MPI「超」入門（C言語編）

東京大学情報基盤センター

FOTRAN編は以下

http://nkl.cc.u-tokyo.ac.jp/seminars/T2Kfvm/MPIprogf.pdf

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（Point-to-Point 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，ライブラリ，そのものではない
    - http://phase.hpcc.jp/phase/mpi-j/ml/mpi-j-html/contents.html
- 歴史
  - 1992　MPIフォーラム
  - 1994　MPI-1規格
  - 1997　MPI-2規格（拡張版），現在はMPI-3が検討されている
- 実装
  - mpich　アルゴンヌ国立研究所
  - LAM
  - 各ベンダー
  - C/C++,FOTRAN,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-unix.mcs.anl.gov/mpi/
- 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
- 同様の試みとしてPVM（Parallel Virtual Machine）があったが，こちらはそれほど広がらず

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.
- http://www-unix.mcs.anl.gov/mpi/www/
  – API（Application Interface）の説明

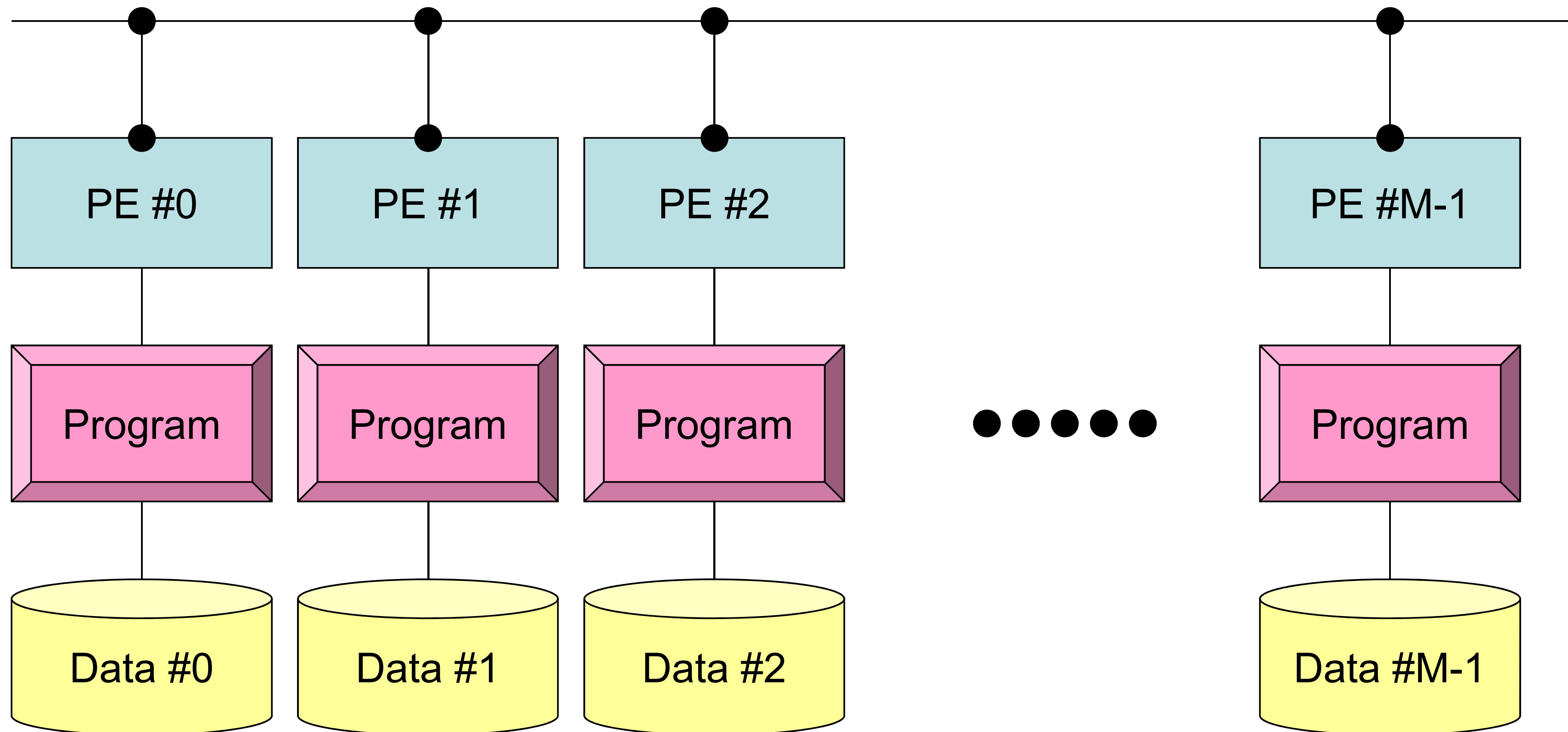
MPIを学ぶにあたって（1/2）

- 文法
  - 「MPI-1」の基本的な機能（10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/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。



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
通信以外は，単体CPUのときと同じ，というのが理想

用語

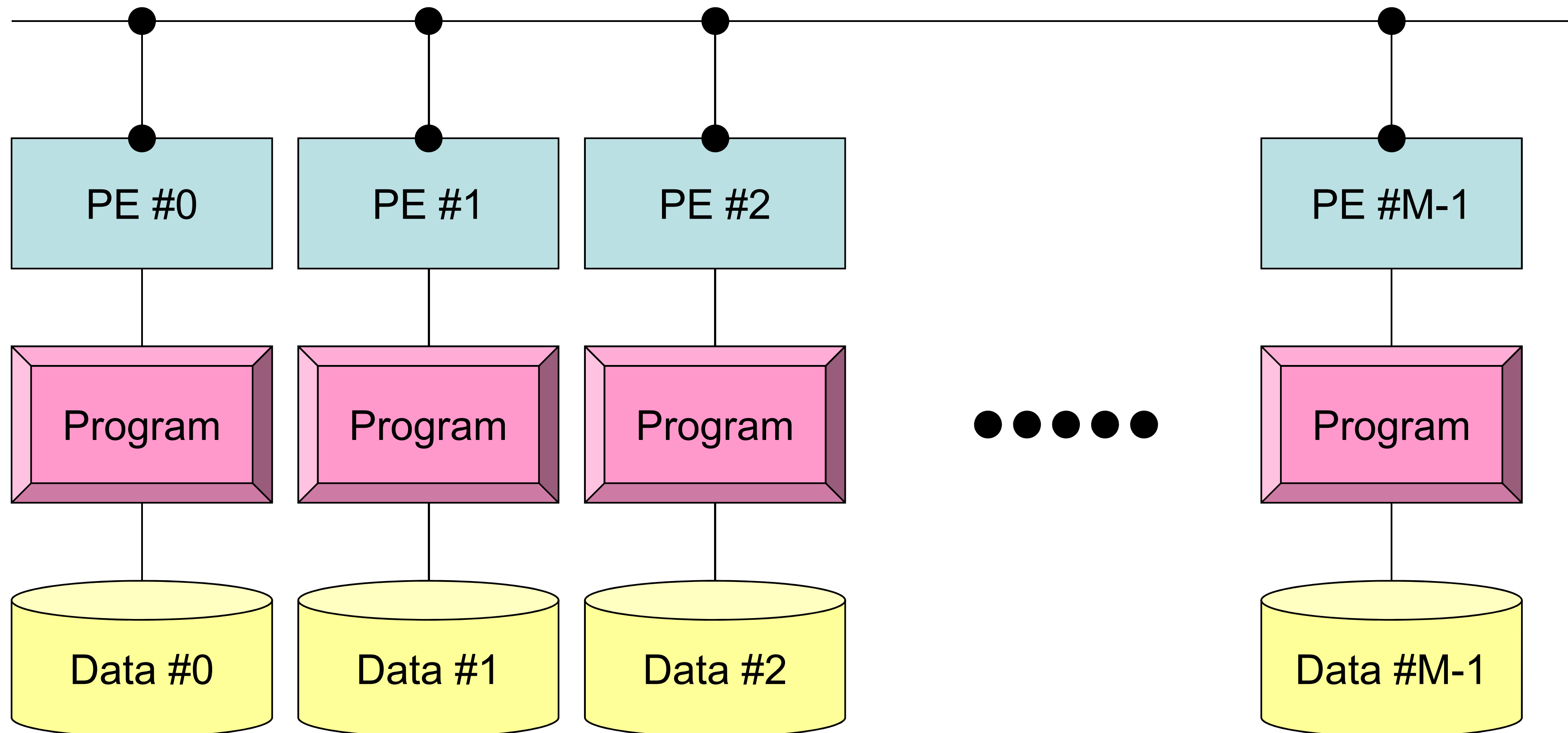
- プロセッサ，コア
  - ハードウェアとしての各演算装置。シングルコアではプロセッサ＝コア
- プロセス
  - MPI計算のための実行単位，ハードウェア的な「コア」とほぼ同義。
  - しかし1つの「プロセッサ・コア」で複数の「プロセス」を起動する場合もある（効率的ではないが）。
- PE（Processing Element）
  - 本来，「プロセッサ」の意味なのであるが，本講義では「プロセス」の意味で使う場合も多い。次項の「領域」とほぼ同義でも使用。
    - マルチコアの場合は：「コア＝PE」という意味で使うことが多い。
- 領域
  - 「プロセス」とほぼ同じ意味であるが，SPMD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
通信以外は，単体CPUのときと同じ，というのが理想

MPIを学ぶにあたって（2/2）

- 繰り返すが，決して難しいものではない。
- 以上のようなこともあって，文法を教える授業は2~3回程度で充分と考えている（今回はもっと短い：正味90分くらいか）。
- とにかくSPMDの考え方を掴むこと！

内 容

- 環境管理
- グループ通信
  - Collective Communication
- 1対1通信
  - Point-to-Point Communication

- 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（Point-to-Point Communication）

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/c をコンパイルしてみよう！

```
>$ cd <$FVM>/S1
>$ mpicc -Os -noparallel hello.c
>$ mpif90 -Oss -noparallel hello.f
```

FORTRAN

$> mpif90 -Oss -noparallel hello.f

"mpif90":

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

C言語

$> mpicc -Os -noparallel hello.c

"mpicc":

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

ジョブ実行

- 実行方法
  - 基本的にバッチジョブのみ
  - インタラクティヴの実行は「基本的に」できません
- 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

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

```
#@$-r hello            実行ジョブ名（qstatで表示）
#@$-q lecture          実行キュー名
#@$-N 1                使用ノード数
#@$-J T4               ノードあたりMPIプロセス数（T1~T16）
#@$-e err              標準エラー出力ファイル名
#@$-o hello.lst        標準出力ファイル名
#@$-lM 28GB            1ノードあたりメモリ使用量（固定）
#@$-lT 00:05:00        実行時間（上限15分，この場合は5分）
#@$

cd $PBS_O_WORKDIR                          実行ディレクトリ移動
mpirun numactl --localalloc ./a.out        mpirun
```

ジョブスクリプト（詳細）

```
#@$-r hello          実行ジョブ名（qstatで表示）
#@$-q lecture        実行キュー名
#@$-N 1              使用ノード数
#@$-J T4             ノードあたりMPIプロセス数（T1~T16）
#@$-e err            標準エラー出力ファイル名
#@$-o hello.lst      標準出力ファイル名
#@$-lM 28GB          1ノードあたりメモリ使用量（固定）
#@$-lT 00:05:00      実行時間（上限15分，この場合は5分）
#@$

cd $PBS_O_WORKDIR                              実行ディレクトリ移動
mpirun numactl --localalloc ./a.out       mpirun
```

- **mpirun -np XXは不要：N×Jがプロセス数**
- **普通は「mpirun -np 4 a.out」のように走らせる**

ジョブ投入

```
>$ cd <$FVM>/S1
>$ qsub hello.sh

>$ cat hello.lst

  Hello World 0
  Hello World 3
  Hello World 2
  Hello World 1
```

利用可能なキュー

```
#@$-r hello        実行ジョブ名（qstatで表示）
#@$-q lecture      実行キュー名
#@$-N 1            使用ノード数
#@$-J T4           ノードあたりMPIプロセス数（T1~T16）
```

- 以下の2種類のキューを利用可能
  - **lecture**
    - 4ノード（64コア），15分，アカウント有効期間中利用可能
    - 1回に1ジョブのみ実行可能（全教育ユーザーで共有）
  - **tutorial**
    - 4ノード（64コア），15分，講義時間のみ
    - **lecture**よりは多くのジョブを投入可能（混み具合による）

ジョブ投入，確認等

- ジョブの投入　　qsub スクリプト名
- ジョブの確認　　qstat
- キューの状態の確認　　qstat -b
- ジョブの取り消し・強制終了　　qdel ジョブID

```
[t15026@ha8000-3 S1]$ qstat -b
2008/08/24 (Sun) 12:59:33:    BATCH QUEUES on HA8000 cluster
NQS schedule stop time : 2008/08/29 (Fri)  9:00:00 (Remain: 116h  0m 27s)
QUEUE NAME       STATUS     TOTAL    RUNNING  RUNLIMIT  QUEUED  HELD     IN-TRANSIT
lecture          AVAILBL      0        0         1         0       0        0
lecture5         STOPPED      0        0         4         0       0        0
[t15026@ha8000-3 S1]$ qsub go.sh
Request 79880.batch1 submitted to queue: lecture.
[t15026@ha8000-3 S1]$ qstat
2008/08/24 (Sun) 12:59:43:    REQUESTS on HA8000 cluster
NQS schedule stop time : 2008/08/29 (Fri)  9:00:00 (Remain: 116h  0m 17s)
   REQUEST      NAME       OWNER      QUEUE      PRI NICE   CPU    MEM    STATE
79880.batch1    S1-3       t15026     lecture    0    0   unlimit   28GB  QUEUED
[t15026@ha8000-3 S1]$ qdel 79880
deleting request 79880.batch1.
[t15026@ha8000-3 S1]$ qstat
2008/08/24 (Sun) 12:59:51:    REQUESTS on HA8000 cluster
NQS schedule stop time : 2008/08/29 (Fri)  9:00:00 (Remain: 116h  0m  9s)
   REQUEST      NAME       OWNER      QUEUE      PRI NICE   CPU    MEM    STATE
No requests.
```

結果確認

- ジョブが終了するとメールがきます
  - ジョブスクリプトに `-mu` オプションを書けば任意のメールアドレスに送信できます
  - `~/.forward` を設定しておけばオプションを書かなくても自分のメールアドレスに送信できます
- 結果の確認
  - 標準出力:
  - 標準エラー出力

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。
- 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？

```
#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();
}
```

```
#@$-r hello            実行ジョブ名（qstatで表示）
#@$-q lecture          実行キュー名
#@$-N 1                使用ノード数
#@$-J T4               ノードあたりMPIプロセス数（T1~T16）
#@$-e err              標準エラー出力ファイル名
#@$-o hello.lst        標準出力ファイル名
#@$-lM 28GB            1ノードあたりメモリ使用量（固定）
#@$-lT 00:05:00        実行時間（上限15分，この場合は5分）
#@$

cd $PBS_O_WORKDIR                      実行ディレクトリ移動
mpirun numactl --localalloc ./a.out    mpirun
```

- **mpirun -np 4 <prog>** により4つのプロセスが立ち上がる（今の場合はT4）。
  - 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。

MPI_Init

- MPIを起動する。他のMPI関数より前にコールする必要がある（必須）
- **MPI_Init (argc, argv)**

```
#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_Finalize

- MPIを終了する。他の全てのMPI関数より後にコールする必要がある（必須）。
- 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・・・

- **MPI_Finalize ()**

```
#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_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」にもどる。必須では無いが，利用することが多い。

- **MPI_Comm_size (comm, size)**
  - **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

```
#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();
}
```

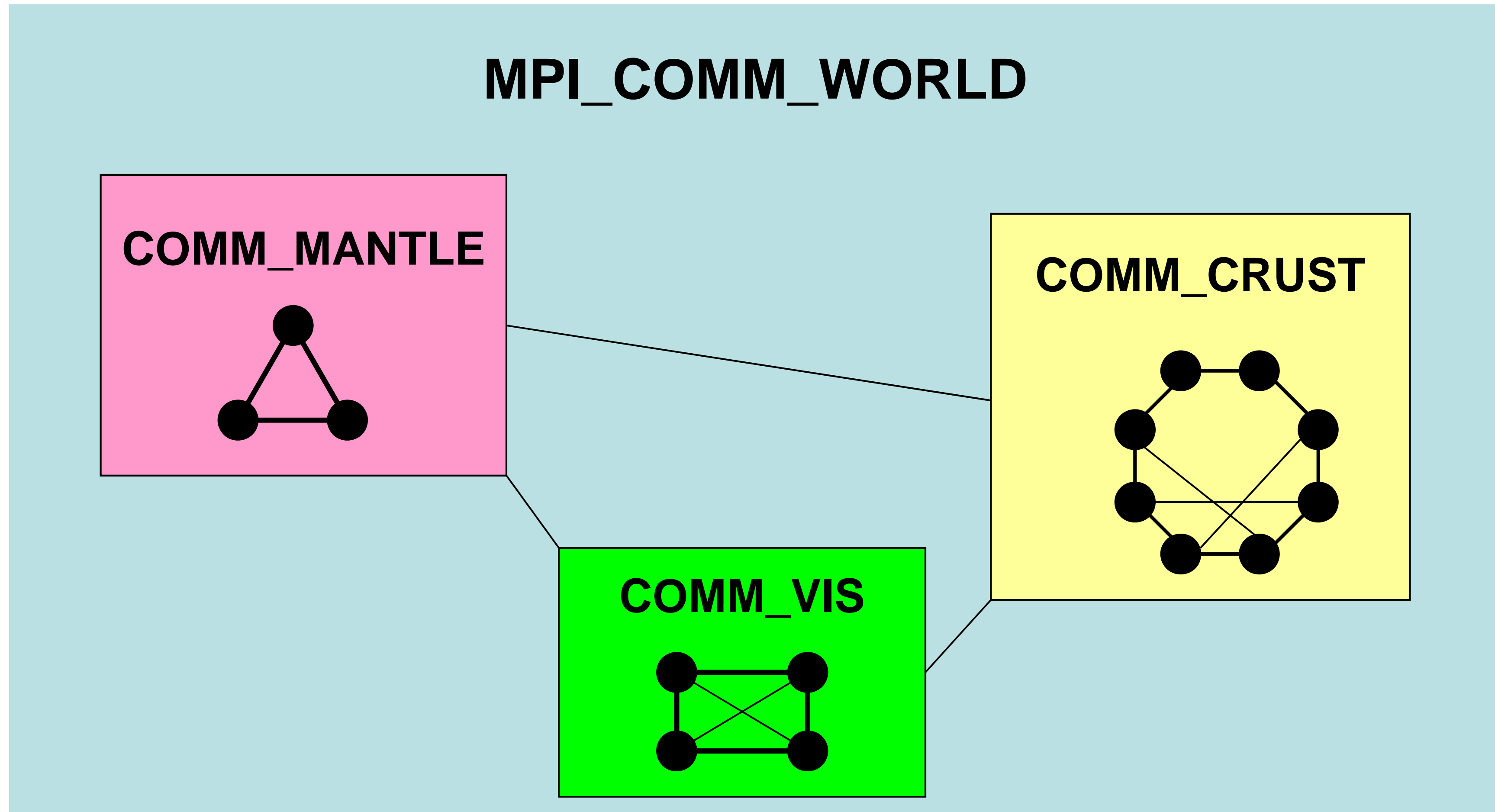
コミュニケータとは？

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

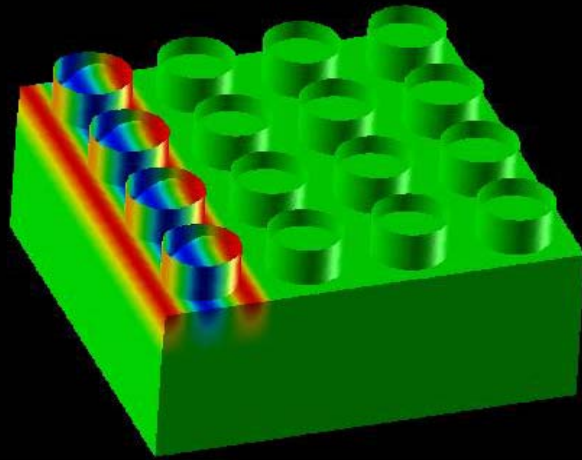
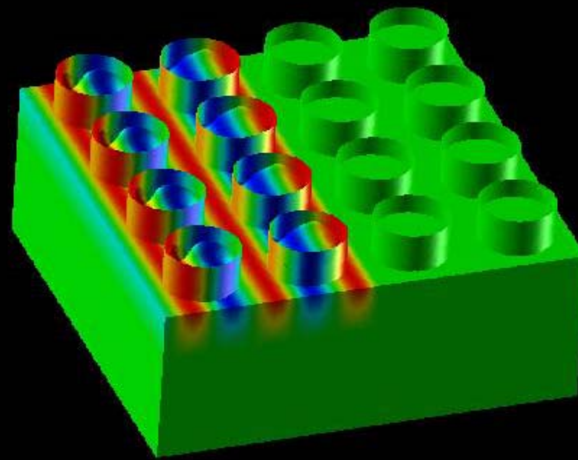
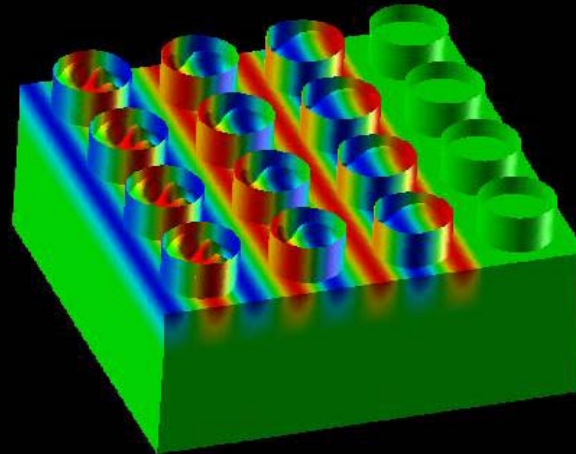
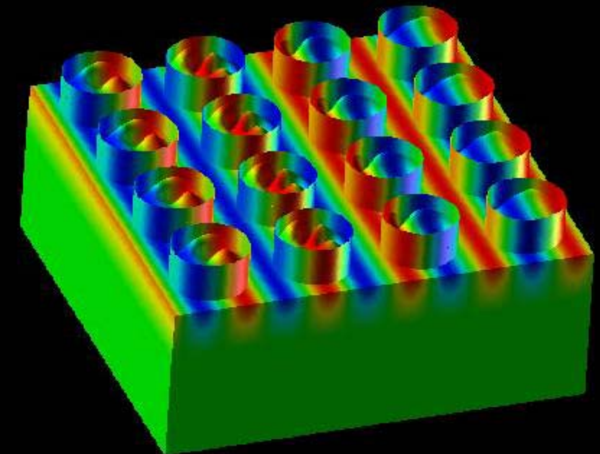
- 通信を実施するためのプロセスのグループを示す。
- MPIにおいて，通信を実施する単位として必ず指定する必要がある。
- mpirunで起動した全プロセスは，デフォルトで「**MPI_COMM_WORLD**」というコミュニケータで表されるグループに属する。
- 複数のコミュニケータを使用し，異なったプロセス数を割り当てることによって，複雑な処理を実施することも可能。
  - 例えば計算用グループ，可視化用グループ
- この授業では「**MPI_COMM_WORLD**」のみでOK。

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



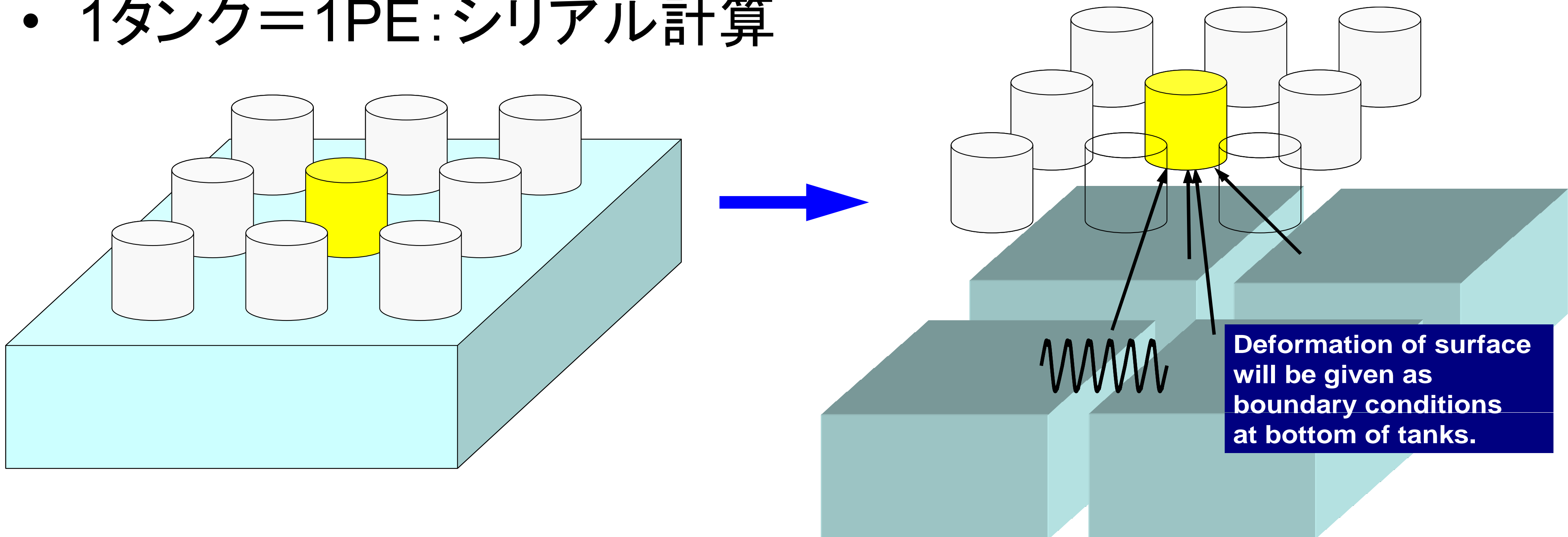
地盤・石油タンク連成シミュレーション

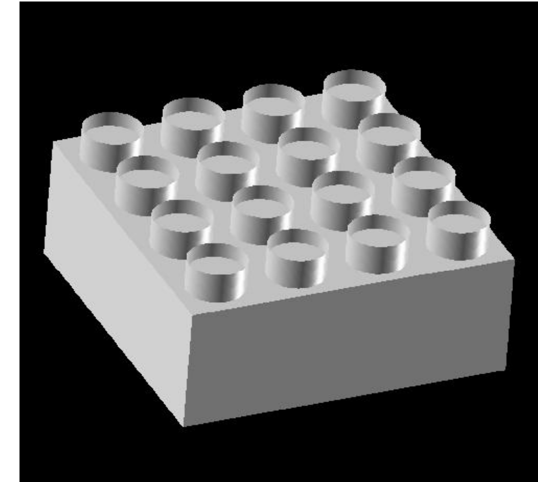
動 画

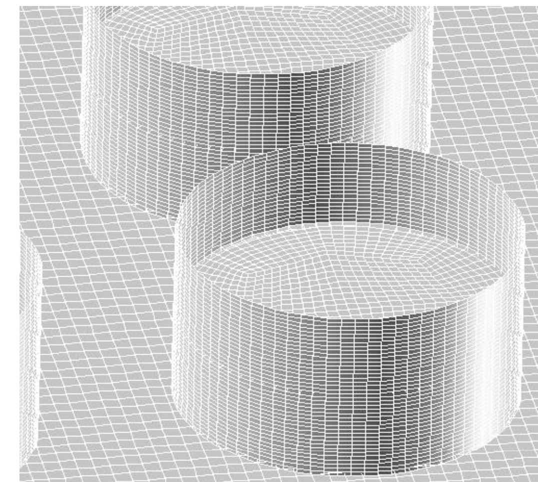
対象とするアプリケーション

- 地盤・石油タンク振動
  - 地盤⇒タンクへの「一方向」連成
  - 地盤表層の変位 ⇒ タンク底面の強制変位として与える
- このアプリケーションに対して，連成シミュレーションのためのフレームワークを開発，実装
- 1タンク＝1PE：シリアル計算



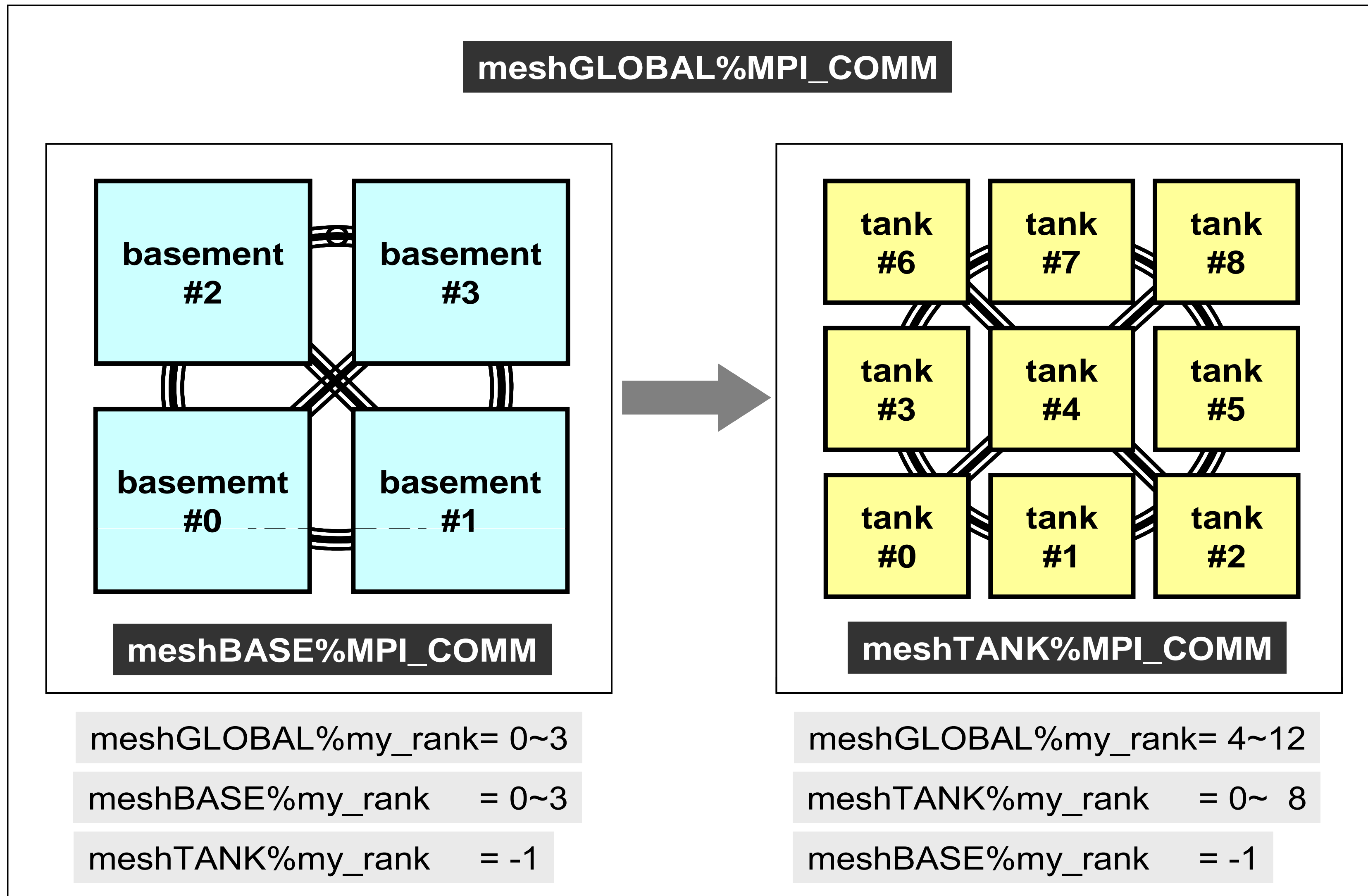
地盤，タンクモデル





- 地盤モデル：FORTRAN
  - 並列FEM，三次元弾性動解析
    - 前進オイラー陽解法，EBE
  - 各要素は一辺2mの立方体
  - 240m×240m×100m
- タンクモデル：C
  - シリアルFEM（EP），三次元弾性動解析
    - 後退オイラー陰解法，スカイライン法
    - シェル要素＋ポテンシャル流（非粘性）
  - 直径:42.7m，高さ:24.9m，厚さ:20mm，液面:12.45m，スロッシング周期:7.6sec.
  - 周方向80分割，高さ方向:0.6m幅
  - 60m間隔で4×4に配置
- 合計自由度数:2,918,169

3種類のコミュニケータの生成



MPI_Comm_rank

- コミュニケーター「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)**
  - **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）

```
#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_Abort

- MPIプロセスを異常終了する。

- **MPI_Abort (comm, errcode)**
  - <u>**comm**</u>　整数　I　コミュニケータを指定する
  - <u>**errcode**</u>　整数　O　エラーコード

MPI_Wtime

- 時間計測用の関数：精度はいまいち良くない（短い時間の場合）

- **time= MPI_Wtime ()**
  - **time** R8 O 過去のある時間からの経過時間（秒数）

```
…
double Stime, Etime;

Stime= MPI_Wtime ();

(…)

Etime= MPI_Wtime ();
```

MPI_Wtime の例

```
$> cd <$FVM>/S1

$> mpicc -O3 time.c
$> mpif90 -O3 time.f

$> 実行(4プロセス) go4.sh

    0    1.113281E+00
    3    1.113281E+00
    2    1.117188E+00
    1    1.117188E+00
```

プロセス番号　　計算時間

MPI_Wtick

- MPI_Wtimeでの時間計測精度
- 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

```
$> cd <$FVM>/S1

$> mpicc -O3 wtick.c
$> mpif90 -O3 wtick.f

$> （実行:1プロセス） go4.sh
```

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。

- **`MPI_Barrier (comm)`**
  - **`comm`** 整数 I コミュニケータを指定する

- 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（Point-to-Point Communication）

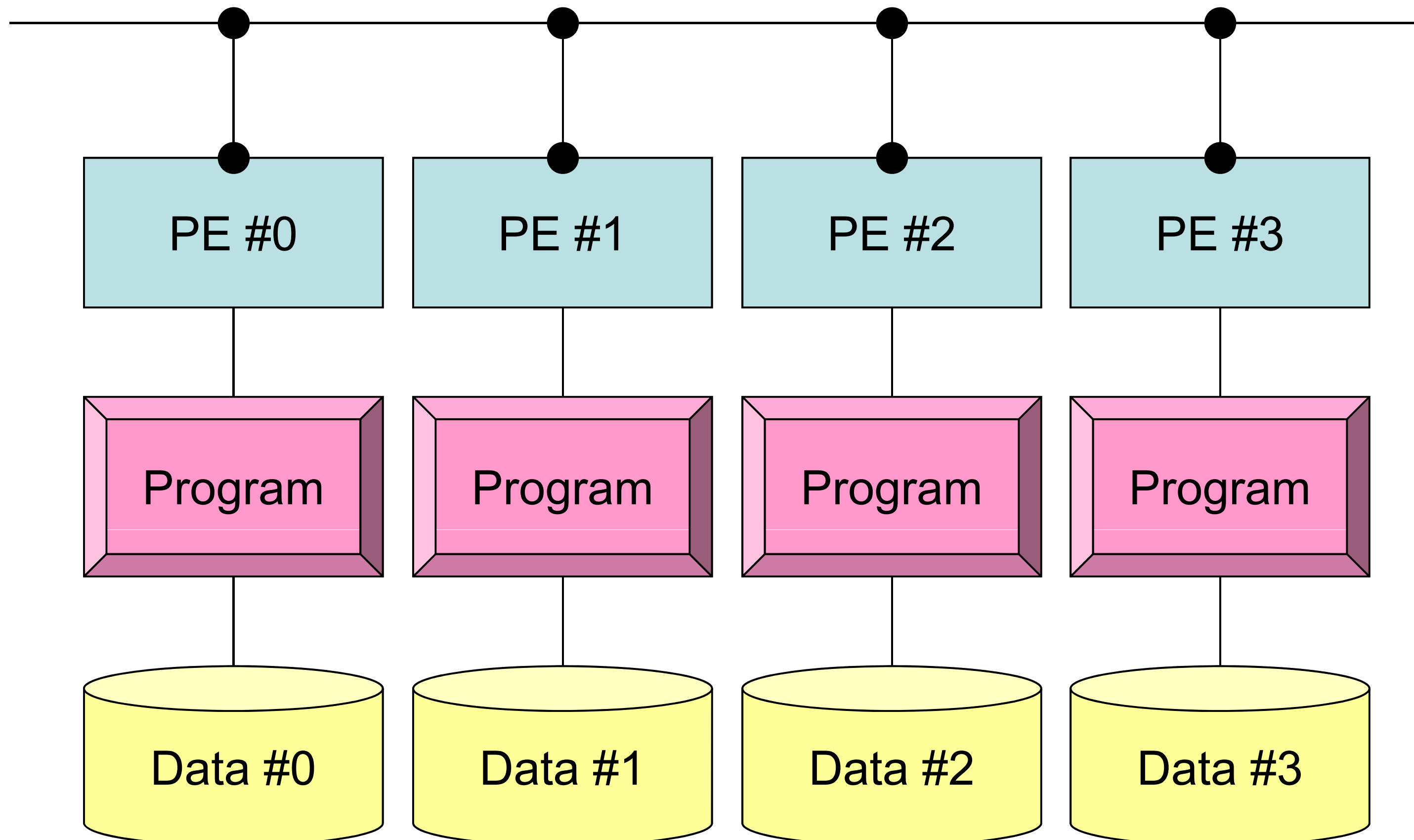
データ構造とアルゴリズム

- 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。
  - もちろん「そうではない」と主張する人もいるが，科学技術計算に関する限り，中島の経験では「データ構造＝アルゴリズム」と言える。
- 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**に対して以下のような計算を考えてみよう:

```
int main(){
        int i,Ng;
        double Vg[20];
        Ng=20;
        for(i=0;i<Ng;i++){
            Vg[i] = 2.0*Vg[i];}
return 0;}
```

- これを4つのプロセッサで分担して計算するとすれば，20/4=5 ずつ記憶し，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：

```
int main(){
        int i,Nl;
        double Vl[5];
        Nl=5;
        for(i=0;i<Nl;i++){
            Vl[i] = 2.0*Vl[i];}
return 0;}
```

- 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。
  - 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。
  - 単体CPUの場合ともほとんど変わらない。

全体データと局所データ

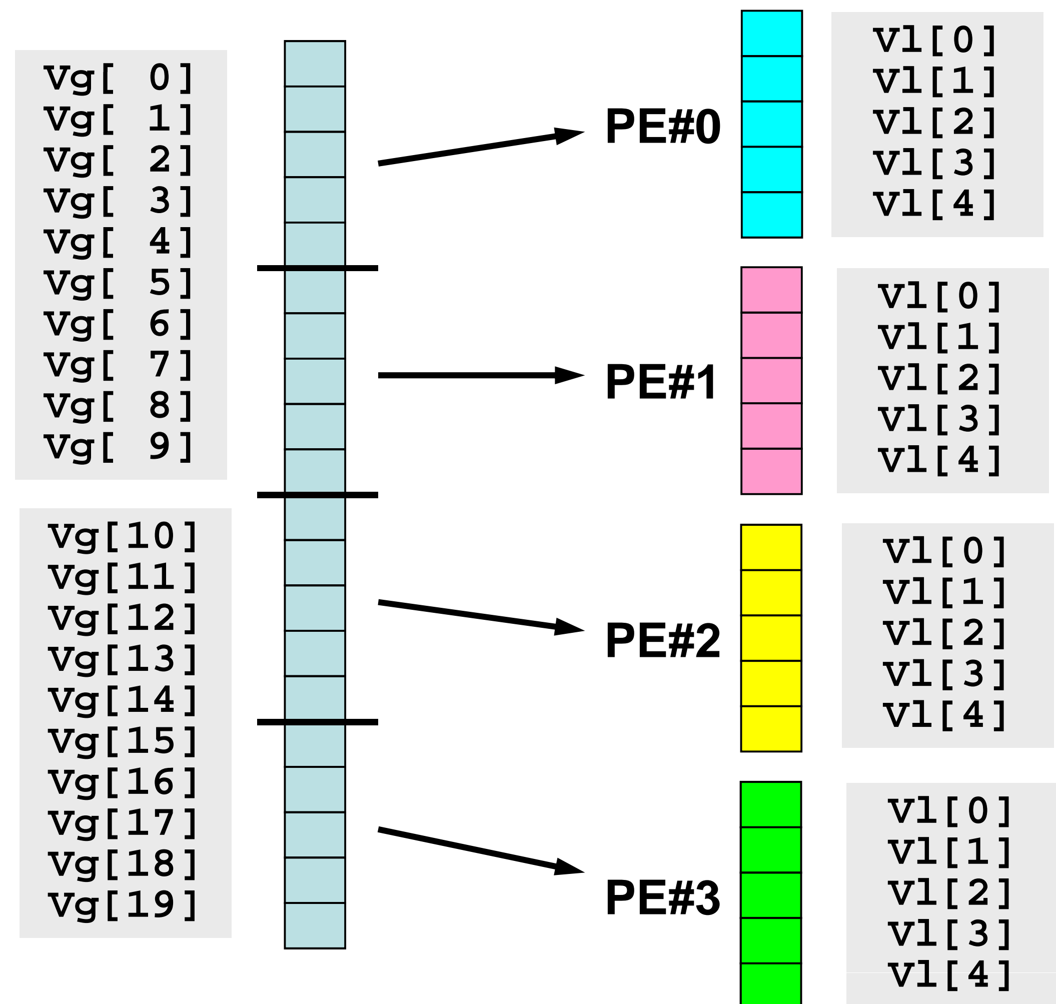
- Vg
  - 領域全体
  - 1番から20番までの「全体番号」を持つ「全体データ（Global Data）」
- Vl
  - 各プロセス（PE，プロセッサ，領域）
  - 1番から5番までの「局所番号」を持つ「局所データ（Local Data）」
  - できるだけ局所データを有効に利用することで，高い並列性能が得られる。

局所データの考え方

「全体データ」Vgの：

- 0～4番成分が0番PE
- 5~9番成分が1番PE
- 10～14番が2番PE
- 15～19番が3番PE

のそれぞれ，「局所データ」Vlの0番～4番成分となる（局所番号が0番～4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」

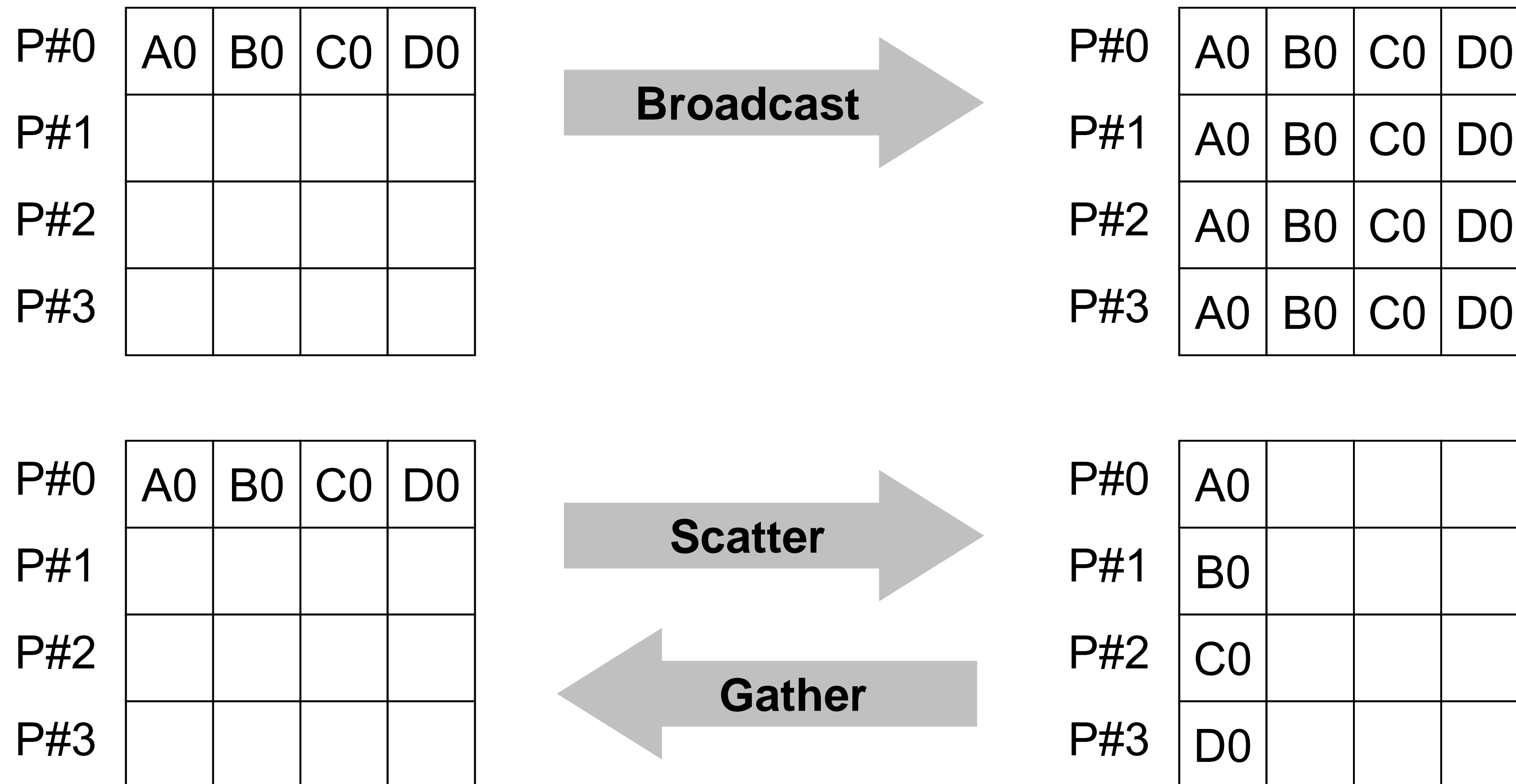
- MPIとは
- MPIの基礎：Hello World
- 全体データと局所データ
- グループ通信（Collective Communication）
- 1対1通信（Point-to-Point Communication）

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)



グループ通信の例（2/4）

All gather

P#0	A0			
P#1	B0			
P#2	C0			
P#3	D0			

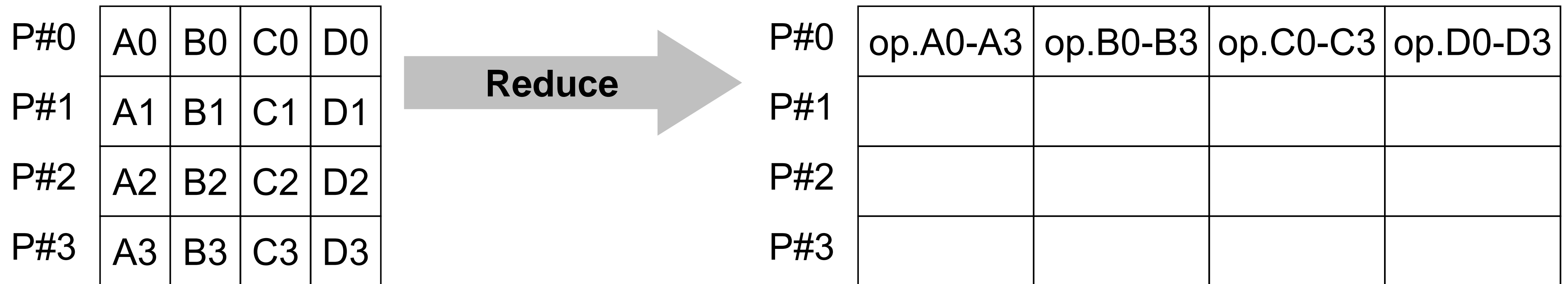
P#0	A0	B0	C0	D0
P#1	A0	B0	C0	D0
P#2	A0	B0	C0	D0
P#3	A0	B0	C0	D0

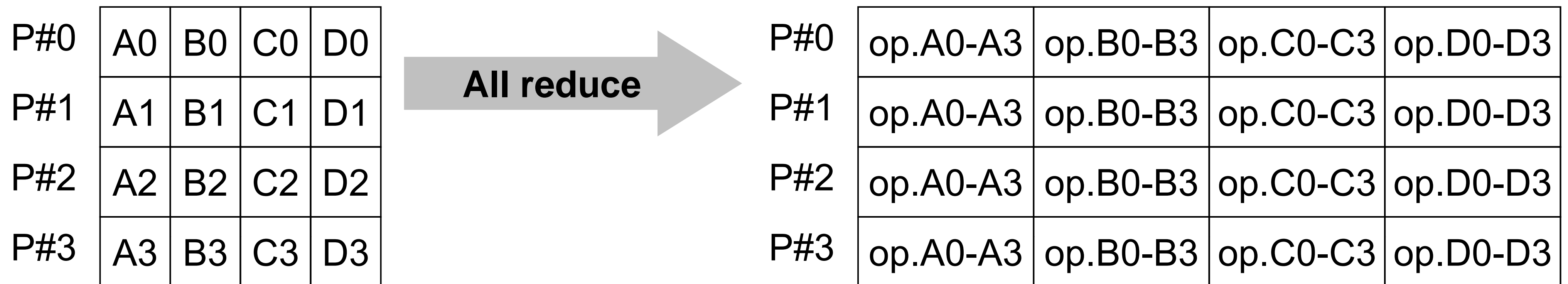
All-to-All

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

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

グループ通信の例(3/4)





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

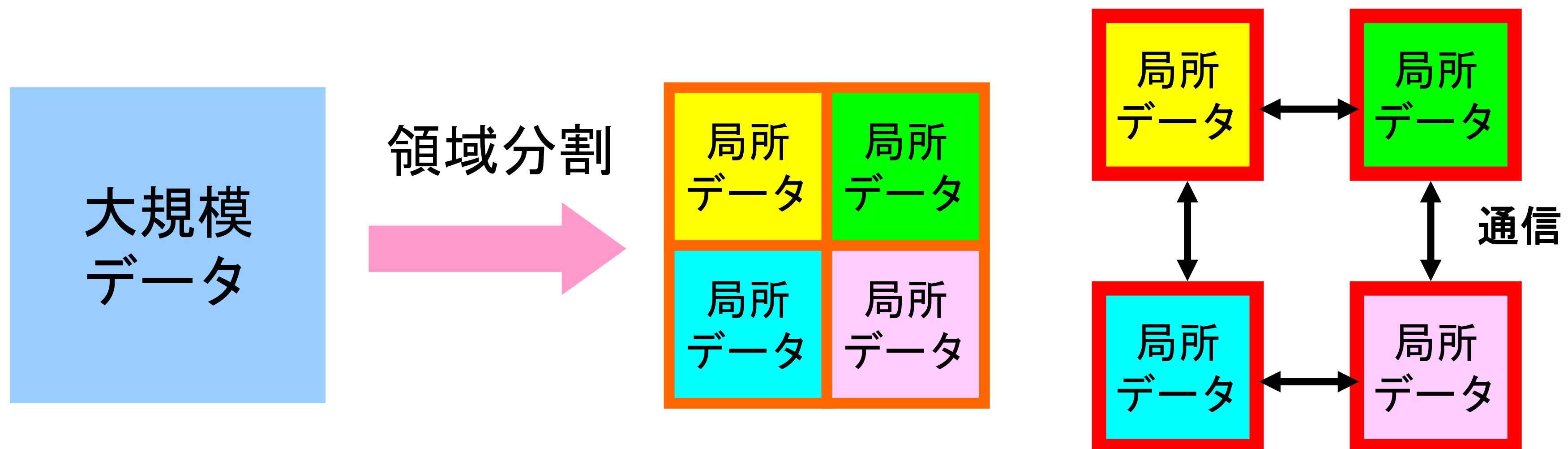
- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み

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。

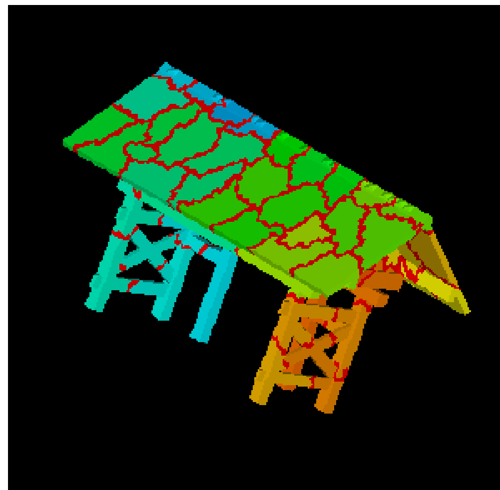
領域分割

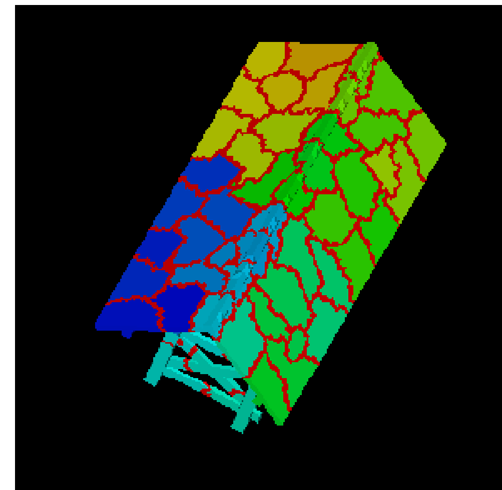
- 1GB程度のPC → 10^6メッシュが限界：FEM
  - 1000km×1000km×1000kmの領域（西南日本）を1kmメッシュで切ると10^9メッシュになる
- 大規模データ → 領域分割，局所データ並列処理
- 全体系計算 → 領域間の通信が必要



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[ 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
```

<$FVM>/S1/dot.f, dot.c

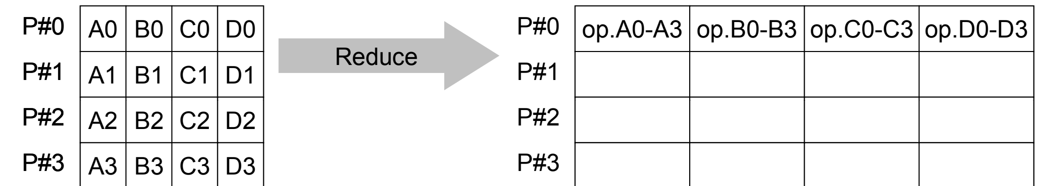
```fortran
implicit REAL*8 (A-H,O-Z)
real(kind=8),dimension(20):: &
     VECp,  VECs

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
```

```c
#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;
}
```

<$FVM>/S1/dot.f, dot.cの実行

```
>$ cd <$FVM>/S1

>$ cc -O3 dot.c
>$ f90 -O3 dot.f

>$ ./a.out

   1          2.          3.
   2          2.          3.
   3          2.          3.
…
  18          2.          3.
  19          2.          3.
  20          2.          3.

dot product       120.
```

MPI_Reduce



- コミュニケーター「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

- **MPI_Reduce (sendbuf,recvbuf,count,datatype,op,root,comm)**
  - **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

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。

MPI_Reduceの例（1/2）

```
MPI_Reduce
(sendbuf,recvbuf,count,datatype,op,root,comm)
```

```
double X0, X1;

MPI_Reduce
(&X0, &X1, 1, MPI_DOUBLE, MPI_MAX, 0, <comm>);
```

```
double X0[4], XMAX[4];

MPI_Reduce
(&X0, &XMAX, 4, MPI_DOUBLE, MPI_MAX, 0, <comm>);
```

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=0~3）

MPI_Reduceの例（2/2）

```
MPI_Reduce
(sendbuf,recvbuf,count,datatype,op,root,comm)
```

```
double X0, XSUM;

MPI_Reduce
(&X0, &XSUM, 1, MPI_DOUBLE, MPI_SUM, 0, <comm>)
```

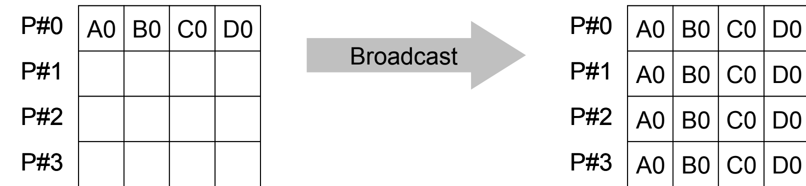
各プロセスにおける，X0の総和が0番PEのXSUMに入る。

```
double X0[4];

MPI_Reduce
(&X0[0], &X0[2], 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>)
```

各プロセスにおける，

- X0[0]の総和が0番プロセスのX0[2]に入る。
- X0[1]の総和が0番プロセスのX0[3]に入る。

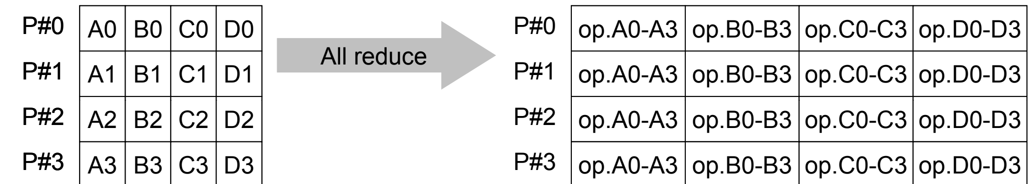
MPI_Bcast



- 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。

- **MPI_Bcast (buffer,count,datatype,root,comm)**
  - <u>**buffer**</u>　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
  - <u>**count**</u>　整数　I　メッセージのサイズ
  - <u>**datatype**</u>　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.
  - <u>**root**</u>　整数　I　送信元プロセスのID（ランク）
  - <u>**comm**</u>　整数　I　コミュニケータを指定する

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)`
  - 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

MPI_Reduce/Allreduceの “op”

```
MPI_Reduce
(sendbuf,recvbuf,count,datatype,op,root,comm)
```

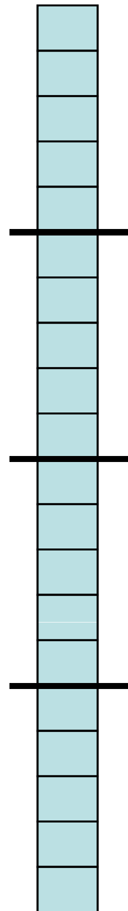
- **MPI_MAX**, **MPI_MIN** 最大値，最小値
- **MPI_SUM**, **MPI_PROD** 総和，積
- **MPI_LAND** 論理AND

局所データの考え方(1/2)

- 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)

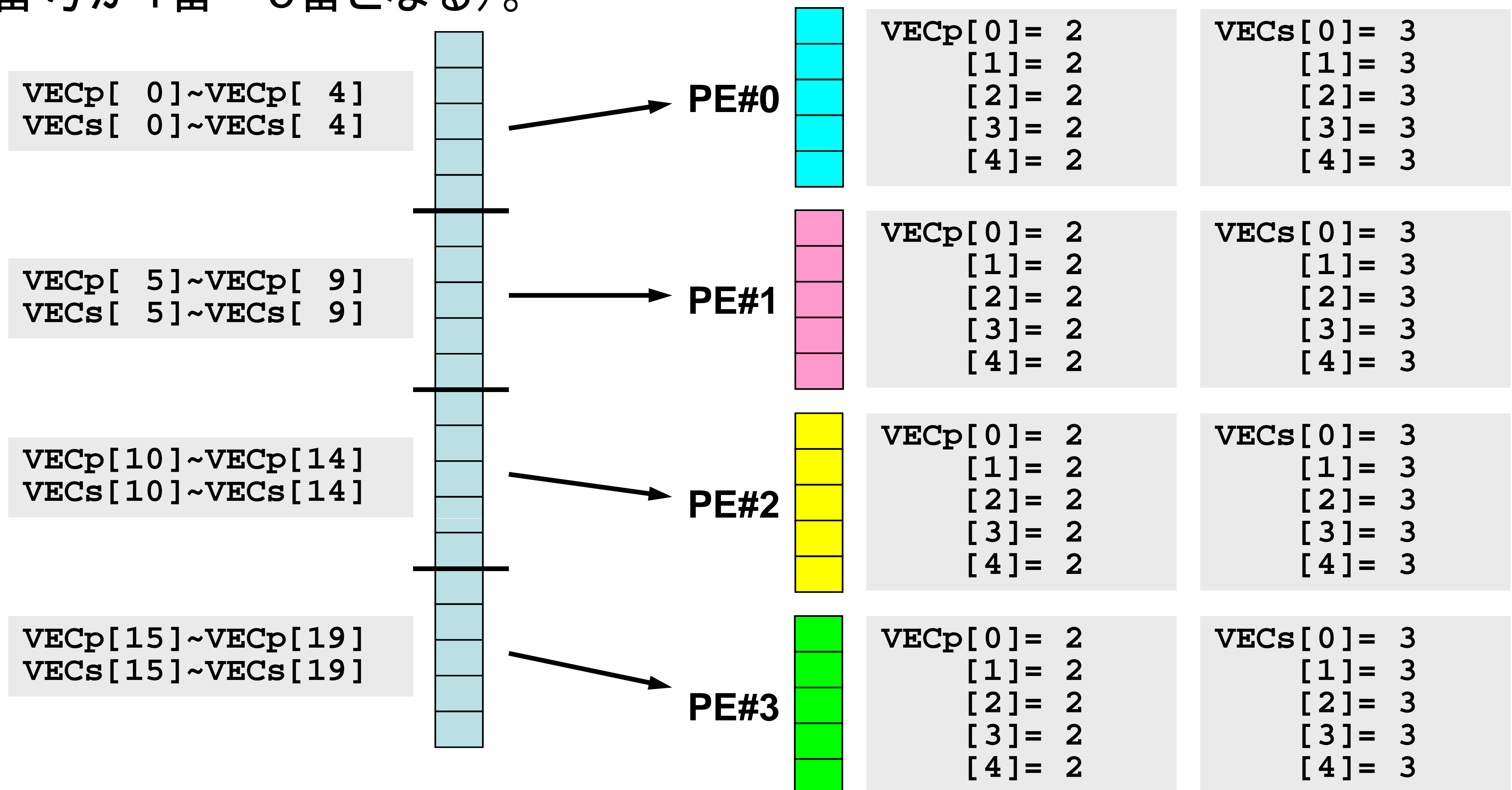
```
VECp[ 0]=  2
    [ 1]=  2
    [ 2]=  2
…
    [17]=  2
    [18]=  2
    [19]=  2
```

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



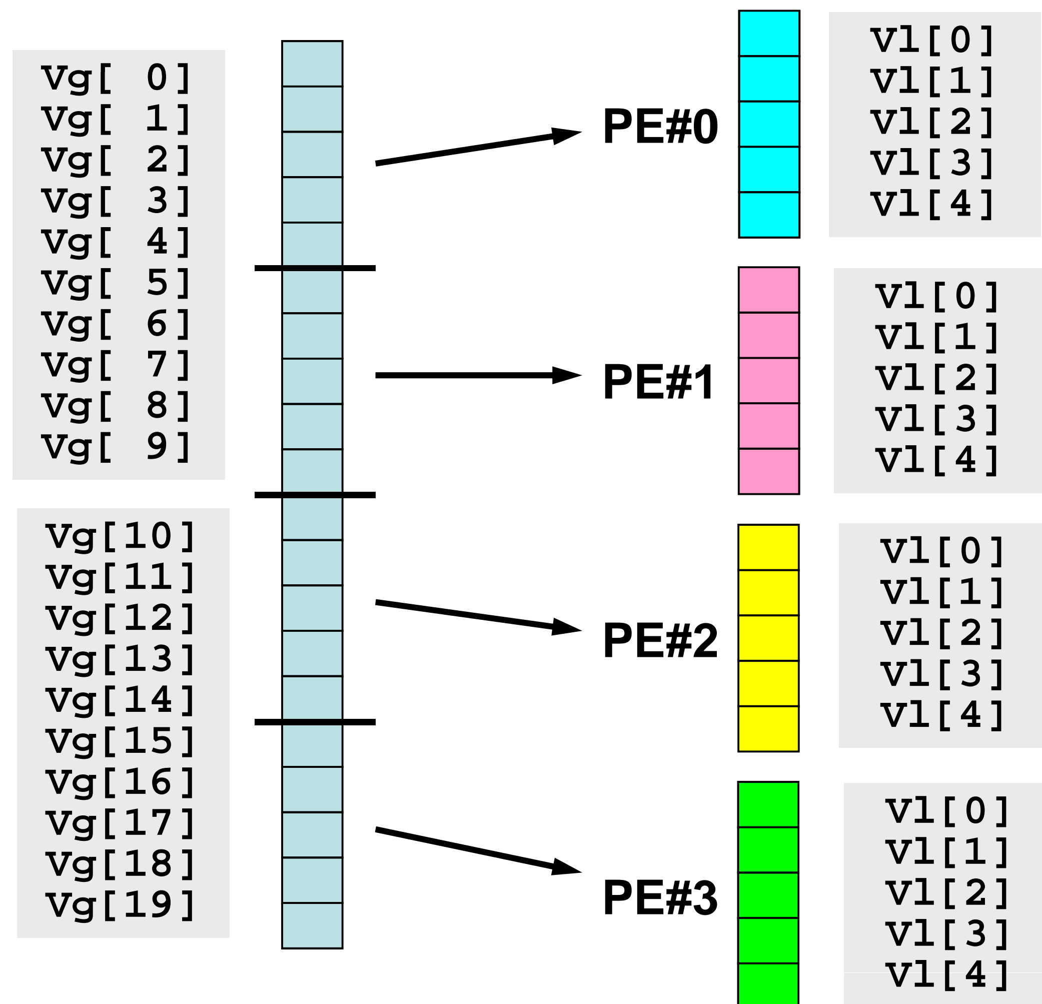
局所データの考え方(2/2)

- 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で紹介する。



内積の並列計算例(1/2)

<$FVM>/S1/allreduce.c

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

int main(int argc, char **argv){
        int i,N;
        int PeTot, MyRank;
        double VECp[5], VECs[5];
        double sumA, sumR, sum0;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sumA= 0.0;
        sumR= 0.0;

        N=5;
        for(i=0;i<N;i++){
          VECp[i] = 2.0;
          VECs[i] = 3.0;
        }

        sum0 = 0.0;
        for(i=0;i<N;i++){
                sum0 += VECp[i] * VECs[i];
        }
```

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(2/2)

<$FVM>/S1/allreduce.c

```
MPI_Reduce(&sum0, &sumR, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Allreduce(&sum0, &sumA, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

<u>内積の計算</u>
各プロセスで計算した結果「sum0」の総和をとる
sumR には，PE#0の場合にのみ計算結果が入る。

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

```
MPI_Bcast(&sumR, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

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

<$FVM>/S1/allreduce.c の実行例

```
$> mpicc -Os   -noparallel allreduce.c
$> mpif90 -Oss -noparallel allreduce.f
$> （実行:4プロセス） go4.sh

(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.**
  - 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(<$FVM>/mpi/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）

- 計算手順
  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
```

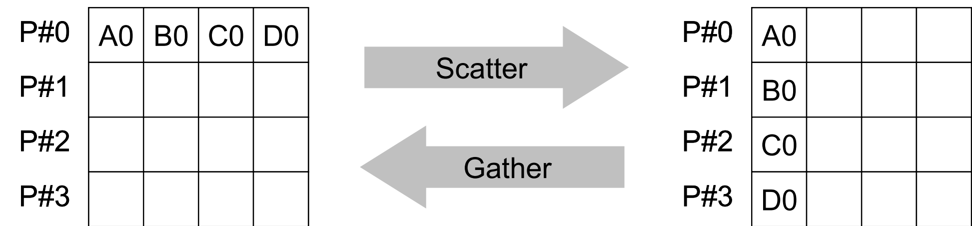
```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

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

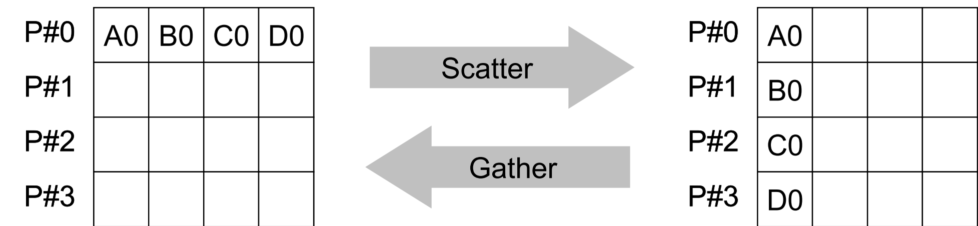
MPI_Scatter



- コミュニケーター「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。

- **MPI_Scatter (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm)**
  - 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

MPI_Scatter（続き）



- **MPI_Scatter (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm)**
  - **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

- 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。

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:

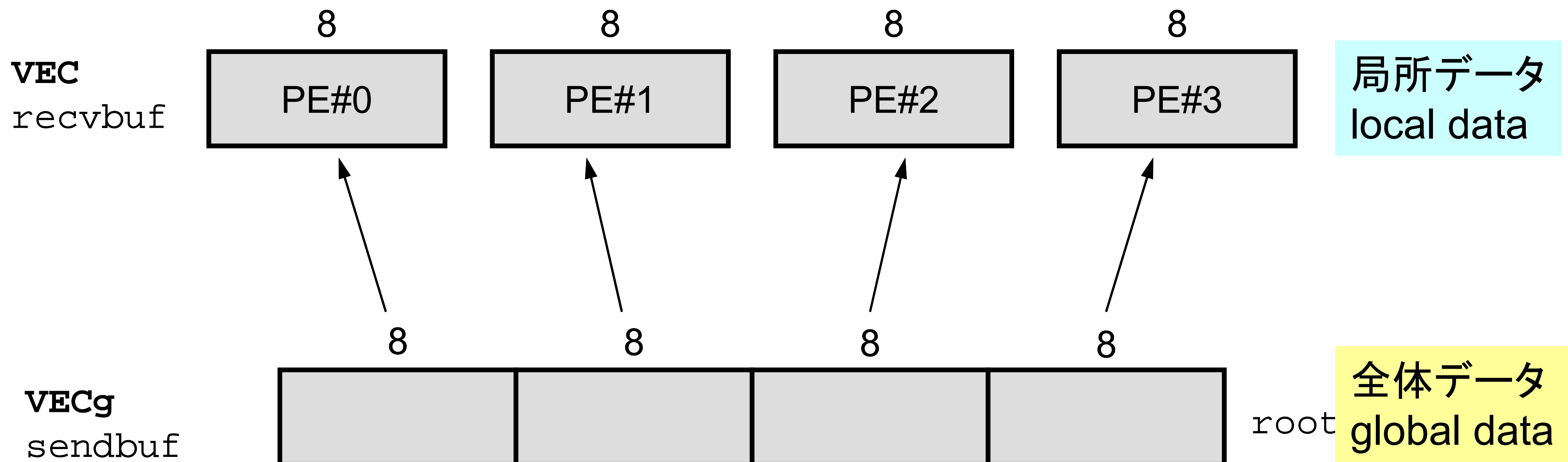
```
integer, parameter :: N =  8
real(kind=8), dimension(N ) :: VEC
...
call MPI_Scatter                          &
     (VECg, N, MPI_DOUBLE_PRECISION, &
      VEC , N, MPI_DOUBLE_PRECISION, &
      0, <comm>, ierr)
```

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

```
MPI_Scatter
(sendbuf, scount, sendtype, recvbuf, rcount,
 recvtype, root, comm)
```

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.
```

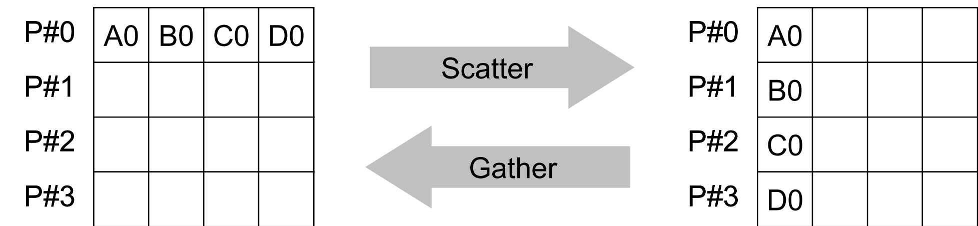
```
PE#3
after 3 1   1401.
after 3 2   1403.
after 3 3   1405.
after 3 4   1406.
after 3 5   1409.
after 3 6   1411.
after 3 7   1421.
after 3 8   1451.
```

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

- **MPI_Gather (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, root, comm)**
  - **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

- 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。

Scatter/Gatherの計算（8/8）

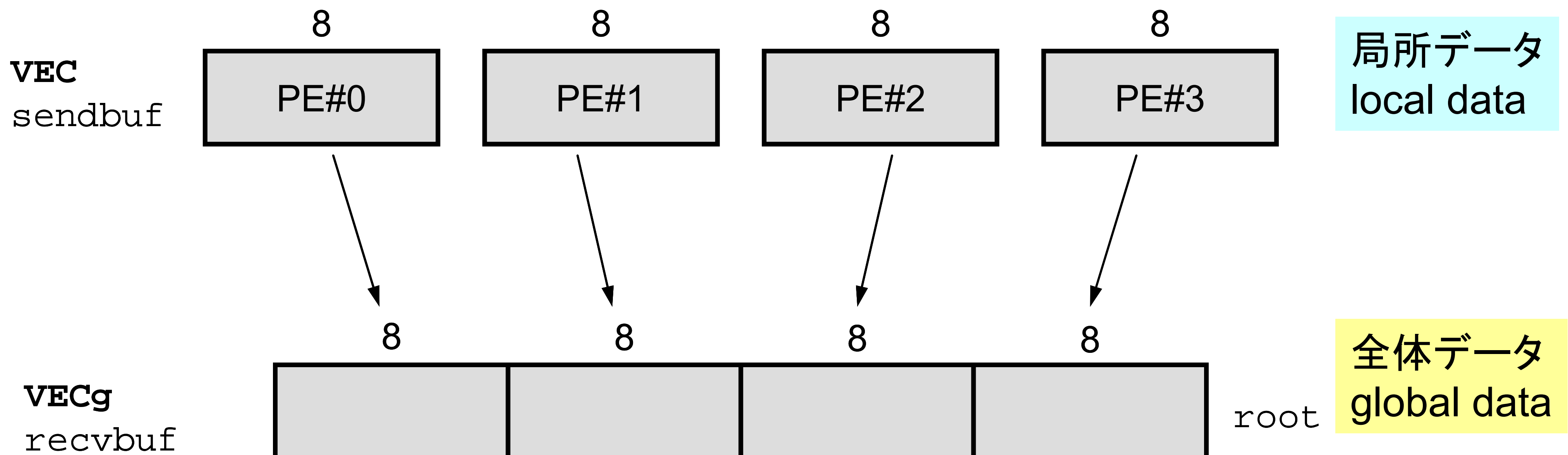
各プロセスの結果を再び長さ32のベクトルにまとめる

- 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



<$FVM>/S1/scatter-gather.c 実行例

```
$> mpicc -Os -noparallel scatter-gather.c
$> mpif90 -Oss -noparallel scatter-gather.f
$> 実行(4プロセス) go4.sh
```

```
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.
```

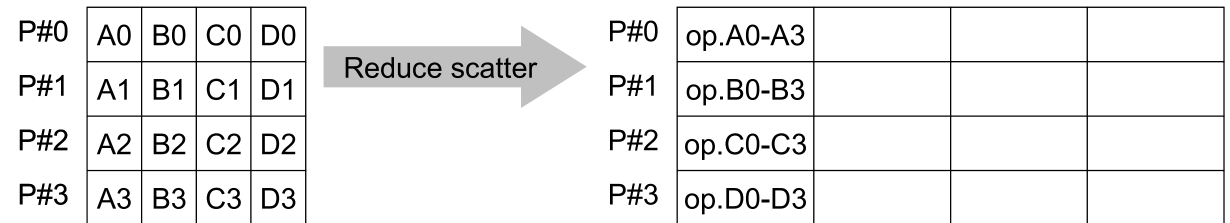
```
PE#0
after 0 1   1101.
after 0 2   1103.
after 0 3   1105.
after 0 4   1106.
after 0 5   1109.
after 0 6   1111.
after 0 7   1121.
after 0 8   1151.
```

```
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.
```

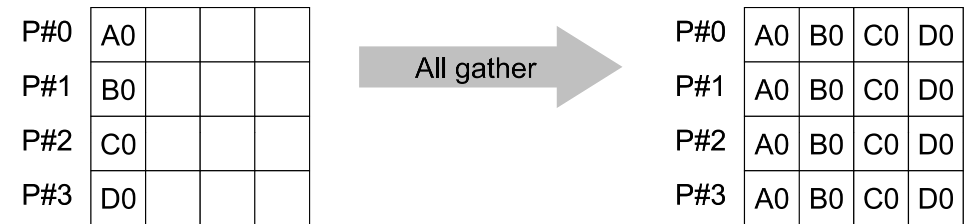
MPI_Reduce_scatter



- MPI_Reduce + MPI_Scatter

- **`MPI_Reduce_Scatter (sendbuf, recvbuf, rcount, datatype, op, comm)`**
  - **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

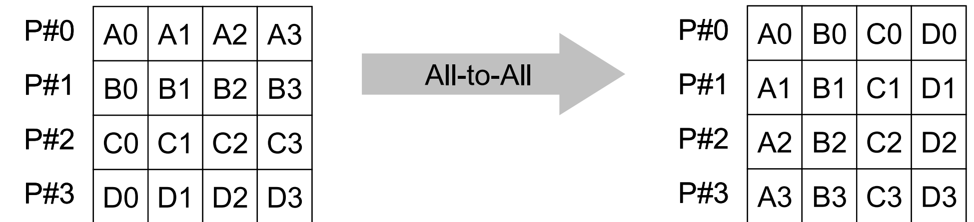
MPI_Allgather



- 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）

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

MPI_Alltoall



- MPI_Allgatherの更なる拡張：転置

- **MPI_Alltoall (sendbuf, scount, sendtype, recvbuf, rcount, recvrype, comm)**
  - **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

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

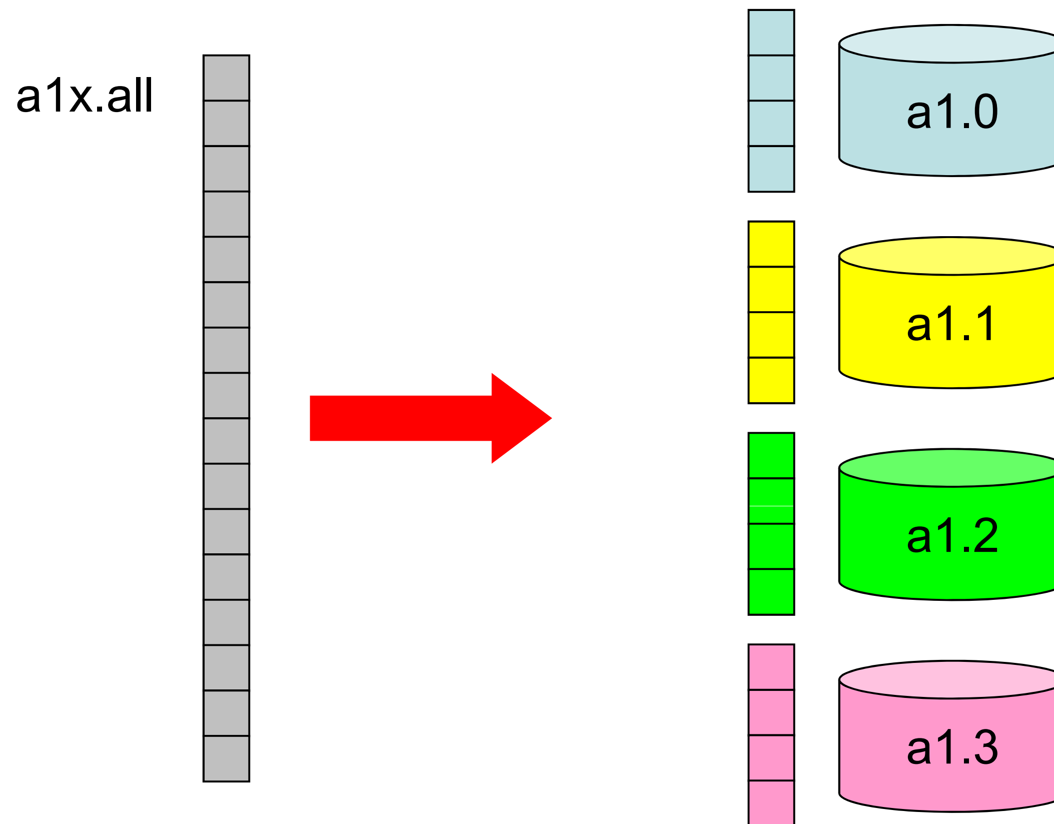
- Scatter/Gatherの例では，PE#0から全体データを読み込み，それを全体にScatterして並列計算を実施した。
- 問題規模が非常に大きい場合，1つのプロセッサで全てのデータを読み込むことは不可能な場合がある。
  - 最初から分割しておいて，「局所データ」を各プロセッサで独立に読み込む
  - あるベクトルに対して，全体操作が必要になった場合は，状況に応じてMPI_Gatherなどを使用する

分散ファイル読み込み：等データ長（1/2）

```
>$ cd <$FVM>/S1
>$ ls a1.*
   a1.0 a1.1 a1.2 a1.3    「a1x.all」を4つに分割したもの

>$ mpicc -Os -noparallel file.c
>$ mpif90 -Oss -noparallel file.f

>$ 実行：4プロセス go4.sh
```

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）

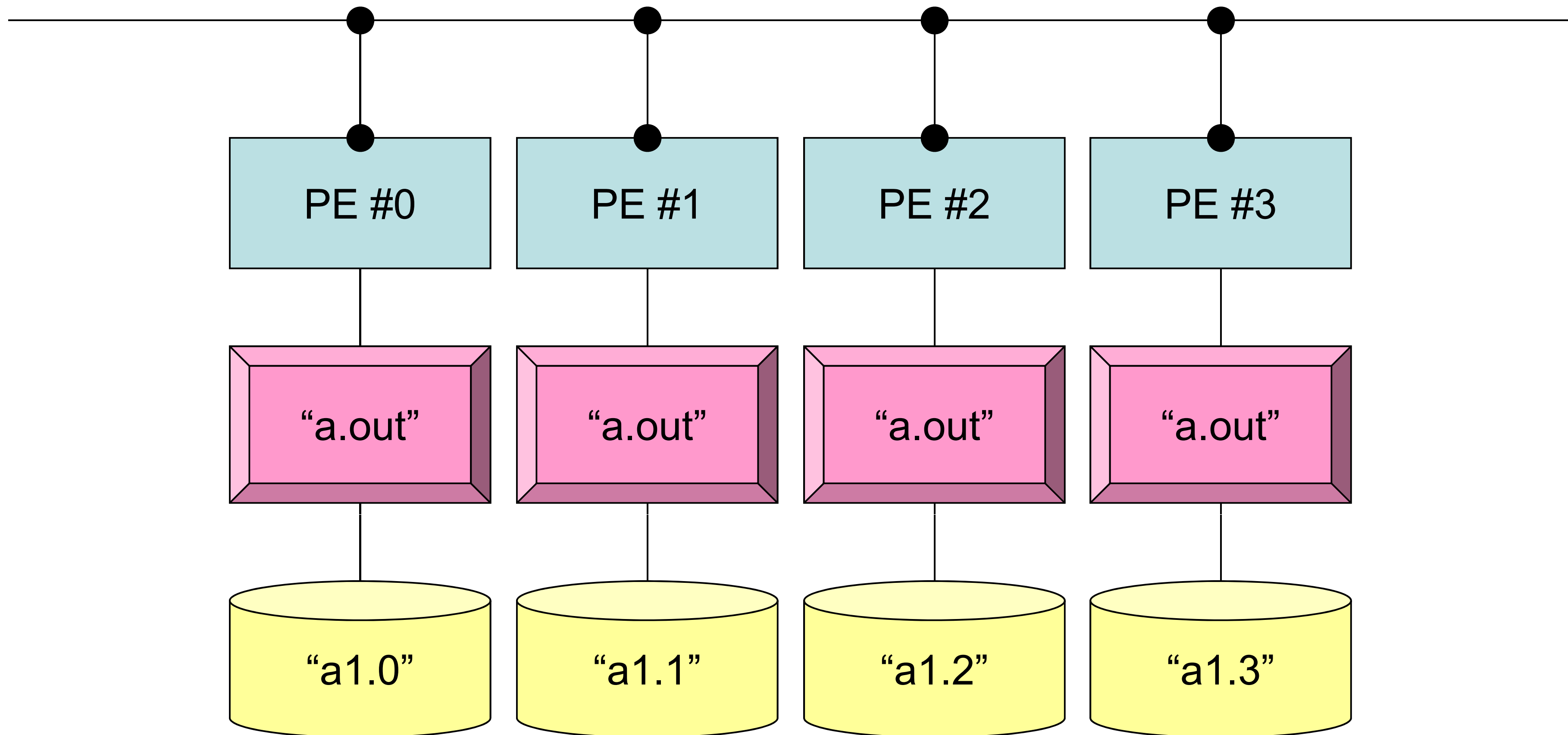
<$FVM>/S1/file.c

```
int main(int argc, char **argv){
        int i;
        int PeTot, MyRank;
        MPI_Comm SolverComm;
        double vec[8];
        char FileName[80];
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sprintf(FileName, "a1.%d", MyRank);

        fp = fopen(FileName, "r");
        if(fp == NULL) MPI_Abort(MPI_COMM_WORLD, -1);
        for(i=0;i<8;i++){
                fscanf(fp, "%lf", &vec[i]);        }

        for(i=0;i<8;i++){
                printf("%5d%5d%10.0f¥n", MyRank, i+1, vec[i]);
        }
        MPI_Finalize();
        return 0;
}
```

Hello とそんなに変わらない

「局所番号（0~7）」で読み込む

SPMDの典型例




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

分散ファイル読み込み：可変長(1/2)

```
>$ cd <$FVM>/S1
>$ ls a2.*
   a2.0 a2.1 a2.2 a2.3
>$ cat a2.0
  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

>$ mpicc -Os -noparallel file2.c
>$ mpif90 -Oss -noparallel file2.f

>$ 実行:4プロセス go4.sh
```

分散ファイルの読み込み：可変長（2/2）

<$FVM>/S1/file2.c

```
int main(int argc, char **argv){
        int i, int PeTot, MyRank;
        MPI_Comm SolverComm;
        double *vec, *vec2, *vecg;
        int num;
        double sum0, sum;
        char filename[80];
        FILE *fp;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        sprintf(filename, "a2.%d", MyRank);
        fp = fopen(filename, "r");
        assert(fp != NULL);

        fscanf(fp, "%d", &num);
        vec = malloc(num * sizeof(double));
        for(i=0;i<num;i++){fscanf(fp, "%lf", &vec[i]);}

        for(i=0;i<num;i++){
                printf(" %5d%5d%5d%10.0f¥n", MyRank, i+1, num, vec[i]);}

        MPI_Finalize();
}
```

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

- 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（Point-to-Point Communication）

有限体積法：全体マトリクスの生成

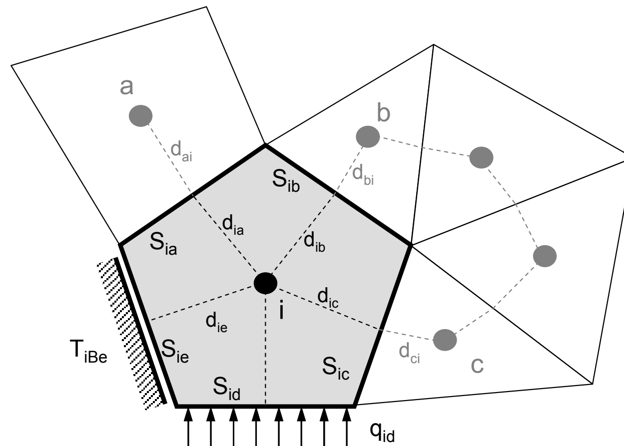
要素iに関する釣り合い

$$\left[\sum_k \frac{S_{ik}}{\frac{d_{ik}}{\lambda_i}+\frac{d_{ki}}{\lambda_k}}+\sum_e \frac{S_{ie}}{\frac{d_{ie}}{\lambda_i}}\right]T_i-\left[\sum_k \frac{S_{ik}}{\frac{d_{ik}}{\lambda_i}+\frac{d_{ki}}{\lambda_k}}T_k\right]=\sum_d S_{id}\dot{q}_{id}+V_i\dot{Q}_i+\sum_e \frac{S_{ie}}{\frac{d_{ie}}{\lambda_i}}T_{iBe}$$

D（対角成分） → **隣接要素の情報必要**

AMAT（非対角成分） → **隣接要素の情報必要**

BFORCE（右辺） → **自分自身（要素i）の情報のみ必要**



前処理付き共役勾配法
Preconditioned Conjugate Gradient Method (CG)

```
Compute r(0)= b-[A]x(0)
for i= 1, 2, …
    solve [M]z(i-1)= r(i-1)
    ρi-1= r(i-1) z(i-1)
    if i=1
      p(1)= z(0)
     else
      βi-1= ρi-1/ρi-2
      p(i)= z(i-1) + βi-1 p(i)
    endif
    q(i)= [A]p(i)
    αi  = ρi-1/p(i)q(i)
    x(i)= x(i-1) + αip(i)
    r(i)= r(i-1) - αiq(i)
    check convergence |r|
end
```

前処理：対角スケーリング

行列ベクトル積：
領域外の値が必要

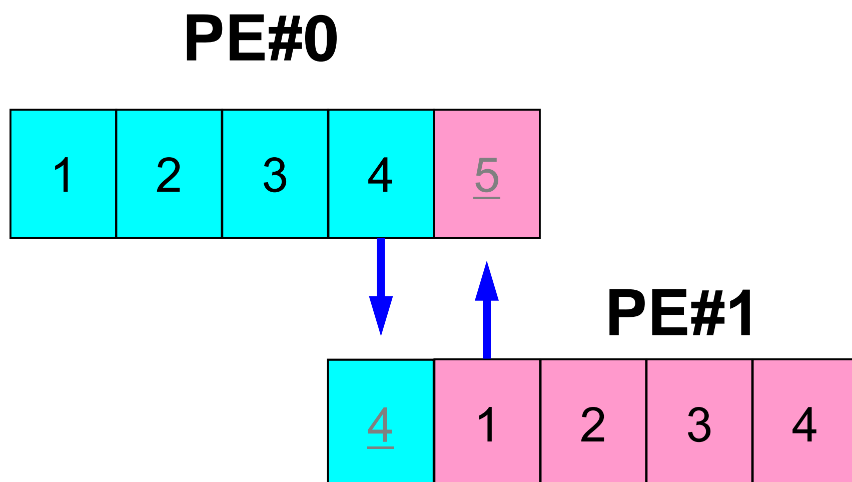
1対1通信とは？

- グループ通信：Collective Communication
  - MPI_Reduce, MPI_Scatter/Gather など
  - 同じコミュニケータ内の全プロセスと通信する
  - 適用分野
    - 境界要素法，スペクトル法，分子動力学等グローバルな相互作用のある手法
    - 内積，最大値などのオペレーション
- 1対1通信：Point-to-Point
  - MPI_Send, MPI_Receive
  - 特定のプロセスとのみ通信がある
    - 隣接領域
  - 適用分野
    - 差分法，有限要素法などローカルな情報を使う手法

1対1通信の方法

- **`MPI_Send`**，　**`MPI_Recv`**というサブルーチンがある。
- しかし，これらは「ブロッキング（blocking）」通信サブルーチンで，デッドロック（dead lock）を起こしやすい。
  - 受信（`Recv`）の完了が確認されないと，送信（`Send`）が終了しない
- もともと非常に「secureな」通信を保障するために，MPI仕様の中に入れられたものであるが，実用上は不便この上ない。
  - したがって実際にアプリケーションレベルで使用されることはほとんど無い（と思う）。
  - 将来にわたってこの部分が改正される予定はないらしい。
- 「そういう機能がある」ということを心の片隅においておいてください。

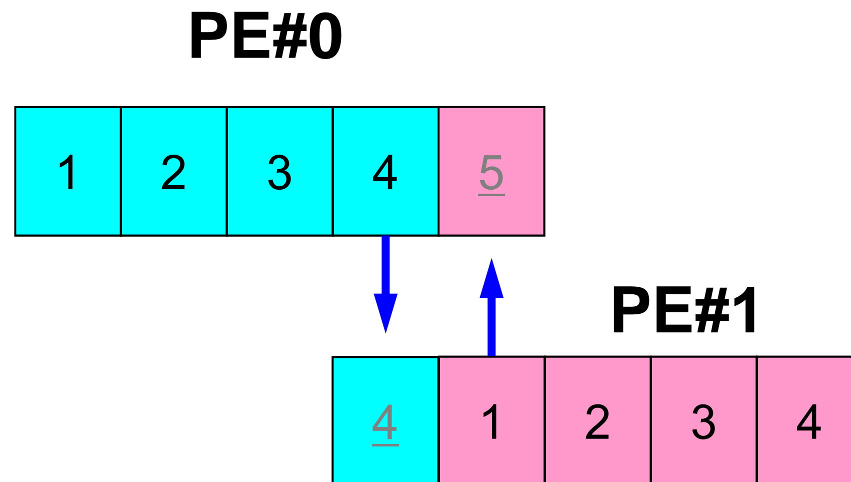
MPI_Send/MPI_Recv




```
if (my_rank.eq.0) NEIB_ID=1
if (my_rank.eq.1) NEIB_ID=0

…
call MPI_Send (NEIB_ID, arg’s)
call MPI_Recv (NEIB_ID, arg’s)
…
```

- 例えば先ほどの例で言えば，このようにしたいところであるが，このようなプログラムを作るとMPI_Send/MPI_Recvのところで止まってしまう。
  - 動く場合もある

MPI_Send/MPI_Recv（続き）

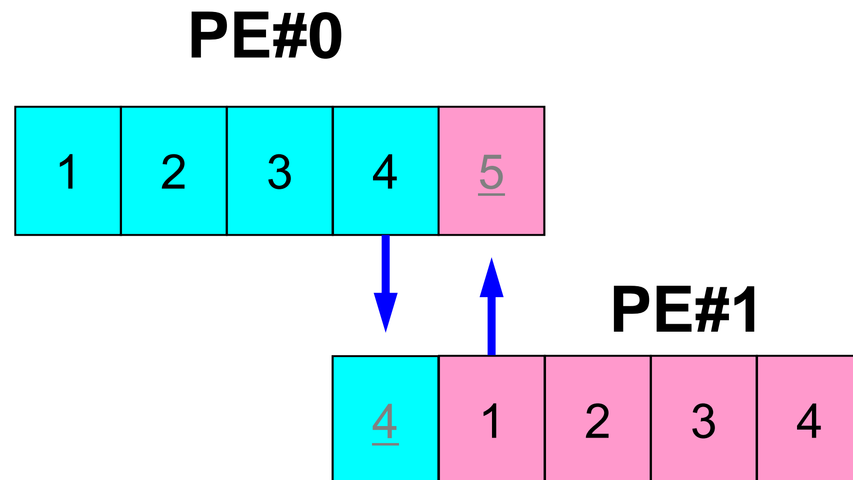



```
if (my_rank.eq.0) NEIB_ID=1
if (my_rank.eq.1) NEIB_ID=0

…
if (my_rank.eq.0) then
  call MPI_Send (NEIB_ID, arg's)
  call MPI_Recv (NEIB_ID, arg's)
endif

if (my_rank.eq.1) then
  call MPI_Recv (NEIB_ID, arg's)
  call MPI_Send (NEIB_ID, arg's)
endif

…
```

- このようにすれば，動く。

1対1通信の方法(実際どうするか)

- MPI_Isend, MPI_Irecv, という「ブロッキングしない(non-blocking)」サブルーチンがある。これと,同期のための「MPI_Waitall」を組み合わせる。
- MPI_Sendrecv というサブルーチンもある(後述)。




```
if (my_rank.eq.0) NEIB_ID=1
if (my_rank.eq.1) NEIB_ID=0

…
call MPI_Isend (NEIB_ID, arg's)
call MPI_Irecv (NEIB_ID, arg's)
…
call MPI_Waitall (for IRECV)
…
call MPI_Waitall (for ISEND)
```

IsendとIrecvで同じ通信識別子を使って,更に整合性が取れるのであればWaitallは一箇所でもOKです(後述)

MPI_Isend

- 送信バッファ「sendbuf」内の，連続した「count」個の送信メッセージを，タグ「tag」を付けて，コミュニケータ内の，「dest」に送信する。「MPI_Waitall」を呼ぶまで，送信バッファの内容を更新してはならない。

- **`MPI_Isend (sendbuf,count,datatype,dest,tag,comm,request)`**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **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
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，送信メッセージの種類を区別するときに使用。通常は「0」でよい。同じメッセージタグ番号同士で通信。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_Waitallで使用。（配列：サイズは同期する必要のある「MPI_Isend」呼び出し数（通常は隣接プロセス数など））

通信識別子（request handle）： request

- **MPI_Isend**
  (sendbuf,count,datatype,dest,tag,comm,request)
  - **sendbuf** 任意 I 送信バッファの先頭アドレス,
  - **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
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，送信メッセージの種類を区別するときに使用。通常は「0」でよい。同じメッセージタグ番号同士で通信。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_Waitallで使用。（配列：サイズは同期する必要のある「MPI_Isend」呼び出し数（通常は隣接プロセス数など））

- 記憶領域を確保するだけで良い

MPI_Irecv

- 受信バッファ「recvbuf」内の，連続した「count」個の送信メッセージを，タグ「tag」を付けて，コミュニケータ内の，「dest」から受信する。「MPI_Waitall」を呼ぶまで，受信バッファの内容を利用した処理を実施してはならない。

- **MPI_Irecv (recvbuf,count,datatype,dest,tag,comm,request)**
  - **recvbuf** 任意 I 受信バッファの先頭アドレス，
  - **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
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，受信メッセージの種類を区別するときに使用。通常は「0」でよい。同じメッセージタグ番号同士で通信。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_Waitallで使用。（配列：サイズは同期する必要のある「MPI_Irecv」呼び出し数（通常は隣接プロセス数など））

MPI_Waitall

- 1対1非ブロッキング通信関数である「MPI_Isend」と「MPI_Irecv」を使用した場合，プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_Waitall」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_Waitall」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば，「MPI_Isend」と「MPI_Irecv」を同時に同期してもよい。
  - 「MPI_Isend/Irecv」で同じ通信識別子を使用すること
- 「MPI_Barrier」と同じような機能であるが，代用はできない。
  - 実装にもよるが，「request」，「status」の内容が正しく更新されず，何度も「MPI_Isend/Irecv」を呼び出すと処理が遅くなる，というような経験もある。

- **MPI_Waitall (count,request,status)**
  - <u>count</u>　整数　I　同期する必要のある「MPI_ISEND」，「MPI_RECV」呼び出し数。
  - <u>request</u>　整数　I/O　通信識別子。「MPI_ISEND」，「MPI_IRECV」で利用した識別子名に対応。（配列サイズ：（count））
  - <u>status</u>　整数　O　状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE,count））
    MPI_STATUS_SIZE: “mpif.h”,”mpi.h”で定められるパラメータ

状況オブジェクト配列（status object）：status

- **MPI_Waitall (count,request,status)**
  - count　整数　I　同期する必要のある「MPI_Isend」，「MPI_Irecv」呼び出し数。
  - request　整数　I/O　通信識別子。「MPI_Isend」，「MPI_Irecv」で利用した識別子名に対応。（配列サイズ：（count））
  - status　整数　O　状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE,count））MPI_STATUS_SIZE: “mpif.h”，”mpi.h”で定められるパラメータ
  - ierr　整数　O　完了コード
- 予め記憶領域を確保しておくだけでよい

MPI_Sendrecv

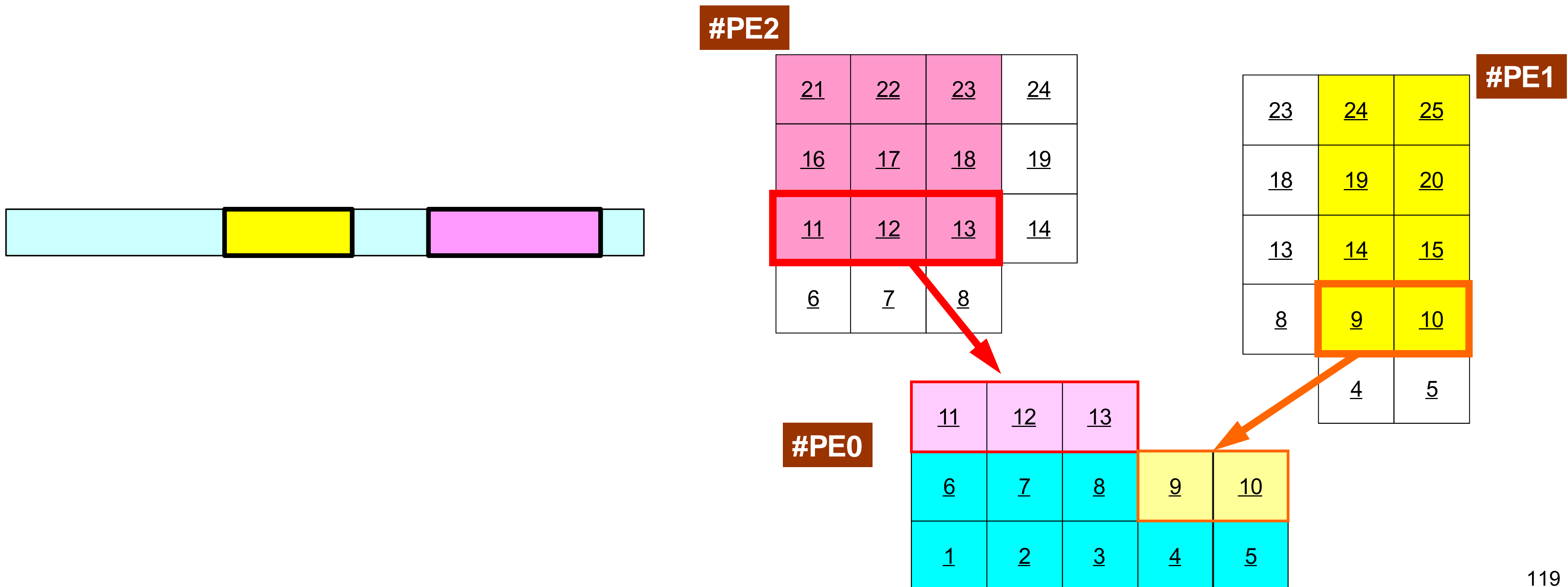
- MPI_Send+MPI_Recv

- **MPI_Sendrecv (sendbuf,sendcount,sendtype,dest,sendtag,recvbuf, recvcount,recvtype,source,recvtag,comm,status)**
  - sendbuf　任意　I　送信バッファの先頭アドレス,
  - sendcount　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
  - dest　整数　I　宛先プロセスのアドレス（ランク）
  - sendtag　整数　I　送信用メッセージタグ，送信メッセージの種類を区別するときに使用。通常は「0」でよい。
  - recvbuf　任意　I　受信バッファの先頭アドレス,
  - recvcount　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
  - source　整数　I　送信元プロセスのアドレス（ランク）
  - sendtag　整数　I　受信用メッセージタグ，送信メッセージの種類を区別するときに使用。通常は「0」でよい。同じメッセージタグ番号同士で通信。
  - comm　整数　I　コミュニケータを指定する
  - status　整数　O　状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE））MPI_STATUS_SIZE: “mpif.h”で定められるパラメータ

RECV(受信):外点への受信

受信バッファに隣接プロセスから連続したデータを受け取る

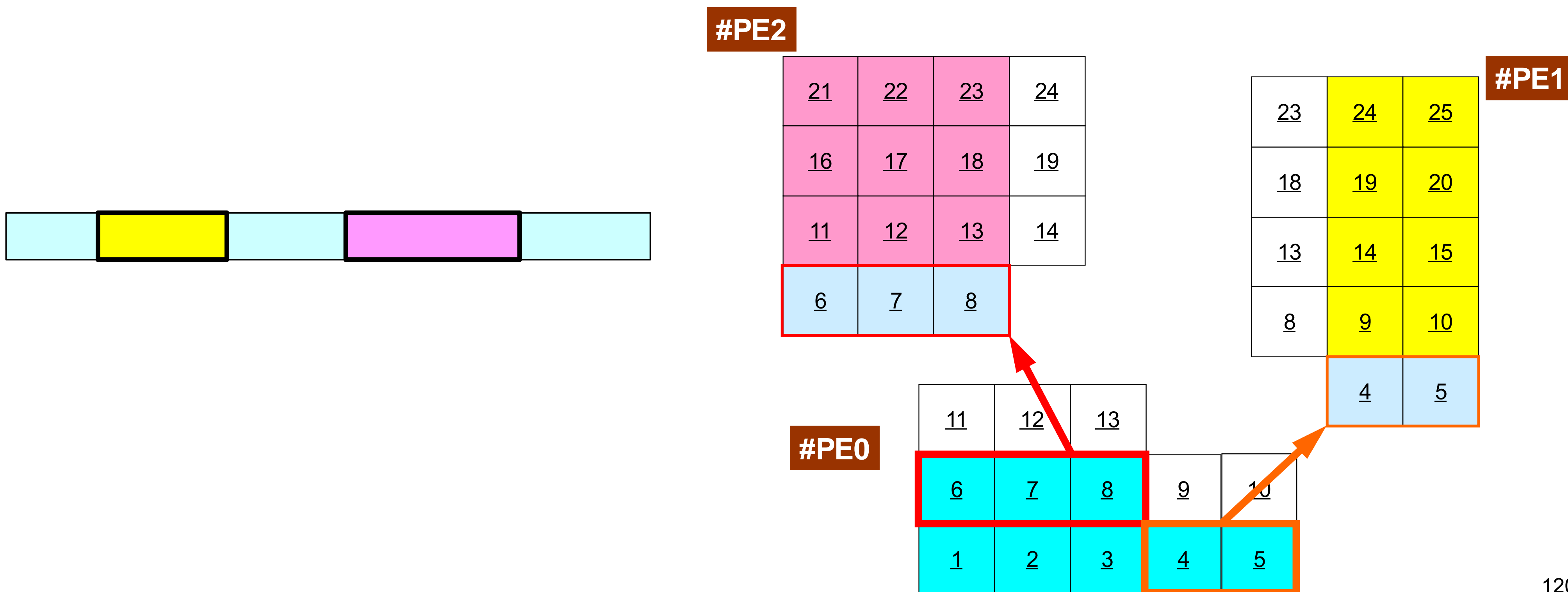
- **MPI_Irecv (recvbuf,count,datatype,dest,tag,comm,request)**
  - recvbuf 任意 I 受信バッファの先頭アドレス,
  - count 整数 I メッセージのサイズ
  - datatype 整数 I メッセージのデータタイプ
  - dest 整数 I 宛先プロセスのアドレス(ランク)



SEND(送信):境界点の送信

送信バッファの連続したデータを隣接プロセスに送る

- **MPI_Isend (sendbuf,count,datatype,dest,tag,comm,request)**
  - sendbuf 任意 I 送信バッファの先頭アドレス,
  - 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
  - dest 整数 I 宛先プロセスのアドレス(ランク)



通信識別子，状況オブジェクト配列の定義の仕方（FORTRAN）

- **MPI_Isend: request**
- **MPI_Irecv: request**
- **MPI_Waitall: request, status**

```
integer request(NEIBPETOT)
integer status (MPI_STAUTS_SIZE,NEIBPETOT)
```

- **MPI_Sendrecv: status**

```
integer status (MPI_STATUS_SIZE)
```

通信識別子，状況オブジェクト配列の定義の仕方（C）：特殊な変数の型がある

- **MPI_Isend: request**
- **MPI_Irecv: request**
- **MPI_Waitall: request, status**

```
MPI_Status  *StatSend, *StatRecv;
MPI_Request *RequestSend, *RequestRecv;
...
StatSend = malloc(sizeof(MPI_Status) * NEIBpetot);
StatRecv = malloc(sizeof(MPI_Status) * NEIBpetot);
RequestSend = malloc(sizeof(MPI_Request) * NEIBpetot);
RequestRecv = malloc(sizeof(MPI_Request) * NEIBpetot);
```

- **MPI_Sendrecv: status**

```
MPI_Status *Status;
...
Status = malloc(sizeof(MPI_Status));
```

利用例(1):スカラー送受信

- PE#0, PE#1間 で8バイト実数VALの値を交換する。

```
if (my_rank.eq.0) NEIB= 1
if (my_rank.eq.1) NEIB= 0

call MPI_Isend (VAL    ,1,MPI_DOUBLE_PRECISION,NEIB,…,req_send,…)
call MPI_Irecv (VALtemp,1,MPI_DOUBLE_PRECISION,NEIB,…,req_recv,…)
call MPI_Waitall (…,req_recv,stat_recv,…):受信バッファ VALtemp を利用可能
call MPI_Waitall (…,req_send,stat_send,…):送信バッファ VAL を変更可能
VAL= VALtemp
```

```
if (my_rank.eq.0) NEIB= 1
if (my_rank.eq.1) NEIB= 0

call MPI_Sendrecv (VAL    ,1,MPI_DOUBLE_PRECISION,NEIB,…              &
                   VALtemp,1,MPI_DOUBLE_PRECISION,NEIB,…, status,…)
VAL= VALtemp
```

受信バッファ名を「VAL」にしても動く場合はあるが, お勧めはしない。

利用例(1):スカラー送受信 C

Isend/Irecv/Waitall

```
$> cd <$FVM>/S2
$> mpicc -Os -noparallel ex1-1.c
$> mpif90 -Oss -noparallel ex1-1.f
$> 実行(2プロセス) go2.sh
```

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char **argv){
        int neib, MyRank, PeTot;
        double VAL, VALx;
        MPI_Status *StatSend, *StatRecv;
        MPI_Request *RequestSend, *RequestRecv;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
        StatSend = malloc(sizeof(MPI_Status) * 1);
        StatRecv = malloc(sizeof(MPI_Status) * 1);
        RequestSend = malloc(sizeof(MPI_Request) * 1);
        RequestRecv = malloc(sizeof(MPI_Request) * 1);

        if(MyRank == 0) {neib= 1; VAL= 10.0;}
        if(MyRank == 1) {neib= 0; VAL= 11.0;}

        MPI_Isend(&VAL , 1, MPI_DOUBLE, neib, 0, MPI_COMM_WORLD, &RequestSend[0]);
        MPI_Irecv(&VALx, 1, MPI_DOUBLE, neib, 0, MPI_COMM_WORLD, &RequestRecv[0]);
        MPI_Waitall(1, RequestRecv, StatRecv);
        MPI_Waitall(1, RequestSend, StatSend);

        VAL=VALx;
        MPI_Finalize();
        return 0; }
```

利用例(1)：スカラー送受信 C

SendRecv

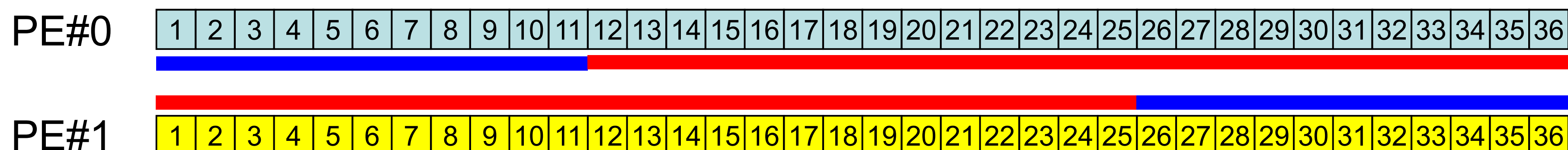
```
$> cd <$FVM>/S2
$> mpicc -Os -noparallel ex1-2.c
$> mpif90 -Oss -noparallel ex1-2.f
$> 実行(2プロセス) go2.sh
```

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char **argv){
        int neib;
        int MyRank, PeTot;
        double VAL, VALtemp;
        MPI_Status *StatSR;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);

        if(MyRank == 0) {neib= 1; VAL= 10.0;}
        if(MyRank == 1) {neib= 0; VAL= 11.0;}

        StatSR = malloc(sizeof(MPI_Status));

        MPI_Sendrecv(&VAL    , 1, MPI_DOUBLE, neib, 0,
                     &VALtemp, 1, MPI_DOUBLE, neib, 0, MPI_COMM_WORLD, StatSR);
        VAL=VALtemp;

        MPI_Finalize();
        return 0;
}
```

利用例(2):配列の送受信(1/4)

- PE#0, PE#1間 で8バイト実数配列VECの値を交換する。
- PE#0⇒PE#1
  - PE#0:VEC(1)~VEC(11)の値を送る(長さ:11)
  - PE#1:VEV(26)~VEC(36)の値として受け取る
- PE#1⇒PE#0
  - PE#1:VEC(1)~VEC(25)の値を送る(長さ:25)
  - PE#0:VEV(12)~VEC(36)の値として受け取る

- 演習:プログラムを作成して見よう!



演　習

- VEC(:)の初期状態を以下のようにする:
  - PE#0　VEC(1-36)= 101,102,103,~,135,136
  - PE#1　VEC(1-36)= 201,202,203,~,235,236

- 次ページのような結果になることを確認せよ

- 以下のそれぞれを使用したプログラムを作成せよ
  - MPI_Isend/Irecv/Waitall
  - MPI_Sendrecv

予測される結果

```
0 #BEFORE#  1        101.
0 #BEFORE#  2        102.
0 #BEFORE#  3        103.
0 #BEFORE#  4        104.
0 #BEFORE#  5        105.
0 #BEFORE#  6        106.
0 #BEFORE#  7        107.
0 #BEFORE#  8        108.
0 #BEFORE#  9        109.
0 #BEFORE# 10        110.
0 #BEFORE# 11        111.
0 #BEFORE# 12        112.
0 #BEFORE# 13        113.
0 #BEFORE# 14        114.
0 #BEFORE# 15        115.
0 #BEFORE# 16        116.
0 #BEFORE# 17        117.
0 #BEFORE# 18        118.
0 #BEFORE# 19        119.
0 #BEFORE# 20        120.
0 #BEFORE# 21        121.
0 #BEFORE# 22        122.
0 #BEFORE# 23        123.
0 #BEFORE# 24        124.
0 #BEFORE# 25        125.
0 #BEFORE# 26        126.
0 #BEFORE# 27        127.
0 #BEFORE# 28        128.
0 #BEFORE# 29        129.
0 #BEFORE# 30        130.
0 #BEFORE# 31        131.
0 #BEFORE# 32        132.
0 #BEFORE# 33        133.
0 #BEFORE# 34        134.
0 #BEFORE# 35        135.
0 #BEFORE# 36        136.
```

```
0 #AFTER #  1        101.
0 #AFTER #  2        102.
0 #AFTER #  3        103.
0 #AFTER #  4        104.
0 #AFTER #  5        105.
0 #AFTER #  6        106.
0 #AFTER #  7        107.
0 #AFTER #  8        108.
0 #AFTER #  9        109.
0 #AFTER # 10        110.
0 #AFTER # 11        111.
0 #AFTER # 12        201.
0 #AFTER # 13        202.
0 #AFTER # 14        203.
0 #AFTER # 15        204.
0 #AFTER # 16        205.
0 #AFTER # 17        206.
0 #AFTER # 18        207.
0 #AFTER # 19        208.
0 #AFTER # 20        209.
0 #AFTER # 21        210.
0 #AFTER # 22        211.
0 #AFTER # 23        212.
0 #AFTER # 24        213.
0 #AFTER # 25        214.
0 #AFTER # 26        215.
0 #AFTER # 27        216.
0 #AFTER # 28        217.
0 #AFTER # 29        218.
0 #AFTER # 30        219.
0 #AFTER # 31        220.
0 #AFTER # 32        221.
0 #AFTER # 33        222.
0 #AFTER # 34        223.
0 #AFTER # 35        224.
0 #AFTER # 36        225.
```

```
1 #BEFORE#  1        201.
1 #BEFORE#  2        202.
1 #BEFORE#  3        203.
1 #BEFORE#  4        204.
1 #BEFORE#  5        205.
1 #BEFORE#  6        206.
1 #BEFORE#  7        207.
1 #BEFORE#  8        208.
1 #BEFORE#  9        209.
1 #BEFORE# 10        210.
1 #BEFORE# 11        211.
1 #BEFORE# 12        212.
1 #BEFORE# 13        213.
1 #BEFORE# 14        214.
1 #BEFORE# 15        215.
1 #BEFORE# 16        216.
1 #BEFORE# 17        217.
1 #BEFORE# 18        218.
1 #BEFORE# 19        219.
1 #BEFORE# 20        220.
1 #BEFORE# 21        221.
1 #BEFORE# 22        222.
1 #BEFORE# 23        223.
1 #BEFORE# 24        224.
1 #BEFORE# 25        225.
1 #BEFORE# 26        226.
1 #BEFORE# 27        227.
1 #BEFORE# 28        228.
1 #BEFORE# 29        229.
1 #BEFORE# 30        230.
1 #BEFORE# 31        231.
1 #BEFORE# 32        232.
1 #BEFORE# 33        233.
1 #BEFORE# 34        234.
1 #BEFORE# 35        235.
1 #BEFORE# 36        236.
```

```
1 #AFTER #  1        201.
1 #AFTER #  2        202.
1 #AFTER #  3        203.
1 #AFTER #  4        204.
1 #AFTER #  5        205.
1 #AFTER #  6        206.
1 #AFTER #  7        207.
1 #AFTER #  8        208.
1 #AFTER #  9        209.
1 #AFTER # 10        210.
1 #AFTER # 11        211.
1 #AFTER # 12        212.
1 #AFTER # 13        213.
1 #AFTER # 14        214.
1 #AFTER # 15        215.
1 #AFTER # 16        216.
1 #AFTER # 17        217.
1 #AFTER # 18        218.
1 #AFTER # 19        219.
1 #AFTER # 20        220.
1 #AFTER # 21        221.
1 #AFTER # 22        222.
1 #AFTER # 23        223.
1 #AFTER # 24        224.
1 #AFTER # 25        225.
1 #AFTER # 26        101.
1 #AFTER # 27        102.
1 #AFTER # 28        103.
1 #AFTER # 29        104.
1 #AFTER # 30        105.
1 #AFTER # 31        106.
1 #AFTER # 32        107.
1 #AFTER # 33        108.
1 #AFTER # 34        109.
1 #AFTER # 35        110.
1 #AFTER # 36        111.
```

利用例（2）：配列の送受信（2/4）

```
if (my_rank.eq.0) then
  call MPI_Isend (VEC( 1),11,MPI_DOUBLE_PRECISION,1,…,req_send,…)
  call MPI_Irecv (VEC(12),25,MPI_DOUBLE_PRECISION,1,…,req_recv,…)
endif

if (my rank.eq.1) then
  call MPI_Isend (VEC( 1),25,MPI_DOUBLE_PRECISION,0,…,req_send,…)
  call MPI_Irecv (VEC(26),11,MPI_DOUBLE_PRECISION,0,…,req_recv,…)
endif

call MPI_Waitall (…,req_recv,stat_recv,…)
call MPI_Waitall (…,req_send,stat_send,…)
```

これでも良いが，操作が煩雑
SPMDらしくない
汎用性が無い

利用例(2):配列の送受信(3/4)

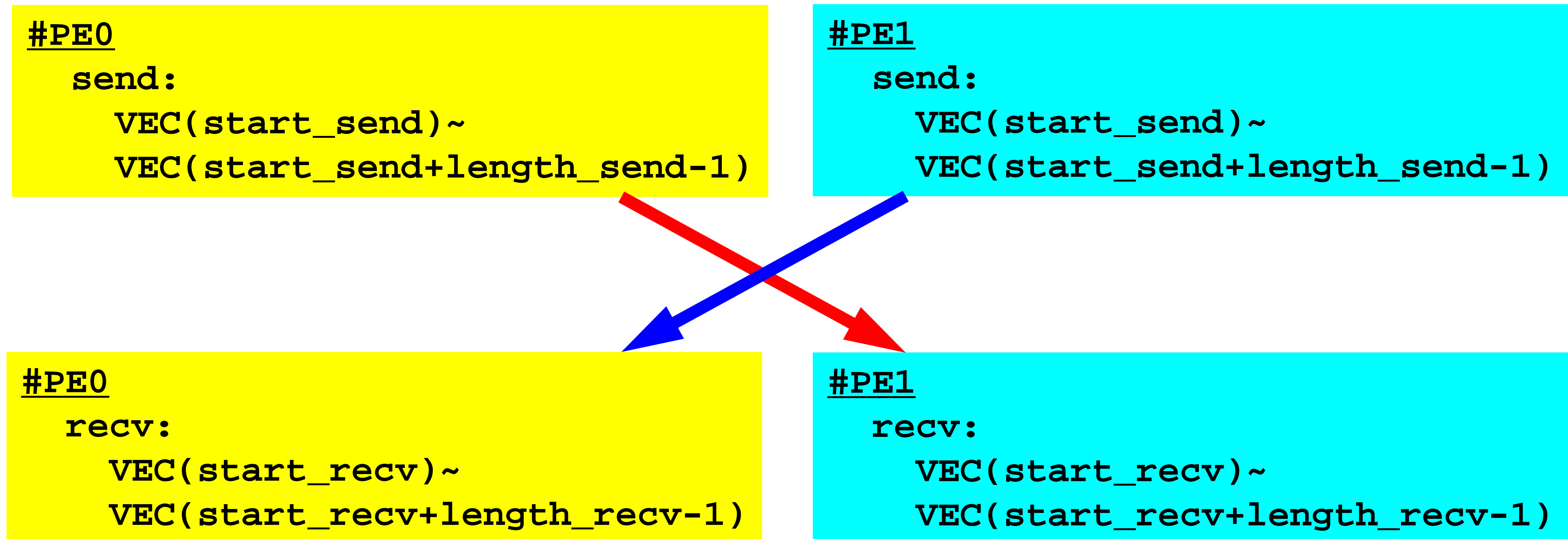
```
if (my_rank.eq.0) then
  NEIB= 1
   start_send=   1
  length_send=  11
   start_recv=  length_send + 1
  length_recv=  25
endif

if (my_rank.eq.1) then
  NEIB= 0
   start_send=  1
  length_send= 25
   start_recv= length_send + 1
  length_recv= 11
endif

call MPI_Isend                                                         &
(VEC(start_send),length_send,MPI_DOUBLE_PRECISION,NEIB,…,req_send,…)
call MPI_Irecv                                                         &
(VEC(start_recv),length_recv,MPI_DOUBLE_PRECISION,NEIB,…,req_recv,…)

call MPI_Waitall (…,req_recv,stat_recv,…)
call MPI_Waitall (…,req_send,stat_send,…)
```

一気にSPMDらしくなる

利用例(2):配列の送受信(4/4)

```
if (my_rank.eq.0) then
  NEIB= 1
   start_send=   1
  length_send=  11
   start_recv=  length_send + 1
  length_recv=  25
endif

if (my_rank.eq.1) then
  NEIB= 0
   start_send=  1
  length_send= 25
   start_recv= length_send + 1
  length_recv= 11
endif

call MPI_Sendrecv                                                   &
(VEC(start_send),length_send,MPI_DOUBLE_PRECISION,NEIB,…            &
 VEC(start_recv),length_recv,MPI_DOUBLE_PRECISION,NEIB,…, status,…)
```

配列の送受信:注意



- 送信側の「length_send」と受信側の「length_recv」は一致している必要がある。
  - PE#0⇒PE#1, PE#1⇒PE#0
- 「送信バッファ」と「受信バッファ」は別のアドレス

解答例(1/3)
Isend/Irecv/Waitall

```
$> cd <$FVM>/S2
$> mpicc -Os  -noparallel ex2a.c
$> mpif90 -Oss -noparallel ex2a.f

$> 実行(2プロセス) go2.sh
```

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

int main(int argc, char **argv){
        int i, neib;
        int MyRank, PeTot;
        double VEC[36];
        int Start_Send, Length_Send;
        int Start_Recv, Length_Recv;

        MPI_Status *StatSend, *StatRecv;
        MPI_Request *RequestSend, *RequestRecv;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &PeTot);
        MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

解答例（2/3）
Isend/Irecv/Waitall

```
     Start_Send= 1;

     if(MyRank == 0) {
       neib= 1;
       Length_Send= 11;
       Length_Recv= 25;
       for (i=0;i<36;i++){VEC[i]=100+i+1;}}

     if(MyRank == 1) {
       neib= 0;
       Length_Send= 25;
       Length_Recv= 11;
       for (i=0;i<36;i++){VEC[i]=200+i+1;}}

     Start_Recv= 1 + Length_Send;

     StatSend = malloc(sizeof(MPI_Status) * 1);
     StatRecv = malloc(sizeof(MPI_Status) * 1);
     RequestSend = malloc(sizeof(MPI_Request) * 1);
     RequestRecv = malloc(sizeof(MPI_Request) * 1);

     for (i=0;i<36;i++) {
       printf("%s%2d%5d%8.0f¥n", "### before", MyRank, i, VEC[i]);}
```

解答例(3/3)
Isend/Irecv/Waitall

```
        MPI_Isend(&VEC[Start_Send-1], Length_Send, MPI_DOUBLE, neib, 0,
                   MPI_COMM_WORLD, &RequestSend[0]);

        MPI_Irecv(&VEC[Start_Recv-1], Length_Recv, MPI_DOUBLE, neib, 0,
                   MPI_COMM_WORLD, &RequestRecv[0]);

        MPI_Waitall(1, RequestRecv, StatRecv);

        MPI_Waitall(1, RequestSend, StatSend);

        for (i=0;i<36;i++) {
          printf("%s%2d%5d%8.0f¥n", "### after ", MyRank, i, VEC[i]);}

        MPI_Finalize();
        return 0;
}
```

- 汎用性がある
- 「データが全て」という気がして来ないだろうか?