MPIによるプログラミング概要（その１）
【Fortran言語編】

RIKEN AICS HPC Summer School 2015
中島研吾（東大・情報基盤センター）
横川三津夫（神戸大・計算科学教育センター）

本schoolの目的

- 並列計算機の使用によって，より大規模で詳細なシミュレーションを高速に実施することが可能になり，新しい科学の開拓が期待される・・・

- 並列計算の目的
  - 高速
  - 大規模
  - 「大規模」の方が「新しい科学」という観点からのウェイトとしては高い．しかし，「高速」ももちろん重要である．
  - +複雑
  - 理想：Scalable
    - N倍の規模の計算をN倍のCPUを使って，「同じ時間で」解く

概要

- MPIとは
- MPIの基礎：Hello Worldを並列で出力する
- 全体データと局所データ
- グループ通信（Collective Communication）
- 1対1通信（Peer-to-Peer Communication）

概要

- MPIとは

- MPIの基礎：Hello Worldを並列で出力する

- 全体データと局所データ
- グループ通信（Collective Communication）
- 1対1通信（Peer-to-Peer Communication）

MPIとは（1/2）

- Message Passing Interface
- 分散メモリ間のメッセージ通信APIの「規格」
  - プログラムやライブラリそのものではない
- 歴史
  - 1992　MPIフォーラム
  - 1994　MPI-1規格
  - 1997　MPI-2規格：MPI I/Oなど
  - 2012　MPI-3規格：
- 実装（こっちはライブラリ）
  - MPICH：　アルゴンヌ国立研究所
  - OpenMP, MVAPICHなど
  - 各ベンダーのMPIライブラリ
    - C/C++，Fortran，Java；　Unix，Linux，Windows，Mac OS

MPIとは（2/2）

- 現状では，MPICH（フリー）が広く使用されている.
  - 部分的に「MPI-2」規格をサポート
  - 2005年11月から「MPICH2」に移行
    - http://www.mpich.org/

- MPIが普及した理由
  - MPIフォーラムによる規格統一
    - どんな計算機でも動く
    - Fortran，Cからサブルーチンとして呼び出すことが可能
  - MPICHの存在
    - フリー，あらゆるアーキテクチュアをサポート

参考文献

- P.Pacheco「MPI並列プログラミング」，培風館，2001（原著1997）
- W.Gropp他「Using MPI second edition」，MIT Press, 1999.
- M.J.Quinn「Parallel Programming in C with MPI and OpenMP」，McGrawhill, 2003.
- W.Gropp他「MPI：The Complete Reference Vol.I, II」，MIT Press, 1998.
- MPICH2
  - http://www.mpich.org/
  - API（Application Interface）の説明

MPIを学ぶにあたって

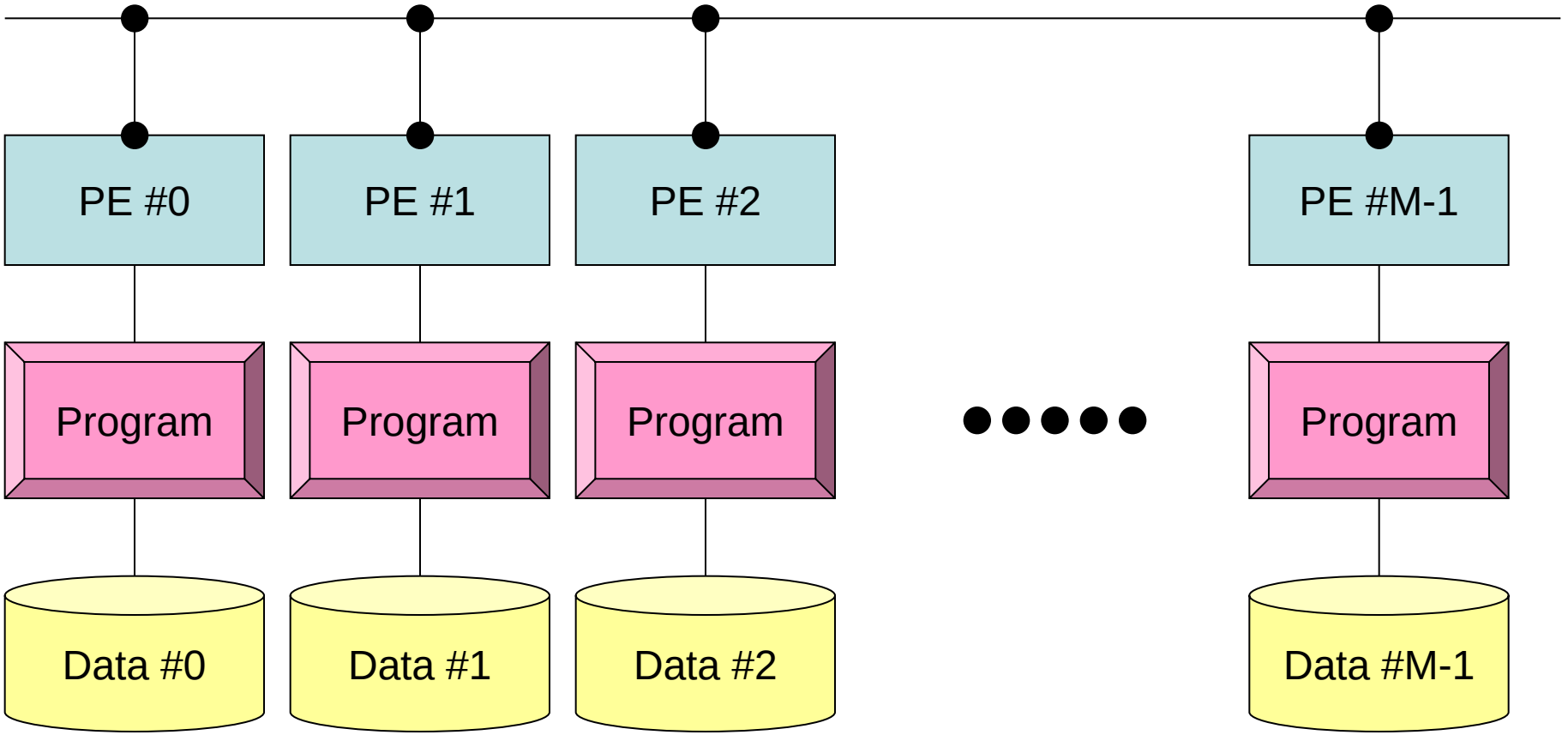
- 文法
  - 「MPI-1」の基本的な機能（10個程度）について習熟する.
    - MPI-2では色々と便利な機能があるが・・・
  - あとは自分に必要な機能について調べる，あるいは知っている人，知っていそうな人に尋ねる.
- 実習の重要性
  - プログラミング
  - その前にまず実行してみること
- SPMD/SIMDのオペレーションに慣れること・・・「つかむ」こと
  - Single Program Multiple Data / Single Instruction Multiple Data
  - 基本的に各プロセスは「同じことをやる」が「データが違う」
    - 大規模なデータを分割し，各部分について各プロセス（プロセッサ）が計算する
  - 全体データと局所データ，全体番号と局所番号

PE: Processing Element
プロセッサ，領域，プロセス

SPMD

この絵が理解できればMPIは9割方，理解できたことになる．コンピュータサイエンスの学科でもこれを上手に教えるのは難しいらしい．

```
mpirun -np M <Program>
```



各プロセスでは「同じプログラムが動く」が「データが違う」
大規模なデータを分割し，各部分について各プロセス（プロセッサ）が計算する
通信以外は，単体CPUのときと同じ，というのが理想

用 語

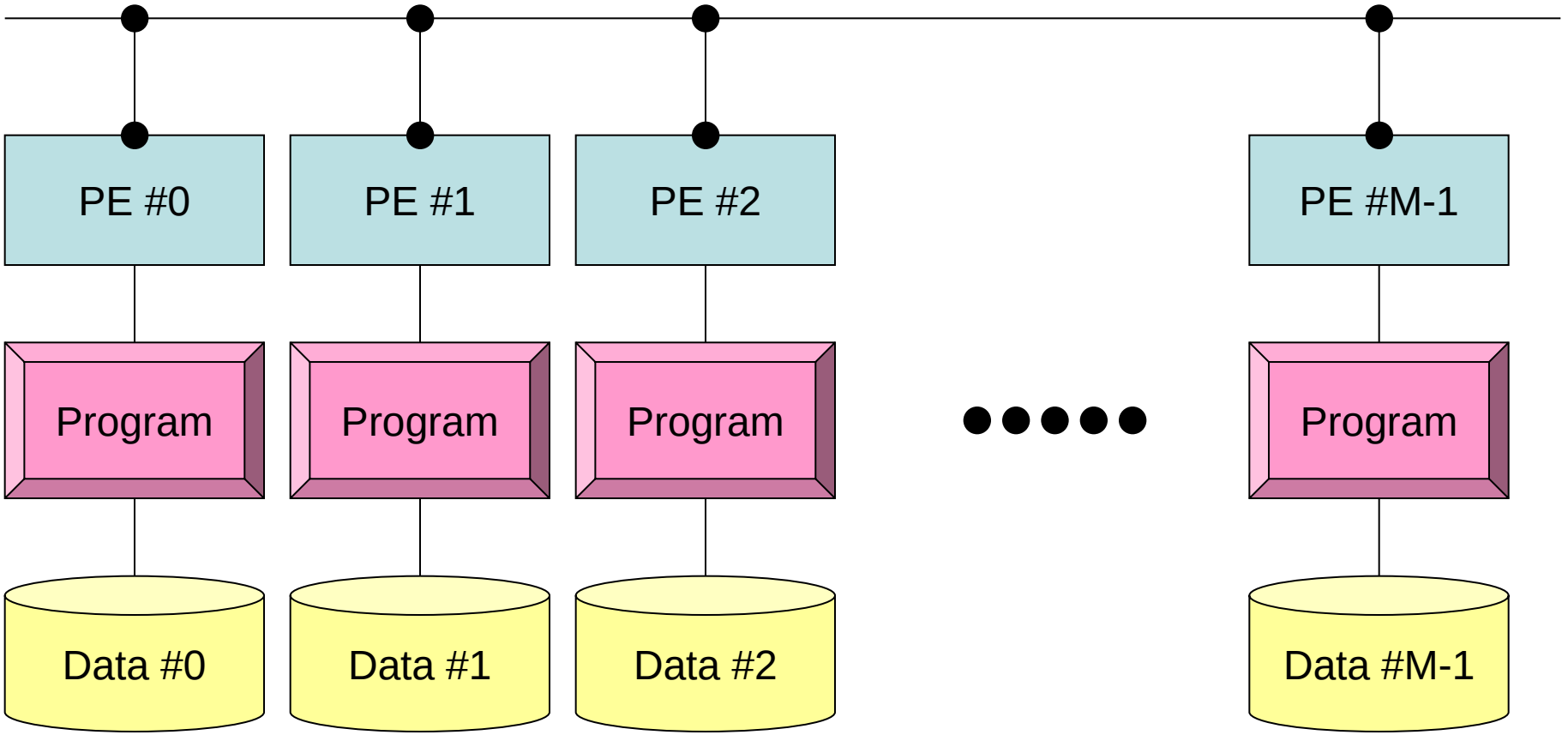
- プロセッサ，コア
  - ハードウェアとしての各演算装置．シングルコアではプロセッサ＝コア
- プロセス
  - MPI計算のための実行単位，ハードウェア的な「コア」とほぼ同義．
  - しかし1つの「プロセッサ・コア」で複数の「プロセス」を起動する場合もある（効率的ではないが）．
- PE（Processing Element）
  - 本来，「プロセッサ」の意味なのであるが，本講義では「プロセス」の意味で使う場合も多い．次項の「領域」とほぼ同義でも使用．
    - マルチコアの場合は：「コア＝PE」という意味で使うことが多い．
- 領域
  - 「プロセス」とほぼ同じ意味であるが，SPMDの「MD」のそれぞれ一つ，「各データ」の意味合いが強い．しばしば「PE」と同義で使用．
- MPIのプロセス番号（PE番号，領域番号）は0から開始
  - したがって8プロセス（PE，領域）ある場合は番号は0～7

PE: Processing Element
プロセッサ，領域，プロセス

SPMD

この絵が理解できればMPIは9割方，理解できたことになる．コンピュータサイエンスの学科でもこれを上手に教えるのは難しいらしい．

```
mpirun -np M <Program>
```



各プロセスでは「同じプログラムが動く」が「データが違う」
大規模なデータを分割し，各部分について各プロセス（プロセッサ）が計算する
通信以外は，単体CPUのときと同じ，というのが理想

講義，課題の予定

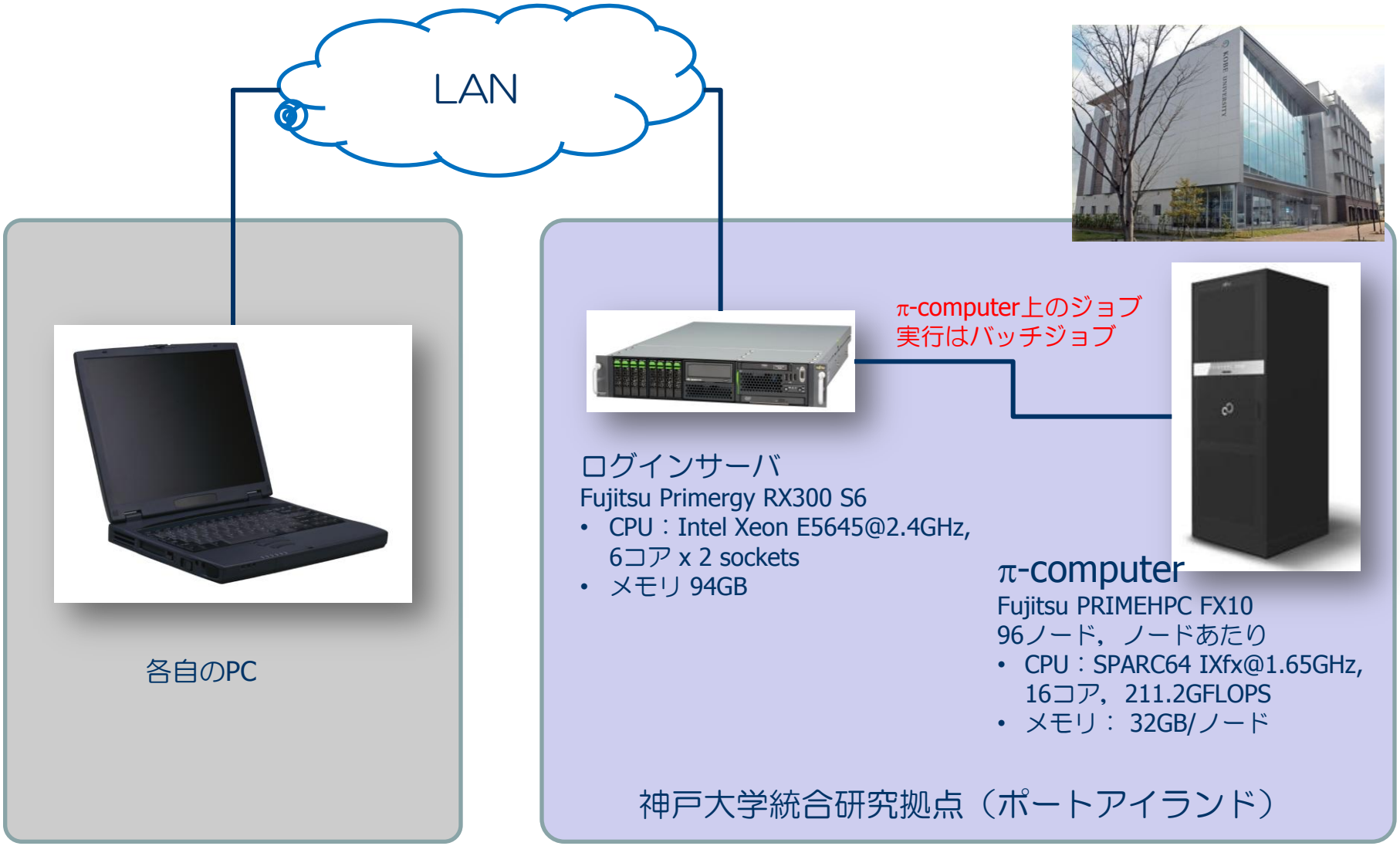
- MPIサブルーチン機能
  - 環境管理
  - グループ通信
  - 1対1通信

- 8月18日（火）
  - 環境管理，グループ通信（Collective Communication）
    - 課題S1
- 8月19日（水）
  - 1対1通信（Point-to-Point Communication）
    - 課題S2：一次元熱伝導解析コードの「並列化」
  - ここまでできればあとはある程度自分で解決できるはず．

概要

- MPIとは
- MPIの基礎：Hello Worldを並列で出力する
- 全体データと局所データ
- グループ通信（Collective Communication）
- 1対1通信（Peer-to-Peer Communication）

schoolで利用するπコンピュータ



ログイン，ディレクトリ作成 on πコンピュータ

```
ssh xxxxxxx@pi.ircpi.kobe-u.ac.jp

ディレクトリ作成
  >$ cd
  >$ mkdir 2015summer  （好きな名前でよい）
  >$ cd 2015summer

  このディレクトリを本講義では <$P-TOP> と呼ぶ
  基本的にファイル類はこのディレクトリにコピー，解凍する
```

ファイルコピー

Fortranユーザ

```
>$ cd <$P-TOP>
>$ cp /tmp/2015summer/F/s1-f.tar .
>$ tar xvf s1-f.tar
```

Cユーザ

```
>$ cd <$P-TOP>
>$ cp /tmp/2015summer/C/s1-c.tar .
>$ tar xvf s1-c.tar
```

ディレクトリ確認

```
>$ ls
   mpi

>$ cd mpi/S1
```

このディレクトリを本講義では <$P-S1> と呼ぶ.

<$P-S1> = <$P-TOP>/mpi/S1

まずはプログラムの例

hello.f

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

hello.c

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

hello.f/hello.c をコンパイルしてみよう！

```
>$ mpifrtpx -Kfast hello.f
>$ mpifccpx -Kfast hello.c
```

Fortran

```
$> mpifrtpx -Kfast hello.f
```

"mpifrtpx":
Fortran90＋MPIによってプログラムをコンパイルする際に
必要なコンパイラ，ライブラリ等がバインドされているコマンド

C言語

```
$> mpifccpx -Kfast hello.c
```

"mpifccpx":
C＋MPIによってプログラムをコンパイルする際に
必要な，コンパイラ，ライブラリ等がバインドされているコマンド

ジョブ実行

- 実行方法
  - 基本的にバッチジョブのみ
  - 会話型の実行は「基本的に」やりません.
- 実行手順
  - ジョブスクリプトを書きます.
  - ジョブを投入します.
  - ジョブの状態を確認します.
  - 結果を確認します.
- その他
  - 実行時には1ノード（16コア）が占有されます.
  - 他のユーザーのジョブに使われることはありません.

ジョブスクリプト

- **<$P-S1>/hello.sh**
- スケジューラへの指令 ＋ シェルスクリプト

```
#!/bin/sh
#PJM -L “node=1”                    ノード数
#PJM -L “elapse=00:00:30”           実行時間
#PJM -L “rscgrp=school”             実行キュー名
#PJM -j
#PJM -o “hello.lst”                 標準出力ファイル名
#PJM --mpi “proc=4”                 MPIプロセス数

mpiexec ./a.out                     実行ファイル名
```

8プロセス	16プロセス	32プロセス	64プロセス	192プロセス
“node=1”	“node=1”	“node=2”	“node=4”	“node=12”
“proc=8”	“proc=16”	“proc=32”	“proc=64”	“proc=192”

ジョブ投入

```
>$ pjsub hello.sh

>$ cat hello.lst

  Hello World Fortran    0    4
  Hello World Fortran    2    4
  Hello World Fortran    3    4
  Hello World Fortran    1    4
```

ジョブ投入，確認等

- ジョブの投入　　`pjsub` スクリプト名
- ジョブの確認　　`pjstat`
- ジョブの取り消し･強制終了　　`pjdel` ジョブID
- キューの状態の確認　　`pjstat --rsc`
- 同時実行･投入可能数　　`pjstat --limit`

```
[pi:~/2015summer/mpi/S1]$ pjstat

  ACCEPT QUEUED  STGIN  READY RUNING RUNOUT STGOUT   HOLD  ERROR   TOTAL
       0      0      0      0      1      0      0      0      0       1
s      0      0      0      0      1      0      0      0      0       1

JOB_ID     JOB_NAME   MD ST  USER     START_DATE      ELAPSE_LIM NODE_REQUIRE
73804      hello.sh   NM RUN yokokawa 07/15 17:12:26  0000:00:10 1
```

環境管理ルーチン＋必須項目

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

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

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

‘mpif.h’, “mpi.h”
環境変数デフォルト値
Fortran90ではuse mpi可

MPI_Init
初期化

MPI_Comm_size
プロセス数取得
mpirun -np XX <prog>

MPI_Comm_rank
プロセスID取得
自分のプロセス番号（0から開始）

MPI_Finalize
MPIプロセス終了

Fortran/Cの違い

- 基本的にインタフェースはほとんど同じ
  - Cの場合，「**MPI_C**omm_size」のように「MPI」は大文字，「MPI_」のあとの最初の文字は大文字，以下小文字

- Fortranはエラーコード（ierr）の戻り値を引数の最後に指定する必要がある.
- Cは変数の特殊な型がある.
  - MPI_Comm, MPI_Datatype, MPI_Op, etc.
- 最初に呼ぶ「MPI_Init」だけは違う
  - `call MPI_INIT (ierr)`
  - `MPI_Init (int *argc, char ***argv)`

何をやっているのか？

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i5)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

```
#!/bin/sh
#PJM -L "node=1"                ノード数
#PJM -L "elapse=00:10:00"       実行時間
#PJM -L "rscgrp=school"         実行キュー名
#PJM -j
#PJM -o "hello.lst"             標準出力ファイル名
#PJM --mpi "proc=4"             MPIプロセス数

mpiexec ./a.out                 実行ファイル名
```

- **mpiexec** により4つのプロセスが立ち上がる（今の場合は”proc=4”）.
  - 同じプログラムが4つ流れる.
  - データの値（my_rank）を書き出す.
- 4つのプロセスは同じことをやっているが，データとして取得したプロセスID（my_rank）は異なる.
- 結果として各プロセスは異なった出力をやっていることになる.
- まさにSPMD

mpi.h, mpif.h

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

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

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

- MPIに関連した様々なパラメータおよび初期値を記述.
- 変数名は「MPI_」で始まっている.
- ここで定められている変数は，MPIサブルーチンの引数として使用する以外は陽に値を変更してはいけない.
- ユーザーは「MPI_」で始まる変数を独自に設定しないのが無難.

Fortran

MPI_INIT

- MPIを起動する．他のMPIサブルーチンより前にコールする必要がある（必須）
- 全実行文の前に置くことを勧める．

- **call MPI_INIT (ierr)**
  - **ierr**　整数　○　完了コード

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

Fortran

MPI_FINALIZE

- MPIを終了する．他の全てのMPIサブルーチンより後にコールする必要がある（必須）．
- 全実行文の後に置くことを勧める
- これを忘れると大変なことになる．
  - 終わったはずなのに終わっていない・・・
- **call MPI_FINALIZE (ierr)**
  - **ierr**　整数　○　完了コード

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

Fortran

MPI_COMM_SIZE

- コミュニケーター「comm」で指定されたグループに含まれるプロセス数の合計が「size」に返ってくる．必須では無いが，利用することが多い．

- **call MPI_COMM_SIZE (comm, size, ierr)**
  - **comm** 整数 I コミュニケータを指定する
  - **size** 整数 O comm.で指定されたグループ内に含まれるプロセス数の合計
  - **ierr** 整数 O 完了コード

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

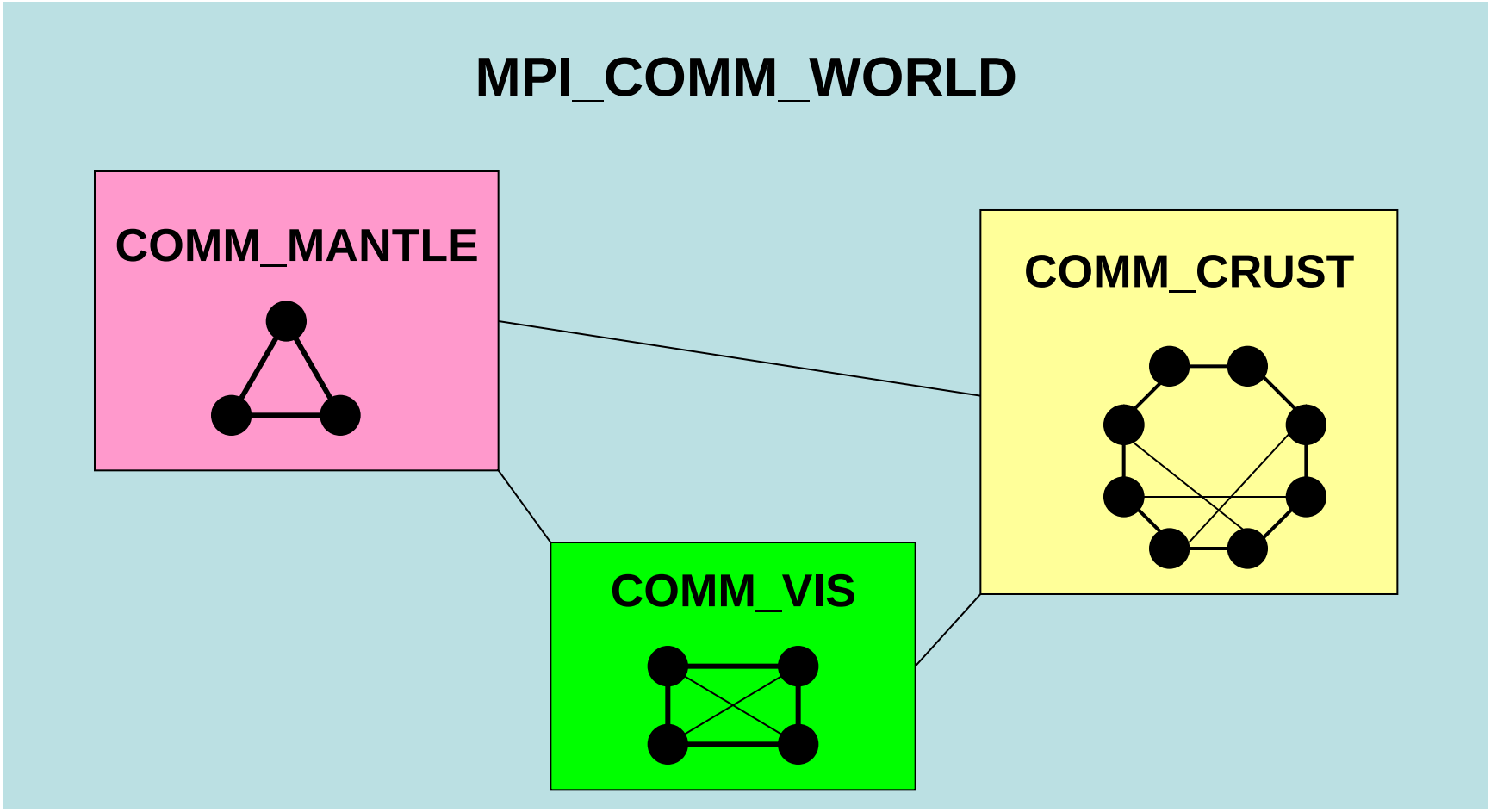
コミュニケータとは？

```
MPI_Comm_Size (MPI_COMM_WORLD, PETOT)
```

- 通信を実施するためのプロセスのグループを示す.
- MPIにおいて，通信を実施する単位として必ず指定する必要がある.
- mpiexecで起動した全プロセスは，デフォルトで「**MPI_COMM_WORLD**」というコミュニケータで表されるグループに属する.
- 複数のコミュニケータを使用し，異なったプロセス数を割り当てることによって，複雑な処理を実施することも可能.
  - 例えば計算用グループ，可視化用グループ
- この授業では「**MPI_COMM_WORLD**」のみでOK.

コミュニケータの概念

あるプロセスが複数のコミュニケータグループに属しても良い



Fortran

MPI_COMM_RANK

- コミュニケータ「comm」で指定されたグループ内におけるプロセスIDが「rank」にもどる．必須では無いが，利用することが多い．
  - プロセスIDのことを「rank（ランク）」と呼ぶことも多い．

- **MPI_COMM_RANK (comm, rank, ierr)**
  - **comm** 整数 I コミュニケータを指定する
  - **rank** 整数 O comm.で指定されたグループにおけるプロセスID
0から始まる（最大はPETOT-1）
  - **ierr** 整数 O 完了コード

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World Fortran', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

Fortran

MPI_ABORT

- MPIプロセスを異常終了する.

- **call MPI_ABORT (comm, errcode, ierr)**
  - **comm** 整数 I コミュニケータを指定する
  - **errcode** 整数 O エラーコード
  - **ierr** 整数 O 完了コード

Fortran

MPI_WTIME

- 時間計測用の関数：精度はいまいち良くない（短い時間を計測する場合）
- **time= MPI_WTIME ()**
  - **time** R8 O 過去のある時間からの経過時間（秒数）：倍精度変数

```
…
real(kind=8):: Stime, Etime

Stime= MPI_WTIME ()
do i= 1, 100000000
  a= 1.d0
enddo
Etime= MPI_WTIME ()

write (*,'(i5,1pe16.6)') my_rank, Etime-Stime
```

MPI_Wtime の例

```
$> mpifccpx –O1 time.c
$> mpifrtpx –O1 time.f

$> pjsub go4.sh
$> cat test.lst
   2    3.399327E-06
   1    3.499910E-06
   0    3.499910E-06
   3    3.399327E-06
```

プロセス番号　　計算時間

MPI_Wtick

- MPI_Wtimeでの時間計測精度を確認する.
- ハードウェア，コンパイラによって異なる

- **time= MPI_Wtick ()**
  - **time** R8 O 時間計測精度（単位：秒）

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'

…
TM= MPI_WTICK ()
write (*,*) TM
…
```

```
double Time;

…
Time = MPI_Wtick();
printf("%5d%16.6E¥n", MyRank, Time);
…
```

MPI_Wtick の例

```
$> mpifccpx -O1 wtick.c
$> mpifrtpx -O1 wtick.f

$> pjsub go1.sh

$> cat test.lst
 1.000000000000000E-07
$>
```

Fortran

MPI_BARRIER

- コミュニケーター「comm」で指定されたグループに含まれるプロセスの同期をとる．コミュニケータ「comm」内の全てのプロセスがこのサブルーチンを通らない限り，次のステップには進まない．
- 主としてデバッグ用に使う．オーバーヘッドが大きいので，実用計算には使わない方が無難．

- **call MPI_BARRIER (comm, ierr)**
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

概要

- MPIとは
- MPIの基礎：Hello World
- 全体データと局所データ
- グループ通信（Collective Communication）
- 1対1通信（Peer-to-Peer Communication）

データ構造とアルゴリズム

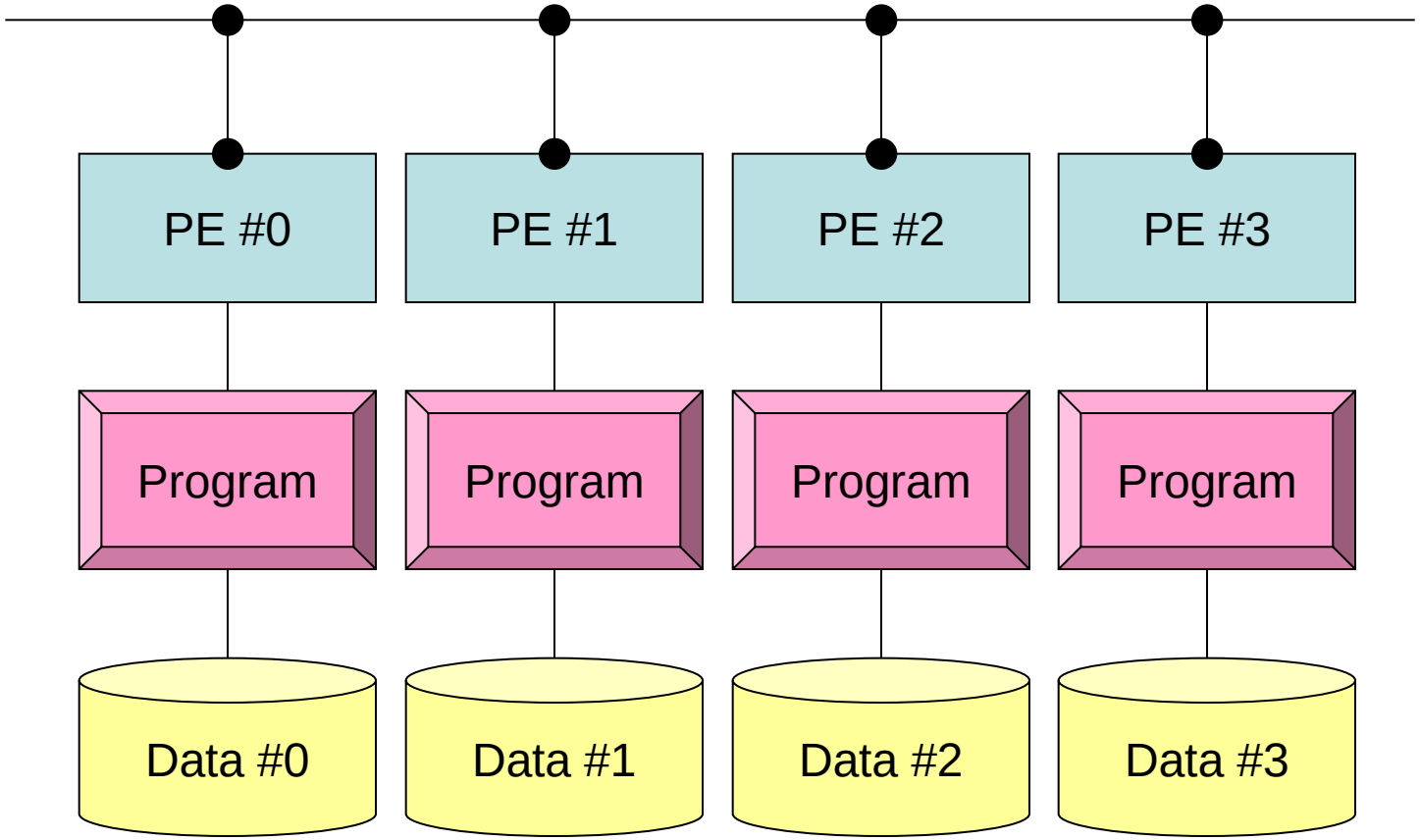
- コンピュータ上で計算を行うプログラムはデータ構造とアルゴリズムから構成される.
- 両者は非常に密接な関係にあり，あるアルゴリズムを実現するためには，それに適したデータ構造が必要である.
  - 極論を言えば「データ構造＝アルゴリズム」と言っても良い.

- 並列計算を始めるにあたって，基本的なアルゴリズムに適したデータ構造を定める必要がある.

SPMD：Single Program Multiple Data

- 一言で「並列計算」と言っても色々なものがあり，基本的なアルゴリズムも様々．
- 共通して言えることは，SPMD（Single Program Multiple Data）
- なるべく単体CPUのときと同じようにできることが理想
  - 通信が必要な部分とそうでない部分を明確にする必要があり．

SPMDに適したデータ構造とは？



SPMDに適したデータ構造(1/2)

- 大規模なデータ領域を分割して，各プロセッサ，プロセスで計算するのがSPMDの基本的な考え方
- 例えば，長さNG(=20)のベクトル**VG**に対して，各要素を2倍する計算を考えてみよう.

```
integer, parameter :: NG= 20
real(kind=8), dimension(20) :: VG

do i= 1, NG
  VG(i)= 2.0 * VG(i)
enddo
```

- これを4つのプロセッサで分担して計算する場合には，各プロセッサが20/4=5 ずつデータを持ち，それぞれが処理すればよい.

SPMDに適したデータ構造(2/2)

- すなわち，こんな感じ:

```
integer, parameter :: NL= 5
real(kind=8), dimension(5) :: VL

do i= 1, NL
  VL(i)= 2.0 * VL(i)
enddo
```

- このようにすれば「一種類の」プログラム(Single Program)で並列計算を実施できる.
  - ただし，各プロセスにおいて，「VL」の中身が違う:Multiple Data
  - 可能な限り計算を「VL」のみで実施することが，並列性能の高い計算へつながる.
  - プログラムの形は，単体CPUの場合とほとんど変わらない.

全体データと局所データ

- VG
  - 領域全体
  - 1番から20番までの「全体番号」を持つ「全体データ（Global Data）」
- VL
  - 各プロセス（PE，プロセッサ，領域）
  - 1番から5番までの「局所番号」を持つ「局所データ（Local Data）」
  - できるだけ局所データを有効に利用することで，高い並列性能が得られる．

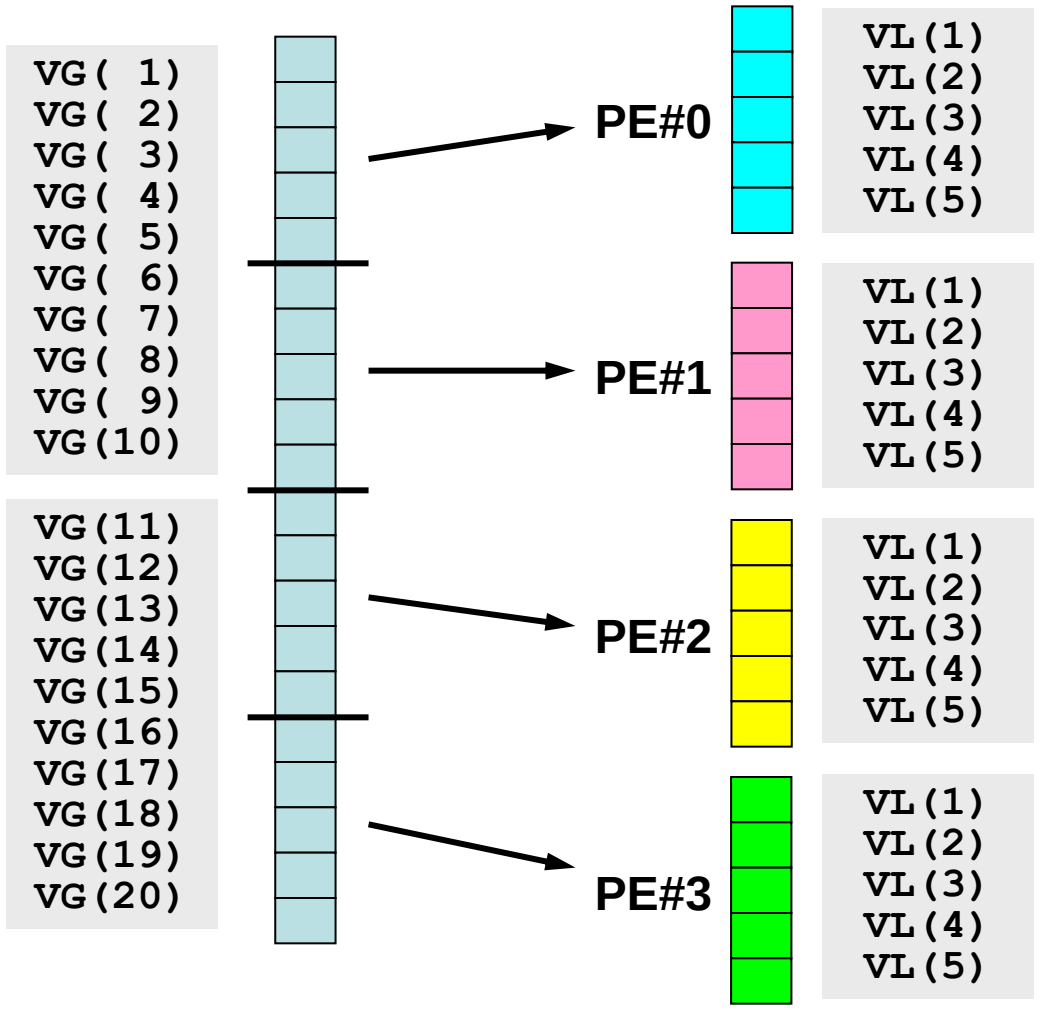
Fortran

局所データの考え方

「全体データ」VGの

- 1～5番成分がPE#0
- 6~10番成分がPE#1
- 11～15番成分がPE#2
- 16～20番成分がPE#3

のそれぞれ，「局所データ」VLの1番～5番成分となる（局所番号が1番～5番となる）.



全体データと局所データ

- VG
  - 領域全体
  - 1番から20番までの「全体番号」を持つ「全体データ（Global Data）」
- VL
  - 各プロセッサ
  - 1番から5番までの「局所番号」を持つ「局所データ（Local Data）」
- この講義で常に注意してほしいこと
  - VG（全体データ）からVL（局所データ）をどのように生成するか.
  - VGからVL，VLからVGへデータの中身をどのようにマッピングするか.
  - VLがプロセスごとに独立して計算できない場合はどうするか.
  - できる限り「局所性」を高めた処理を実施する⇒高い並列性能
    - そのための「データ構造」，「アルゴリズム」を考える.

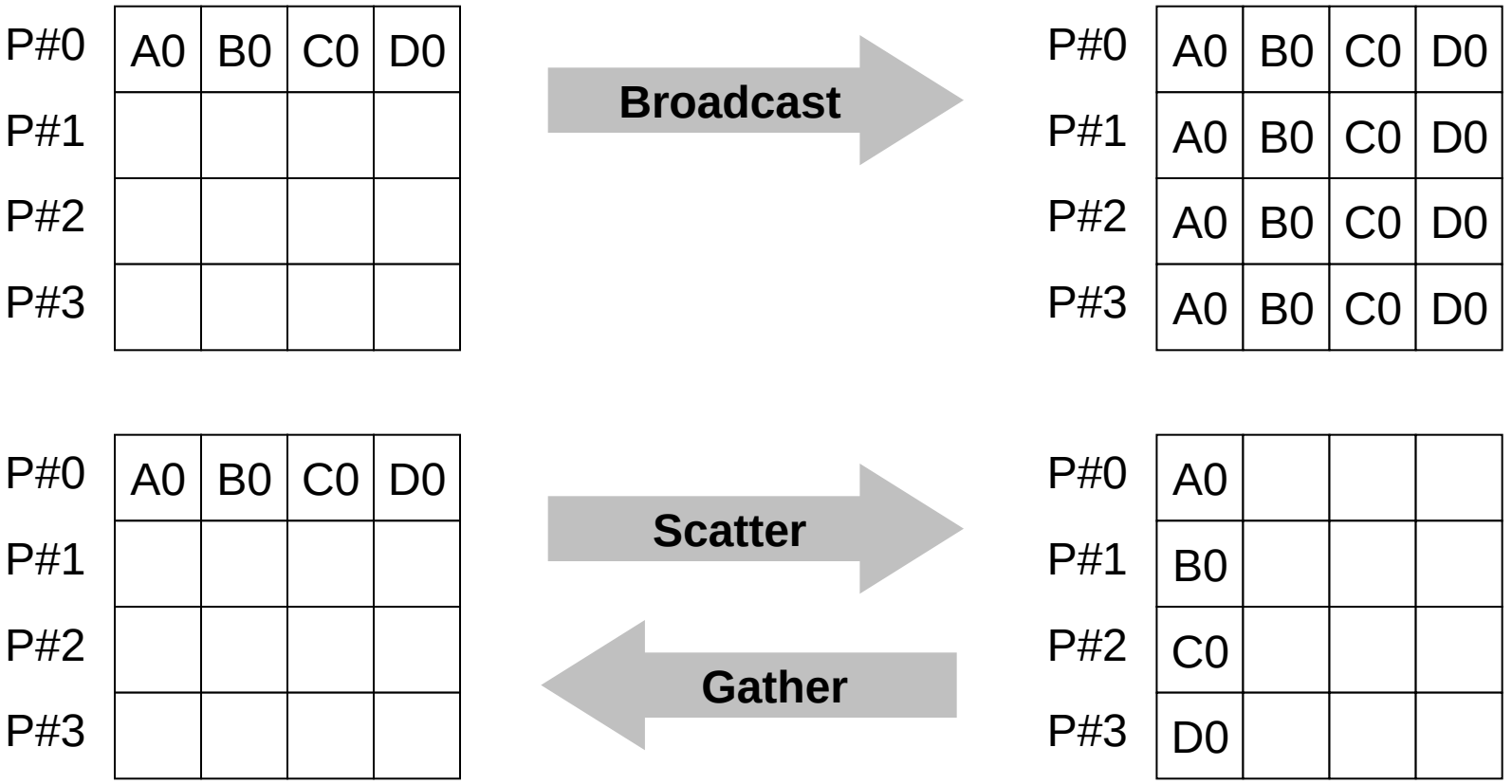
- MPIとは
- MPIの基礎：Hello World
- 全体データと局所データ
- グループ通信（Collective Communication）
- 1対1通信（Peer-to-Peer Communication）

グループ通信とは

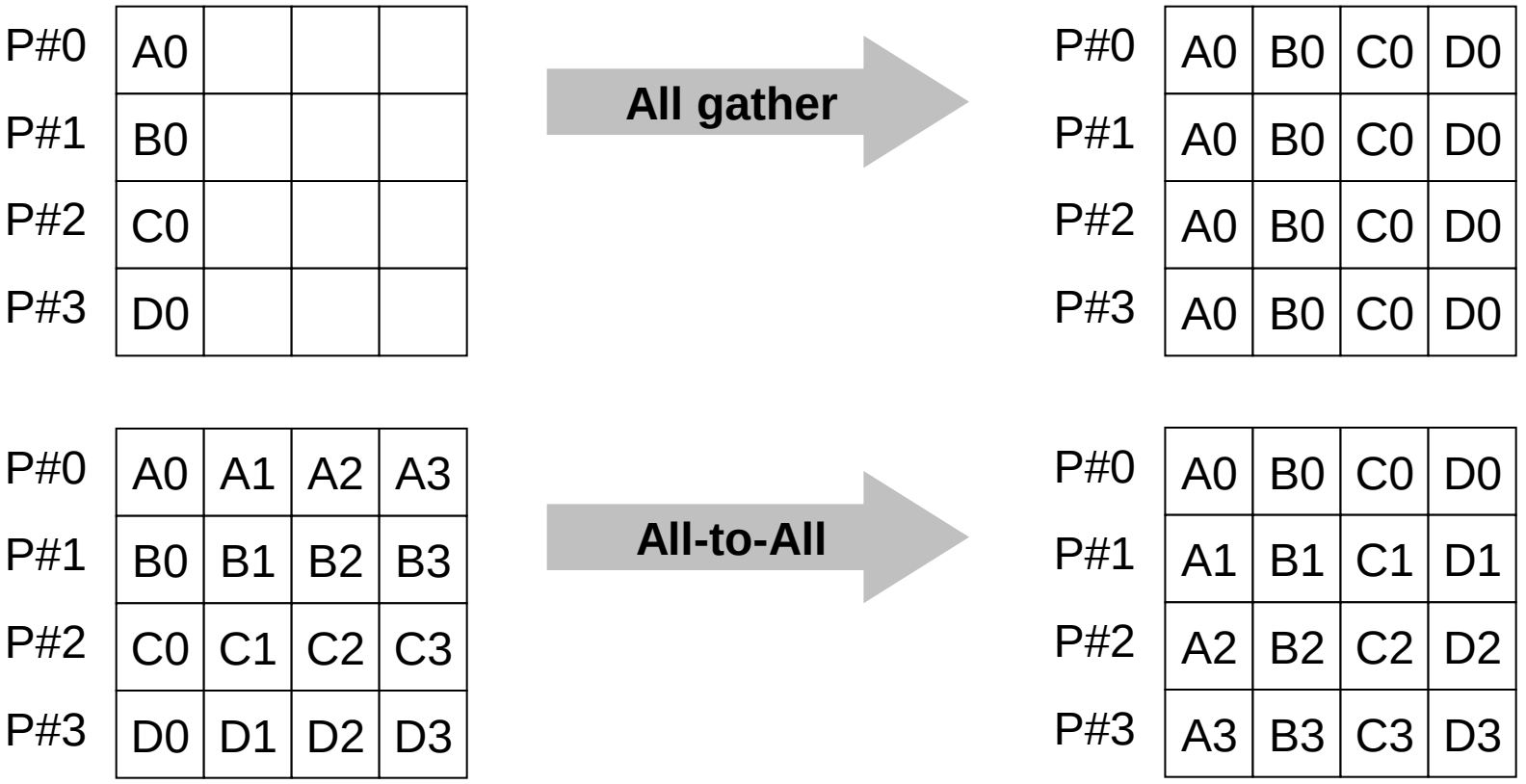
- コミュニケータで指定されるグループ全体に関わる通信.

- 例
  - 制御データの送信
  - 最大値，最小値の判定
  - 総和の計算
  - ベクトルの内積の計算
  - 密行列の転置

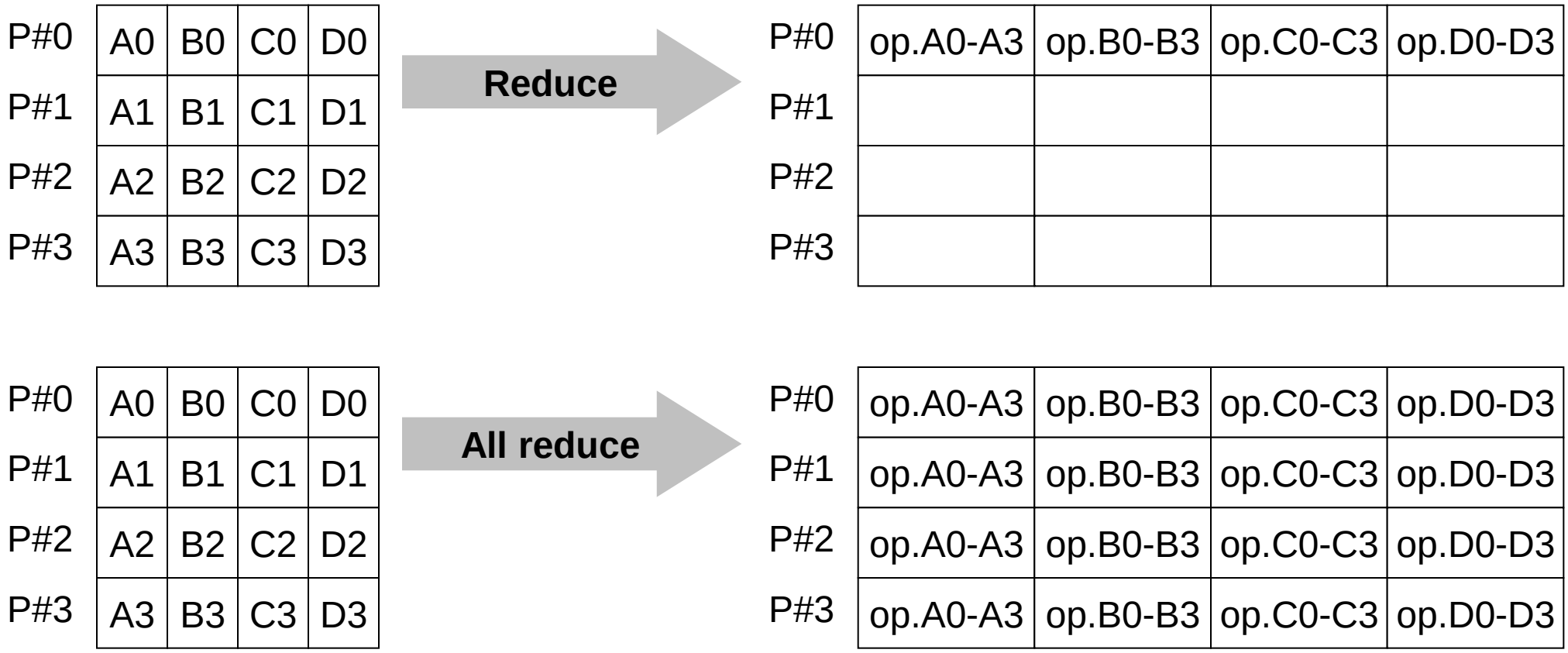
グループ通信の例(1/4)


P#0
P#1
P#2
P#3
A0
B0
C0
D0
Broadcast
P#0
P#1
P#2
P#3
A0
B0
C0
D0
P#0
P#1
P#2
P#3
A0
B0
C0
D0
Scatter
Gather
P#0
P#1
P#2
P#3
A0
B0
C0
D0

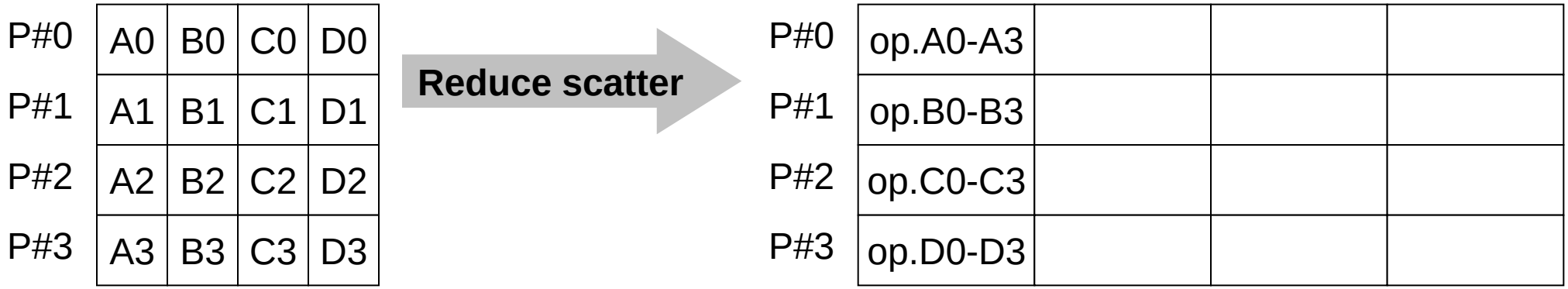
グループ通信の例(2/4)


P#0
P#1
P#2
P#3
A0
B0
C0
D0
All gather
P#0
P#1
P#2
P#3
A0 B0 C0 D0
A0 B0 C0 D0
A0 B0 C0 D0
A0 B0 C0 D0
P#0
P#1
P#2
P#3
A0 A1 A2 A3
B0 B1 B2 B3
C0 C1 C2 C3
D0 D1 D2 D3
All-to-All
P#0
P#1
P#2
P#3
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3

グループ通信の例(3/4)


P#0
P#1
P#2
P#3
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
Reduce
P#0
P#1
P#2
P#3
op.A0-A3 op.B0-B3 op.C0-C3 op.D0-D3
P#0
P#1
P#2
P#3
A0 B0 C0 D0
A1 B1 C1 D1
A2 B2 C2 D2
A3 B3 C3 D3
All reduce
P#0
P#1
P#2
P#3
op.A0-A3 op.B0-B3 op.C0-C3 op.D0-D3
op.A0-A3 op.B0-B3 op.C0-C3 op.D0-D3
op.A0-A3 op.B0-B3 op.C0-C3 op.D0-D3
op.A0-A3 op.B0-B3 op.C0-C3 op.D0-D3

グループ通信の例(4/4)


P#0
A0 B0 C0 D0
P#1
A1 B1 C1 D1
P#2
A2 B2 C2 D2
P#3
A3 B3 C3 D3
Reduce scatter
P#0
op.A0-A3
P#1
op.B0-B3
P#2
op.C0-C3
P#3
op.D0-D3

グループ通信による計算例

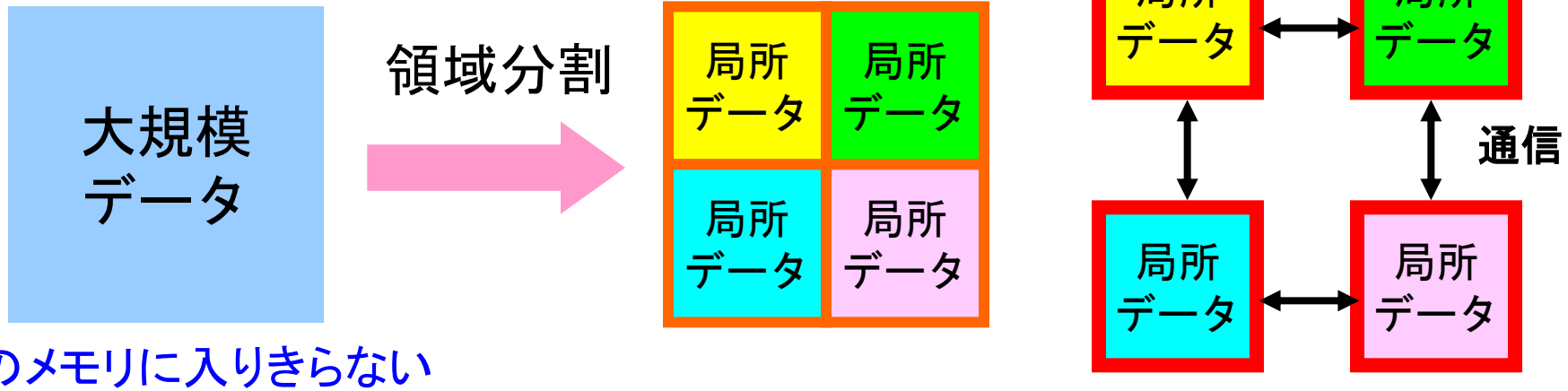
- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み

全体データと局所データ

- 大規模な全体データ（global data）を局所データ（local data）に分割して，SPMDによる並列計算を実施する場合のデータ構造について考える.

領域分割

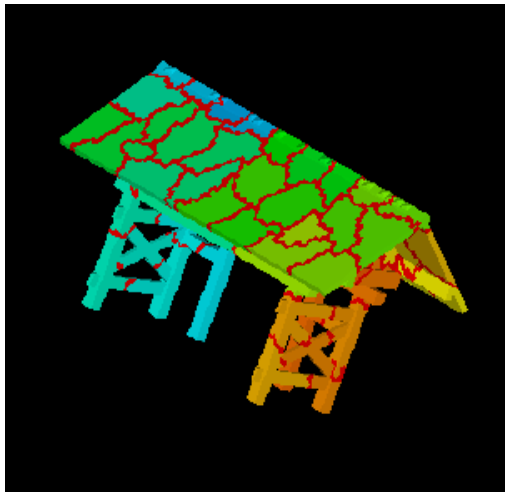
- 1GB程度のPC → 10^6メッシュが限界：FEM
  - 1000km×1000km×100kmの領域（西南日本）を1kmメッシュで切ると10^8メッシュになる
- 大規模データ → 領域分割，局所データ並列処理
- 全体系計算 → 領域間の通信が必要

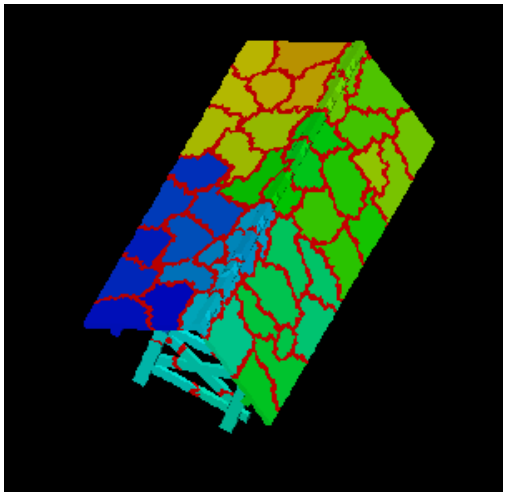


PCのメモリに入りきらない

局所データ構造

- 対象とする計算(のアルゴリズム)に適した局所データ構造を定めることが重要
  - アルゴリズム＝データ構造
- この講義の主たる目的の一つと言ってよい.





全体データと局所データ

- 大規模な全体データ（global data）を局所データ（local data）に分割して，SPMDによる並列計算を実施する場合のデータ構造について考える．

- 下記のような長さ20のベクトル，VECpとVECsの内積計算を4つのプロセッサ，プロセスで並列に実施することを考える．

```
VECp( 1)=  2
     ( 2)=  2
     ( 3)=  2
…
     (18)=  2
     (19)=  2
     (20)=  2
```

```
VECs( 1)=  3
     ( 2)=  3
     ( 3)=  3
…
     (18)=  3
     (19)=  3
     (20)=  3
```

Fortran

```
VECp[ 0]=  2
     [ 1]=  2
     [ 2]=  2
…
     [17]=  2
     [18]=  2
     [19]=  2
```

```
VECs[ 0]=  3
     [ 1]=  3
     [ 2]=  3
…
     [17]=  3
     [18]=  3
     [19]=  3
```

C

<$P-S1>/dot.f, dot.c

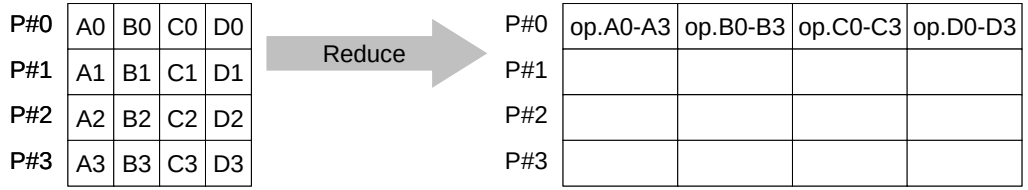
```
implicit REAL*8 (A-H,O-Z)
real(kind=8),dimension(20):: &
    VECp,  VECs

do i= 1, 20
  VECp(i)= 2.0d0
  VECs(i)= 3.0d0
enddo

sum= 0.d0
do ii= 1, 20
  sum= sum + VECp(ii)*VECs(ii)
enddo

stop
end
```

```
#include <stdio.h>
int main(){
   int i;
   double VECp[20], VECs[20]
   double sum;

   for(i=0;i<20;i++){
     VECp[i]= 2.0;
     VECs[i]= 3.0;
   }

   sum = 0.0;
   for(i=0;i<20;i++){
     sum += VECp[i] * VECs[i];
   }
   return 0;
}
```

<$P-S1>/dot.f, dot.cの逐次実行

```
>$ cd <$P-S1>

>$ cc -O3 dot.c
>$ f95 -O3 dot.f

>$ ./a.out
   1       2.00       3.00
   2       2.00       3.00
   3       2.00       3.00
…

  18       2.00       3.00
  19       2.00       3.00
  20       2.00       3.00


dot product    120.00
```

MPI_REDUCE



- コミュニケータ「comm」内の，各プロセスの送信バッファ「sendbuf」について，演算「op」を実施し，その結果を1つの受信プロセス「root」の受信バッファ「recbuf」に格納する．
  - 総和，積，最大，最小 他

- **call MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ

    Fortran MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc

  - **op** 整数 I 計算の種類

    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
    ユーザーによる定義も可能：MPI_OP_CREATE
  - **root** 整数 I 受信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

送信バッファと受信バッファ

- MPIでは「送信バッファ」,「受信バッファ」という変数がしばしば登場する.

- 送信バッファと受信バッファは必ずしも異なった名称の配列である必要はないが，必ずアドレスが異なっていなければならない.

Fortran

MPI_REDUCEの例（1/2）

```
call MPI_REDUCE
(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```

```
real(kind=8):: X0, X1

call MPI_REDUCE
(X0, X1, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0, <comm>, ierr)
```

```
real(kind=8):: X0(4), XMAX(4)

call MPI_REDUCE
(X0, XMAX, 4, MPI_DOUBLE_PRECISION, MPI_MAX, 0, <comm>, ierr)
```

各プロセスにおける，X0(i)の最大値が0番プロセスのXMAX(i)に入る（i=1~4）

Fortran

MPI_REDUCEの例(2/2)

```
call MPI_REDUCE
(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```

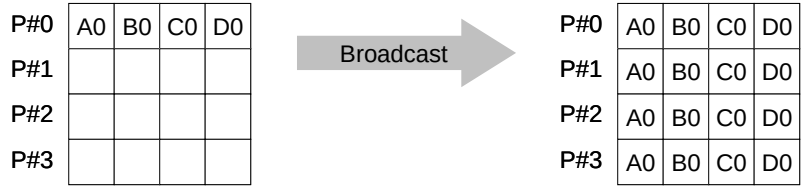
```
real(kind=8):: X0, XSUM

call MPI_REDUCE
(X0, XSUM, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

各プロセスにおける，X0の総和が0番PEのXSUMに入る.

```
real(kind=8):: X0(4)

call MPI_REDUCE
(X0(1), X0(3), 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

各プロセスにおける，
- X0(1)の総和が0番プロセスのX0(3)に入る.
- X0(2)の総和が0番プロセスのX0(4)に入る.

MPI_BCAST



- コミュニケーター「comm」内の一つの送信元プロセス「root」のバッファ「buffer」から，その他全てのプロセスのバッファ「buffer」にメッセージを送信．

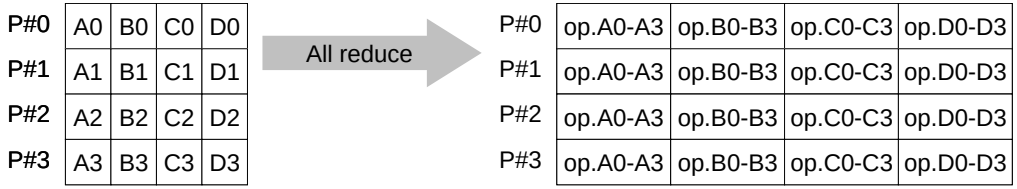
- **call MPI_BCAST (buffer,count,datatype,root,comm,ierr)**
  - **buffer** 任意 I/O バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ

    Fortran MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.

    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
  - **root** 整数 I 送信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

MPI_ALLREDUCE



- MPI_REDUCE + MPI_BCAST
- 総和，最大値を計算したら，各プロセスで利用したい場合が多い

- **call MPI_ALLREDUCE**
  (sendbuf,recvbuf,count,datatype,op, comm,ierr)
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **op** 整数 I 計算の種類
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

MPI_Reduce/Allreduceの “op”

Fortran

```
call MPI_REDUCE
(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```

- **MPI_MAX, MPI_MIN** 最大値，最小値
- **MPI_SUM, MPI_PROD** 総和，積
- **MPI_LAND** 論理AND

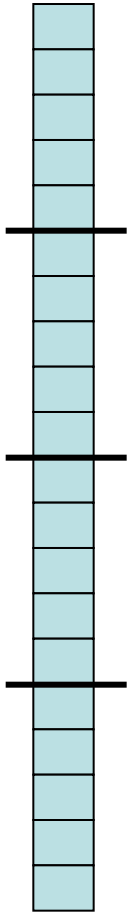
局所データの考え方(1/2)

Fortran

- 長さ20のベクトルを，4つに分割する
- 各プロセスで長さ5のベクトル(1~5)

```
VECp( 1)=  2
     ( 2)=  2
     ( 3)=  2
…
     (18)=  2
     (19)=  2
     (20)=  2
```

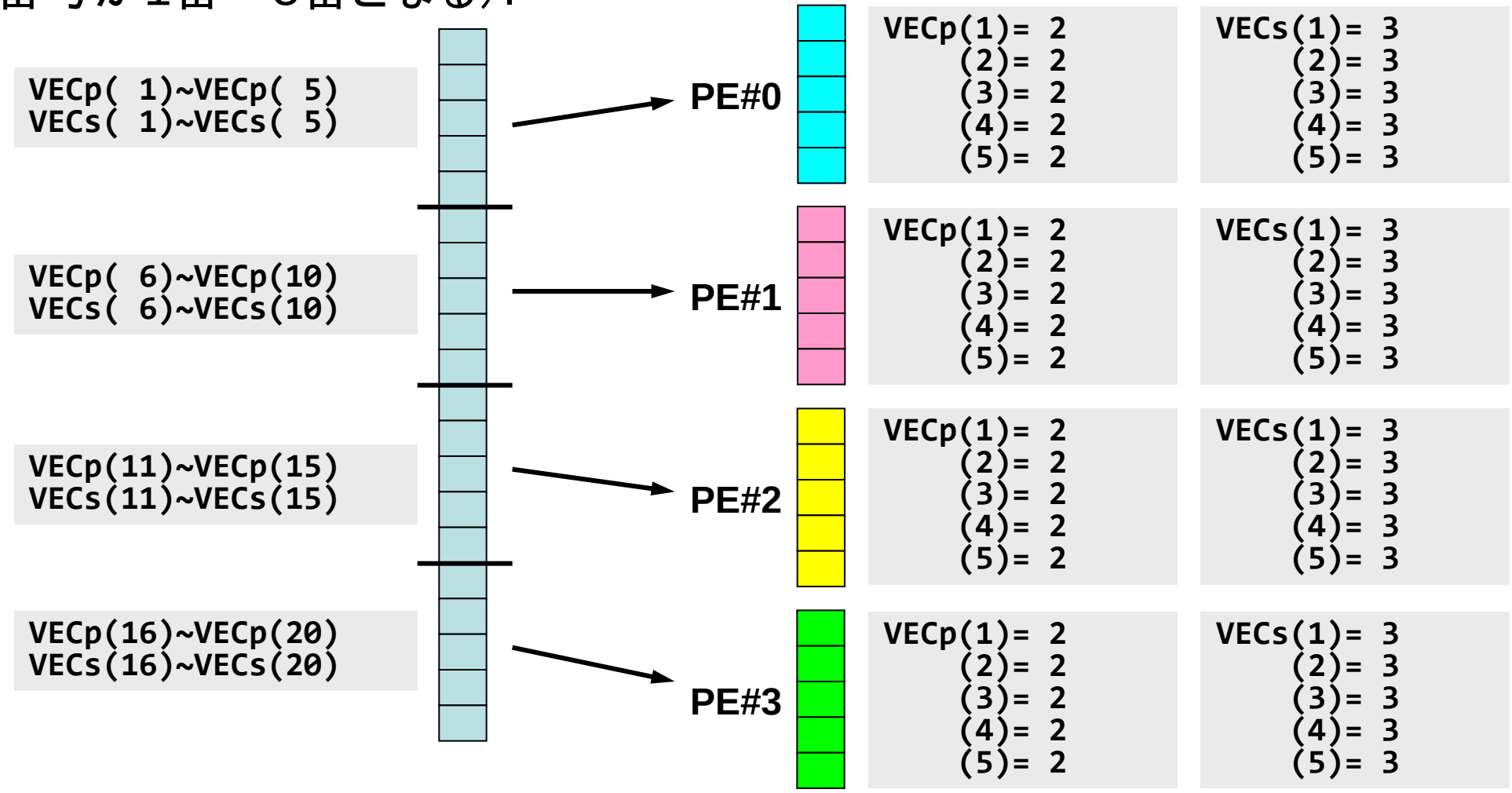
```
VECs( 1)=   3
     ( 2)=   3
     ( 3)=   3
…
     (18)=   3
     (19)=   3
     (20)=   3
```



局所データの考え方(2/2)

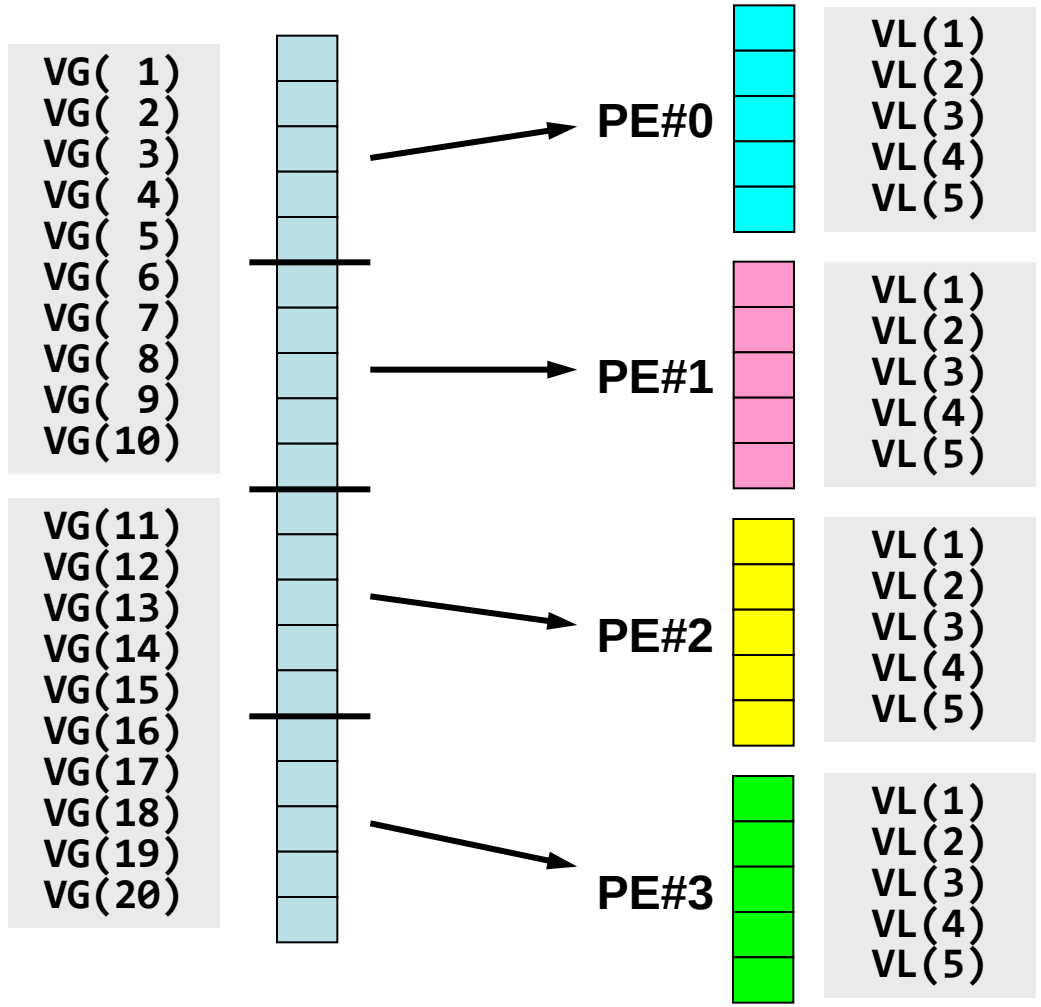
Fortran

- もとのベクトルの1～5番成分が0番PE，6~10番成分が1番PE，11～15番が2番PE，16～20番が3番PEのそれぞれ1番～5番成分となる(局所番号が1番～5番となる)．



とは言え・・・

- 全体を分割して，1から番号をふり直すだけ･･･というのはいかにも簡単である.
- もちろんこれだけでは済まない．済まない例については後半に紹介する.



内積の並列計算例(1/3)

<$P-S1>/allreduce.f

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr
real(kind=8), dimension(5) :: VECp,  VECs

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

sumA= 0.d0
sumR= 0.d0
do i= 1, 5
  VECp(i)= 2.0d0
  VECs(i)= 3.0d0
enddo

sum0= 0.d0
do i= 1, 5
  sum0= sum0 + VECp(i) * VECs(i)
enddo

if (my_rank == 0) then
  write (*,'(a)') '(my_rank, sumALLREDUCE, sumREDUCE)‘
endif
```

各ベクトルを各プロセスで独立に生成する

内積の並列計算例(2/3)

<$P-S1>/allreduce.f

```
!C
!C-- REDUCE
  call MPI_REDUCE (sum0, sumR, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, &
                   MPI_COMM_WORLD, ierr)

!C
!C-- ALL-REDUCE
  call MPI_allREDUCE (sum0, sumA, 1, MPI_DOUBLE_PRECISION, MPI_SUM, &
                      MPI_COMM_WORLD, ierr)

  write (*,'(a,i5, 2(1pe16.6))') 'before BCAST', my_rank, sumA, sumR
```

内積の計算

各プロセスで計算した結果「sum0」の総和をとる

sumR には，PE#0だけに計算結果が入る.

PE#1～PE#3は何も変わらない.

sumA には，MPI_ALLREDUCEによって全プロセスに計算結果が入る.

内積の並列計算例(3/3)

<$P-S1>/allreduce.f

```
!C
!C-- BCAST
  call MPI_BCAST (sumR, 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, &
                  ierr)
  write (*,'(a,i5, 2(1pe16.6))') 'after  BCAST', my_rank, sumA, sumR

  call MPI_FINALIZE (ierr)

  stop
  end
```

MPI_BCASTによって，PE#0以外の場合にも sumR に
計算結果が入る.

<$P-S1>/allreduce.f/c の実行例

```
$> mpifccpx -O3 allreduce.c
$> mpifrtpx -O3 allreduce.f
$> pjsub go4.sh  ← 出力先のファイル名を適当に変更してもよい

(my_rank, sumALLREDUCE, sumREDUCE)
before BCAST    0    1.200000E+02    1.200000E+02
after  BCAST    0    1.200000E+02    1.200000E+02

before BCAST    1    1.200000E+02    0.000000E+00
after  BCAST    1    1.200000E+02    1.200000E+02

before BCAST    3    1.200000E+02    0.000000E+00
after  BCAST    3    1.200000E+02    1.200000E+02

before BCAST    2    1.200000E+02    0.000000E+00
after  BCAST    2    1.200000E+02    1.200000E+02
```

グループ通信による計算例

- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み

全体データと局所データ(1/3)

- ある実数ベクトル**VECg**の各成分に実数αを加えるという，以下のような簡単な計算を，「並列化」することを考えてみよう:

```
      do i= 1, NG
        VECg(i)= VECg(i) + ALPHA
      enddo
```

```
for (i=0; i<NG; i++){
    VECg[i]= VECg[i] + ALPHA
}
```

全体データと局所データ(2/3)

- 簡単のために,
  - **NG=32**
  - **ALPHA=1000.0**
  - MPIプロセス数=4

- ベクトル**VECg**として以下のような32個の成分を持つベクトルを仮定する(<$P-S1>/a1x.all):

```
(101.0,  103.0,  105.0,  106.0,  109.0,  111.0,  121.0,  151.0,
 201.0,  203.0,  205.0,  206.0,  209.0,  211.0,  221.0,  251.0,
 301.0,  303.0,  305.0,  306.0,  309.0,  311.0,  321.0,  351.0,
 401.0,  403.0,  405.0,  406.0,  409.0,  411.0,  421.0,  451.0)
```

全体データと局所データ(3/3)

- 並列計算の方針
  ① 長さ32のベクトル**VECg**をあるプロセス(例えば0番)で読み込む.
  - 全体データ
  ② 4つのプロセスへ均等に(長さ8ずつ)割り振る.
  - 局所データ, 局所番号
  ③ 各プロセスでベクトル(長さ8)の各成分に**ALPHA**を加える.
  ④ 各プロセスの結果を再び長さ32のベクトルにまとめる.

- もちろんこの程度の規模であれば1プロセッサで計算できるのであるが・・・

Scatter/Gatherの計算（1/8）

長さ32のベクトルVECgをあるプロセス（例えば0番）で読み込む.

- プロセス0番から「全体データ」を読み込む

```
include   'mpif.h'
integer,  parameter :: NG= 32
real(kind=8), dimension(NG):: VECg

call MPI_INIT (ierr)
call MPI_COMM_SIZE (<comm>, PETOT  , ierr)
call MPI_COMM_RANK (<comm>, my_rank, ierr)

if (my_rank.eq.0) then
  open (21, file= 'a1x.all', status= 'unknown')
  do i= 1, NG
    read (21,*) VECg(i)
  enddo
  close (21)
endif
```

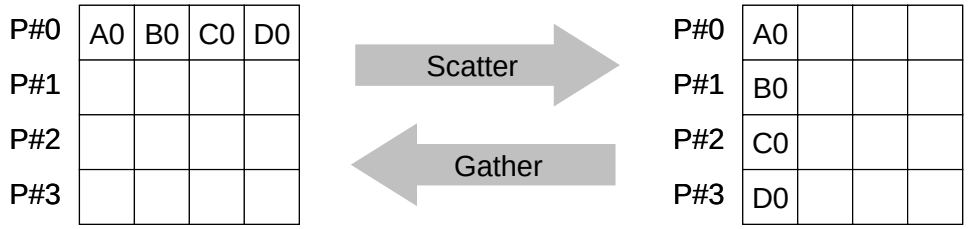
```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

int main(int argc, char **argv){
  int i, NG=32;
  int PeTot, MyRank, MPI_Comm;
  double VECg[32];
  char filename[80];
  FILE *fp;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(<comm>, &PeTot);
  MPI_Comm_rank(<comm>, &MyRank);

        fp = fopen("a1x.all", "r");
        if(!MyRank) for(i=0;i<NG;i++){
                fscanf(fp, "%lf", &VECg[i]);
        }
```

Scatter/Gatherの計算（2/8）

4つのプロセスへ均等に（長さ8ずつ）割り振る.

- MPI_Scatter の利用

MPI_SCATTER

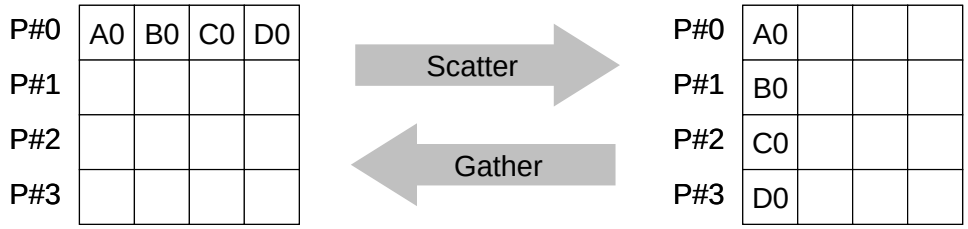


- コミュニケータ「comm」内の一つの送信元プロセス「root」の送信バッファ「sendbuf」から各プロセスに先頭から「scount」ずつのサイズのメッセージを送信し，その他全てのプロセスの受信バッファ「recvbuf」に，サイズ「rcount」のメッセージを格納.

- **`call MPI_SCATTER (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm, ierr)`**
  - **`sendbuf`** 任意 I 送信バッファの先頭アドレス，
  - **`scount`** 整数 I 送信メッセージのサイズ
  - **`sendtype`** 整数 I 送信メッセージのデータタイプ
  - **`recvbuf`** 任意 O 受信バッファの先頭アドレス，
  - **`rcount`** 整数 I 受信メッセージのサイズ
  - **`recvtype`** 整数 I 受信メッセージのデータタイプ
  - **`root`** 整数 I 送信プロセスのID（ランク）
  - **`comm`** 整数 I コミュニケータを指定する
  - **`ierr`** 整数 O 完了コード

Fortran

MPI_SCATTER（続き）



- **call MPI_SCATTER (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm, ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **scount** 整数 I 送信メッセージのサイズ
  - **sendtype** 整数 I 送信メッセージのデータタイプ
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
  - **rcount** 整数 I 受信メッセージのサイズ
  - **recvtype** 整数 I 受信メッセージのデータタイプ
  - **root** 整数 I 送信プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード
- 通常は
  - **scount = rcount**
  - **sendtype= recvtype**
- この関数によって，プロセスroot番のsendbuf（送信バッファ）の先頭アドレスからscount個ずつの成分が，commで表されるコミュニケータを持つ各プロセスに送信され，recvbuf（受信バッファ）のrcount個の成分として受信される．

Fortran

Scatter/Gatherの計算（3/8）

4つのプロセスへ均等に（長さ8ずつ）割り振る.

- 各プロセスにおいて長さ8の受信バッファ「**VEC**」（＝局所データ）を定義しておく.
- プロセス0番から送信される送信バッファ「**VECg**」の8個ずつの成分が，4つの各プロセスにおいて受信バッファ「**VEC**」の1番目から8番目の成分として受信される
- **N=8** として引数は下記のようになる：

```
integer, parameter :: N =  8
real(kind=8), dimension(N ) :: VEC
...
call MPI_Scatter                              &
     (VECg, N, MPI_DOUBLE_PRECISION, &
      VEC , N, MPI_DOUBLE_PRECISION, &
      0, <comm>, ierr)
```

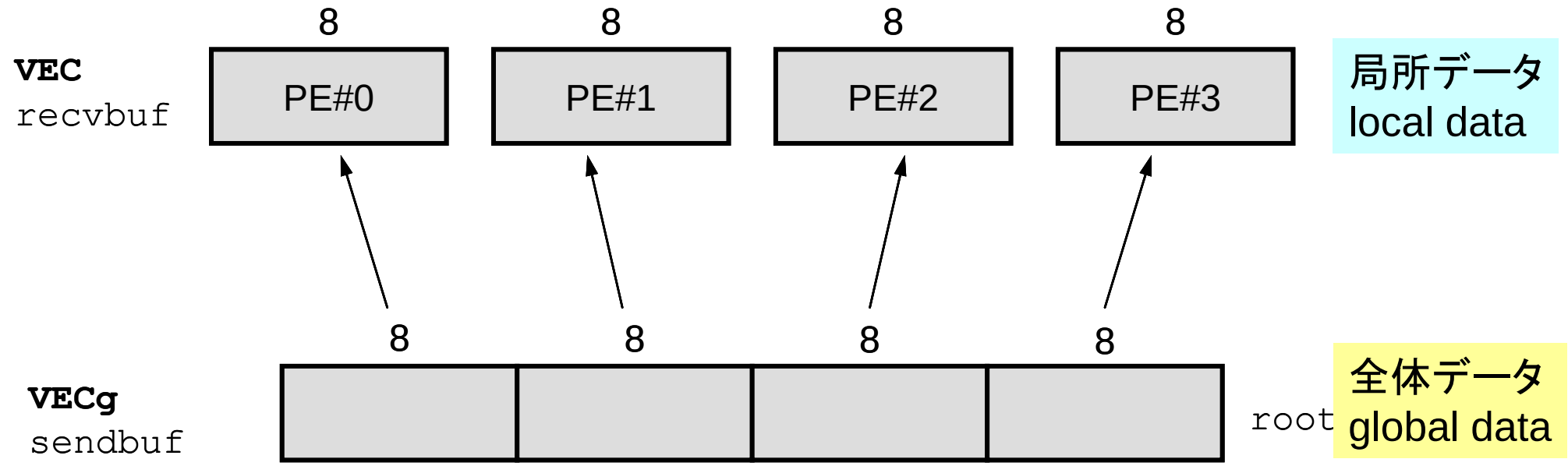
```
int  N=8;
double VEC [8];
...
MPI_Scatter (&VECg, N, MPI_DOUBLE, &VEC, N,
MPI_DOUBLE, 0, <comm>);
```

```
call MPI_SCATTER
(sendbuf, scount, sendtype, recvbuf, rcount,
 recvtype, root, comm, ierr)
```

Scatter/Gatherの計算（4/8）

4つのプロセスへ均等に（長さ8ずつ）割り振る.

- rootプロセス（0番）から各プロセスへ8個ずつの成分がscatterされる.
- **VECg**の1番目から8番目の成分が0番プロセスにおける**VEC**の1番目から8番目，9番目から16番目の成分が1番プロセスにおける**VEC**の1番目から8番目という具合に格納される.
  - **VECg**：全体データ，**VEC**：局所データ



Scatter/Gatherの計算（5/8）

4つのプロセスへ均等に（長さ8ずつ）割り振る.

- 全体データ（global data）としては**VECg**の1番から32番までの要素番号を持っていた各成分が，それぞれのプロセスにおける局所データ（local data）としては，**VEC**の1番から8番までの局所番号を持った成分として格納される. **VEC**の成分を各プロセスごとに書き出してみると:

```
do i= 1, N
  write (*,'(a, 2i8,f10.0)') 'before',  my_rank, i, VEC(i)
enddo
```

```
for(i=0;i<N;i++){
printf("before %5d %5d %10.0F¥n", MyRank, i+1, VEC[i]);}
```

Scatter/Gatherの計算（5/8）

4つのプロセスへ均等に（長さ8ずつ）割り振る.

- 全体データ（global data）としては**VECg**の1番から32番までの要素番号を持っていた各成分が，それぞれのプロセスにおける局所データ（local data）としては，**VEC**の1番から8番までの局所番号を持った成分として格納される. **VEC**の成分を各プロセスごとに書き出してみると:

PE#0
```
before 0 1   101.
before 0 2   103.
before 0 3   105.
before 0 4   106.
before 0 5   109.
before 0 6   111.
before 0 7   121.
before 0 8   151.
```

PE#1
```
before 1 1   201.
before 1 2   203.
before 1 3   205.
before 1 4   206.
before 1 5   209.
before 1 6   211.
before 1 7   221.
before 1 8   251.
```

PE#2
```
before 2 1   301.
before 2 2   303.
before 2 3   305.
before 2 4   306.
before 2 5   309.
before 2 6   311.
before 2 7   321.
before 2 8   351.
```

PE#3
```
before 3 1   401.
before 3 2   403.
before 3 3   405.
before 3 4   406.
before 3 5   409.
before 3 6   411.
before 3 7   421.
before 3 8   451.
```

Scatter/Gatherの計算（6/8）

各プロセスでベクトル（長さ8）の各成分にALPHAを加える

- 各プロセスでの計算は，以下のようになる：

```
real(kind=8), parameter :: ALPHA= 1000.
do i= 1, N
  VEC(i)= VEC(i) + ALPHA
enddo
```

```
double ALPHA=1000.;
...
for(i=0;i<N;i++){
  VEC[i]= VEC[i] + ALPHA;}
```

- 計算結果は以下のようになる：

PE#0
```
after 0 1  1101.
after 0 2  1103.
after 0 3  1105.
after 0 4  1106.
after 0 5  1109.
after 0 6  1111.
after 0 7  1121.
after 0 8  1151.
```

PE#1
```
after 1 1  1201.
after 1 2  1203.
after 1 3  1205.
after 1 4  1206.
after 1 5  1209.
after 1 6  1211.
after 1 7  1221.
after 1 8  1251.
```

PE#2
```
after 2 1  1301.
after 2 2  1303.
after 2 3  1305.
after 2 4  1306.
after 2 5  1309.
after 2 6  1311.
after 2 7  1321.
after 2 8  1351.
```

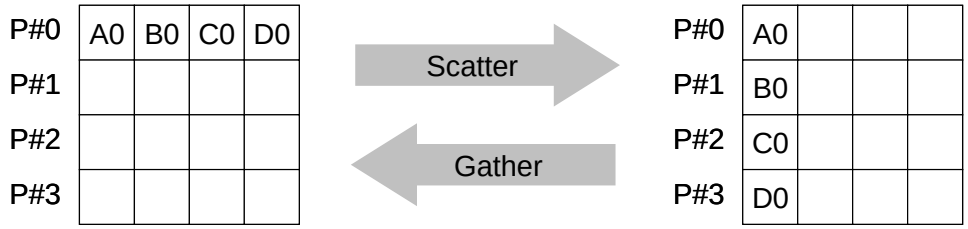
PE#3
```
after 3 1  1401.
after 3 2  1403.
after 3 3  1405.
after 3 4  1406.
after 3 5  1409.
after 3 6  1411.
after 3 7  1421.
after 3 8  1451.
```

Scatter/Gatherの計算（7/8）

各プロセスの結果を再び長さ32のベクトルにまとめる

- これには，MPI_Scatter と丁度逆の MPI_Gather という関数が用意されている.

MPI_GATHER



- MPI_SCATTERの逆

- **call MPI_GATHER (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm, ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス,
  - **scount** 整数 I 送信メッセージのサイズ
  - **sendtype** 整数 I 送信メッセージのデータタイプ
  - **recvbuf** 任意 O 受信バッファの先頭アドレス,
  - **rcount** 整数 I 受信メッセージのサイズ
  - **recvtype** 整数 I 受信メッセージのデータタイプ
  - **root** 整数 I 受信プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

- ここで，受信バッファ recvbuf の値はroot番のプロセスに集められる.

Fortran

Scatter/Gatherの計算（8/8）

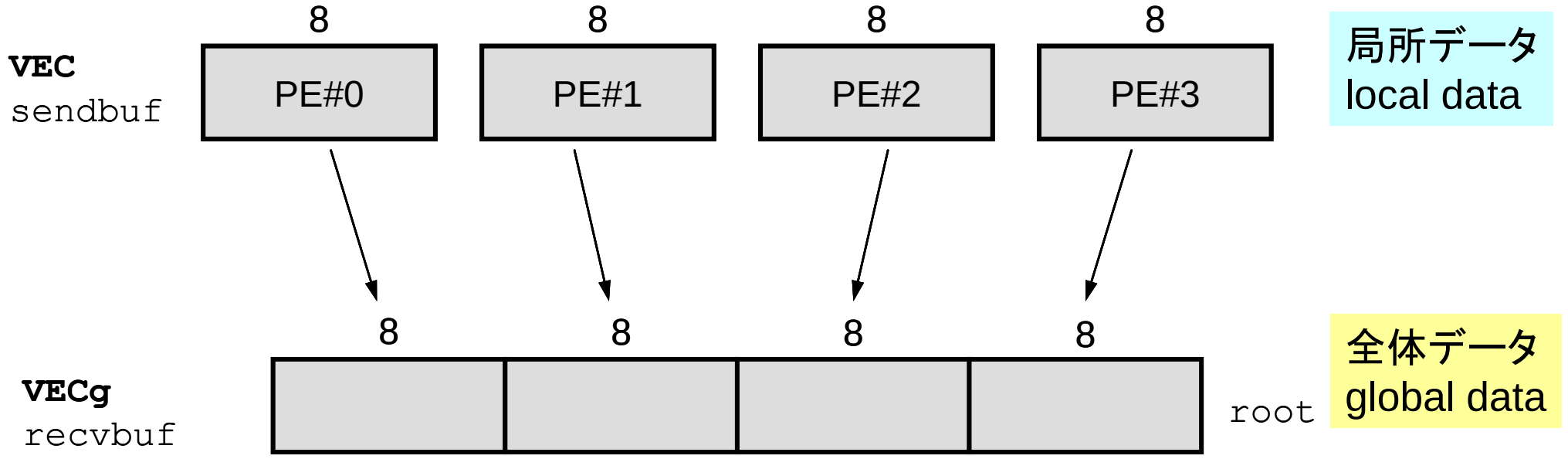
各プロセスの結果を再び長さ32のベクトルにまとめる

- 本例題の場合，root=0として，各プロセスから送信される**VEC**の成分を0番プロセスにおいて**VECg**として受信するものとすると以下のようになる:

```
call MPI_Gather                                    &
     (VEC , N, MPI_DOUBLE_PRECISION, &
      VECg, N, MPI_DOUBLE_PRECISION, &
      0, <comm>, ierr)
```

```
MPI_Gather (&VEC, N, MPI_DOUBLE, &VECg, N,
MPI_DOUBLE, 0, <comm>);
```

- 各プロセスから8個ずつの成分がrootプロセスへgatherされる



<$P-S1>/scatter-gather.f/c 実行例

```
$> mpifccpx -Kfast scatter-gather.c
$> mpifrtpx -Kfast scatter-gather.f
$> pjsub go4.sh ← 出力先のファイル名を適当に変更してもよい
```

PE#0
```
before 0 1   101.
before 0 2   103.
before 0 3   105.
before 0 4   106.
before 0 5   109.
before 0 6   111.
before 0 7   121.
before 0 8   151.
```

PE#1
```
before 1 1   201.
before 1 2   203.
before 1 3   205.
before 1 4   206.
before 1 5   209.
before 1 6   211.
before 1 7   221.
before 1 8   251.
```

PE#2
```
before 2 1   301.
before 2 2   303.
before 2 3   305.
before 2 4   306.
before 2 5   309.
before 2 6   311.
before 2 7   321.
before 2 8   351.
```

PE#3
```
before 3 1   401.
before 3 2   403.
before 3 3   405.
before 3 4   406.
before 3 5   409.
before 3 6   411.
before 3 7   421.
before 3 8   451.
```

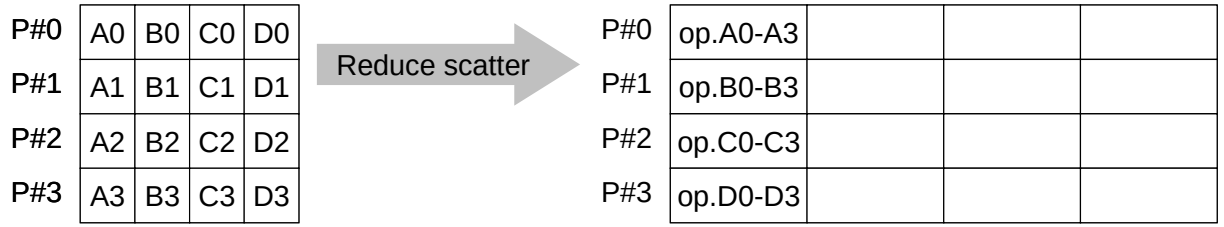
PE#0
```
after 0 1   1101.
after 0 2   1103.
after 0 3   1105.
after 0 4   1106.
after 0 5   1109.
after 0 6   1111.
after 0 7   1121.
after 0 8   1151.
```

PE#1
```
after 1 1   1201.
after 1 2   1203.
after 1 3   1205.
after 1 4   1206.
after 1 5   1209.
after 1 6   1211.
after 1 7   1221.
after 1 8   1251.
```

PE#2
```
after 2 1   1301.
after 2 2   1303.
after 2 3   1305.
after 2 4   1306.
after 2 5   1309.
after 2 6   1311.
after 2 7   1321.
after 2 8   1351.
```

PE#3
```
after 3 1   1401.
after 3 2   1403.
after 3 3   1405.
after 3 4   1406.
after 3 5   1409.
after 3 6   1411.
after 3 7   1421.
after 3 8   1451.
```

MPI_REDUCE_SCATTER

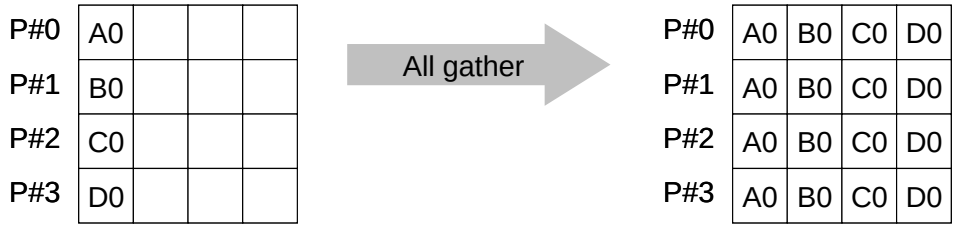


- MPI_REDUCE + MPI_SCATTER

- **call MPI_REDUCE_SCATTER (sendbuf, recvbuf, rcount, datatype, op, comm, ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス,
  - **recvbuf** 任意 O 受信バッファの先頭アドレス,
  - **rcount** 整数 I 受信メッセージのサイズ(配列:サイズ＝プロセス数)
  - **datatype** 整数 I メッセージのデータタイプ
  - **op** 整数 I 計算の種類
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

MPI_ALLGATHER

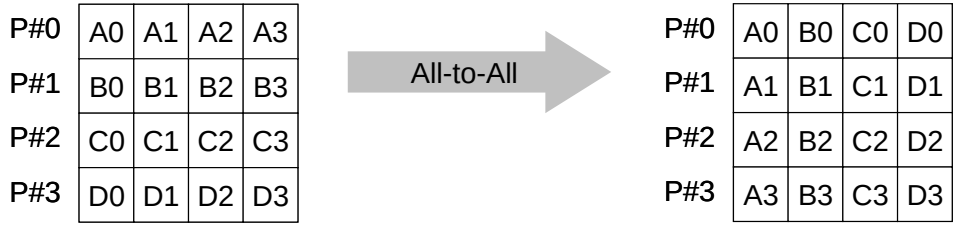


- MPI_GATHER＋MPI_BCAST
  - Gatherしたものを，全てのPEにBCASTする（各プロセスで同じデータを持つ）

- **call MPI_ALLGATHER (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, comm, ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **scount** 整数 I 送信メッセージのサイズ
  - **sendtype** 整数 I 送信メッセージのデータタイプ
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
  - **rcount** 整数 I 受信メッセージのサイズ
  - **recvtype** 整数 I 受信メッセージのデータタイプ
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

MPI_ALLTOALL



- MPI_ALLGATHERの更なる拡張：転置

- **call MPI_ALLTOALL (sendbuf, scount, sendtype, recvbuf, rcount, recvrype, comm, ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **scount** 整数 I 送信メッセージのサイズ
  - **sendtype** 整数 I 送信メッセージのデータタイプ
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
  - **rcount** 整数 I 受信メッセージのサイズ
  - **recvtype** 整数 I 受信メッセージのデータタイプ
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

グループ通信による計算例

- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み

分散ファイルを使用したオペレーション

- Scatter/Gatherの例では，PE#0から全体データを読み込み，それを全体にScatterして並列計算を実施した．

- 問題規模が非常に大きい場合，1つのプロセッサで全てのデータを読み込むことは不可能な場合がある．
  - 最初から分割しておいて，「局所データ」を各プロセッサで独立に読み込む．
  - あるベクトルに対して，全体操作が必要になった場合は，状況に応じてMPI_Gatherなどを使用する

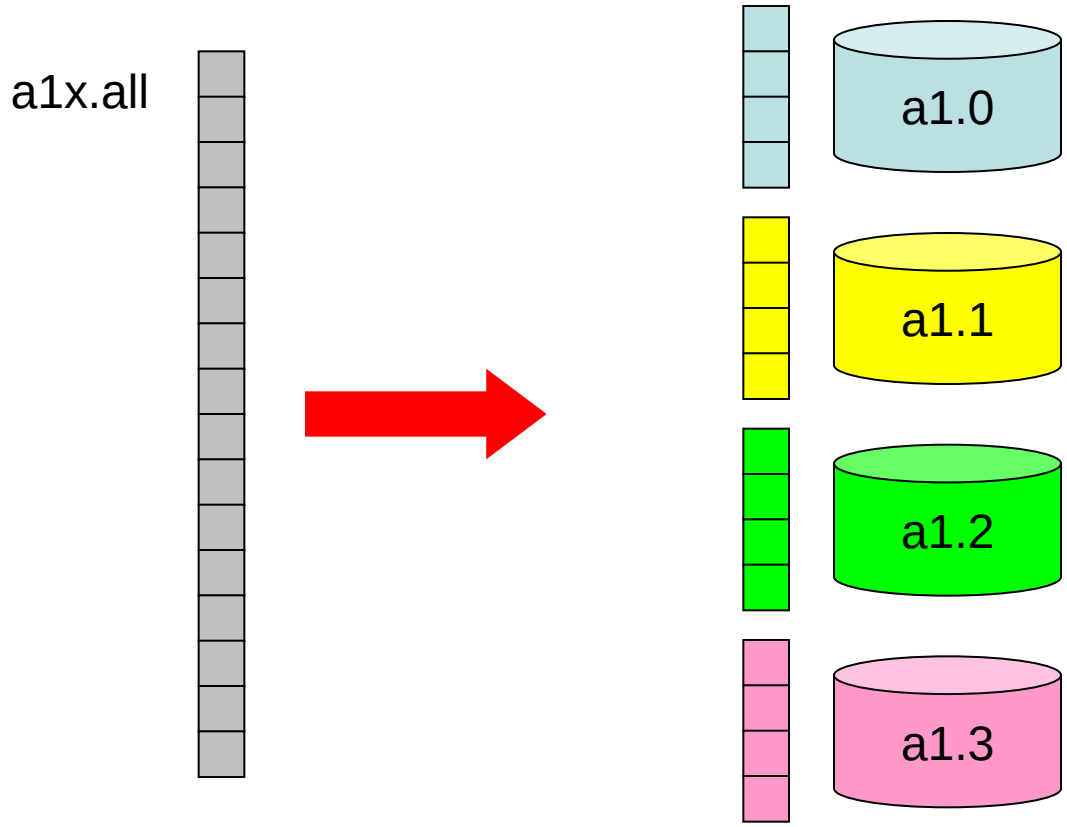
分散ファイル読み込み：等データ長（1/2）

```
>$ cd <$P-S1>
>$ ls a1.*
   a1.0 a1.1 a1.2 a1.3

>$ mpifccpx -Kfast file.c
>$ mpifrtpx -Kfast file.f

>$ pjsub go4.sh
```

「a1x.all」を4つに分割したもの

分散ファイルの操作

- 「a1.0~a1.3」は全体ベクトル「a1x.all」を領域に分割したもの，と考えることができる．



分散ファイル読み込み：等データ長（2/2）

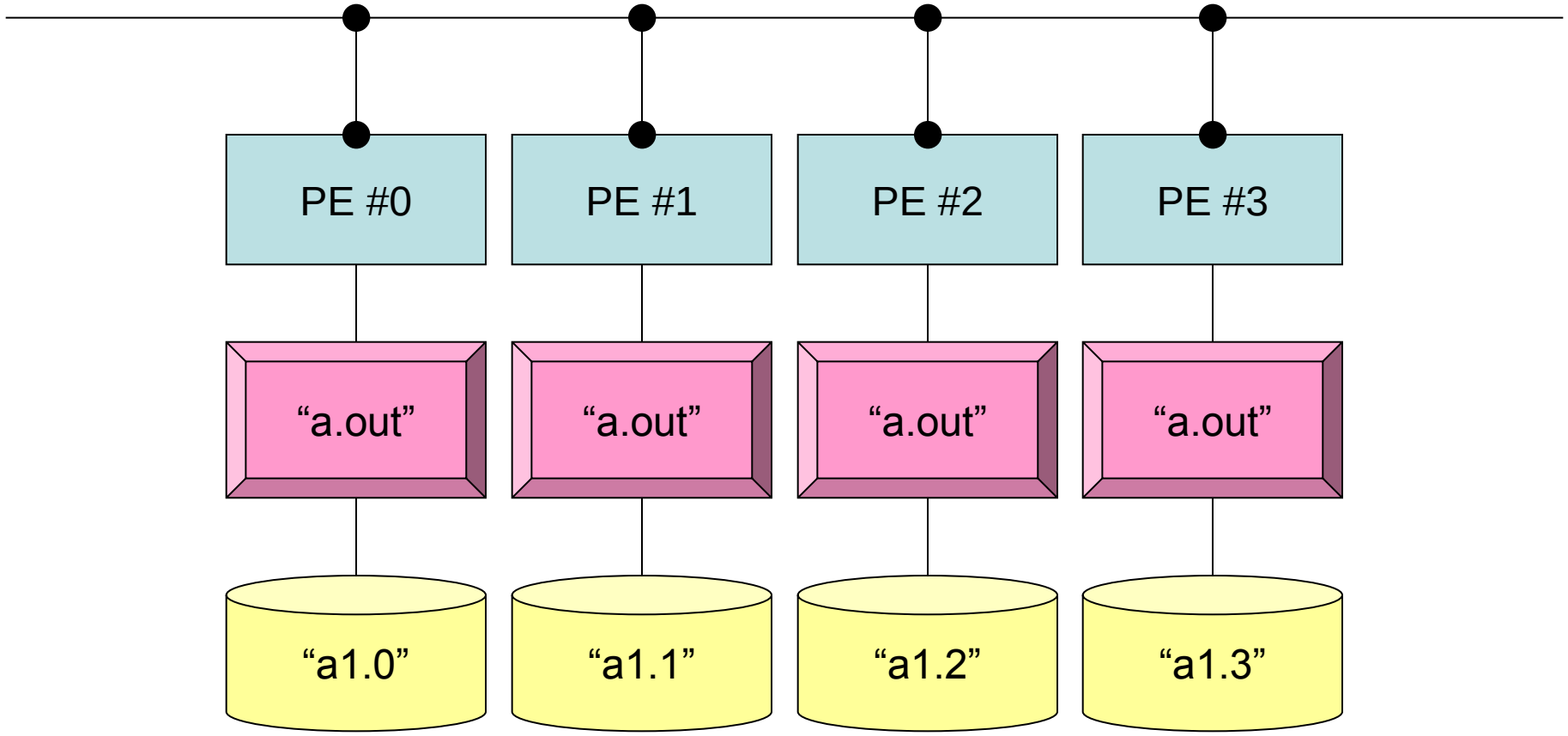
<$P-S1>/file.f

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr
real(kind=8), dimension(8) :: VEC
character(len=80)          :: filename

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

if (my_rank.eq.0) filename= 'a1.0'
if (my_rank.eq.1) filename= 'a1.1'
if (my_rank.eq.2) filename= 'a1.2'
if (my_rank.eq.3) filename= 'a1.3'

open (21, file= filename, status= 'unknown')
  do i= 1, 8
    read (21,*) VEC(i)
  enddo
close (21)

call MPI_FINALIZE (ierr)

stop
end
```

Hello とそんなに変わらない

「局所番号（1~8）」で読み込む

SPMDの典型例




```
mpiexec -np 4 a.out
```

分散ファイル読み込み：可変長（1/2）

ファイル内のデータ数が均等でない場合はどうするか？

```
>$ cd <$P-S1>
>$ ls a2.*
   a2.0 a2.1 a2.2 a2.3
>$ cat a2.1
  5          ← 各PEにおける成分数
  201.0      ← 成分の並び
  203.0
  205.0
  206.0
  209.0

>$ mpifccpx -Kfast file2.c
>$ mpifrtpx -Kfast file2.f

>$ pjsub go4.sh
```

分散ファイルの読み込み：可変長（2/2）

<$P-S1>/file2.f

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr
real(kind=8), dimension(:), allocatable :: VEC
character(len=80)                       :: filename

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

 if (my_rank.eq.0) filename= 'a2.0'
 if (my_rank.eq.1) filename= 'a2.1'
 if (my_rank.eq.2) filename= 'a2.2'
 if (my_rank.eq.3) filename= 'a2.3'

 open (21, file= filename, status= 'unknown')
   read (21,*) N
   allocate (VEC(N))
   do i= 1, N
     read (21,*) VEC(i)
   enddo
 close(21)

call MPI_FINALIZE (ierr)
stop
end
```

Nが各データ（プロセッサ）で異なる

局所データの作成法

- 全体データ（N=NG）を入力
  - Scatterして各プロセスに分割
  - 各プロセスで演算
  - 必要に応じて局所データをGather（またはAllgather）して全体データを生成
- 局所データ（N=NL）を生成，あるいは（あらかじめ分割生成して）入力
  - 各プロセスで局所データを生成，あるいは入力
  - 各プロセスで演算
  - 必要に応じて局所データをGather（またはAllgather）して全体データを生成
- 将来的には後者が中心となるが，全体的なデータの動きを理解するために，しばらくは前者についても併用

グループ通信による計算例

- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み
- MPI_Allgatherv

MPI_GATHERV, MPI_SCATTERV

- これまで紹介してきた，MPI_GATHETR, MPI_SCATTERなどは，各プロセッサからの送信，受信メッセージが均等な場合.

- 末尾に「V」が付くと，各ベクトルが可変長さの場合となる.
  - MPI_GATHERV
  - MPI_SCATTERV
  - MPI_ALLGATHERV
  - MPI_ALLTOALLV

Fortran

MPI_ALLGATHERV

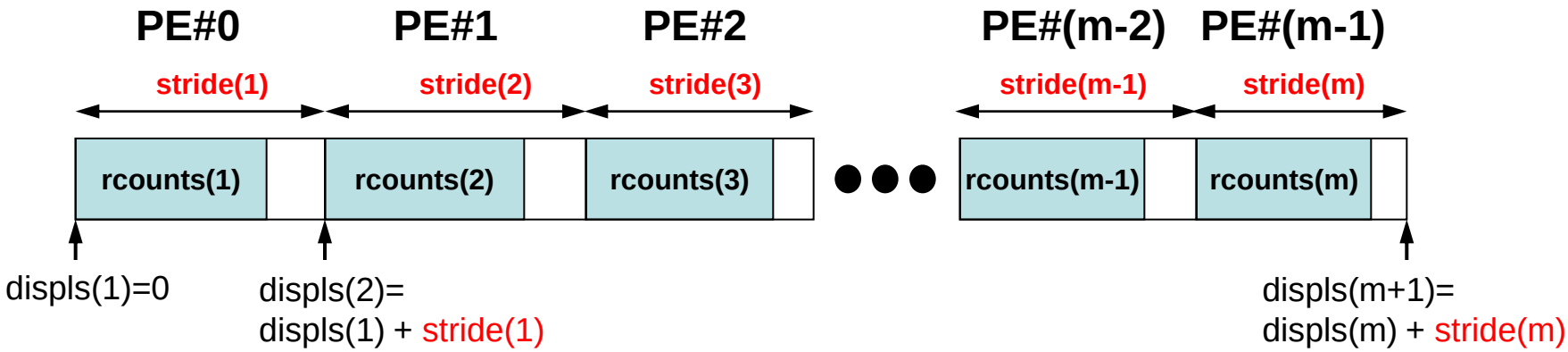
- MPI_ALLGATHER の可変長さベクトル版
  - 「局所データ」から「全体データ」を生成する

- **call MPI_ALLGATHERV (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm, ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **scount** 整数 I 送信メッセージのサイズ
  - **sendtype** 整数 I 送信メッセージのデータタイプ
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
  - **rcounts** **整数 I 受信メッセージのサイズ（配列：サイズ＝PETOT）**
  - **displs** **整数 I 受信メッセージのインデックス（配列：サイズ＝PETOT+1）**
  - **recvtype** 整数 I 受信メッセージのデータタイプ
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

MPI_ALLGATHERV（続き）

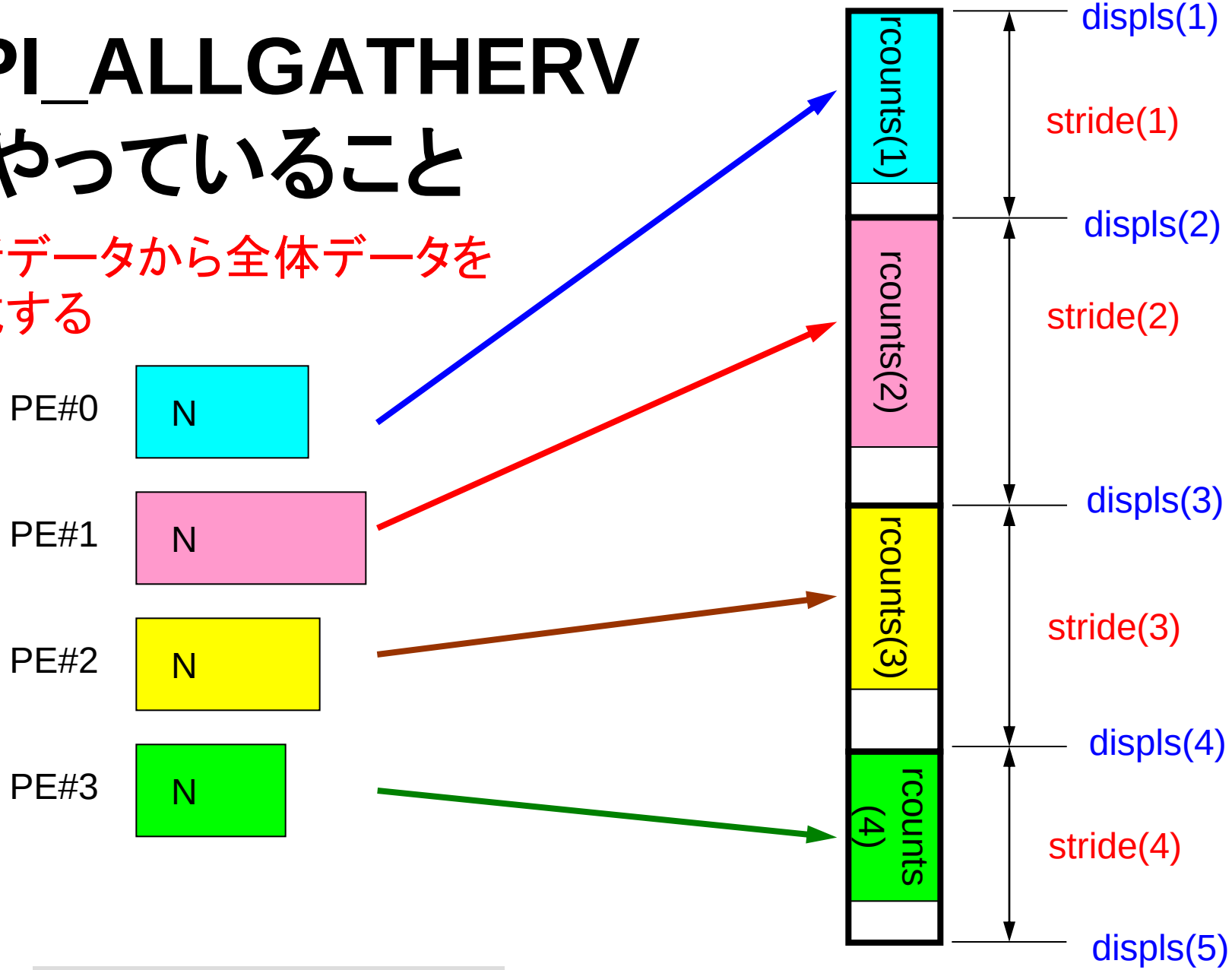
- `call MPI_ALLGATHERV (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm, ierr)`
  - `rcounts` 整数 I 受信メッセージのサイズ（配列：サイズ＝PETOT）
  - `displs` 整数 I 受信メッセージのインデックス（配列：サイズ＝PETOT+1）
  - この2つの配列は，最終的に生成される「全体データ」のサイズに関する配列であるため，各プロセスで配列の全ての値が必要になる：
    - もちろん各プロセスで共通の値を持つ必要がある．
  - 通常は`stride(i)=rcounts(i)`



size(recvbuf)= displs(PETOT+1)= sum(stride)

MPI_ALLGATHERV でやっていること

局所データから全体データを生成する

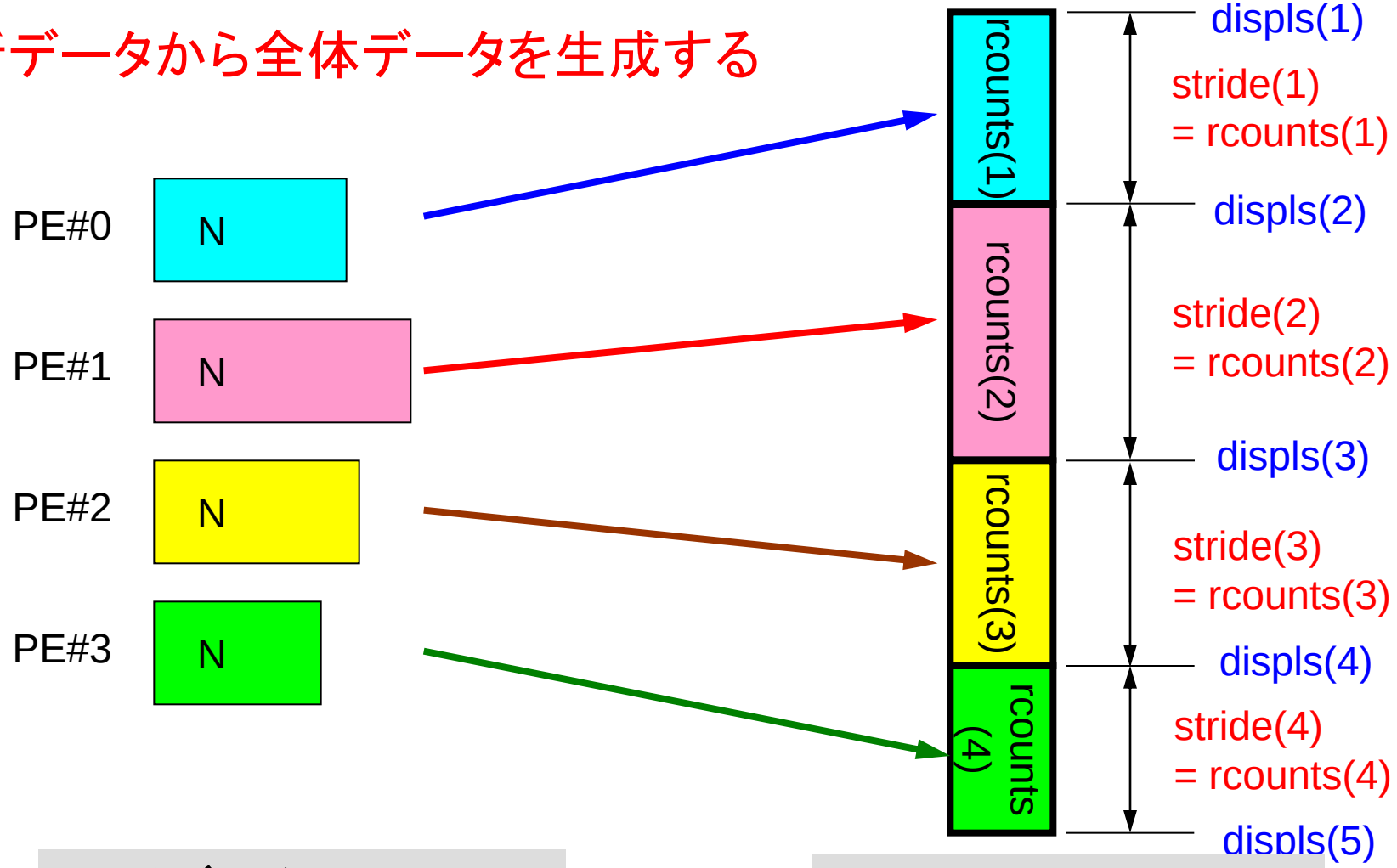


局所データ：sendbuf

全体データ：recvbuf

MPI_ALLGATHERV でやっていること

局所データから全体データを生成する



局所データ：sendbuf

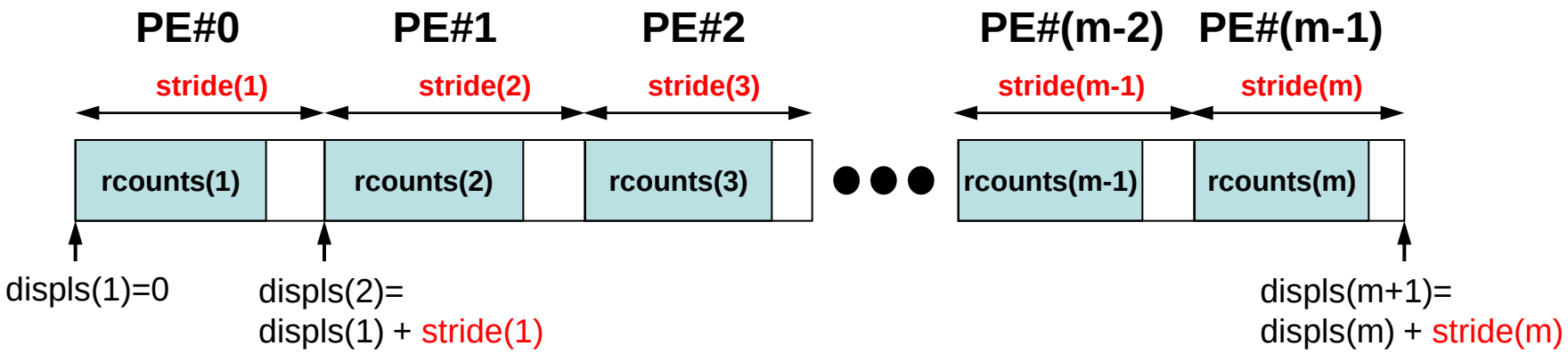
全体データ：recvbuf

MPI_ALLGATHERV詳細(1/2)

Fortran

- call MPI_ALLGATHERV (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm, ierr)
  - rcounts 整数 I 受信メッセージのサイズ(配列:サイズ=PETOT)
  - displs 整数 I 受信メッセージのインデックス(配列:サイズ=PETOT+1)

- rcounts
  - 各PEにおけるメッセージサイズ:局所データのサイズ
- displs
  - 各局所データの全体データにおけるインデックス
  - displs(PETOT+1)が全体データのサイズ

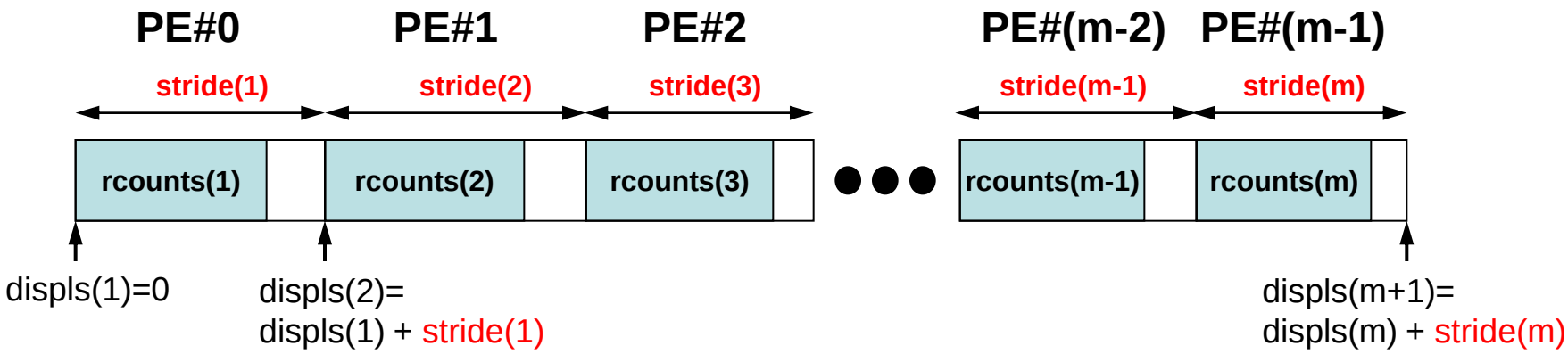


size(recvbuf)= displs(PETOT+1)= sum(stride)

MPI_ALLGATHERV詳細(2/2)

Fortran

- rcountsとdisplsは各プロセスで共通の値が必要
  - 各プロセスのベクトルの大きさ N をallgatherして，rcountsに相当するベクトルを作る.
  - rcountsから各プロセスにおいてdisplsを作る(同じものができる).
    - stride(i)= rcounts(i) とする
  - rcountsの和にしたがってrecvbufの記憶領域を確保する.



size(recvbuf)= displs(PETOT+1)= sum(stride)

MPI_ALLGATHERV使用準備 例題：<$P-S1>/agv.f，<$P-S1>/agv.c

- “a2.0”~”a2.3”から，全体ベクトルを生成する.
- 各ファイルのベクトルのサイズが，8,5,7,3であるから，長さ23（=8+5+7+3）のベクトルができることになる.

a2.0~a2.3

PE#0

```
   8
101.0
103.0
105.0
106.0
109.0
111.0
121.0
151.0
```

PE#1

```
   5
201.0
203.0
205.0
206.0
209.0
```

PE#2

```
   7
301.0
303.0
305.0
306.0
311.0
321.0
351.0
```

PE#3

```
   3
401.0
403.0
405.0
```

MPI_ALLGATHERV 使用準備(1/4)

<$P-S1>/agv.f

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'

integer :: PETOT, my_rank, SOLVER_COMM, ierr
real(kind=8), dimension(:), allocatable :: VEC
real(kind=8), dimension(:), allocatable :: VEC2
real(kind=8), dimension(:), allocatable :: VECg
integer(kind=4), dimension(:), allocatable :: rcounts
integer(kind=4), dimension(:), allocatable :: displs
character(len=80)                          :: filename

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

if (my_rank.eq.0) filename= 'a2.0'
if (my_rank.eq.1) filename= 'a2.1'
if (my_rank.eq.2) filename= 'a2.2'
if (my_rank.eq.3) filename= 'a2.3'

open (21, file= filename, status= 'unknown')
  read (21,*) N
  allocate (VEC(N))
  do i= 1, N
    read (21,*) VEC(i)
  enddo
```

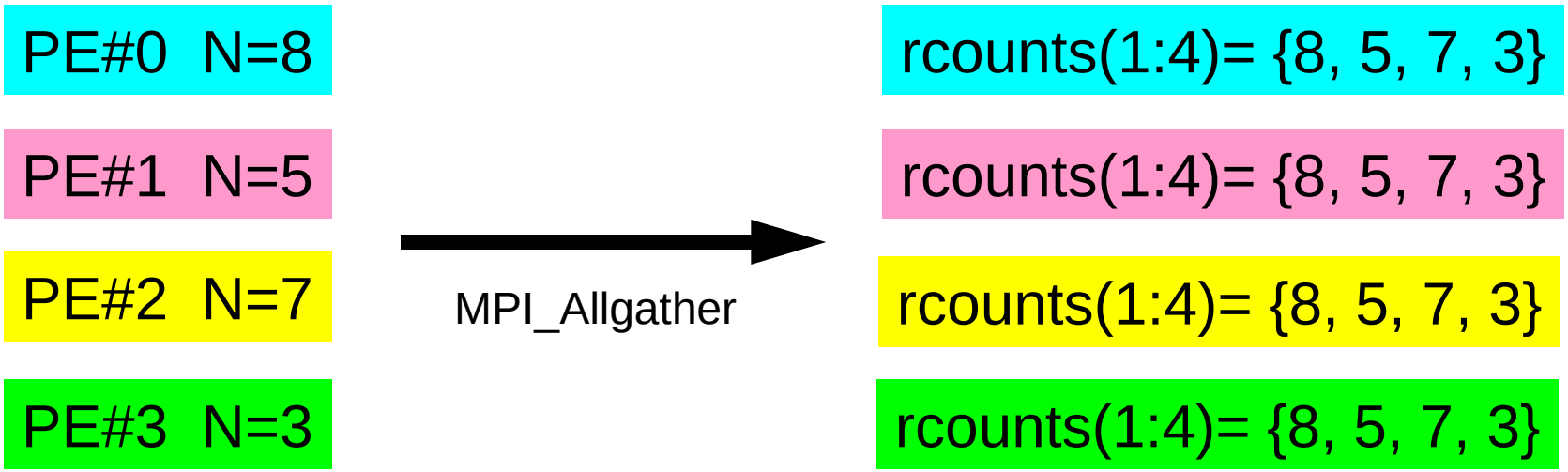
N(NL)の値が各PEで異なることに注意

MPI_ALLGATHERV 使用準備(2/4)

<$P-S1>/agv.f

```
 allocate (rcounts(PETOT), displs(PETOT+1))
 rcounts= 0
 write (*,'(a,10i8)') "before", my_rank, N, rcounts

 call MPI_allGATHER ( N     , 1, MPI_INTEGER,                    &
&                     rcounts, 1, MPI_INTEGER,                    &
&                     MPI_COMM_WORLD, ierr)

 write (*,'(a,10i8)') "after ", my_rank, N, rcounts
 displs(1)= 0
```

各PEにrcountsを生成



MPI_ALLGATHERV 使用準備(2/4)

<$P-S1>/agv.f

```
 allocate (rcounts(PETOT), displs(PETOT+1))
 rcounts= 0
 write (*,'(a,10i8)') "before", my_rank, N, rcounts

 call MPI_allGATHER ( N     , 1, MPI_INTEGER,                    &
&                     rcounts, 1, MPI_INTEGER,                    &
&                     MPI_COMM_WORLD, ierr)

 write (*,'(a,10i8)') "after ", my_rank, N, rcounts
 displs(1)= 0

 do ip= 1, PETOT
   displs(ip+1)= displs(ip) + rcounts(ip)
 enddo

 write (*,'(a,10i8)') "displs", my_rank, displs

 call MPI_FINALIZE (ierr)

 stop
 end
```

各PEにrcountsを生成

各PEでdisplsを生成

MPI_ALLGATHERV 使用準備(3/4)

```
> mpifrtpx -Kfast agv.f
> mpifccpx -Kfast agv.c
> pjsub go4.sh

before       0       8       0       0       0       0
after        0       8       8       5       7       3
displs       0               0       8      13      20      23

before       1       5       0       0       0       0
after        1       5       8       5       7       3
displs       1               0       8      13      20      23

before       3       3       0       0       0       0
after        3       3       8       5       7       3
displs       3               0       8      13      20      23

before       2       7       0       0       0       0
after        2       7       8       5       7       3
displs       2               0       8      13      20      23
```

```
write (*,‘(a,10i8)’) “before”, my_rank, N, rcounts
write (*,'(a,10i8)') "after ", my_rank, N, rcounts
write (*,'(a,i8,8x,10i8)') "displs", my_rank, displs
```

MPI_ALLGATHERV 使用準備(4/4)

- 引数で定義されていないのは「recvbuf」だけ.
- サイズは・・・「displs(PETOT+1)」
  - 各PEで,「**allocate (recvbuf(displs(PETOT+1))**」のようにして記憶領域を確保する

```
call MPI_allGATHERv
     ( VEC , N, MPI_DOUBLE_PRECISION,
       recvbuf, rcounts, displs, MPI_DOUBLE_PRECISION,
       MPI_COMM_WORLD, ierr)
```

課題S1（1/2）

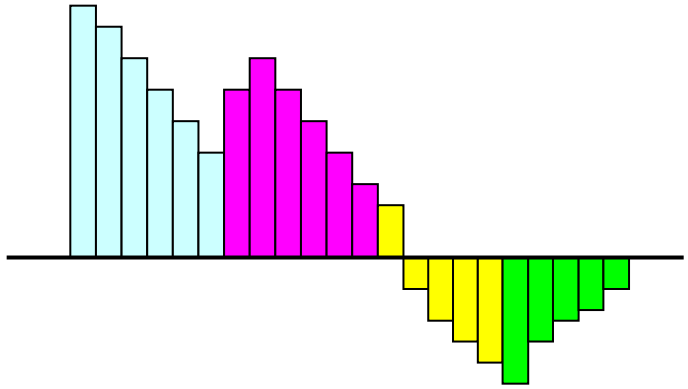
- 「<$P-S1>/a1.0～a1.3」，「<$P-S1>/a2.0~a2.3」から局所ベクトル情報を読み込み，全体ベクトルのノルム（||x||）を求めるプログラムを作成する（S1-1）.
  - ノルム ||x|| は，各要素の2乗の和の平方根である.
  - <$P-S1>file.f，<$T-S1>file2.fをそれぞれ参考にする.

- 「<$P-S1>/a2.0~a2.3」から局所ベクトル情報を読み込み，「全体ベクトル」情報を各プロセッサに生成するプログラムを作成する．MPI_Allgathervを使用する（S1-2）.

課題S1（2/2）

- 下記の数値積分を台形公式によって求めるプログラムを作成する．MPI_Reduce，MPI_Bcast等を使用して並列化を実施し，プロセッサ数を変化させた場合の計算時間を測定する（S1-3）．

$$\int_0^1 \frac{4}{1+x^2}\,dx$$



$$\frac{1}{2}\Delta x\left(f_1 + f_{N+1} + \sum_{i=2}^{N} 2f_i\right)$$