MPIによるプログラミング概要
C言語編（1/2）

中島 研吾
東京大学情報基盤センター

並列計算の意義・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: Weak Scaling
    - 同じ問題をN倍のCPUを使って「1/Nの時間で」解く: Strong Scaling

概要

- MPIとは
- MPIの基礎：Hello World
- 集団通信（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
    - https://www.mpi-forum.org/docs/
    - http://phase.hpcc.jp/phase/mpi-j/ml/mpi-j-html/contents.html
- 歴史
  - 1992　MPIフォーラム
    - https://www.mpi-forum.org/
  - 1994　MPI-1規格
  - 1997　MPI-2規格：MPI I/O他
  - 2012　MPI-3規格：非同期Collective通信他
- 実装
  - mpich　アルゴンヌ国立研究所
  - OpenMPI, MVAPICH 他
  - 各ベンダー
  - C/C++,FOTRAN,Java ; Unix，Linux，Windows，Mac OS

MPIとは（2/2）

- mpich（フリー）が広く使用されていた
  - 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
    - 現状はMPI 5.0
  - 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)

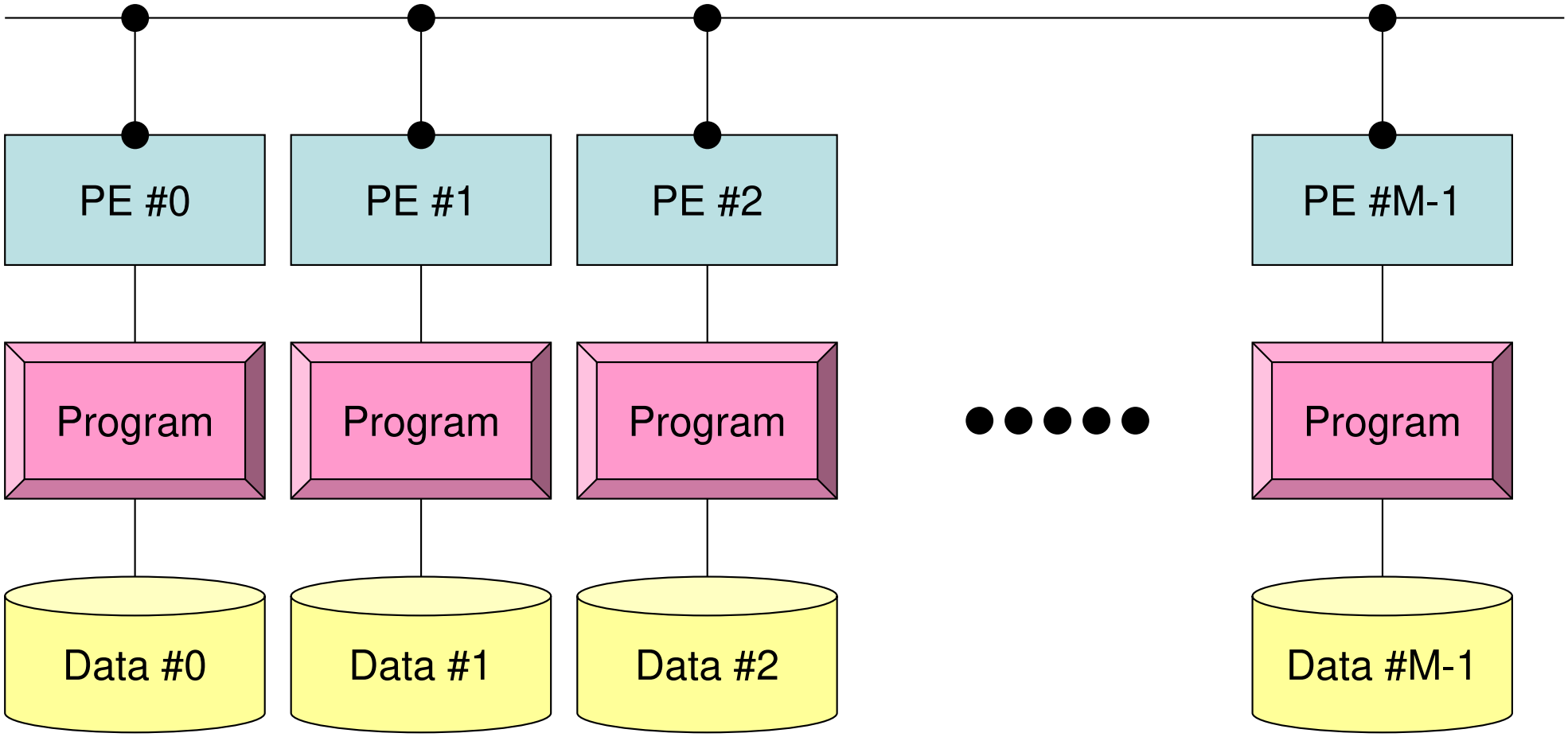
- 文法
  - 「MPI-1」の基本的な機能(10程度)について習熟する
    - MPI-2では色々と便利な機能があるが・・・
  - あとは自分に必要な機能について調べる，あるいは知っている人，知っていそうな人に尋ねる
- 実習の重要性
  - プログラミング
  - その前にまず実行してみること
- SPMD/SIMDのオペレーションに慣れること・・・「つかむ」こと
  - Single Program/Instruction Multiple Data
  - 基本的に各プロセスは「同じことをやる」が「データが違う」
    - 大規模なデータを分割し，各部分について各プロセス(プ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

```
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

用 語

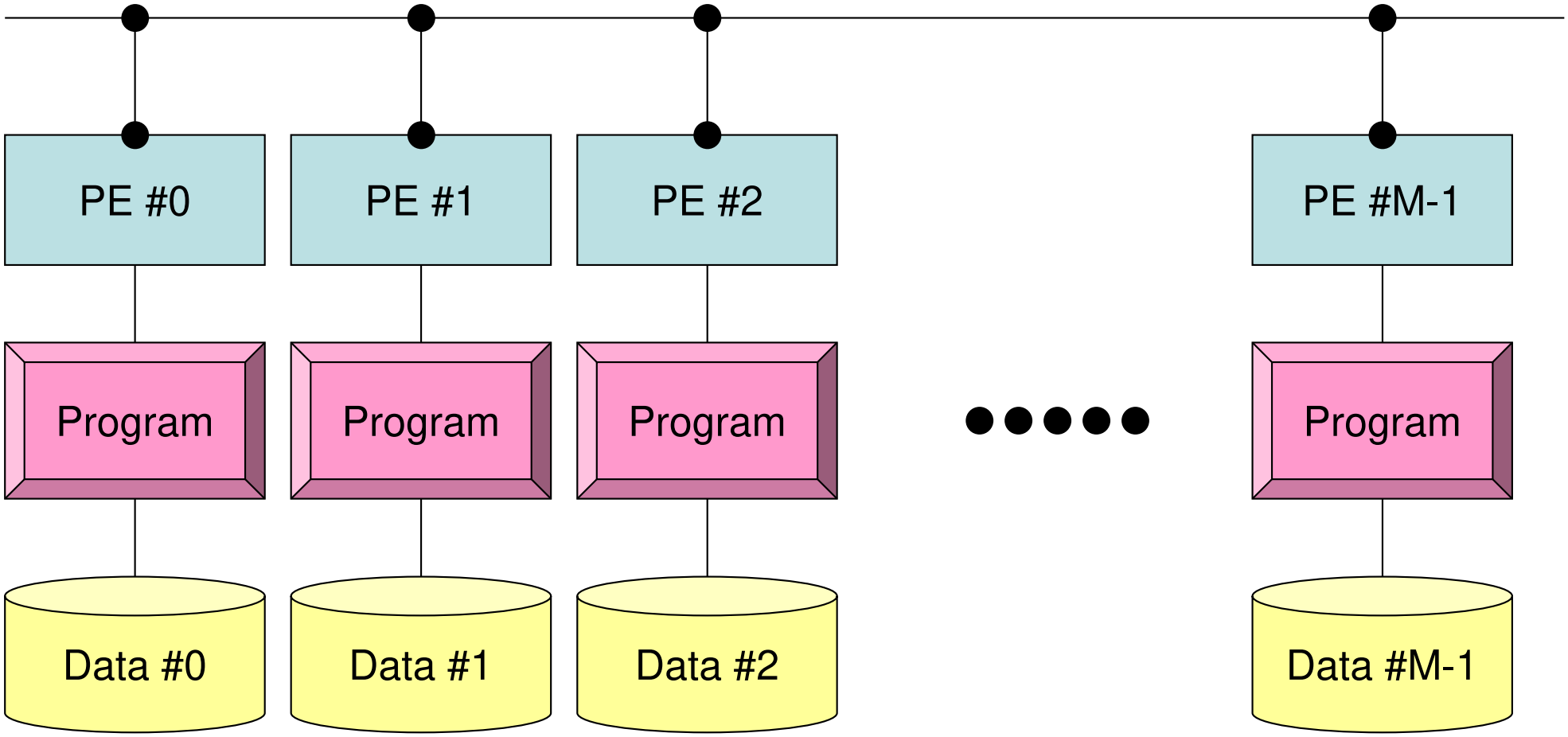
- プロセッサ，コア
  - ハードウェアとしての各演算装置。シングルコアではプロセッサ＝コア
- プロセス
  - MPI計算のための実行単位，ハードウェア的な「コア」とほぼ同義。
  - しかし1つの「プロセッサ・コア」で複数の「プロセス」を起動する場合もある（効率的ではないが）。
- PE（Processing Element）
  - 本来，「プロセッサ」の意味なのであるが，本講義では「プロセス」の意味で使う場合も多い。次項の「領域」とほぼ同義でも使用。
    - マルチコアの場合は：「コア＝PE」という意味で使うことが多い。
- 領域
  - 「プロセス」とほぼ同じ意味であるが，SPMD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

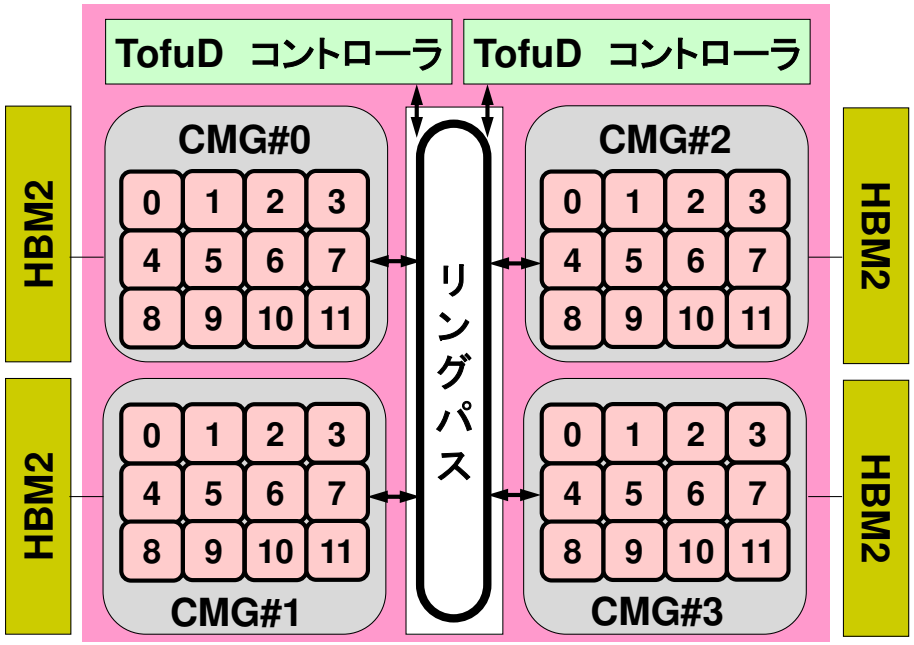
- 繰り返すが，決して難しいものではない。
- 以上のようなこともあって，文法を教える授業は2~3回程度で充分と考えている。
- とにかくSPMDの考え方を掴むこと！

授業・課題の予定（普段の講義）

- MPIサブルーチン機能
  - 環境管理
  - 集団通信
  - 1対1通信

- 90分×4～5コマ
  - 環境管理，集団通信（Collective Communication）
  - 1対1通信（Point-to-Point Communication）
  - ここまでできればあとはある程度自分で解決できます

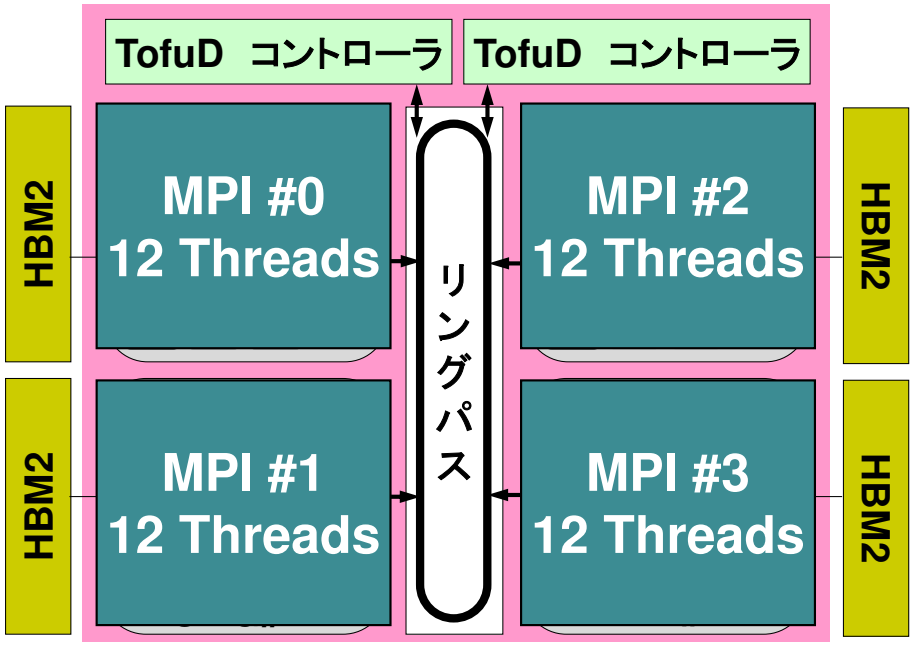
A64FXプロセッサ



プロセッサ名	A64FX
プロセッサ数(コア数)	1 (48+アシスタントコア2 or 4)
周波数	2.2 GHz
理論演算性能	3.3792 TFLOPS
メモリ容量	32 GiB
メモリ帯域幅	1,024 GB/s
L1 Cache	64 KiB/core (Inst/Data)
L2 Cache	8 MiB/CMG

- 4つのCMG（Core Memory Group），12計算コア/CMG
- NUMAアーキテクチャ（Non-Uniform Memory Access）
  - ✓ メモリは各CMGに搭載されていて独立，異なるCMGのローカルメモリ上のデータをアクセスすることは可能
  - ✓ ローカルメモリ上のデータを使って計算するのが効率的
- 大規模並列：各CMGに1-MPIプロセス（12-OpenMPスレッド），プロセッサ内4プロセスのハイブリッド推奨

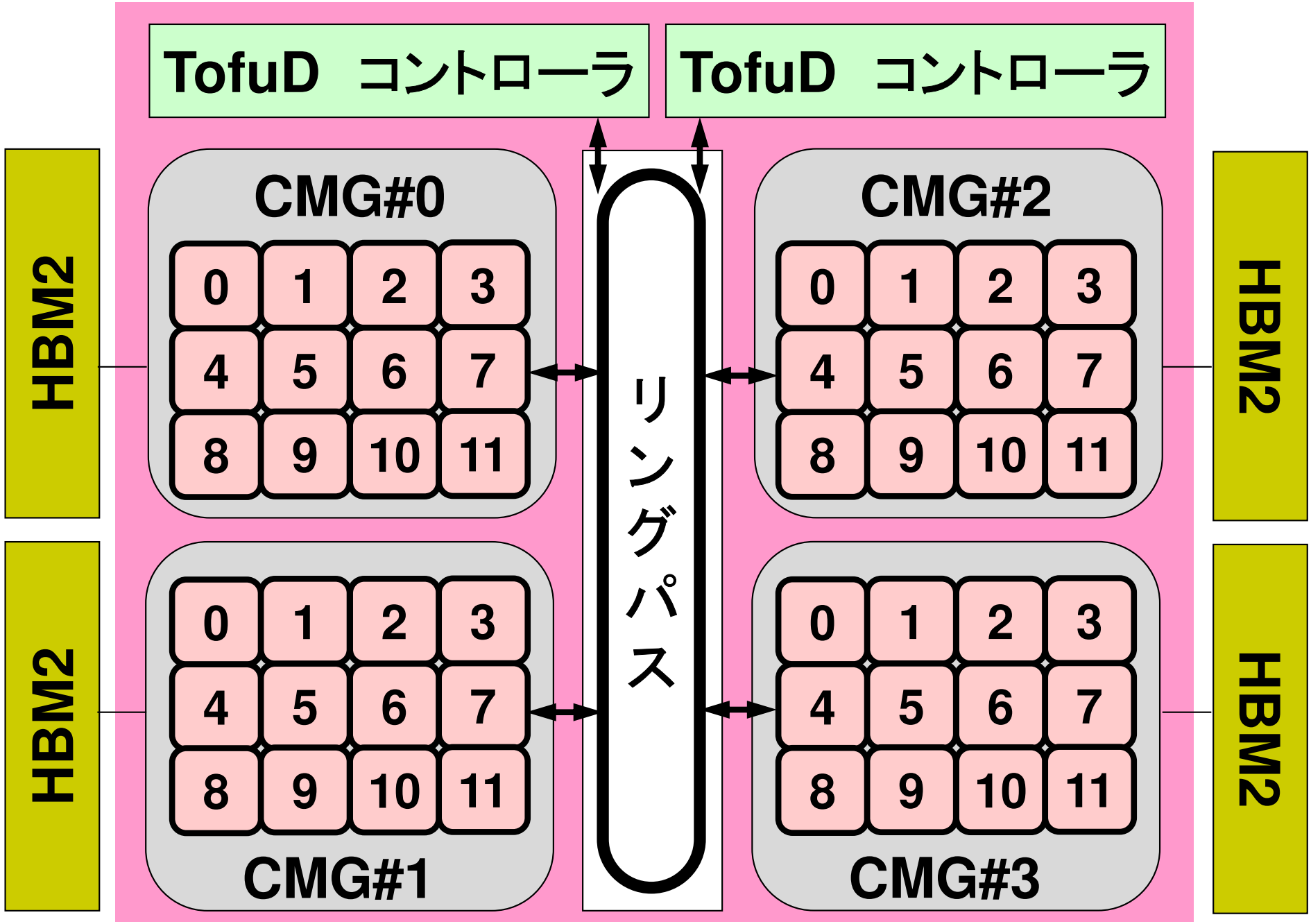
A64FXプロセッサ



プロセッサ名	A64FX
プロセッサ数(コア数)	1 (48+アシスタントコア2 or 4)
周波数	2.2 GHz
理論演算性能	3.3792 TFLOPS
メモリ容量	32 GiB
メモリ帯域幅	1,024 GB/s
L1 Cache	64 KiB/core (Inst/Data)
L2 Cache	8 MiB/CMG

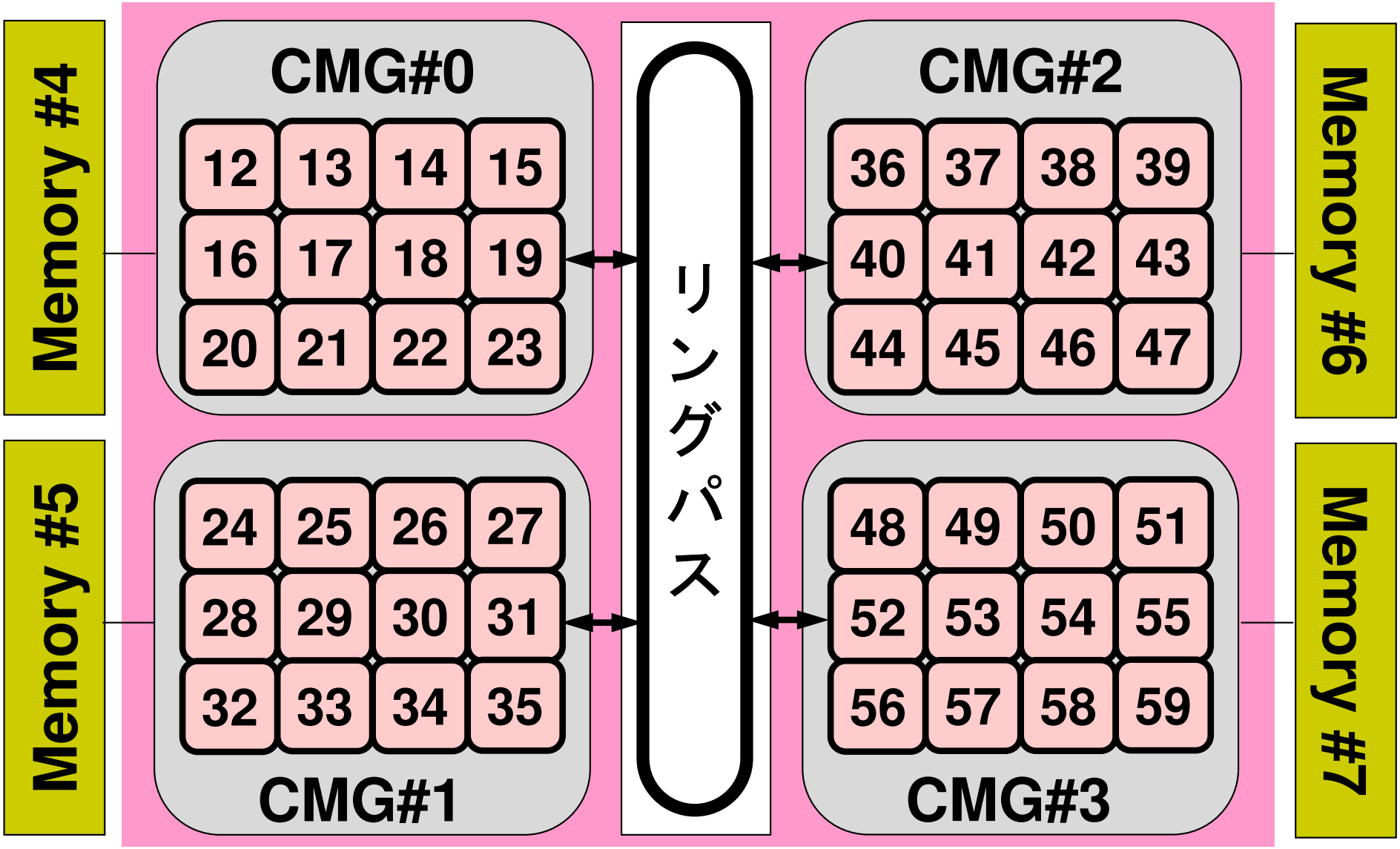
- 4つのCMG（Core Memory Group），12計算コア/CMG
- NUMAアーキテクチャ（Non-Uniform Memory Access）
  - ✓ メモリは各CMGに搭載されていて独立，異なるCMGのローカルメモリ上のデータをアクセスすることは可能
  - ✓ ローカルメモリ上のデータを使って計算するのが効率的
- 大規模並列：各CMGに1-MPIプロセス（12-OpenMPスレッド），プロセッサ内4プロセスのハイブリッド推奨

A64FX：CMG（Core Memory Group）



CMG番号，コア番号，メモリ番号（1/2）

CMG:#0-#3，Core:#12-59，Memory:#4-#7



- MPIとは
- MPIの基礎：Hello World
- 集団通信（Collective Communication）
- 1対1通信（Point-to-Point Communication）

ログイン，ディレクトリ作成 on Odyssey

```
ssh t00***@wisteria.cc.u-tokyo.ac.jp
```

ディレクトリ作成

```
>$ cd /work/gt00/t00***

>$ mkdir pFEM  （好きな名前でよい）
>$ cd pFEM

>$ module load fj    ログインしたら必ずこれをタイプ（コンパイラ）
```

このディレクトリを本講義では <$O-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

Odyssey

Your PC

ファイルコピー on Odyssey

FORTRANユーザー

```
>$ cd /work/gt00/t00XXX/pFEM
>$ module load fj
>$ cp /work/gt00/z30088/pFEM/F/s1-f.tar .
>$ tar xvf s1-f.tar
```

Cユーザー

```
>$ cd /work/gt00/t00XXX/pFEM
>$ module load fj
>$ cp /work/gt00/z30088/pFEM/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 <$O-S1> と呼ぶ。

<$O-S1> = <$O-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/c をコンパイルしてみよう！

```
>$ cd /work/gt00/t00XXX/pFEM/mpi/S1
>$ module load fj
>$ mpifrtpx -Kfast hello.f
>$ mpifccpx -Nclang -Kfast hello.c
```

FORTRAN

“mpifrtpx”:

Fujitsu 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

“mpifccpx”:

Fujitsu 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

Cコンパイラ:2種類のモード

tradモード (-Nnoclang オプション) （デフォルト）	• 京およびPRIMEHPC FX100以前のシステム向け富士通コンパイラをベースとする。 • tradモードは，従来の富士通コンパイラとの互換を重視する場合に適している。 • サポートしている仕様は，C89/C99/C11，OpenMP 3.1/OpenMP 4.5(一部) • オプション省略時（デフォルト）は，-Nnoclangオプション適用 • （概して）遅い
clangモード (-Nclang オプション)	• オープンソースソフトウェアであるClang/LLVMコンパイラをベースとする。 • clangモードは，最新言語仕様を使用したプログラムや，オープンソースソフトウェアを翻訳する場合に適している。 • サポートしている仕様は，C89/C99/C11，OpenMP 4.5/OpenMP 5.0(一部) • （概して）tradモードより速い，最適化すると差は縮まるが，今回はデフォルトでclangモード使用

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（56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

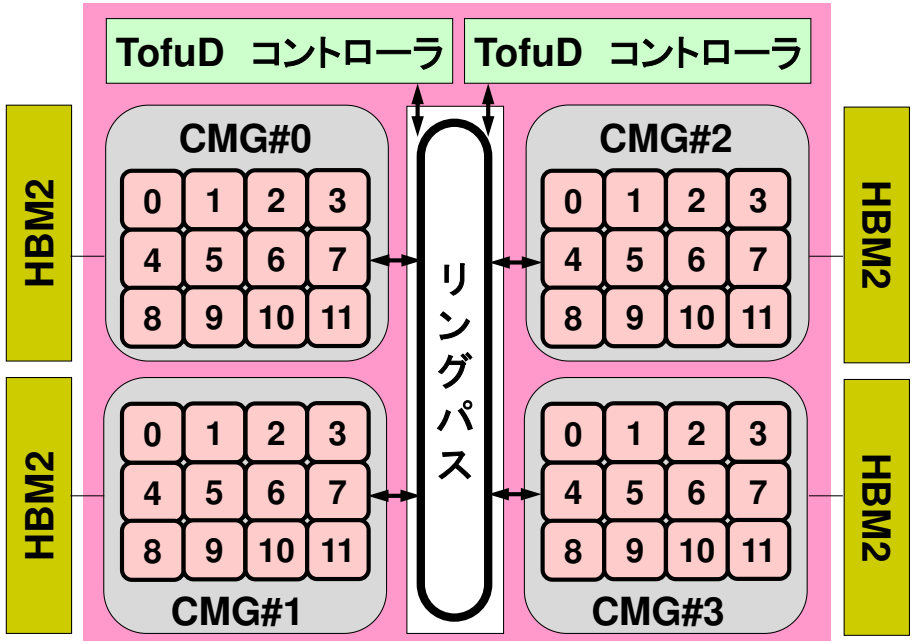
ジョブスクリプト

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

```
#!/bin/sh
#PJM -N "hello"                    Job Name
#PJM -L rscgrp=tutorial-o          Name of "Resource Group"
#PJM -L node=1                     Node#
#PJM -mpi proc=4                   Total MPI Process#
#PJM -L elapse=00:15:00            Computation Time
#PJM -g gt00                       Group Name (Wallet)
#PJM -j
#PJM -e err                        Standard Error
#PJM -o hello.lst                  Standard Output

module load fj                     必須
module load fjmpi

mpiexec ./a.out
```

プロセス数

```
#PJM -L node=1;#PJM --mpi proc= 1    1-node, 1-proc, 1-proc/n
#PJM -L node=1;#PJM --mpi proc= 4    1-node, 4-proc, 4-proc/n
#PJM -L node=1;#PJM --mpi proc=12    1-node,12-proc,12-proc/n
#PJM -L node=1;#PJM --mpi proc=24    1-node,24-proc,24-proc/n
#PJM -L node=1;#PJM --mpi proc=48    1-node,48-proc,48-proc/n

#PJM -L node= 4;#PJM --mpi proc=192   4-node,192-proc,48-proc/n
#PJM -L node= 8;#PJM --mpi proc=384   8-node,384-proc,48-proc/n
#PJM -L node=12;#PJM --mpi proc=576  12-node,576-proc,48-proc/n
```



ジョブ投入

```
>$ cd /work/gt00/t00XXX/pFEM/mpi/S1
>$ module load fj
>$ pjsub hello.sh

>$ cat hello.lst

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

利用可能なResource Group（Queue）

- 以下の2種類のResource Groupを利用可能
- 最大12ノードを使える
  - **lecture-o**
    - 12ノード（576コア），15分，アカウント有効期間中利用可能
    - 全教育ユーザーで共有
  - **tutorial-o**
    - 12ノード（576コア），15分，講義・演習実施時間帯
    - **Lecture-o**よりは多くのジョブを投入可能（混み具合による）

様々なコマンド

• ジョブ実行	`pjsub SCRIPT NAME`
• ジョブ実行状況	`pjstat`
• ジョブ停止	`pjdel JOB ID`
• ジョブキューの状況	`pjstat --rsc`
• ジョブキューの状況(詳細)	`pjstat --rsc -x`
• 実行ジョブ情報	`pjstat -a`
• ジョブ実行履歴	`pjstat -H`
• ジョブ実行制限	`pjstat --limit`

```
[t00470@wisteria01 run]$ pjsub f2_48.sh
[INFO] PJM 0000 pjsub Job 15713 submitted.

[t00470@wisteria01 run]$ pjsub f3_48.sh
[INFO] PJM 0000 pjsub Job 15714 submitted.

[t00470@wisteria01 run]$ pjstat
Wisteria/BDEC-01 scheduled stop time: 2021/05/28(Fri) 09:00:00 (Remain:  4days 1:25:56)

JOB_ID       JOB_NAME   STATUS  PROJECT    RSCGROUP          START_DATE        ELAPSE       TOKEN            NODE  GPU
15713        f2_48      RUNNING gt00       lecture-o         05/24 07:34:03    00:00:02         -               1    -
15714        f3_48      QUEUED  gt00       lecture-o         --/-- --:--:--    00:00:00         -               1    -

[t00470@wisteria01 run]$ pjstat
Wisteria/BDEC-01 scheduled stop time: 2021/05/28(Fri) 09:00:00 (Remain:  4days 1:25:56)

JOB_ID       JOB_NAME   STATUS  PROJECT    RSCGROUP          START_DATE        ELAPSE       TOKEN            NODE  GPU
15713        f2_48      RUNNING gt00       lecture-o         05/24 07:34:03    00:00:02         -               1    -
15714        f3_48      RUNNING gt00       lecture-o         (05/24 07:34)     00:00:00         -               1    -

[t00XYZ@wisteria01 ~]$ pjdel 15714
[INFO] PJM 0100 pjdel Accepted Job 15714

[t00XYZ@wisteria01 ~]$ pjstat
Wisteria/BDEC-01 scheduled stop time: 2021/05/28(Fri) 09:00:00 (Remain:  4days 1:25:56)

JOB_ID       JOB_NAME   STATUS  PROJECT    RSCGROUP          START_DATE        ELAPSE       TOKEN            NODE  GPU
15713        f2_48      RUNNING gt00       lecture-o         05/24 07:34:03    00:00:02         -               1    -

[t00XYZ@wisteria01 ~]$ pjstat
Wisteria/BDEC-01 scheduled stop time: 2021/05/28(Fri) 09:00:00 (Remain:  4days 1:21:45)

No unfinished job found.
```

```
[t00XYZ@wisteria01 ~]$ pjstat --rsc
SYSTEM: Odyssey
RSCGRP                  STATUS                  NODE
lecture-o               [ENABLE,START]            96
tutorial-o              [DISABLE,STOP]         2x12x16

SYSTEM: Aquarius
RSCGRP                  STATUS                  NODE  GPU
lecture-a               [ENABLE,START]             7   56
tutorial-a              [DISABLE,STOP]             2   16

[t00XYZ@wisteria01 ~]$ pjstat --rsc -x
SYSTEM: Odyssey
RSCGRP                  STATUS          MIN_NODE MAX_NODE MAX_ELAPSE REMAIN_ELAPSE MEM(GiB) PROJECT
lecture-o               [ENABLE,START]         1       12   00:15:00      00:15:00       28 gt00
tutorial-o              [DISABLE,STOP]         1       12   00:15:00      --:--:--       28 gt00

SYSTEM: Aquarius
RSCGRP                  STATUS          MIN_NODE MAX_NODE AVAIL_GPU MAX_ELAPSE REMAIN_ELAPSE MEM(GiB) PROJECT
lecture-a               [ENABLE,START]         1        1     1,2,4   00:15:00      00:15:00      448 gt00
tutorial-a              [DISABLE,STOP]         1        1     1,2,4   00:15:00      --:--:--      448 gt00

[t00XYZ@wisteria01 ~]$ pjstat --limit
SYSTEM: Odyssey
PROJECT        ACCEPT        RUN  BULK_ACCEPT    BULK_RUN          NODE
gt00          0/  128     0/   16       0/    8     0/   16       0/ 2304

SYSTEM: Aquarius
PROJECT        ACCEPT        RUN  BULK_ACCEPT    BULK_RUN           GPU
gt00          0/    4     0/    2       0/    0     0/    0       0/   64
```

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？

```
#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();
}
```

```
#!/bin/sh
#PJM -N "hello"                 Job Name
#PJM -L rscgrp=tutorial-o       Name of "Resource Group"
#PJM -L node=1                  Node#
#PJM -mpi proc=4                Total MPI Process#
#PJM -L elapse=00:15:00         Computation Time
#PJM -g gt00                    Group Name (Wallet)
#PJM -j
#PJM -e err                     Standard Error
#PJM -o hello.lst               Standard Output

module load fj                  必須
module load fjmpi

mpiexec ./a.out
```

- **mpiexec.hydra** により4つのプロセスが立ち上がる（今の場合は”node=1: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（myid）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。

C

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）
- 全実行文の前に置くことを勧める

- **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();
}
```

C

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）。
- 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・・・

- **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();
}
```

C

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** MPI_Comm 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();
}
```

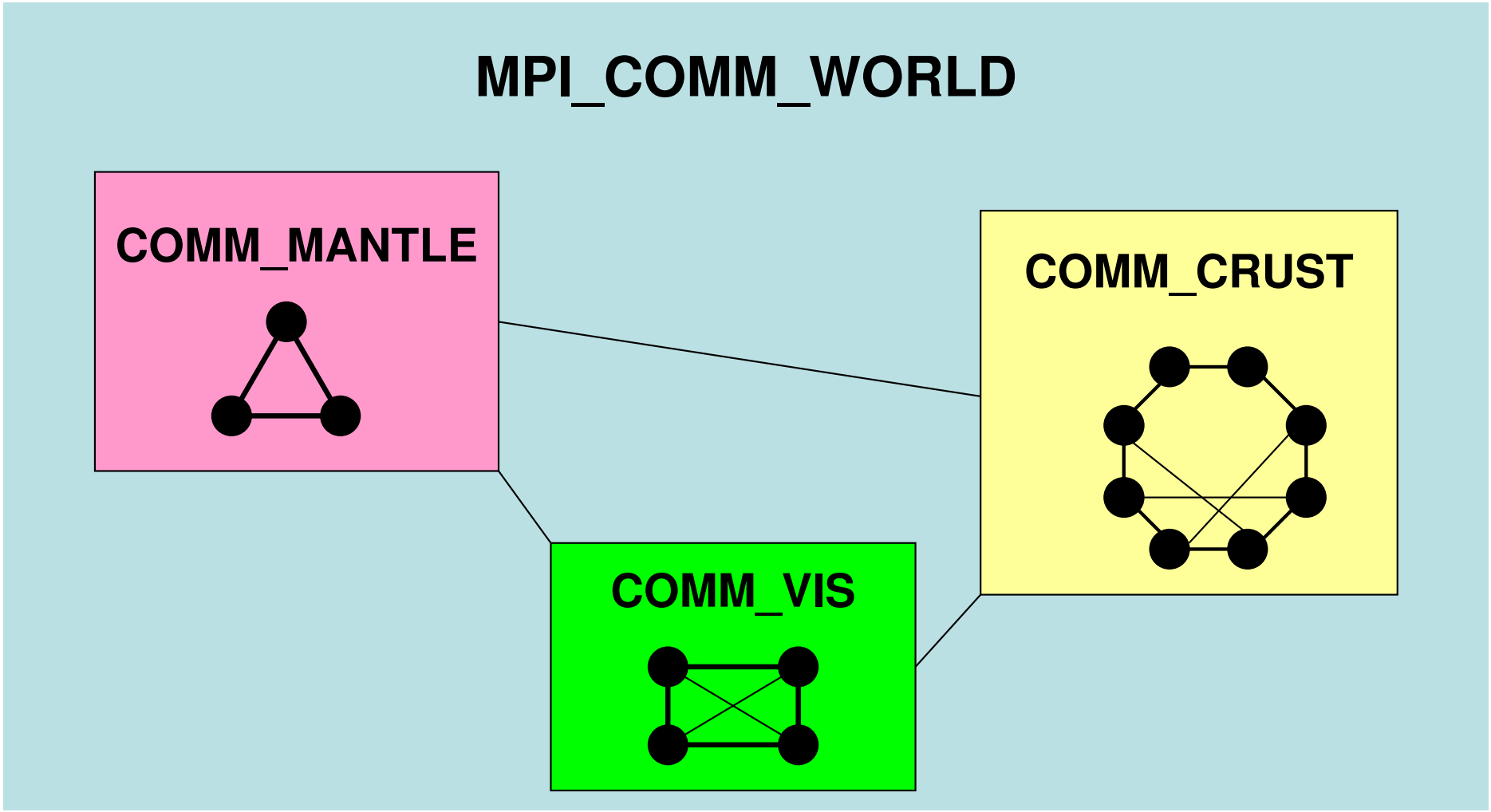
コミュニケータとは？

```
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。

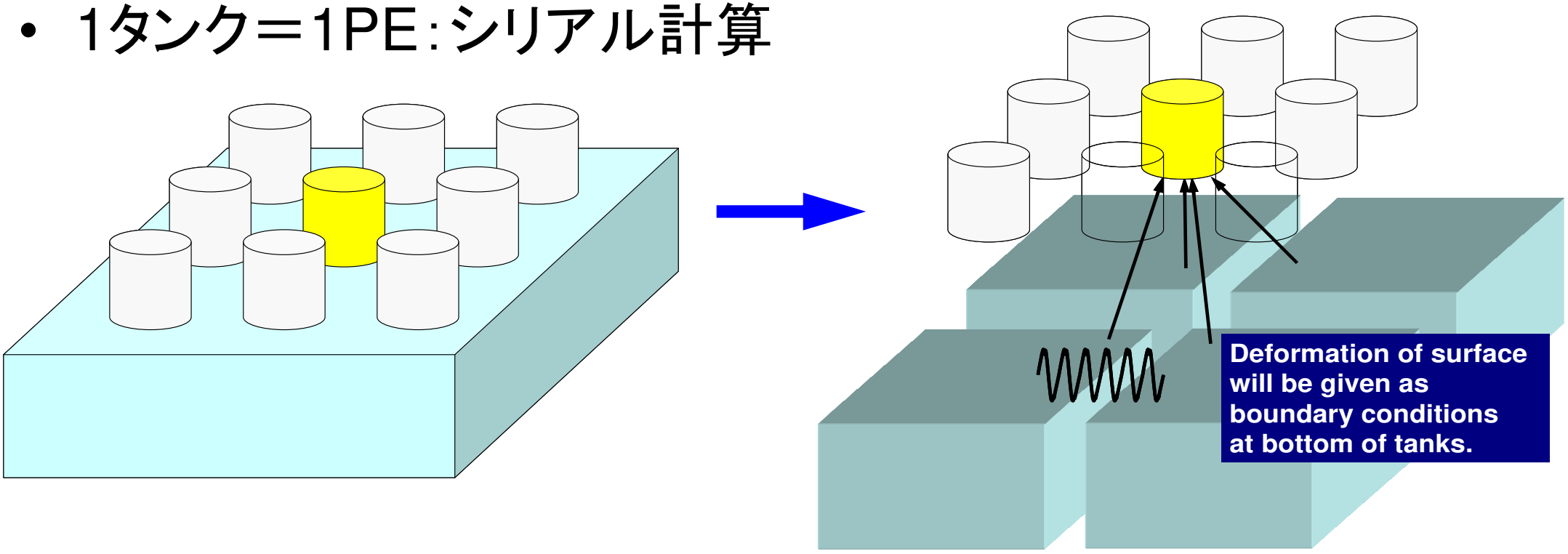
コミュニケータの概念

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



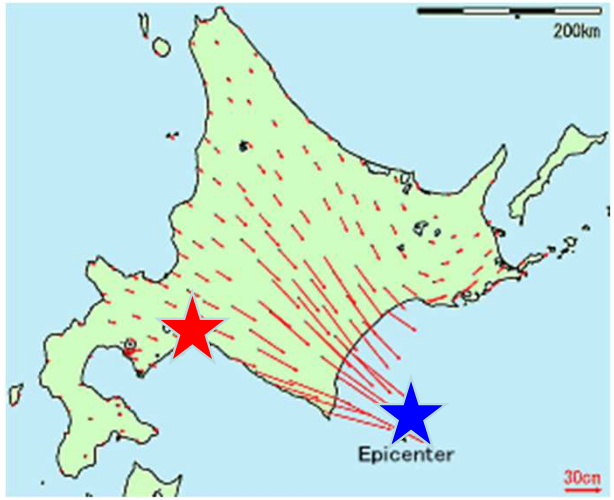
対象とするアプリケーション

- 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



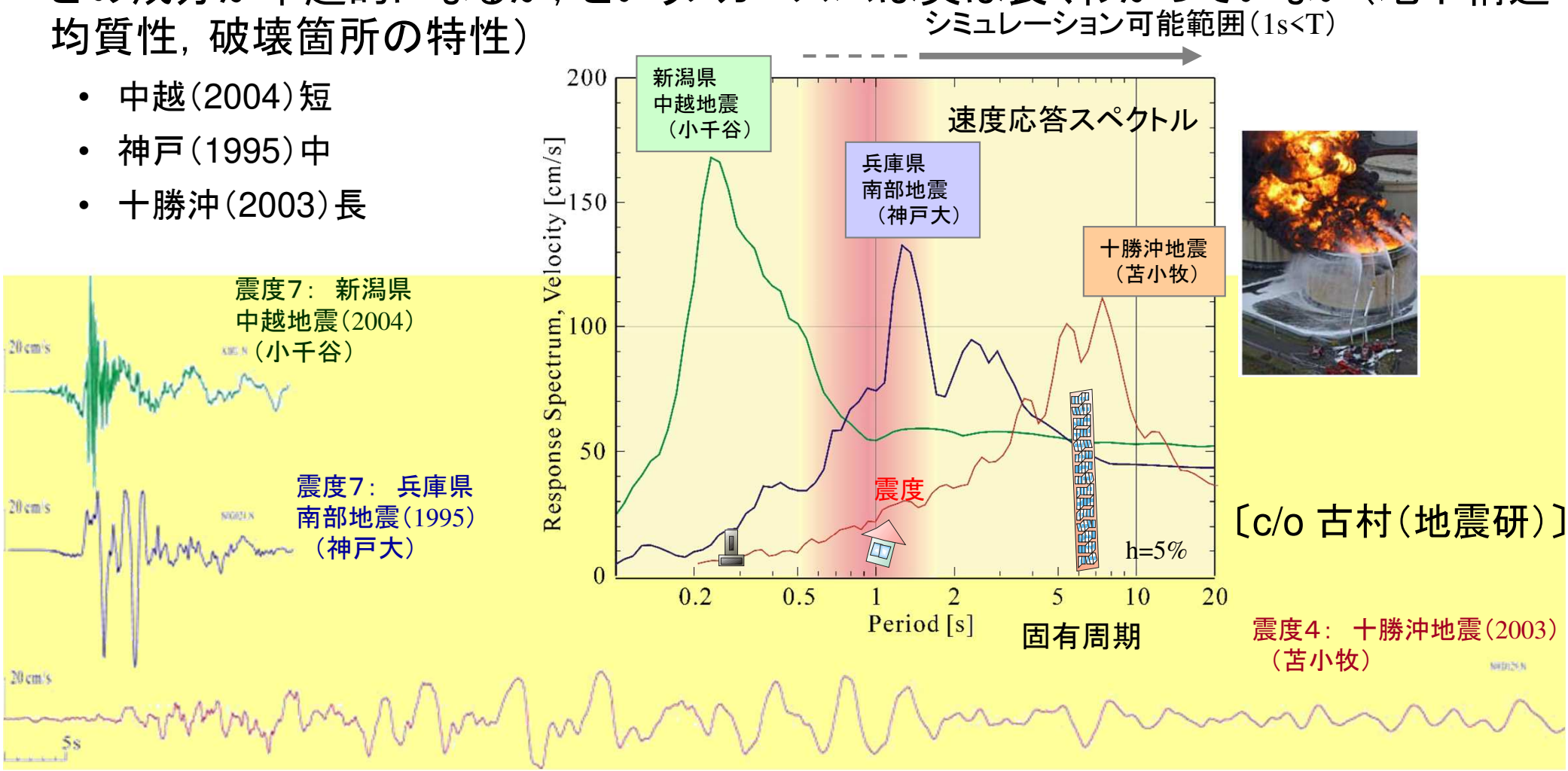
2003年　十勝沖地震

長周期地震波動（表面波）：苫小牧の石油タンクが激しく揺れ，金具がこすれた火花が，液面揺動（スロッシング）する石油に引火して大火災に



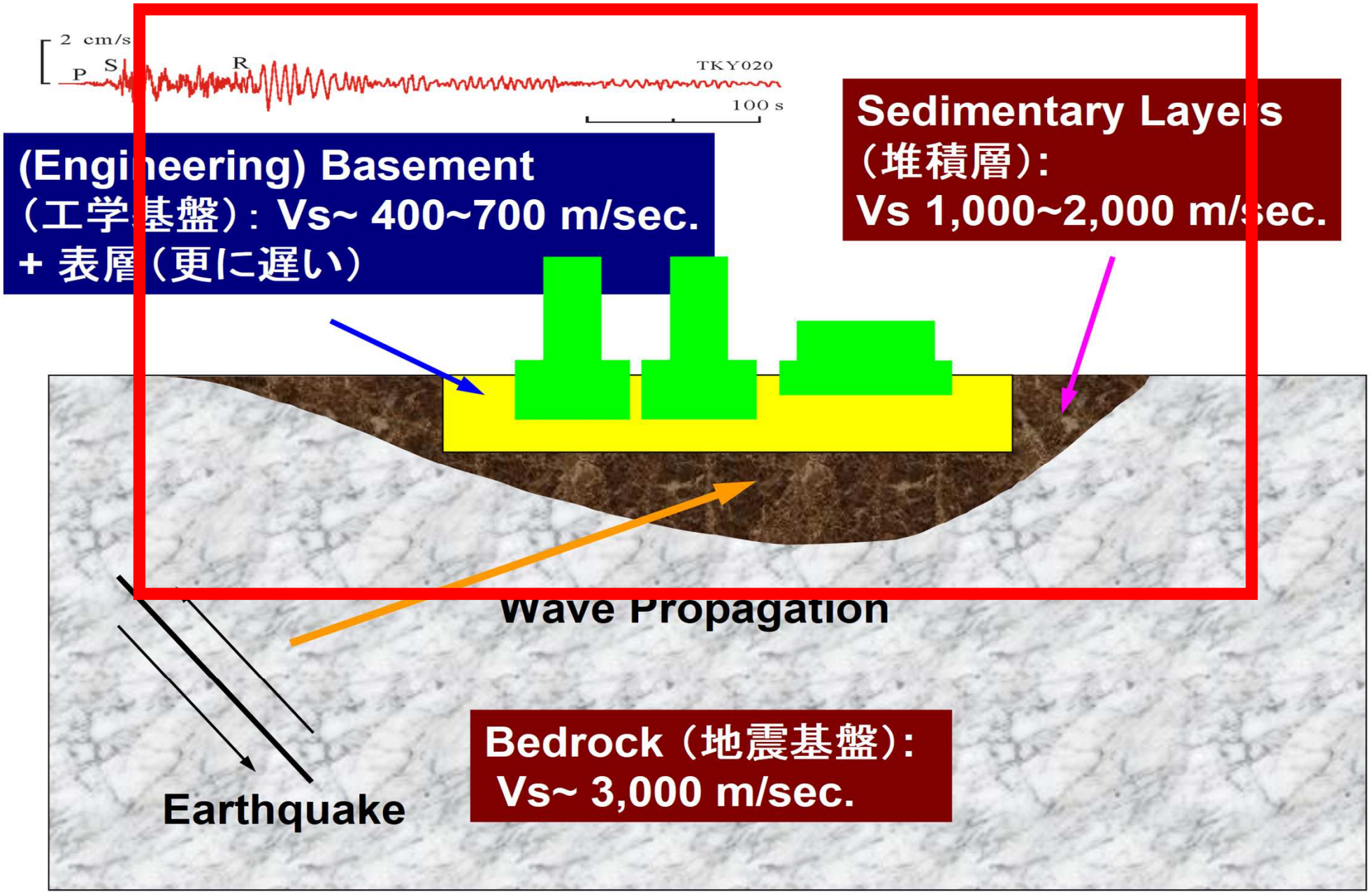
地震波：様々な波長の成分の合成

- 卓越成分と同じ固有周期の建物がもっとも激しく揺れる：一種の「共鳴」
  - 人工構造物の固有周期（振動周期）は0.1~10 sec　大きな建物ほど大きい
    - 長周期の波は長く続き，遠くまで届く：測定場所によってもスペクトル分布は異なる
  - どの成分が卓越的になるか，というメカニズムは実は良くわかっていない（地下構造不均質性，破壊箇所の特性）
    - 中越（2004）短
    - 神戸（1995）中
    - 十勝沖（2003）長



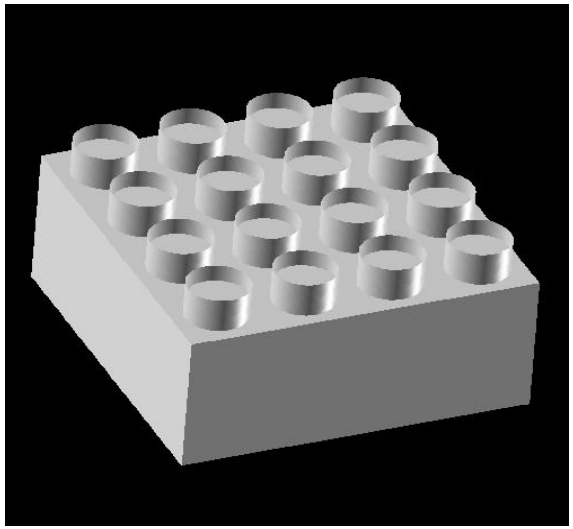
〔c/o 古村（地震研）〕

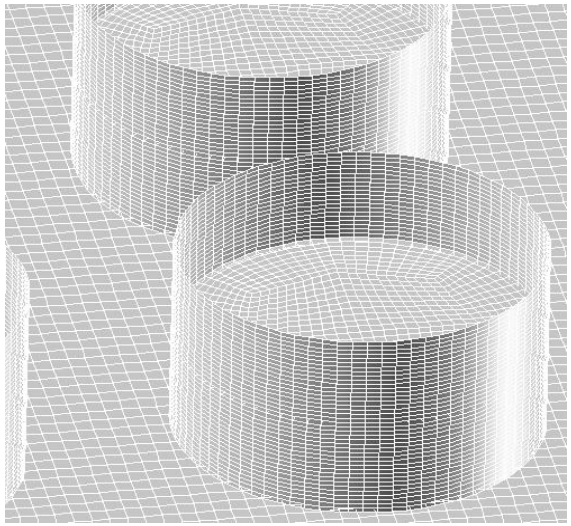
地盤・石油タンク振動連成シミュレーション



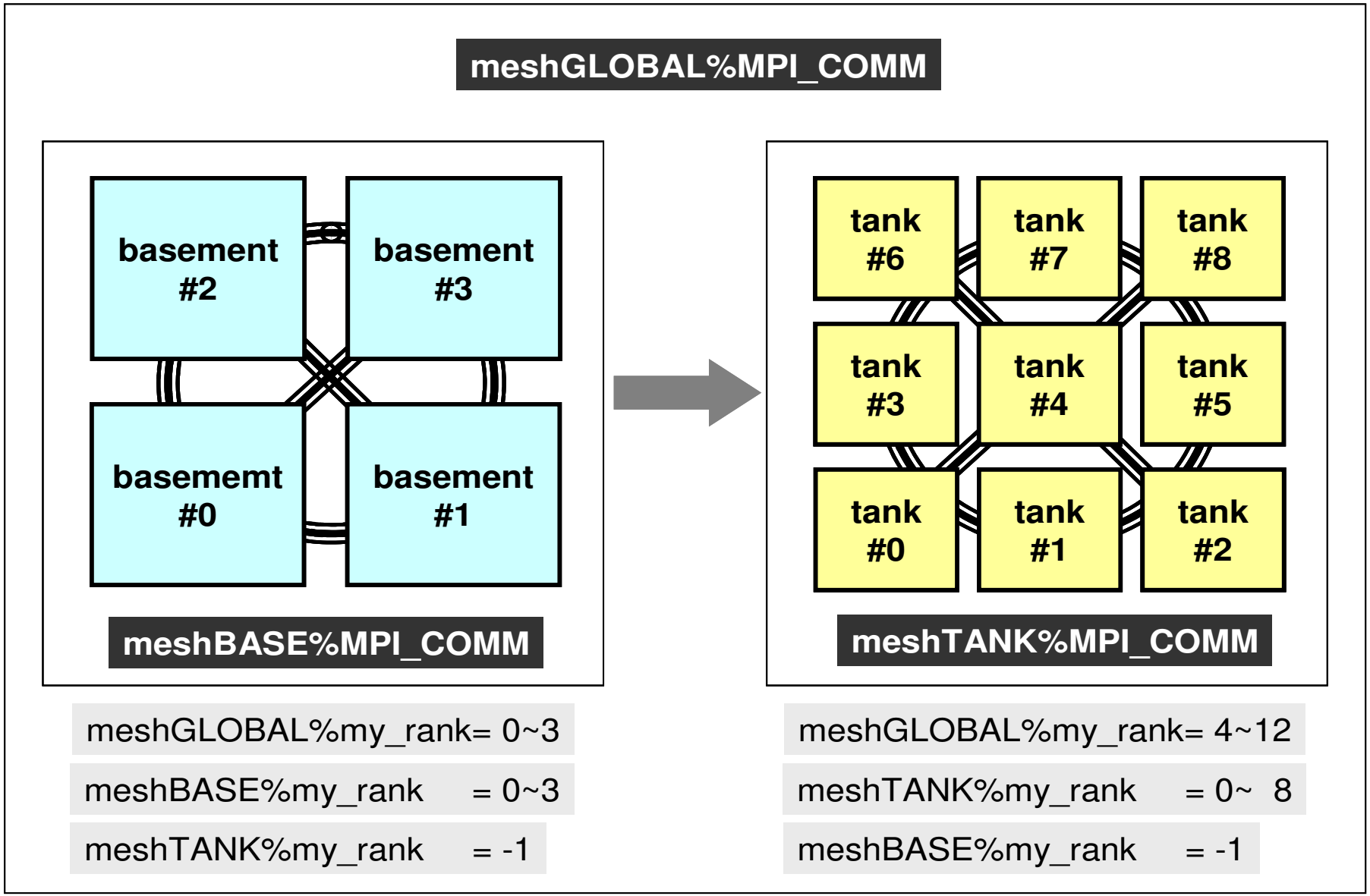
地盤，タンクモデル

- 地盤モデル（市村）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（長嶋）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

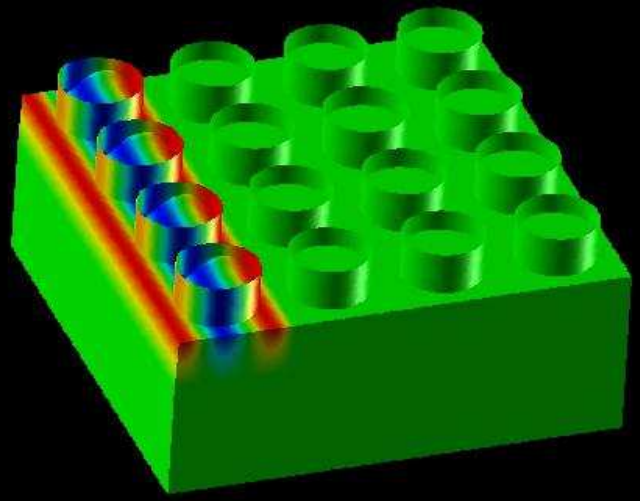
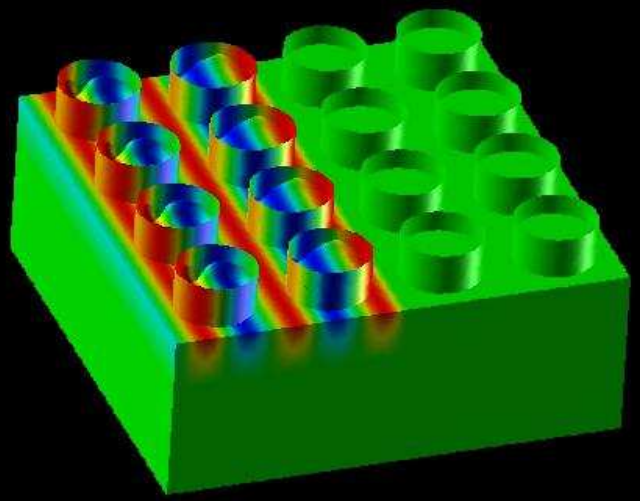
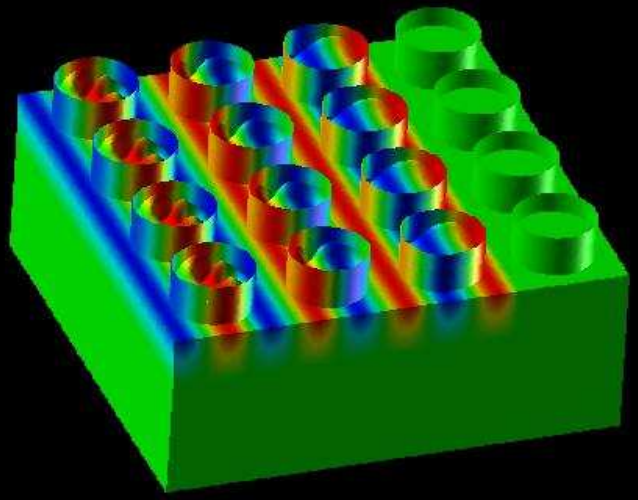
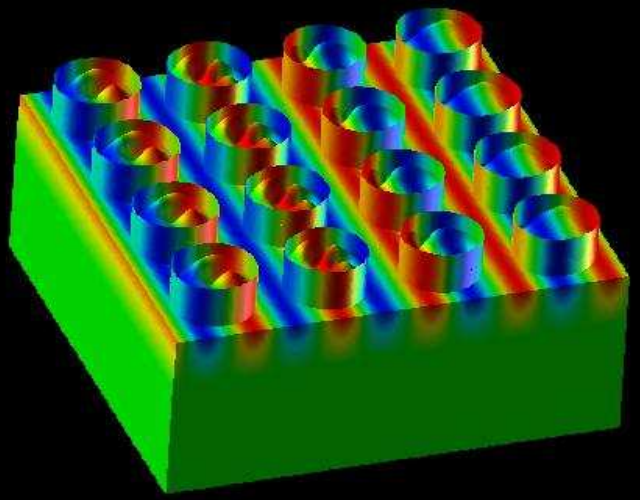


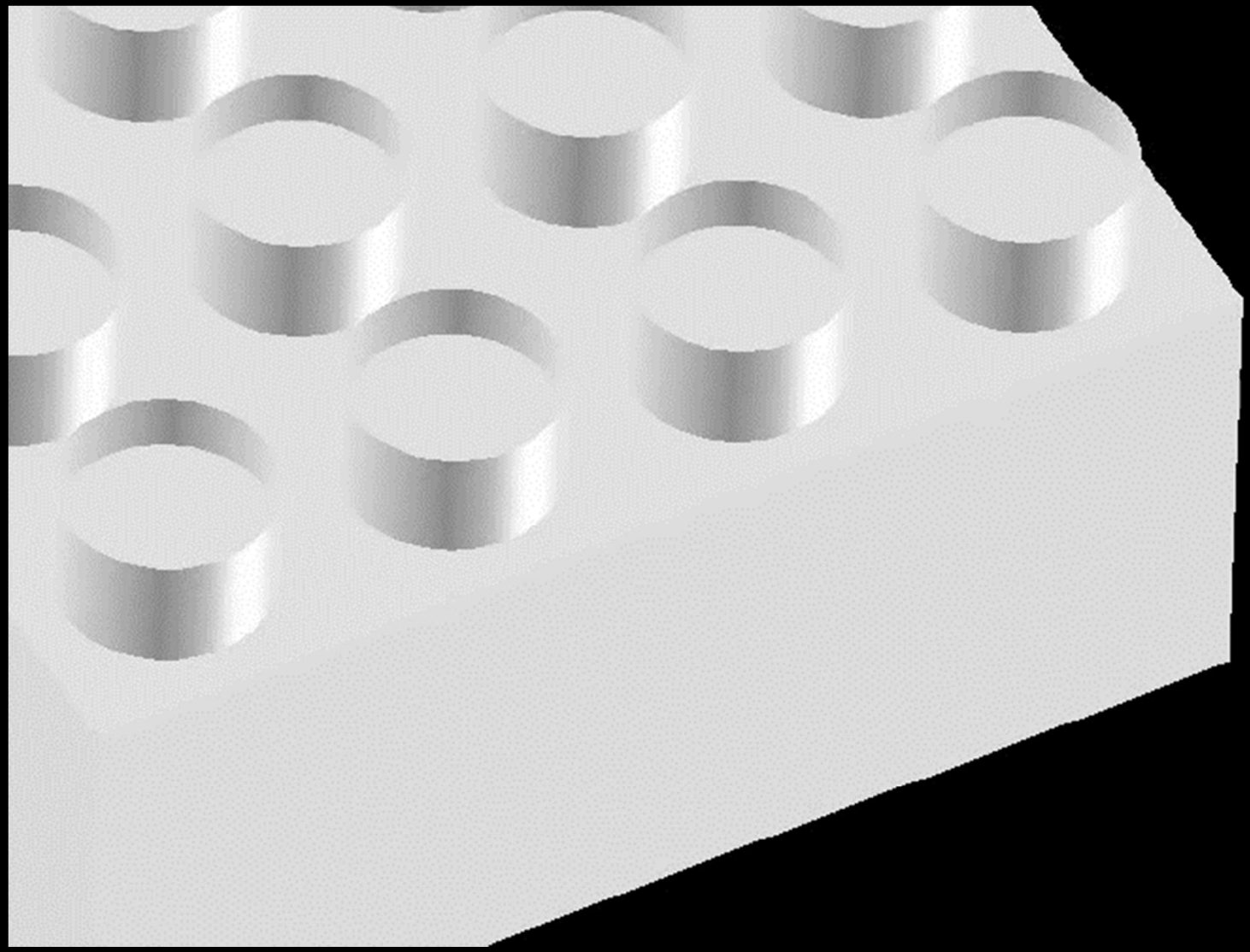


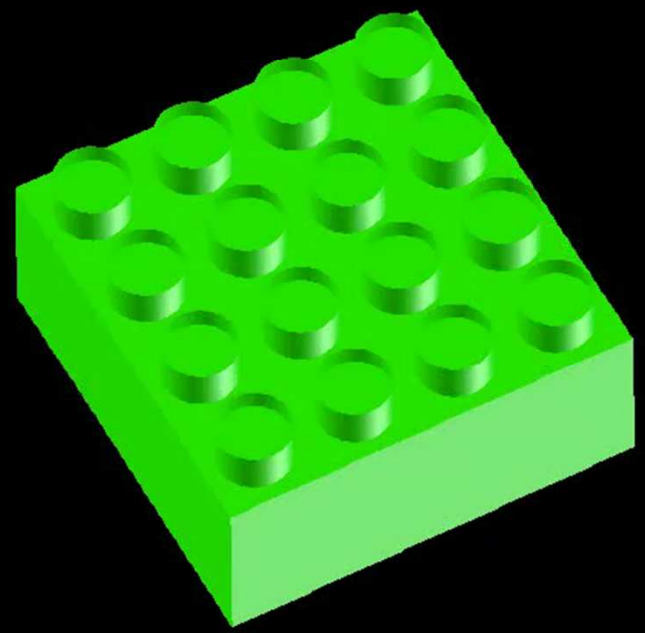
3種類のコミュニケータの生成



地盤・石油タンク連成シミュレーション





C

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** MPI_Comm 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();
}
```

C

MPI_Abort

- MPIプロセスを異常終了する。

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

C

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 /work/gt00/t00XXX/pFEM/mpi/S1
$> module load fj

$> mpifccpx -Nclang -O1 time.c
$> mpifrtpx -O1 time.f

$> 実行(4プロセス) pjsub go4.sh

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

   プロセス       計算時間
    番号
```

go4.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial-o
#PJM -L node=1
#PJM --mpi proc=4
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

module load fj
module load fjmpi

mpiexec ./a.out
```

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 /work/gt00/t00XXX/pFEM/mpi/S1
$> module load fj

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

$>（実行:1プロセス）pjsub go1.sh
```

go1.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial-o
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

module load fj
module load fjmpi

mpiexec ./a.out
```

C

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`**　　MPI_Comm　I　　コミュニケータを指定する

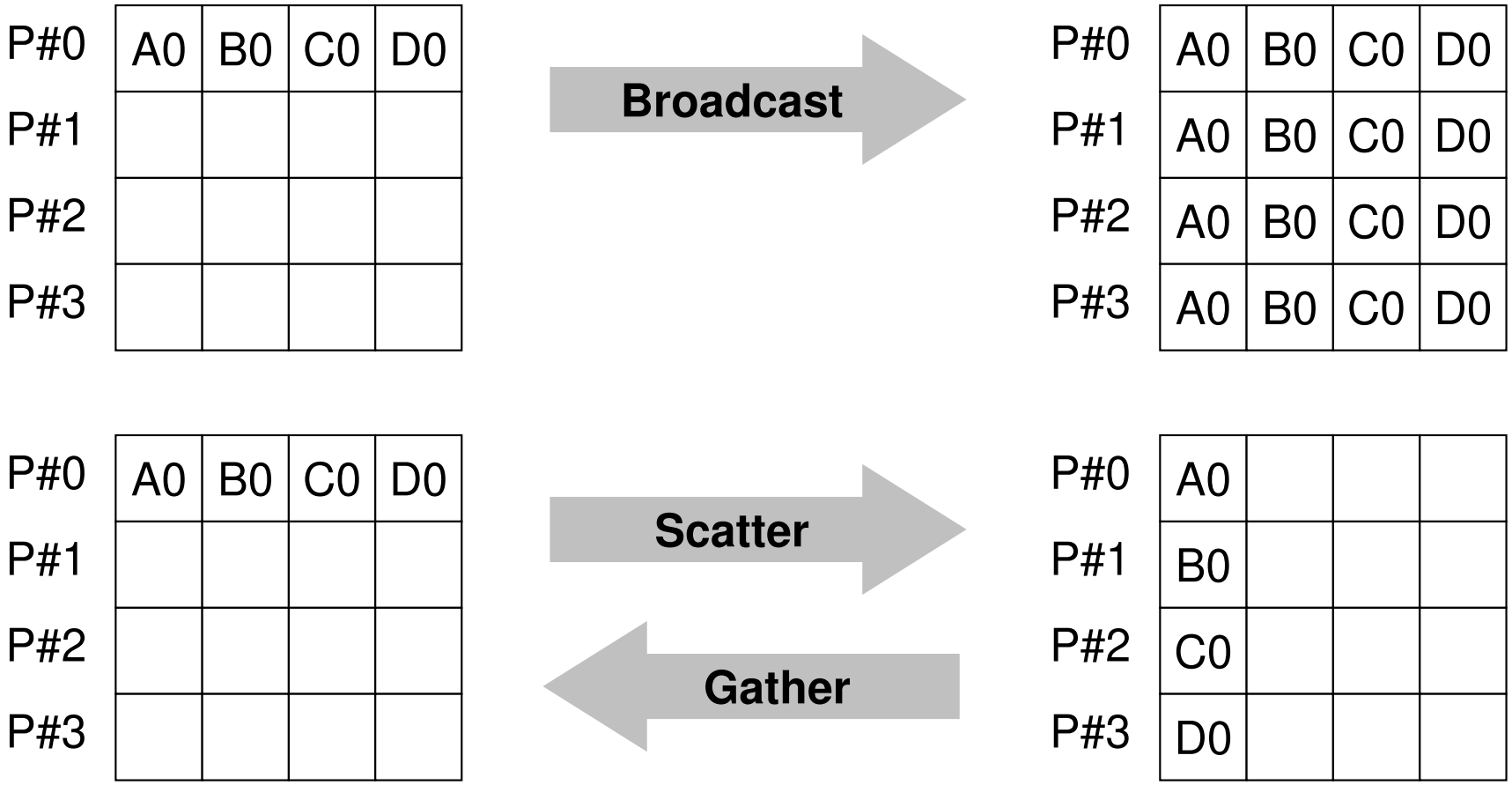
- MPIとは
- MPIの基礎：Hello World
- 集団通信（Collective Communication）
- 1対1通信（Point-to-Point Communication）

集団通信とは

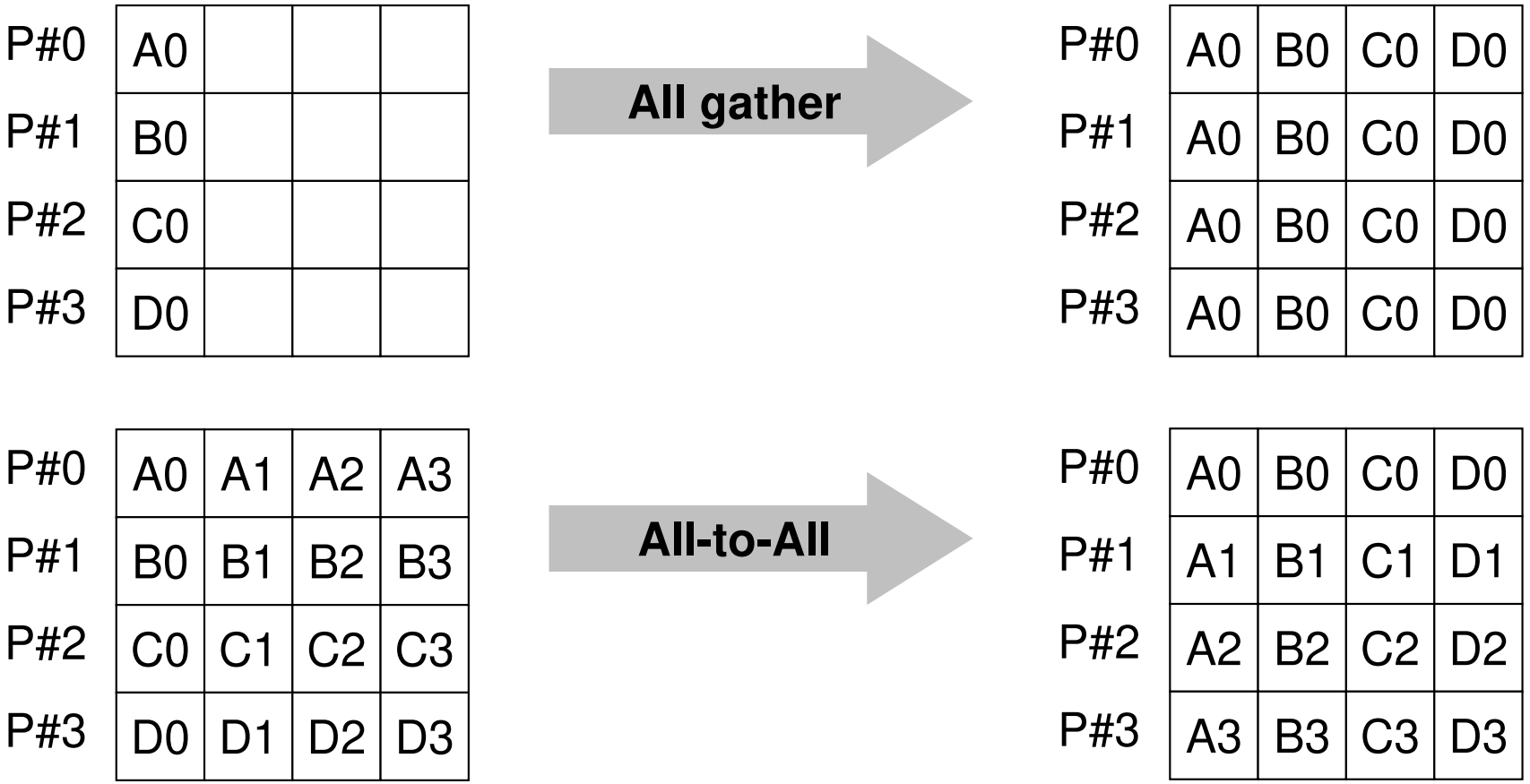
- コミュニケータで指定されるグループ全体に関わる通信。

- 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

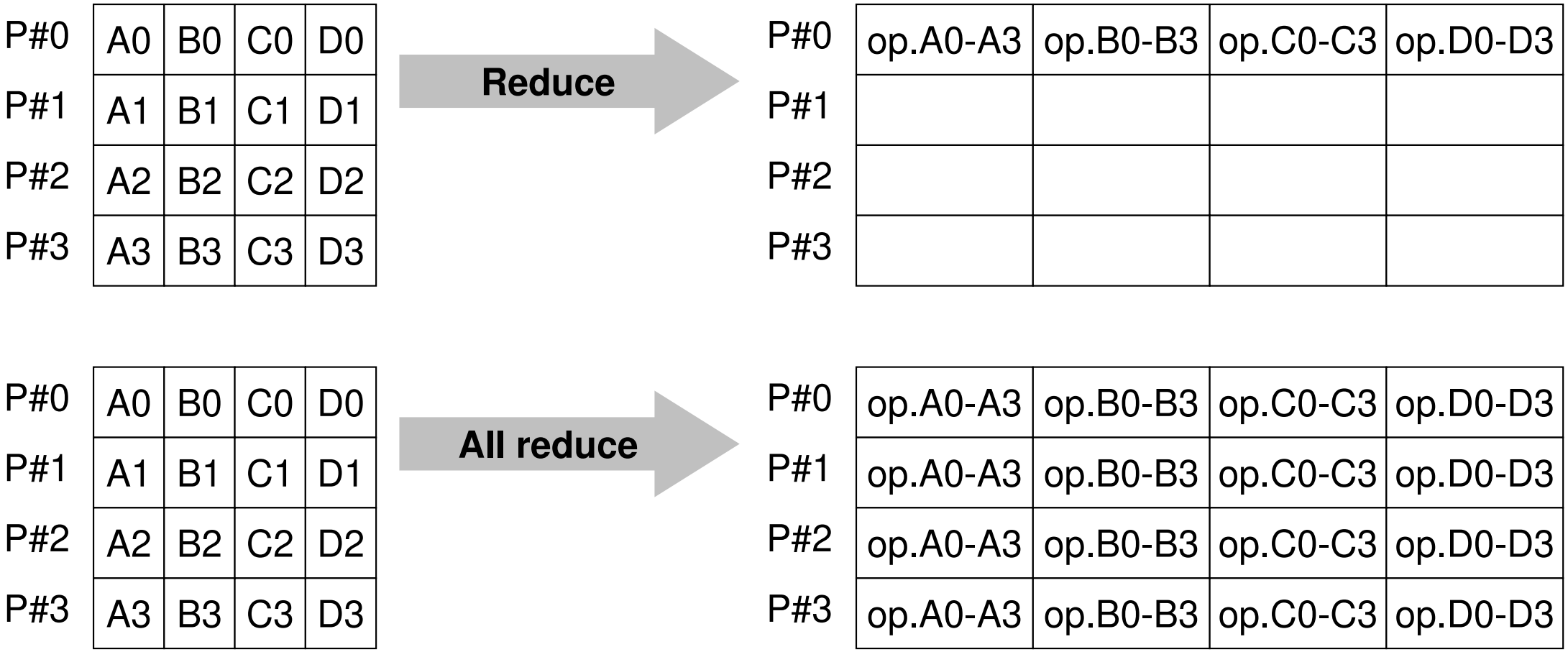
集団通信の例(1/4)



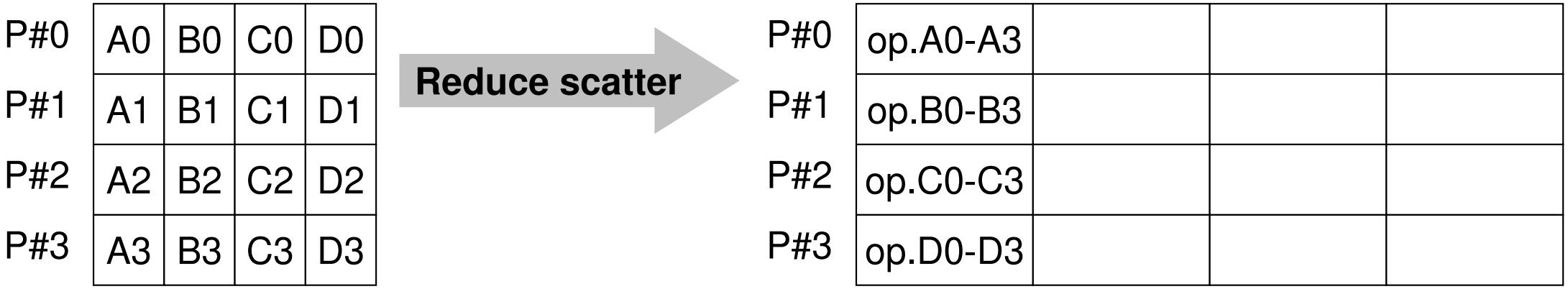
集団通信の例(2/4)


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

集団通信の例(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

集団通信の例(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

集団通信による計算例

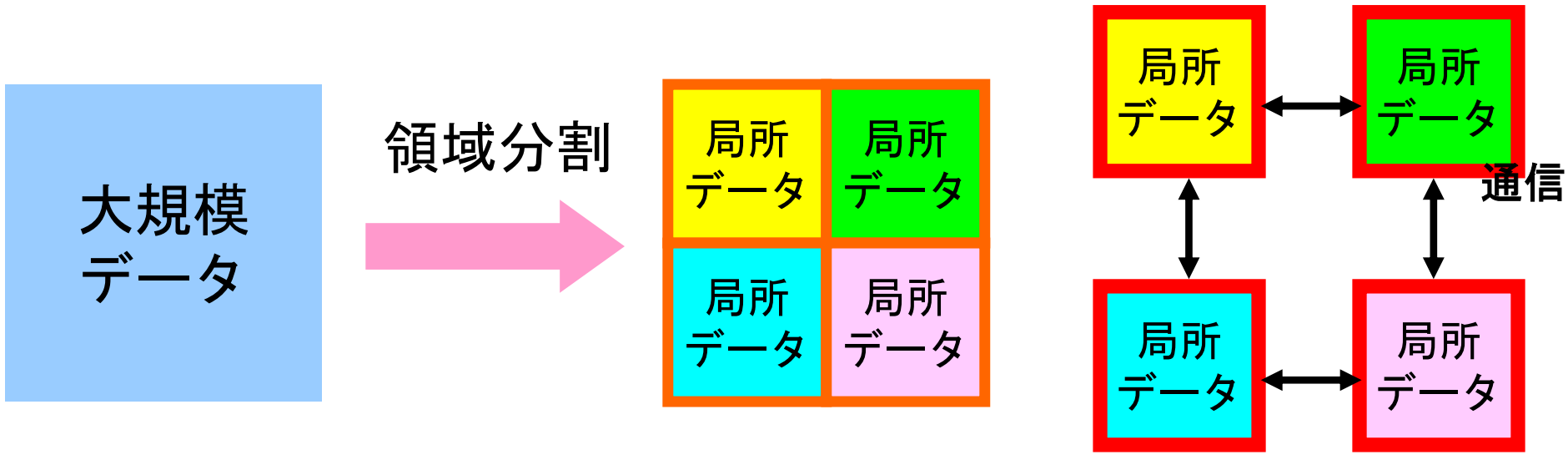
- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み
- MPI_Allgatherv

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。

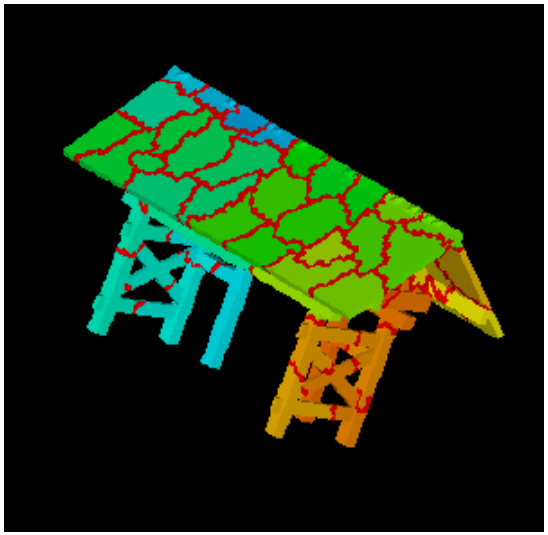
領域分割

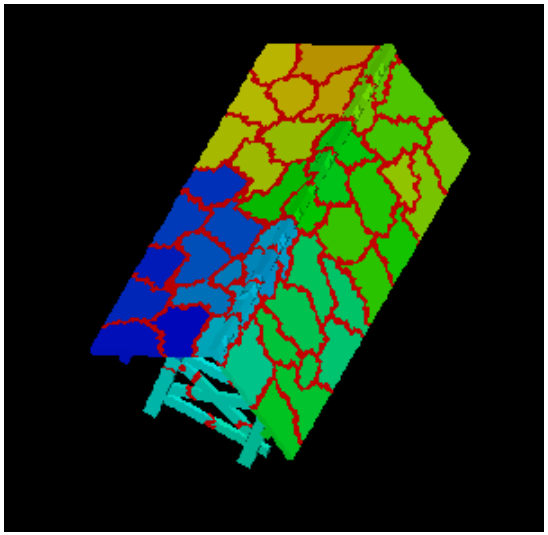
- 1GB程度のPC → 10^6メッシュが限界：FEM
  - 1000km×1000km×100kmの領域（西南日本）を1kmメッシュで切ると10^8メッシュになる
- 大規模データ → 領域分割，局所データ並列処理
- 全体系計算 → 領域間の通信が必要



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
```

```
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
```

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

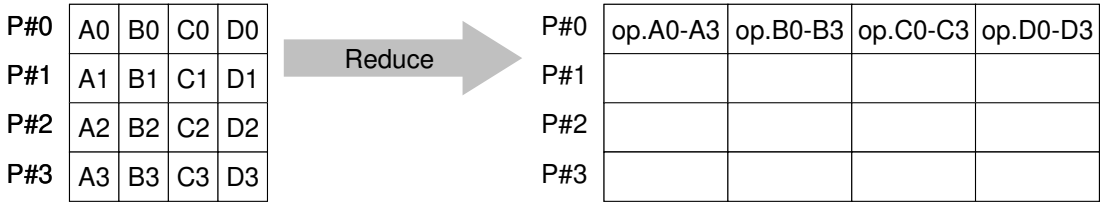
```
implicit REAL*8 (A-H,O-Z)
real(kind=8),dimension(20):: &
    VECp,  VECs

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;
}
```

<$O-S1>/dot.f, dot.cの実行（やらないでほしいが）

```
>$ cd /work/gt00/t00XXX/pFEM/mpi/S1

>$ gcc dot.c
>$ gfortran 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** MPI_Datatype 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** MPI_Op 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** MPI_Comm I コミュニケータを指定する

C

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。

Send/Receive Buffer (1/3)
A: Scalar

```
call MPI_REDUCE
(A,recvbuf, 1,datatype,op,root,comm,ierr)

MPI_Reduce
(A,recvbuf,1,datatype,op,root,comm)
```

Send/Receive Buffer (2/3)
A: Array

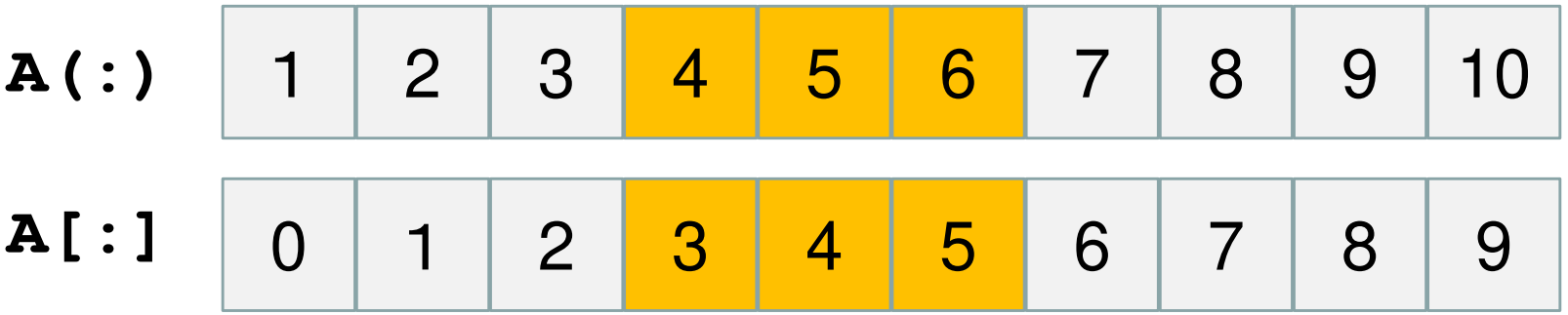
```
call MPI_REDUCE
(A,recvbuf, 3,datatype,op,root,comm,ierr)

MPI_Reduce
(A,recvbuf,3,datatype,op,root,comm)
```

- Starting Address of Send Buffer
  - `A(1)`: Fortran, `A[0]`: C
  - 3 (continuous) components of `A (A(1)-A(3), A[0]-A[2])` are sent

Send/Receive Buffer (3/3) A: Array

```
call MPI_REDUCE
(A(4),recvbuf, 3,datatype,op,root,comm,ierr)

MPI_Reduce
(A[3],recvbuf,3,datatype,op,root,comm)
```

- Starting Address of Send Buffer
  - `A(4)`: Fortran, `A[3]`: C
  - 3 (continuous) components of `A (A(4)-A(6), A[3]-A[5])` are sent



C

MPI_Reduceの例(1/2) C

```
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)

C

MPI_Reduceの例(2/2) C

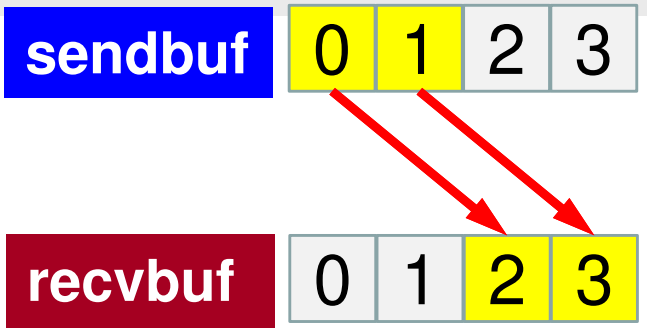
```
MPI_Reduce
(sendbuf,recvbuf,count,datatype,op,root,comm)
```

```
double X0, XSUM;

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

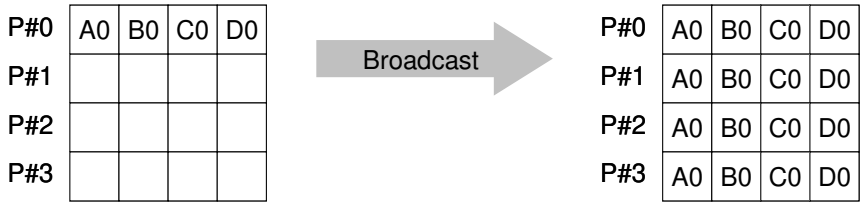
各プロセスにおける，X0の総和が0番PEのXSUMに入る。

```
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

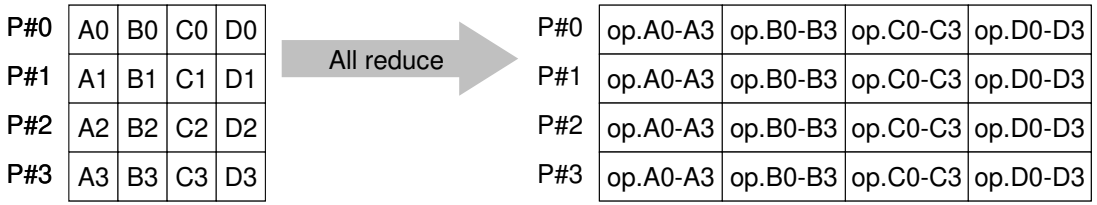


- コミュニケーター「comm」内の一つの送信元プロセス「root」のバッファ「buffer」から，その他全てのプロセスのバッファ「buffer」にメッセージを送信。

- **MPI_Bcast (buffer,count,datatype,root,comm)**
  - **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** MPI_Datatype 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** MPI_Comm I コミュニケータを指定する

C

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`** MPI_Datatype I メッセージのデータタイプ
  - **`op`** MPI_Op I 計算の種類
  - **`comm`** MPI_Comm I コミュニケータを指定する

C

MPI_Reduce/Allreduceの“op”

C

```
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

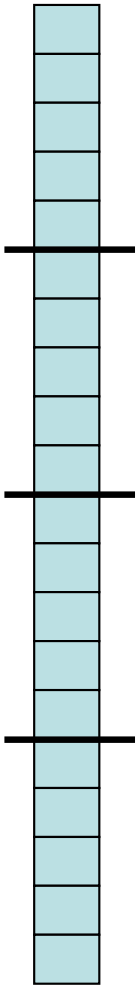
局所データの考え方(1/2)

C

- 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
```

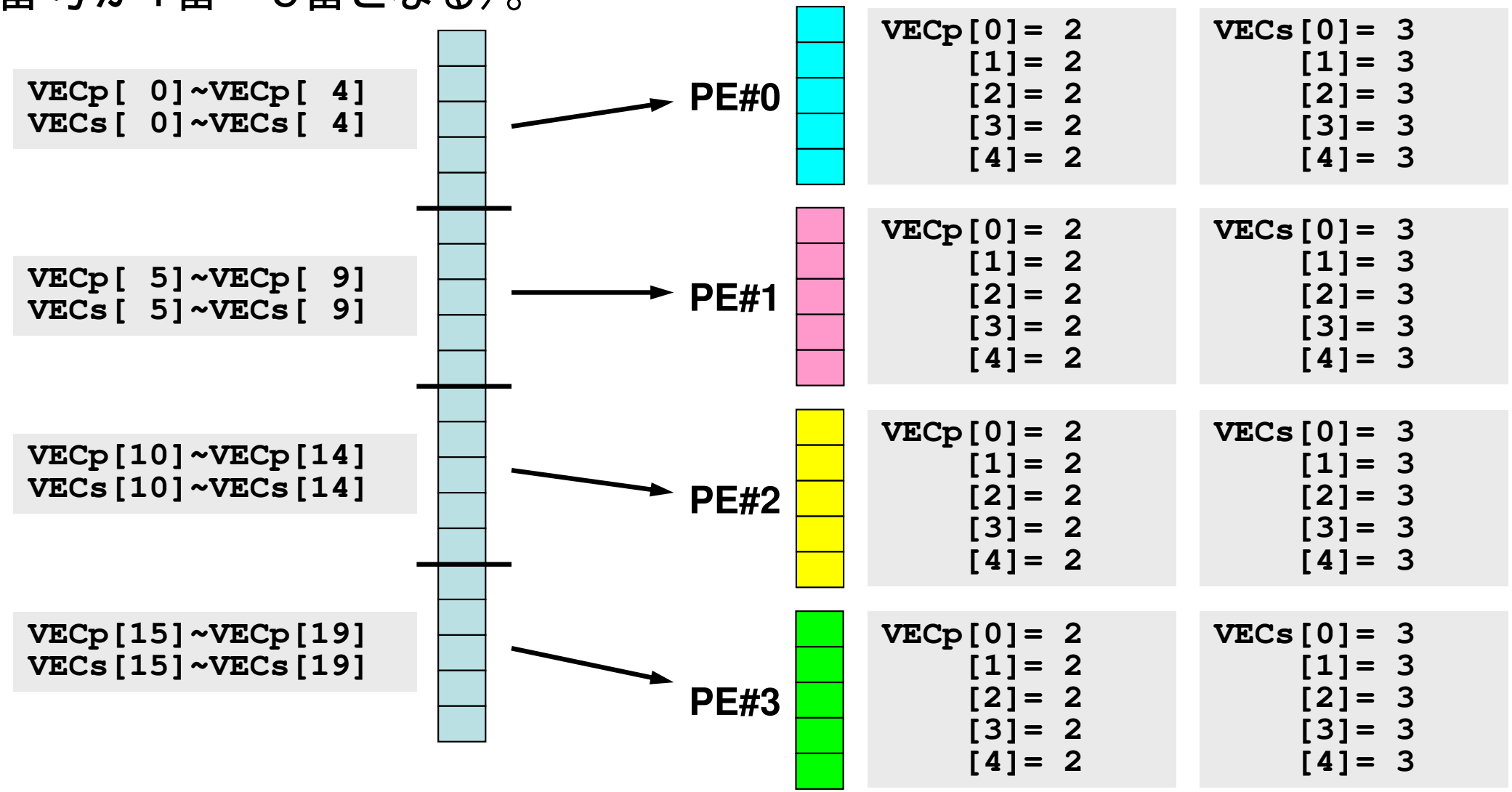
```
VECs[ 0]=  3
     [ 1]=  3
     [ 2]=  3
…
     [17]=  3
     [18]=  3
     [19]=  3
```



C

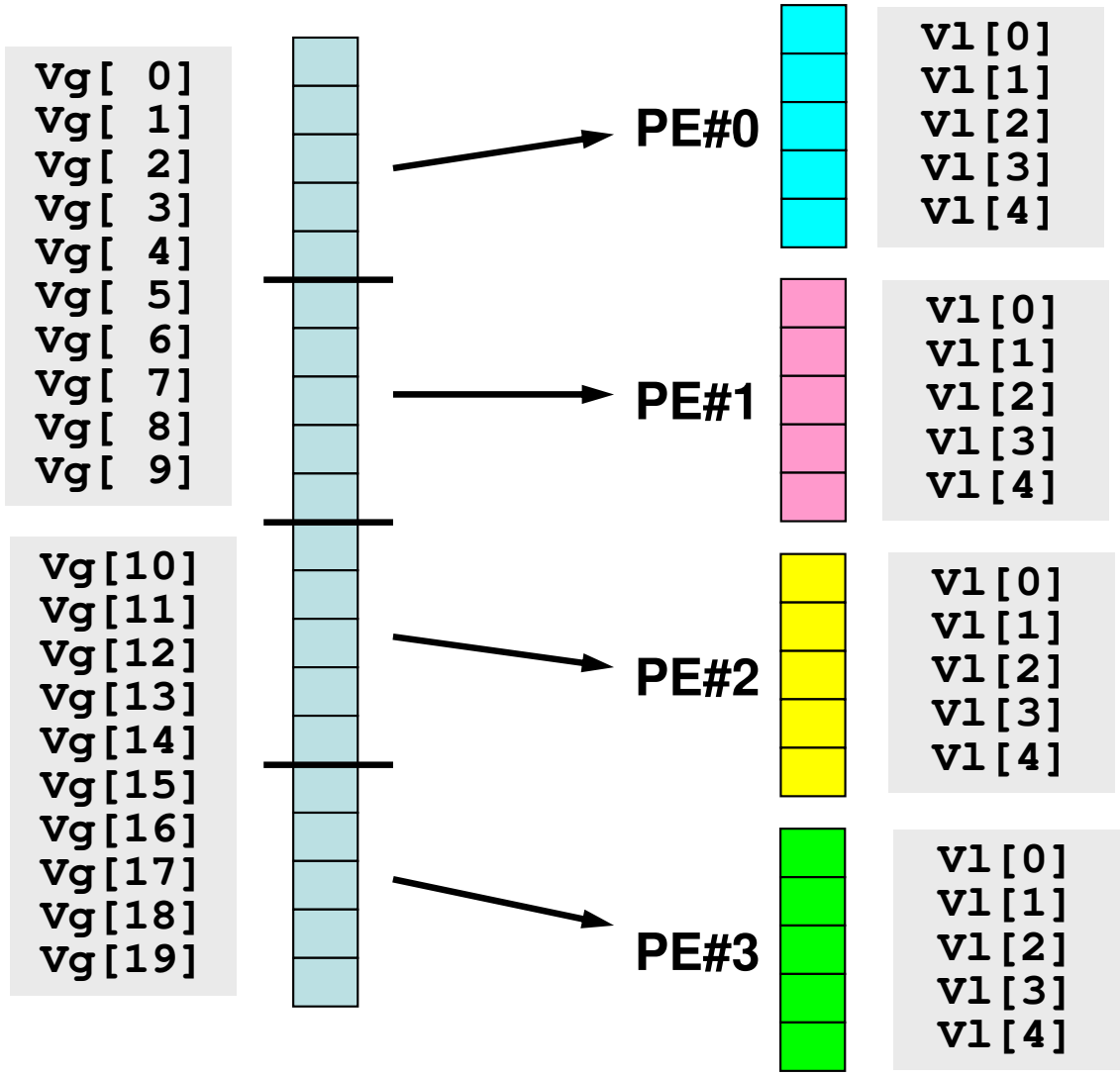
局所データの考え方(2/2)

- 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(0)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)

<$O-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(2/3)

<$O-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);
        printf("before BCAST %5d %15.0F %15.0F¥n", MyRank, sumA, sumR);

        MPI_Bcast(&sumR, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        printf("after  BCAST %5d %15.0F %15.0F¥n", MyRank, sumA, sumR);

        MPI_Finalize();

        return 0;
}
```

内積の並列計算例(3/3)

<$O-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);
```

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。

<$O-S1>/allreduce.f/c の実行例

```
$> cd /work/gt00/t00XXX/pFEM/mpi/S1
$> module load fj
$> mpifrtpx -Kfast allreduce.f
$> mpifccpx -Nclang -Kfast allreduce.c
$> (実行:4プロセス) pjsub 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
```

集団通信による計算例

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

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(<$O-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)

- 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
```

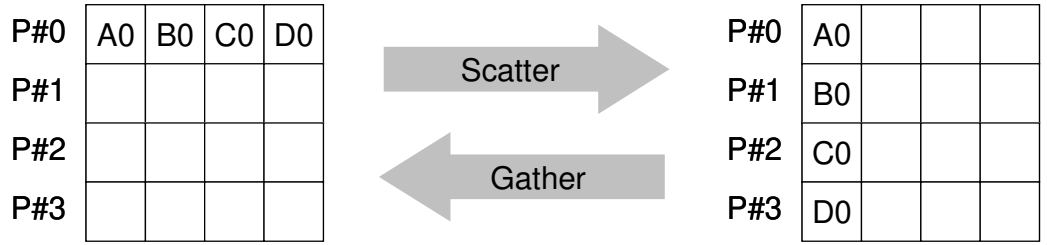
```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

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

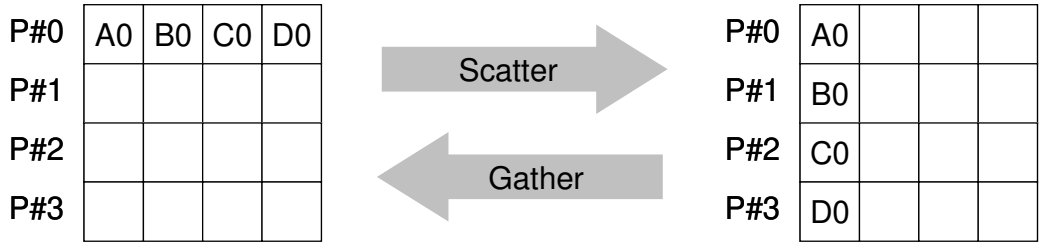


- コミュニケーター「comm」内の一つの送信元プロセス「root」の送信バッファ「sendbuf」から各プロセスに先頭から「scount」ずつのサイズのメッセージを送信し，その他全てのプロセスの受信バッファ「recvbuf」に，サイズ「rcount」のメッセージを格納。

- **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** MPI_Datatype 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** MPI_Datatype 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** MPI_comm I コミュニケータを指定する

C

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** MPI_Datatype 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** MPI_Datatype 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** MPI_comm 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。

C

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)
```

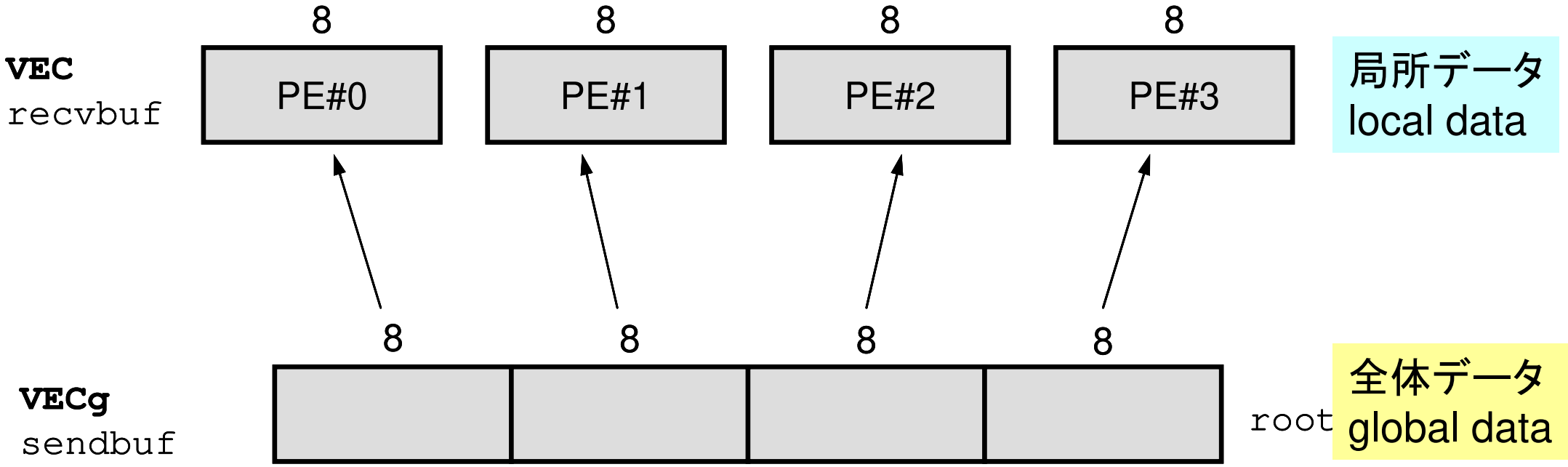
```
int  N=8;
double VEC [8];
...
MPI_Scatter (&VECg, N, MPI_DOUBLE, &VEC, N,
MPI_DOUBLE, 0, <comm>);
```

```
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.
```

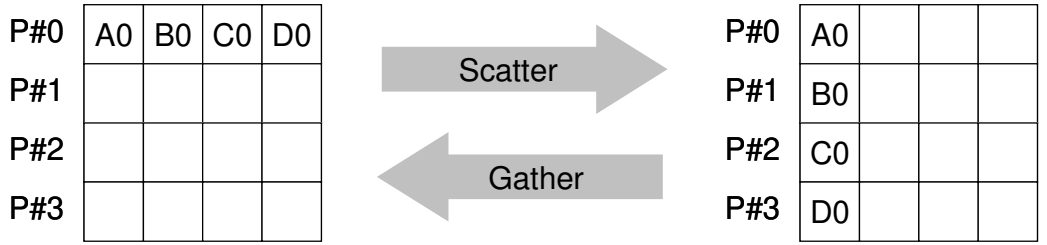
PE#3
```
after 3 1  1401.
after 3 2  1403.
after 3 3  1405.
after 3 4  1406.
after 3 5  1409.
after 3 6  1411.
after 3 7  1421.
after 3 8  1451.
```

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

- **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** MPI_Datatype 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** MPI_Datatype 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** MPI_comm 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。

C

Scatter/Gatherの計算（8/8）

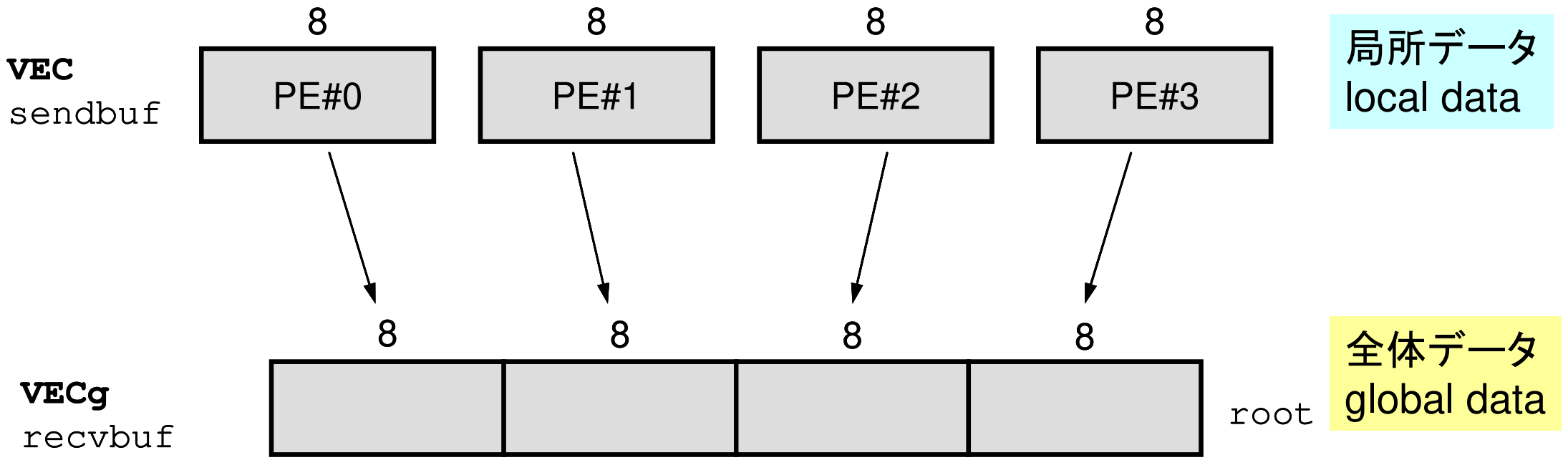
各プロセスの結果を再び長さ32のベクトルにまとめる

- 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



<$O-S1>/scatter-gather.f/c 実行例

```
$> cd /work/gt00/t00XXX/pFEM/mpi/S1
$> module load fj
$> mpifccpx -Nclang -Kfast scatter-gather.c
$> mpifrtpx -Kfast 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.
```

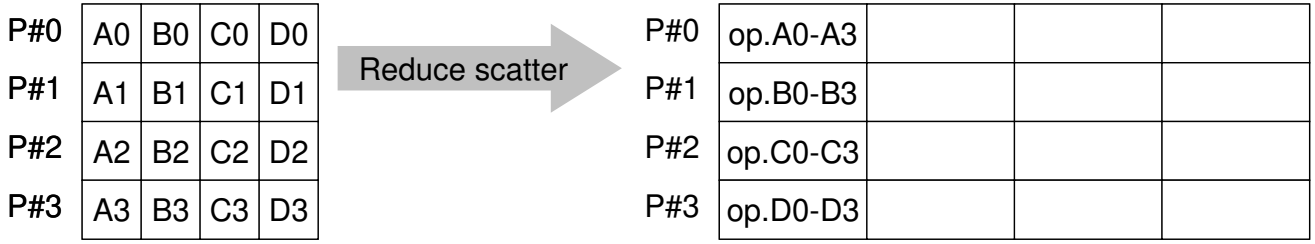
PE#0
```
after 0 1   1101.
after 0 2   1103.
after 0 3   1105.
after 0 4   1106.
after 0 5   1109.
after 0 6   1111.
after 0 7   1121.
after 0 8   1151.
```

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

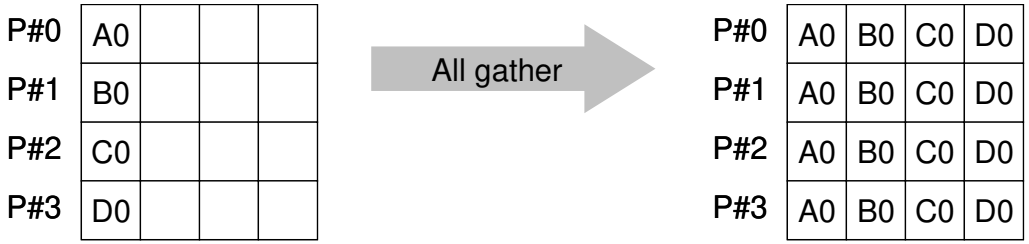


- MPI_Reduce + MPI_Scatter

- **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** MPI_Datatype I メッセージのデータタイプ
  - **op** MPI_Op I 計算の種類
  - **comm** MPI_Comm I コミュニケータを指定する

C

MPI_Allgather

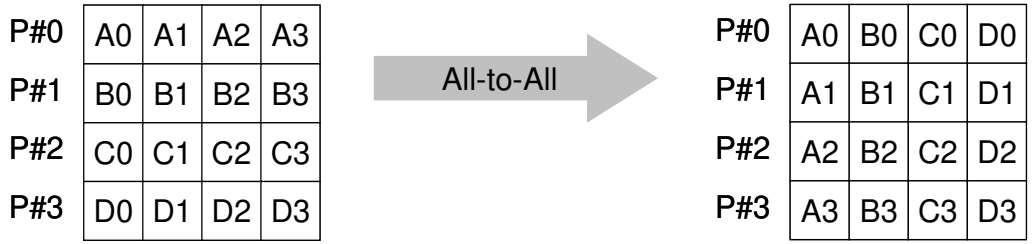


- MPI_Gather＋MPI_Bcast
  - Gatherしたものを，全てのPEにBcastする（各プロセスで同じデータを持つ）

- **MPI_Allgather (sendbuf, scount, sendtype, recvbuf, rcount, recvtype, comm)**
  - **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** MPI_Datatype 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** MPI_Datatype I 受信メッセージのデータタイプ
  - **comm** MPI_Comm I コミュニケータを指定する

C

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** MPI_Datatype 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** MPI_Datatype I 受信メッセージのデータタイプ
  - **comm** MPI_Comm I コミュニケータを指定する

C

集団通信による計算例

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

分散ファイルを使用したオペレーション

- 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することが可能（MPI_Scatter/Gather利用）。

- 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 /work/gt00/t00XXX/pFEM/mpi/S1
>$ module load fj
>$ ls a1.*
   a1.0 a1.1 a1.2 a1.3

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

>$ 実行：4プロセス pjsub go4.sh
```

a1.0
101.0
103.0
105.0
106.0
109.0
111.0
121.0
151.0

a1.1
201.0
203.0
205.0
206.0
209.0
211.0
221.0
251.0

a1.2
301.0
303.0
305.0
306.0
309.0
311.0
321.0
351.0

a1.3
401.0
403.0
405.0
406.0
409.0
411.0
421.0
451.0

go4.sh

```
#!/bin/sh
#PJM -N “test“                          Job Name
#PJM -L rscgrp=tutorial-o               Name of “QUEUE”
#PJM -L node=1                          Node#
#PJM --mpi proc=4                       Total MPI Process#
#PJM -L elapse=00:15:00                 Computation Time
#PJM -g gt00                            Group Name (Wallet)
#PJM -j
#PJM -e err                             Standard Error
#PJM -o test.lst                        Standard Output

module load fj
module load fjmpi

mpiexec ./a.out
```

分散ファイル読み込み：等データ長（2/2）

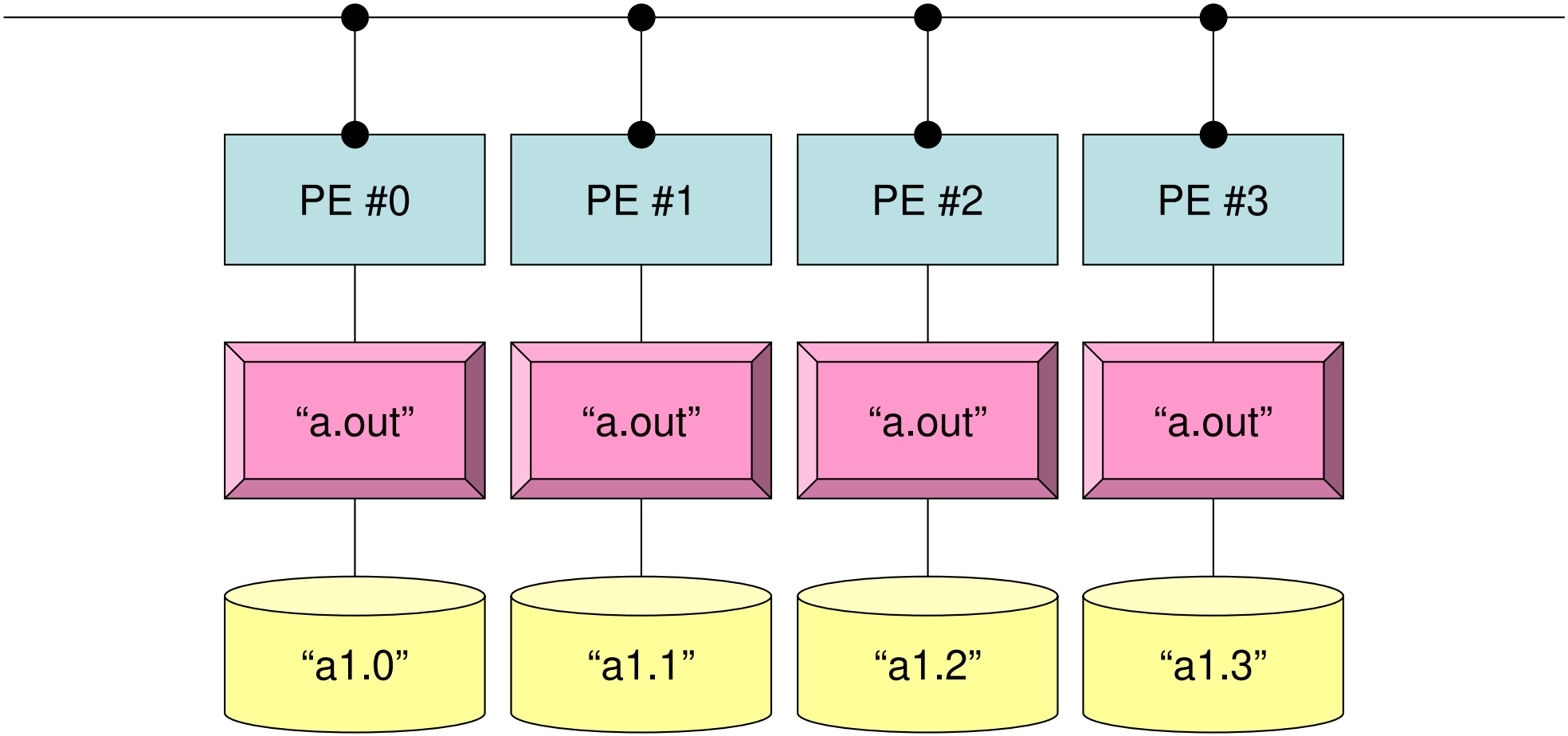
<$O-S1>/file.c

```
int main(int argc, char **argv){
        int i;
        int PeTot, MyRank;
        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 /work/gt00/t00XXX/pFEM/mpi/S1
>$ module load fj
>$ 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 -Nclang -Kfast file2.c
>$ mpifrtpx -Kfast file2.f

>$ 実行：4プロセス pjsub go4.sh
```

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
```

分散ファイルの読み込み：可変長（2/2）

<$O-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();
}
```

num 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

集団通信による計算例

- 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_Gather，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

C

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

- **MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, 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** MPI_Datatype 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** MPI_Datatype I 受信メッセージのデータタイプ
  - **comm** MPI_Comm I コミュニケータを指定する

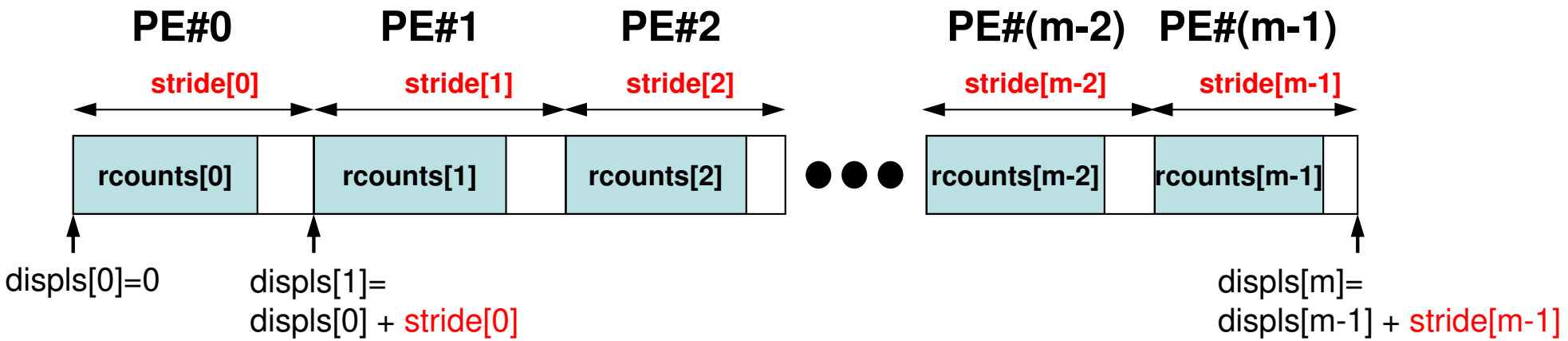
C

MPI_Allgatherv（続き）

- **MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm)**
  - **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。

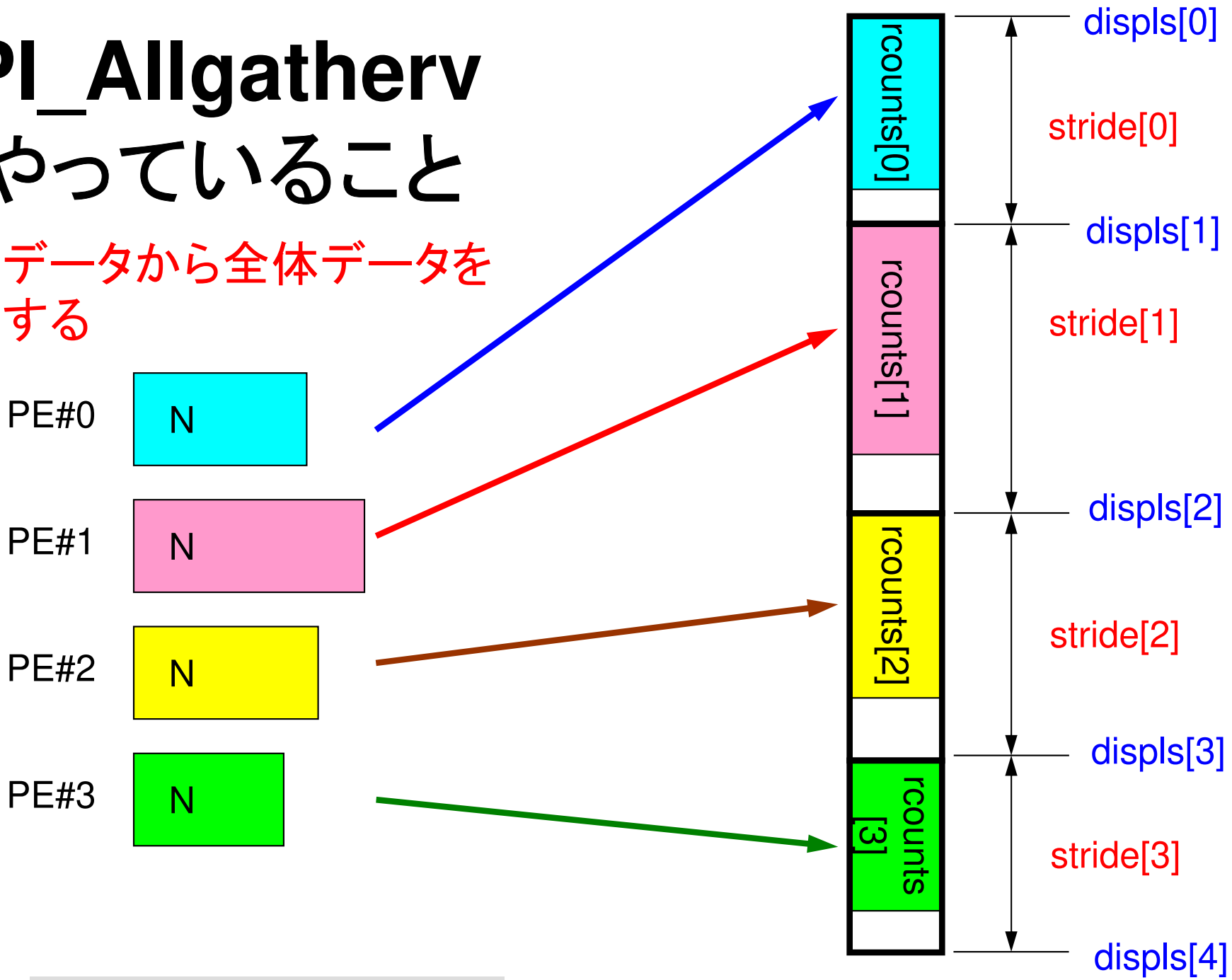
  - 通常はstride(i)=rcounts(i)



size[recvbuf]= displs[PETOT]= 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

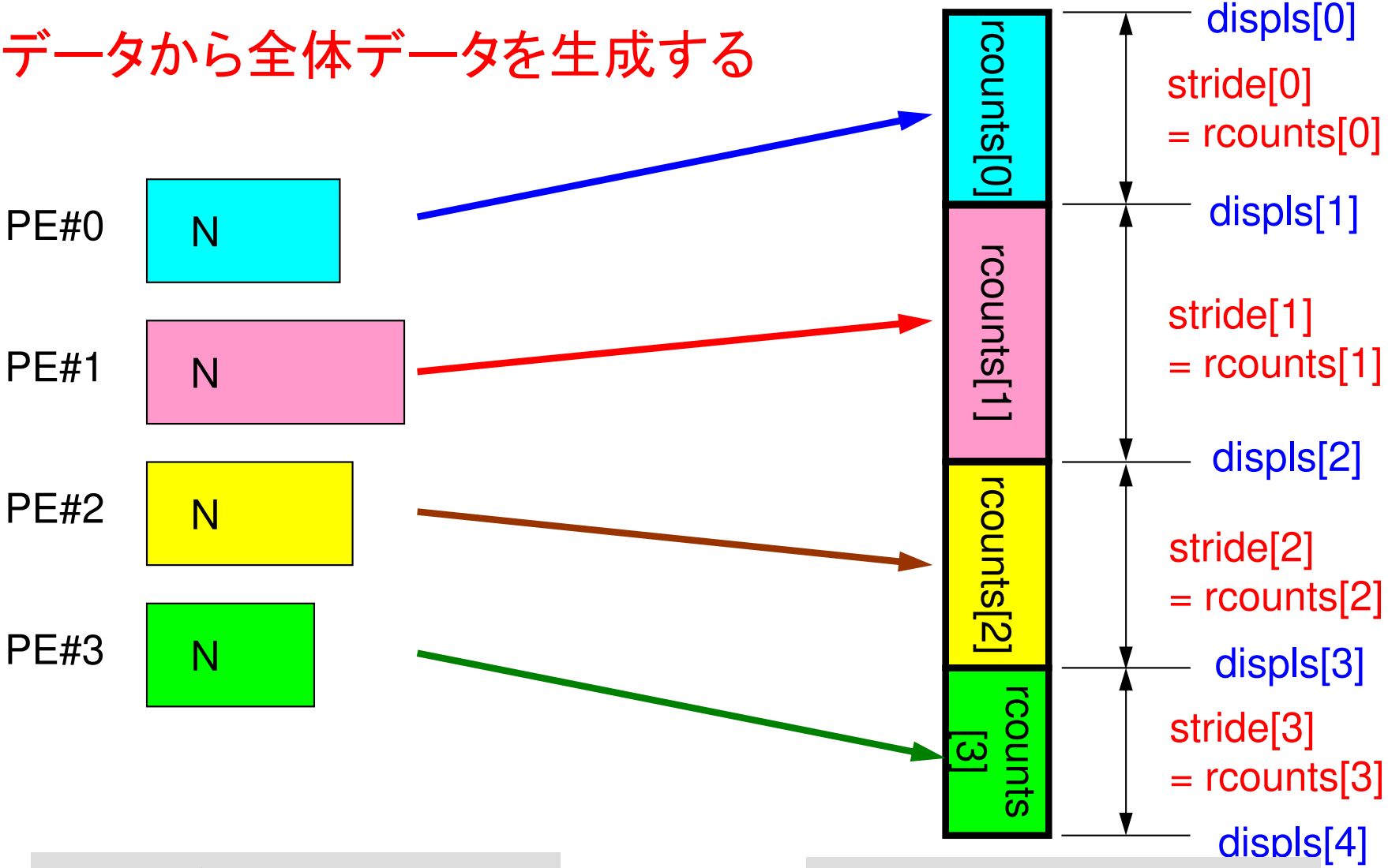


局所データ：sendbuf

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

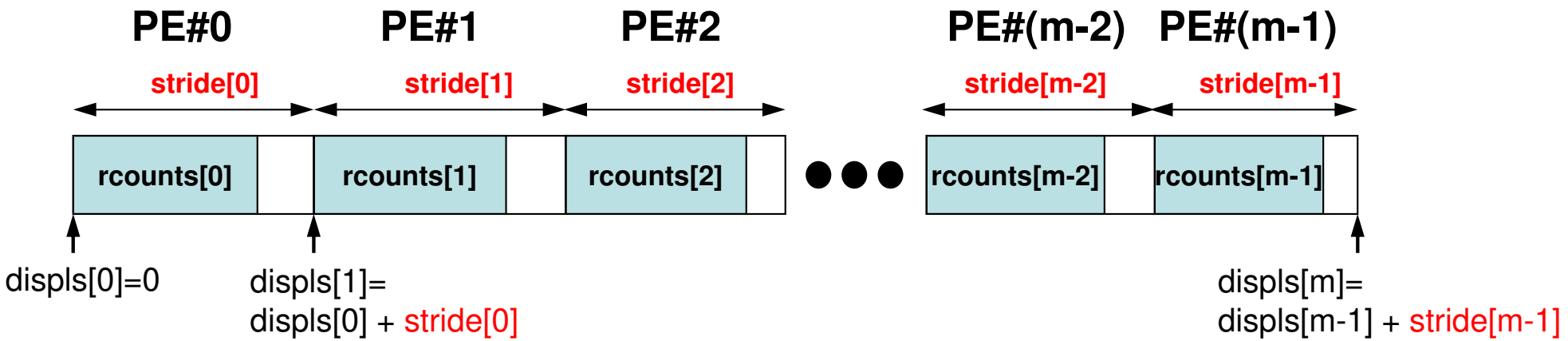
全体データ：recvbuf

C

MPI_Allgatherv詳細(1/2)

- **MPI_Allgatherv (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm)**
  - **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

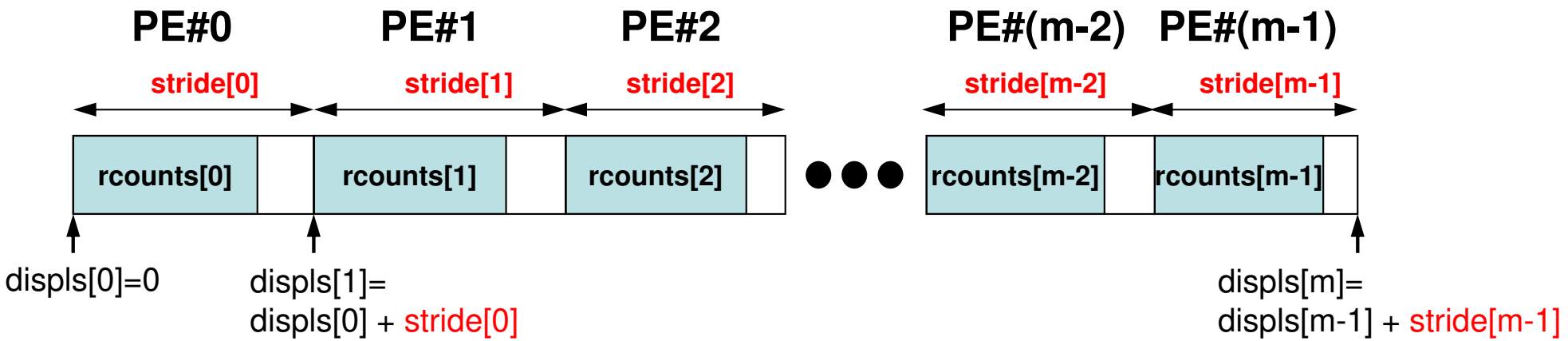


size[recvbuf]= displs[PETOT]= sum[stride]

C

MPI_Allgatherv詳細(2/2)

- **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]= sum[stride]

MPI_Allgatherv使用準備
例題：<$O-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
```

C

MPI_Allgatherv 使用準備(1/4)

<$O-S1>/agv.c

```
int main(int argc, char **argv){
        int i;
        int PeTot, MyRank;
        MPI_Comm SolverComm;
        double *vec, *vec2, *vecg;
        int *Rcounts, *Displs;
        int n;
        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", &n);
        vec = malloc(n * sizeof(double));
        for(i=0;i<n;i++){
                fscanf(fp, "%lf", &vec[i]);
        }
```

n(NL)の値が各PEで異なることに注意

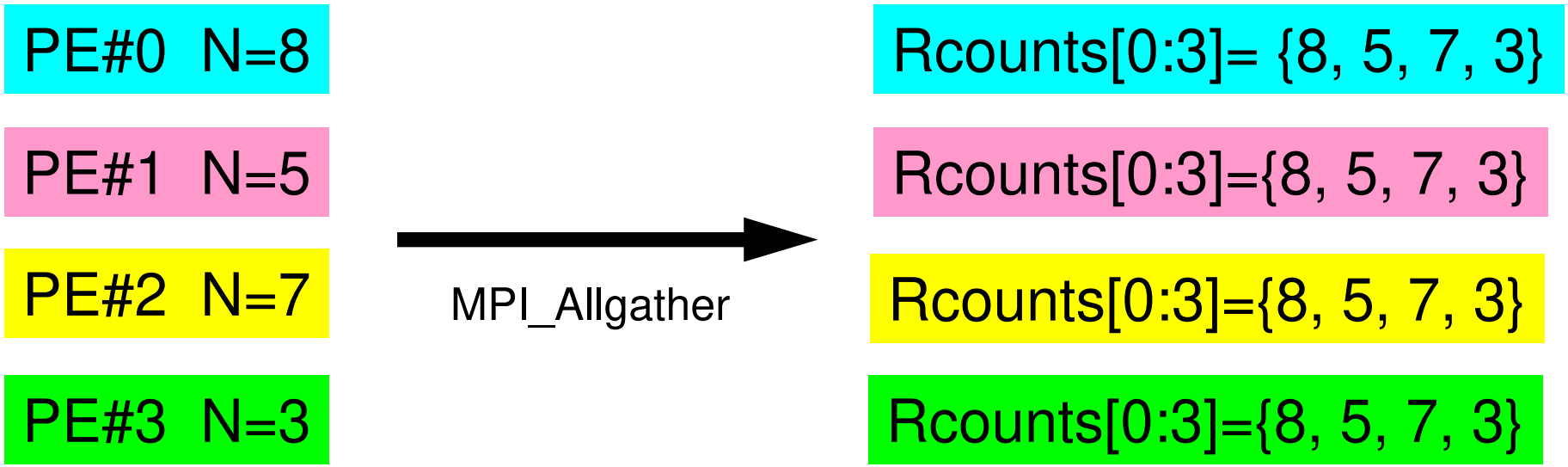
C

MPI_Allgatherv 使用準備(2/4)

<$O-S1>/agv.c

```
Rcounts= calloc(PeTot, sizeof(int));
Displs = calloc(PeTot+1, sizeof(int));

printf("before %d %d", MyRank, n);
for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

MPI_Allgather(&n, 1, MPI_INT, Rcounts, 1, MPI_INT, MPI_COMM_WORLD);

printf("after  %d %d", MyRank, n);
for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

Displs[0] = 0;
```

各PEにRcountsを生成



C

MPI_Allgatherv 使用準備(2/4)

<$O-S1>/agv.c

```
 Rcounts= calloc(PeTot, sizeof(int));
 Displs = calloc(PeTot+1, sizeof(int));

 printf("before %d %d", MyRank, n);
 for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

 MPI_Allgather(&n, 1, MPI_INT, Rcounts, 1, MPI_INT, MPI_COMM_WORLD);

 printf("after  %d %d", MyRank, n);
 for(i=0;i<PeTot;i++){printf(" %d", Rcounts[i]);}

 Displs[0] = 0;
 for(i=0;i<PeTot;i++){
                 Displs[i+1] = Displs[i] + Rcounts[i];}

        printf("CoundIndex  %d ", MyRank);
        for(i=0;i<PeTot+1;i++){
                printf(" %d", Displs[i]);
        }
        MPI_Finalize();
        return 0;
}
```

各PEにRcountsを生成

各PEでDisplsを生成

MPI_Allgatherv 使用準備(3/4)

```
$> cd /work/gt00/t00XXX/pFEM/mpi/S1
$> module load fj
$> mpifccpx -Nclang -Kfast agv.c
$> 実行:4プロセス 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
```

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]」

```
MPI_Allgatherv
      ( VEC , N, MPI_DOUBLE,
        recvbuf, rcounts, displs, MPI_DOUBLE,
        MPI_COMM_WORLD);
```

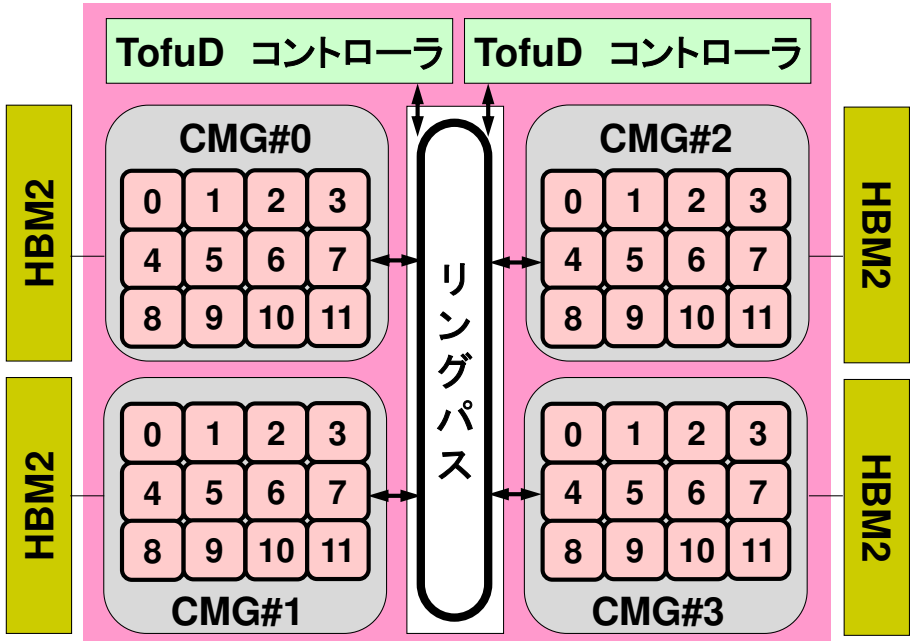
課題S1

- 内容
  - 「<$O-S1>/a1.0～a1.3」，「 <$O-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）。
    - <$O-S1>file.f，<$T-S1>file2.fをそれぞれ参考にする。
  - 「<$O-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）。
  - 下記の数値積分の結果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）。

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

プロセス数

```
#PJM -L node=1;#PJM --mpi proc= 1    1-node, 1-proc, 1-proc/n
#PJM -L node=1;#PJM --mpi proc= 4    1-node, 4-proc, 4-proc/n
#PJM -L node=1;#PJM --mpi proc=12    1-node,12-proc,12-proc/n
#PJM -L node=1;#PJM --mpi proc=24    1-node,24-proc,24-proc/n
#PJM -L node=1;#PJM --mpi proc=48    1-node,48-proc,48-proc/n

#PJM -L node= 4;#PJM --mpi proc=192   4-node,192-proc,48-proc/n
#PJM -L node= 8;#PJM --mpi proc=384   8-node,384-proc,48-proc/n
#PJM -L node=12;#PJM --mpi proc=576  12-node,576-proc,48-proc/n
```



a012.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial-o
#PJM -L node=1
#PJM --mpi proc=12
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

module load fj
module load fjmpi
mpiexec ./a.out
mpiexec numactl -l ./a.out
```

a048.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial-o
#PJM -L node=1
#PJM --mpi proc=48
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

module load fj
module load fjmpi
mpiexec ./a.out
mpiexec numactl -l ./a.out
```

a384.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial-o
#PJM -L node=8
#PJM --mpi proc=384
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

module load fj
module load fjmpi
mpiexec ./a.out
mpiexec numactl -l ./a.out
```

a576.sh

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial-o
#PJM -L node=12
#PJM --mpi proc=576
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

module load fj
module load fjmpi
mpiexec ./a.out
mpiexec numactl -l ./a.out
```

numactl -l/--localalloc ローカルなメモリを使用（効果はほとんど無い）