XcalableMP

並列プログラミング言語XcalableMP チュートリアル

中尾昌広

筑波大学計算科学研究センター

チュートリアルの流れ

- XcalableMPの概要
- XcalableMPのプログラム例
- XcalableMPのプログラム方法
  - データマッピング
  - ワークマッピング
  - ノード間通信
- その他の機能

XcalableMPの開発背景

- 並列プログラミングの現状
  - 大半はMPI (Message Passing Interface)を利用
  - MPIはプログラミングコストが大きい

- 目標
  - もっと生産性の高い並列プログラミング言語を！！
  - 広く用いてもらえる並列プログラミング言語XcalableMPの開発

リファレンス実装を筑波大学で開発（オープンソース）

Omni XcalableMP Compiler（http://www.xcalablemp.org）

XcalableMP（XMP）

- 分散メモリシステムを対象としたプログラミングモデル
- High Performance Fortranなどの既存並列言語の機能を引き継ぎつつ，その反省点を踏まえて開発
- OpenMP likeな指示文を逐次プログラムに挿入して並列化
  - 既存の逐次プログラムからの移行を容易化
  - 教育コストの低減

} 生産性の向上

- Performance Awareness
  - 通信が発生する箇所は明示的に指示
  - それ以外はローカルメモリにアクセス

} 性能チューニングの容易化

- C言語とFortran言語に対応【本スライドではC言語版を説明】

開発体制

- 大学・企業・研究所のメンバで構成される、
  次世代並列プログラミング言語検討委員会を作成
  - コミュニティの経験（特にHigh Performance Fortran）と意向を取り入れた仕様検討
  - 開発後の普及体制までを考慮（実用化を重視）
  - 標準化を目指して、world-wide communityに提案
- メンバー
  - 大学　：筑波大、東大、京都大、九州大
  - 企業　：富士通、NEC、日立
  - 研究所：理研、NIFS、JAXA、JAMSTEC/ES

現在はPCクラスタコンソーシアムの並列プログラミング言語
XcalableMP規格部会で言語規格活動を行っています

XMPのプログラム例

```
int array[MAX];
#pragma xmp nodes p(*)
#pragma xmp template t(0:MAX-1)
#pragma xmp distribute t(block) onto p
#pragma xmp align array[i] with t(i)

main(){
#pragma xmp loop on t(i) reduction(+:res)
  for(i = 0; i < MAX; i++){
    array[i] = func(i);
    res += array[i];
  }
}
```

data mapping

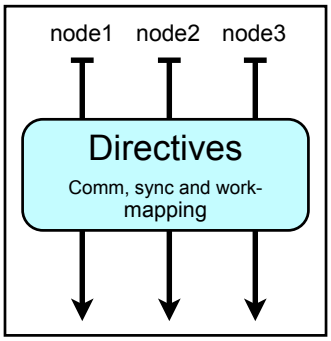
work mapping & reduction

同じプログラムをMPIで書くと

```
int array[MAX];

main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    dx = MAX/size;
    llimit = rank * dx;
    if(rank != (size -1)) ulimit = llimit + dx;
    else ulimit = MAX;

    temp_res = 0;
    for(i=llimit; i < ulimit; i++){
            array[i] = func(i);
            temp_res += array[i];
            }

    MPI_Allreduce(&temp_res, &res, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Finalize( );
}
```

XMPのプログラミングモデル（1/2）

- 基本的な実行モデルはSPMD
  - 各ノードで同じプログラムが並列実行
  - 指示文により通信 or 同期などが発生
- XMPが提供する機能
  - データマッピング（配列データを各プロセスに分散配置）
  - ワークマッピング（各プロセスで処理を分担）
  - （ブロードキャストや同期などの）定型的な集合通信
  - 簡易な片側通信の記法
- MPIおよびOpenMPとの混在利用も可能

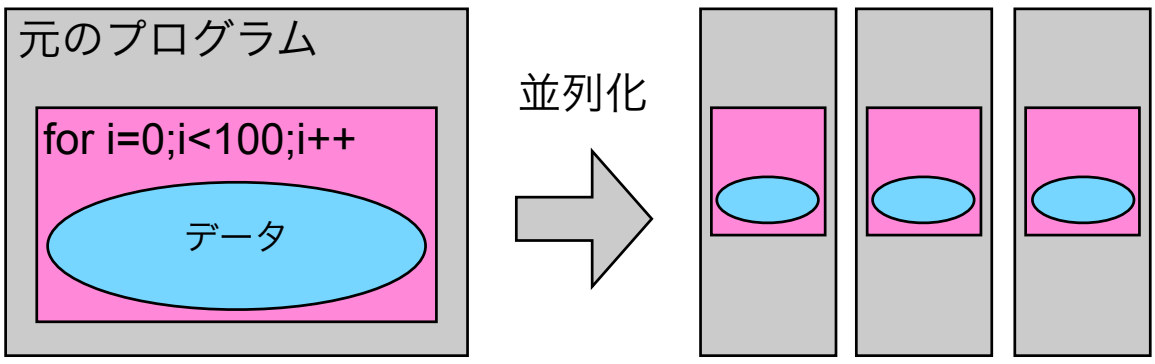


XMPのプログラミングモデル（2/2）

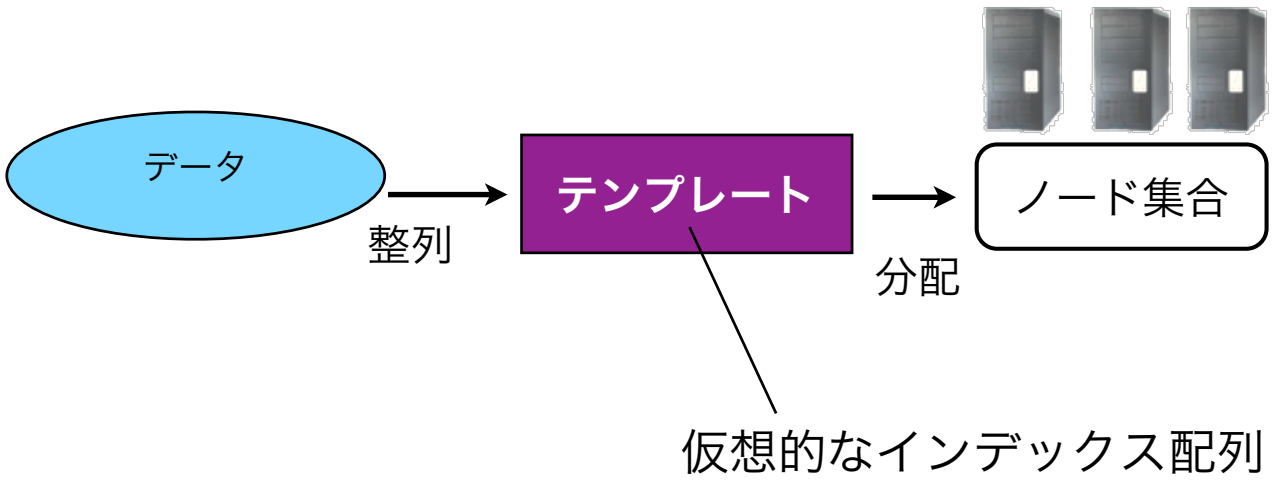
- グローバルビューモデル（like as HPF）
  - プログラマはデータマッピングとワークマッピングを
    指示文を使って記述する
- ローカルビューモデル（like as Coarray Fortran）
  - 言語拡張により片側通信を簡易に記述
  - Fortran版のXMPはCoarray Fortranの上位互換になるように設計

データマッピング（1/3）

- データを各ノードに割り当てること
- 高速な並列プログラムを作成するためには
  - 処理の分割のみでなく，データの配置が大切
  - データの移動が少なくなるように，
    はじめからローカルメモリにデータを配置する



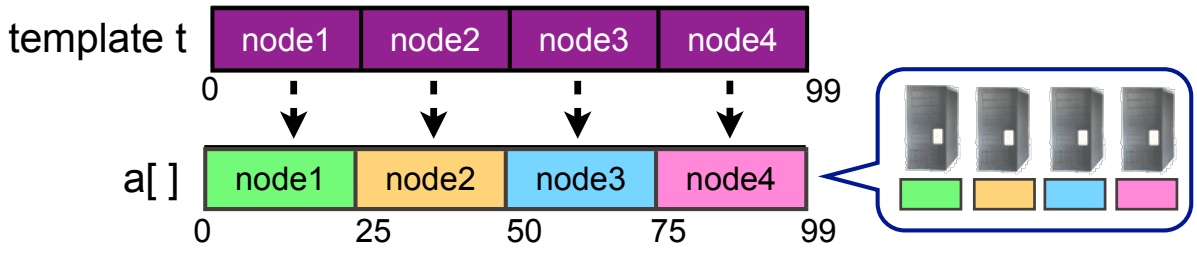
データマッピング（2/3）



データマッピング（3/3）

```
#pragma xmp template t(0:99)          // 0から99のindexを持つtemplateを作成
#pragma xmp nodes p(4)                // ノード集合を定義（今回は4ノード）
#pragma xmp distribute t(block) onto p // templateをノード集合pに分配する
int a[100];
#pragma xmp align a[i] with t(i)      // 配列aをtemplateに整列する
main(){
  int i;
#pragma xmp loop on t(i)
  for(i=0;i<100;i++)
    a[ i ] = func( i );
}
```

このループの並列化を考える
配列aをマッピングする



データのマッピング

1. 仮想的なインデックスの作成

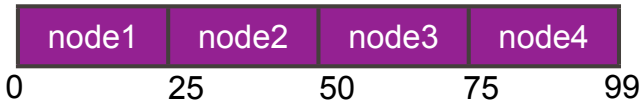
#pragma xmp template t(0:99)

2. 計算に用いるノードの定義

#pragma xmp nodes p(4)

3. templateをノードに分配

#pragma xmp distribute t(block) onto p



4. templateをもとに配列を整列

```
int a[100];
#pragma xmp align a[i] with t(i)
```

データ分散の例

例1

```
#pragma xmp template t(0:19)
#pragma xmp nodes p(4)
#pragma xmp distribute t(block) onto p
```

node	index
p(1)	0, 1, 2, 3, 4
p(2)	5, 6, 7, 8, 9
p(3)	10, 11, 12, 13, 14
p(4)	15, 16, 17, 18, 19

例2

```
#pragma xmp template t(0:19)
#pragma xmp nodes p(4)
#pragma xmp distribute t(cyclic) onto p
```

node	index
p(1)	0, 4, 8, 12, 16
p(2)	1, 5, 9, 13, 17
p(3)	2, 6, 10, 14, 18
p(4)	3, 7, 11, 15, 19

block-cyclic分散やユーザが自由に要素数を設定することも可能

Loop指示文の例（1/2）

```
#pragma xmp template t(0:19)
#pragma xmp nodes p(4)
#pragma xmp distribute t(block) onto p
int a[20];
#pragma xmp align a[i] with t(i)

int main(void){
  int i;
#pragma xmp loop on t(i)
  for(i=0;i<20;i++)
    a[ i ] = func( i );

  return(0);
}
```

並列化したいfor文の直前に挿入する

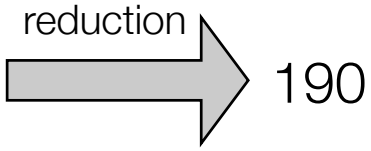
ループが各ノードに分かれて並列処理される

Loop指示文の例（2/2）

```
int main(void){
  int i, sum = 0;

#pragma xmp loop on t(i) reduction(+:sum)
  for(i=0;i<20;i++)
    sum += i;

  return(0);
}
```

ループ文の終了時に集計計算を行う.
提供している演算子は
+, *, &, &&, |, || など

ノード	ループで行われる計算	各sum
p(1)	0+1+2+3+4	10
p(2)	5+6+7+8+9	35
p(3)	10+11+12+13+14	60
p(4)	15+16+17+18+19	85



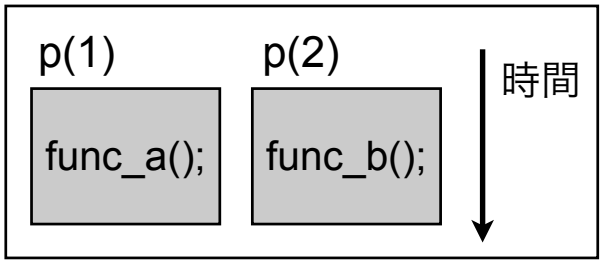
Task指示文

#pragma xmp task on *[nodes-ref | template-ref]*

- 直後の処理を *nodes-ref*または*template-ref* で指定したノードが実行する

```
#pragma xmp nodes p(2)

#pragma xmp tasks
{
#pragma xmp task on p(1)
  func_a();
#pragma xmp task on p(2)
  func_b();
}
```

ノード番号1はfunc_a()を
ノード番号2はfunc_b()を
並列に実行する



通信+αの指示文一覧

指示文	機能
reduction	集約計算
bcast	ブロードキャスト
barrier	バリア同期
shadow/reflect	袖領域の生成/同期
gmove	分散配列用のデータ通信
coarray	片側通信（Put/Get）

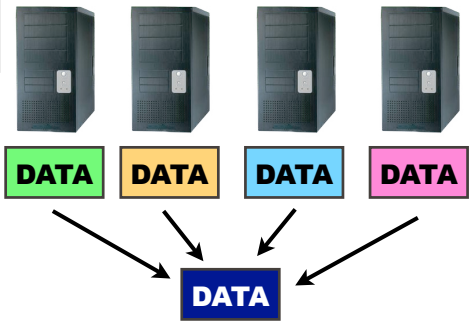
reduction指示文

#pragma xmp reduction(*reduction-kind : variable[, variable]...*)
[on nodes-ref | template-ref]

- ループ指示文の項を参照．下記の2つの結果は同じである．

```
sum=0;
#pragma xmp loop on t(i) reduction(+:sum)
  for(i=0;i<20;i++)
    sum += func( i );
```

```
sum = 0;
#pragma xmp loop on t(i)
  for(i=0;i<20;i++)
    sum += func( i );

#pragma xmp reduction(+:sum)
```



最大値や総計を計算

bcast指示文

#pragma xmp bcast *variable [, variable] ...*
[from nodes-ref] [on nodes-ref | template-ref]

- 変数*variable* を *from* 節で指定したノードが
  on 節で指定したノードにブロードキャストする

```
#pragma xmp nodes p(4)
・・・
#pragma xmp bcast (a) from p(1) on p(1:5)
```

p(1)が持つ変数aを
指定したpに送信する

from節が無い場合はp(1)が選ばれる

on節が無い場合はすべてのプロセスに送信する

barrier指示文

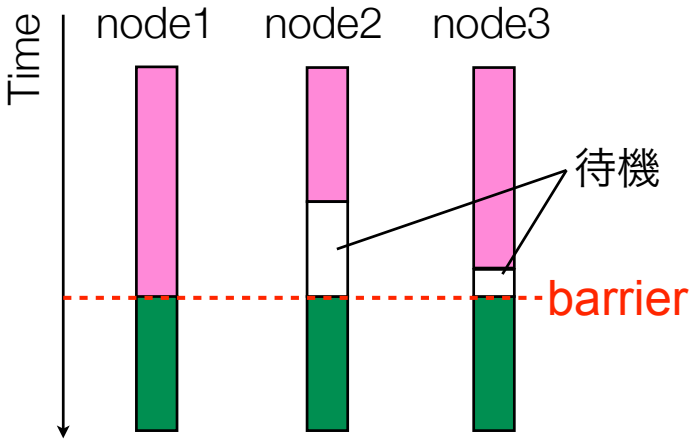
```
#pragma xmp barrier
```

- 同期を取る

```
func_a();
#pragma xmp barrier
func_b();
```

：func_a

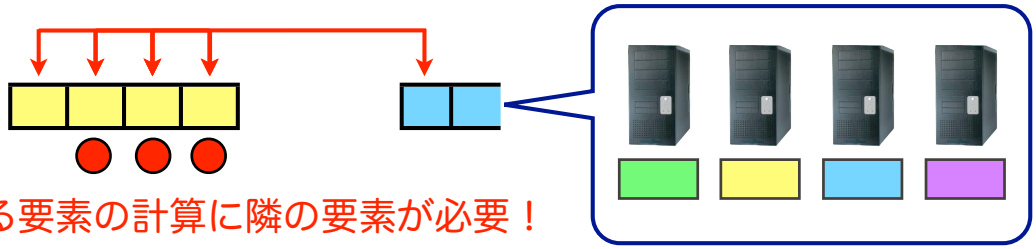
：func_b



shadow指示文

```
#pragma xmp shadow array-name[shadow-width
 [, shadow-width] ...]
```

- 他のノードに割り当てられた要素を参照
- *shadow-width*は参照範囲の大きさ。int型の定数 or 範囲



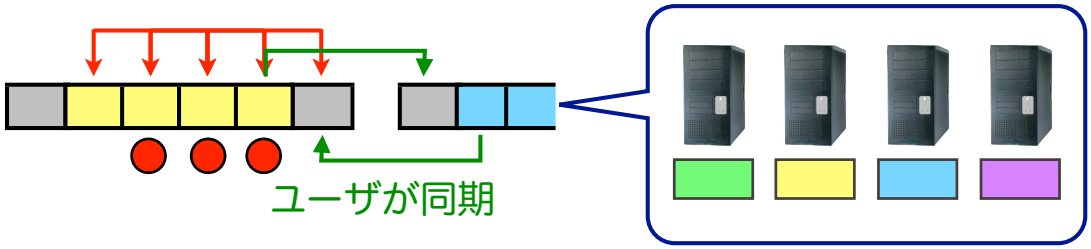
ローカルにあるデータかどうかを場合分けし,

通信を挿入するのは面倒

shadow指示文

#pragma xmp shadow *array-name[shadow-width*
[, shadow-width] ...]

- 他のノードに割り当てられた要素を参照



元の配列要素の横に，配列要素を追加する。
その要素を袖領域（シャドウ領域）と呼ぶ

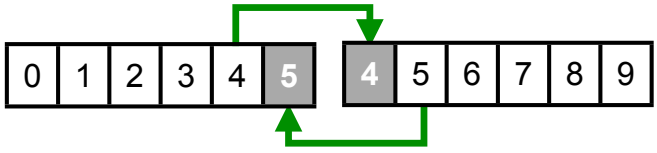
shadow指示文の具体例

```
#pragma xmp template t(0:9)
#pragma xmp nodes p(2)
int a[10], b[10];
#pragma align [i] with t(i) :: a, b
#pragma xmp shadow a[1:1]
・・・
#pragma xmp loop on t(i)
for(i=0;i<10;i++)
  a[i] = init(i);     // a[]の初期化
#pragma xmp reflect (a)

#pragma xmp loop on t(i)
for(i=1;i<9;i++)
  b[i] = a[i-1] + a[i] + a[i+1];
・・・
```

分散配列の端に、同期をとりたい
要素を1つ追加する
（下図の灰色が袖領域）

p(1)　　p(2)

袖領域の同期をとりたい場合
reflect指示文を用いる



コピーが発生する

gmove指示文

#pragma xmp gmove *[in | out]*

代入文

- 分散された配列についてデータ移動を行う
  （コロンを使って処理範囲を指定）

```
#pragma xmp gmove
a[0:3] = b[3:3];
```

開始点　転送要素数

分散配列b[3]からb[5]の要素が
分散配列a[0]からa[2]に
転送される

どのデータがどのノードに配置されているかを意識する必要がない

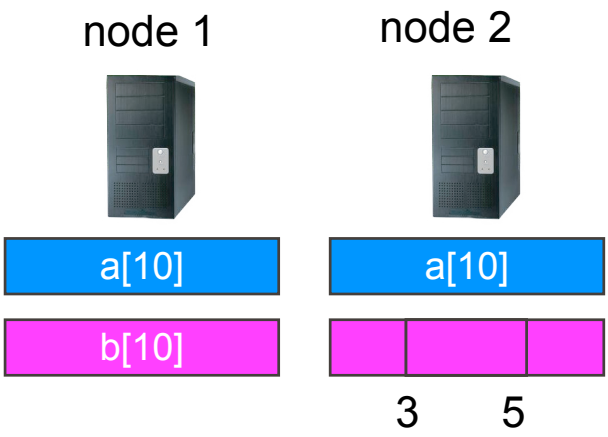
Co-array

- ローカルデータに対する片側通信（Get/Put）の記述
- Fortran版XMPはCo-array Fortranと互換

ノード2が持つb[3:3]の
データをa[0:3]に転送する

```
#pragma xmp coarray b:[*]
      :
if(me == 1)
  a[0:3] = b[3:3]:[2];       // Get
```

コロンの後の[]はノード番号を表す

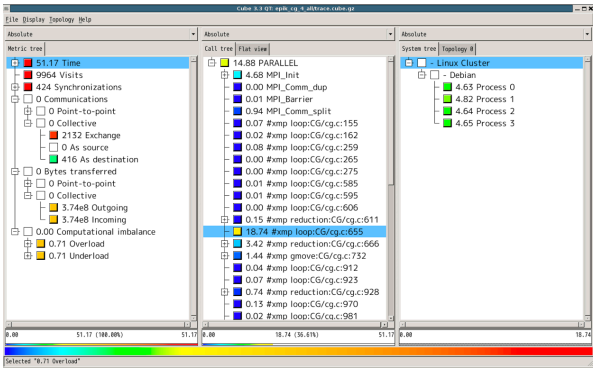


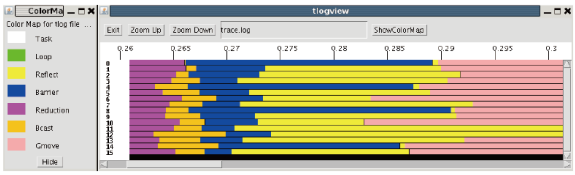
その他の機能

- Easy Hybrid Programming

```
#pragma xmp loop on t(i) threads
for(i=2;i<=10;i++){...}
```

- Interface of scalasca and tlog profiling tools
  （この機能はOmni XMP Compilerのみ）





```
#pragma xmp gmove profile
    :
#pragma xmp loop on t(i) profile
```

まとめ

- 簡易に並列アプリケーションを作成できる並列言語 XcalableMPの説明
- Omni XcalableMP Compiler http://www.xcalablemp.org
  - マニュアル
  - チュートリアル
  - サポートメーリングリスト
- 今後の拡張予定
  - For accelerators(GPU, etc)
  - Parallel I/O
  - Interface of MPI library, and so on