並列分散コンピューティング (7)OpenMP, MPI

大瀧保広

今日の内容

- 並列システムのプログラムを作成する場合、
  マルチスレッドやマルチプロセスのプログラミングが
  必要となるが、素で書くのは辛い。
- 実際に分散プログラミングを行う場合に利用可能なものを
  超簡単に紹介する。

  - OpenMP　→ 共有メモリ型が対象（スレッド並列）
  - MPI　→ 分散メモリ型が対象（プロセス並列）

今回の資料だけでは効果的に使えるようにはなりません。

OpenMPとは

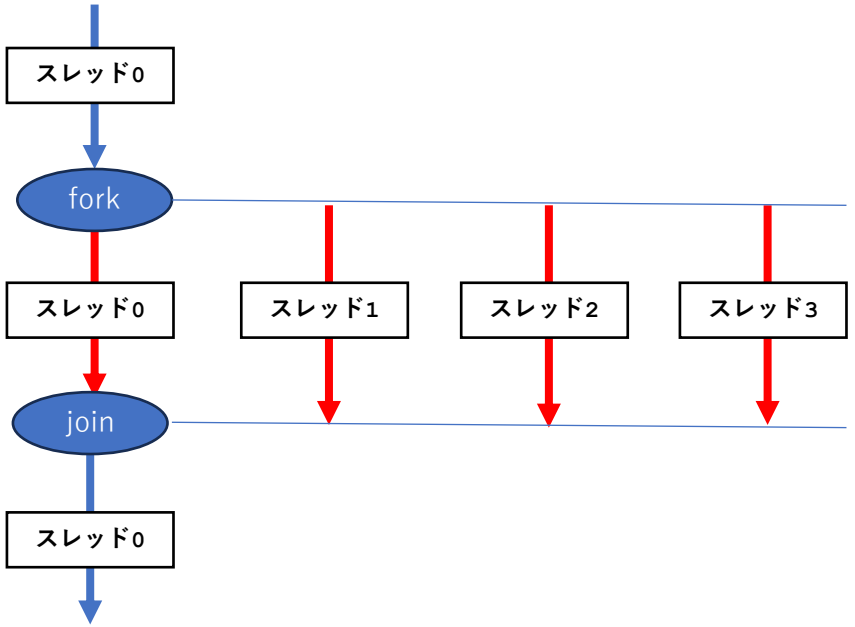
- OpenMP C and C++ Application Program Interface
- 共有メモリ型並列計算機用 にプログラムを並列化することを目的として、以下のものを規格化したもの。
  1. 指示文
  2. ライブラリ
  3. 環境変数

並列実行の指示はユーザが与えるのであって、コンパイラによる自動並列化ではない。

- https://www.sstc.co.jp/biz/projects/OpenMP.html

OpenMPによるスレッド並列

■Fork-Join モデル




```
...    非並列処理

#pragma omp parallel
{
    ... 並列リージョン
}

...    非並列処理
```

OpenMPコードの書き方とコンパイル

- ヘッダーファイル omp.h を include した上で、#pragma omp で始まる指示文を書く。
- コンパイル時に OpenMP用のオプション **-fopenmp** を付ける。
- オプションを指定しない場合は、指示文はコメントとして認識され、逐次実行の実行ファイルが生成される。
  - とはいえ、#pragmaを無視したという警告がでることがある。
  - 指示文の書き方によって、逐次実行と並列実行の実行結果が変わる点に注意

```
#include <omp.h>

処理A
#pragma omp parallel
{
    処理B
}
処理C
```

「Pragma（プラグマ）」とは
コンパイラに特定の情報を渡すために
使用するコンパイラ指令

サンプルプログラム（追加配布）

- 以前 配布した PDC.zip に含まれていないので、
  改めて以下のファイルを入手してください。

- WSL内で以下のコマンドを実行する。

```
cd
wget http://nenya.cis.ibaraki.ac.jp/PDC2.zip
unzip PDC2.zip
```

ホームディレクトリに PDC2 というフォルダができます。

サンプルプログラム：PDC2/OpenMP/basic.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

int main(void)
{
        printf("AAAAAAAA¥n");
        #pragma omp parallel
        {
                printf("BBBBBBBB¥n");
        }
        printf("CCCCCCCC¥n");

        return EXIT_SUCCESS;
}
```

```
$ gcc -o basic-omp basic.c -fopenmp
$ ./basic-omp
AAAAAAAA
BBBBBBBB
BBBBBBBB
BBBBBBBB
BBBBBBBB
CCCCCCCC

$ gcc -o basic-single basic.c
$ ./basic-single
AAAAAAAA
BBBBBBBB
CCCCCCCC
$
```

スレッドはいくつ生成する？

- #pragma omp parallel　で生成されるスレッドのデフォルト数は実行環境に依存する。
- 明示的にしてすることもできる。
  スレッド数の指定方法は2つ。

  - プログラム中で指定
    omp_set_num_threads(10);

  - シェルの環境変数で指定　←推奨

```
$ export OMP_NUM_THREADS=10
$ ./omp1-omp
```

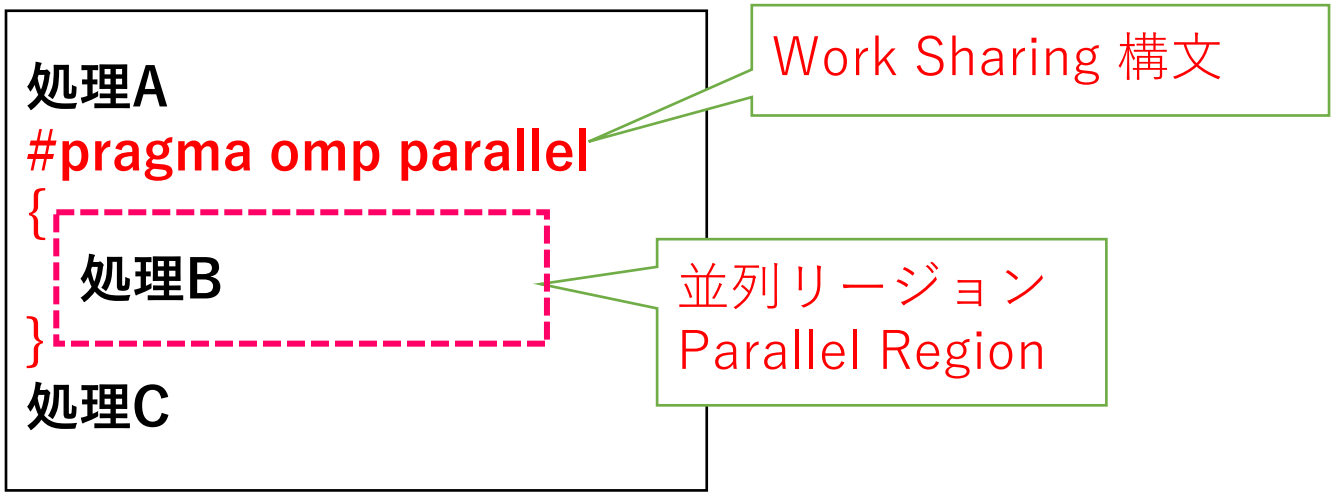
スレッドはいくつ生成する？

- 多くすればいいというものではない。
  使用可能なコア数とプログラムの性質に応じて設定する。

スレッド数 とコア数	**説明**
スレッド数 = コア数	一般的な選択
スレッド数 < コア数	一定のスレッド数を超えると性能が上がらない、 もしくは性能が低下する場合に選択
スレッド数 > コア数	計算よりもファイル I/Oや通信が主体で、 スレッド中でコアが遊んでしまうようなプログラムの 場合には増やしても大丈夫

OpenMP
よく使う構文の説明

OpenMPの用語



- 並列リージョンに書かれた処理が並列化の対象となる
- 「並列化がどのように行われるか」はWork Sharing構文次第

以下、代表的なものをいくつか紹介する

Work Sharing構文

以下の2種類がある。

1. parallel指示文と組み合わせるもの
   - ■parallel for 構文
   - ■parallel sections 構文

   など

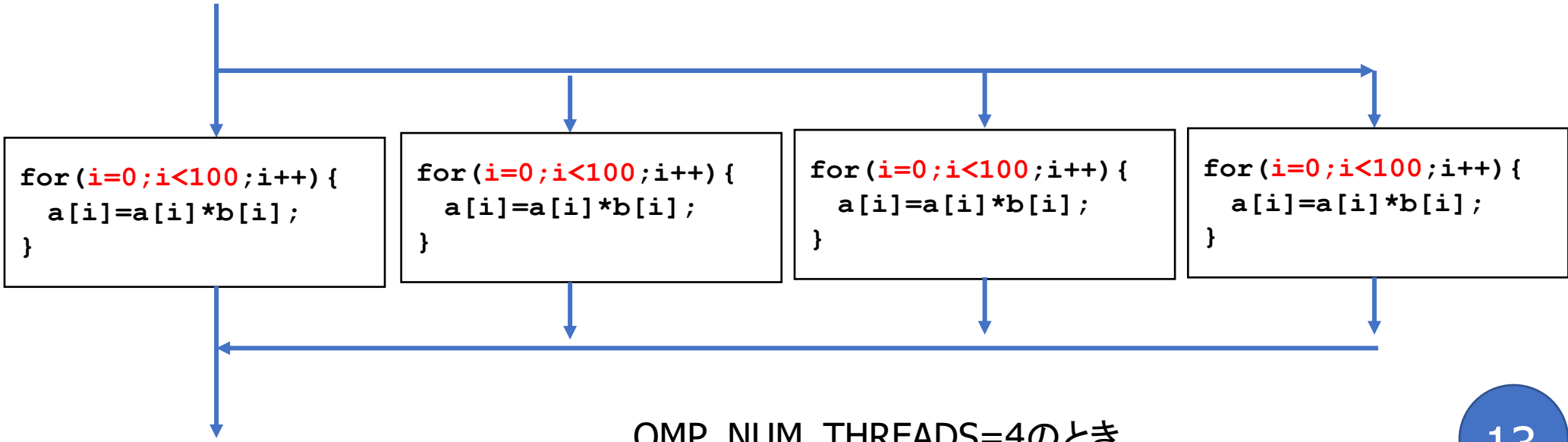
2. 並列リージョン**内**で記述するもの
   - ■section構文
   - ■single構文

   など

omp parallelだけだとこうなる

```
#pragma omp parallel
{
  for (i=0;i<100;i++) {
    a[i]=a[i]*b[i];
  }
}
```

【動きのイメージ】
#pragma omp parallel で
スレッドを生成しただけでは、
全てのスレッドが全ループを計算しようとする。

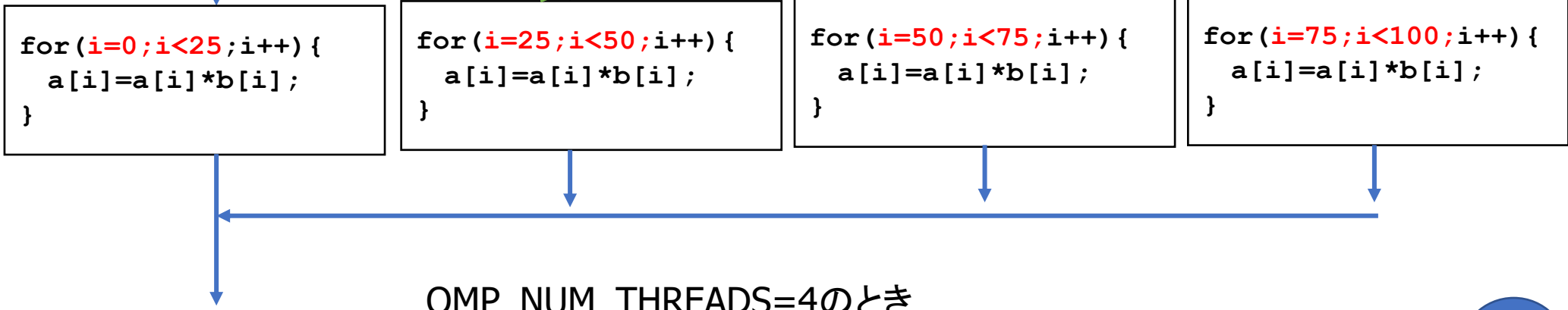


OMP_NUM_THREADS=4のとき

omp for構文

```
#pragma omp parallel
{
   #pragma omp for
    for (i=0;i<100;i++) {
       a[i]=a[i]*b[i];
    }
}
```

【動きのイメージ】
ワークシェアリング構文である
#pragma omp for は、ループを自動的に
スレッド数で均等に分割する



OMP_NUM_THREADS=4のとき

for構文

```
#pragma omp parallel for
{
    for (i=0;i<100;i++) {
       a[i]=a[i]*b[i];
    }
}
```

スレッド生成とループ並列を1行で記述することもできる。

ところで、変数 iは衝突しないの？
#pragma直下の for文の制御変数は、**Private変数**（スレッドごとに独立した変数）として確保される。

```
for(i=0;i<25;i++){
  a[i]=a[i]*b[i];
}
```

```
for(i=25;i<50;i++){
  a[i]=a[i]*b[i];
}
```

```
for(i=50;i<75;i++){
  a[i]=a[i]*b[i];
}
```

```
for(i=75;i<100;i++){
  a[i]=a[i]*b[i];
}
```

OMP_NUM_THREADS=4のとき

サンプル：for.c

- omp for のプログラム例

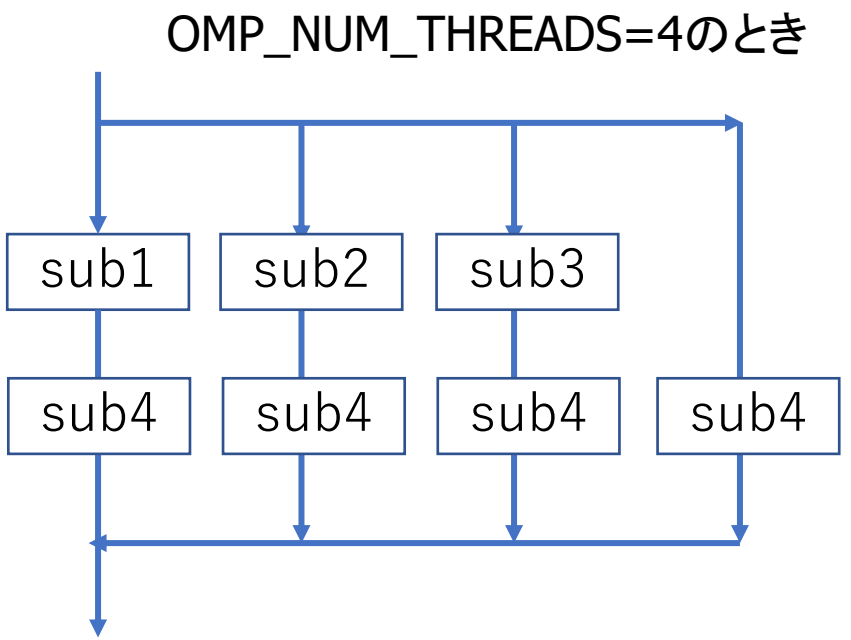
- スレッド数3で実行し、どのように並列化されているか確認せよ。

- forループの直前に以下のpragmaを追加し、動きがどのように変わったか確認せよ。

#pragma omp for

Parallel sections構文 と section構文

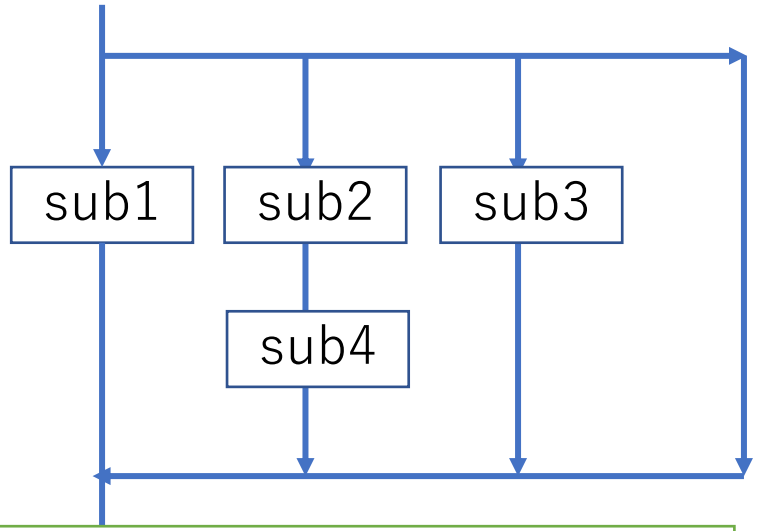
```
#pragma omp parallel
{
  #pragma omp sections
  {
        #pragma omp section
         {sub(1);}
        #pragma omp section
         {sub(2); }
        #pragma omp section
        {sub(3);}
  }
  {sub(4);}
}
```



single構文

```
#pragma omp parallel
{
  #pragma omp sections
  {
        #pragma omp section
         {sub(1);}
        #pragma omp section
         {sub(2); }
        #pragma omp section
        {sub(3);}
  }
  #pragma omp single
   {sub(4);}
}
```

OMP_NUM_THREADS=4のとき

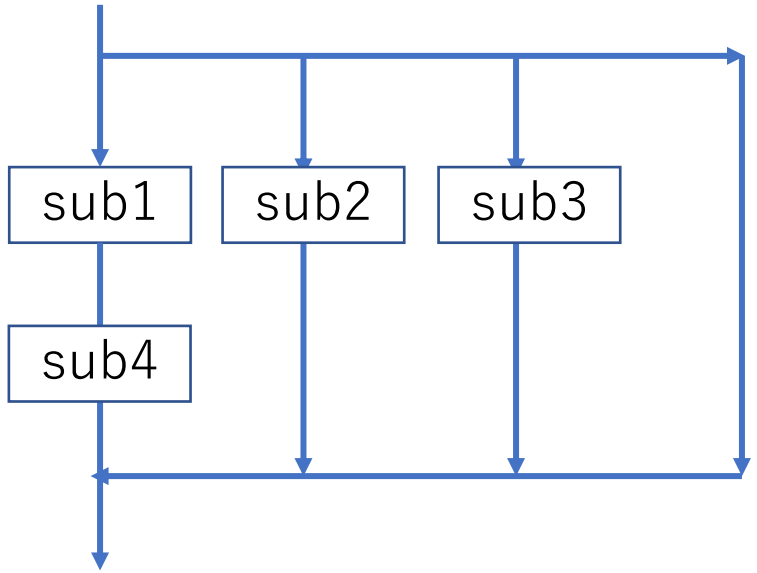


データ出力など、全スレッドで行う必要がない処理はsingleを指定すると、{}内がどこか1つのスレッドで実行される。
ただし、どのスレッドで実行されるかはわからない。

master構文

```
#pragma omp parallel
{
  #pragma omp sections
  {
       #pragma omp section
        {sub(1);}
       #pragma omp section
        {sub(2); }
       #pragma omp section
       {sub(3);}
  }
  #pragma omp master
   {sub(4);}
}
```

OMP_NUM_THREADS=4のとき



single構文と同じだが、
マスタースレッドで実行する。

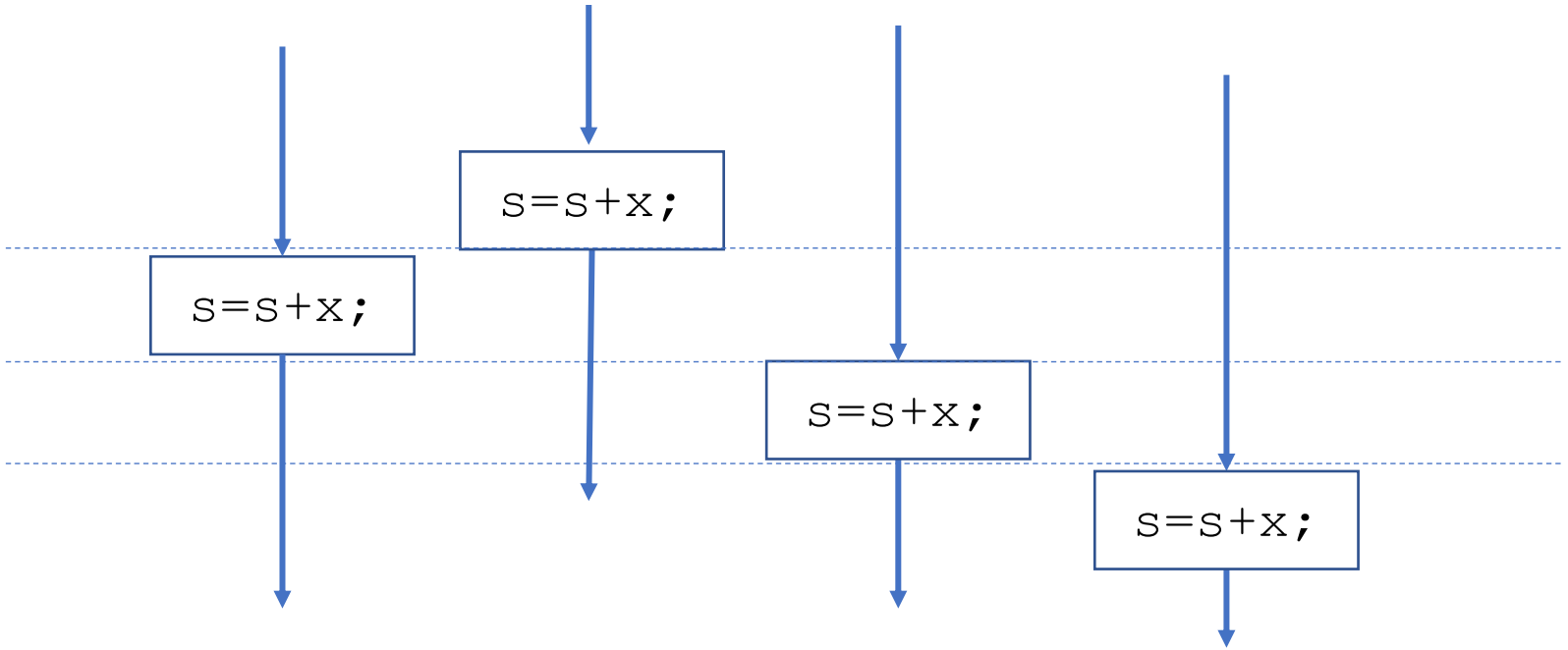
サンプル：section.c

- Parallel sections を使ったプログラム例
- 実行し、どの関数がどのスレッドで実行されているか、確認せよ。
- 最後のsub(4)の直前に 以下の行を追加し、実行せよ。
  #pragma omp single
- singleをmaster に変更して動きの違いを観察せよ。
  #pragma omp master

Critical補助構文

```
#pragma omp critical
{
        s=s+x;
}
```

クリティカルセクションの明示
この部分が排他的に実行される。



サンプル：critical.c

- 共有変数 s に1を足すプログラム。

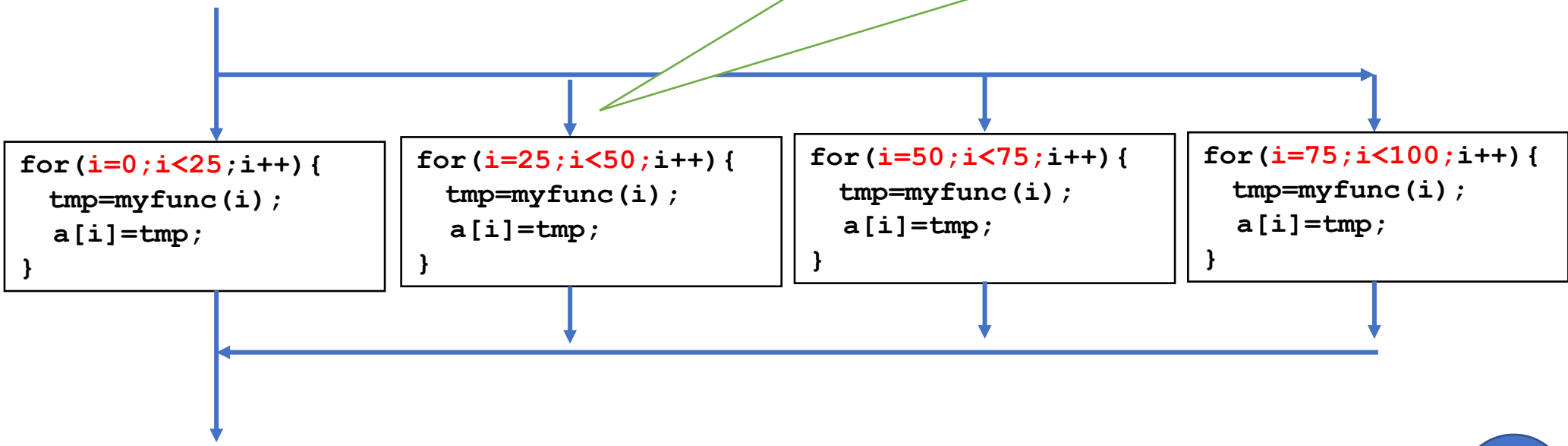
- 初期状態では critical補助構文は記述されていない。
  - このままコンパイル→実行してみよ。

  - Sに1を足す処理に対してcritical補助構文を適用し、実行してみよ。

Private補助構文（を使わないと）

```
#pragma omp for
for (i=0;i<100;i++) {
  tmp=myfunc(i);
  a[i]=tmp;
}
```

変数 iはプライベート変数になるが、
変数tmpはプライベート変数にならない。
各スレッドが同じtmpを上書きしてしまい、
正しく動かない

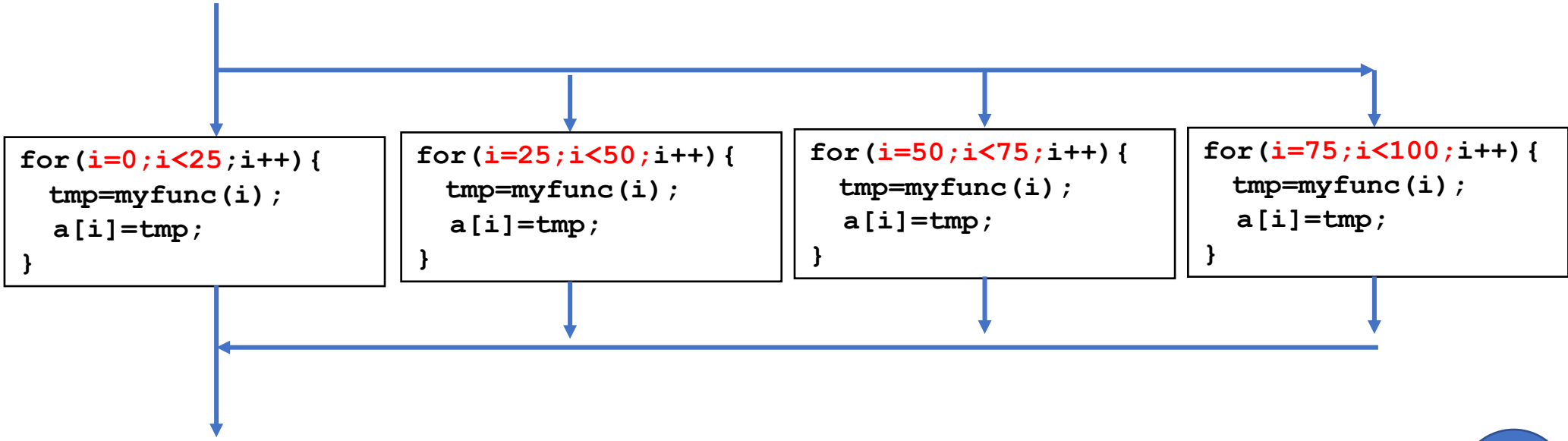


OMP_NUM_THREADS=4のとき

Private補助構文

```
#pragma omp for private(tmp)
for (i=0;i<100;i++) {
  tmp=myfunc(i);
  a[i]=tmp;
}
```

privateで明示的にプライベート変数にすることで、各スレッドごとに異なるtmpの領域が確保される。



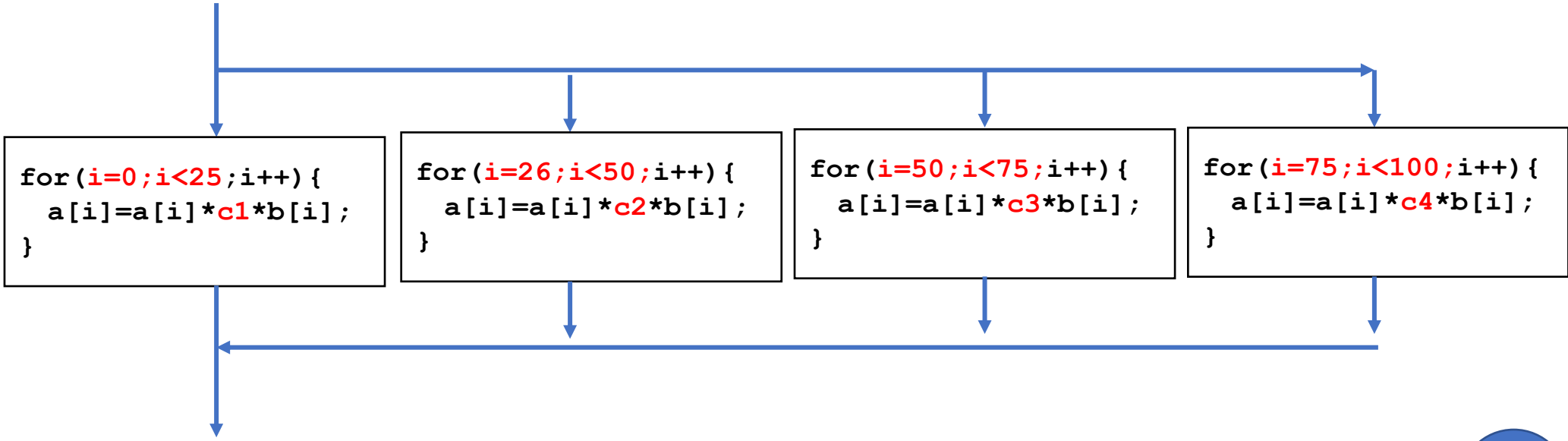
OMP_NUM_THREADS=4のとき

Private補助構文

変数は複数個を並べることができる

```
#pragma omp parallel for private(c)
for (i=0;i<100;i++) {
  a[i]=a[i]*c*b[i];
}
```

変数cが各スレッドでローカルな
変数領域を確保して実行される。
↓
高速化される



OMP_NUM_THREADS=4のとき

Private補助構文の重要性

```
#pragma omp parallel for
for (i=0;i<100;i++){
 for (j=0;j<100;j++){
   a[i]=a[i]*b[j];
 }
}
```

private(j) がない場合
変数 j は指示文直下の for 文の制御変数ではないため、共有変数 である。
各スレッドで同じ j をインクリメントする。
↓
正しく動かない。
（逐次実行と結果が異なる）

```
#pragma omp parallel for private(j)
for (i=0;i<100;i++){
 for (j=0;j<100;j++){
   a[i]=a[i]*b[j];
 }
}
```

private(j) がある場合
変数 j は各スレッドごとに独立した変数となる。
↓
逐次と同じ結果が得られる。

サンプル：private.c

- parallel sectionsの中の2つのsectionは並行に実行されるが所望の動きをしていない。
- printf文をいれて変数iの動きを確認せよ。
- section のところで変数 i をprivate指定して動きを確認せよ。

```
int i;
int a[BUFSIZE], b[BUFSIZE];

#pragma omp parallel sections
{
  #pragma omp section
  for (i=0; i<BUFSIZE; i++){
      a[i] = i;
  }
  #pragma omp section
  for (i=0; i<BUFSIZE; i++){
      b[i] = i;
  }
}

#pragma omp parallel for
for (i=0; i<BUFSIZE; i++){
    a[i]=a[i] - b[i];
}

for (i=0; i<BUFSIZE; i++){
   if ( a[i]!=0)
       printf("a[%d]=%d b[%d]=%d¥n",i, a[i],i, b[i]);
}
```

Reduction補助構文

```
d=0;
#pragma omp parallel for reduction(+:d)
for (i=0;i<100;i++){
  d=d+a[i]*b[i];
}
```

各スレッドで結果を足し込み一つの結果を得る　場合などに使用する。

リダクションが可能な演算にのみ使用可能。変数はスカラー型のみ。

イメージとしては、
各スレッドごとにd1, d2, d3, ... を並列に求め、
d= d1+ d2 + d3 + ... の計算を
リダクションを用いて計算している。

サンプル：reduction.c

- コンパイル→実行してみよ
- parallel for 構文の後ろに reduction(+:d) を追加してみよ。

```
int i;
int d=0;
int a[BUFSIZE], b[BUFSIZE];

for (i=0; i<BUFSIZE; i++){ a[i] = i;
for (i=0; i<BUFSIZE; i++){ b[i] = 2*i


#pragma omp parallel for
for (i=0; i<BUFSIZE; i++){
      d =d + a[i]*b[i];
}

printf("Final d=%d¥n",d);
```

よく使われるOpenMP関連の関数

■最大スレッド数を設定する

```
int nthread=10;
omp_set_num_threads( nthread );
```

■最大スレッド数を取得する

```
int nthread;
nthread= omp_get_num_threads();
```

■スレッドIDを取得する

```
int myid;
myid= get_num_thread_num ();
```

OpenMPの はまりどころ

変数がPrivateになるかSharedになるか

変数が並列化でどのように扱われるかを理解していないと
並列化後の処理が正しい結果にならない。

- Parallels構文によって一番外側のループ制御変数のみが
  自動的にPrivate(スレッドローカル)変数になる。
- 並列領域内で明示的に宣言される変数 は Privateになる。
- 並列領域内でPrivate宣言せずに利用するそれ以外の変数は、
  すべて共有変数(shared variable)になる。
- それ以外のグルーバル変数はすべて共有変数となる。

- ループ内で呼ばれる関数内で宣言される変数はPrivate..
  そもそも呼び出された関数の中のローカル変数なので。

その上で、さらに共有変数の罠

- OpenMPの共有変数は**データの一貫性を保証しない**。
  - 各スレッドで同じグローバル変数を参照していると思っていても、実際にはレジスタ上のキャッシュを参照していることがあるため、値がスレッド毎に違うことがある。

- 一貫性が必要なグローバル変数を書き換えたときには、Flush構文などで
  レジスタ→物理メモリ
  の同期処理をかけないと危ない。

念押し：並列化の責任はユーザにある

- どのループを並列化するか(そもそも並列化可能か) という判断はすべてプログラマが行う
- データ依存性などの関係で、本来 並列化できないループであっても、指示文を書けば 無理やり並列化は行われる。
  しかし当然、正しい結果は得られない。

- 必要なPrivate宣言を忘れると簡単にバグる。
  （逐次実行と結果が変わってしまう）
  - private変数の宣言を書き忘れても、コンパイラはエラーを出さない。

最後に：OpenMPは万能ではない

- OpenMPを用いた並列化は、 parallel構文を用いた単純なforループ並列化 が主となる。
  ほどほどの高速化を手軽に行いたい場面では有用。

- 複雑な並列化はかえってプログラミングコストがかかるので、OpenMPを利用する利点が失われる。

↓

複雑なスレッドプログラミングには向かない。

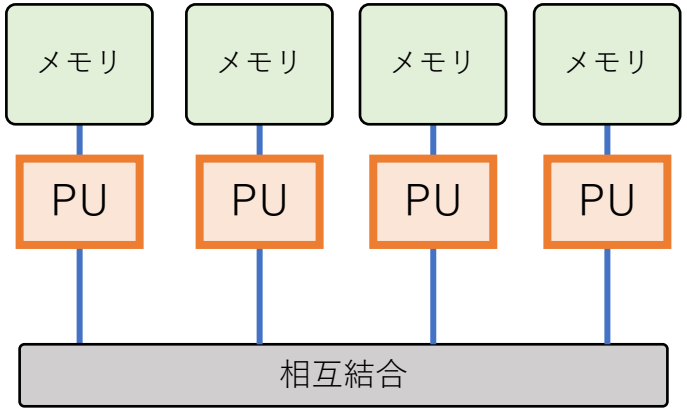
究極まで高速化を求めるような場面にも向かない。

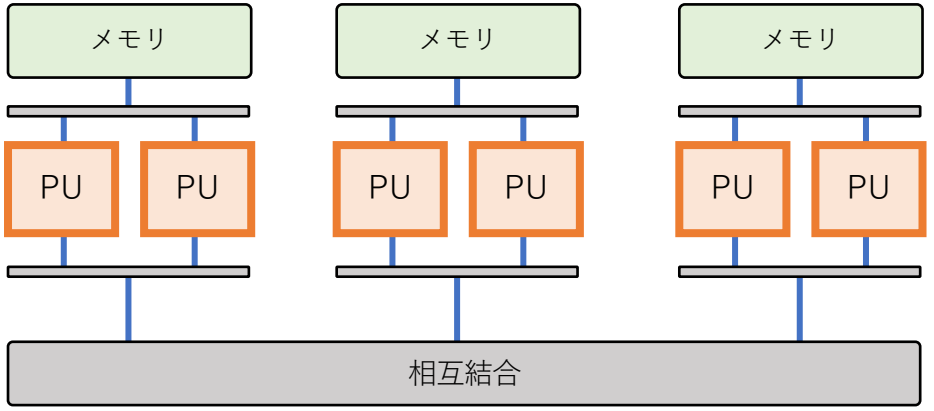
MPI
Message Passing Interface

分散メモリ環境のため並列化ミドルウェア（のひとつ）

分散メモリシステムにおける並列化

- 分散メモリシステムにおける並列化は大きく 2 つに分類される。
  - メッセージパッシング方式　← MPIはこっち
  - データ並列方式





分散メモリシステムにおける並列化

- **メッセージパッシング方式**
  - プロセッサ間でメッセージの「通信」をしながら計算を行う
  - 送信側と受信側で、以下の2つの処理が対になるように適切に実装されることが重要。
    - processor i ： processor j に送信する
    - processor j ： processor i から受信する
- 通信処理を書くのは大変

「プログラム」から見た「通信」

一般的なTCP/IP通信プログラムの場合

- **接続手順**
  - socket, bind, listen, connect, accept, ...
- **通信相手の指定**
  IPアドレス（ホスト名）, ポート番号, ...
- **通信内容**
  - バイト列
- **その他**
  - 基本的に一対一通信（IPによるユニキャスト）
  - IPのマルチキャストやブロードキャストは
    アプリとしては使いにくい。

MPIは何をしてくれるのか

直感的に並列プログラムを記述できるように、通信を抽象化する

- 接続手順
  - MPI_Init を呼ぶだけ（ただし事前にいろいろ仕込みは必要）
- 通信相手の指定
  - MPI_Comm_rank()で得られる識別番号（ランク）で指定
- 通信内容
  - 「データ型」単位
  - 新たな「データ型」も定義可能
- その他
  - 一対一, 一対多, 多対多　　などの通信をサポート

MPI (Message Passing Interface)とは

- MPIは通信を行う部分の規格
  - 厳密には言語、ライブラリなどのことではない。
  - 各種言語（C, C++, Fortran）で利用可能
  - OpenMPと組み合わせて利用することが多い

- MPIの実装
  MPI規格に基づく実装は各種ある。
  フリーで利用できるもので有名どころは以下の2つ。

  - MPICH
    https://www.mcs.anl.gov/research/projects/mpi/

  - Open MPI
    https://www.open-mpi.org/about/members/

MPICHを使うための準備

Ubuntu上では追加のパッケージのインストールが必要です。

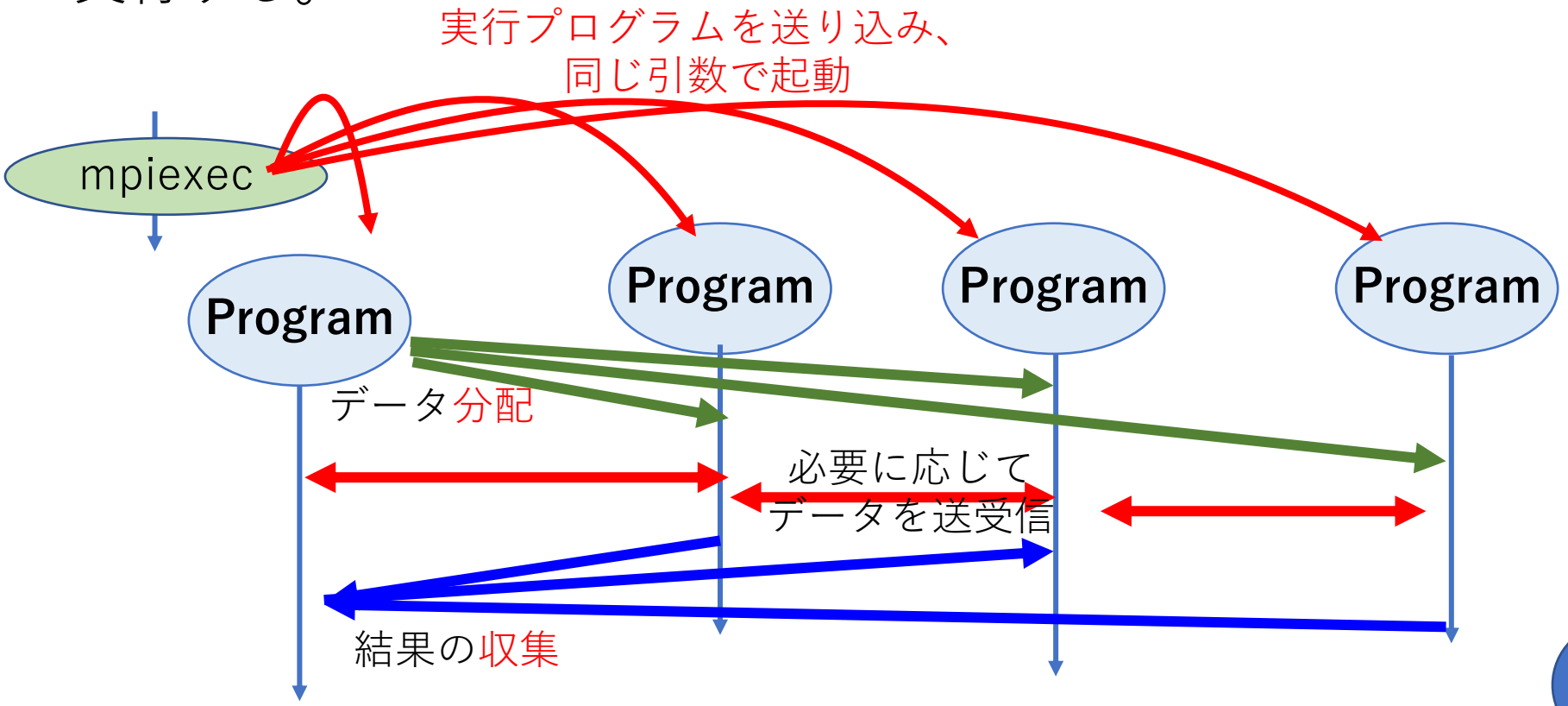
% **sudo apt install openssh-server**

% **sudo apt install mpich**

MPIプログラミングの考え方

■基本的にはSingle Program, Multi Data (**SPMD**)

「同じプログラム（処理）」を「異なるデータ」に対して実行する。



プログラムの構造

forkシステムコールによる
子プロセスの生成と同じ考え方

■実行するプロセスによって処理を変えるには、
同じプログラムの異なるコード部分を実行させる。
そのために,
　プログラム内部でプロセスを識別して処理を分ける
ように記述する。

本質的にはプロセスIDなのだが、
システムのプロセスIDは使えないことに注意！

■プロセスの識別はランク（rank）を使用する。
　■ルートプロセスはランク0
　■それ以外は１から順に割り振られる

サンプルプログラム：mpi-hello.c

```
#include <mpi.h>
#define MAXSTR 100
int main(int argc, char **argv )
{
     char message[MAXSTR];
     int myrank;
     MPI_Status status;

     MPI_Init( &argc, &argv );
     MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

     if (myrank == 0) {
          strcpy(message,"Hello, there");
          MPI_Send(message, strlen(message)+1,
                   MPI_CHAR, 1, 99, MPI_COMM_WORLD);
          printf("rank%d sent: message¥n", myrank, message);
     } else {
          MPI_Recv(message, MAXSTR,
                   MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
          printf("rank%d received: %s¥n", myrank, message);
     }
     MPI_Finalize();
}
```

MPI関連のヘッダファイル

Rank 0側と Rank 1側の処理を書き、
myrankをみて処理を振り分けるように書く

コンパイルと実行

- コンパイル
  - MPIが提供する専用のコンパイルコマンドを利用する

```
mpicc -o mpihello mpi-hello.c
```

- 実行
  - MPIが提供する専用の実行コマンドを利用
  - でき上がった実行ファイルをそのまま実行するのではない。

```
mpiexec -np 2 ./mpihello
```

プログラム制御に関するMPI関数

■ **int MPI_Init(int *argc, char ***argv)**

MPIの実行環境の初期化を行う。

- argc コマンド行の引数の数
- argv コマンド行の引数

■ **int MPI_Comm_size(MPI_Comm comm, int *size)**

通信を行うグループのサイズを決める。

- comm 通信を行うグループの指定
- size グループ内のタスクの数を受け取る

■ **int MPI_Comm_rank(MPI_Comm comm,int *rank)**

通信を行うグループのプロセスにタスク番号を与える。

- comm 通信を行うグループの指定。
- rank commの中でのタスク番号を受け取る(0,1,2,..)

■ **int MP_Finalize(void)**

MPIの実行環境を終了する。

通信に関するMPI関数

Blocking型の送信関数

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

- buf 送信データバッファ
- count 送信データの個数
- datatype データタイプ
- dest メッセージの送信先を指定
- tag メッセージタグ
- comm 通信を行うグループの指定

通信に関するMPI関数

Blocking型の受信関数

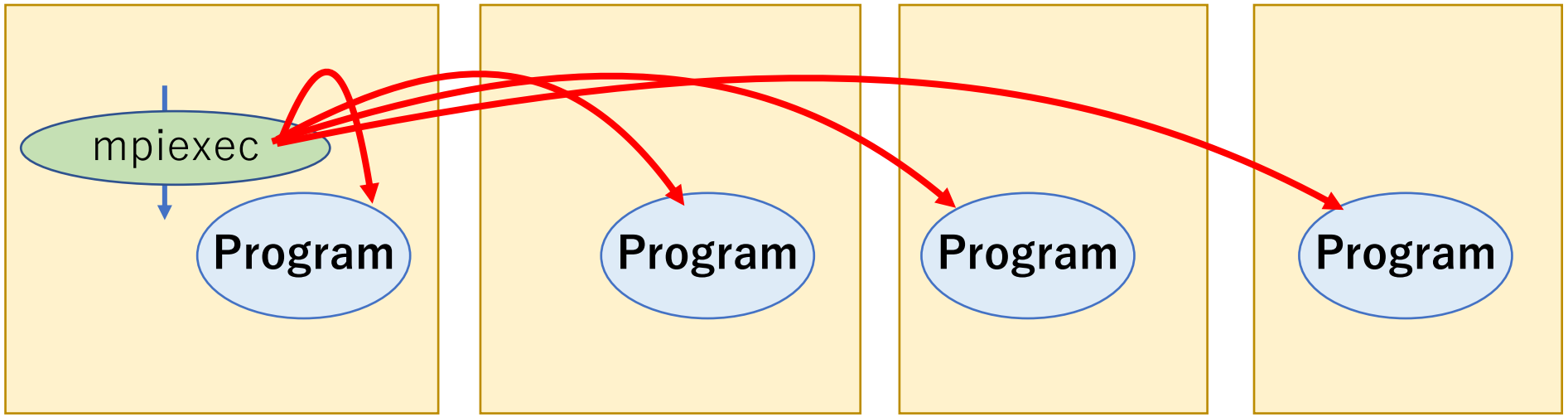
```
int MPI_Recv(void *buf,
        int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm,
        MPI_Status *status)
```

- buf 受信データバッファ
- count 受信データの個数
- datatype データタイプ
- source 送信元のタスク番号を指定。MPI_ANY_SOURCEで任意の送信元
- tag メッセージタグ。MPI_ANY_TAGで任意のタグを指定
- comm 通信を行うグループの指定
- status MPI_Status構造体で受信状況を返す。
  送信元、タグ、メッセージの大きさなど

MPIのデータ型

MPI DataType	C言語における型
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsinged short
MPI_UNSIGNED	unsinged int
MPI_UNSIGNED_LONG	unsinged long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	対応する型はない
MPI_PACKED	対応する型はない

MPIは事前の仕込みが大変



複数の計算機を使うための仕込み

MPIが本領を発揮するのは複数の計算機上で処理を行う状況。

- 適当なテキストファイルに、プロセスを起動したい計算機のリストを作成し、mpiexecの –h オプションで渡す。

```
mpiexec –np 3 -f  hostfile   ./mpi-prog
```

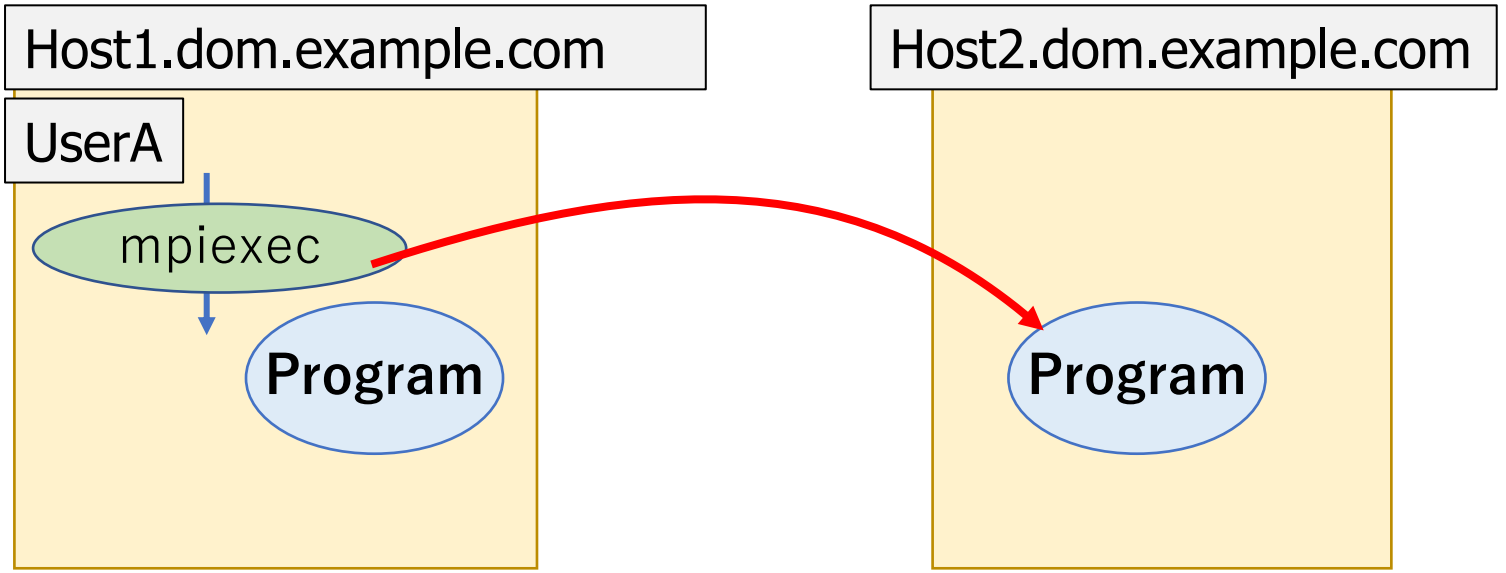
```
host1.dom.example.com
host2.dom.example.com
...
```

- 同一の実行環境の整備
  - mpiexecは各計算機に同じプログラムを送り込んで起動する。
  - 作成されたプログラムが実行できる同一の環境をすべての計算機で整えておく必要がある。(100台あったら100台に同じ設定をする)

sshによるログイン許可（公開鍵認証方式で）

mpiexecは各ホストへのプログラムの送り込み＆起動にssh コマンドを使う。

- プログラムを実行する同一ユーザのアカウントが必要
- ssh接続時にパスワード認証で一時停止するのは困る。
  →公開鍵認証による認証許可設定が必須。



公開鍵認証認証でSSH接続するには

1.SSHキーペアの生成:

ssh-keygen -t rsa

保護用のパスフレーズを聞かれるが enterを打つだけOK

■**公開鍵のコピー**:

ssh-copy-id　username@hostname

ここで usernameとhostnameは

コピー先のユーザ名とホスト名。

例えば： ssh-copy-id yohtaki@localhost

コピー時に、コピー先のパスワードの入力を求められる。

■ssh localhost ls

でパスワードを聞かれずに ls コマンドが実行されればOK。

今日のまとめ

- 並列システムのプログラム作成や実行を少しでも楽にするために様々なツールが提供されている。
- プログラマの指示で手っ取り早く、マルチスレッドによる並列化を行う方法としてOpenMP がある。
- 複数の計算機を用いて同一の処理を異なるデータに対して行うタイプの処理（SPMD）の、実装上の煩雑さを解消する枠組みとしてMPIがある。
  - データの通信まわりの抽象化と
    プログラムの送り込み＋起動を簡便にする。