



TOHOKU
UNIVERSITY



Cyberscience
Center

NEC

Orchestrating a brighter world

2021年度 並列プログラミング入門Ⅱ (MPI)

2021年 10月 8日

東北大学サイバーサイエンスセンター

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た。
無断転載等は、ご遠慮下さい。

-
- 並列化概要
 - OpenMPプログラミング編
 - MPIプログラミング編

MPIプログラミング編

MPIプログラミング編・目次

1. MPI概要
 2. 演習問題1
 3. 演習問題2
 4. 演習問題3
 5. 演習問題4
 6. MPIプログラミング
 7. 演習問題5
 8. 演習問題6
- 付録1.主な手続き
- 2.参考文献, Webサイト

演習問題の構成

■ 演習問題の環境を自分のホームディレクトリ配下にコピーします。

```
/mnt/stfs/ap/lecture/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
```

```
$ cd <環境をコピーしたいディレクトリ>  
$ cp -r /mnt/stfs/ap/lecture/MPI/ .
```

1. 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

(サイバーサイエンスセンターのSX-Aurora TSUBASAでは、 12TByte
まで利用可能)

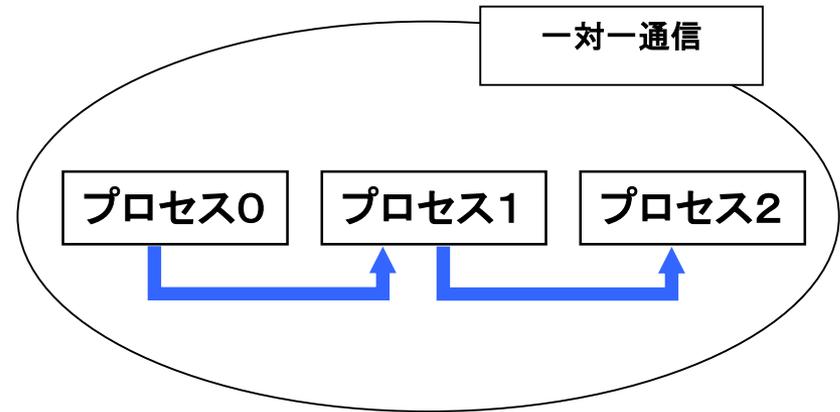
MPIの主な機能

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

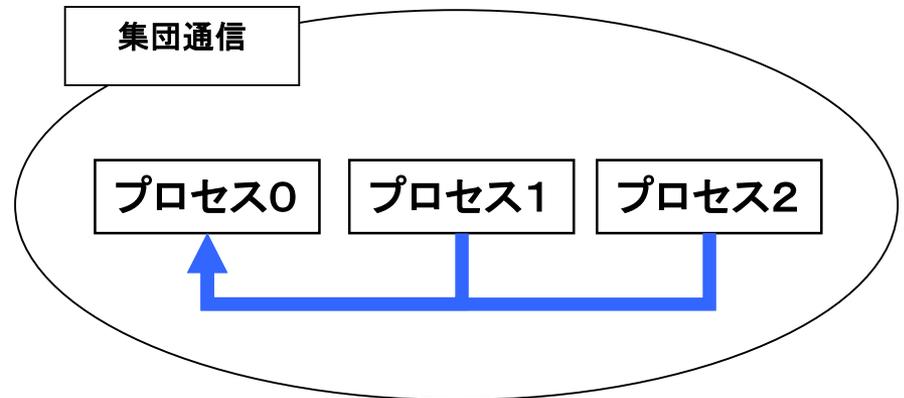
一対一通信

- 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
```

実行例

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

- 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

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に戻る
- 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に戻る
- 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) then  
  write(6,*) "..."  
endif
```

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 -np [総MPIプロセス数] ロードモジュール名
```

実行スクリプト例

32mpi 8smpのジョブを32ノード(32VE)で実行する際のスクリプト例

```
#!/bin/csh

#PBS -q sxf
#PBS --venode 32
#PBS -l elapstim_req=20:00:00
#PBS -N Test_Job
#PBS -v OMP_NUM_THREADS=8

cd $PBS_O_WORKDIR

mpirun -np 32 ./a.out
```

NQSVオプション(#PBSで指定)

- q キュー名を指定 (必須)
- venode 使用VE数を指定 (必須)
- l 経過時間(hh:mm:ss)の申告 (強く推奨)
- N ジョブ名を指定 (任意)
- v (実行する全てのノードに対して)環境変数を設定する

OMP_NUM_THREADS=8を実行する全てのノードに対して設定する

\$PBS_O_WORKDIR:ジョブスクリプトをqsubしたディレクトリ

SX-Aurora TSUBASAのジョブクラス

利用形態	キュー名	VE数	実行形態(※)	最大経過時間 既定値/最大値	メモリサイズ
無料	sxf	1	1VE	1時間/1時間	48GB×VE数
共有	sx	1	1VE	72時間/720時間	
		2~256	8VE単位で確保 (VHを共用しない)		
	sxmix	2~8	1VE単位で確保 (VHを共用する)		

MPIプログラム例(Hello World)

```
program sample1
write(6,*) "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_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
write(6,*) "Hello World My rank=",myrank,"(",nprocs,"processes)"
call MPI_FINALIZE(ierr)
stop
end
```

MPIプログラム

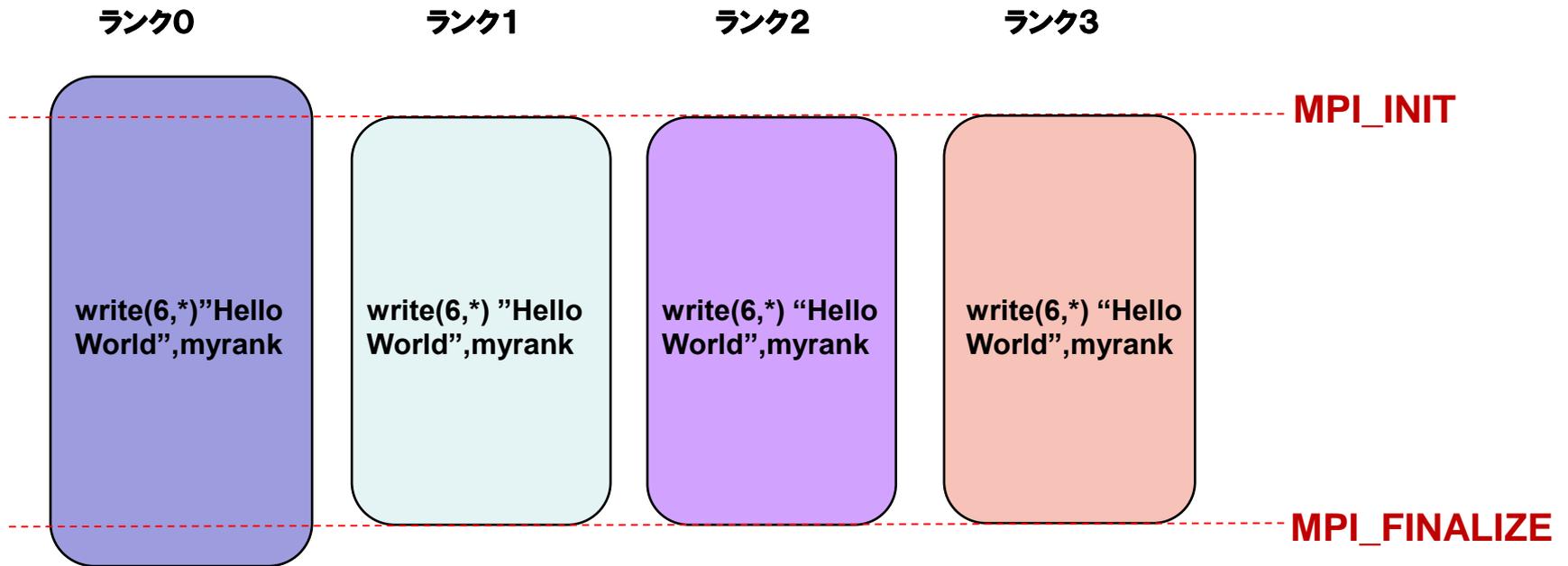
sample2.f

- 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 -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「`write(6, *) "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)

2. 演習問題1

- (1-1)P13 のプログラム(sample2.f)をコンパイル, 実行してください
- (1-2)P13 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

演習問題1-1 (practice_1)

■ P13のソースファイル(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

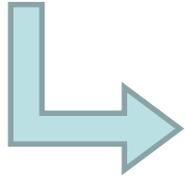
項目	対象	備考
作業ディレクトリ	practice_1	
使用ソースファイル	practice1.f (sample2.fをコピー)	編集 不要
ジョブファイル	run.sh	そのまま投入

- 手順① : 作業ディレクトリを移動して, ファイルをコピーしてください.
% cd MPI/practice_1
% cp ../sample/sample2.f practice1.f
- 手順② : コンパイルしてください.
% mpifort practice1.f

演習問題1-1 (practice_1)つづき

- 手順③：実行スクリプトを確認します。

```
% cat run.sh
```



```
#!/bin/csh
#PBS -q sx
#PBS --venode 1
#PBS -N practice_p1
cd $PBS_O_WORKDIR
mpirun -np 8 ./a.out
```

```
VEノード数 : 1
MPIプロセス数 : 8
ジョブクラス : sx
ジョブ名 : practice_p1
実行ディレクトリ : カレントディレクトリ
```

- 手順④：ジョブを投入します。

```
% qsub run.sh
```

- 手順⑤：実行結果を確認します。結果は practice_p1.oXXXX(XXXXにはリクエストIDが入ります)として格納されます。

```
% cat practice_p1.oXXXX
```

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

項目	対象	備考
作業ディレクトリ	practice_1	
使用ソースファイル	practice1-2.f	編集 必要
ジョブファイル	run.sh	そのまま投入

- 手順① : エディタでソースファイルを編集してください.
 % vi practice1-2.f (vi以外のエディタでも構いません)
 ※次ページにプログラム編集のヒントがあります.

演習問題1-2 (practice_1)

ヒント

以下の の部分を埋めてください。 (P13を参考にしてください)

```
program sample1_2

integer ierr,myrank,nprocs



if() write(6,*) "Hello World",myrank

stop
end
```

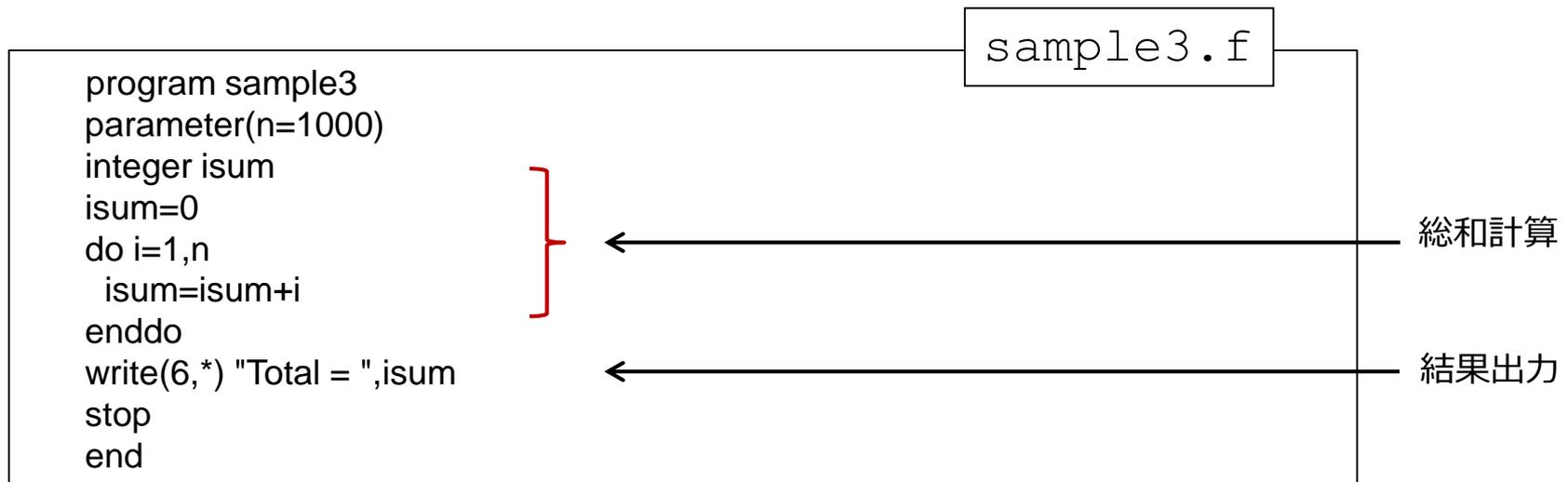
- MPIのインクルードファイルを指定
- MPIの初期化
- プログラム実行時のプロセス数の取得
- プロセス番号の取得
- IF文で, 自分自身がランク0である場合に出力を行うようにします.
- MPIの終了

演習問題1-2 (practice_1) つづき

- 手順②：コンパイルを実行します.
% mpinfort practice1-2.f
- 手順③：ジョブを投入します.
% qsub run.sh
- 手順④：実行結果を確認します。結果はpractice_p1.oXXXX(XXXXにはリクエストIDが入ります)として格納されます。
% cat practice_p1.oXXXX

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)

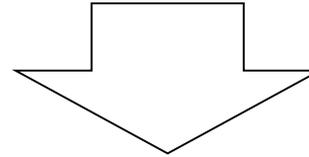


総和プログラムのMPI化

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, write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

i=1,250 で和を取る
i=251,500 で和を取る
i=501,750 で和を取る
i=751,1000 で和を取る
結果出力

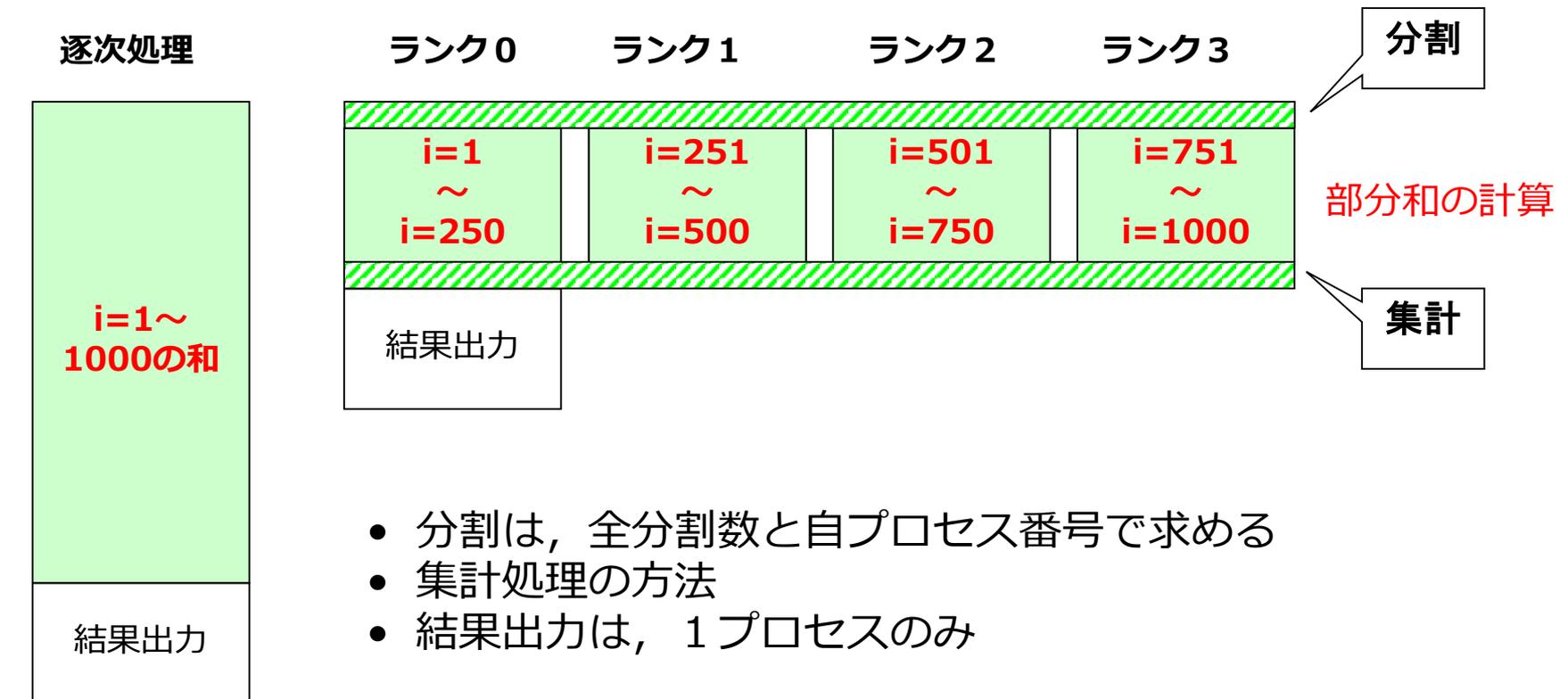
i=1,1000までの和を取る処理は,

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

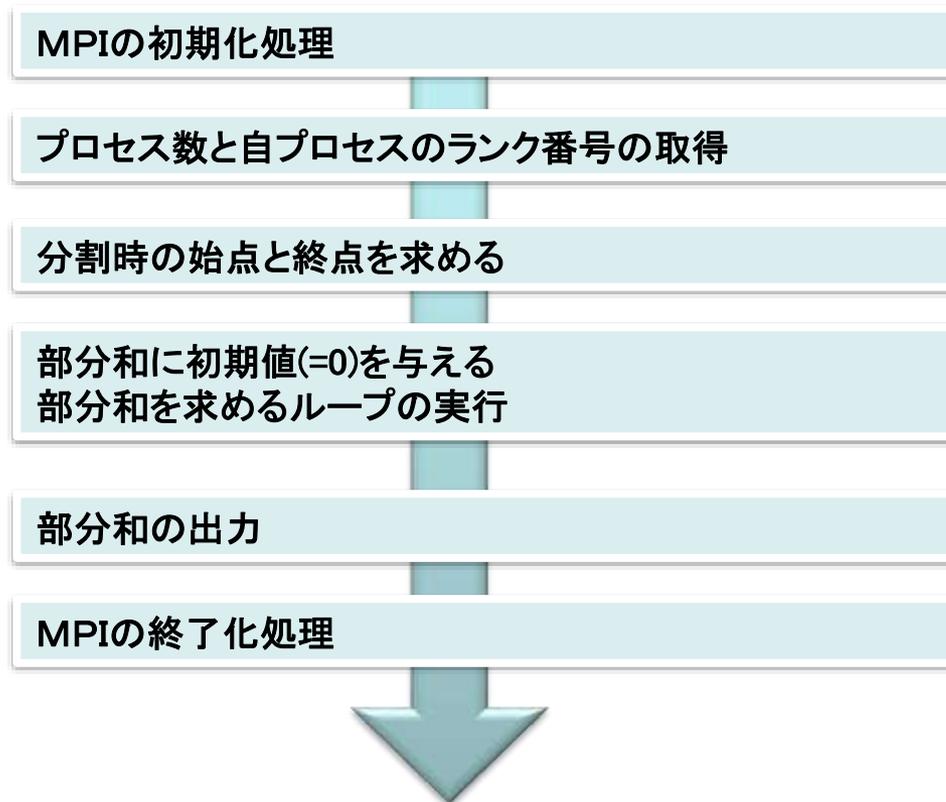
数値例

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

3. 演習問題2 (practice_2)

1から1000の総和を8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



演習問題2 (practice_2) つづき

項目	対象	備考
作業ディレクトリ	practice_2	
使用ソースファイル	practice2.f	編集 必要
ジョブファイル	run.sh	そのまま投入

- 手順①：作業ディレクトリを移動してください。

```
% cd MPI/practice_2
```

- 手順②：エディタでソースファイルを編集してください。ソースファイル practice2.f は ディレクトリ practice_2/ に用意しています。

```
% vi practice2.f
```

※次ページにプログラム編集のヒントがあります。

演習問題2 (practice_2) つづき

ヒント

以下の  の部分を埋めてください。

```
program sample2
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=
ied=
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
```

- 始点の計算式を入れてください
- 終点の計算式を入れてください

演習問題2 (practice_2)つづき

- 手順③：コンパイルを実行します。

```
% mpinfort practice2.f
```

- 手順④：ジョブを投入します。

```
% qsub run.sh
```

- 手順⑤：実行結果を確認します。結果はpractice_p2.oXXXX(XXXXにはリクエストIDが入ります)として格納されます。

```
% cat practice_p2.oXXXX
```

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

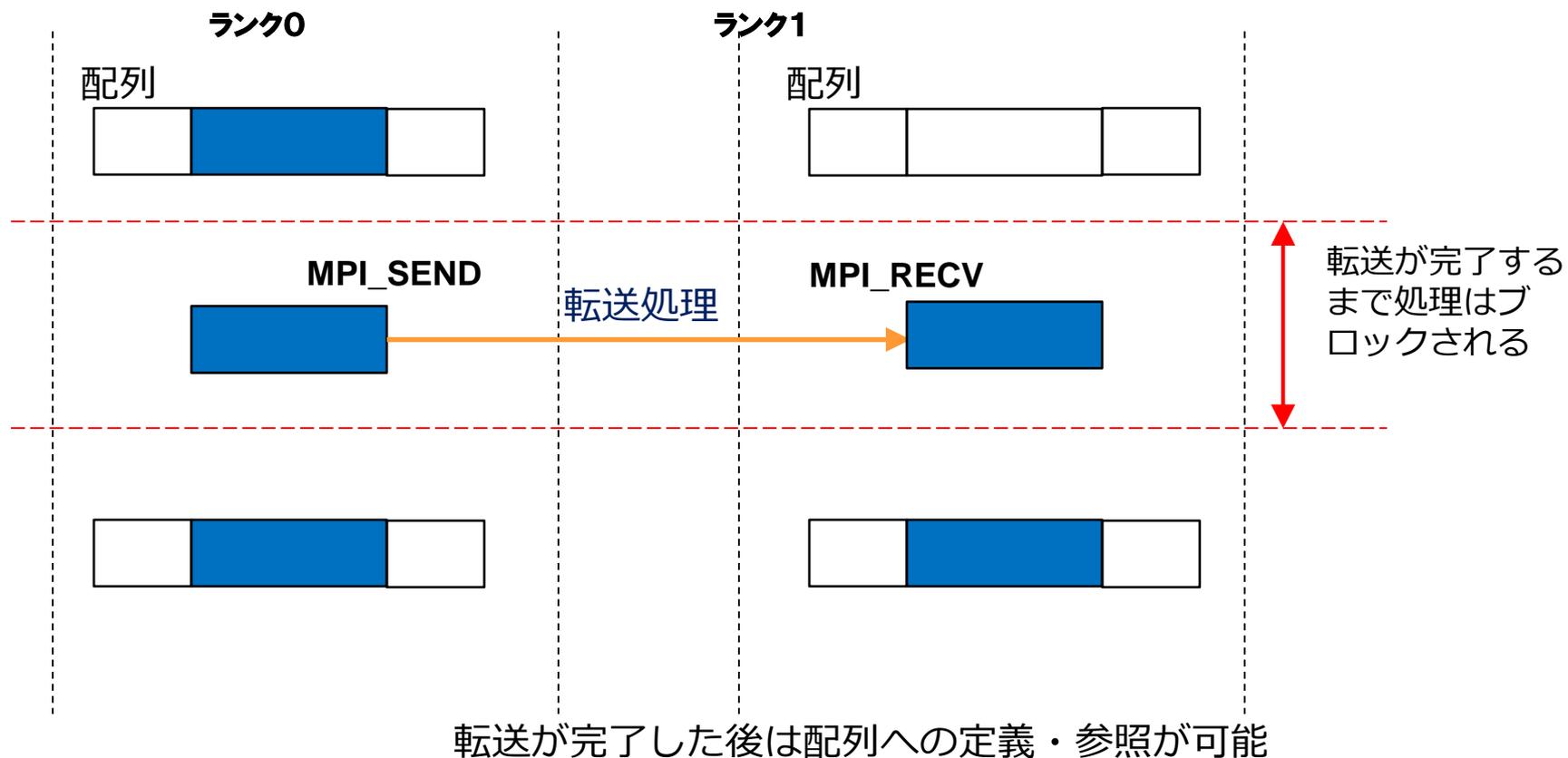
- 各プロセスは独立したメモリ空間を有する
- 他のプロセスのデータを直接アクセスすることは不可
- データ転送により他のプロセスのデータをアクセスすることが可能

■ 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,itag,ierr
integer status(MPI_STATUS_SIZE)
real work(10)
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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

4. 演習問題3 (practice_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

4. 演習問題3 (practice_3) つづき

項目	対象	備考
作業ディレクトリ	practice_3	
使用ソースファイル	practice3.f	編集 必要
ジョブファイル	run.sh	そのまま投入

- 手順①：作業ディレクトリを移動してください。

```
% cd MPI/practice_3
```

- 手順②：エディタでソースファイルを編集してください。演習問題2の回答例としてpractice3.fをディレクトリ practice_3/ に用意しています。ご自身が作成された演習問題2の回答を使用したい場合は、practice_2/からpractice2.fをコピーして使用してください。

```
% vi practice3.f
```

※次ページにプログラム編集のヒントがあります。

4. 演習問題3 (practice_3)つづき

ヒント

以下の の部分を埋めてください。 (P33を参考にしてください)

```
program example3
include 'mpif.h'
integer ierr,myrank,nprocs,ist,ied,itag
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
itag=1
if (  ) then
  call  (  )
& 
else
  call  (  )
& 
  isum=isum+isum2
  call  (  )
& 
  isum=isum+isum2
  call  (  )
& 
  isum=isum+isum2
  write(6,6000) 
6000 format("Total Sum = ",i10)
endif
call MPI_FINALIZE(ierr)
stop
end
```

- ランク0以外であれば、ランク0に送信します。
- ランク0は、ランク1~3からデータを受信します。
- 総和を出力します。

4. 演習問題3 (practice_3)つづき

- 手順③：コンパイルを実行します.

```
% mpinfort practice3.f
```

- 手順④：ジョブを投入します.

```
% qsub run.sh
```

- 手順⑤：実行結果を確認します. 結果はpractice_p3.oXXXX(XXXXにはリクエストIDが入ります)として格納されます.

```
% cat practice_p3.oXXXX
```

MPI集団通信

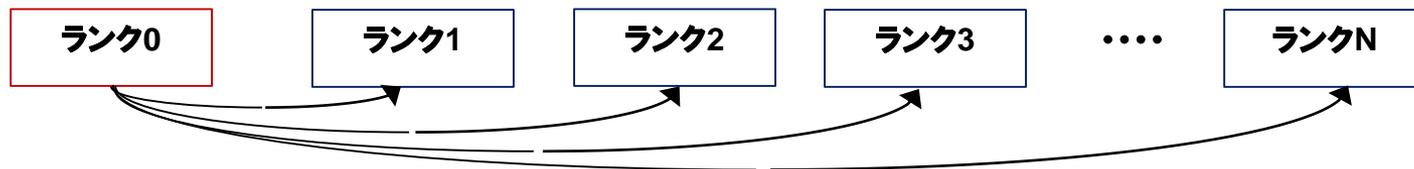
- あるプロセスから同じコミュニケータを持つ全プロセスに対して同時に通信を行う
- または同じコミュニケータを持つプロセス間でデータを共有する

(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,isum,ierr
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_REDUCE(myrank,isum,1,MPI_INTEGER,MPI_SUM,0,
+ MPI_COMM_WORLD,ierr)
if(myrank.eq.0) write(6,*)"Result = ",isum
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

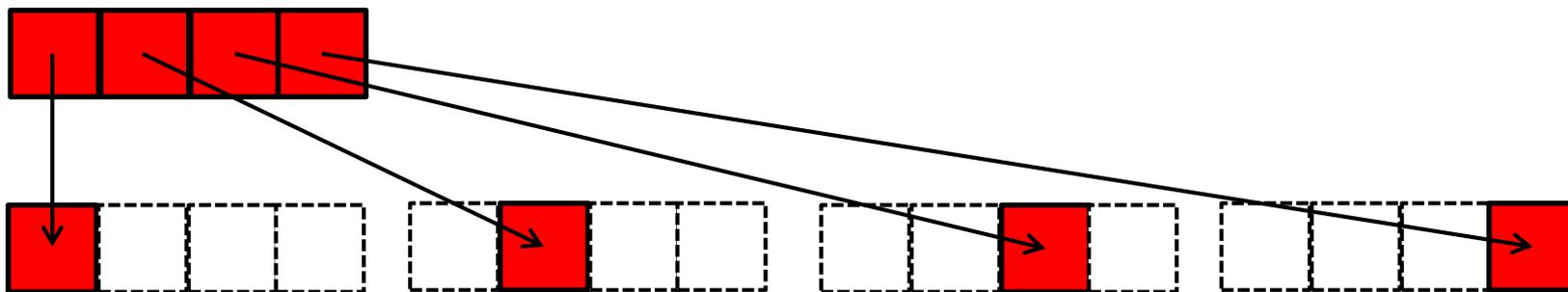
MPI_REDUCEの詳細は付録1.3.3

```
$ mpirun -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。

代表プロセス



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

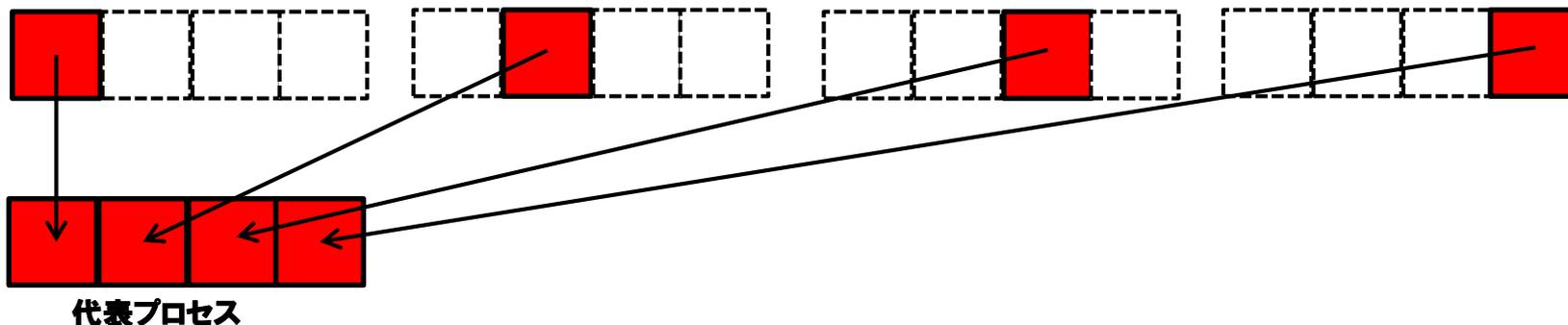
- 送信バッファと受信バッファはメモリ上の重なりがあってはならない(MPI1.0仕様)
- 各プロセスへのメッセージ長が一定でない場合はMPI_SCATTERVを使用する。

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

5. 演習問題4 (practice_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

項目	対象	備考
作業ディレクトリ	practice_4	
使用ソースファイル	practice4.f	編集 必要
ジョブファイル	run.sh	そのまま投入

- 手順①：作業ディレクトリを移動してください。

```
% cd MPI/practice_4
```

- 手順②：エディタでソースファイルを編集してください。演習問題3の回答例としてpractice4.fをディレクトリ practice_4/ に用意しています。ご自身が作成された演習問題3の回答を使用したい場合は、practice_3/からpractice3.fをコピーして使用してください。

```
% vi practice4.f
```

※次ページにプログラム編集のヒントがあります。

5. 演習問題4 (practice_4) つづき

ヒント

以下の の部分を埋めてください。

```
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
call MPI_REDUCE(,,,MPI_INTEGER,,,
& MPI_COMM_WORLD,ierr)
if() write(6,6000) isum2
6000 format("Total Sum = ",i10)
call MPI_FINALIZE(ierr)
stop
end
```

- 送信データのアドレス
- 受信データのアドレス
- 送信データの要素数
- リダクション演算の機能コード
- rootプロセスのランク

- ランク0が集計結果を出力させるようにします。

5. 演習問題4 (practice_4) つづき

- 手順③：コンパイルを実行します。

```
% mpinfort practice4.f
```

- 手順④：ジョブを投入します。

```
% qsub run.sh
```

- 手順⑤：実行結果を確認します。結果はpractice_p4.oXXXX(XXXXにはリクエストIDが入ります)として格納されます。

```
% cat practice_p4.oXXXX
```

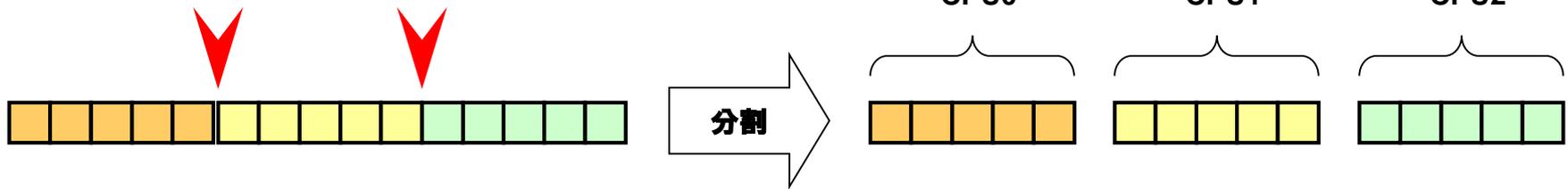
6. MPIプログラミング

- 通信の発生
- デッドロック
- 配列の縮小
- ファイルの入出力
- MPI + OpenMPのハイブリッド実行

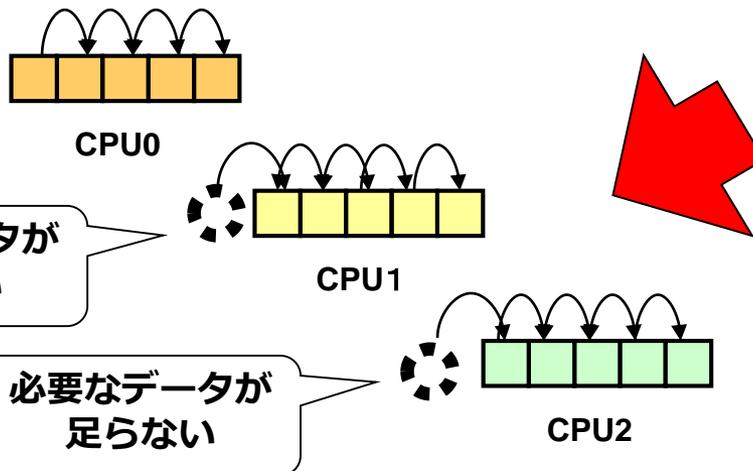
通信の発生

■ 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

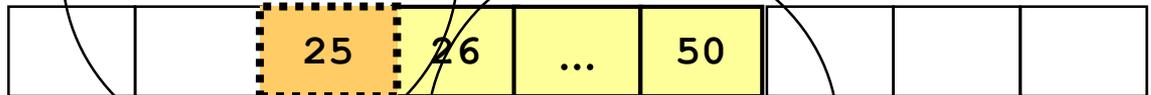
```
do i=1, 100  
  b(i)=a(i)-a(i-1)  
enddo
```

並列版

プロセス0



プロセス1

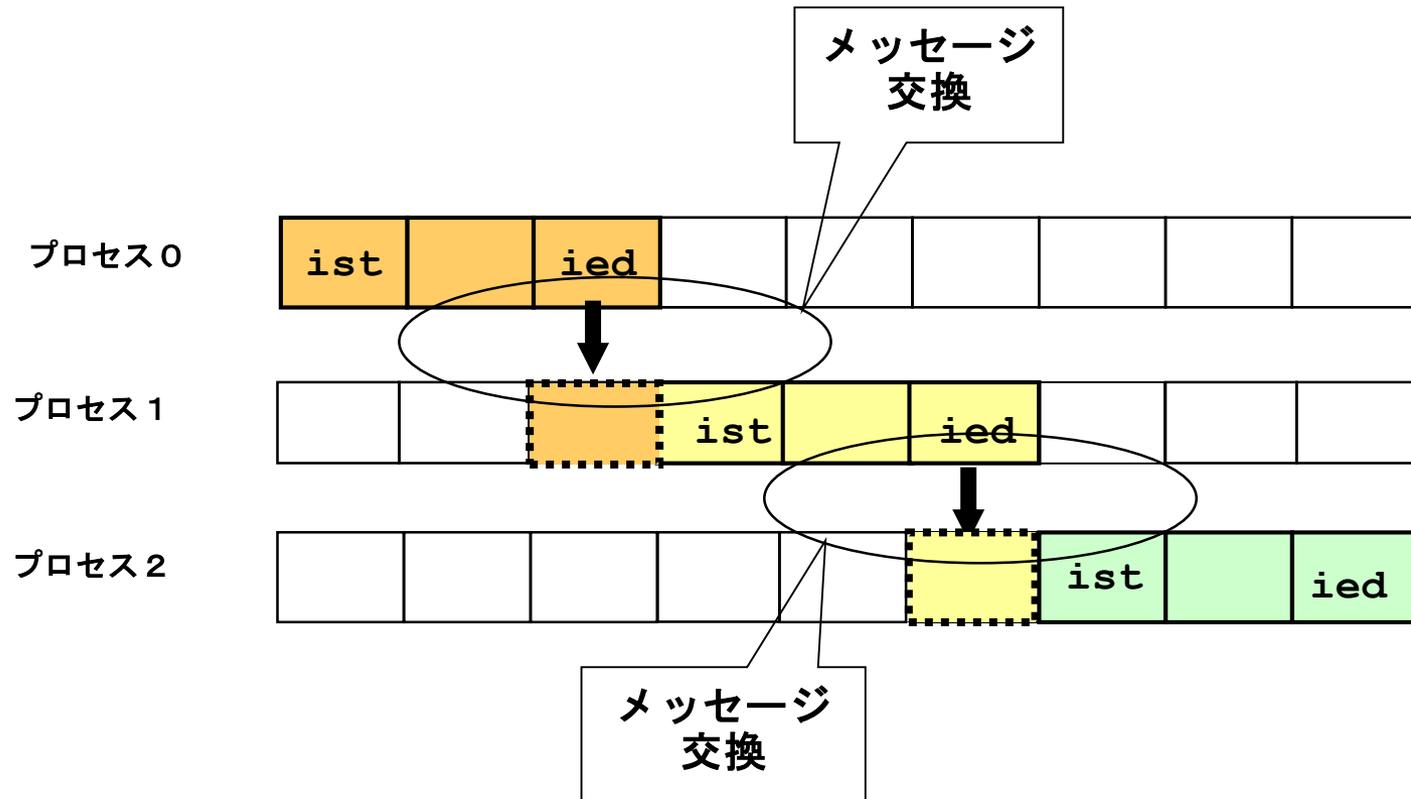


プロセス2

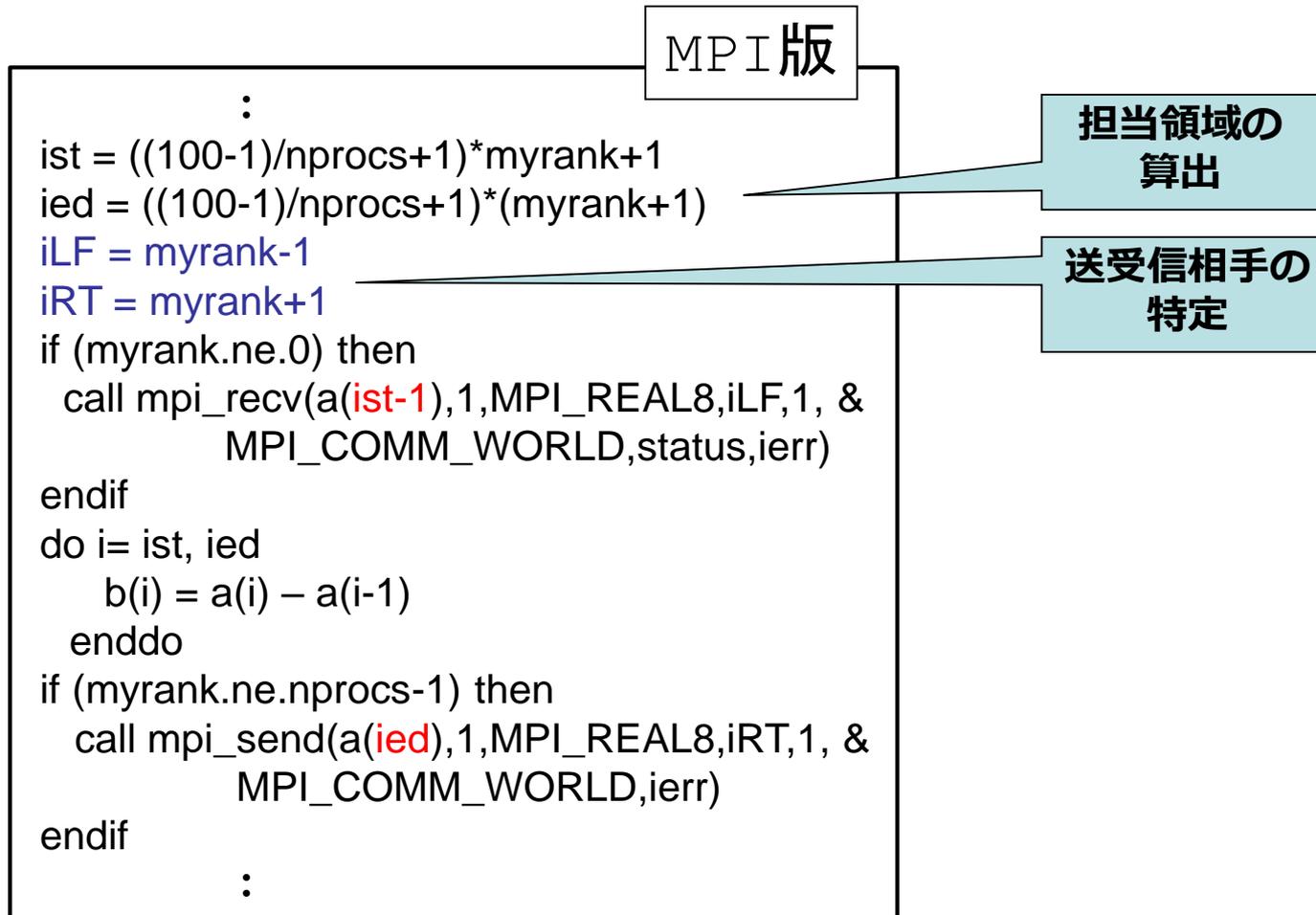


不足データの送受信

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

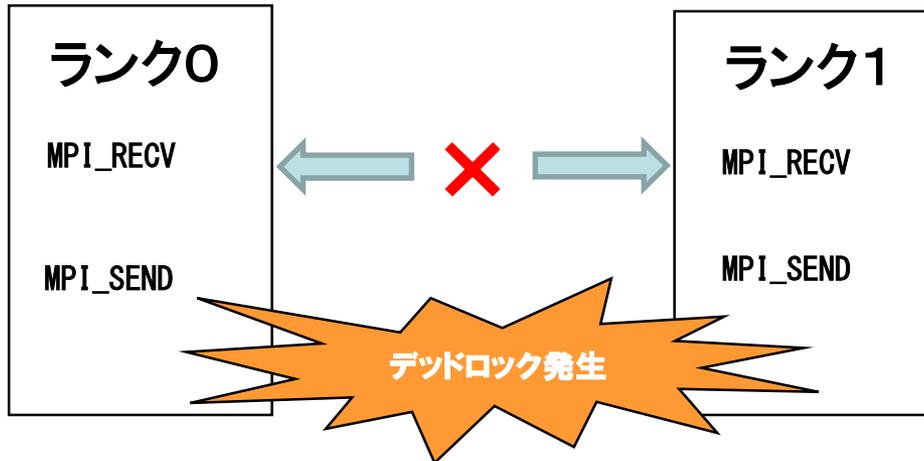


デッドロック

```
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
```

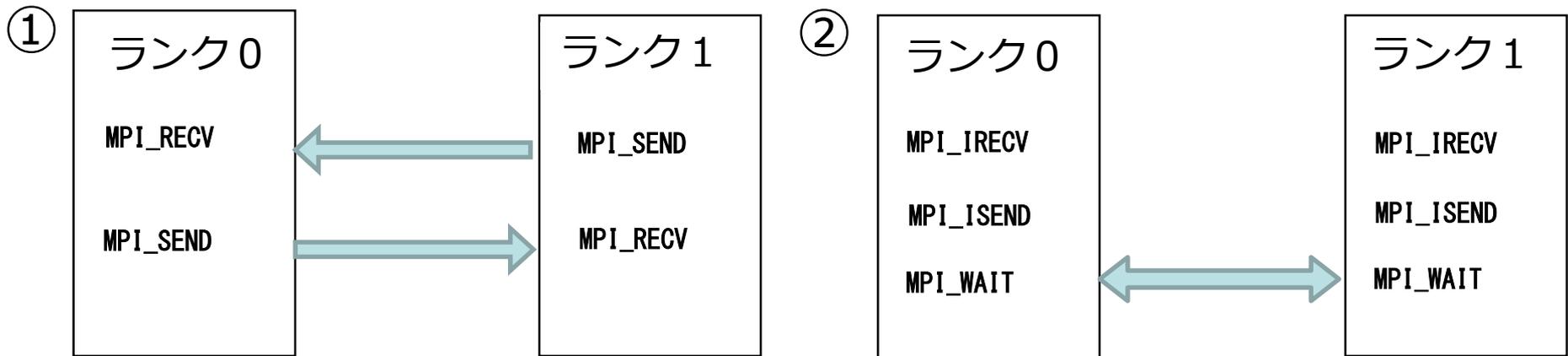
- ランク0とランク1が同時にMPI_RECV(同期型1対1通信)を行うと、送受信が完了せず、待ち状態となる。
- このような待ち状態をデッドロックという。

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

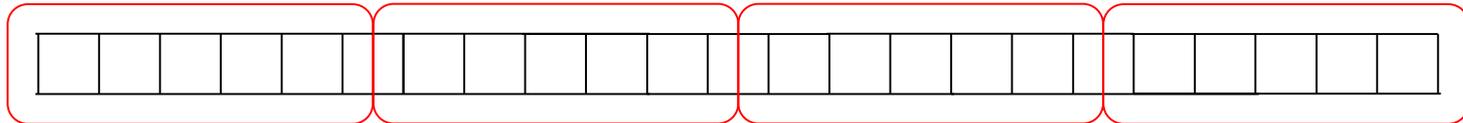
MPI_ISENDの詳細は付録1.2.7

MPI_IRecvの詳細は付録1.2.8

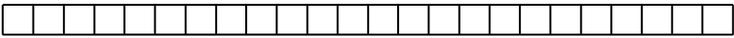
MPI_WAITの詳細は付録1.2.10

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

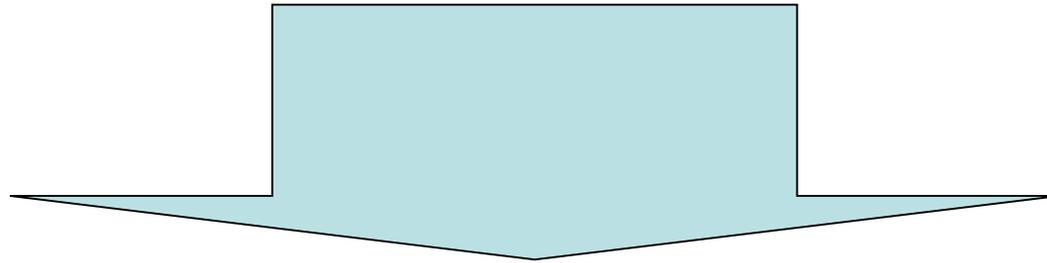
```
real(8)::a(100)
do i=26,50
  c=c+a(i)*b(i)
enddo
```

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

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幾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。

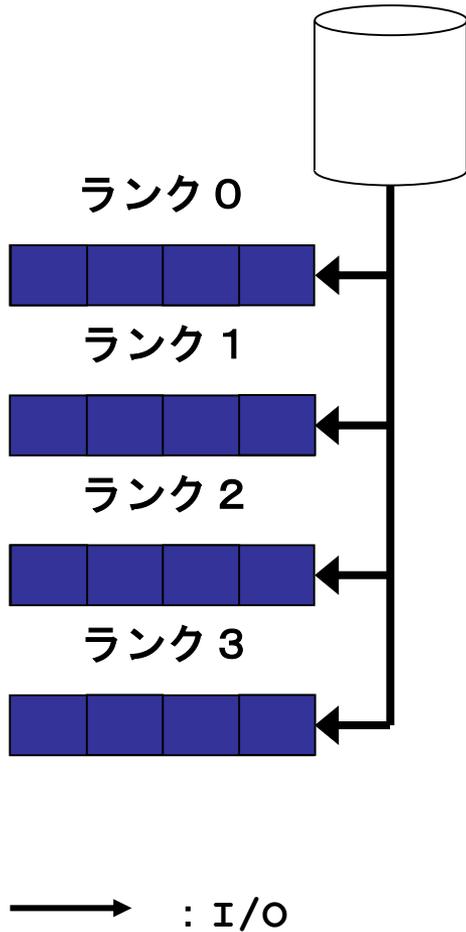
1. ファイル入力

- 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

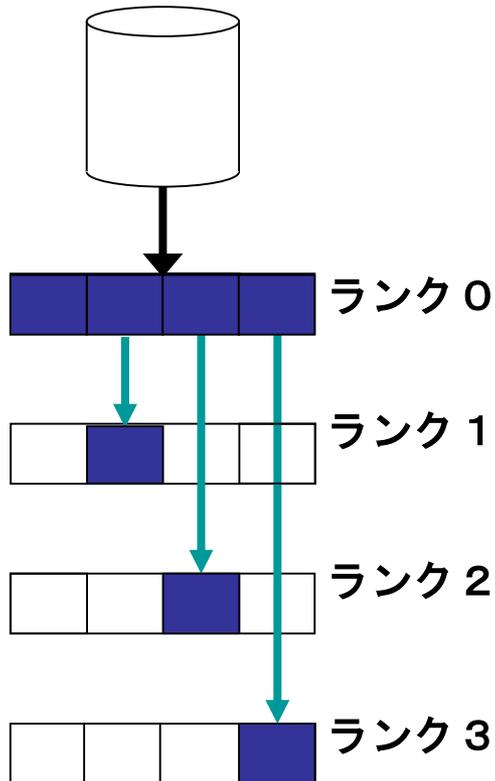


etc1.f

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

代表プロセス入力

etc2.f



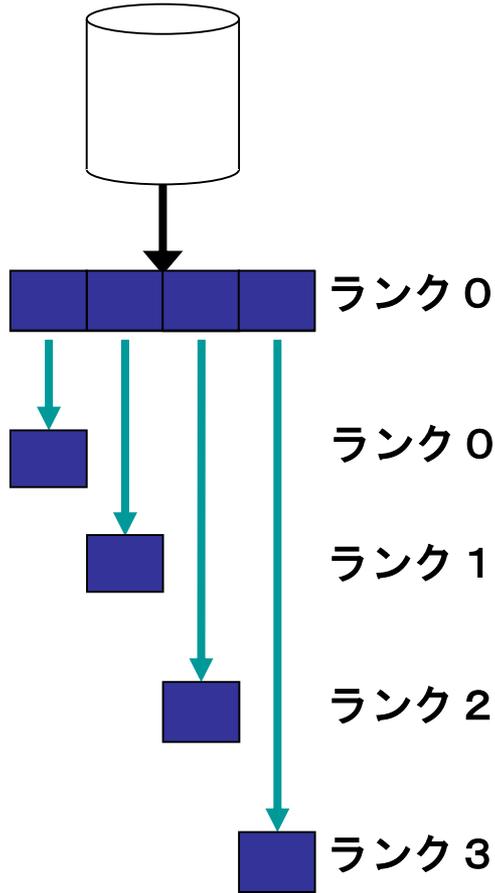
→ : MPI通信

→ : I/O

```
include 'mpif.h'  
integer,parameter :: numdat=100  
integer::senddata(recvdata(numdat)  
integer::myrank,nprocs,icount,isum,ierr  
call MPI_INIT(ierr)  
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)  
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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



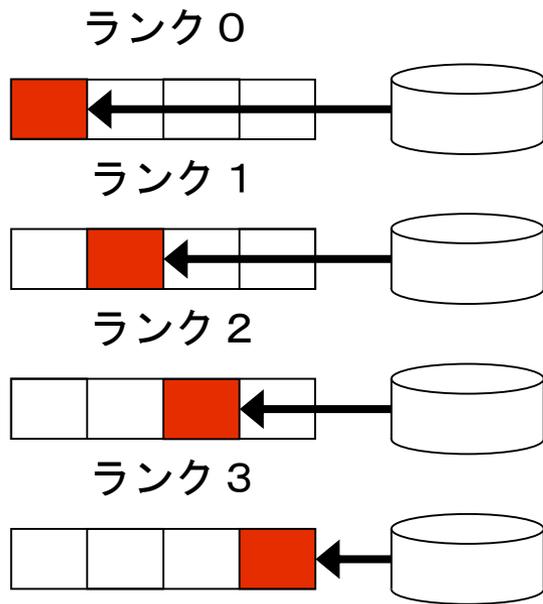
```
include 'mpif.h'
integer,parameter :: numdat=100
integer,allocatable :: idat(:),work(:)
integer :: nprocs,myrank,ierr
integer :: ist,ied,isum
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
ist = ((numdat-1)/nprocs+1)*myrank+1
ied = ((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
deallocate(idat)
call MPI_FINALIZE(ierr)
stop
end
```

→ : MPI通信

→ : I/O

分散ファイル入力

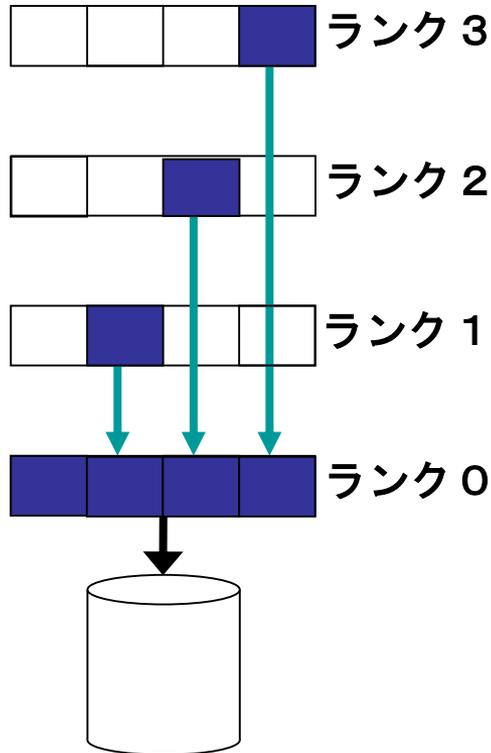
etc3.f



→ : I/O

```
include 'mpif.h'  
integer,parameter :: numdat=100  
integer::buf(numdat)  
integer :: myrank,nprocs,ist,ied,isum,ierr  
c  
call MPI_INIT(ierr)  
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)  
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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



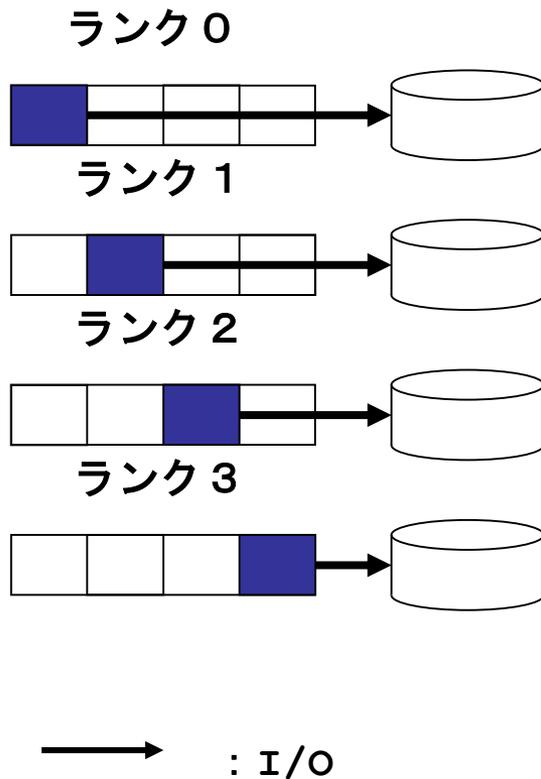
→ : MPI通信
→ : I/O

etc4.f

```
include 'mpif.h'  
parameter (numdat=100)  
integer senddata(numdat),recvdata(numdat)  
integer myrank,nprocs,icount,ierr  
call MPI_INIT(ierr)  
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)  
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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



etc5.f

```
include 'mpif.h'  
integer,parameter :: numdat=100  
integer :: buf(numdat)  
integer :: myrank,nprocs,ist,ied,ierr  
call MPI_INIT(ierr)  
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)  
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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
```

7. 演習問題 5

- P60のetc4.fをP58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。

7. 演習問題5 (practice_5) つづき

項目	対象	備考
作業ディレクトリ	practice_5	
使用ソースファイル	practice5.f	編集 必要
ジョブファイル	run.sh	そのまま投入

- 手順①：作業ディレクトリを移動してください。

```
% cd MPI/practice_5
```

- 手順②：エディタでソースファイルを編集してください。ソースファイルpractice5.fは practice_5/ に用意しています。

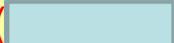
```
% vi practice5.f
```

※次ページにプログラム編集のヒントがあります。

7. 演習問題5 (practice_5) つづき

ヒント

以下の  の部分を埋めてください。

```
include 'mpif.h'
integer,parameter :: numdat=100
integer,allocatable :: senddata(:), 
integer :: myrank,nprocs,icount,ierr
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((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
```

➤ 必要な分だけallocate
します。

次ページへつづく。

7. 演習問題5 (practice_5) つづき

```
call MPI_GATHER( [redacted]  
& [redacted],MPI_INTEGER,[redacted]  
& [redacted],MPI_INTEGER,0,MPI_COMM_WORLD,  
& ierr)  
if(myrank.eq.0) then  
  open(60,file='fort.60')  
  write(60,'(10I8)' recvdata  
  deallocate([redacted])  
endif  
deallocate([redacted])  
call MPI_FINALIZE(ierr)  
stop  
end
```

- 送信データの開始アドレス
- 送信データの要素数
- 受信領域の開始アドレス
- 個々のプロセスから受信する要素数
- 最後にdeallocateします。

7. 演習問題5 (practice_5) つづき

- 手順③：コンパイルを実行します.

```
% mpinfort practice5.f
```

- 手順④：ジョブを投入します.

```
% qsub run.sh
```

- 手順⑤：実行結果を確認します. 結果はpractice_p5.oXXXX(XXXXにはリクエストIDが入ります)として格納されます.

```
% cat practice_p5.oXXXX
```

MPI + OpenMPのハイブリッド実行 (1/2)

- MPI(Message Passing Interface)とOpenMPを組み合わせることにより, 複数ノード(複数VE)のSX-Aurora TSUBASAが利用可能
- VE間をMPI並列(最大256VE), VE内をOpenMP並列(最大8スレッド)
- ハイブリッド実行のメリットは以下
 - ① 並列数の増加による処理時間の短縮
 - ② MPI転送ネックのプログラムにおいて, 転送回数の削減(コードによっては転送量の削減も)

MPI + OpenMPのハイブリッド実行 (2/2)

■ コンパイルはmpinfortコマンドを利用

```
mpinfort -fopenmp [オプション] ソースファイル名
```

■ ハイブリッド実行のスク립トの例

```
#!/bin/csh
#PBS -q ①
#PBS --venode ②
#PBS -l elapstim_req=③
#PBS -v OMP_NUM_THREADS=④

cd $PBS_O_WORKDIR
mpirun -np ⑤ ./a.out
```

- ① キュー名を指定 (必須)
- ② 使用VE数を指定. (必須)
- ③ 使用計算時間 (経過時間) を設定.
「hh:mm:ss」のように指定. (強く推奨)
- ④ スレッド数を指定. (共有並列の場合必須)
- ⑤ 総MPIプロセス数を指定.
(VE内がすべて共有メモリ並列の場合は,
②=⑤)

※詳細は「AOBA-A プログラム開発・実行環境 利用手順書」を参照

<https://www.ss.cc.tohoku.ac.jp/sscc/wp-content/uploads/pdf/AOBA-Aプログラム開発・実行環境利用手順書.pdf>

8. 演習問題6 (practice_6)

行列積プログラムをMPIで並列化してください

```
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
```

- ◆ 左記に記載する行列積を行うプログラムをMPI化して8プロセスで実行してください。出力はプロセス0で行ってください。

8. 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]

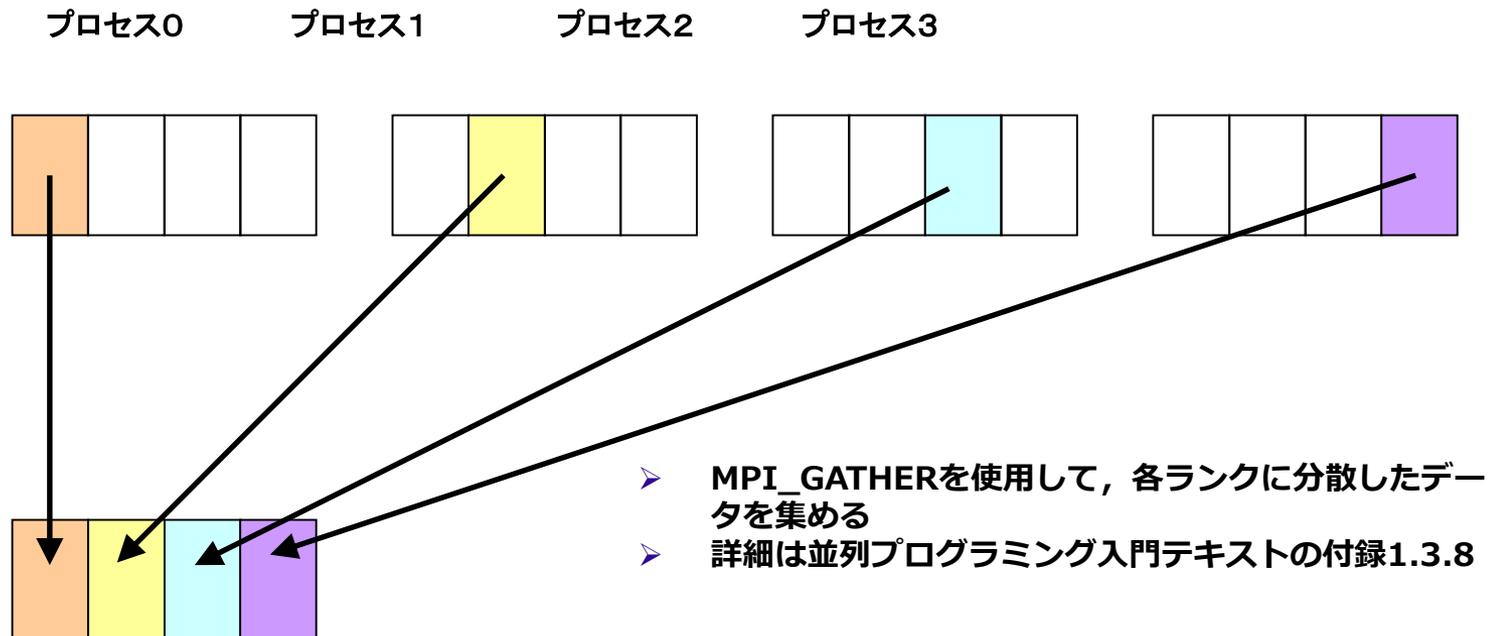
```
call MPI_BARRIER(MPI_COMM_WORLD,IERR)  
t2=MPI_WTIME()
```

③t2-t1が計測区間の時間となる

8. 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)



8. 演習問題6 (practice_6) つづき

項目	対象	備考
作業ディレクトリ	practice_6	
使用ソースファイル	practice6.f	編集 必要
ジョブファイル	run.sh	そのまま投入

- 手順①：作業ディレクトリを移動してください。

```
% cd MPI/practice_6
```

- 手順②：エディタでソースファイルを編集してください。逐次版の行列積プログラムのソースファイルpractice6.fは practice_6/ に用意しています。

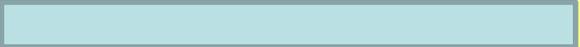
```
% vi practice6.f
```

※次ページにプログラム編集のヒントがあります。

8. 演習問題6 (practice_6) つづき

ヒント

以下の  の部分を埋めてください。

```
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=
ied=
n2=n/nprocs
```

▶ 分割点の始点と終点を
決めます。

次ページへつづく。

8. 演習問題6 (practice_6) 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)
```

t1=

➤ バリア処理

➤ 時間計測区間の開始

次ページへつづく.

8. 演習問題6 (practice_6) つづき

```
do j= [ ]
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), [ ], MPI_REAL8, d, [ ]
& [ ], MPI_REAL8, 0, MPI_COMM_WORLD, ierr)
[ ]
t2= [ ]
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
```

- 各プロセスが担当する区間の計算を実施.
- 計算結果をランク0に集めます.
- 各プロセスから転送する配列のサイズ
- バリア処理
- 時間計測区間の終了.

8. 演習問題6 (practice_6) つづき

- 手順③：コンパイルを実行します。

```
% mpinfort practice6.f
```

- 手順④：ジョブを投入します。

```
% qsub run.sh
```

- 手順⑤：実行結果を確認します。結果はpractice_p6.oXXXX(XXXXにはジョブIDが入ります)として格納されます。

```
% cat practice_p6.oXXXX
```

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
- 付録 1.5 プログラミング作法

※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'
integer :: myrank,nprocs,ist,ied,ierr,isum1
parameter(numdat=100)
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myrank, 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）

[P9に戻る](#)

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レーチンも呼び出してはならない

[P9に戻る](#)

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

[P10に戻る](#)

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)

P10に戻る

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

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

$$\begin{aligned} \text{ist} &= ((100-1)/4+1)*0+1 \\ &= 1 \\ \text{ied} &= ((100-1)/4+1)*(0+1) \\ &= 25 \end{aligned}$$

$$\begin{aligned} \text{ist} &= ((100-1)/4+1)*1+1 \\ &= 26 \\ \text{ied} &= ((100-1)/4+1)*(1+1) \\ &= 50 \end{aligned}$$

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

$$\begin{aligned} \text{ist} &= ((100-1)/4+1)*3+1 \\ &= 76 \\ \text{ied} &= ((100-1)/4+1)*(3+1) \\ &= 100 \end{aligned}$$

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

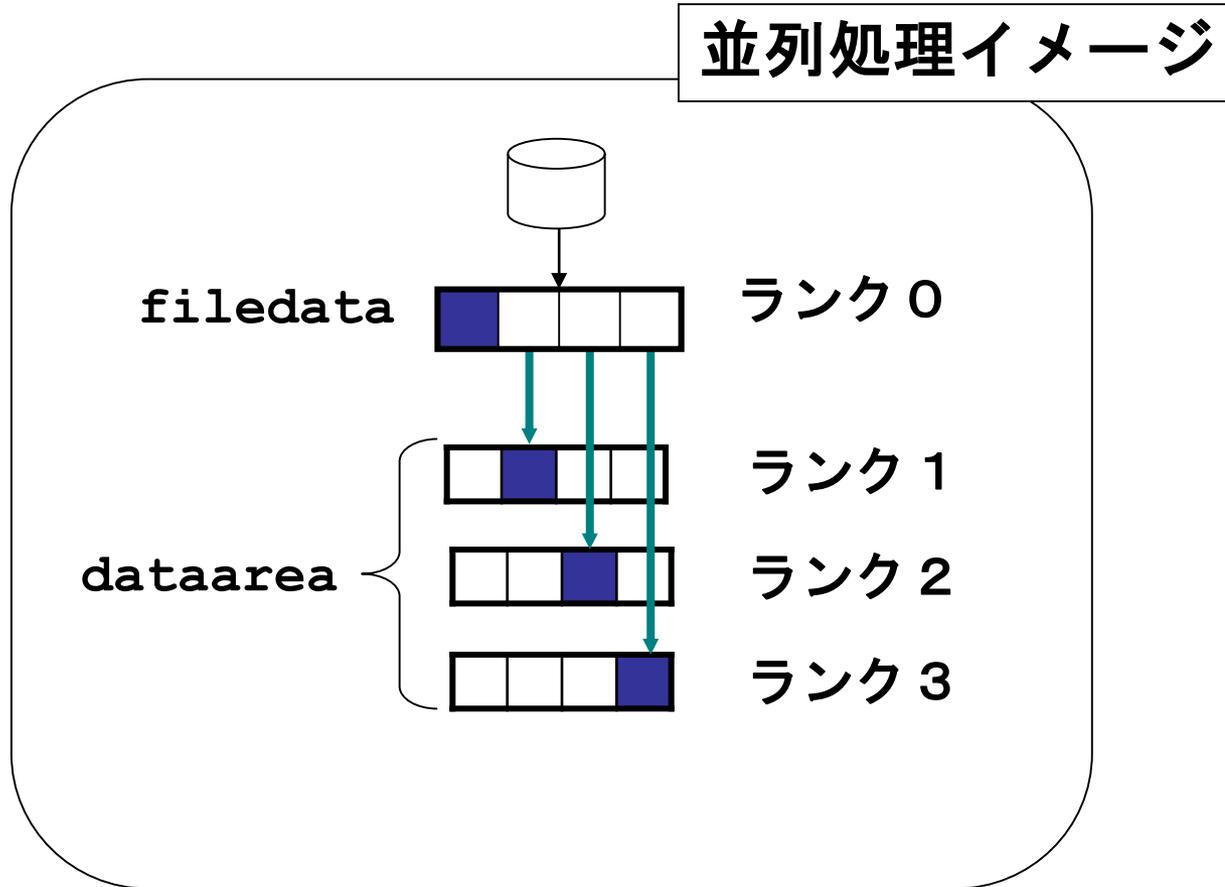
■ 1から100までをnprocで分割

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
```

付録 1.2.2 処理イメージ

並列処理イメージ



付録 1.2.3 プログラム例 (MPI版)

etc9.f

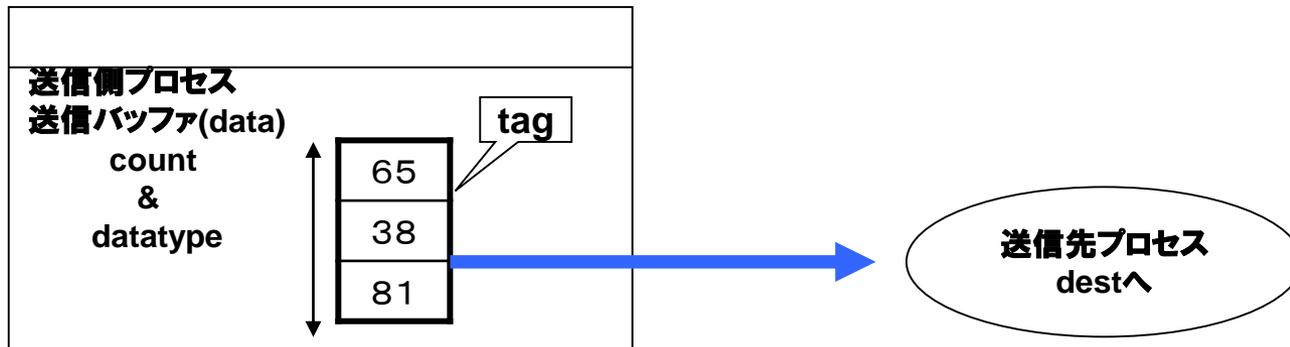
```
include 'mpif.h'
parameter(numdat=100)
integer status(MPI_STATUS_SIZE),senddata(numdat),recvdata(numdat)
integer source,dest,tag
integer myrank,nprocs,icount,ierr,isum
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
icount=(numdat-1)/nprocs+1
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 1.2.4 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



付録 1.2.4 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

付録 1.2.4 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のどちらで受信してもよい

付録 1.2.5 MPIで定義された変数の型 (FORTRAN)

MPIの データタイプ	FORTRAN言語の 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

付録 1.2.6 MPIで定義された変数の型 (C)

MPI データタイプ	C言語 対応する型	
MPI_CHAR	char	
MPI_SHORT	short	
MPI_INT	int	
MPI_LONG	long	
MPI_LONG_LONG	long long	
MPI_LONG_LONG_INT	long long	
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED_INT	unsigned int	
MPI_UNSIGNED_LONG	unsigned long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	など

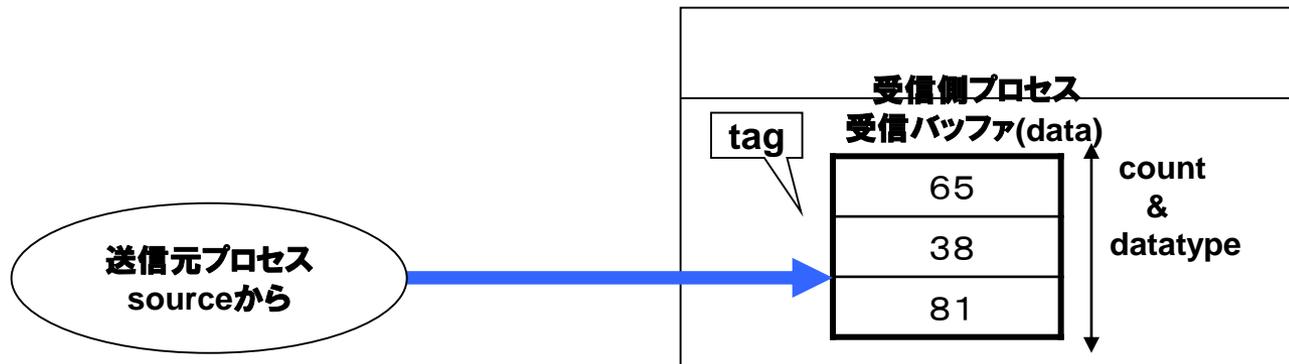
[P32に戻る](#)

付録 1.2.7 MPI_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



付録 1.2.7 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

付録 1.2.7 MPI_RECV ブロッキング型受信 (続き)

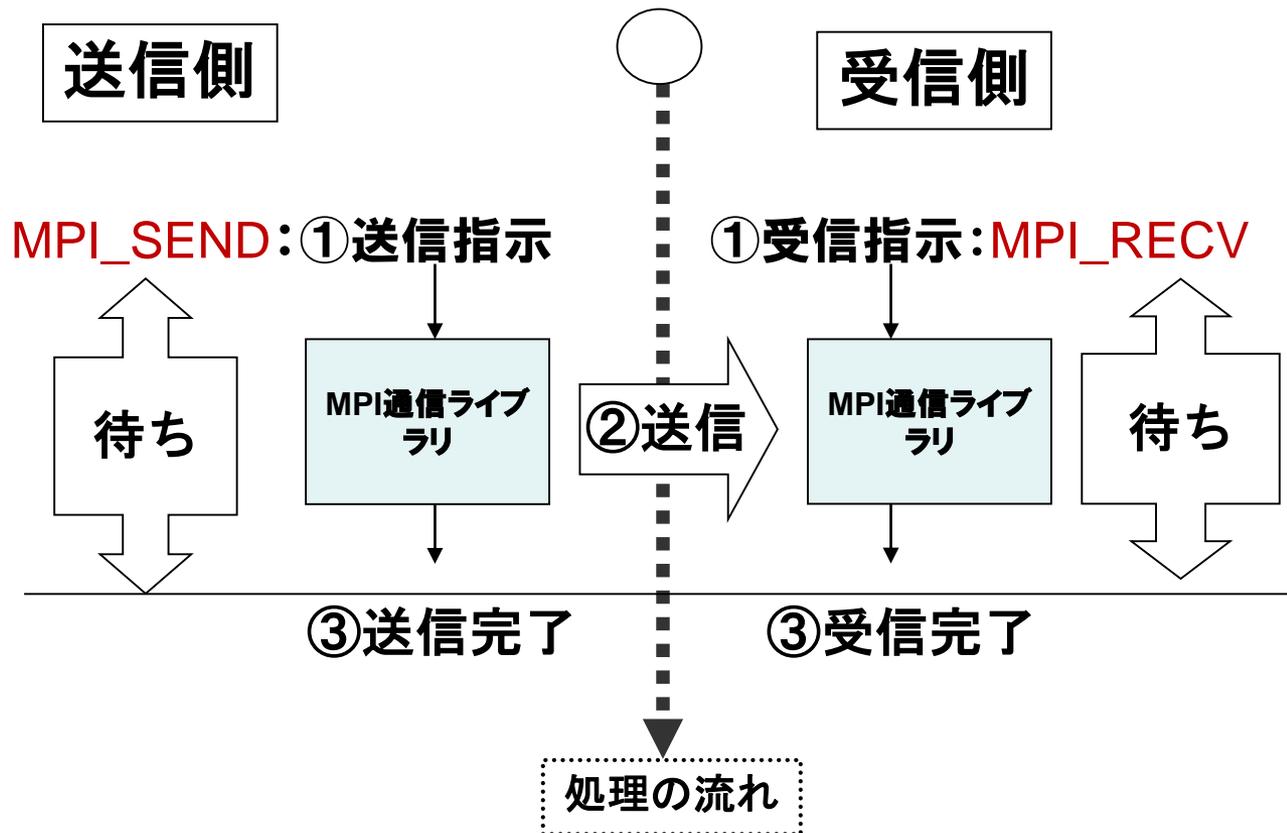
メモ

- 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

[P32に戻る](#)

付録 1.2.8 ブロッキング型通信の動作

■ MPI_SEND, MPI_RECV

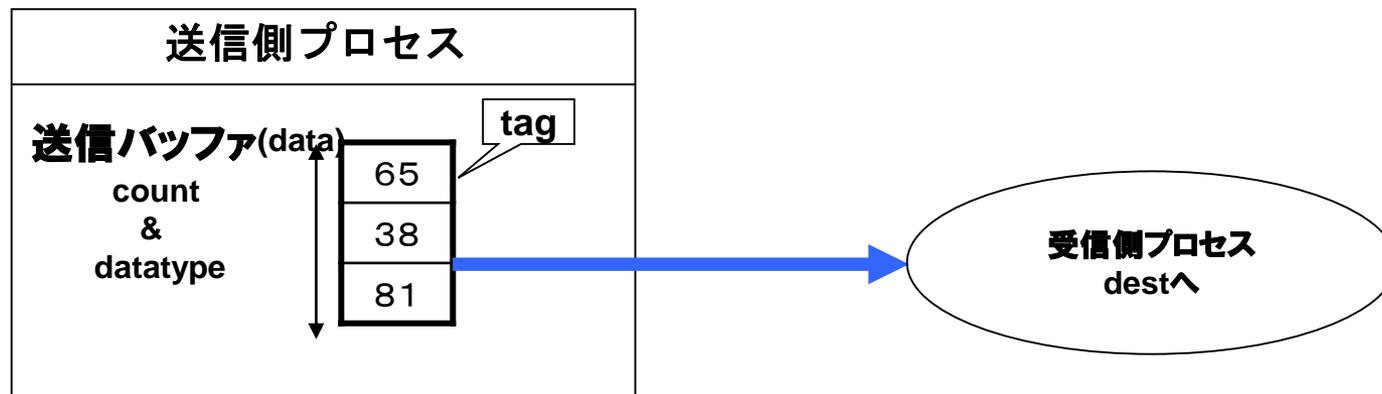


付録 1.2.9 MPI_ISEND 非ブロッキング型送信

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



付録 1.2.9 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

付録 1.2.9 MPI_ISEND 非ブロッキング型送信 (続き)

メモ

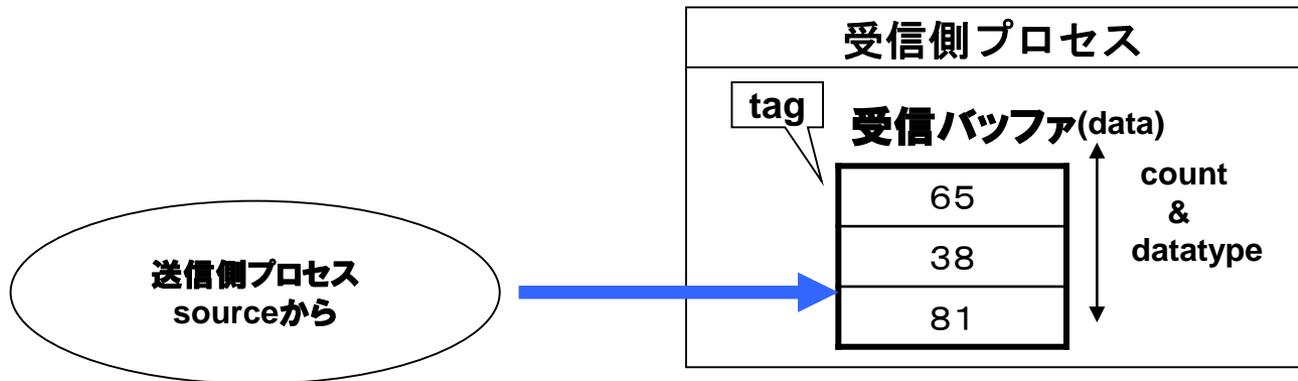
- メッセージの大きさはバイト数ではなく、要素の個数(count)で表す
- datatypeはMPI_SENDの項を参照
- タグはメッセージを区別するために使用する
- requestには要求した通信の識別子が戻され、MPI_WAIT等で通信の完了を確認する際に使用する
- 本ルーチンコール後、受信処理の完了を待たずにプログラムの処理を続行する
- MPI_WAITまたはMPI_WAITALLで処理の完了を確認するまでは、dataの内容を更新してはならない
- MPI_ISENDで送信したデータは、MPI_IRECV, MPI_RECVのどちらで受信してもよい
- 通信の完了もMPI_WAIT,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

付録 1.2.10 MPI_Irecv 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



付録 1.2.10 MPI_Irecv 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_Irecv(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

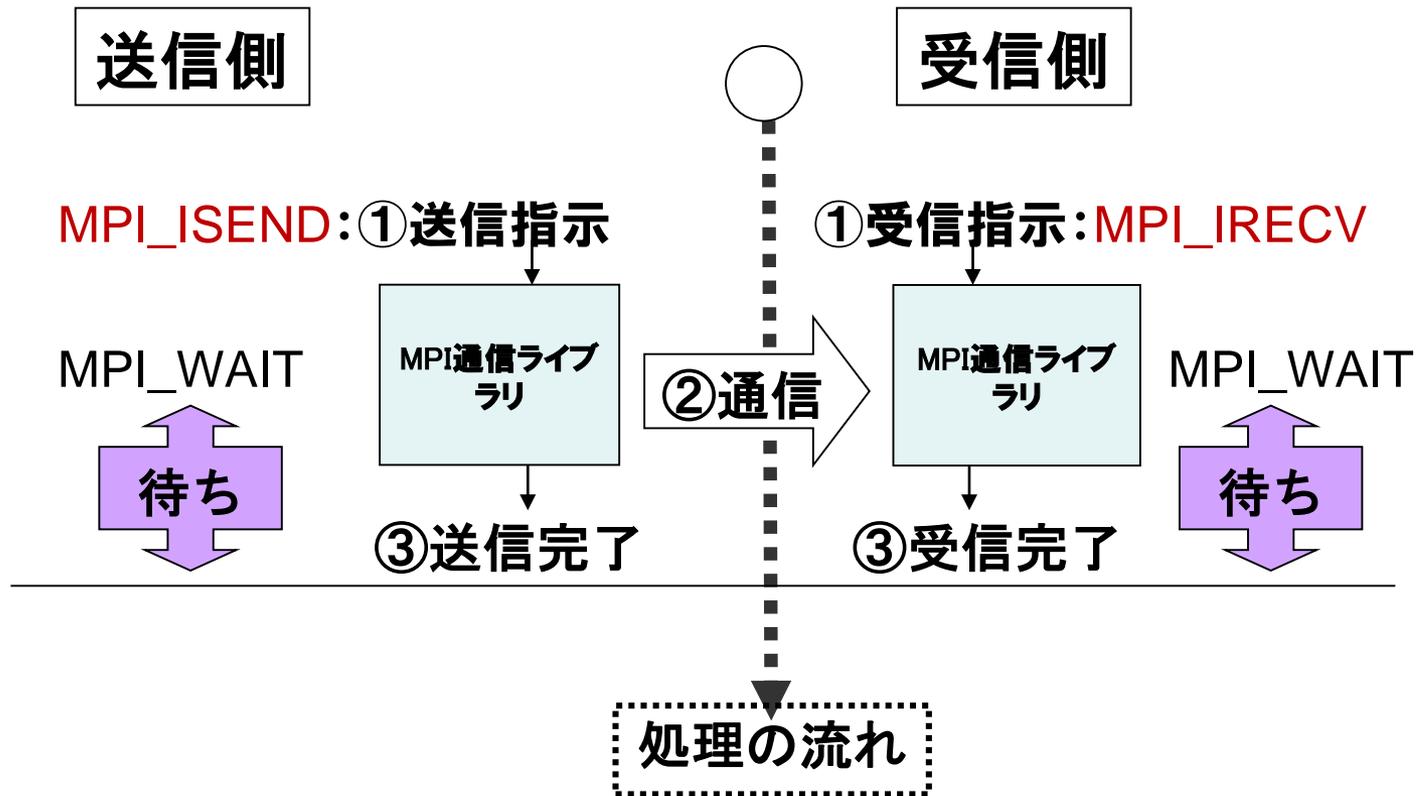
付録 1.2.10 MPI_Irecv 非ブロッキング型受信 (続き)

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_Irecv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 1.2.11 非ブロッキング型通信の動作

■ MPI_ISEND, MPI_IRECVの動作



付録 1.2.12 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 1.2.13 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 大きさは(count)
array_of_status	status	OUT	メッセージ情報の配列 大きさは(count)

付録 1.2.13 MPI_WAITALL 通信完了の待合わせ (続き)

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 1.2.14 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てもMPI_WAITALLでまとめて待ち合わせても良い

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

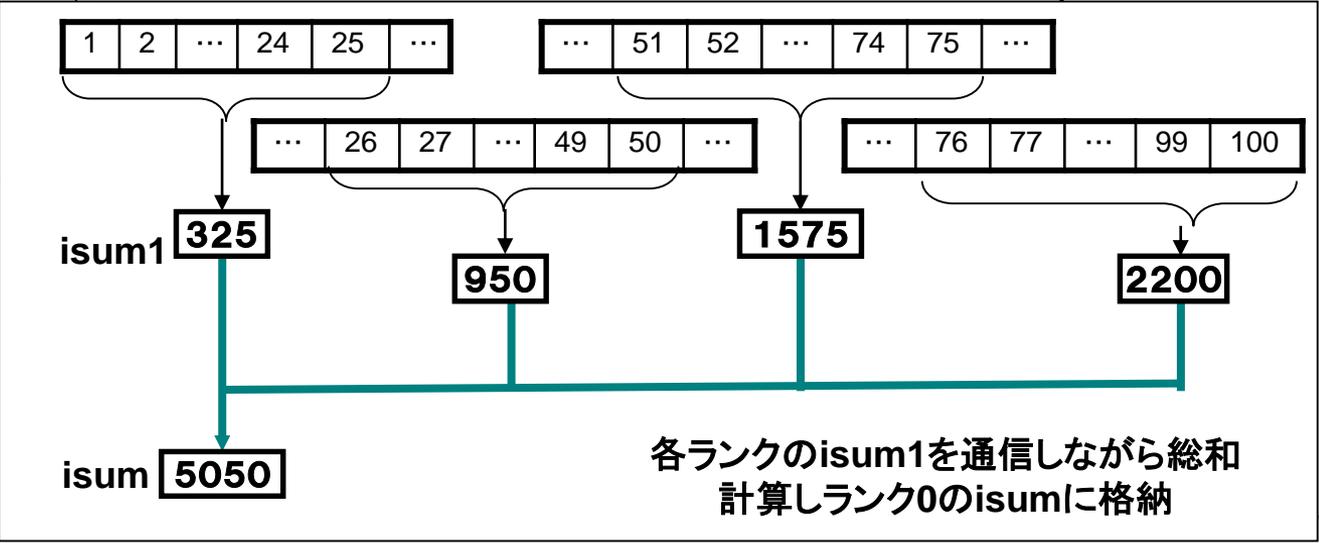
付録 1.3.2 プログラム例

etc10.f

```
include 'mpif.h'
parameter(numdat=100)
integer :: myrank,nprocs,ist,ied,ierr,isum1,isum
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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

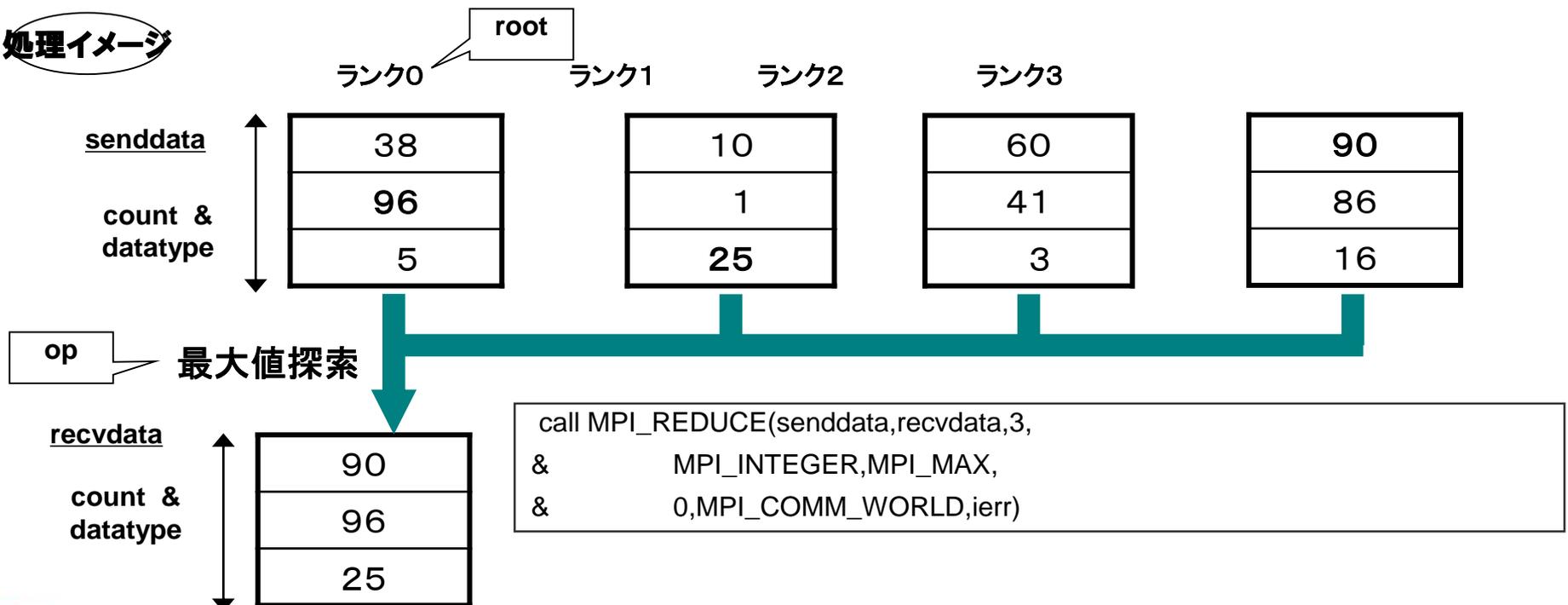


付録 1.3.3 MPI_REDUCE リダクション演算

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



付録 1.3.3 MPI_REDUCE リダクション演算 (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 (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

付録 1.3.3 MPI_REDUCE リダクション演算 (続き)

■ 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

付録 1.3.4 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

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

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

$$\text{dsum2} = a(4) + a(5) = 8 + 6 = 14 = 0.0001E5$$

$$\text{dsum2} + a(6) = 0.0001E5 + 1E5 = 1.0001E+5$$

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

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

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

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

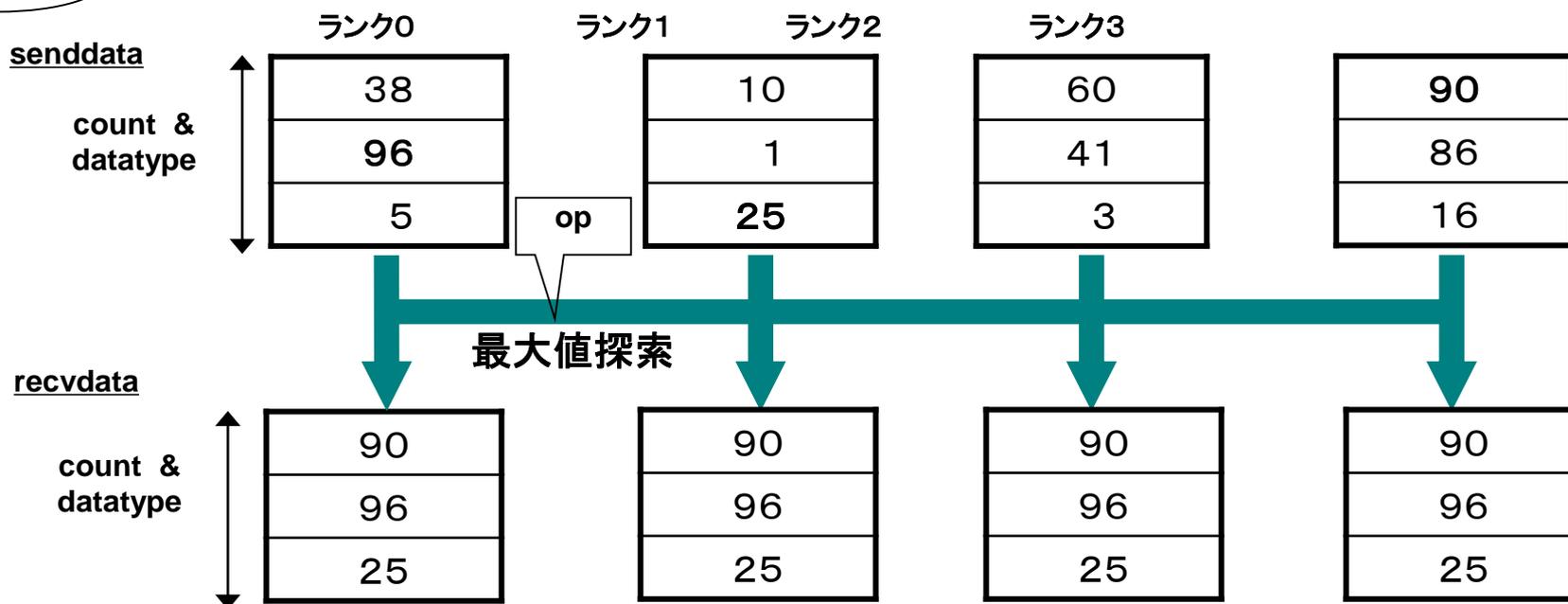
- 通信に参加する全プロセスが、同じ集団通信手続きをコールしなければならない
- 送信バッファと受信バッファの実際に使用する部分は、メモリ上で重なってはならない
(MPI-2では、MPI_IN_PLACEを用いることで可能になります)
- 基本的に集団通信処理の直前や直後での同期処理は不要

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)
```

付録 1.3.5 MPI_ALLREDUCE リダクション演算 (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

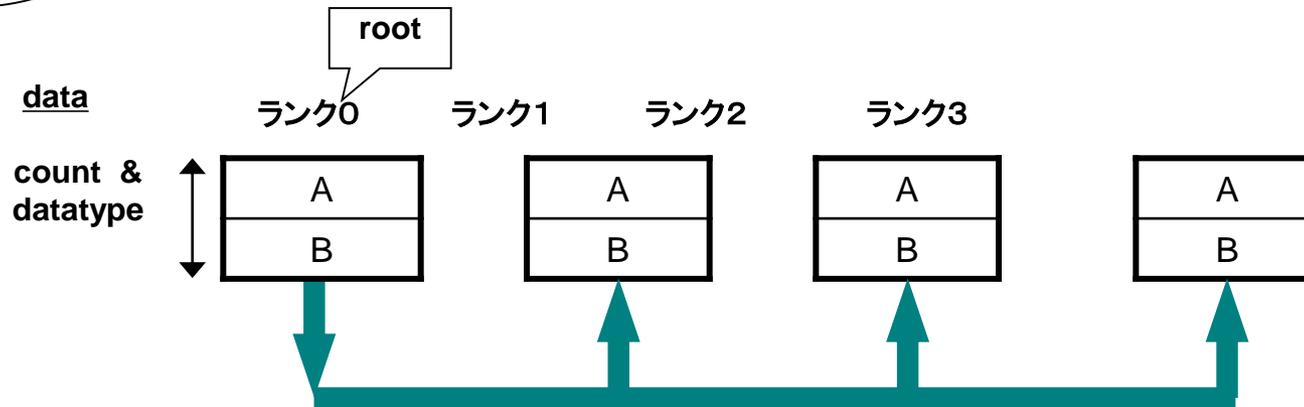
- MPI_REDUCEの計算結果を全プロセスに送信するのと機能的に同じ

付録 1.3.6 MPI_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



付録 1.3.6 MPI_BCAST ブロードキャスト (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

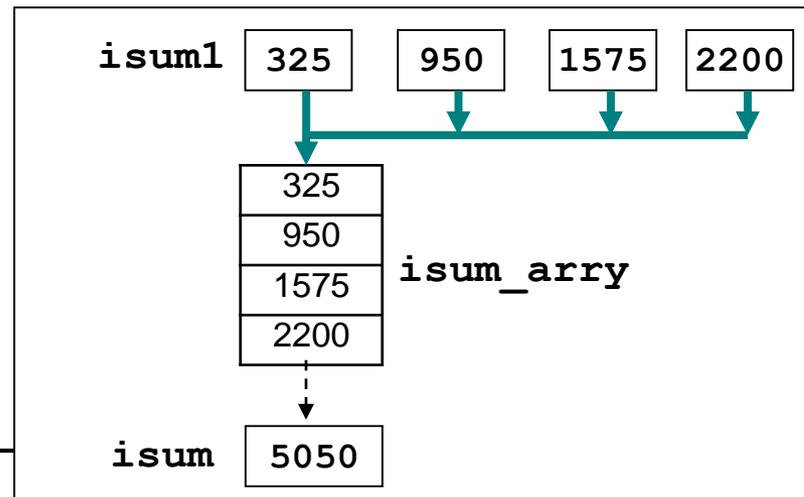
メモ

- 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_arr(10)
integer myrank,nprocs,ist,ied,ierr,isum,isum1
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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_arr, 1,
& MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
if(myrank.eq.0) then
  isum=0
  do i=1,nprocs
    isum=isum+isum_arr(i)
  enddo
  write(6,*)'isum=',isum
endif
call MPI_FINALIZE(ierr)
stop
end
```

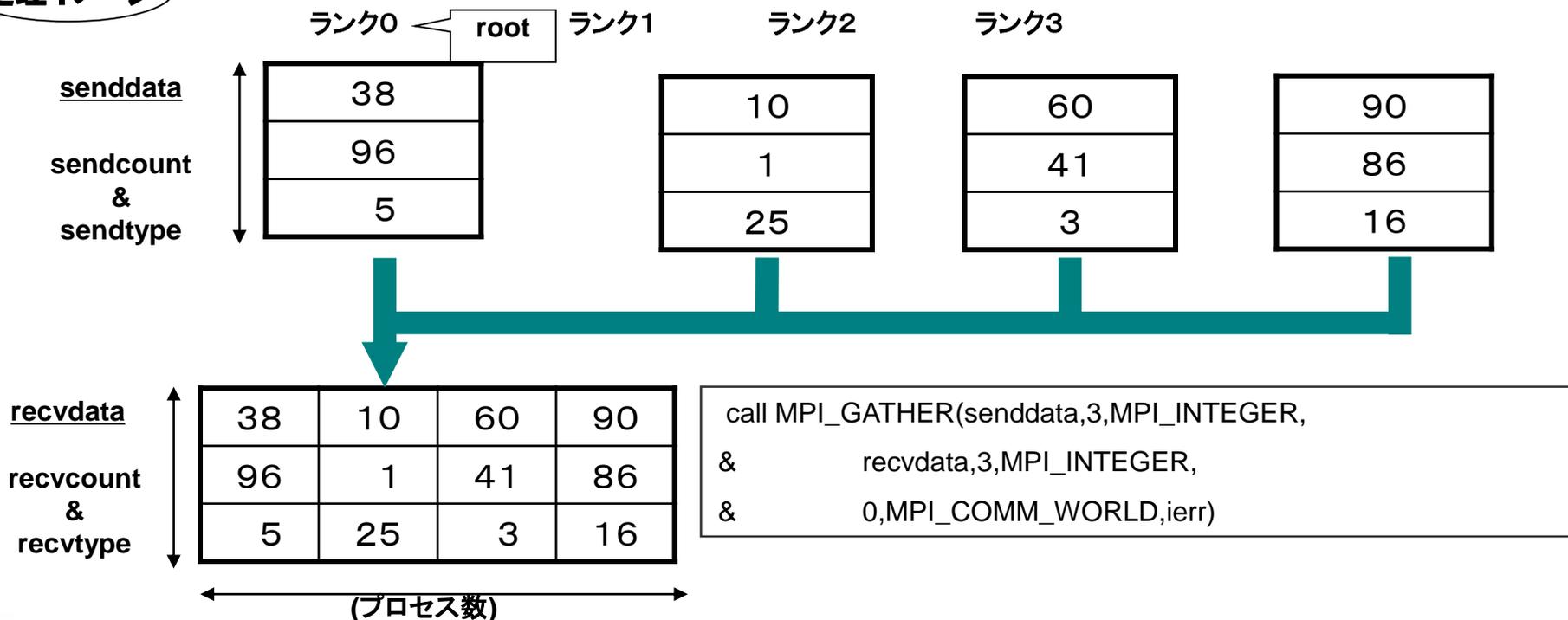


付録 1.3.8 MPI_GATHER データの集積

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



付録 1.3.8 MPI_GATHER データの集積 (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)
```

付録 1.3.8 MPI_GATHER データの集積 (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

[P40に戻る](#)

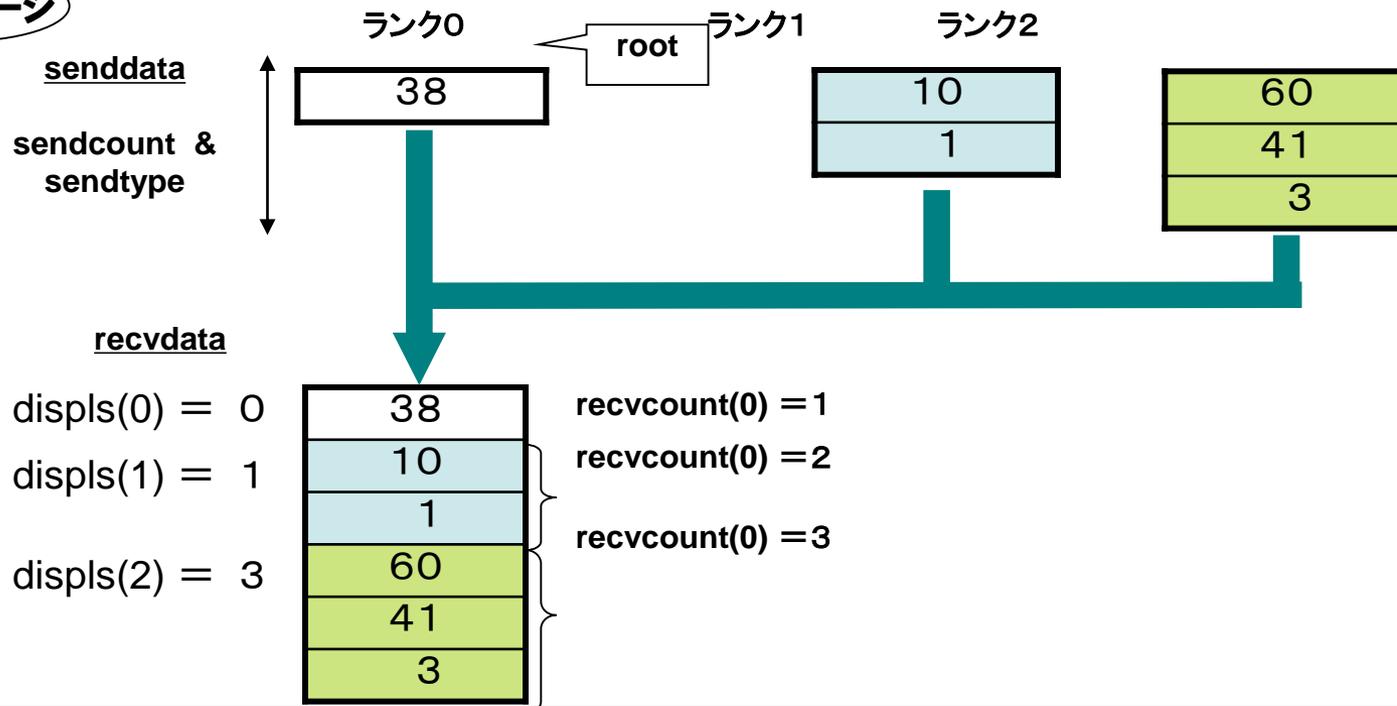
[P60に戻る](#)

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



付録 1.3.9 MPI_GATHERV データの集積 (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)
```

付録 1.3.9 MPI_GATHERV データの集積 (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 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 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

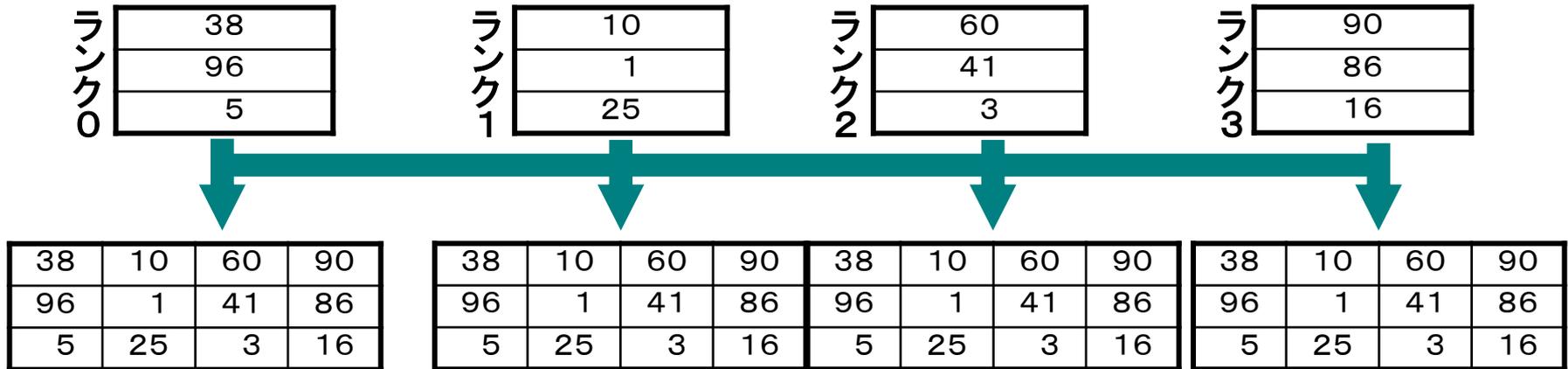
☆ …rootプロセスだけが意味を持つ

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



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

付録 1.3.10 MPI_ALLGATHER 全プロセスでデータ集積(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)
```

付録 1.3.10 MPI_ALLGATHER 全プロセスでデータ集積(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

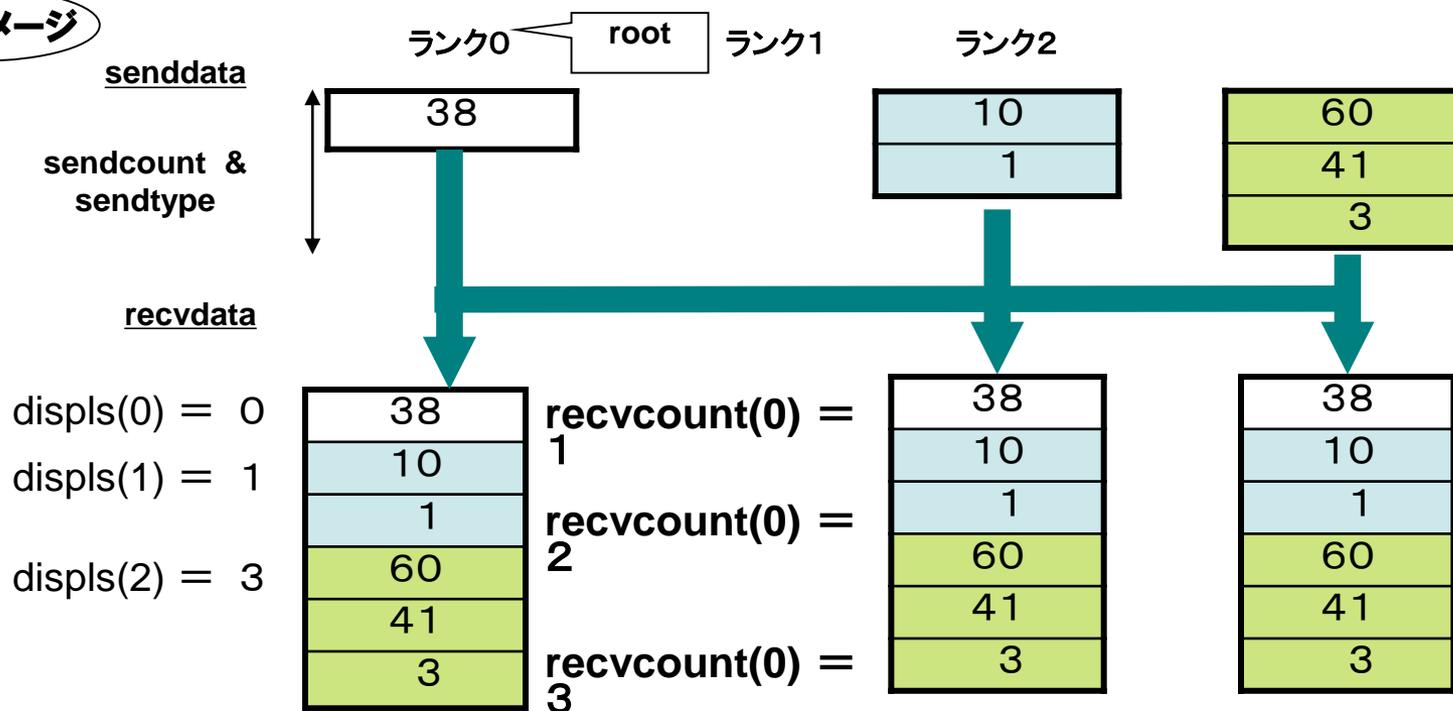
- MPI_GATHERの結果を全プロセスに送信するのと機能的に同じ

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



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)
```

付録 1.3.11 MPI_ALLGATHERV 全プロセスでデータ集積 (続き)

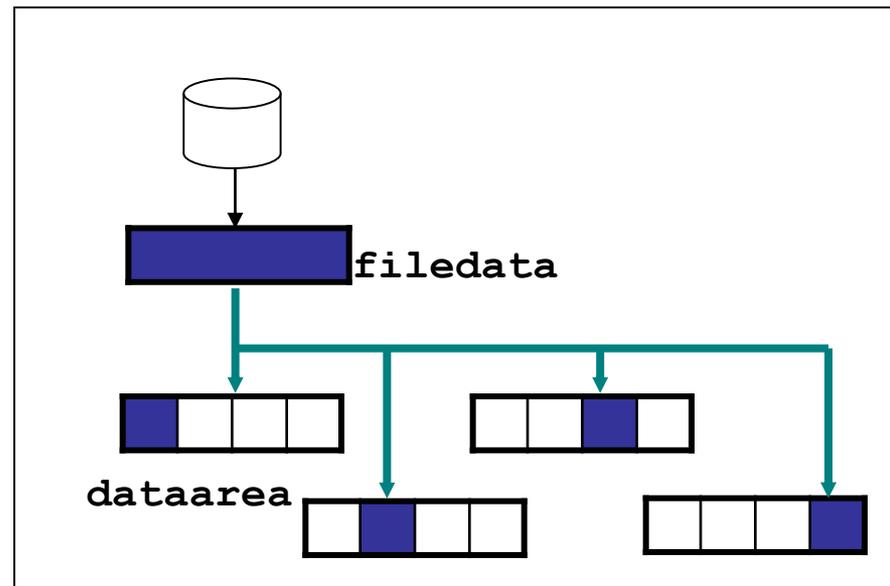
引数

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)

etc12.f

```
include 'mpif.h'
integer filedata(100),dataarea(100)
integer :: myrank,nprocs,ist,ied,ierr,isum1, isum
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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
```

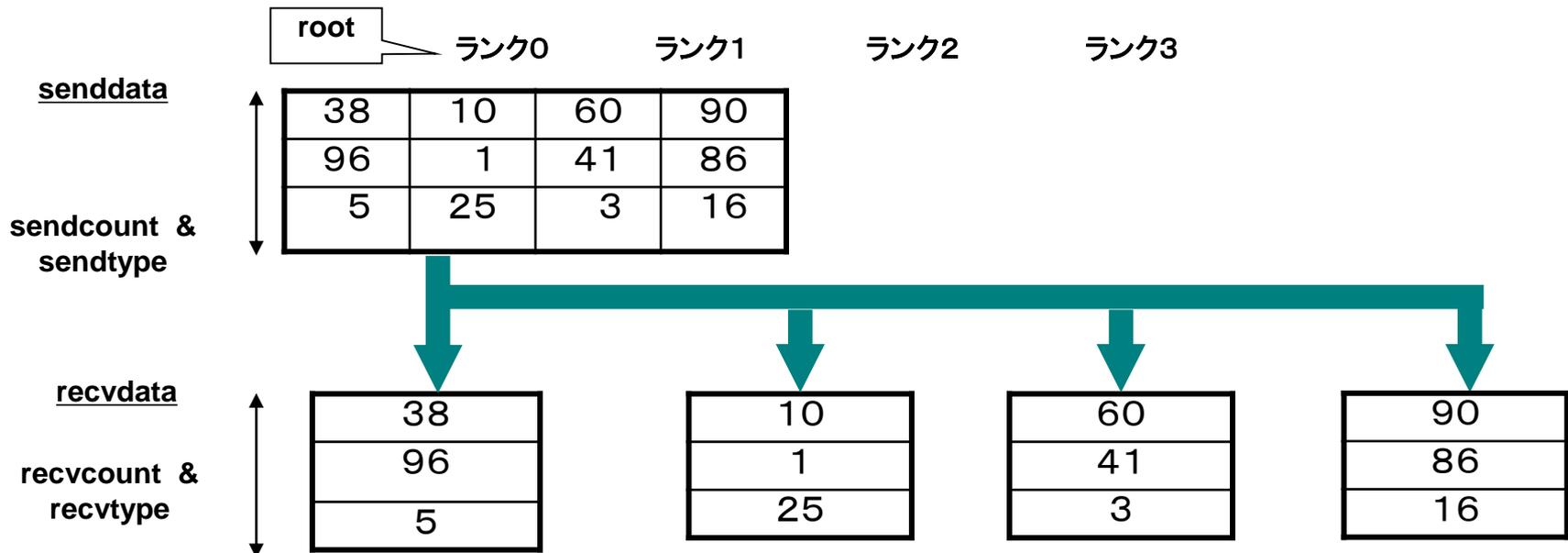


付録 1.3.13 MPI_SCATTER データの分配

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



付録 1.3.13 MPI_SCATTER データの分配 (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)
```

付録 1.3.13 MPI_SCATTER データの分配 (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

[P39に戻る](#)

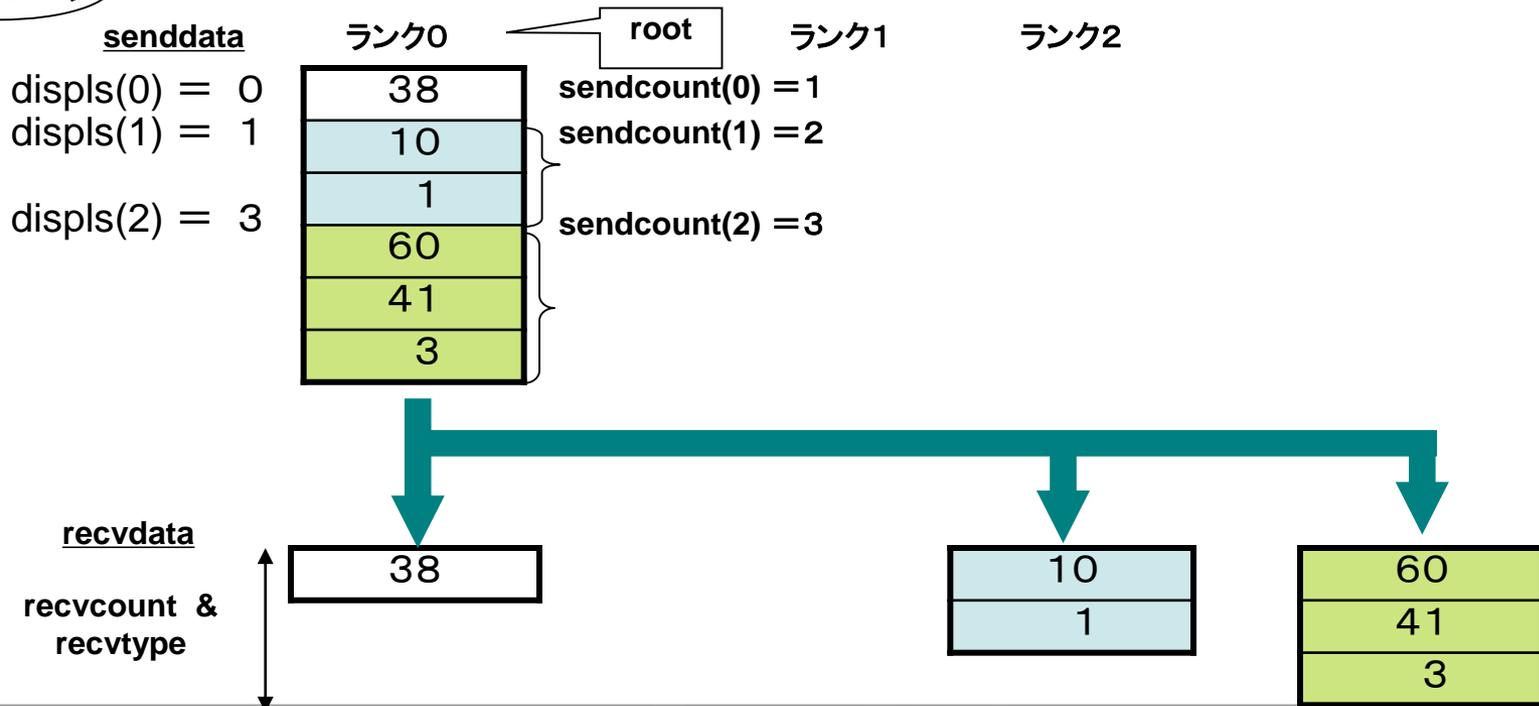
[P57に戻る](#)

付録 1.3.14 MPI_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



付録 1.3.14 MPI_SCATTERV データの分配 (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)
```

付録 1.3.14 MPI_SCATTERV データの分配 (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 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

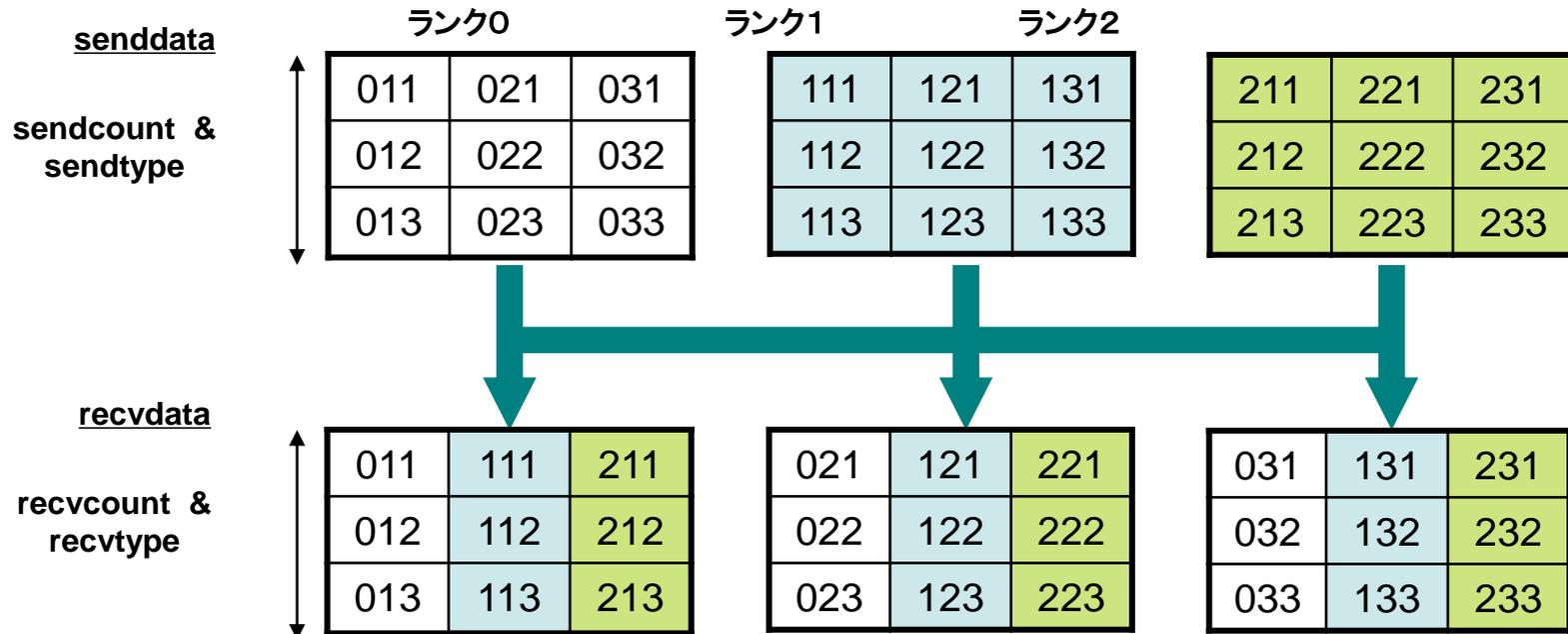
P39に戻る

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



付録 1.3.15 MPI_ALLTOALL データ配置 (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)
```

付録 1.3.15 MPI_ALLTOALL データ配置 (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

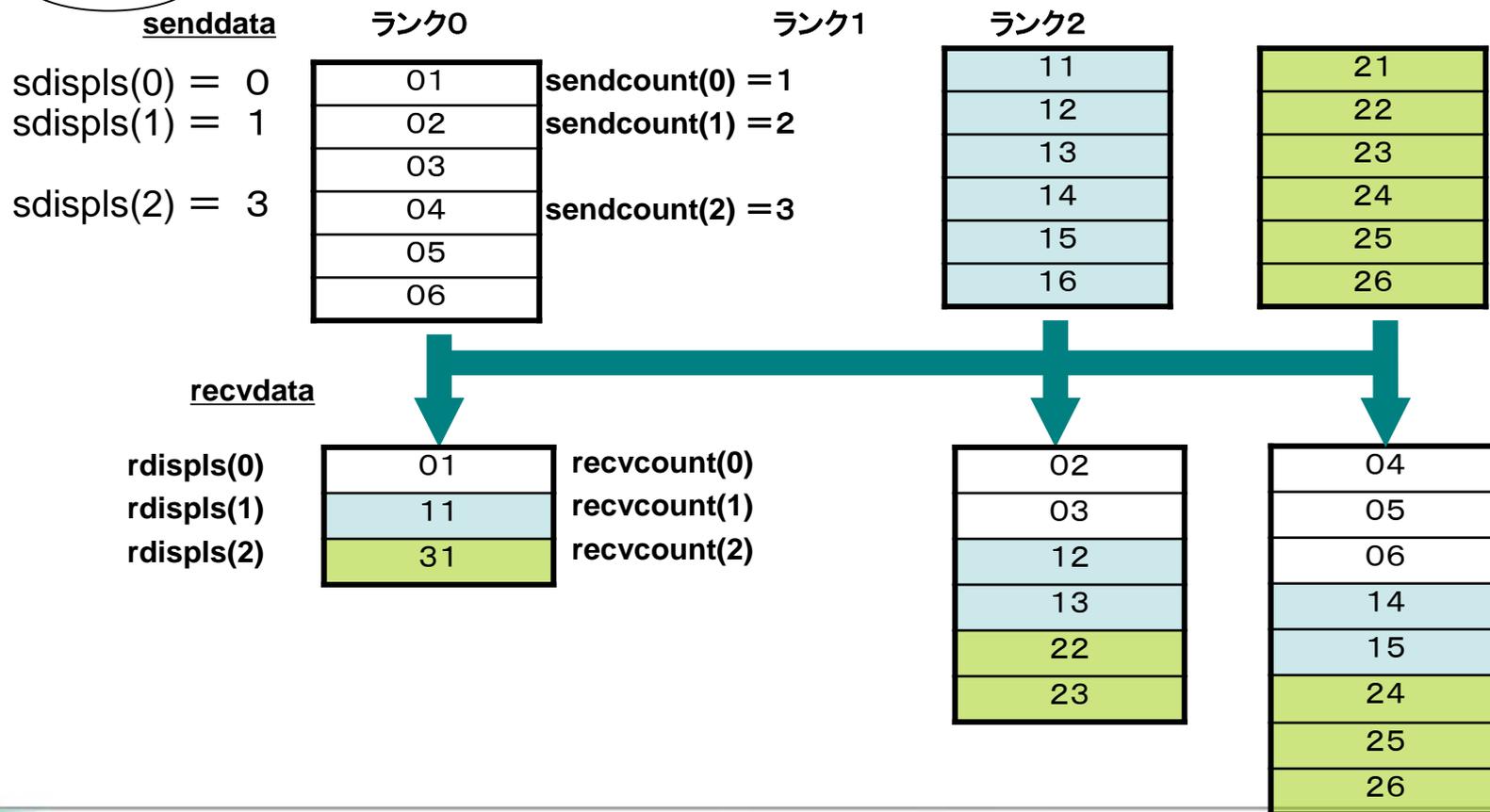
- 全対全スキヤッタ／ギャザ、または全交換とも呼ばれる

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



付録 1.3.16 MPI_ALLTOALLV データ配置

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)
```

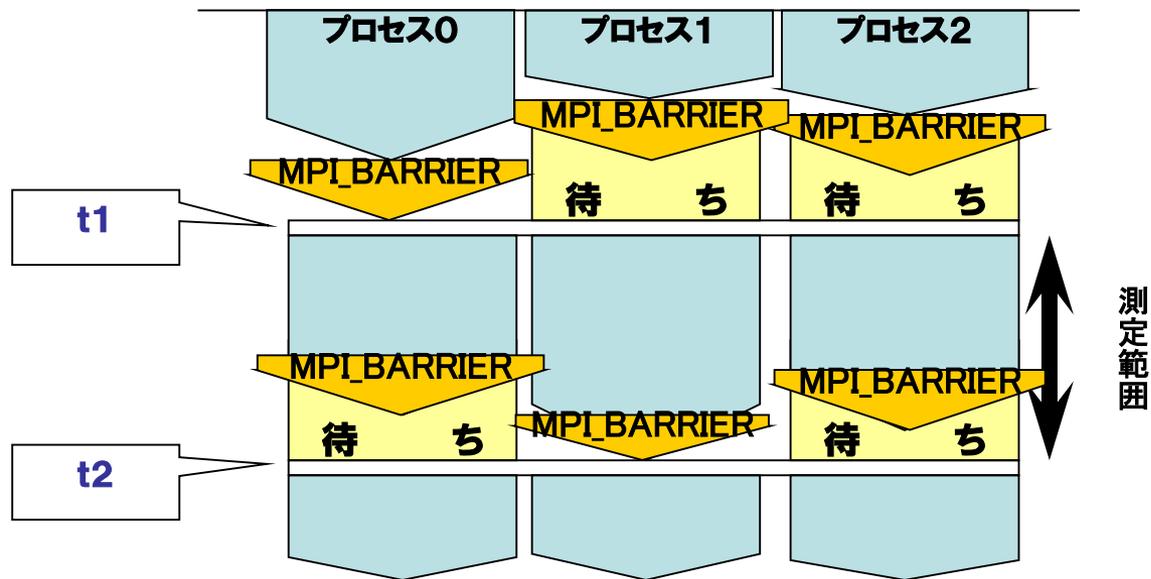
付録 1.3.16 MPI_ALLTOALLV データ配置

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)



$$(\text{測定時間}) = t_2 - t_1$$

付録 1.4.1 計時 (プログラム例)

etc13.f

```
include 'mpif.h'
parameter(numdat=100)
integer myrank,nprocs,ist,ied,ierr,isum1,isum
real*8 t1,t2,tt
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,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良い
- 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」

http://acc.riken.jp/wp-content/uploads/2015/06/secure_4467_parallel-programming_main.pdf

■ 「並列プログラミング虎の巻MPI版」

https://www.hpci-office.jp/invite2/documents2/mpi-all_20160801_20181206.pdf

演習問題解答例

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) write(6,*) "Hello World",myrank
call MPI_FINALIZE(ierr)
stop
end
```

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
```

演習問題3 (practice_3) 解答例

```
program example3
include 'mpif.h'
integer ierr,myrank,nprocs,ist,ied,itag
integer status(MPI_STATUS_SIZE)
parameter(n=1000)
integer isum,i sum2
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
  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
```

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
  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
```

※ 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。

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
```

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
  deallocate(recvdata)
endif
deallocate(senddata)
call MPI_FINALIZE(ierr)
stop
end
```

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
```

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)
```

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,iend
  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
```