(財)計算科学振興財団、大学院GP「大学連合による計算科学の最先端人材育成」

第1回　社会人向けスパコン実践セミナー　資料

# MPI入門

2009年2月17日　13:15～14:45

九州大学情報基盤研究開発センター

南里　豪志

# 講義の流れ

- **並列プログラムの概要**
  - 通常のプログラムと並列プログラムの違い
  - 並列プログラム作成手段と並列計算機の構造

- **OpenMPによる並列プログラム作成**
  - 処理を複数コアに分割して並列実行する方法

- **MPIによる並列プログラム作成 （午後）**
  - プロセス間通信による並列処理
  - 処理の分割 ＋ データの配置 ＋ 通信

# MPI (Message Passing Interface)

- **並列プログラム作成法(プログラミングモデル)のひとつ**
  - C言語やFortranプログラムから呼び出す通信用ルーチンやプロセス番号問い合わせルーチン等を使って、並列プログラムを記述する。
    - 実はMPIは特定のソフトウェアの名前ではなく、各ルーチンを定義した規格名。
  - ほとんどの並列計算機で MPIライブラリを利用可能
    - MPIライブラリ　＝　MPI規格に準拠して作成されたMPIルーチン群
- **プロセス並列によるプログラミングモデル**
  - メモリを共有しないので、必要に応じてプロセス間で通信(Message Passing)を行う。
- **ともかく、MPIによる並列プログラムの例を見てみましょう**

```c
#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での並列処理の準備

自分のランク(プロセス番号)を取得

全体のプロセス数を取得

もし自分のランクが 0であれば

まず、自分用のデータを入力して myval に格納

i = 1～procs－1 について

データを一つ入力して valに格納し

MPI_Sendでランク i に val の値を送信

ランク0以外のプロセスは
MPI_Recvでランク 0から値を受信して myval に格納

各プロセスが、自分の myvalを表示

並列実行の終了処理

```
program test1
  implicit none
  include 'mpif.h'
  integer :: myid, procs, ierr, i
  integer, dimension(MPI_STATUS_SIZE) :: status
  real(8) :: myval, val

  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)
  end if

  print *, 'PROCS: ', procs,'MYID: ', myid, 'MYVAL: ', myval
  call MPI_Finalize(ierr)
  stop
end program
```

MPIでの並列処理の準備

ランク(自分のプロセス番号)を取得

全体のプロセス数を取得

もし自分のランクが 0であれば

まず、自分用のデータを入力して myval に格納

その後、ランク 1 ～ ランク procs－1 について

データを入力して valに格納し

MPI_Sendでランク i に val の値を送信

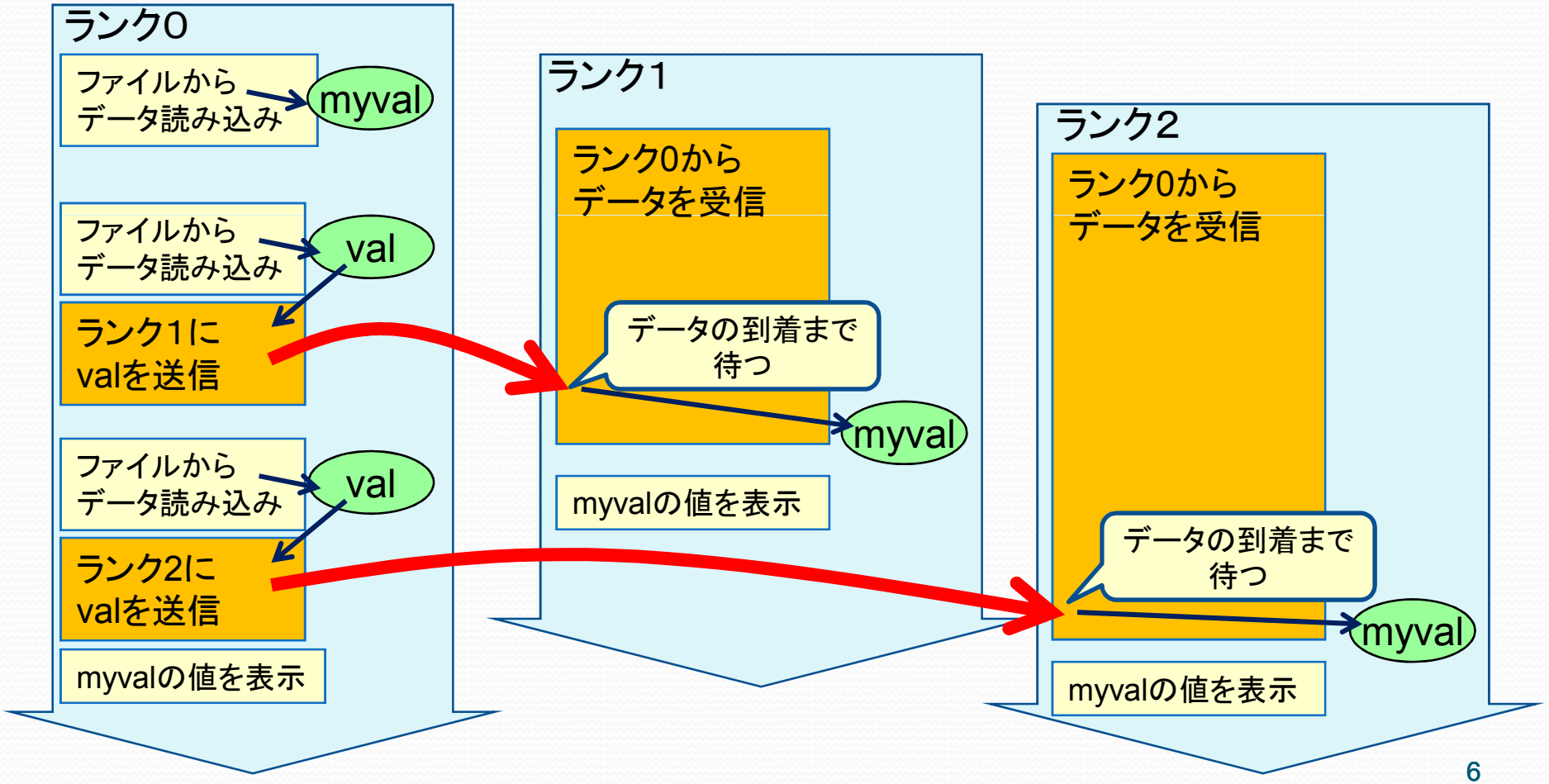
ランク0以外のプロセスは
MPI_Recvでランク 0から値を受信して myval に格納

並列実行の終了処理

各プロセスが、自分の myvalを表示

# プログラム例の実行の流れ

- それぞれのプロセスが自分に割り当てられた仕事を実行



# 実行例

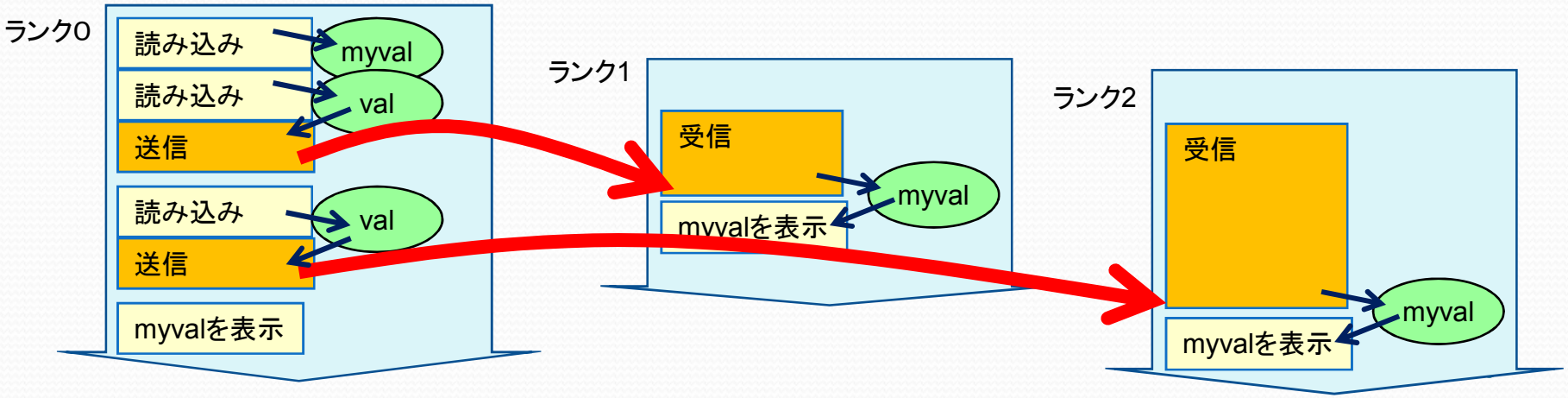
- 各プロセスがそれぞれ勝手に表示するので、表示の順番は毎回変わる可能性がある。

```
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
```

プロセス 1の myval
プロセス 2の myval
プロセス 0の myval
プロセス 3の myval

# MPIプログラムの特徴

- **全プロセスが同じプログラムを実行**
  - OpenMPと同じ SPMD(Single Program Multiple Data)型のモデル
  - 処理の割り当てには、プロセスの番号（＝ランク）を利用
    - プログラム例では 0番はデータ入力と送信、1～3番は受信
- **MPIは新しいプログラミング言語ではなく、通信等のためのルーチン群。**
  - 新しく文法を覚えなくてもいい
- **他のプロセスの変数を直接見ることはできない。**
  - 必要に応じて通信をする。



# MPIによるプログラム並列化の手順

0. 並列化するかどうかを吟味
   －OpenMPと同様だが、並列化に要する作業時間が長いので、気楽には試せない

1. 並列化の対象部分を選択
   －OpenMPと同様、処理に時間を要する部分を優先して選択

2. データのプロセスへの配置方法を選択

3. データ配置に応じてプログラムを書き換え
   －MPIの基本ルーチン追加

4. 必要に応じて通信ルーチン追加

5. 動作確認とデバッグ
   －プログラムが複雑なので、デバッグに要する時間も大

# MPIプログラムの構成

```
#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;
}
```

ヘッダファイル

基本的なルーチン

通信ルーチン

# 主なMPIルーチン

- **基本的なルーチン（環境管理、問い合わせ）**
  - どのMPIプログラムにも必ず必要なルーチン

    **MPI_Init（初期化）, MPI_Finalize（終了）**
  - ほとんどのMPIプログラムで利用

    **MPI_Comm_size（プロセス数取得）, MPI_Comm_rank（プロセス番号取得）**
  - その他

    **MPI_Wtime（経過時間計測）, ...**

- **通信ルーチン**
  - 一対一通信ルーチン: 送信プロセスと受信プロセスの間で通信

    **MPI_Send, MPI_Isend, MPI_Ssend（送信）,**
    **MPI_Recv, MPI_Irecv（受信）, MPI_Wait（処理待ち）, ...**

  - 集団通信ルーチン: 全プロセスで一斉に行う通信

    **MPI_Bcast（データコピー）, MPI_Reduce（データ集約）,**
    **MPI_Gather（データ収集）, ...**

# 初期化
## MPI_Init

- MPIの並列処理開始処理
  - プロセスの起動やプロセス間通信路の確立等。
  - 他のMPIルーチンを呼ぶ前に、
    必ずこのルーチンを呼ぶ。

C, C++:
```
int MPI_Init(int *argc, char **argv);
```
Fortran:
```
subroutine MPI_Init(ierr)
```

- C, C++ の場合の引数:
  - 引数に main関数の2つの引数へのポインタを渡す。
    - 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 の場合の引数:
  - 引数にエラーコード格納用の整数変数を指定する。
- Fortranの場合、ほとんどのMPIルーチンで
  引数の最後にエラーコード格納用の変数を
  指定する。

プログラム例

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
   int myid, procs, ierr;
   double myval, val;
  MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   MPI_Comm_size(MPI_COMM_WORLD, &procs);

   ...
```

```
program test1
  implicit none
  include 'mpif.h'
  integer :: myid, procs, ierr, i
  integer, dimension(MPI_STATUS_SIZE) :: status
  real(8) :: myval, val

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

# 終了処理
# MPI_Finalize

- **並列処理の終了**
  - 確立した通信路の切断や、確保した作業領域の解放等
  - このルーチン実行後はMPIルーチンを呼び出せない
- **Fortranの場合の引数**
  - エラーコード格納用変数

C, C++:
```
int MPI_Finalize();
```
Fortran:
```
subroutine MPI_Finalize(ierr)
```

プログラム例

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

```
  ...
  print *, 'PROCS: ', procs,'MYID: ', myid, 'MYVAL: ', myval
  call MPI_Finalize(ierr)
  stop
end program
```

- **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させる。**
  - そうしないと、一部のプロセスだけが先に終了してしまうため、エラーになる。

# プロセス番号(ランク)の取得
## MPI_Comm_rank

- そのプロセスのランクを取得する
  - 引数:
    **コミュニケータ**,
    ランクを格納する変数
    (C言語の場合はポインタ)

C, C++:
```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```
Fortran:
```
subroutine MPI_Comm_rank(comm, rank, ierr)
```

- ランク: プロセスを識別するための番号

- コミュニケータ: プロセスのグループを表す識別子
  - 例えば、プロセスを半分に分けて、それぞれ別のことをやらせる、という時に、グループに分けると便利。
    - 今回の講義ではグループ分けについては扱わない
  - 通常は, MPI_COMM_WORLD を指定
    - MPI_COMM_WORLD: 全プロセス

プログラム例

```
...
int myid, procs, ierr;
...
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &procs);

if (myid == 0){
 ...
```

```
...
integer :: myid, procs, ierr, i

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

# プロセス数の取得 MPI_Comm_size

- **そのコミュニケータに含まれるプロセスの数を取得する**
  - 引数:
    コミュニケータ,
    プロセス数を格納する変数
    (C言語の場合はポインタ)

C, C++:

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

Fortran:

```
subroutine MPI_Comm_size(comm, size, ierr)
```

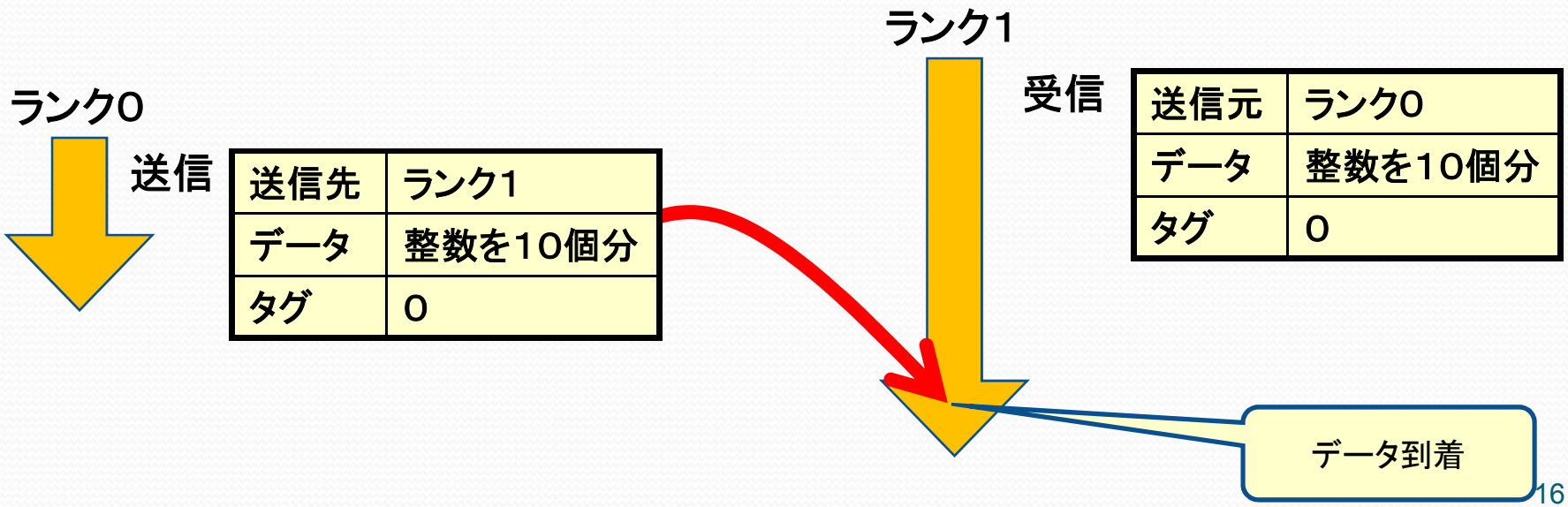
プログラム例

```
...
int myid, procs, ierr;
...
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &procs);

if (myid == 0){
 ...
```

```
...
integer :: myid, procs, ierr, i

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

# MPIにおける通信：一対一通信

- 一対一通信： 送信プロセスと受信プロセスの間で行われる通信
- 送信プロセスでの送信ルーチンと
  受信プロセスでの受信ルーチンが、
  それぞれ”適切”に呼び出されると、通信が行われる。
  - 送信元と送信先のランクが正しく設定されていて、
  - 送信側と受信側でデータの大きさが等しく、
  - 送信側と受信側でデータに付けられた番号（タグ）が等しい



# 送信
## MPI_Send

- **送信内容の指定**
  - 引数:
    送信データの場所(アドレス),
    送信データの数,
    送信データの型,
    送信先のランク,
    タグ(通常は0),
    コミュニケータ
    (通常は MPI_COMM_WORLD)
- **主なデータ型:**

| | C, C++ | 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 |
| 文字 | MPI_CHAR | MPI_CHARACTER |

- **タグ: メッセージに付ける番号(整数)**
  - 不特定のプロセスから届く通信を処理するタイプのプログラムで使用
  - 通常は、0 を指定しておいて良い

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

```
...
if (myid == 0){
  printf("value for proc 0: ");
  scanf("%f", &myval);
  for (i = 1; i < procs; i++){
    printf("value for proc %d: ", i);
    scanf("%f", &val);
    MPI_Send(&val, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
  }
...
```

```
...
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
...
```

# MPI_Sendの利用例

- 整数変数 d の値を送信(整数1個)

```
MPI_Send(&d, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

```
call MPI_Send(d, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, &
              ierr)
```

- 実数配列 mat の最初の要素から100番目の要素までを送信

```
MPI_Send(mat, 100, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
```

```
call MPI_Send(mat, 100, MPI_DOUBLE_PRECISION, 1, 0, &
              MPI_COMM_WORLD, ierr)
```

- 整数配列 data の10番目の要素から50個を送信

```
MPI_Send(&(data[9]), 50, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

```
call MPI_Send(data[10], 50, MPI_INTEGER, 1, 0, &
              MPI_COMM_WORLD, ierr)
```

# 受信
## MPI_Recv

- **受信内容の指定**
  - 引数:
    受信データを格納するアドレス,
    受信データの数,
    受信データの型,
    送信元のランク,
    タグ(通常は0),
    コミュニケータ
    (通常は MPI_COMM_WORLD),
    ステータス

C, C++:

```
int MPI_Recv(void *b, int c, MPI_Datatype d, int src,
             int t, MPI_Comm  comm, MPI_Status *st);
```

Fortran:

```
subroutine MPI_Recv(b, c, d, dest, t, comm, st, ierr)
```

```
...
} else
  MPI_Recv(&myval, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
...
```

```
...
integer, dimension(MPI_STATUS_SIZE) :: status
...
else
  call MPI_Recv(myval, 1, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD, status)
end if
...
```

- **ステータス st : メッセージの情報を格納する整数配列**
  - Fortran では以下のように整数配列として宣言する
    `integer, dimension(MPI_STATUS_SIZE) :: st`
  - 送信元ランクやタグの値を参照可能(通常は、あまり使わない)

# ノンブロッキング通信
## MPI_Isend, MPI_Irecv

- **ノンブロッキング： 完了を待たずに次の処理に移る**
  - 複数の処理を並行的に行う

- **ノンブロッキング送信 MPI_Isend**
  - 送信されるデータが送信プロセスから送出されるまで待たずに次の処理を実行する
  - 大きなデータを送信する場合に有効 （かも）

- **ノンブロッキング受信 MPI_Irecv**
  - 受信するデータがまだ届いてなくても、待たずに次の処理を実行する
  - 大きなデータを受信する場合に有効 （かも）

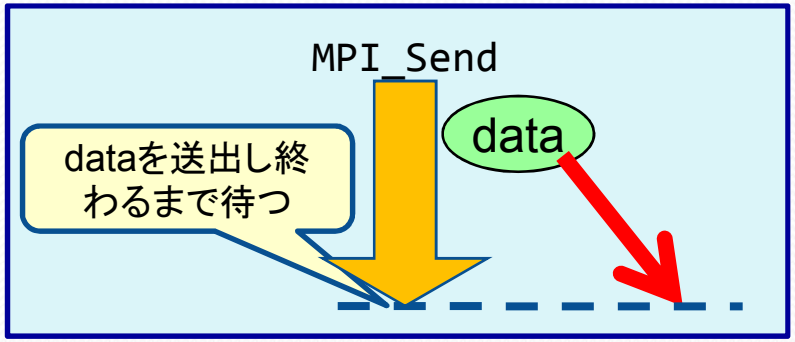
- **ノンブロッキング送信、受信の完了待ち MPI_Wait, MPI_Waitall**
  - MPI_Isend や MPI_Irecv で指示した送信、受信の完了を待つ。

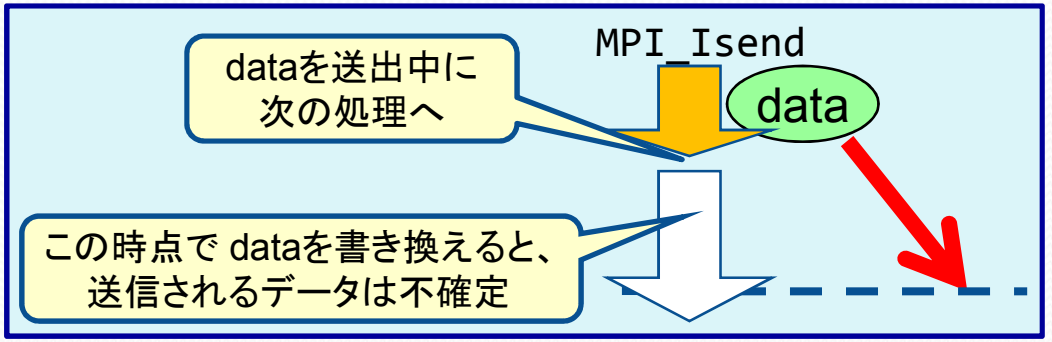
C, C++:
```
int MPI_Isend(void *b, int c, MPI_Datatype d, int dest,
              int t, MPI_Comm  comm, MPI_Request *r);
```
Fortran:
```
subroutine MPI_Isend(b, c, d, dest, t, comm, r, ierr)
```

# MPI_Send と MPI_Isendの違い

- **MPI_Sendは、送信データを書き換えても良い状態になるまで待つ。**
  - ネットワークにデータを送出し終わるか、一時的にデータのコピーを作成するまで。

- **MPI_Isendは、待たない。**
  **＝ MPI_Isendの直後に送信対象データを書き換えた場合、**
  **書き換え前の値と書き換え後の値のどちらが送信されるか分からない**




- **大きなデータの送信を行う場合や連続して通信を行う場合は、MPI_Isendの方が速いかもしれない。**
  - データの送出処理を行っている間に別の処理を行える。
  - MPI_Waitを実行するまでは送信データを書き換えないように注意。

C, C++:

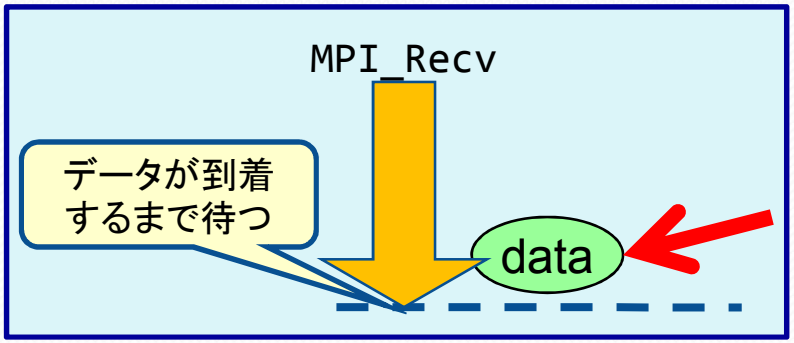
```
int MPI_Irecv(void *b, int c, MPI_Datatype d, int src,
              int t, MPI_Comm  comm, MPI_Request *r);
```

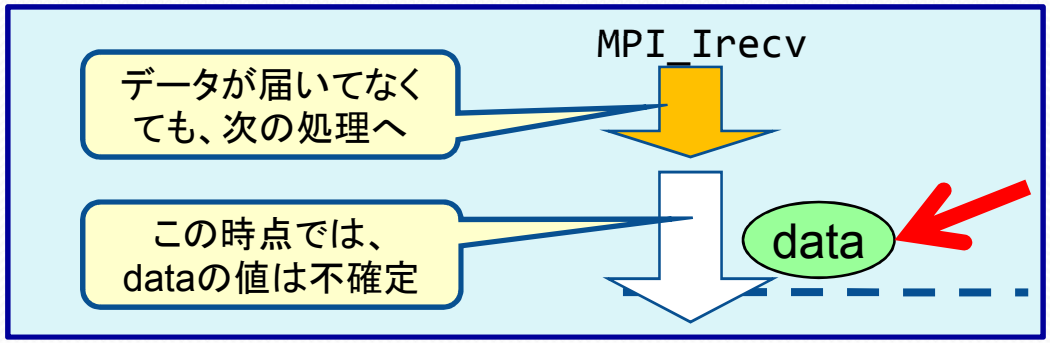
Fortran:

```
subroutine MPI_Irecv(b, c, d, dest, t, comm, r, ierr)
```

# MPI_Recv と MPI_Irecvの違い

- **MPI_Recvは、データが到着するまで待つ。**
- **MPI_Irecvは、待たない。**
  **＝ MPI_Irecvの直後に受信データを参照しても、正しい値かどうか分からない。**

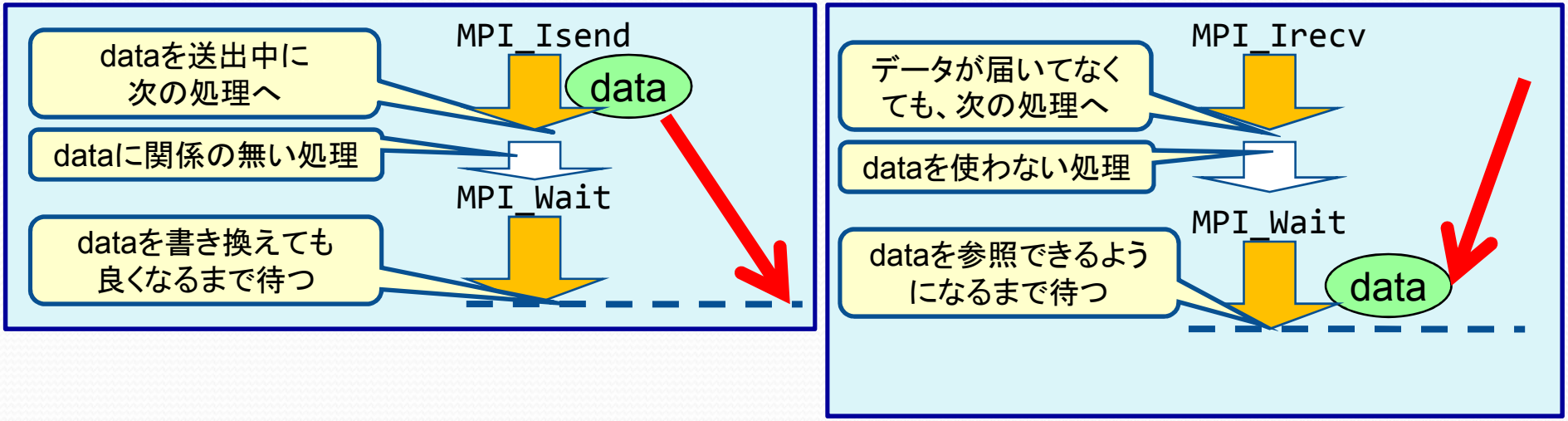



- **大きなデータの受信を行う場合や連続して通信を行う場合は、MPI_Irecvの方が速いかもしれない。**
  - データの受信処理を行っている間に別の処理を行える。
  - MPI_Waitを実行するまでは受信データを参照しないように注意。

# MPI_Wait, MPI_Waitall

- ノンブロッキング通信（MPI_Isend、 MPI_Irecv）の完了を待つ。
  ＝　送信データを書き換えたり受信データを参照したり出来るようになる
  - MPI_Isend，MPI_Irecvを使った後に必ず実行。



# ノンブロッキング通信を使った例

```
...

double myval;
double *val;
MPI_Request *req;
MPI_Status status, *st;
...

if (myid == 0){
   val = (double *)malloc((procs-1) * sizeof(double));
   req = (MPI_Request *) malloc((procs-1) * sizeof(MPI_Request));
   st = (MPI_Status *)malloc((procs-1) * sizeof(MPI_Status));
    ...
   for (i = 1; i < procs; i++){
      fscanf("%lf", &(val[i-1]));
      MPI_Isend(&(val[i-1]), 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &(req[i-1]));
   }
   fclose(fp);
   MPI_Waitall(procs-1, req, st);
} else
   MPI_Recv(&myval, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD &status);
 ...
```

# ノンブロッキング通信を使った例

```
...
integer, dimension(MPI_STATUS_SIZE) :: st1
integer, dimension(:,:), allocatable :: st2
integer, dimension(:), allocatable :: req
...

if (myid == 0) then
  allocate(req(procs-1))
  allocate(st2(MPI_STATUS_SIZE, procs-1))
  open(10, file='test.dat')
  read(10, *) myval
  do i = 1, procs-1
    read(10, *) val
    call MPI_Isend(val, 1, MPI_DOUBLE_PRECISION, i, 0, &
                   MPI_COMM_WORLD, req(i), ierr)
  end do
  close(10)
  call MPI_Waitall(procs-1, req, st2, ierr)
else
  call MPI_Recv(myval, 1, MPI_DOUBLE_PRECISION, 0, 0, MPI_COMM_WORLD, st1)
end if
...
```

# 集団通信

- 全プロセスで行う通信
- 例)

**MPI_Bcast**

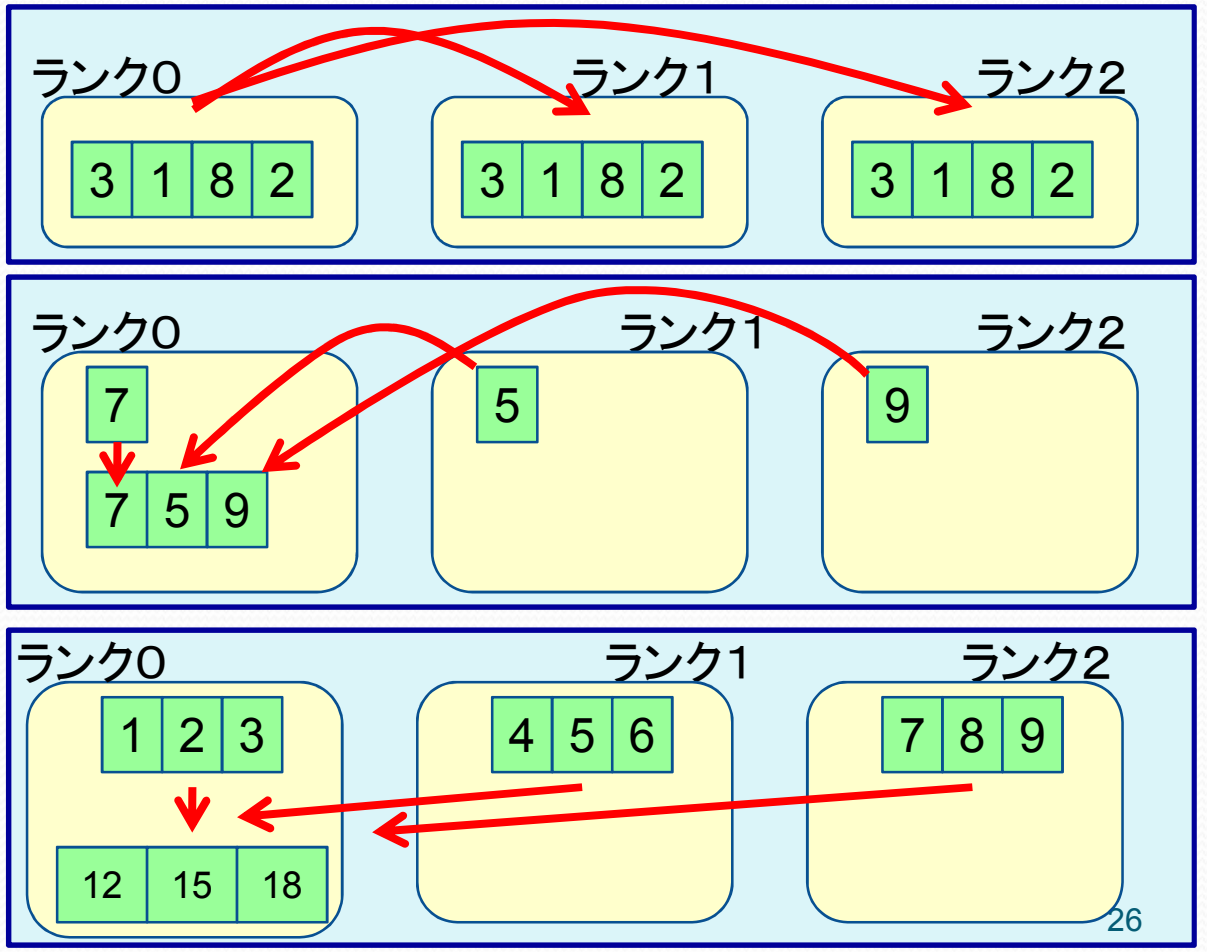
- 全プロセスにコピー

**MPI_Gather**

- 各プロセスのデータを一つの行列にとりまとめ

**MPI_Reduce**

- 各プロセスのデータを集約計算(総和、最大値、最小値等)して一つの行列にとりまとめ



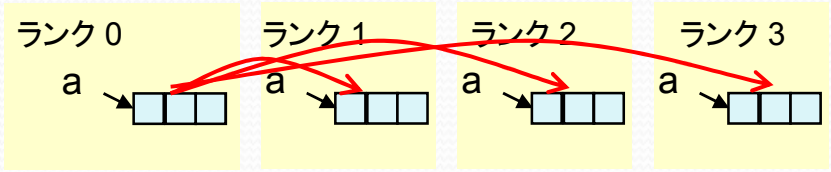
# MPI_Bcast

- 全プロセスへのデータのコピー
  - 引数:
    コピー対象のデータのアドレス,
    データの数, データの型,
    rootランク(オリジナルデータを
    持つプロセスのランク),
    コミュニケータ(通常は
    MPI_COMM_WORLD)

C, C++:
```
int MPI_Bcast(void *b, int c, MPI_Datatype d,
              int root, MPI_Comm comm);
```
Fortran:
```
subroutine MPI_Bcast(b, c, d, root, comm, ierr)
```

- 例)

```
MPI_Bcast(a, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```




  - rootランク(4番目の引数)のプロセスのデータを各プロセスにコピーする

# MPI_Gather

- 全プロセスからのデータを1つのプロセスに収集
  - 引数:
    収集元のデータのアドレス,
    データの数, データの型,
    収集先のデータのアドレス,
    データの数, データの型,
    rootランク(データを収集するプロセスのランク),
    コミュニケータ(通常はMPI_COMM_WORLD)

C, C++:

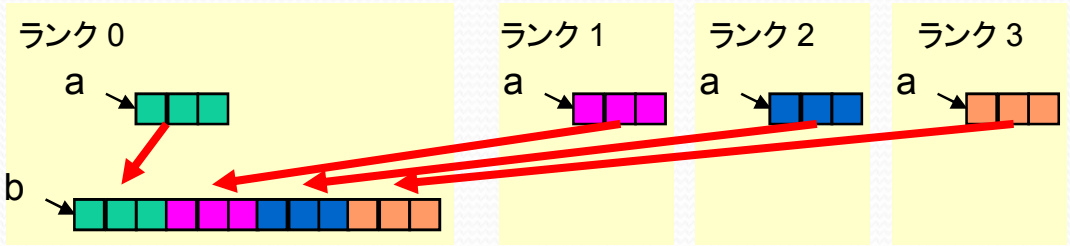
```
int MPI_Gather(void *sb, int sc MPI_Datatype st, void *rb, int rc,
               MPI_Datatype rt, int  root, MPI_Comm comm);
```

Fortran:

```
subroutine MPI_Gather(sb, sc, st, rb, rc, rt, root, comm, ierr)
```

- 例)

```
MPI_Gather(a, 3, MPI_DOUBLE, b, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```




- 各プロセスのデータを rootランク(7番目の引数)のプロセスの配列に、プロセス番号順に並べて格納する。

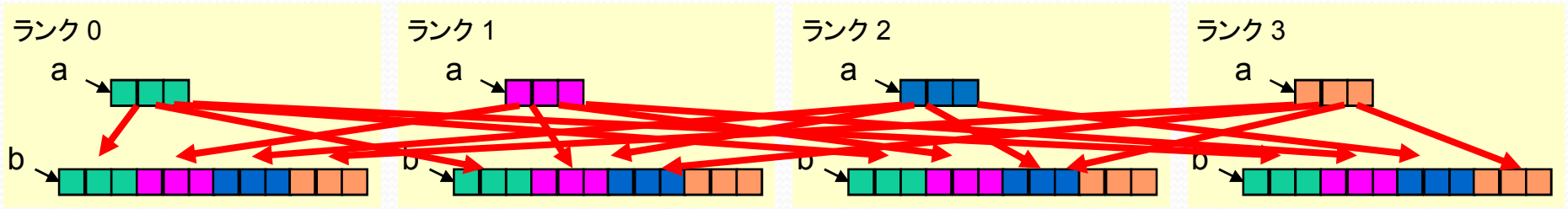
# MPI_Allgather

C, C++:
```
int MPI_Allgather(void *sb, int sc MPI_Datatype st,  void *rb,
                  int rc, MPI_Datatype rt, MPI_Comm comm);
```
Fortran:
```
subroutine MPI_Gather(sb, sc, st, rb, rc, rt, root, comm, ierr)
```

- **MPI_Gatherの結果を全プロセスにコピー**
  - 引数:
    収集元のデータのアドレス,
    データの数, データの型,
    収集先のデータのアドレス,
    データの数, データの型,
    コミュニケータ(通常は
    MPI_COMM_WORLD)

- **例)**

```
MPI_Allgather(a, 3, MPI_DOUBLE, b, 3, MPI_DOUBLE, MPI_COMM_WORLD);
```



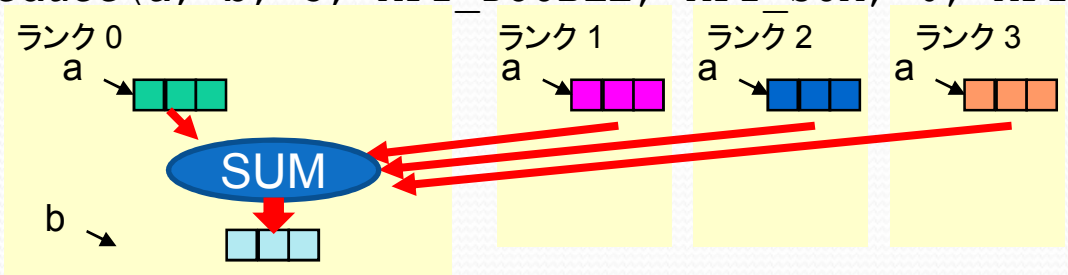

- 各プロセスのデータを各プロセスの配列に、プロセス番号順に並べて格納する。

# MPI_Reduce

C, C++:
```
int MPI_Reduce(void *sb, void *rb, int c, MPI_Datatype t,
               MPI_Op op, int root, MPI_Comm comm);
```
Fortran:
```
subroutine MPI_Reduce(sb, rb, c, t, op, root, comm, ierr)
```

- **全プロセスからのデータを集めて計算(総和等)をする**
  - 引数:
    収集元のデータのアドレス,
    計算結果を格納するアドレス,
    データの数, データの型,
    rootランク(計算結果を格納するプロセスのランク),
    コミュニケータ(通常はMPI_COMM_WORLD)
- **例)**

```
MPI_Reduce(a, b, 3, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```



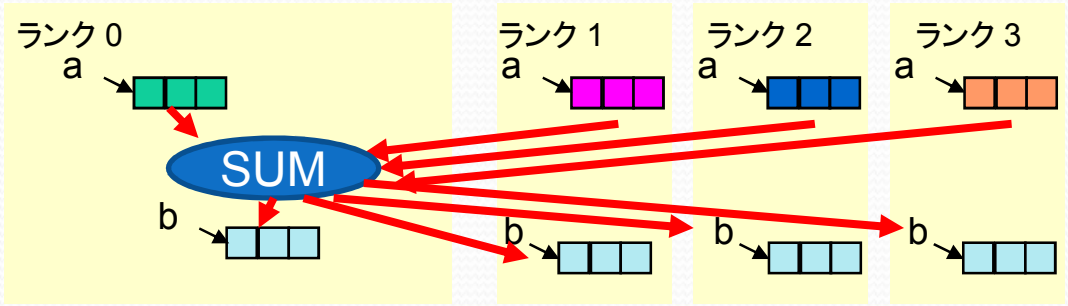

  - 各プロセスのデータを rootランク(6番目の引数)のプロセスに集め、op(5番目の引数)で指示された計算を適用する。
- **op で指示できる計算:MPI_SUM(和), MPI_MAX(最大値), MPI_MIN(最小値) 等**

# MPI_Allreduce

C, C++:
```
int MPI_Allreduce(void *sb, void *rb, int c,
                  MPI_Datatype t, MPI_Op op, MPI_Comm comm);
```
Fortran:
```
subroutine MPI_Reduce(sb, rb, c, t, op, comm, ierr)
```

- MPI_Reduceの結果を全プロセスにコピー
  - 引数:
    収集元のデータのアドレス,
    計算結果を格納するアドレス,
    データの数, データの型,
    コミュニケータ(通常はMPI_COMM_WORLD)
- 例)

```
MPI_Allreduce(a, b, 3, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```




  - 各プロセスのデータを rootランク(6番目の引数)のプロセスに集め、op(5番目の引数)で指示された計算を適用後、全プロセスにコピー。

# 集団通信の利用に当たって

- **プログラム中で必ず全プロセスが実行するよう、記述する。**
  - 特に MPI_Bcast等は送信元プロセスだけ実行するように書いてしまいがちなので注意。
    間違いの例）

```
if (myid == 0)
  MPI_Bcast(a, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- **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等

# MPIプログラムでは”デッドロック”に注意

- デッドロック： 何らかの理由で、プログラムを進行させることができなくなった状態

- MPIでデッドロックが発生しやすい場所：
  1. MPI_Recv, MPI_Wait, MPI_Waitall
     ⇒ 対応する MPI_Send等の送信が実行されなければ先に進めない。

間違いの例）

```
if (myid == 0){
  ランク1から MPI_Recv
  ランク1へ MPI_Send
}
if (myid == 1){
  ランク0から MPI_Recv
  ランク0へ MPI_Send
}
```

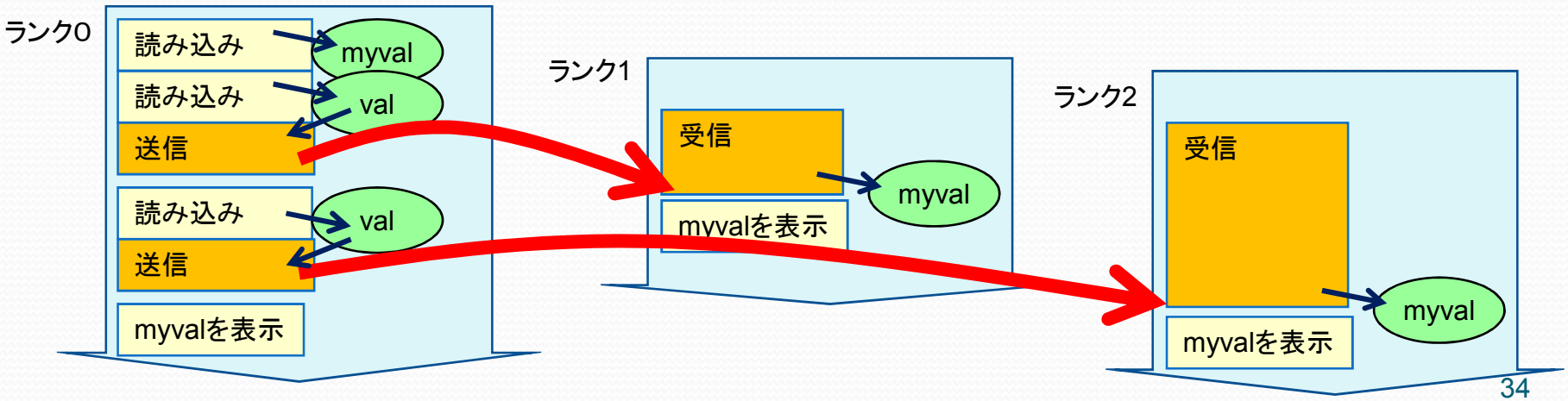
改善例）

```
if (myid == 0){
  ランク1から MPI_Irecv
  ランク1へ MPI_Send
  MPI_Wait
}
if (myid == 1){
  ランク0から MPI_Irecv
  ランク0へ MPI_Send
  MPI_Wait
}
```

  2. 集団通信ルーチン
     ⇒ 基本的に全部のプロセスが同じルーチンを実行するまで先に進めない

# ここまでのまとめ

- **MPIでは、一つのプログラムを複数のプロセスが実行する**
- **各プロセスには、そのランク（番号）に応じて仕事を割り当てる**
- **各プロセスはそれぞれ自分だけの記憶場所（メモリ）を持っている**
- **他のプロセスが持っているデータを参照するには、通信する**
- **MPIルーチンの種類**
  - MPIの環境に関するルーチン MPI_Init, MPI_Finalize, MPI_Comm_rank　等
  - 一対一通信　MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, MPI_Wait　等
  - 集団通信　MPI_Bcast, MPI_Gather, MPI_Allgather, MPI_Reduce, 等



# 演習　MPIプログラムの実行

- 演習用の計算機にログインして、以下を実行

```
$ cd test
$ cat test-mpi.c
$ mpicc test-mpi.c -o test-mpi
$ cat test-mpi.sh
$ qsub test-mpi.sh
Request 7129.pcj submitted to queue: PCL-A.
$ qstat
```

ジョブの受付番号

何度か qstat を実行して、自分が投入したジョブが消えてから

```
$ ls
```

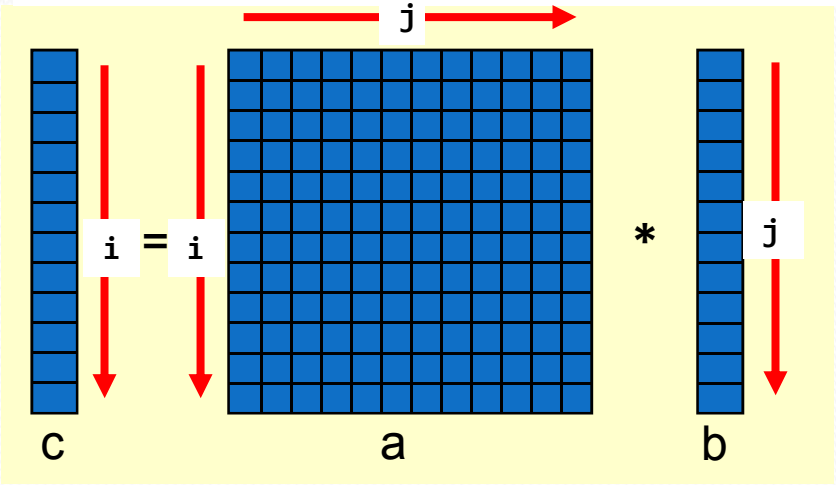
test-mpi.sh.e???? と test-mpi.sh.o???? というファイルができていることを確認(????はジョブの番号)

```
$ cat test-mpi.sh.o????
```

# プログラムの並列化事例
## 対象： 行列・ベクトル積プログラム

並列化前のプログラム

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

#define N 100

int main(int argc, char *argv[])
{
  int i, j;
  double *a, *b, *c;

  a = (double *)malloc(N*N*sizeof(double));
  b = (double *)malloc(N*sizeof(double));
  c = (double *)malloc(N*sizeof(double));

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i*N+j] = i + j;

  for (i = 0; i < N; i++){
    b[i] = i;
    c[i] = 0;
  }

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      c[i] += a[i*N+j] * b[j];

  for (i = 0; i < N; i++)
    printf("(%d: %.2f) ", i, c[i]);
  printf("¥n");

  return 0;
}
```



# ループのプロセスへの割り当て方法：ループの各繰り返しで全く別の計算を行う場合

- 単純に、ループを均等に分けるだけでよい

ランク0の仕事

```
for (i = 0; i < N/4; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

ランク1の仕事

```
for (i = N/4; i < N/2; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

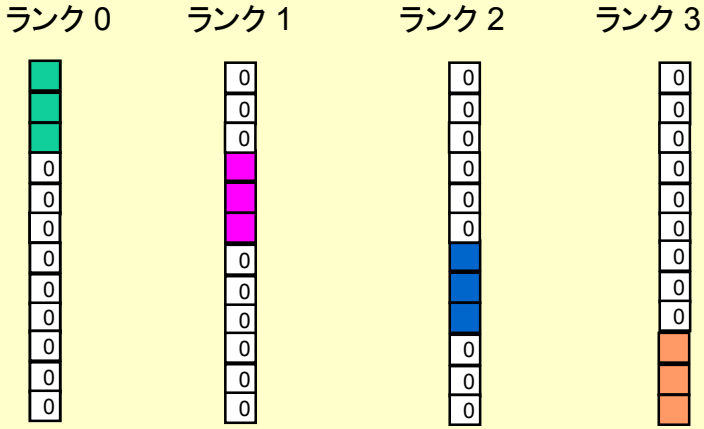
ランク2の仕事

```
for (i = N/2; i < N*3/4; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

ランク3の仕事

```
for (i = N*3/4; i < N; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

各ランクの c[i]






37

# ループのプロセスへの割り当て方法：ループ全体で総和の計算を行う場合

- まずプロセス毎に部分和を計算しておいて、最後に全プロセスの総和を計算する。

ランク0の仕事

```
for (i = 0; i < N; i++)
  for (j = 0; j < N/4; j++)
    ctmp[i] += a[i*N+j] * b[j];
```

ランク1の仕事

```
for (i = 0; i < N; i++)
  for (j = N/4; j < N/2; j++)
    ctmp[i] += a[i*N+j] * b[j];
```

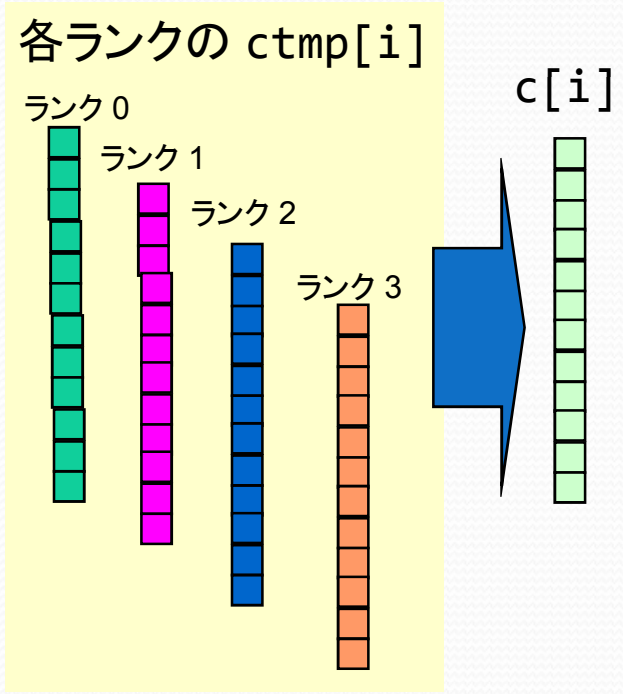
ランク2の仕事

```
for (i = 0; i < N; i++)
  for (j = N/2; j < N*3/4; j++)
    ctmp[i] += a[i*N+j] * b[j];
```

ランク3の仕事

```
for (i = 0; i < N; i++)
  for (j = N*3/4; j < N; j++)
    ctmp[i] += a[i*N+j] * b[j];
```

最後に全プロセスの ctmp[i]の総和を計算して c[i] に格納



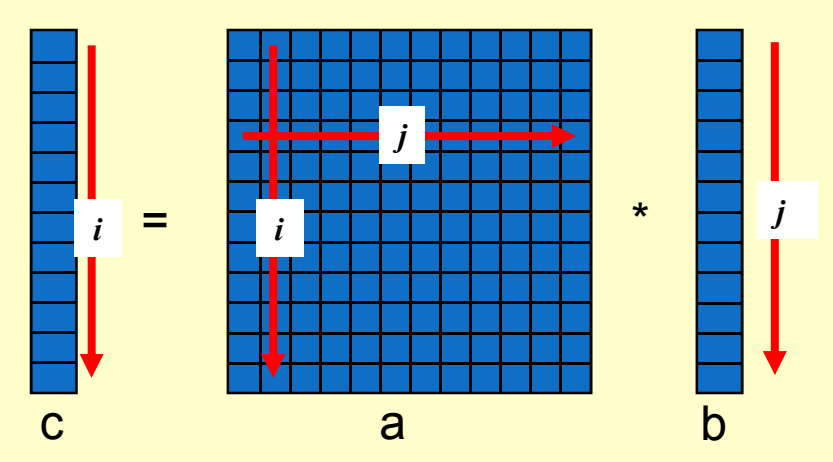
# では、どのループを並列化するか?

- 基本的な考え方:
  OpenMPの時と少し違う

  1. なるべく配列の連続した要素への参照が長く続くように

  2. なるべく他のプロセスで計算した結果への参照が少なくてすむように

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```



# どのようにループをプロセスに分担させるか

- Block分割： 連続した繰り返し毎に分割

```
for (i = (100/procs)*myid;
      i < (100/procs)*(myid+1); i++)
  a[i] = work(i);
```

```
プロセス0: i =  0,  1,  2, ..., 24
プロセス1: i = 25, 26, 27, ..., 49
プロセス2: i = 50, 51, 52, ..., 74
プロセス3: i = 75, 76, 77, ..., 99
```

  - 実際はプロセス数で割りきれない場合があるので、もう少し複雑になる。
  - 通常、このBlock分割で十分な高速化が得られる場合が多い。

- Cyclic分割： とびとびに分割

```
for (i = myid; i < 100; i += procs)
  a[i] = work(i);
```

```
プロセス0: i =  0,  4,  8, 12, ..., 96
プロセス1: i =  1,  5,  9, 13, ..., 97
プロセス2: i =  2,  6, 10, 14, ..., 98
プロセス3: i =  3,  7, 11, 15, ..., 99
```

- Block-Cyclic分割： 上記二つの組み合わせ

```
for (i = myid*4; i < 100; i += procs*4)
  for (ii = i; ii < i+4; i++)
  a[i] = work(i);
```

```
プロセス0: i =  0, 1, 2, 3,16,17,18,19, ...
プロセス1: i =  4, 5, 6, 7,20,21,22,23, ...
プロセス2: i =  8, 9,10,11,24,25,26,27, ...
プロセス3: i = 12,13,14,15,28,29,30,31, ...
```

```
procs : プロセス数
myid : ランク(プロセス番号)
```

# 別の分担のさせ方

- プロセス毎に全く別の処理を割り当てる。

```
if (myid == 0)
  work1();
else if (myid == 1)
  work2();
else if (myid == 2)
  work3();
```

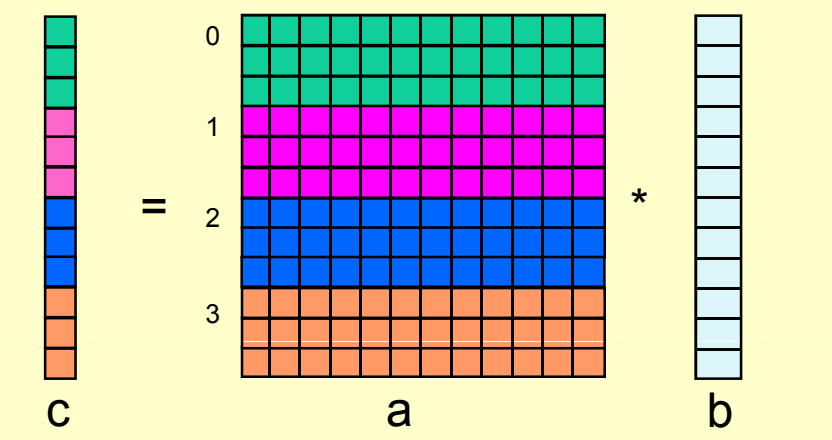
  - 割り当てる仕事の量が均等でなければ、並列処理の効率が悪い。
    ⇒ プロセス数が変化する場合は割り当てが難しい。

  - この講義では、この分担方法については扱わない。

# ベクトル行列積のループ並列化例(1)
## 外側ループを Block分割する

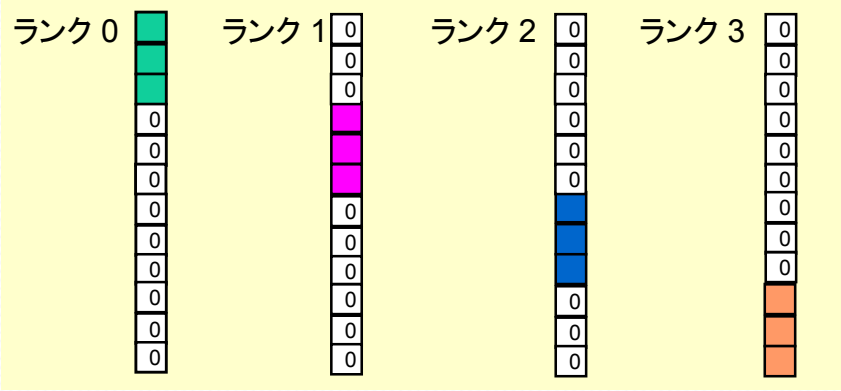
procs : プロセス数
myproc : ランク(プロセス番号)

```
nmod = N%procs;
ndiv = N/procs;
if (nmod == 0){
  start = myid * ndiv;
  end = start + ndiv - 1;
} else {
  start = myid * (ndiv + 1);
  if (myid == (procs - 1))
    end = N - 1;
  else
    end = start + ndiv ;
}
for (i = start; i <= end; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

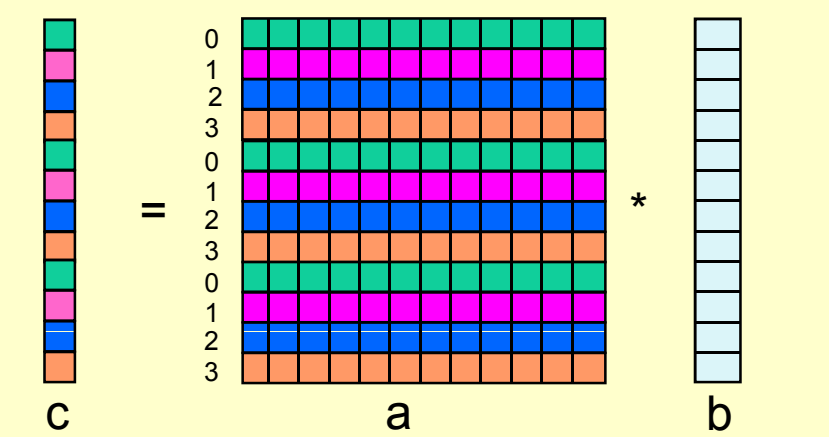



- 繰り返しの数 Nをプロセス数で等分したブロック毎にプロセスに割り当てる。
- 連続した領域を割り当てられるのでメモリアクセスや通信の効率が良い。
- Nがプロセス数で割りきれない場合の処理が多少複雑。
- 外側ループは独立した計算なので、計算結果が各プロセスに分散

計算後のCの値(自分が担当した要素以外は 0)



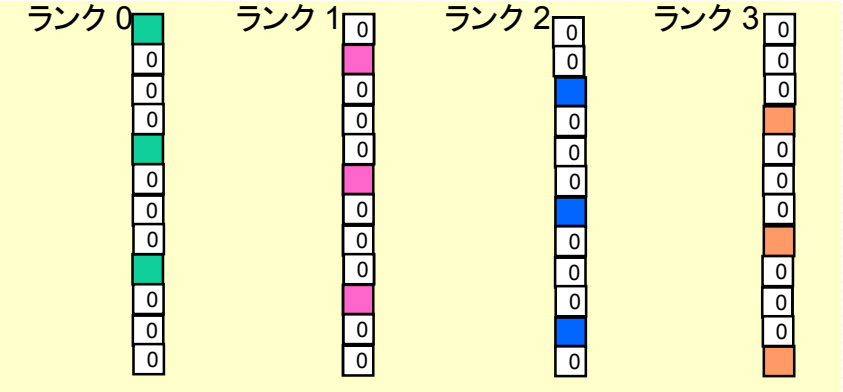
# ベクトル行列積のループ並列化例(2)
# 外側ループを Cyclic分割する



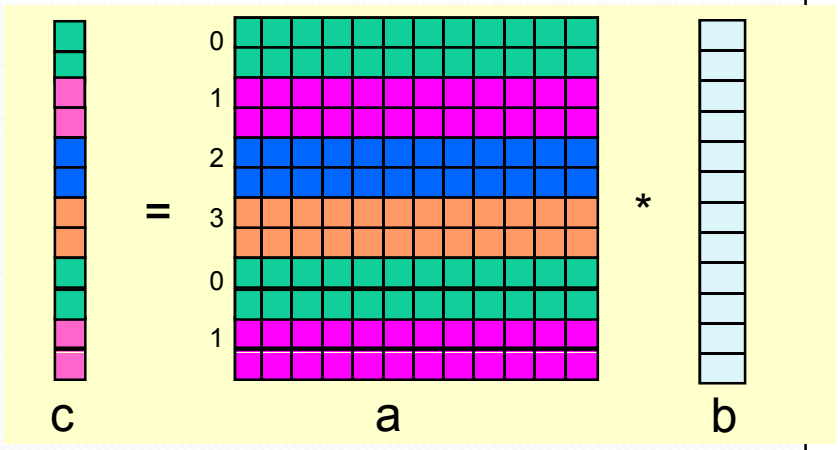

```
procs : プロセス数
myid : ランク(プロセス番号)

for (i = myid; i < N; i+=procs)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

- ループの各繰り返しを一つずつ順番に各プロセスに割り当てる。
- Nの値によらず、均等に処理を割り当てることができる。
  - Block分割で均等に処理を割り当てられない場合に選択。
- 計算後の値がとびとびになるので、最後に計算結果を取りまとめる処理が複雑になり、コストも高くなる。

計算後のCの値(自分が担当した要素以外は 0)



# ベクトル行列積のループ並列化例(3)
# 外側ループを Block-Cyclic分割する

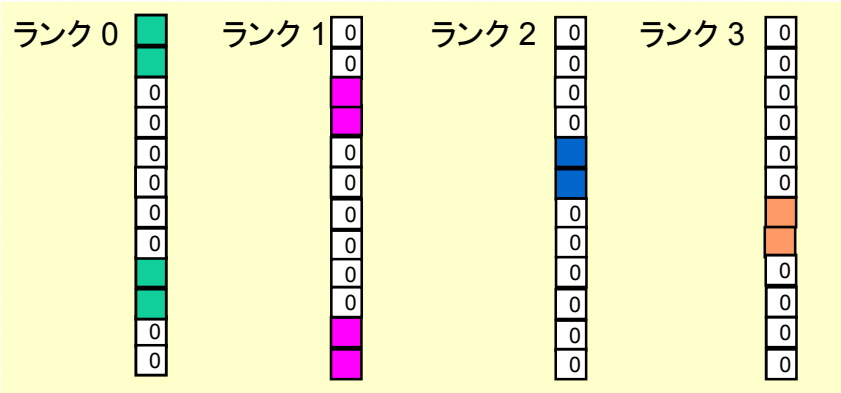



```
procs : プロセス数
myproc : ランク(プロセス番号)
```

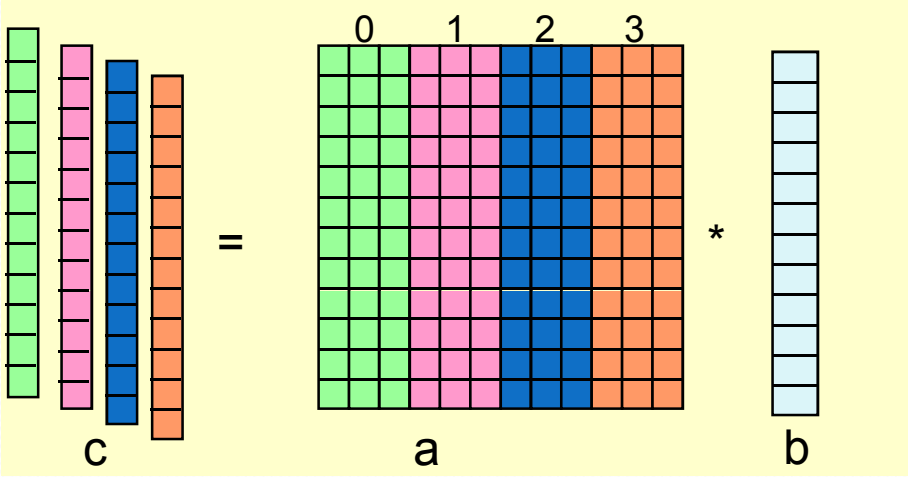
```
bs = 2;

for (k = myid*bs; k < N;
       k += procs*bs){
  end = (k+bs) > N ? N : (k + bs);
  for (i = k; i < end; i++)
    for (j = 0; j < N; j++)
      c[i] += a[i*N+j] * b[j];
}
```

- 一定サイズのブロック単位で、プロセスに順に割り当てる。
- キャッシュサイズを意識した並列化を行う場合に選択。
  - 例えば同じブロックを何度も参照する計算の場合、ブロックサイズをキャッシュサイズ以内に設定し、ブロックを再利用するようにループを変形する。

計算後のCの値(自分が担当した要素以外は 0)



# ベクトル行列積のループ並列化例(4)
# 内側ループを Block分割する



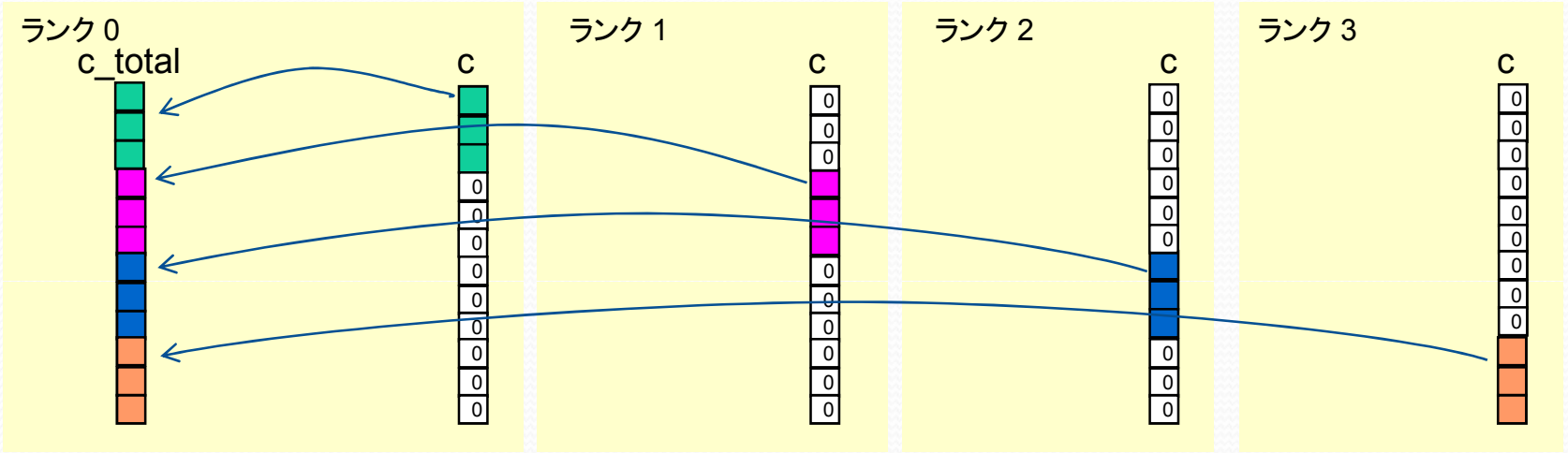

```
procs : プロセス数
myproc : ランク(プロセス番号)
```

```
nmod = N%procs;
ndiv = N/procs;
if (nmod == 0){
  start = myid * ndiv;
  end = start + ndiv - 1;
} else {
  start = myid * (ndiv + 1);
  if (myid == (procs - 1))
    end = N - 1;
  else
    end = start + ndiv ;
}
for (i = 0; i < N; i++)
  for (j = start; j <= end; j++)
    c[i] += a[i*N+j] * b[j];
```

- 内側ループは総和計算なので、
  各プロセスの計算結果は部分和
- この後、全プロセスでの総和計算が必要
- 内側ループの Cyclic分割、Block-Cyclic分割は割愛

# 計算結果のとりまとめ(1)

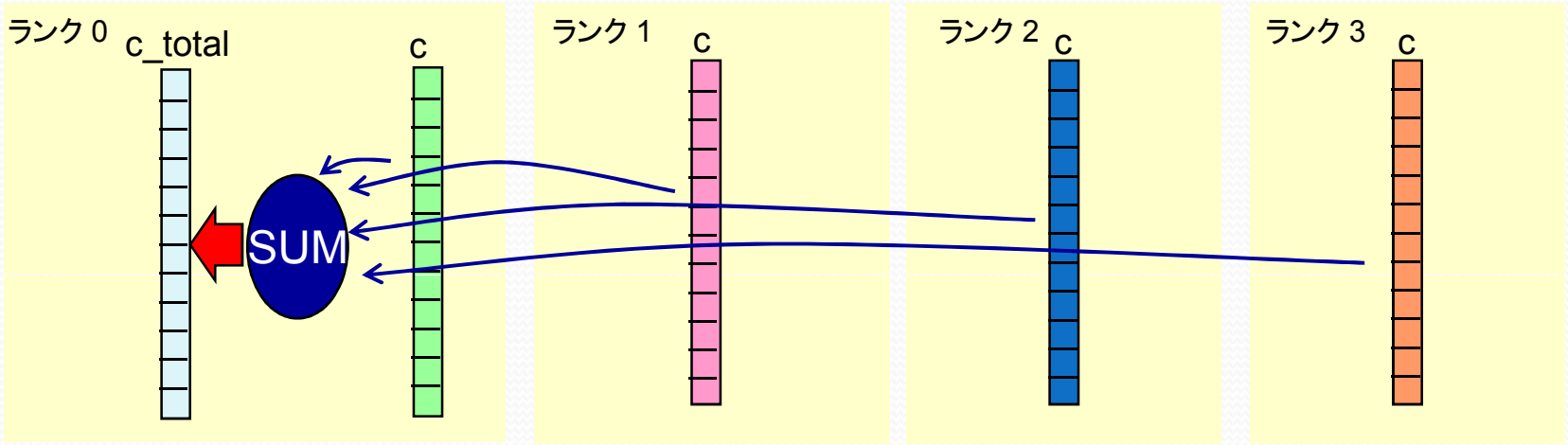
- 各プロセスに分かれている計算結果を集約する




- プログラムによっては、結果を分散させたままでよい場合もある。
  - 今回のプログラム例では、計算結果のベクトルを別の計算に適用したい場合や、計算結果をまとめて表示したい場合にとりまとめ。
- ループの分割方法によって取りまとめ方が違う
  - Block分割： 集団通信 MPI_Gather を利用
    - 結果を全員に持たせたい場合は MPI_Allgatherを利用
  - その他： 一対一通信（MPI_Send, MPI_Recv等）を利用
    - 本講義では割愛。

# 計算結果のとりまとめ(2)

- 全プロセスの総和を計算する




- 各プロセスに部分和を計算させるプログラムの場合、必ず行う。

- 分割の方法によらず、集団通信 MPI_Reduceを利用
  - 結果を全員に持たせたい場合は MPI_Allreduceを利用

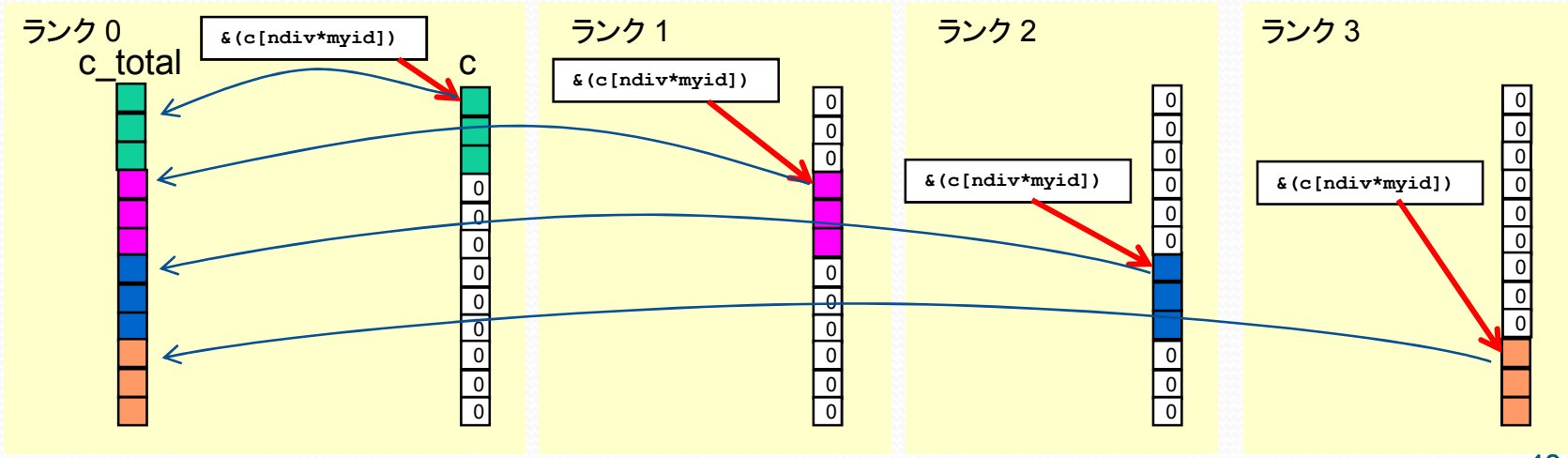
# MPI_Gatherによるとりまとめの例（1）

- 外側ループをBlock分割で分担し、かつ Nがプロセス数で割りきれる場合

```
c = (double *)malloc(N*sizeof(double));
c_total = (double *)malloc(N*sizeof(double));
  ...
MPI_Gather(&(c[ndiv*myid]), ndiv, MPI_DOUBLE, c_total, ndiv,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
ndiv = N / procs      （小数点以下は切り捨て）
```

例） N = 10、 procs（プロセス数）= 4 の場合



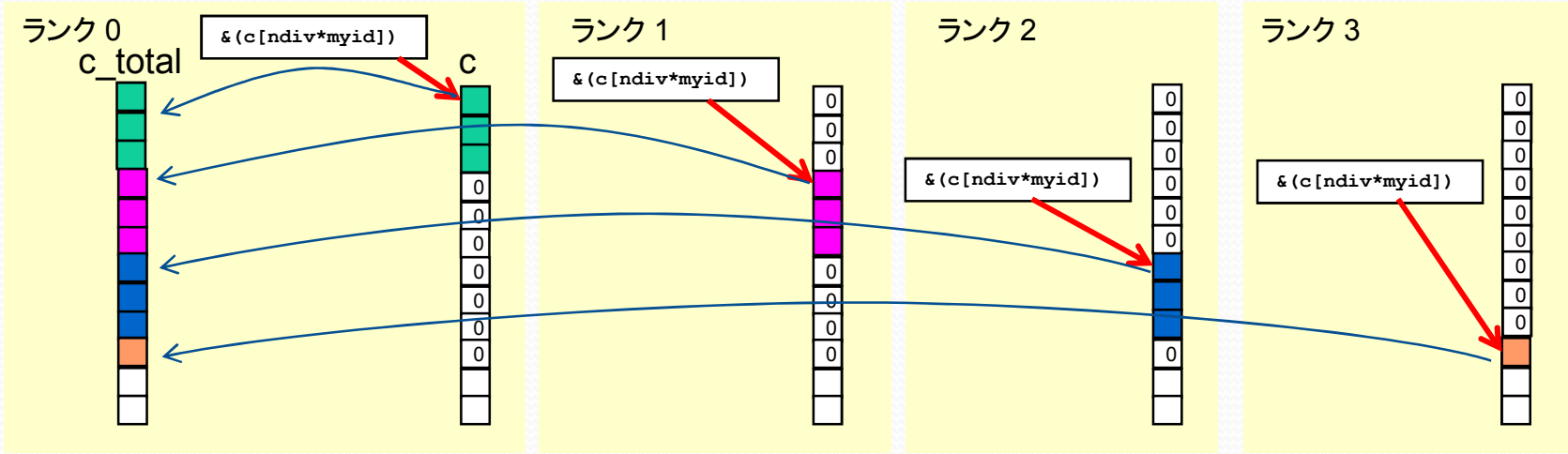
# MPI_Gatherによるとりまとめの例（2）

- **Nがプロセス数で割りきれない場合**
  - 最後のプロセスだけ割り当てられる要素数が少ない。
  - しかし、MPI_Gatherは全プロセスで送信サイズが同じである必要がある。
  - 解決策の一つとして、以下のように配列 cを若干大きめに確保する方法がある。

```
c = (double *)malloc((ndiv+1)*procs*sizeof(double));
c_total = (double *)malloc((ndiv+1)*procs*sizeof(double));
...
MPI_Gather(&(c[(ndiv+1)*myid]), ndiv+1, MPI_DOUBLE, c_total,
           ndiv+1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
ndiv = N / procs        （小数点以下は切り捨て）
```

例） N = 10、 procs（プロセス数）= 4 の場合



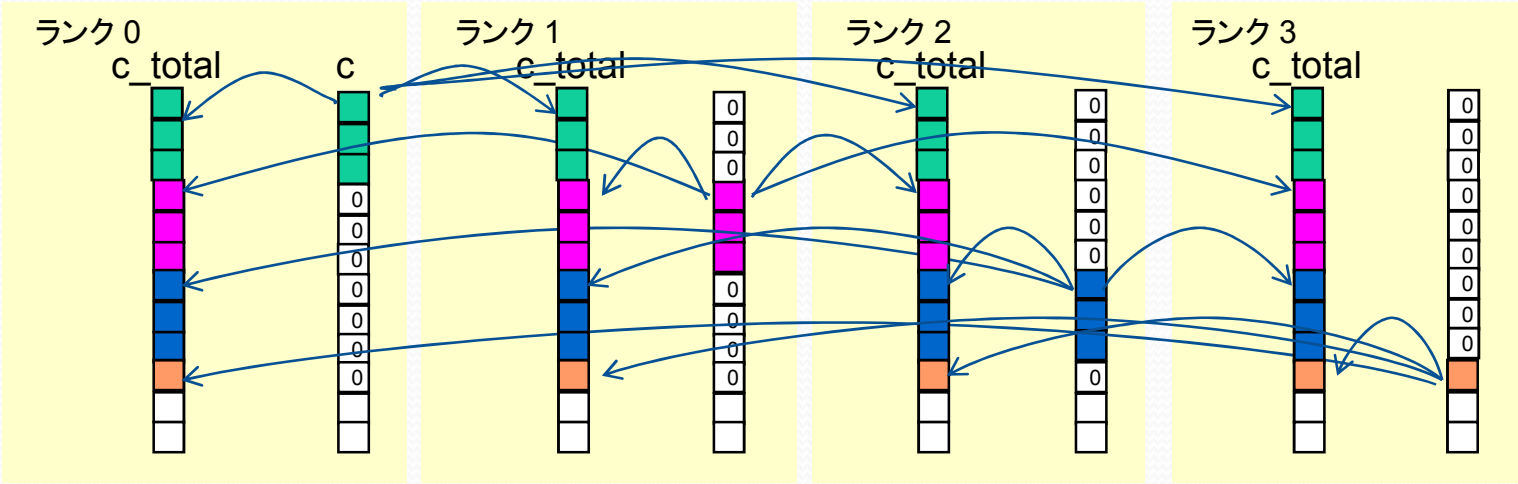
# MPI_Allgatherによるとりまとめの例

- とりまとめた配列を全プロセスに持たせる

```
c = (double *)malloc((ndiv+1)*procs*sizeof(double));
c_total = (double *)malloc((ndiv+1)*procs*sizeof(double));
...
MPI_Allgather(&(c[(ndiv+1)*myid]), ndiv+1, MPI_DOUBLE, c_total,
              ndiv+1, MPI_DOUBLE, MPI_COMM_WORLD);
```

```
ndiv = N / procs      （小数点以下は切り捨て）
```

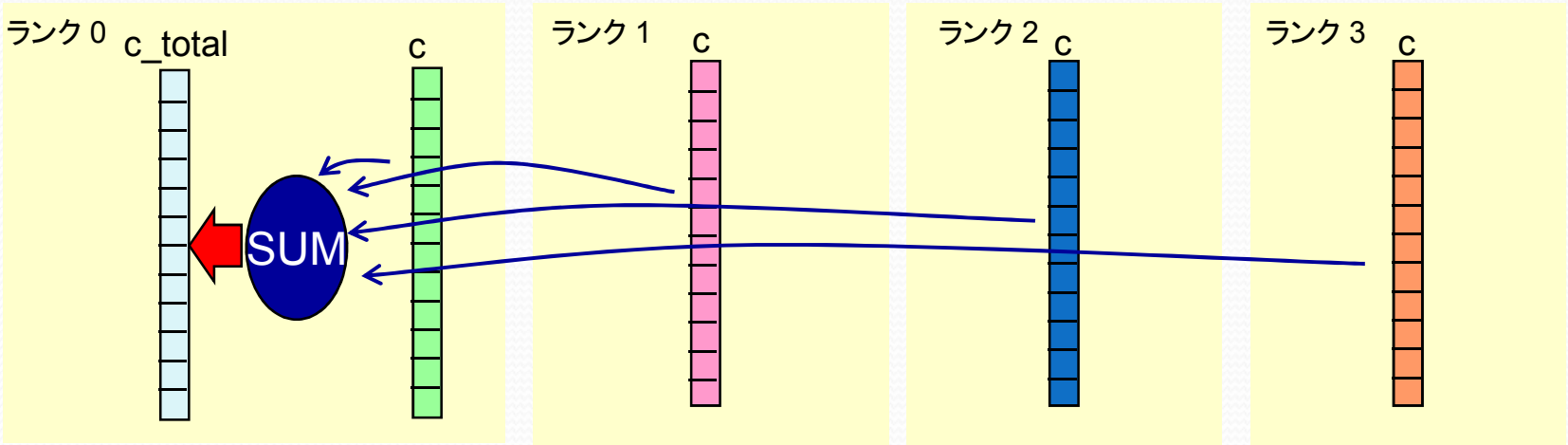
例） N = 10、 procs（プロセス数）= 4 の場合



# MPI_Reduceによるとりまとめの例

- 全プロセスの総和

```
MPI_Reduce(c, c_total, N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```



# ちょっと、ここまでのまとめ

- **仕事をどのようにプロセスに分配するか。**
  **⇒ ランクを使って処理を分割し、分担させる**

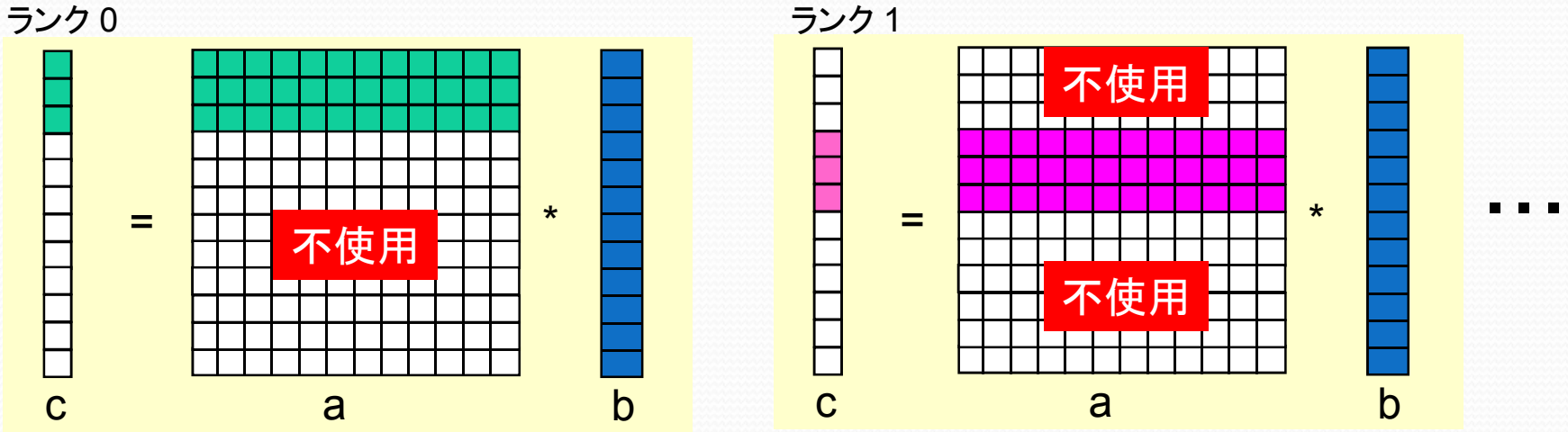
- **ループを分配する場合、各プロセスの担当範囲をランクから計算**
  - Block, Cyclic, Block-Cyclic
    - 割り切れない場合、多少複雑な計算
  - 担当範囲の計算方法は、MPIに限った話ではなく、他の並列化手法でも共通

- **必要に応じて、最後の取りまとめ**
  - MPIの通信ルーチンを使って一箇所に集める。

# もう一つ、プロセス並列処理に特有の話：どのようにデータを各プロセスに配置するか？

- **ここまでの並列化例では、基本的に全てのプロセスが全ての配列を重複して所有**
  - 利点： データのサイズや構造を変えずに並列化できる。
    - 並列化が容易
  - 欠点： プロセス数を増やしても、扱えるデータ量が変わらない。
    - 実際には使わない領域が大量に存在する。

```
double a[N*N], b[N], c[N];

for (i = myid*N/procs; i < (myid+1)*N/procs; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

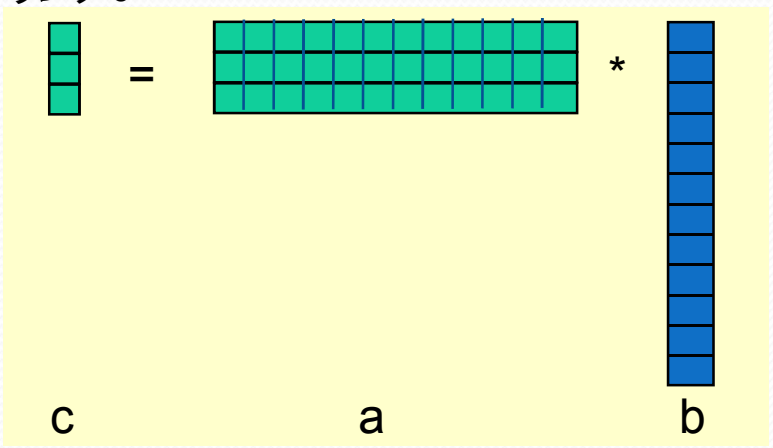


# どのようにデータを各プロセスに配置するか？（続き）

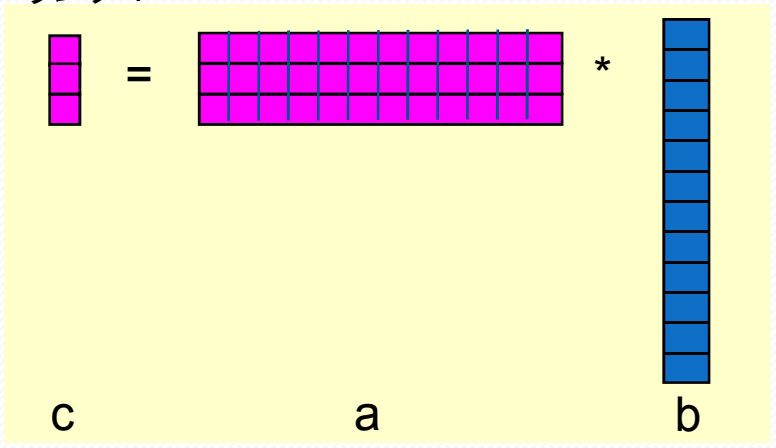
- 一方、データを分割して各プロセスに配置することも可能
  - 利点： メモリの有効利用
    - プロセス数に応じて扱えるデータ量も増加
  - 欠点： 各プロセスの配列のサイズが変わる。
    - 並列化にともなってプログラム全体の書き換えが必要。

```
double *a, b[N], *c;
a = (double *)malloc(N*N/procs*sizeof(double));
c = (double *)malloc(N/procs*sizeof(double));
for (i = 0; i < N/procs; i++)
  for (j = 0; j < N; j++)
    c[i] += a[i*N+j] * b[j];
```

ランク 0




ランク 1

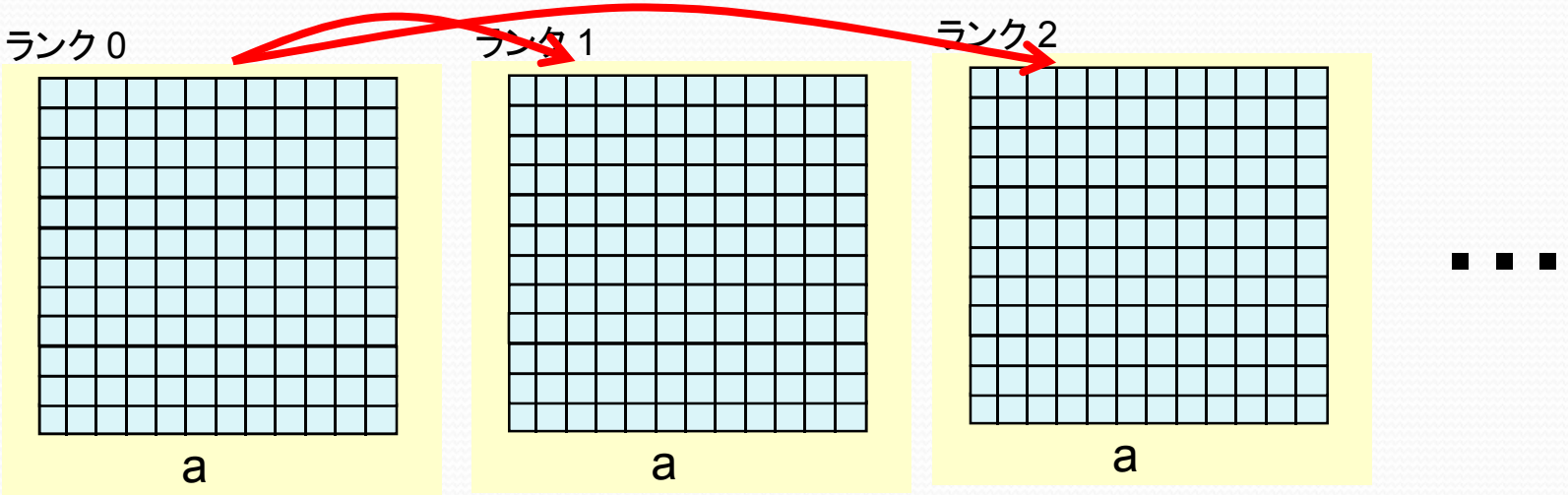



...

# データ配置によるプログラムの違い：重複配置における初期値データの配布

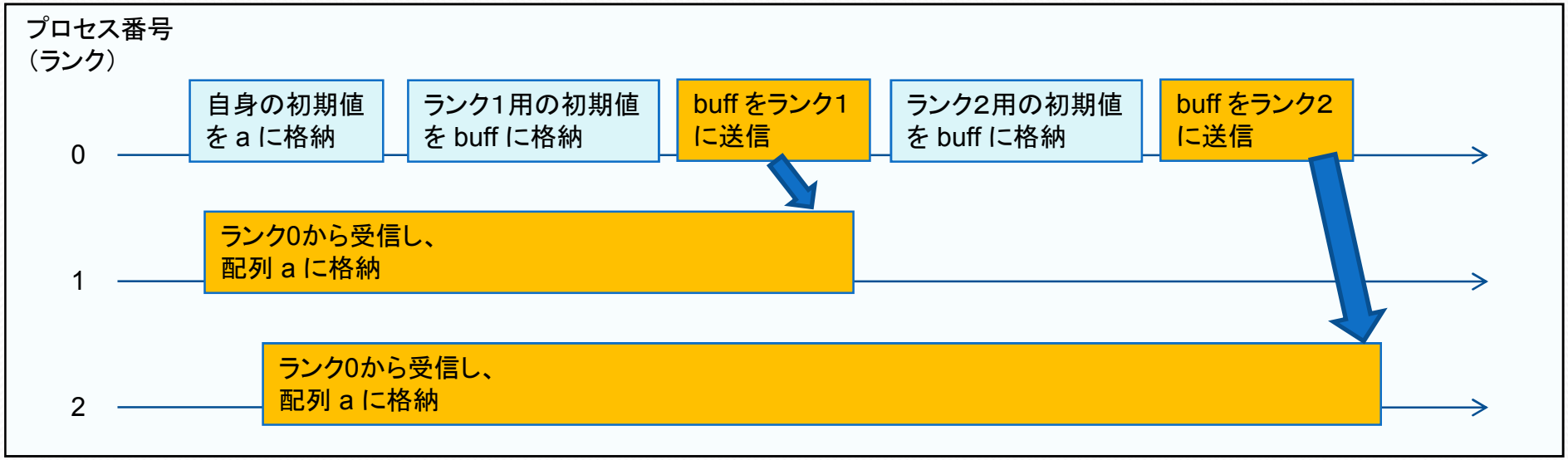
- **初期値データ： ここではランク0が初期値データをまとめて生成すると仮定。**
  - プログラムによって初期値をファイルから読み込む場合とプログラム中で生成する場合がある。
  - それぞれの場合について、初期値の入力もしくは生成を並列に行えることもあるが、ランク0がまとめて行うことのほうが多い。
- **コピー配置の場合： ランク0で全初期値を生成して各ランクに配布。**
  - 配布には集団通信 MPI_Bcast を利用。

```
MPI_Bcast(a, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```



# データ配置によるプログラムの違い：分割配置における初期値データの配布

- **分割配置の場合、**
  - ランク0はデータを生成した後、個別に各プロセスに MPI_Send
  - 他のランクはMPI_Recvでランク0からのデータ送信を待って配列に格納



# 並列化後のベクトル・行列積（Block分割、重複配置）

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define N 1000

int main(int argc, char *argv[])
{
  int i, j, myid, procs, nmod, ndiv,
      start, end;
  double *a, *b, *c, *c_total;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);

  nmod = N%procs;
  ndiv = N/procs;
  if (nmod == 0){
    start = myid * ndiv;
    end = start + ndiv - 1;
  } else {
    start = myid * (ndiv + 1);
    if (myid == (procs - 1))
      end = N - 1;
    else
      end = start + ndiv ;
  }
```

```
  a = (double *)malloc(N*N*sizeof(double));
  b = (double *)malloc(N*sizeof(double));
  if (nmod == 0)
    c = (double *)malloc(N*sizeof(double));
  else
    c = (double *)malloc((ndiv+1)
                         *procs*sizeof(double));

  if (myid == 0){
    if (nmod == 0)
      c_total = (double *)malloc(N
                  *sizeof(double));
    else
      c_total = (double *)malloc((ndiv+1)
                         *procs*sizeof(double));

    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        a[i*N+j] = i;
    for (i = 0; i < N; i++)
      b[i] = i;
  }

  MPI_Bcast(a, N*N, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
  MPI_Bcast(b, N, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
```

次ページへ続く

# 並列化後のベクトル・行列積（Block分割、重複配置）

前ページより

```
  for (i = start; i < end; i++)
    c[i] = 0;

  for (i = start; i <= end; i++)
    for (j = 0; j < N; j++)
      c[i] += a[i*N+j] * b[j];

  if (nmod == 0)
    MPI_Gather(&(c[ndiv*myid]), ndiv,
               MPI_DOUBLE, c_total, ndiv,
               MPI_DOUBLE, 0, MPI_COMM_WORLD);
  else
    MPI_Gather(&(c[(ndiv+1)*myid]), ndiv+1,
               MPI_DOUBLE, c_total, ndiv+1,
               MPI_DOUBLE, 0, MPI_COMM_WORLD);

  if (myid == 0){
    for (i = 0; i < N; i++)
      printf("(%d: %.2f) ", i, c_total[i]);
    printf("¥n");
  }

  MPI_Finalize();

  return 0;
}
```

# 並列化後のベクトル・行列積（Block分割、分割配置）

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

#define N 1000

int main(int argc, char *argv[])
{
  int i, j, myid, procs, nmod, ndiv, start,
      end, p, base_size, final_size;
  double *a, *b, *c, *buf, *c_local;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);

  nmod = N%procs;
  ndiv = N/procs;
  if (nmod == 0){
    end = ndiv-1;
    base_size = ndiv;
    final_size = base_size;
  } else {
    base_size = ndiv + 1;
    final_size = N - base_size*(procs-1);
    if (myid == (procs - 1))
      end = final_size - 1;
    else
      end = base_size - 1;
  }

  a = (double *)malloc(N*base_size
                       *sizeof(double));
  b = (double *)malloc(N*sizeof(double));
  c_local = (double *)malloc(base_size
              *sizeof(double));
  if (myid == 0){
    c = (double *)malloc(base_size
                *procs*sizeof(double));
    buf = (double *)malloc(N*base_size
                *sizeof(double));
    for (i = 0; i < base_size; i++)
      for (j = 0; j < N; j++)
        a[i*N + j] = i;
    for (p = 1; p < procs-1; p++){
      for (i = 0; i < base_size; i++)
        for (j = 0; j < N; j++)
          buf[i*N+j] = i + base_size*p;
      MPI_Send(buf, N*base_size, MPI_DOUBLE,
                  p, 0, MPI_COMM_WORLD);
    }
```

次ページへ続く

# 並列化後のベクトル・行列積（Block分割、分割配置）

前ページより

```
    for (i = 0; i < final_size; i++)
      for (j = 0; j < N; j++)
        buf[i*N+j] = i + base_size*(procs-1);
    MPI_Send(buf, N*final_size, MPI_DOUBLE,
             procs-1, 0, MPI_COMM_WORLD);

    for (i = 0; i < N; i++)
      b[i] = i;
  } else{
    if (myid == (procs - 1))
      MPI_Recv(a, N*final_size, MPI_DOUBLE,
            0, 0, MPI_COMM_WORLD, &status);
    else
      MPI_Recv(a, N*base_size, MPI_DOUBLE,
            0, 0, MPI_COMM_WORLD, &status);
  }

  MPI_Bcast(b, N, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);

  for (i = 0; i < base_size; i++)
    c_local[i] = 0;

  for (i = 0; i <= end; i++)
    for (j = 0; j < N; j++)
      c_local[i] += a[i*N+j] * b[j];
```

```
    MPI_Gather(c_local, base_size,
       MPI_DOUBLE, c, base_size,
       MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if (myid == 0){
      for (i = 0; i < N; i++)
        printf("(%d: %.2f) ", i, c[i]);
      printf("¥n");
    }

    MPI_Finalize();

    return 0;
  }
```

# 並列化手法のまとめ

- **選択肢**
  - ループの分担： Block分割、Cyclic分割、Block-Cyclic分割
  - データの配置： コピーして配置、分割して配置
- **ほとんどの場合、Block分割で十分な並列化効果**
  - 負荷バランスが悪い場合は Cyclic分割を検討。
  - 同じデータを何度も参照する場合、Block-Cyclic分割でキャッシュの最適化を図る。
- **データ配置はメモリが不足しなければコピーして配置の方が簡単**
  - ただし、分割して配置するとプロセスごとの使用メモリ量が減るため、キャッシュの利用効率が向上して性能が上がる場合もある。

# MPIプログラムの時間計測
## MPI_Wtime

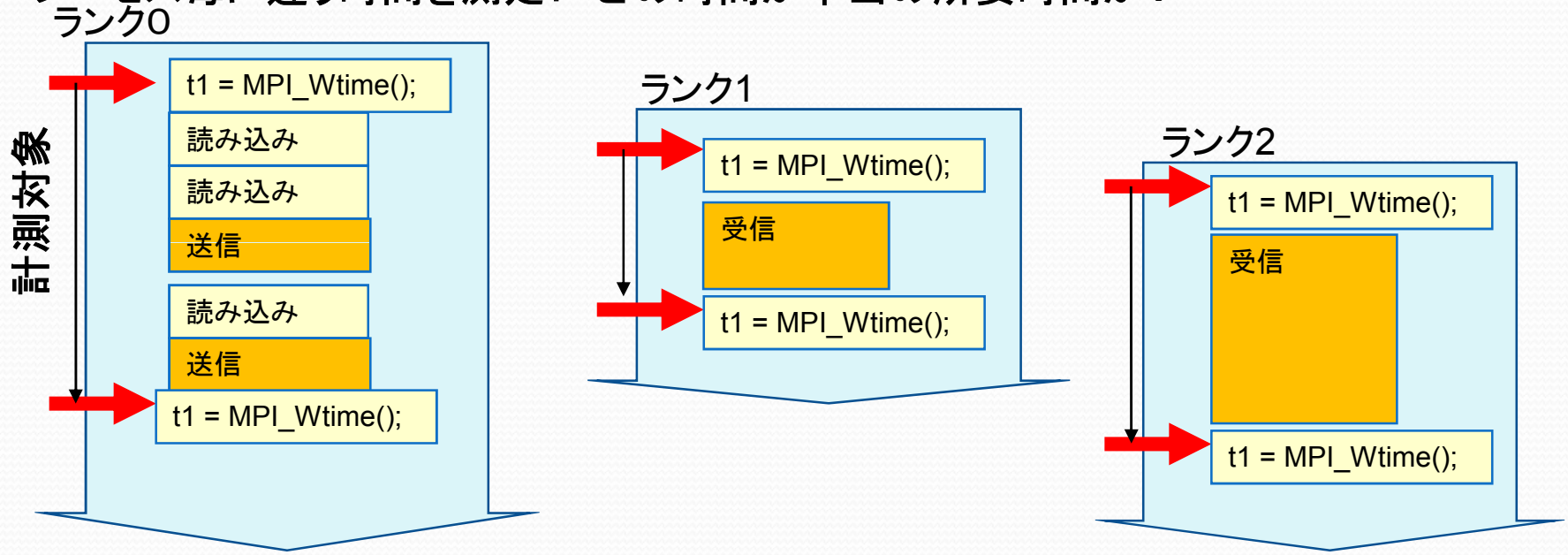
- 現在時間(秒)を実数で返す関数
- 利用例

計測対象

```
 ...
double t1, t2;
 ...
t1 = MPI_Wtime();
if (myid == 0){
   printf("value for proc 0: ");
   scanf("%f", &myval);
   for (i = 1; i < procs; i++){
      printf("value for proc %d: ", i);
      scanf("%f", &val);
      MPI_Send(&val, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
   }
 } else
   MPI_Recv(&myval, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
t2 = MPI_Wtime();

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

# 並列プログラムにおける時間計測の問題

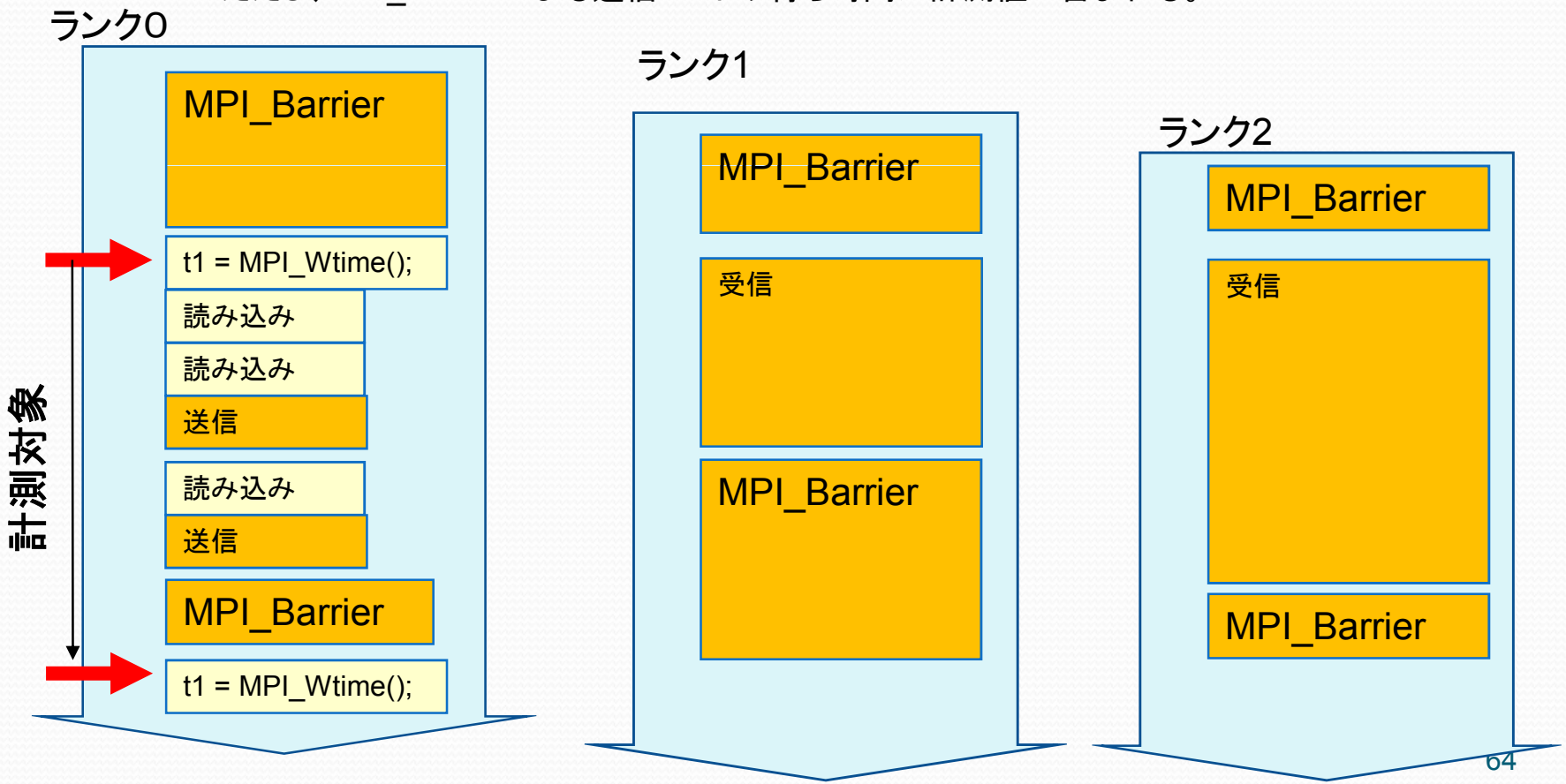
- プロセス毎に違う時間を測定： どの時間が本当の所要時間か？




- 特に、MPI_Sendは受信側プロセスが受信するのを待たずに終了するので、MPI_Sendで終わるプロセスでは、他のプロセスがまだ仕事をしている時に終了時刻の計測を行うことになる。

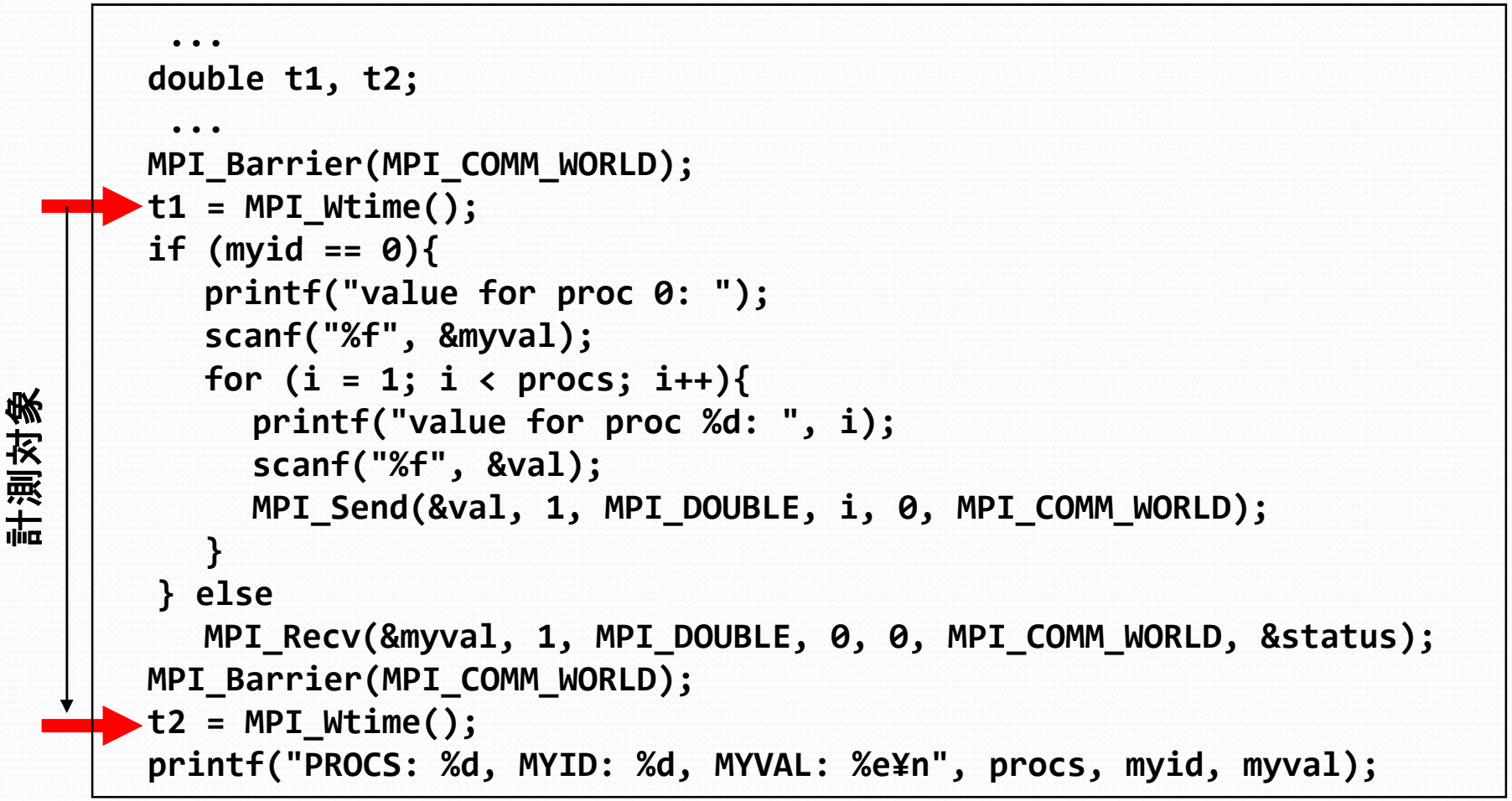
# 集団通信 MPI_Barrierを使った解決策

- **全プロセスを同期させる集団通信 MPI_Barrierを時間計測前に実行する。**
  - 全プロセスで開始時刻と終了時刻をほぼ揃うので、ランク0だけで計測できる。
    - ただし、MPI_Barrierによる通信コストや待ち時間が計測値に含まれる。



# MPI_Barrierを使ったプログラム例

- グループ内の全プロセスがMPI_Barrierを実行するまで、次の処理に移らない。




```
 ...
double t1, t2;
 ...
MPI_Barrier(MPI_COMM_WORLD);
t1 = MPI_Wtime();
if (myid == 0){
   printf("value for proc 0: ");
   scanf("%f", &myval);
   for (i = 1; i < procs; i++){
      printf("value for proc %d: ", i);
      scanf("%f", &val);
      MPI_Send(&val, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
   }
 } else
   MPI_Recv(&myval, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
MPI_Barrier(MPI_COMM_WORLD);
t2 = MPI_Wtime();
printf("PROCS: %d, MYID: %d, MYVAL: %e¥n", procs, myid, myval);
```

# まとめ

- **並列計算機の能力を発揮させるには並列プログラムが必要**
- **MPIは”プロセス並列”で並列プログラムを作成するための規格**
  - C言語やFortranから呼び出すプロセス間通信ルーチンやその他の補助ルーチンの定義
- **プログラムの並列化　＝　処理の分割とデータの配置**
  - MPIでは、プロセスのランクに応じて処理を割り当て
  - データの配置はコピー配置もしくは分割配置
- **並列プログラムの処理時間計測**
  - どの時間を計測するかが重要

# MPIの利点と欠点

- **利点1．高速化に向けた細かいチューニングが可能**
  - 通信のタイミングや転送するデータの大きさ、さらに処理のプロセスへの分担のさせ方やデータの配置方法等、性能に影響する事項をプログラムで直接指示できるので、慣れれば高い性能を得られやすい。
- **利点2．ほぼ全ての並列計算機で同じMPIプログラムを利用可能**
  - 現在利用されているほとんどの並列計算機には、MPIのライブラリが実装されている。
  - MPIの規格に準拠していれば、基本的に互換性は確保されている。
- **欠点1．並列プログラムの作成が複雑**
  - プロセス毎のデータ配置やプロセス間の通信等を全て自分で記述しないといけないため、習得には多少時間を要する。
  - 既存のプログラムを並列化する場合、プログラム構造の大幅な変更が必要。

# MPIの関連情報

- MPI仕様（日本語訳）
  http://phase.hpcc.jp/phase/mpi-j/ml/

- 理化学研究所の講習会資料
  http://accc.riken.jp/HPC/training/mpi/mpi_all_2007-02-07.pdf

- 本講義資料に関する質問は以下まで:

九州大学情報基盤研究開発センター　南里　豪志
Email　nanri@cc.kyushu-u.ac.jp