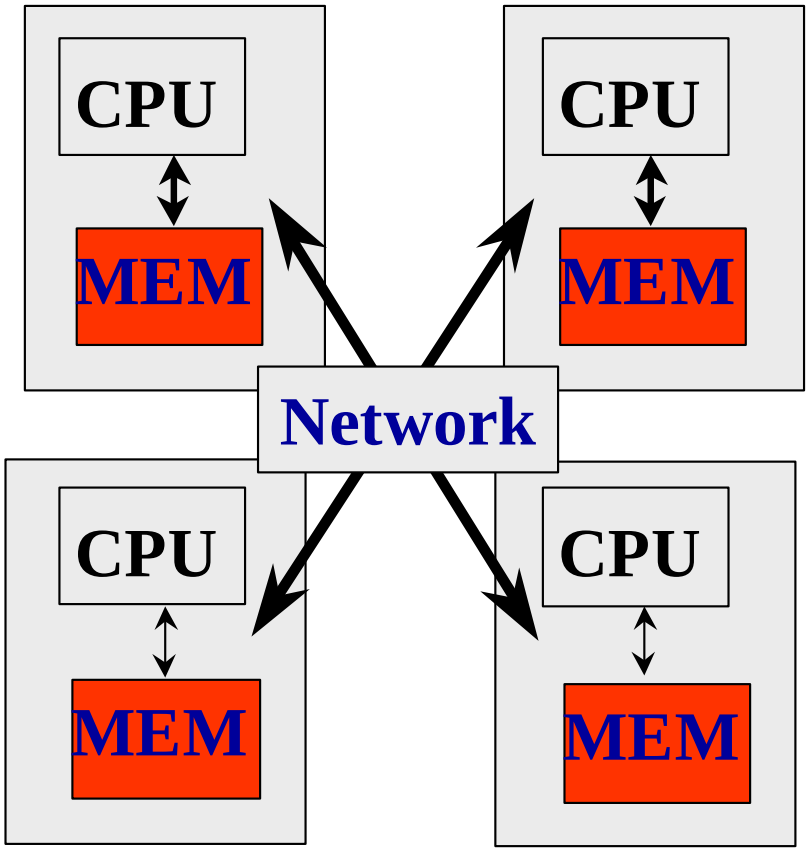
並列プログラミング環境

プログラミング環境特論

2008年1月24日

建部修見

分散メモリ型計算機



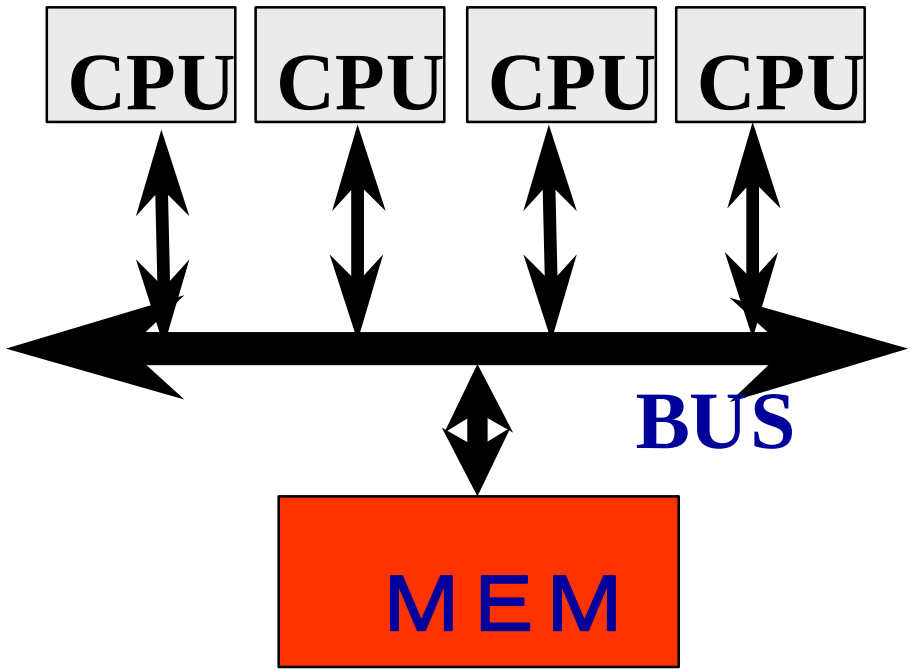
◆CPUとメモリという一つの計算機システムが、ネットワークで結合されているシステム

◆それぞれの計算機で実行されているプログラムはネットワークを通じて、データ(メッセージ)を交換し、動作する

◆超並列（MPP：Massively Parallel Processing)コンピュータ

◆クラスタ計算機

共有メモリ型計算機



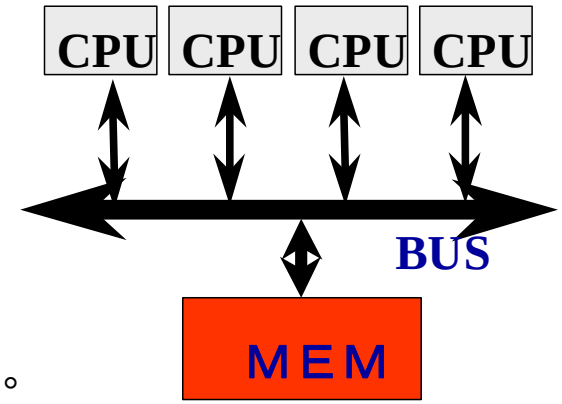
◆複数のCPUが一つのメモリにアクセスするシステム。

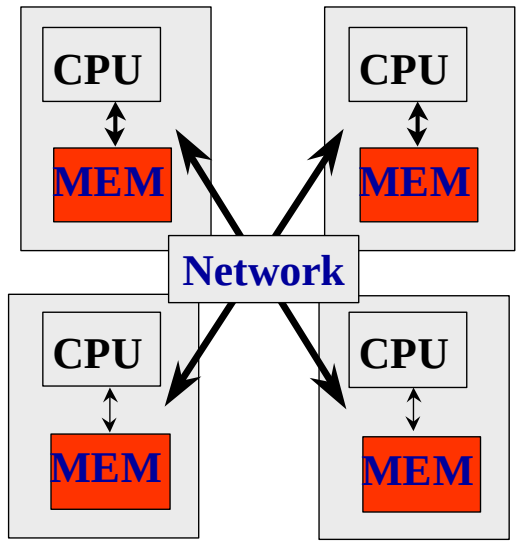
◆それぞれのCPUで実行されているプログラム（スレッド）は、メモリ上のデータにお互いにアクセスすることで、データを交換し、動作する。

◆大規模サーバ

並列処理の利点

- 計算能力が増える。
  - １つのCPUよりも多数のCPU。
- メモリの読み出し能力（バンド幅）が増える。
  - それぞれのCPUがここのメモリを読み出すことができる。
- ディスク等、入出力のバンド幅が増える。
  - それぞれのCPUが並列にディスクを読み出すことができる。
- キャッシュメモリが効果的に利用できる。
  - 単一のプロセッサではキャッシュに載らないデータでも、処理単位が小さくなることによって、キャッシュを効果的に使うことができる。
- 低コスト
  - マイクロプロセッサをつかえば。





→ クラスタ技術

並列プログラミング

- メッセージ通信 (Message Passing)
  - 分散メモリシステム（共有メモリでも、可）
  - プログラミングが面倒、難しい
  - プログラマがデータの移動を制御
  - プロセッサ数に対してスケーラブル

- 共有メモリ (shared memory)
  - 共有メモリシステム（DSMシステムon分散メモリ）
  - プログラミングしやすい（逐次プログラムから）
  - システムがデータの移動を行ってくれる
  - プロセッサ数に対してスケーラブルではないことが多い。

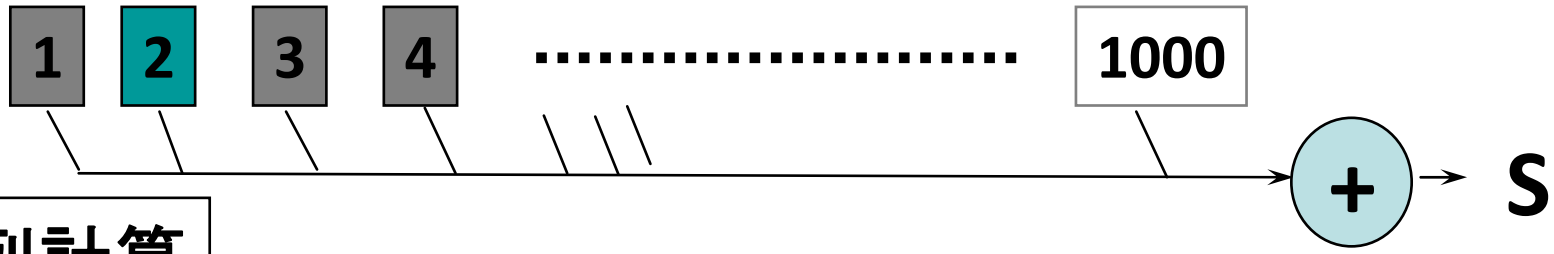
並列プログラミング

- メッセージ通信プログラミング
  - MPI, PVM
- 共有メモリプログラミング
  - マルチスレッドプログラミング
    - Pthread, Solaris thread, NT thread
  - OpenMP
    - 指示文によるannotation
    - thread制御など共有メモリ向け
  - HPF
    - 指示文によるannotation
    - 並列構文
    - distributionなど分散メモリ向け
- 自動並列化
  - 逐次プログラムをコンパイラで並列化
    - コンパイラによる解析には制限がある。指示文によるhint
- Fancy parallel programming languages

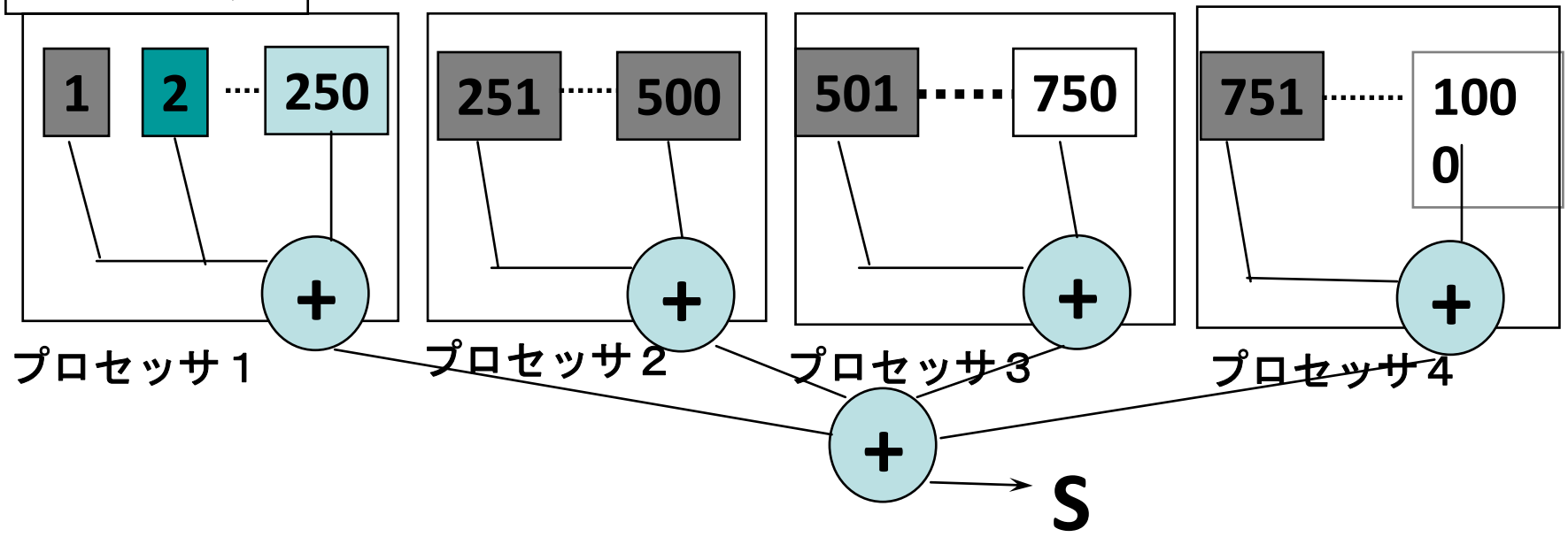
並列処理の簡単な例

for (i = 0; i < 1000; i++)
S += A[i]

逐次計算



並列計算



POSIXスレッドによるプログラミング

- スレッドの生成

Pthread, Solaris thread

```
for (t = 1; t < n_thd; t++){
  r = pthread_create(thd_main, t)
}
thd_main(0);
for (t = 1; t < n_thd; t++)
            pthread_join();
```

- ループの担当部分の分割
- 足し合わせの同期

```
double s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c, b, e, i;  double ss;
  c = 1000 / n_thd;
  b = c * id;
  e = b + c;
  ss = 0.0;
  for (i = b; i < e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return (0);
}
```

OpenMPによるプログラミング

これだけで、OK!

```
#pragma omp parallel for reduction(+:s)
  for (i = 0; i < 1000; i++) s += a[i];
```

OpenMPとは

- 共有メモリマルチプロセッサの並列プログラミングのためのプログラミングモデル
  - ベース言語(Fortran/C/C++)をdirective（指示文）で並列プログラミングできるように拡張
- 米国コンパイラ関係のISVを中心に仕様を決定
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - 現在、OpenMP 3.0が策定中
- URL
  - http://www.openmp.org/

背景

- 共有メモリマルチプロセッサシステムの普及
  - SGI Cray Origin
    - ASCI Blue Mountain System
  - SUN Enterprise
  - PC-based SMPシステム
  - そして、いまや　マルチコア！
- 共有メモリマルチプロセッサシステムの並列化指示文の共通化の必要性
  - 各社で並列化指示文が異なり、移植性がない。
    - SGI Power Fortran/C
    - SUN Impact
    - KAI/KAP
- OpenMPの指示文は並列実行モデルへのAPIを提供
  - 従来の指示文は並列化コンパイラのためのヒントを与えるもの

科学技術計算とOpenMP

- 科学技術計算が主なターゲット（これまで）
  - 並列性が高い
  - コードの5%が95%の実行時間を占める(?)
    - 5%を簡単に並列化する
- 共有メモリマルチプロセッサシステムがターゲット
  - small-scale（～16プロセッサ）からmedium-scale（～64プロセッサ）を対象
  - 従来はマルチスレッドプログラミング
    - pthreadはOS-oriented, general-purpose
- 共有メモリモデルは逐次からの移行が簡単
  - 簡単に、少しずつ並列化ができる。
    - （でも、デバックはむずかしいかも）

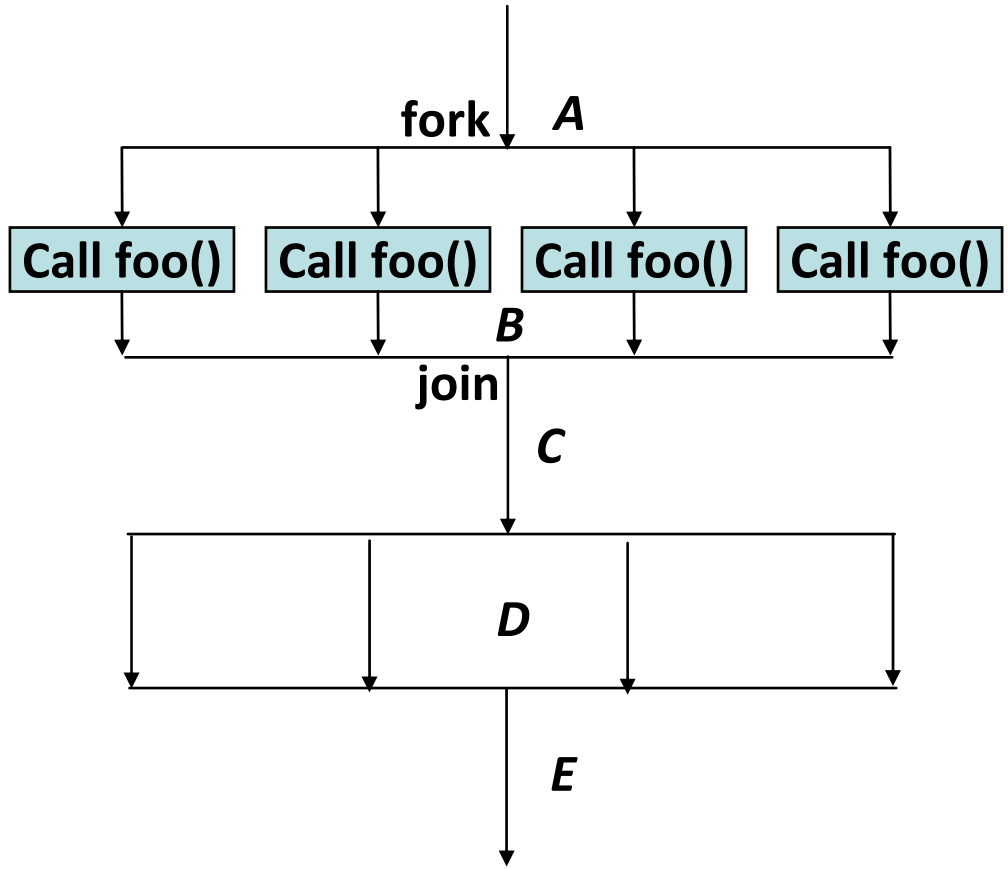
OpenMPのAPI

- 新しい言語ではない！
  - コンパイラ指示文（directives/pragma）、ライブラリ、環境変数によりベース言語を拡張
  - ベース言語：Fortran77, f90, C, C++
    - Fortran： !$OMPから始まる指示行
    - C: #pragma omp のpragma指示行
- 自動並列化ではない！
  - 並列実行・同期をプログラマが明示
- 指示文を無視することにより、逐次で実行可
  - incrementalに並列化
  - プログラム開発、デバックの面から実用的
  - 逐次版と並列版を同じソースで管理ができる

OpenMPの実行モデル

- 逐次実行から始まる
- Fork-joinモデル
- parallel region
  - 関数呼び出しも重複実行

```
 … A …
#pragma omp parallel
{
  foo(); /* ..B... */
}
… C ….
#pragma omp parallel
{
… D …
}
… E …
```



Parallel Region

- 複数のスレッド(team)によって、並列実行される部分
  - Parallel構文で指定
  - 同じParallel regionを実行するスレッドをteamと呼ぶ
  - region内をteam内のスレッドで重複実行
    - 関数呼び出しも重複実行

Fortran:

```
!$OMP PARALLEL
      …
      … parallel region
      …
!$OMP END PARALLEL
```

C:

```
#pragma omp parallel
{
  …
     … Parallel region…
     …
}
```

Work sharing構文

- Team内のスレッドで分担して実行する部分を指定
  - parallel region内で用いる
  - for 構文
    - イタレーションを分担して実行
    - データ並列
  - sections構文
    - 各セクションを分担して実行
    - タスク並列
  - single構文
    - 一つのスレッドのみが実行
  - parallel 構文と組み合わせた記法
    - parallel for 構文
    - parallel sections構文

For構文

- Forループ（DOループ）のイタレーションを並列実行
- 指示文の直後のforループは*canonical shape*でなくてはならない

```
#pragma omp for [clause…]
 for(var=lb; var logical-op ub; incr-expr)
   body
```

  - *var*は整数型のループ変数（強制的にprivate）
  - *incr-expr*
    - ++*var*,*var*++,--*var*,*var*--,*var*+=*incr*,*var*-=*incr*
  - *logical-op*
    - ＜、＜＝、＞、＞＝
  - ループの外の飛び出しはなし、breakもなし
  - *clause*で並列ループのスケジューリング、データ属性を指定

例

疎行列ベクトル積ルーチン

```
matvec(double a[],int row_start,int col_idx[],
            double x[],double y[],int n)
{
   int i, j, start, end;   double t;
#pragma omp parallel for private(j,t,start,end)
   for (i = 0; i < n; i++){
      start = row_start[i];
      end = row_start[i+1];
      t = 0.0;
      for (j = start; j < end; j++)
         t += a[j] * x[col_idx[j]];
      y[i] = t;
  }
}
```

並列ループのスケジューリング

- プロセッサ数4の場合

逐次

schedule(static,*n*)

schedule(static)

schedule(dynamic,*n*)

schedule(guided,*n*)

Data scope属性指定

- parallel構文、work sharing構文で指示節で指定
- shared(*var_list*)
  - 構文内で指定された変数がスレッド間で共有される
- private(*var_list*)
  - 構文内で指定された変数がprivate
- firstprivate(*var_list*)
  - privateと同様であるが、直前の値で初期化される
- lastprivate(*var_list*)
  - privateと同様であるが、構文が終了時に逐次実行された場合の最後の値を反映する
- reduction(*op*:*var_list*)
  - reductionアクセスをすることを指定、スカラ変数のみ
  - 実行中はprivate、構文終了後に反映

Barrier 指示文

- バリア同期を行う
  - チーム内のスレッドが同期点に達するまで、待つ
  - それまでのメモリ書き込みもflushする
  - 並列リージョンの終わり、work sharing構文でnowait指示節が指定されない限り、暗黙的にバリア同期が行われる。

```
#pragma omp barrier
```

MPIによるプログラミング

- MPI (Message Passing Interface)
- 現在、分散メモリシステムにおける標準的なプログラミングライブラリ
  - 100ノード以上では必須
  - 面倒だが、性能は出る
    - アセンブラでプログラミングと同じ
- メッセージをやり取りして通信を行う
  - Send/Receive
- コレクティブ通信
  - 総和など

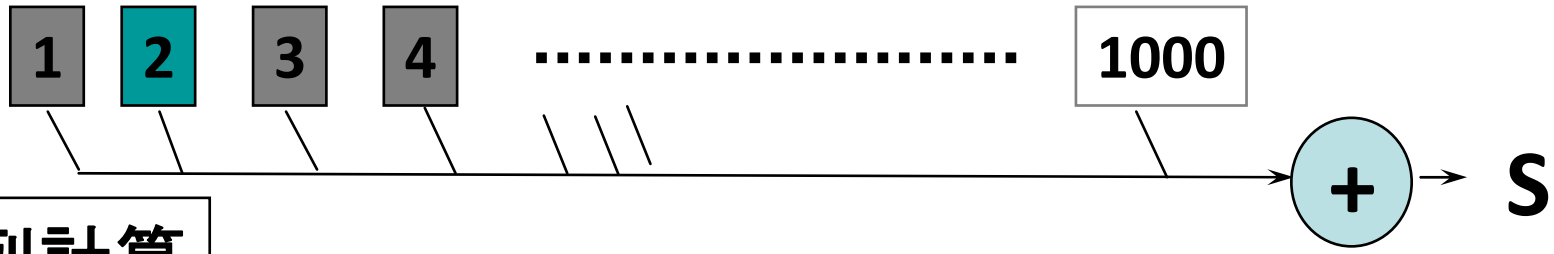
MPI – Message Passing Interface

- 仕様書
  - http://www.mpi-forum.org/
  - http://phase.hpcc.jp/phase/mpi-j/ml/
- メッセージパッシングインターフェースの標準
  - 1992 主に米国，欧州の40組織，80人以上が集まり活動開始
  - 1994 MPI-1.0
  - 1995 MPI-1.1
  - 1997 MPI-2
- 特徴
  - 豊富な通信モード，コレクティブ通信
  - 通信ドメイン（コミュニケータ），プロセストポロジなど
  - MPI-1.1でも定義されている関数の数は128
  - FORTRAN77, C, C++（, Fortran90）
  - 基本データ型のデータサイズに非依存

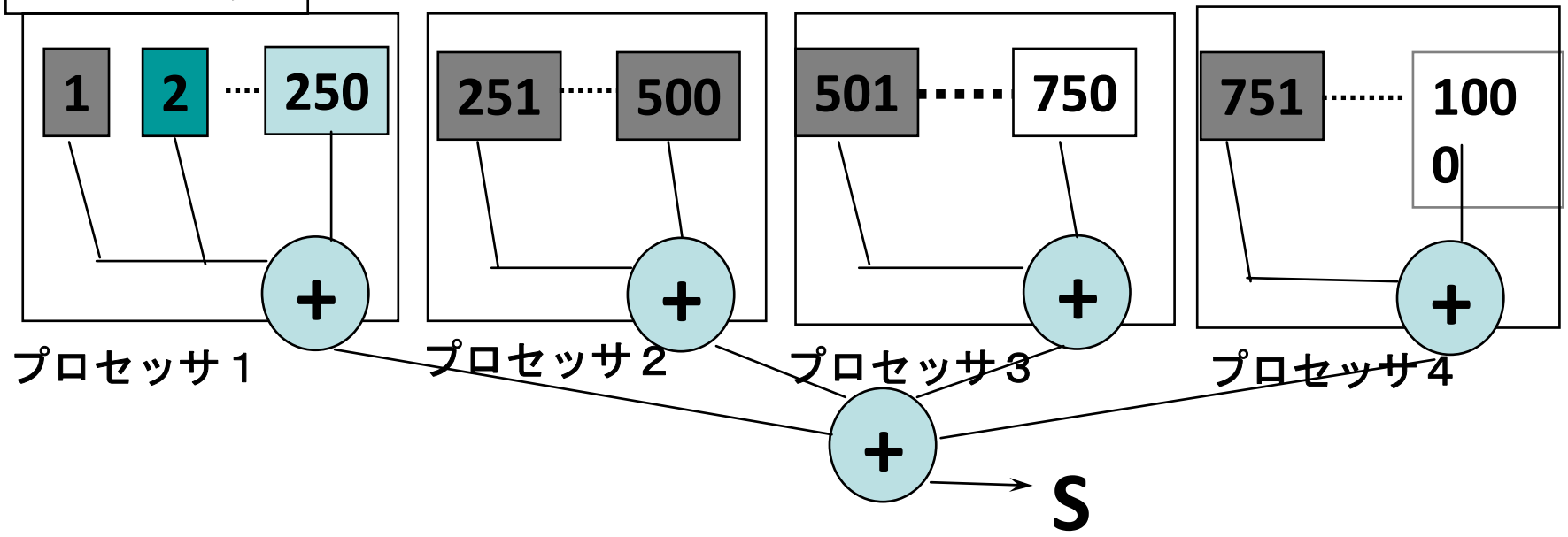
並列処理の簡単な例

for (i = 0; i < 1000; i++)
S += A[i]

逐次計算



並列計算



MPIでプログラミングしてみると

```
#include <mpi.h>

double A[1000 / N_PE];

int main(int argc, char *argv[])
{
   double sum, mysum;

   MPI_Init(&argc,&argv);
   mysum = 0.0;
   for (i = 0; i < 1000 / N_PE; i++){
            mysum += A[i];
   }
   MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
   MPI_Finalize();
   return (0);
}
```

解説

- まず、宣言されたデータは各プロセッサに重複して取られている。
  - なので、ひとつのプロセッサではプロセッサ数N_PEで割った分だけでいい
- 各プロセッサでは、mainからプログラムが実行される
- SPMD (single program/multiple data)
  - 大体、同じようなところを違うデータ（つまり、実行されているノードにあるデータ）に対して実行するようなプログラムのこと
- 初期化
  - MPI_Init

解説（続き）

- コミュニケータ
  - 通信のcontextと保持する仕組み
  - MPI_COMM_WORLDだけつかえば、当分の間十分
- 計算・通信
  - 各プロセッサで部分和を計算して、集計
  - コレクティブ通信

```
MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE,
        MPI_SUM, 0, MPI_COMM_WORLD);
```

- 最後にexitの前で、全プロセッサで！

```
MPI_Finalize();
```

コミュニケータ (Communicator)

- 通信領域（通信ドメイン）を指定
  - Opaqueオブジェクト
  - プロセスグループ
  - プロセストポロジ
- モジュール間でのメッセージの分離
- MPI_COMM_WORLD
  - 全体のプロセスを含むコミュニケータ
- MPI_COMM_SELF, MPI_COMM_NULL

コミュニケータに対する操作

- int MPI_Comm_size(MPI_Comm comm, int *size);
- コミュニケータcommのプロセスグループの総数をsizeに返す
- Int MPI_Comm_rank(MPI_Comm comm, int *rank);
- コミュニケータcommのプロセスグループにおける自プロセスのランク番号をrankに返す

コレクティブ通信

- コミュニケータのプロセスグループ全体が参加する通信
  - 総和（リダクション）
  - ブロードキャスト
  - scatter/gather
  - 全対全（転置）
  - バリア同期など

OpenMPとMPIのプログラム例：Cpi

- 積分して、円周率を求めるプログラム
- MPICHのテストプログラム

$$\pi = \int_0^1 \frac{4}{1+t^2}\, dt$$

- OpenMP版（cpi-seq.c)
  - ループを並列化するだけ, 1行のみ

- MPI版(cpi-mpi.c)
  - 変数nの値をBcast
  - 最後にreduction
  - 計算は、プロセッサごとに飛び飛びにやっている

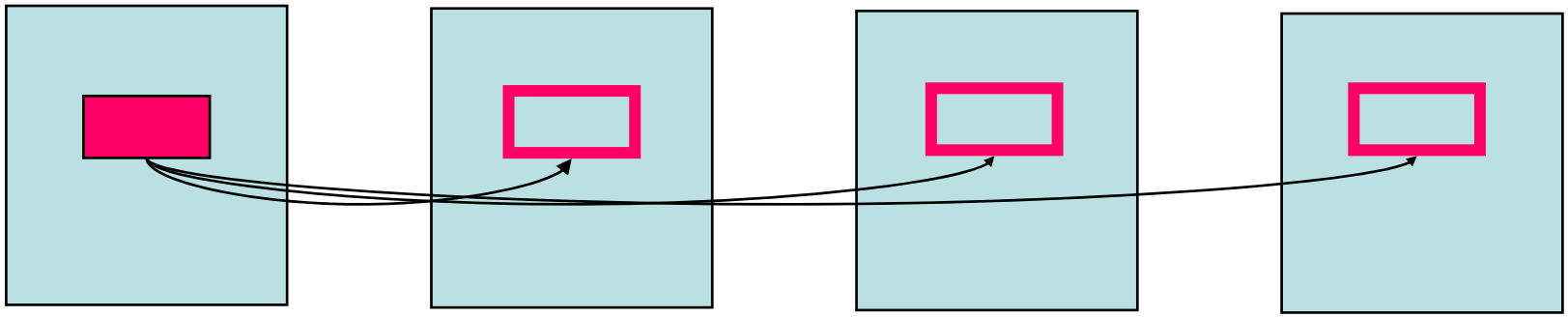
```
…
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

h = 1.0 / n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs){
        x = h * (i - 0.5);
        sum += f(x);
}
mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

集団通信: ブロードキャスト

```
MPI_Bcast(
  void      *data_buffer,   // ブロードキャスト用送受信バッファのアドレス
  int       count,          // ブロードキャストデータの個数
  MPI_Datatype data_type,   // ブロードキャストデータの型(*1)
  int       source,         // ブロードキャスト元プロセスのランク
  MPI_Comm  communicator    // 送受信を行うグループ
);
```

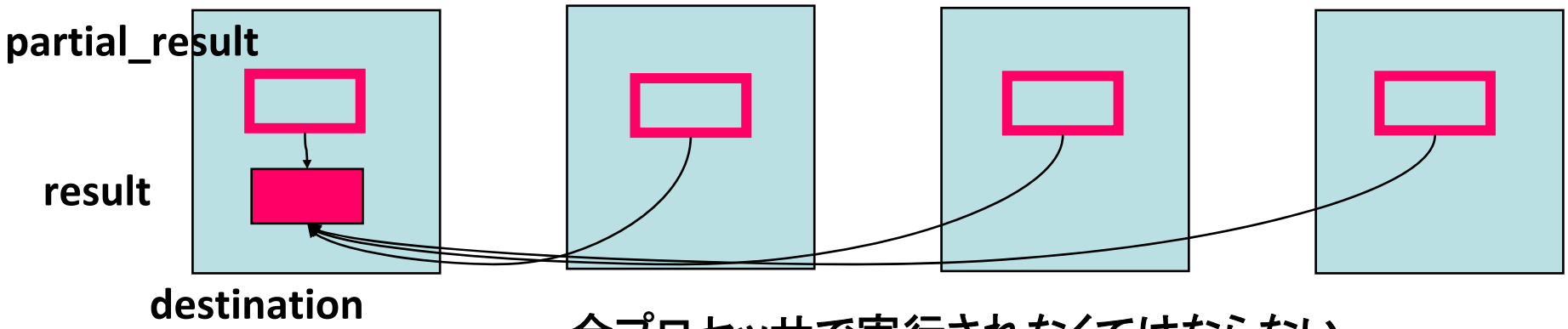


source

全プロセッサで実行されなくてはならない

集団通信: リダクション

```
MPI_Reduce(
  void      *partial_result, // 各ノードの処理結果が格納されているアドレス
  void      *result,         // 集計結果を格納するアドレス
  int       count,           // データの個数
  MPI_Datatype data_type,    // データの型(*1)
  MPI_Op    operator,        // リデュースオペレーションの指定(*2)
  int       destination,     // 集計結果を得るプロセス
  MPI_Comm  communicator     // 送受信を行うグループ
);
```

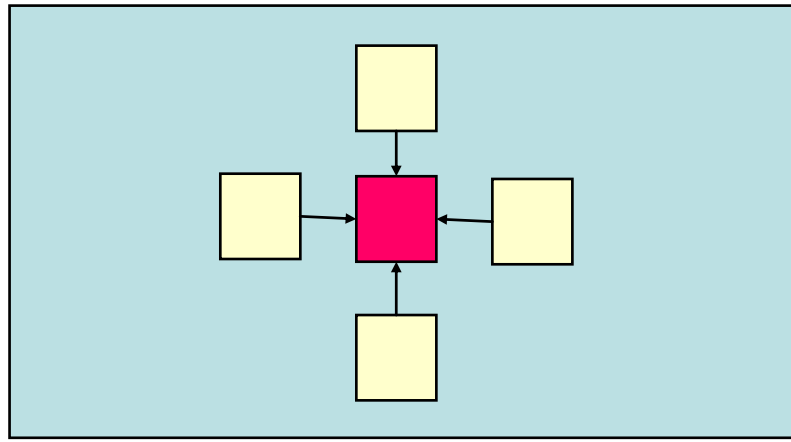


全プロセッサで実行されなくてはならない

Resultを全プロセッサで受け取る場合は、MPI_AllReduce

OpenMPとMPIのプログラム例：laplace

- Laplace方程式の陽的解法
  - 上下左右の4点の平均で、updateしていくプログラム
  - Oldとnewを用意して直前の値をコピー
  - 典型的な領域分割
  - 最後に残差をとる

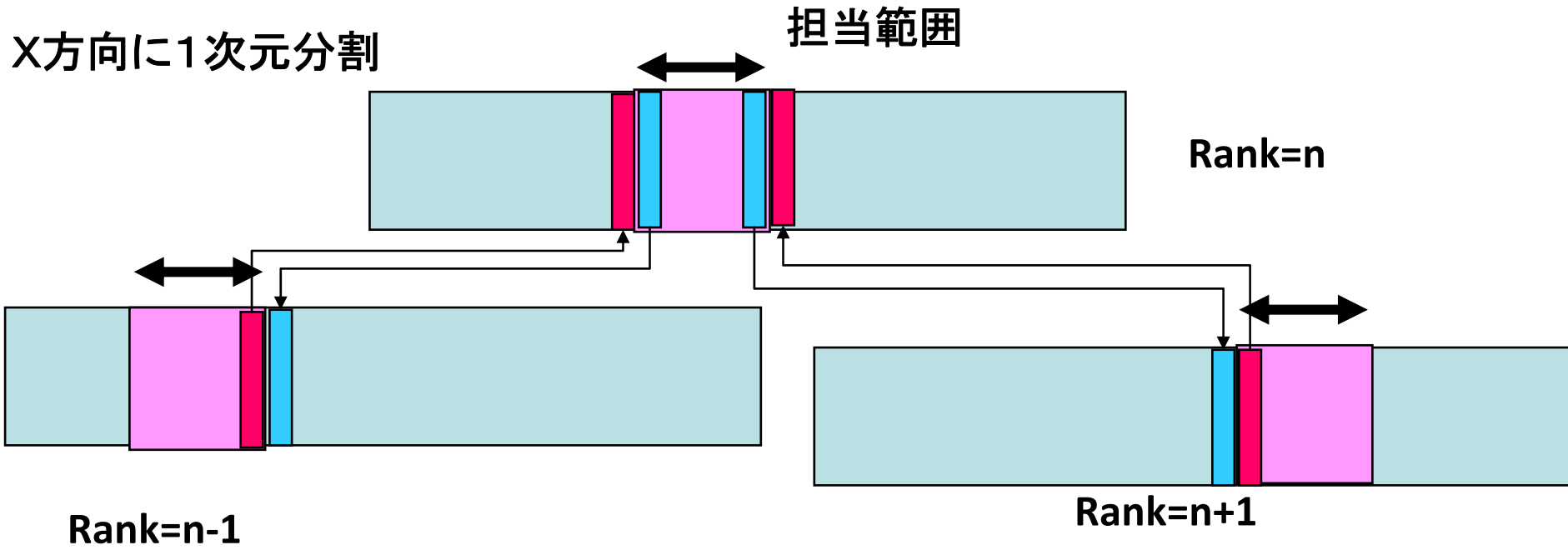


- OpenMP版 lap.c
  - 3つのループを外側で並列化
    - OpenMPは1次元のみ
  - Parallel指示文とfor指示文を離してつかってみた

- MPI版
  - 結構たいへん

隣接通信

- 隣の部分を繰り返しごとに通信しなくてはならない



- 非同期通信を使う方法
- 同期通信を使う方法
- Sendrecvを使う方法

メッセージ通信

- Send/Receive

```
MPI_Send(
  void          *send_data_buffer, //  送信データが格納されているメモリのアドレス
  int           count,             //  送信データの個数
  MPI_Datatype  data_type,         //  送信データの型(*1)
  int           destination,       //  送信先プロセスのランク
  int           tag,               //  送信データの識別を行うタグ
  MPI_Comm      communicator       //  送受信を行うグループ.
);
```

```
MPI_Recv(
  void          *recv_data_buffer, //  受信データが格納されるメモリのアドレス
  int           count,             //  受信データの個数
  MPI_Datatype  data_type,         //  受信データの型(*1)
  int           source,            //  送信元プロセスのランク
  int           tag,               //  受信データの識別を行うためのタグ.
  MPI_Comm      communicator,      //  送受信を行うグループ.
  MPI_Status    *status            //  受信に関する情報を格納する変数のアドレス
);
```

メッセージ通信

- メッセージはデータアドレスとサイズ
  - 型がある　MPI_INT，MPI_DOUBLE，…
  - Binaryの場合は、MPI_BYTEで、サイズにbyte数を指定
- Source/destinationは、プロセッサ番号（rank）とタグを指定
  - 送信元を指定しない場合はMPI_ANY_SOURCEを指定
  - 同じタグを持っているSendとRecvがマッチ
  - どのようなタグでもRecvしたい場合はMPI_ANY_TAGを指定
- Statusで，実際に受信したメッセージサイズ，タグ，送信元などが分かる
- 注意
  - これは、同期通信
  - つまり、recvが完了しないと、sendは完了しない。

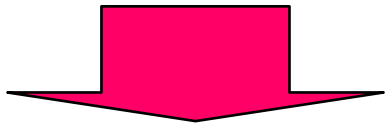
注：正確には、sendはバッファにあるデータを送りだした時点で終了する。しかし、recvされないと送りだしができないことがあるので、「相手がrecvしないとsendが終了しない」として理解したほうが安全。

非同期通信

- Send/recvを実行して、後で終了をチェックする通信方法
  - 通常のsend/recv（同期通信）では、オペレーションが終了するまで、終わらない

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype,
      int dest, int tag,   MPI_Comm comm, MPI_Request *request )
```

```
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
 int source, int tag, MPI_Comm comm, MPI_Request *request )
```



```
int MPI_Wait ( MPI_Request *request, MPI_Status *status)
```

プロセストポロジ

- int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart);
- ndims次元のハイパーキューブのトポロジをもつコミュニケータcomm_cartを作成
- dimsはそれぞれの次元のプロセス数
- periodsはそれぞれの次元が周期的かどうか
- reorderは新旧のコミュニケータでrankの順番を変更するかどうか

シフト通信の相手先

- int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest);
- directionはシフトする次元
  - ndims次元であれば0～ndims-1
- dispだけシフトしたとき，受け取り先がrank_source，送信先がrank_destに返る
- 周期的ではない場合，境界を超えるとMPI_PROC_NULLが返される

```
/* calculate process ranks for ‘down’ and ‘up’ */
MPI_Cart_shift(comm, 0, 1, &down, &up);

/* recv from down */
MPI_Irecv(&uu[x_start-1][1], YSIZE, MPI_DOUBLE, down, TAG_1,
          comm, &req1);
/* recv from up */
MPI_Irecv(&uu[x_end][1], YSIZE, MPI_DOUBLE, up, TAG_2,
          comm, &req2);

/* send to down */
MPI_Send(&u[x_start][1], YSIZE, MPI_DOUBLE, down, TAG_2, comm);
/* send to up */
MPI_Send(&u[x_end-1][1], YSIZE, MPI_DOUBLE, up, TAG_1, comm);

MPI_Wait(&req1, &status1);
MPI_Wait(&req2, &status2);
```

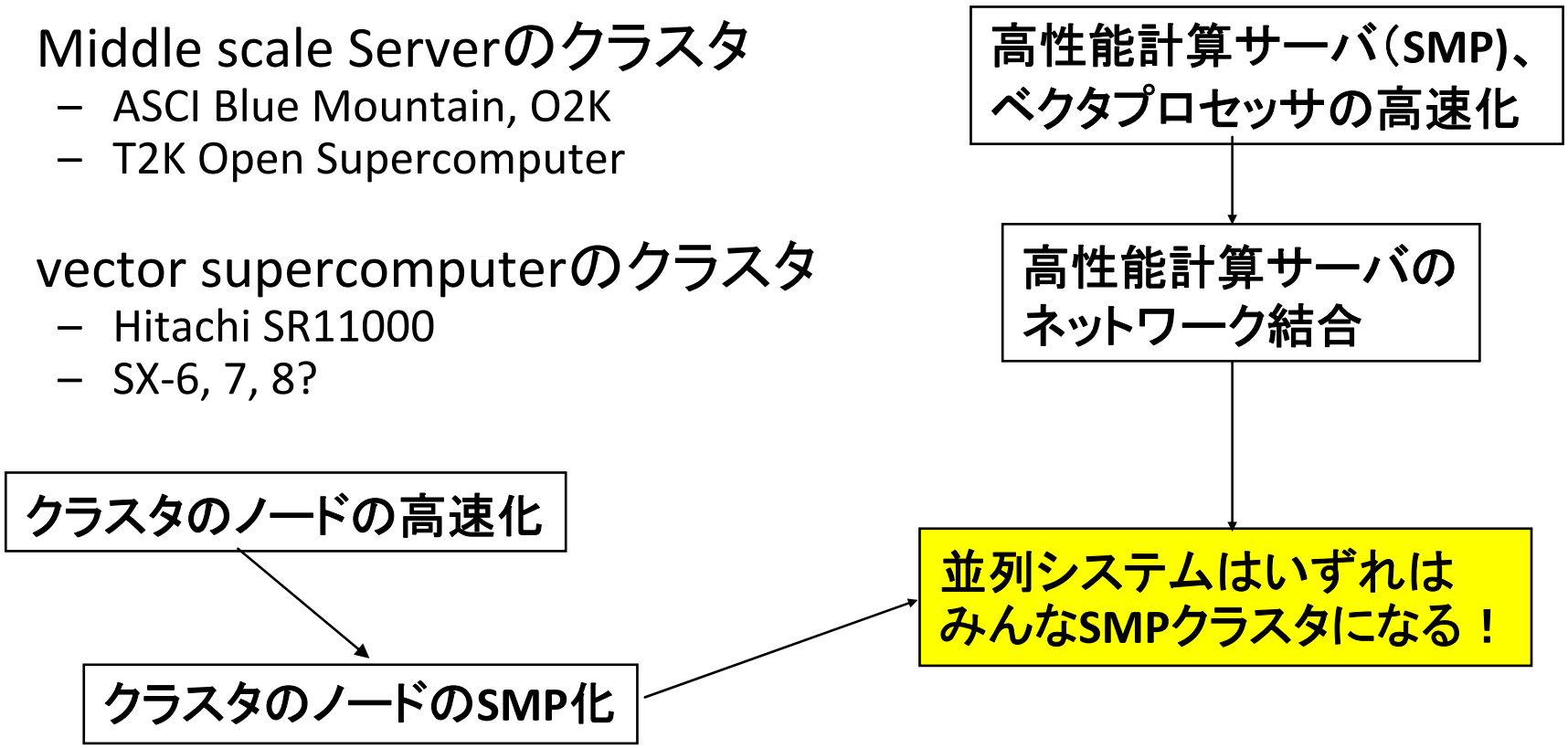
端（0とnumprocs-1)のプロセッサについてはMPI_PROC_NULLが指定され特別な処理は必要ない

改善すべき点

- 配列の一部しか使っていないので、使うところだけにする
  - 配列のindexの計算が面倒になる
  - 大規模計算では本質的な点
- １次元分割だけだが、２次元分割したほうが効率がよい
  - 通信量が減る
  - 多くのプロセッサが使える

SMPクラスタ

- PC-based SMPクラスタ
  - マルチコア
- Middle scale Serverのクラスタ
  - ASCI Blue Mountain, O2K
  - T2K Open Supercomputer
- vector supercomputerのクラスタ
  - Hitachi SR11000
  - SX-6, 7, 8?



MPIとOpenMPの混在プログラミング

- 分散メモリはMPIで、中のSMPはOpenMPで

- MPI+OpenMP
  - はじめにMPIのプログラムを作る
  - 並列にできるループを並列実行指示文を入れる
    - 並列部分はSMP上で並列に実行される。

- OpenMP+MPI
  - OpenMPによるマルチスレッドプログラム
  - single構文・master構文・critical構文内で、メッセージ通信を行う。
    - Thread-safeなMPIが必要
    - いくつかの点で、動作の定義が不明な点がある
      - マルチスレッド環境でのMPI
      - OpenMPのthreadprivate変数の定義？

- SMP内でデータを共用することができるときに効果がある。
  - 必ずしもそうならないことがある（メモリバス容量の問題？）

おわりに

- これからの高速化には、並列化は必須

- １６プロセッサぐらいでよければ、OpenMP
- それ以上になれば、MPIが必須
  - だだし、プログラミングのコストと実行時間のトレードオフか
  - 長期的には、MPIに変わるプログラミング言語が待たれる

- 科学技術計算の並列化はそれほど難しくない
  - 内在する並列性がある
  - 大体のパターンが決まっている
  - 並列プログラムの「デザインパターン」
  - 性能も…

おまけ – Open Source OpenMP

- GNU GCC 4.2以降

  % cc -fopenmp . . .
- Omni OpenMP Compiler
  - http://phase.hpcc.jp/Omni/
  - 佐藤（三）先生

Open Source MPI

- OpenMPI
  - http://www.open-mpi.org/
- MPICH2
  - http://www-unix.mcs.anl.gov/mpi/mpich2/
- YAMPII
  - http://www.il.is.s.u-tokyo.ac.jp/yampii/

コンパイル・実行の仕方

- コンパイル

  % mpicc … test.c …

  - MPI用のコンパイルコマンドがある
  - 手動で-lmpiをリンクすることもできる

- 実行

  % mpiexec –n #procs a.out …

  - a.outが#procsプロセスで実行される
  - 以前の処理系ではmpirunが利用され，de factoとなっているが，ポータブルではない

    % mpirun –np #procs a.out …

  - 実行されるプロセス群はマシン構成ファイルなどで指定する
  - あらかじめデーモンプロセスを立ち上げる必要があるものも

OpenMPIでのアプリケーションの実行

```
% cat hosts
host1 slots=4
host2 slots=4
host3 slots=4
host4 slots=4
% mpiexec –hostfile hosts –n 16 program
```