連絡事項

- 「履修アンケート」未送付の人は中島まで送付してください。
- 前回講義データ
  - input.dat, cinput.dat
  - 1.0d-07 ⇒ 1.d-7 コンパイラによっては受け付けない場合あり

MPIによるプログラミング概要
その1

2007年4月25日

中島研吾

並列計算プログラミング（616-2057）・先端計算機演習I（616-4009）

本授業の理念・・・より

- 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。
  - 理想：Scalable
    - N倍の規模の計算をN倍のCPUを使って，「同じ時間で」解く

概要

- MPIとは
- PC Cluster “CENJU” について

- 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）

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）
- 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
- 竹内則夫，平野 広和「FORTRAN77とFORTRAN90」，森北，1994.
  - FORTRAN90の特徴
    - 配列のallocate，deallocate
    - ポインタ，構造体を使うことができる
    - モジュール文により，機能的なプログラミングが可能：コモンブロックの代替にもなる
    - 自由書式

MPIを学ぶにあたって（1/2）

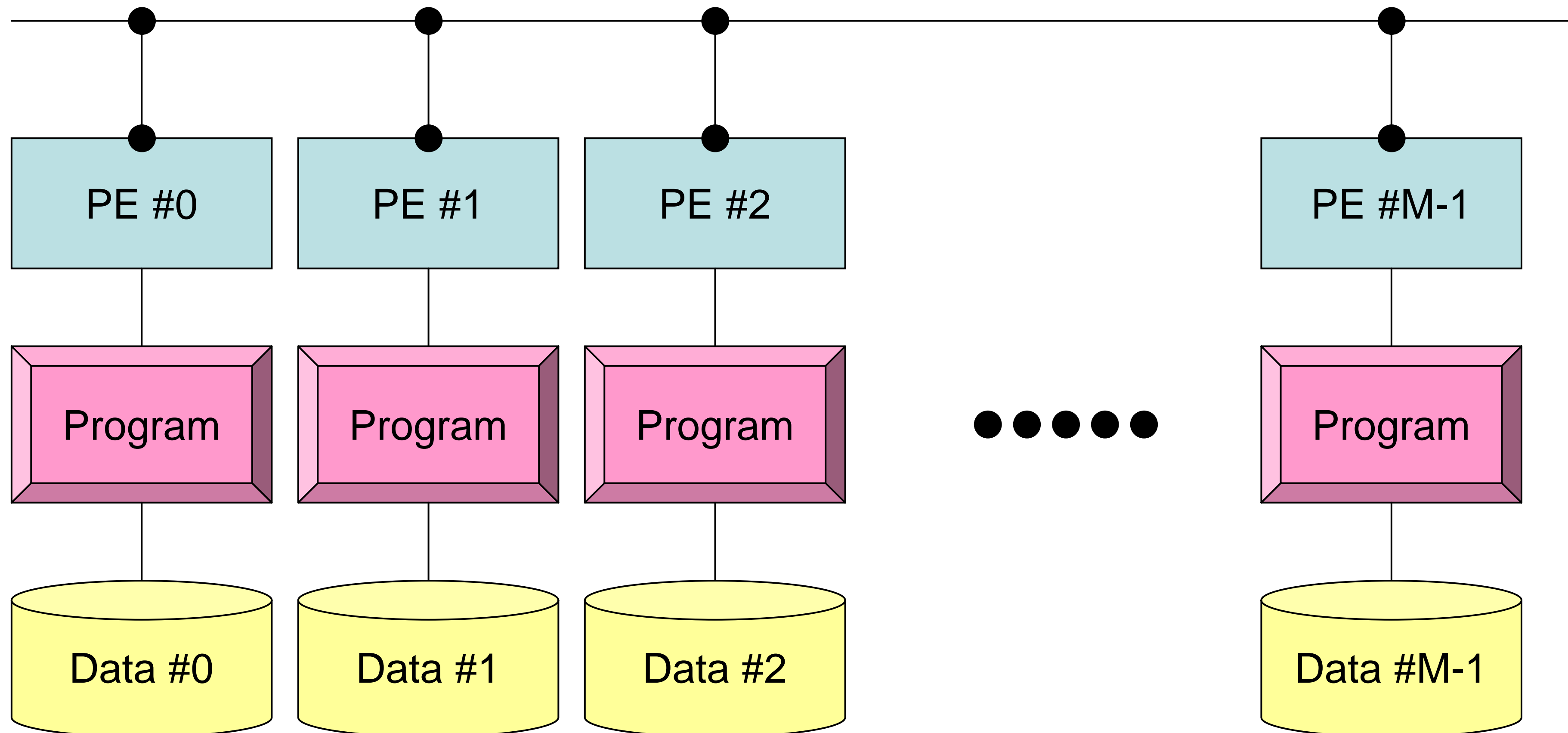
- 文法
  - 「MPI-1」の基本的な機能（10程度）について習熟する
  - あとは自分に必要な機能について調べる，あるいは知っている人，知っていそうな人に尋ねる
- 実習の重要性
  - プログラミング
  - その前にまず実行してみること
- SPMD/SIMDのオペレーションに慣れること･･･「つかむ」こと
  - Single Program/Instruction Multiple Data
  - 基本的に各プロセスは「同じことをやる」が「データが違う」
    - 大規模なデータを分割し，各部分について各プロセス（プロセッサ）が計算する
  - 全体データと局所データ，全体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

PE: Processing Element
プロセッサ，領域，プロセス

SPMD/SIMD

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

用 語

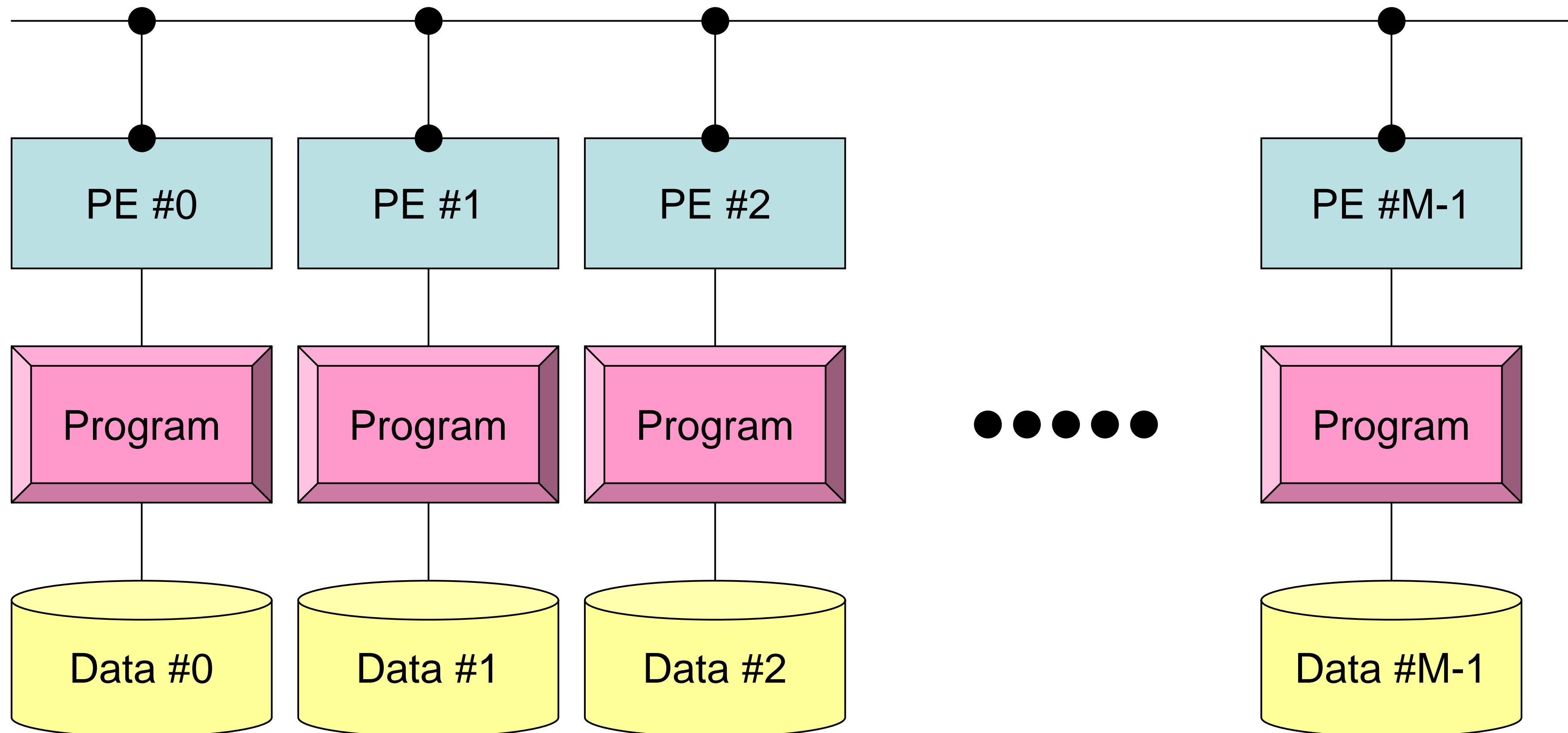
- プロセッサ，コア
  - ハードウェアとしての各演算装置。シングルコアではプロセッサ＝コア
- プロセス
  - MPI計算のための実行単位，ハードウェア的な「コア」とほぼ同義。
  - しかし1つの「プロセッサ・コア」で複数の「プロセス」を起動する場合もある（効率的ではないが）。
- PE（Processing Element）
  - 本来，「プロセッサ」の意味なのであるが，本講義では「プロセス」の意味で使う場合も多い。次項の「領域」とほぼ同義でも使用。
    - マルチコアの場合は：「コア＝PE」という意味で使うことが多い。
- 領域
  - 「プロセス」とほぼ同じ意味であるが，SPMD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

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

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

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

- 2007年4月25日，5月2日，5月9日，（＋5月16日）
  - 環境管理，グループ通信（Collective Communication）
    - 課題S1
  - 1対1通信（Point-to-Point Communication）
    - 課題S2：一次元熱伝導解析コードの「並列化」
  - ここまでできればあとはある程度自分で解決できます

- MPIとは
- PC Cluster "CENJU" について

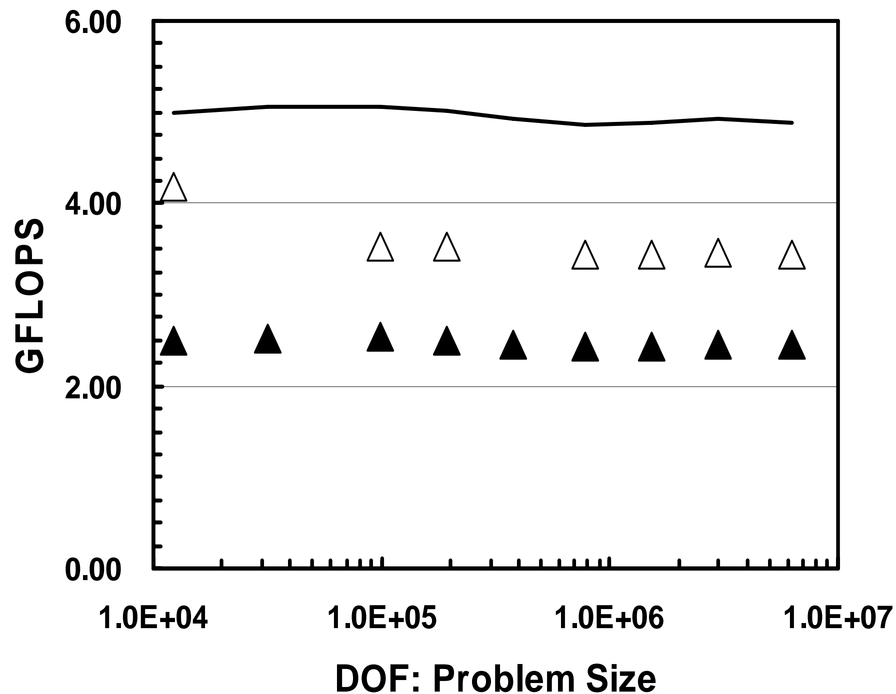
- MPIの基礎：Hello World

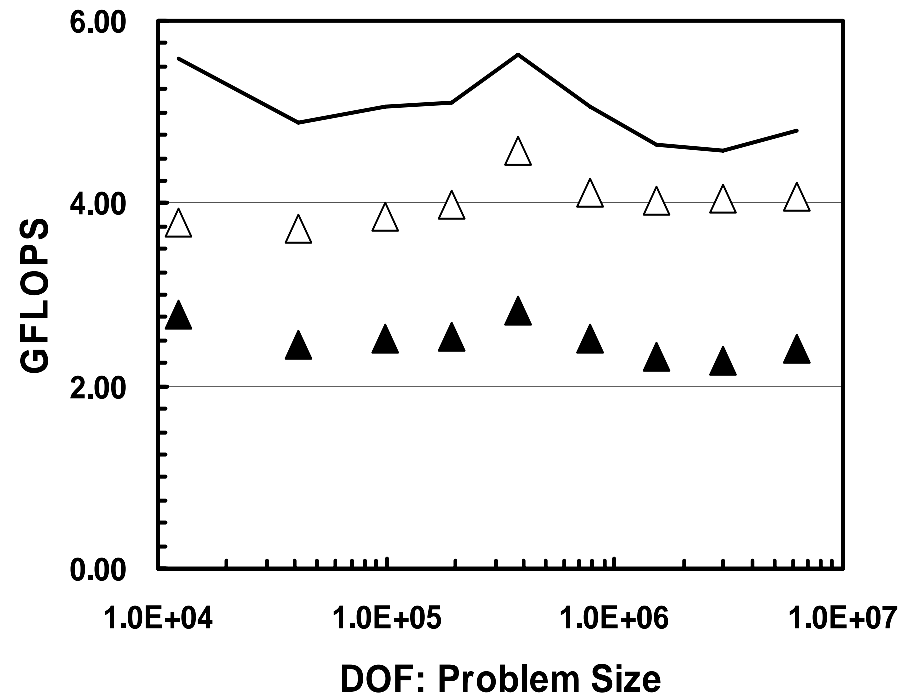
- 全体データと局所データ
- グループ通信（Collective Communication）

PCクラスタの概要

- cenju.eps.s.u-tokyo.ac.jp
  - http://www-solid.eps.s.u-tokyo.ac.jp/~nakajima/07s/PCcluster/index.html
- Opteron 1.8GHz x 16 nodes + Control node (1PE)
  - 各ノード
    - 2 PE's on each node (32 PE's total), 100GB local HD
    - 2 GB RAM on each node
    - 1 MB L2 cache on each node
  - Gigabit Ethernet
  - 1 TB HD
- SSH login，SCP
  - 外部から直接接続可能
- PGI Compiler
- プリンタは接続されていない

cenjuの性能





Opteronは各CPUにコントローラが内蔵されているために，Xeon，Pentiumと比較して，メモリを効率的に使うことができる。したがって2PE/nodeとしても性能はある程度保たれる。

VT64 Opteron Cluster システム構成図

(Opteron 16Node/32CPU + 1Nod e/1CPU)

VISUAL TECHNOLOGY
Jan. / 2004 Ver. 1

※1ラック分で 100V/30A 電源 x 2 を使用
(各Node、各接続機器へは、UPSより電源を供給)

【 管理Nodeの構成 】
- ・CPU　: AMD Opteron 146 (1.8GHz)
  : 1MB L2 cash , AMD 8131+8111 chipset
- ・メモリ　: 2GB (512MB x 4)
- ・HDD　: UltraATA100/133 120GB
- ・NIC　: 1000Base-T x 2 (On-board)
- ・Video　: ATIRageXL 8MB (On-board)
- ・Etc.　: FDD
- ・OS　: Turbolinux8 for AMD64
- ・消費電力 : 最大250W

<1.28TB RAID System>
- ・HDD　: UltraATA100/133 160GB x 8 (RAID 0,1,5,10)
  ※管理NodeとSCSIで接続
- ・消費電力 : 最大600W

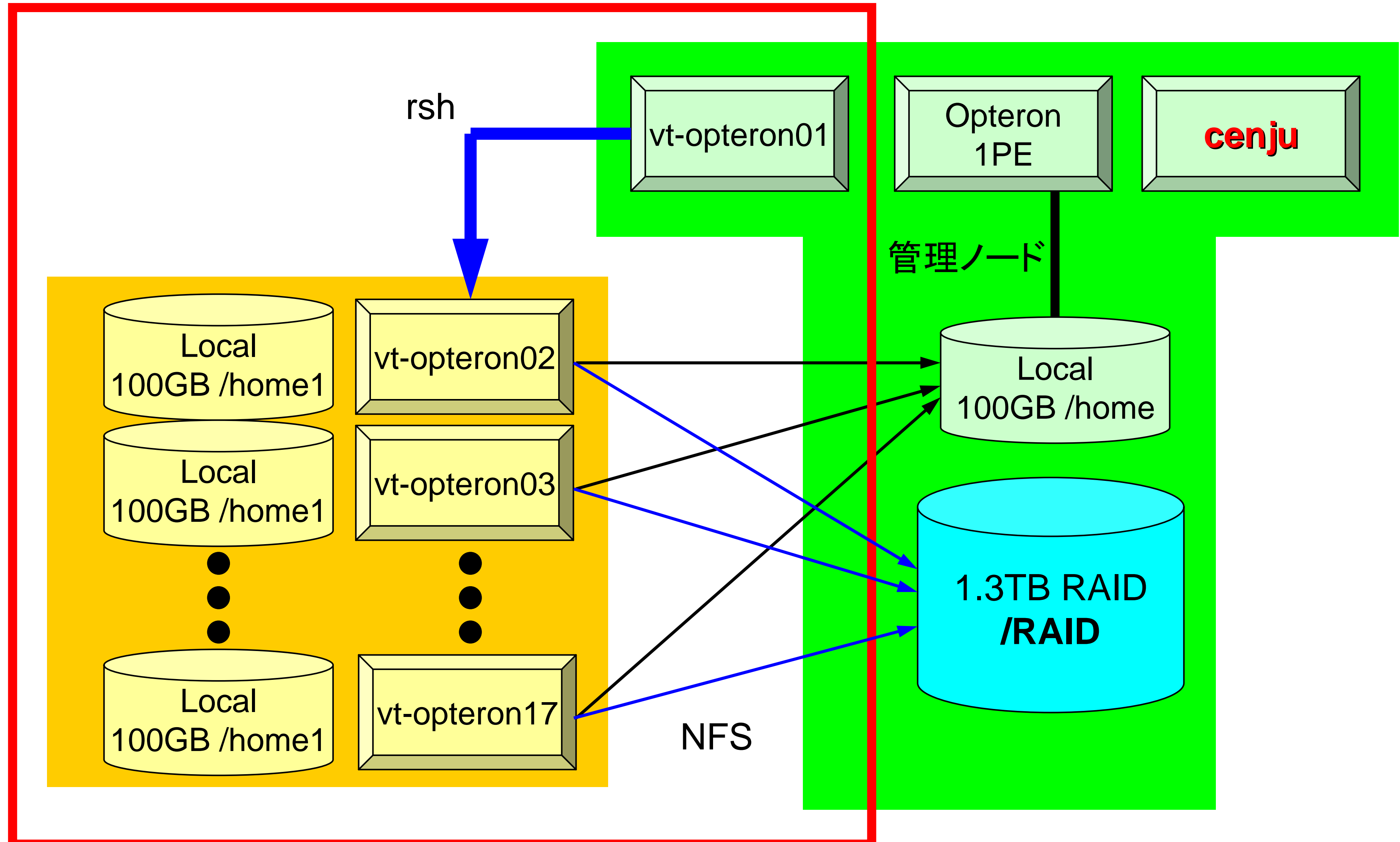
【 計算Node1Nodeあたりの構成 】
- ・CPU　: AMD Opteron 246(1.8GHz) Dual
  : 1MB L2 cash , AMD 8131+8111 chipset
- ・メモリ　: ~~1GB (512MB x 2)~~ 2GB
- ・HDD　: UltraATA100/133 120GB
- ・NIC　: 1000Base-T x 2 (On-board)
- ・Video　: ATIRageXL 8MB (On-board)
- ・Etc.　: FDD
- ・OS　: Turbolinux8 for AMD64
- ・消費電力 : 最大250W

【 システム構成 】
- ・ケース　: 38U 19inchラックキャビネットに搭載
  ※各Node、UPSより電源供給
- ・ノード　: 16Node/32CPU + 1Node/1CPU
  ※システム内部プライベートIP設定
  ※内1台管理ノード兼用、グローバルIP、システム内部プライベートIP設定
- ・ストレージ : 1.28TB RAID subsystem
  ※管理Nodeに接続
- ・Network : Gigabit Ethernet接続
  ※24ch 1000Base-T Switching HUBにて接続
  ※消費電力=最大72W (x4)
  ※システム内部ネットワーク、システム外部ネットワーク(管理ノード)共に1000Base-T

【 OS、コンパイラ 、その他ソフトウエア 】
- ・Turbolinux8 for AMD64
- ・Compiler PGI Workstation 5.0 for AMD64
- ・GNU Compiler
- ・並列化ソフトウエア(ノード間通信ライブラリ) : MPICH
- ・クラスタ管理ソフトウェア　: VT-CMS Lite

- 制御ノード＋16ノード
- ネットワークから見えているのは，制御ノードのみ
  - cenju.eps.s.u-tokyo.ac.jp
- ローカルノード名
  - 制御：vt-opteron01
  - 計算：vt-opteron02～17

PCクラスタのファイルシステム



利用にあたってのルール(1/2)

- 基本的に本授業以外の目的では使わない
  - 個別に相談には乗ります

- ID，パスワードの管理徹底
  - 初期パスワードは速やかに変更
  - (あたりまえの話であるが)他の人には教えないこと
- ジョブ関連
  - 複数PEを使う場合にはバッチジョブ(qsub使用：後述)
  - 利用ノードの上限は当面4ノード(8 CPU)
  - 一回のジョブ投入は2つまで
  - ジョブの制限時間は10分まで

利用にあたってのルール（2/2）

- ハードディスク使用
  - 特に制限はしないが，バックアップ目的等に使用しないこと
  - coreファイルを貯めない
    - 作業ディレクトリに「core」というディレクトリを作っておくと良い
  - 「/home」はすぐパンクするので，出力ファイル等は「/RAID」の下に自分のUIDでディレクトリを作り，そこへ保存する。
- 問い合わせ
  - 中島（1号館716） nakajima＠eps.s.u-tokyo.ac.jp

プログラミング環境

- GUIモード
  - startx
- エディター
  - vi, xemacs
- コンパイラ
  - PGIコンパイラ（Portland Group）: version 6に更新
  - CはGNUもある
- MPI
  - mpich 1.2.5
- OpenMP
  - コンパイラがサポート：複数ノード利用のMPIでは動かない：

PGIコンパイラ

- 詳細はhttp://www.pgroup.com/
- F77, F90, C, C++, OpenMP, HPFをサポート
- 呼び出しコマンド
  - pgf77, pgf90, pgcc, pgCC
- コンパイルオプション
  - -O2, -O3がお勧め, -fast(-O2＋高速化), -fastsse(-fast＋sse/sse2命令セット)(Streaming SIMD Extensions)
- マニュアル
  - /usr/pgi/linux86-64/6.0/doc
  - manpage
- 64ビットモードにバグがあった(特にC)が, アップグレードにより, ほぼ解決した模様

並列計算関連

- MPI
  - mpif77, mpif90, mpicc, mpiCC
- OpenMP
  - 「-mp」をコンパイルオプションとして指定
  - export OMP_NUM_THREADS=2 （bash）
  - setenv OMP_NUM_THREADS 2 （csh）

- MPIとは
- PC Cluster “CENJU” について

- 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）

ログイン，ディレクトリ作成

```
ssh XXXXX@cenju.eps.s.u-tokyo.ac.jp
```

パスワード変更

```
>$ passwd
```

RAIDの下に自分のディレクトリ作成

```
>$ cd /RAID
>$ mkdir <自分のUserID>
```

ディレクトリ作成

```
>$ cd
>$ mkdir 07summer   （好きな名前でよい）
>$ cd 07summer
```

このディレクトリを本講義では <$07S> **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

この下に課題番号に応じて **S1, S2, S1-ref** などのディレクトリを作る

<$S1> = <$07S>/S1
<$S2> = <$07S>/S2

ファイルコピー

FORTRANユーザー

```
>$ cp /home/nakajima/class/2007summer/F/s1-f.tar .
>$ tar xvf s1-f.tar
```

Cユーザー

```
>$ cp /home/nakajima/class/2007summer/C/s1-c.tar .
>$ tar xvf s1-c.tar
```

一次元熱伝導方程式

```
>$ cp /home/nakajima/class/2007summer/intro.tar .
>$ tar xvf intro.tar
```

作業ディレクトリ

ディレクトリ確認

```
>$ ls
    S1

>$ cd S1
```

このディレクトリを本講義では `<$S1>` **と呼ぶ。**

```
<$S1> = <$07S>/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();
}
```

実行してみよう（4-PE を使用）

FORTRAN ユーザー

```
$> cd <$S1>
$> mpif90 -O3 hello.f
$> mpirun -np 4 a.out

Hello World FORTRAN        0        4
FORTRAN STOP
Hello World FORTRAN        1        4
FORTRAN STOP
Hello World FORTRAN        3        4
Hello World FORTRAN        2        4
FORTRAN STOP
FORTRAN STOP
```

Cユーザー

```
$> cd <$S1>
$> mpicc  -O3 hello.c
$> mpirun -np 4 a.out

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

このようにならない人は至急中島まで連絡してください（挙手）。

プログラムのコンパイル，実行

FORTRAN

```
$> mpif90 -O3 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 -O3 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

実行

```
$> mpirun -np 4 a.out
   mpirun -np <プロセス数> <プログラム名>
```

プロセス数＝ハードウェア的CPU数とは限らない
1CPUで複数のプロセスを立ち上げることも可能である

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

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（ierror）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？

```
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
```

```
$> mpif90 -O -o hello.f
$> mpirun -np 4 a.out

Hello World FORTRAN        0        4
Hello World FORTRAN        1        4
FORTRAN STOP
FORTRAN STOP
Hello World FORTRAN        3        4
Hello World FORTRAN        2        4
FORTRAN STOP
FORTRAN STOP
```

- `mpirun -np 4 <prog>` により4つのプロセスが立ち上がる。
  - 同じプログラムが4つ流れる。
  - データの値（PETOT, 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サブルーチンより前にコールする必要がある（必須）

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

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

MPI_FINALIZE

- MPIを終了する。他の全てのMPIサブルーチンより後にコールする必要がある（必須）。
- これを忘れると大変なことになる。
  - 終わったはずなのに終わっていない・・・

- **call MPI_FINALIZE (ierr)**
  - **ierr** 整数 O 完了コード

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

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。

- **call MPI_COMM_SIZE (comm, size, ierr)**
  - <u>**comm**</u> 整数 I コミュニケータを指定する
  - <u>**size**</u> 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
  - <u>**ierr**</u> 整数 O 完了コード

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

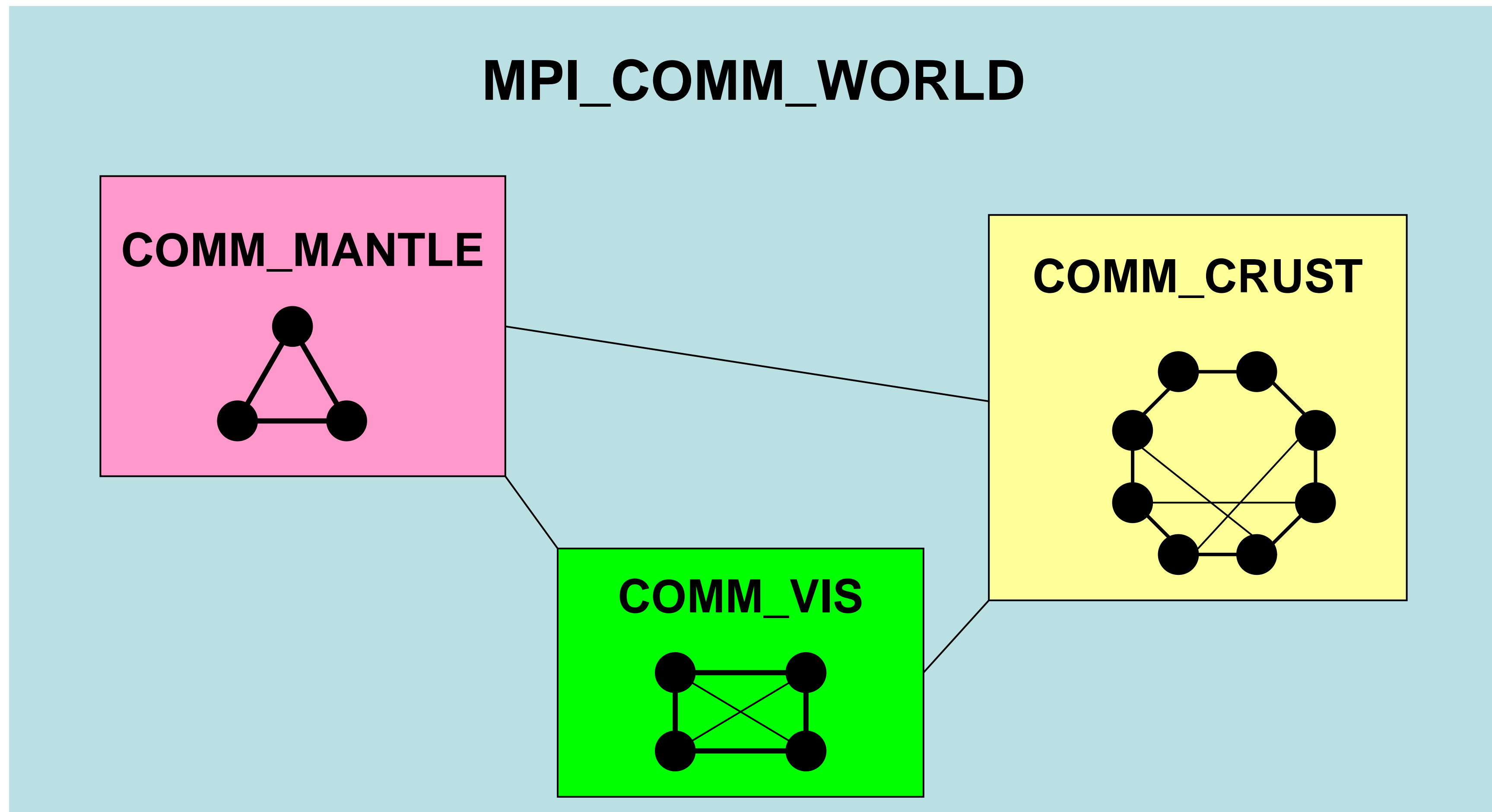
コミュニケータとは？

```
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
```

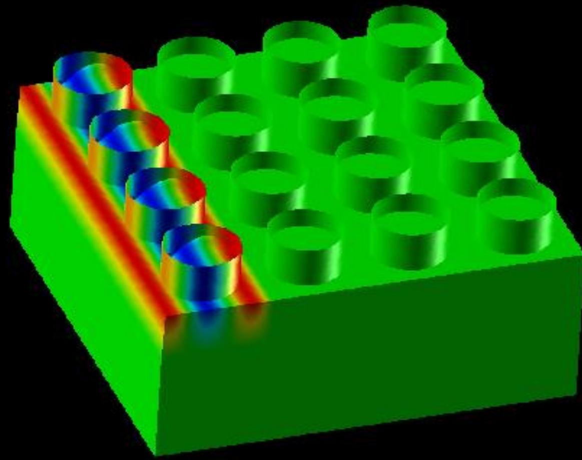
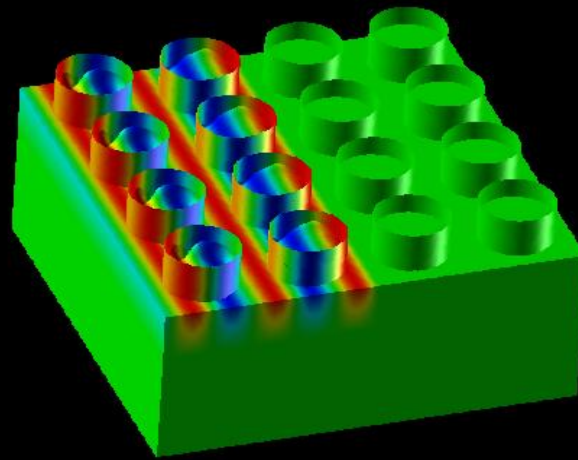
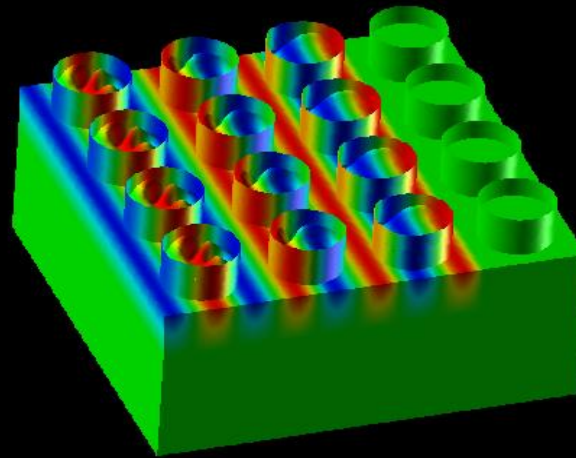
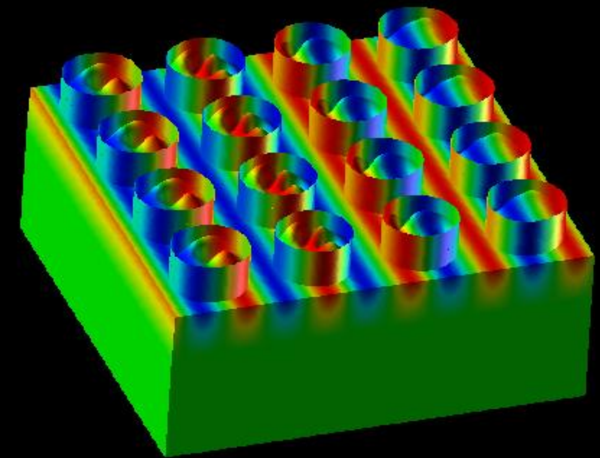
- 通信を実施するためのプロセスのグループを示す。
- MPIにおいて，通信を実施する単位として必ず指定する必要がある。
- mpirunで起動した全プロセスは，デフォルトで「**MPI_COMM_WORLD**」というコミュニケータで表されるグループに属する。
- 複数のコミュニケータを使用し，異なったプロセス数を割り当てることによって，複雑な処理を実施することも可能。
  - 例えば計算用グループ，可視化用グループ
- この授業では「**MPI_COMM_WORLD**」のみでOK。

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

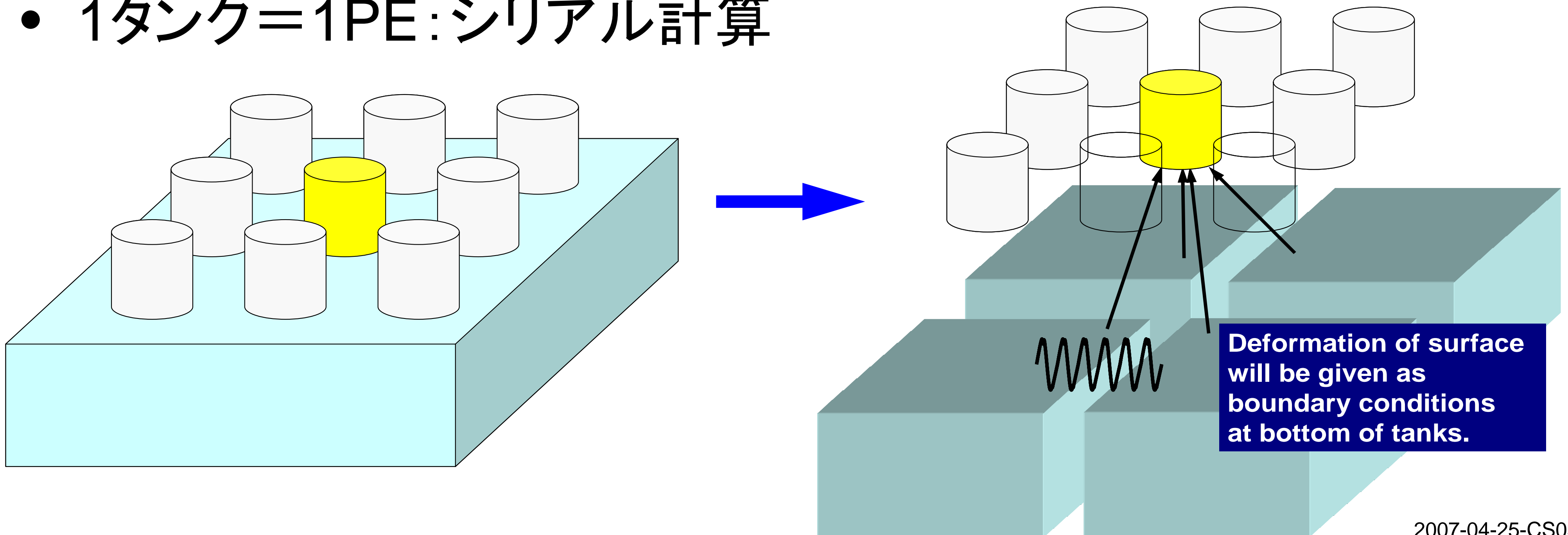


地盤・石油タンク連成シミュレーション

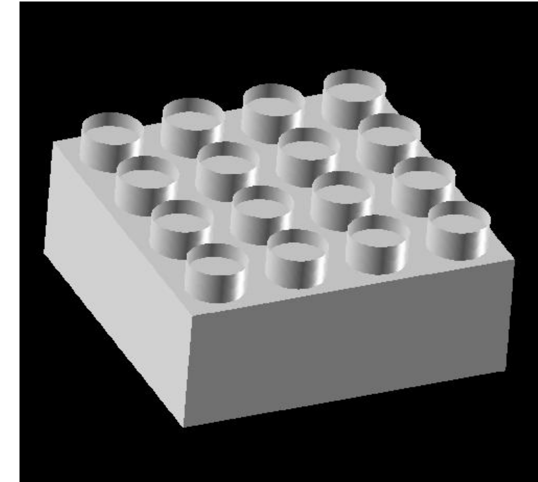
対象とするアプリケーション

- **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

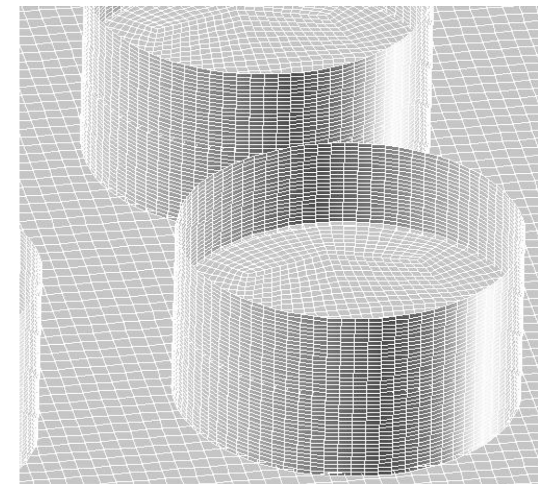


地盤，タンクモデル

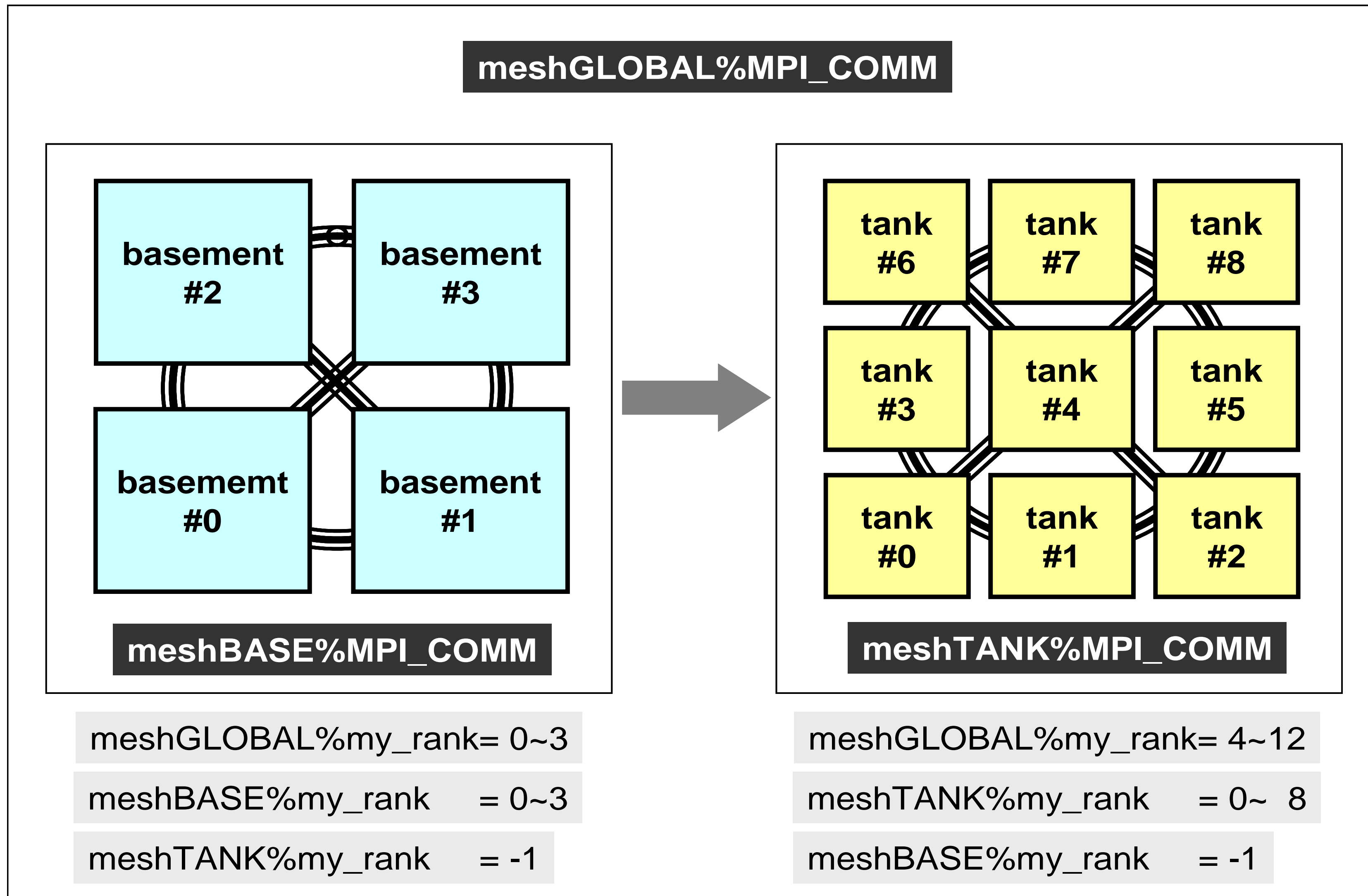
- **地盤モデル（市村）**FORTRAN
  - 並列FEM，三次元弾性動解析
    - 前進オイラー陽解法，EBE
  - 各要素は一辺2mの立方体
  - 240m×240m×100m



- **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



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

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

MPI_ABORT

- MPIプロセスを異常終了する。

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

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）

```
…
real(kind=8):: Stime, Etime

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

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

MPI_WTIME の例

```
$> cd <$S1>
$> mpif90 time.f or mpicc -O3 time.c

$> mpirun -np 4 a.out

    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
```

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

```
$> cd <$S1>

$> mpif90 -O3 wtick.f
$> mpirun -np 1 a.out
  3.906250000000000E-003
```
この値が計測精度

```
$> mpicc -O3 wtick.c
$> mpirun -np 1 a.out
   0    3.906250E-03
```
この値が計測精度

MPI_BARRIER

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

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

ジョブ実行

- 小規模，短時間であれば，「mpirun -np …」とフォアグラウンドで端末画面から実行しても良いが，できるだけ「バッチジョブ」で実行する。
  - 空いているCPUを自動的に探し出してくれるなど，効率が良い

- PBS（Portable Batch System）
  - NASAで開発されたバッチジョブ管理システム
  - もともとパブリックドメイン（OpenPBS），最近は商用化されたPBS Proも広く使用されている
    - cenjuでは当然「OpenPBS」が使用されている
  - http://www.openpbs.org/

ジョブ実行の手順

- シェルスクリプトを作成する
  - 各<S1>ディテクトリの ”go.sh” を参照

- 基本コマンド
  - qsub　ジョブ投入
  - qdel　ジョブ削除
  - qstat　キュー，ジョブの状態表示
  - pbsnodes　計算ノード状態の確認

シェルスクリプトの例

```
#!/bin/sh
#PBS -N hello-test           ジョブ名
#PBS -o test.lst             標準出力ファイル名
#PBS -e err                  エラー出力ファイル名
#PBS -l nodes=2:ppn=2        使用ノード数(nodes)，ノードあたりPE数(ppn)
cd $PBS_O_WORKDIR
NPROCS=`wc -l < $PBS_NODEFILE`
mpirun -v -machinefile $PBS_NODEFILE -np $NPROCS a.out
```

ロードモジュール名：任意

- 太字の部分は変えなくて良い
  - ノードあたりのプロセッサ数は2にしておいてください
  - 奇数CPU使用の場合には最終行の「$NPROCS」をCPU数（例えば3）に変えてください
- 行先頭に「!」がくるとコメント行

シェルスクリプトの例：CPU数3の例

```
#!/bin/sh
#PBS -N hello-test              ジョブ名
#PBS -o test.lst                標準出力ファイル名
#PBS -e err                     エラー出力ファイル名
#PBS -l nodes=2:ppn=2           使用ノード数（nodes），ノードあたりPE数（ppn）
cd $PBS_O_WORKDIR
NPROCS=`wc -l < $PBS_NODEFILE`
mpirun -v -machinefile $PBS_NODEFILE -np 3 a.out
```

- 「#PBS -l nodes=2:ppn=2」の情報は結果的に無視される

実行 qsub（1/2）

- qsub <オプション> <シェルスクリプト名>で実行
- オプション
  - **-N** ＜ジョブの名前＞
  - **-o** ＜標準出力ファイル名＞
  - **-e** ＜エラー出力ファイル名＞
  - **-l nodes=XX:ppn=YY**
  - **-l walltime=10:00** 実行時間制限（この場合は10分）

- 上記オプションはシェルスクリプト内で「#PBS」として指定することもできるし，qsub実行時のオプションとしても指定できる（実行時指定が優先）。

実行 qsub（2/2）

- 標準出力，標準エラー出力名を指定しない場合は:
  - ジョブ名（-Nで指定）＋「.oジョブ番号」「. eジョブ番号」
    - e.g. cube-test.o205, mpi-test.e325
  - ジョブ名が指定されていない場合には，「シェルスクリプト名」＋「.oジョブ番号」「. eジョブ番号」
    - e.g. go.sh.o205, go.sh.e325

実行してみよう

```
$> cd S1

$> cat go.sh
  #!/bin/sh
  #PBS -N test
  #PBS -o test.lst
  #PBS -e err
  #PBS -l nodes=2:ppn=2
  #PBS -l walltime=10:00
  cd $PBS_O_WORKDIR
  NPROCS=`wc -l < $PBS_NODEFILE`
  mpirun -v -machinefile $PBS_NODEFILE -np $NPROCS a.out

$> qsub go.sh

$> cat test.lst
  Hello World FORTRAN       1       4
  Hello World FORTRAN       3       4
  Hello World FORTRAN       2       4
  Hello World FORTRAN       0       4
```

ジョブの実行状況（1/4）

qstat -a：実行状況全般

```
vt-opteron01.localdomain:
                                                                   Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
219.vt-opteron0 nakajima default  cube-test   27273   8  --     -- 7200: R   --
220.vt-opteron0 nakajima default  cube-test   27508   8  --     -- 7200: R   --
221.vt-opteron0 nakajima default  cube-test   25792   8  --     -- 7200: R   --
222.vt-opteron0 nakajima default  cube-test   26027   8  --     -- 7200: R   --
223.vt-opteron0 nakajima default  cube-test     --    8  --     -- 7200: Q   --
```

- 実行時間は長時間ジョブでないとうまく表示できないらしい（設定の問題，バグの可能性もあり）。
- S:ジョブの実行状況
  - R: 実行中，Q：待ち

ジョブの実行状況（2/4）

qstat -f <JOB ID>：ジョブ実行状況の詳細

```
[nakajima]$ qstat -f 219

Job Id: 219.vt-opteron01.localdomain
    Job_Name = cube-test
    Job_Owner = nakajima@vt-opteron01.localdomain
    resources_used.cput = 00:00:00
    resources_used.mem = 3844kb
    resources_used.vmem = 21864kb
    resources_used.walltime = 00:01:20      実行時間
    job_state = R
    queue = default
    server = vt-opteron01.localdomain
```

ジョブの実行状況（3/4）

qdel <JOB ID>：実行中止

```
[nakajima]$ qdel 220

[nakajima]$ qstat
Job id           Name             User             Time Use S Queue
---------------- ---------------- ---------------- -------- - -----
219.vt-opteron01 cube-test        nakajima         00:00:00 R default
221.vt-opteron01 cube-test        nakajima                0 R default
222.vt-opteron01 cube-test        nakajima                0 R default
223.vt-opteron01 cube-test        nakajima                0 R default
```

ジョブの実行状況（4/4）

pbsnodes -a：ノードの使用状況

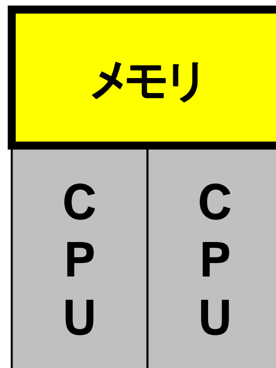
```
vt-opteron02.localdomain
     state = job-exclusive
     np = 2
     ntype = cluster
     jobs = 0/219.vt-opteron01.localdomain, 1/223.vt-opteron01.localdomain

vt-opteron03.localdomain
     state = job-exclusive
     np = 2
     ntype = cluster
     jobs = 0/219.vt-opteron01.localdomain, 1/223.vt-opteron01.localdomain

vt-opteron04.localdomain
     state = job-exclusive
     np = 2
     ntype = cluster
     jobs = 0/219.vt-opteron01.localdomain, 1/223.vt-opteron01.localdomain

vt-opteron05.localdomain
     state = job-exclusive
     np = 2
     ntype = cluster
     jobs = 0/219.vt-opteron01.localdomain, 1/223.vt-opteron01.localdomain

（中略）

vt-opteron10.localdomain
     state = free
     np = 2
     ntype = cluster
```

補足事項:ノード, CPU(1/4)

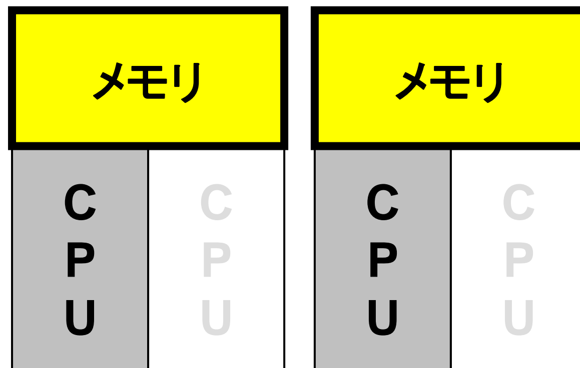
- Opteronの各ノードは2つのCPUから構成されている。
- 1つのメモリユニットを共有している。
- ノード内の使用CPU数は環境変数等で制御が可能

補足事項：ノード，CPU（2/4）

- 例えば 2プロセッサ使用する場合：



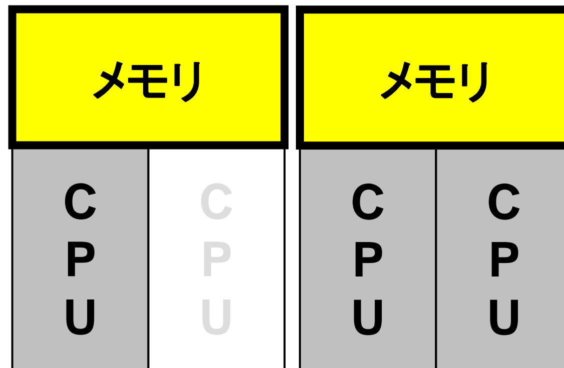

```
#!/bin/sh
#PBS -N test
#PBS -o x1a.lst
#PBS -l nodes=1:ppn=2
#PBS -e err
cd $PBS_O_WORKDIR
NPROCS=`wc -l < $PBS_NODEFILE`
mpirun -v -machinefile $PBS_NODEFILE -np $NPROCS a.out
```



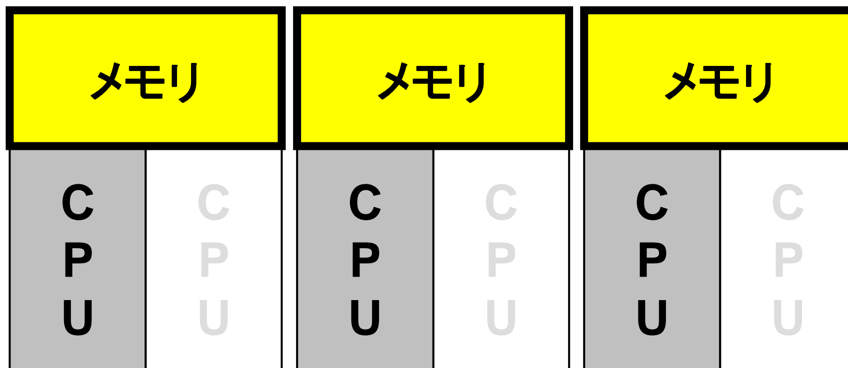

```
#!/bin/sh
#PBS -N test
#PBS -o x1b.lst
#PBS -l nodes=2:ppn=1
#PBS -e err
cd $PBS_O_WORKDIR
NPROCS=`wc -l < $PBS_NODEFILE`
mpirun -v -machinefile $PBS_NODEFILE -np $NPROCS a.out
```

補足事項：ノード，CPU（3/4）

- 例えば 3プロセッサ使用する場合：




```
#!/bin/sh
#PBS -N test
#PBS -o x1a.lst
#PBS -l nodes=2:ppn=2
#PBS -e err
cd $PBS_O_WORKDIR
NPROCS=`wc -l < $PBS_NODEFILE`
mpirun -v -machinefile $PBS_NODEFILE -np 3 a.out
```




```
#!/bin/sh
#PBS -N test
#PBS -o x1b.lst
#PBS -l nodes=3:ppn=1
#PBS -e err
cd $PBS_O_WORKDIR
NPROCS=`wc -l < $PBS_NODEFILE`
mpirun -v -machinefile $PBS_NODEFILE -np $NPROCS a.out
```

補足事項：ノード，CPU（4/4）

- Opteronは各CPUに独立のメモリコントローラがあるため，理想的には，ppn=1，ppn=2では違いはないはずであるが，実際にはオーバーヘッドはある。
  - 使用メモリ量が増加するとこの傾向は顕著
  - 基本的にppn=1とした方が速いのであるが，MPIにおいてもノード内，ノード間の差はあるため，一概にどちらが速いとは言えない。
    - メモリへの負担，通信オーバヘッドのバランスによって決まる
- 以下は課題S1-3（台形積分）でN=10^9とした場合の例

```
nodes=1: ppn=1    19.75390625000000
nodes=2: ppn=1     9.88281250000000
nodes=1: ppn=2     9.94140625000000
```

無効プロセスへの対処（重要）（1/2）

- 以下のような原因でジョブが異常終了した場合，プロセスが完全に落ちずに残ってしまう場合がある:
  - MPI_FINALIZEが入っていない
  - 送信，受信の整合性が取れていない
- マシン動作の不安定の原因になりますので，ときどき注意して，このような「無効プロセス」を killしてください。

無効プロセスへの対処（重要）（2/2）

- 無効プロセスを探すには「vt-opteron01」で：

```
>ps -ef|grep nakajima
nakajima  12581     1  0 Jun07 ?  00:00:00 rsh vt-opteron02.localdomain -l
nakajima  12583     1  0 Jun07 ?  00:00:00 rsh vt-opteron02.localdomain -l
nakajima  12585     1  0 Jun07 ?  00:00:00 rsh vt-opteron03.localdomain -l
nakajima  18372     1  0 01:12 ?  00:00:00 rsh vt-opteron02.localdomain -l
nakajima  18706     1  0 01:14 ?  00:00:00 rsh vt-opteron02.localdomain -l
nakajima  18795     1  0 01:14 ?  00:00:00 rsh vt-opteron02.localdomain -l
nakajima  19545     1  0 01:17 ?  00:00:00 rsh vt-opteron02.localdomain -l
```

- のようにして，見つかった無効ジョブをkillします。上記の例だと，vt-optero02,03にも無効ジョブがある可能性があります。「vt-opteron01」のジョブを消しただけではこれらは連動して消えてくれないので：

```
>rsh vt-opteron02 ps -ef|grep nakajima
```

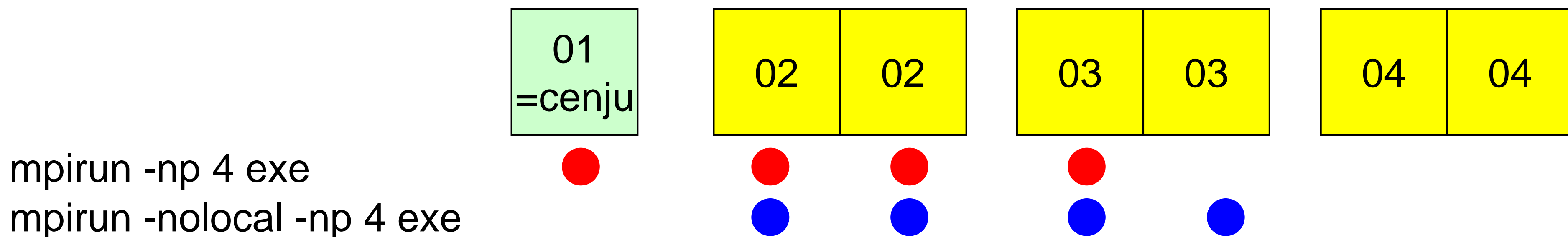
- のようにして，rshを使って同じようなプロセスを繰り返してください。

MPIプログラムのデバッグ方法

- 手近で使えるデバッガーは無い。
- まず，シリアルで完全なプログラムを作ること。
  - 続いてMPIを挿入し，1CPUで計算して，シリアルのときと同じ答えになることを確認すること(`mpirun -np 1 <prog>`)。
- MPI_BARRIERが有効。
- **引数の型，数，順番等に充分気をつけること。**
  - **バグの可能性として最も高い**

フォアグラウンドでのmpirun 使用時

- バッチジョブの場合vt-opteron02~vt-opteron17が使用される。
- フォアグラウンドで端末から「mpirun -np XX exe」とすると
  - 1プロセスは「vt-opteron01（=cenju）」が使われてしまう。
  - 以下プロセス数に従って，vt-opteron02，03・・・が「ppn=2」として使用される。
  - 「vt-opteron01」を使用したくない場合は「-nolocal」のオプションを加える。
- 他のユーザーと競合する可能性があるので，時間計測時にはバッチジョブを使用してください。



- MPIとは
- PC Cluster “CENJU” について

- 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）

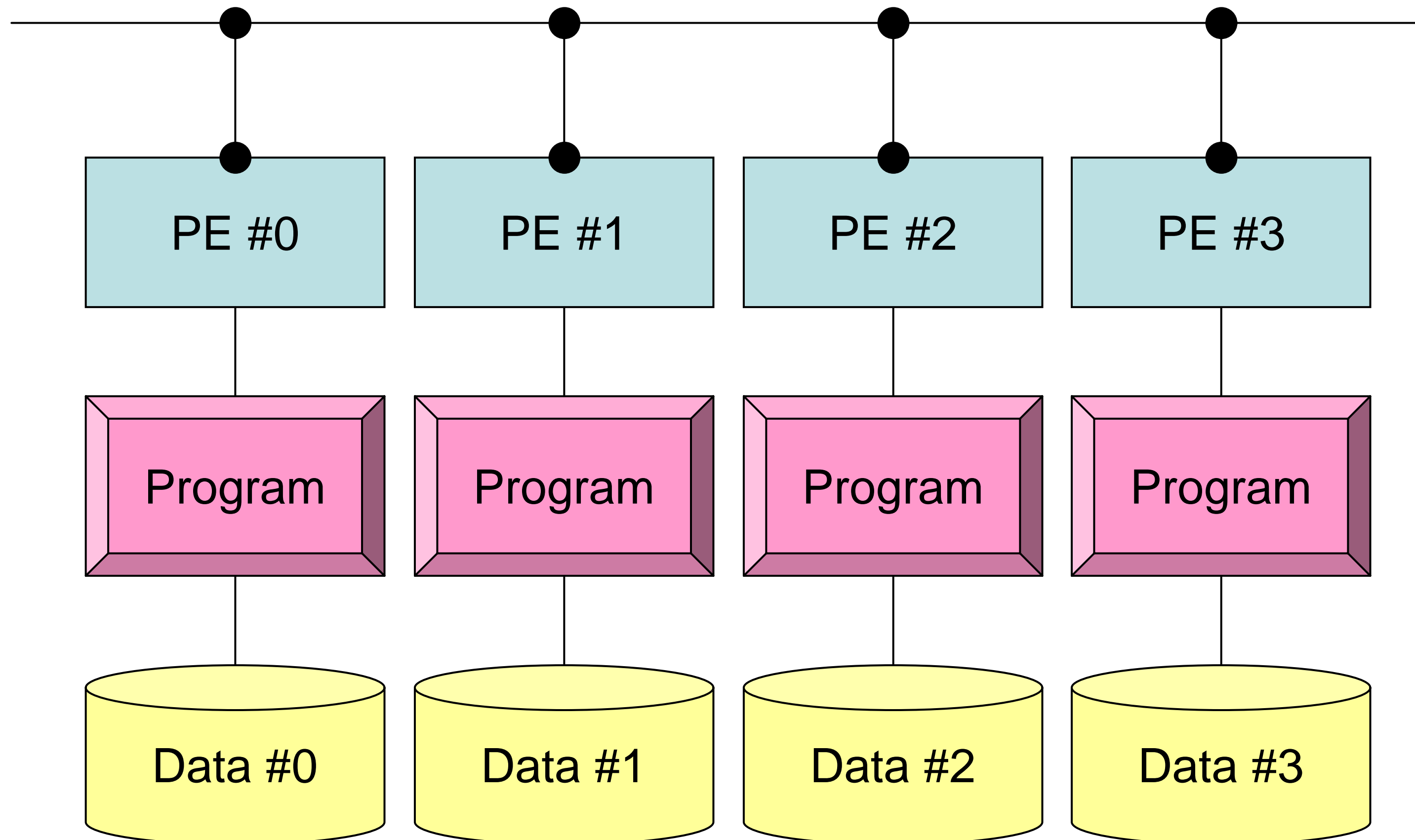
データ構造とアルゴリズム

- 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）

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:

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

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

- これを4つのプロセッサで分担して計算するとすれば，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:

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

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

- このようにすれば「一種類の」プログラム（Single Program）で並列計算を実施できる。
  - 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。

全体データと局所データ

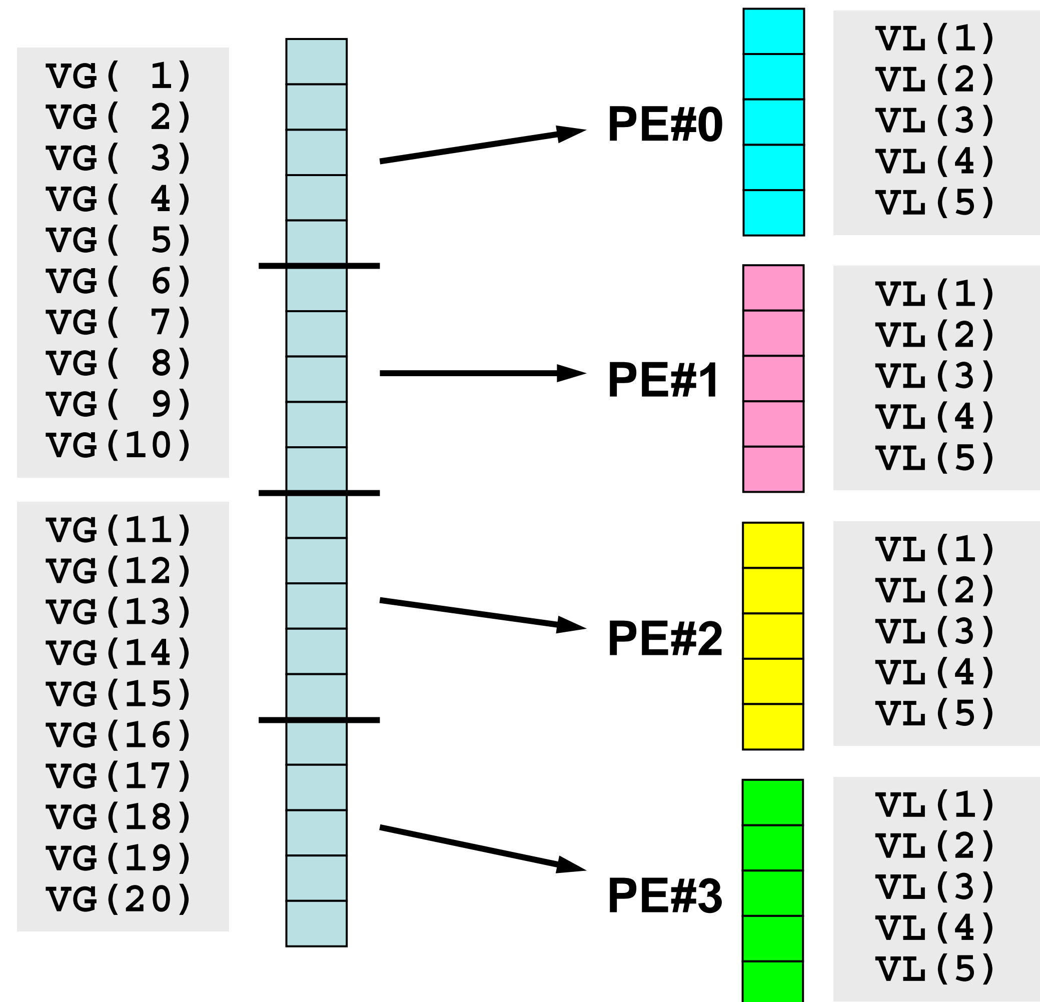
- VG
  - 領域全体
  - 1番から20番までの「全体番号」を持つ「全体データ（Global Data）」
- VL
  - 各プロセス（PE，プロセッサ，領域）
  - 1番から5番までの「局所番号」を持つ「局所データ（Local Data）」
  - できるだけ局所データを有効に利用することで，高い並列性能が得られる。

局所データの考え方

「全体データ」VGの：

- 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，「局所データ」VLの1番～5番成分となる（局所番号が1番～5番となる）。



全体データと局所データ

- VG
  - 領域全体
  - 1番から20番までの「全体番号」を持つ「全体データ（Global Data）」
- VL
  - 各プロセッサ
  - 1番から5番までの「局所番号」を持つ「局所データ（Local Data）」

- この講義で常に注意してほしいこと
  - VG（全体データ）からVL（局所データ）をどのように生成するか。
  - VGからVL，VLからVGへデータの中身をどのようにマッピングするか。
  - VLがプロセスごとに独立して計算できない場合はどうするか。
  - できる限り「局所性」を高めた処理を実施する⇒高い並列性能

- MPIとは
- PC Cluster “CENJU” について

- MPIの基礎：Hello World

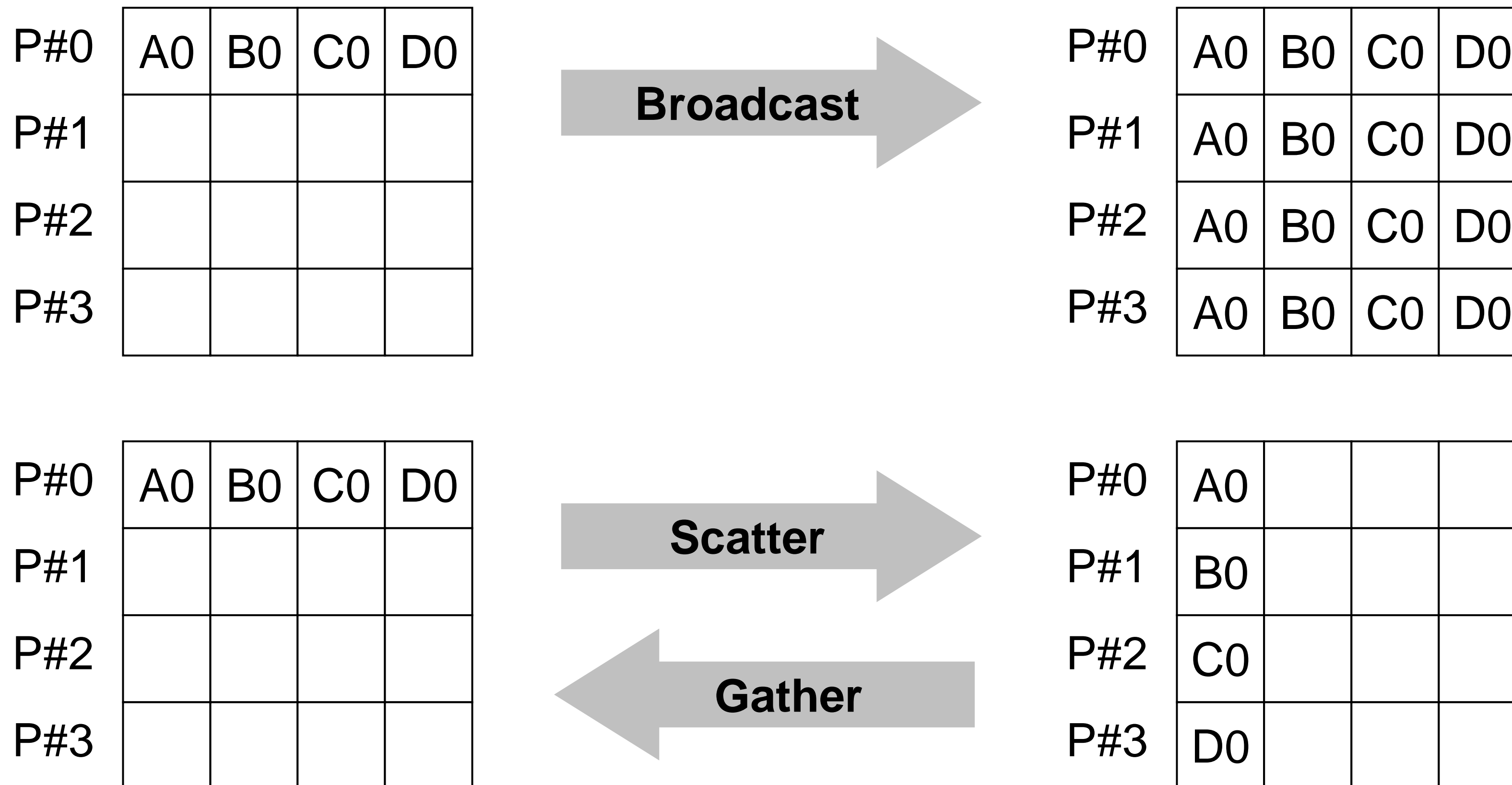
- 全体データと局所データ
- グループ通信（Collective Communication）

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

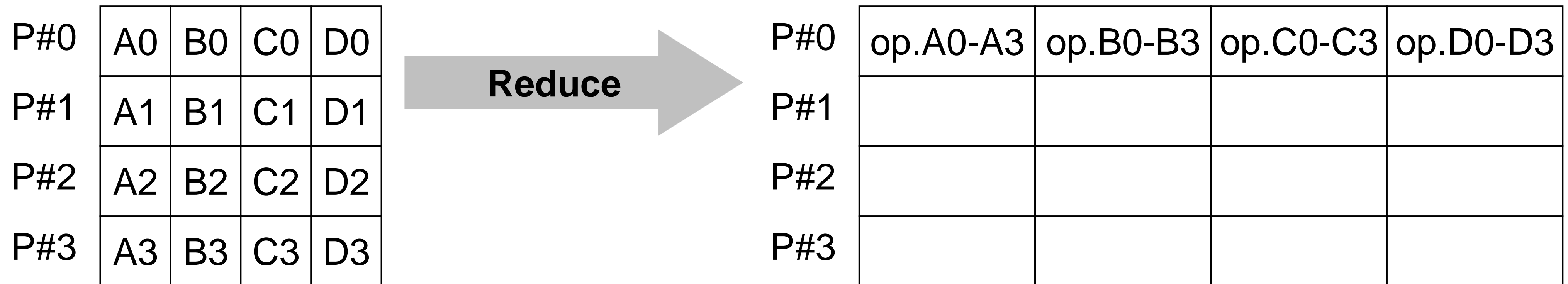
All-to-All

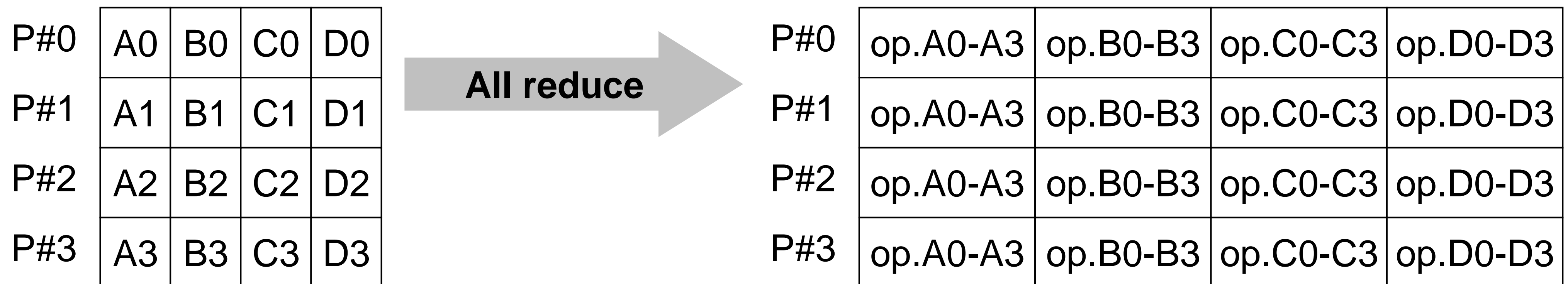
P#0	A0	A1	A2	A3
P#1	B0	B1	B2	B3
P#2	C0	C1	C2	C3
P#3	D0	D1	D2	D3

→

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
- 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。 補足資料
- 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
```

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

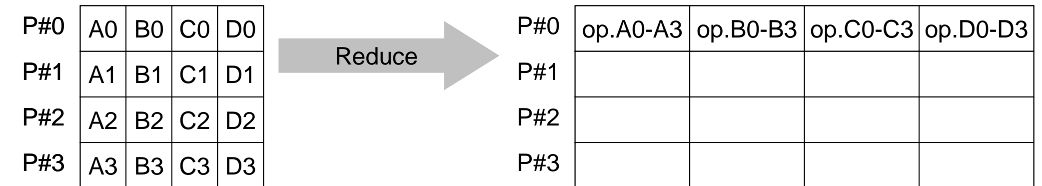
```fortran
implicit REAL*8 (A-H,O-Z)
real(kind=8),dimension(20):: &
     VECp,  VECs

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;
}
```

<$S1>/dot.f, dot.cの実行

```
>$ cd <$S1>

>$ pgf90 -O3 dot.f
or
>$ pgcc -O3 dot.c

>$ ./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

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

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

  - **op** 整数 I 計算の種類

    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
    ユーザーによる定義も可能： MPI_OP_CREATE

  - **root** 整数 I 受信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

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）

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

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

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

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

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

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

MPI_REDUCEの例（2/2）

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

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

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

各プロセスにおける，X0の総和が0番PEのXSUMに入る。

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

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

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

MPI_BCAST

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

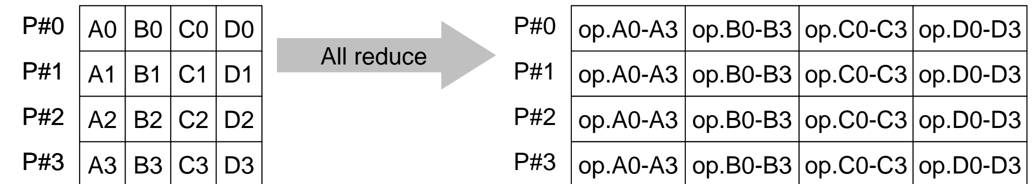
Broadcast →

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

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

- `call MPI_BCAST (buffer,count,datatype,root,comm,ierr)`
  - **buffer** 任意 I/O バッファの先頭アドレス，タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
    FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
  - **root** 整数 I 送信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

MPI_ALLREDUCE

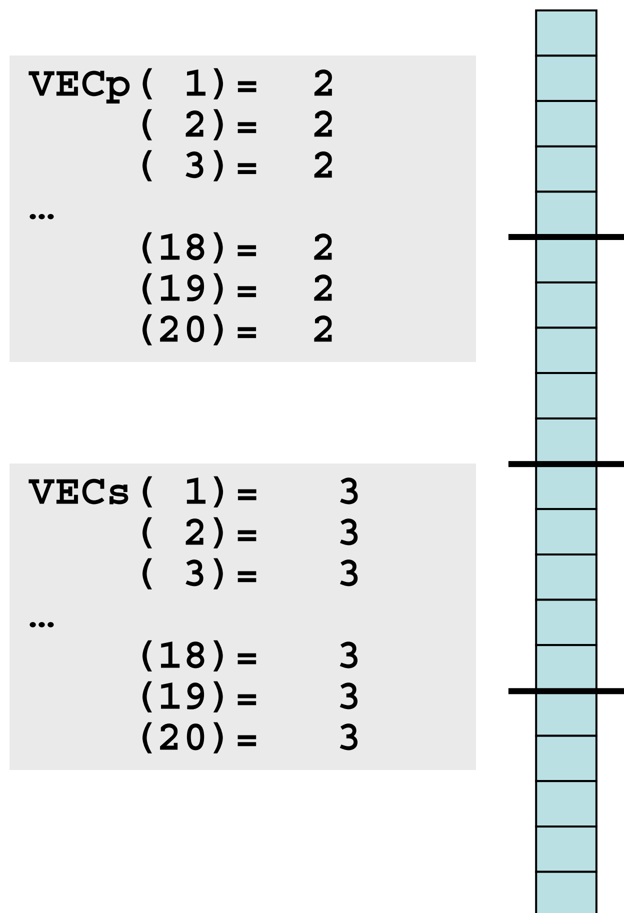


- MPI_REDUCE + MPI_BCAST
- 総和，最大値を計算したら，各プロセスで利用したい場合が多い

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

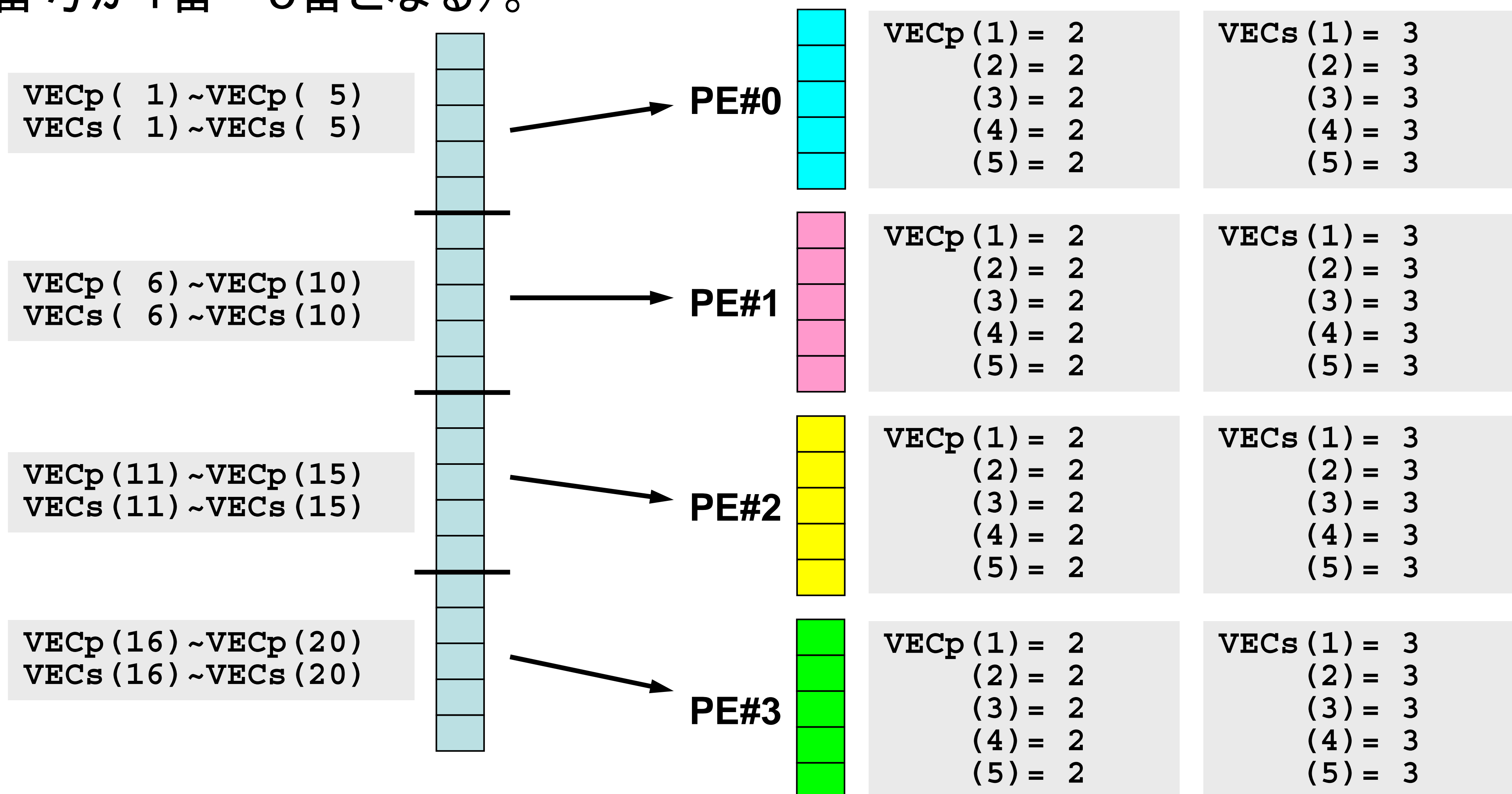
局所データの考え方（1/2）

- 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）



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)。



内積の並列計算例(1/3)

<$S1>/allreduce.f

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

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

sumA= 0.d0
sumR= 0.d0
do i= 1, 5
  VECp(i)= 2.d0
  VECs(i)= 3.d0
enddo

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

if (my_rank.eq.0) then
  write (*,'(a)') '(my_rank, sumALLREDUCE, sumREDUCE)'
endif
```

各ベクトルを各プロセスで独立に生成する

内積の並列計算例（2/3）

<$S1>/allreduce.f

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

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

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

内積の計算
各プロセスで計算した結果「sum0」の総和をとる
sumR には，PE#0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。

内積の並列計算例(3/3)

<$S1>/allreduce.f

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

  call MPI_FINALIZE (ierr)

  stop
  end
```

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

<$S1>/allreduce.f の実行例

```
$> mpif90 -O3 allreduce.f
$> mpirun -np 4 a.out

(my_rank, sumALLREDUCE,sumREDUCE)
before BCAST    0    1.200000E+02    1.200000E+02
after  BCAST    0    1.200000E+02    1.200000E+02
FORTRAN STOP
before BCAST    1    1.200000E+02    0.000000E+00
after  BCAST    1    1.200000E+02    1.200000E+02
FORTRAN STOP
before BCAST    3    1.200000E+02    0.000000E+00
after  BCAST    3    1.200000E+02    1.200000E+02
FORTRAN STOP
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
- 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（<$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
```

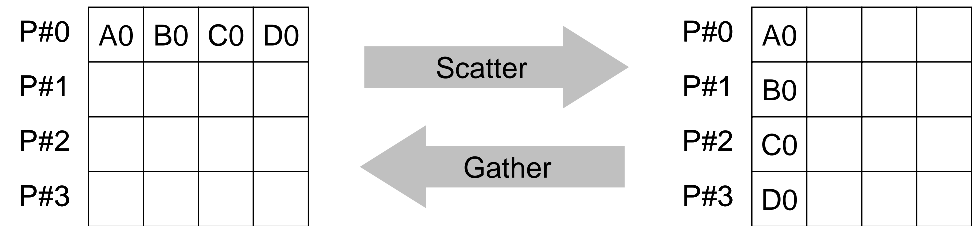
```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

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

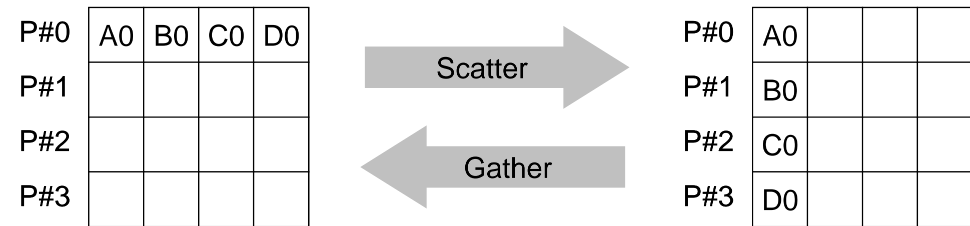
MPI_SCATTER



- 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。

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

MPI_SCATTER (続き)



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

Scatter/Gatherの計算 (3/8)

4つのプロセスへ均等に（長さ8ずつ）割り振る。

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

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

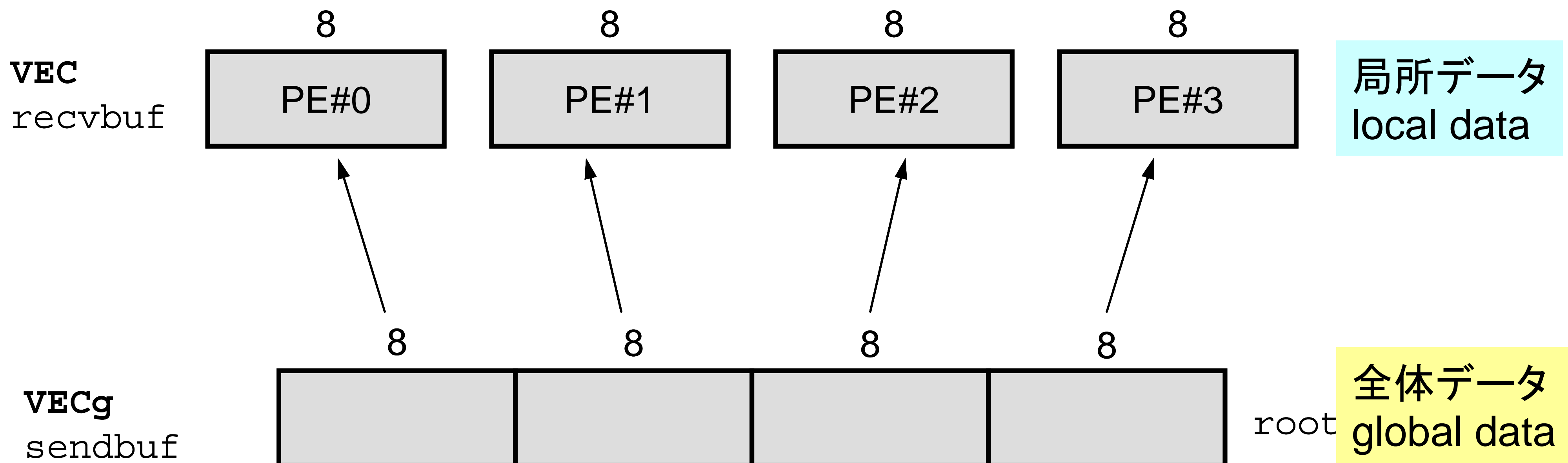
```
int  N=8;
double VEC [8];
...
MPI_Scatter (&VECg, N, MPI_DOUBLE, &VEC, N,
MPI_DOUBLE, 0, <comm>);
```

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

Scatter/Gatherの計算（4/8）

4つのプロセスへ均等に（長さ8ずつ）割り振る。

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



Scatter/Gatherの計算 (5/8)

4つのプロセスへ均等に(長さ8ずつ)割り振る。

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

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

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

Scatter/Gatherの計算（5/8）

4つのプロセスへ均等に（長さ8ずつ）割り振る。

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

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

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

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

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

Scatter/Gatherの計算 (6/8)

各プロセスでベクトル（長さ8）の各成分にALPHAを加える

- 各プロセスでの計算は，以下のようになる：

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

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

- 計算結果は以下のようになる：

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

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

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

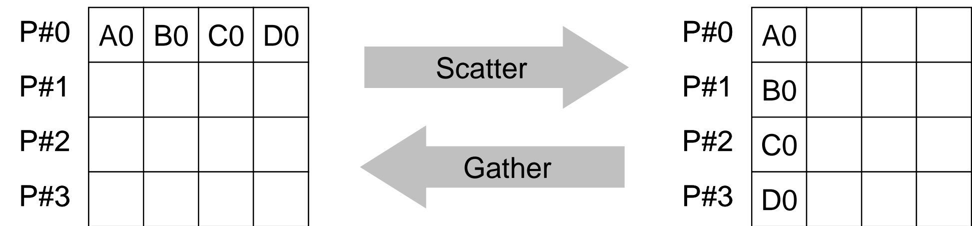
```
PE#3
after 3 1  1401.
after 3 2  1403.
after 3 3  1405.
after 3 4  1406.
after 3 5  1409.
after 3 6  1411.
after 3 7  1421.
after 3 8  1451.
```

Scatter/Gatherの計算（7/8）

各プロセスの結果を再び長さ32のベクトルにまとめる

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

MPI_GATHER



- MPI_SCATTERの逆

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

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

Scatter/Gatherの計算（8/8）

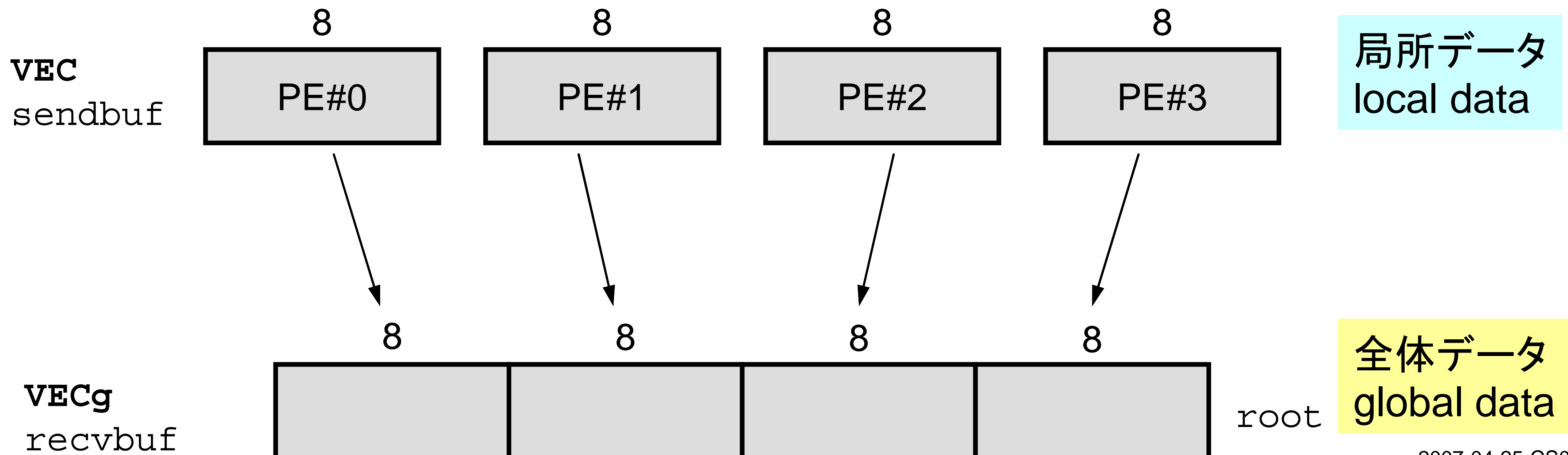
各プロセスの結果を再び長さ32のベクトルにまとめる

- 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



<$S1>/scatter-gather.f の実行例

```
$> mpif90 -O3 scatter-gather.f
$> mpirun -np 4 a.out
```

```
$> mpicc -O3 scatter-gather.c
$> mpirun -np 4 a.out
```

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

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

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

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

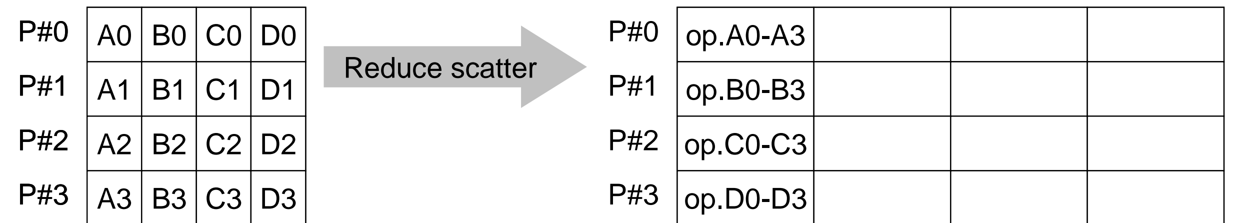
```
PE#0
after 0 1   1101.
after 0 2   1103.
after 0 3   1105.
after 0 4   1106.
after 0 5   1109.
after 0 6   1111.
after 0 7   1121.
after 0 8   1151.
```

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

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

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

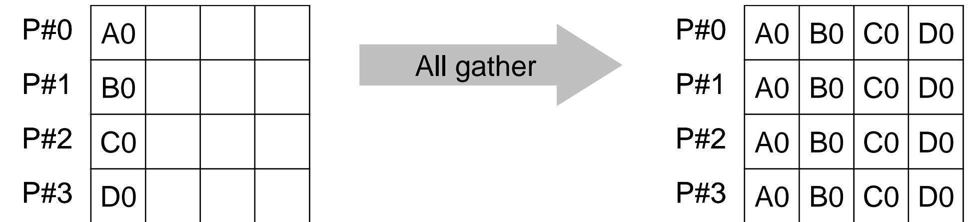
MPI_REDUCE_SCATTER



- MPI_REDUCE + MPI_SCATTER

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

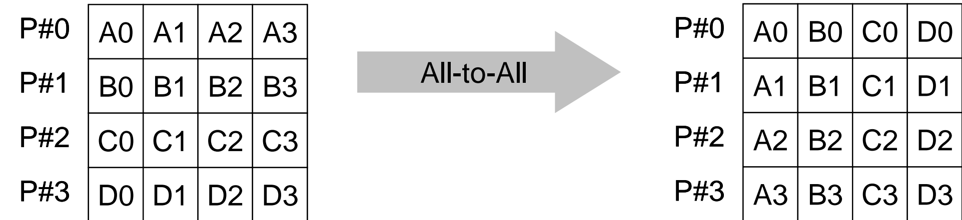
MPI_ALLGATHER



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

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

MPI_ALLTOALL



- MPI_ALLGATHERの更なる拡張：転置

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

グループ通信による計算例

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

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）

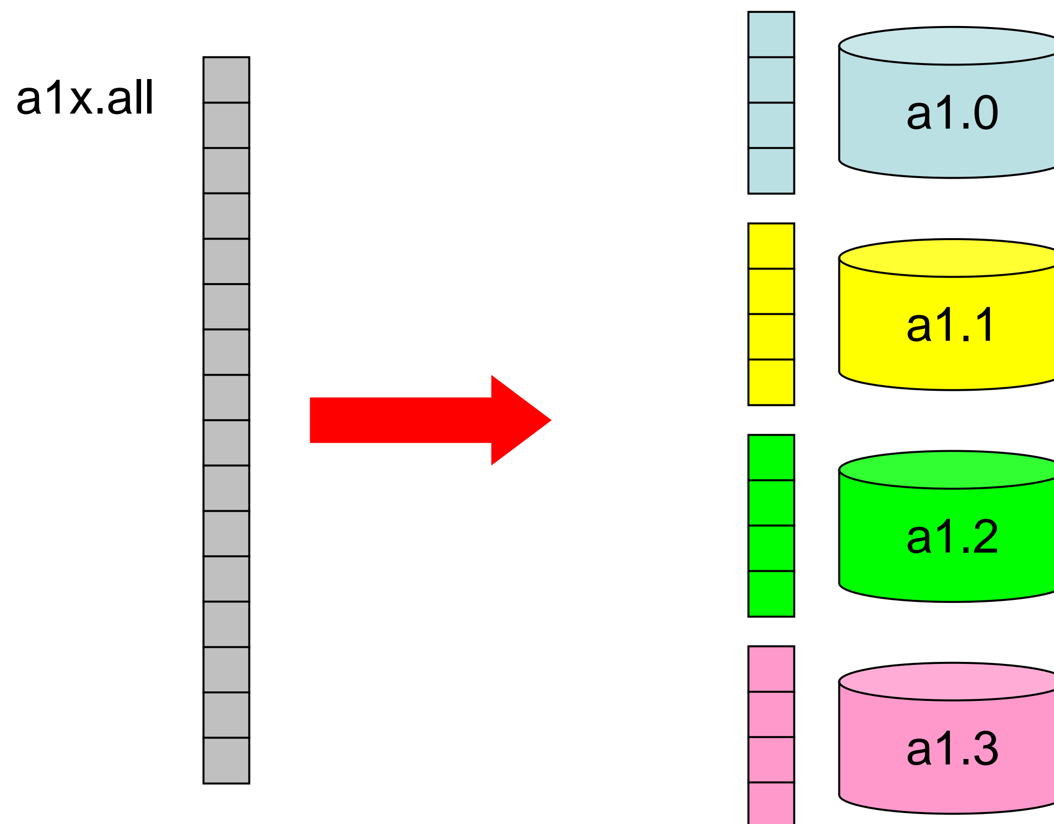
```
>$ cd <$S1>
>$ ls a1.*
   a1.0 a1.1 a1.2 a1.3
```

「a1x.all」を4つに分割したもの

```
>$ mpif90 -O3 file.f
or
>$ mpicc -O3 file.c

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

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)

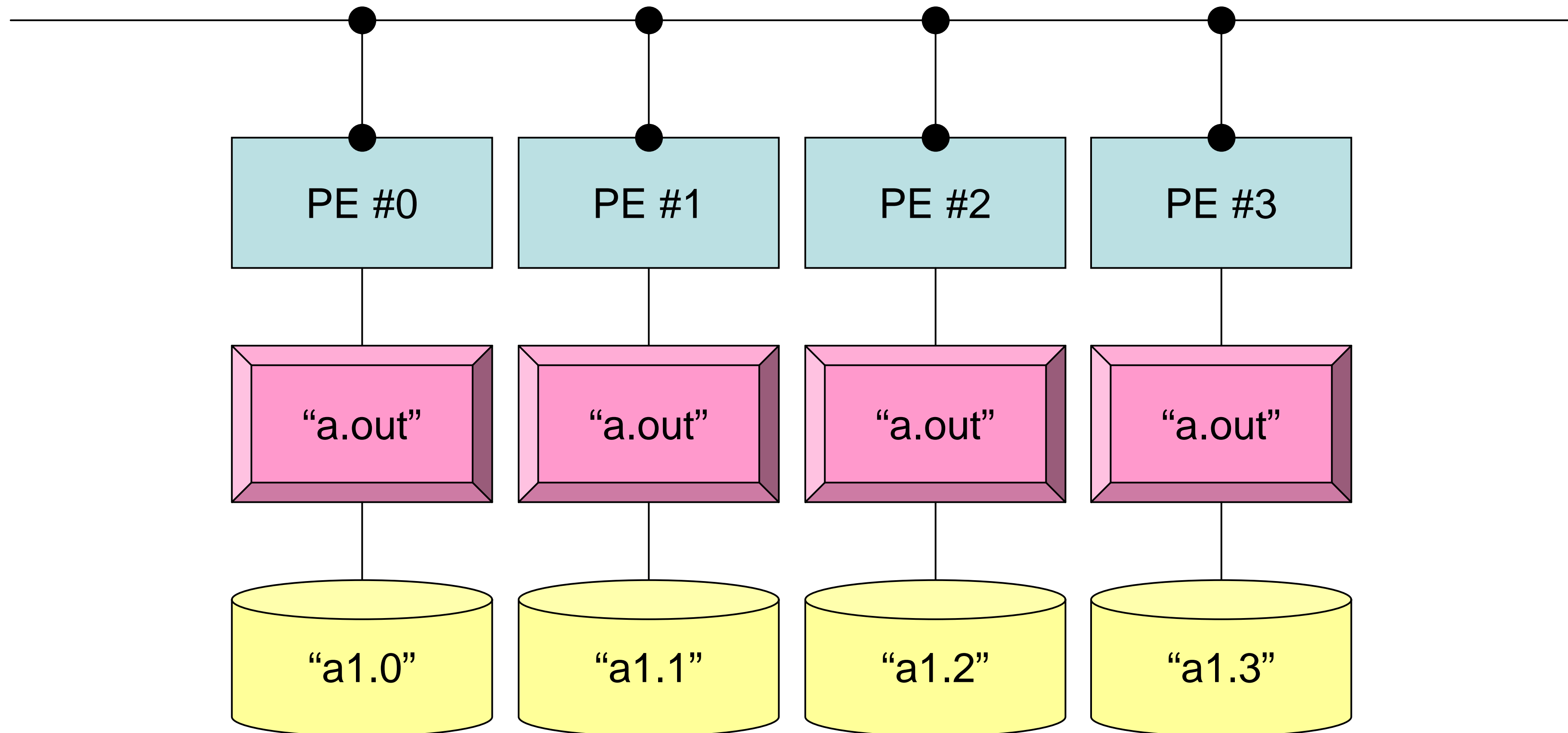
<$S1>/file.f

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

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

Hello とそんなに変わらない

「局所番号(1~8)」で読み込む

SPMDの典型例




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

分散ファイル読み込み：可変長(1/2)

```
>$ cd <$S1>
>$ 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
>$ mpif90 -O3 file2.f
or
>$ mpicc -O3 file2.c

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

分散ファイルの読み込み：可変長(2/2)

<$S1>/file2.f

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

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

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

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

call MPI_FINALIZE (ierr)
stop
end
```

Nが各データ(プロセッサ)で異なる

局所データの作成法

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

次回までの宿題

- 本講義資料の復習
  - 「全体データ(番号)」と「局所データ(番号)」の考え方の理解
- <$S1>内のプログラムの理解

```
hello.f
time.f

dot.f
allreduce.f

scatter-gather.f

file.f
file2.f
```

```
hello.c
time.c

dot.c
allreduce.c

scatter-gather.c

file.c
file2.c
```