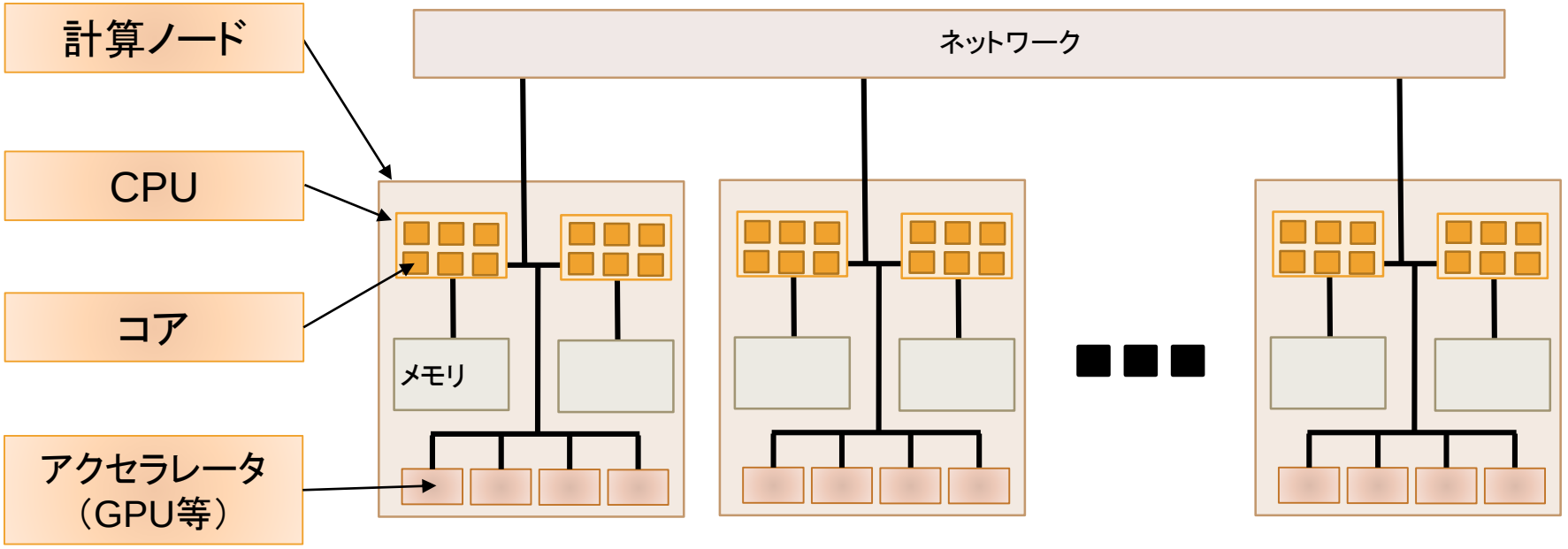
並列プログラミング超入門講習会 MPIコース

九州大学情報基盤研究開発センター

並列計算機の構成

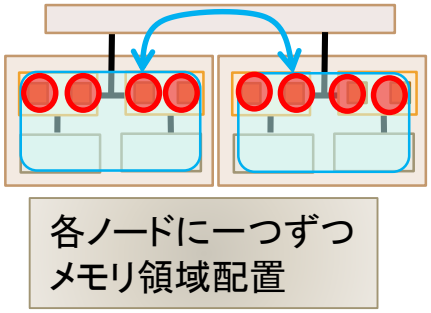


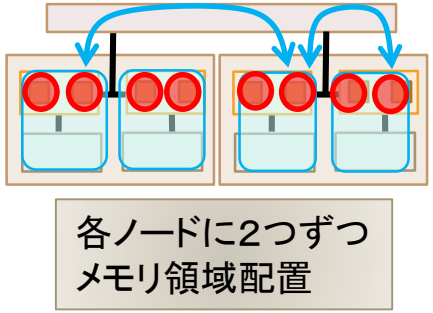
例：スーパーコンピュータシステム ITO サブシステム B

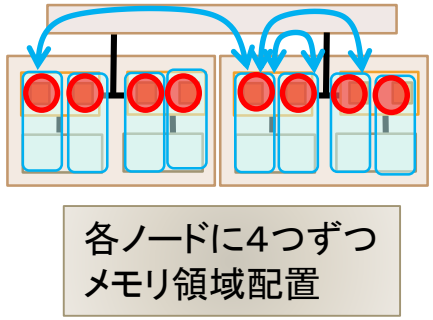
ノード数	CPU数 / ノード	コア数 / CPU	GPU数 / ノード
128	2	18	4

MPI (Message Passing Interface) による並列計算

- 主に複数の計算ノードによるクラスタ型計算機向け
  - 複数のメモリ領域の間で「通信」をしながら計算
  - ノード内をさらに複数のメモリ領域に分けることも可能
    - ノード内で通信が必要となるが, その方が高速な場合がある







- OpenMPと組み合わせて利用することが多い
  - OpenMP:一つのメモリ領域内で並列計算

MPIコースの内容

- 第一部：
  - MPIプログラムの基本構成
  - コンパイルと実行
  - MPIの初期化と終了
  - 実習 1
- 第二部：
  - 計算とデータの分割
  - 実習 2
- 第三部：
  - 通信の記述
  - 実習 3
- 第四部：
  - より高度な MPIプログラミングに向けて
  - 実習 4

プログラム中の通信

- インターネットプロトコル（TCP, UDP）の場合
  - 接続：
    - socket, bind, listen, connect, accept, …
  - ホストの識別：
    - ホスト名, IPアドレス, ポート番号, …
  - 転送
    - バイト単位
    - 基本的に一対一通信のみ

```
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
memset(&echoServAddr, 0, sizeof(echoServAddr));
echoServAddr.sin_family = AF_INET;
echoServAddr.sin_addr.s_addr = inet_addr(servIP);
echoServAddr.sin_port = htons(echoServPort);
connect(sock, (struct sockaddr *) &echoServAddr,
        sizeof(echoServAddr));
echoStringLen = strlen(echoString);
send(sock, echoString, echoStringLen, 0);

totalBytesRcvd = 0;
printf("Received: ");
while (totalBytesRcvd < echoStringLen){
  bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0);
  totalBytesRcvd += bytesRcvd;
  echoBuffer[bytesRcvd] = '¥0' ;
  printf(echoBuffer);
}
printf("¥n");
close(sock);
```

TCPによるサーバプログラム例

```
servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
memset(&echoServAddr, 0, sizeof(echoServAddr));
echoServAddr.sin_family = AF_INET;
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);
echoServAddr.sin_port = htons(echoServPort);
bind(servSock, (struct sockaddr *) &echoServAddr,
     sizeof(echoServAddr));
listen(servSock, MAXPENDING);
for (;;){
  clntLen = sizeof(echoClntAddr);
  clntSock = accept(servSock,(struct sockaddr *)&echoClntAddr,
                    &clntLen);
  recvMsgSize = recv(clntSock, echoBuffer, RCVBUFSIZE, 0);
  while (recvMsgSize > 0){
    send(clntSock, echoBuffer, recvMsgSize, 0);
    recvMsgSize = recv(clntSock, echoBuffer, RCVBUFSIZE, 0);
  }
  close(clntSock);
}
```

TCPによるクライアントプログラム例

MPI (Message Passing Interface)

- 並列計算向けの通信関数群
  - C, C++, Fortranのプログラムから呼び出し
  - ほぼ全ての並列計算機で利用可能

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

- 接続：
  - MPI_Init
- ホストの識別：
  - MPI_Comm_rank()（識別番号の取得）
- 転送
  - データ型単位
    - 新たにデータ型を定義可能
  - 一対一，一対多，多対多

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
  int myid, procs, ierr, i;
  double myval, val;
  MPI_Status status;
  FILE *fp;
  char s[64];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  if (myid == 0) {
    fp = fopen("test.dat", "r");
    fscanf(fp, "%lf", &myval);
    for (i = 1; i < procs; i++){
      fscanf(fp, "%lf", &val);
      MPI_Send(&val, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    }
    fclose(fp);
  } else
    MPI_Recv(&myval, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);


  printf("PROCS: %d, MYID: %d, MYVAL: %e¥n", procs, myid, myval);
  MPI_Finalize();

  return 0;
}
```

MPIのプログラム例

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
  int myid, procs, i;
  double myval, val;
  MPI_Status status;
  FILE *fp;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  if (myid == 0) {
    fp = fopen("test.dat", "r");
    fscanf(fp, "%lf", &myval);
    for (i = 1; i < procs; i++){
      fscanf(fp, "%lf", &val);
      MPI_Send(&val, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    }
    fclose(fp);
  } else
    MPI_Recv(&myval, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

  printf("PROCS: %d, MYID: %d, MYVAL: %e¥n", procs, myid, myval);
  MPI_Finalize();

  return 0;
}
```

MPIの準備

自分のプロセス番号（＝ランク）を取得

実行に参加しているプロセス数を取得

ランク0か否か

最初のデータは, myvalに格納

i ＝ 1～procs－1

次のデータを読み込み, valに格納

MPI_Sendにより, valの値をランク iに送信

ランク0以外のプロセスは, MPI_Recvでランク0から受信し, myvalに格納

myvalの値を表示

MPIの終了処理

MPIのプログラム例（Fortran）

```
program ex1
implicit none
include "mpif.h"

integer :: myid, procs, i
real(8) :: myval, val
integer :: ierr
integer, dimension(MPI_STATUS_SIZE) :: status

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, procs, ierr)

if (myid == 0) then
   open(10, file="test.dat")
   read(10, *) myval
   do i = 1, procs-1
      read(10, *) val
      call MPI_Send(val, 1, MPI_DOUBLE_PRECISION, i, 0, MPI_COMM_WORLD, ierr)
   end do
   close(10)
else
   call MPI_Recv(myval, 1, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD, status, ierr)
end if

print *, "PROCS: ", procs, " MYID: ", myid, " MYVAL: ", myval
call MPI_Finalize(ierr)

end program
```

MPIの準備

自分のプロセス番号（＝ランク）を取得

実行に参加しているプロセス数を取得

ランク0か否か

最初のデータは, myvalに格納

i ＝ 1～procs－1

次のデータを読み込み, valに格納

MPI_Sendにより, valの値をランク iに送信

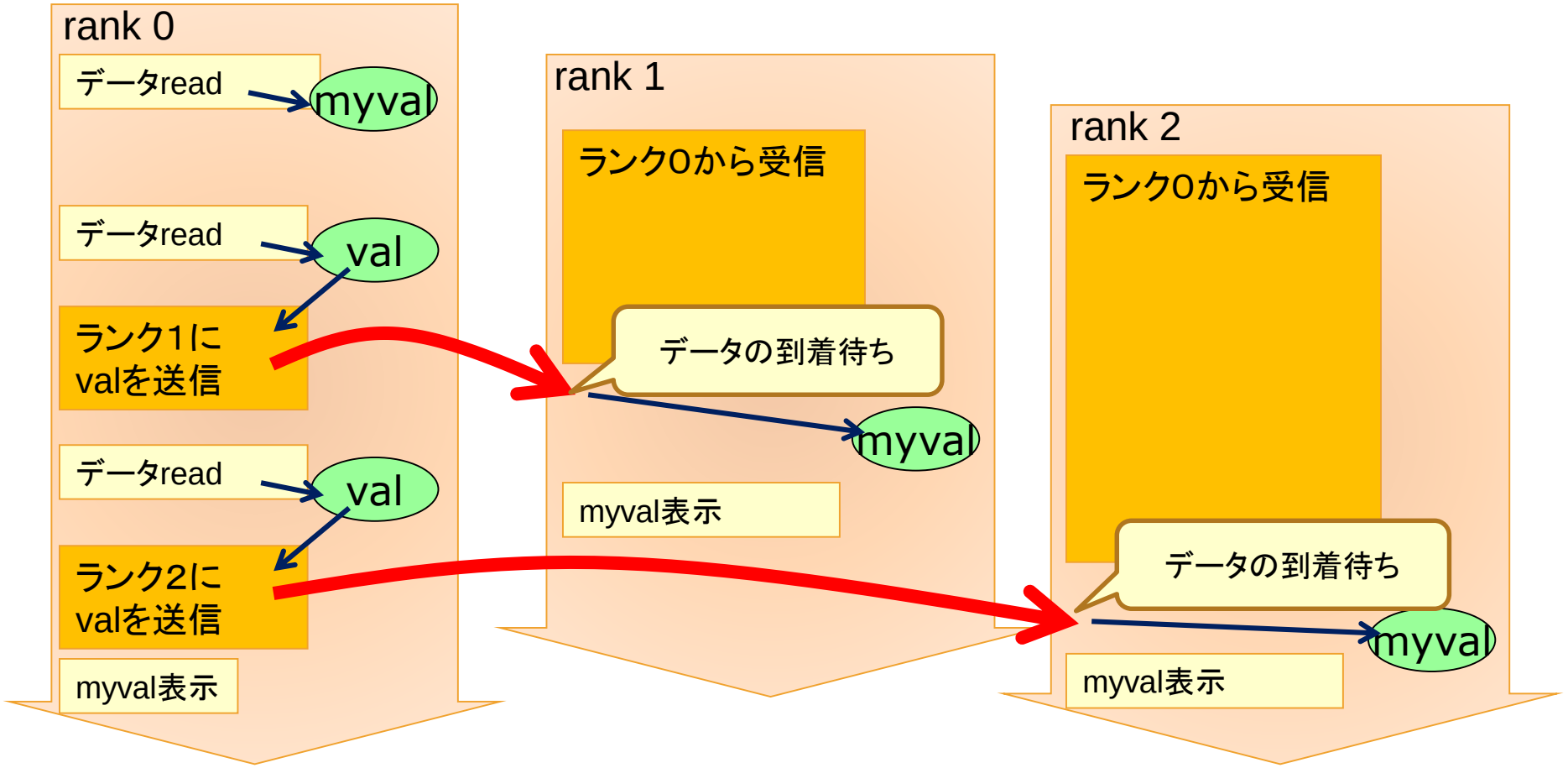
ランク0以外のプロセスは, MPI_Recvでランク0から受信し, myvalに格納

myvalの値を表示

MPIの終了処理

プログラム例の実行の流れ

- 複数の”プロセス”が，自分の番号（ランク）に応じて実行



実行例

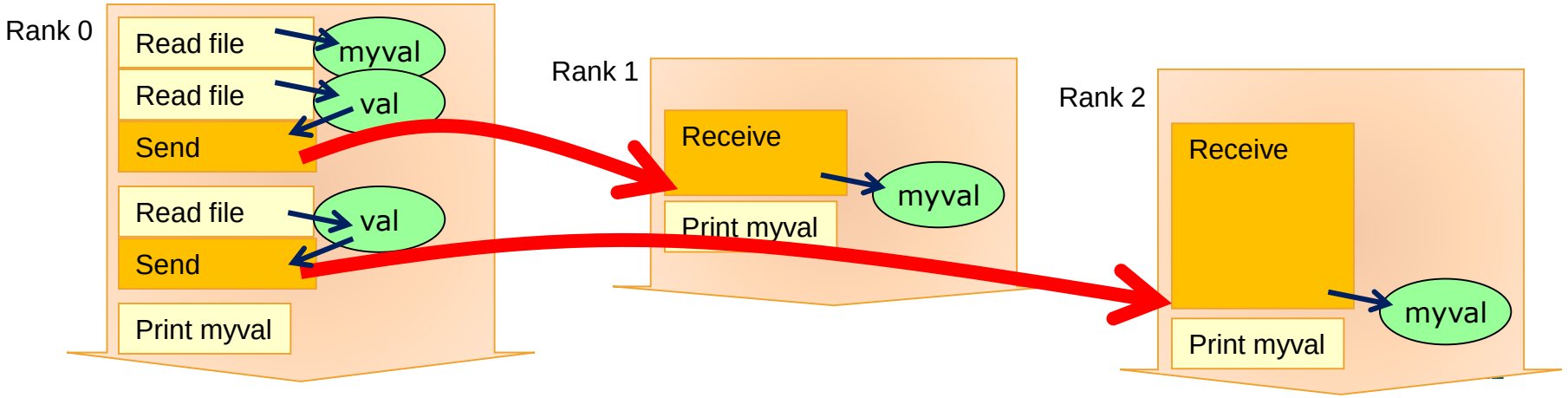
- 各プロセスがそれぞれ勝手に表示するので, 表示の順番は毎回変わる可能性がある

```
PROCS:  4 MYID:  1 MYVAL:  20.0000000000000000
PROCS:  4 MYID:  2 MYVAL:  30.0000000000000000
PROCS:  4 MYID:  0 MYVAL:  10.0000000000000000
PROCS:  4 MYID:  3 MYVAL:  40.0000000000000000
```

rank 1
rank 2
rank 0
rank 3

MPIインタフェースの特徴

- C/C++, Fortranプログラムから呼び出す関数（サブルーチン）
- 基本的に, 各プロセスが同じプログラムを実行する
- ランク（＝プロセス番号）を使って，プロセス毎に違う仕事を実行
- 他のプロセスの変数を直接読み書きすることはできない



MPIプログラムの基本構造

必須行

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
  ...

  MPI_Init(&argc, &argv);

  ...

  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);

  ...

  MPI_Send(&val, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
  ...
  MPI_Recv(&myval, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

  ...

  MPI_Finalize();

  return 0;
}
```

ヘッダファイル "mpi.h"

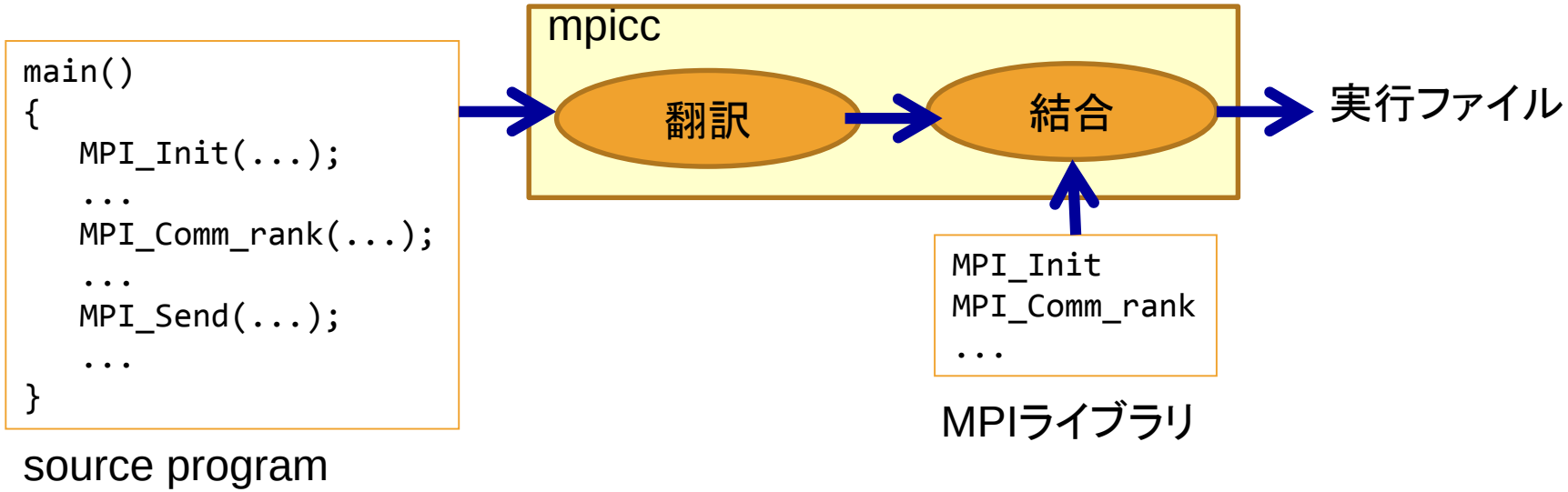
MPIの準備

MPI関数を使用可能な範囲

MPIの終了

MPIライブラリ

- MPI関数の実体は，MPIライブラリに格納されている
  - MPI用コンパイルコマンド（mpicc等）で
    MPIライブラリをプログラムに結合



MPIプログラムのコンパイル

- MPIライブラリが提供するコンパイルコマンドを利用

MPIライブラリ	コンパイルコマンド	
	C/C++	Fortran
Intel MPI	**mpiicc**	**mpiifort**
Open MPI	**mpicc**	**mpifort**
MVAPICH2	**mpicc**	**mpifort**
富士通	**mpifcc**	**mpifrt**

- 例）

```
mpiicc test.c -o test
```

MPIプログラムの実行

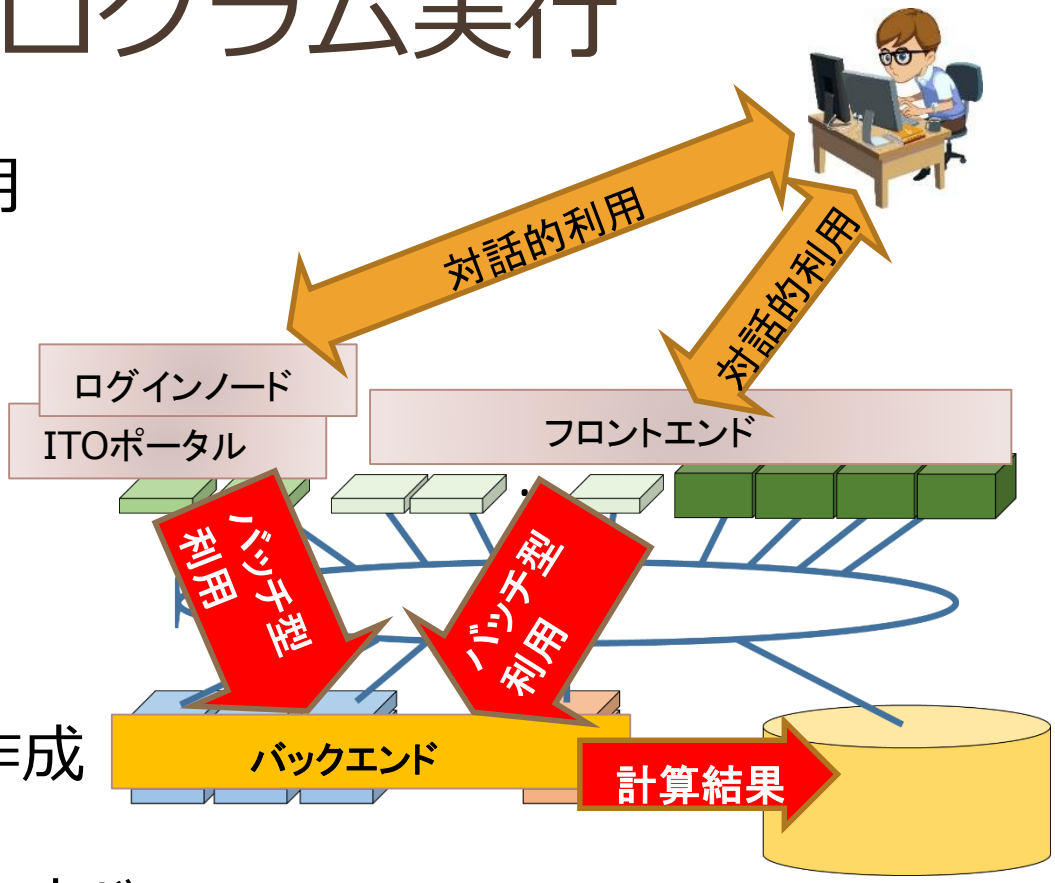
- 通常, mpiexec コマンドで実行
  - -np オプションでプロセス数を指定
  - 例）

```
mpiexec -np 8 ./test
```

  - 計算機や MPIライブラリによって,
    コマンド名, オプション, 環境変数
    が違う
    - マニュアル参照

ITOでの MPIプログラム実行

- 「バッチ型」による利用



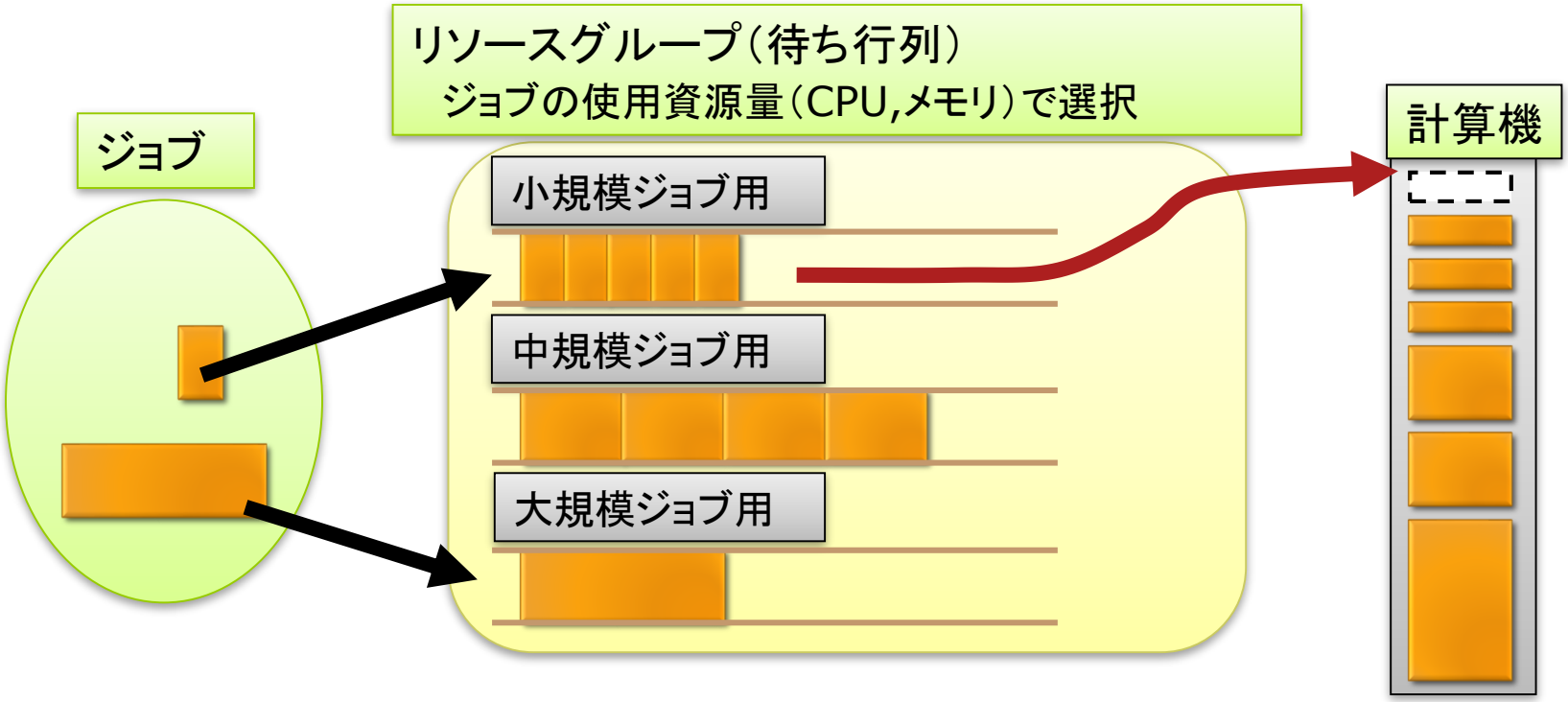
- 基本的な流れ：
  1．ジョブスクリプト作成
  2．ジョブ投入
  （システムの空き状況に応じて，
  順にジョブを実行）
  3．完了を待って，出力ファイル確認

バッチシステムの仕組み

- 処理してほしい内容を記述したファイルを投入
  - ジョブとして受付
- 資源の空き状況に応じて順に処理される
  - 要求内容や空き状況によっては先を越されることも

```
#!/bin/sh

#PJM -L "vnode=4"
#PJM -L "vnode-core=36"
#PJM -L "rscunit=ito-b"
#PJM -L "rscgrp=ito-g-16"
#PJM -L "elapse=10:00"

mpiexec -np 16 ./a.out
```

ジョブスクリプトの例



バッチ処理に用いるコマンド

- バッチジョブの投入
  pjsub

- バッチジョブの状況
  pjstat

- バッチジョブのキャンセル
  pjdel

pjsub バッチジョブの投入

- コマンド

```
$ pjsub オプション ジョブスクリプトファイル名
```

  - オプション：使用する資源等に関する指定
    - いつも同じ指定をするのであれば, ジョブスクリプトファイルの中に記述
    - pjsubコマンドでのオプション指定が優先
  - ジョブスクリプトファイル：依頼する処理内容
    - シェルスクリプトとして記述

- 例）ジョブスクリプトファイル test.sh を投入

```
$ pjsub test.sh
[INFO] PJM 0000 pjsub Job 28246 submitted.
```

ジョブ ID

今回の実習で使うジョブスクリプト

```
#!/bin/bash
#PJM -L "rscunit=ito-a"
#PJM -L "rscgrp=ito-a-lecture"
#PJM -L "vnode=2"
#PJM -L "vnode-core=36"
#PJM -L "elapse=00:05:00"
#PJM -j

module load intel/2017

export I_MPI_FABRICS=shm:ofa
export I_MPI_HYDRA_BOOTSTRAP=rsh
export I_MPI_HYDRA_BOOTSTRAP_EXEC=/bin/pjrsh
export I_MPI_HYDRA_HOST_FILE=${PJM_O_NODEINF}

export I_MPI_PERHOST=2

mpiexec.hydra -np 4 ./ex1
```

使用ノード数 2

ノード当たり使用可能最大コア数 36

実行時間 5分以内

Intel MPIのバージョン

Intel MPIの設定

ノードあたりのプロセス数

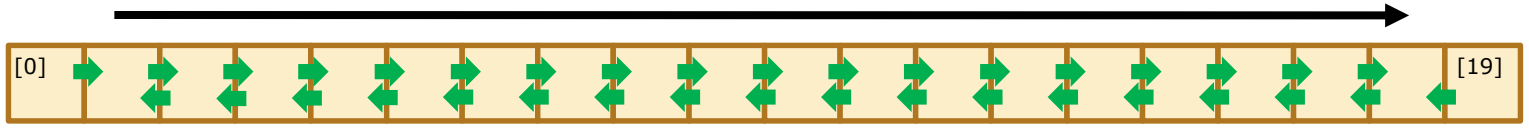
実行コマンド
（プロセス数 <= vnode * I_MPI_PERHOST
となるように，設定）

MPIプログラムの作成

- 以下の逐次プログラムを例に，並列化の手順例を紹介

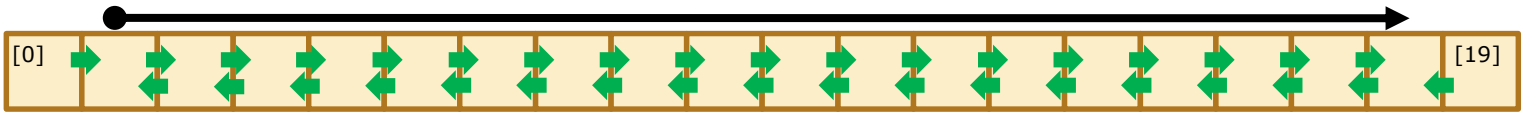
```
a = (double *)malloc(N*sizeof(double));
newa = (double *)malloc(N*sizeof(double));

for (i = 1; i < N-1; i++)
  a[i] = 0.0;
a[0] = 100.0;
a[N-1] = 10.0;

for (j = 0; j < REPEAT; j++){
  for (i = 1; i <= N-2; i++)
    newa[i] = (a[i-1]+a[i]+a[i+1])/3.0;

  for (i = 1; i <= N-2; i++)
    a[i] = newa[i];

  printf("Step %2d: ", j);
  for (i = 0; i < N; i++)
    printf(" %6.2f", a[i]);
  printf("¥n");
}
```

```
allocate(a(0:n-1))
allocate(newa(0:n-1))

a(1:n-2) = 0.0
a(0) = 100.0
a(n-1) = 10.0

do j = 1, repeat
  do i = 1, n-2
    newa(i) = (a(i-1) + a(i) + a(i+1)) / 3.0
  end do

  a(1:n-2) = newa(1:n-2)

  write(*,'(a5,i2,a1)',advance='no') "Step ",j,":"
  do i = 0, n-1
     write(*,'(f7.2)',advance='no') a(i)
  end do
  write(*,*)
end do
```

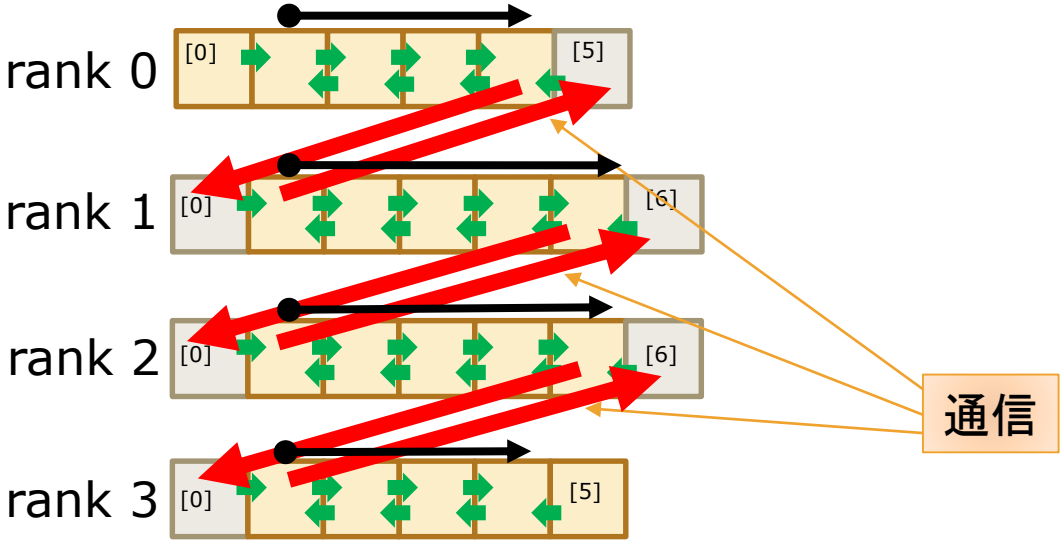


並列化の目標

- 並列化前



- 並列化後



並列化の手順（例）

- ステップ１：MPIの必須関数追加
- ステップ２：計算の並列化とデータの分割
  - ループをプロセスで分担
  - （必要に応じて）ループの割当に合わせて配列を分割
- ステップ３：通信関数追加
  - 隣接プロセスの計算結果取得
  - 全プロセスの計算結果結合

ステップ１：MPIの必須関数追加

- MPI_Init
  - MPIの初期化処理
- MPI_Finalize
  - MPIの終了処理
- ヘッダファイルの include
  - C/C++: mpi.h
  - Fortran: mpif.h

MPI_Init

C/C++:
```
int MPI_Init(int *argc, char **argv);
```

Fortran:
```
call MPI_Init(ierr)
  integer :: ierr
```

- MPI利用開始
  - プロセスの起動やプロセス間通信路の確立等
  - 他のMPI関数を呼ぶ前に，必ずこの関数を呼ぶ
- 引数
  - C/C++
    - *argc*, *argv* : main関数の2つの引数へのポインタ
      - 各プロセス起動時に実行ファイル名やオプションを共有するために参照
  - Fortran
    - *ierr* :エラー番号を返す整数変数
      - 全ての Fortran用 MPIルーチンに共通

MPI_Finalize

C/C++:
```
int MPI_Finalize();
```

Fortran:
```
call MPI_Finalize(ierr)
  integer :: ierr
```

- MPI利用終了
  - このルーチン実行後はMPIルーチンを呼び出せない
  - プログラム終了前に全プロセスで必ずこのルーチンを実行

MPI_Init, _Finalizeの追加

```
/* Initialize MPI */
MPI_Init(&argc, &argv);
```

```
/* Finalize MPI */
MPI_Finalize();
```

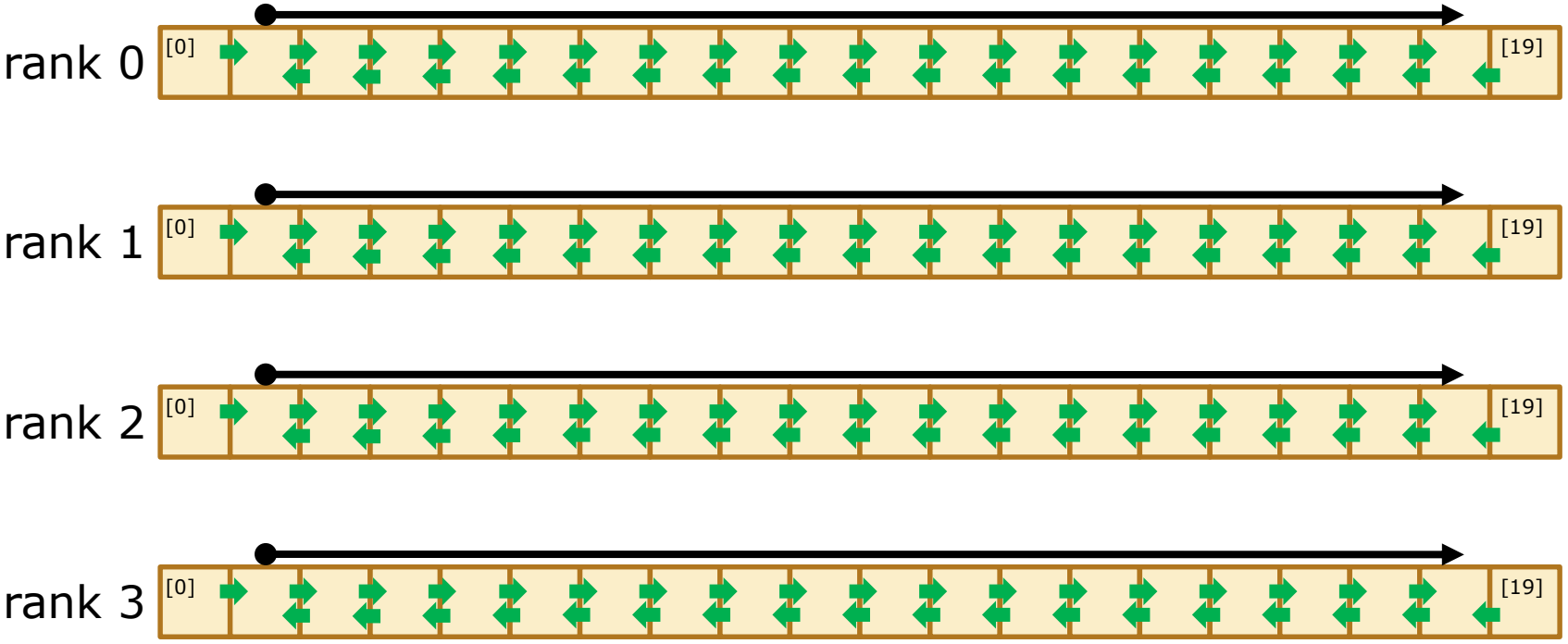
実習時に追加するコード（C/C++）

```
! Initialize MPI
call MPI_Init(ierr);
```

```
! Finalize MPI
call MPI_Finalize(ierr);
```

実習時に追加するコード（Fortran）

- 全プロセスが同じプログラムを実行



実習１ 準備

- ITOにログイン

- /home/tmp/mpi/mpiex-2018 を自分のホームにコピーし, mpiex-2018ディレクトリに移動

```
cp -r /home/tmp/mpi/mpiex-2018 .
cd mpiex-2018
```

実習1 MPI_Init, _Finalizeを追加

- ex1.c もしくは ex1.f90 を編集し, 全プロセスで同じプログラムを並列実行するプログラムに変更
  - ヘッダファイル, 変数宣言も追加

```
/* Add header file */
#include "mpi.h"
```

```
! Add header file
include 'mpif.h'
```

```
! Add necessary variable
integer :: ierr
```

- コンパイル後, ジョブ投入

```
module load intel/2017
mpiicc ex1.c -o ex1
```

```
module load intel/2017
mpiifort ex1.f90 -o ex1
```

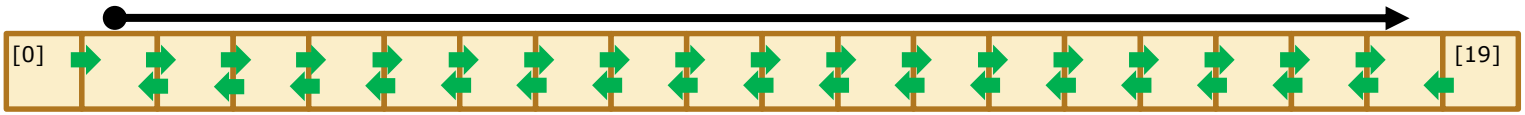
```
cat ex1.sh
pjsub ex1.sh
```

- 結果の確認

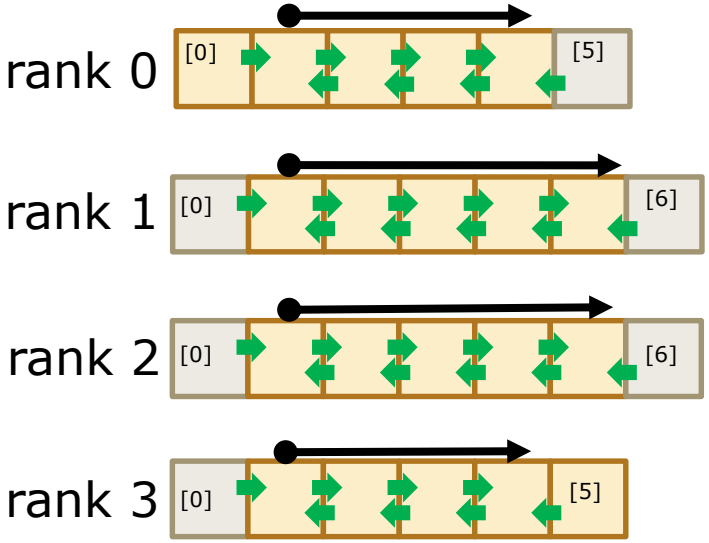
```
ls
cat ex1.sh.oジョブ番号
```

ステップ２：　計算の並列化とデータの分割

- 分割前

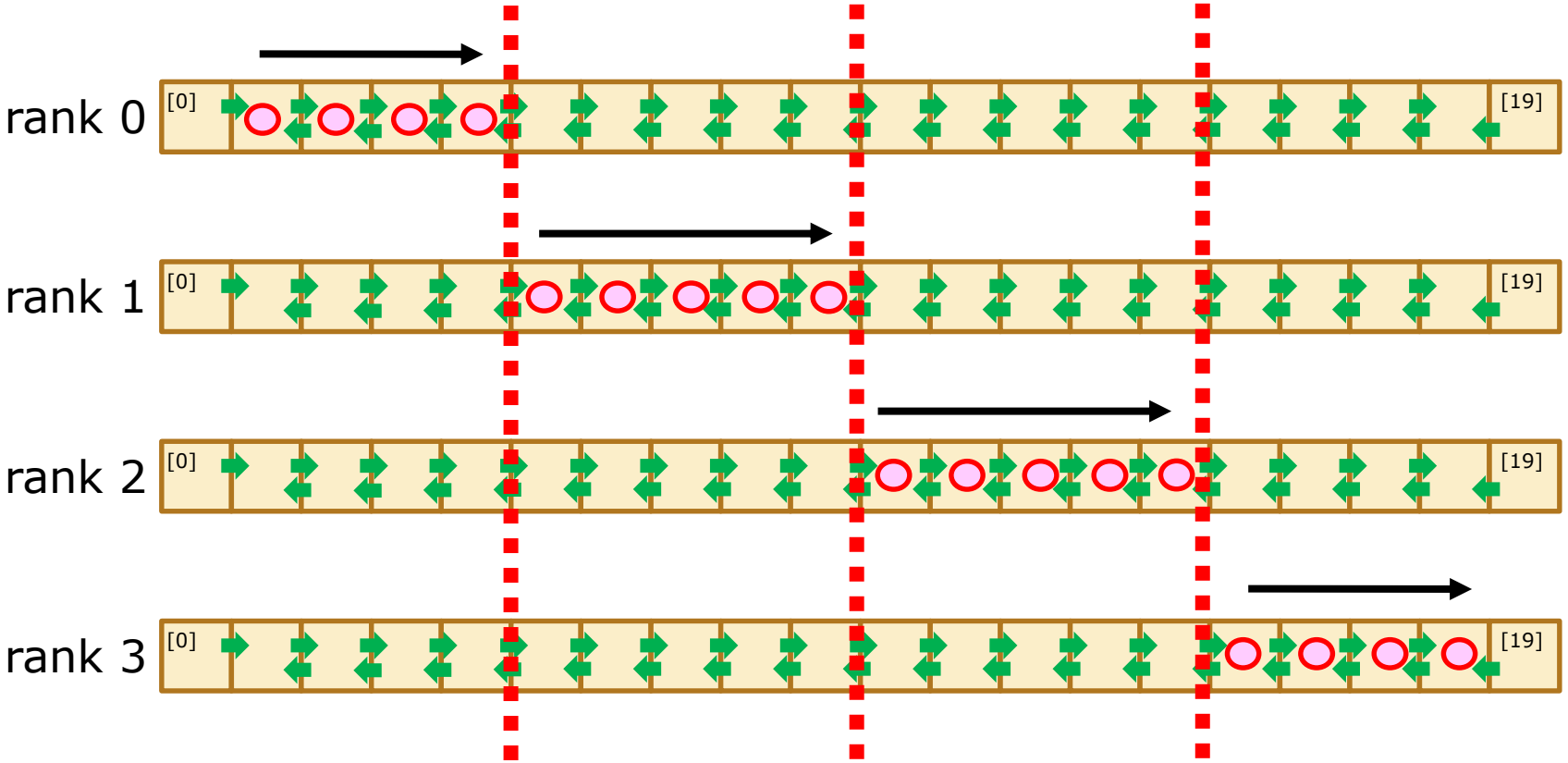


- 分割後



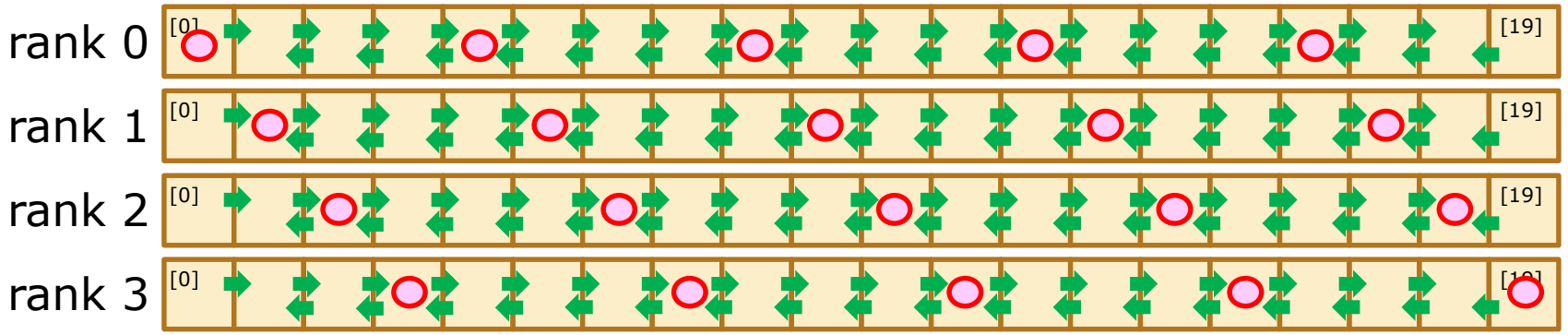
計算の並列化

- ループをプロセスに割り当て
  - 今回は「ブロック分割」を適用
    - ループを部分ループに分割してプロセスに割り当て

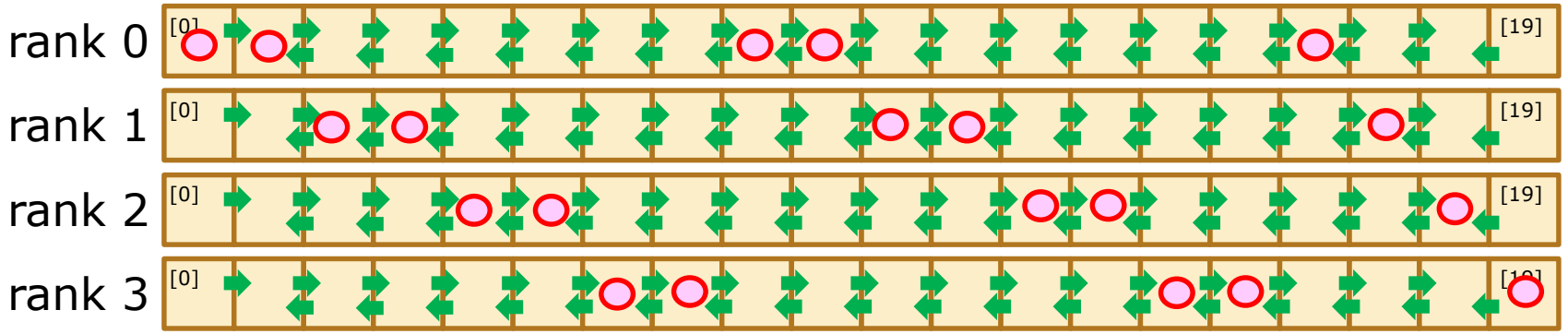


他の割当方法

- サイクリック

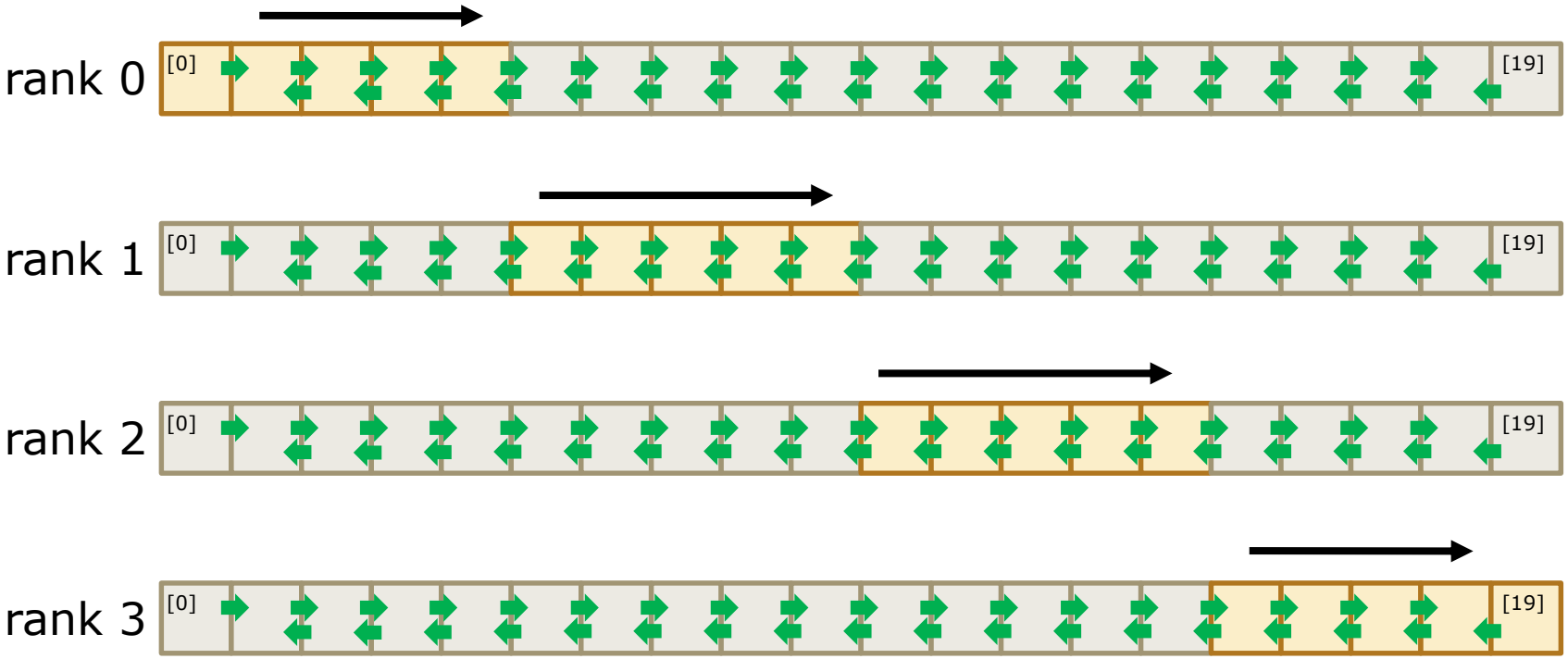


- ブロックサイクリック



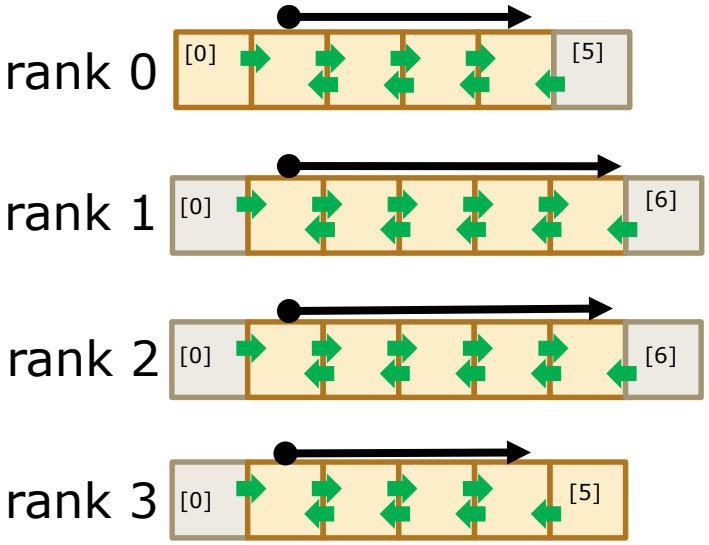
（必要に応じて）データの分割

- １台の計算ノードではメモリが不足する場合，データ分割
  - 通常，各プロセスが，自分の担当範囲のデータを配置



（必要に応じて）データ分割を調整

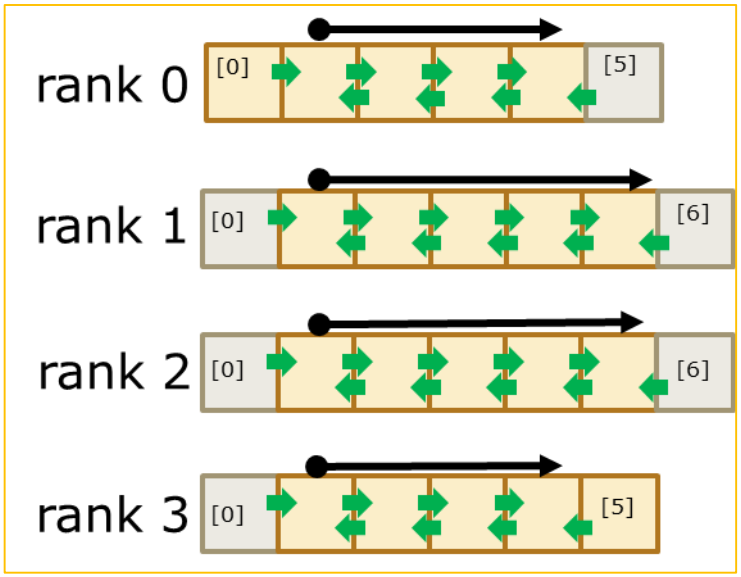
- 今回のプログラムでは, 隣接の要素も参照するので, 少し余裕を持って配列を配置しておく
  - あとで通信に利用する



Nがプロセス数で割り切れる場合のループとデータ分割

- ランク myid, プロセス数 procs を利用

ランク	配列の大きさ	ループ範囲
myid == 0	N/procs + 1	1 ... N/procs - 1
1 <= myid <= procs - 2	N/procs + 2	1 ... N/procs
myid == procs - 1	N/procs + 1	1 ... N/procs - 1



MPI_Comm_rank

C/C++:
```
int MPI_Comm_rank(MPI_Comm comm,
                  int *rank);
```

Fortran:
```
call MPI_Comm_rank(comm, rank, ierr)
  integer :: comm, rank, ierr
```

- そのプロセスのランク取得
- 引数
  - *comm* : “コミュニケータ”
  - *rank* : ランクを格納する場所
- コミュニケータ
  - プロセスのグループを表す識別子
  - 通常は，MPI_COMM_WORLD を指定
    - MPI_COMM_WORLD: 実行に参加する全プロセスによるグループ
    - プロセスを複数の小グループに分けて，
      それぞれ別の仕事をさせることも可能

MPI_Comm_size

C/C++:
```
int MPI_Comm_size(MPI_Comm comm,
                  int *size);
```

Fortran:
```
call MPI_Comm_size(comm, size, ierr)
  integer :: comm, size, ierr
```

- プロセス数取得
- 引数
  - *comm* : “コミュニケータ”
  - *size* : プロセス数を格納する場所

Nがプロセス数で割り切れる場合の配列サイズとループ終了値の計算例

```
/* Get myid and procs */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &procs);
```

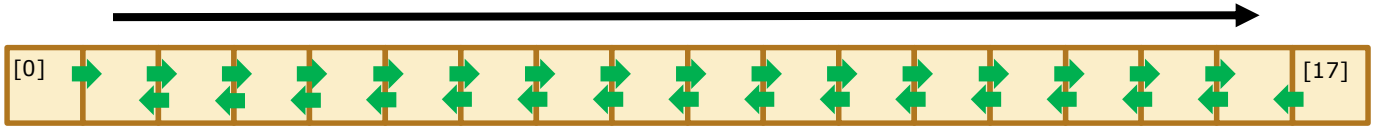
```
/* Divide N into procs */
divN = N / procs;
modN = N % procs;
if (modN != 0)
  printf("ERROR: modN is not 0 ¥n");

if ((myid == 0) || (myid == procs-1)){
  sizea = divN + 1;
  loopend = divN - 1;
} else {
  sizea = divN + 2;
  loopend = divN;
}
```

```
! Get myid and procs
call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, procs, ierr)
```

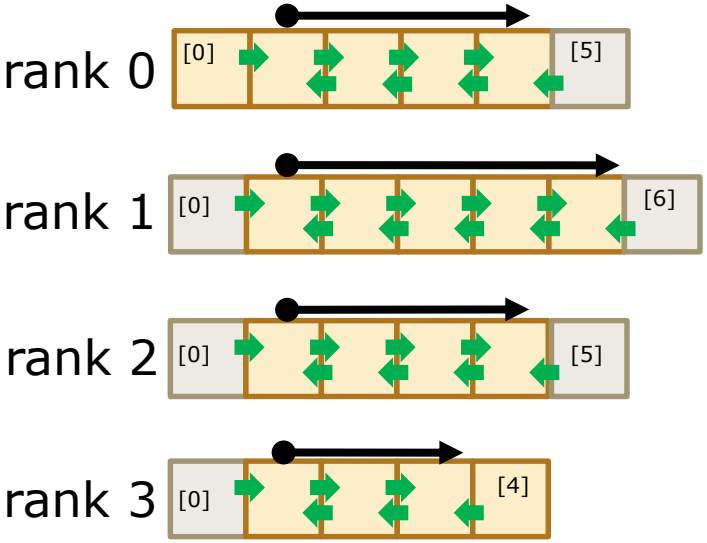
```
! Divide N into procs
divN = floor(real(N/procs))
modN = mod(N, procs)
if (modN /= 0) then
  write(*, *) "Error modN is not 0"
end if

if ((myid == 0) .or. (myid == procs-1)) then
   sizea = divN + 1
   loopend = divN - 1
else
   sizea = divN + 2
   loopend = divN
endif
```

Nがプロセス数で割り切れない場合

- 並列化前（N = 18）



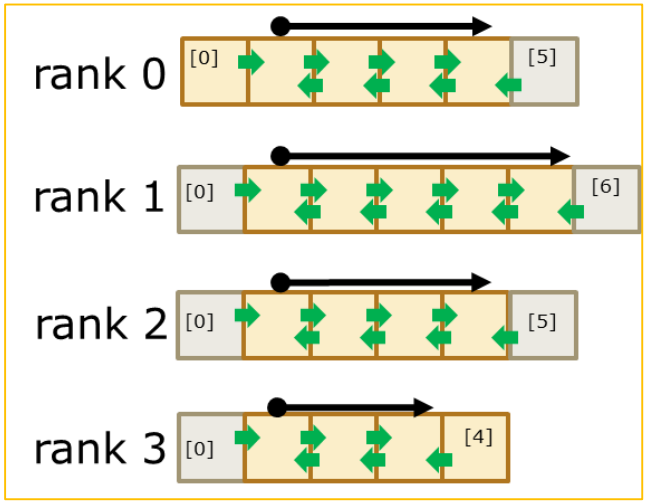
- 並列化後（プロセス数 4）



Nがプロセス数で割り切れない場合のループとデータ分割の例

- 余った分（= N % procs）を前半のプロセスに分配
  - divN = N / procs （切り捨て）

ランク	配列の大きさ	ループ範囲
myid == 0	divN + 2	1 ... divN
1 <= myid <= (N % procs) - 1	divN + 3	1 ... divN +1
(N % procs) <= myid <= procs - 2	divN + 2	1 ... divN
myid == procs - 1	divN + 1	1 ... divN - 1



Nがプロセス数で割り切れない場合の配列サイズとループ終了値の計算例

```
/* Divide N into procs */
divN = N / procs;
modN = N % procs;

if ((myid == 0) || (myid == procs-1) {
  sizea = divN + 1;
  loopend = divN - 1;
} else {
  sizea = divN + 2;
  loopend = divN;
}

if (myid < modN) {
  sizea++;
  loopend++;
}
```

```
! Divide N into procs
divN = floor(real(N/procs))
modN = mod(N, procs)

if ((myid == 0) .or. (myid == procs-1))
then
   sizea = divN + 1
   loopend = divN - 1
else
   sizea = divN + 2
   loopend = divN
endif

if (myid < modN) then
   sizea = sizea + 1
   loopend = divN + 1
end if
```

配列の初期化例

- 配列サイズ変更
- 初期化ループの範囲変更

```
/* Initialize local a */
a = (double *)malloc(sizea*sizeof(double));
newa = (double *)malloc(sizea*sizeof(double));
for (i = 1; i <= loopend; i++)
  a[i] = 0.0;
if (myid == 0)
  a[0] = 100.0;
if (myid == procs - 1)
  a[sizea - 1] = 10.0;
```

```
! Initialize local a
allocate(a(0:sizea-1))
allocate(newa(0:sizea-1))
a(1:loopend) = 0.0
if (myid == 0) then
   a(0) = 100.0
end if
if (myid == procs - 1) then
   a(sizea-1) = 10.0
end if
```

ループの並列化例

- ループの範囲修正
  - 表示部では, とりあえずランク0の配列のみ表示することにする

```
/* Calculate local area */
for (i = 1; i <= loopend; i++)
  newa[i] = (a[i-1]+a[i]+a[i+1])/3.0;
```

```
! Calculate local area
do i = 1, loopend
  newa(i) = (a(i-1) + a(i) + a(i+1)) / 3.0
end do
```

```
/* Update data */
for (i = 1; i <= loopend; i++)
  a[i] = newa[i];
```

```
! Update data
a(1:loopend) = newa(1:loopend)
```

```
/* Print local area */
if (myid == 0) {
  printf("Step %2d: ", j);
  for (i = 0; i < sizea; i++)
    printf(" %6.2f", a[i]);
  printf("¥n");
}
```

```
! Print local area
if (myid == 0) then
  write(*,'(a5,i2,a1)',advance='no') "Step ",j,":"
  do i = 0, sizea-1
    write(*,'(f7.2)',advance='no') a(i)
  end do
  write(*,*)
end if
```

実習2 計算の並列化とデータ分割

- ex2.c もしくは ex2.f90 を編集し, 計算を並列化してデータを分割するプログラムに変更
  - 必要な変数宣言を追加

```
/* Add necessary variables */
double *a, *newa;
int i, j;
int myid, procs, modN, divN, sizea, loopend;
```

```
! Add necessary variable
integer :: i, j, ierr
integer :: myid, procs, modN, divN, sizea, loopend
```

  - ex2-answer.c, ex2-answer.f90に回答例があります

- コンパイル後, ジョブ投入

```
mpiicc ex2.c -o ex2
pjsub ex2.sh
```

```
mpiifort ex2.f90 -o ex2
pjsub ex2.sh
```

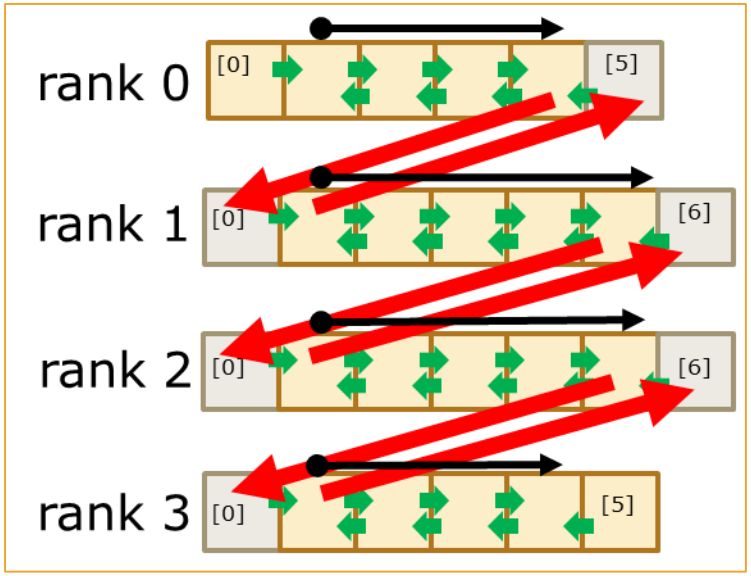
- 結果の確認

```
pjstat
ls
cat ex2.sh.oジョブ番号
```

ステップ３：通信の追加

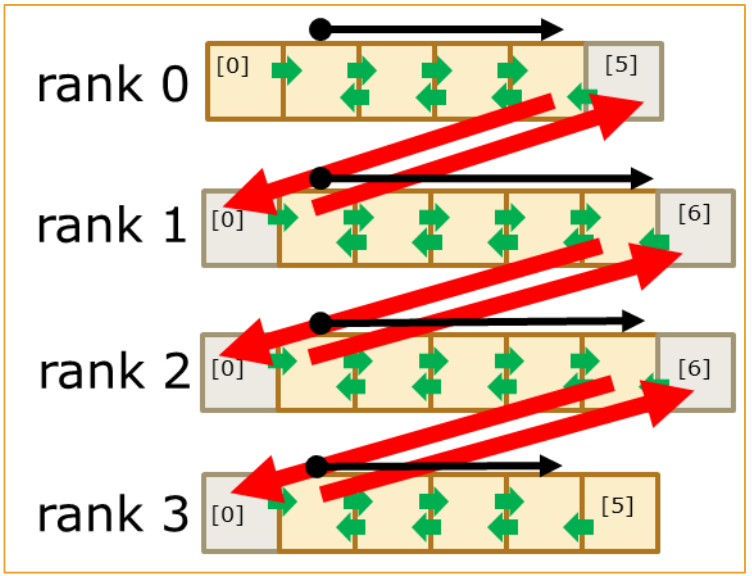
- 他のプロセスの計算結果を参照

- 今回のプログラムで使用する通信：
  - 一対一通信
    - 隣のプロセスと，境界部分の値を交換
  - 集団通信（グループ通信）
    - 全プロセスの計算結果の連結に利用
      - 配列表示用



隣のプロセスとの値交換

- 左のプロセス（myid-1）との交換（ランク 0以外）：
  - a[1] を送信
  - a[0] に受信
- 右のプロセス（myid+1）との交換（ランクprocs-1 以外）：
  - a[loopend] を送信
  - a[loopend+1] に受信

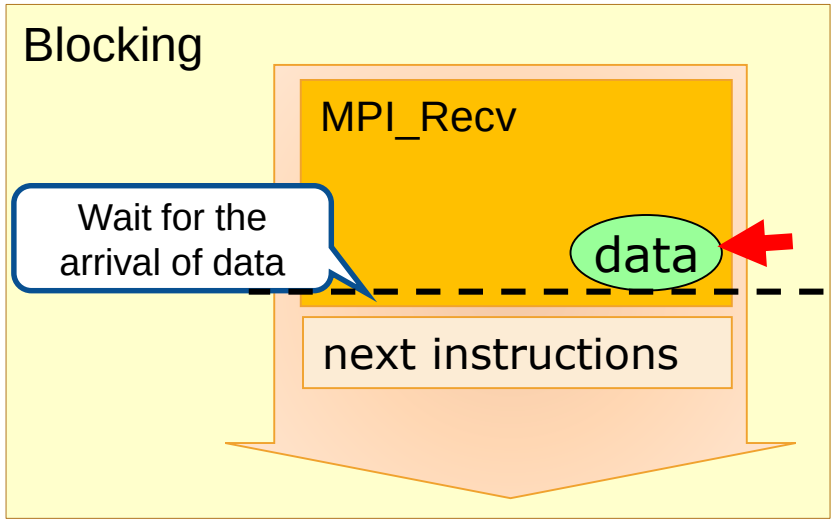


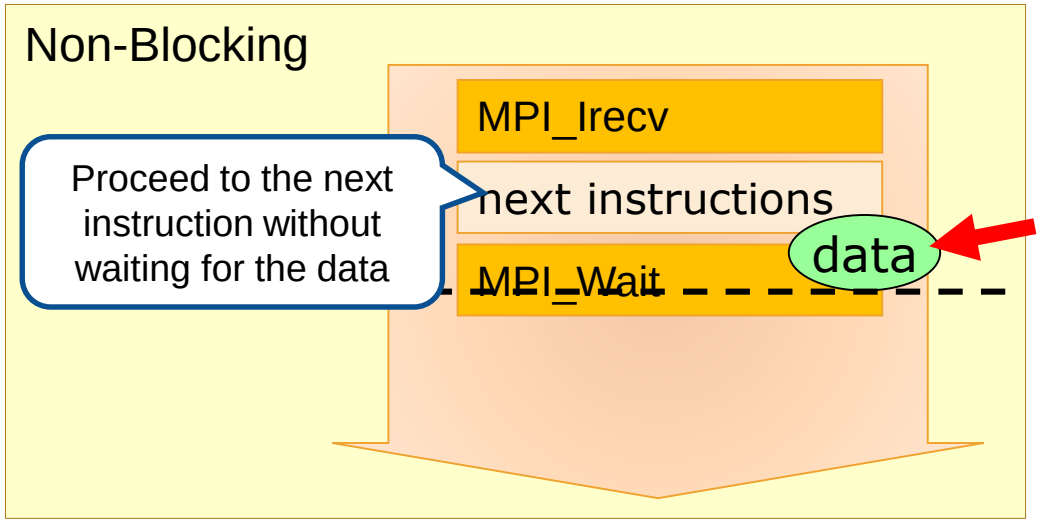
一対一通信関数

- MPI_Send, MPI_Recv
  - ブロッキング送信, 受信
- MPI_Isend, MPI_Irecv
  - 非ブロッキング送信, 受信
- MPI_Wait, MPI_Waitall
  - 非ブロッキング送信, 受信の完了待ち

ブロッキング通信 vs 非ブロッキング通信

- ブロッキング通信：通信の完了を待って，次の命令へ
- 非ブロッキング通信：通信の完了を待たずに，次の命令へ
  - 通信完了は，別途，MPI_Wait関数等で待つ





非ブロッキング通信の利点

- デッドロックの回避
  - お互いに相手のデータの送信待ち, という状態を回避
- 通信時間の隠蔽
  - データが転送されている間に他の計算を進める

非ブロッキング通信による デッドロックの回避

- デッドロック：
  何らかの理由でプログラムを進行できなくなった状態
  - MPIプログラムにおけるデッドロックの例：
    - お互い, 相手からデータが送信されないと
      自分の送信を始めない

```
if (myid == 0){
  MPI_Recv from rank 1
  MPI_Send to rank 1
}
if (myid == 1){
  MPI_Recv from rank 0
  MPI_Send to rank 1
}
```

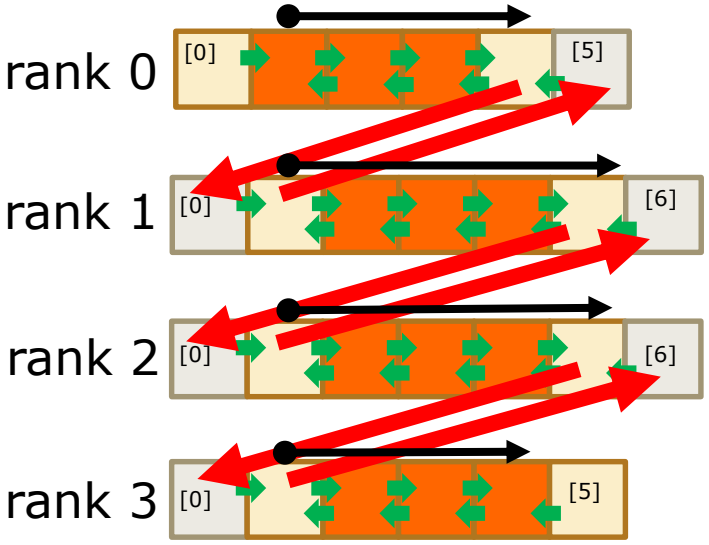
- 非ブロッキング通信で回避：
  - 受信開始後, 完了を待たずに送信開始

```
if (myid == 0){
  MPI_Irecv from rank 1
  MPI_Send to rank 1
  MPI_Wait
}
if (myid == 1){
  MPI_Irecv from rank 0
  MPI_Send to rank 0
  MPI_Wait
}
```

非ブロッキング通信による通信時間の隠蔽

- 非ブロッキング通信開始後, その通信と無関係な計算実行
  ⇒ 見かけ上, 通信時間が 0 に近づく

- 今回のプログラム
  - 通信中に, 隣のプロセスの値を必要としない範囲の計算が可能



非ブロッキング通信の注意点

- 通信開始後, 完了待ち（Wait）までは,
  通信対象領域を読み書きしない
  - MPI_Isend
    - 完了前に送信対象領域を書き換えると,
      書き換え前と後のどちらのデータが送信されるか不明

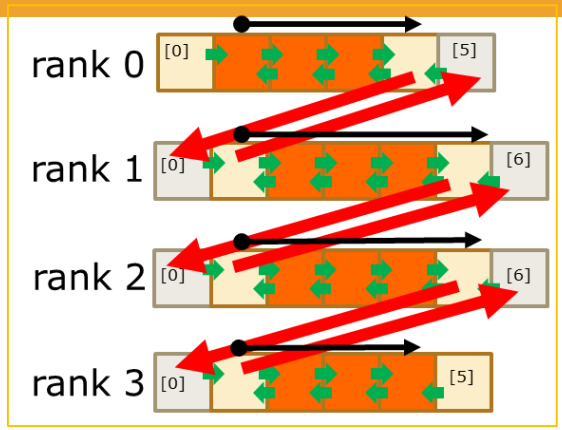
```
MPI_Isend(A)
A = ...
MPI_Wait()
```

  - MPI_Irecv
    - 完了前に受信対象領域を参照すると,
      データ到着前と後のどちらのデータを参照するか不明

```
MPI_Irecv(A)
... = A
MPI_Wait()
```

通信を隠蔽するプログラムの流れ

- 計算を，内部と境界部に分ける




```
/* Exchange values */
if (myid > 0) { /* left */
  MPI_Isend(左にa[1] を送信開始);
  MPI_Irecv(左からa[0] に受信開始);
}
if (myid < procs-1) {/* right */
  MPI_Isend(右にa[loopend]を送信開始);
  MPI_Irecv(右からa[loopend+1]に受信開始);
}

/* Modify loop to calculate internal area only */
for (i = 2; i <= loopend-1; i++)
  newa[i] = (a[i-1]+a[i]+a[i+1])/3.0;

/* Wait for all non-blocking communications */
MPI_Waitall(全ての非ブロッキング通信);

/* Calculate edges */
newa[1] =
 (a[0]+a[1]+a[2])/3.0;
newa[loopend] = (a[loopend-1]+a[loopend]+
                 a[loopend+1])/3.0;
```

```
! Exchange values
if (myid > 0) then
  call MPI_Isend(左にa(1)を送信開始)
  call MPI_Irecv(左からa(0)に受信開始)
end if
if (myid < procs - 1) then
  call MPI_Isend(右にa(loopend)を送信開始)
  call MPI_Irecv(右からa(loopend+1を受信開始)
end if

! Modify loop to calculate internal area only
do i = 2, loopend-1
  newa(i) = (a(i-1) + a(i) + a(i+1)) / 3.0
end do

! Wait for all non-blocking communications
call MPI_Waitall(全ての非ブロッキング通信)

! Calculate edges
newa(1) = (a(0) + a(1) + a(2)) / 3.0
newa(loopend) = (a(loopend-1) + a(loopend) + &
                 a(loopend+1)) / 3.0
```

MPI_Send

C/C++:
```
int MPI_Send(void *b, int c, MPI_Datatype d,
             int dest, int t, MPI_Comm comm);
```

Fortran:
```
call MPI_Send(b, c, d, dest, t, comm, ierr)
   <type>  :: b
   integer :: c, d, dest, t, comm, ierr
```

- ブロッキング送信
- 引数
  - ***b*** : 送信データの先頭
  - ***c*** : 要素数
  - ***d*** : データ型
  - ***dest*** : 送信先ランク
  - ***t*** : タグ
    - メッセージにつける番号
    - 不規則な通信の記述に利用
    - 通常は 0 で可
  - ***comm*** : コミュニケータ
  - ***r*** : リクエスト情報の格納場所

MPIの主なデータ型

データ型	MPIのデータ型（C/C++）	MPIのデータ型（Fortran）
整数	MPI_INT	MPI_INTEGER
単精度実数	MPI_FLOAT	MPI_REAL
倍精度実数	MPI_DOUBLE	MPI_DOUBLE_PRECISION
単精度複素数		MPI_COMPLEX
倍精度複素数		MPI_DOUBLE_COMPLEX
文字	MPI_CHAR	MPI_CHARACTER

MPI_Recv

C/C++:
```
int MPI_Recv(void *b, int c, MPI_Datatype d,
             int dest, int t, MPI_Comm comm,
             MPI_Status *s);
```

Fortran:
```
call MPI_Isend(b, c, d, dest, t, comm, s, ierr)
   <type>  :: b
   integer :: c, d, dest, t, comm, ierr
   integer,dimension(MPI_STATUS_SIZE) :: s
```

- ブロッキング受信
- 引数
  - ***b*** : 受信データの格納場所の先頭
  - ***c*** : 要素数
  - ***d*** : データ型
  - ***dest*** : 受信元ランク
  - ***t*** : タグ
  - ***comm*** : コミュニケータ
  - ***s*** : 受信したデータの情報の格納場所
    - 送信元ランク, タグの値, 等
    - 不規則な通信に利用
    - 通常は MPI_STATUS_IGNORE を指定

MPI_Isend

C/C++:
```
int MPI_Isend(void *b, int c, MPI_Datatype d,
              int dest, int t, MPI_Comm comm,
              MPI_Request *r);
```

Fortran:
```
call MPI_Isend(b, c, d, dest, t, comm, r, ierr)
   <type>  :: b
   integer :: c, d, dest, t, comm, r, ierr
```

- 非ブロッキング送信
- 引数
  - ***b*** : 送信データの先頭
  - ***c*** : 要素数
  - ***d*** : データ型
  - ***dest*** : 送信先ランク
  - ***t*** : タグ
  - ***comm*** : コミュニケータ
  - ***r*** : リクエスト情報の格納場所

MPI_Irecv

C/C++:
```
int MPI_Irecv(void *b, int c, MPI_Datatype d,
              int dest, int t, MPI_Comm comm,
              MPI_Request *r);
```

Fortran:
```
call MPI_Irecv(b, c, d, dest, t, comm, r, ierr)
   <type> :: b
   integer :: c, d, dest, t, comm, r, ierr
```

- 非ブロッキング受信
- 引数
  - ***b*** : 受信データの格納場所の先頭
  - ***c*** : 要素数
  - ***d*** : データ型
  - ***dest*** : 受信元ランク
  - ***t*** : タグ
  - ***comm*** : コミュニケータ
  - ***r*** : リクエスト情報の格納場所

MPI_Wait

C/C++:
```
int MPI_Wait (MPI_Request *r, MPI_Status *s);
```

Fortran:
```
call MPI_Wait(r, s)
                                    integer :: r
integer,dimension(MPI_STATUS_SIZE) :: s
```

- 非ブロッキング通信の完了待ち
- 引数
  - ***r*** : リクエスト情報の格納場所
  - ***s*** : 受信したデータの情報の格納場所
    - MPI_STATUS_IGNORE指定可

リクエスト情報

- 非ブロッキング通信の完了待ちに必要な情報
- 各非ブロッキング通信発行毎に, 別の場所に保存
  - 完了待ち後は, 再利用可能

```
for () {
 MPI_Isend(..., &r1);
 MPI_Irecv(..., &r2);
  ...
 MPI_Wait(&r1, MPI_IGNORE_STATUS);
 MPI_Wait(&r2, MPI_IGNORE_STATUS);
}
```

非ブロッキング通信の完了待ち（C/C++）

```
do
 call MPI_Isend(..., r1, ...);
 call MPI_Irecv(..., r2, ...);
  ...
 call MPI_Wait(r1, MPI_IGNORE_STATUS);
 call MPI_Wait(r2, MPI_IGNORE_STATUS);
end do
```

非ブロッキング通信の完了待ち（Fortran）

MPI_Waitall

C/C++:
```
int MPI_Waitall (int c, MPI_Request *r,
                 MPI_Status *s);
```

Fortran:
```
 call MPI_Irecv(c, r, s)
                                    integer :: c
                     integer,dimension(:) :: r
integer,dimension(MPI_STATUS_SIZE,:) :: s
```

- 複数の非ブロッキング通信の完了待ち
- 引数
  - ***c*** : 待つリクエストの数
  - ***r*** : リクエスト情報の配列
  - ***s*** : 受信したデータの情報の格納場所の配列
    - MPI_STATUS_IGNORE指定可

非ブロッキング通信による データ交換と並列計算の例（C/C++）

```
/* Allocate an array of requests */
reqs = (MPI_Request *)malloc(4*sizeof(MPI_REQUEST));
```

```
/* Exchange values */
nreqs = 0
if (myid > 0) { /* left */
  MPI_Isend(&(a[1]), 1, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD, &(reqs[nreqs]));
  nreqs++;
  MPI_Irecv(&(a[0]), 1, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD, &(reqs[nreqs]));
  nreqs++;
}
if (myid < procs-1) {/* right */
  MPI_Isend(&(a[loopend]), 1, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD, &(reqs[nreqs]));
  nreqs++;
  MPI_Irecv(&(a[loopend+1]), 1, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD, &(reqs[nreqs]));
  nreqs++;
}
```

```
/* Modify loop to calculate internal area only */
       for (i = 2; i <= loopend - 1; i++)
           newa[i] = (a[i-1] + a[i] + a[i+1])/3.0;
```

```
/* Wait for all non-blocking communications */
MPI_Waitall(nreqs, reqs, MPI_STATUS_IGNORE);
```

```
/* Calculate edges */
newa[1] = (a[0] + a[1] + a[2])/3.0;
newa[loopend] = (a[loopend-1] + a[loopend] + a[loopend+1])/3.0;
```

非ブロッキング通信による データ交換と並列計算の例（Fortran）

```
! Allocate an array of requests
allocate(reqs(0:3))
```

```
! Exchange values
nreqs = 0
if (myid > 0) then
  call MPI_Isend(a(1), 1, MPI_DOUBLE_PRECISION, myid - 1, 0, MPI_COMM_WORLD, reqs(nreqs), ierr)
  nreqs = nreqs + 1
  call MPI_Irecv(a(0), 1, MPI_DOUBLE_PRECISION, myid - 1, 0, MPI_COMM_WORLD, reqs(nreqs), ierr)
  nreqs = nreqs + 1
end if
  if (myid < procs - 1) then
    call MPI_Isend(a(loopend), 1, MPI_DOUBLE_PRECISION, myid + 1, 0, MPI_COMM_WORLD, reqs(nreqs), ierr)
    nreqs = nreqs + 1
    call MPI_Irecv(a(loopend+1), 1, MPI_DOUBLE_PRECISION, myid + 1, 0, MPI_COMM_WORLD, reqs(nreqs), ierr)
    nreqs = nreqs + 1
end if
```

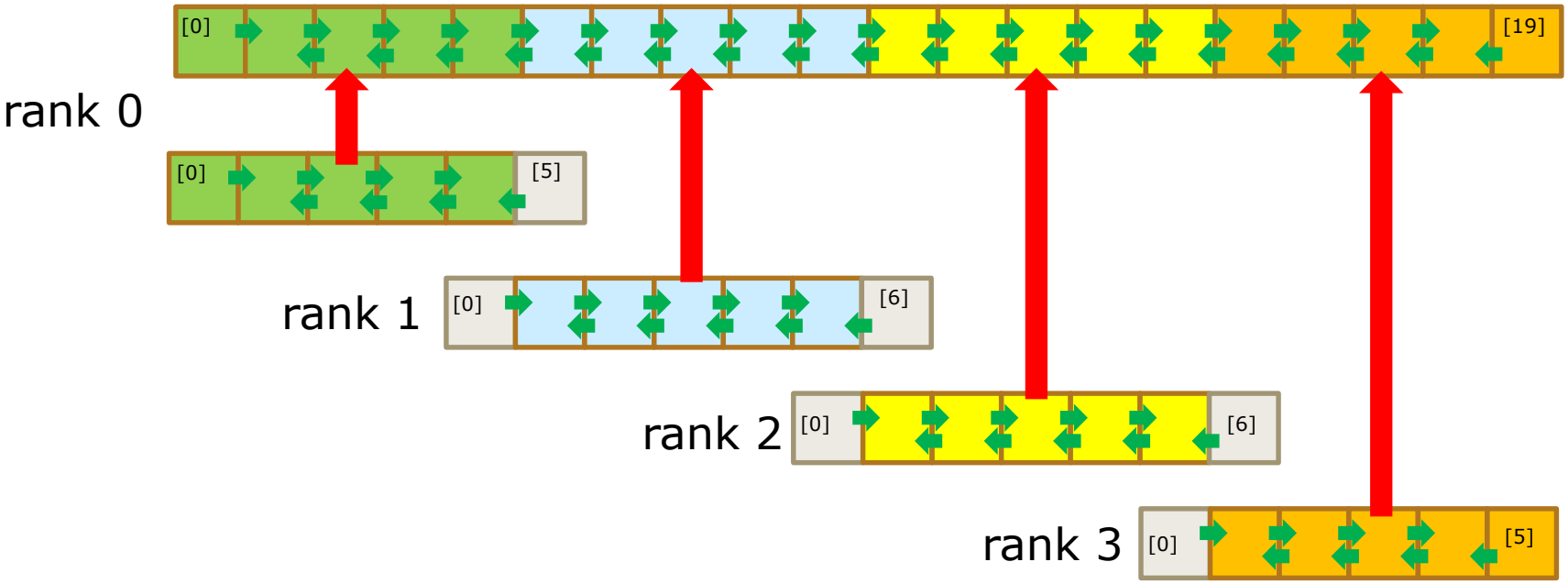
```
  ! Modify loop to calculate internal area only
   do i = 2, loopend-1
      newa(i) = (a(i-1) + a(i) + a(i+1)) / 3.0
   end do
```

```
 ! Wait for all non-blocking communications
   call MPI_Waitall(nreqs, reqs, MPI_STATUS_IGNORE, ierr)
```

```
 ! Calculate edges
   newa(1) = (a(0) + a(1) + a(2)) / 3.0
   newa(loopend) = (a(loopend-1) + a(loopend) + a(loopend+1)) / 3.0
```

全プロセスの計算結果の収集

- 今回のプログラムでは,
  計算結果を一つのプロセスにまとめて, 表示したい
  - 各プロセスで表示するとバラバラになるため
- 集団通信関数の一つ, MPI_Gather関数を利用



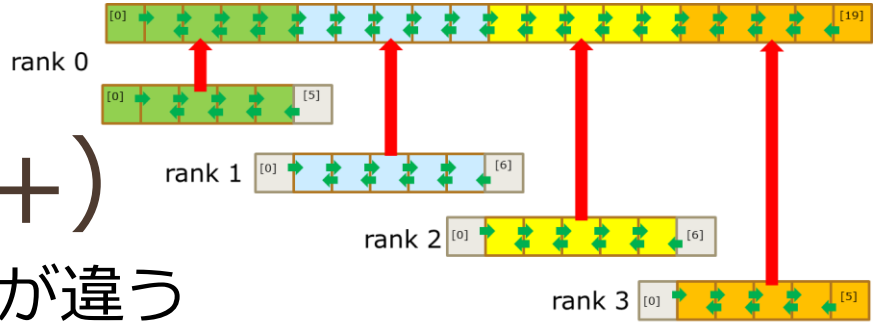
MPI_Gather

C/C++:
```
int MPI_Gather(
  void *b1, int c1, MPI_Datatype d1,
  void *b2, int c2, MPI_Datatype d2,
  int root, MPI_Comm  comm);
```

Fortran:
```
call MPI_Gather(b1, c1, d1, b2, c2, d2, root,
                comm, ierr)
<type>,dimension(:) :: b1, b2
  integer :: c1, d1, c2, d2, root, comm, ierr
```

- 全プロセスのデータを一つの配列にプロセス順に連結して格納
- 引数
  - ***b1***：送信データの先頭
  - ***c1***：送信データの要素数
  - ***d1***：送信データのデータ型
  - ***b2***：受信データの先頭
  - ***c2***：受信データの要素数
  - ***d2***：受信データのデータ型
  - ***root***：収集したデータを格納するランク
  - ***comm***：コミュニケータ
- 注意）*b2, c2, d2* は, rootのみで意味を持つ

データを集めて表示する例（C/C++）



- rank0のみMPI_Gatherの先頭が違う

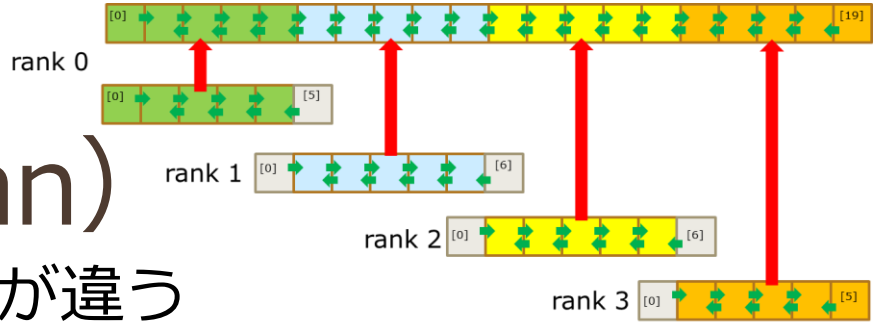
```
/* Allocate an array for MPI_Gather */
if (myid == 0)
  worka = (double *)malloc(N*sizeof(double));
```

```
/* Set start address for MPI_Gather */
if (myid == 0)
  starta = 0;
else
  starta = 1;
```

```
/* Gather data to rank 0 and print */
MPI_Gather(&(a[starta]), divN, MPI_DOUBLE, worka, divN, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (myid == 0) {
  printf("Step %2d: ", j);
  for (i = 0; i < N; i++)
    printf(" %6.2f", worka[i]);
  printf("¥n");
}
```

```
/* Free the array for MPI_Gather */
   if (myid == 0)
       free(worka);
```

データを集めて表示する例（Fortran）



- rank0のみMPI_Gatherの先頭が違う

```
! Allocate an array for MPI_Gather
if (myid == 0) then
   allocate(worka(0:n-1))
end if
```

```
! Set start address for MPI_Gather
if (myid == 0) then
  starta = 0
else
  starta = 1
end if
```

```
!Gather data into rank 0 and print
call MPI_Gather(a(starta), divN, MPI_DOUBLE_PRECISION, worka, divN, MPI_DOUBLE_PRECISION, 0,&
                MPI_COMM_WORLD, ierr)
if (myid == 0) then
  write(*,'(a5,i2,a1)',advance='no') "Step ",j,":"
  do i = 0, n-1
    write(*,'(f7.2)',advance='no') worka(i)
  end do
  write(*,*)
end if
```

```
! Free the array for MPI_Gather
if (myid == 0) then
   deallocate(worka)
end if
```

終了時の注意

- 動的配列の解放は全プロセスの通信が完了後
  - 未完了の通信による領域違反を防ぐ
  - 同期関数を利用

```
free(a);
free(newa);

MPI_Finalize();
```

動的配列の解放部分（C/C++）

```
deallocate(a)
deallocate(newa)

call MPI_Finalize(ierr)
```

動的配列の解放部分（Fortran）

MPI_Barrier

C/C++:
```
int MPI_Barrier(MPI_Comm comm);
```

Fortran:
```
call MPI_Barrier(comm, ierr)
  integer :: comm, ierr
```

- 全プロセスが到着するまで待つ（同期）
- 引数
  - ***comm*** : コミュニケータ

```
/* Wait for other process before free */
MPI_Barrier(MPI_COMM_WORLD);
```

```
! Wait for other process before free
call MPI_Barrier(MPI_COMM_WORLD, ierr)
```

実習3 通信

- ex3.c もしくは ex3.f90 を編集し, 隣のプロセスとのデータ交換, および計算結果の収集のための通信を行うプログラムに変更
  - 変数定義の追加

```
/* Add necessary variables */
double *a, *newa, *worka;
int i, j, nreqs, starta;
int myid, procs, modN, divN, sizea, loopend;
MPI_Request *reqs;
```

```
! Add necessary variables
real(8), dimension(:), allocatable :: a, newa, worka
integer :: i, j, ierr, nreqs, starta
integer :: myid, procs, modN, divN, sizea, loopend
integer, dimension(:), allocatable :: reqs
```

  - ex3-answer.c, ex3-answer.f90 に回答例があります

- コンパイル後, ジョブ投入

```
mpiicc ex3.c -o ex3
pjsub ex3.sh
```

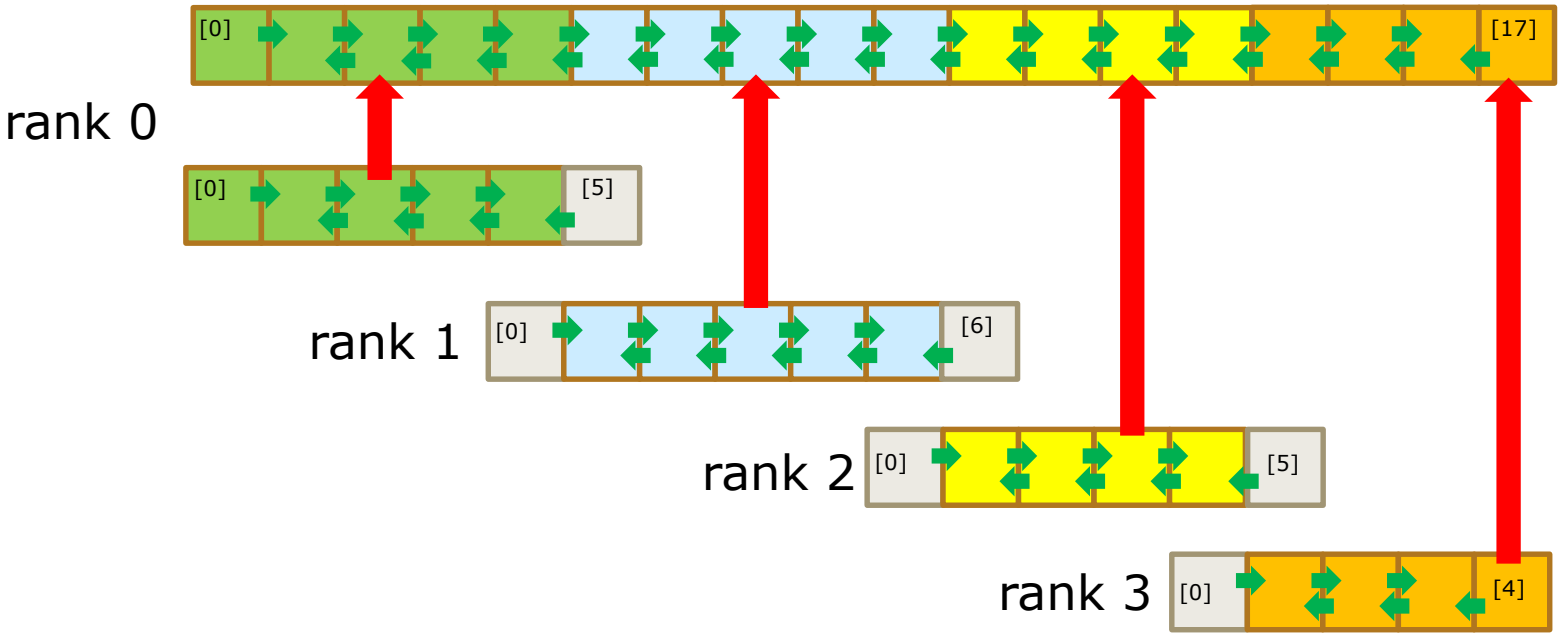
```
mpiifort ex3.f90 -o ex3
pjsub ex3.sh
```

- 結果の確認

```
pjstat
ls
cat ex3.sh.oジョブ番号
```

Nがプロセス数で割り切れない場合の全プロセスの計算結果の収集

- 送信データの要素数が, プロセスごとに異なる



MPI_Gatherv

C/C++:
```
int MPI_Gatherv(
  void *b1, int c1, MPI_Datatype d1,
  void *b2, const int x[], const int y[],
  MPI_Datatype d2,
  int root, MPI_Comm  comm);
```

Fortran:
```
 call MPI_Gatherv(b1, c1, d1, b2, x, y, d2, root,
                  comm, ierr)
 <type>,dimension(:) :: b1, b2
              integer :: c1, d1, d2, root, comm, ierr
integer,dimension(:) :: x, y
```

- プロセスごとにサイズが違うデータの連結
- 引数
  - ***b1*** : 送信データの先頭
  - ***c1*** : 送信データの要素数
  - ***d1*** : 送信データのデータ型
  - ***b2*** : 受信データの先頭
  - ***x*** : 各ランクからの送信データ要素数の配列
  - ***y*** : 各ランクからの送信データを格納する位置の配列
  - ***d2*** : 受信データのデータ型
  - ***root*** : 収集したデータを格納するランク
  - ***comm*** : コミュニケータ

Nがプロセス数で割り切れない場合にデータを集めて表示する例（C/C++）

```
/* Allocate an array for MPI_Gather */
if (myid == 0)
  worka = (double *)malloc(N * sizeof(double));
/* Prepare tables and gatherlen for MPI_Gatherv */
elems = (int *)malloc(procs*sizeof(int));
displs = (int *)malloc(procs*sizeof(int));
for (i = 0; i < modN; i++)
  elems[i] = divN + 1;
for (i = modN; i < procs; i++)
  elems[i] = divN;
displs[0] = 0;
for (i = 1; i < procs; i++)
  displs[i] = displs[i-1] + elems[i-1];
if (myid < modN)
  gatherlen = divN + 1;
else
  gatherlen = divN;
```

```
/* Gather data into rank 0 and print */
if (modN == 0)
  MPI_Gather(&(a[starta]), divN, MPI_DOUBLE, worka, divN, MPI_DOUBLE, 0, MPI_COMM_WORLD);
else
  MPI_Gatherv(&(a[starta]), gatherlen, MPI_DOUBLE, worka, elems, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (myid == 0) {
  printf("Step %2d: ", j);
  for (i = 0; i < N; i++)
     printf(" %6.2f", worka[i]);
  printf("¥n");
}
```

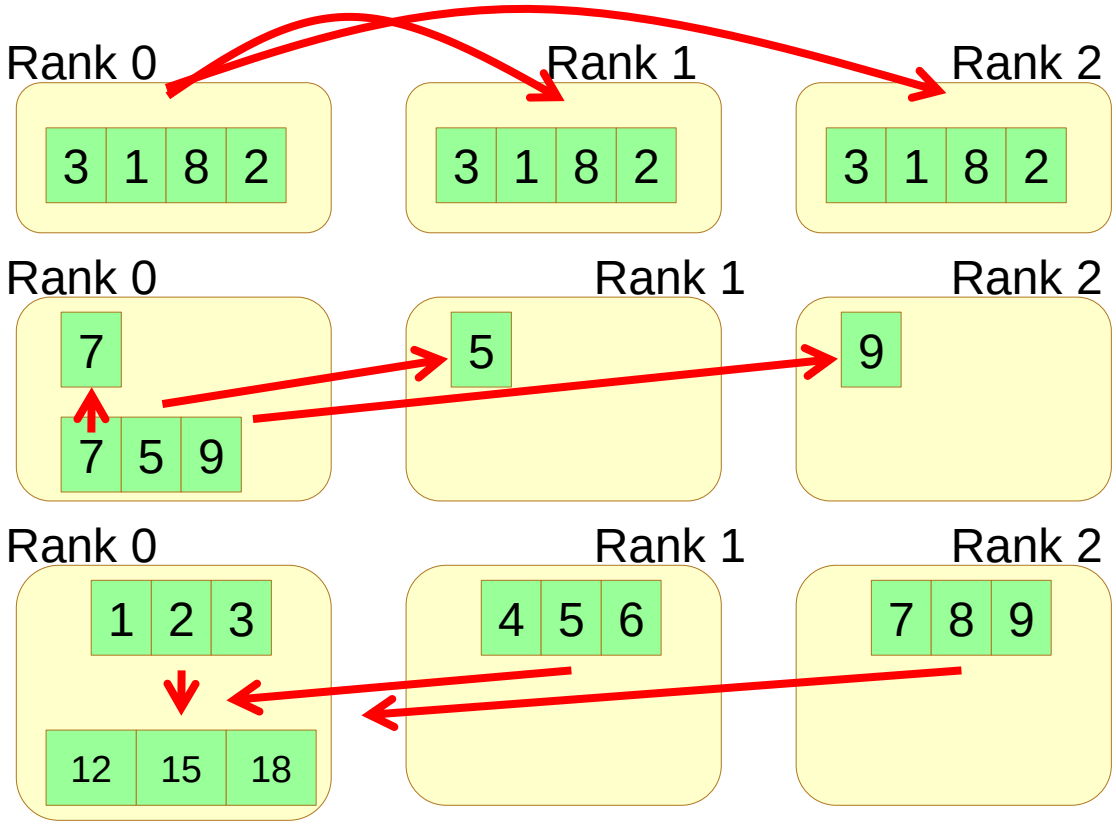
Nがプロセス数で割り切れない場合にデータを集めて表示する例（Fortran）

```
! Allocate an array for MPI_Gather
if (myid == 0) then
   allocate(worka(0:n-1))
end if
! Prepare tables and gatherlen for MPI_Gatherv
allocate(elems(0:procs-1))
allocate(displs(0:procs-1))
do i = 0, modN-1
   elems(i) = divN + 1
end do
do i = modN, procs - 1
   elems(i) = divN
end do
displs(0) = 0
do i = 1, procs - 1
   displs(i) = displs(i-1) + elems(i-1)
end do
if (myid < modN) then
   gatherlen = divN + 1
else
   gatherlen = divN
end if
```

```
!Gather data into rank 0 and print
if (modN == 0) then
  call MPI_Gather(a(starta), divN, MPI_DOUBLE_PRECISION, worka, divN, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
else
  call MPI_Gatherv(a(starta), divN, MPI_DOUBLE_PRECISION, worka, elems, displs, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
end if
if (myid == 0) then
  write(*, '(a5,i2,a1)', advance='no') "Step ", j, ":"
  do i = 0, n-1
    write(*,'(f7.2)',advance='no') worka(i)
  end do
  write(*,*)
end if
```

その他の集団通信の例

- MPI_Bcast
  - 全プロセスにコピー
- MPI_Scatter
  - 全プロセスに分散
- MPI_Reduce
  - 全プロセスの値を集約



- MPI_Allgather, MPI_Allreduce
  - MPI_Gather, MPI_Reduceの結果を全プロセスにコピー

集団通信の利用に当たって

- 同じ関数を全プロセスが実行するよう, 記述する
  - 例えば MPI_Bcastは，root rankが送信するデータを他のランクが受信する
  - このように集団通信は全プロセスが参加する
- 送信データと受信データの場所を別々に指定するタイプの集団通信では, 送信データの範囲と受信データの範囲が重ならないように指定する
  - MPI_Gather，MPI_Allgather, MPI_Gatherv, MPI_Allgatherv, MPI_Recude, MPI_Allreduce, MPI_Alltoall, MPI_Alltoallv, etc.

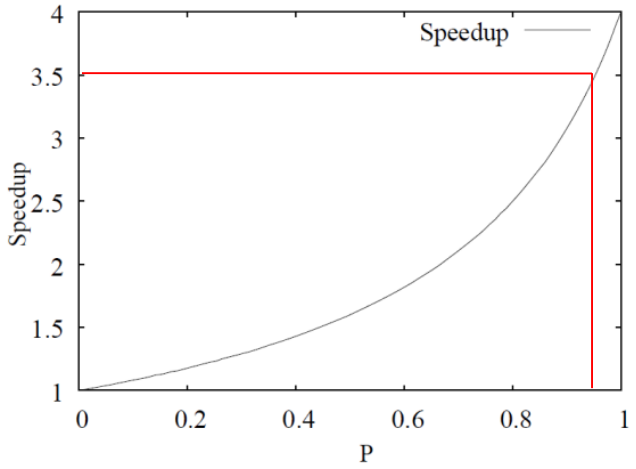
並列処理に対する期待と現実

- プログラマ:
  「ＣＰＵを 4台使うんだから，並列化で 4倍速くなって欲しい」
- 計算機製作者:
  「ＣＰＵ 4台で 3倍くらい速くなれば十分だろう」
- Why?
  - アムダールの法則
  - 負荷のバランス
  - 通信のコスト

アムダールの法則

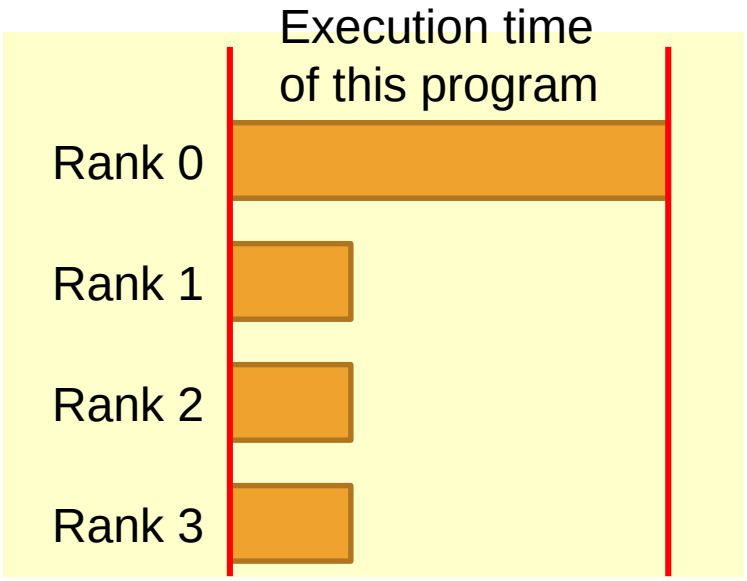
- プログラム中の高速化した部分しか高速化されない
- 並列化にあてはめて考えると：
  並列化による性能向上率の理論的な限界
  ＝1/((1−P)+P/N)
  - P: プログラム中の並列化対象部分が全処理時間に占める割合
  - N: プロセス数

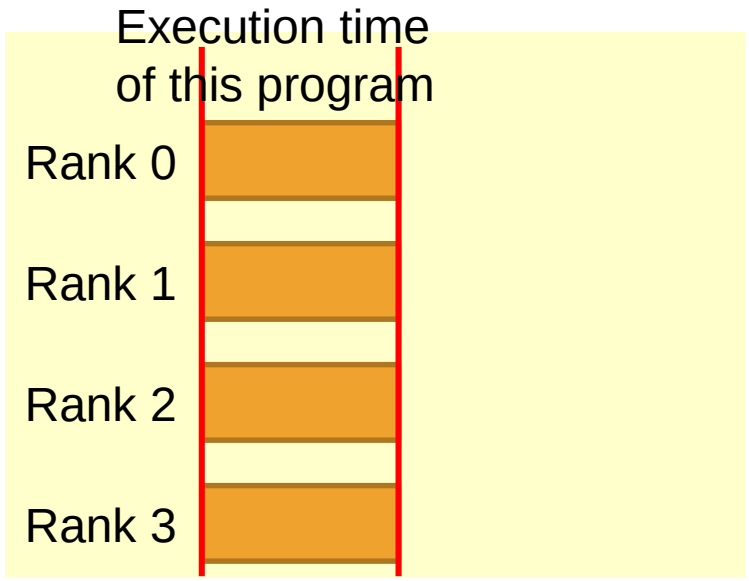
  - Example) N=4 で 3.5倍以上高速化するためには 95%以上の部分の並列化が必要



負荷のバランス

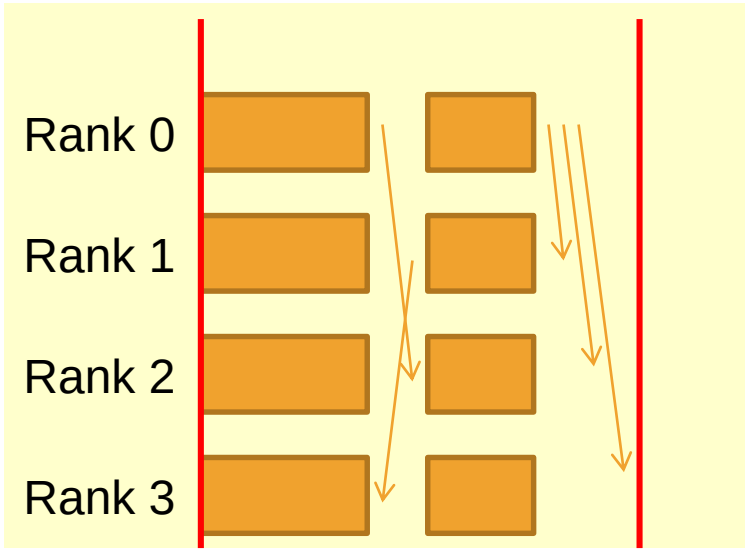
- 並列プログラムの処理時間は
  「最も遅いプロセスの処理時間」
  である





通信時間

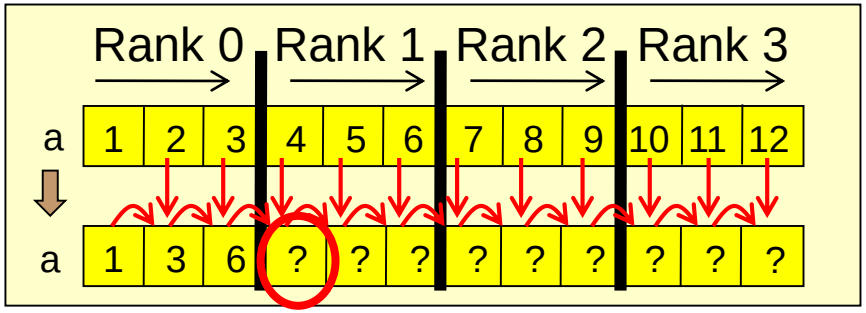
- 並列化前は不要だった時間
  ＝　並列化によるオーバーヘッド



並列化が難しいプログラム

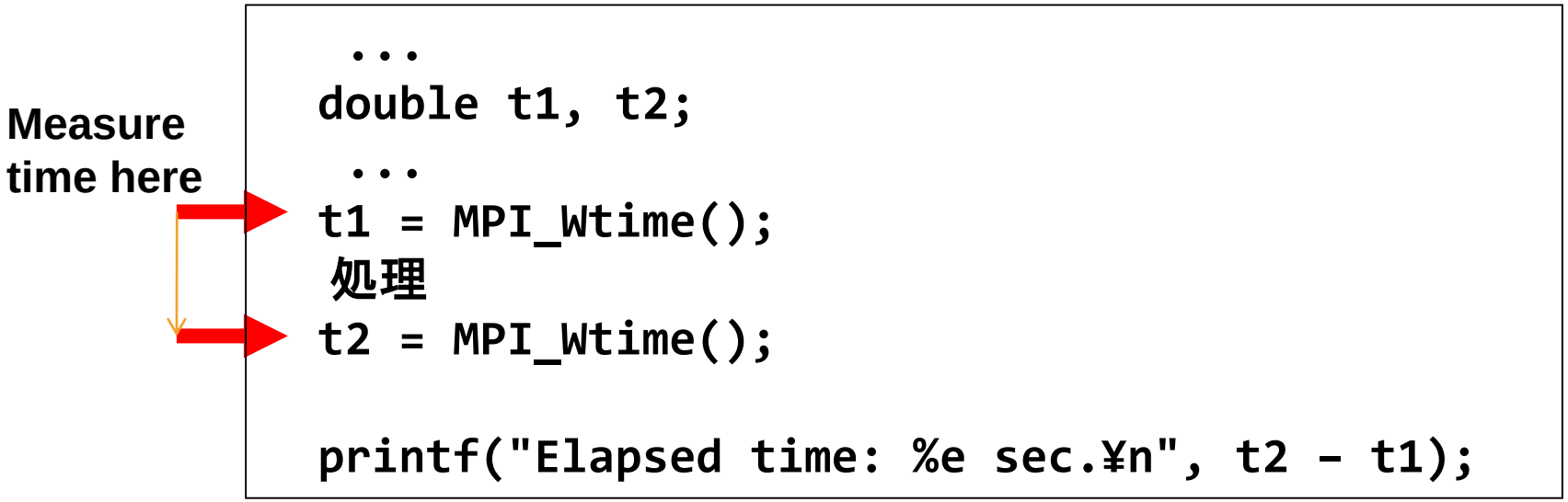
- 実行の順序によって結果が変わる
  - Example:
    前に計算した値を参照して計算

```
for (i = 1; i < 12; i++)
    a[i] = a[i] + a[i-1];
```



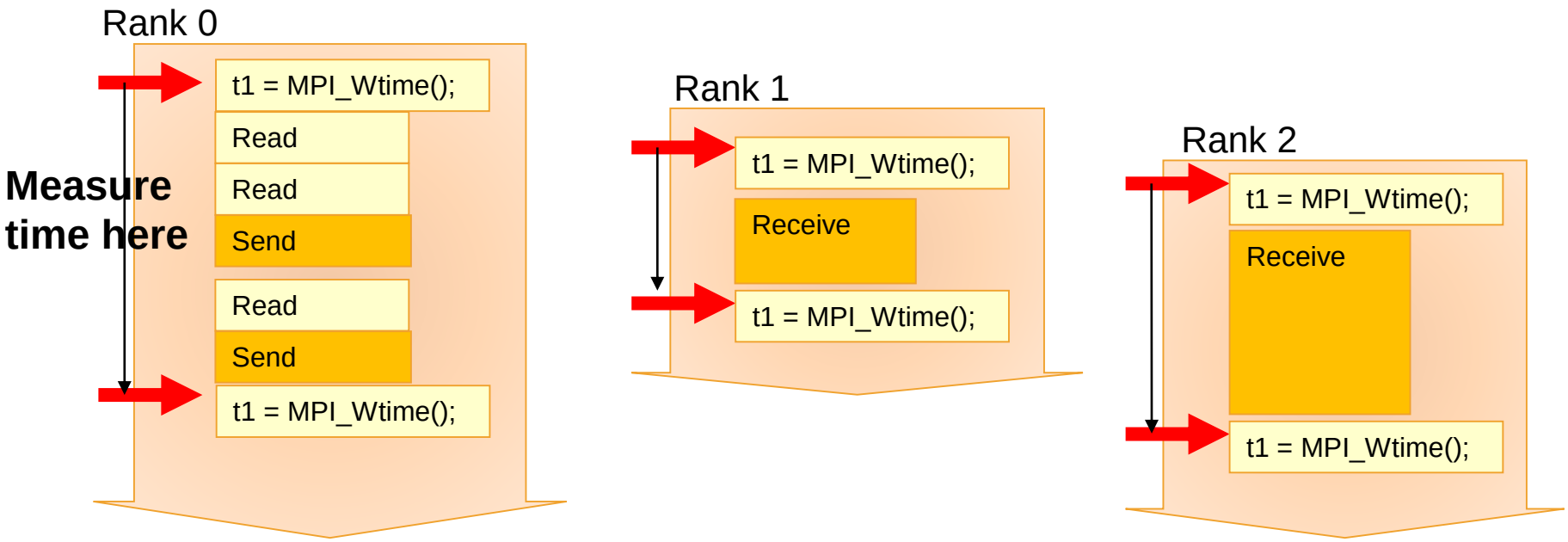
MPIプログラムの時間計測

- MPI_Wtime
  - 現在時間（秒）を実数で返す関数 Returns the current time in seconds.
  - Example)



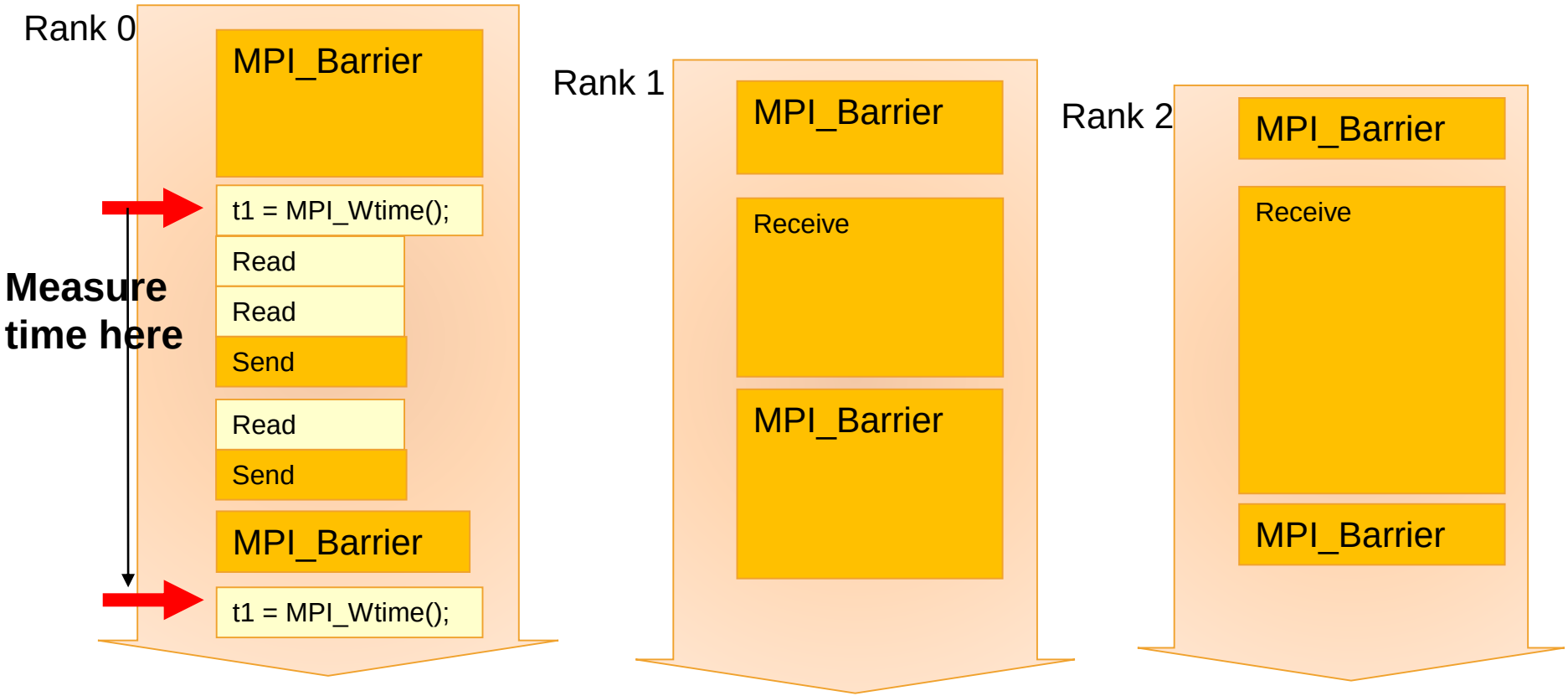
並列プログラムにおける時間計測の問題

- プロセス毎に違う時間を測定：
  どの時間が本当の所要時間か?



集団通信 MPI_Barrierを使った解決策

- 時間計測前にMPI_Barrierで同期



実習4 性能計測

- 配列サイズを大きくして時間計測関数を追加してある ex4.c もしくは ex4.f90をコンパイル後, ジョブ投入

```
mpiicc ex4.c -o ex4
pjsub ex4.sh
```

```
mpiifort ex4.f90 -o ex4
pjsub ex4.sh
```

- 結果の確認

```
pjstat
ls
cat ex4.sh.oジョブ番号
```

MPI + OpenMP

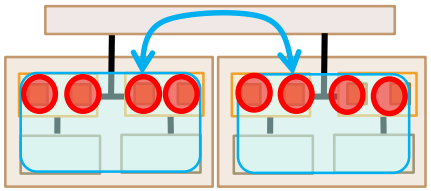
- MPIの各プロセスを, OpenMPのスレッドで並列化可能
  - 並列リージョン内での MPI関数利用は要注意

```
 /* Exchange values */
 if (myid > 0) { /* left */
   MPI_Isend(左にlocala[1] を送信開始);
   MPI_Irecv(左からlocala[0] に受信開始);
 }
 if (myid < procs-1) {/* right */
   MPI_Isend(右にlocala[loopend]を送信開始);
   MPI_Irecv(右からlocala[loopend+1]に受信開始);
 }
 /* Modify loop to calculate internal area only */
#pragma omp parallel for
 for (i = 2; i <= loopend-1; i++)
   localnewa[i] =
    (locala[i-1]+locala[i]+locala[i+1])/3.0;

 /* Wait for all non-blockings */
 MPI_Waitall(全ての非ブロッキング通信);

 /* Calculate edges */
 localnewa[1] =
  (locala[0]+locala[1]+locala[2])/3.0;
 localnewa[loopend] = (locala[loopend-1]
   +locala[loopend]+locala[loopend+1])/3.0;
```

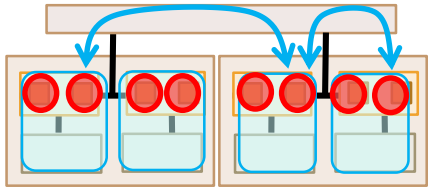
```
! Exchange values
if (myid > 0) then
  call MPI_Isend(左にa(1)を送信開始)
  call MPI_Irecv(左からa(0)に受信開始)
end if
if (myid < procs - 1) then
  call MPI_Isend(右にa(loopend)を送信開始)
  call MPI_Irecv(右からa(loopend+1を受信開始)
end if

! Modify loop to calculate internal area only
!$omp parallel do
do i = 2, loopend-1
  newa(i) = (a(i-1) + a(i) + a(i+1)) / 3.0
end do

! Wait for all non-blocking communications
call MPI_Waitall(全ての非ブロッキング通信)

! Calculate edges
newa(1) = (a(0) + a(1) + a(2)) / 3.0
newa(loopend) = (a(loopend-1) + a(loopend) + &
                 a(loopend+1)) / 3.0
```

MPI + OpenMP の実行

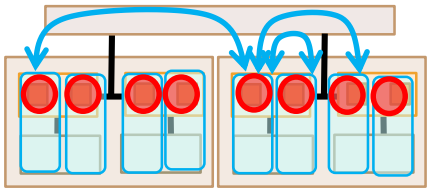
- プロセス数とスレッド数を調整



2プロセス x 4スレッド



4プロセス x 2スレッド



8プロセス
（OpenMP無し）

  - 実行例）４プロセス x ２スレッド

```
export OMP_NUM_THREADS=2
mpiexec -np 4 ./test
```

- 並列リージョンで MPI関数を呼ぶプログラム
  - 使用しているMPIライブラリが対応しているか, 管理者に確認
  - MPI_Initを MPI_Init_threadに変更

```
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &p);
if (p != MPI_THREAD_MULTIPLE)
  printf("Error¥n");
```

MPIの参考資料

- 片桐 孝洋「スパコンプログラミング入門: 並列処理とMPIの学習」, 東京大学出版会, 2013
- P. Pacheco 「MPI並列プログラミング」, 培風館, 2001
- M. J. Quinn "Parallel Programming in C with MPI and OpenMP", McGraw-Hill, 2003