並列処理とMPI通信ライブラリ 入門

東京大学
石川　裕

目次

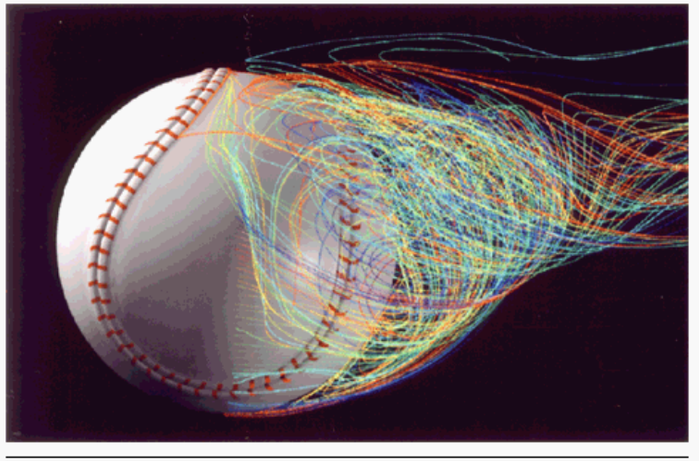
- 並列プログラムの分類
- MPI通信ライブラリの概要
- MPI通信ライブラリを使った簡単なプログラム例
- 性能指標
- 様々なMPI通信ライブラリ性能
  - Point to Point 遅延＆バンド幅
  - NASA並列ベンチマーク結果

並列プログラムの分類

- データ並列
  - 気象予測
  - 原子炉シミュレーション
  - 車、飛行機の設計
  - 天体・宇宙
  - コンピュータグラフィックス
- Embarrassingly Parallel(EP)
  - データベース検索、パラメータサーチ
- コントロール並列
  - 並列オペレーティングシステム
  - ウインドウシステム

データ並列の例

- 魔球の正体
  - 姫野龍太郎先生＠理化学研究所
  - ナックルボールのような変化球がなぜできるのかをコンピュータシミュレーションで解明
    - 流体力学(Computational Fluid Dynamics)



野球ボールまわりの非対称な流れのようす

http://www.riken.go.jp/r-world/research/lab/wako/info/environ/index.htmlより

データ並列:どのような処理？

- 大量データ＆計算
  - 魔球の正体の解明では、
    - 1,870万元の速度に関する連立方程式
    - 623万元の圧力の連立方程式
    - これらを10万回解く
    - 90年代後半最速のコンピュータ(スーパコンピュータ)の1CPUで全部の計算を終了するには1,000～2,000時間要する

出展:湯浅、安村、中田編、「はじめての並列プログラミング」、共立出版、1999、ISBN4-320-02940-2

データ並列:どのような処理？

- 連立方程式の解法の一つヤコビの反復法をとりあげる

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \cdots. & a_{1n} \\ a_{21} & a_{22} & a_{23} \cdots. & a_{2n} \\ a_{31} & a_{32} & a_{33} \cdots. & a_{3n} \\ \cdots\cdots\cdots\cdots\cdots\cdots & & & \\ a_{n1} & a_{n2} & a_{n3} \cdots. & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ . \\ b_n \end{pmatrix}$$

$$x_1' = \{b_1 - (a_{12} * x_2 + a_{13} * x_3 \cdots. \quad a_{1n} * x_n)\} / a_{11}$$
$$x_2' = \{b_2 - (a_{21} * x_1 + a_{23} * x_3 \cdots. \quad a_{2n} * x_n)\} / a_{22}$$
$$x_3' = \{b_3 - (a_{31} * x_1 + a_{32} * x_2 \cdots. \quad a_{2n} * x_n)\} / a_{33}$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$x_n' = \{b_n - (a_{n1} * x_1 + a_{n2} * x_2 \cdots. \quad a_{nn-1} * x_{n-1})\} / a_{nn}$$

1. x1, x2, … xnの初期値を0として、上記の式を使ってx1', x2', … xn'を求める。
2. x1', x2', … xn'の値をx1, x2, … xnに代入し、
3. 再び上記の式を使ってx1', x2', … xn'を求める。これを繰り返す。

データ並列:どのような処理？

- 例えば以下の3元連立方程式ヤコビの反復法で解く

$$
\begin{aligned}
2x + y + z &= 7 \\
x + 2y - z &= 2 \\
x + y + 4z &= 15
\end{aligned}
$$

1）x_1, x_2, x_3の初期値を0として、下記の式を使ってx_1', x_2', x_3'を求める。
2）x_1', x_2', x_3'の値をx_1, x_2, x_3に代入し、再び下記の式を使ってx_1', x_2', x_3'を求める。これを繰り返す。

$$
\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}
\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}
=
\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}
$$

$$
\begin{aligned}
x_1' &= \{b_1 - (a_{12} * x_2 + a_{13} * x_3)\} / a_{11} \\
x_2' &= \{b_2 - (a_{21} * x_1 + a_{23} * x_3)\} / a_{22} \\
x_3' &= \{b_3 - (a_{31} * x_1 + a_{32} * x_2)\} / a_{33}
\end{aligned}
$$

データ並列:どのような処理？

- 魔球の正体では、N=1,870万およびN=623万元の2つの連立方程式を10万回解く。

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \cdots. & a_{1n} \\ a_{21} & a_{22} & a_{23} \cdots. & a_{2n} \\ a_{31} & a_{32} & a_{33} \cdots. & a_{3n} \\ \cdots\cdots\cdots\cdots\cdots\cdots & & & \\ a_{n1} & a_{n2} & a_{n3} \cdots. & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ . \\ b_n \end{pmatrix}$$

$$x_1' = \{b_1 - (a_{12} * x_2 + a_{13} * x_3 \cdots. \quad a_{1n} * x_n)\} / a_{11}$$
$$x_2' = \{b_2 - (a_{21} * x_1 + a_{23} * x_3 \cdots. \quad a_{2n} * x_n)\} / a_{22}$$
$$x_3' = \{b_3 - (a_{31} * x_1 + a_{32} * x_2 \cdots. \quad a_{2n} * x_n)\} / a_{33}$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$x_n' = \{b_n - (a_{n1} * x_1 + a_{n2} * x_2 \cdots. \quad a_{nn-1} * x_{n-1})\} / a_{nn}$$

データ並列:どのように並列処理する？

- スカラー並列コンピュータの場合

$$
\begin{aligned}
x_1' &= \{b_1 - (a_{12} * x_2 + a_{13} * x_3 \cdots . \quad a_{1n} * x_n)\} / a_{11} \\
x_2' &= \{b_2 - (a_{21} * x_1 + a_{23} * x_3 \cdots . \quad a_{2n} * x_n)\} / a_{22} \\
x_3' &= \{b_3 - (a_{31} * x_1 + a_{32} * x_2 \cdots . \quad a_{2n} * x_n)\} / a_{33} \\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
x_n' &= \{b_n - (a_{n1} * x_1 + a_{n2} * x_2 \cdots . \quad a_{nn-1} * x_{n-1})\} / a_{nn}
\end{aligned}
$$

各式は
独立して
解ける

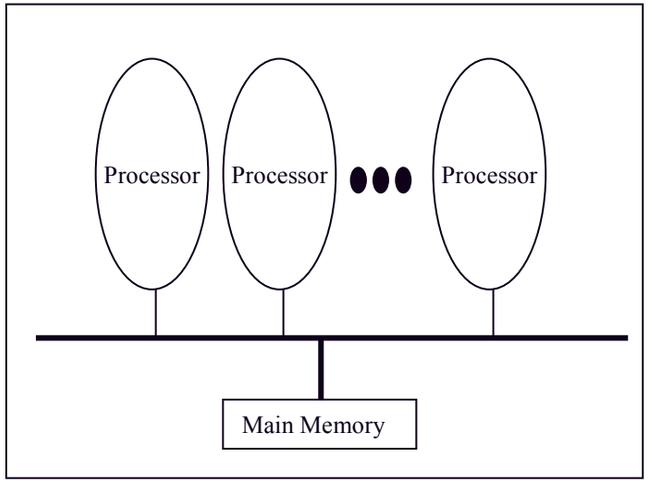
プロセッサ毎に各式を計算させよう

プロセッサ	プロセッサ	プロセッサ
$x_1' = \{7 - (x_2 + x_3)\} / 2$	$x_2' = \{2 - (x_1 - x_3)\} / 2$	$x_3' = \{15 - (x_1 + x_2)\} / 4$
$x_1' = 7 / 2 = 3.5$	$x_2' = 1$	$x_3' = 15 / 4 = 3.75$
データ交換		
$x_1' = \{7 - (1 + 3.75)\} / 2$	$x_2' = \{2 - (3.5 - 3.75)\} / 2$	$x_3' = \{15 - (3.5 + 1)\} / 4$
データ交換		

どうやってデータ交換するか？

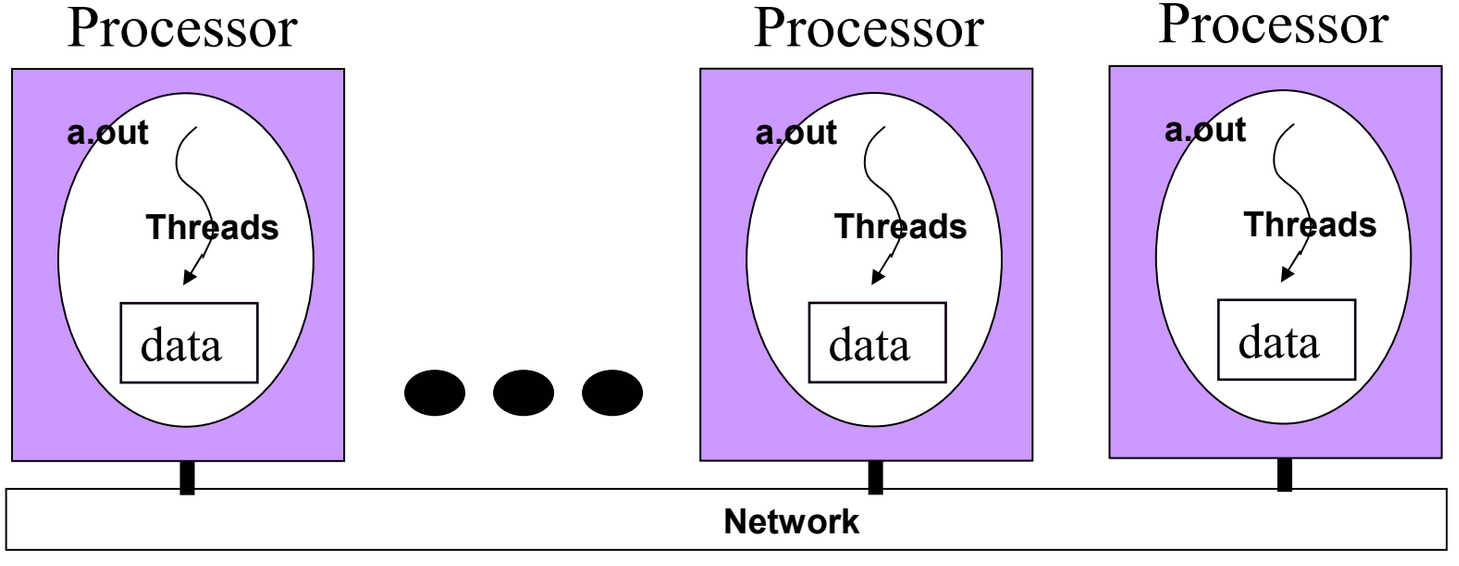
データ並列:どのように並列処理する？

- スカラー並列コンピュータの場合
  - 共有メモリをもつスカラー並列コンピュータ
    - 各プロセッサは共有するメモリをアクセスできる
    - スレッドプログラミング

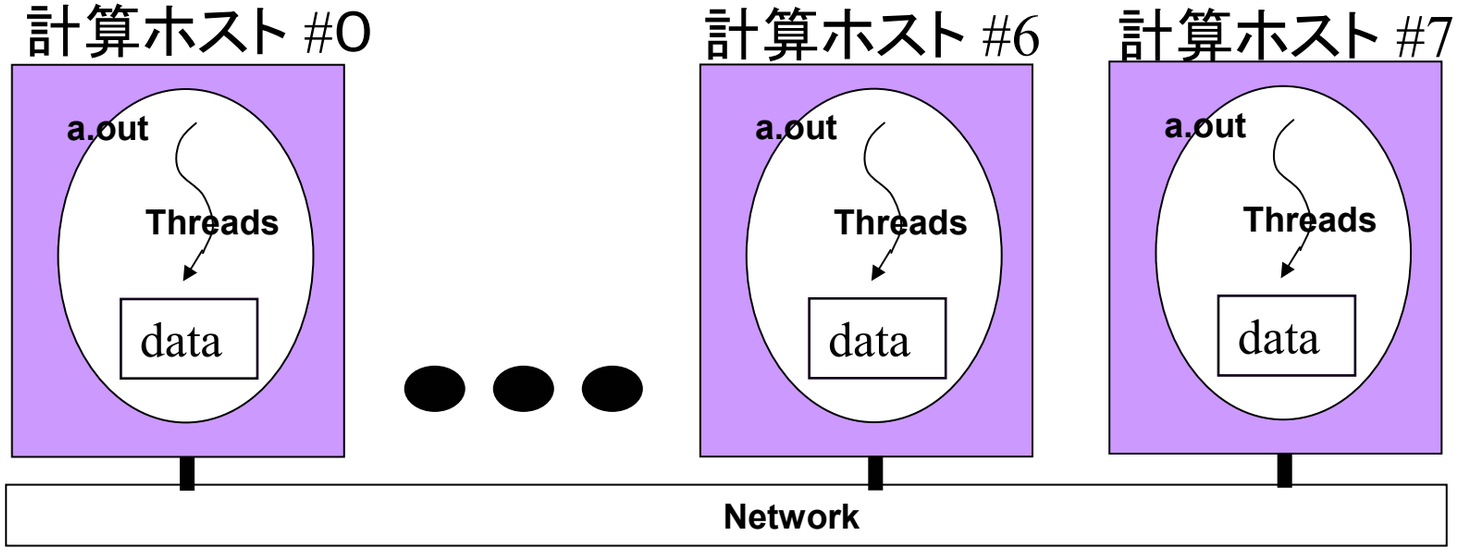


データ並列:どのように並列処理する？

- スカラー並列コンピュータの場合
  - 分散メモリ型並列コンピュータ
    - それぞれの計算ホストはローカルなメモリ上に独自のデータを持つ
    - 通信ライブラリでデータ交換、同期を行う



SPMD（Single Program Multiple Data）



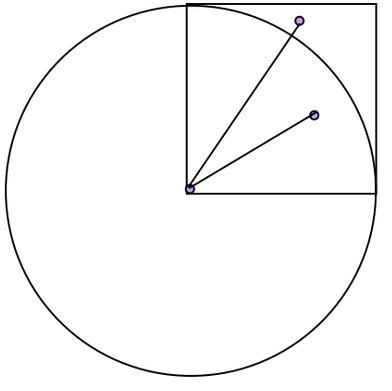
- 同一プログラムが実行される
- それぞれの計算ホストはローカルなメモリ上に独自のデータを持つ
- 通信ライブラリでデータ交換、同期を行う

データ並列:なぜデータ並列と呼ぶ？

- データ並列とは？
  - あるデータの塊に対する演算を並列に実行するから
  - データの量に応じて並列度が上がる

Embarrassingly Parallel(EP)の例

- Embarrassingly Parallel Computation
  - A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process
  - 例:モンテカルロ計算
    - 乱数を使って近似解を求める
    - 例:円周率を求める
      - 乱数を振って点(a, b)を決める
      - その点が1/4円の中に納まっているか計算する
        - $a^2 + b^2 < 1$
      - N回中、K回が1/4円に収まっていれば
        - $\pi / 4 = k/n$
  - 通信性能は重要ではない
  - 並列性が高い

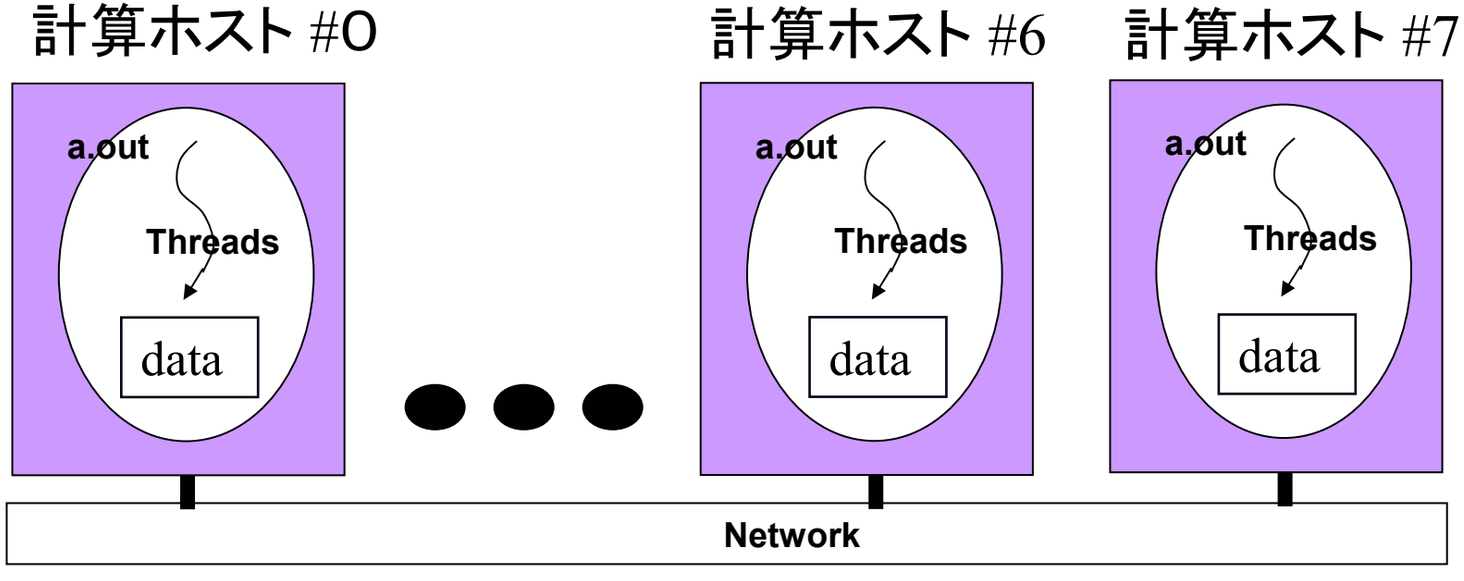


MPI通信ライブラリの概要

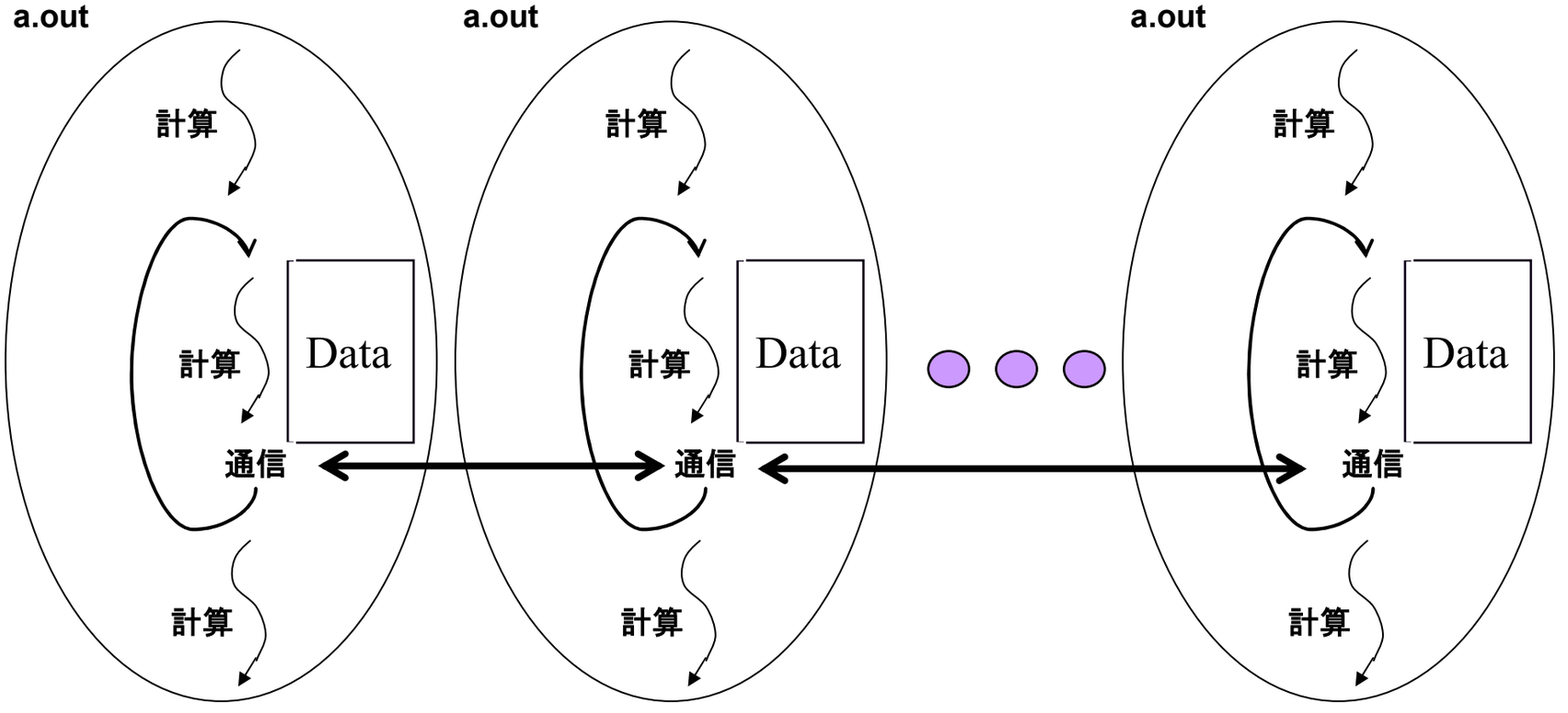
- 背景
  - 1990年初頭、各コンピュータベンダは独自の通信ライブラリを提供
  - ユーザプログラムのポータビリティ問題が深刻化
    - あるマシン上で開発したアプリケーションが他のベンダのマシンで動かず、プログラムを書き直さなければならない
  - 米国国立研究所と並列計算機メーカが中心となり策定した通信ライブラリ
- 歴史
  - 1994年　MPI-1.0
  - 1995年　MPI-1.1
  - 1997年　MPI-2.0 & MPI-1.2

MPI-1.2通信ライブラリが想定する実行モデル SPMD（Single Program Multiple Data）

- 同一プログラムが実行される
- それぞれの計算ホストはローカルなメモリ上に独自のデータを持つ
- 通信ライブラリでデータ交換、同期を行う



MPIアプリケーションの典型的実行パターン



- 逐次計算部分
- 並列実行部分

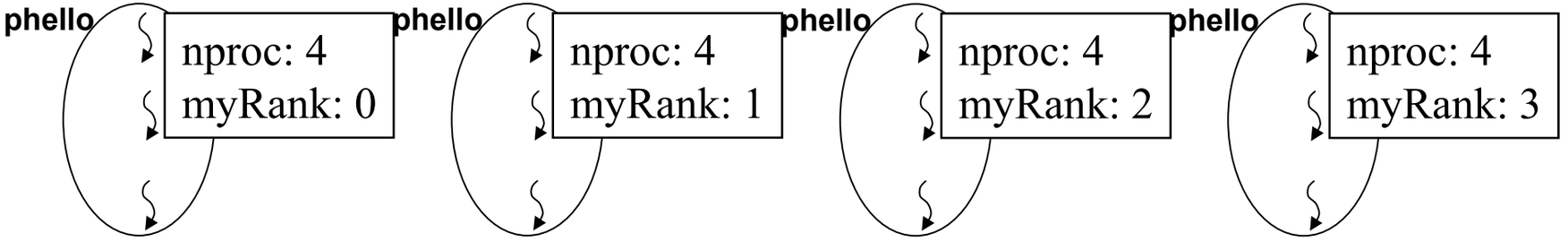
```
loop {
        ローカル計算
        通信
}
```

簡単なMPIプログラム: 並列hello world

```
#include <mpi.h>
main(int argc, char **argv)
{
   int       nprocs, myRank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
   printf("hello world");          fflush(stdout);
   printf("I'm from Rank %d", myRank);   fflush(stdout);
   MPI_Finalize();
}
```

```
% mpicc -o phello phello.c
% mpirun -np 4 ./phello
hello world
I'm from Rank 0
hello world
I'm from Rank 1
hello world
hello world
I'm from Rank 3
I'm from Rank 2
```



使用したMPI通信ライブラリ関数

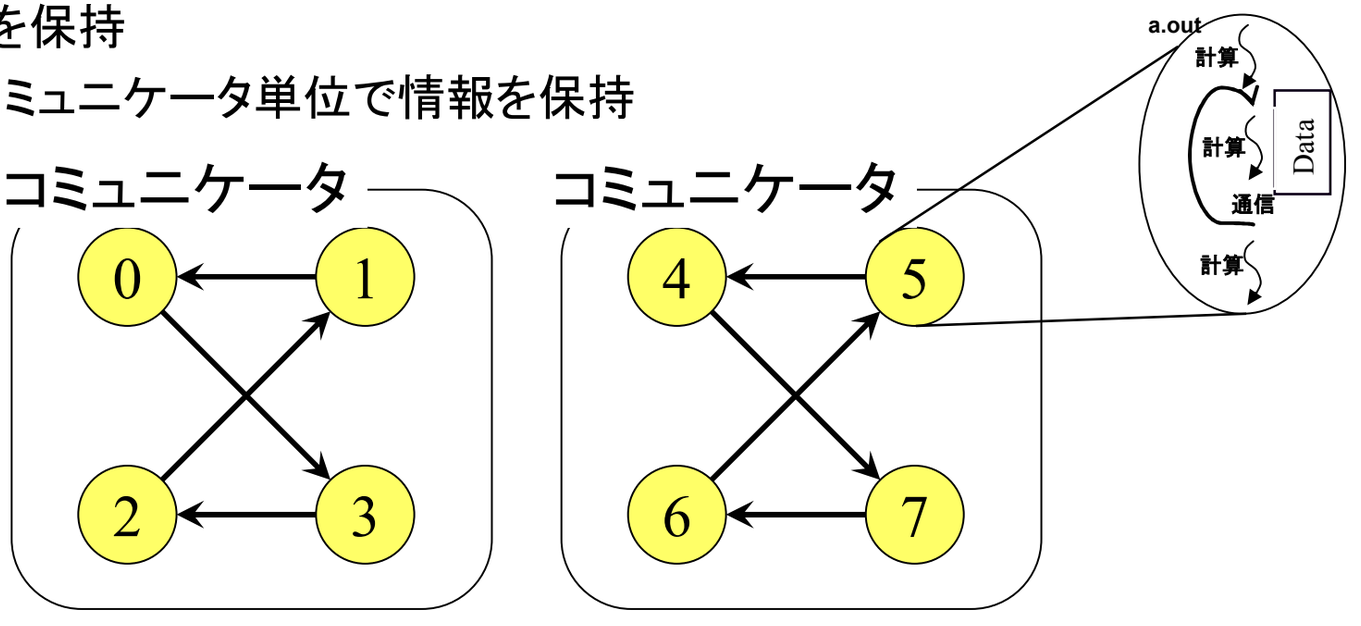
- MPI_Init(&argc, &argv)
  - 本関数はMPI通信ライブラリの初期化を行う
  - MPIプログラムの最初に必ず呼ばなければいけない
- MPI_Comm_size(MPI_COMM_WORLD, &size);
  - プロセッサの数を得る
- MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  - 自分のランク(プロセッサ番号)を得る
- MPI_Finalize()
  - MPIの終了

MPI通信ライブラリの概要

- MPI-1.2
  - コミュニケータと呼ばれる媒体による通信
  - アプリケーション記述を容易にする集団通信機能
    - バリア同期、全対全、スキャッタ/ギャザ、リダクション、
- MPI-2.0
  - リモートメモリ書き込み・読み出し（One-sided Communication）
  - 動的プロセス生成
  - 並列 I/O

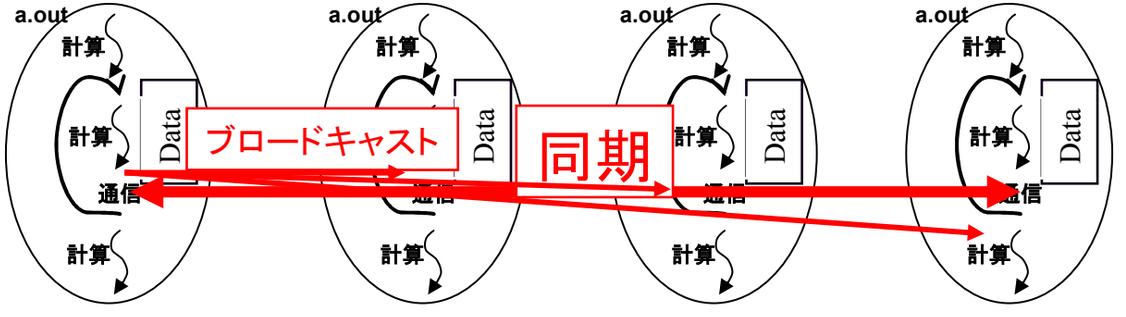
MPI通信ライブラリの特徴

- コミュニケータとは？
  - 通信に必要な情報（コンテキスト）を保持
  - プロセスのグループを保持
    - プロセスは順序付けされ、それぞれランク（番号）を持つ
  - 仮想トポロジーを保持
  - 属性を保持
    - コミュニケータ単位で情報を保持



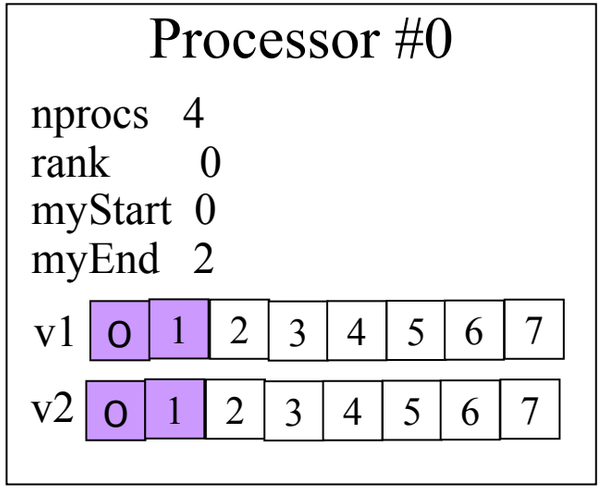
MPIが提供する通信パターン

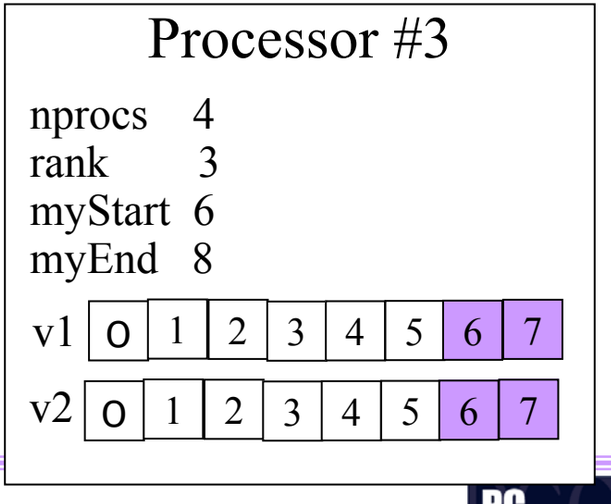
- 1対1通信
  - Blocking
    - MPI_Bsend (Buffered mode), MPI_Ssend (Synchronous mode), MPI_Rsend (Ready mode)
    - MPI_Recv
  - Nonblocking
    - MPI_Ibsend, MPI_Issend, MPI_Irsend
    - MPI_Irecv
- 集団通信
  - バリア同期
  - ブロードキャスト
  - ギャザ
  - スキャッタ
  - リデュース
  - スキャン



簡単なMPIプログラム：ベクトルの内積

```
#include <mpi.h>
void       initdata(char*, double*, double*, int, int);
double     v1[SIZE], v2[SIZE];
main(int argc, char **argv)
{
    int       nprocs, rank,  myStart, myEnd, i;
    double tmp, result;
   if (argc != 2) usage();
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   initdata(argv[1], v1, v2, nprocs, rank);
   myStart = (SIZE/nprocs)*rank;
   myEnd = (SIZE/nprocs)*rank  + SIZE/nprocs;
   tmp = 0;
   for (i = myStart; i < myEnd; i++)
              tmp += v1[i]*v2[i];
    MPI_Reduce(&tmp, &result, 1, MPI_DOUBLE,
                     MPI_SUM, 0, MPI_COMM_WORLD);
   MPI_Finalize();
}
```





使用したMPI通信ライブラリの関数

- MPI_Reduce(&ldata, &result, count,
  MPI_DOUBLE, MPI_SUM, root, COMM_WORLD)
  - 各ノード上のldata(データタイプはMPI_DOUBLE)の値を加算(MPI_SUM)し、その結果をrootで示されるノードのresult変数に格納する。
  - データタイプ
    - MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, ….
  - 操作
    - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, ….

ベクトルの内積（データの配布）

その１

```
#include <stdio.h>
#include <mpi.h>
void
initdata(char *file, double *v1, double *v2, int nprocs, int rank)
{
    FILE          *fp;
    int            dst;
    MPI_Status   stat;
    if (rank == 0) {
        if ((fp = fopen(file, “r”)) == NULL) usage2();
       fread(v1, sizeof(double), SIZE, fp);
       fread(v2, sizeof(double), SIZE, fp);
       flose(fp);
       for (dst = 1; dst < nprocs; dst++) {
             MPI_Send(v1, SIZE, MPI_DOUBLE, dst, 0, MPI_COMM_WORLD);
             MPI_Send(v2, SIZE, MPI_DOUBLE, dst, 0, MPI_COMM_WORLD);
       }
   } else {
     MPI_Recv(v1, SIZE, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);
     MPI_Recv(v2, SIZE, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &stat);
   }
}
```

MPI通信ライブラリの関数

- MPI_Send(&data, count, MPI_DOUBLE, dst, tag,
  MPI_COMM_WORLD)
  - double型のdataをcountサイズ分、dstノードに送信する。Tagを指定することにより、メッセージを分類して送ることが可能。
- MPI_Recv(&data, count, MPI_DOUBLE, src, tag,
  MPI_COMM_WORLD , &stat)
  - double型のdataをcountサイズ分、srcノードから受信する。Tagで指定したメッセージのみを受信。　statにエラーが生じたかどうかの状態が格納される。

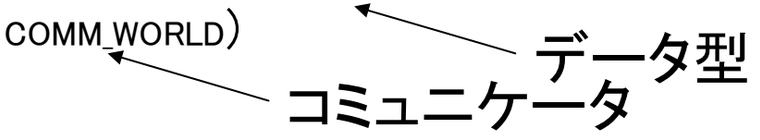
ベクトルの内積（データの配布）

その2

```
#include <stdio.h>
#include <mpi.h>
void
initdata(char *file, double *v1, double *v2, int nprocs, int rank)
{
    FILE          *fp;
    int            dst;
    MPI_Status   stat;
    if (rank == 0) {
        if ((fp = fopen(file, “r”)) == NULL) usage2();
       fread(v1, sizeof(double), SIZE, fp);
       fread(v1, sizeof(double), SIZE, fp);
       fclose(fp);
    }
    MPI_Bcast(v1, SIZE, MPI_DOUBLE, 0,  MPI_COMM_WORLD);
    MPI_Bcast(v2, SIZE, MPI_DOUBLE, 0,  MPI_COMM_WORLD);
}
```

使用したMPI通信ライブラリの関数

- MPI_Bcast(&data, count, MPI_DOUBLE, root,
  COMM_WORLD)

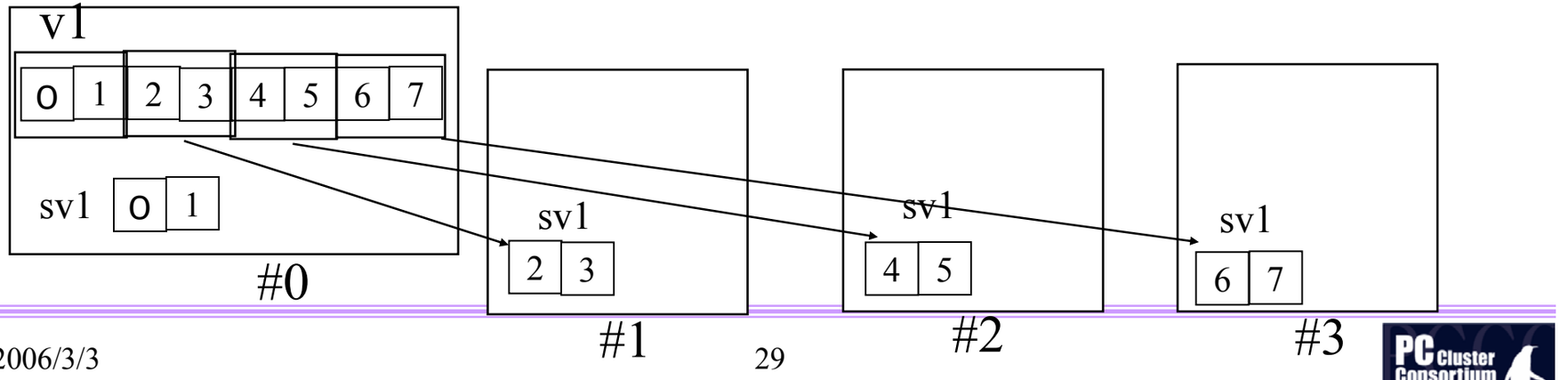


  - rootで示されるノード上のdouble型のdataをcountサイズ分、全てのノードに送信。

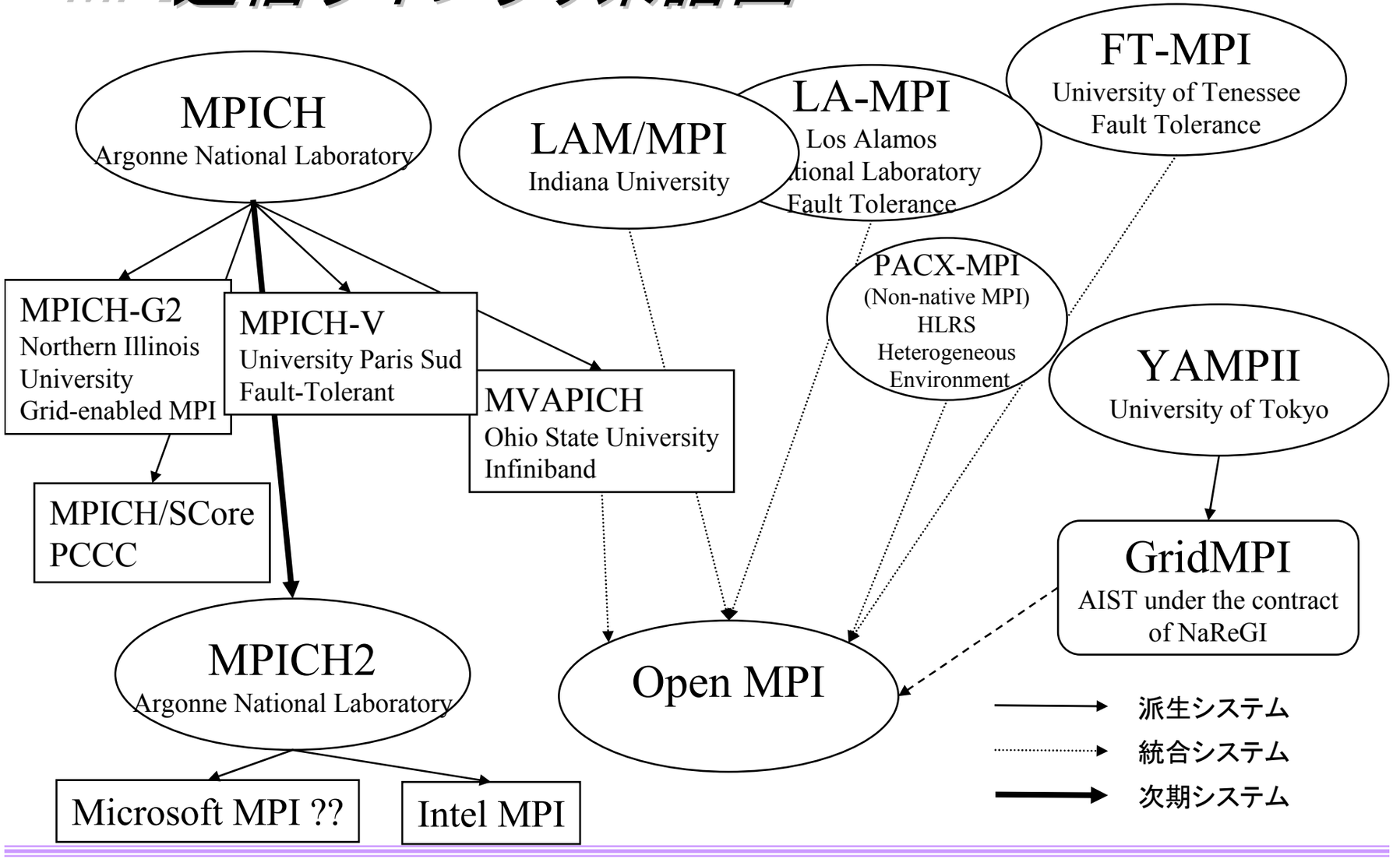
MPI通信ライブラリの関数

- MPI_Scatter(&sdata, scount, MPI_DOUBLE,
  &rdata, rcount, MPI_DOUBLE,
  root, MPI_COMM_WORLD)
  - rootで示されるノード上のdouble型のsdataをcountサイズ分を分割してノードに送信。

MPI_Scatter(v1, 2, MPI_DOBULE, sv1, 2, 0, MPI_COMM_WORLD);



MPI通信ライブラリ系譜図



MPICHとその派生通信ライブラリ

- 歴史
  - 米国アルゴンヌ国立研究所によって開発
- 特徴
  - MPI規格化時のリファレンスモデル
  - 様々なプラットフォームに移植される
    - MPICH/SCoreもその一つ
    - MVAPICH
  - MPICHの派生
    - MPICH-V
    - MPICH-G2
- MPICH2
  - MPICHの最新版
  - Intel社MPIは、MPICH2をベースとしている

LAM/MPI

- 歴史
  - Ohio Supercomputing Centerで開発、その後、Notre Dameに開発が引きつがれる。さらにIndiana大学に引き継がれている。
- 特徴
  - Linux上系ディストリビューションに同梱されている
  - 最新版7.1.1
  - Dynamic Shared Objects
  - サポートデバイス
    - Socket, Infiniband, Myrinet

Open MPI

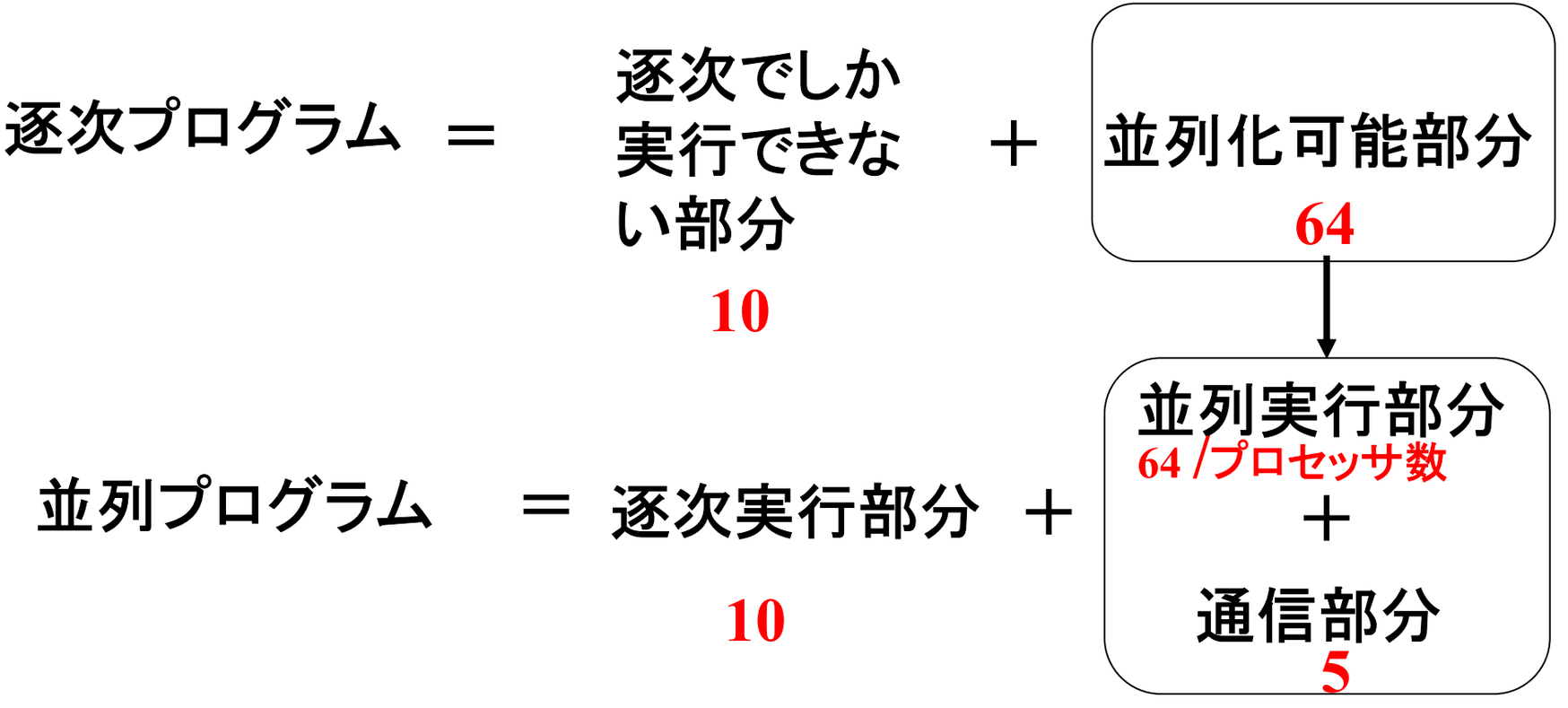
- 歴史
  - 2004年、LAM/MPI, FT-MPI, LA-MPI, MVAPIC, PACX-MPI 開発グループが集まって開発。2004年SC04でアナウンス
  - 2005年秋リリース
- 特徴
  - MPI-2
  - Thread safety
  - Network and process fault tolerance
  - Run-time tunable
  - Component Architecture

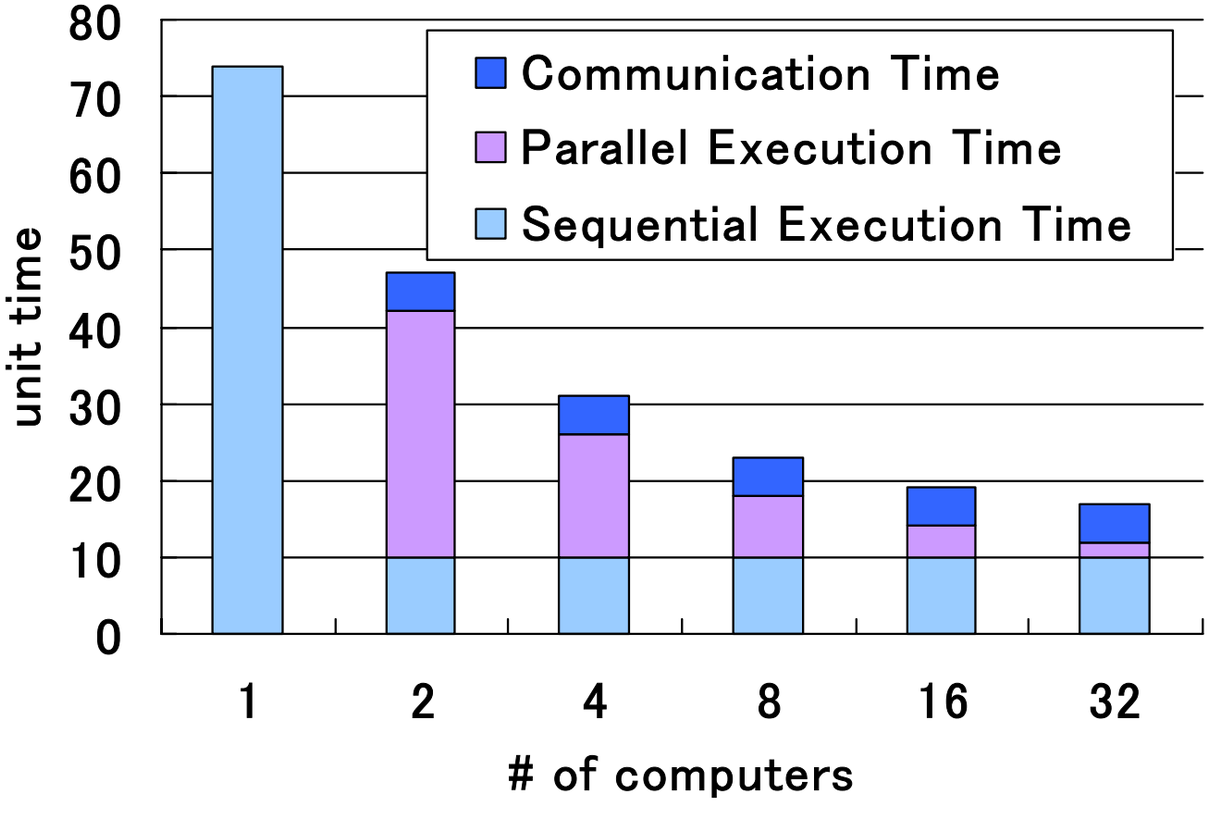
YAMPII and GridMPI

- 歴史
  - 2002年1月から個人的にYAMPIIの開発開始。ライセンスはLGPL
    - 既にいくつものMPI実装があり研究として出来ないし資金もなかったため（常磐高速バスの中＆自宅）
  - 2003年から文部科学省リーディングプロジェクト超高速コンピュータ網形成プロジェクト（通称NaReGIプロジェクト）におけるGridMPIの核としてYAMPIIを使用
- 特徴
  - YAMPII　　http://www.il.is.s.u-tokyo.ac.jp/yampii/
    - MPI-2 (version 0.9では限定的)
    - Thread Safe
    - 同一バイナリでSCore環境でもLAN環境でも実行可能
    - 性能と安定の両立
    - MPE (MPI Parallel Environment)利用可能
  - GridMPI　　http://www.gridmpi.org/
    - グリッド環境上で高性能通信環境を実現
      - 通信遅延が大きい通信路におけるTCP/IP性能劣化問題を解決
    - IMPI (Inter-operable MPI)プロトコル規格を踏襲

並列性能

アムダールの法則:プログラムの並列化できない部分が並列性能を制限する





実行時間＝逐次処理時間（10）＋64/プロセッサ数＋通信時間（5）

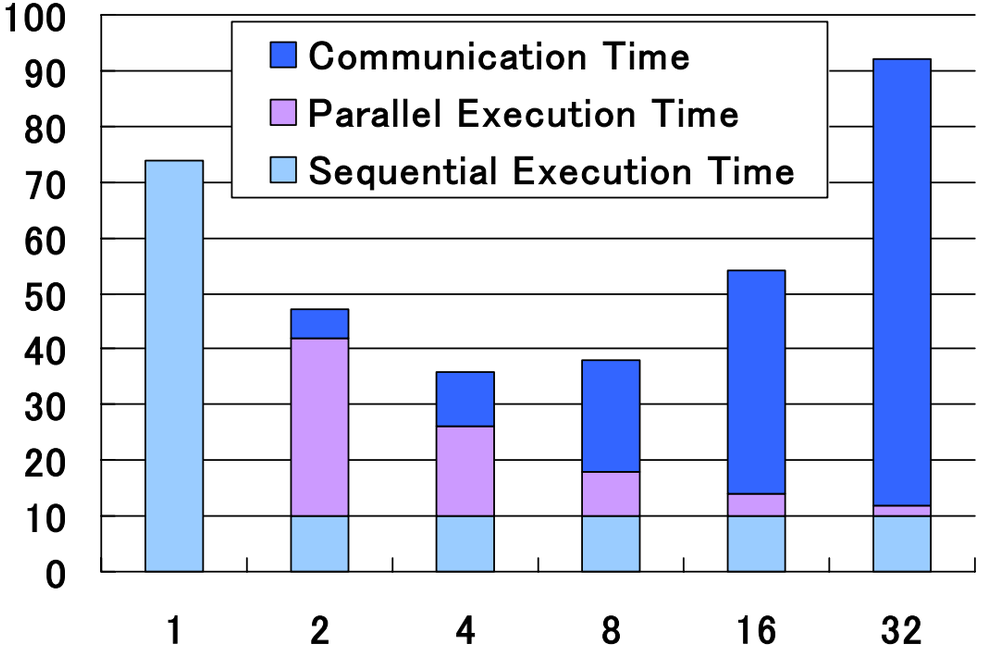
並列性能

アムダールの法則:プログラムの並列化できない部分が並列性能を制限する

逐次プログラム ＝ 逐次でしか実行できない部分 **10** ＋ 並列化可能部分 **64**

並列プログラム ＝ 逐次実行部分 **10** ＋ 並列実行部分 **64** /プロセッサ数 ＋ 通信部分 **5** ＊プロセッサ数

大抵は、通信時間はプロセッサ数を乗じるほど時間はかからないが、ここではわかりやすく極端な例とした。



実行時間＝逐次処理時間（10）＋64/プロセッサ数＋通信時間（5＊プロセッサ数）

性能の指標

- 通信性能
  - バンド幅： 1秒間に転送できるデータ容量
  - 通信遅延：最小メッセージを
- 通信パターン
  - １対１通信性能 (point to point)
  - 双方向通信性能 (bidirection)
  - バイセクションバンド幅 (bisection bandwidth)
  - 集団通信性能
- アプリケーションレベル性能
- 台数効果 (scalability)
  - 台数に応じて性能が向上するかどうか

性能指標

- アプリケーションレベル性能
  - 実際に使用するアプリケーションを使用して性能を測るのが一番
  - アプリケーション実行には入出力に関する制限があり、どのような環境でも簡単に実行できるとは限らない
  - 並列処理の特徴を抽出したプログラムを作る
    - ベンチマーク
- ベンチマーク
  - Linpack
    - 連立一次方程式をLU分解により解くコード
    - TOP500リストのランキングに使用
  - NASA Parallel Benchmarks
    - NASAが開発した流体系ベンチマークプログラム集
  - 姫野ベンチマーク
    - ポアソン方程式をヤコビ反復法で解くコード
  - SPEC HPC
  - SPLASH
  - STREAM
  - ……

NAS Parallel Benchmarks

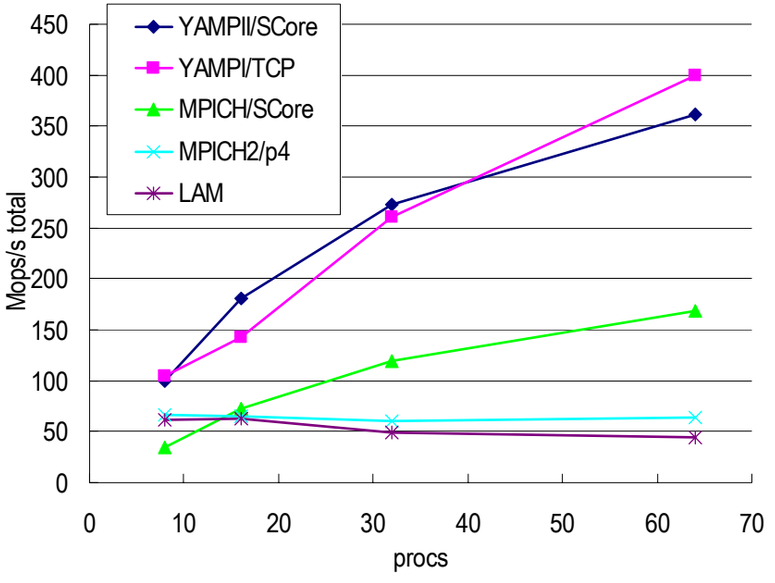
- FT
  - 3-D partial differential equation solution using FFT
  - 全対全通信
- IS
  - Integer Sort
  - 全対全通信
- CG
  - conjugate gradient method
  - 隣接通信
- LU
  - LU solver
- BT
  - block tridiagonal solver
- MG
  - multigrid kernel
- EP
  - Embarrassingly parallel kernel

評価環境

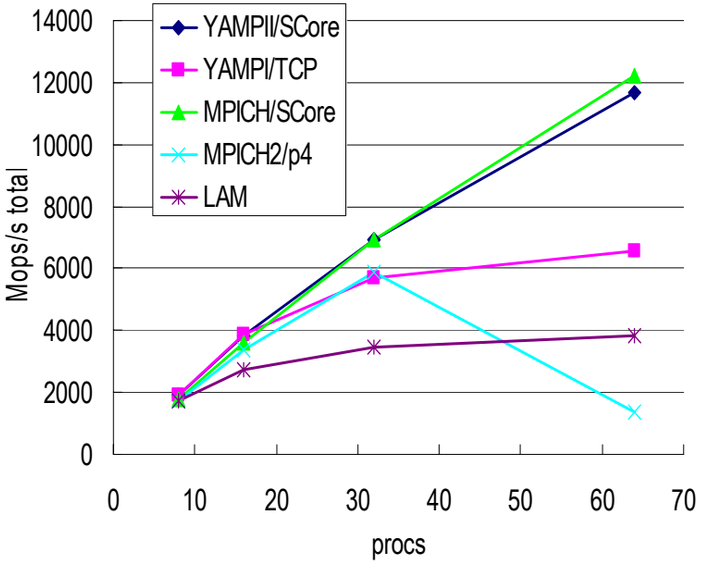
- ノード計算機
  - 日本AMD社 AMD Asia Cluster Lab
    - プロセッサ：Opteron 246 x 1
    - ノード数：64台
    - ネットワーク：Broadcom 1GE
- ネットワークスイッチ
  - Baystack5510-48T x 2
- ソフトウェア環境
  - SCore 5.8.3 on Fedora Core 3
  - YAMPII Version 0.9-alpha
  - LAM/MPI Version 7.0.6
  - MPICH 2.0

NAS Parallel Benchmarks

IS (Class B)



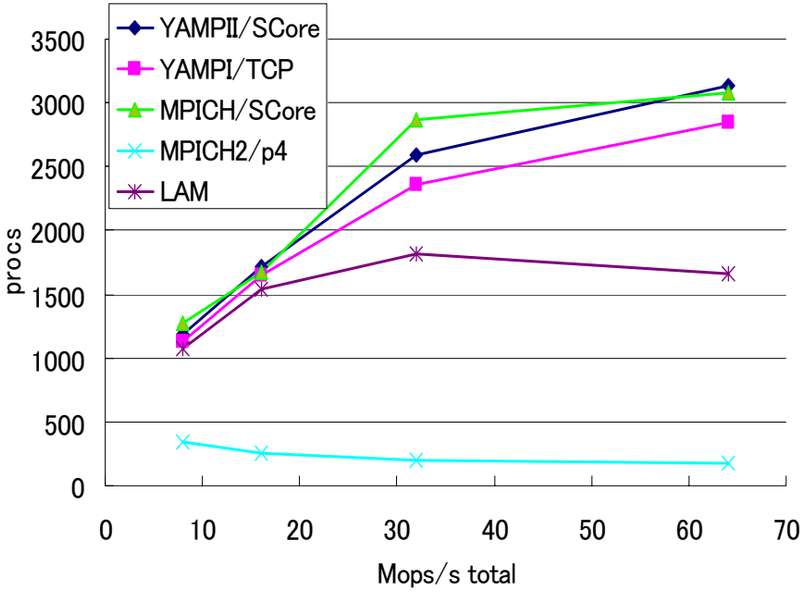
FT (Class B)



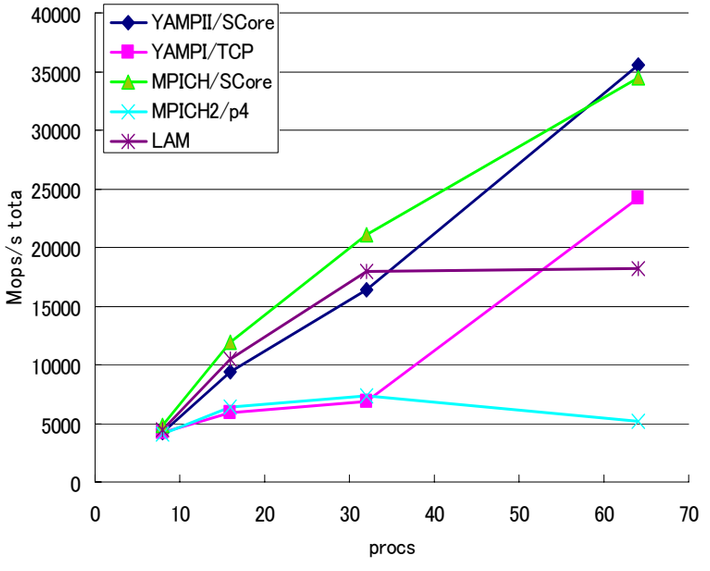
- ISは全対全通信かつメッセージ量が多いアプリケーション
  - YAMPIIの性能が良いのは全対全アルゴリズムの違い
- SCore上のYAMPIIが性能が一番高い
- FTの64プロセス実行では、YAMPIIがMPICHに負けているのは今後の課題（多分MPI_Wait実装の問題）

NAS Parallel Benchmarks

CG (Class B)



LU (Class B)



- 32台のところでYAMPIIが遅い
  - LUの32台の性能はパラメータ設定を間違えた可能性あり
- 今後、結果の解析が必要

まとめ

- 並列プログラムの分類
  - データ並列
  - コントロール並列
  - ジョブ配布
- MPI通信ライブラリの概要
  - MPICH, LAM/MPI, Open MPI, YAMPII, GridMPI
- MPI通信ライブラリを使った簡単なプログラム例
- MPI通信ライブラリ性能