

並列プログラミング入門 (MPI)

大阪大学サイバーメディアセンター
日本電気株式会社

**本資料は、東北大学サイバーサイエンスセンターとNECの共同により作成され、大阪大学サイバーメディアセンターの環境で実行確認を行い、修正を加えたものです。
無断転載等は、ご遠慮下さい。**

目次

1. 並列化概要
 2. MPI概要
 3. 演習問題1
 4. 演習問題2
 5. 演習問題3
 6. 演習問題4
 7. MPIプログラミング
 8. 演習問題5
 9. 実行方法と性能解析
 10. 演習問題6
- 付録1.主な手続き
- 2.参考文献, Webサイト

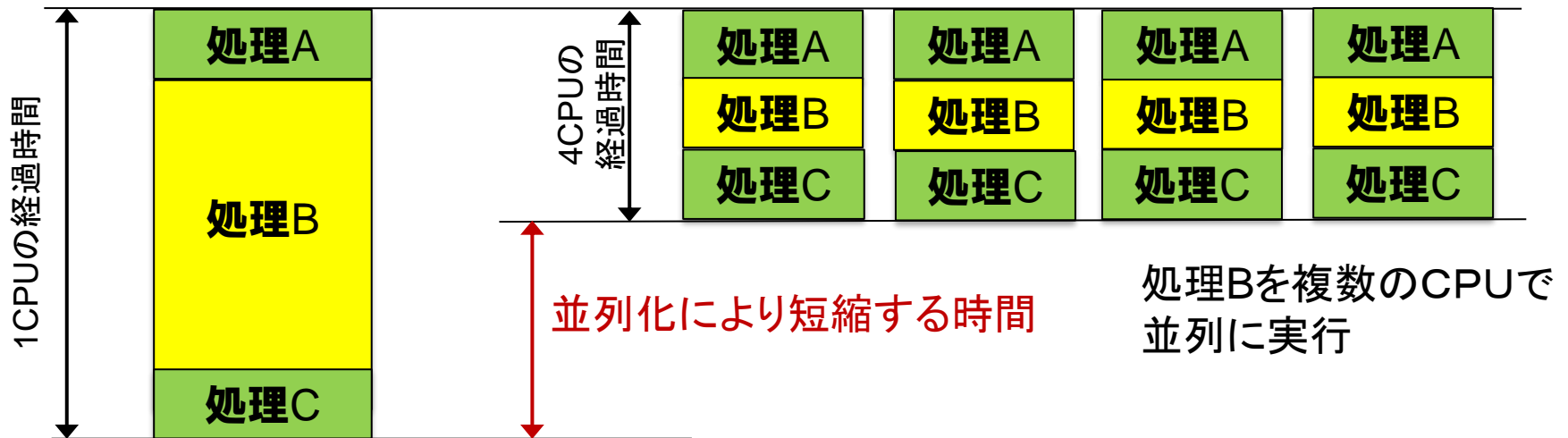
1. 並列化概要

並列処理・並列実行

- 仕事(処理)を複数のコアに分割し、同時に実行すること

並列化

- 並列処理を可能とするために、処理の分割を行うこと



並列化の効果

行列積プログラム

```
implicit real(8)(a-h,o-z)
parameter ( n=10240 )
real(8) a(n,n),b(n,n),c(n,n)
real(4) etime,cp1(2),cp2(2),t1,t2,t3
do j = 1,n
  do i = 1,n
    a(i,j) = 0.0d0
    b(i,j) = n+1-max(i,j)
    c(i,j) = n+1-max(i,j)
  enddo
enddo
write(6,50) ' Matrix Size = ',n
50 format(1x,a,i5)
t1=etime(cp1)
do j=1,n
  do k=1,n
    do i=1,n
      a(i,j)=a(i,j)+b(i,k)*c(k,j)
    end do
  end do
end do
t2=etime(cp2)
t3=cp2(1)-cp1(1)
write(6,60) ' Execution Time = ',t2,' sec',' A(n,n) = ',a(n,n)
60 format(1x,a,f10.3,a,1x,a,d24.15)
stop
end
```

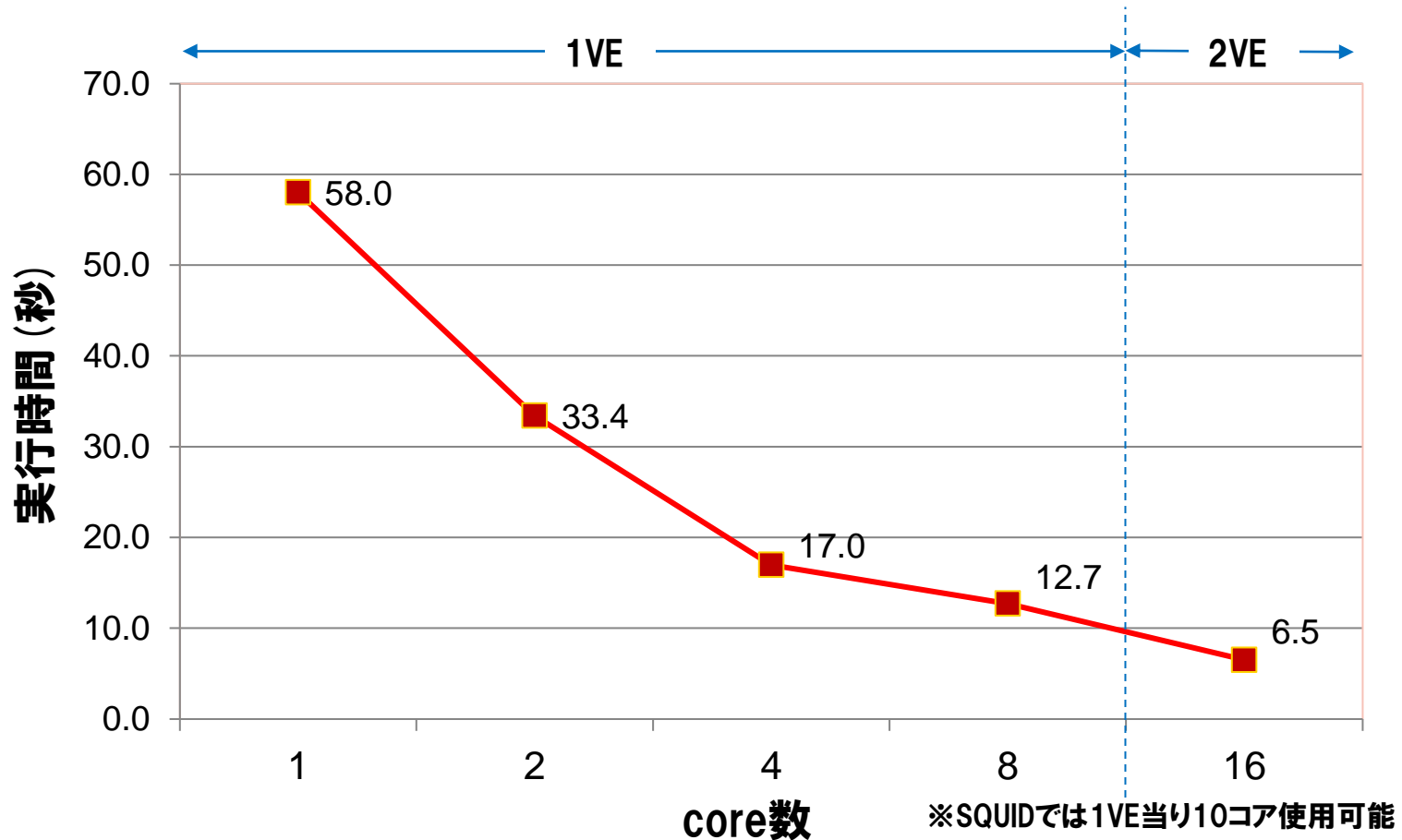
● SX-Aurora TSUBASA 1coreの 実行時間は約58秒

```
***** Program Information *****
Real Time (sec) : 58.099338
User Time (sec) : 57.871755
Vector Time (sec) : 57.833009
Inst. Count : 68257454157
V. Inst. Count : 16782152502
V. Element Count : 4296230848942
V. Load Element Count : 2147588508684
FLOP Count : 2147483648345
MOPS : 93686.977745
MOPS (Real) : 93313.422266
MFLOPS : 37110.243188
MFLOPS (Real) : 36962.274548
A. V. Length : 255.999989
V. Op. Ratio (%) : 99.050525
L1 Cache Miss (sec) : 0.002656
CPU Port Conf. (sec) : 0.000000
V. Arith. Exec. (sec) : 27.333839
V. Load Exec. (sec) : 30.498928
VLD LLC Hit Element Ratio (%) : 49.992749
FMA Element Count : 1073741824000
Power Throttling (sec) : 0.000000
Thermal Throttling (sec) : 0.000000
Memory Size Used (MB) : 3856.000000
Non Swappable Memory Size Used (MB) : 228.000000
```

● 複数のcoreを用いることで実行時間を 短縮することが可能に

並列化の効果

- 並列化により複数のcoreを利用して実行時間を短縮
- MPIを用いることで, SX-Aurora TSUBASAの複数ノードが利用可能



16coreで約9分1の時間に短縮

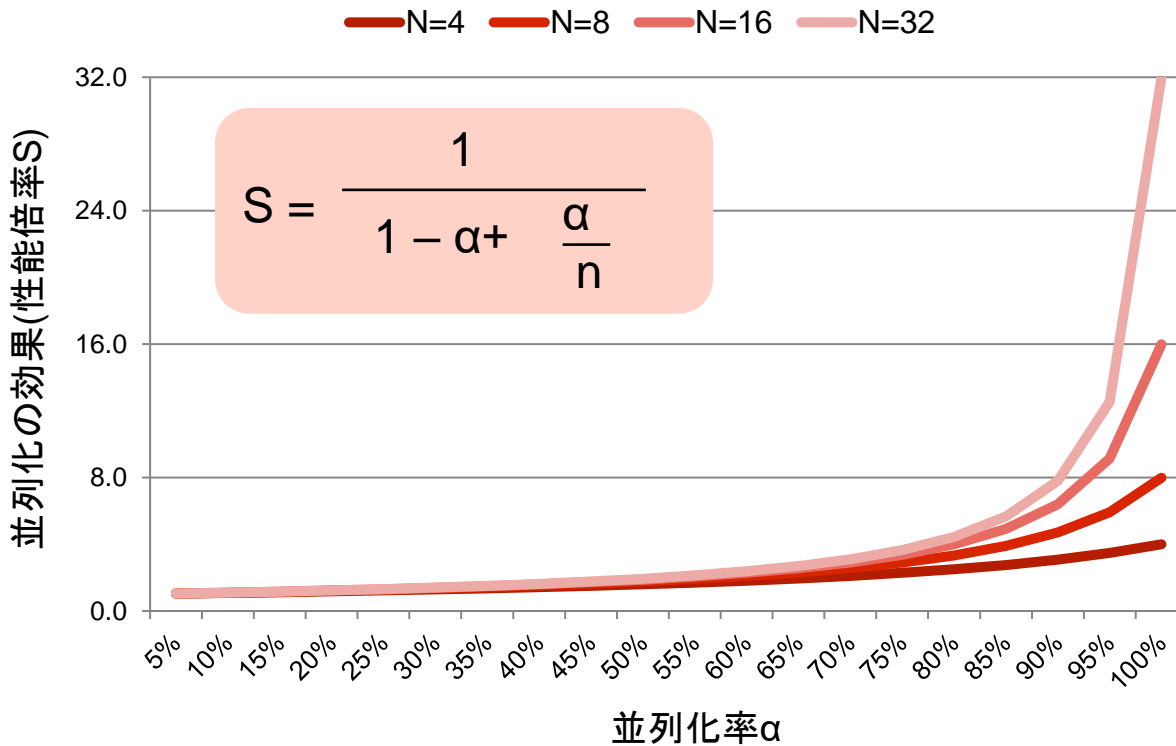
並列化の効果

■ 並列に実行可能(あるいは効果のある)部分と並列に実行不可(あるいは効果のない)部分を見つけ、並列に実行可能部分を複数のCPUに割り当てる。

■ できるだけ多くの部分を並列化の対象としなければ、CPU数に応じた効果が得られない。

$$\text{並列化率}\alpha = \frac{\text{並列化対象部分の時間}}{\text{全体の処理時間}} \\ (\text{並列化の対象部分と非対象部分の時間の合計})$$

並列化の効果



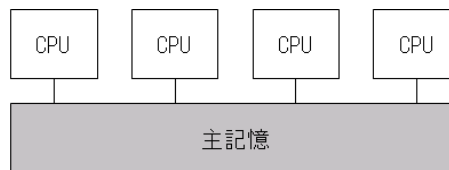
- Nは並列数
- 並列化率が100%から下がるにしたがって性能倍率は急速に低下する

並列化率100%はあり得ない(データの入出力で必ず逐次処理発生)が、可能な限り100%に近づかなければ並列化の効果は得られない

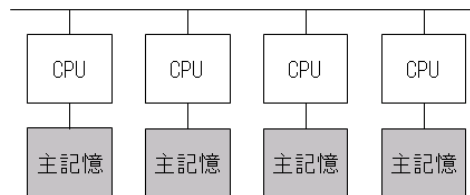
並列処理モデル

コンピュータアーキテクチャに応じた処理の分担(分割)のさせ方によって幾つかの並列処理がある

1. 共有メモリ並列処理



2. 分散メモリ並列処理



MPI(Message Passing Interface)は分散メモリ並列処理のための並列手法である

2. MPI概要

- **分散メモリ並列処理におけるメッセージパッシングの標準規格**
 - 複数のプロセス間でのデータをやり取りするために用いるメッセージ通信操作の仕様標準
- **FORTRAN, Cから呼び出すサブプログラムのライブラリ**
- **ポータビリティに優れている**
 - 標準化されたライブラリインターフェースによって、様々なMPI実装環境で同じソースをコンパイル・実行できる
- **プログラムの負担が比較的大きい**
 - プログラムを分析して、データ・処理を分割し、通信の内容とタイミングをユーザが自ら記述する必要がある
- **大容量のメモリ空間を利用可能**
 - 複数ノードを利用するプログラムの実行により、大きなメモリ空間を利用可能になる
(SQUIDのSX-Aurora TSUBASAは13.5TByteまで利用可能)

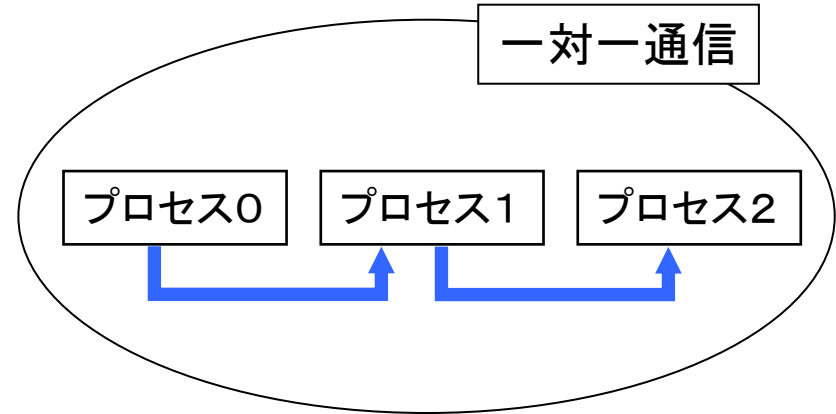
MPIの主な機能

プロセス管理

- MPIプログラムの初期化や終了処理などを行う

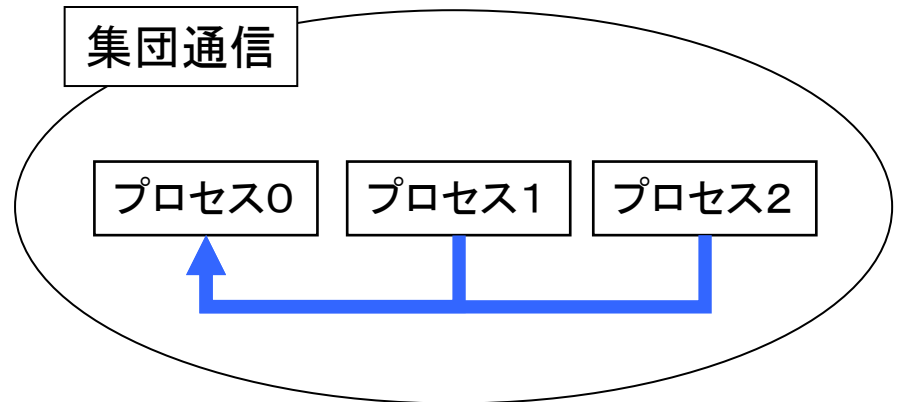
一対一通信

- 一対一で行う通信



集団通信

- グループ内のプロセス全体が関わる通信操作



MPIプログラムの基本構造

a.outのイメージ

```
PROGRAM MAIN  
CALL MPI_INIT(IERR)
```



MPI並列の対象

```
CALL MPI_FINALIZE(IERR)  
STOP  
END
```

- MPI_INITがcallされ, MPI_FINALIZEがcallされるまでの区間がMPI並列の対象
- MPI_INITがcallされた時点でプロセスが生成される(mpirunコマンドで指定するプロセス数. 下の例ではプロセス数は「4」)
- PROGRAM文以降で最初の処理の前(宣言文の後)でMPI_INITをcallする
- STOP文の直前にMPI_FINALIZEをcallする

実行例

```
%mpirun -np 4 ./a.out
```

MPI_INITは付録1.1.4参照

MPI_FINALIZEは付録1.1.5参照

MPIプログラムの基本構造

● プログラム実行時のプロセス数を得る

CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)

- mpirunコマンドで指定するプロセス数がNPROCSに返る
- ループの分割数を決める場合などに使用
- MPI_COMM_WORLDは「コミュニケータ」と呼ばれ、同じ通信の集まりを識別するフラグ
- 集団通信は同じコミュニケータを持つ集団間で行う

● プロセス番号を得る(プロセス番号は0から始まって、プロセス数-1まで)

CALL MPI_COMM_RANK(MPI_COMM_WORLD,MYRANK,IERR)

- 自プロセス番号がMYRANKに返る
 - プロセス番号は「ランク」とも呼ばれる
 - 特定のプロセスでのみ処理を実行する場合などに使用
- ```
if(myrank.eq.0) write(6.*)
```

MPI\_COMM\_SIZEは付録1.1.7参照

MPI\_COMM\_RANKは付録1.1.8参照

# コンパイル・実行コマンド

## MPIプログラムのコンパイル

```
mpifort [オプション] ソースファイル名
```

※オプションはnfortと同様.

## MPIプログラムの実行

```
mpirun -venode -np [総MPIプロセス数] ロードモジュール名
```

- -venode・・・ホストの指定をVHではなくVE単位で行います。

# 実行スクリプト例

## 160mpiのジョブを16VEで実行する際のスクリプト例

```
#!/bin/bash

#PBS -q SQUID
#PBS --group=【グループ名】
#PBS -l elapstim_req=1:00:00
#PBS --venode=16
#PBS -T necmpi

cd $PBS_O_WORKDIR
module load BaseVEC/20xx
mpirun -venode -np 160 ./a.out
```

\$PBS\_O\_WORKDIR  
ジョブスクリプトをqsubしたディレクトリ

「20xx」は最新バージョンを指定する

### NQS V オプション(#PBSで指定)

- q バッチリクエストを投入するキュー名を指定
- group 所属するグループ名を指定
- l 使用する資源制限値を指定
- T ジョブの種類を指定
- v (実行する全てのノードに対して) 環境変数を設定

参考: <http://www.hpc.cmc.osaka-u.ac.jp/system/manual/squid-use/jobscript/#vopt>

## ジョブクラス一覧(ベクトルノード群の一部)

| ジョブクラス  | 最大core数  | メモリ容量      | 利用方法 |
|---------|----------|------------|------|
| DBG     | 40       | 192GB      | 共有   |
| SQUID   | 2560     | 12TB       |      |
| mySQUID | 10*占有VE数 | 10GB*占有VE数 | 占有   |

参考: <http://www.hpc.cmc.osaka-u.ac.jp/system/manual/squid-use/jobclass/>

# MPIプログラム例 (Hello World)

```
program sample1
print *, "Hello World"
stop
end
```

逐次プログラム

sample1.f

```
program sample2
include 'mpif.h'
integer ierr,myrank,nprocs
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
print *, "Hello World My rank=",myrank,"(",nprocs,"processes)"
call MPI_FINALIZE(ierr)
stop
end
```

MPIプログラム

sample2.f

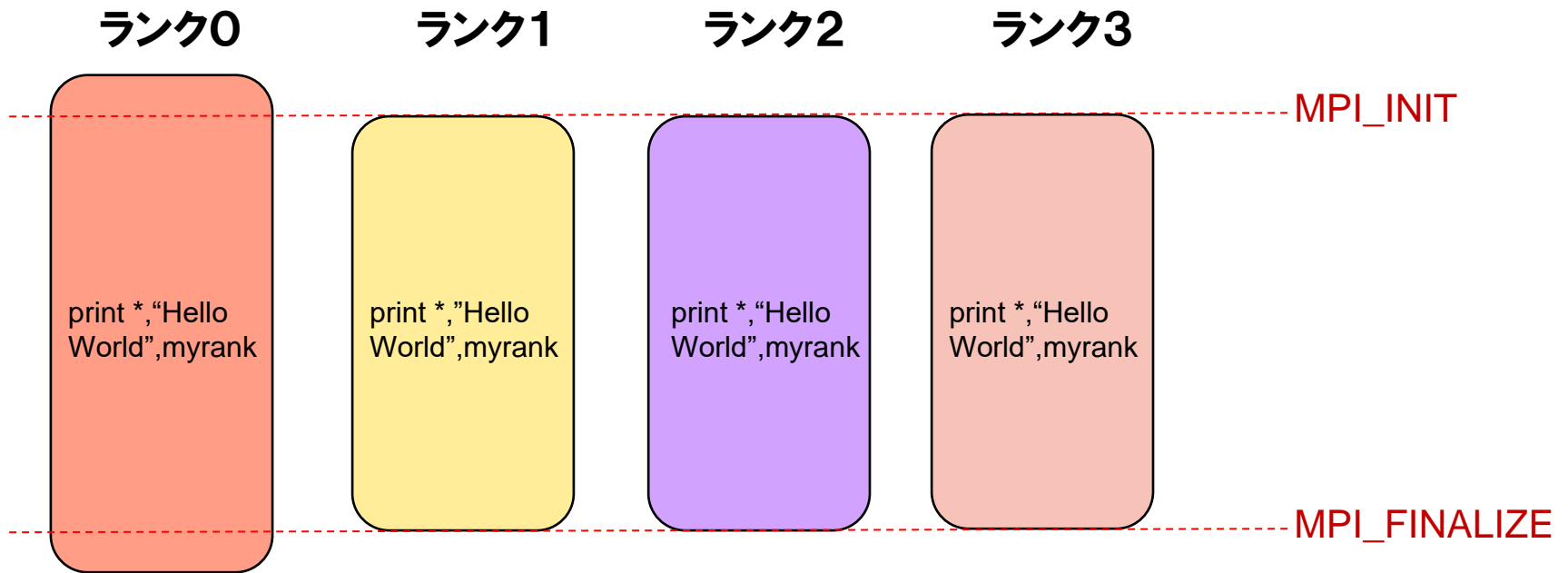
- MPIのインクルードファイルを指定する
- MPIの初期化 (MPI\_INIT)
- MPIの終了化 (MPI\_FINALIZE)

```
%mpirun -venode -np 4 ./a.out
Hello World My rank= 3 (4 processes)
Hello World My rank= 2 (4 processes)
Hello World My rank= 0 (4 processes)
Hello World My rank= 1 (4 processes)
```

- 4プロセスで実行
- 「Hello World」とランク番号,プロセス数が4回出力



# MPIプログラムの動作



- ① mpirunコマンドを実行 (-npサブオプションのプロセス数は4)
- ② ランク0のプロセスが生成
- ③ MPI\_INITをcallする時点でランク1,2,3のプロセスが生成
- ④ 各ランクで「print \*, "Hello World", myrank」が実行
- ⑤ 出力する順番はタイミングで決まる (ランク番号順ではない)

# 演習問題の構成

## ディレクトリ構成

MPI/

|-- practice\_1 演習問題1

|-- practice\_2 演習問題2

|-- practice\_3 演習問題3

|-- practice\_4 演習問題4

|-- practice\_5 演習問題5

|-- practice\_6 演習問題6

|-- sample テキスト内のsampleX.fとして掲載しているプログラム

|-- etc その他, テキスト内のetcX.fとして掲載しているプログラム

### 3. 演習問題1

---

■ P16 のプログラム (sample2.f) をコンパイル, 実行してください

■ P16 のMPIプログラム「Hello World」の結果をランク0のみが出力するように書き換えてください

### 3. 演習問題1-1 (practice\_1)

■ P16のプログラム (sample2.f) をpractice1.f としてコピーし、コンパイル・実行してください

#### ① ファイルのコピー

```
% cd MPI/practice_1
% cp ../sample/sample2.f practice1.f
```

#### ② コンパイル方法

```
% module load BaseVEC/20xx
% mpinfort practice1.f
```

「20xx」は最新バージョンを指定する

SX-Aurora TSUBASA用コンパイラ環境を設定する為のコマンド。  
ログイン後に1度実行することで、セッション中は設定が保存されます。

### 3. 演習問題1-1 (practice\_1) つづき

#### ③ 実行スクリプトの確認

```
% cat run.sh
```



```
#!/bin/bash
#PBS -q 【キュー名】
#PBS --group=【グループ名】
#PBS -l elapstim_req=00:03:00
#PBS -T necmpi
#PBS --venode=1
#PBS -v NMPI_PROGINF=detail
module load BaseVEC/20xx
cd ${PBS_O_WORKDIR}
mpirun -venode -np 4 ./a.out
```

「20xx」は最新バージョンを指定する

#### ④ ジョブの投入(実行)

```
% qsub run.sh
```

#### ⑤ 実行結果の確認

```
% cat run.sh.oXXXX
```

※XXXXはシステムにより付与されるジョブID

### 3. 演習問題1-2 (practice\_1)

■ 演習問題1-1で使ったMPIプログラム「Hello World」の結果をランク0のみが出力するように書き換えてください

① ファイルのコピー

```
% cp practice1.f practice1-2.f
```

② プログラムの編集

```
% vi practice1-2.f
```

③ コンパイル

```
% mpifort practice1-2.f
```

④ 実行

```
% qsub run.sh
```

解答例(P164)

# 総和プログラムのMPI化

- 1から1000の総和を求める (逐次実行プログラム)

sample3.f

```
program sample3
parameter (n=1000)
integer isum
isum=0
do i=1,n
 isum=isum+i
enddo
print *, "Total = ", isum
stop
end
```

← 総和計算

← 結果出力

# 総和プログラムのMPI化

- 逐次プログラム処理イメージ



- ・ 総和計算部分は, DOループ
- ・ 結果出力は, print文
  - 最後の1回だけ

- ・ 処理時間が一番大きいDOループが  
並列処理のターゲット



# 総和プログラムのMPI化

## 4分割の処理イメージ

|                              |
|------------------------------|
| <b>i=1, 250<br/>で和を取る</b>    |
| <b>i=251, 500<br/>で和を取る</b>  |
| <b>i=501, 750<br/>で和を取る</b>  |
| <b>i=751, 1000<br/>で和を取る</b> |
| 結果出力                         |

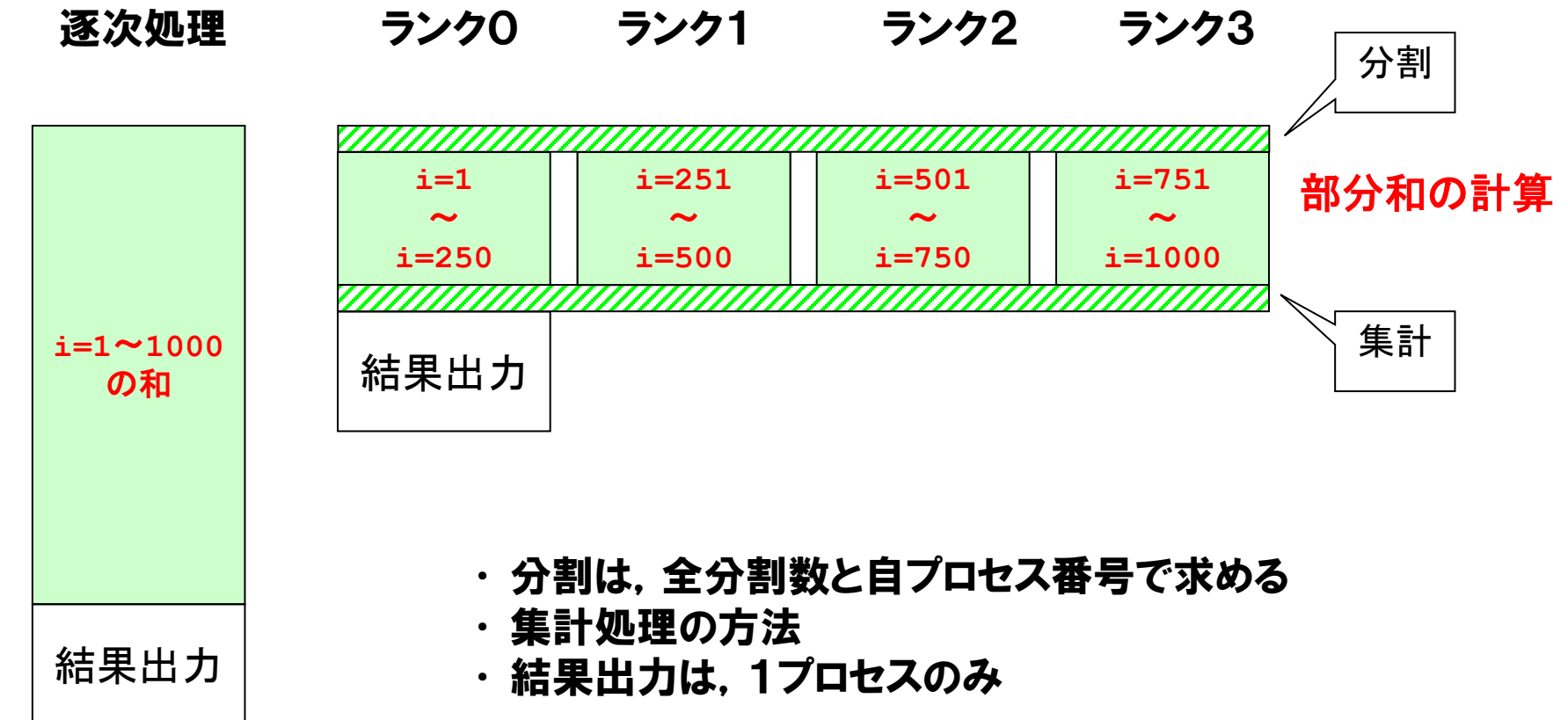
**i=1,1000までの和を取る処理は,**

**i= 1, 250までの和を取る処理  
i=251, 500までの和を取る処理  
i=501, 750までの和を取る処理  
i=751,1000までの和を取る処理**

**に分割することができる。  
しかも順不同。**

# 総和プログラムのMPI化

## ● 並列処理のイメージ (4分割)



# 総和プログラムのMPI化

- 分割の方法 (n=1000の場合)
  - 始点の求め方
    - $((n-1) / nprocs + 1) * myrank + 1$
  - 終点の求め方
    - $((n-1) / nprocs + 1) * (myrank + 1)$

但し、全分割数はnprocs, 自プロセス番号はmyrank  
本例は、nがプロセス数で割り切れることを前提としている

## 数値例

| nprocs=4 | 始点  | 終点   |
|----------|-----|------|
| myrank=0 | 1   | 250  |
| myrank=1 | 251 | 500  |
| myrank=2 | 501 | 750  |
| myrank=3 | 751 | 1000 |

## 4. 演習問題2

1から1000の総和を4分割してMPI並列で実行し、部分和を各ランクから出力してください

◆ ヒント:プログラムの流れは下記のとおり

MPIの初期化処理

プロセス数と自プロセスのランク番号の取得

分割時の始点と終点を求める

部分和に初期値(=0)を与える  
部分和を求めるループの実行

部分和の出力

MPIの終了化処理



## 4. 演習問題2 (practice\_2) つづき

### ① ディレクトリの移動

```
% cd MPI/practice_2
```

### ② プログラムの編集

```
% vi practice2.f
```

### ③ コンパイル

```
% mpifort practice2.f
```

### ④ 実行

```
% qsub run.sh
```

解答例(P165)

# MPIデータ転送

## ■ 各プロセスは独立したプログラムと考える

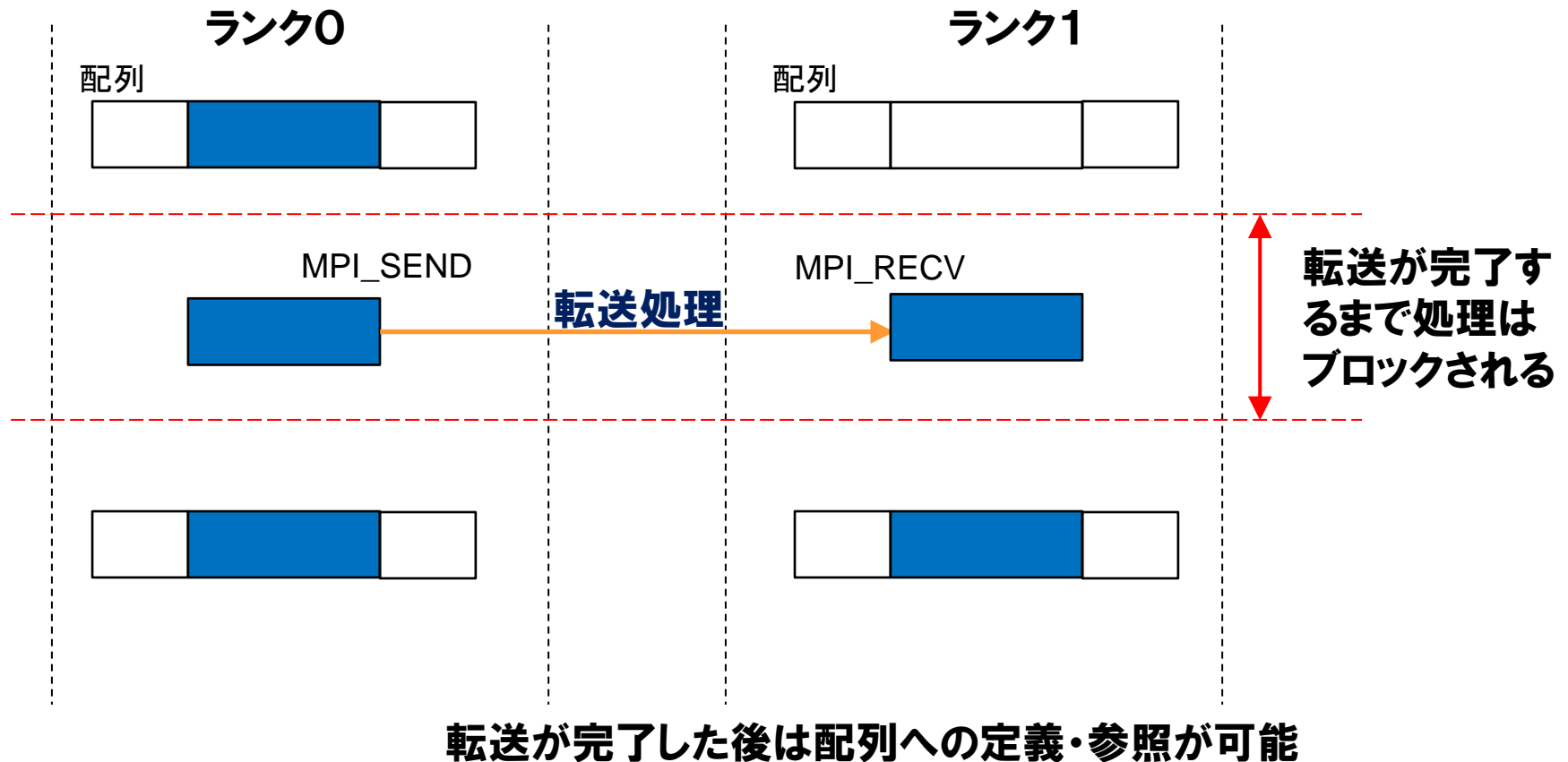
- 各プロセスは独立したメモリ空間を有する
- 他のプロセスのデータを直接アクセスすることは不可
- データ転送により他のプロセスのデータをアクセスすることが可能

## ■ MPI\_SEND/MPI\_RECV

- 同期型の1対1通信
- 特定のプロセス間でデータの送受信を行う
- データ転送が完了するまで処理は中断

# MPI\_SEND/MPI\_RECV

## ランク0の配列の一部部分をランク1へ転送



# MPI\_SEND/MPI\_RECV

sample4.f

```
program sample4
include 'mpif.h'
integer nprocs,myrank
integer status(MPI_STATUS_SIZE)
real work(10)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
itag=1
work=0.0
if(myrank.eq.0) then
 do i=1,10
 work(i)=float(i)
 enddo
 call MPI_SEND(work(4),3,MPI_REAL,1,itag,MPI_COMM_WORLD,ierr)
else if(myrank.eq.1) then
 call MPI_RECV(work(4),3,MPI_REAL,0,itag,MPI_COMM_WORLD,
+ status,ierr)
 write(6,*) work
endif
call MPI_FINALIZE(ierr)
stop
end
```

詳細は付録1.2.3

詳細は付録1.2.5



## 5. 演習問題3

**演習問題2のプログラムの各ランクの部分和をランク0に集めて、総和を計算し出力してください**

◆ ヒント: 転送処理は以下

ランク1,2,3(0以外)

```
call MPI_SEND(isum,1,MPI_INTEGER,0,
& itag,MPI_COMM_WORLD,ierr)
```

ランク0

```
call MPI_RECV(isum2,1,MPI_INTEGER,1,
& itag,MPI_COMM_WORLD,status,ierr)
call MPI_RECV(isum2,1,MPI_INTEGER,2,
& itag,MPI_COMM_WORLD,status,ierr)
call MPI_RECV(isum2,1,MPI_INTEGER,3,
& itag,MPI_COMM_WORLD,status,ierr)
```

※isumで受信するとランク0の部分和が上書きされてしまう

## 5. 演習問題3 (practice\_3) つづき

### ① ディレクトリの移動

```
% cd MPI/practice_3
```

### ② プログラムの編集(演習問題2の回答例を practice3.f として用意しています。)

```
% vi practice3.f
```

### ③ コンパイル

```
% mpinfort practice3.f
```

### ④ 実行

```
% qsub run.sh
```

解答例(P166)

# MPI集団通信

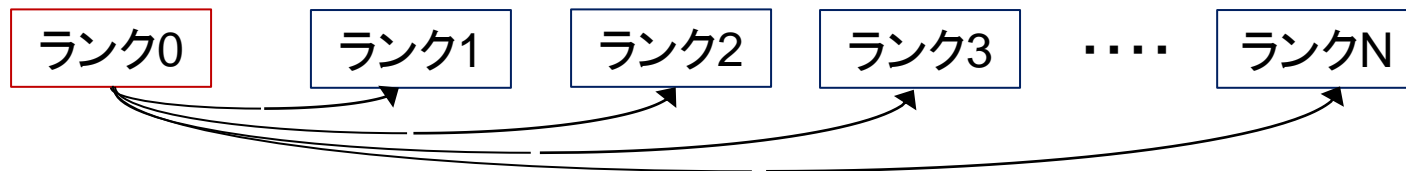
あるプロセスから同じコミュニケータを持つ全プロセスに対して同時に通信を行う

または同じコミュニケータを持つプロセス間でデータを共有する

(例) 代表プロセスのデータを同じコミュニケータを持つ全プロセスへ送信する

**CALL MPI\_BCAST (DATA,N,MPI\_REAL,0,MPI\_COMM\_WORLD,IERR)**

- N個の実数型データを格納するDATAをランク0 から送信
- コミュニケータMPI\_COMM\_WORLDを持つ全プロセスに送信される
- MPI\_BCASTがcallされる時に同期処理が発生 (通信に参加する全プロセスの足並みを揃える)



詳細は付録1.3.6

# MPI\_REDUCE

- 同じコミュニケータを持つプロセス間で総和、最大、最少などの演算を行い、結果を代表プロセスに返す

sample5.f

```
program sample5
include 'mpif.h'
integer myrank,nprocs
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_REDUCE(myrank,ism,1,MPI_INTEGER,MPI_SUM,0,
+ MPI_COMM_WORLD,ierr)
if(myrank.eq.0) write(6,*)"Result = ",ism
call MPI_FINALIZE(ierr)
stop
end
```

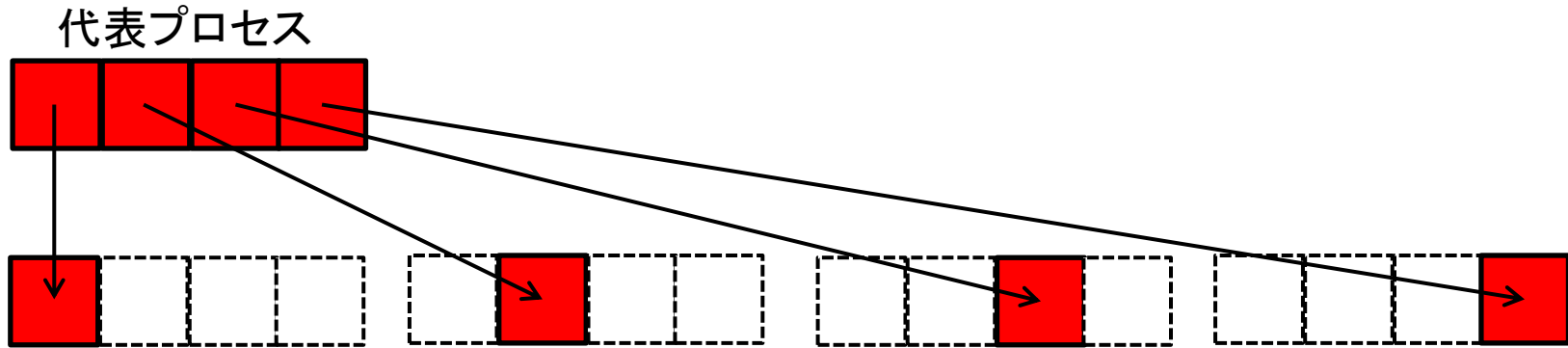
コミュニケータ  
MPI\_COMM\_WORLDを持つ  
プロセスのランク番号  
の合計をランク0に集計  
して出力する

詳細は付録1.3.3

```
%mpirun -venode -np 4 ./a.out
Result = 6
```

# MPI\_SCATTER

- 同じコミュニケータを持つプロセス内の代表プロセスの送信バッファから、全プロセスの受信バッファにメッセージを送信する。
- 各プロセスへのメッセージ長は一定である。



```
call MPI_SCATTER(senddata, icount, MPI_INTEGER,
& recvdata(icount*myrank+1), icount,
& MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
```

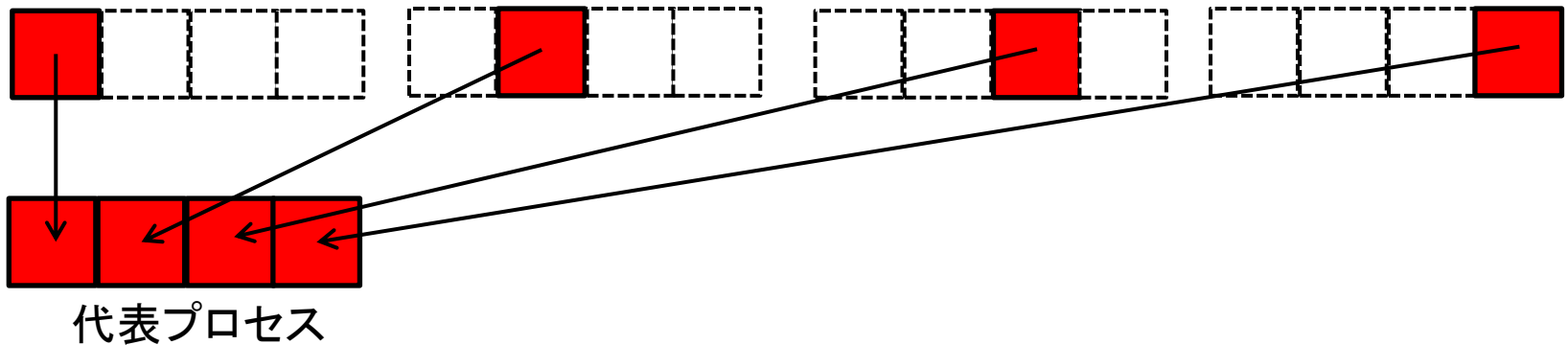
- 送信バッファと受信バッファはメモリ上の重なりがあってはならない (MPI1.0仕様)
- 各プロセスへのメッセージ長が一定でない場合はMPI\_SCATTERVを使用する。

MPI\_SCATTERの詳細は付録1.3.13

MPI\_SCATTERVの詳細は付録1.3.14

# MPI\_GATHER

- 同じコミュニケータを持つプロセス内の全プロセスの送信バッファから、代表プロセスの受信バッファにメッセージを送信する。
- 各プロセスからのメッセージ長は一定である。



```
call MPI_GATHER(senddata(icount*myrank+1),
& icount, MPI_INTEGER, recvdata,
& icount, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
```

- 送信バッファと受信バッファはメモリ上の重なりがあってはならない (MPI1.0仕様)
- 各プロセスへのメッセージ長が一定でない場合はMPI\_GATHERVを使用する。

MPI\_GATHERの詳細は付録1.3.8

MPI\_GATHERVの詳細は付録1.3.9

## 6. 演習問題4

■ 演習問題3のプログラムで、各ランクの部分和をMPI\_REDUCEを使用してランク0に集計して、ランク0から結果を出力してください

① ディレクトリの移動

```
% cd MPI/practice_4
```

② プログラムの編集(演習問題3の回答例を practice4.f として用意しています。)

```
% vi practice4.f
```

③ コンパイル

```
% mpifort practice4.f
```

④ 実行

```
% qsub run.sh
```

解答例(P168)

# 7. MPIプログラミング

---

**7.1 並列化の対象**

**7.2 空間分割の種類**

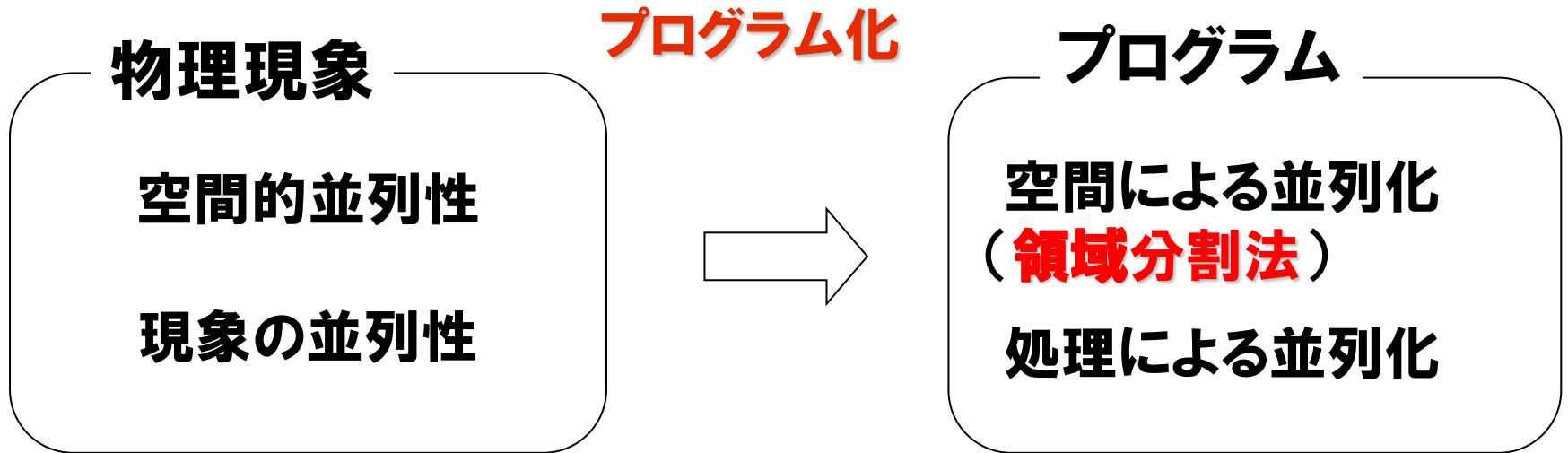
**7.3 通信の発生**

**7.4 配列の縮小**

**7.5 ファイルの入出力**

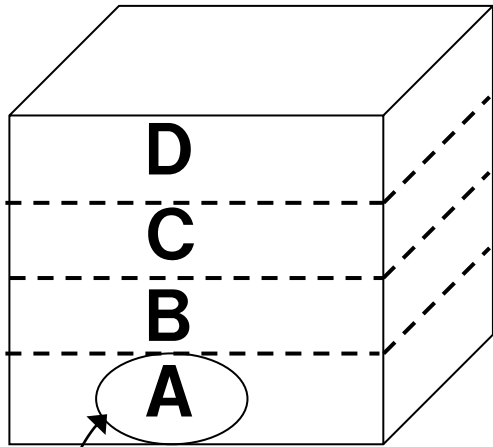


# 7.1 並列化の対象

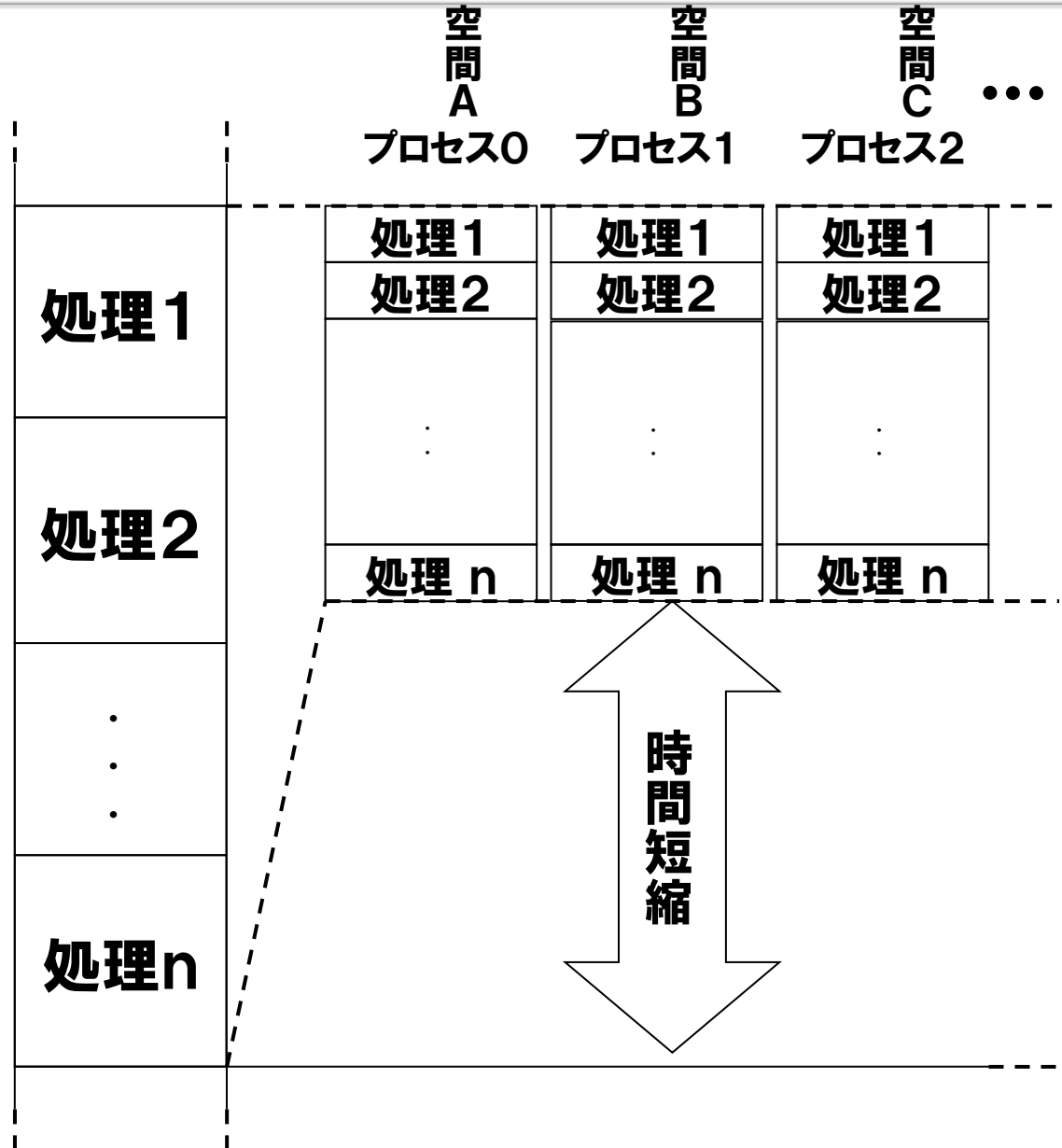


# 空間による並列化(イメージ)

## 領域分割法



各々, CPUに割り当てる

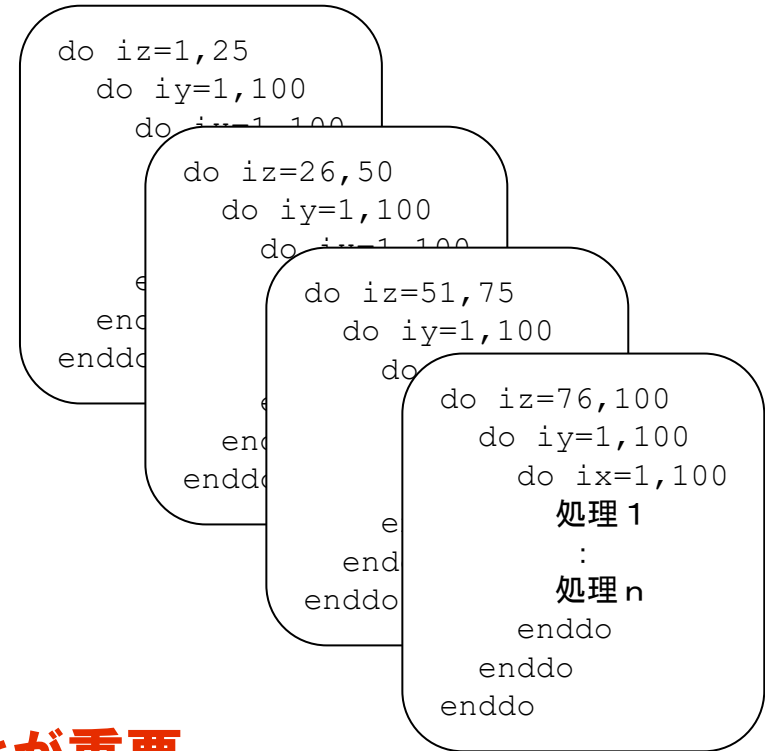
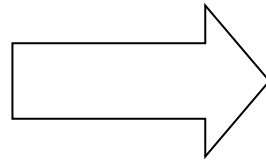


# 空間による並列化の例

## DOループ(FORTRAN)単位での並列処理

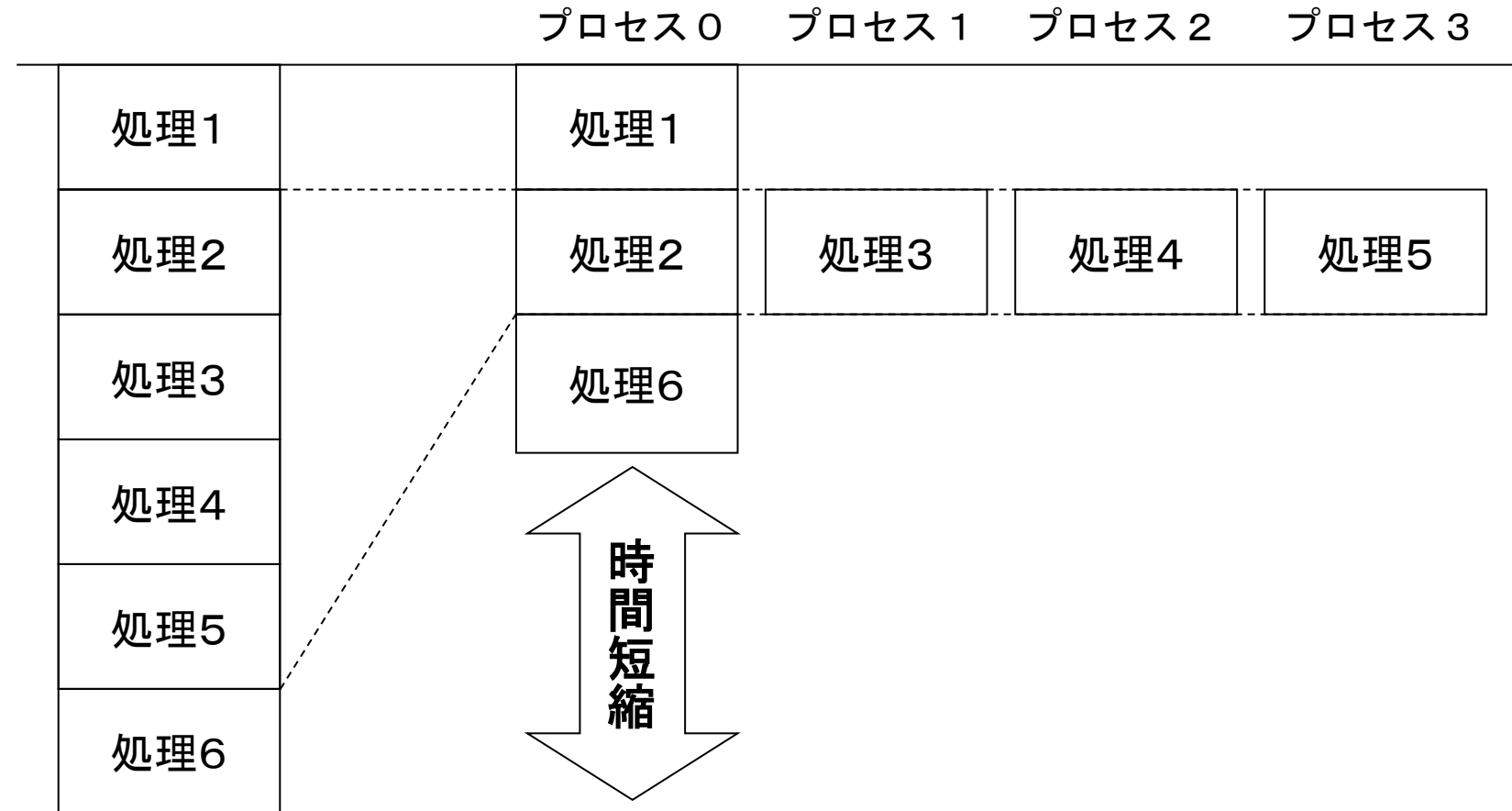
### 例)領域分割法

```
do iz=1,100
 do iy=1,100
 do ix=1,100
 処理1
 :
 処理n
 enddo
 enddo
enddo
```



より外側のループで並列化することが重要

# 処理による並列化(イメージ)

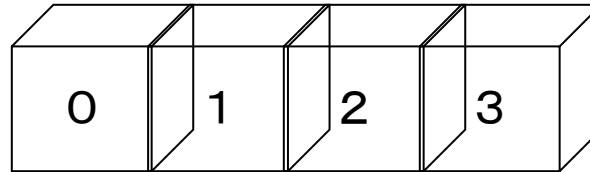


## 7.2 空間分割の分類

### ① ブロック分割

- 空間を分割数の塊に分割する

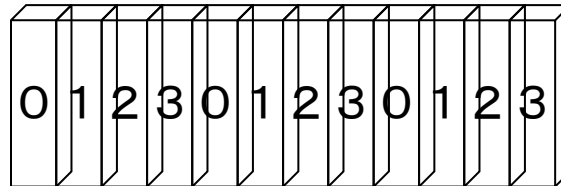
例)4分割



### ② サイクリック分割

- 帯状に細分し, 巡回的に番号付ける

例)4分割



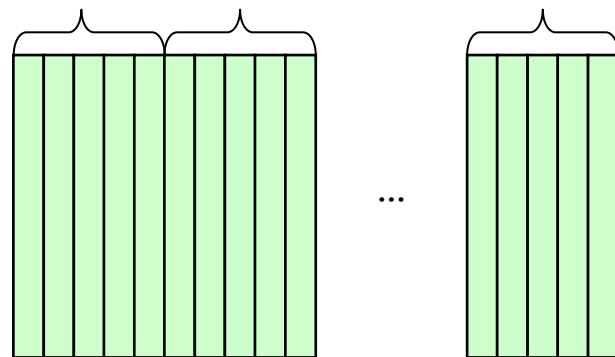
# ブロック分割

## 処理量が均等なループを分割する場合

```
do i=1,100
...
enddo
```

繰り返し数をプロセス  
毎に均等に割り当てる

プロセス: 0 1 ... nproc-1



```
do i=1,25
...
enddo
```

```
do i=26,50
...
enddo
```

```
do i=51,75
...
enddo
```

```
do i=76,100
...
enddo
```

# サイクリック分割

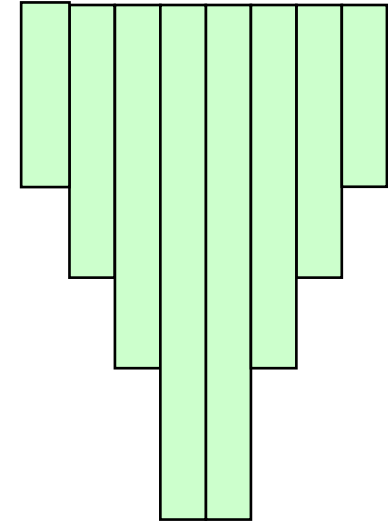
## 処理量が不均等なループを分割する場合

```
do i=1,n
...
enddo
```

繰り返し数をプロセス毎に  
巡回的に割り当てる

```
common /commpi/ myrank,nprocs
do i=myrank+1,n,nprocs
...
enddo
```

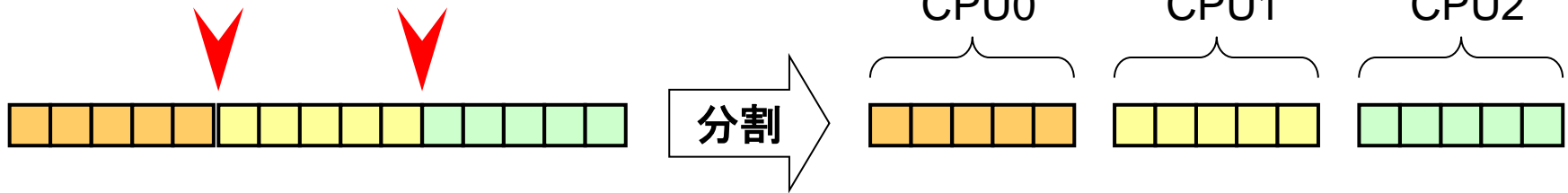
プロセス: 0 1 2 3 0 1 2 3



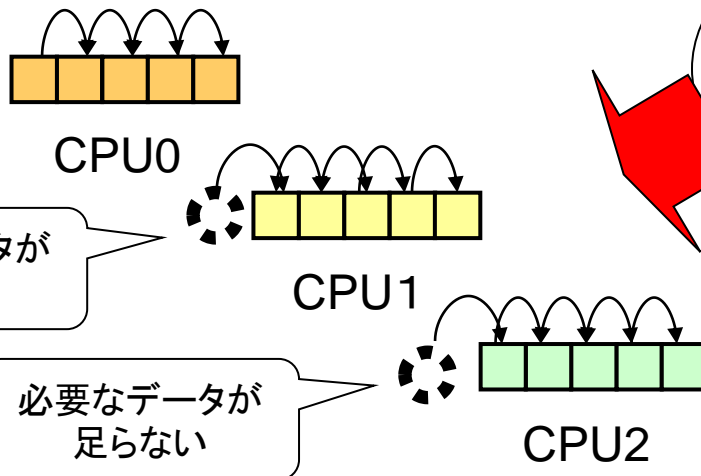
# 7.3 通信の発生

## 袖領域

配列を3つの配列に分割すると、



しかし、



他のCPUが保持するデータを参照する仕組みが必要

データの送受信



# 境界を跨ぐ例

■ 対象のDOループに含まれる配列の添え字が $i+1$ や $i-1$ の場合、ループを分割した時にできる境界を跨ぐ

逐次版

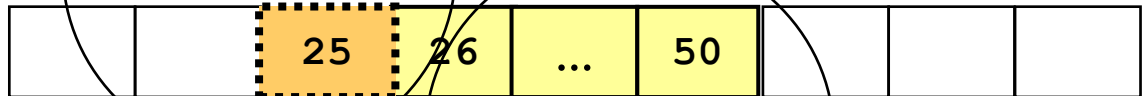
```
do i=1, 100
 b(i)=a(i)-a(i-1)
enddo
```

並列版

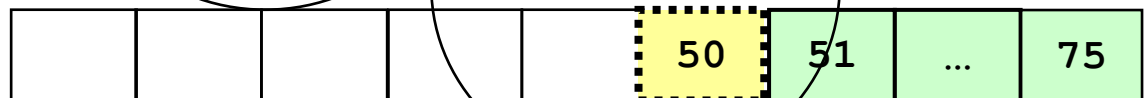
プロセス0



プロセス1

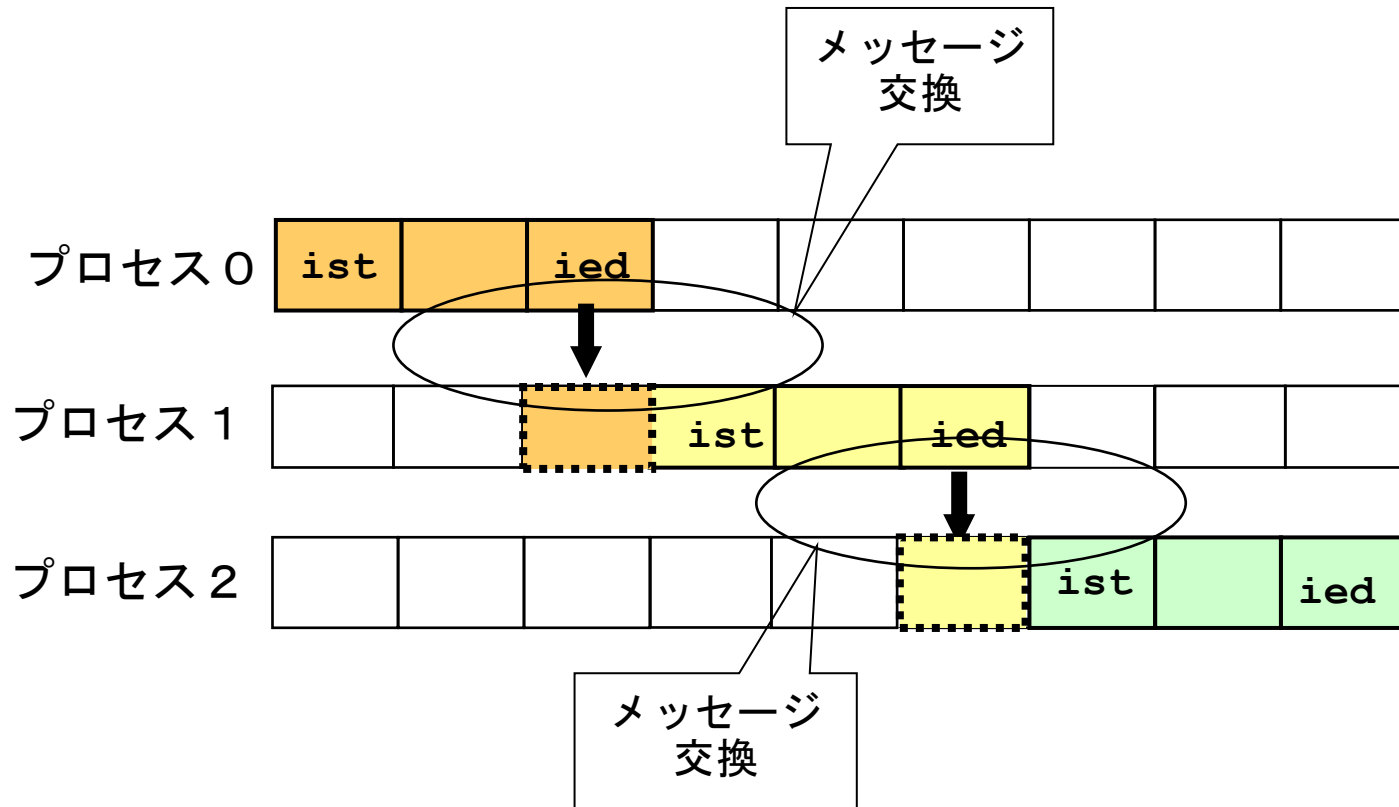


プロセス2



# 不足データの送受信

分割境界におけるデータを補うには、メッセージ交換によるデータの送受信が必要



# 領域分割時のメッセージ交換

MPI版

```

:
ist = ((100-1)/nprocs+1)*myrank+1
ied = ((100-1)/nprocs+1)*(myrank+1)
iLF = myrank-1
iRT = myrank+1
if (myrank.ne.0) then
 call mpi_recv(a(ist-1),1,MPI_REAL8,iLF,1, &
 MPI_COMM_WORLD,status,ierr)
endif
do i= ist, ied
 b(i) = a(i) - a(i-1)
enddo
if (myrank.ne.nprocs-1) then
 call mpi_send(a(ied),1,MPI_REAL8,iRT,1, &
 MPI_COMM_WORLD,ierr)
endif
:
```

担当領域の  
算出

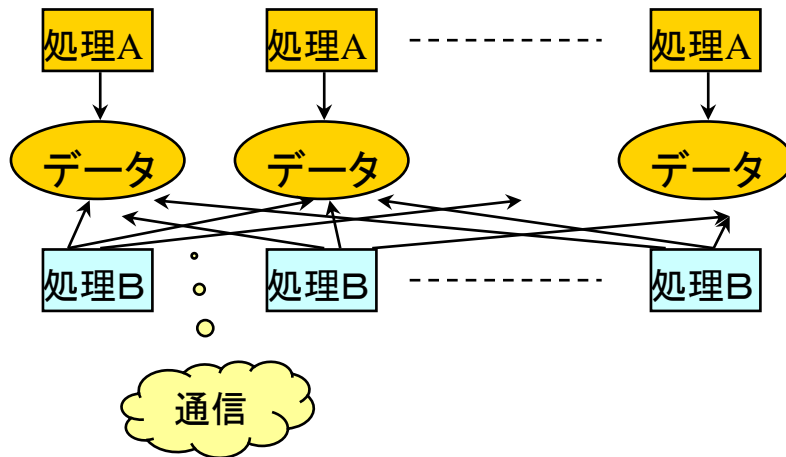
送受信相手の  
特定

# アクセス方法が変わる例

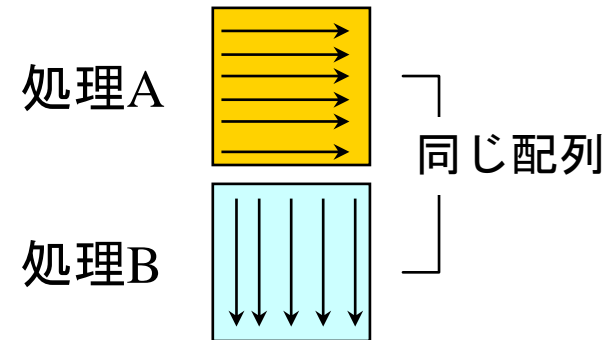
## ・ データ分割

- 分割後の処理と、これを扱うデータの分割が必ずしも一致しない→**データ通信が必要**

分散メモリ型コンピュータ



アクセス方法

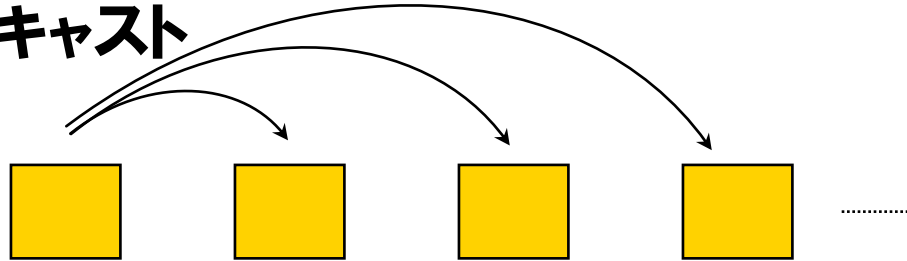


# 主なデータ通信のパターン

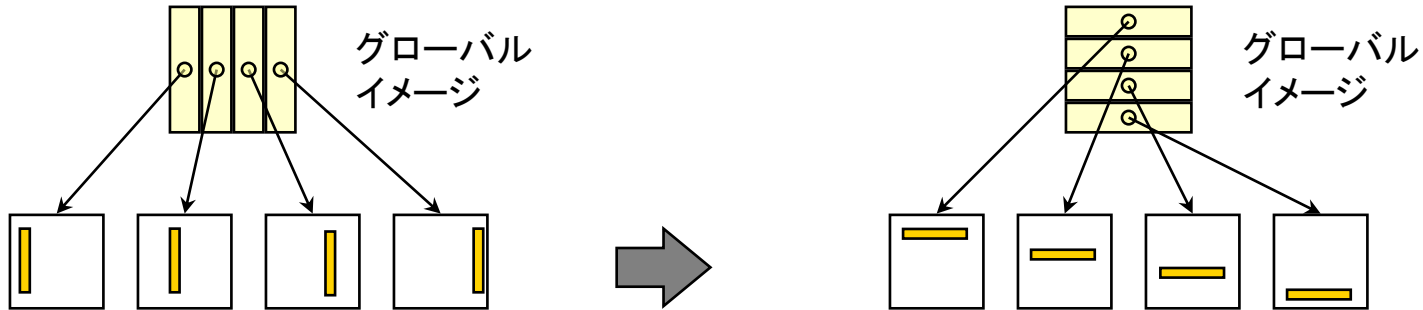
## ・ シフト通信



## ・ ブロードキャスト



## ・ 転置



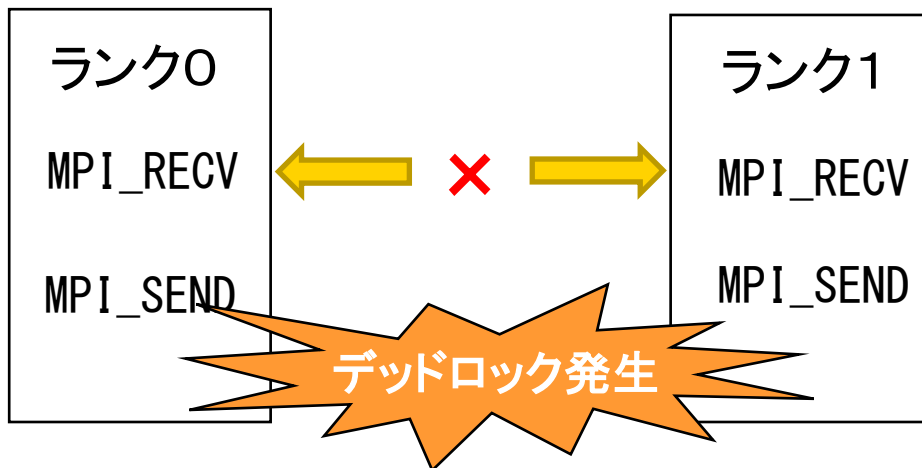
**MPIは、通信をプログラム上で明示的に記述**

# デッドロック

```
if(myrank.eq.0) then
 call MPI_Recv(rdata,1,MPI_REAL,1,
+ itag,MPI_COMM_WORLD,status,ierr)
else if(myrank.eq.1) then
 call MPI_Recv(rdata,1,MPI_REAL,0,
+ itag,MPI_COMM_WORLD,status,ierr)
endif
if(myrank.eq.0) then
 call MPI_Send(sdata,1,MPI_REAL,1,
+ itag,MPI_COMM_WORLD,ierr)
else if(myrank.eq.1) then
 call MPI_Send(sdata,1,MPI_REAL,0,
+ itag,MPI_COMM_WORLD,ierr)
endif
```

- ランク0とランク1が同時にMPI\_RECV (同期型1対1通信) を行くと、送受信が完了せず、待ち状態となる。
- このような待ち状態をデッドロックという。

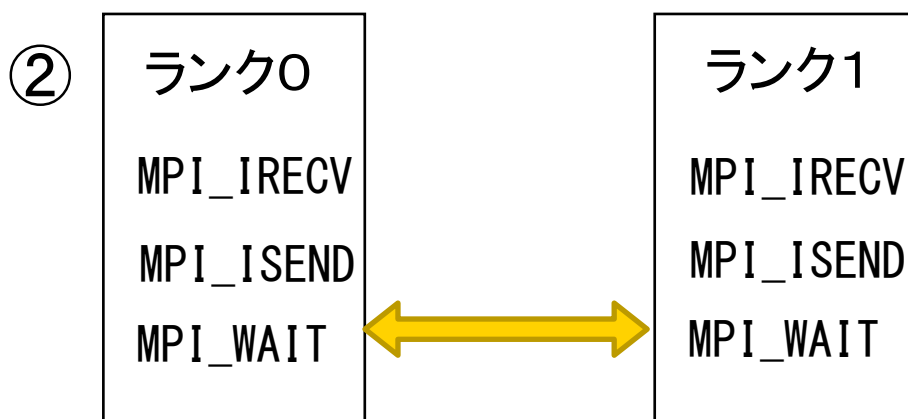
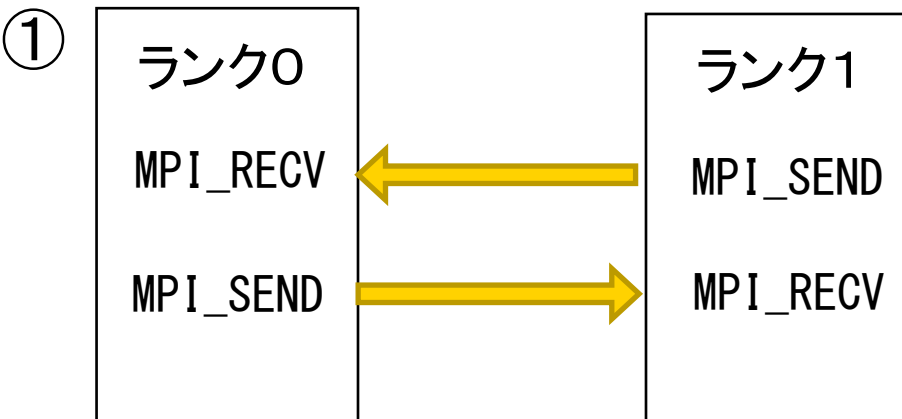
# デッドロック



※ランク0とランク1から同時にMPI\_RECVを実行するとデータが送信されるのを待つ状態で止まってしまう。

● デッドロックの回避方法としては、以下が挙げられる。

- ① MPI\_RECVとMPI\_SENDの正しい呼び出し順序に修正
- ② 非同期型にMPI\_IRecvとMPI\_Isendに置き換える



# デッドロックの回避①

```
if(myrank.eq.0) then
 call MPI_Recv(rdata,1,MPI_REAL,1,
+ itag,MPI_COMM_WORLD,status,ierr)
else if(myrank.eq.1) then
 call MPI_Send(sdata,1,MPI_REAL,0,
+ itag,MPI_COMM_WORLD,ierr)
endif
if(myrank.eq.0) then
 call MPI_Send(sdata,1,MPI_REAL,1,
+ itag,MPI_COMM_WORLD,ierr)
else if(myrank.eq.1) then
 call MPI_Recv(rdata,1,MPI_REAL,0,
+ itag,MPI_COMM_WORLD,status,ierr)
endif
```

- MPI\_SENDとMPI\_RECVが対になるように呼び出し順序を変更



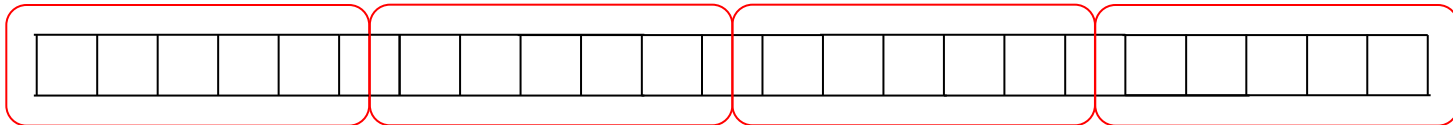
## デッドロックの回避②

```
if(myrank.eq.0) then
 call MPI_IRecv(rdata,1,MPI_REAL,1,
+ itag,MPI_COMM_WORLD,ireq1,ierr)
else if(myrank.eq.1) then
 call MPI_IRecv(rdata,1,MPI_REAL,0,
+ itag,MPI_COMM_WORLD,ireq1,ierr)
endif
if(myrank.eq.0) then
 call MPI_ISend(sdata,1,MPI_REAL,1,
+ itag,MPI_COMM_WORLD,ireq2,ierr)
else if(myrank.eq.1) then
 call MPI_ISend(sdata,1,MPI_REAL,0,
+ itag,MPI_COMM_WORLD,ireq2,ierr)
endif
call MPI_WAIT(ireq1,status,ierr)
call MPI_WAIT(ireq2,status,ierr)
```

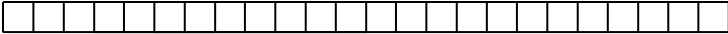
- 非同期型のMPI\_ISENDとMPI\_IRecvに置き換える
- MPI\_ISENDとMPI\_IRecv,MPI\_WAITの詳細は付録1.2.7～10

# 7.4 配列の縮小

配列a (100)



 ... 各プロセスが担当する領域

各プロセスは, 

全体の配列を持つ必要がない



**メモリ領域の節約ができる**

# 縮小イメージ(内積)

## プロセス0

```
real(8)::a(100)
do i=1,25
 c=c+a(i)*b(i)
enddo
```

## プロセス1

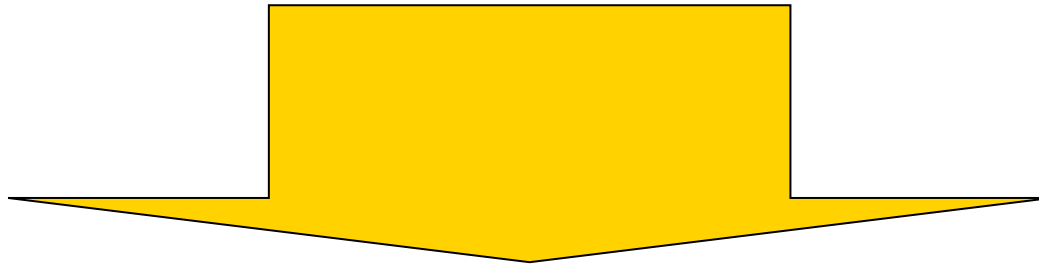
```
real(8)::a(100)
do i=26,50
 c=c+a(i)*b(i)
enddo
```

## プロセス2

```
real(8)::a(100)
do i=51,75
 c=c+a(i)*b(i)
enddo
```

## プロセス3

```
real(8)::a(100)
do i=76,100
 c=c+a(i)*b(i)
enddo
```



```
real(8)::a(25)
do i=1,25
 c=c+a(i)*b(i)
enddo
```

## プロセス0

```
real(8)::a(25)
do i=1,25
 c=c+a(i)*b(i)
enddo
```

## プロセス1

```
real(8)::a(25)
do i=1,25
 c=c+a(i)*b(i)
enddo
```

## プロセス2

```
real(8)::a(25)
do i=1,25
 c=c*a(i)*b(i)
enddo
```

## プロセス3

## 7.5 ファイル入出力

MPIによって並列化されたプログラムのファイル入出力にはいくつかのパターンがあり、それぞれに特徴があるため、実際のプログラム例を記載する。

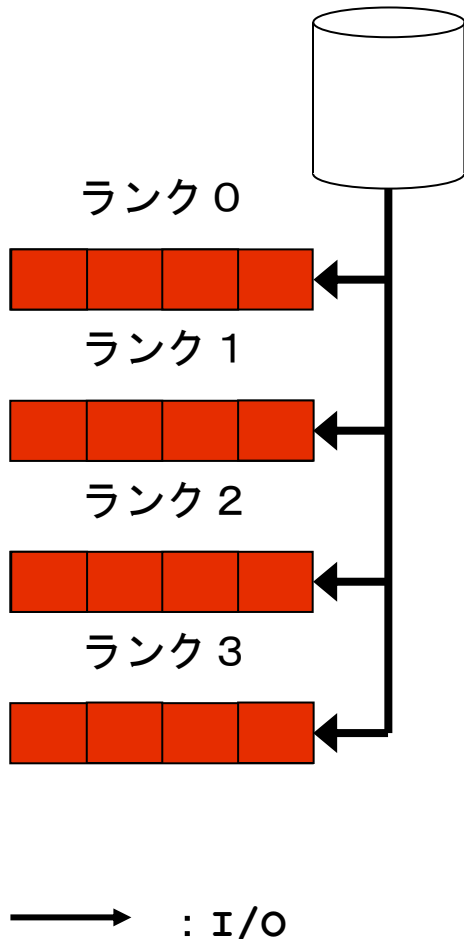
### 1. ファイル入力

- ① 全プロセス同一ファイル入力
  - 逐次プログラムから移行し易い
- ② 代表プロセス入力
  - メモリの削減が可能
- ③ 分散ファイル入力
  - メモリ削減に加え、I/O時間の削減が可能

### 2. ファイル出力

- ① 代表プロセス出力
  - ファイルを1つにまとめる
- ② 分散ファイル出力
  - I/O時間の削減が可能

# 全プロセス同一ファイル入力

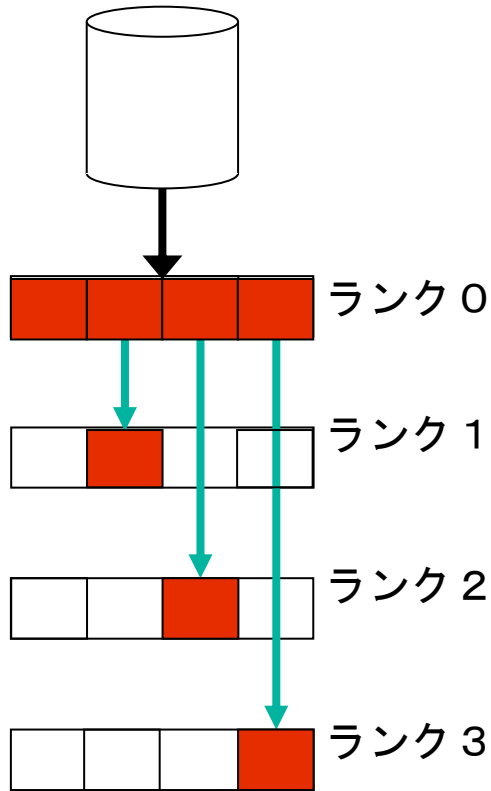


etc1.f

```
include 'mpif.h'
integer,parameter::numdat=100
integer::idat(numdat)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
ist=((numdat-1)/nprocs+1)*myrank+1
ied=((numdat-1)/nprocs+1)*(myrank+1)
open(10,file='fort.10')
read(10,*) idat
isum=0
do i=ist,ied
 isum=isum+idat(i)
enddo
write(6,*) myrank,':partial sum=',isum
call MPI_FINALIZE(ierr)
stop
end
```

# 代表プロセス入力

etc2.f

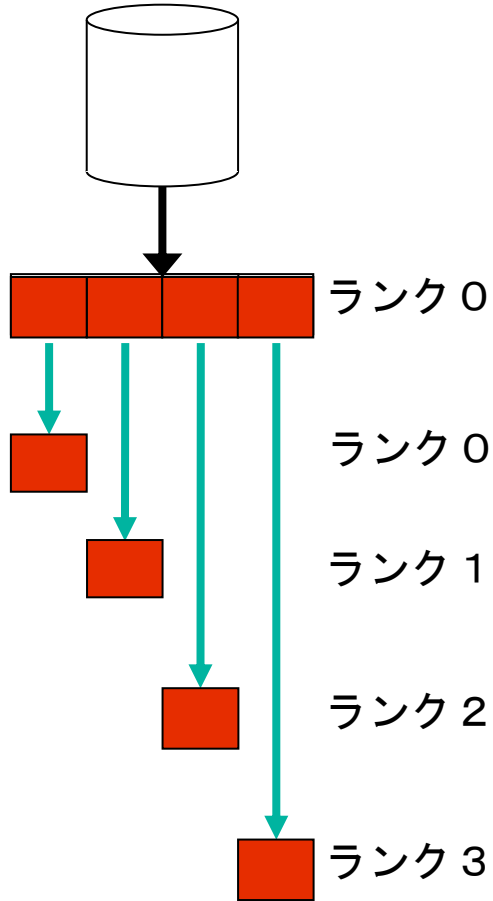


→ : MPI通信  
→ : I/O

```
include 'mpif.h'
integer,parameter :: numdat=100
integer::senddata(numdat),recvdata(numdat)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
if (myrank.eq.0) then
 open(10,file='fort.10')
 read(10,*) senddata
endif
icount=(numdat-1)/nprocs+1
call MPI_SCATTER(senddata,icount,MPI_INTEGER,
& recvdata(icount*myrank+1),icount,
& MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
isum=0
do i=1,icount
 isum=isum+recvdata(icount*myrank+i)
enddo
write(6,*)myrank,':partial sum=',isum
call MPI_FINALIZE(ierr)
stop
end
```

**MPI\_SCATTERの詳細は付録1.3.13**

# 代表プロセス入力+メモリ削減

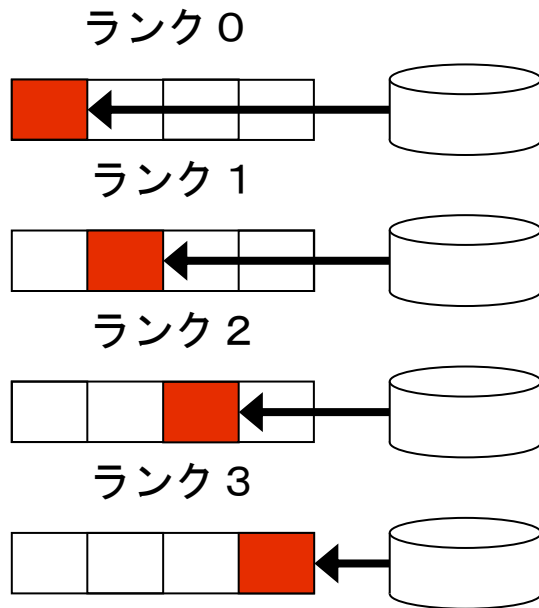


→ : MPI通信

→ : I/O

```
include 'mpif.h'
integer,parameter :: numdat=100
integer,allocatable :: idat(:),work(:)
integer :: nprocs,myrank,ierr
integer :: ist,ied
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
ist = ((numdat-1)/nprocs+1)*myrank+1
ied = ((numdat-1)/nprocs+1)*(myrank+1)
allocate(idat(ist:ied))
if(myrank.eq.0) then
 allocate(work(numdat))
 open(10,file='fort.10')
 read(10,*) work
endif
call MPI_SCATTER(work,ied-ist+1,MPI_INTEGER,
+ idat(ist),ied-ist+1,MPI_INTEGER,0,
+ MPI_COMM_WORLD,ierr)
if(myrank.eq.0) deallocate(work)
isum=0
do i=ist,ied
 isum = isum + idat(i)
enddo
write(6,*) myrank,';partial sum=',isum
call MPI_FINALIZE(ierr)
stop
end
```

# 分散ファイル入力



→ : I/O

etc3.f

```
include 'mpif.h'
integer, parameter :: numdat=100
integer :: buf(numdat)

c
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

c
ist=((numdat-1)/nprocs+1)*myrank+1
ied=((numdat-1)/nprocs+1)*(myrank+1)
read(10+myrank, *) (buf(i), i=ist, ied)

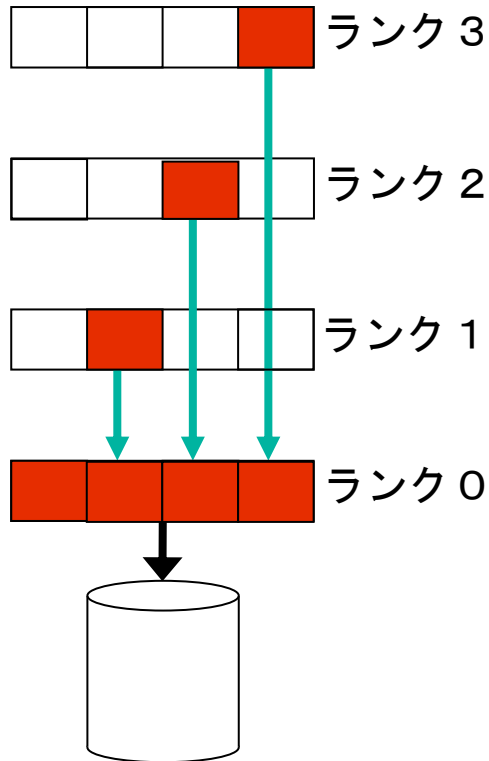
c
isum=0
do i=ist, ied
 isum = isum + buf(i)
enddo

c
write(6, *) myrank, ' : partial sum=', isum
call MPI_FINALIZE(ierr)
stop
end
```



# 代表プロセス出力

etc4.f

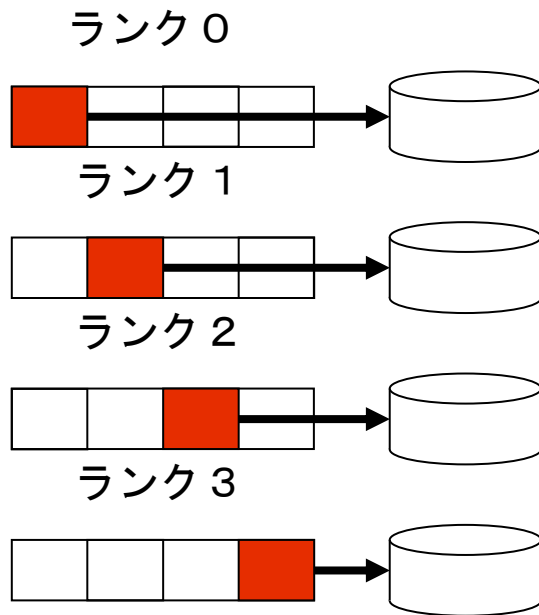


→ : MPI通信  
→ : I/O

```
include 'mpif.h'
parameter (numdat=100)
integer senddata(numdat), recvdata(numdat)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
icount=(numdat-1)/nprocs+1
do i=1,icount
 senddata(icount*myrank+i)=icount*myrank+i
enddo
call MPI_GATHER(senddata(icount*myrank+1),
& icount,MPI_INTEGER,recvdata,
& icount,MPI_INTEGER,0,MPI_COMM_WORLD,
& ierr)
if (myrank.eq.0) then
 open(60,file='fort.60')
 write(60,'(10I8)') recvdata
endif
call MPI_FINALIZE(ierr)
stop
end
```

**MPI\_GATHERの詳細は付録1.3.8**

# 分散ファイル出力



→ : I/O

etc5.f

```
include 'mpif.h'
integer, parameter :: numdat=100
integer :: buf(numdat)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
ist=((numdat-1)/nprocs+1)*myrank+1
ied=((numdat-1)/nprocs+1)*(myrank+1)
do i=ist, ied
 buf(i)=i
enddo
write(60+myrank, '(10I8)') (buf(i), i=ist, ied)
call MPI_FINALIZE(ierr)
stop
end
```

## 8. 演習問題5

■ P65のetc4.fをP63の「代表プロセス入力＋メモリ削減」の例のように、各プロセスに必要な領域だけ確保するように修正してください。

◆ ヒント:

- ① senddata,recvdataを動的に確保するようにallocatable宣言する
- ② 各プロセスが確保する領域(ist,jed)を求める
- ③ 各プロセスで必要なsenddataの領域を確保する(allocate)
- ④ ランク0でrecvdataの領域を確保する(allocate)

## 8. 演習問題5 (practice\_5) つづき

### ① ディレクトリの移動

```
% cd MPI/practice_5
```

### ② プログラムの編集

```
% vi practice5.f
```

### ③ コンパイル

```
% mpinfort practice5.f
```

### ④ 実行

```
% qsub run.sh
```

解答例(P170)

# 9. 実行方法と性能解析

---

9.1 サイバーメディアセンターのコンピュータ

9.2 SX-Aurora TSUBASAのコンパイル・実行

9.3 SX-Aurora TSUBASAにおける環境変数

9.4 SX-Aurora TSUBASAの簡易性能解析機能

9.5 NEC Ftrace Viewer

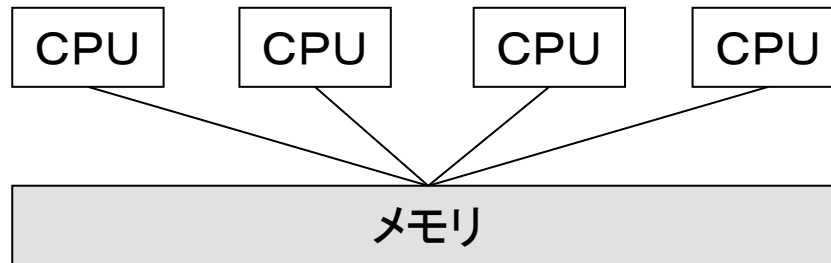
# 9.1 サイバーメディアセンターのコンピュータ

## ■シングルノード型

■ 複数のCPUが同一のメモリを参照できる

- SX-Aurora TSUBASA
- PCクラスタ

■ コンパイラによる自動並列化機能やOpenMPなどが利用できる



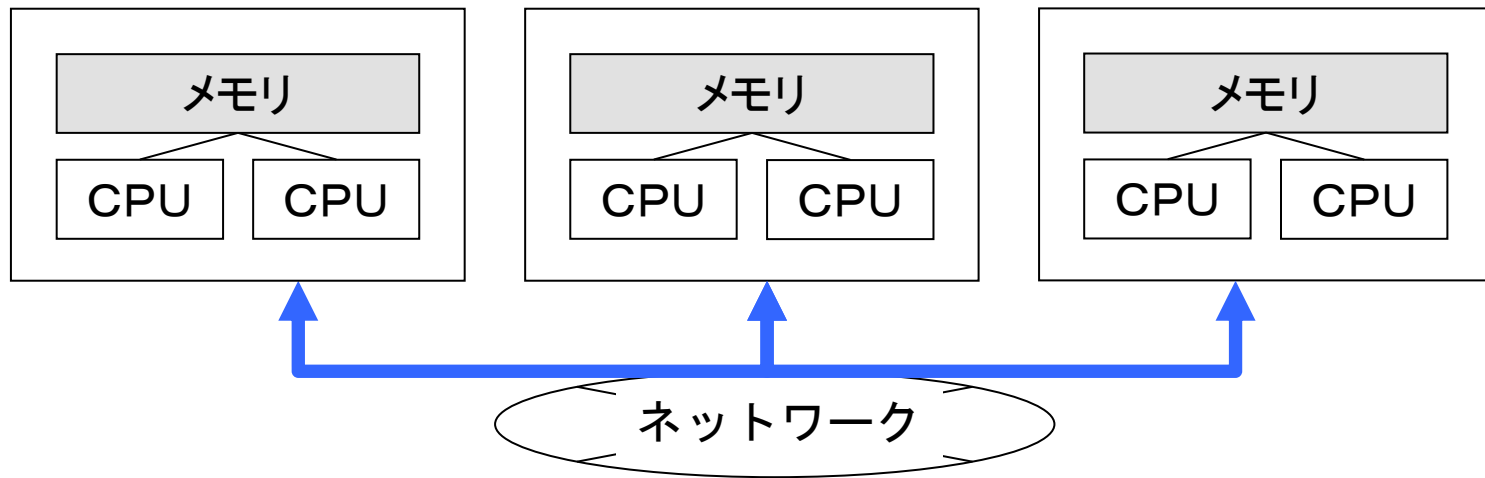
# 9.1 サイバーメディアセンターのコンピュータ

## ■マルチノード型(SMPクラスタ)

■ 複数の共有メモリ型コンピュータがネットワークを介して接続されている

- SX-Aurora TSUBASA
- PCクラスタ

■ SMP間は、MPIによる並列化を行う



## 9.2 SX-Aurora TSUBASAのコンパイル・実行

### ■ SX-Aurora TSUBASAのコンパイル方法

#### ① 環境設定の読み込み

SQUIDでは、コンパイラ、ライブラリ、アプリケーション環境などの環境変数設定を Environment modules というツールで管理しています。まずは以下のコマンドを実行して、ベクトルノード向けの推奨環境を読み込んでください。

```
% module load BaseVEC/20xx
```

「20xx」は最新バージョンを指定する

#### ② コンパイル方法

フロントエンド上で以下のコマンドを実行してください。

```
% mpinfort オプション MPIソースファイル名
```



## 9.3 SX-Aurora TSUBASAにおける環境変数

### ■NMPI\_PROGINF

■ 実行性能情報をMPIプロセス毎に詳細に表示させたり、全MPIプロセスの情報を集計編集して表示させることが可能

■ 表示は、MPIプログラムの実行において、MPI\_FINALIZE手続きを呼び出した際にMPI\_COMM\_WORLD (MPIUNIVERSE=0) のランク0のMPIプロセスから標準エラー出力に対して行われる

■ NMPI\_PROGINFの値と表示内容は以下の通り

- NO 性能情報を出力しない(既定値)
- YES 性能情報を集約形式で出力
- DETAIL 性能情報を詳細集約形式で出力
- ALL 性能情報を拡張形式で出力
- ALL\_DETAIL 性能情報を詳細拡張形式で出力

# NMPI\_PROGINF出力例(Detail指定時)

| Global Data of 4 Vector processes |                                     | Min [U, R]           | Max [U, R]           | Average       |
|-----------------------------------|-------------------------------------|----------------------|----------------------|---------------|
| =====                             |                                     |                      |                      |               |
| a.                                | Real Time (sec)                     | 16.970 [0, 3]        | 16.970 [0, 0]        | 16.970        |
| b.                                | User Time (sec)                     | 16.965 [0, 0]        | 16.965 [0, 1]        | 16.965        |
| c.                                | Vector Time (sec)                   | 16.796 [0, 0]        | 16.963 [0, 1]        | 16.874        |
| d.                                | Inst. Count                         | 17053548337 [0, 1]   | 17440762295 [0, 0]   | 17263359528   |
| e.                                | V. Inst. Count                      | 4198659475 [0, 1]    | 4204884402 [0, 0]    | 4201854256    |
| f.                                | V. Element Count                    | 1074848210983 [0, 1] | 1075027941241 [0, 0] | 1074899697744 |
| g.                                | V. Load Element Count               | 536897217651 [0, 1]  | 536990823082 [0, 0]  | 536924988473  |
| h.                                | FLOP Count                          | 536870912015 [0, 1]  | 536870912208 [0, 0]  | 536870912063  |
| i.                                | MOPS                                | 79940.380 [0, 1]     | 79974.629 [0, 0]     | 79955.934     |
| j.                                | MOPS (Real)                         | 79914.100 [0, 1]     | 79946.632 [0, 0]     | 79929.576     |
| k.                                | MFLOPS                              | 31646.962 [0, 1]     | 31647.435 [0, 0]     | 31647.096     |
| l.                                | MFLOPS (Real)                       | 31636.356 [0, 0]     | 31636.995 [0, 3]     | 31636.663     |
| m.                                | A. V. Length                        | 255.662 [0, 0]       | 255.998 [0, 1]       | 255.816       |
| n.                                | V. Op. Ratio (%)                    | 99.024 [0, 0]        | 99.052 [0, 1]        | 99.037        |
| o.                                | L1 Cache Miss (sec)                 | 0.000 [0, 1]         | 0.011 [0, 0]         | 0.005         |
| p.                                | CPU Port Conf. (sec)                | 0.000 [0, 0]         | 0.000 [0, 0]         | 0.000         |
| q.                                | V. Arith. Exec. (sec)               | 6.963 [0, 3]         | 6.968 [0, 2]         | 6.965         |
| r.                                | V. Load Exec. (sec)                 | 9.783 [0, 0]         | 9.995 [0, 1]         | 9.882         |
| s.                                | VLD LLC Hit Element Ratio (%)       | 49.991 [0, 1]        | 50.000 [0, 0]        | 49.994        |
| t.                                | FMA Element Count                   | 268435456000 [0, 0]  | 268435456000 [0, 0]  | 268435456000  |
| u.                                | Power Throttling (sec)              | 0.000 [0, 0]         | 0.000 [0, 0]         | 0.000         |
| v.                                | Thermal Throttling (sec)            | 0.000 [0, 0]         | 0.000 [0, 0]         | 0.000         |
| w.                                | Memory Size Used (MB)               | 3784.000 [0, 0]      | 3784.000 [0, 0]      | 3784.000      |
| x.                                | Non Swappable Memory Size Used (MB) | 92.000 [0, 1]        | 156.000 [0, 0]       | 108.000       |

# NMPI\_PROGINF項目説明

- a. 経過時間
- b. ユーザCPU時間
- c. ベクトル命令実行時間
- d. 実行命令回数
- e. ベクトル命令実行回数
- f. ベクトル命令で処理された要素の個数
- g. ベクトルロードされた要素の個数
- h. 浮動小数点演算命令で処理された要素の個数
- i. ユーザーCPU時間1秒あたりの100万演算実行回数
- j. 経過時間1秒あたりの100万演算実行回数
- k. ユーザーCPU時間1秒あたりの100万浮動小数点演算実行回数
- l. 経過時間1秒あたりの100万浮動小数点演算実行回数
- m. 平均ベクトル長
- n. ベクトル演算率
- o. L1キャッシュミス時間
- p. CPUポート競合時間
- q. ベクトル演算実行時間
- r. ベクトルロード命令実行時間
- s. ベクトルロード命令によりロードされた要素のうち、LLCからロードされた要素の割合
- t. FMA命令実行要素数
- u. 電力要因によりHWが減速した時間
- v. 温度要因によりHWが減速した時間
- w. メモリ最大使用量
- x. Partial Precess Swapping機能でスワップアウトできないメモリの最大使用量

# MPISEPSELECT

## 標準出力および標準エラー出力の出力先を制御する

- 値が1の時, 標準出力だけstdout.\$IDに出力される
- 値が2の時, 標準エラー出力だけがstderr.\$IDに出力される(既定値)
- 値が3の時, 標準出力はstdout.\$IDに, 標準エラー出力はstderr.\$IDに出力される
- 値が4の時, 標準出力および標準エラー出力が, std.\$IDに出力される
- その他の時, 標準出力も標準エラー出力もファイルに出力しない

mpisep.shの使用例 (値=3を指定する場合)

```
#PBS -v MPISEPSELECT=3
```

```
mpirun -venode -np 4 /opt/nec/ve/bin/mpisep.sh a.out
```

## 9.4 SX-Aurora TSUBASAの簡易性能解析機能 ftrace

### ●使用方法

■ 測定対象のソースプログラムを翻訳時オプション `-ftrace` を指定してコンパイルすると、測定ルーチン呼び出す命令列がオブジェクト中に生成され、測定ルーチンをリンクした実行可能プログラムが生成される

■ 実行可能プログラムを実行すると、カレントディレクトリに解析情報ファイルとして `ftrace.out` が生成される（MPIプログラムの場合は、グループID、ランク番号が付与された名前となる）

■ `ftrace` コマンドを実行すると、解析リストが標準出力ファイルに出力される

```
% ftrace -f ftrace.out.* -all
```

# 簡易性能解析機能 ftrace 出力例

MPI実行時のftraceは、以下の赤枠部分の情報が追加出力される。

```

FTRACE ANALYSIS LIST

```

```
Execution Date : Fri Oct 1 18:33:34 2021 JST
Total CPU Time : 0:01'08"519 (68.519 sec.)
```

| FREQUENCY | EXCLUSIVE<br>TIME[sec] ( % ) | AVER. TIME<br>[msec] | MOPS    | MFLOPS  | V. OP<br>RATIO | AVER.<br>V. LEN | VECTOR<br>TIME | L1CACHE<br>MISS | CPU<br>CONF | PORT | VLD<br>HIT | LLC<br>E. % | PROC. NAME |
|-----------|------------------------------|----------------------|---------|---------|----------------|-----------------|----------------|-----------------|-------------|------|------------|-------------|------------|
| 4         | 68.519 (100.0)               | 17129.680            | 79182.1 | 31341.6 | 99.04          | 255.9           | 67.370         | 0.035           | 0.000       |      |            | 49.99       | EXAMPLE6   |
| 4         | 68.519 (100.0)               | 17129.680            | 79182.1 | 31341.6 | 99.04          | 255.9           | 67.370         | 0.035           | 0.000       |      |            | 49.99       | total      |

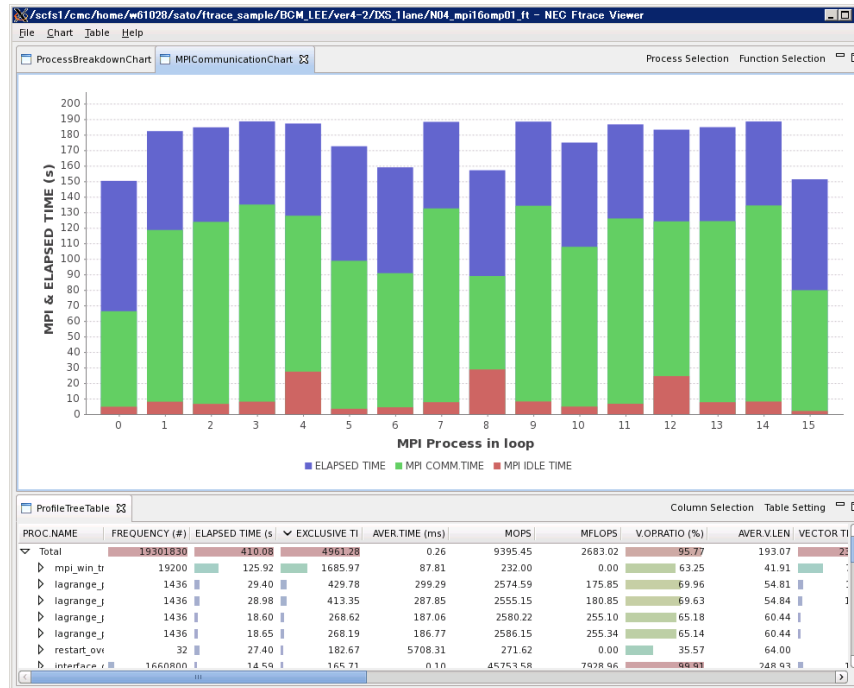
| (a)                  | (b)                 | (c)                     | (d)                | (e)                    | (f)                 | (g)   | (h)                 |            |
|----------------------|---------------------|-------------------------|--------------------|------------------------|---------------------|-------|---------------------|------------|
| ELAPSED<br>TIME[sec] | COMM. TIME<br>[sec] | COMM. TIME<br>/ ELAPSED | IDLE TIME<br>[sec] | IDLE TIME<br>/ ELAPSED | AVER. LEN<br>[byte] | COUNT | TOTAL LEN<br>[byte] | PROC. NAME |
| 17.409               | 0.566               |                         | 0.328              |                        | 300.0M              | 4     | 1.2G                | EXAMPLE6   |

- a. 経過時間
- b. MPI手続の実行に費やした経過時間
- c. 各プロセスにおいて、MPI手続の実行に費やした経過時間が、経過時間に占める割合
- d. メッセージ待ちに費やした経過時間
- e. 各プロセスにおいて、メッセージ待ちに費やした経過時間が、経過時間に占める割合
- f. MPI手続きあたりの平均通信量
- g. MPI手続きによる転送回数
- h. MPI手続きによる総通信量

# 9.5 NEC Ftrace Viewer

簡易性能解析機能(ftrace)情報をグラフィカルに表示するためのツール

- 関数・ルーチン単位の性能情報を絞り込み、多彩なグラフ形式で表示できます。
- 自動並列化機能・OpenMP、MPIを利用したプログラムのスレッド・プロセス毎の性能情報を容易に把握できます。



参考: [https://www.hpc.nec/documents/sdk/pdfs/g2at01-NEC\\_Ftrace\\_Viewer\\_User\\_Guide\\_ja.pdf](https://www.hpc.nec/documents/sdk/pdfs/g2at01-NEC_Ftrace_Viewer_User_Guide_ja.pdf)

# 10. 演習問題6

## 行列積プログラムをMPIで並列化してください

sample6.f

```
implicit real(8)(a-h,o-z)
parameter (n=12000)
real(8) a(n,n),b(n,n),c(n,n)
real(4) etime,cp1(2),cp2(2),t1,t2,t3
do j = 1,n
 do i = 1,n
 a(i,j) = 0.0d0
 b(i,j) = n+1-max(i,j)
 c(i,j) = n+1-max(i,j)
 enddo
enddo
write(6,50) ' Matrix Size = ',n
50 format(1x,a,i5)
t1=etime(cp1)
do j=1,n
 do k=1,n
 do i=1,n
 a(i,j)=a(i,j)+b(i,k)*c(k,j)
 end do
 end do
end do
t2=etime(cp2)
t3=cp2(1)-cp1(1)
write(6,60) ' Execution Time = ',t2,' sec',' A(n,n) = ',a(n,n)
60 format(1x,a,f10.3,a,1x,a,d24.15)
stop
end
```

- ◆ 左記の行列積を行うプログラムをMPI化して4プロセスで実行してください。



# 10. 演習問題6 (practice\_6) つづき

◆ ヒント: プログラムの流れは下記のとおり

MPIの初期化処理

プロセス数と自プロセスのランク番号の取得

分割時の始点と終点を求める

解を格納する配列aの初期化  
行列bとcの値の設定

各プロセスが担当する範囲の行列積を計算

解を格納する配列aをランク0に集める

ランク0が結果を出力

MPIの終了化処理

時間計測はMPI\_Wtimeを使用する

①時間を格納する変数はreal\*8で定義する

```
real*8 t1,t2
```

②測定する区間の始まりと終わりの時間を計測する

```
call MPI_BARRIER (MPI_COMM_WORLD,IERR)
t1=MPI_WTIME ()
[測定区間]
```

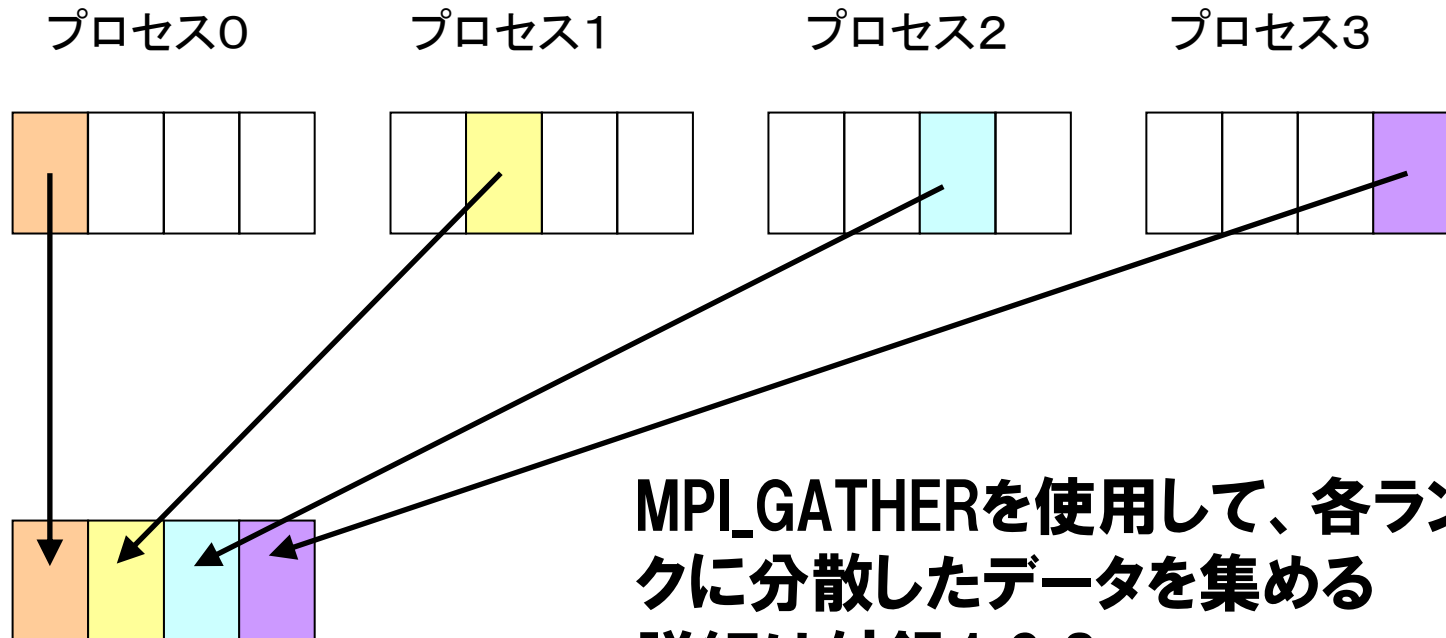
```
call MPI_BARRIER (MPI_COMM_WORLD,IERR)
t2=MPI_WTIME ()
```

③ $t2-t1$ が計測区間の時間となる

# 10. 演習問題6 (practice\_6) つづき

## ■データの転送方法(行列-ベクトル積)

- プロセス0はプロセス1, 2, 3から計算結果を格納した配列xを受け取る(下図)



# 10. 演習問題6 (practice\_6) つづき

## ① ディレクトリの移動

```
% cd MPI/practice_6
```

## ② プログラムの編集

```
% vi practice6.f
```

## ③ コンパイル

```
% mpifort practice6.f
```

## ④ 実行

```
% qsub run.sh
```

解答例(P172)

---

# 付録

# 付録

---

**付録1. 主な手続き**

**付録2. 参考文献, Webサイト**

# 付録1. 主な手続き

---

付録1.1 プロセス管理

付録1.2 一対一通信

付録1.3 集団通信

付録1.4 その他の手続き

※但し，本テキストでは，コミュニケータ（comm）は，MPI\_COMM\_WORLDとする。

# 付録1.1 プロセス管理

---

## 付録1.1.1 プロセス管理とは

■ MPI環境の初期化・終了処理や環境の問い合わせを行う

# 付録1. 1. 2 プログラム例(FORTRAN)

etc6.f

```
include 'mpif.h'
parameter (numdat=100)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
ist=((numdat-1)/nprocs+1)*myrank+1
ied=((numdat-1)/nprocs+1)*(myrank+1)
isum1=0
do i=ist,ied
 isum1=isum1+i
enddo
call MPI_REDUCE(isum1,isum,1,MPI_INTEGER,MPI_SUM,
& 0,MPI_COMM_WORLD,ierr)
if (myrank.eq.0) write(6,*) 'sum=',isum
call MPI_FINALIZE(ierr)
stop
end
```



# 付録1. 1. 2 プログラム例(C)

etc7.c

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
 int numdat=100;
 int myrank, nprocs;
 int i, ist, ied, isum1, isum;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 ist=((numdat-1)/nprocs+1)*myrank+1;
 ied=((numdat-1)/nprocs+1)*(myrank+1);
 isum1=0;
 for(i=ist;i<ied+1;i++) isum1 += i;
 MPI_Reduce(&isum1,&isum,1,MPI_INT,MPI_SUM,
 0,MPI_COMM_WORLD);
 if(myrank==0) printf("isum=%d¥n", isum);
 MPI_Finalize();
}
```

# 付録1. 1.3 インクルードファイル

## 書式

```
include 'mpif.h' ...FORTRAN
```

```
#include "mpi.h" ...C
```

## メモ

- MPI手続きを使うサブルーチン・関数では、必ずインクルードしなければならない
- MPIで使用する MPI\_xxx といった定数を定義している
- ユーザは、このファイルの中身まで知る必要はない

mpif.h

```
 :
 INTEGER MPI_LOR, MPI BOR, MPI_LXOR, MPI_BXOR,
 INTEGER MPI_MAXLOC, MPI_REPLACE
 PARAMETER (MPI_MAX = 100)
 PARAMETER (MPI_MIN = 101)
 PARAMETER (MPI_SUM = 102)
 :
```

# 付録1. 1.4 MPI\_INIT MPI環境の初期化

## 機能概要

- MPI環境の初期化処理を行う
- 引数は返却コード `ierr` のみ (FORTRANの場合)

## 書式

```
integer ierr
CALL MPI_INIT (ierr)

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

## メモ

- 他のMPIルーチンより前に1度だけ呼び出されなければならない
- 返却コードは、コールしたMPIルーチンが正常に終了すれば、`MPI_SUCCESS`を返す(他のMPIルーチンでも同じ)
- 当該手続きを呼び出す前に設定した変数・配列は、他のプロセスには引き継がれない(引き継ぐには通信が必要)

# 付録1. 1. 5 MPI\_FINALIZE MPI環境の終了

## 機能概要

- MPI環境の終了処理を行う
- 引数は返却コード `ierr` のみ (FORTRANの場合)

## 書式

```
integer ierr
CALL MPI_FINALIZE(ierr)
```

```
int MPI_Finalize (void)
```

## メモ

- プログラムが終了する前に、必ず1度実行する必要がある
  - 異常終了処理には、MPI\_ABORTを用いる
- この手続きが呼び出された後は、いかなるMPIルーチンも呼び出してはならない

# 付録1. 1. 6 MPI\_ABORT MPI環境の中断

## 機能概要

- MPI環境の異常終了処理を行う

## 書式

```
integer comm, errcode, ierr
CALL MPI_ABORT(comm, errcode, ierr)
```

```
int MPI_Abort (MPI_Comm comm, int errcode)
```

## 引数

| 引数      | 値      | 入出力 |         |
|---------|--------|-----|---------|
| comm    | handle | IN  | コミュニケータ |
| errcode | 整数     | IN  | エラーコード  |

## メモ

- すべてのプロセスを即時に異常終了しようとする
- 引数にコミュニケータを必要とするが MPI\_COMM\_WORLDを想定

# 付録1. 1. 7 MPI\_COMM\_SIZE MPIプロセス数の取得

## 機能概要

- 指定したコミュニケータにおける全プロセス数を取得する

## 書式

```
integer comm, nprocs, ierr
CALL MPI_COMM_SIZE (comm, nprocs, ierr)
```

```
int MPI_Comm_size (MPI_Comm comm, int *nprocs)
```

## 引数

| 引数     | 値      | 入出力 |                 |
|--------|--------|-----|-----------------|
| comm   | handle | IN  | コミュニケータ         |
| nprocs | 整数     | OUT | コミュニケータ内の総プロセス数 |

## メモ

- commがMPI\_COMM\_WORLDの場合, 利用可能なプロセスの総数を返す

# 付録1. 1. 8 MPI\_COMM\_RANK ランク番号の取得

## 機能概要

- 指定したコミュニケータにおける自プロセスのランク番号を取得する

## 書式

```
integer comm, myrank, ierr
CALL MPI_COMM_RANK(comm, myrank, ierr)
```

```
int MPI_Comm_rank(MPI_Comm comm, int *myrank)
```

## 引数

| 引数     | 値      | 入出力 |                |
|--------|--------|-----|----------------|
| comm   | handle | IN  | コミュニケータ        |
| myrank | 整数     | OUT | コミュニケータ中のランク番号 |

## メモ

- 自プロセスと他プロセスの区別, 認識に用いる
- 0からnproc-1までの範囲で呼び出したプロセスのランクを返す (nprocsはMPI\_COMM\_SIZEの返却値)

# 付録1. 1. 9 ランク番号と総プロセス数を使った処理の分割

## 1から100までをnprocで分割

myrank=0  
nprocs=4

myrank=1  
nprocs=4

myrank=2  
nprocs=4

myrank=3  
nprocs=4

$$\text{ist} = ((100-1)/\text{nprocs}+1) * \text{myrank} + 1$$
$$\text{ied} = ((100-1)/\text{nprocs}+1) * (\text{myrank}+1)$$

$$\text{ist} = ((100-1)/\mathbf{4}+1) * \mathbf{0} + 1$$
$$= \mathbf{1}$$
$$\text{ied} = ((100-1)/\mathbf{4}+1) * (\mathbf{0}+1)$$
$$= \mathbf{25}$$

$$\text{ist} = ((100-1)/\mathbf{4}+1) * \mathbf{1} + 1$$
$$= \mathbf{26}$$
$$\text{ied} = ((100-1)/\mathbf{4}+1) * (\mathbf{1}+1)$$
$$= \mathbf{50}$$

$$\text{ist} = ((100-1)/\mathbf{4}+1) * \mathbf{2} + 1$$
$$= \mathbf{51}$$
$$\text{ied} = ((100-1)/\mathbf{4}+1) * (\mathbf{2}+1)$$
$$= \mathbf{75}$$

$$\text{ist} = ((100-1)/\mathbf{4}+1) * \mathbf{3} + 1$$
$$= \mathbf{76}$$
$$\text{ied} = ((100-1)/\mathbf{4}+1) * (\mathbf{3}+1)$$
$$= \mathbf{100}$$



# 付録1.2 一対一通信

---

## 付録1.2.1 一対一通信とは

- 一組の送信プロセスと受信プロセスが行うメッセージ交換
- メッセージの交換は、データを送受信することで行われる
- 一対一通信は、送信処理と受信処理に分かれている
- ブロッキング型通信と非ブロッキング型通信がある

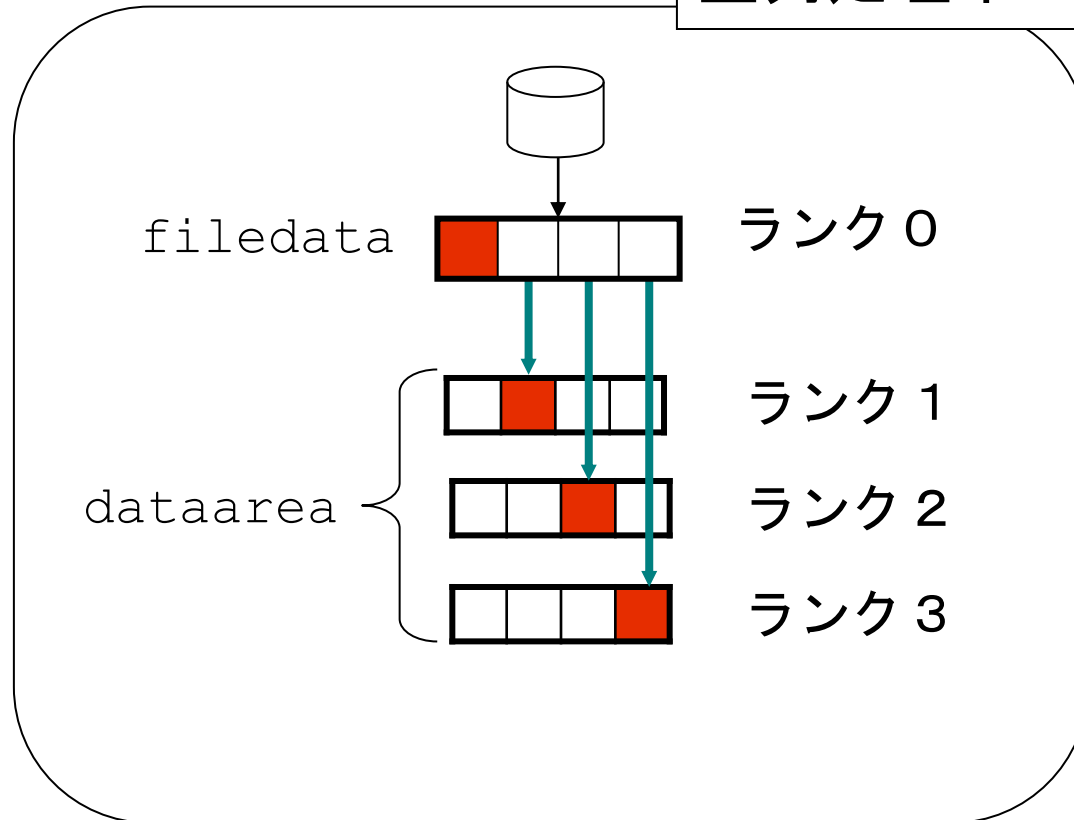
## 付録1.2.2 プログラム例

逐次版 (etc8.f)

```
integer a(100), isum
open(10, file='fort.10')
read(10, *) a
isum=0
do i=1, 100
 isum=isum+a(i)
enddo
write(6, *) 'SUM=', isum
stop
end
```

# 処理イメージ

## 並列処理イメージ



# プログラム例(MPI版)

etc9.f

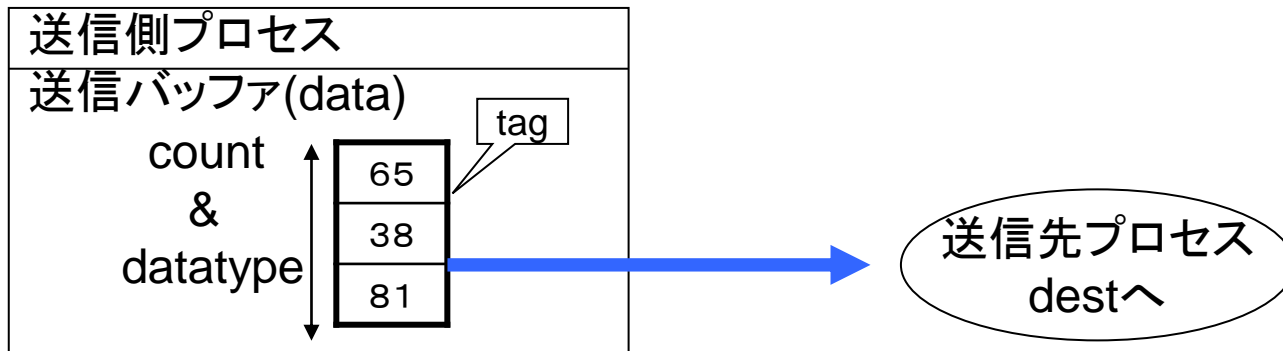
```
include 'mpif.h'
parameter (numdat=100)
integer status (MPI_STATUS_SIZE), senddata (numdat), recvdata (numdat)
integer source, dest, tag
call MPI_INIT(ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, nprocs, ierr)
icount=(numdat-1)/nprocs+1
if (myrank .eq. 0) then
 open(10, file='fort. 10')
 read(10, *) senddata
 do i=1, nprocs-1
 dest=i
 tag=myrank
 call MPI_SEND (senddata (icount*i+1), icount, MPI_INTEGER,
& dest, tag, MPI_COMM_WORLD, ierr)
 enddo
 recvdata=senddata
else
 source=0
 tag=source
 call MPI_RECV (recvdata (icount*myrank+1), icount, MPI_INTEGER,
& source, tag, MPI_COMM_WORLD, status, ierr)
endif
isum=0
do i=1, icount
 isum=isum+recvdata (icount*myrank+i)
enddo
call MPI_FINALIZE(ierr)
write(6, *) myrank, ':SUM= ', isum
stop ; end
```

# 付録1. 2. 3 MPI\_SEND ブロッキング型送信

## 機能概要

- 送信バッファ(data)内のデータ型がdatatypeで連続したcount個のタグ(tag)付き要素をコミュニケータcomm内のランクdestなるプロセスに送信する

## 処理イメージ



# MPI\_SEND ブロッキング型送信

## 書式

```
任意の型 data(*)
integer count, datatype, dest, tag, comm, ierr
CALL MPI_SEND (data, count, datatype, dest, tag, comm, ierr)
```

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

## 引数

| 引数       | 値      | 入出力 |                   |
|----------|--------|-----|-------------------|
| data     | 任意     | IN  | 送信データの開始アドレス      |
| count    | 整数     | IN  | 送信データの要素数(0以上の整数) |
| datatype | handle | IN  | 送信データのタイプ         |
| dest     | 整数     | IN  | 通信相手のランク          |
| tag      | 整数     | IN  | メッセージタグ           |
| comm     | handle | IN  | コミュニケータ           |

# MPI\_SEND ブロッキング型送信(続き)

## メモ

- **メッセージの大きさはバイト数ではなく、要素の個数(count)で表す**
- **datatypeは次ページ以降に一覧を示す**
- **タグはメッセージを区別するために使用する**
- **本ルーチン呼び出し後、転送処理が完了するまで処理を待ち合わせる**
- **MPI\_SENDで送信したデータは、MPI\_IRecv, MPI\_RECVのどちらかで受信してもよい**

## 付録1. 2. 4 MPIで定義された変数の型(FORTRAN)

| MPIの<br>データタイプ          | FORTRAN言語の<br>対応する型 |    |
|-------------------------|---------------------|----|
| MPI_INTEGER             | INTEGER             |    |
| MPI_INTEGER2            | INTEGER*2           |    |
| MPI_INTEGER4            | INTEGER*4           |    |
| MPI_REAL                | REAL                |    |
| MPI_REAL4               | REAL*4              |    |
| MPI_REAL8               | REAL*8              |    |
| MPI_DOUBLE_PRECISION    | DOUBLE PRECISION    |    |
| MPI_REAL16              | REAL*16             |    |
| MPI_QUADRUPLE_PRECISION | QUADRUPLE PRECISION |    |
| MPI_COMPLEX             | COMPLEX             |    |
| MPI_COMPLEX8            | COMPLEX*8           |    |
| MPI_COMPLEX16           | COMPLEX*16          |    |
| MPI_DOUBLE_COMPLEX      | DOUBLE COMPLEX      |    |
| MPI_COMPLEX32           | COMPLEX*32          |    |
| MPI_LOGICAL             | LOGICAL             |    |
| MPI_LOGICAL1            | LOGICAL*1           |    |
| MPI_LOGICAL4            | LOGICAL*4           |    |
| MPI_CHARACTER           | CHARACTER           | など |



# MPIで定義された変数の型(C)

## MPI

### データタイプ

MPI\_CHAR

MPI\_SHORT

MPI\_INT

MPI\_LONG

MPI\_LONG\_LONG

MPI\_LONG\_LONG\_INT

MPI\_UNSIGNED\_CHAR

MPI\_UNSIGNED\_SHORT

MPI\_UNSIGNED\_INT

MPI\_UNSIGNED\_LONG

MPI\_FLOAT

MPI\_DOUBLE

MPI\_LONG\_DOUBLE

## C言語

### 対応する型

char

short

int

long

long long

long long

unsigned char

unsigned short

unsigned int

unsigned long

float

double

long double

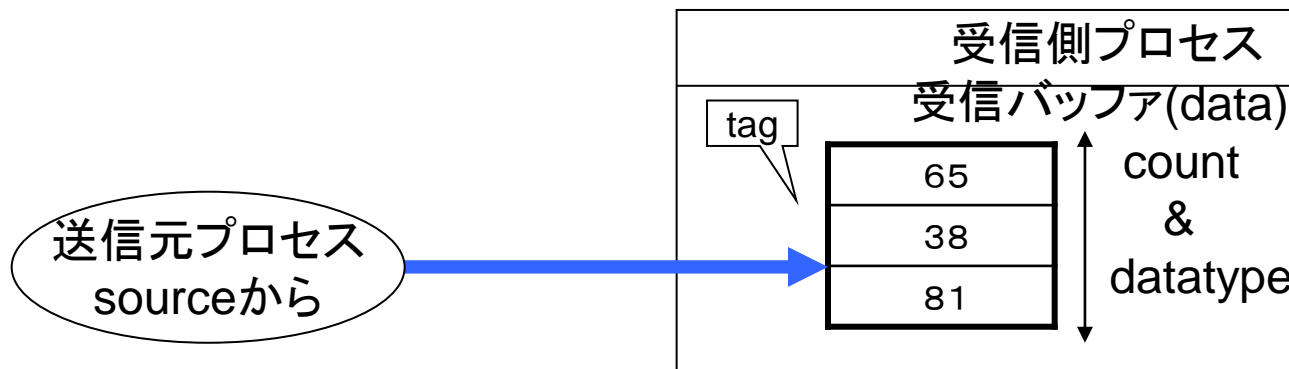
など

# 付録1. 2. 5 MPL\_RECV ブロッキング型受信

## 機能概要

- コミュニケータcomm内のランクsourceなるプロセスから送信されたデータ型がdatatypeで連続したcount個のタグ(tag)付き要素を受信バッファ(data)に同期受信する

## 処理イメージ



# MPI\_RECV ブロッキング型受信(続き)

## 書式

```
任意の型 data(*)
integer count, datatype, source, tag, comm,
 status(MPI_STATUS_SIZE), ierr
CALL MPI_RECV(data, count, datatype, source, tag,
 comm, status, ierr)
```

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

## 引数

| 引数       | 値      | 入出力 |                   |
|----------|--------|-----|-------------------|
| data     | 任意     | OUT | 受信データの開始アドレス      |
| count    | 整数     | IN  | 受信データの要素の数(0以上の値) |
| datatype | handle | IN  | 受信データのタイプ         |
| source   | 整数     | IN  | 通信相手のランク          |
| tag      | 整数     | IN  | メッセージタグ           |
| comm     | handle | IN  | コミュニケータ           |
| status   | status | OUT | メッセージ情報           |

# MPI\_RECV ブロッキング型受信(続き)

メモ

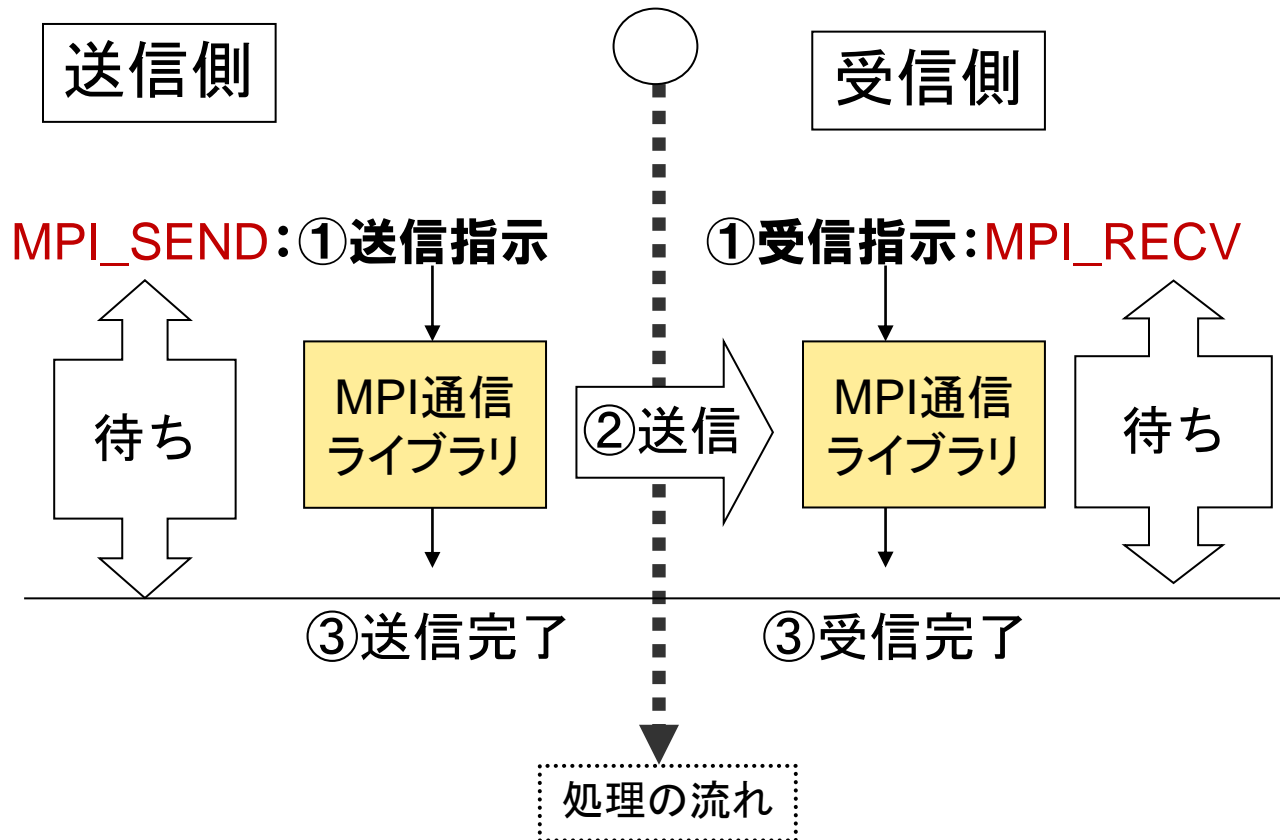
**転送処理が完了するまで処理を待ち合わせる**

**引数statusは通信の完了状況が格納される**

- FORTRANでは大きさがMPI\_STATUS\_SIZEの整数配列
- CではMPI\_Statusという型の構造体で、送信元やタグ、エラーコードなどが格納される

## 付録1.2.6 ブロッキング型通信の動作

- MPI\_SEND, MPI\_RECV

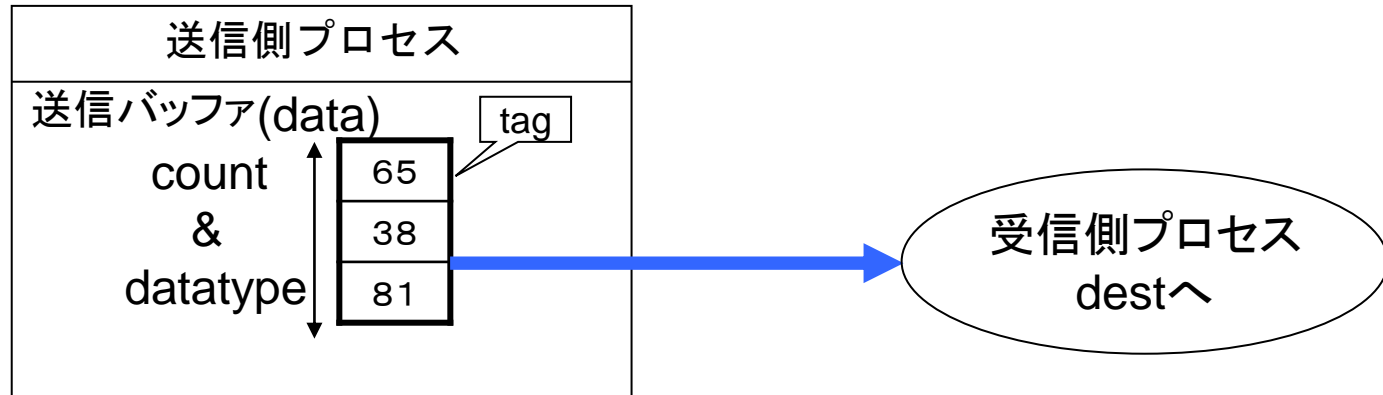


## 付録1. 2. 7 MPI\_SEND 非ブロッキング型送信

### 機能概要

- 送信バッファ(data)内のデータ型がdatatypeで連続したcount個のタグ(tag)付き要素をコミュニケータcomm内のランクdestなるプロセスに送信する

### 処理イメージ



# MPI\_ISEND 非ブロッキング型送信(続き)

## 書式

任意の型 data (\*)

```
integer count, datatype, dest, tag, comm, request, ierr
CALL MPI_ISEND(data, count, datatype, dest, tag,
 comm, request, ierr)
```

## 引数

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

| 引数       | 値      | 入出力 |                   |
|----------|--------|-----|-------------------|
| data     | 任意     | IN  | 送信データの開始アドレス      |
| count    | 整数     | IN  | 送信データの要素の数(0以上の値) |
| datatype | handle | IN  | 送信データのタイプ         |
| dest     | 整数     | IN  | 通信相手のランク          |
| tag      | 整数     | IN  | メッセージタグ           |
| comm     | handle | IN  | コミュニケータ           |
| request  | handle | OUT | 通信識別子             |

# MPI\_ISEND 非ブロッキング型送信(続き)

## メモ

- **メッセージの大きさはバイト数ではなく、要素の個数(count)で表す**
- **datatypeはMPI\_SENDの項を参照**
- **タグはメッセージを区別するために使用する**
- **requestには要求した通信の識別子が戻され、MPI\_WAIT等で通信の完了を確認する際に使用する**
- **本ルーチンコール後、受信処理の完了を待たずにプログラムの処理を続行する**
- **MPI\_WAITまたはMPI\_WAITALLで処理の完了を確認するまでは、dataの内容を更新してはならない**
- **MPI\_ISENDで送信したデータは、MPI\_IRecv, MPI\_RECVのどちらで受信してもよい**
- **通信の完了もMPI\_WAIT, MPI\_WAITALLのどちらを使用してもよい**

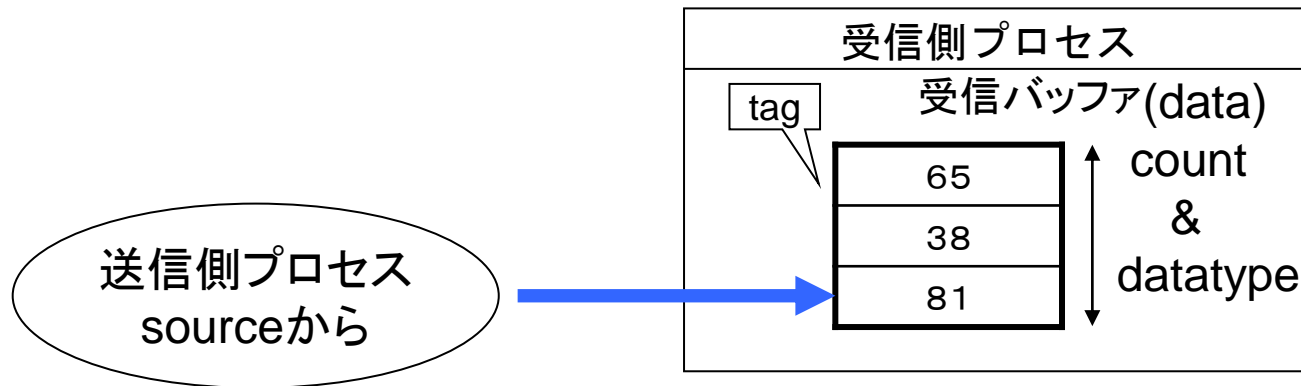


# 付録1.2.8 非ブロッキング型受信

## 機能概要

- コミュニケータcomm内のランクsourceなるプロセスから送信されたデータ型がdatatypeで連続したcount個のタグ(tag)付き要素を受信バッファ(data)に受信する

## 処理イメージ



# MPI\_RECV 非ブロッキング型受信(続き)

## 書式

```
任意の型 data(*)
integer count, datatype, source, tag, comm, request, ierr
CALL MPI_RECV(data, count, datatype, source, tag,
 comm, request, ierr)
```

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

## 引数

| 引数       | 値      | 入出力 |                   |
|----------|--------|-----|-------------------|
| data     | 任意     | OUT | 受信データの開始アドレス      |
| count    | 整数     | IN  | 受信データの要素の数(0以上の値) |
| datatype | handle | IN  | 受信データのタイプ         |
| source   | 整数     | IN  | 通信相手のランク          |
| tag      | 整数     | IN  | メッセージタグ           |
| comm     | handle | IN  | コミュニケータ           |
| request  | status | OUT | メッセージ情報           |

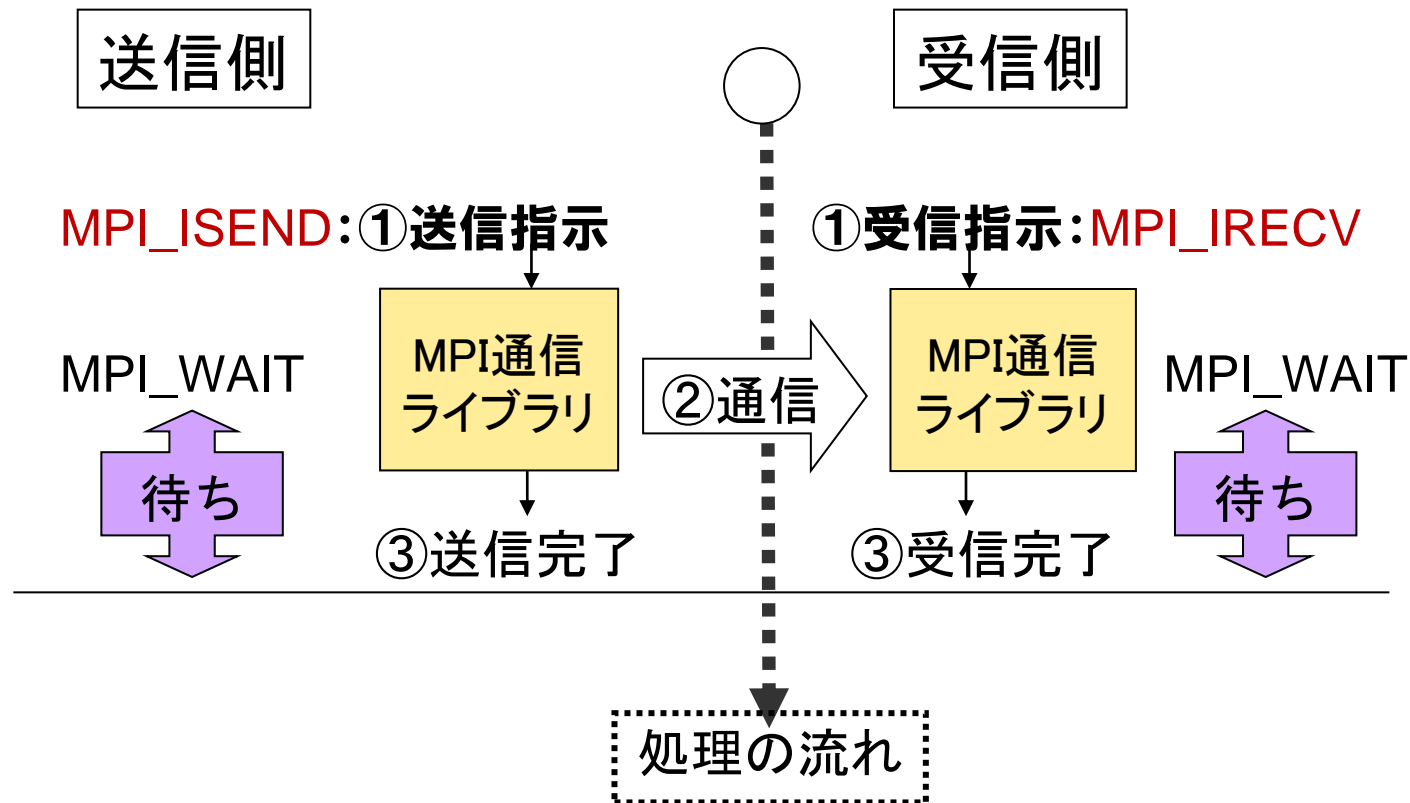
## MPI\_RECV 非ブロッキング型受信(続き)

メモ

- **メッセージの大きさは要素の個数(count)で表す**
- **datatypeはMPI\_SENDの項を参照**
- **タグは送信側で付けられた値もしくは、MPI\_ANY\_TAGを指定する**
- **requestは要求した通信の識別子が戻され、MPI\_WAIT等で通信の完了を確認する際に使用する**
- **本ルーチンコール後、処理の完了を待たずにプログラムの処理を続行する**
- **MPI\_WAITまたはMPI\_WAITALLで処理の完了を確認するまでは、dataの内容を使用してはならない**
- **MPI\_ISEND, MPI\_SENDのどちらで送信したデータもMPI\_RECVで受信してよい**
- **通信の完了もMPI\_WAIT, MPI\_WAITALLのどちらを使用してもよい**

## 付録1.2.9 非ブロッキング型通信の動作

- MPI\_ISEND, MPI\_IRECVの動作



## 付録1. 2. 10 MPI\_WAIT 通信完了の待ち合わせ

### 機能概要

- 非同期通信処理が完了するまで待ち合わせる

### 書式

```
integer request, status(MPI_STATUS_SIZE), ierr
CALL MPI_WAIT(request, status, ierr)
```

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

### 引数

| 引数      | 値      | 入出力   |         |
|---------|--------|-------|---------|
| request | handle | INOUT | 通信識別子   |
| status  | status | out   | メッセージ情報 |

### メモ

- requestには、MPI\_ISEND, MPI\_IRECVをコールして返されたメッセージ情報requestを指定する
- statusには、FORTRANではMPI\_STATUS\_SIZEの整数配列, CではMPI\_Status型の構造体を指定する

## 付録1. 2. 11 MPI\_WAITALL 通信完了の待合わせ

### 機能概要

- 1つ以上の非同期通信全ての完了を待ち合わせる

### 書式

```
integer count, array_of_requests(count),
 array_of_status(MPI_STATUS_SIZE,*), ierr
call MPI_WAITALL(count,array_of_requests,
 array_of_status,ierr)
```

```
int MPI_Waitall(int count,
 MPI_Request *array_of_requests,
 MPI_Status *array_of_status)
```

### 引数

| 引数                | 値      | 入出力   |                           |
|-------------------|--------|-------|---------------------------|
| count             | 整数     | IN    | 待ち合わせる通信の数                |
| array_of_requests | handle | INOUT | 通信識別子の配列<br>大きさは(count)   |
| array_of_status   | status | OUT   | メッセージ情報の配列<br>大きさは(count) |

# MPI\_WAITALL 通信完了の待ち合わせ

メモ

array\_of\_statusは、Fortranでは整数配列で大きさは(count, MPI\_STATUS\_SIZE)

CではMPI\_Statusの構造体の配列で、大きさは(count)

array\_of\_statusには、array\_of\_requestsに指定されたrequestと同じ順番で、そのrequestに対応する通信の完了状態が格納される

## 付録1. 2. 12 一対一通信まとめ

|       | 送信        | 受信        | 待ち合せ          |
|-------|-----------|-----------|---------------|
| 同期通信  | MPI_SEND  | MPI_RECV  |               |
| 非同期通信 | MPI_ISEND | MPI_Irecv | MPI_WAIT(ALL) |

■ MPI\_SEND, MPI\_ISENDのどちらで送信した場合でも,  
MPI\_RECV, MPI\_Irecvのどちらで受信してもよい

(“I”は immediate の頭文字)

■ MPI\_ISEND, MPI\_Irecvは, MPI\_WAITで個別に待ち合わせしても  
MPI\_WAITALLでまとめて待ち合わせしても良い



# 付録1.3 集団通信

## 付録1.3.1 集団通信とは

■ コミュニケータ内の全プロセスで行う同期的通信

- 総和計算などのリダクション演算
- 入力データの配布などに用いられるブロードキャスト
- FFTで良く用いられる転置
- その他ギャザ／スキヤッタなど

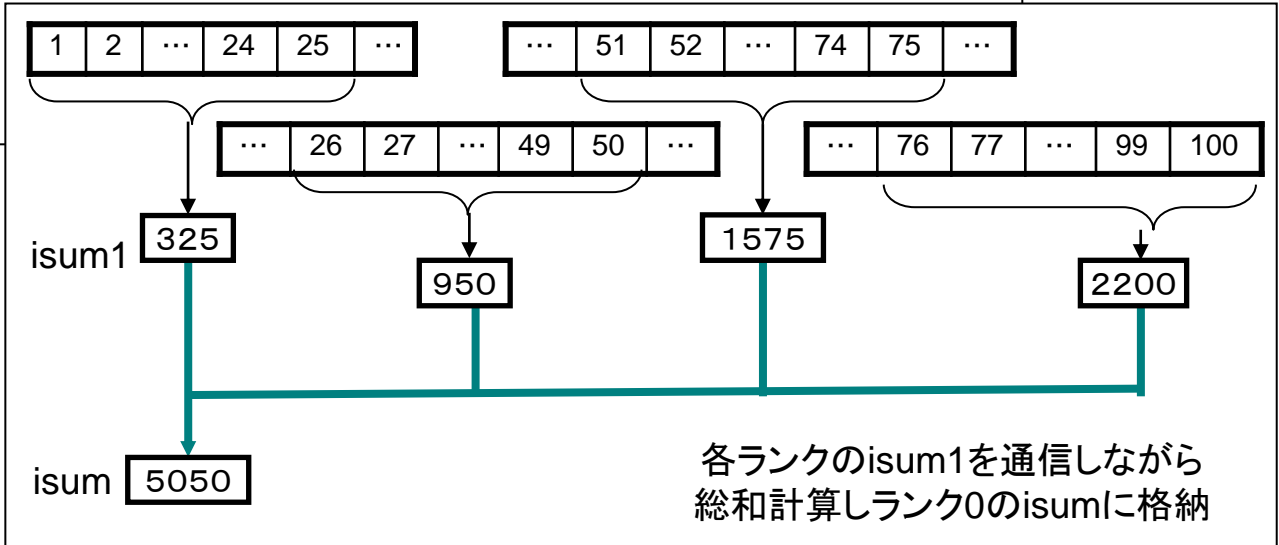
# 付録1. 3. 2 プログラム例

etc10.f

```
include 'mpif.h'
parameter (numdat=100)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
ist=((numdat-1)/nprocs+1)*myrank+1
ied=((numdat-1)/nprocs+1)*(myrank+1)
isum1=0
do i=ist, ied
 isum1=isum1+i
enddo
call MPI_REDUCE(isum1, isum, 1, MPI_INTEGER, MPI_SUM,
& 0, MPI_COMM_WORLD, ierr)
if(myrank.eq.0) write(6,*) 'isum=', isum
call MPI_FINALIZE(ierr)
stop
end
```

プロセス毎の小計しか  
わからない

各プロセスの小計を集  
計する

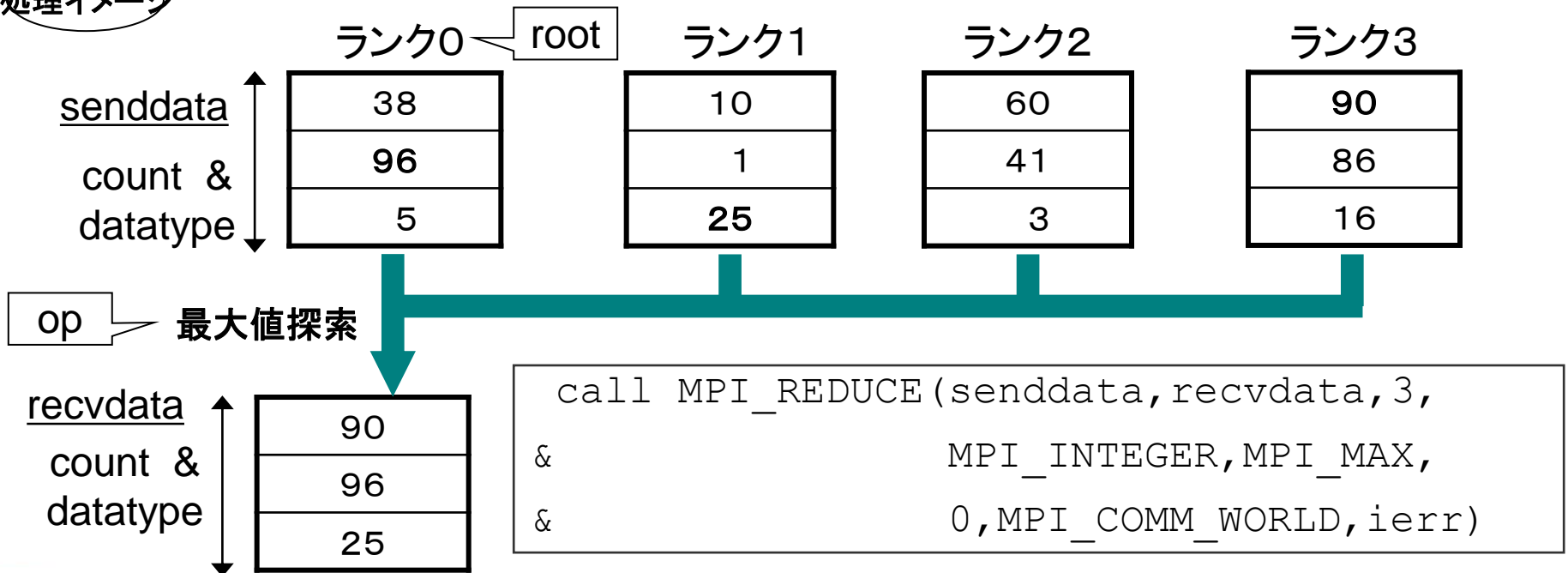


# 付録1.3.3 MPI\_REDUCE リダクション演算

## 機能概要

- コミュニケータcomm内の全プロセスが、送信バッファのデータ(senddata)を通信しながら、opで指定された演算を行い、結果を宛先(root)プロセスの受信バッファ(recvdata)に格納する
- 送信データが配列の場合は、要素毎に演算を行う

## 処理イメージ



# MPI\_REDUCE(続き)

## 書式

```
任意の型 senddata(*), recvdata(*)
integer count, datatype, op, root, comm, ierr
call MPI_REDUCE(senddata, recvdata, count, datatype, op,
 root, comm, ierr)
```

```
int MPI_Reduce(void* senddata, void* recvdata, int count,
 MPI_Datatype datatype, MPI_Op op, int root,
 MPI_Comm comm)
```

## 引数

| 引数       | 値      | 入出力 |                                 |
|----------|--------|-----|---------------------------------|
| senddata | 任意     | IN  | 送信データのアドレス                      |
| recvdata | 任意     | OUT | 受信データのアドレス<br>(rootプロセスだけ意味を持つ) |
| count    | 整数     | IN  | 送信データの要素の数                      |
| datatype | handle | IN  | 送信データのタイプ                       |
| op       | handle | IN  | リダクション演算の機能コード                  |
| root     | 整数     | IN  | rootプロセスのランク                    |
| comm     | handle | IN  | コミュニケーター                        |

# MPI\_REDUCEで使える演算

| 機能名        | 機能         |
|------------|------------|
| MPI_MAX    | 最大値        |
| MPI_MIN    | 最小値        |
| MPI_SUM    | 総和         |
| MPI_PROD   | 累積         |
| MPI_MAXLOC | 最大値と対応情報取得 |
| MPI_MINLOC | 最小値と対応情報取得 |
| MPI_BAND   | ビット積       |
| MPI_BOR    | ビット和       |
| MPI_BXOR   | 排他的ビット和    |
| MPI_LAND   | 論理積        |
| MPI_LOR    | 論理和        |
| MPI_LXOR   | 排他的論理和     |

# 総和計算の丸め誤差

総和計算において、逐次処理と並列処理とで結果が異なる場合がある



並列処理に限らず、部分和をとってから総和を算出する等、加算順序の変更により結果が異なっている可能性がある

例 (有効桁数を小数点以下4桁として)  
配列aに右の数値が入っていたとする

|      |   |   |   |   |      |
|------|---|---|---|---|------|
| 1E+5 | 7 | 4 | 8 | 6 | 1E+5 |
|------|---|---|---|---|------|

## 逐次処理

$$\text{dsum} = a(1) + a(2) = 1E5 + 0.00007E5$$

有効桁数以下切捨てで

$$= 1.0000E+5$$

同様に a(3), a(4), a(5) まで足し込んだdsumは 1.0000E+5

$$\text{dsum} = \text{dsum} + a(6)$$

$$= 1.0000E+5 + 1.0000E+5$$

$$= \underline{2.0000E+5}$$

## 2並列

$$\text{dsum1} = a(1) + a(2) = 1E5 + 0.00007E5 = 1.0000E+5$$

$$\text{dsum1} + a(3) = 1E5 + 0.00004E5 = 1.0000E+5$$

$$\text{dsum2} = a(4) + a(5) = 8 + 6 = 14 = 0.00001E5$$

$$\text{dsum2} + a(6) = 0.00001E5 + 1E5 = 1.0001E+5$$

$$\text{dsum} = \text{dsum1} + \text{dsum2}$$

$$= 1.0000E+5 + 1.0001E+5$$

$$= \underline{2.0001E+5}$$

加算順序の違いで異なる結果になった

## 付録1.3.4 注意事項

---

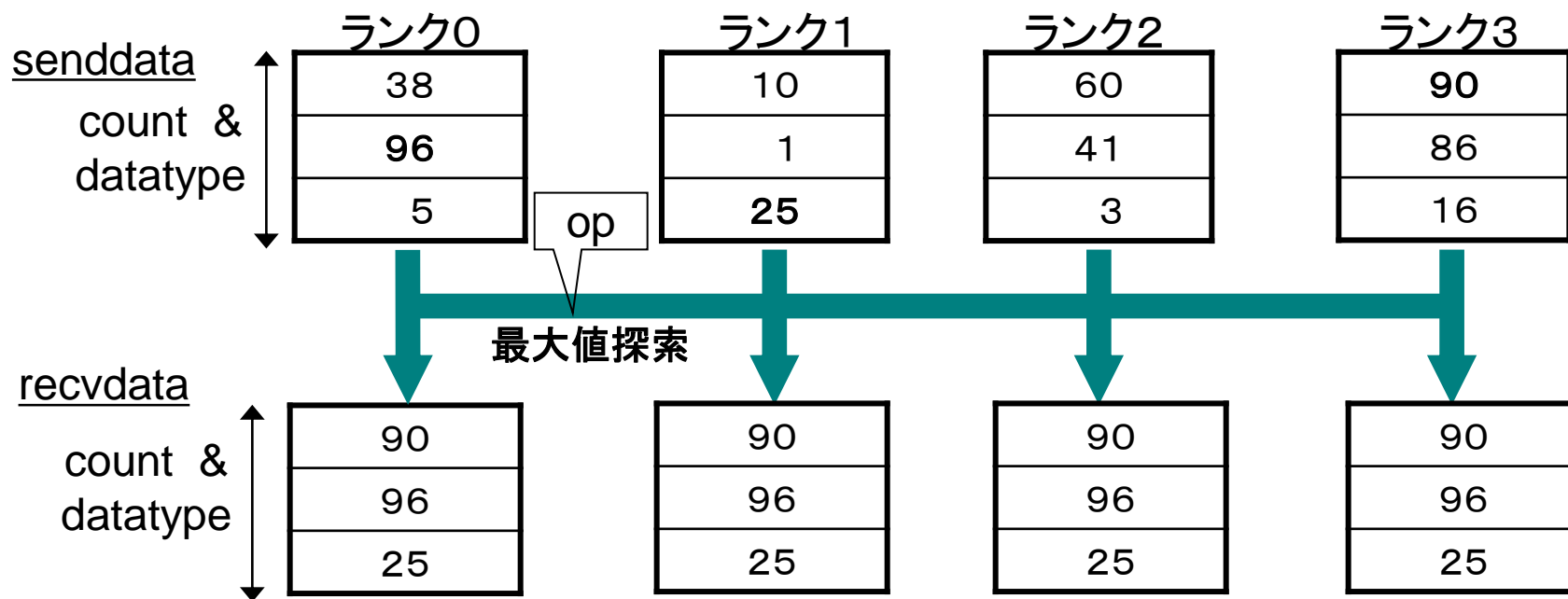
- **通信に参加する全プロセスが、同じ集団通信手続きをコールしなければならない**
- **送信バッファと受信バッファの実際に使用する部分は、メモリ上で重なってはならない**  
(MPI-2では、MPI\_IN\_PLACEを用いることで可能になります)
- **基本的に集団通信処理の直前や直後での同期処理は不要**

# 付録1. 3. 5 MPI\_ALLREDUCE リダクション演算

## 機能概要

- コミュニケータcomm内の全プロセスが、送信バッファのデータ (senddata)を通信しながら、opで指定された演算を行い、結果を全プロセスの受信バッファ(recvdata)に格納する

## 処理イメージ



```
call MPI_ALLREDUCE (senddata, recvdata, 3, MPI_INTEGER, MPI_MAX,
& MPI_COMM_WORLD, ierr)
```



# MPI\_ALLREDUCE(続き)

## 書式

```
任意の型 senddata(*), recvdata(*)
integer count, datatype, op, comm, ierr
call MPI_ALLREDUCE(senddata, recvdata, count, datatype, op,
 comm, ierr)
```

```
int MPI_Allreduce(void* senddata, void* recvdata, int count,
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

## 引数

| 引数       | 値      | 入出力 |                |
|----------|--------|-----|----------------|
| senddata | 任意     | IN  | 送信データのアドレス     |
| recvdata | 任意     | OUT | 受信データのアドレス     |
| count    | 整数     | IN  | 送信データの要素の数     |
| datatype | handle | IN  | 送信データのタイプ      |
| op       | handle | IN  | リダクション演算の機能コード |
| comm     | handle | IN  | コミュニケータ        |

## メモ

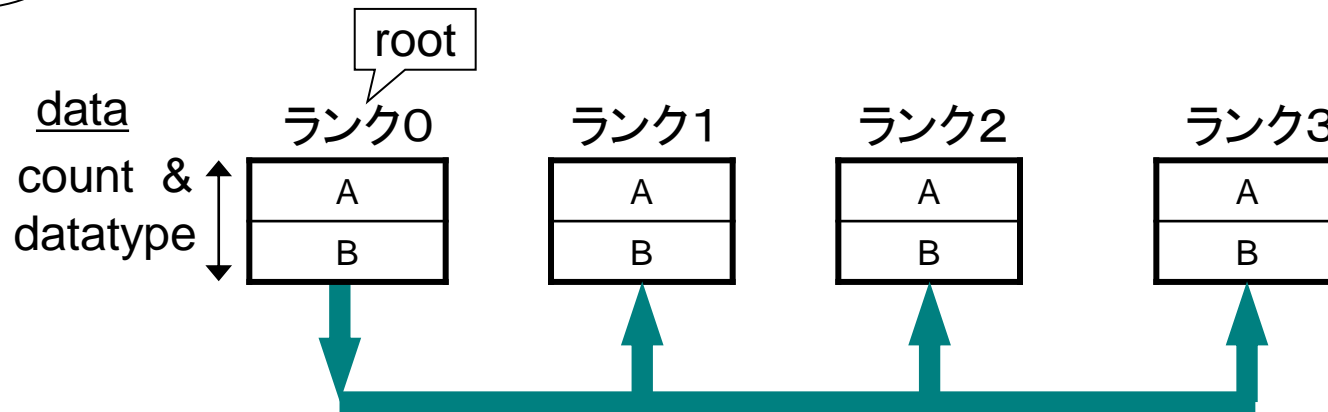
- ・ MPI\_REDUCEの計算結果を全プロセスに送信するのと機能的に同じ

# 付録1.3.6 MPL\_BCAST ブロードキャスト

## 機能概要

- 1つの送信元プロセス(root)の送信バッファ(data)のデータをコミュニケータcomm内全てのプロセスの受信バッファ(data)に送信する

## 処理イメージ



# MPI\_BCAST (続き)

## 書式

**任意の型** data (\*)

```
integer count, datatype, root, comm, ierr
```

```
call MPI_BCAST(data, count, datatype, root, comm, ierr)
```

```
int MPI_Bcast(void* data, int count, MPI_Datatype
 datatype, int root, MPI_Comm comm)
```

## 引数

| 引数       | 値      | 入出力   |                    |
|----------|--------|-------|--------------------|
| data     | 任意     | INOUT | データの開始アドレス         |
| count    | 整数     | IN    | データの要素の数           |
| datatype | handle | IN    | データのタイプ            |
| root     | 整数     | IN    | ブロードキャスト送信プロセスのランク |
| comm     | handle | IN    | コミュニケータ            |

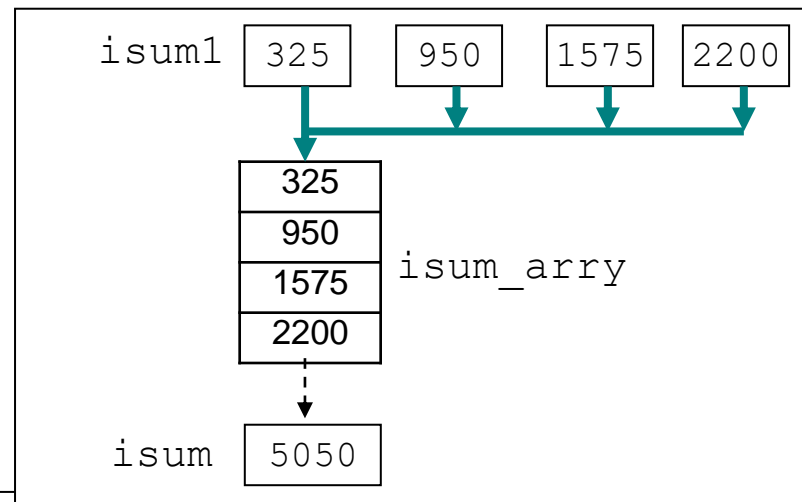
## メモ

- dataはrootプロセスでは送信データ, その他のプロセスでは受信データになる

# 付録1.3.7 プログラム例(総和計算)

etc11.f

```
include 'mpif.h'
parameter (numdat=100)
integer isum_array(10)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs, ierr)
ist=((numdat-1)/nprocs+1)*myrank+1
ied=((numdat-1)/nprocs+1)*(myrank+1)
isum1=0
do i=ist,ied
 isum1=isum1+i
enddo
call MPI_GATHER(isum1, 1, MPI_INTEGER, isum_array, 1,
& MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
if(myrank.eq.0) then
 isum=0
 do i=1,nprocs
 isum=isum+isum_array(i)
 enddo
 write(6,*)' isum=', isum
endif
call MPI_FINALIZE(ierr)
stop
end
```

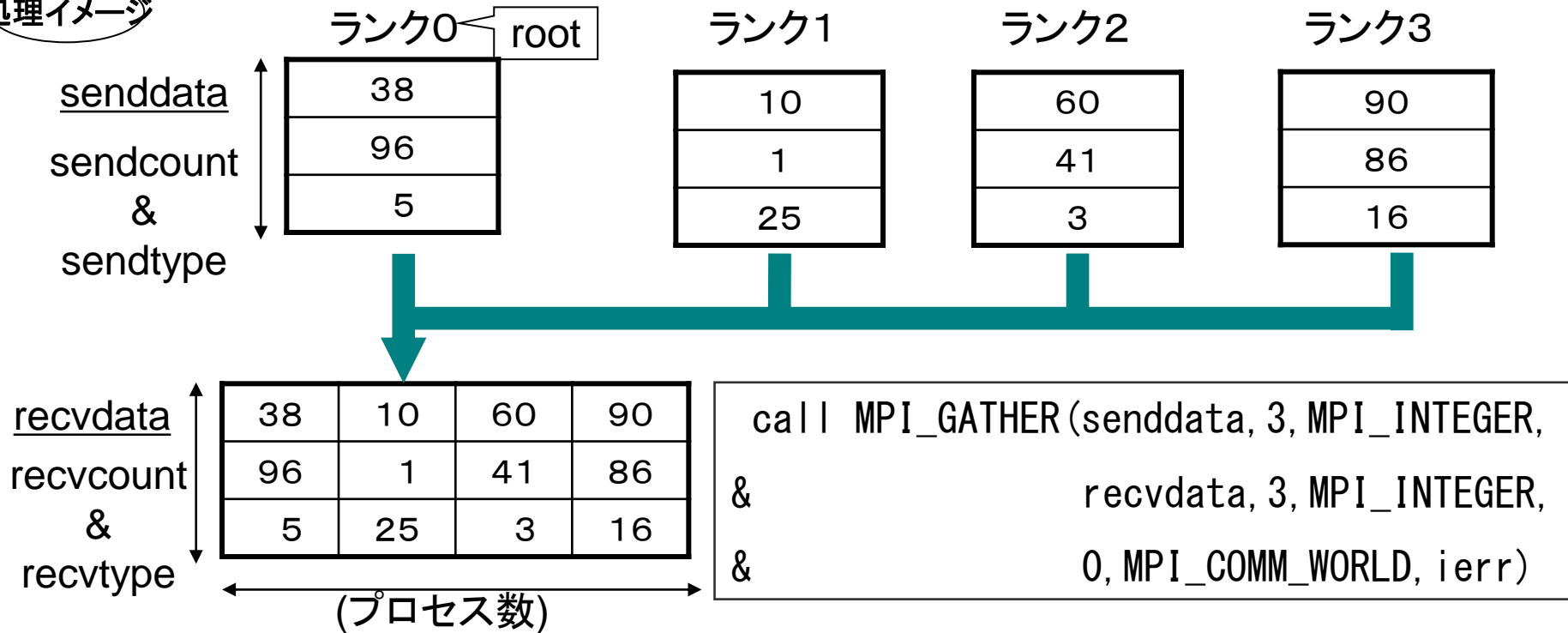


# 付録1. 3. 8 MPI\_GATHER データの集積

## 機能概要

- コミュニケータcomm内の全プロセスの送信バッファ(senddata)から、1つのプロセス(root)の受信バッファ(recvdata)へメッセージを送信する
- メッセージの長さは一定で、送信元プロセスのランクが小さい順に受信バッファに格納される

## 処理イメージ



# MPI\_GATHER(続き)

## 書式

```
任意の型 senddata(*), recvdata(*)
integer sendcount, sendtype, recvcount, recvtype,
 root, comm, ierr
call MPI_GATHER(senddata, sendcount, sendtype,
 recvdata, recvcount, recvtype,
 root, comm, ierr)
```

```
int MPI_Gather(void* senddata, int sendcount,
 MPI_Datatype sendtype, void* recvarea,
 int recvcount, MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

# MPI\_GATHER( 続き )

## 引数

| 引数        | 値      | 入出力 |                    |
|-----------|--------|-----|--------------------|
| senddata  | 任意     | IN  | 送信データの開始アドレス       |
| sendcount | 整数     | IN  | 送信データの要素の数         |
| sendtype  | handle | IN  | 送信データのタイプ          |
| recvdata  | 任意     | OUT | 受信領域の開始アドレス ☆      |
| recvcount | 整数     | IN  | 個々のプロセスから受信する要素数 ☆ |
| recvtype  | handle | IN  | 受信領域のデータタイプ ☆      |
| root      | 整数     | IN  | rootプロセスのランク       |
| comm      | handle | IN  | コミュニケータ            |

☆…rootプロセスだけ意味を持つ

## メモ

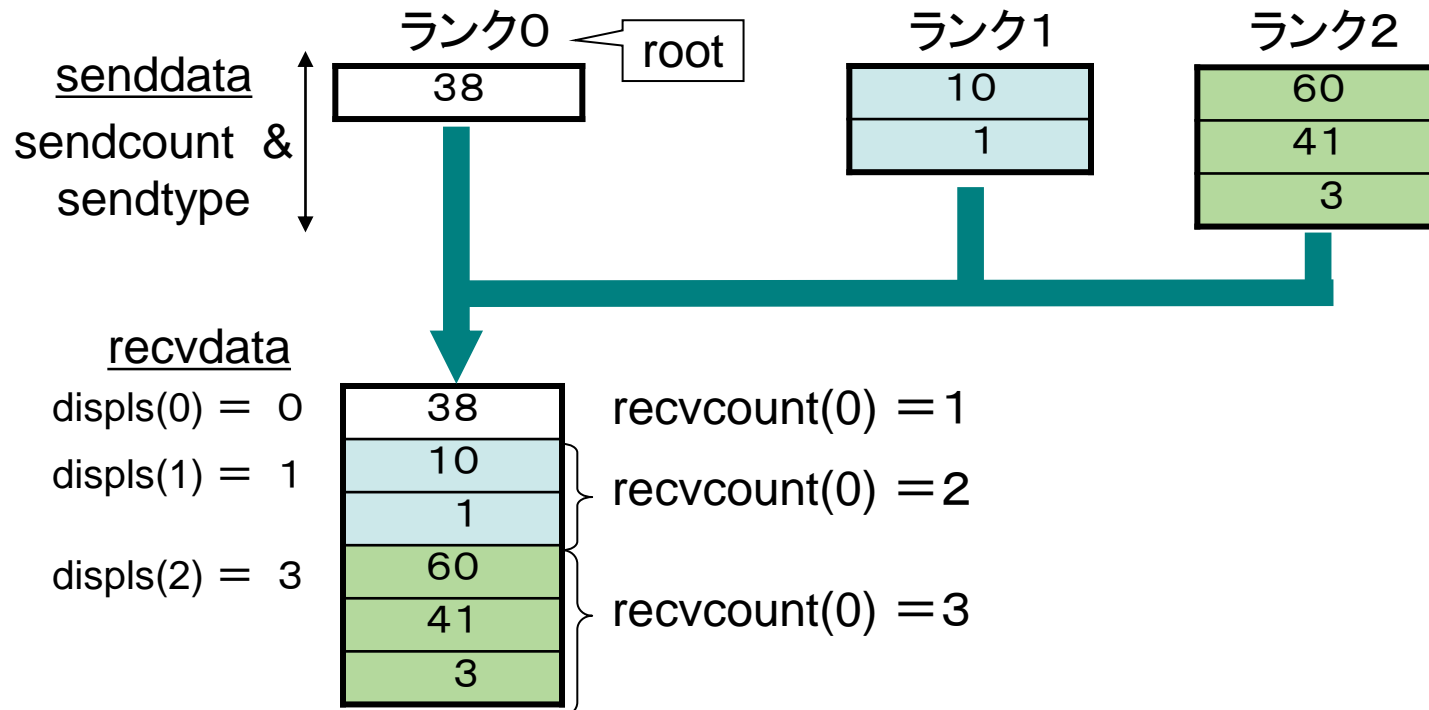
- ・ メッセージの長さは一定で、送信元プロセスのランクが小さい順に受信バッファに格納される

# 付録1. 3. 9 MPI\_GATHERV データの集積

## 機能概要

- コミュニケータcomm内の全プロセスの送信バッファ(senddata)から、1つのプロセス(root)の受信バッファ(recvdata)へメッセージを送信する
- 送信元毎に受信データ長(recvcnt)と受信バッファ内の位置(displs)を変えることができる

## 処理イメージ





# MPI\_GATHERV( 続き )

## 書式

```
任意の型 senddata (*), recvdata (*)
integer sendcount, sendtype, recvcount (*),
 displs (*), recvtype, root, comm, ierr
call MPI_GATHERV(senddata, sendcount, sendtype,
 recvdata, recvcount, displs,
 recvtype, root, comm, ierr)
```

```
int MPI_Gatherv(void* senddata, int sendcount,
 MPI_Datatype sendtype, void* recvdata,
 int *recvcount, int *displs,
 MPI_Datatype recvtype, int root,
 MPI_Comm comm)
```

## MPI\_GATHERV( 続き )

| 引数        | 値      | 入出力 |                                     |
|-----------|--------|-----|-------------------------------------|
| senddata  | 任意     | IN  | 送信データの開始アドレス                        |
| sendcount | 整数     | IN  | 送信データの要素の                           |
| sendtype  | handle | IN  | 送信データのタイプ                           |
| recvdata  | 任意     | OUT | 受信領域の開始アドレス ☆                       |
| recvcount | 整数     | IN  | 個々のプロセスから受信する<br>要素数の配列 ☆           |
| displs    | 整数     | IN  | 受信データを置き始めるrecvdataからの<br>相対位置の配列 ☆ |
| recvtype  | handle | IN  | 受信領域のデータタイプ ☆                       |
| root      | 整数     | IN  | rootプロセスのランク                        |
| comm      | handle | IN  | コミュニケータ                             |

☆ …rootプロセスだけが意味を持つ

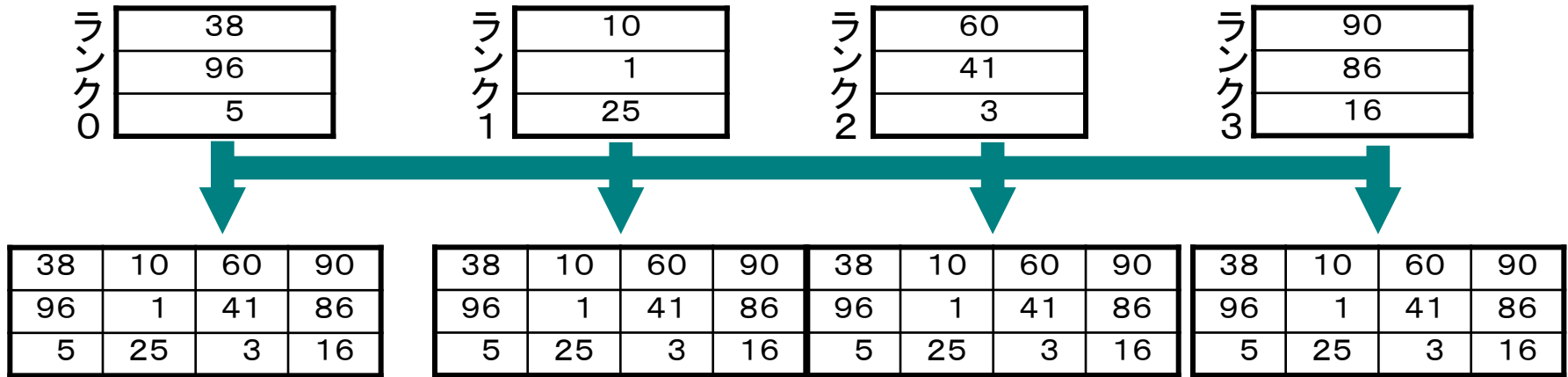
# 付録1. 3. 10 MPI\_ALLGATHER 全プロセスでデータ集積

## 機能概要

- コミュニケータ(comm)内の全プロセスの送信バッファ(senddata)から、全プロセスの受信バッファ(recvdata)へ互いにメッセージを送信する
- メッセージの長さは一定で、送信元プロセスのランクが小さい順に受信バッファに格納される

## 処理イメージ

### 1.3.10 MPI\_ALLGATHER 全プロセスでデータ集積



```
call MPI_ALLGATHER (senddata, 3, MPI_INTEGER,
& recvdata, 3, MPI_INTEGER,
& 0, MPI_COMM_WORLD, ierr)
```

# MPI\_ALLGATHER( 続き )

## 書式

```
任意の型 senddata(*), recvdata(*)
integer sendcount, sendtype, recvcount, recvtype,
 comm, ierr
call MPI_ALLGATHER(senddata, sendcount, sendtype,
 recvdata, recvcount, recvtype,
 comm, ierr)
```

```
int MPI_Allgather(void* senddata, int sendcount,
 MPI_Datatype sendtype, void* recvdata,
 int recvcount, MPI_Datatype recvtype,
 MPI_Comm comm)
```

# MPI\_ALLGATHER( 続き )

## 引数

| 引数        | 値      | 入出力 |                   |
|-----------|--------|-----|-------------------|
| senddata  | 任意     | IN  | 送信領域の開始アドレス       |
| sendcount | 整数     | IN  | 送信データの要素の数        |
| sendtype  | handle | IN  | 送信データのタイプ         |
| recvdata  | 任意     | OUT | 受信領域の開始アドレス       |
| recvcount | 整数     | IN  | 個々のプロセスから受信する要素の数 |
| recvtype  | handle | IN  | 受信データのタイプ         |
| comm      | handle | IN  | コミュニケータ           |

## メモ

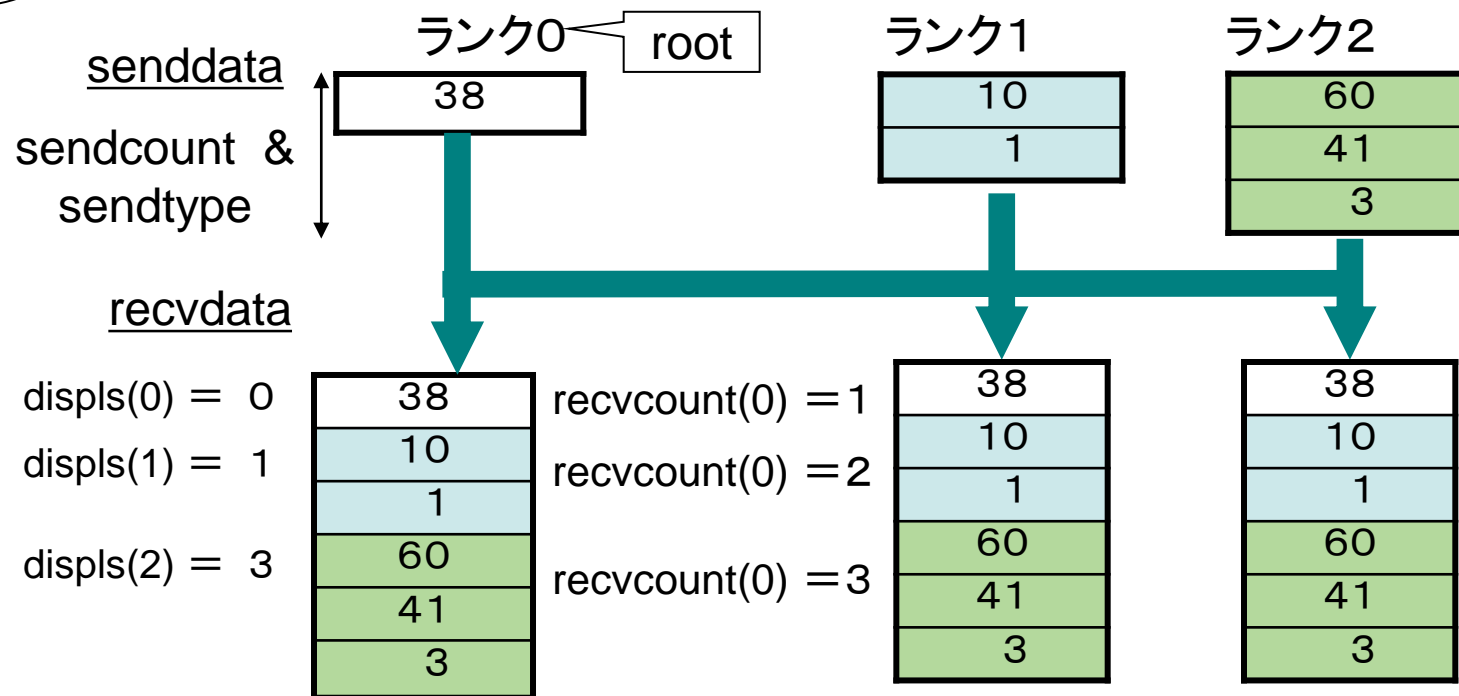
- ・ MPI\_GATHERの結果を全プロセスに送信するのと機能的に同じ

# 付録1. 3. 11 MPI\_ALLGATHERV 全プロセスでデータ集積

## 機能概要

- コミュニケータcomm内の全プロセスの送信バッファ(senddata)から、全プロセスの受信バッファ(recvdata)へメッセージを送信する
- 送信元毎に受信データ長(recvcount)と受信バッファ内の位置(displs)を変えることができる

## 処理イメージ



# MPI\_ALLGATHERV(続き)

## 書式

```
任意の型 senddata(*), recvdata(*)
integer sendcount, sendtype, recvcount(*), displs(*),
 recvtype, comm, ierr
call MPI_ALLGATHERV(senddata, sendcount, sendtype,
 recvdata, recvcount, displs,
 recvtype, comm, ierr)
```

```
int MPI_Allgatherv(void* senddata, int sendcount,
 MPI_Datatype sendtype, void* recvdata,
 int *recvcount, int *displs,
 MPI_Datatype recvtype, MPI_Comm comm)
```

# MPI\_ALLGATHERV( 続き )

## 引数

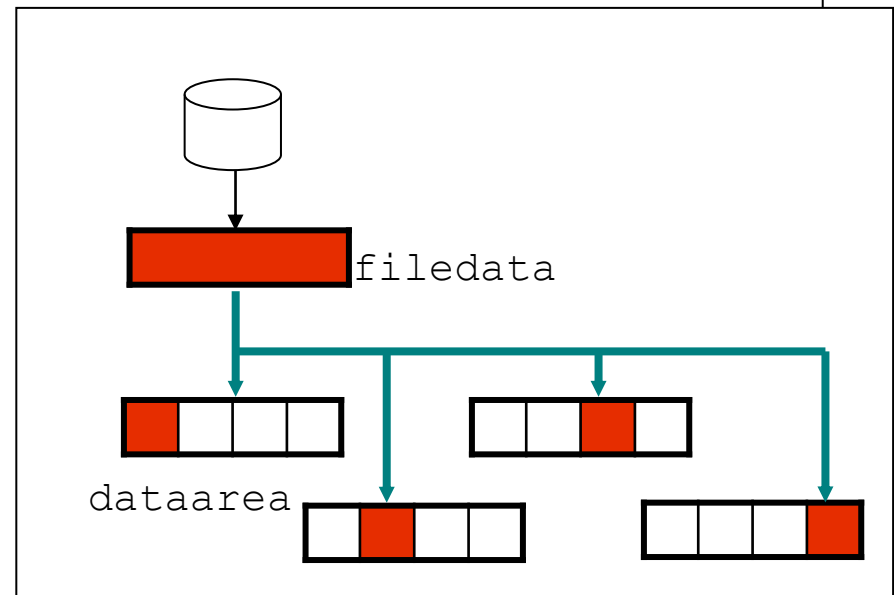
| 引数        | 値      | 入出力 |                                |
|-----------|--------|-----|--------------------------------|
| senddata  | 任意     | IN  | 送信領域の開始アドレス                    |
| sendcount | 整数     | IN  | 送信データの要素の数                     |
| sendtype  | handle | IN  | 送信データのタイプ                      |
| recvdata  | 任意     | OUT | 受信領域の開始アドレス                    |
| recvcount | 整数     | OUT | 受信データの要素の数                     |
| displs    | 整数     | IN  | 受信データを置くrecvdataからの相対位置(プロセス毎) |
| recvtype  | handle | IN  | 受信データのタイプ                      |
| comm      | handle | IN  | コミュニケータ                        |



# 付録1.3.12 プログラム例(代表プロセスによるファイル入力)

```
include 'mpif.h'
integer filedata(100),dataarea(100)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
icount=(100-1)/nprocs+1
if(myrank==0)then
 open(10,file='fort.10')
 read(10,*)filedata
end if
call MPI_SCATTER(filedata, icount, MPI_INTEGER,
& dataarea(icount*myrank+1), icount, MPI_INTEGER,
& 0, MPI_COMM_WORLD,ierr)
isum1=0
ist=icount*myrank+1
ied=icount*(myrank+1)
do i=ist,ied
 isum1=isum1+dataarea(i)
enddo
call MPI_REDUCE(isum1, isum, 1,
& MPI_INTEGER, MPI_SUM,
& 0, MPI_COMM_WORLD, ierr)
if(myrank==0)
& write(6,*)'sum=',isum
call MPI_FINALIZE(ierr)
stop
end
```

etc12.f

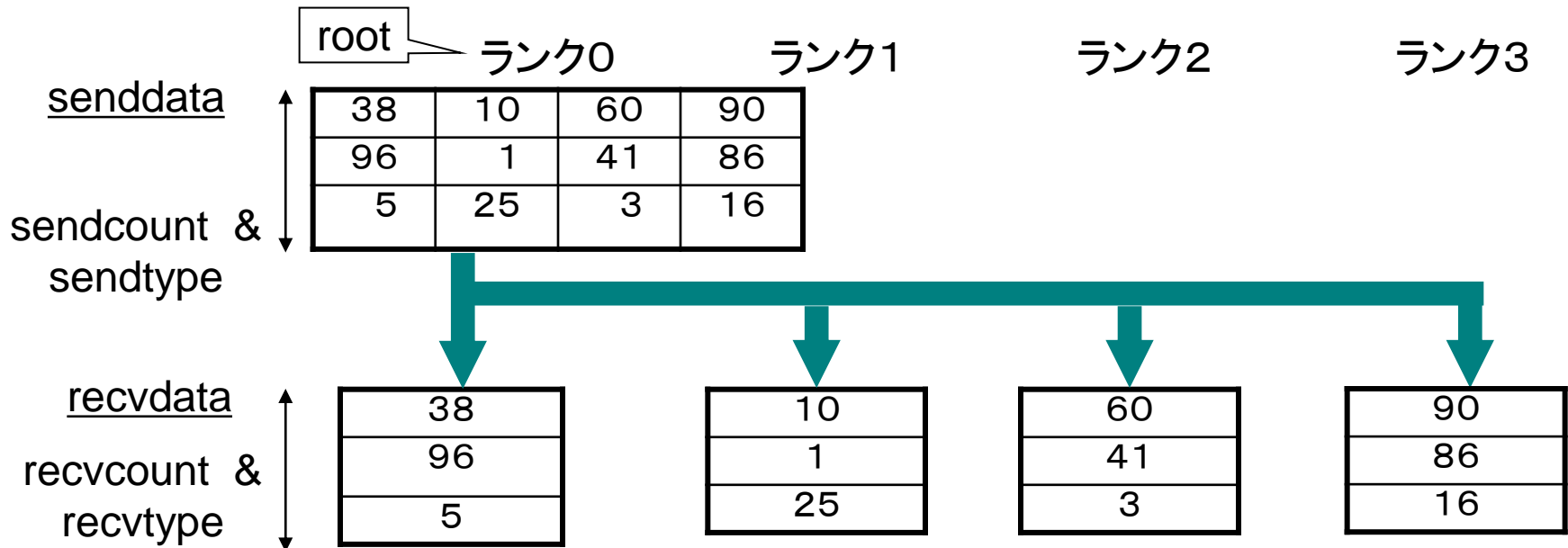


# 付録1.3.13 MPL\_SCATTER データの分配

## 機能概要

- 一つの送信元プロセス(root)の送信バッファ(senddata)から、コミュニケータcomm内の全てのプロセスの受信バッファ(recvdata)にデータを送信する
- 各プロセスへのメッセージ長は一定である

## 処理イメージ



# MPI\_SCATTER(続き)

## 書式

```
任意の型 senddata(*), recvdata(*),
integer sendcount, sendtype, recvcount, recvtype,
 root, comm, ierr
call MPI_SCATTER (senddata, sendcount, sendtype,
 recvdata, recvcount, recvtype,
 root, comm, ierr)
```

```
int MPI_Scatter(void* senddata, int sendcount,
 MPI_Datatype sendtype, void* recvdata,
 int recvcount, MPI_Datatype recvtype,
 int root, MPI_Comm comm)
```

# MPL\_SCATTER( 続き )

## 引数

| 引数        | 値      | 入出力 |                  |
|-----------|--------|-----|------------------|
| senddata  | 任意     | IN  | 送信領域のアドレス ☆      |
| sendcount | 整数     | IN  | 各プロセスへ送信する要素数 ☆  |
| sendtype  | handle | IN  | 送信領域の要素のデータタイプ ☆ |
| recvdata  | 任意     | OUT | 受信データのアドレス       |
| recvcount | 整数     | IN  | 受信データの要素の数       |
| recvtype  | handle | IN  | 受信データのタイプ        |
| root      | 整数     | IN  | rootプロセスのランク     |
| comm      | handle | IN  | コミュニケータ          |

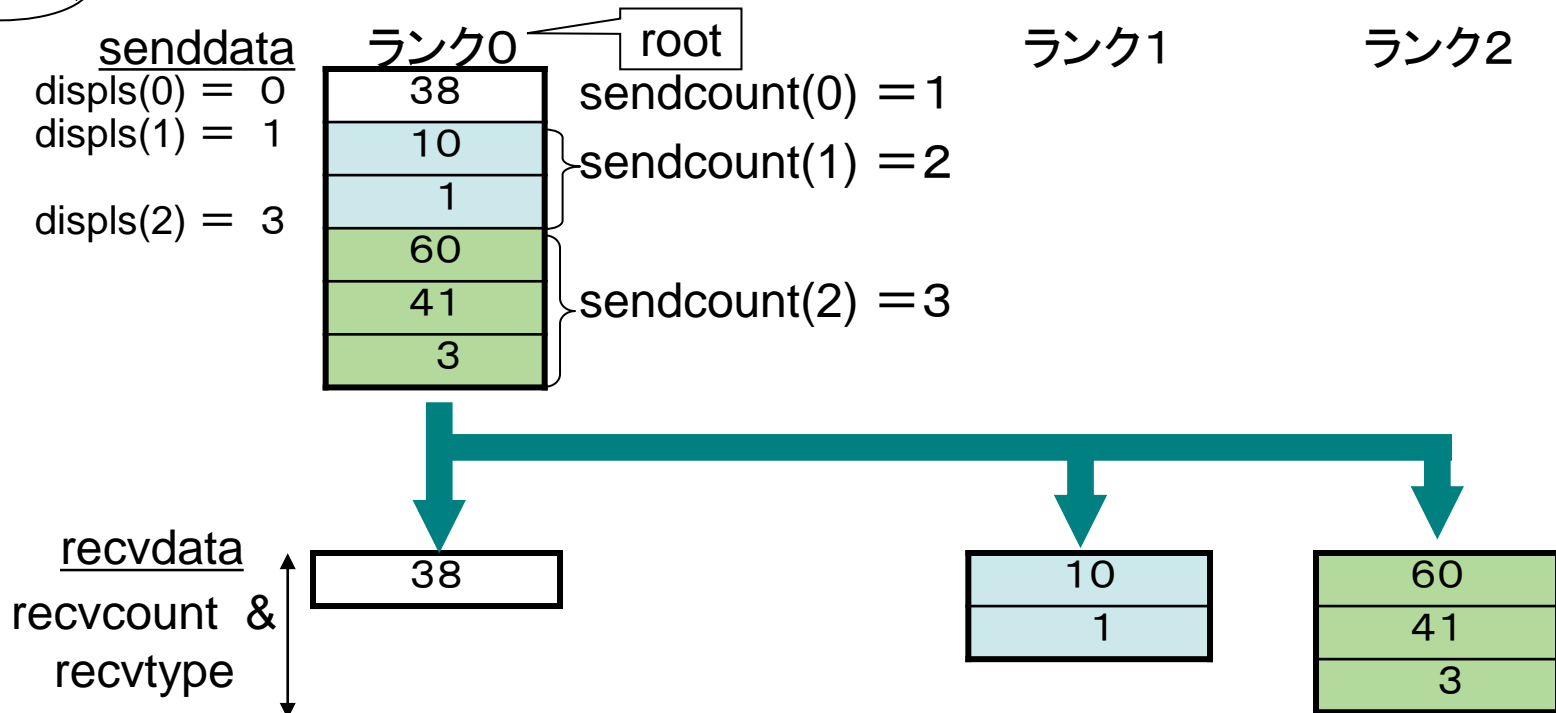
☆… rootプロセスだけ意味を持つ

# 付録1.3.14 MPL\_SCATTERV データの分配

## 機能概要

- 一つの送信元プロセス(root)の送信バッファ(senddata)から、コミュニケータcomm内の全てのプロセスの受信バッファ(recvdata)にデータを送信する
- 送信先毎に送信データ長(sendcount)とバッファ内の位置(displs)を変えることができる

## 処理イメージ



# MPI\_SCATTERV( 続き )

## 書式

```
任意の型 senddata(*), recvdata(*)
integer sendcount(*), displs(*), sendtype, recvcount,
 recvtype, root, comm, ierr
call MPI_SCATTERV(senddata, sendcount, displs, sendtype,
 recvdata, recvcount, recvtype, root,
 comm, ierr)
```

```
int MPI_Scatterv(void* senddata, int *sendcount,
 int *displs, MPI_Datatype sendtype,
 void* recvdata, int recvcount,
 MPI_Datatype recvtype, int root,
 MPI_Comm comm)
```

# MPI\_SCATTERV( 続き )

## 引数

| 引数        | 値      | 入出力 |                                      |
|-----------|--------|-----|--------------------------------------|
| senddata  | 任意     | IN  | 送信領域のアドレス ☆                          |
| sendcount | 整数     | IN  | 各プロセスへ送信する要素数 ☆                      |
| displs    | 整数     | IN  | プロセス毎の送信データの始まる<br>senddataからの相対位置 ☆ |
| sendtype  | handle | IN  | 送信データのタイプ ☆                          |
| recvdata  | 任意     | OUT | 受信データのアドレス                           |
| recvcount | 整数     | IN  | 受信データの要素の数                           |
| recvtype  | handle | IN  | 受信データのタイプ                            |
| root      | 整数     | IN  | rootプロセスのランク                         |
| comm      | handle | IN  | コミュニケータ                              |

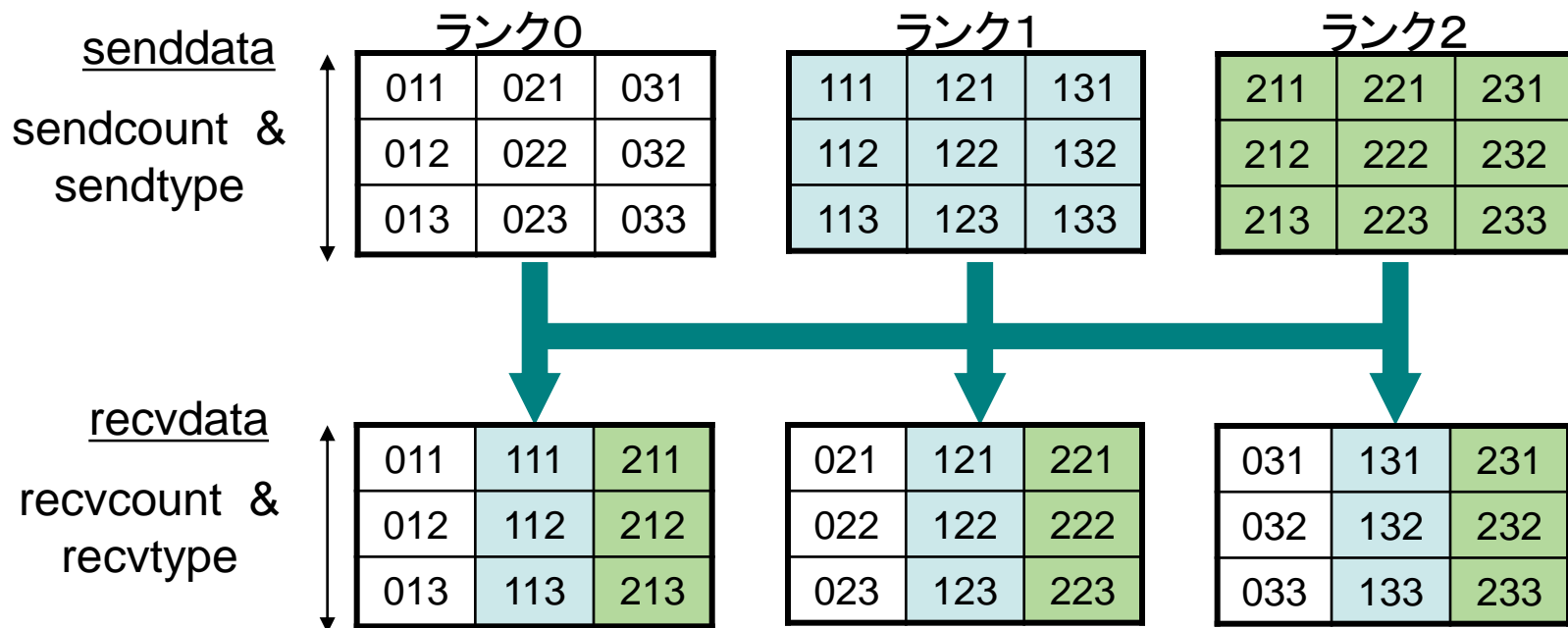
☆… rootプロセスだけ意味を持つ

# 付録1.3.15 MPI\_ALLTOALL データ配置

## 機能概要

- コミュニケータcomm内の全プロセスが、それぞれの送信バッファ (senddata)から、他の全てのプロセスの受信バッファ(recvdata)にデータを分配する
- 各プロセスへのメッセージ長は一定である

## 処理イメージ





# MPI\_ALLTOALL(続き)

## 書式

```
任意の型 senddata(*), recvdata(*)
integer sendcount, sendtype, recvcount, recvtype,
 comm, ierr
call MPI_ALLTOALL(senddata, sendcount, sendtype,
 recvdata, recvcount, recvtype,
 comm, ierr)
```

```
int MPI_Alltoall(void* senddata, int sendcount,
 MPI_Datatype sendtype, void* recvdata,
 int recvcount, MPI_Datatype recvtype,
 MPI_Comm comm)
```

# MPI\_ALLTOALL(続き)

## 引数

| 引数        | 値      | 入出力 |                 |
|-----------|--------|-----|-----------------|
| senddata  | 任意     | IN  | 送信領域の開始アドレス     |
| sendcount | 整数     | IN  | 各プロセスへ送信する要素の数  |
| sendtype  | handle | IN  | 送信データのタイプ       |
| recvdata  | 任意     | OUT | 受信領域の開始アドレス     |
| recvcount | 整数     | IN  | 各プロセスから受信する要素の数 |
| recvtype  | handle | IN  | 受信データのタイプ       |
| comm      | handle | IN  | コミュニケータ         |

## メモ

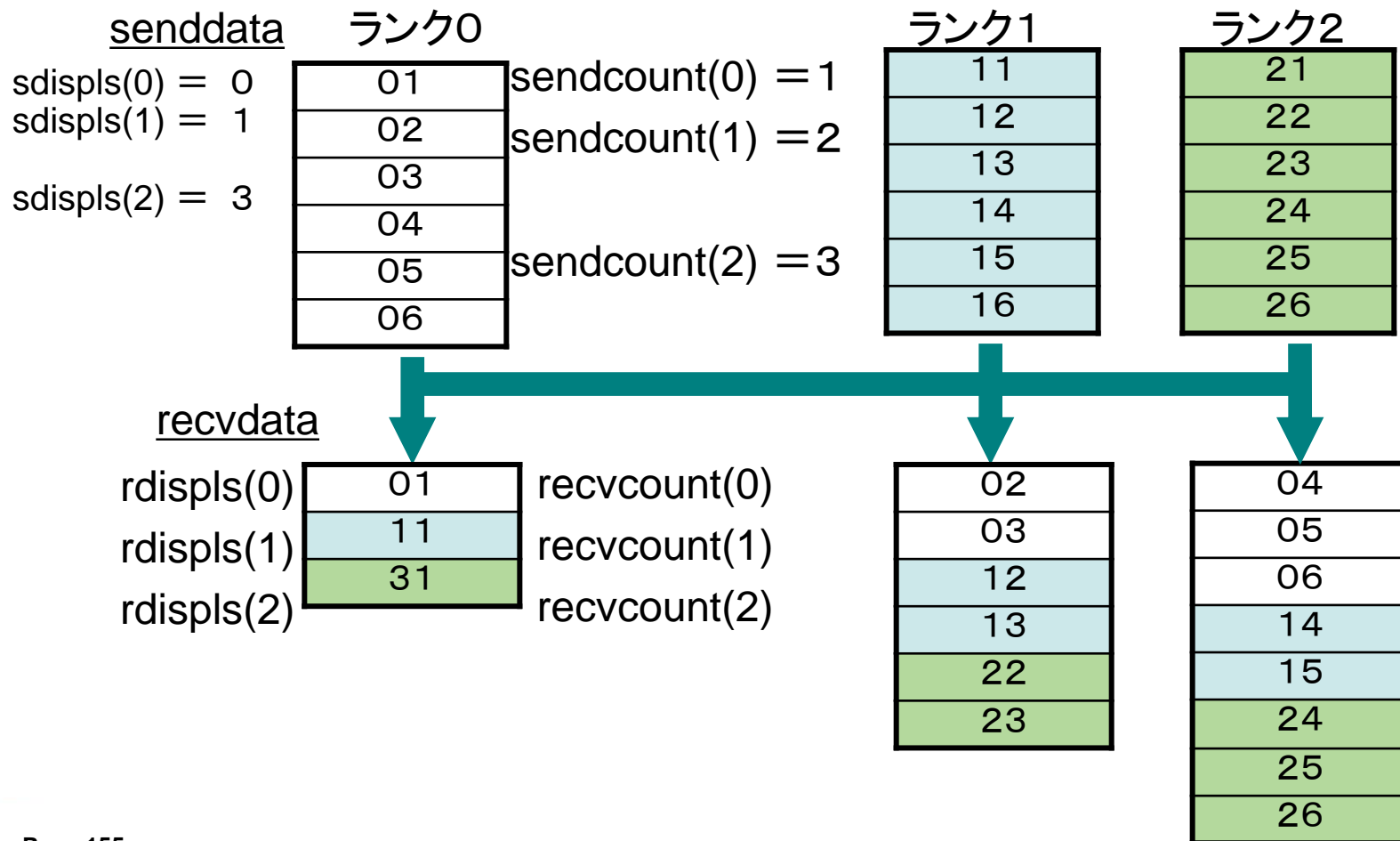
- 全対全スキヤッタ／ギャザ, または全交換とも呼ばれる

# 付録1.3.16 MPI\_ALLTOALLV データ配置

## 機能概要

- コミュニケータcomm内の全プロセスが、それぞれの送信バッファ(senddata)から他の全てのプロセスの受信バッファ(recvdata)にデータを分配する
- 送信元毎にメッセージ長を変えることができる

## 処理イメージ



# MPI\_ALLTOALLV(続き)

## 書式

```
任意の型 senddata(*), recvdata(*)
integer sendcount(*), sdispls(*), sendtype,
 recvcount(*), rdispls(*), recvtype,
 comm, ierr
call MPI_ALLTOALLV(senddata, sendcount, sdispls, sendtype,
 recvdata, recvcount, rdispls, recvtype,
 comm, ierr)
```

```
int MPI_Alltoallv(void* senddata, int *sendcount,
 int *sdispls, MPI_Datatype sendtype,
 void* recvdata, int *recvcount,
 int *rdispls, MPI_Datatype recvtype,
 MPI_Comm comm)
```

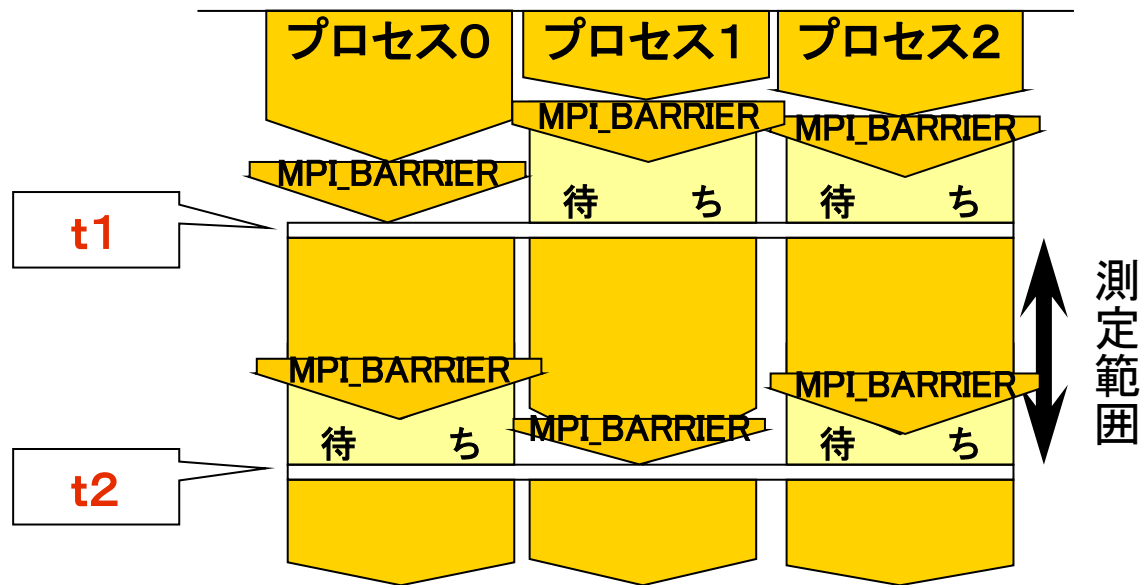
# MPI\_ALLTOALLV(続き)

## 引数

| 引数        | 値      | 入出力 |                                   |
|-----------|--------|-----|-----------------------------------|
| senddata  | 任意     | IN  | 送信領域の開始アドレス                       |
| sendcount | 整数     | IN  | 送信する要素の数(プロセス毎)                   |
| sdispls   | 整数     | IN  | 送信データの始まるsenddataからの相対位置(プロセス毎)   |
| sendtype  | handle | IN  | 送信データのデータタイプ                      |
| recvdata  | 任意     | OUT | 受信領域の開始アドレス                       |
| recvcount | 整数     | IN  | 受信する要素の数(プロセス毎)                   |
| rdispls   | 整数     | IN  | 受信データを置き始めるrecvdataからの相対位置(プロセス毎) |
| recvtype  | handle | IN  | 受信バッファの要素のデータタイプ                  |
| comm      | handle | IN  | コミュニケータ                           |

# 付録1.4 その他の手続き

## 付録1.4.1 計時(イメージ)



$$(\text{測定時間}) = t2 - t1$$

# 計時プログラム例

etc13.f

```
include 'mpif.h'
parameter(numdat=100)
real*8 t1,t2,tt
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
ist=((numdat-1)/nprocs+1)*myrank+1
ied=((numdat-1)/nprocs+1)*(myrank+1)
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
t1=MPI_WTIME()
isum=0
do i=ist,ied
 isum=isum+i
enddo
call MPI_REDUCE(isum,isum0,1,MPI_INTEGER,
& MPI_SUM,0,MPI_COMM_WORLD,ierr)
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
t2=MPI_WTIME()
tt=t2-t1
if(myrank.eq.0)write(6,*)'sum=',isum0,',',time=',tt
call MPI_FINALIZE(ierr)
stop
end
```

## 付録1.4.2 MPI\_WTIME 経過時間の測定

### 機能概要

- 過去のある時刻からの経過時間(秒数)を倍精度実数で返す

### 書式

```
DOUBLE PRECISION MPI_WTIME ()
```

```
double MPI_Wtime (void)
```

### メモ

- 引数はない
- この関数を実行したプロセスのみの時間を取得できる
  - プログラム全体の経過時間を知るには同期を取る必要がある
- 得られる値は経過時間であり, システムによる中断があればその時間も含まれる



# 付録1.4.3 MPI\_BARRIER バリア同期

## 機能概要

- コミュニケータ(comm)内の全てのプロセスで同期をとる

## 書式

```
integer comm, ierr
call MPI_BARRIER (comm, ierr)
```

```
int MPI_Barrier (MPI_Comm comm)
```

## 引数

| 引数   | 値      | 入出力 |         |
|------|--------|-----|---------|
| comm | handle | IN  | コミュニケータ |

## メモ

- MPI\_BARRIERをコールすると, commに含まれる全てのプロセスがMPI\_BARRIERをコールするまで待ち状態に入る

# 付録1.5 プログラミング作法

## 1. FORTRAN

- ① ほとんどのMPI手続きはサブルーチンであり、引数の最後に整数型の返却コード(本書では*ierr*)を必要とする
- ② 関数は引数に返却コードを持たない

## 2. C

- ① 接頭辞MPI\_とそれに続く1文字は大文字、以降の文字は小文字
- ② 但し、定数はすべて大文字
- ③ ほとんどの関数は戻り値として返却コードを返すため、引数に返却コードは必要ない

## 3. 共通

- ① 引数説明にある「*handle*」は、FORTRANでは整数型、Cでは書式説明に記載した型を指定する
- ② 引数説明にある「*status*」は、FORTRANではMPI\_STATUS\_SIZEの整数配列、CではMPI\_Status型の構造体を指定する
- ③ 接頭辞MPI\_で始まる変数や関数は宣言しない方がよい
- ④ 成功した場合の返却コードはMPI\_SUCCESSとなる

## 付録2. 参考文献, Webサイト

### MPI並列プログラミング, Peter S. Pacheco著, 秋葉 博訳



出版社: 培風館 (2001/07)  
ISBN-10: 456301544X  
ISBN-13: 978-4563015442

### 「並列プログラミング入門 MPI版」

➤ (旧「並列プログラミング虎の巻MPI版」)(青山幸也 著)

---

# 演習問題解答例

### 3. 演習問題1-2 (practice\_1) 解答例

```
program example1
 include 'mpif.h'
 integer ierr,myrank
 call MPI_INIT(ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
 if(myrank.eq.0) print *, "Hello World",myrank
 call MPI_FINALIZE(ierr)
 stop
end
```

```
% mpifort practice1.f
% qsub run.sh
% cat run.sh.oXXXX
Hello World 0
```

## 4. 演習問題2 (practice\_2) 解答例

```
program example2
 include 'mpif.h'
 integer ierr,myrank,nprocs,ist,ied
 parameter (n=1000)
 integer isum
 call MPI_INIT(ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
 ist=((n-1)/nprocs+1)*myrank+1
 ied=((n-1)/nprocs+1)*(myrank+1)
 isum=0
 do i=ist,ied
 isum=isum+i
 enddo
 write(6,6000) myrank, isum
6000 format("Total of Rank:",i2,i10)
 call MPI_FINALIZE(ierr)
 stop
end
```

```
% mpifort practice2.f
```

```
% qsub run.sh
```

```
% cat run.sh.oXXXX
```

```
Total of Rank: 0 31375
```

```
Total of Rank: 2 156375
```

```
Total of Rank: 3 218875
```

```
Total of Rank: 1 93875
```

## 5. 演習問題3 (practice\_3) 解答例

```
program example3
include 'mpif.h'
integer ierr,myrank,nprocs,ist,ied
integer status(MPI_STATUS_SIZE)
parameter (n=1000)
integer isum, isum2
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
ist=((n-1)/nprocs+1)*myrank+1
ied=((n-1)/nprocs+1)*(myrank+1)
isum=0
do i=ist,ied
 isum=isum+i
enddo
```

## 5. 演習問題3 (practice\_3) 解答例(つづき)

```
itag=1
if(myrank.ne.0) then
 call MPI_SEND(isum, 1, MPI_INTEGER, 0,
& itag, MPI_COMM_WORLD, ierr)
else
 call MPI_RECV(isum2, 1, MPI_INTEGER, 1,
& itag, MPI_COMM_WORLD, status, ierr)
 isum=isum+isum2
 call MPI_RECV(isum2, 1, MPI_INTEGER, 2,
& itag, MPI_COMM_WORLD, status, ierr)
 isum=isum+isum2
 call MPI_RECV(isum2, 1, MPI_INTEGER, 3,
& itag, MPI_COMM_WORLD, status, ierr)
 isum=isum+isum2
 write(6, 6000) isum
6000 format("Total Sum = ", i10)
endif
call MPI_FINALIZE(ierr)
stop
end
```

```
% mpinfort practice3.f
% qsub run.sh
% cat run.sh.oXXXX
Total Sum = 500500
```



## 6. 演習問題4 (practice\_4) 解答例

```
program example4
include 'mpif.h'
integer ierr,myrank,nprocs,ist,ied
parameter (n=1000)
integer isum,isum2
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
ist=((n-1)/nprocs+1)*myrank+1
ied=((n-1)/nprocs+1)*(myrank+1)
isum=0
do i=ist,ied
 isum=isum+i
enddo
```

## 6. 演習問題4 (practice\_4) 解答例 (つづき)

```
call MPI_REDUCE(isum, isum2, 1, MPI_INTEGER, MPI_SUM, 0,
& MPI_COMM_WORLD, ierr)
if(myrank.eq.0) write(6, 6000) isum2
6000 format("Total Sum = ", i10)
call MPI_FINALIZE(ierr)
stop
end
```

```
% mpifort practice4.f
% qsub run.sh
% cat run.sh.oXXXX
Total Sum = 500500
```

※ MPI\_REDUCEでは送信するデータと受信するデータの領域に重なりがあってはならない。isumとisum2に分けて使用。

## 8. 演習問題5 (practice\_5) 解答例

```
include 'mpif.h'
integer, parameter :: numdat=100
integer, allocatable :: senddata (:), recvdata (:)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
ist = ((numdat-1)/nprocs+1)*myrank+1
ied = ((numdat-1)/nprocs+1)*(myrank+1)
allocate(senddata(ist:ied))
if(myrank.eq.0) allocate(recvdata(numdat))
icount=(numdat-1)/nprocs+1
do i=1, icount
 senddata(icount*myrank+i)=icount*myrank+i
enddo
```

## 8. 演習問題5 (practice\_5) 解答例(つづき)

```
call MPI_GATHER(senddata(icount*myrank+1),
& icount, MPI_INTEGER, recvdata,
& icount, MPI_INTEGER, 0, MPI_COMM_WORLD,
& ierr)
if(myrank.eq.0) then
 open(60, file='fort.60')
 write(60, '(10I8)') recvdata
endif
call MPI_FINALIZE(ierr)
stop
end
```

```
% mpifort practice5.f
```

```
% qsub run.sh
```

```
% cat fort.60
```

```
1 2 3 4 5 ...
```

## 9. 演習問題6 (practice\_6) 解答例

```
program example6
implicit real (8) (a-h, o-z)
include 'mpif.h'
integer ierr, myrank, nprocs, ist, ied
parameter (n=12000)
real (8) a(n, n), b(n, n), c(n, n)
real (8) d(n, n)
real (8) t1, t2
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
ist=((n-1)/nprocs+1)*myrank+1
ied=((n-1)/nprocs+1)*(myrank+1)
n2=n/nprocs
```

## 9. 演習問題6 (practice\_6) 解答例(つづき)

```
do j = 1, n
 do i = 1, n
 a(i, j) = 0.0d0
 b(i, j) = n+1-max(i, j)
 c(i, j) = n+1-max(i, j)
 enddo
enddo
if(myrank.eq.0) then
write(6,50) ' Matrix Size = ', n
endif
50 format(1x, a, i5)
```

## 9. 演習問題6 (practice\_6) 解答例(つづき)

```
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
t1=MPI_WTIME()
do j=ist, ied
 do k=1, n
 do i=1, n
 a(i, j)=a(i, j)+b(i, k)*c(k, j)
 end do
 end do
end do
call MPI_GATHER(a(1, ist), n*n2, MPI_REAL8, d, n*n2
& , MPI_REAL8, 0, MPI_COMM_WORLD, ierr)
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
t2=MPI_WTIME()
if(myrank.eq.0) then
write(6, 60) ' Execution Time = ', t2-t1, ' sec', ' A(n, n) = ', d(n, n)
endif
60 format(1x, a, f10.3, a, 1x, a, d24.15)
call MPI_FINALIZE(ierr)
stop
end
```

```
% mpinfort practice6.f
% qsub run.sh
% cat run.sh.oXXXX
Matrix Size = 12000
Execution Time = 27.358 sec
A(n,n) = 0.12000000000000000D+05
```