	MPI::ERR_DISP
MPI_ERR_DUP_DATAREP	MPI::ERR_DUP_DATAREP
MPI_ERR_FILE_EXISTS	MPI::ERR_FILE_EXISTS
MPI_ERR_FILE_IN_USE	MPI::ERR_FILE_IN_USE
MPI_ERR_FILE	MPI::ERR_FILE
MPI_ERR_INFO_KEY	MPI::ERR_INFO_VALUE
MPI_ERR_INFO_NOKEY	MPI::ERR_INFO_NOKEY
MPI_ERR_INFO_VALUE	MPI::ERR_INFO_KEY
MPI_ERR_INFO	MPI::ERR_INFO

（次ページに続く）

戻り値 （続き）

MPI_ERR_IO	MPI::ERR_IO
MPI_ERR_KEYVAL	MPI::ERR_KEYVAL
MPI_ERR_LOCKTYPE	MPI::ERR_LOCKTYPE
MPI_ERR_NAME	MPI::ERR_NAME
MPI_ERR_NO_MEM	MPI::ERR_NO_MEM
MPI_ERR_NOT_SAME	MPI::ERR_NOT_SAME
MPI_ERR_NO_SPACE	MPI::ERR_NO_SPACE
MPI_ERR_NO_SUCH_FILE	MPI::ERR_NO_SUCH_FILE
MPI_ERR_PORT	MPI::ERR_PORT
MPI_ERR_QUOTA	MPI::ERR_QUOTA
MPI_ERR_READ_ONLY	MPI::ERR_READ_ONLY
MPI_ERR_RMA_CONFLICT	MPI::ERR_RMA_CONFLICT
MPI_ERR_RMA_SYNC	MPI::ERR_RMA_SYNC
MPI_ERR_SERVICE	MPI::ERR_SERVICE
MPI_ERR_SIZE	MPI::ERR_SIZE
MPI_ERR_SPAWN	MPI::ERR_SPAWN
MPI_ERR_UNSUPPORTED_DATAREP	MPI::ERR_UNSUPPORTED_DATAREP
MPI_ERR_UNSUPPORTED_OPERATION	MPI::ERR_UNSUPPORTED_OPERATION
MPI_ERR_WIN	MPI::ERR_WIN
MPI_ERR_LASTCODE	MPI::ERR_LASTCODE

バッファアドレス定数

C言語型: `void * const` Fortran言語型: （定義済みメモリ位置）	C++言語型: `void * const`
MPI_BOTTOM	MPI::BOTTOM
MPI_IN_PLACE	MPI::IN_PLACE

様々な定数

C言語型: `const int` (または名前無し`enum`)	C++言語型:
Fortran言語型: `INTEGER`	`const int` (または名前無し`enum`)
MPI_PROC_NULL	MPI::PROC_NULL
MPI_ANY_SOURCE	MPI::ANY_SOURCE
MPI_ANY_TAG	MPI::ANY_TAG
MPI_UNDEFINED	MPI::UNDEFINED
MPI_BSEND_OVERHEAD	MPI::BSEND_OVERHEAD
MPI_KEYVAL_INVALID	MPI::KEYVAL_INVALID
MPI_LOCK_EXCLUSIVE	MPI::LOCK_EXCLUSIVE
MPI_LOCK_SHARED	MPI::LOCK_SHARED
MPI_ROOT	MPI::ROOT

ステータスのサイズと予約されたインデックス値（**Fortran**言語のみ）

Fortran言語型: `INTEGER`	
MPI_STATUS_SIZE	C++言語では定義されない
MPI_SOURCE	C++言語では定義されない
MPI_TAG	C++言語では定義されない
MPI_ERROR	C++言語では定義されない

可変のアドレスサイズ（**Fortran**言語のみ）

Fortran言語型: `INTEGER`	
MPI_ADDRESS_KIND	C++言語では定義されない
MPI_INTEGER_KIND	C++言語では定義されない
MPI_OFFSET_KIND	C++言語では定義されない

エラー処理指示子

C言語型: `MPI_Errhandler`	C++言語型: `MPI::Errhandler`
Fortran言語型: `INTEGER`	
MPI_ERRORS_ARE_FATAL	MPI::ERRORS_ARE_FATAL
MPI_ERRORS_RETURN	MPI::ERRORS_RETURN
	MPI::ERRORS_THROW_EXCEPTIONS

文字列の最大サイズ

C言語型: `const int` (または名前無し`enum`) Fortran言語型: `INTEGER`	C++言語型: `const int` (または名前無し`enum`)
MPI_MAX_PROCESSOR_NAME	MPI::MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING	MPI::MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING	MPI::MAX_DATAREP_STRING
MPI_MAX_INFO_KEY	MPI::MAX_INFO_KEY
MPI_MAX_INFO_VAL	MPI::MAX_INFO_VAL
MPI_MAX_OBJECT_NAME	MPI::MAX_OBJECT_NAME
MPI_MAX_PORT_NAME	MPI::MAX_PORT_NAME

名前付き定義済みデータ型		C言語／C++言語型
C言語型: MPI_Datatype Fortran言語型: INTEGER	C++言語型: MPI::Datatype	
MPI_CHAR	MPI::CHAR	char （印字可能文字として 扱われる）
MPI_SHORT	MPI::SHORT	signed short int
MPI_INT	MPI::INT	signed int
MPI_LONG	MPI::LONG	signed long
MPI_LONG_LONG_INT	MPI::LONG_LONG_INT	signed long long
MPI_LONG_LONG	MPI::LONG_LONG	long long （同義）
MPI_SIGNED_CHAR	MPI::SIGNED_CHAR	signed char （整数値として扱われる）
MPI_UNSIGNED_CHAR	MPI::UNSIGNED_CHAR	unsigned char （整数値として扱われる）
MPI_UNSIGNED_SHORT	MPI::UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	MPI::UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	MPI::UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	MPI::UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	MPI::FLOAT	float
MPI_DOUBLE	MPI::DOUBLE	double
MPI_LONG_DOUBLE	MPI::LONG_DOUBLE	long double
MPI_WCHAR	MPI::WCHAR	wchar_t （<stddef.h>で定義される） （印字可能文字として 扱われる）
MPI_C_BOOL	（C言語のハンドル型を使用）	_Bool
MPI_INT8_T	（C言語のハンドル型を使用）	int8_t
MPI_INT16_T	（C言語のハンドル型を使用）	int16_t
MPI_INT32_T	（C言語のハンドル型を使用）	int32_t
MPI_INT64_T	（C言語のハンドル型を使用）	int64_t
MPI_UINT8_T	（C言語のハンドル型を使用）	uint8_t
MPI_UINT16_T	（C言語のハンドル型を使用）	uint16_t
MPI_UINT32_T	（C言語のハンドル型を使用）	uint32_t
MPI_UINT64_T	（C言語のハンドル型を使用）	uint64_t
MPI_AINT	（C言語のハンドル型を使用）	MPI_Aint
MPI_OFFSET	（C言語のハンドル型を使用）	MPI_Offset
MPI_C_COMPLEX	（C言語のハンドル型を使用）	float _Complex
MPI_C_FLOAT_COMPLEX	（C言語のハンドル型を使用）	float _Complex
MPI_C_DOUBLE_COMPLEX	（C言語のハンドル型を使用）	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	（C言語のハンドル型を使用）	long double _Complex
MPI_BYTE	MPI::BYTE	（任意のC言語／C++言語型）
MPI_PACKED	MPI::PACKED	（任意のC言語／C++言語型）

名前付き定義済みデータ型		**Fortran言語型**
C言語型: `MPI_Datatype`	C++言語型: `MPI::Datatype`	
Fortran言語型: `INTEGER`		
MPI_INTEGER	MPI::INTEGER	`INTEGER`
MPI_REAL	MPI::REAL	`REAL`
MPI_DOUBLE_PRECISION	MPI::DOUBLE_PRECISION	`DOUBLE PRECISION`
MPI_COMPLEX	MPI::F_COMPLEX	`COMPLEX`
MPI_LOGICAL	MPI::LOGICAL	`LOGICAL`
MPI_CHARACTER	MPI::CHARACTER	`CHARACTER(1)`
MPI_AINT	（C言語のハンドル型を使用）	`INTEGER (KIND=MPI_ADDRESS_KIND)`
MPI_OFFSET	（C言語のハンドル型を使用）	`INTEGER (KIND=MPI_OFFSET_KIND)`
MPI_BYTE	MPI::BYTE	（任意のFortran言語型）
MPI_PACKED	MPI::PACKED	（任意のFortran言語型）

名前付き定義済みデータ型	**C++言語型**
C言語型: MPI_Datatype	
Fortran言語型: INTEGER	
またはTYPE(MPI_Datatype)	
MPI_CXX_BOOL	`bool`
MPI_CXX_COMPLEX	`std::complex<float>`
MPI_CXX_DOUBLE_COMPLEX	`std::complex<double>`
MPI_CXX_LONG_DOUBLE_COMPLEX	`std::complex<long double>`

オプションのデータ型 （Fortran言語）		**Fortran言語型**
C言語型: `MPI_Datatype` Fortran言語型: `INTEGER`	C++言語型: `MPI::Datatype`	
MPI_DOUBLE_COMPLEX	MPI::F_DOUBLE_COMPLEX	`DOUBLE COMPLEX`
MPI_INTEGER1	MPI::INTEGER1	`INTEGER*1`
MPI_INTEGER2	MPI::INTEGER2	`INTEGER*8`
MPI_INTEGER4	MPI::INTEGER4	`INTEGER*4`
MPI_INTEGER8	MPI::INTEGER8	`INTEGER*8`
MPI_INTEGER16	MPI::INTEGER16	`INTEGER*16`
MPI_REAL2	MPI::REAL2	`REAL*2`
MPI_REAL4	MPI::REAL4	`REAL*4`
MPI_REAL8	MPI::REAL8	`REAL*8`
MPI_REAL16	MPI::F_REAL16	`REAL*16`
MPI_COMPLEX4	MPI::F_COMPLEX4	`COMPLEX*4`
MPI_COMPLEX8	MPI::F_COMPLEX8	`COMPLEX*8`
MPI_COMPLEX16	MPI::F_COMPLEX16	`COMPLEX*16`
MPI_COMPLEX32	MPI::F_COMPLEX32	`COMPLEX*32`

リダクション関数のためのデータ型 （**C言語／C++言語**）

C言語型: `MPI_Datatype` Fortran言語型: `INTEGER`	C++言語型: `MPI::Datatype`
MPI_FLOAT_INT	MPI::FLOAT_INT
MPI_DOUBLE_INT	MPI::DOUBLE_INT
MPI_LONG_INT	MPI::LONG_INT
MPI_2INT	MPI::TWOINT
MPI_SHORT_INT	MPI::SHORT_INT
MPI_LONG_DOUBLE_INT	MPI::LONG_DOUBLE_INT

リダクション関数のためのデータ型 （**Fortran言語**）

C言語型: `MPI_Datatype` Fortran言語型: `INTEGER`	C++言語型: `MPI::Datatype`
MPI_2REAL	MPI::TWOREAL
MPI_2DOUBLE_PRECISION	MPI::TWODOUBLE_PRECISION
MPI_2INTEGER	MPI::TWOINTEGER

派生データ型作成のための特別なデータ型

C言語型: MPI_Datatype Fortran言語型: INTEGER	C++言語型: MPI::Datatype
MPI_UB	MPI::UB
MPI_LB	MPI::LB

予約されたコミュニケータ

C言語型: MPI_Comm Fortran言語型: INTEGER	C++言語型: MPI::Intracomm
MPI_COMM_WORLD	MPI::COMM_WORLD
MPI_COMM_SELF	MPI::COMM_SELF

コミュニケータとグループ比較の結果

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_IDENT	MPI::IDENT
MPI_CONGRUENT	MPI::CONGRUENT
MPI_SIMILAR	MPI::SIMILAR
MPI_UNEQUAL	MPI::UNEQUAL

環境問い合わせのキー

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_TAG_UB	MPI::TAG_UB
MPI_IO	MPI::IO
MPI_HOST	MPI::HOST
MPI_WTIME_IS_GLOBAL	MPI::WTIME_IS_GLOBAL

集団操作

C言語型: MPI_Op	C++言語型: const MPI::Op
Fortran言語型: INTEGER	
MPI_MAX	MPI::MAX
MPI_MIN	MPI::MIN
MPI_SUM	MPI::SUM
MPI_PROD	MPI::PROD
MPI_MAXLOC	MPI::MAXLOC
MPI_MINLOC	MPI::MINLOC
MPI_BAND	MPI::BAND
MPI_BOR	MPI::BOR
MPI_BXOR	MPI::BXOR
MPI_LAND	MPI::LAND
MPI_LOR	MPI::LOR
MPI_LXOR	MPI::LXOR
MPI_REPLACE	MPI::REPLACE

nullハンドル

C言語／Fortran言語での名前 C言語／Fortran言語での型	C++言語での名前 C++言語型
MPI_GROUP_NULL MPI_Group / INTEGER	MPI::GROUP_NULL const MPI::Group
MPI_COMM_NULL MPI_Comm / INTEGER	MPI::COMM_NULL [1])
MPI_DATATYPE_NULL MPI_Datatype / INTEGER	MPI::DATATYPE_NULL const MPI::Datatype
MPI_REQUEST_NULL MPI_Request / INTEGER	MPI::REQUEST_NULL const MPI::Request
MPI_OP_NULL MPI_Op / INTEGER	MPI::OP_NULL const MPI::Op
MPI_ERRHANDLER_NULL MPI_Errhandler / INTEGER	MPI::ERRHANDLER_NULL const MPI::Errhandler
MPI_FILE_NULL MPI_File / INTEGER	MPI::FILE_NULL
MPI_INFO_NULL MPI_Info / INTEGER	MPI::INFO_NULL const MPI::Info
MPI_WIN_NULL MPI_Win / INTEGER	MPI::WIN_NULL

[1]) C++言語型: クラス階層とMPI::COMM_NULLの型については495ページの第16.1.7節を参照

空のグループ

C言語型: MPI_Group Fortran言語型: INTEGER	C++言語型: const MPI::Group
MPI_GROUP_EMPTY	MPI::GROUP_EMPTY

トポロジー

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_GRAPH	MPI::GRAPH
MPI_CART	MPI::CART
MPI_DIST_GRAPH	MPI::DIST_GRAPH

定義済み機能

C言語／Fortran言語での名前 C言語／Fortran言語型	C++言語での名前 C++言語型
MPI_COMM_NULL_COPY_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_FN	MPI_COMM_NULL_COPY_FN C言語と同じ[1])
MPI_COMM_DUP_FN MPI_Comm_copy_attr_function / COMM_COPY_ATTR_FN	MPI_COMM_DUP_FN C言語と同じ[1])
MPI_COMM_NULL_DELETE_FN MPI_Comm_delete_attr_function / COMM_DELETE_ATTR_FN	MPI_COMM_NULL_DELETE_FN C言語と同じ[1])
MPI_WIN_NULL_COPY_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_FN	MPI_WIN_NULL_COPY_FN C言語と同じ[1])
MPI_WIN_DUP_FN MPI_Win_copy_attr_function / WIN_COPY_ATTR_FN	MPI_WIN_DUP_FN C言語と同じ[1])
MPI_WIN_NULL_DELETE_FN MPI_Win_delete_attr_function / WIN_DELETE_ATTR_FN	MPI_WIN_NULL_DELETE_FN C言語と同じ[1])
MPI_TYPE_NULL_COPY_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_FN	MPI_TYPE_NULL_COPY_FN C言語と同じ[1])
MPI_TYPE_DUP_FN MPI_Type_copy_attr_function / TYPE_COPY_ATTR_FN	MPI_TYPE_DUP_FN C言語と同じ[1])
MPI_TYPE_NULL_DELETE_FN MPI_Type_delete_attr_function / TYPE_DELETE_ATTR_FN	MPI_TYPE_NULL_DELETE_FN C言語と同じ[1])

[1] 236ページの第6.7.2節の MPI_COMM_NULL_COPY_FNなどに関する実装者へのアドバイス参照

廃止された定義済み関数

C言語／Fortran言語での名前 C言語／Fortran言語型	C++言語での名前 C++言語型
MPI_NULL_COPY_FN MPI_Copy_function / COPY_FUNCTION	MPI::NULL_COPY_FN MPI::Copy_function
MPI_DUP_FN MPI_Copy_function / COPY_FUNCTION	MPI::DUP_FN MPI::Copy_function
MPI_NULL_DELETE_FN MPI_Delete_function / DELETE_FUNCTION	MPI::NULL_DELETE_FN MPI::Delete_function

定義済み属性キー

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_APPNUM	MPI::APPNUM
MPI_LASTUSEDCODE	MPI::LASTUSEDCODE
MPI_UNIVERSE_SIZE	MPI::UNIVERSE_SIZE
MPI_WIN_BASE	MPI::WIN_BASE
MPI_WIN_DISP_UNIT	MPI::WIN_DISP_UNIT
MPI_WIN_SIZE	MPI::WIN_SIZE

モード定数

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_MODE_APPEND	MPI::MODE_APPEND
MPI_MODE_CREATE	MPI::MODE_CREATE
MPI_MODE_DELETE_ON_CLOSE	MPI::MODE_DELETE_ON_CLOSE
MPI_MODE_EXCL	MPI::MODE_EXCL
MPI_MODE_NOCHECK	MPI::MODE_NOCHECK
MPI_MODE_NOPRECEDE	MPI::MODE_NOPRECEDE
MPI_MODE_NOPUT	MPI::MODE_NOPUT
MPI_MODE_NOSTORE	MPI::MODE_NOSTORE
MPI_MODE_NOSUCCEED	MPI::MODE_NOSUCCEED
MPI_MODE_RDONLY	MPI::MODE_RDONLY
MPI_MODE_RDWR	MPI::MODE_RDWR
MPI_MODE_SEQUENTIAL	MPI::MODE_SEQUENTIAL
MPI_MODE_UNIQUE_OPEN	MPI::MODE_UNIQUE_OPEN
MPI_MODE_WRONLY	MPI::MODE_WRONLY

データ型デコード定数

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_COMBINER_CONTIGUOUS	MPI::COMBINER_CONTIGUOUS
MPI_COMBINER_DARRAY	MPI::COMBINER_DARRAY
MPI_COMBINER_DUP	MPI::COMBINER_DUP
MPI_COMBINER_F90_COMPLEX	MPI::COMBINER_F90_COMPLEX
MPI_COMBINER_F90_INTEGER	MPI::COMBINER_F90_INTEGER
MPI_COMBINER_F90_REAL	MPI::COMBINER_F90_REAL
MPI_COMBINER_HINDEXED_INTEGER	MPI::COMBINER_HINDEXED_INTEGER
MPI_COMBINER_HINDEXED	MPI::COMBINER_HINDEXED
MPI_COMBINER_HVECTOR_INTEGER	MPI::COMBINER_HVECTOR_INTEGER
MPI_COMBINER_HVECTOR	MPI::COMBINER_HVECTOR
MPI_COMBINER_INDEXED_BLOCK	MPI::COMBINER_INDEXED_BLOCK
MPI_COMBINER_INDEXED	MPI::COMBINER_INDEXED
MPI_COMBINER_NAMED	MPI::COMBINER_NAMED
MPI_COMBINER_RESIZED	MPI::COMBINER_RESIZED
MPI_COMBINER_STRUCT_INTEGER	MPI::COMBINER_STRUCT_INTEGER
MPI_COMBINER_STRUCT	MPI::COMBINER_STRUCT
MPI_COMBINER_SUBARRAY	MPI::COMBINER_SUBARRAY
MPI_COMBINER_VECTOR	MPI::COMBINER_VECTOR

スレッド定数

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_THREAD_FUNNELED	MPI::THREAD_FUNNELED
MPI_THREAD_MULTIPLE	MPI::THREAD_MULTIPLE
MPI_THREAD_SERIALIZED	MPI::THREAD_SERIALIZED
MPI_THREAD_SINGLE	MPI::THREAD_SINGLE

ファイル操作定数（その1）

C言語型: const MPI_Offset (または名前無しenum) Fortran言語型: INTEGER (KIND=MPI_OFFSET_KIND)	C++言語型: const MPI::Offset (または名前無しenum)
MPI_DISPLACEMENT_CURRENT	MPI::DISPLACEMENT_CURRENT

ファイル操作定数（その2）

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_DISTRIBUTE_BLOCK	MPI::DISTRIBUTE_BLOCK
MPI_DISTRIBUTE_CYCLIC	MPI::DISTRIBUTE_CYCLIC
MPI_DISTRIBUTE_DFLT_DARG	MPI::DISTRIBUTE_DFLT_DARG
MPI_DISTRIBUTE_NONE	MPI::DISTRIBUTE_NONE
MPI_ORDER_C	MPI::ORDER_C
MPI_ORDER_FORTRAN	MPI::ORDER_FORTRAN
MPI_SEEK_CUR	MPI::SEEK_CUR
MPI_SEEK_END	MPI::SEEK_END
MPI_SEEK_SET	MPI::SEEK_SET

Fortran 90言語データ型マッチング定数

C言語型: const int (または名前無しenum) Fortran言語型: INTEGER	C++言語型: const int (または名前無しenum)
MPI_TYPECLASS_COMPLEX	MPI::TYPECLASS_COMPLEX
MPI_TYPECLASS_INTEGER	MPI::TYPECLASS_INTEGER
MPI_TYPECLASS_REAL	MPI::TYPECLASS_REAL

空または無視すべき入力を指定する定数

C言語／Fortran言語での名前 C言語／Fortran言語型	C++言語での名前 C++言語型
MPI_ARGVS_NULL char*** / CHARACTER*(*)の2次元配列	MPI::ARGVS_NULL const char ***
MPI_ARGV_NULL char** / CHARACTER*(*)の配列	MPI::ARGV_NULL const char **
MPI_ERRCODES_IGNORE int* / INTEGERの配列	C++言語では定義されない
MPI_STATUSES_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)	C++言語では定義されない
MPI_STATUS_IGNORE MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)	C++言語では定義されない
MPI_UNWEIGHTED	C++言語では定義されない

無視すべき入力を指定するC言語定数 （C++言語またはFortran言語には存在しない）
C言語型: MPI_Fint*
MPI_F_STATUSES_IGNORE
MPI_F_STATUS_IGNORE

C言語／C++言語プリプロセッサ定数とFortran言語パラメータ
C/C++言語型: const int (または名前無しenum)
Fortran言語型: INTEGER
MPI_SUBVERSION
MPI_VERSION

A.1.2 型

以下はmpi.hファイルに含むC言語の型定義である.

```
  /* C opaque types */
MPI_Aint
MPI_Fint
MPI_Offset
MPI_Status

/* C handles to assorted structures */
MPI_Comm
MPI_Datatype
MPI_Errhandler
MPI_File
MPI_Group
MPI_Info
MPI_Op
MPI_Request
MPI_Win

// C++ opaque types (all within the MPI namespace)
MPI::Aint
MPI::Offset
MPI::Status

// C++ handles to assorted structures (classes,
// all within the MPI namespace)
MPI::Comm
MPI::Intracomm
MPI::Graphcomm
MPI::Distgraphcomm
MPI::Cartcomm
MPI::Intercomm
MPI::Datatype
```

MPI::Errhandler
MPI::Exception
MPI::File
MPI::Group
MPI::Info
MPI::Op
MPI::Request
MPI::Prequest
MPI::Grequest
MPI::Win

A.1.3 プロトタイプ宣言

以下は，ユーザ定義関数のためのC言語のtypedefの定義であり，これらもmpi.hファイルに含まれている.

```
/* prototypes for user-defined functions */
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
             MPI_Datatype *datatype);

typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm,
             int comm_keyval, void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int*flag);
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm,
             int comm_keyval, void *attribute_val, void *extra_state);

typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
             void *extra_state, void *attribute_val_in,
             void *attribute_val_out, int *flag);
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
             void *attribute_val, void *extra_state);

typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
             int type_keyval, void *extra_state,
             void *attribute_val_in, void *attribute_val_out, int *flag);
typedef int MPI_Type_delete_attr_function(MPI_Datatype type,
             int type_keyval, void *attribute_val, void *extra_state);

typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);

typedef int MPI_Grequest_query_function(void *extra_state,
            MPI_Status *status);
typedef int MPI_Grequest_free_function(void *extra_state);
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);

typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
            MPI_Aint *file_extent, void *extra_state);
typedef int MPI_Datarep_conversion_function(void *userbuf,
```

```
            MPI_Datatype datatype, int count, void *filebuf,
            MPI_Offset position, void *extra_state);
```

Fortran言語における，ユーザ定義サブルーチンの宣言方法の例は以下の通りである.

MPI_OP_CREATEへのユーザ関数の引数は以下の様に宣言するべきである:

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
   <type> INVEC(LEN), INOUTVEC(LEN)
   INTEGER LEN, TYPE
```

MPI_COMM_CREATE_KEYVALへ渡す複製関数と削除関数の引数は以下の様に宣言するべきである:

```
SUBROUTINE COMM_COPY_ATTR_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
   INTEGER OLDCOMM, COMM_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT
   LOGICAL FLAG

SUBROUTINE COMM_DELETE_ATTR_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
   INTEGER COMM, COMM_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

MPI_WIN_CREATE_KEYVALへ渡す複製関数と削除関数の引数は以下の様に宣言するべきである:

```
SUBROUTINE WIN_COPY_ATTR_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
   INTEGER OLDWIN, WIN_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT
   LOGICAL FLAG

SUBROUTINE WIN_DELETE_ATTR_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
   INTEGER WIN, WIN_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

MPI_TYPE_CREATE_KEYVALへ渡す複製関数と削除関数の引数は以下の様に宣言するべきである:

```
SUBROUTINE TYPE_COPY_ATTR_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
   INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
   LOGICAL FLAG

SUBROUTINE TYPE_DELETE_ATTR_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
              EXTRA_STATE, IERROR)
   INTEGER TYPE, TYPE_KEYVAL, IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

MPI_COMM_CREATE_ERRHANDLERへのハンドラ関数引数は以下の様に宣言するべきである:

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
   INTEGER COMM, ERROR_CODE
```

MPI_WIN_CREATE_ERRHANDLERへのハンドラ関数引数は以下の様に宣言するべきである:

```
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
   INTEGER WIN, ERROR_CODE
```

MPI_FILE_CREATE_ERRHANDLERへのハンドラ関数引数は以下の様に宣言するべきである:

```
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
   INTEGER FILE, ERROR_CODE
```

MPI_GREQUEST_STARTへ渡すクエリ関数，開放関数，キャンセル関数の引数は以下の様に宣言するべきである:

```
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
   INTEGER STATUS(MPI_STATUS_SIZE), IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
   LOGICAL COMPLETE
```

MPI_REGISTER_DATAREPへ渡す拡張関数と変換関数の引数は以下の様に宣言するべきである:

```
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
             POSITION, EXTRA_STATE, IERROR)
    <TYPE> USERBUF(*), FILEBUF(*)
    INTEGER COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) POSITION
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

以下は，C++言語のtypedefの定義であり，これらもmpi.hファイルに含まれている.

```
namespace MPI {
  typedef void User_function(const void* invec, void *inoutvec,
              int len, const Datatype& datatype);

  typedef int Comm::Copy_attr_function(const Comm& oldcomm,
              int comm_keyval, void* extra_state, void* attribute_val_in,
              void* attribute_val_out, bool& flag);
  typedef int Comm::Delete_attr_function(Comm& comm, int
              comm_keyval, void* attribute_val, void* extra_state);

  typedef int Win::Copy_attr_function(const Win& oldwin,
              int win_keyval, void* extra_state, void* attribute_val_in,
              void* attribute_val_out, bool& flag);
  typedef int Win::Delete_attr_function(Win& win, int
              win_keyval, void* attribute_val, void* extra_state);

  typedef int Datatype::Copy_attr_function(const Datatype& oldtype,
              int type_keyval, void* extra_state,
              const void* attribute_val_in, void* attribute_val_out,
              bool& flag);
  typedef int Datatype::Delete_attr_function(Datatype& type,
              int type_keyval, void* attribute_val, void* extra_state);

  typedef void Comm::Errhandler_function(Comm &, int *, ...);
  typedef void Win::Errhandler_function(Win &, int *, ...);
  typedef void File::Errhandler_function(File &, int *, ...);

  typedef int Grequest::Query_function(void* extra_state, Status& status);
  typedef int Grequest::Free_function(void* extra_state);
  typedef int Grequest::Cancel_function(void* extra_state, bool complete);

  typedef void Datarep_extent_function(const Datatype& datatype,
               Aint& file_extent, void* extra_state);
  typedef void Datarep_conversion_function(void* userbuf,
               Datatype& datatype, int count, void* filebuf,
               Offset position, void* extra_state);
}
```

A.1.4 廃止されたプロトタイプ宣言

以下は，廃止されたユーザ定義関数のためのC言語のtypedefの定義であり，これらもmpi.hファイルに含まれている.

```
/* prototypes for user-defined functions */
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
              void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
              void *attribute_val, void *extra_state);
typedef void MPI_Handler_function(MPI_Comm *, int *, ...);
```

以下は，廃止されたFortran言語ユーザ定義コールバックサブルーチンプロトタイプである． MPI_KEYVAL_CREATEへ渡す廃止された複製関数と削除関数の引数は，以下のように宣言すべきである:

```
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
                ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
   INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
         ATTRIBUTE_VAL_OUT, IERR
   LOGICAL FLAG



SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

廃止された，エラーハンドラのためのハンドラ関数は以下の様に宣言すべきである:

```
SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE)
   INTEGER COMM, ERROR_CODE
```

A.1.5 Infoキー

access_style
appnum
arch
cb_block_size
cb_buffer_size
cb_nodes
chunked_item
chunked_size
chunked
collective_buffering
file_perm
filename
file
host
io_node_list
ip_address
ip_port
nb_proc
no_locks
num_io_nodes
path
soft
striping_factor
striping_unit
wdir

A.1.6 Info値

false
random
read_mostly
read_once
reverse_sequential
sequential
true
write_mostly
write_once

A.2 C言語呼び出し形式

A.2.1 1対1通信 C 言語呼び出し形式

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Buffer_attach(void* buffer, int size)
int MPI_Buffer_detach(void* buffer_addr, int* size)
int MPI_Cancel(MPI_Request *request)
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
MPI_Status *status)
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Request_free(MPI_Request *request)
int MPI_Request_get_status(MPI_Request request, int *flag,
MPI_Status *status)
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype,
int source, int recvtag, MPI_Comm comm, MPI_Status *status)
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Start(MPI_Request *request)
int MPI_Startall(int count, MPI_Request *array_of_requests)
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Test_cancelled(MPI_Status *status, int *flag)
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
MPI_Status *array_of_statuses)
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
int *flag, MPI_Status *status)
int MPI_Testsome(int incount, MPI_Request *array_of_requests,
int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Waitall(int count, MPI_Request *array_of_requests,
MPI_Status *array_of_statuses)
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
MPI_Status *status)
int MPI_Waitsome(int incount, MPI_Request *array_of_requests,
int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)
```

A.2.2 データ型 C 言語呼び出し形式

```
int MPI_Get_address(void *location, MPI_Aint *address)
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
int outsize, int *position, MPI_Comm comm)
int MPI_Pack_external(char *datarep, void *inbuf, int incount,
MPI_Datatype datatype, void *outbuf, MPI_Aint outsize, MPI_Aint *position)
int MPI_Pack_external_size(char *datarep, int incount,
MPI_Datatype datatype, MPI_Aint *size)
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
int *size)
int MPI_Type_commit(MPI_Datatype *datatype)
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)
int MPI_Type_create_darray(int size, int rank, int ndims,
int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int
array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_create_hindexed(int count, int array_of_blocklengths[],
MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
MPI_Datatype *newtype)
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_create_indexed_block(int count, int blocklength,
int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint
extent, MPI_Datatype *newtype)
int MPI_Type_create_struct(int count, int array_of_blocklengths[],
MPI_Aint array_of_displacements[], MPI_Datatype array_of_types[],
MPI_Datatype *newtype)
```

```
int MPI_Type_create_subarray(int ndims, int array_of_sizes[],
int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype
oldtype, MPI_Datatype *newtype)
int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)
int MPI_Type_free(MPI_Datatype *datatype)
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
int max_addresses, int max_datatypes, int array_of_integers[],
MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[])
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
int *num_addresses, int *num_datatypes, int *combiner)
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
MPI_Aint *extent)
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
MPI_Aint *true_extent)
int MPI_Type_indexed(int count, int *array_of_blocklengths,
int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_size(MPI_Datatype datatype, int *size)
int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
int outcount, MPI_Datatype datatype, MPI_Comm comm)
int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
MPI_Aint *position, void *outbuf, int outcount, MPI_Datatype datatype)
```

A.2.3 集団的通信 C 言語呼び出し形式

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,
MPI_Comm comm)
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls,
MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[], int rdispls[],
MPI_Datatype recvtypes[], MPI_Comm comm)
int MPI_Barrier(MPI_Comm comm)
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm )
int MPI_Exscan(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,
int root, MPI_Comm comm)
int MPI_Op_commutative(MPI_Op op, int *commute)
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Reduce_local(void* inbuf, void* inoutbuf, int count,
MPI_Datatype datatype, MPI_Op op)
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Reduce_scatter_block(void* sendbuf, void* recvbuf, int recvcount,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)
int MPI_op_free( MPI_Op *op)
```

A.2.4 グループ，コンテクスト，コミュニケータ，キャッシング C 言語呼び出し形式

```
int MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,
void *attribute_val_in, void *attribute_val_out, int *flag)
int MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval,
void *extra_state, void *attribute_val_in, void *attribute_val_out,
int *flag)
int MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval, void
*attribute_val, void *extra_state)
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
MPI_Comm_delete_attr_function *comm_delete_attr_fn, int *comm_keyval,
void *extra_state)
int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_free(MPI_Comm *comm)
int MPI_Comm_free_keyval(int *comm_keyval)
int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
int *flag)
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
```

```
int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
int MPI_Comm_set_name(MPI_Comm comm, char *comm_name)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_free(MPI_Group *group)
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
MPI_Group *newgroup)
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
MPI_Group *newgroup)
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_size(MPI_Group group, int *size)
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
MPI_Group group2, int *ranks2)
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
MPI_Comm *newintracomm)
int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,
void *extra_state, void *attribute_val_in, void *attribute_val_out,
int *flag)
int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval,
void *extra_state, void *attribute_val_in, void *attribute_val_out,
int *flag)
int MPI_TYPE_NULL_DELETE_FN(MPI_Datatype type, int type_keyval, void
*attribute_val, void *extra_state)
int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
MPI_Type_delete_attr_function *type_delete_attr_fn, int *type_keyval,
void *extra_state)
int MPI_Type_delete_attr(MPI_Datatype type, int type_keyval)
int MPI_Type_free_keyval(int *type_keyval)
int MPI_Type_get_attr(MPI_Datatype type, int type_keyval, void
*attribute_val, int *flag)
int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen)
int MPI_Type_set_attr(MPI_Datatype type, int type_keyval,
void *attribute_val)
int MPI_Type_set_name(MPI_Datatype type, char *type_name)
int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
void *attribute_val_in, void *attribute_val_out, int *flag)
```

```
int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state,
void *attribute_val_in, void *attribute_val_out, int *flag)
int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void
*attribute_val, void *extra_state)
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
MPI_Win_delete_attr_function *win_delete_attr_fn, int *win_keyval,
void *extra_state)
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
int MPI_Win_free_keyval(int *win_keyval)
int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val,
int *flag)
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
int MPI_Win_set_name(MPI_Win win, char *win_name)
```

A.2.5 プロセストポロジー C 言語呼び出し形式

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
int reorder, MPI_Comm *comm_cart)
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
int *coords)
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods,
int *newrank)
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
int *rank_source, int *rank_dest)
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
int MPI_Dims_create(int nnodes, int ndims, int *dims)
int MPI_Dist_graph_create(MPI_Comm comm_old, int n, int sources[],
int degrees[], int destinations[], int weights[], MPI_Info info,
int reorder, MPI_Comm *comm_dist_graph)
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
int sources[], int sourceweights[], int outdegree, int destinations[], int
destweights[], MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
int sourceweights[], int maxoutdegree, int destinations[],
int destweights[])
int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
int *outdegree, int *weighted)
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
int reorder, MPI_Comm *comm_graph)
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
int *edges)
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges,
int *newrank)
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
int *neighbors)
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
int MPI_Topo_test(MPI_Comm comm, int *status)
```

A.2.6 MPI環境管理 C 言語呼び出し形式

```
double MPI_Wtick(void)
double MPI_Wtime(void)
int MPI_Abort(MPI_Comm comm, int errorcode)
int MPI_Add_error_class(int *errorclass)
int MPI_Add_error_code(int errorclass, int *errorcode)
int MPI_Add_error_string(int errorcode, char *string)
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *function,
MPI_Errhandler *errhandler)
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
int MPI_Error_class(int errorcode, int *errorclass)
int MPI_Error_string(int errorcode, char *string, int *resultlen)
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
int MPI_File_create_errhandler(MPI_File_errhandler_function *function,
MPI_Errhandler *errhandler)
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
int MPI_Finalize(void)
int MPI_Finalized(int *flag)
int MPI_Free_mem(void *base)
int MPI_Get_processor_name(char *name, int *resultlen)
int MPI_Get_version(int *version, int *subversion)
int MPI_Init(int *argc, char ***argv)
int MPI_Initialized(int *flag)
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
int MPI_Win_create_errhandler(MPI_Win_errhandler_function *function,
MPI_Errhandler *errhandler)
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

A.2.7 Infoオブジェクト C 言語呼び出し形式

```
int MPI_Info_create(MPI_Info *info)
int MPI_Info_delete(MPI_Info info, char *key)
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
int MPI_Info_free(MPI_Info *info)
int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value,
int *flag)
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
```

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen,
int *flag)
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

A.2.8 プロセスの生成と管理 C 言語呼び出し形式

```
int MPI_Close_port(char *port_name)
int MPI_Comm_accept(char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_connect(char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_disconnect(MPI_Comm *comm)
int MPI_Comm_get_parent(MPI_Comm *parent)
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info
info, int root, MPI_Comm comm, MPI_Comm *intercomm,
int array_of_errcodes[])
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[],
int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)
int MPI_Open_port(MPI_Info info, char *port_name)
int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)
int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)
```

A.2.9 片方向通信 C 言語呼び出し形式

```
int MPI_Accumulate(void *origin_addr, int origin_count,
MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp,
int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int target_count,
MPI_Datatype target_datatype, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, MPI_Win *win)
int MPI_Win_fence(int assert, MPI_Win win)
int MPI_Win_free(MPI_Win *win)
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_test(MPI_Win win, int *flag)
int MPI_Win_unlock(int rank, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

A.2.10 外部インターフェイス C 言語呼び出し形式

```
int MPI_Grequest_complete(MPI_Request request)
int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function
*cancel_fn, void *extra_state, MPI_Request *request)
int MPI_Init_thread(int *argc, char *((*argv)[]), int required,
int *provided)
int MPI_Is_thread_main(int *flag)
int MPI_Query_thread(int *provided)
int MPI_Status_set_cancelled(MPI_Status *status, int flag)
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
int count)
```

A.2.11 入出力 C 言語呼び出し形式

```
int MPI_File_close(MPI_File *fh)
int MPI_File_delete(char *filename, MPI_Info info)
int MPI_File_get_amode(MPI_File fh, int *amode)
int MPI_File_get_atomicity(MPI_File fh, int *flag)
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
MPI_Offset *disp)
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
MPI_Aint *extent)
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
MPI_Datatype *filetype, char *datarep)
int MPI_File_iread(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
MPI_File *fh)
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
MPI_Status *status)
int MPI_File_read_all(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype)
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_read_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
int MPI_File_set_atomicity(MPI_File fh, int flag)
int MPI_File_set_info(MPI_File fh, MPI_Info info)
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
MPI_Datatype filetype, char *datarep, MPI_Info info)
int MPI_File_sync(MPI_File fh)
int MPI_File_write(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_all(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_all_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
int MPI_File_write_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype)
int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_write_ordered(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_write_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_Register_datarep(char *datarep,
MPI_Datarep_conversion_function *read_conversion_fn,
MPI_Datarep_conversion_function *write_conversion_fn,
MPI_Datarep_extent_function *dtype_file_extent_fn, void *extra_state)
```

A.2.12 言語呼び出し形式 C 言語呼び出し形式

```
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
MPI_Fint MPI_File_c2f(MPI_File file)
MPI_File MPI_File_f2c(MPI_Fint file)
MPI_Fint MPI_Group_c2f(MPI_Group group)
MPI_Group MPI_Group_f2c(MPI_Fint group)
MPI_Fint MPI_Info_c2f(MPI_Info info)
MPI_Info MPI_Info_f2c(MPI_Fint info)
MPI_Fint MPI_Op_c2f(MPI_Op op)
MPI_Op MPI_Op_f2c(MPI_Fint op)
MPI_Fint MPI_Request_c2f(MPI_Request request)
MPI_Request MPI_Request_f2c(MPI_Fint request)
int MPI_Status_c2f(MPI_Status *c_status, MPI_Fint *f_status)
int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Win_c2f(MPI_Win win)
MPI_Win MPI_Win_f2c(MPI_Fint win)
```

A.2.13 プロファイルインターフェイス C 言語呼び出し形式

```
int MPI_Pcontrol(const int level, ...)
```

A.2.14 廃止された C 言語呼び出し形式

```
int MPI_Address(void* location, MPI_Aint *address)
int MPI_Attr_delete(MPI_Comm comm, int keyval)
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
void *attribute_val_in, void *attribute_val_out, int *flag)
int MPI_Errhandler_create(MPI_Handler_function *function,
MPI_Errhandler *errhandler)
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
*delete_fn, int *keyval, void* extra_state)
int MPI_Keyval_free(int *keyval)
int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,
void *attribute_val_in, void *attribute_val_out, int *flag)
```

```
int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,
void *extra_state)
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
int MPI_Type_struct(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
MPI_Datatype *newtype)
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
```

A.3 Fortran言語呼び出し形式

A.3.1 1対1通信 Fortran 言語呼び出し形式

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
    <type> BUFFER(*)
    INTEGER SIZE, IERROR
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    <type> BUFFER_ADDR(*)
    INTEGER SIZE, IERROR
MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
MPI_REQUEST_GET_STATUS( REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

```
MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*)
    INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
    IERROR
MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

A.3.2 データ型 Fortran 言語呼び出し形式

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
POSITION, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    CHARACTER*(*) DATAREP
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER, OLDTYPE,
NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
    ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)
```

```
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
    NEWTYPE, IERROR
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
    INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
    ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
    INTEGER TYPE, NEWTYPE, IERROR
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES, IERROR)
    INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,
    ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND = MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
DATATYPE, IERROR)
    INTEGER OUTCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
```

A.3.3 集団的通信 Fortran 言語呼び出し形式

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, IERROR
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, IERROR
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
    LOGICAL COMMUTE
    INTEGER OP, IERROR
MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
    EXTERNAL FUNCTION
    LOGICAL COMMUTE
    INTEGER OP, IERROR
MPI_OP_FREE( OP, IERROR)
```

```
    INTEGER OP, IERROR
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
MPI_REDUCE_LOCAL(INBUF, INOUBUF, COUNT, DATATYPE, OP, IERROR)
    <type> INBUF(*), INOUTBUF(*)
    INTEGER COUNT, DATATYPE, OP, IERROR
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, IERROR
```

A.3.4 グループ，コンテクスト，コミュニケータ，キャッシング Fortran 言語呼び出し形式

```
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

```
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
    INTEGER COMM_KEYVAL, IERROR
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
    INTEGER COMM, RESULTLEN, IERROR
    CHARACTER*(*) COMM_NAME
MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) COMM_NAME
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
TAG, NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
    INTEGER INTERCOMM, INTRACOMM, IERROR
    LOGICAL HIGH
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
EXTRA_STATE, IERROR)
    EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
    INTEGER TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
MPI_TYPE_DELETE_ATTR(TYPE, TYPE_KEYVAL, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR
MPI_TYPE_GET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
MPI_TYPE_GET_NAME(TYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER TYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME
MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
MPI_TYPE_NULL_DELETE_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
IERROR)
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
MPI_TYPE_SET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
```

```
    INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
MPI_TYPE_SET_NAME(TYPE, TYPE_NAME, IERROR)
    INTEGER TYPE, IERROR
    CHARACTER*(*) TYPE_NAME
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
    CHARACTER*(*) WIN_NAME
MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
        ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
```

A.3.5 プロセストポロジー Fortran 言語呼び出し形式

```
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, COMM_DIST_GRAPH,
IERROR)
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
    INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
        DESTINATIONS(*), DESTWEIGHTS(*), IERROR
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
    INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
    LOGICAL WEIGHTED
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
IERROR)
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
    LOGICAL REORDER
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

A.3.6 MPI環境管理 Fortran 言語呼び出し形式

```
DOUBLE PRECISION MPI_WTICK()
DOUBLE PRECISION MPI_WTIME()
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) STRING
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
MPI_COMM_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
    INTEGER FH, ERRORCODE, IERROR
MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
MPI_FINALIZE(IERROR)
    INTEGER IERROR
MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR
MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN,IERROR
```

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
MPI_INIT(IERROR)
    INTEGER IERROR
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR

MPI_WIN_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

A.3.7 Infoオブジェクト Fortran 言語呼び出し形式

```
MPI_INFO_CREATE(INFO, IERROR)
    INTEGER INFO, IERROR
MPI_INFO_DELETE(INFO, KEY, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY
MPI_INFO_DUP(INFO, NEWINFO, IERROR)
    INTEGER INFO, NEWINFO, IERROR
MPI_INFO_FREE(INFO, IERROR)
    INTEGER INFO, IERROR
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    CHARACTER*(*) KEY, VALUE
    LOGICAL FLAG
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
    INTEGER INFO, NKEYS, IERROR
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
    INTEGER INFO, N, IERROR
    CHARACTER*(*) KEY
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    LOGICAL FLAG
    CHARACTER*(*) KEY
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY, VALUE
```

A.3.8 プロセスの生成と管理 Fortran 言語呼び出し形式

```
MPI_CLOSE_PORT(PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER IERROR
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
```

```
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
MPI_COMM_DISCONNECT(COMM, IERROR)
    INTEGER COMM, IERROR
MPI_COMM_GET_PARENT(PARENT, IERROR)
    INTEGER PARENT, IERROR
MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
    INTEGER FD, INTERCOMM, IERROR
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERROR
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES,
IERROR)
    INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, IERROR
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
```

A.3.9 片方向通信 Fortran 言語呼び出し形式

```
MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE,TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR
MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
```

```
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
    INTEGER WIN, GROUP, IERROR
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

A.3.10 外部インターフェイス Fortran 言語呼び出し形式

```
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
IERROR)
    INTEGER REQUEST, IERROR
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
    INTEGER REQUIRED, PROVIDED, IERROR
MPI_IS_THREAD_MAIN(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR
MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

A.3.11 入出力 Fortran 言語呼び出し形式

```
MPI_FILE_CLOSE(FH, IERROR)
    INTEGER FH, IERROR
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
    INTEGER FH, GROUP, IERROR
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
```

```
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER COMM, AMODE, INFO, FH, IERROR
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
```

```
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
MPI_FILE_SET_INFO(FH, INFO, IERROR)
    INTEGER FH, INFO, IERROR
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
    INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
MPI_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, IERROR
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
```

```
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER IERROR
```

A.3.12 言語呼び出し形式 Fortran 言語呼び出し形式

```
MPI_SIZEOF(X, SIZE, IERROR)
    <type> X
    INTEGER SIZE, IERROR
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
    INTEGER R, NEWTYPE, IERROR
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, TYPE, IERROR)
    INTEGER TYPECLASS, SIZE, TYPE, IERROR
```

A.3.13 プロファイルインターフェイス Fortran 言語呼び出し形式

```
MPI_PCONTROL(LEVEL)
    INTEGER LEVEL
```

A.3.14 廃止された Fortran 言語呼び出し形式

```
MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER ADDRESS, IERROR
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG
MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)
    EXTERNAL FUNCTION
    INTEGER ERRHANDLER, IERROR
MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

```
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, EXTRA_STATE, IERROR
MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR
MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG
MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
    INTEGER DATATYPE, EXTENT, IERROR
MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    OLDTYPE, NEWTYPE, IERROR
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
MPI_TYPE_LB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
ARRAY_OF_TYPES, NEWTYPE, IERROR)
    INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
    ARRAY_OF_TYPES(*), NEWTYPE, IERROR
MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)
    INTEGER DATATYPE, DISPLACEMENT, IERROR
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

A.4 C++言語呼び出し形式（廃止された）

A.4.1 1対1通信 C++ 言語呼び出し形式

```
namespace MPI {
```

{void Attach_buffer(void* buffer, int size) （廃止された呼び出し形式, 第15.2節を参照）
}

{void Comm::Bsend(const void* buf, int count, const Datatype& datatype,
int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{Request Comm::Bsend_init(const void* buf, int count, const
Datatype& datatype, int dest, int tag) const （廃止された呼び出し
形式, 第15.2節を参照）}

{void Request::Cancel() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Detach_buffer(void*& buffer) （廃止された呼び出し形式, 第15.2節を参照）}

{void Request::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{int Status::Get_count(const Datatype& datatype) const （廃止された呼び出し形式,
第15.2節を参照）}

{int Status::Get_error() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Status::Get_source() const （廃止された呼び出し形式, 第15.2節を参照）}

{bool Request::Get_status() const （廃止された呼び出し形式, 第15.2節を参照）}

{bool Request::Get_status(Status& status) const （廃止された呼び出し形式, 第15.2節を
参照）}

{int Status::Get_tag() const （廃止された呼び出し形式, 第15.2節を参照）}

{Request Comm::Ibsend(const void* buf, int count, const Datatype& datatype,
int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{bool Comm::Iprobe(int source, int tag) const （廃止された呼び出し形式, 第15.2節を参
照）}

{bool Comm::Iprobe(int source, int tag, Status& status) const （廃止された呼び
出し形式, 第15.2節を参照）}

{Request Comm::Irecv(void* buf, int count, const Datatype& datatype, int source, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{Request Comm::Irsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{bool Status::Is_cancelled() const （廃止された呼び出し形式, 第15.2節を参照）}

{Request Comm::Isend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{Request Comm::Issend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Probe(int source, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Probe(int source, int tag, Status& status) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag, Status& status) const （廃止された呼び出し形式, 第15.2節を参照）}

{Prequest Comm::Recv_init(void* buf, int count, const Datatype& datatype, int source, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Rsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{Prequest Comm::Rsend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Send(const void* buf, int count, const Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{Prequest Comm::Send_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Sendrecv(const void *sendbuf, int sendcount, const Datatype& sendtype, int dest, int sendtag, void *recvbuf,

```
              int recvcount, const Datatype& recvtype, int source,
              int recvtag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Sendrecv(const void *sendbuf, int sendcount, const
              Datatype& sendtype, int dest, int sendtag, void *recvbuf,
              int recvcount, const Datatype& recvtype, int source,
              int recvtag, Status& status) const （廃止された呼び出し形式, 第15.2節
              を参照）}

{void Comm::Sendrecv_replace(void* buf, int count, const
              Datatype& datatype, int dest, int sendtag, int source,
              int recvtag) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Sendrecv_replace(void* buf, int count, const
              Datatype& datatype, int dest, int sendtag, int source,
              int recvtag, Status& status) const （廃止された呼び出し形式, 第15.2節
              を参照）}

{void Status::Set_error(int error) （廃止された呼び出し形式, 第15.2節を参照）}

{void Status::Set_source(int source) （廃止された呼び出し形式, 第15.2節を参照）}

{void Status::Set_tag(int tag) （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Ssend(const void* buf, int count, const Datatype& datatype,
              int dest, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{Prequest Comm::Ssend_init(const void* buf, int count, const
              Datatype& datatype, int dest, int tag) const （廃止された呼び出し
              形式, 第15.2節を参照）}

{void Prequest::Start() （廃止された呼び出し形式, 第15.2節を参照）}

{static void Prequest::Startall(int count, Prequest array_of_requests[])
              （廃止された呼び出し形式, 第15.2節を参照）}

{bool Request::Test() （廃止された呼び出し形式, 第15.2節を参照）}

{bool Request::Test(Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{static bool Request::Testall(int count, Request array_of_requests[]) （廃止
              された呼び出し形式, 第15.2節を参照）}

{static bool Request::Testall(int count, Request array_of_requests[],
              Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}
```

{static bool Request::Testany(int count, Request array_of_requests[], int& index) （廃止された呼び出し形式, 第15.2節を参照）}

{static bool Request::Testany(int count, Request array_of_requests[], int& index, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Request::Testsome(int incount, Request array_of_requests[], int array_of_indices[]) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Request::Testsome(int incount, Request array_of_requests[], int array_of_indices[], Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}

{void Request::Wait() （廃止された呼び出し形式, 第15.2節を参照）}

{void Request::Wait(Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{static void Request::Waitall(int count, Request array_of_requests[]) （廃止された呼び出し形式, 第15.2節を参照）}

{static void Request::Waitall(int count, Request array_of_requests[], Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Request::Waitany(int count, Request array_of_requests[]) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Request::Waitany(int count, Request array_of_requests[], Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Request::Waitsome(int incount, Request array_of_requests[], int array_of_indices[]) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Request::Waitsome(int incount, Request array_of_requests[], int array_of_indices[], Status array_of_statuses[]) （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.2 データ型 C++ 言語呼び出し形式

namespace MPI {

{void Datatype::Commit() （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_contiguous(int count) const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_darray(int size, int rank, int ndims,
const int array_of_gsizes[], const int array_of_distribs[],
const int array_of_dargs[], const int array_of_psizes[],
int order) const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_hindexed(int count,
const int array_of_blocklengths[],
const Aint array_of_displacements[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_hvector(int count, int blocklength, Aint stride)
const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_indexed(int count,
const int array_of_blocklengths[],
const int array_of_displacements[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_indexed_block(int count, int blocklength,
const int array_of_displacements[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_resized(const Aint lb, const Aint extent) const
（廃止された呼び出し形式, 第15.2節を参照）}

{static Datatype Datatype::Create_struct(int count,
const int array_of_blocklengths[], const Aint
array_of_displacements[], const Datatype array_of_types[])
（廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_subarray(int ndims, const int array_of_sizes[],
const int array_of_subsizes[], const int array_of_starts[],
int order) const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Create_vector(int count, int blocklength, int stride)
const （廃止された呼び出し形式, 第15.2節を参照）}

{Datatype Datatype::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{Aint Get_address(void* location) （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Get_contents(int max_integers, int max_addresses,
int max_datatypes, int array_of_integers[],
Aint array_of_addresses[], Datatype array_of_datatypes[])
const （廃止された呼び出し形式, 第15.2節を参照）}

{int Status::Get_elements(const Datatype& datatype) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Get_envelope(int& num_integers, int& num_addresses, int& num_datatypes, int& combiner) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Get_extent(Aint& lb, Aint& extent) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Datatype::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Get_true_extent(Aint& true_lb, Aint& true_extent) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Pack(const void* inbuf, int incount, void *outbuf, int outsize, int& position, const Comm &comm) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Pack_external(const char* datarep, const void* inbuf, int incount, void* outbuf, Aint outsize, Aint& position) const （廃止された呼び出し形式, 第15.2節を参照）}

{Aint Datatype::Pack_external_size(const char* datarep, int incount) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Datatype::Pack_size(int incount, const Comm& comm) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Unpack(const void* inbuf, int insize, void *outbuf, int outcount, int& position, const Comm& comm) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Unpack_external(const char* datarep, const void* inbuf, Aint insize, Aint& position, void* outbuf, int outcount) const （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.3 集団的通信 C++ 言語呼び出し形式

namespace MPI {

{void Comm::Allgather(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount,

const Datatype& recvtype) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Allgatherv(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int displs[], const Datatype& recvtype) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Allreduce(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Alltoall(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Alltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], const Datatype& recvtype) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Alltoallw(const void* sendbuf, const int sendcounts[], const int sdispls[], const Datatype sendtypes[], void* recvbuf, const int recvcounts[], const int rdispls[], const Datatype recvtypes[]) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Barrier() const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Bcast(void* buffer, int count, const Datatype& datatype, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Op::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Gather(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Gatherv(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int displs[], const Datatype& recvtype, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Op::Init(User_function* function, bool commute) （廃止された呼び出し形式, 第15.2節を参照）}

{bool Op::Is_commutative() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Reduce(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Op::Reduce_local(const void* inbuf, void* inoutbuf, int count, const Datatype& datatype) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Reduce_scatter(const void* sendbuf, void* recvbuf, int recvcounts[], const Datatype& datatype, const Op& op) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Reduce_scatter_block(const void* sendbuf, void* recvbuf, int recvcount, const Datatype& datatype, const Op& op) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Intracomm::Scan(const void* sendbuf, void* recvbuf, int count, const Datatype& datatype, const Op& op) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Scatter(const void* sendbuf, int sendcount, const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Scatterv(const void* sendbuf, const int sendcounts[], const int displs[], const Datatype& sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.4 グループ，コンテクスト，コミュニケータ，キャッシング C++ 言語呼び出し形式

namespace MPI {

{Comm& Comm::Clone() const = 0 （廃止された呼び出し形式, 第15.2節を参照）}

{Cartcomm& Cartcomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}

{Distgraphcomm& Distgraphcomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}

{Graphcomm& Graphcomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}

{Intercomm& Intercomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}

{Intracomm& Intracomm::Clone() const （廃止された呼び出し形式, 第15.2節を参照）}

{static int Comm::Compare(const Comm& comm1, const Comm& comm2) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Group::Compare(const Group& group1, const Group& group2) （廃止された呼び出し形式, 第15.2節を参照）}

{Intracomm Intracomm::Create(const Group& group) const （廃止された呼び出し形式, 第15.2節を参照）}

{Intercomm Intercomm::Create(const Group& group) const （廃止された呼び出し形式, 第15.2節を参照）}

{Intercomm Intracomm::Create_intercomm(int local_leader, const Comm& peer_comm, int remote_leader, int tag) const （廃止された呼び出し形式, 第15.2節を参照）}

{static int Comm::Create_keyval(Comm::Copy_attr_function* comm_copy_attr_fn, Comm::Delete_attr_function* comm_delete_attr_fn, void* extra_state) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Datatype::Create_keyval(Datatype::Copy_attr_function* type_copy_attr_fn, Datatype::Delete_attr_function* type_delete_attr_fn, void* extra_state) （廃止された呼び出し形式, 第15.2節を参照）}

{static int Win::Create_keyval(Win::Copy_attr_function* win_copy_attr_fn, Win::Delete_attr_function* win_delete_attr_fn, void* extra_state) （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Delete_attr(int comm_keyval) （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Delete_attr(int type_keyval) （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Delete_attr(int win_keyval) （廃止された呼び出し形式, 第15.2節を参照）}

{static Group Group::Difference(const Group& group1, const Group& group2)
（廃止された呼び出し形式, 第15.2節を参照）}

{Cartcomm Cartcomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

{Distgraphcomm Distgraphcomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

{Graphcomm Graphcomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

{Intercomm Intercomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

{Intracomm Intracomm::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

{Group Group::Excl(int n, const int ranks[]) const （廃止された呼び出し形式,
第15.2節を参照）}

{void Comm::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{void Group::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{static void Comm::Free_keyval(int& comm_keyval) （廃止された呼び出し形式, 第15.2節
を参照）}

{static void Datatype::Free_keyval(int& type_keyval) （廃止された呼び出し形式,
第15.2節を参照）}

{static void Win::Free_keyval(int& win_keyval) （廃止された呼び出し形式, 第15.2節を
参照）}

{bool Comm::Get_attr(int comm_keyval, void* attribute_val) const （廃止された
呼び出し形式, 第15.2節を参照）}

{bool Datatype::Get_attr(int type_keyval, void* attribute_val) const （廃止さ
れた呼び出し形式, 第15.2節を参照）}

{bool Win::Get_attr(int win_keyval, void* attribute_val) const （廃止された呼
び出し形式, 第15.2節を参照）}

{Group Comm::Get_group() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Get_name(char* comm_name, int& resultlen) const （廃止された呼び出
し形式, 第15.2節を参照）}

{void Datatype::Get_name(char* type_name, int& resultlen) const （廃止された呼
び出し形式, 第15.2節を参照）}

{void Win::Get_name(char* win_name, int& resultlen) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Comm::Get_rank() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Group::Get_rank() const （廃止された呼び出し形式, 第15.2節を参照）}

{Group Intercomm::Get_remote_group() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Intercomm::Get_remote_size() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Comm::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Group::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}

{Group Group::Incl(int n, const int ranks[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{static Group Group::Intersect(const Group& group1, const Group& group2) （廃止された呼び出し形式, 第15.2節を参照）}

{bool Comm::Is_inter() const （廃止された呼び出し形式, 第15.2節を参照）}

{Intracomm Intercomm::Merge(bool high) const （廃止された呼び出し形式, 第15.2節を参照）}

{Group Group::Range_excl(int n, const int ranges[][3]) const （廃止された呼び出し形式, 第15.2節を参照）}

{Group Group::Range_incl(int n, const int ranges[][3]) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Set_attr(int comm_keyval, const void* attribute_val) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Set_attr(int type_keyval, const void* attribute_val) （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Set_attr(int win_keyval, const void* attribute_val) （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Set_name(const char* comm_name) （廃止された呼び出し形式, 第15.2節を参照）}

{void Datatype::Set_name(const char* type_name) （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Set_name(const char* win_name) （廃止された呼び出し形式, 第15.2節を参照）}

{Intercomm Intercomm::Split(int color, int key) const （廃止された呼び出し形式, 第15.2節を参照）}

{Intracomm Intracomm::Split(int color, int key) const （廃止された呼び出し形式, 第15.2節を参照）}

{static void Group::Translate_ranks (const Group& group1, int n, const int ranks1[], const Group& group2, int ranks2[]) （廃止された呼び出し形式, 第15.2節を参照）}

{static Group Group::Union(const Group& group1, const Group& group2) （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.5 プロセストポロジー C++ 言語呼び出し形式

namespace MPI {

{void Compute_dims(int nnodes, int ndims, int dims[]) （廃止された呼び出し形式, 第15.2節を参照）}

{Cartcomm Intracomm::Create_cart(int ndims, const int dims[], const bool periods[], bool reorder) const （廃止された呼び出し形式, 第15.2節を参照）}

{Graphcomm Intracomm::Create_graph(int nnodes, const int index[], const int edges[], bool reorder) const （廃止された呼び出し形式, 第15.2節を参照）}

{Distgraphcomm Intracomm::Dist_graph_create(int n, const int sources[], const int degrees[], const int destinations[], const int weights[], const Info& info, bool reorder) const （廃止された呼び出し形式, 第15.2節を参照）}

{Distgraphcomm Intracomm::Dist_graph_create(int n, const int sources[], const int degrees[], const int destinations[], const Info& info, bool reorder) const （廃止された呼び出し形式, 第15.2節を参照）}

{Distgraphcomm Intracomm::Dist_graph_create_adjacent(int indegree, const int sources[], const int sourceweights[], int outdegree, const int destinations[], const int destweights[], const Info& info, bool reorder) const （廃止された呼び出し形式, 第15.2節を参照）}

{Distgraphcomm Intracomm::Dist_graph_create_adjacent(int indegree, const int sources[], int outdegree, const int destinations[], const Info& info, bool reorder) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Cartcomm::Get_cart_rank(const int coords[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Cartcomm::Get_dim() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Graphcomm::Get_dims(int nnodes[], int nedges[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Distgraphcomm::Get_dist_neighbors(int maxindegree, int sources[], int sourceweights[], int maxoutdegree, int destinations[], int destweights[]) （廃止された呼び出し形式, 第15.2節を参照）}

{void Distgraphcomm::Get_dist_neighbors_count(int& indegree, int& outdegree, bool& weighted) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Distgraphcomm::Get_dist_neighbors_count(int rank, int indegree[], int outdegree[], bool& weighted) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Graphcomm::Get_neighbors(int rank, int maxneighbors, int neighbors[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Graphcomm::Get_neighbors_count(int rank) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Cartcomm::Get_topo(int maxdims, int dims[], bool periods[], int coords[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Graphcomm::Get_topo(int maxindex, int maxedges, int index[], int edges[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Comm::Get_topology() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Cartcomm::Map(int ndims, const int dims[], const bool periods[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Graphcomm::Map(int nnodes, const int index[], const int edges[]) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Cartcomm::Shift(int direction, int disp, int& rank_source, int& rank_dest) const （廃止された呼び出し形式, 第15.2節を参照）}

{Cartcomm Cartcomm::Sub(const bool remain_dims[]) const （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.6 MPI環境管理 C++ 言語呼び出し形式

namespace MPI {

{void Comm::Abort(int errorcode) （廃止された呼び出し形式, 第15.2節を参照）}

{int Add_error_class() （廃止された呼び出し形式, 第15.2節を参照）}

{int Add_error_code(int errorclass) （廃止された呼び出し形式, 第15.2節を参照）}

{void Add_error_string(int errorcode, const char* string) （廃止された呼び出し形式, 第15.2節を参照）}

{void* Alloc_mem(Aint size, const Info& info) （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Call_errhandler(int errorcode) const （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Call_errhandler(int errorcode) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Call_errhandler(int errorcode) const （廃止された呼び出し形式, 第15.2節を参照）}

{static Errhandler Comm::Create_errhandler(Comm::Errhandler_function* function) （廃止された呼び出し形式, 第15.2節を参照）}

{static Errhandler File::Create_errhandler(File::Errhandler_function* function) （廃止された呼び出し形式, 第15.2節を参照）}

{static Errhandler Win::Create_errhandler(Win::Errhandler_function* function) （廃止された呼び出し形式, 第15.2節を参照）}

{void Finalize() （廃止された呼び出し形式, 第15.2節を参照）}

{void Errhandler::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{void Free_mem(void *base) （廃止された呼び出し形式, 第15.2節を参照）}

{Errhandler Comm::Get_errhandler() const （廃止された呼び出し形式, 第15.2節を参照）}

{Errhandler File::Get_errhandler() const （廃止された呼び出し形式, 第15.2節を参照）}

{Errhandler Win::Get_errhandler() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Get_error_class(int errorcode) （廃止された呼び出し形式, 第15.2節を参照）}

{void Get_error_string(int errorcode, char* name, int& resultlen) （廃止された呼び出し形式, 第15.2節を参照）}

{void Get_processor_name(char* name, int& resultlen) （廃止された呼び出し形式, 第15.2節を参照）}

{void Get_version(int& version, int& subversion) （廃止された呼び出し形式, 第15.2節を参照）}

{void Init() （廃止された呼び出し形式, 第15.2節を参照）}

{void Init(int& argc, char**& argv) （廃止された呼び出し形式, 第15.2節を参照）}

{bool Is_finalized() （廃止された呼び出し形式, 第15.2節を参照）}

{bool Is_initialized() （廃止された呼び出し形式, 第15.2節を参照）}

{void Comm::Set_errhandler(const Errhandler& errhandler) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Set_errhandler(const Errhandler& errhandler) （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Set_errhandler(const Errhandler& errhandler) （廃止された呼び出し形式, 第15.2節を参照）}

{double Wtick() （廃止された呼び出し形式, 第15.2節を参照）}

{double Wtime() （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.7 Infoオブジェクト C++ 言語呼び出し形式

```
namespace MPI {
```

{static Info Info::Create() （廃止された呼び出し形式, 第15.2節を参照）}

{void Info::Delete(const char* key) （廃止された呼び出し形式, 第15.2節を参照）}

{Info Info::Dup() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Info::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{bool Info::Get(const char* key, int valuelen, char* value) const （廃止された呼び出し形式, 第15.2節を参照）}

{int Info::Get_nkeys() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Info::Get_nthkey(int n, char* key) const （廃止された呼び出し形式, 第15.2節を参照）}

{bool Info::Get_valuelen(const char* key, int& valuelen) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Info::Set(const char* key, const char* value) （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.8 プロセスの生成と管理 C++ 言語呼び出し形式

```
namespace MPI {
```

{Intercomm Intracomm::Accept(const char* port_name, const Info& info, int root) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Close_port(const char* port_name) （廃止された呼び出し形式, 第15.2節を参照）}

{Intercomm Intracomm::Connect(const char* port_name, const Info& info, int root) const （廃止された呼び出し形式, 第15.2節を参照）}

```
{void Comm::Disconnect()} (廃止された呼び出し形式, 第15.2節を参照)

{static Intercomm Comm::Get_parent()} (廃止された呼び出し形式, 第15.2節を参照)

{static Intercomm Comm::Join(const int fd)} (廃止された呼び出し形式, 第15.2節を参照)

{void Lookup_name(const char* service_name, const Info& info,
              char* port_name)} (廃止された呼び出し形式, 第15.2節を参照)

{void Open_port(const Info& info, char* port_name)} (廃止された呼び出し形式,
              第15.2節を参照)

{void Publish_name(const char* service_name, const Info& info,
              const char* port_name)} (廃止された呼び出し形式, 第15.2節を参照)

{Intercomm Intracomm::Spawn(const char* command, const char* argv[],
              int maxprocs, const Info& info, int root) const} (廃止された呼び出
              し形式, 第15.2節を参照)

{Intercomm Intracomm::Spawn(const char* command, const char* argv[],
              int maxprocs, const Info& info, int root,
              int array_of_errcodes[]) const} (廃止された呼び出し形式, 第15.2節を参
              照)

{Intercomm Intracomm::Spawn_multiple(int count,
              const char* array_of_commands[], const char** array_of_argv[],
              const int array_of_maxprocs[], const Info array_of_info[],
              int root)} (廃止された呼び出し形式, 第15.2節を参照)

{Intercomm Intracomm::Spawn_multiple(int count,
              const char* array_of_commands[], const char** array_of_argv[],
              const int array_of_maxprocs[], const Info array_of_info[],
              int root, int array_of_errcodes[])} (廃止された呼び出し形式, 第15.2節
              を参照)

{void Unpublish_name(const char* service_name, const Info& info,
              const char* port_name)} (廃止された呼び出し形式, 第15.2節を参照)

};
```

A.4.9 片方向通信 C++ 言語呼び出し形式

```
namespace MPI {
```

{void Win::Accumulate(const void* origin_addr, int origin_count, const Datatype& origin_datatype, int target_rank, Aint target_disp, int target_count, const Datatype& target_datatype, const Op& op) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Complete() const （廃止された呼び出し形式, 第15.2節を参照）}

{static Win Win::Create(const void* base, Aint size, int disp_unit, const Info& info, const Intracomm& comm) （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Fence(int assert) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Free() （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Get(void *origin_addr, int origin_count, const Datatype& origin_datatype, int target_rank, Aint target_disp, int target_count, const Datatype& target_datatype) const （廃止された呼び出し形式, 第15.2節を参照）}

{Group Win::Get_group() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Lock(int lock_type, int rank, int assert) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Post(const Group& group, int assert) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Put(const void* origin_addr, int origin_count, const Datatype& origin_datatype, int target_rank, Aint target_disp, int target_count, const Datatype& target_datatype) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Start(const Group& group, int assert) const （廃止された呼び出し形式, 第15.2節を参照）}

{bool Win::Test() const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Unlock(int rank) const （廃止された呼び出し形式, 第15.2節を参照）}

{void Win::Wait() const （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.10 外部インターフェイス C++ 言語呼び出し形式

namespace MPI {

{void Grequest::Complete() （廃止された呼び出し形式, 第15.2節を参照）}

{int Init_thread(int required) （廃止された呼び出し形式, 第15.2節を参照）}

{int Init_thread(int& argc, char**& argv, int required) （廃止された呼び出し形式, 第15.2節を参照）}

{bool Is_thread_main() （廃止された呼び出し形式, 第15.2節を参照）}

{int Query_thread() （廃止された呼び出し形式, 第15.2節を参照）}

{void Status::Set_cancelled(bool flag) （廃止された呼び出し形式, 第15.2節を参照）}

{void Status::Set_elements(const Datatype& datatype, int count) （廃止された呼び出し形式, 第15.2節を参照）}

{static Grequest Grequest::Start(const Grequest::Query_function* query_fn, const Grequest::Free_function* free_fn, const Grequest::Cancel_function* cancel_fn, void *extra_state) （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.11 入出力 C++ 言語呼び出し形式

namespace MPI {

{void File::Close() （廃止された呼び出し形式, 第15.2節を参照）}

{static void File::Delete(const char* filename, const Info& info) （廃止された呼び出し形式, 第15.2節を参照）}

{int File::Get_amode() const （廃止された呼び出し形式, 第15.2節を参照）}

{bool File::Get_atomicity() const （廃止された呼び出し形式, 第15.2節を参照）}

{Offset File::Get_byte_offset(const Offset disp) const （廃止された呼び出し形式, 第15.2節を参照）}

{Group File::Get_group() const （廃止された呼び出し形式, 第15.2節を参照）}

{Info File::Get_info() const （廃止された呼び出し形式, 第15.2節を参照）}

{Offset File::Get_position() const （廃止された呼び出し形式, 第15.2節を参照）}

{Offset File::Get_position_shared() const （廃止された呼び出し形式, 第15.2節を参照）}

{Offset File::Get_size() const （廃止された呼び出し形式, 第15.2節を参照）}

{Aint File::Get_type_extent(const Datatype& datatype) const （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Get_view(Offset& disp, Datatype& etype, Datatype& filetype, char* datarep) const （廃止された呼び出し形式, 第15.2節を参照）}

{Request File::Iread(void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{Request File::Iread_at(Offset offset, void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{Request File::Iread_shared(void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{Request File::Iwrite(const void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{Request File::Iwrite_at(Offset offset, const void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{Request File::Iwrite_shared(const void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{static File File::Open(const Intracomm& comm, const char* filename, int amode, const Info& info) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Preallocate(Offset size) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read(void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read(void* buf, int count, const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_all(void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_all(void* buf, int count, const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_all_begin(void* buf, int count, const Datatype& datatype)
（廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_all_end(void* buf) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_all_end(void* buf, Status& status) （廃止された呼び出し形式,
第15.2節を参照）}

{void File::Read_at(Offset offset, void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_at(Offset offset, void* buf, int count,
const Datatype& datatype, Status& status) （廃止された呼び出し形式,
第15.2節を参照）}

{void File::Read_at_all(Offset offset, void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_at_all(Offset offset, void* buf, int count,
const Datatype& datatype, Status& status) （廃止された呼び出し形式,
第15.2節を参照）}

{void File::Read_at_all_begin(Offset offset, void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_at_all_end(void* buf) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_at_all_end(void* buf, Status& status) （廃止された呼び出し形式,
第15.2節を参照）}

{void File::Read_ordered(void* buf, int count, const Datatype& datatype)
（廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_ordered(void* buf, int count, const Datatype& datatype,
Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_ordered_begin(void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_ordered_end(void* buf) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_ordered_end(void* buf, Status& status) （廃止された呼び出し形式,
第15.2節を参照）}

{void File::Read_shared(void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Read_shared(void* buf, int count, const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void Register_datarep(const char* datarep, Datarep_conversion_function* read_conversion_fn, Datarep_conversion_function* write_conversion_fn, Datarep_extent_function* dtype_file_extent_fn, void* extra_state) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Seek(Offset offset, int whence) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Seek_shared(Offset offset, int whence) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Set_atomicity(bool flag) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Set_info(const Info& info) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Set_size(Offset size) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Set_view(Offset disp, const Datatype& etype, const Datatype& filetype, const char* datarep, const Info& info) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Sync() （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write(const void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write(const void* buf, int count, const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_all(const void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_all(const void* buf, int count, const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_all_begin(const void* buf, int count, const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_all_end(const void* buf) （廃止された呼び出し形式, 第15.2節を参照）
}

{void File::Write_all_end(const void* buf, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_at(Offset offset, const void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_at(Offset offset, const void* buf, int count,
const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_at_all(Offset offset, const void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_at_all(Offset offset, const void* buf, int count,
const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_at_all_begin(Offset offset, const void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_at_all_end(const void* buf) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_at_all_end(const void* buf, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_ordered(const void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_ordered(const void* buf, int count,
const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_ordered_begin(const void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_ordered_end(const void* buf) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_ordered_end(const void* buf, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_shared(const void* buf, int count,
const Datatype& datatype) （廃止された呼び出し形式, 第15.2節を参照）}

{void File::Write_shared(const void* buf, int count,
const Datatype& datatype, Status& status) （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.12 言語呼び出し形式 C++ 言語呼び出し形式

namespace MPI {

{static Datatype Datatype::Create_f90_complex(int p, int r) （廃止された呼び出し形式, 第15.2節を参照）}

{static Datatype Datatype::Create_f90_integer(int r) （廃止された呼び出し形式, 第15.2節を参照）}

{static Datatype Datatype::Create_f90_real(int p, int r) （廃止された呼び出し形式, 第15.2節を参照）}

Exception::Exception(int error_code)

{int Exception::Get_error_class() const （廃止された呼び出し形式, 第15.2節を参照）}

{int Exception::Get_error_code() const （廃止された呼び出し形式, 第15.2節を参照）}

{const char* Exception::Get_error_string() const （廃止された呼び出し形式, 第15.2節を参照）}

{static Datatype Datatype::Match_size(int typeclass, int size) （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.13 プロファイルインターフェイス C++ 言語呼び出し形式

namespace MPI {

{void Pcontrol(const int level, ...) （廃止された呼び出し形式, 第15.2節を参照）}

};

A.4.14 全てのMPIクラスにおける C++言語呼び出し形式

C++言語は, 全てのクラスが4つの特別な機能を持つ必要がある:デフォルトのコンストラクタ, コピーコンストラクタ, デストラクタそして割当てられた演算子である. これら機能の呼び出し形式を以下に列挙する. これらの意味論セクション 16.1.5にて述べている.

2つのコンストラクタは**仮想**ではない. プロトタイプ関数の呼び出し形式 は, ⟨CLASS⟩型よりもすべてのMPI クラス の関数をそれぞれ列挙する方法を使う. ⟨CLASS⟩トークンは通知されたとき以外, Group, Datatypeなどの有効なMPI-2クラス名によってによって置き換えることができる. 加えて呼び出し形式はcomparison and inter-language セクション 16.1.5と 16.1.9の 操作性ために提供される.

A.4.15 コンストラクション／デストラクション

```
namespace MPI {

   ⟨CLASS⟩::⟨CLASS⟩()

   ⟨CLASS⟩::~⟨CLASS⟩()

};
```

A.4.16 複製／割当て

```
namespace MPI {

   ⟨CLASS⟩::⟨CLASS⟩(const ⟨CLASS⟩& data)

   ⟨CLASS⟩& ⟨CLASS⟩::operator=(const ⟨CLASS⟩& data)

};
```

A.4.17 比較

Statusインスタンスが，根本的なオブジェクトへのハンドラではないため，operator==()とoperator!=()関数はStatusクラスに宣言されていない.

```
namespace MPI {

   bool ⟨CLASS⟩::operator==(const ⟨CLASS⟩& data) const

   bool ⟨CLASS⟩::operator!=(const ⟨CLASS⟩& data) const

};
```

A.4.18 言語間操作性

C++言語オブジェクトのMPI::STATUS_IGNORE とMPI::STATUSES_IGNOREがないため, 結果として C言語かFortran言語に働きかける(MPI_STATUS_IGNORE と MPI_STATUSES_IGNORE)C++言語が未定義である.

```
namespace MPI {

   ⟨CLASS⟩& ⟨CLASS⟩::operator=(const MPI_⟨CLASS⟩& data)

   ⟨CLASS⟩::⟨CLASS⟩(const MPI_⟨CLASS⟩& data)

   ⟨CLASS⟩::operator MPI_⟨CLASS⟩() const

};
```

付録B章

変更履歴

この付録ではMPI標準のこれまでの版からの変更をまとめる． MPIライブラリの実装が必要となるような，あるいはユーザ観点でのMPIの理解が変わるような，大きな変更のみ（すなわち, 明確化と新機能）を記載する． 校正修正，フォーマット変更，誤植修正，そして些細な明確化は記載しない．

B.1 2.1から2.2への変更点

1. 17ページの 第2.5.4節
   定義済みの名前付き定数ハンドルを（他の定数と同様に）初期化式や代入に使用して良いことが保証された．すなわち，MPI_INITの呼び出し前でも良い．

2. 19ページの 第2.6節， 22ページの 第2.6.4節， 487ページの 第16.1節
   C++言語呼び出し形式は廃止されたため，将来的にMPIの仕様から削除されることがある．

3. 32ページの 第3.2.2節
   印字可能な文字のためのMPI_CHARは，C言語の（signed charの代わりに）charとして定義している．この変更はどのようなアプリケーションやMPI ライブラリ（いくつかのコメント行以外）に影響を与えてはならない．なぜなら，印字可能な文字は，過去も現在もC言語のchar型，signed char型，unsigned char型のいずれにも格納でき，また，MPI_CHARは定義済みのリダクション操作には使えないためである．

4. 32ページの 第3.2.2節
   MPI_(U)INT{8,16,32,64}_T， MPI_AINT， MPI_OFFSET， MPI_C_BOOL，
   MPI_C_COMPLEX， MPI_C_FLOAT_COMPLEX， MPI_C_DOUBLE_COMPLEX，
   MPI_C_LONG_DOUBLE_COMPLEXは，定義済みMPIデータ型として有効になった．

5. 46ページの 第3.4節， 58ページの 第3.7.2節， 78ページの 第3.9節， 141ページの 第5.1節

ブロッキング，ノンブロッキング，集団APIの送信バッファへの読込みアクセスの制限が解除された．送信バッファが操作中であっても読込みアクセスを許可する．

6. 56ページの 第3.7節
IBSENDとIRSENDのユーザへのアドバイスが若干変更になった．

7. 61ページの 第3.7.3節
MPI_REQUEST_FREEのユーザへのアドバイスで，アクティブな要求の解放のアドバイスを削除した．

8. 73ページの 第3.7.6節
MPI_REQUEST_GET_STATUSの入力として非アクティブかnullリクエストを許可するように変更した．

9. 167ページの 第5.8節
“in place”オプションがグループ内コミュニケータを使用したMPI_ALLTOALL，MPI_ALLTOALLV，MPI_ALLTOALLWに追加された．

10. 174ページの 第5.9.2節
定義済みのパラメータ化されたデータ型（例えば，MPI_TYPE_CREATE_F90_REALが返す型）と選択可能な名前付き定義済みデータ型（例えば，MPI_REAL8）をリダクション操作の有効なデータ型のリストに追加した．

11. 174ページの 第5.9.2節
MPI_(U)INT{8,16,32,64}_Tは，定義済みのリダクション操作を目的として全てC言語の整数型と認識する．MPI_AINTとMPI_OFFSETはFortran言語の整数型と認識する．MPI_C_BOOLは論理型と認識する．MPI_C_COMPLEX，MPI_C_FLOAT_COMPLEX，MPI_C_DOUBLE_COMPLEX，MPI_C_LONG_DOUBLE_COMPLEXは複素数型と認識する．

12. 186ページの 第5.9.7節
ローカルルーチンのMPI_REDUCE_LOCALとMPI_OP_COMMUTATIVEを追加した．

13. 187ページの 第5.10.1節
集団関数のMPI_REDUCE_SCATTER_BLOCKをMPI標準に追加した．

14. 190ページの 第5.11.2節
MPI_EXSCANにインプレイス引数を追加した．

15. 211ページの 第6.4.2節 225ページの 第6.6節
グループ間コミュニケータに関するMPI_COMM_CREATEをサポートしていない実装については，この関数を追加しなければならない．グループ間コミュニケータの操作の挙動が標準で示されているが，それは既にほとんどの実装にて提供されていると考えたからである．MPI_COMM_CREATEとMPI_COMM_SPLIT両方のC++言

語呼び出し形式は，グループ間コミュニケータを明確にを許可していることも留意すること．

16. 211ページの 第6.4.2節
MPI_COMM_CREATEは，commがグループ内コミュニケータの場合，互いに疎な複数のサブグループを入力とすることが許されるよう拡張された．commがグループ間コミュニケータの場合，commの同じローカルグループの全てのプロセスはグループに同じ値を指定しなければならないと明文化された．

17. 262ページの 第7.5.4節
スケーラブル分散グラフトポロジーインターフェイスの新しい関数を追加した． この節では， MPI_DIST_GRAPH_CREATE_ADJACENT関数とMPI_DIST_GRAPH_CREATE関数， MPI_UNWEIGHTED定数， C++言語の派生クラスDistgraphcommが追加された．

18. 268ページの 第7.5.5節
スケーラブル分散グラフトポロジーインターフェイスのため，MPI_DIST_NEIGHBORS_COUNT関数とMPI_DIST_NEIGHBORS関数，MPI_DIST_GRAPH定数を追加した．

19. 268ページの 第7.5.5節
MPI_GRAPH_NEIGHBORSとMPI_GRAPH_NEIGHBORS_COUNTによって複製された隣接に関する曖昧さを削除した．

20. 281ページの 第8.1.1節
サブバージョンを1から2へ変更した．

21. 286ページの 第8.3節， 485ページの 第15.2節， 553ページの 付録A.1.3
関数ポインタのtypedef名をMPI_{Comm,File,Win}_errhandler_fnから MPI_{Comm,File,Win}_errhandler_functionに変更した． 古い“_fn”名は望ましくない．

22. 306ページの 第8.7.1節
MPI_COMM_SELFの属性削除コールバックはLIFOで呼び出されるようになった．実装者は，MPI_INIT/MPI_INIT_THREADから戻る前にMPI_COMM_SELFの全ての実装内部の属性削除コールバックを登録しなければならない．

23. 357ページの 第11.3.4節
MPI_ACCUMULATEのMPI_REPLACEが定義済みのデータ型でのみ使うことができる制限がMPI 2.1で追加されたが削除された． MPI_REPLACEは， MPI 2.0の時と同様に派生データ型を使用できるようになった． また， MPI_REPLACEは MPI_ACCUMULATEの中だけで使用でき，MPI_REDUCEなどのリダクション集団操作の中では使用できないということを明確にした．

24. 389ページの 第12.2節
MPI::Grequest::Start()のC++言語呼び出し形式のquery_fn，free_fn，cancel_fnの引数に“*”を追加した．これは，関数ポインタを引数とする他のMPI関数との一貫性を保つためである．

25. 448ページの 第13.5.2節， 450ページの 表13.2
external32表現の宣言済みのデータ型としてMPI_(U)INT{8,16,32,64}_T， MPI_AINT，MPI_OFFSET， MPI_C_COMPLEX， MPI_C_FLOAT_COMPLEX，MPI_C_DOUBLE_COMPLEX， MPI_C_LONG_DOUBLE_COMPLEX， MPI_C_BOOLを追加した．

26. 529ページの 第16.3.7節
MPIが内部的にどのように属性を格納するかではなく，どのようにMPIの 実装が振る舞うかの説明だけに説明文を変更した．誤ったMPI-2.1 例16.17は， 530ページ-531ページの3つの新しい例である例16.17，例16.18， 例16.19に置き換えられ，言語間の属性の詳細な動きを明示している．古い例の動作にマッチする動作をする実装は更新する必要がある．

27. 537ページの 付録A.1.1
MPI::Fint型（ 552ページの 第A.1.2節にあるMPI_Fintと比較すること）を削除した．

28. 537ページの 付録A.1.1 表 *Named Predefined Datatypes*
定義済みのデータ型として， MPI_(U)INT{8,16,32,64}_T， MPI_AINT，MPI_OFFSET， MPI_C_BOOL， MPI_C_FLOAT_COMPLEX， MPI_C_COMPLEX，MPI_C_DOUBLE_COMPLEX，MPI_C_LONG_DOUBLE_COMPLEXを追加した．

B.2 2.0から2.1への変更点

1. 32ページの 第3.2.2節， 492ページの 第16.1.6節， 537ページの 付録A.1
MPI_LONG_LONGは，オプションとして追加されなければならない．MPI_LONG_LONG_INTと同義である．

2. 32ページの 第3.2.2節， 492ページの 第16.1.6節， 537ページの 付録A.1
MPI_LONG_LONG_INT， MPI_LONG_LONG （同義の別名），MPI_UNSIGNED_LONG_LONG， MPI_SIGNED_CHAR， とMPI_WCHARは， オプションから公式になり3言語全ての言語呼び出し形式で定義された．

3. Section 38ページの 第3.2.5節長さ0のデータ型に対するMPI_GET_COUNTについて長さゼロのデータ型に対して0バイトが転送された場合，MPI_GET_COUNT関数のcount引数に返される値はゼロである．転送バイト数がゼロより大きいならば，MPI_UNDEFINEDを返される．

4. 87ページの 第4.1節 派生データ型の基本ルールについて
ほとんどのデータ型コンストラクタは，反復回数やブロック長の引数を持つ．これらの引数に対し許される値は非負整数である．ただし，これらの値が0の場合，型マップの中には要素を何も生じさせないし，データ型の上下限や範囲に対し何の影響も与えない．

5. 137ページの 第4.3節
MPI_PACK_EXTERNALによってパックされたデータの送信と受信には，MPI_BYTEが使用されなくてはならない．

6. 185ページの 第5.9.6節
commがMPI_ALLREDUCEのグループ間コミュニケータの場合， 両グループは，同一の型シグネチャとなるようなcountとdatatypeを引数として渡さなけらばならない． （すなわち，両グループが同じcount値を提供する必要はない．）

7. 202ページの 第6.3.1節 MPI_GROUP_TRANSLATE_RANKSとMPI_PROC_NULLについて
MPI_GROUP_TRANSLATE_RANKSの入力としてMPI_PROC_NULLは有効なランクで，変換されたランクとしてMPI_PROC_NULLを返却する．

8. 234ページの 第6.7節属性キャッシュ関数について

> **実装者へのアドバイス** 高品質な実装であれば， MPI_XXX_CREATE_KEYVALの呼び出しによって生成されたkeyval が， MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR,MPI_YYY_FREE_KEYVALへの呼び出しで誤った型のオブジェクトを指定して使用された場合，エラーを発生させなくてはならない．そのためには，関連するユーザ関数の型に関する情報をkeyval毎に管理する必要がある． （**実装者へのアドバイス終わり**）

9. 248ページの 第6.8節 MPI_COMM_GET_NAMEについて
C言語では，name[resultlen]にnull文字が追加で保存される． resultlenは，MPI_MAX_OBJECT_NAME-1よりも大きくなってはいけない． Fortran言語では，nameの右に空白文字がパディングされる． resultlenは，MPI_MAX_OBJECT_NAMEよりも大きくなってはいけない．

10. 257ページの 第7.4節 MPI_GRAPH_CREATEとMPI_CART_CREATEについて
全ての入力引数は，comm_oldグループのすべてのプロセスで同一値を持たなくてはならない．

11. 258ページの 第7.5.1節 MPI_CART_CREATEについて
ndimsがゼロの場合，0次元のカルテシアントポロジーが生成される．グループサイズよりも大きな格子を指定した場合，またはndimsが負の値の場合は誤りである．

12. 260ページの 第7.5.3節 MPI_GRAPH_CREATEについて
もしグラフが空ならば，すなわちnnodes == 0ならば，全プロセスで
MPI_COMM_NULLが返却される．

13. 260ページの 第7.5.3節 MPI_GRAPH_CREATEについて
プロセスの隣接リスト内で1つのプロセスを複数回定義することができる（つまり，2つのプロセスの間に複数のエッジが存在する可能性がある）．また，プロセスはそれ自体の隣接とすることもできる（つまり，グラフ内の自己参照ループ）．隣接行列は非対称とすることができる．

> **ユーザへのアドバイス** 複数のエッジまたは非対称の隣接行列を使用した場合の性能の詳細は定義されていない．ノード隣接エッジの定義では通信の方向は示されていない． （**ユーザへのアドバイス終わり**）

14. 268ページの 第7.5.5節 MPI_CARTDIM_GETとMPI_CART_GETについて
commに0次元のカルテシアントポロジーが関連付けされている場合，
MPI_CARTDIM_GETはndims=0を返し，MPI_CART_GETはすべての出力引数を変更しないで保持する．

15. 268ページの 第7.5.5節 MPI_CART_RANKについて
commに0次元のカルテシアントポロジーが関連付けされている場合，coordは意味を持たず，rankに0が返される．

16. 268ページの 第7.5.5節 MPI_CART_COORDSについて
commに0次元のカルテシアントポロジーが関連付けされている場合，coordsは変更されない．

17. 275ページの 第7.5.6節 MPI_CART_SHIFTについて
ある方向について負の値またはカルテシアンコミュニケータの次元数以上の値を指定してMPI_CART_SHIFTを呼び出すのは誤りである．このことは，0次元のカルテシアントポロジーが関連付けられたcommを使用するのは，MPI_CART_SHIFTの呼び出しは誤りとなることを示している．

18. 276ページの 第7.5.7節 MPI_CART_SUBについて
remain_dimsのすべてのエントリが偽か，またはcommにすでに0次元のカルテシアントポロジーが関連付けられている場合，newcommに0次元のカルテシアントポロジーが関連付けられる．

18.1. 281ページの 第8.1.1節
サブバージョン番号が0から1に変更になった．

19. 282ページの 第8.1.2節 MPI_GET_PROCESSOR_NAMEについて
C言語では加えて，name[resultlen]にnull文字が保存される．resultlenは，
MPI_MAX_PROCESSOR_NAME-1よりも大きくならない．Fortran言語では，nameの

右にスペースがパディングされる．resultlenは，MPI_MAX_PROCESSOR_NAMEよりも大きくならない．

20. 286ページの 第8.3節
MPI_{COMM,WIN,FILE}_GET_ERRHANDLER の動作は，新しいエラーハンドラオブジェクトが生成されるのと同様である．つまり，エラーハンドラが不要となった時点で，MPI_ERRHANDLER_GETまたはMPI_{COMM,WIN,FILE}_GET_ERRHANDLERから返されたエラーハンドラを使用してMPI_ERRHANDLER_FREEを呼び出し，エラーハンドラを解放のためにマークする必要がある．この動作は，MPI_COMM_GROUPおよびMPI_GROUP_FREEと同様である．

21. 301ページの 第8.7節，MPI_FINALIZEの説明参照
MPI_FINALIZEは接続されている全てのプロセスに対して集団的である．スポーン（spawn），アクセプト（accept），またはコネクト（connect）が行われたプロセスがない場合，これはMPI_COMM_WORLDが対象となる．それ以外の場合，かつて接続していたまたは現在も接続中である全てのプロセスの集合に対して集団的である．これらの事項は，343ページの第10.5.4節で説明している．

22. 301ページの 第8.7節 MPI_ABORTについて

ユーザへのアドバイス エラーコードが実行可能ファイルから返されるか，MPIプロセスの起動メカニズム（mpiexecなど）から返されるかは，MPIライブラリの品質に関する事柄であり，必須事項ではない．（**ユーザへのアドバイス終わり**）

実装者へのアドバイス 高品質な実装では，可能なかぎり，MPIプロセスの起動メカニズム（mpiexec，シングルトンinitなど）からエラーコードを返そうとする．（**実装者へのアドバイス終わり**）

23. 311ページの 第9節
実装では，キーを識別するかどうかに関係なく，infoオブジェクト を任意の(key,value)のペアのキャッシュとしてサポートする必要がある．識別しないキーを無視するため，MPI_Infoの形式でヒントを取得する各関数を用意する必要がある．infoオブジェクトのこの記述は，キーは識別するが付加された値は識別しない場合に，特定の関数が対処する方法を定義するものではない．MPI_INFO_GET_NKEYS，MPI_INFO_GET_NTHKEY，MPI_INFO_GET_VALUELEN，MPI_INFO_GETは，レイヤー化された機能でもInfoオブジェクトを使用できるように，すべての (key, value)のペアを保持する必要がある．

24. 351ページの 第11.3節
MPI_PROC_NULLはMPI RMA 呼び出しのMPI_ACCUMULATE，MPI_GET，

MPI_PUTにおける有効なターゲットランクである．効果は MPI の1対1通信の MPI_PROC_NULLの場合と同様である． 項番25も参照のこと．

25. 351ページの 第11.3節
ランクMPI_PROC_NULLをもつどんな RMA 操作の後にも，それでもなお，そのアクセスエポックを開始した同期メソッドでアクセスエポックを終わらせる必要がある． 項番24も参照のこと．

26. 357ページの 第11.3.4節
MPI_ACCUMULATEのMPI_REPLACEは， 他の定義済み操作のように，定義済みMPIデータ型のためだけに定義されている．

27. 415ページの 第13.2.8節 MPI_FILE_SET_VIEWとMPI_FILE_SET_INFOについて
有効なヒントのサブセットを指定したinfoオブジェクトをMPI_FILE_SET_VIEWまたはMPI_FILE_SET_INFOに渡す場合，このinfoで指定しない設定済みのヒントやデフォルトのヒントには影響しない．

28. 415ページの 第13.2.8節 MPI_FILE_GET_INFOについて
fhで関連付けられたファイルのヒントが存在しない場合，新しく生成された，キー値のペアが格納されていないinfoオブジェクトへのハンドルが返される．

29. 419ページの 第13.3節
ファイルがMPI_MODE_SEQUENTIALモードを持っていなかった場合，MPI_FILE_SET_VIEWのdisp同様，MPI_DISPLACEMENT_CURRENTも無効である．

30. 448ページの 第13.5.2節
16バイト“Double Extended”形式のbiasについてbias = +10383と定義されていた．正しい値は16383である．

31. 488ページの 第16.1.4節
この節の例では，バッファは`const void* buf`として宣言されるべきである．

32. 511ページの 第16.2.5節 MPI_TYPE_CREATE_F90_xxxxについて

> **実装者へのアドバイス** アプリケーションは同じ(xxxx,p,r)の組み合わせで MPI_TYPE_CREATE_F90_xxxxの呼び出しを繰り返すことがよくある．アプリケーションでは，返された定義済みの名前無しデータ型ハンドルを解放することはできない．非常に多数のハンドルが生成されるのを防止するため，MPI 実装では同じ(REAL/COMPLEX/INTEGER,p,r)の組み合わせに対して同じデータ型ハンドルを返す必要がある．その前のMPI_TYPE_CREATE_F90_xxxxの呼び出しでの(p,r)の組み合わせをチェックし，ハッシュテーブルにより前に生成されたハンドルを検出すると，(xxxx,p,r)の同じ組み合わせで以前に生成されたデータ型を検索する際のオーバーヘッドが抑えられる． （**実装者へのアドバイス終わり**）

33. 537ページの 第A.1.1節
MPI_BOTTOMは，`void * const MPI::BOTTOM`として宣言された．

参考文献

[1] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2

[2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2

[3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 6.1

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993. 1.2

[5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990. 1.2

[6] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993. 13.1

[7] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992. 1.2

[8] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2

[9] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, April 1994. 1.2

[10] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers. In *Proceedings of the 1990 Supercomputing Conference*, pages 647–656, 1990. 7.1

[11] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)*, pages II–1 – II–4, 1991. 7.1

[12] Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 1.2, 7.4

[13] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38. 13.1

[14] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–75, April 1993. 1.2

[15] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993. 1.2

[16] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991. 1.2

[17] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992. 1.2

[18] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[19] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 6.1.2

[20] C++ Forum. Working paper for draft proposed international standard for information systems — programming language C++. Technical report, American National Standards Institute, 1995.

[21] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994. 1.3

[22] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. `http://www.mpi-forum.org`. 1.3

[23] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994. 10.1

[24] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. 1.2

[25] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 1.2

[26] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from `http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90`. 16.2.4

[27] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.5.2

[28] International Organization for Standardization, Geneva. *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, 1987. 13.5.2

[29] International Organization for Standardization, Geneva. *Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996. 12.4, 13.2.1

[30] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993. 4.1.4

[31] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1

[32] O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989. 7.1

[33] S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Also available at `http://www.netbsd.org/Documentation/lite2/psd/`. 10.5.5

[34] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990. 1.2

[35] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 13.1

[36] William J. Nitzberg. *Collective Parallel I/O*. PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 13.1

[37] *4.4BSD Programmer's Supplementary Documents (PSD)*. O'Reilly and Associates, 1994. 10.5.5

[38] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988. 1.2

[39] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. 13.1

[40] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. 1.2, 6.1.2

[41] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. 1.2

[42] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993. 6.1

[43] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, April 1994. `http://www.erc.msstate.edu/mpi/mpix.html`. 5.2.2

[44] Anthony Skjellum, Ziyang Lu, Purushotham V. Bangalore, and Nathan E. Doss. Explicit parallel programming in C++ based on the message-passing interface (MPI). In Gregory V. Wilson, editor, *Parallel Programming Using C++*, Engineering Computation Series. MIT Press, July 1996. ISBN 0-262-73118-5.

[45] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994. 6.1.2, 6.5.6

[46] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996. 13.1

[47] *The Unicode Standard, Version 2.0*. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.5.2

[48] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2

例の索引

全ての例の索引である．内容，あるいは，使われているMPI関数により列挙されている．MPI関数のうち，名前が全て大文字のものはFortran言語の例であり，そうでないものはC言語／C++言語の例である．

MPI定数と定義済みハンドルの索引

予め定義されたMPI定数とハンドルの索引

MPI宣言の索引

アドレスkind整数，ハンドル，などのC言語／C++言語に必要な宣言の索引．下線が付いたページ番号は主たる参照（重要なコンセプトが複数ページに現れる場合は，複数ある）

MPIコールバック関数プロトタイプの索引

属性キャッシングあるいはユーザ定義のリダクション操作のためのC言語のtypedef名の索引．C言語名の記述の近くにあるtyepdefのC++言語名およびFortran言語の例

MPI関数の索引

下線がついたページ番号は関数定義であることを示す.