

MPIによるプログラミング概要

課題S1・S2出題

Fortran編

2012年夏季集中講義
中島研吾

並列計算プログラミング(616-2057)・先端計算機演習(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。
 - +複雑
 - 理想 : Scalable
 - N倍の規模の計算をN倍のCPUを使って、「同じ時間で」解く

概要

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

MPIとは (1/2)

- Message Passing Interface
- 分散メモリ間のメッセージ通信APIの「規格」
 - プログラム, ライブラリ, そのものではない
 - <http://phase.hpcc.jp/phase/mpi-j/ml/mpi-j-html/contents.html>
- 歴史
 - 1992 MPIフォーラム
 - 1994 MPI-1規格
 - 1997 MPI-2規格(拡張版), 現在はMPI-3が検討されている
- 実装
 - mpich アルゴンヌ国立研究所
 - LAM
 - 各ベンダー
 - C/C++, FOTRAN, Java ; Unix, Linux, Windows, Mac OS

MPIとは (2/2)

- 現状では, mpich(フリー)が広く使用されている。
 - 部分的に「MPI-2」規格をサポート
 - 2005年11月から「MPICH2」に移行
 - <http://www-unix.mcs.anl.gov/mpi/>
- MPIが普及した理由
 - MPIフォーラムによる規格統一
 - どんな計算機でも動く
 - FORTRAN, Cからサブルーチンとして呼び出すことが可能
 - mpichの存在
 - フリー, あらゆるアーキテクチャをサポート
- 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に

- SPMD/SIMDのオペレーションに慣れること...「つかむ」こと

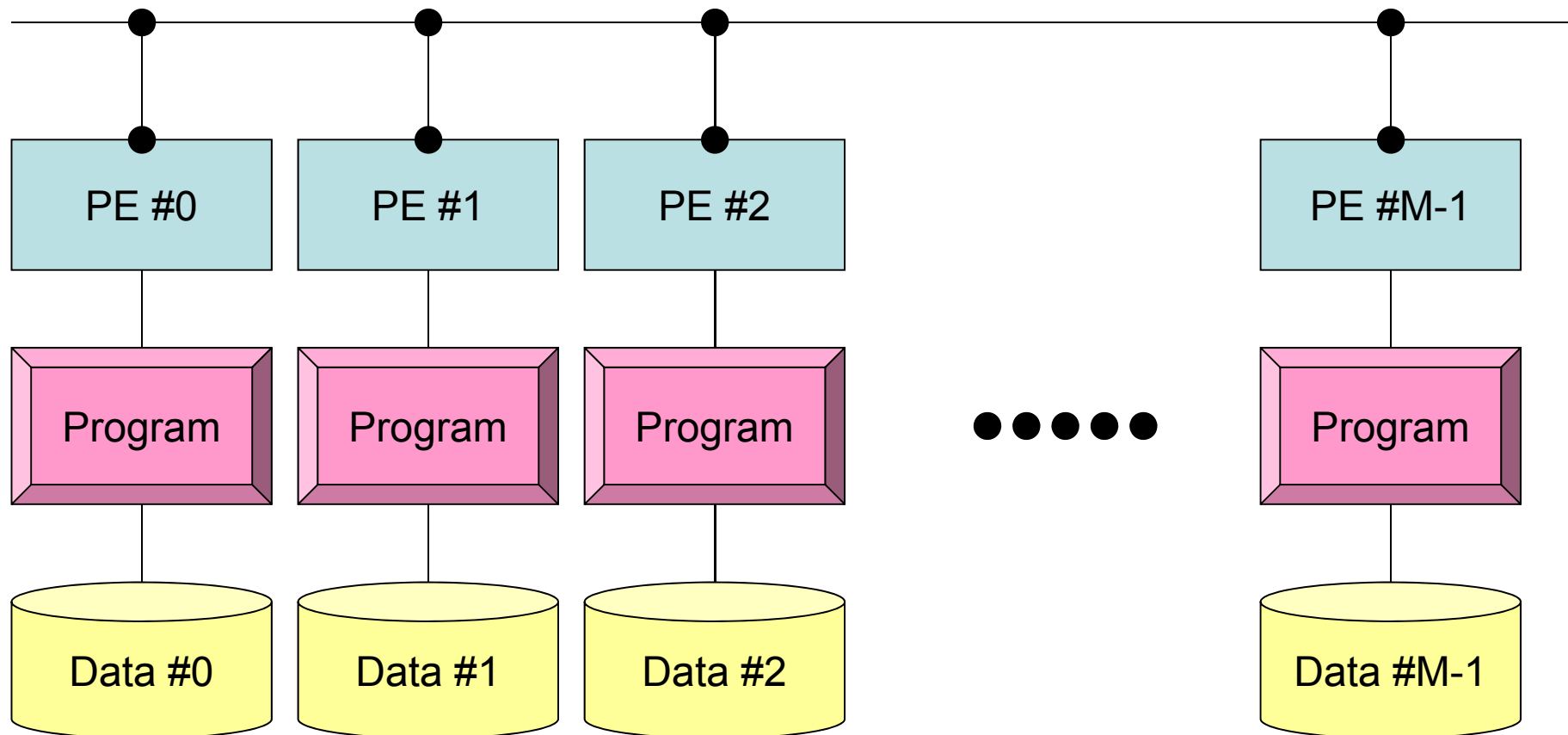
- Single Program/Instruction Multiple Data
- 基本的に各プロセスは「同じことをやる」が「データが違う」
 - 大規模なデータを分割し, 各部分について各プロセス(プロセッサ)が計算する
- 全体データと局所データ, 全体番号と局所番号

PE: Processing Element
プロセッサ, 領域, プロセス

SPMD

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

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



各プロセスは「同じことをやる」が「データが違う」
大規模なデータを分割し、各部分について各プロセス(プロセッサ)が計算する
通信以外は、単体CPUのときと同じ、というのが理想

用語

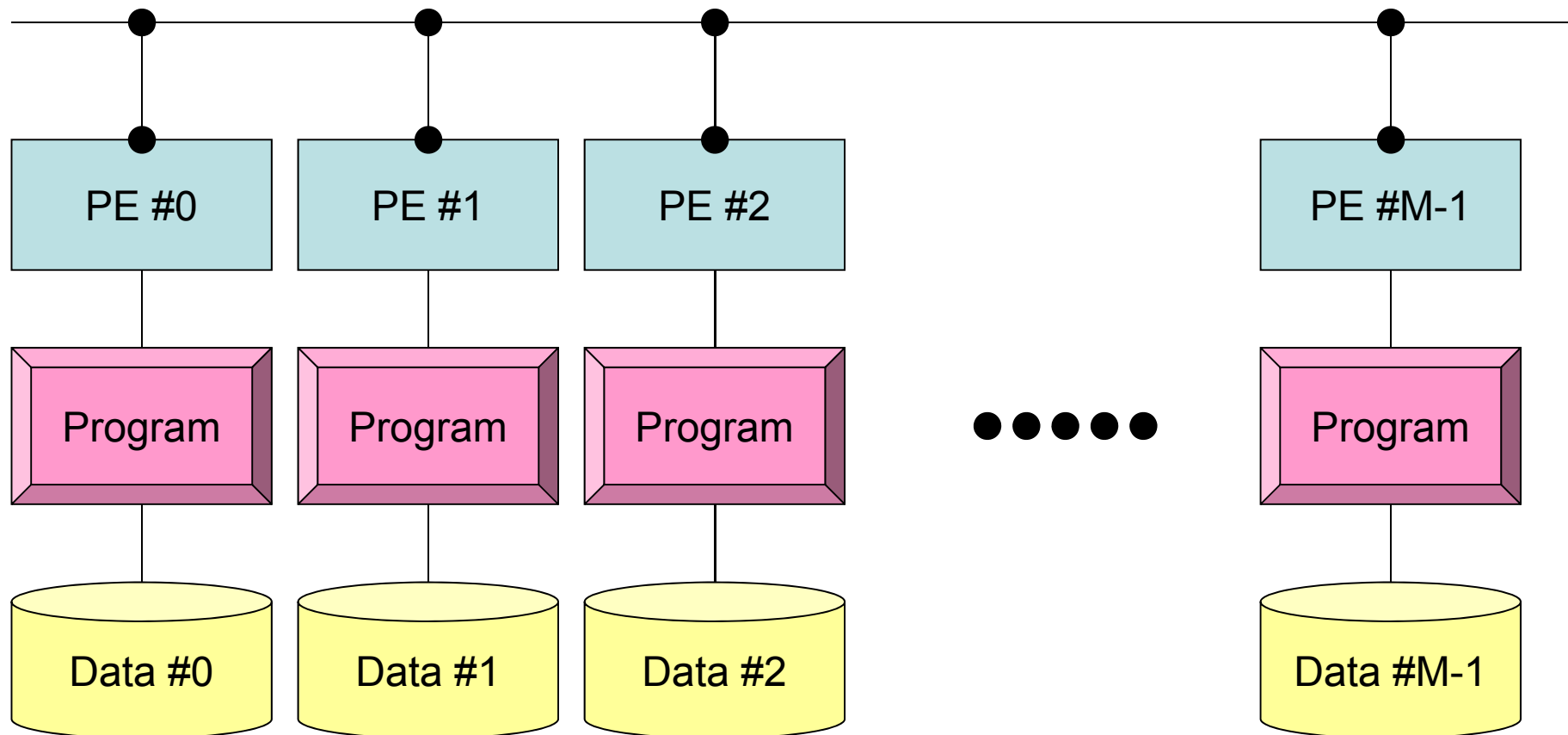
- プロセッサ, コア
 - ハードウェアとしての各演算装置。シングルコアではプロセッサ＝コア
- プロセス
 - MPI計算のための実行単位, ハードウェア的な「コア」とほぼ同義。
 - しかし1つの「プロセッサ・コア」で複数の「プロセス」を起動する場合もある(効率的ではないが)。
- PE (Processing Element)
 - 本来, 「プロセッサ」の意味なのであるが, 本講義では「プロセス」の意味で使う場合も多い。次項の「領域」とほぼ同義でも使用。
 - マルチコアの場合は: 「コア＝PE」という意味で使うことが多い。
- 領域
 - 「プロセス」とほぼ同じ意味であるが, SPMD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

PE: Processing Element
プロセッサ, 領域, プロセス

SPMD

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

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



各プロセスは「同じことをやる」が「データが違う」
大規模なデータを分割し、各部分について各プロセス(プロセッサ)が計算する
通信以外は、単体CPUのときと同じ、というのが理想

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
- 90分 × 5コマ
 - 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
- MPIの基礎: Hello World
- 全体データと局所データ
- グループ通信 (Collective Communication)
- 1対1通信 (Peer-to-Peer Communication)

ログイン, ディレクトリ作成 on Oakleaf-FX

```
ssh t61**@oakleaf-fx.cc.u-tokyo.ac.jp
```

ディレクトリ作成

```
>$ cd
```

```
>$ mkdir 2012summer (好きな名前でもいい)
```

```
>$ cd 2012summer
```

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

Oakleaf-FX

ECCS2012

ファイルコピー-on Oakleaf-FX

FORTRANユーザー

```
>$ cd <$0-TOP>  
>$ cp /home/z30088/class_eps/F/s1-f.tar .  
>$ tar xvf s1-f.tar
```

Cユーザー

```
>$ cd <$0-TOP>  
>$ cp /home/z30088/class_eps/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 <\$0-S1> と呼ぶ。

<\$0-S1> = <\$0-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 <$0-S1>  
>$ mpifrtpx -Kfast hello.f  
>$ mpifccpx -Kfast hello.c
```

FORTRAN

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

“mpifrtpx”:

FORTRAN90+MPIによってプログラムをコンパイルする際に
必要な, コンパイラ, ライブラリ等がバインドされている

C言語

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

“mpifccpx”:

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

ジョブ実行

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

ジョブスクリプト

- `<$0-S1>/hello.sh`
- スケジューラへの指令 + シェルスクリプト

<code>#!/bin/sh</code>	
<code>#PJM -L "node=1"</code>	ノード数
<code>#PJM -L "elapsed=00:10:00"</code>	実行時間
<code>#PJM -L "rscgrp=lecture"</code>	実行キュー名
<code>#PJM -g "gt61"</code>	グループ名（俗称：財布）
<code>#PJM -j</code>	
<code>#PJM -o "hello.lst"</code>	標準出力ファイル名
<code>#PJM --mpi "proc=4"</code>	MPIプロセス数
 <code>mpirun ./a.out</code>	 実行ファイル名

8プロセス
"node=1"
"proc=8"

16プロセス
"node=1"
"proc=16"

32プロセス
"node=2"
"proc=32"

64プロセス
"node=4"
"proc=64"

192プロセス
"node=12"
"proc=192"

ジョブ投入

```
>$ cd <$0-S1>
>$ pjsub hello.sh

>$ cat hello.lst

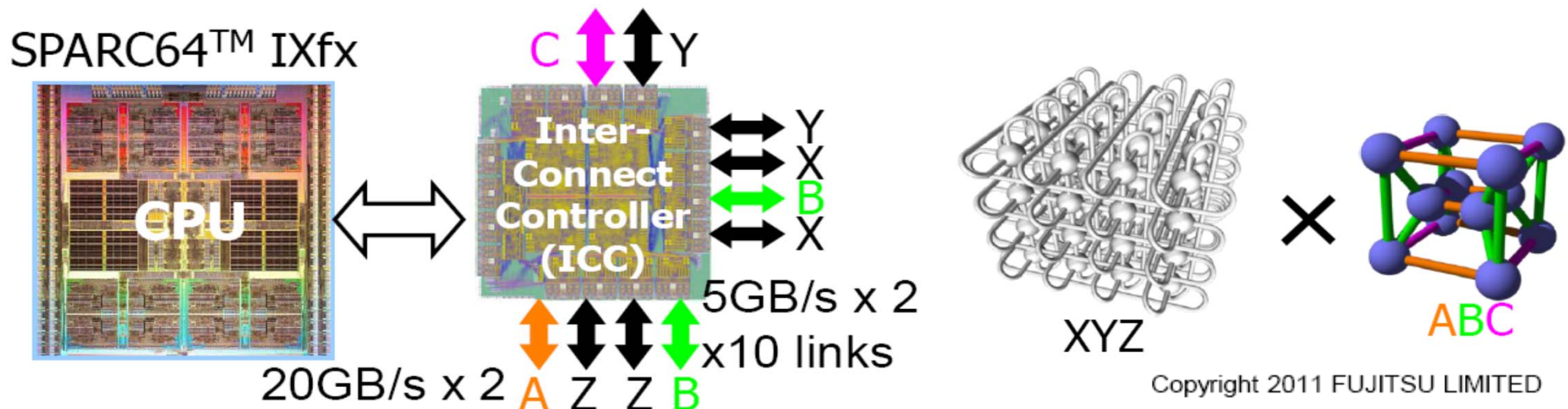
Hello World 0
Hello World 3
Hello World 2
Hello World 1
```

利用可能なキュー

- 以下の2種類のキューを利用可能
- 1 Tofu(12ノード)を使える
 - **lecture**
 - 12ノード(192コア), 15分, アカウント有効期間中利用可能(～10月末)
 - 全教育ユーザーで共有
 - **lecture1**
 - 12ノード(192コア), 15分, 講義・演習実施時間帯
 - **lecture**よりは多くのジョブを投入可能(混み具合による)

Tofuインターコネクト

- ノードグループ
 - 12ノード
 - A軸・C軸: システムボード内4ノード結合, B軸: 3ボード結合
- 6D: (X,Y,Z,A,B,C)
 - ABC 3D Mesh: ノードグループの12ノードを結合: $2 \times 2 \times 3$
 - XYZ 3D Mesh: ABC 3D Meshグループを結合: $10 \times 5 \times 8$
- ネットワークトポロジーを指定したJob Submission可能
 - 実行されたXYZは知ることができる



ジョブ投入, 確認等

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

```
[z30088@oakleaf-fx-6 S2-ref]$ pjstat
```

```
Oakleaf-FX scheduled stop time: 2012/09/28(Fri) 09:00:00 (Remain: 31days 20:01:46)
```

JOB_ID	JOB_NAME	STATUS	PROJECT	RSCGROUP	START_DATE	ELAPSE	TOKEN	NODE:COORD
334730	go. sh	RUNNING	gt61	lecture	08/27 12:58:08	00:00:05	0.0	1

環境管理ルーチン＋必須項目

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

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

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

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

'mpif.h', "mpi.h"

環境変数デフォルト値

FORTRAN90ではuse mpi可

MPI_Init

初期化

MPI_Comm_size

プロセス数取得

mpirun -np XX <prog>

MPI_Comm_rank

プロセスID取得

自分のプロセス番号(0から開始)

MPI_Finalize

MPIプロセス終了

FORTRAN/Cの違い

- 基本的にインタフェースはほとんど同じ
 - Cの場合, 「**MPI_Comm_size**」のように「MPI」は大文字, 「MPI_」のあとの最初の文字は大文字, 以下小文字
- FORTRANはエラーコード(ierr)の戻り値を引数の最後に指定する必要がある。
- 最初に呼ぶ「MPI_INIT」だけは違う
 - `call MPI_INIT (ierr)`
 - `MPI_Init (int *argc, char ***argv)`

何をやっているのか？

```
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
```

#!/bin/sh	
#PJM -L "node=1"	ノード数
#PJM -L "elapsed=00:10:00"	実行時間
#PJM -L "rscgrp=lecture"	実行キュー名
#PJM -g "gt61"	グループ名 (俗称: 財布)
#PJM -j	
#PJM -o "hello.lst"	標準出力ファイル名
#PJM --mpi "proc=4"	MPIプロセス数
mpiexec ./a.out	実行ファイル名

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

mpi.h, mpif.h

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

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

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

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

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

MPI_INIT

- MPIを起動する。他のMPIサブルーチンより前にコールする必要がある(必須)
- 全実行文の前に置くことを勧める。
- **call MPI_INIT (ierr)**
 - ierr 整数 0 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）。
- 全実行文の後に置くことを勧める
- **これを忘れると大変なことになる。**
 - 終わったはずなのに終わっていない……
- **call MPI_FINALIZE (ierr)**
 - **ierr** 整数 0 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「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)**
 - comm 整数 I コミュニケータを指定する
 - size 整数 0 comm.で指定されたグループ内に含まれるプロセス数の合計
 - ierr 整数 0 完了コード

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

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

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

call MPI_FINALIZE (ierr)

stop
end
```

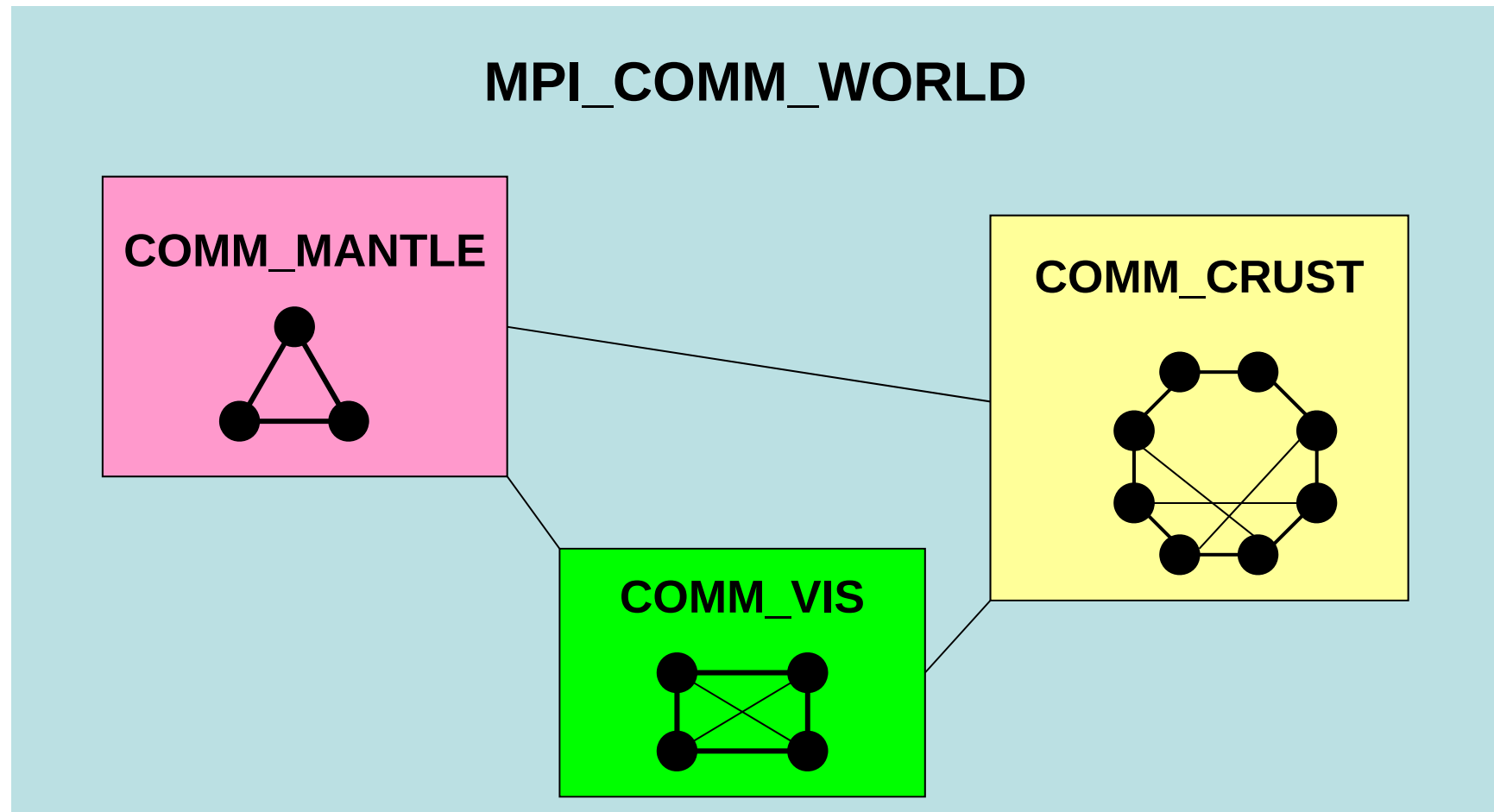
コミュニケータとは？

MPI_Comm_Size (MPI_COMM_WORLD, PETOT)

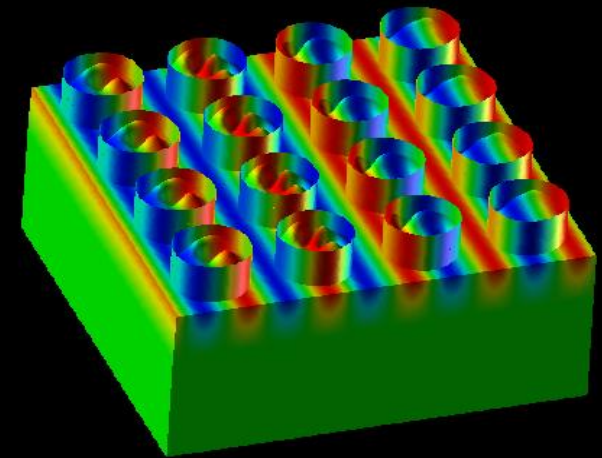
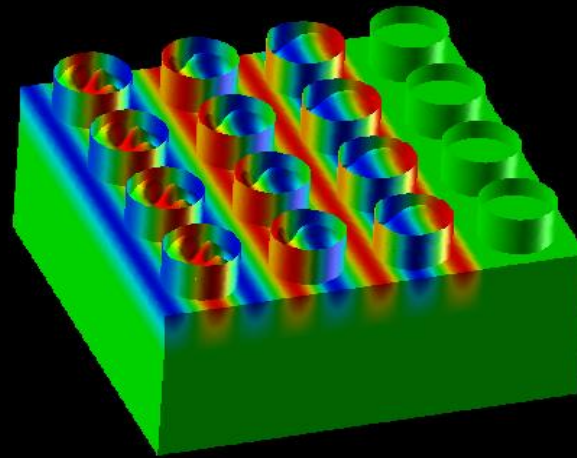
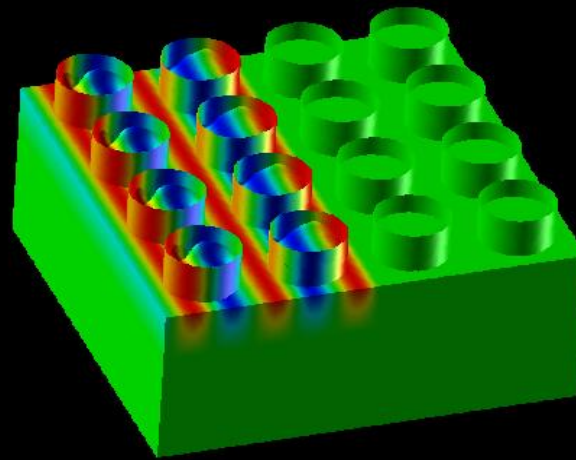
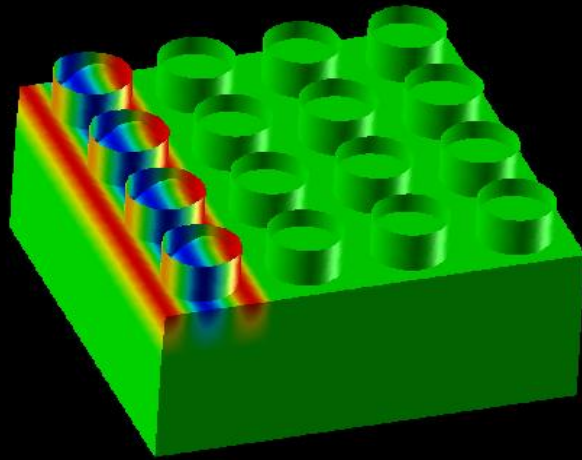
- 通信を実施するためのプロセスのグループを示す。
- MPIにおいて、通信を実施する単位として必ず指定する必要がある。
- mpirunで起動した全プロセスは、デフォルトで「**MPI_COMM_WORLD**」というコミュニケータで表されるグループに属する。
- 複数のコミュニケータを使用し、異なったプロセス数を割り当てることによって、複雑な処理を実施することも可能。
 - 例えば計算用グループ、可視化用グループ
- この授業では「**MPI_COMM_WORLD**」のみでOK。

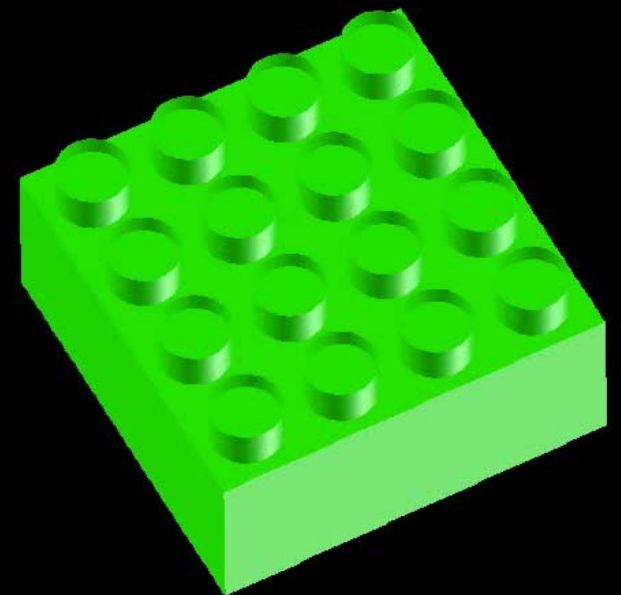
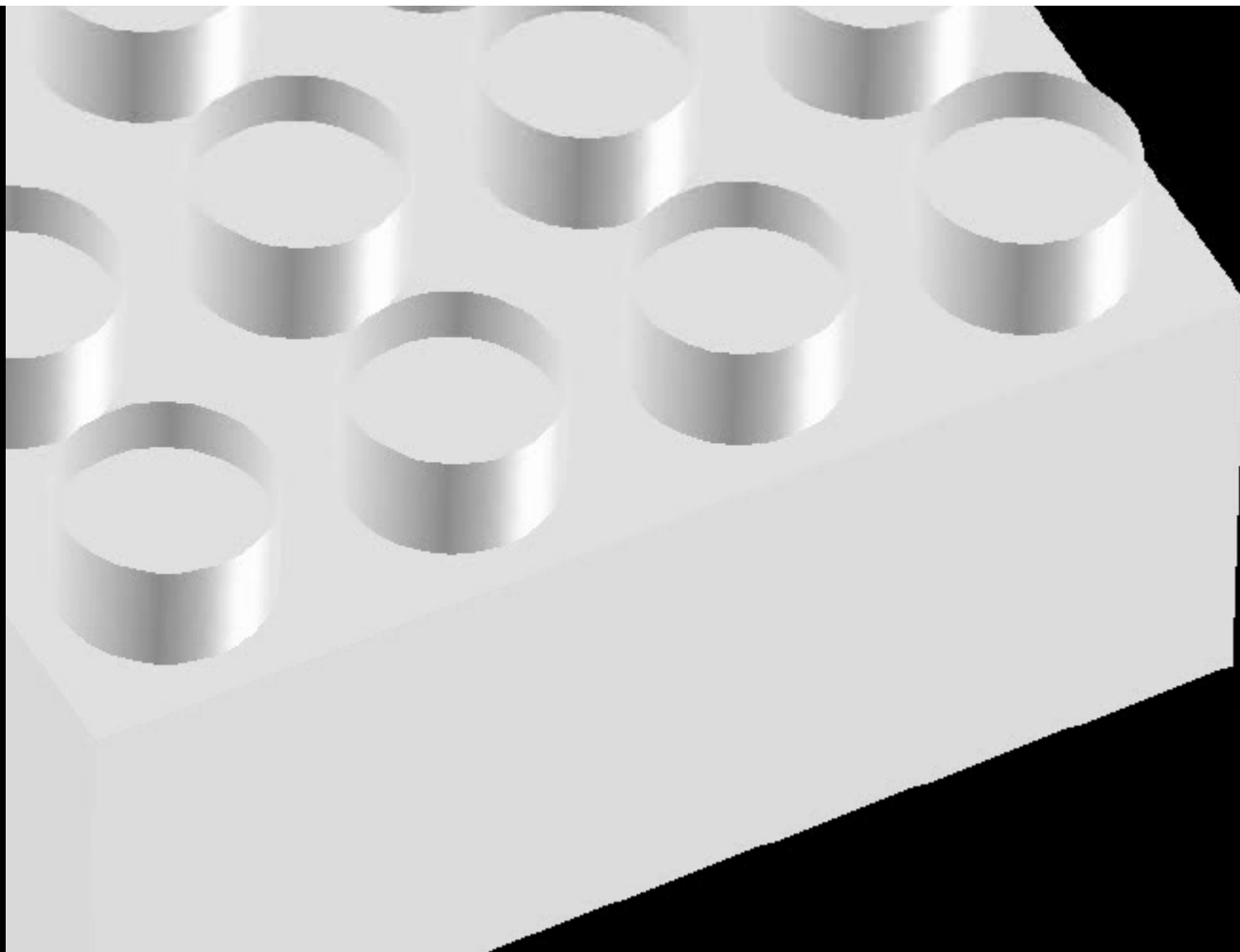
コミュニケータの概念

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



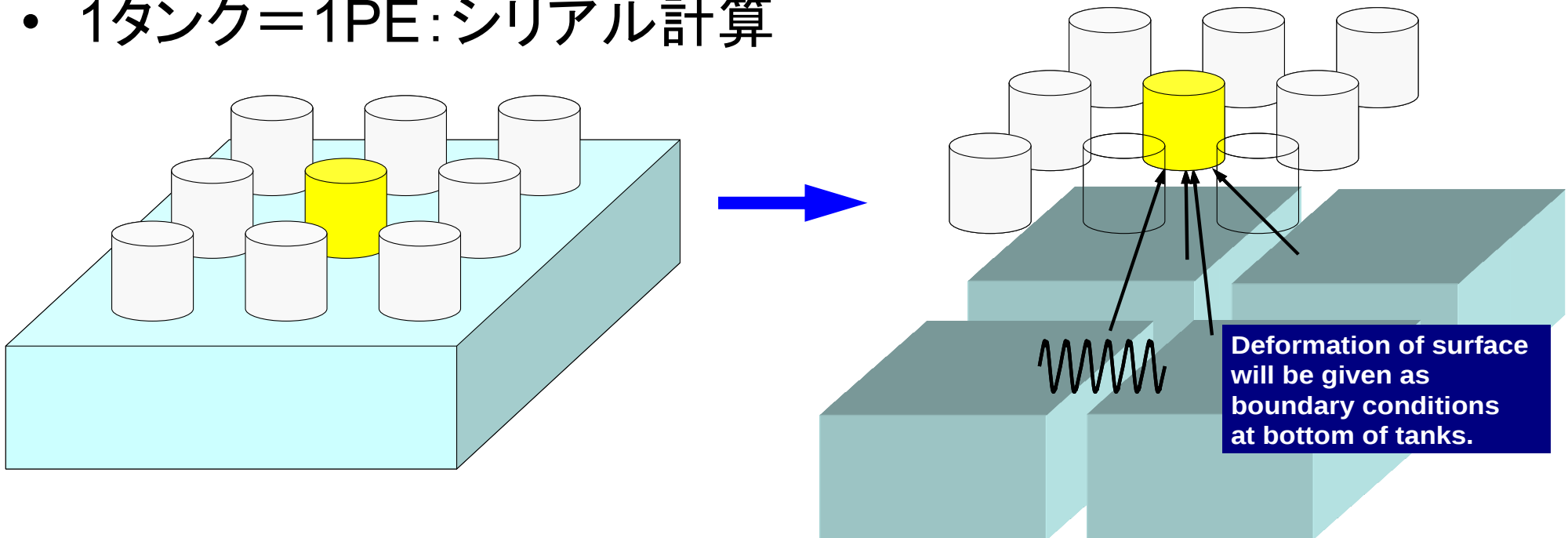
地盤・石油タンク連成シミュレーション





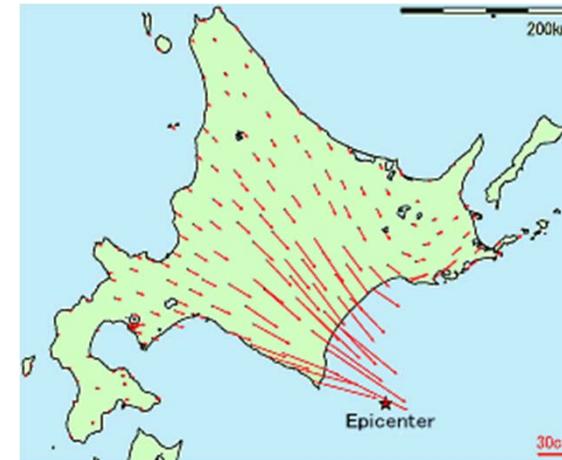
対象とするアプリケーション

- 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

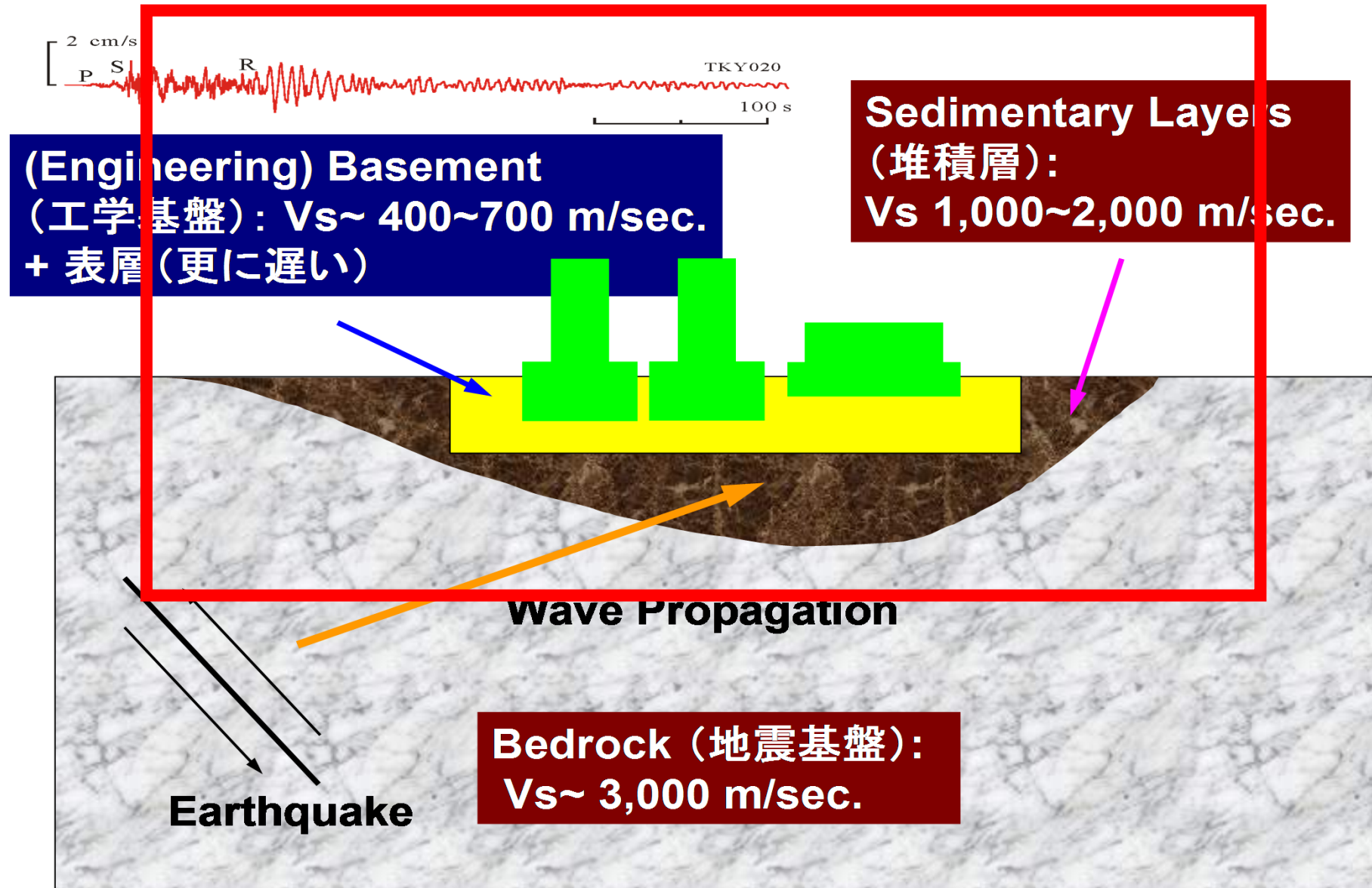


2003年 十勝沖地震

長周期地震波動(表面波)のために苫小牧の
石油タンクがスロッシングを起こし火災発生

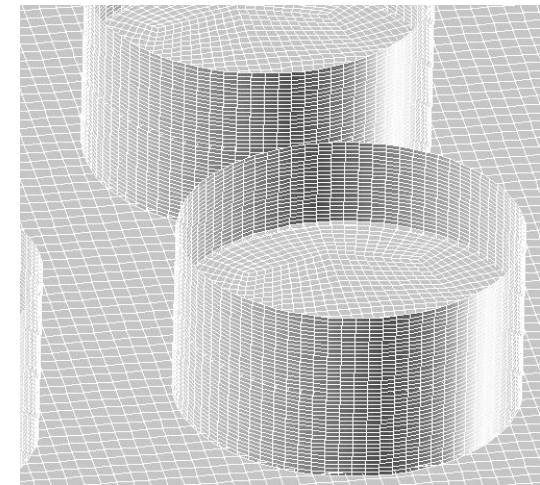
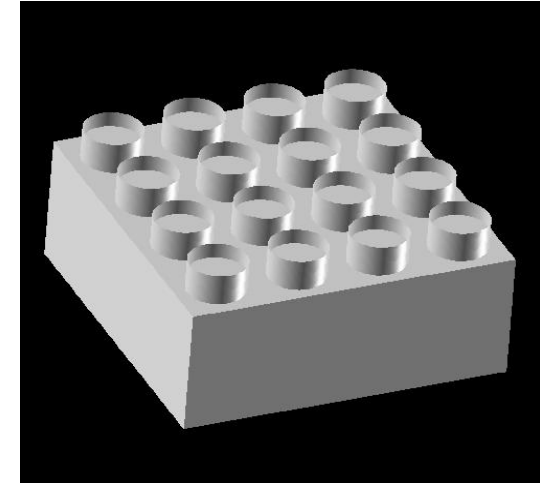


地盤・石油タンク振動連成シミュレーション

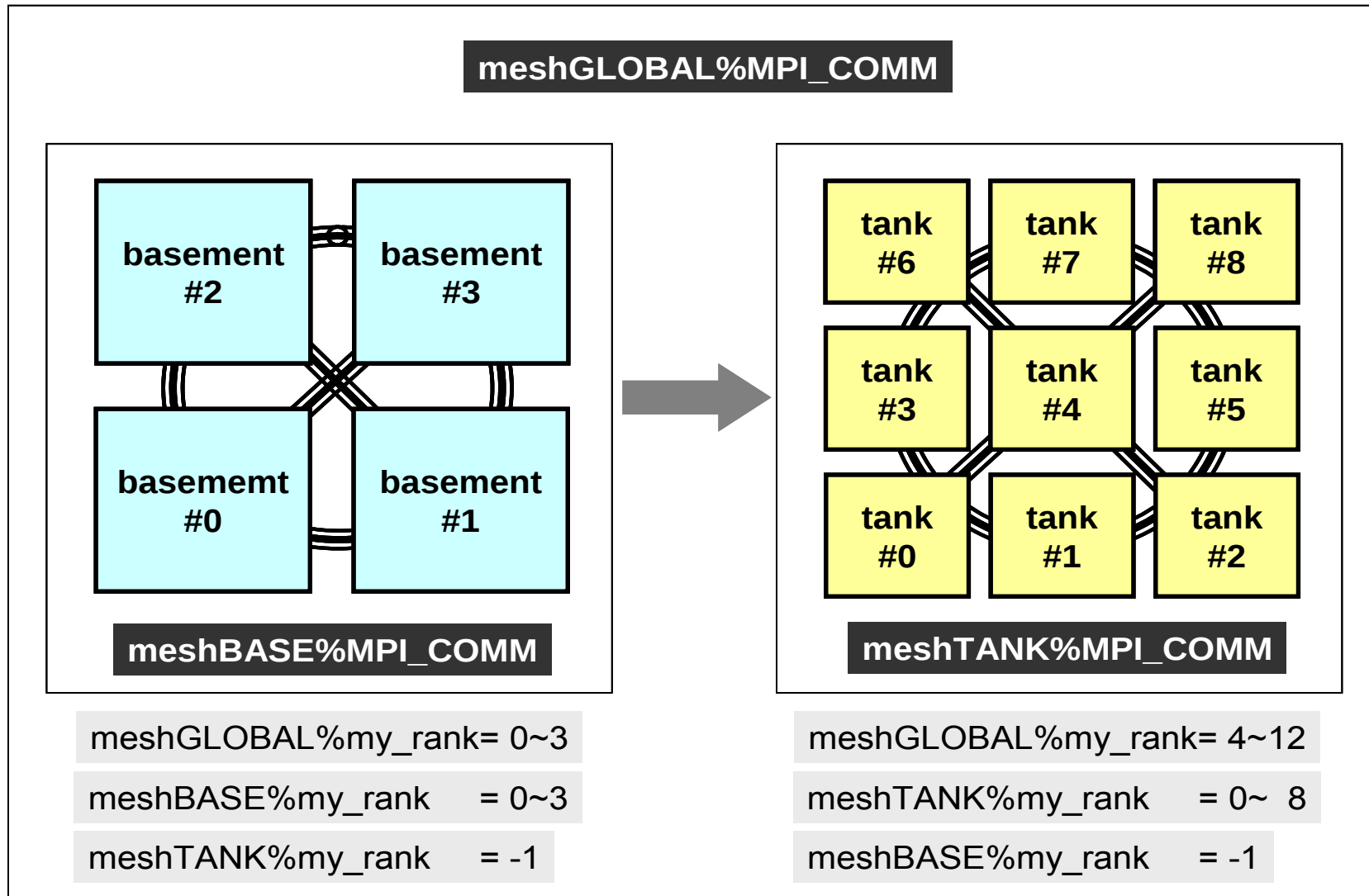


地盤，タンクモデル

- 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
 - $240\text{m} \times 240\text{m} \times 100\text{m}$
- 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 コミュニケーターを指定する
 - rank 整数 0 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 0 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 0 エラーコード
 - ierr 整数 0 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 0 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, 1000000000  
  a= 1.d0  
enddo  
Etime= MPI_WTIME ( )  
  
write (*, '(i5,1pe16.6)') my_rank, Etime-Stime
```

MPI_Wtime の例

```
$> cd <$0-S1>
```

```
$> mpifccpx -O1 time.c
```

```
$> mpifrtpx -O1 time.f
```

```
$> 実行(4プロセス) go4.sh
```

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

プロセス
番号

計算時間

MPI_Wtick

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

- **time= MPI_Wtick ()**

– time R8 0 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 <$0-S1>
```

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

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

```
$> (実行:1プロセス) go1.sh
```

MPI_BARRIER

- コミュニケータ「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ため、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 0 完了コード

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

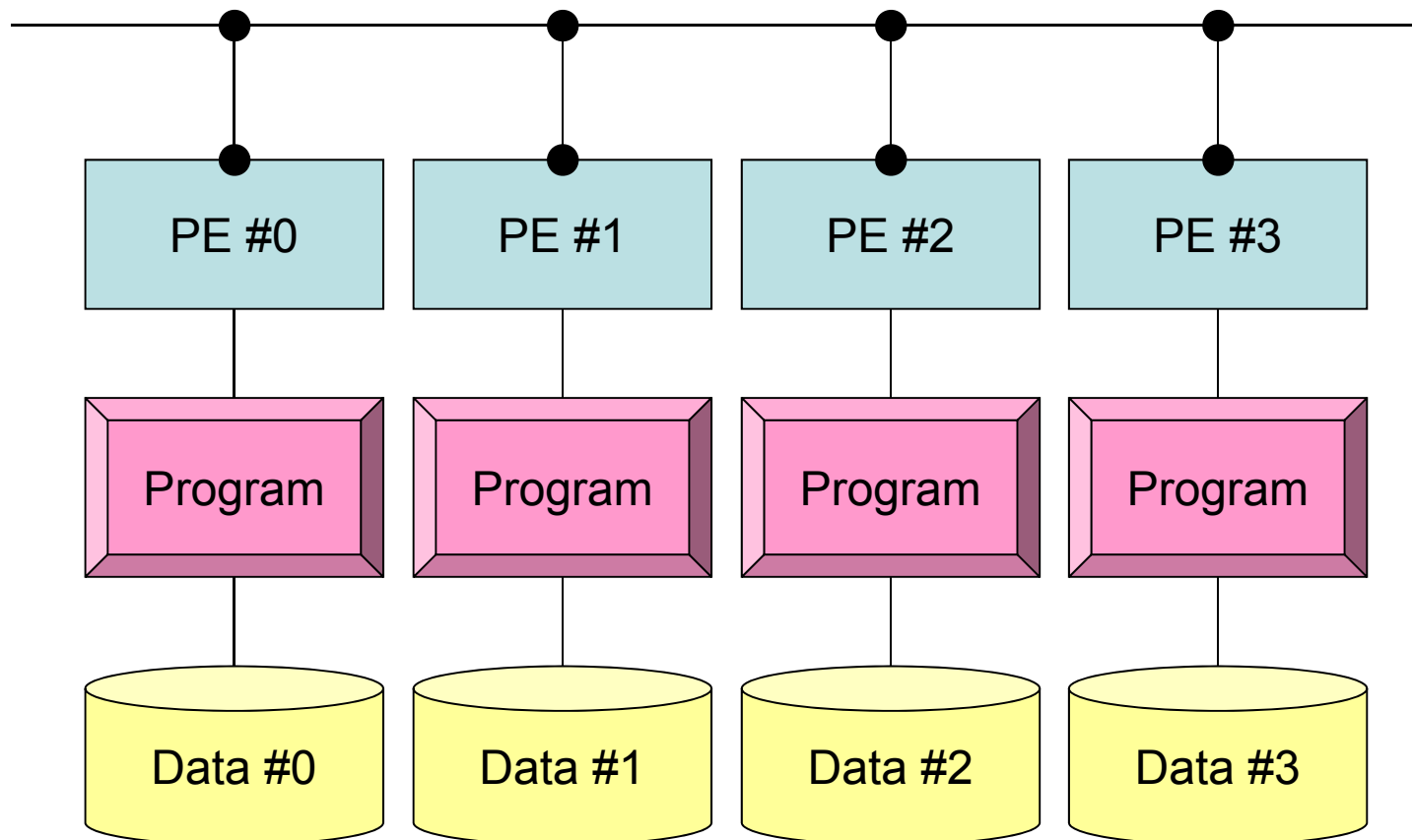
データ構造とアルゴリズム

- コンピュータ上で計算を行うプログラムはデータ構造とアルゴリズムから構成される。
- 両者は非常に密接な関係にあり、あるアルゴリズムを実現するためには、それに適したデータ構造が必要である。
 - 極論を言えば「データ構造＝アルゴリズム」と言っても良い。
 - もちろん「そうではない」と主張する人もいるが、科学技術計算に関する限り、中島の経験では「データ構造＝アルゴリズム」と言える。
- 並列計算を始めるにあたって、基本的なアルゴリズムに適したデータ構造を定める必要がある。

SPMD: Single Program Multiple Data

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

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 $N_g(=20)$ のベクトル V_g 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。
 - 単体CPUの場合ともほとんど変わらない。

全体データと局所データ

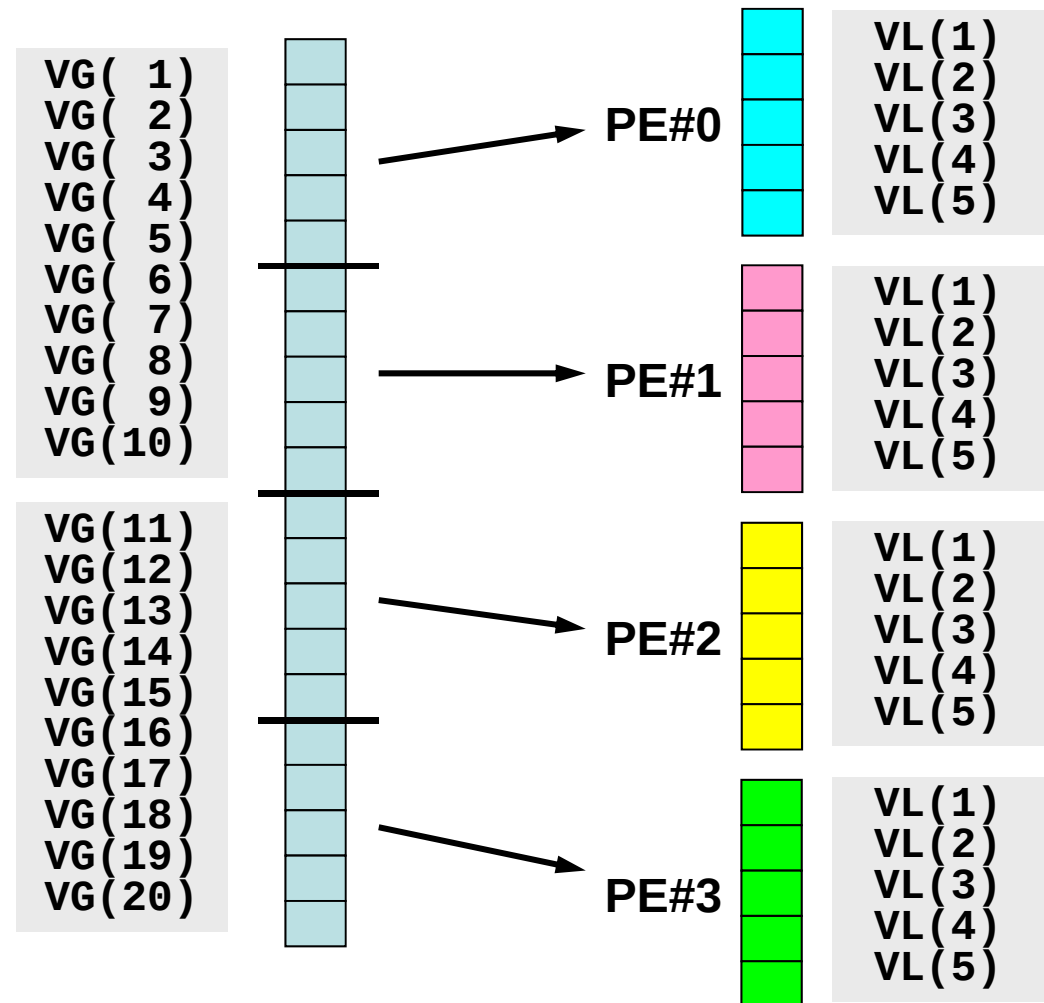
- Vg
 - 領域全体
 - 1番から20番までの「全体番号」を持つ「全体データ(Global Data)」
- VI
 - 各プロセス(PE, プロセッサ, 領域)
 - 1番から5番までの「局所番号」を持つ「局所データ(Local Data)」
 - できるだけ局所データを有効に利用することで, 高い並列性能が得られる。

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

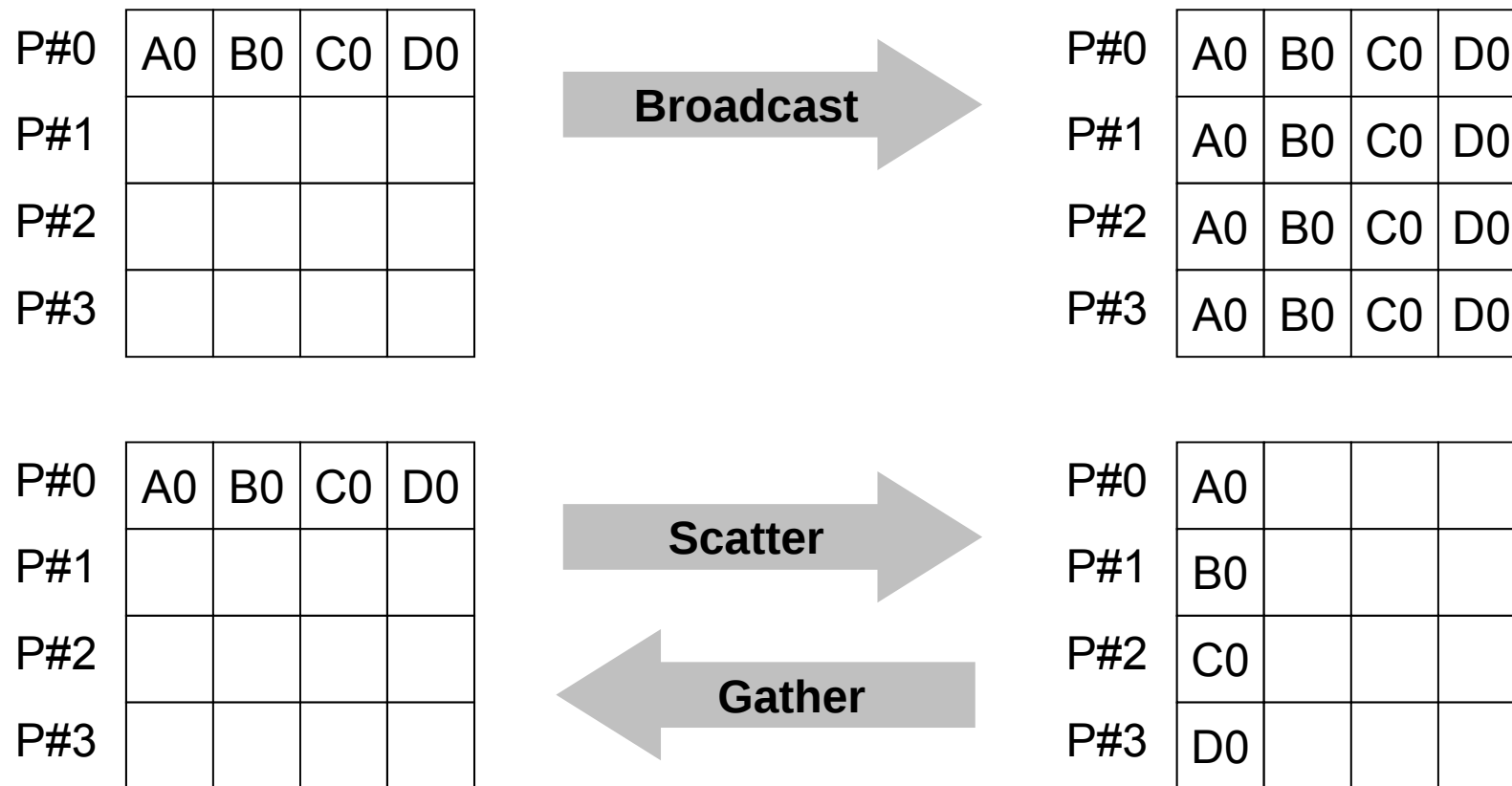
- Vg
 - 領域全体
 - 1番から20番までの「全体番号」を持つ「全体データ(Global Data)」
- VI
 - 各プロセッサ
 - 1番から5番までの「局所番号」を持つ「局所データ(Local Data)」
- この講義で常に注意してほしいこと
 - Vg(全体データ)からVI(局所データ)をどのように生成するか。
 - VgからVI, VIからVgへデータの中身をどのようにマッピングするか。
 - VIがプロセスごとに独立して計算できない場合はどうするか。
 - できる限り「局所性」を高めた処理を実施する⇒高い並列性能
 - そのための「データ構造」,「アルゴリズム」

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

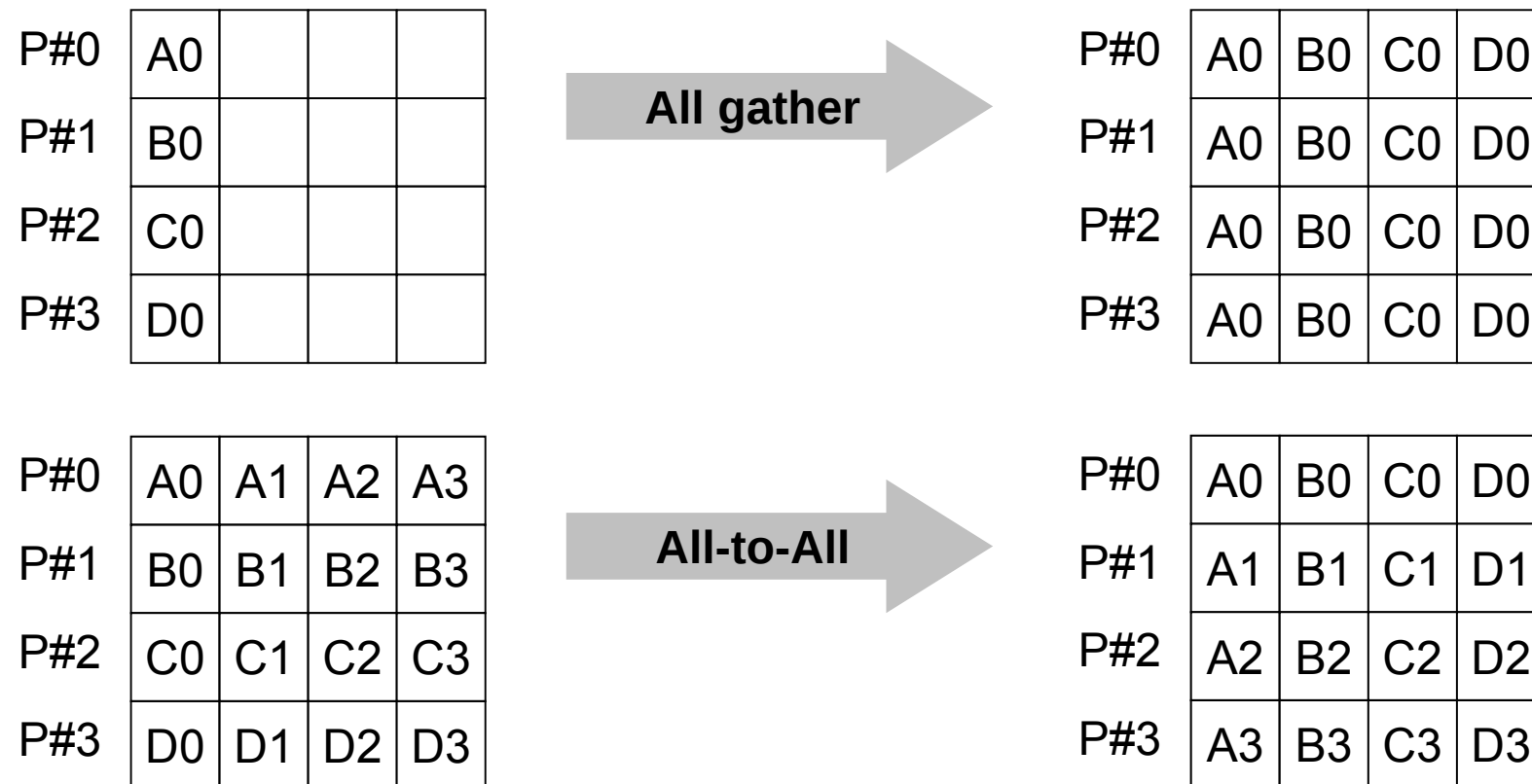
グループ通信とは

- コミュニケータで指定されるグループ全体に関わる通信。
- 例
 - 制御データの送信
 - 最大値, 最小値の判定
 - 総和の計算
 - ベクトルの内積の計算
 - 密行列の転置

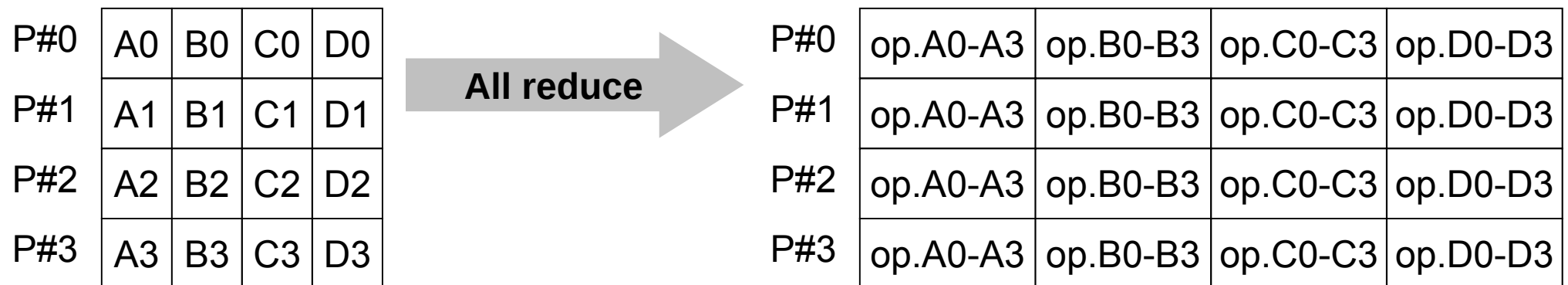
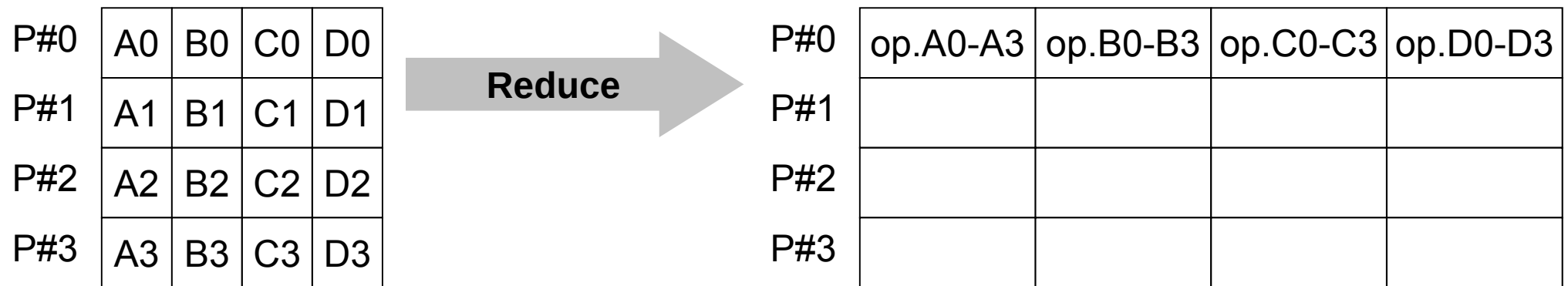
グループ通信の例(1/4)



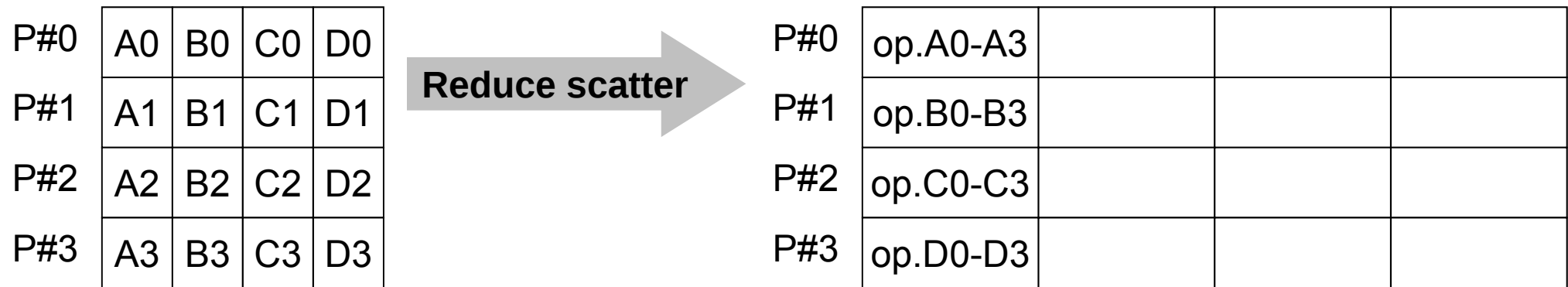
グループ通信の例(2/4)



グループ通信の例(3/4)



グループ通信の例(4/4)



グループ通信による計算例

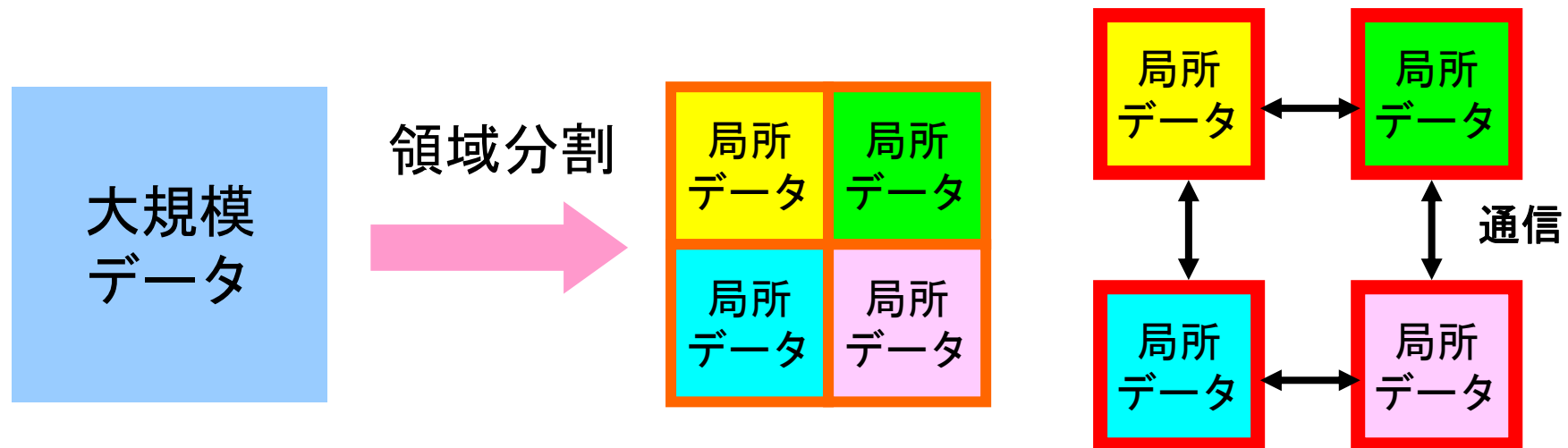
- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み

全体データと局所データ

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

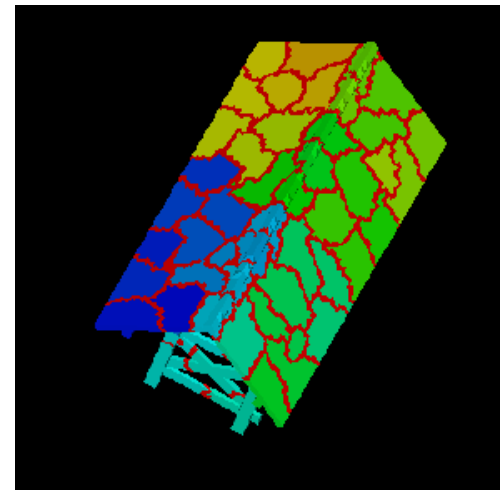
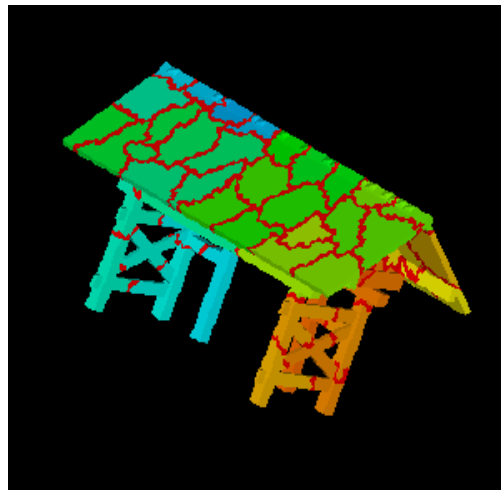
領域分割

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



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(実は不可)

```
>$ cd <$T-S1>
```

```
>$ cc -O3 dot.c
```

```
>$ f90 -O3 dot.f
```

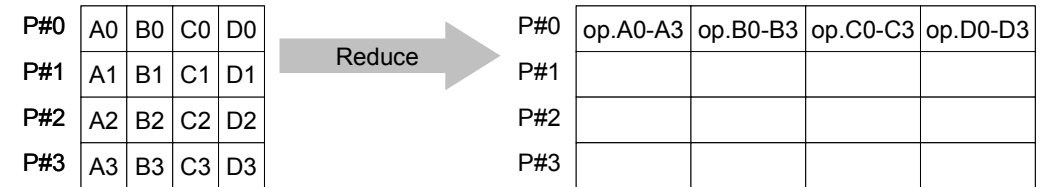
```
>$ ./a.out
```

1	2.	3.
2	2.	3.
3	2.	3.

...

18	2.	3.
19	2.	3.
20	2.	3.

dot product 120.



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)
```

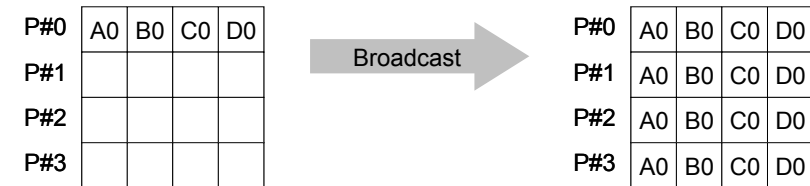
```
real(kind=8):: X0, XSUM  
  
call MPI_REDUCE  
(X0, XSUM, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

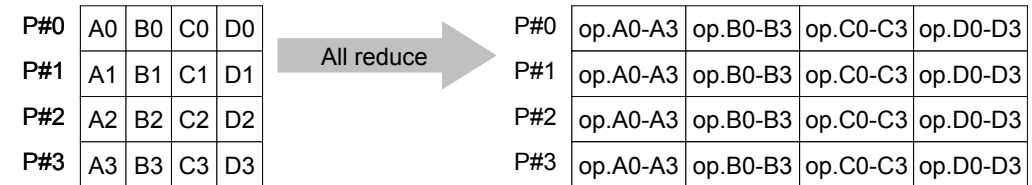
各プロセスにおける, X0の総和が0番PEのXSUMに入る。

```
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。





Fortran

MPI_Reduce/Allreduceの“op”

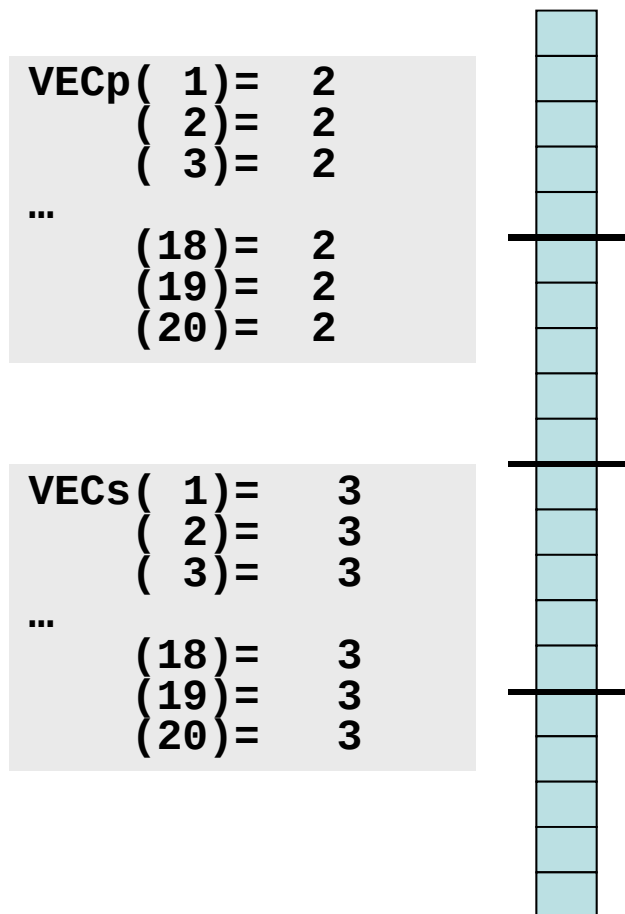
call MPI_REDUCE

(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)

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

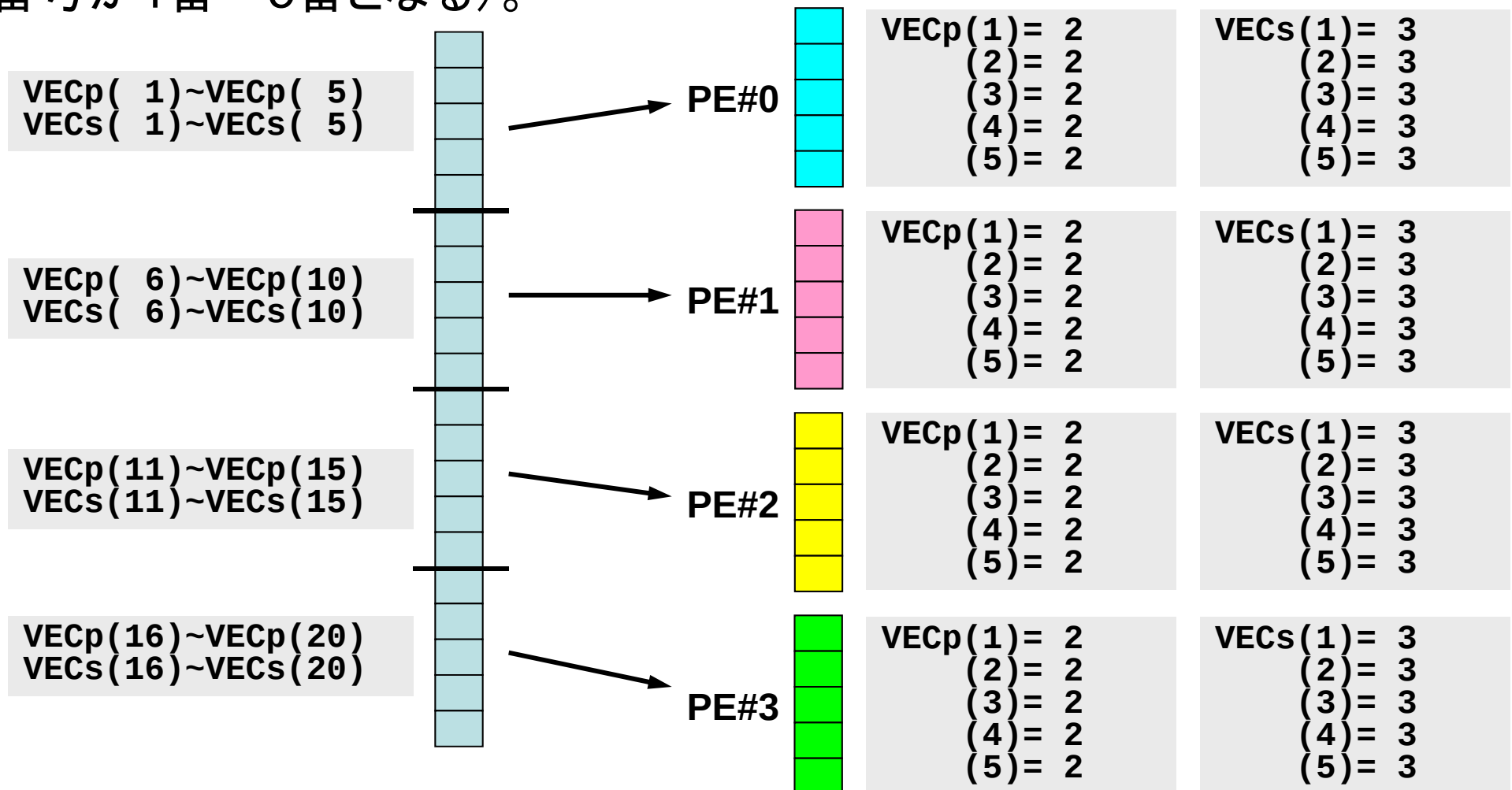
局所データの考え方(1/2)

- 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)



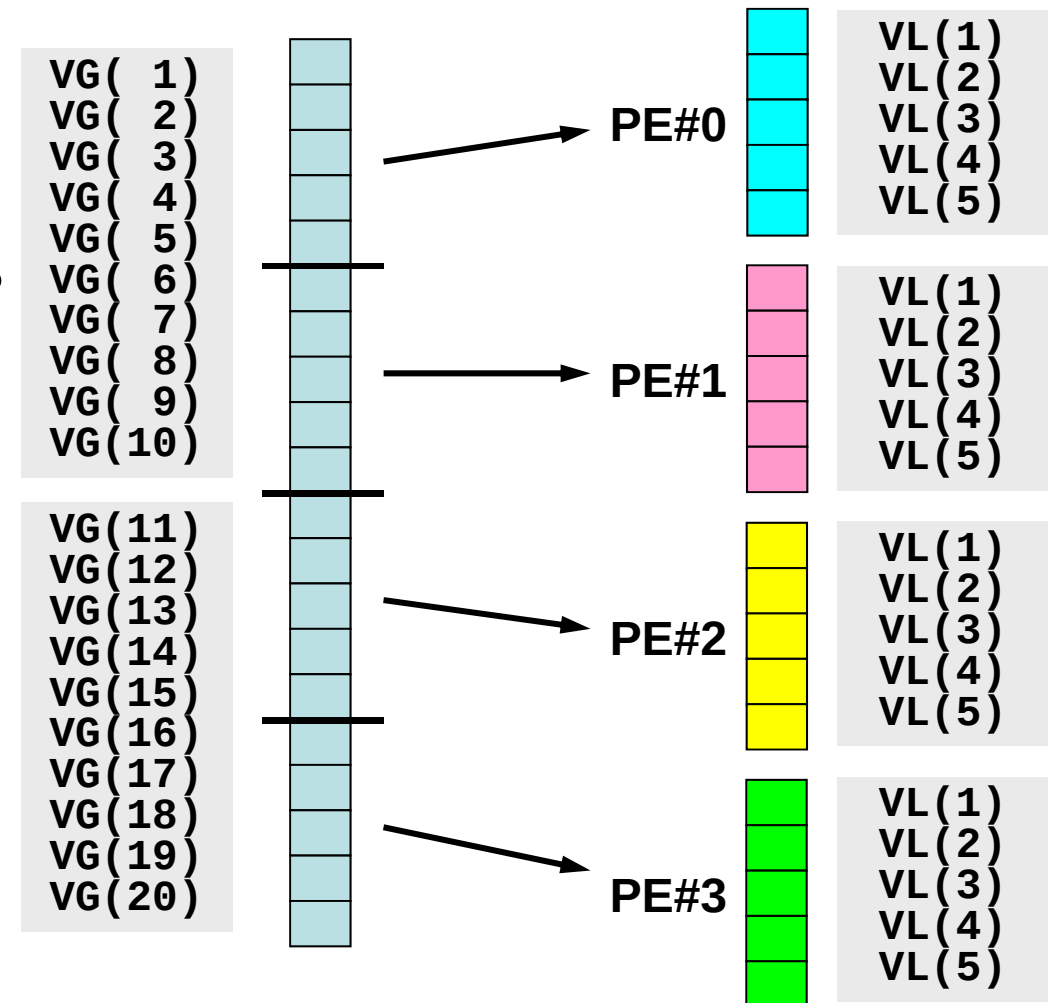
局所データの考え方(2/2)

- 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)。



とは言え . . .

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



内積の並列計算例(1/3)

<\$0-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)

<\$0-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)

<\$0-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。

<\$0-S1>/allreduce.f/c の実行例

```
$> mpifccpx -Kfast allreduce.c  
$> mpifrtpx -Kfast allreduce.f  
$> (実行:4プロセス) go4.sh
```

(my_rank, sumALLREDUCE, sumREDUCE)				
before	BCAST	0	1.2000000E+02	1.2000000E+02
after	BCAST	0	1.2000000E+02	1.2000000E+02
before	BCAST	1	1.2000000E+02	0.0000000E+00
after	BCAST	1	1.2000000E+02	1.2000000E+02
before	BCAST	3	1.2000000E+02	0.0000000E+00
after	BCAST	3	1.2000000E+02	1.2000000E+02
before	BCAST	2	1.2000000E+02	0.0000000E+00
after	BCAST	2	1.2000000E+02	1.2000000E+02

グループ通信による計算例

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

全体データと局所データ(1/3)

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

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

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

全体データと局所データ(2/3)

- 簡単のために,
 - NG=32
 - ALPHA=1000.
 - MPIプロセス数=4
- ベクトルVECgとして以下のような32個の成分を持つベクトルを仮定する(<\$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 VEC_g 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);

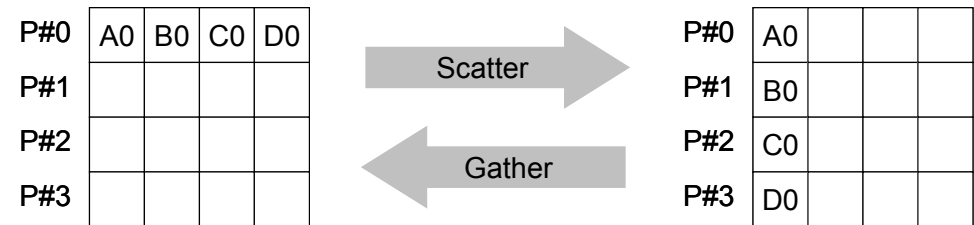
  fp = fopen("a1x.all", "r");
  if(!MyRank) for(i=0;i<NG;i++) {
    fscanf(fp, "%lf", &VECg[i]);
  }
```

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

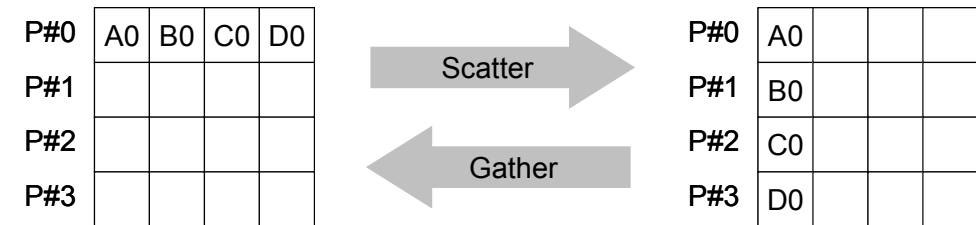
MPI_SCATTER



- 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 0 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 0 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 0 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 0 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)
```

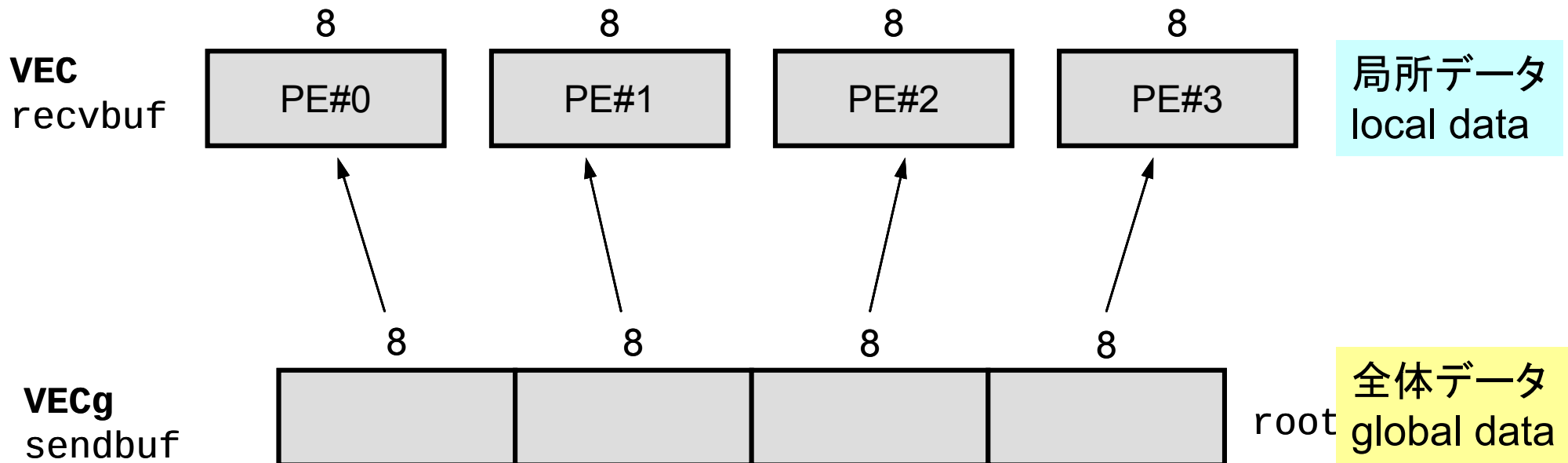
```
int N=8;
double VEC [8];
...
MPI_Scatter (&VECg, N, MPI_DOUBLE, &VEC, N,
MPI_DOUBLE, 0, <comm>);
```

```
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

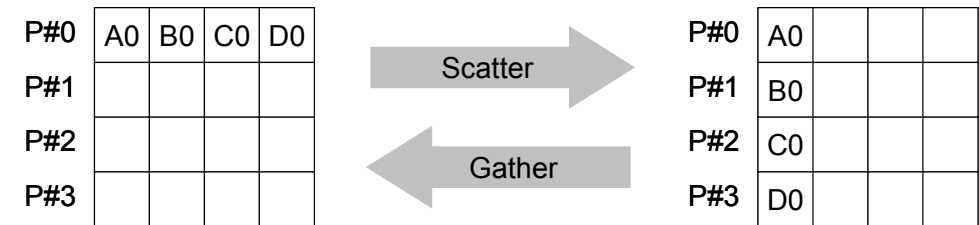
```
after 3 1 1401.  
after 3 2 1403.  
after 3 3 1405.  
after 3 4 1406.  
after 3 5 1409.  
after 3 6 1411.  
after 3 7 1421.  
after 3 8 1451.
```

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 0 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 0 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)

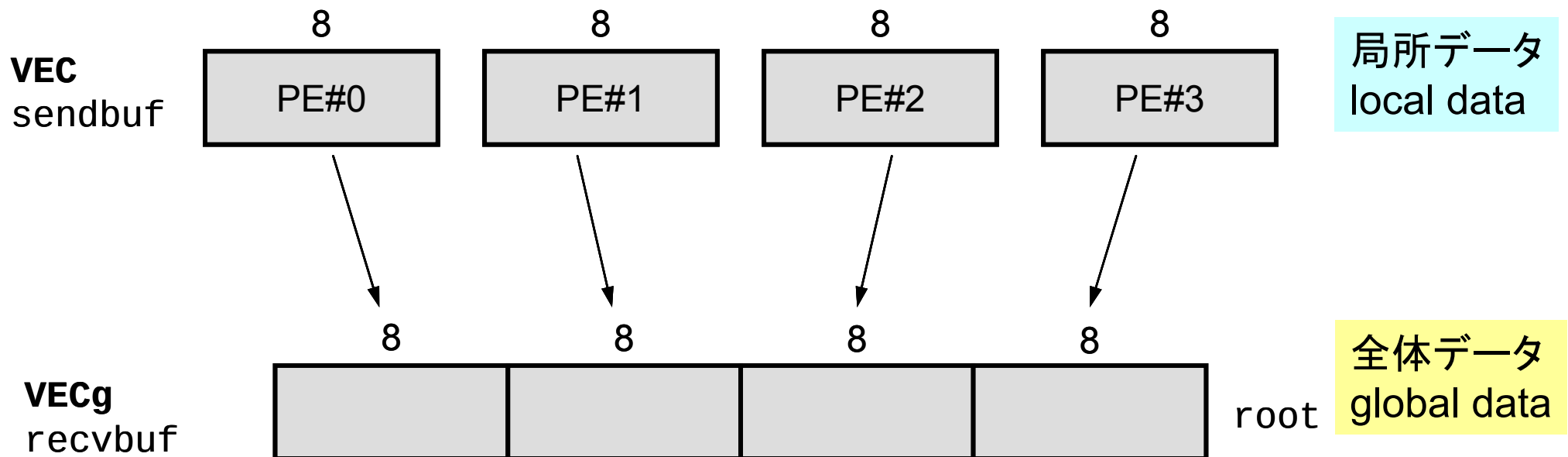
各プロセスの結果を再び長さ32のベクトルにまとめる

- 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



<\$0-S1>/scatter-gather.f/c

実行例

```
$> mpifccpx -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.

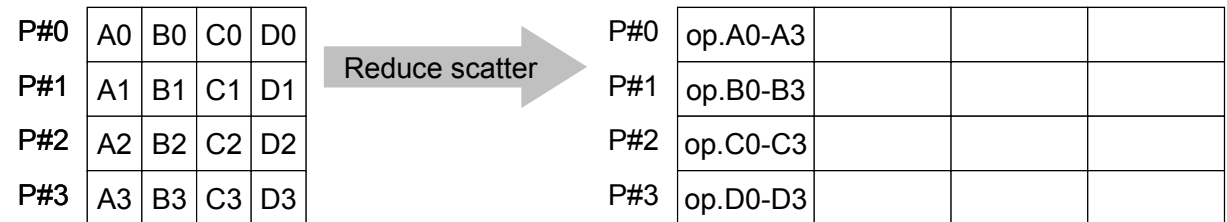
PE#2

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

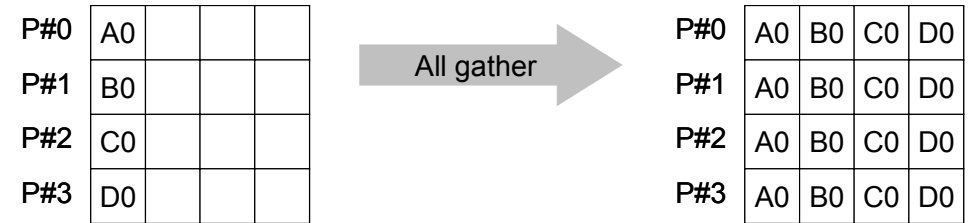
after 3 1 1401.
after 3 2 1403.
after 3 3 1405.
after 3 4 1406.
after 3 5 1409.
after 3 6 1411.
after 3 7 1421.
after 3 8 1451.

MPI_REDUCE_SCATTER



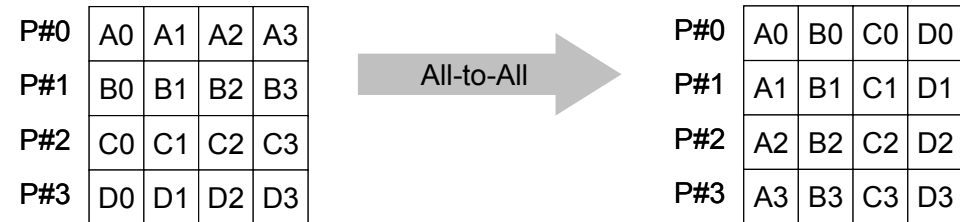
- MPI_REDUCE + MPI_SCATTER
- call MPI_REDUCE_SCATTER (sendbuf, recvbuf, rcount, datatype, op, comm, ierr)
 - sendbuf 任意 I 送信バッファの先頭アドレス,
 - recvbuf 任意 0 受信バッファの先頭アドレス,
 - rcount 整数 I 受信メッセージのサイズ(配列: サイズ=プロセス数)
 - datatype 整数 I メッセージのデータタイプ
 - op 整数 I 計算の種類
 - comm 整数 I コミュニケータを指定する
 - ierr 整数 0 完了コード

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 0 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 0 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 0 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 0 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

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)

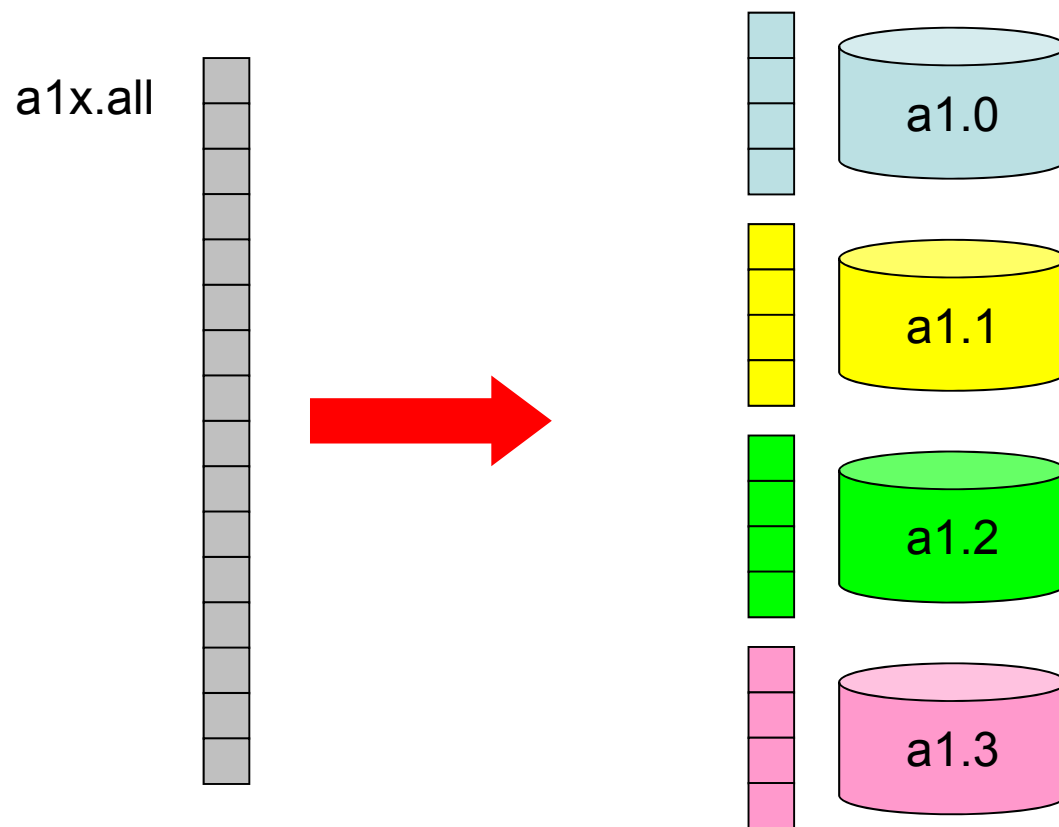
```
>$ cd <$0-S1>
>$ ls a1.*
a1.0 a1.1 a1.2 a1.3    「a1x.a11」を4つに分割したもの

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

>$ 実行:4プロセス go4.sh
```

分散ファイルの操作

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



分散ファイル読み込み: 等データ長 (2/2)

<\$0-\$1>/file.f

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

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

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

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

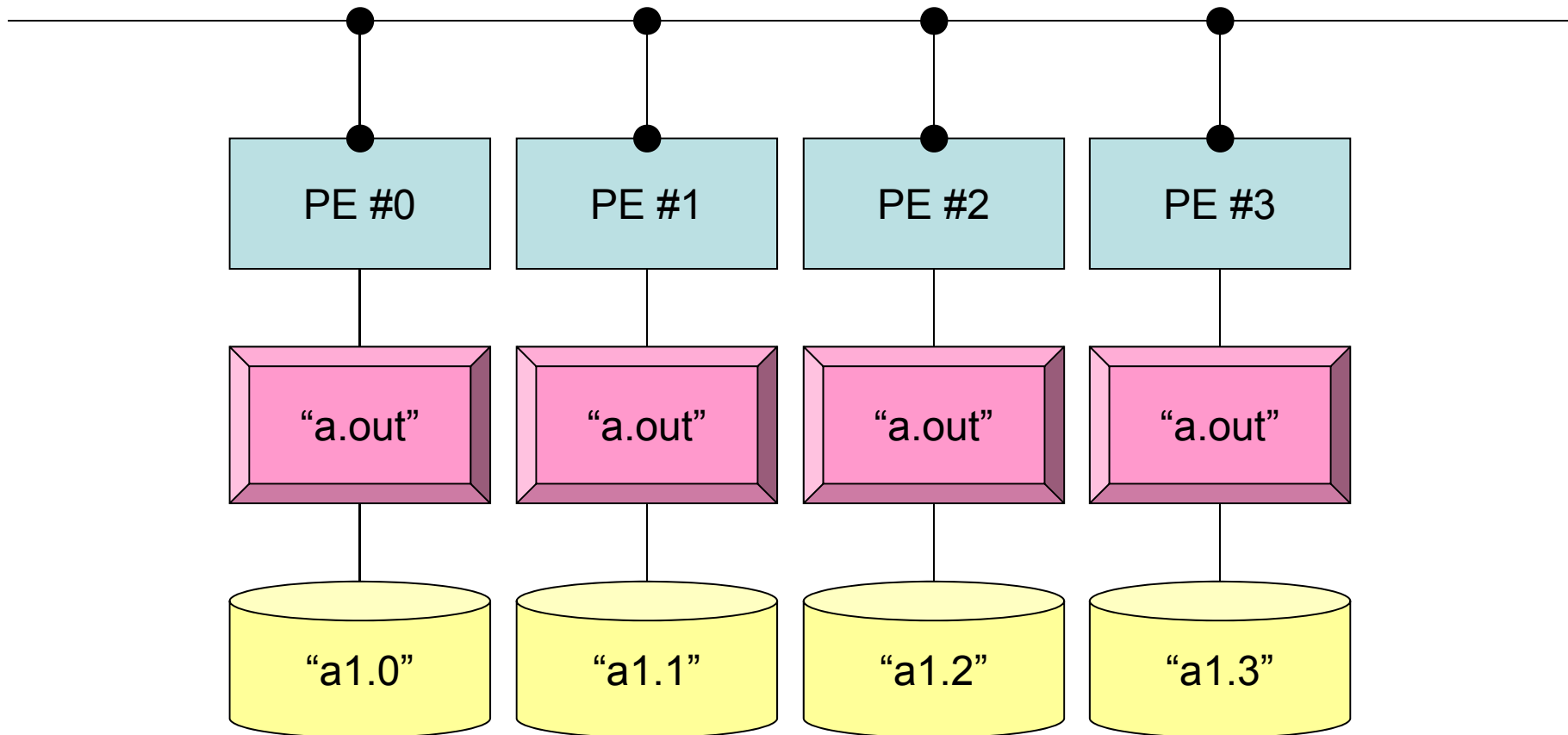
call MPI_FINALIZE (ierr)

stop
end
```

Hello とそんなに
変わらない

「局所番号(1~8)」で
読み込む

SPMDの典型例



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

分散ファイル読み込み: 可変長 (1/2)

```
>$ cd <$0-S1>
>$ 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

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

>$ 実行:4プロセス go4.sh
```

分散ファイルの読み込み: 可変長 (2/2)

<\$0-S1>/file2.f

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

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

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

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

call MPI_FINALIZE (ierr)
stop
end
```

Nが各データ(プロセッサ)で異なる

局所データの作成法

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

グループ通信による計算例

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

MPI_GATHERV, MPI_SCATTERV

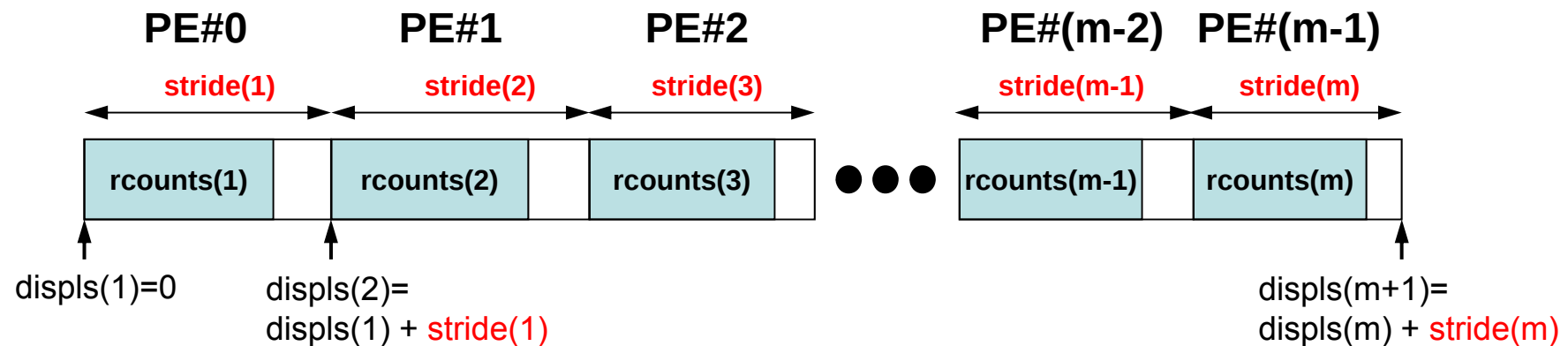
- これまで紹介してきた, MPI_GATHER, MPI_SCATTERなどは, 各プロセッサからの送信, 受信メッセージが均等な場合。
- 末尾に「V」が付くと, 各ベクトルが可変長さの場合となる。
 - MPI_GATHERV
 - MPI_SCATTERV
 - MPI_ALLGATHERV
 - MPI_ALLTOALLV

MPI_ALLGATHERV

- MPI_ALLGATHER の可変長さベクトル版
 - 「局所データ」から「全体データ」を生成する
- call MPI_ALLGATHERV (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm, ierr)
 - sendbuf 任意 I 送信バッファの先頭アドレス,
 - scount 整数 I 送信メッセージのサイズ
 - sendtype 整数 I 送信メッセージのデータタイプ
 - recvbuf 任意 0 受信バッファの先頭アドレス,
 - rcounts 整数 **I** 受信メッセージのサイズ(配列: サイズ=PETOT)
 - displs 整数 **I** 受信メッセージのインデックス(配列: サイズ=PETOT+1)
 - recvtype 整数 I 受信メッセージのデータタイプ
 - comm 整数 I コミュニケータを指定する
 - ierr 整数 0 完了コード

MPI_ALLGATHERV(続き)

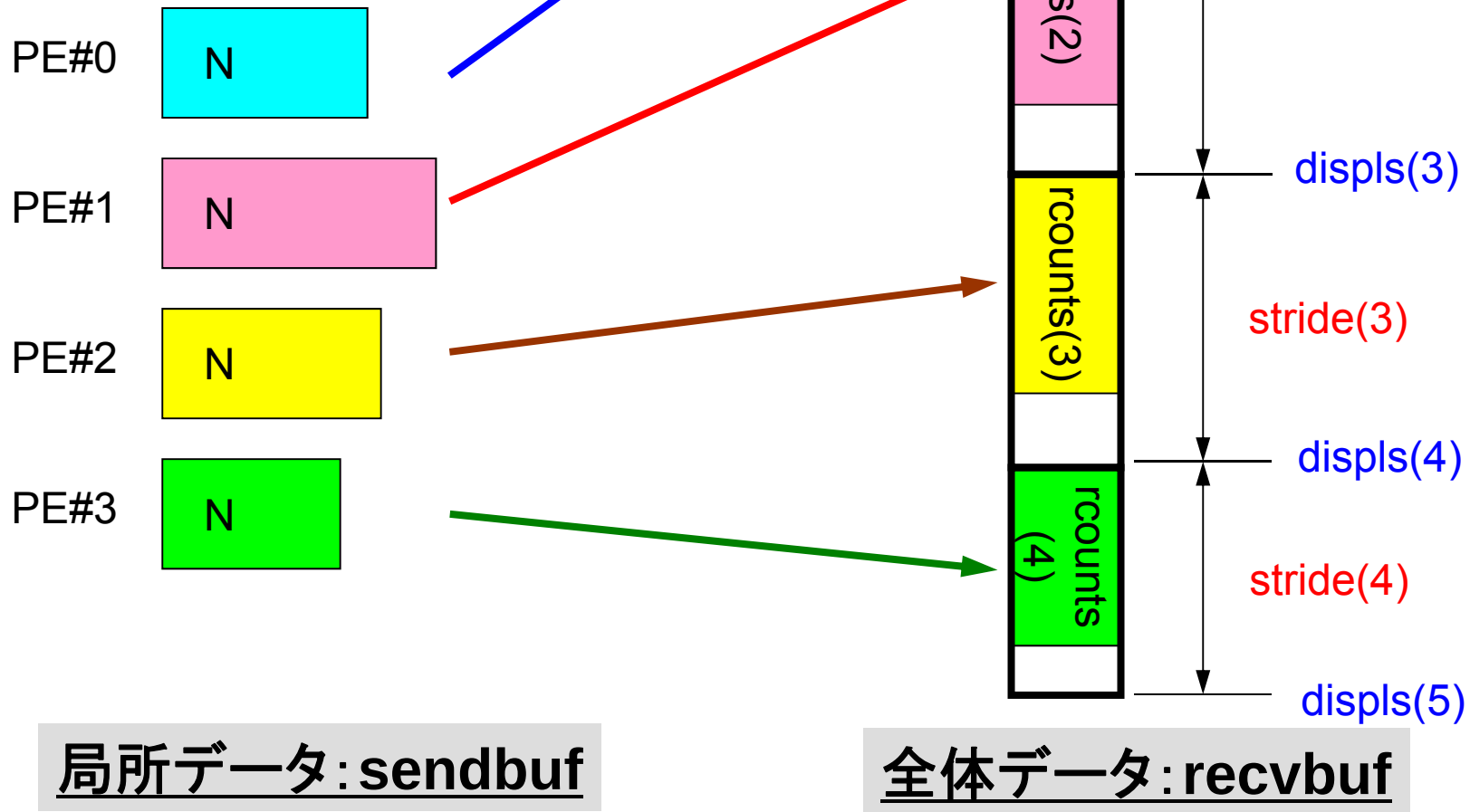
- call MPI_ALLGATHERV (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm, ierr)
 - rcounts 整数 I 受信メッセージのサイズ(配列:サイズ=PETOT)
 - displs 整数 I 受信メッセージのインデックス(配列:サイズ=PETOT+1)
 - この2つの配列は、最終的に生成される「全体データ」のサイズに関する配列であるため、各プロセスで配列の全ての値が必要になる:
 - もちろん各プロセスで共通の値を持つ必要がある。
 - 通常は $\text{stride}(i)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



$$\text{size(recvbuf)} = \text{displs}(\text{PETOT}+1) = \text{sum}(\text{stride})$$

MPI_ALLGATHERV でやっていること

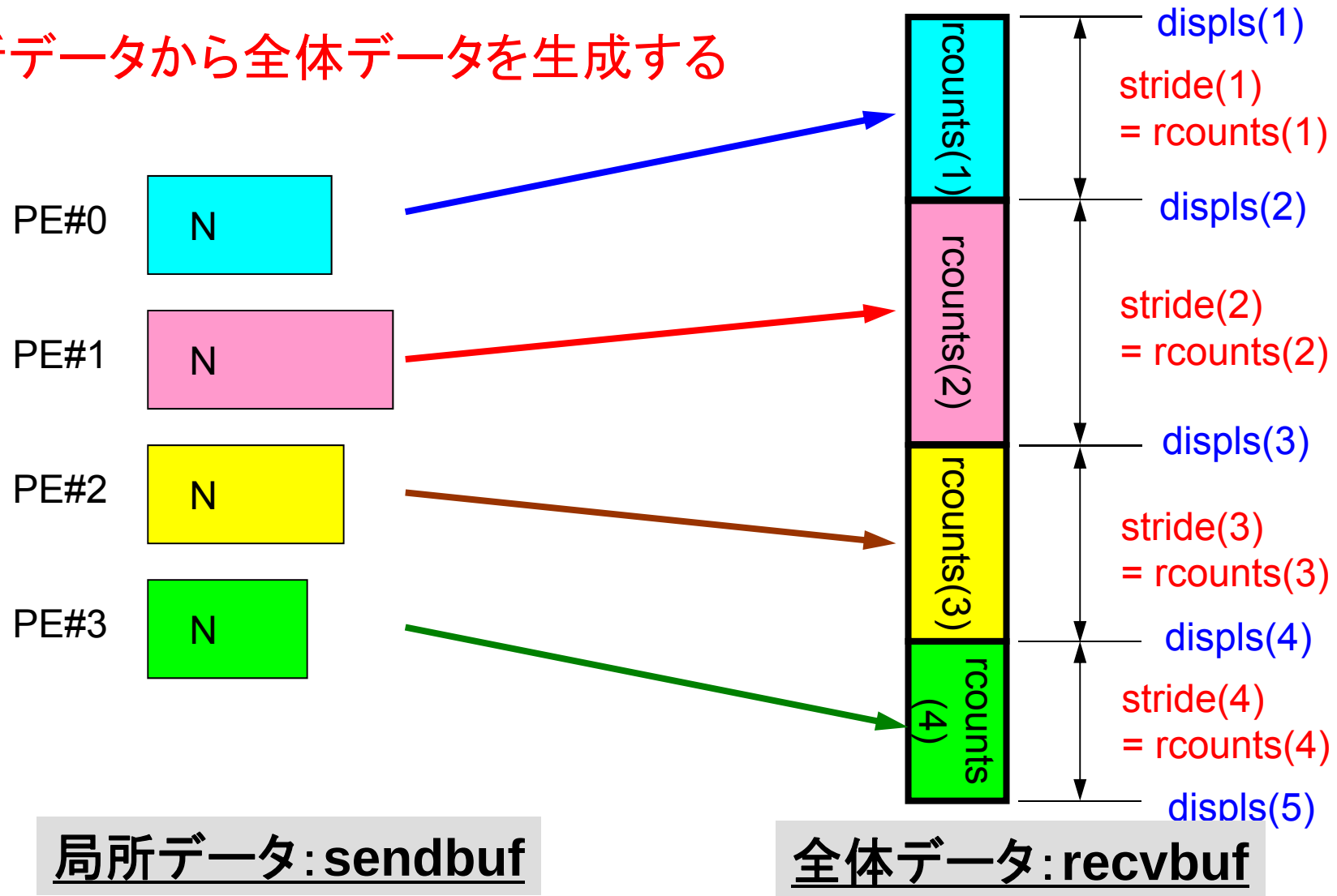
局所データから全体データを
生成する



MPI_ALLGATHERV

でやっていること

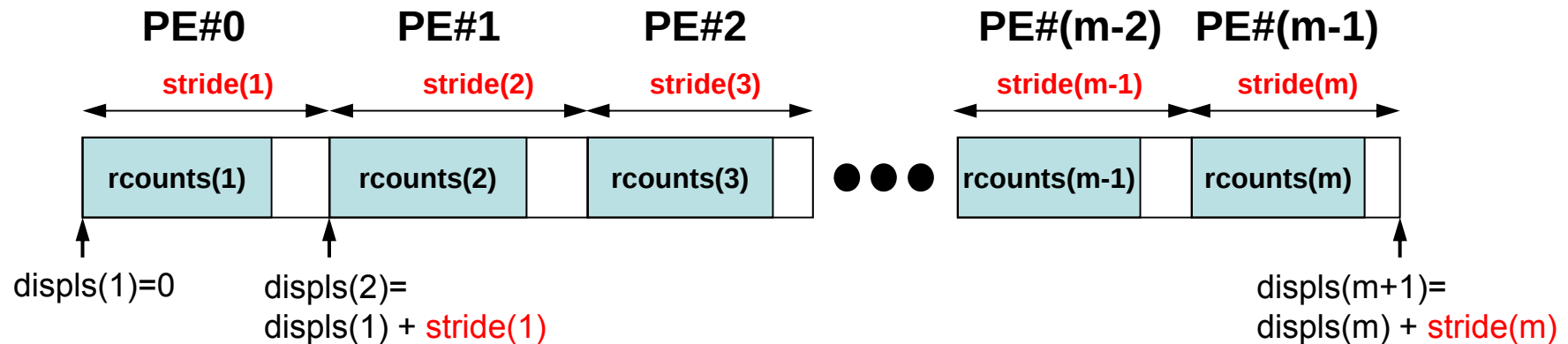
局所データから全体データを生成する



MPI_ALLGATHERV詳細(1/2)

Fortran

- call MPI_ALLGATHERV (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm, ierr)
 - rcounts 整数 I 受信メッセージのサイズ(配列: サイズ=PETOT)
 - displs 整数 I 受信メッセージのインデックス(配列: サイズ=PETOT+1)
- rcounts**
 - 各PEにおけるメッセージサイズ: 局所データのサイズ
- displs**
 - 各局所データの全体データにおけるインデックス
 - **displs(PETOT+1)**が全体データのサイズ

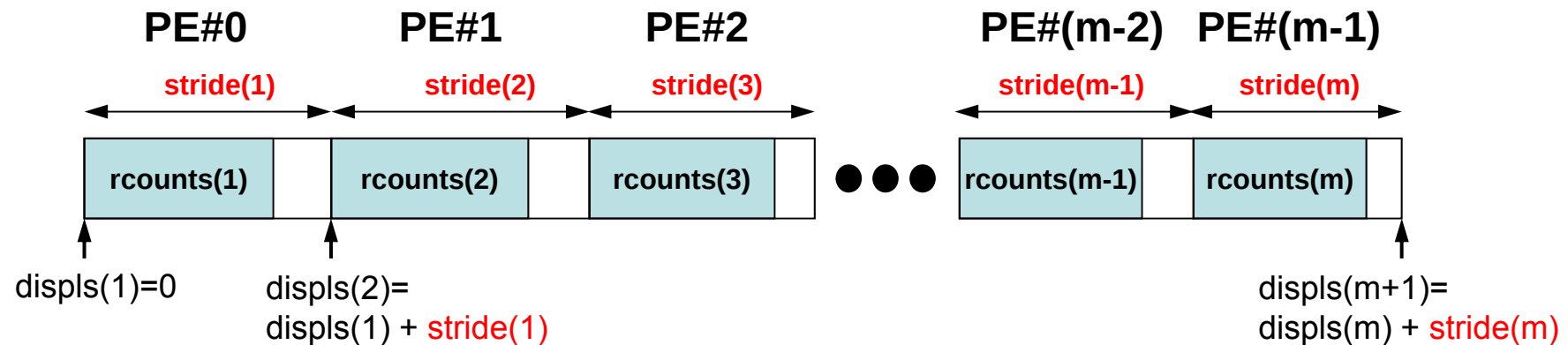


$$size(recvbuf) = displs(PETOT+1) = \text{sum}(\text{stride})$$

MPI_ALLGATHERV詳細(2/2)

Fortran

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



$\text{size}(\text{recvbuf}) = \text{displs}(\text{PETOT}+1) = \text{sum}(\text{stride})$

MPI_ALLGATHERV使用準備

例題: <\$O-S1>/agv.f, <\$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

MPI_ALLGATHERV 使用準備(1/4)

<\$0-\$1>/agv.f

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

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

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

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

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

N(NL)の値が各PEで異なることに注意

MPI_ALLGATHERV 使用準備 (2/4)

<\$0-\$1>/agv.f

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

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

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

&
&
各PEにrcountsを
生成

PE#0 N=8

PE#1 N=5

PE#2 N=7

PE#3 N=3



MPI_Allgather

rcounts(1:4)= {8, 5, 7, 3}

rcounts(1:4)= {8, 5, 7, 3}

rcounts(1:4)= {8, 5, 7, 3}

rcounts(1:4)= {8, 5, 7, 3}

MPI_ALLGATHERV 使用準備(2/4)

<\$0-\$1>/agv.f

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

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

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

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

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

stop
end
```

&
&
各PEにrcountsを
生成

各PEでdisplsを
生成

MPI_ALLGATHERV 使用準備(3/4)

```
> cd <$0-S1>
> mpifrtpx -Kfast agv.f, mpifccpx -Kfast agv.c
> バッチジョブ実行(4プロセス)
```

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

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

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

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

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

MPI_ALLGATHERV 使用準備(4/4)

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

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

課題S1 (1/2)

- 提出期限: 2012年10月12日(金) 1700
- 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)。

課題S1 (2/2)

- 内容(続き)

- 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)。

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

- 提出物(レポート): 最高級仕様

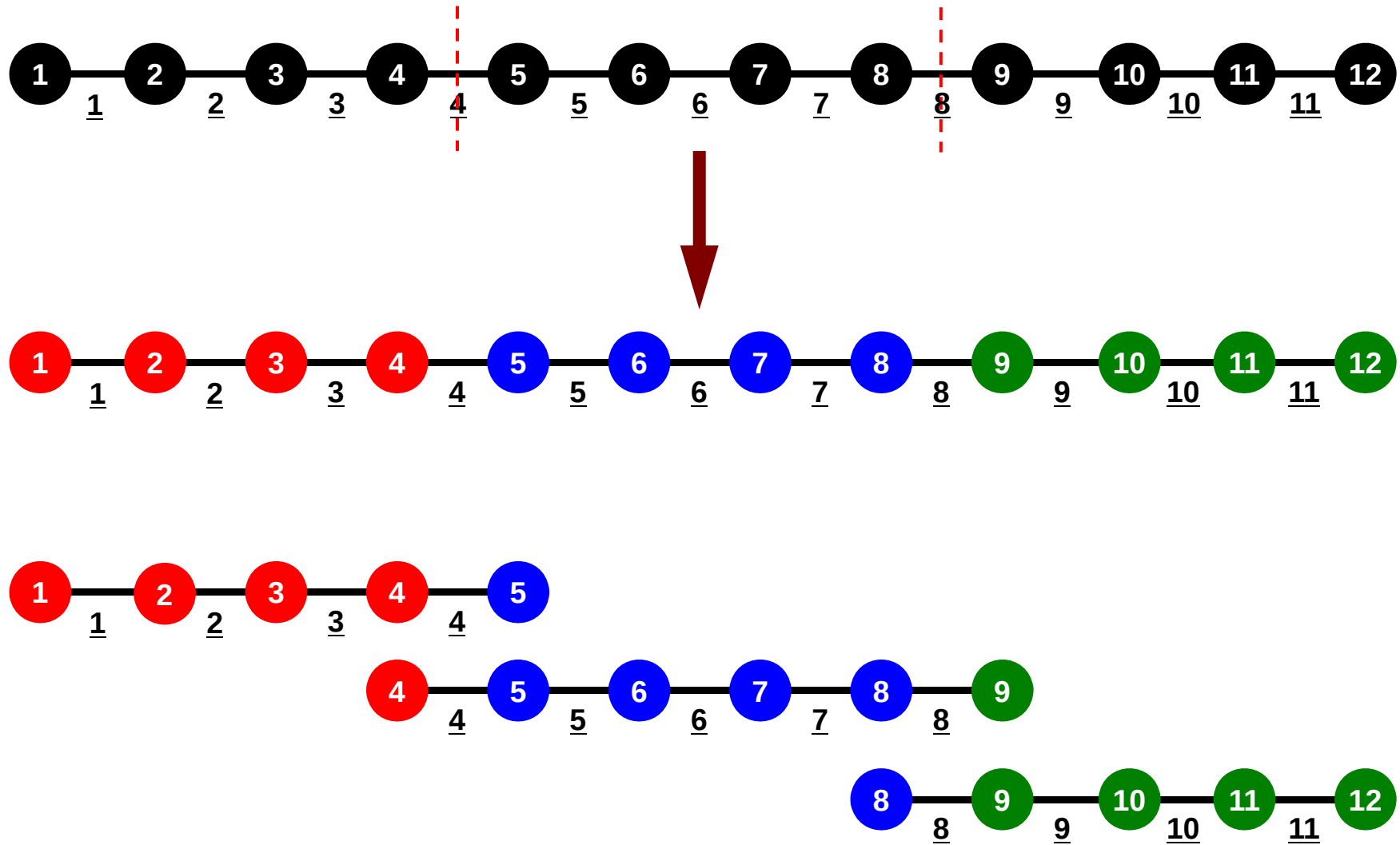
- 表紙: 氏名, 学籍番号, 課題番号を明記
- 各サブ課題につきA4 2枚以内(図表含む)でまとめること
 - 基本方針(フロー図), プログラム構造・説明, 考察・課題
- プログラムリスト
- 結果出力リスト(最小限にとどめること)

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

1対1通信

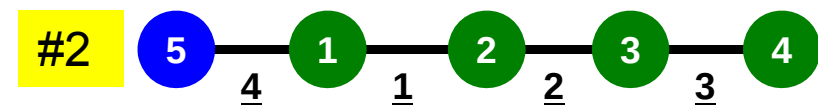
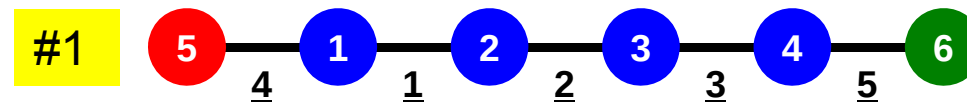
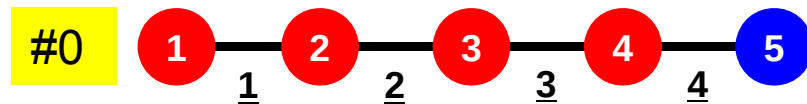
- 1対1通信とは ?
- 二次元問題, 一般化された通信テーブル
- 課題S2

一次元問題: 11要素, 12節点, 3領域



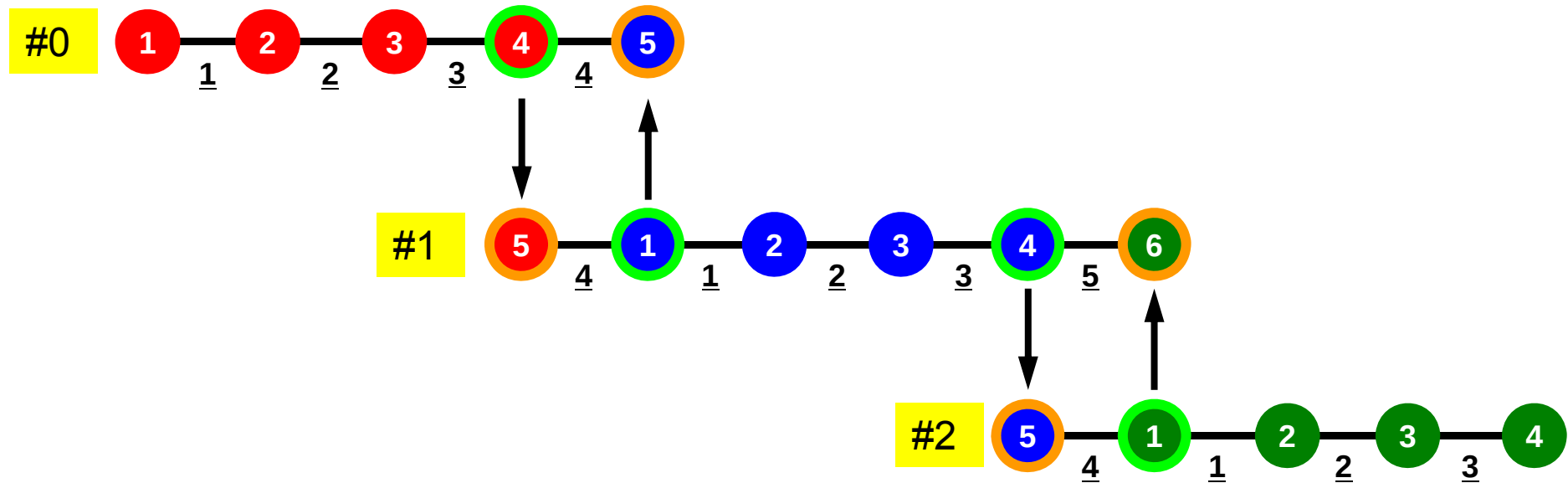
一次元問題: 11要素, 12節点, 3領域

局所番号: 節点・要素とも1からふる



一次元問題: 11要素, 12節点, 3領域

外点・境界点



前処理付き共役勾配法

Preconditioned Conjugate Gradient Method (CG)

```

Compute  $\mathbf{r}^{(0)} = \mathbf{b} - [\mathbf{A}]\mathbf{x}^{(0)}$ 
for i= 1, 2, ...
  solve  $[\mathbf{M}]\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$ 
   $\rho_{i-1} = \mathbf{r}^{(i-1)} \mathbf{z}^{(i-1)}$ 
  if i=1
     $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
     $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta_{i-1} \mathbf{p}^{(i-1)}$ 
  endif
   $\mathbf{q}^{(i)} = [\mathbf{A}]\mathbf{p}^{(i)}$ 
   $\alpha_i = \rho_{i-1} / \mathbf{p}^{(i)} \mathbf{q}^{(i)}$ 
   $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha_i \mathbf{p}^{(i)}$ 
   $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha_i \mathbf{q}^{(i)}$ 
  check convergence  $|\mathbf{r}|$ 
end

```

前処理: 対角スケーリング

前処理, ベクトル定数倍の加減

局所的な計算(内点のみ)が可能⇒並列処理

```
!C
!C-- {z} = [Minv] {r}

do i= 1, N
  W(i, Z) = W(i, DD) * W(i, R)
enddo
```

```
!C
!C-- {x} = {x} + ALPHA*{p}
!C   {r} = {r} - ALPHA*{q}

do i= 1, N
  PHI(i) = PHI(i) + ALPHA * W(i, P)
  W(i, R) = W(i, R) - ALPHA * W(i, Q)
enddo
```

1
2
3
4
5
6
7
8
9
10
11
12

内積

全体で和をとる必要がある⇒通信？

```
!C
!C-- ALPHA= RHO / {p} {q}

C1= 0. d0
do i= 1, N
  C1= C1 + W(i, P)*W(i, Q)
enddo
ALPHA= RHO / C1
```

1
2
3
4
5
6
7
8
9
10
11
12

行列ベクトル積

外点の値が必要⇒1対1通信

```
!C
!C-- {q} = [A] {p}

do i= 1, N
  W(i, Q) = DIAG(i)*W(i, P)
  do j= INDEX(i-1)+1, INDEX(i)
    W(i, Q) = W(i, Q) + AMAT(j)*W(ITEM(j), P)
  enddo
enddo
```



行列ベクトル積: ローカルに計算実施可能

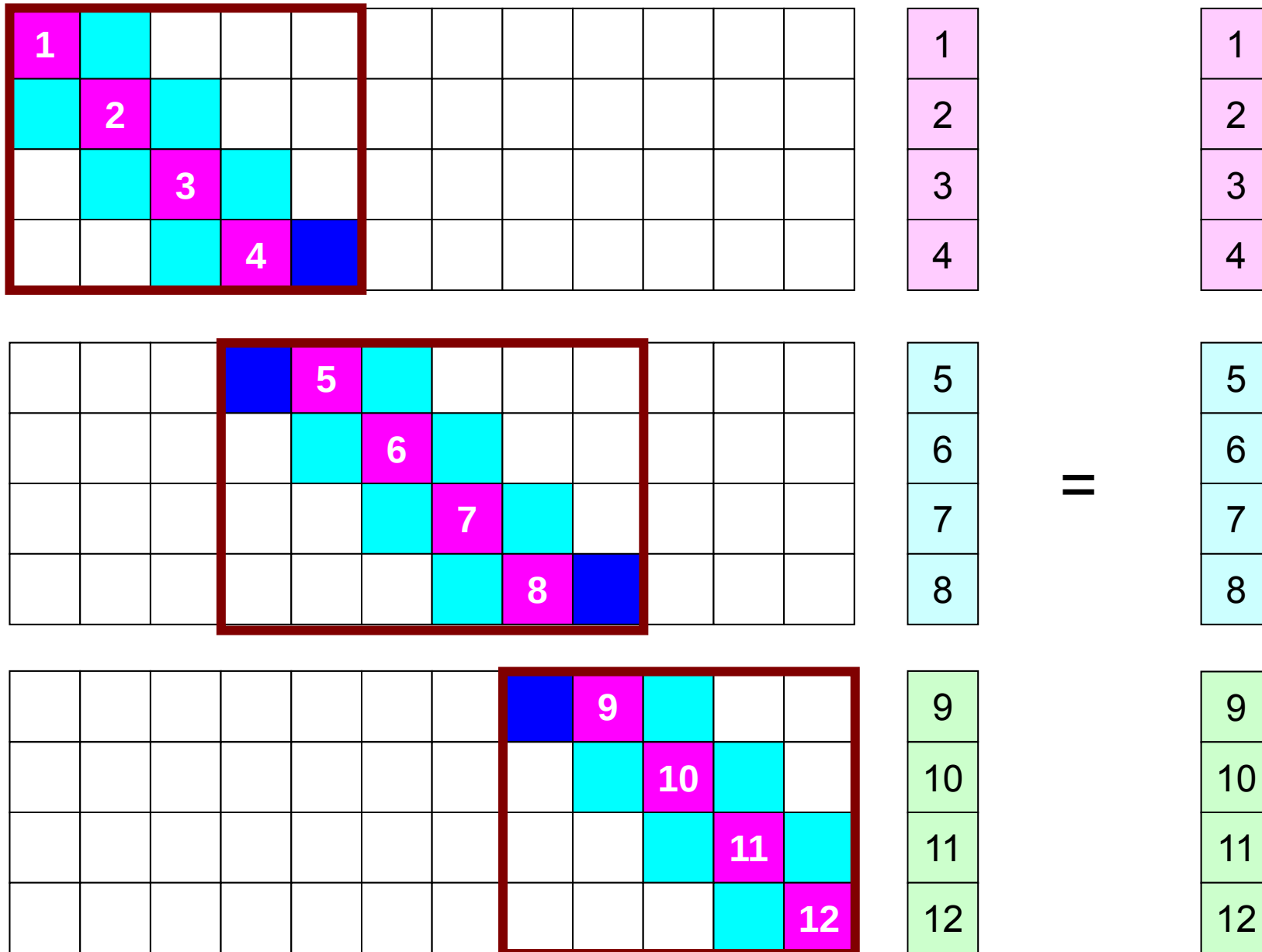
1											
	2										
		3									
			4								
				5							
					6						
						7					
							8				
								9			
									10		
										11	
											12

1
2
3
4
5
6
7
8
9
10
11
12

=

1
2
3
4
5
6
7
8
9
10
11
12

行列ベクトル積: ローカルに計算実施可能



行列ベクトル積: ローカルに計算実施可能

1				
	2			
		3		
			4	

1
2
3
4

1
2
3
4

	5			
		6		
			7	
				8

5
6
7
8

=

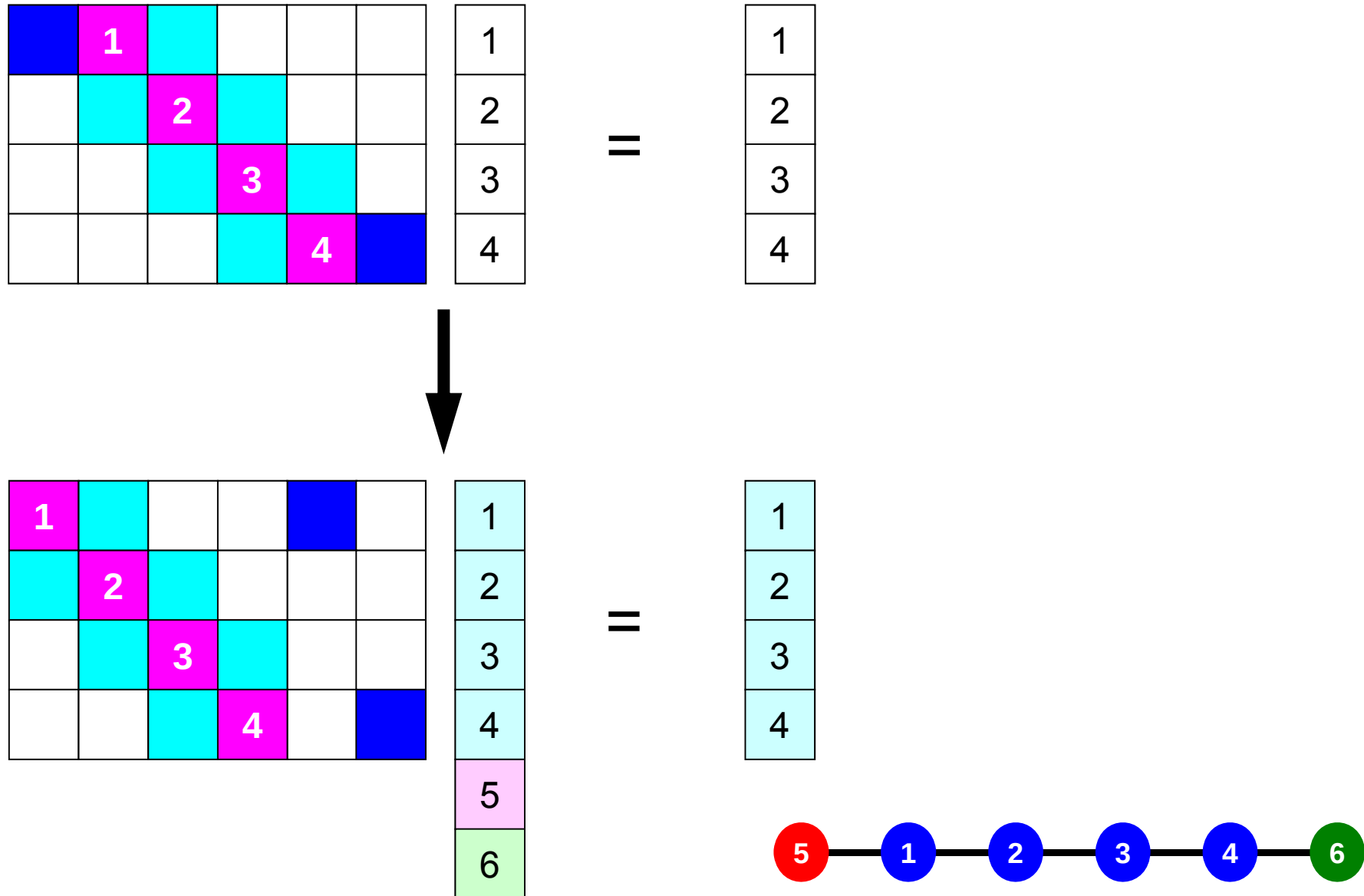
5
6
7
8

	9			
		10		
			11	
				12

9
10
11
12

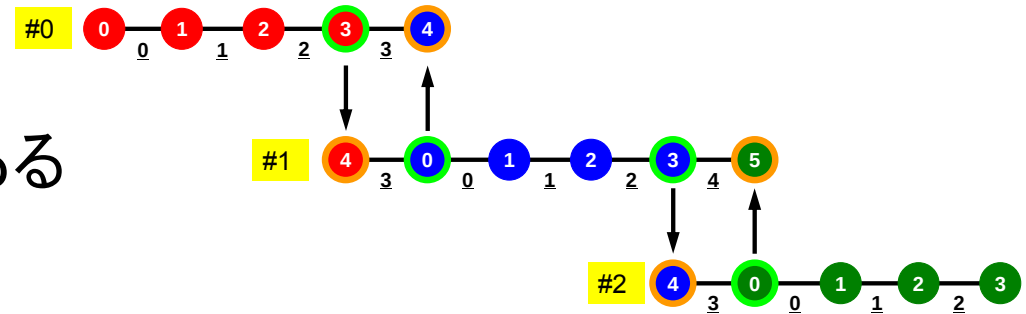
9
10
11
12

行列ベクトル積: ローカル計算 #1



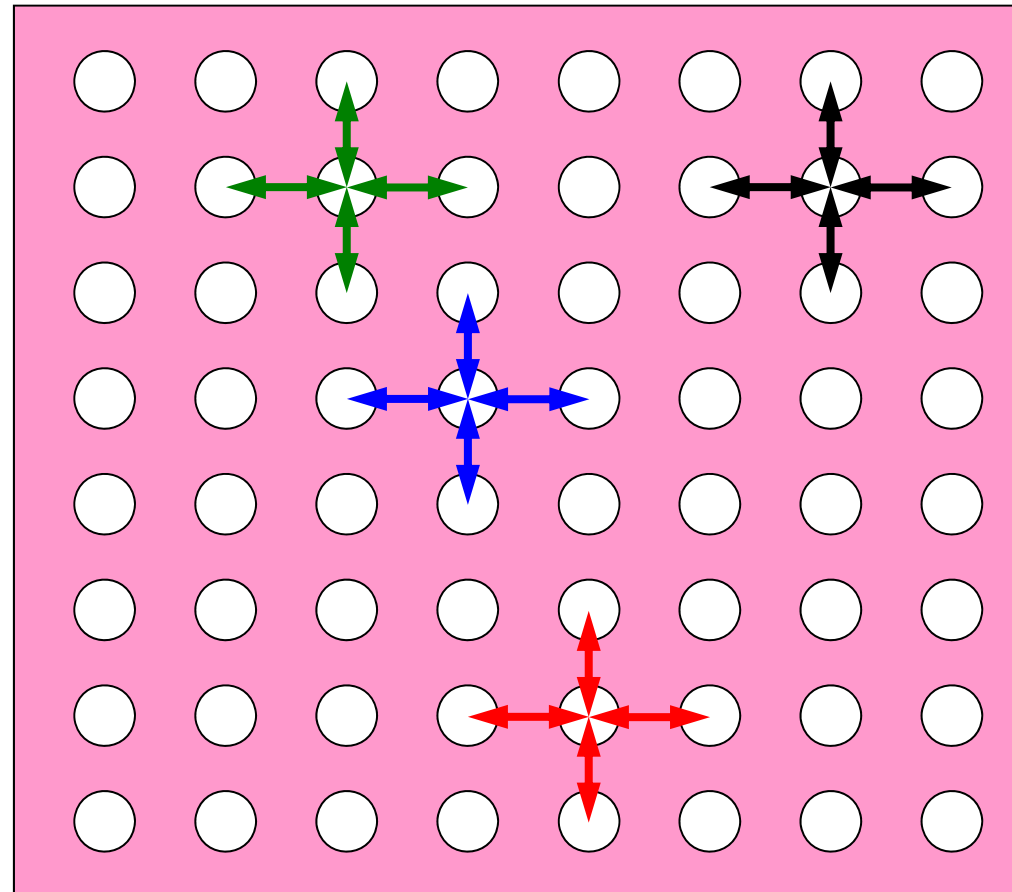
1対1通信とは？

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



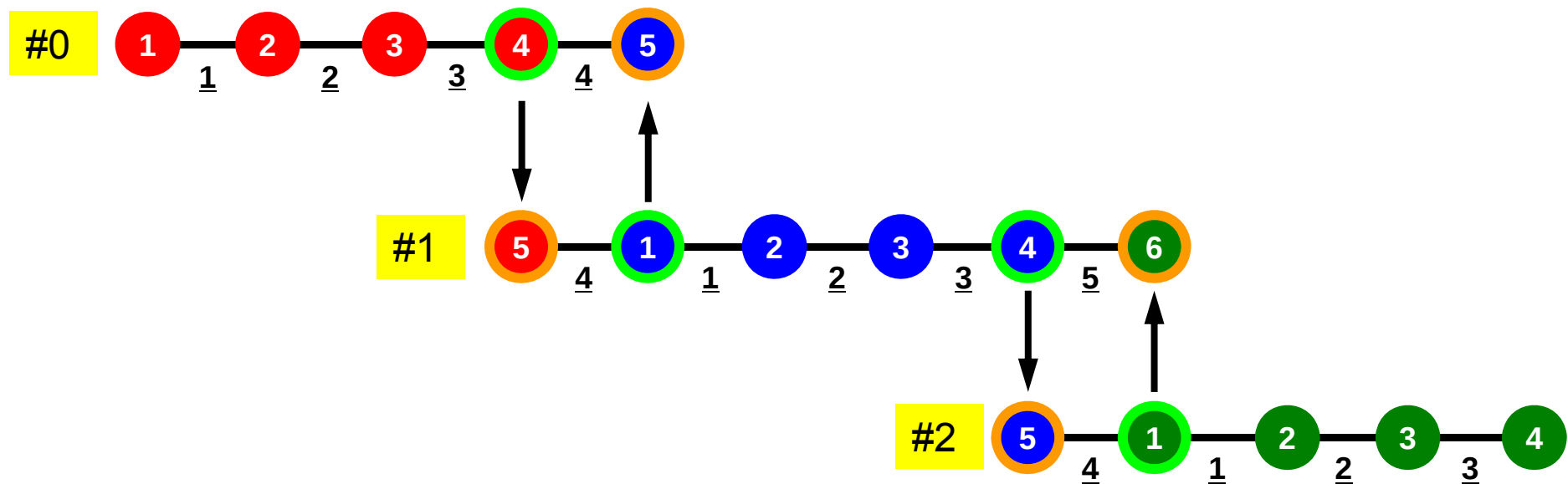
グループ通信, 1対1通信

近接PE(領域)のみとの相互作用
差分法, 有限要素法



1対1通信が必要になる場面: 1DFEM

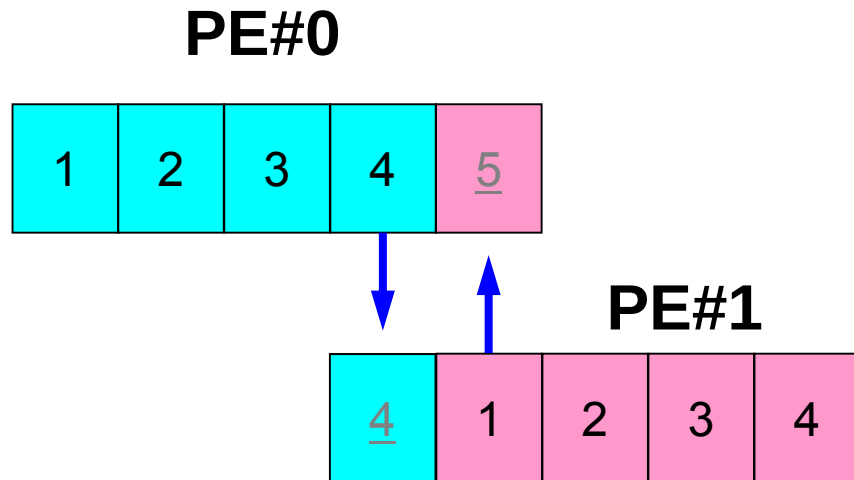
FEMのオペレーションのためには隣接領域の情報が必要
マトリクス生成, 反復法



1対1通信の方法

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

MPI_SEND/MPI_RECV

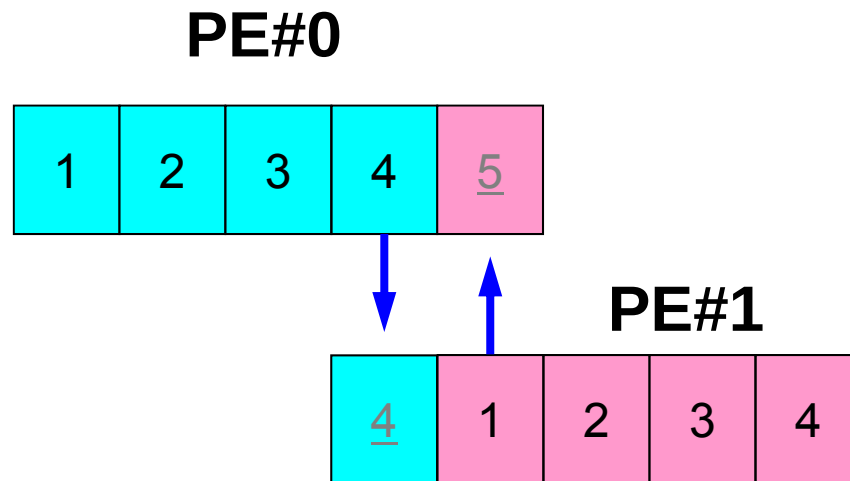


```
if (my_rank.eq.0) NEIB_ID=1
if (my_rank.eq.1) NEIB_ID=0

...
call MPI_SEND (NEIB_ID, arg's)
call MPI_RECV (NEIB_ID, arg's)
...
```

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

MPI_SEND/MPI_RECV(続き)



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

...
if (my_rank.eq.0) then
  call MPI_SEND (NEIB_ID, arg's)
  call MPI_RECV (NEIB_ID, arg's)
endif

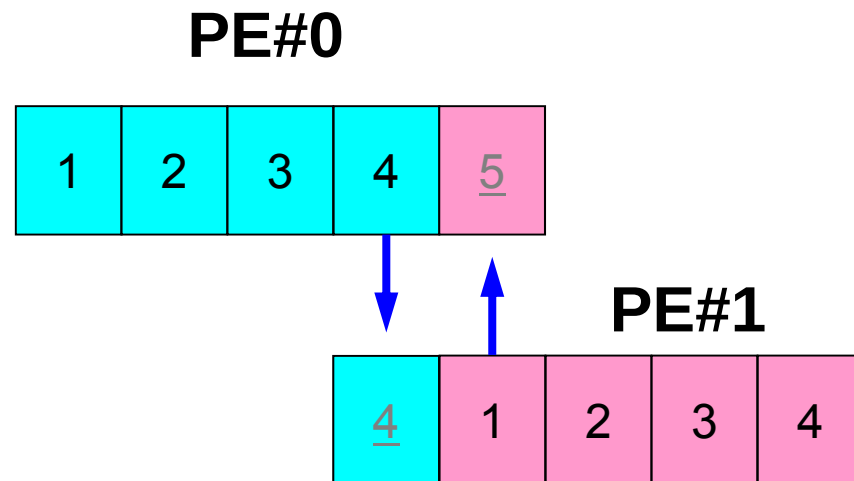
if (my_rank.eq.1) then
  call MPI_RECV (NEIB_ID, arg's)
  call MPI_SEND (NEIB_ID, arg's)
endif

...
```

- このようにすれば, 動く。

1対1通信の方法（実際どうするか）

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



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

...
call MPI_Isend (NEIB_ID, arg's)
call MPI_Irecv (NEIB_ID, arg's)
...
call MPI_Waitall (for Irecv)
...
call MPI_Waitall (for Isend)
```

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

MPI_ISEND

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

- call MPI_ISEND**

(sendbuf, count, datatype, dest, tag, comm, request, ierr)

- | | | | |
|-------------------|----|---|--|
| - <u>sendbuf</u> | 任意 | I | 送信バッファの先頭アドレス, |
| - <u>count</u> | 整数 | I | メッセージのサイズ |
| - <u>datatype</u> | 整数 | I | メッセージのデータタイプ |
| - <u>dest</u> | 整数 | I | 宛先プロセスのアドレス(ランク) |
| - <u>tag</u> | 整数 | I | メッセージタグ, 送信メッセージの種類を区別するときに使用。
通常は「0」でよい。同じメッセージタグ番号同士で通信。 |
| - <u>comm</u> | 整数 | I | コミュニケータを指定する |
| - <u>request</u> | 整数 | 0 | 通信識別子。MPI_WAITALLで使用。
(配列: サイズは同期する必要のある「MPI_ISEND」呼び出し
数(通常は隣接プロセス数など)): C言語については後述 |
| - <u>ierr</u> | 整数 | 0 | 完了コード |

通信識別子 (request handle) : request

- `call MPI_ISEND`

`(sendbuf, count, datatype, dest, tag, comm, request, ierr)`

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

- 以下のような形で宣言しておく(記憶領域を確保するだけで良い: Cについては後述)

```
allocate (request(NEIBPETOT))
```

MPI_IRECV

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

- call MPI_IRECV**

(recvbuf, count, datatype, dest, tag, comm, request, ierr)

- | | | | |
|-------------------|----|---|--|
| - <u>recvbuf</u> | 任意 | I | 受信バッファの先頭アドレス, |
| - <u>count</u> | 整数 | I | メッセージのサイズ |
| - <u>datatype</u> | 整数 | I | メッセージのデータタイプ |
| - <u>dest</u> | 整数 | I | 宛先プロセスのアドレス(ランク) |
| - <u>tag</u> | 整数 | I | メッセージタグ, 受信メッセージの種類を区別するときに使用。
通常は「0」でよい。同じメッセージタグ番号同士で通信。 |
| - <u>comm</u> | 整数 | I | コミュニケータを指定する |
| - <u>request</u> | 整数 | 0 | 通信識別子。MPI_WAITALLで使用。
(配列: サイズは同期する必要がある「MPI_IRECV」呼び出し
数(通常は隣接プロセス数など)): C言語については後述 |
| - <u>ierr</u> | 整数 | 0 | 完了コード |

MPI_WAITALL

- 1対1非ブロッキング通信サブルーチンである「MPI_ISEND」と「MPI_Irecv」を使用した場合、プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_WAITALL」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_WAITALL」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば、「MPI_ISEND」と「MPI_Irecv」を同時に同期してもよい。
 - 「MPI_ISEND/Irecv」で同じ通信識別子を使用すること
- 「MPI_BARRIER」と同じような機能であるが、代用はできない。
 - 実装にもよるが、「request」、「status」の内容が正しく更新されず、何度も「MPI_ISEND/Irecv」を呼び出すと処理が遅くなる、というような経験もある。
- **call MPI_WAITALL (count, request, status, ierr)**
 - **count** 整数 I 同期する必要のある「MPI_ISEND」、「MPI_RECV」呼び出し数。
 - **request** 整数 I/O 通信識別子。「MPI_ISEND」、「MPI_Irecv」で利用した識別子名に対応。(配列サイズ: (count))
 - **status** 整数 0 状況オブジェクト配列(配列サイズ: (MPI_STATUS_SIZE, count))
MPI_STATUS_SIZE: “mpif.h”, “mpi.h”で定められる
パラメータ: C言語については後述
 - **ierr** 整数 0 完了コード

状況オブジェクト配列 (status object) : status

- **call MPI_WAITALL (count, request, status, ierr)**
 - **count** 整数 I 同期する必要のある「MPI_ISEND」, 「MPI_RECV」呼び出し数。
 - **request** 整数 I/O 通信識別子。「MPI_ISEND」, 「MPI_Irecv」で利用した識別子名に対応。(配列サイズ: (count))
 - **status** 整数 0 状況オブジェクト配列(配列サイズ: (MPI_STATUS_SIZE, count))
MPI_STATUS_SIZE: “mpif.h”, “mpi.h”で定められる
パラメータ
 - **ierr** 整数 0 完了コード
- 以下のように予め記憶領域を確保しておくだけでよい(Cについては後述):

```
allocate (stat(MPI_STATUS_SIZE, NEIBPETOT))
```

MPI_SENDRECV

- MPI_SEND+MPI_RECV: 結構制約は多いのでお勧めしない
- **call MPI_SENDRECV**
(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierr)
 - sendbuf 任意 I 送信バッファの先頭アドレス,
 - sendcount 整数 I 送信メッセージのサイズ
 - sendtype 整数 I 送信メッセージのデータタイプ
 - dest 整数 I 宛先プロセスのアドレス(ランク)
 - sendtag 整数 I 送信用メッセージタグ, 送信メッセージの種類を区別するときに使用。
通常は「0」でよい。
 - recvbuf 任意 I 受信バッファの先頭アドレス,
 - recvcount 整数 I 受信メッセージのサイズ
 - recvtype 整数 I 受信メッセージのデータタイプ
 - source 整数 I 送信元プロセスのアドレス(ランク)
 - recvtag 整数 I 受信用メッセージタグ, 送信メッセージの種類を区別するときに使用。
通常は「0」でよい。同じメッセージタグ番号同士で通信。
 - comm 整数 I コミュニケータを指定する
 - status 整数 0 状況オブジェクト配列(配列サイズ: (MPI_STATUS_SIZE))
MPI_STATUS_SIZE: “mpif.h”で定められるパラメータ
C言語については後述
 - ierr 整数 0 完了コード

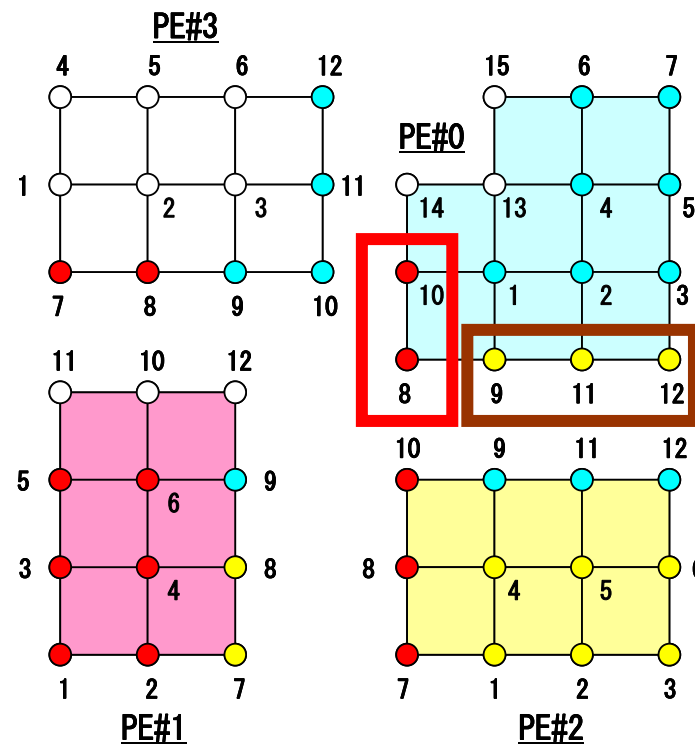
RECV(受信): 外点への受信

受信バッファに隣接プロセスから連続したデータを受け取る

- MPI_Irecv

(recvbuf, count, datatype, dest, tag, comm, request)

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



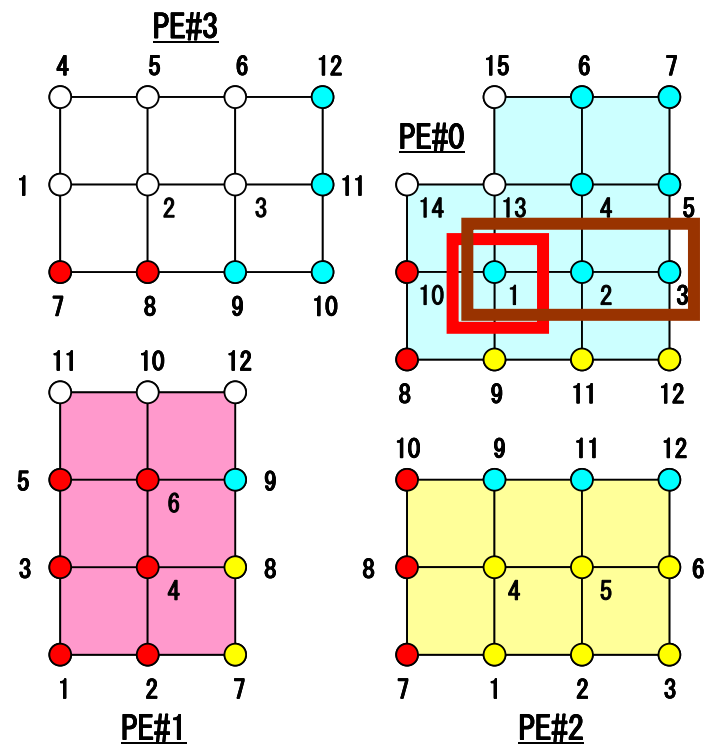
SEND(送信): 境界点の送信

送信バッファの連続したデータを隣接プロセスに送る

- MPI_Isend

(sendbuf, count, datatype, dest, tag, comm, request)

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



通信識別子, 状況オブジェクト配列の定義の仕方 (FORTRAN)

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

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

- **MPI_Sendrecv: status**

```
integer status (MPI_STATUS_SIZE)
```

ファイルコピー・ディレクトリ確認

FORTRANユーザー

```
>$ cd <$0-TOP>
>$ cp /home/z30088/class_eps/F/s2-f.tar .
>$ tar xvf s2-f.tar
```

Cユーザー

```
>$ cd <$0-TOP>
>$ cp /home/z30088/class_eps/C/s2-c.tar .
>$ tar xvf s2-c.tar
```

ディレクトリ確認

```
>$ ls
mpi
```

```
>$ cd mpi/S2
```

このディレクトリを本講義では <\$0-S2> と呼ぶ。

<\$0-S2> = <\$0-TOP>/mpi/S2

利用例(1): スカラー送受信

- PE#0, PE#1間 で8バイト実数VALの値を交換する。

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

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

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

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

受信バッファ名を「VAL」にしても動く場合はあるが、お勧めはしない。

利用例(1): スカラー送受信 FORTRAN

Isend/Irecv/Waitall

```
$> cd <$0-S2>
$> mpifrtpx -Kfast ex1-1.f
$> バッチジョブ実行(2プロセス)
```

```
implicit REAL*8 (A-H,0-Z)
include 'mpif.h'
integer(kind=4) :: my_rank, PETOT, NEIB
real (kind=8) :: VAL, VALtemp
integer(kind=4), dimension(MPI_STATUS_SIZE,1) :: stat_send, stat_recv
integer(kind=4), dimension(1) :: request_send, request_recv

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

if (my_rank.eq.0) then
  NEIB= 1
  VAL = 10.d0
else
  NEIB= 0
  VAL = 11.d0
endif

call MPI_ISEND (VAL, 1, MPI_DOUBLE_PRECISION, NEIB, 0, MPI_COMM_WORLD, request_send(1), ierr)
call MPI_IRecv (VALx, 1, MPI_DOUBLE_PRECISION, NEIB, 0, MPI_COMM_WORLD, request_recv(1), ierr)
call MPI_WAITALL (1, request_recv, stat_recv, ierr)
call MPI_WAITALL (1, request_send, stat_send, ierr)
VAL= VALx

call MPI_FINALIZE (ierr)
end
```

利用例(1): スカラー送受信 FORTRAN

SendRecv

```
$> cd <$0-S2>
$> mpifrtpx -Kfast ex1-2.f
$> バッチジョブ実行(2プロセス)
```

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer(kind=4) :: my_rank, PETOT, NEIB
real (kind=8) :: VAL, VALtemp
integer(kind=4) :: status(MPI_STATUS_SIZE)

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

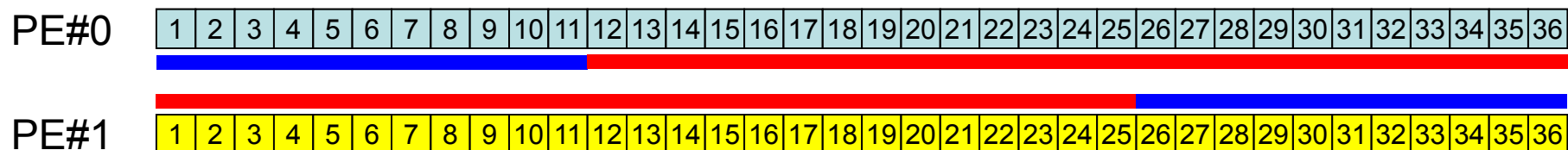
if (my_rank.eq.0) then
  NEIB= 1
  VAL = 10.d0
endif
if (my_rank.eq.1) then
  NEIB= 0
  VAL = 11.d0
endif

call MPI_SENDRECV
& (VAL, 1, MPI_DOUBLE_PRECISION, NEIB, 0, &
& VALtemp, 1, MPI_DOUBLE_PRECISION, NEIB, 0, MPI_COMM_WORLD, status, ierr)

VAL= VALtemp
call MPI_FINALIZE (ierr)
end
```

利用例(2): 配列の送受信(1/4)

- PE#0, PE#1間 で8バイト実数配列VECの値を交換する。
- PE#0⇒PE#1
 - PE#0: VEC(1)～VEC(11)の値を送る(長さ:11)
 - PE#1: VEV(26)～VEC(36)の値として受け取る
- PE#1⇒PE#0
 - PE#1: VEC(1)～VEC(25)の値を送る(長さ:25)
 - PE#0: VEV(12)～VEC(36)の値として受け取る
- 演習: プログラムを作成して見よう!



演習t1

演 習

- VEC(:)の初期状態を以下のようにする:
 - PE#0 VEC(1-36)= 101,102,103,~,135,136
 - PE#1 VEC(1-36)= 201,202,203,~,235,236
- 次ページのような結果になることを確認せよ
- 以下のそれぞれを使用したプログラムを作成せよ
 - MPI_Isend/Irecv/Waitall
 - MPI_Sendrecv

予測される結果

演習t1

```

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

```

```

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

```

```

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

```

```

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

```

利用例(2): 配列の送受信(2/4)

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

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

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

これでも良いが、操作が煩雑
SPMDらしくない
汎用性がない

利用例(2): 配列の送受信(3/4)

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

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

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

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

一気にSPMDらしくなる

利用例(2): 配列の送受信(4/4)

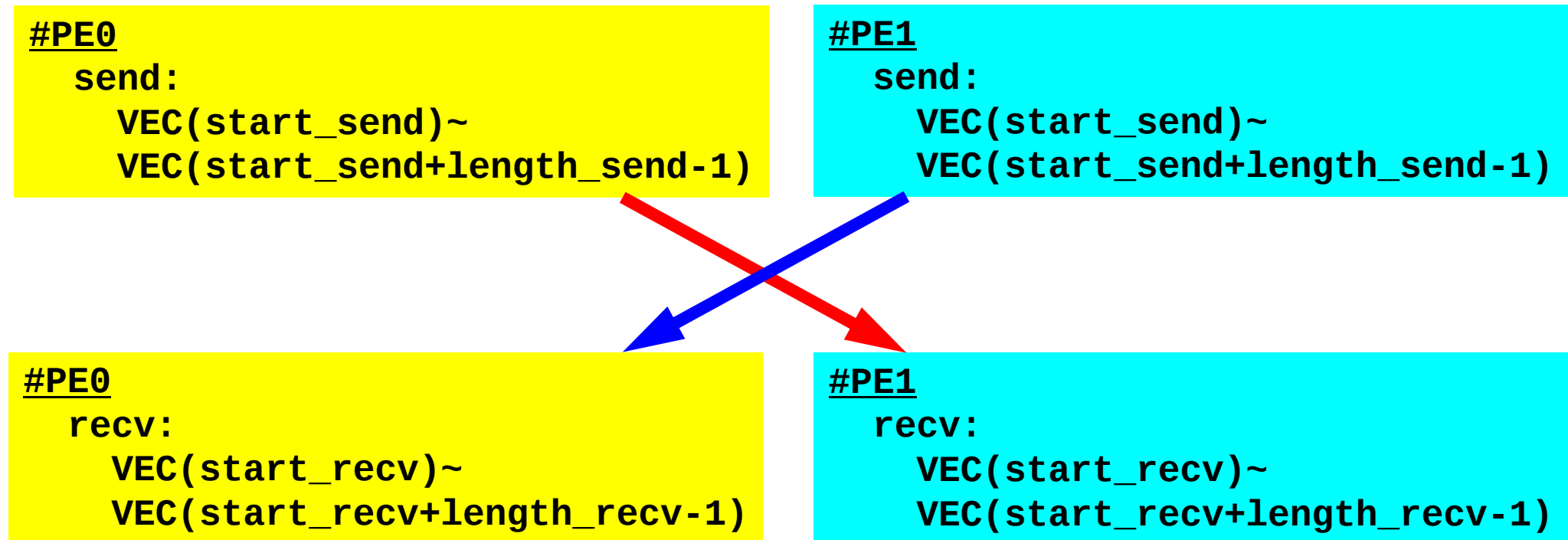
```
if (my_rank.eq.0) then
  NEIB= 1
  start_send= 1
  length_send= 11
  start_recv= length_send + 1
  length_recv= 25
endif

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

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

演習t1

配列の送受信:注意



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

1対1通信

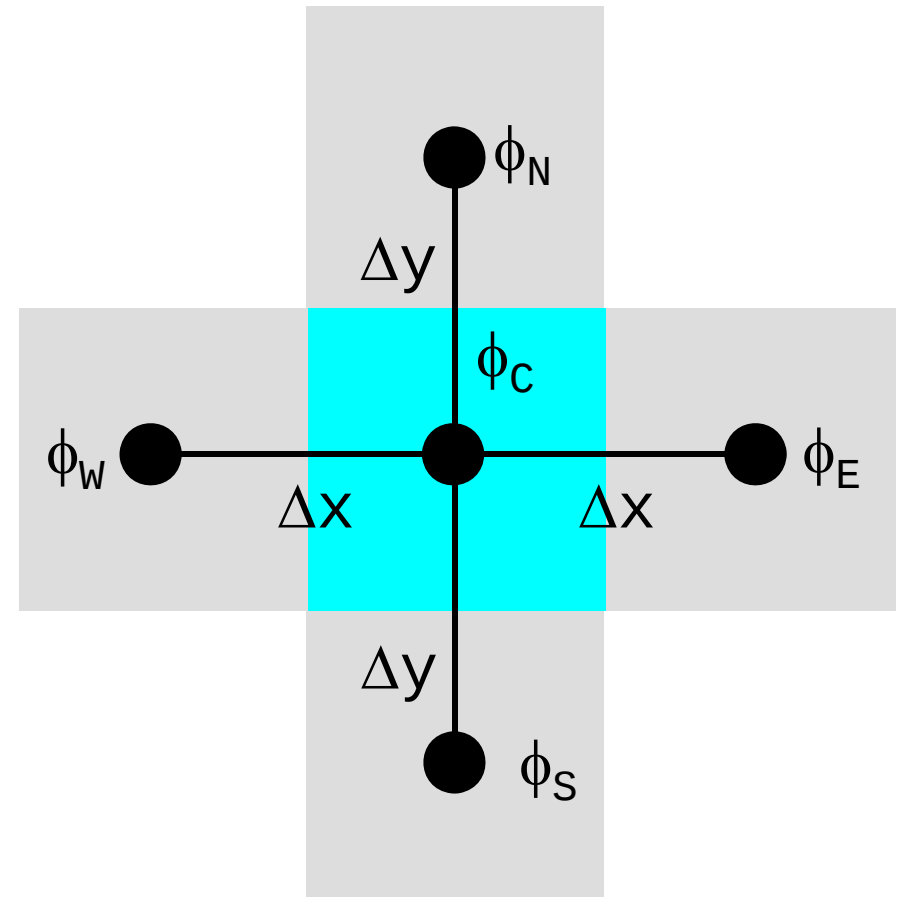
- 1対1通信とは ?
- 二次元問題, 一般化された通信テーブル
 - 二次元差分法
 - 問題設定
 - 局所データ構造と通信テーブル
 - 実装例
- 課題S2

[illegible]

二次元中央差分法(5点差分法)の定式化

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f$$

$$\left(\frac{\phi_E - 2\phi_C + \phi_W}{\Delta x^2} \right) + \left(\frac{\phi_N - 2\phi_C + \phi_S}{\Delta y^2} \right) = f_C$$



4領域に分割

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>	<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>	<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>	<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>	<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

4領域に分割: 全体番号

PE#3

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>

PE#2

<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>

PE#0

<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>

PE#1

<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

4領域に分割: 局所番号

PE#3

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

PE#2

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

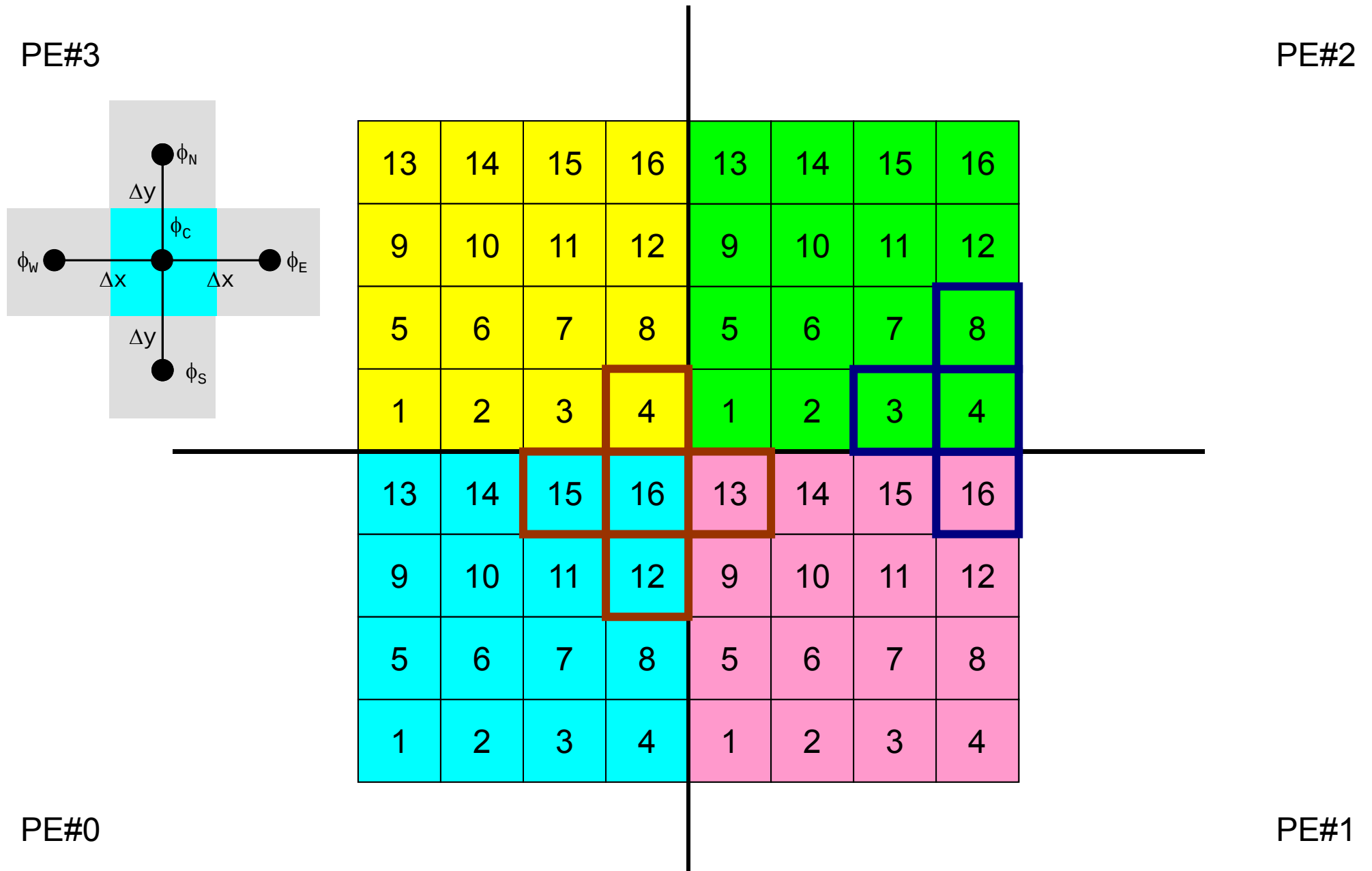
PE#0

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

PE#1

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

オーバーラップ領域の値が必要: 外点



オーバーラップ領域の値が必要: 外点

PE#3

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

PE#2

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

PE#0

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

PE#1

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

外点の局所番号はどうする？

PE#3

13	14	15	16	?
9	10	11	12	?
5	6	7	8	?
1	2	3	4	?
?	?	?	?	

PE#2

?	13	14	15	16
?	9	10	11	12
?	5	6	7	8
?	1	2	3	4
	?	?	?	?

PE#0

?	?	?	?	
13	14	15	16	?
9	10	11	12	?
5	6	7	8	?
1	2	3	4	?

PE#1

	?	?	?	?
?	13	14	15	16
?	9	10	11	12
?	5	6	7	8
?	1	2	3	4

オーバーラップ領域の値が必要

PE#3

13	14	15	16	?
9	10	11	12	?
5	6	7	8	?
1	2	3	4	?

?	?	?	?
---	---	---	---

PE#0

?	?	?	?	
13	14	15	16	?
9	10	11	12	?
5	6	7	8	?
1	2	3	4	?

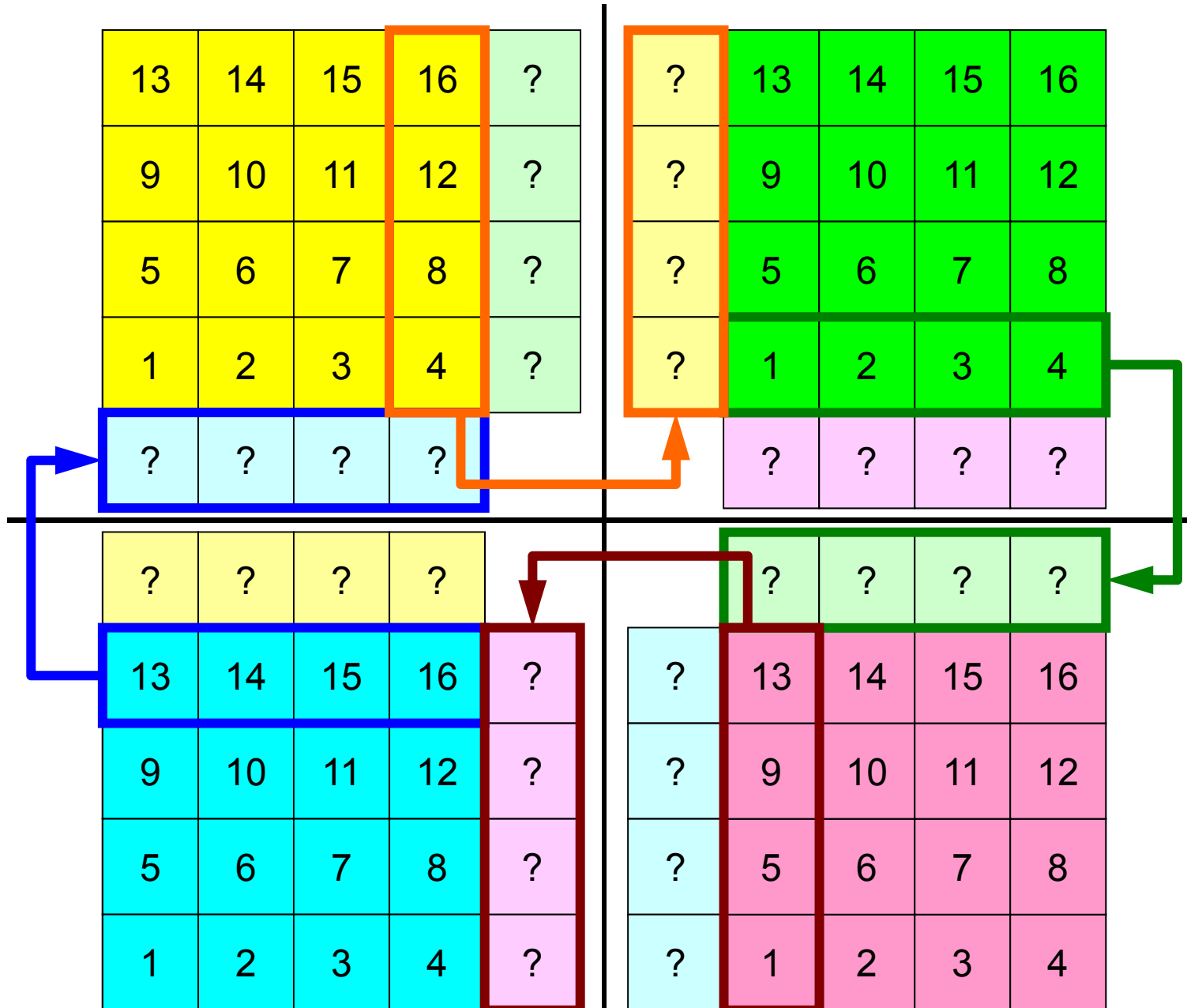
PE#2

?	13	14	15	16
?	9	10	11	12
?	5	6	7	8
?	1	2	3	4

?	?	?	?
---	---	---	---

PE#1

		?	?	?	?
?	13	14	15	16	
?	9	10	11	12	
?	5	6	7	8	
?	1	2	3	4	



オーバーラップ領域の値が必要

PE#3

13	14	15	16	?
9	10	11	12	?
5	6	7	8	?
1	2	3	4	?
?	?	?	?	

PE#2

?	13	14	15	16
?	9	10	11	12
?	5	6	7	8
?	1	2	3	4
	?	?	?	?

PE#0

?	?	?	?	
13	14	15	16	?
9	10	11	12	?
5	6	7	8	?
1	2	3	4	?

PE#1

	?	?	?	?
?	13	14	15	16
?	9	10	11	12
?	5	6	7	8
?	1	2	3	4

1対1通信

- 1対1通信とは ?
- 二次元問題, 一般化された通信テーブル
 - 二次元差分法
 - 問題設定
 - 局所データ構造と通信テーブル
 - 実装例
- 課題S2

問題設定：全体データ

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>	<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>	<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>	<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>	<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

- $8 \times 8 = 64$ 要素に分割された二次元領域を考える。
- 各要素には1～64までの全体要素番号が振られている。
 - 簡単のため、この「全体要素番号」を各要素における従属変数値(温度のようなもの)とする
 - $\Rightarrow$ 「計算結果」のようなもの

問題設定：局所分散データ

PE#2

57	58	59	60
49	50	51	52
41	42	43	44
33	34	35	36

PE#3

61	62	63	64
53	54	55	56
45	46	47	48
37	38	39	40

- 左記のような4領域に分割された二次元領域において、外点の情報(全体要素番号)を隣接領域から受信する方法

— □ はPE#0が受信する情報

25	26	27	28
17	18	19	20
9	10	11	12
1	2	3	4

29	30	31	32
21	22	23	24
13	14	15	16
5	6	7	8

PE#0

PE#1

PE#2

57	58	59	60	
49	50	51	52	
41	42	43	44	
33	34	35	36	

PE#3

	61	62	63	64
	53	54	55	56
	45	46	47	48
	37	38	39	40

25	26	27	28	
17	18	19	20	
9	10	11	12	
1	2	3	4	

PE#0

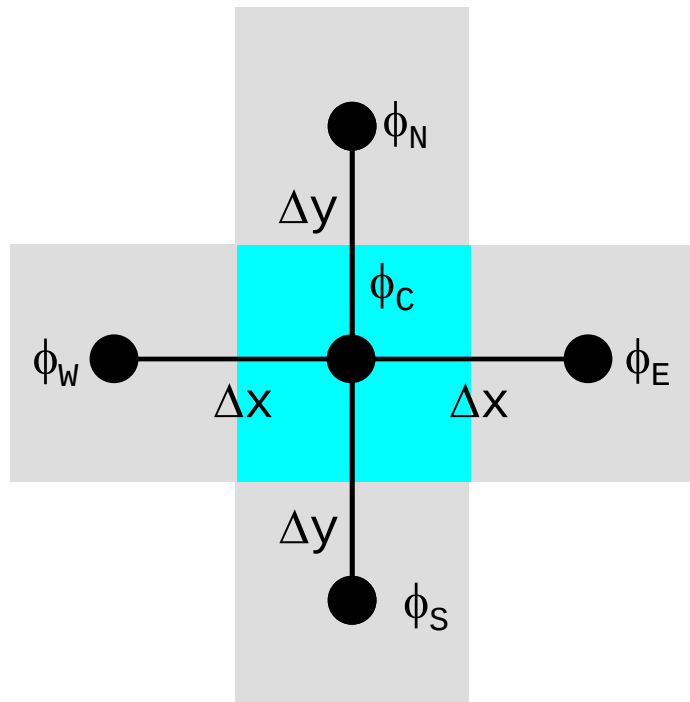
	29	30	31	32
	21	22	23	24
	13	14	15	16
	5	6	7	8

PE#1

二次元差分法のオペレーション

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f$$

$$\left(\frac{\phi_E - 2\phi_C + \phi_W}{\Delta x^2} \right) + \left(\frac{\phi_N - 2\phi_C + \phi_S}{\Delta y^2} \right) = f_C$$

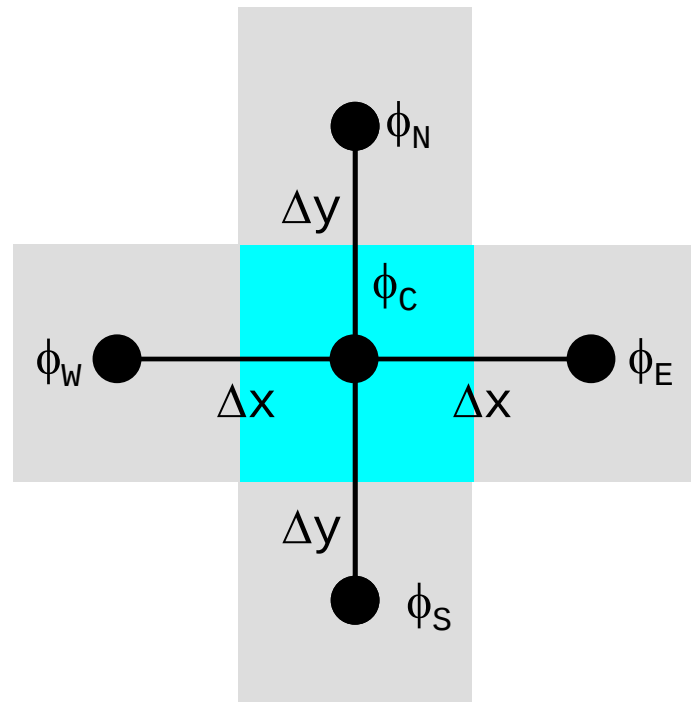


<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>	<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>	<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>	<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>	<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

二次元差分法のオペレーション

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f$$

$$\left(\frac{\phi_E - 2\phi_C + \phi_W}{\Delta x^2} \right) + \left(\frac{\phi_N - 2\phi_C + \phi_S}{\Delta y^2} \right) = f_C$$



57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

演算内容(1/3)

<u>PE#2</u>	<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>	<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>	<u>PE#3</u>
	<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>	<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>	
	<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>	<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>	
	<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>	<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>	
	<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>	
	<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>	
	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>	
<u>PE#0</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>PE#1</u>

- 各PEの内点 ($i=1 \sim N (=16)$) において局所データを読み込み, 「境界点」のデータを各隣接領域における「外点」として配信

演算内容(2/3):送信,受信前

PE#2

1: <u>33</u>	9: <u>49</u>	17: <u>?</u>
2: <u>34</u>	10: <u>50</u>	18: <u>?</u>
3: <u>35</u>	11: <u>51</u>	19: <u>?</u>
4: <u>36</u>	12: <u>52</u>	20: <u>?</u>
5: <u>41</u>	13: <u>57</u>	21: <u>?</u>
6: <u>42</u>	14: <u>58</u>	22: <u>?</u>
7: <u>43</u>	15: <u>59</u>	23: <u>?</u>
8: <u>44</u>	16: <u>60</u>	24: <u>?</u>

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>	
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>	
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>	
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>	

PE#3

1: <u>37</u>	9: <u>53</u>	17: <u>?</u>
2: <u>38</u>	10: <u>54</u>	18: <u>?</u>
3: <u>39</u>	11: <u>55</u>	19: <u>?</u>
4: <u>40</u>	12: <u>56</u>	20: <u>?</u>
5: <u>45</u>	13: <u>61</u>	21: <u>?</u>
6: <u>46</u>	14: <u>62</u>	22: <u>?</u>
7: <u>47</u>	15: <u>63</u>	23: <u>?</u>
8: <u>48</u>	16: <u>64</u>	24: <u>?</u>

	<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
	<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
	<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
	<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>

1: <u>1</u>	9: <u>17</u>	17: <u>?</u>
2: <u>2</u>	10: <u>18</u>	18: <u>?</u>
3: <u>3</u>	11: <u>19</u>	19: <u>?</u>
4: <u>4</u>	12: <u>20</u>	20: <u>?</u>
5: <u>9</u>	13: <u>25</u>	21: <u>?</u>
6: <u>10</u>	14: <u>26</u>	22: <u>?</u>
7: <u>11</u>	15: <u>27</u>	23: <u>?</u>
8: <u>12</u>	16: <u>28</u>	24: <u>?</u>

<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	

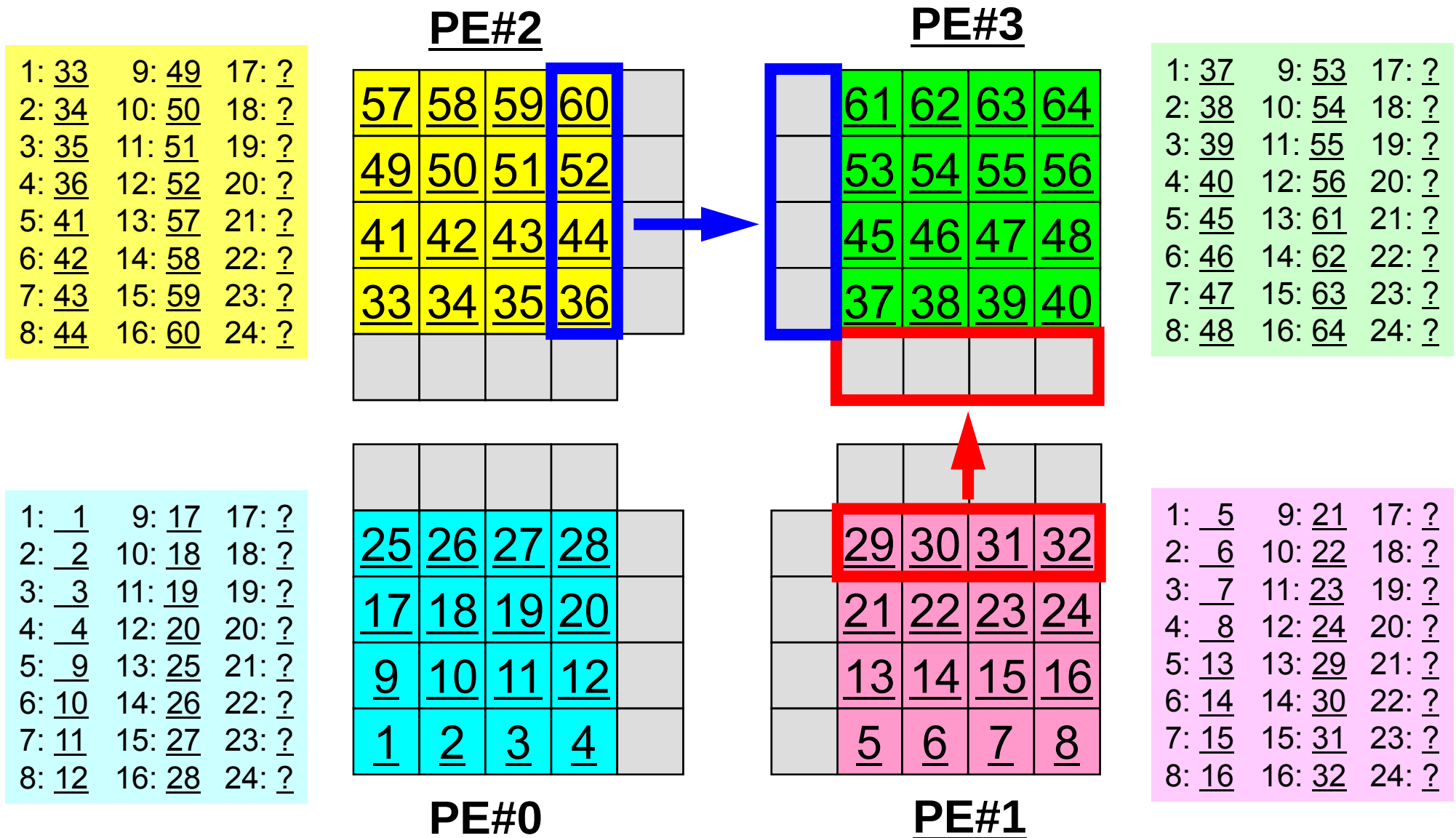
PE#0

PE#1

	<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

1: <u>5</u>	9: <u>21</u>	17: <u>?</u>
2: <u>6</u>	10: <u>22</u>	18: <u>?</u>
3: <u>7</u>	11: <u>23</u>	19: <u>?</u>
4: <u>8</u>	12: <u>24</u>	20: <u>?</u>
5: <u>13</u>	13: <u>29</u>	21: <u>?</u>
6: <u>14</u>	14: <u>30</u>	22: <u>?</u>
7: <u>15</u>	15: <u>31</u>	23: <u>?</u>
8: <u>16</u>	16: <u>32</u>	24: <u>?</u>

演算内容(2/3):送信, 受信前



演算内容(3/3):送信,受信後

PE#2

1: 33 9: 49 17: 37
 2: 34 10: 50 18: 45
 3: 35 11: 51 19: 53
 4: 36 12: 52 20: 61
 5: 41 13: 57 21: 25
 6: 42 14: 58 22: 26
 7: 43 15: 59 23: 27
 8: 44 16: 60 24: 28

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>	<u>61</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>	<u>53</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>	<u>45</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>	<u>37</u>
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	

PE#3

1: 37 9: 53 17: 36
 2: 38 10: 54 18: 44
 3: 39 11: 55 19: 52
 4: 40 12: 56 20: 60
 5: 45 13: 61 21: 29
 6: 46 14: 62 22: 30
 7: 47 15: 63 23: 31
 8: 48 16: 64 24: 32

<u>60</u>	<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>52</u>	<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>44</u>	<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>36</u>	<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>
	<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>

PE#0

1: 1 9: 17 17: 5
 2: 2 10: 18 18: 14
 3: 3 11: 19 19: 21
 4: 4 12: 20 20: 29
 5: 9 13: 25 21: 33
 6: 10 14: 26 22: 34
 7: 11 15: 27 23: 35
 8: 12 16: 28 24: 36

<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>	
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>

PE#1

	<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>
<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

1: 5 9: 21 17: 4
 2: 6 10: 22 18: 12
 3: 7 11: 23 19: 20
 4: 8 12: 24 20: 28
 5: 13 13: 29 21: 37
 6: 14 14: 30 22: 38
 7: 15 15: 31 23: 39
 8: 16 16: 32 24: 40

1対1通信

- 1対1通信とは ?
- 二次元問題, 一般化された通信テーブル
 - 二次元差分法
 - 問題設定
 - 局所データ構造と通信テーブル
 - 実装例
- 課題S2

各領域データ(局所分散データ)仕様

PE#0における局所分散データ

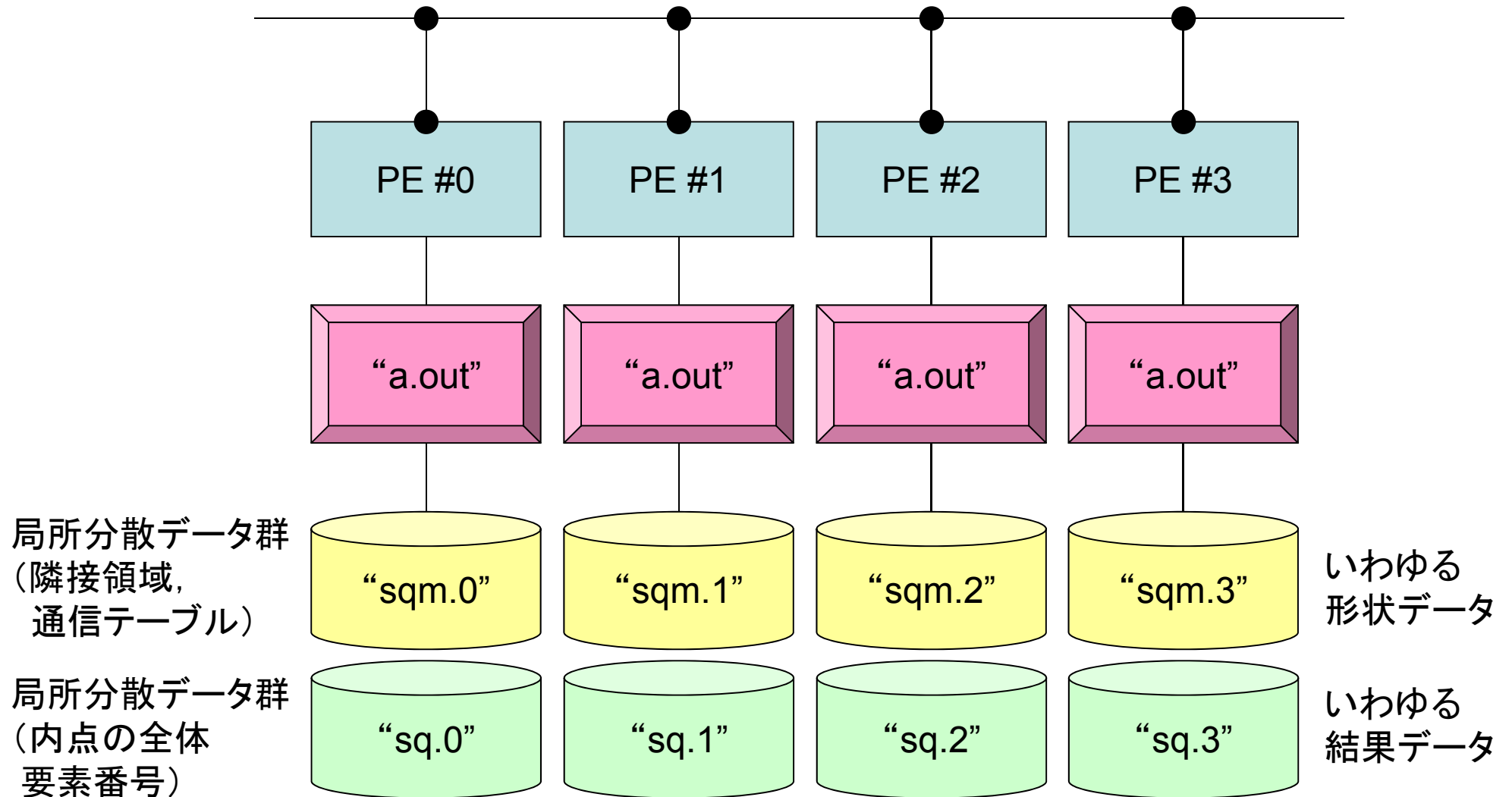
<u>PE#2</u>				
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	
<u>PE#0</u>				<u>PE#1</u>

<u>PE#2</u>				
13	14	15	16	
9	10	11	12	
5	6	7	8	
1	2	3	4	
<u>PE#0</u>				<u>PE#1</u>

各要素における値(全体番号)

局所番号

SPMD...



二次元差分法: PE#0

各領域に必要な情報(1/4)

内点 (Internal Points)

その領域にアサインされた要素

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

二次元差分法: PE#0

各領域に必要な情報(2/4)

PE#3

●	●	●	●	
13	14	15	16	●
9	10	11	12	●
5	6	7	8	●
1	2	3	4	●

PE#1

内点 (Internal Points)

その領域にアサインされた要素

外点 (External Points)

他の領域にアサインされた要素であるがその領域の計算を実施するのに必要な要素
(オーバーラップ領域の要素)

- ・袖領域
- ・Halo (後光, 光輪, (太陽・月の) 暈 (かさ), 暈輪 (うんりん))



二次元差分法: PE#0

各領域に必要な情報(4/4)

PE#3

13	14	15	16	
9	10	11	12	
5	6	7	8	
1	2	3	4	

PE#1

内点 (Internal Points)

その領域にアサインされた要素

外点 (External Points)

他の領域にアサインされた要素であるがその領域の計算を実施するのに必要な要素
(オーバーラップ領域の要素)

境界点 (Boundary Points)

内点のうち、他の領域の外点となっている要素
他の領域の計算に使用される要素

二次元差分法: PE#0

各領域に必要な情報(4/4)

PE#3

13	14	15	16	
9	10	11	12	
5	6	7	8	
1	2	3	4	

PE#1

内点 (Internal Points)

その領域にアサインされた要素

外点 (External Points)

他の領域にアサインされた要素であるがその領域の計算を実施するのに必要な要素
(オーバーラップ領域の要素)

境界点 (Boundary Points)

内点のうち、他の領域の外点となっている要素
他の領域の計算に使用される要素

領域間相互の関係

通信テーブル: 外点, 境界点の関係
隣接領域

各領域データ(局所データ)仕様

21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

- 内点, 外点
 - 内点～外点となるように局所番号をつける
- 隣接領域情報
 - オーバーラップ要素を共有する領域
 - 隣接領域数, 番号
- 外点情報
 - どの領域から, 何個の, どの外点の情報を「受信:import」するか
- 境界点情報
 - 何個の, どの境界点の情報を, どの領域に「送信:export」するか

各領域データ(局所分散データ)仕様

PE#0における局所分散データ

<u>PE#2</u>				
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	
<u>PE#0</u>				<u>PE#1</u>

<u>PE#2</u>				
21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17
<u>PE#0</u>				<u>PE#1</u>

各要素における値(全体番号)

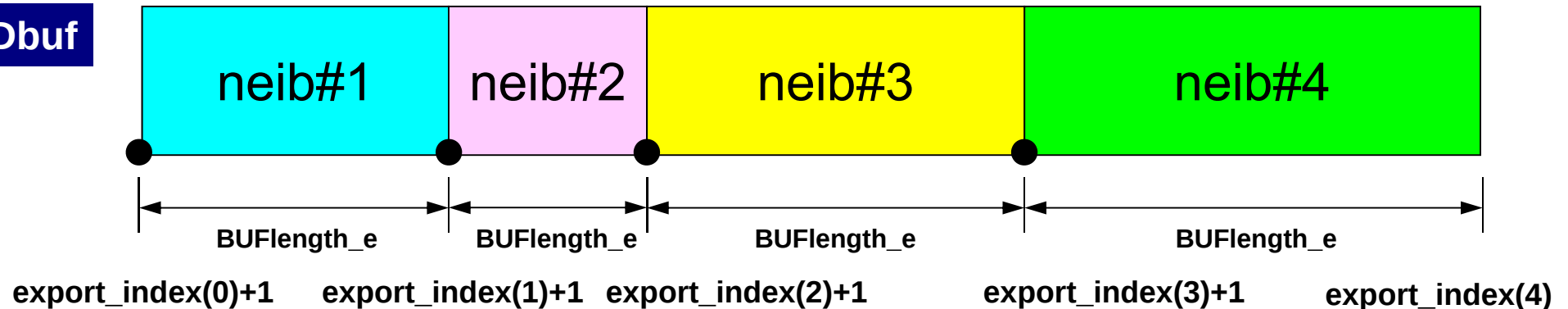
局所番号

一般化された通信テーブル: 送信

- 送信相手
 - NEIBPETOT, NEIBPE(neib)
- それぞれの送信相手に送るメッセージサイズ
 - export_index(neib), neib= 0, NEIBPETOT
- 「境界点」番号
 - export_item(k), k= 1, export_index(NEIBPETOT)
- それぞれの送信相手に送るメッセージ
 - SENDbuf(k), k= 1, export_index(NEIBPETOT)

送信 (MPI_Isend/Irecv/Waitall) Fortran

SENDbuf



```

do neib= 1, NEIBPETOT
  do k= export_index(neib-1)+1, export_index(neib)
    kk= export_item(k)
    SENDbuf(k)= VAL(kk)
  enddo
enddo

do neib= 1, NEIBPETOT
  iS_e= export_index(neib-1) + 1
  iE_e= export_index(neib)
  BUFlength_e= iE_e + 1 - iS_e

  call MPI_ISEND
&      (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&      MPI_COMM_WORLD, request_send(neib), ierr)
  enddo

call MPI_WAITALL (NEIBPETOT, request_send, stat_recv, ierr)

```

送信バッファへの代入
温度などの変数を直接送信, 受信に使うのではなく, このようなバッファへ一回代入して計算することを勧める。

一般化された通信テーブル: 受信

- 受信相手
 - NEIBPETOT, NEIBPE(neib)
- それぞれの受信相手から受け取るメッセージサイズ
 - import_index(neib), neib= 0, NEIBPETOT
- 「外点」番号
 - import_item(k), k= 1, import_index(NEIBPETOT)
- それぞれの受信相手から受け取るメッセージ
 - RECVbuf(k), k= 1, import_index(NEIBPETOT)

受信 (MPI_Isend/Irecv/Waitall)

Fortran

```

do neib= 1, NEIBPETOT
  iS_i= import_index(neib-1) + 1
  iE_i= import_index(neib )
  BUFlength_i= iE_i + 1 - iS_i

  call MPI_Irecv
&      (RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&      MPI_COMM_WORLD, request_recv(neib), ierr)
enddo

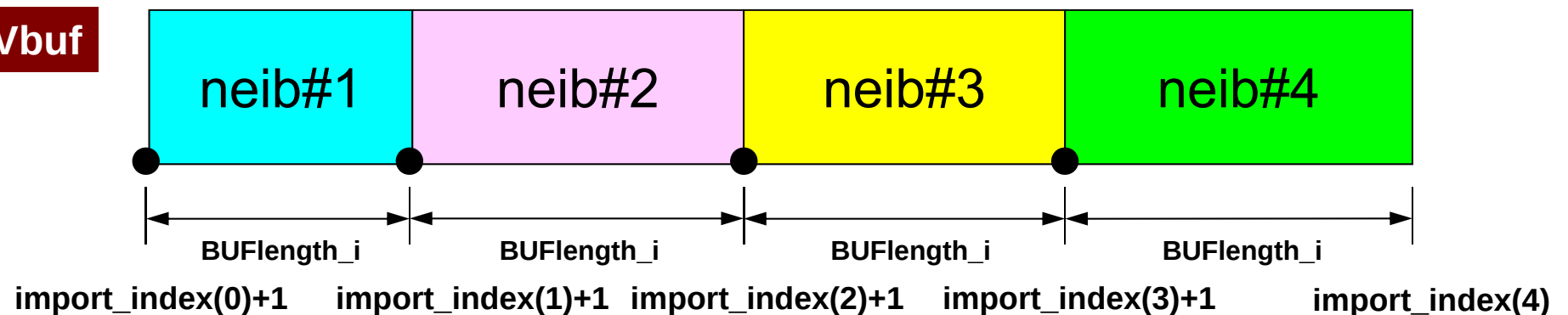
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

do neib= 1, NEIBPETOT
  do k= import_index(neib-1)+1, import_index(neib)
    kk= import_item(k)
    VAL(kk)= RECVbuf(k)
  enddo
enddo

```

受信バッファから代入

RECVbuf



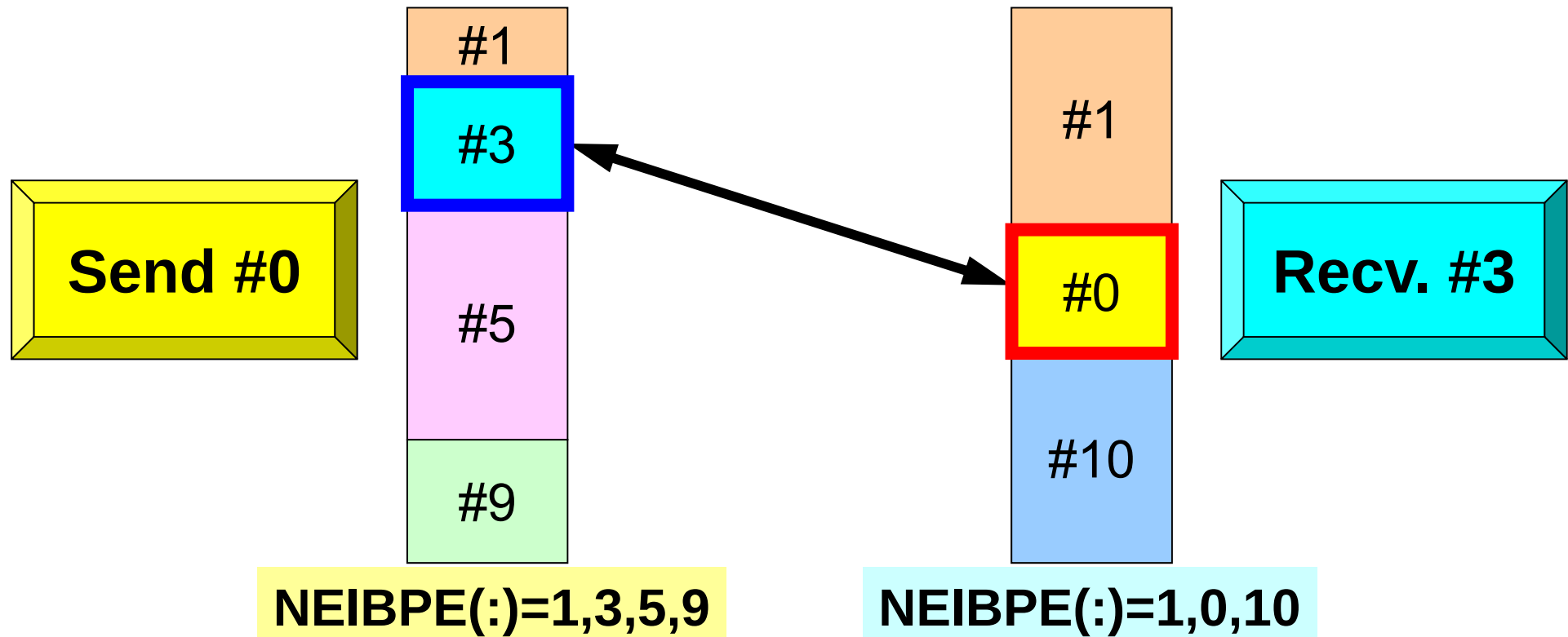
送信と受信の関係

```
do neib= 1, NEIBPETOT  
  iS_e= export_index(neib-1) + 1  
  iE_e= export_index(neib )  
  BUFlength_e= iE_e + 1 - iS_e  
  
  call MPI_ISEND  
&      (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&  
&      MPI_COMM_WORLD, request_send(neib), ierr)  
enddo
```

```
do neib= 1, NEIBPETOT  
  iS_i= import_index(neib-1) + 1  
  iE_i= import_index(neib )  
  BUFlength_i= iE_i + 1 - iS_i  
  
  call MPI_Irecv  
&      (RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&  
&      MPI_COMM_WORLD, request_recv(neib), ierr)  
enddo
```

- 送信元・受信先プロセス番号, メッセージサイズ, 内容の整合性 !
- NEIBPE(neib)がマッチしたときに通信が起こる。

送信と受信の関係 (#0⇒#3)



- 送信元・受信先プロセス番号, メッセージサイズ, 内容の整合性 !
- NEIBPE(neib)がマッチしたときに通信が起こる。

一般化された通信テーブル(1/6)

PE#3

21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

PE#1

```
#NEIBPEtot
2
#NEIBPE
1 3
#NODE
24 16
#IMPORT_index
4 8
#IMPORT_items
17
18
19
20
21
22
23
24
#EXPORT_index
4 8
#EXPORT_items
4
8
12
16
13
14
15
16
```

一般化された通信テーブル(2/6)

PE#3

21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

PE#1

```

#NEIBPEtot 隣接領域数
2
#NEIBPE 隣接領域番号
13
#NODE
24 16 内点+外点, 内点数
#IMPORT_index
4 8
#IMPORT_items
17
18
19
20
21
22
23
24
#EXPORT_index
4 8
#EXPORT_items
4
8
12
16
13
14
15
16

```

一般化された通信テーブル(3/6)

PE#3

21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

PE#1

```
#NEIBPEtot
2
#NEIBPE
1 3
#NODE
24 16
#IMPORT_index
4 8
#IMPORT_items
17
18
19
20
21
22
23
24
#EXPORT_index
4 8
#EXPORT_items
4
8
12
16
13
14
15
16
```

隣接領域1(#1)から4つ(1~4),
隣接領域2(#3)から4つ(5~8)が
「import(受信)」されることを示
す。

一般化された通信テーブル(4/6)

PE#3

21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

PE#1

#NEIBPEtot

2

#NEIBPE

1 3

#NODE

24 16

#IMPORT_index

4 8

#IMPORT_items

17

18 隣接領域1(#1)から

19 「import」する要素(1~4)

20

21 隣接領域2(#3)から

22 「import」する要素(5~8)

23

24

#EXPORT_index

4 8

#EXPORT_items

4

8

12

16

13

14

15

16

一般化された通信テーブル(5/6)

PE#3

21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

PE#1

```

#NEIBPetot
2
#NEIBPE
1 3
#NODE
24 16
#IMPORT_index
4 8
#IMPORT_items
17
18
19
20
21
22
23
24
#EXPORT_index
4 8
#EXPORT_items
4
8
12
16
13
14
15
16

```

隣接領域1(#1)へ4つ(1~4),
隣接領域2(#3)へ4つ(5~8)が
「export(送信)」されることを示す。

一般化された通信テーブル(6/6)

PE#3

21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

PE#1

#NEIBPEtot

2

#NEIBPE

1 3

#NODE

24 16

#IMPORT_index

4 8

#IMPORT_items

17

18

19

20

21

22

23

24

#EXPORT_index

4 8

#EXPORT_items

4

8

12

16

13

14

15

16

隣接領域1(#1)へ
「export」する要素(1~4)

隣接領域2(#3)へ
「export」する要素(5~8)

一般化された通信テーブル(6/6)

PE#3

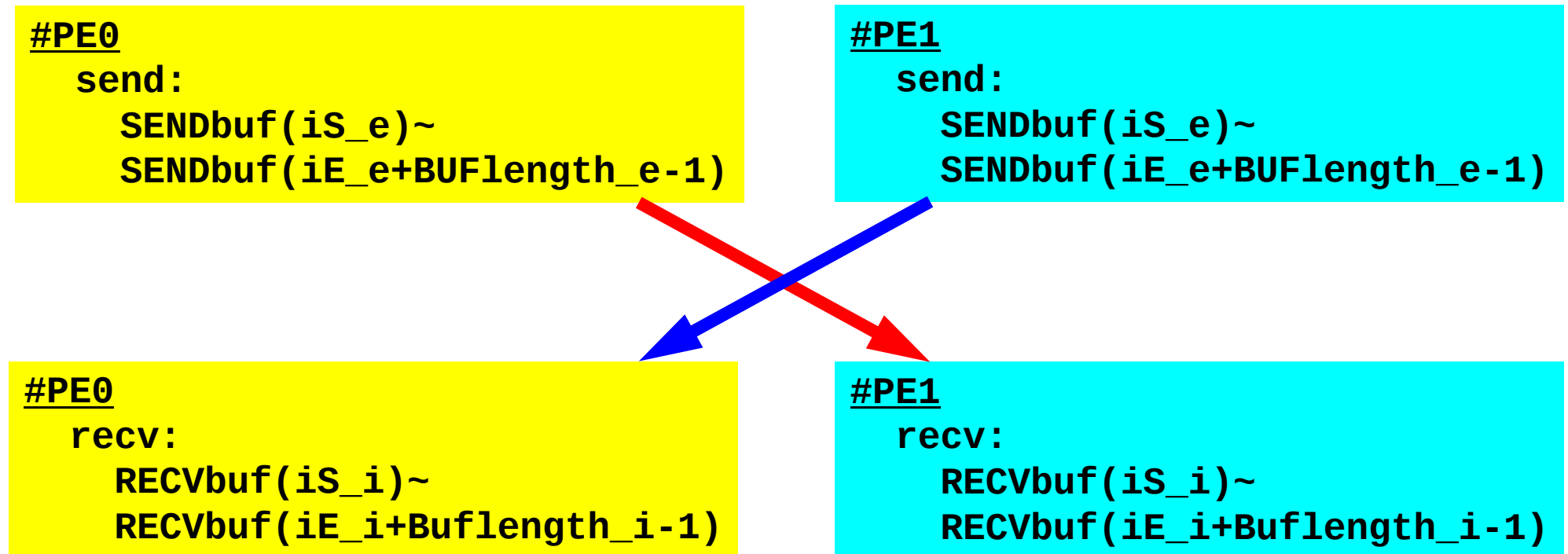
21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17

PE#1

「外点」はその要素が本来所属している領域からのみ受信される。

「境界点」は複数の領域において「外点」となっている可能性があるので、複数の領域に送信されることもある(16番要素の例)。

配列の送受信:注意



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

1対1通信

- 1対1通信とは ?
- 二次元問題, 一般化された通信テーブル
 - 二次元差分法
 - 問題設定
 - 局所データ構造と通信テーブル
 - 実装例
- 課題S2

サンプルプログラム：二次元データの例

```
$ cd <$0-S2>
```

```
$ mpifrtpx -Kfast sq-sr1.f
```

```
$ mpifccpx -Kfast sq-sr1.c
```

```
$ 実行:4プロセス go4.sh
```

プログラム例: sq-sr1.f (1/6)

初期化

Fortran

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

integer(kind=4) :: my_rank, PETOT
integer(kind=4) :: N, NP, NEIBPETOT, BUFLength

integer(kind=4), dimension(:), allocatable :: VAL
integer(kind=4), dimension(:), allocatable :: SENDbuf, RECVbuf
integer(kind=4), dimension(:), allocatable :: NEIBPE

integer(kind=4), dimension(:), allocatable :: import_index, import_item
integer(kind=4), dimension(:), allocatable :: export_index, export_item

integer(kind=4), dimension(:, :), allocatable :: stat_send, stat_recv
integer(kind=4), dimension(: ), allocatable :: request_send
integer(kind=4), dimension(: ), allocatable :: request_recv

character(len=80)          :: filename, line

!C
!C +-----+
!C | INIT. MPI |
!C +-----+
!C===
      call MPI_INIT          (ierr)
      call MPI_COMM_SIZE    (MPI_COMM_WORLD, PETOT, ierr )
      call MPI_COMM_RANK    (MPI_COMM_WORLD, my_rank, ierr )
```

プログラム例: sq-sr1.f (2/6)

Fortran

局所分散メッシュデータ(sqm.*)読み込み

```
!C
!C-- MESH
  if (my_rank.eq.0) filename= 'sqm.0'
  if (my_rank.eq.1) filename= 'sqm.1'
  if (my_rank.eq.2) filename= 'sqm.2'
  if (my_rank.eq.3) filename= 'sqm.3'
  open (21, file= filename, status= 'unknown')
    read (21, *) NEIBPETOT
      allocate (NEIBPE(NEIBPETOT))
      allocate (import_index(0:NEIBPETOT))
      allocate (export_index(0:NEIBPETOT))
      import_index= 0
      export_index= 0

      read (21, *) (NEIBPE(neib), neib= 1, NEIBPETOT)
      read (21, *) NP, N
      read (21, '(a80)') line
      read (21, *) (import_index(neib), neib= 1, NEIBPETOT)
        nn= import_index(NEIBPETOT)
        allocate (import_item(nn))

      do i= 1, nn
        read (21, *) import_item(i)
      enddo
      read (21, '(a80)') line
      read (21, *) (export_index(neib), neib= 1, NEIBPETOT)
        nn= export_index(NEIBPETOT)
        allocate (export_item(nn))

      do i= 1, nn
        read (21, *) export_item(i)
      enddo
  close (21)
```

プログラム例: sq-sr1.f (2/6)

Fortran

局所分散メッシュデータ(sqm.*)読み込み

```

!C
!C-- MESH
      if (my_rank.eq.0) filename= 'sqm.0'
      if (my_rank.eq.1) filename= 'sqm.1'
      if (my_rank.eq.2) filename= 'sqm.2'
      if (my_rank.eq.3) filename= 'sqm.3'
      open (21, file= filename, status= 'unknown')
         read (21,*) NEIBPETOT
         allocate (NEIBPE(NEIBPETOT))
         allocate (import_index(0:NEIBPETOT))
         allocate (export_index(0:NEIBPETOT))
         import_index= 0
         export_index= 0

         read (21,*) (NEIBPE(neib), neib= 1, NEIBPETOT)
         read (21,*) NP, N

         read (21,*) (import_index(neib), neib= 1, NEIBPETOT)
         nn= import_index(NEIBPETOT)
         allocate (import_item(nn))

         do i= 1, nn
            read (21,*) import_item(i)
         enddo

         read (21,*) (export_index(neib), neib= 1, NEIBPETOT)
         nn= export_index(NEIBPETOT)
         allocate (export_item(nn))

         do i= 1, nn
            read (21,*) export_item(i)
         enddo
      close (21)

```

```

#NEIBPETot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16

```

プログラム例: sq-sr1.c (2/6)

Fortran

局所分散メッシュデータ(sqm.*)読み込み

```
!C
!C-- MESH
if (my_rank.eq.0) filename= 'sqm.0'
if (my_rank.eq.1) filename= 'sqm.1'
if (my_rank.eq.2) filename= 'sqm.2'
if (my_rank.eq.3) filename= 'sqm.3'
open (21, file= filename, status= 'unknown')
  read (21, *) NEIBPETOT
    allocate (NEIBPE(NEIBPETOT))
    allocate (import_index(0:NEIBPETOT))
    allocate (export_index(0:NEIBPETOT))
    import_index= 0
    export_index= 0

  read (21, *) (NEIBPE(neib), neib= 1, NEIBPETOT)
  read (21, *) NP, N
  ) line
  NP 総要素数
  N   内点数
  port_index(neib), neib= 1, NEIBPETOT)
  = import_index(NEIBPETOT)
  locate (import_item(nn))
do i= 1, nn
  read (21, *) import_item(i)
enddo
read (21, '(a80)') line
read (21, *) (export_index(neib), neib= 1, NEIBPETOT)
  nn= export_index(NEIBPETOT)
  allocate (export_item(nn))
do i= 1, nn
  read (21, *) export_item(i)
enddo
close (21)
```

```
#NEIBPETot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16
```

プログラム例: sq-sr1.c (2/6)

Fortran

局所分散メッシュデータ(sqm.*)読み込み

```
!C
!C-- MESH
  if (my_rank.eq.0) filename= 'sqm.0'
  if (my_rank.eq.1) filename= 'sqm.1'
  if (my_rank.eq.2) filename= 'sqm.2'
  if (my_rank.eq.3) filename= 'sqm.3'
  open (21, file= filename, status= 'unknown')
    read (21, *) NEIBPETOT
      allocate (NEIBPE(NEIBPETOT))
      allocate (import_index(0:NEIBPETOT))
      allocate (export_index(0:NEIBPETOT))
      import_index= 0
      export_index= 0

      read (21, *) (NEIBPE(neib), neib= 1, NEIBPETOT)
      read (21, *) NP, N

      read (21, *) (import_index(neib), neib= 1, NEIBPETOT)
      nn= import_index(NEIBPETOT)
      allocate (import_item(nn))

      do i= 1, nn
        read (21, *) import_item(i)
      enddo

      read (21, *) (export_index(neib), neib= 1, NEIBPETOT)
      nn= export_index(NEIBPETOT)
      allocate (export_item(nn))

      do i= 1, nn
        read (21, *) export_item(i)
      enddo
  close (21)
```

```
#NEIBPETot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16
```

プログラム例: sq-sr1.f (2/6)

局所分散メッシュデータ(sqm.*)読み込み

Fortran

```

!C
!C-- MESH
      if (my_rank.eq.0) filename= 'sqm.0'
      if (my_rank.eq.1) filename= 'sqm.1'
      if (my_rank.eq.2) filename= 'sqm.2'
      if (my_rank.eq.3) filename= 'sqm.3'
      open (21, file= filename, status= 'unknown')
      read (21,*) NEIBPETOT
      allocate (NEIBPE(NEIBPETOT))
      allocate (import_index(0:NEIBPETOT))
      allocate (export_index(0:NEIBPETOT))
      import_index= 0
      export_index= 0

      read (21,*) (NEIBPE(neib), neib= 1, NEIBPETOT)
      read (21,*) NP, N

      read (21,*) (import_index(neib), neib= 1, NEIBPETOT)
      nn= import_index(NEIBPETOT)
      allocate (import_item(nn))

      do i= 1, nn
        read (21,*) import_item(i)
      enddo

      read (21,*) (export_index(neib), neib= 1, NEIBPETOT)
      nn= export_index(NEIBPETOT)
      allocate (export_item(nn))

      do i= 1, nn
        read (21,*) export_item(i)
      enddo
      close (21)

```

```

#NEIBPETot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16

```

PE#0 受信

```
#NEIBPEtot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16
```

<u>PE#2</u>				
21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17
<u>PE#0</u>				<u>PE#1</u>

プログラム例: sq-sr1.f (2/6)

Fortran

局所分散メッシュデータ(sqm.*)読み込み

```

!C
!C-- MESH
      if (my_rank.eq.0) filename= 'sqm.0'
      if (my_rank.eq.1) filename= 'sqm.1'
      if (my_rank.eq.2) filename= 'sqm.2'
      if (my_rank.eq.3) filename= 'sqm.3'
      open (21, file= filename, status= 'unknown')
      read (21, *) NEIBPETOT
      allocate (NEIBPE(NEIBPETOT))
      allocate (import_index(0:NEIBPETOT))
      allocate (export_index(0:NEIBPETOT))
      import_index= 0
      export_index= 0

      read (21, *) (NEIBPE(neib), neib= 1, NEIBPETOT)
      read (21, *) NP, N

      read (21, *) (import_index(neib), neib= 1, NEIBPETOT)
      nn= import_index(NEIBPETOT)
      allocate (import_item(nn))

      do i= 1, nn
        read (21, *) import_item(i)
      enddo

      read (21, *) (export_index(neib), neib= 1, NEIBPETOT)
      nn= export_index(NEIBPETOT)
      allocate (export_item(nn))

      do i= 1, nn
        read (21, *) export_item(i)
      enddo
      close (21)

```

```

#NEIBPETot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16

```

プログラム例: sq-sr1.f (2/6)

Fortran

局所分散メッシュデータ(sqm.*)読み込み

```

!C
!C-- MESH
      if (my_rank.eq.0) filename= 'sqm.0'
      if (my_rank.eq.1) filename= 'sqm.1'
      if (my_rank.eq.2) filename= 'sqm.2'
      if (my_rank.eq.3) filename= 'sqm.3'
      open (21, file= filename, status= 'unknown')
      read (21, *) NEIBPETOT
      allocate (NEIBPE(NEIBPETOT))
      allocate (import_index(0:NEIBPETOT))
      allocate (export_index(0:NEIBPETOT))
      import_index= 0
      export_index= 0

      read (21, *) (NEIBPE(neib), neib= 1, NEIBPETOT)
      read (21, *) NP, N

      read (21, *) (import_index(neib), neib= 1, NEIBPETOT)
      nn= import_index(NEIBPETOT)
      allocate (import_item(nn))

      do i= 1, nn
        read (21, *) import_item(i)
      enddo

      read (21, *) (export_index(neib), neib= 1, NEIBPETOT)
      nn= export_index(NEIBPETOT)
      allocate (export_item(nn))

      do i= 1, nn
        read (21, *) export_item(i)
      enddo
      close (21)

```

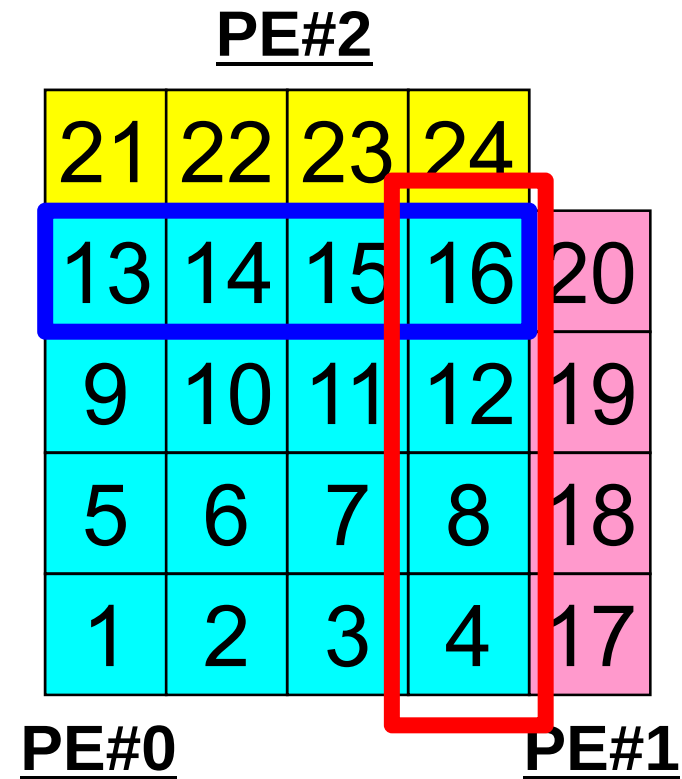
```

#NEIBPETot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16

```

PE#0 送信

```
#NEIBPEtot  
2  
#NEIBPE  
1 2  
#NODE  
24 16  
#IMPORTindex  
4 8  
#IMPORTitems  
17  
18  
19  
20  
21  
22  
23  
24  
#EXPORTindex  
4 8  
#EXPORTitems  
4  
8  
12  
16  
13  
14  
15  
16
```



プログラム例: sq-sr1.f (3/6)

Fortran

局所分散データ(全体番号の値)(sq.*)読み込み

```
!C
!C-- VAL.
      if (my_rank.eq.0) filename= 'sq.0'
      if (my_rank.eq.1) filename= 'sq.1'
      if (my_rank.eq.2) filename= 'sq.2'
      if (my_rank.eq.3) filename= 'sq.3'

      allocate (VAL(NP))
      VAL= 0
      open (21, file= filename, status= 'unknown')
        do i= 1, N
          read (21, *) VAL(i)
        enddo
      close (21)
!C==
```

N : 内点数

VAL : 全体要素番号を読み込む

この時点で外点の値はわかっていない

PE#2

<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>	
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	

PE#0

PE#1

1
2
3
4
9
10
11
12
17
18
19
20
25
26
27
28

プログラム例: sq-sr1.f (4/6)

送・受信バッファ準備

Fortran

```
!C
!C +-----+
!C |  BUFFER  |
!C +-----+
!C===
      allocate (SENDbuf(export_index(NEIBPETOT)))
      allocate (RECVbuf(import_index(NEIBPETOT)))

      SENDbuf= 0
      RECVbuf= 0

      do neib= 1, NEIBPETOT
        iS= export_index(neib-1) + 1
        iE= export_index(neib  )
        do i= iS, iE
          SENDbuf(i)= VAL(export_item(i))
        enddo
      enddo
!C===
```

送信バッファに「境界点」の情報を入れる。送信バッファの `export_index(neib-1)+1` から `export_index(neib)` までに `NEIBPE(neib)` に送信する情報を格納する。

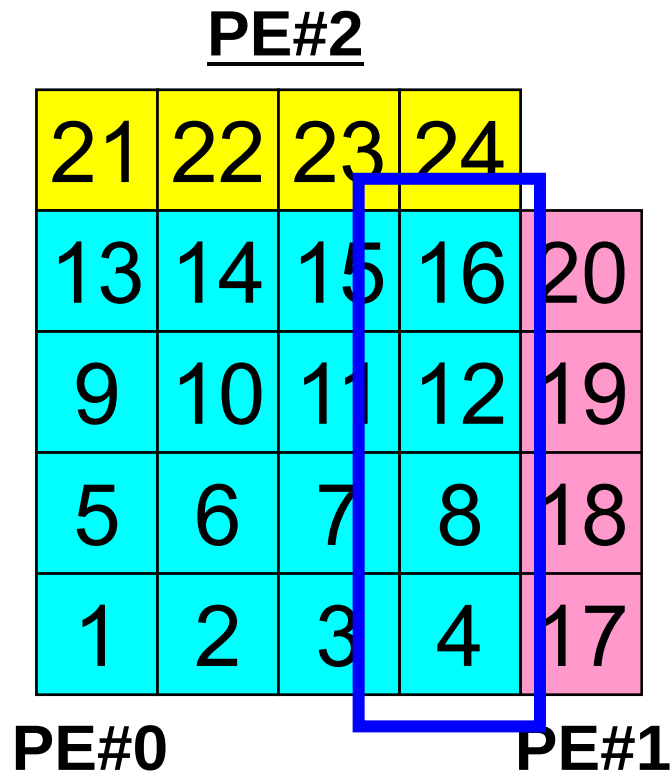
送信バッファの効能

```

do neib= 1, NEIBPETOT
  iS_e= export_index(neib-1) + 1
  iE_e= export_index(neib )
  BUFlength_e= iE_e + 1 - iS_e

  call MPI_ISEND
&      (VAL(...), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&      MPI_COMM_WORLD, request_send(neib), ierr)
enddo

```

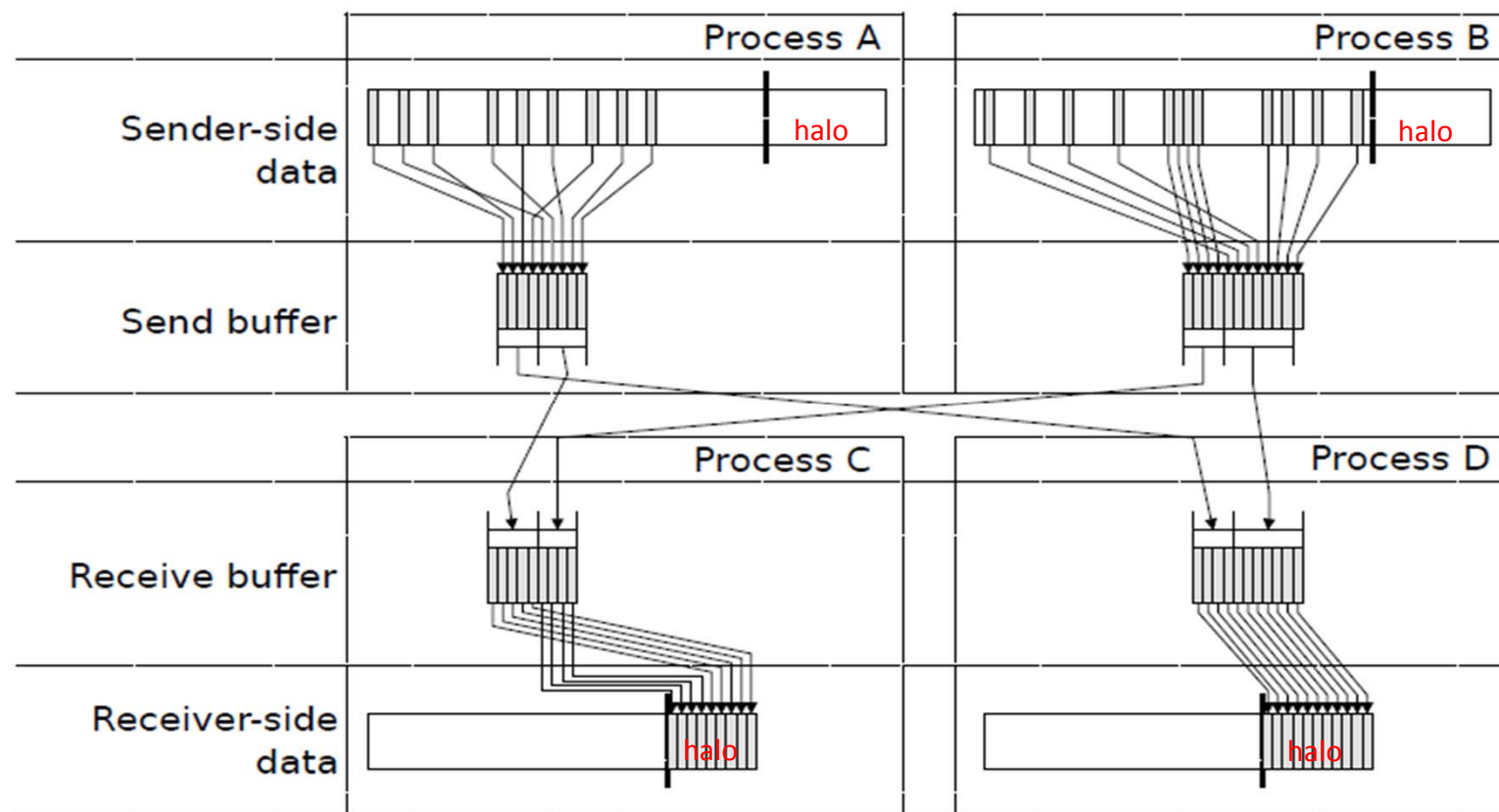


たとえば, この境界点は連続してない
ので,

- ・ 送信バッファの先頭アドレス
- ・ そこから数えて●●のサイズの
メッセージ

というような方法が困難

Communication Pattern using 1D Structure



Dr. Osni Marques
(Lawrence Berkeley National
Laboratory) より借用

プログラム例: sq-sr1.f (5/6)

Fortran

送信 (MPI_Isend)

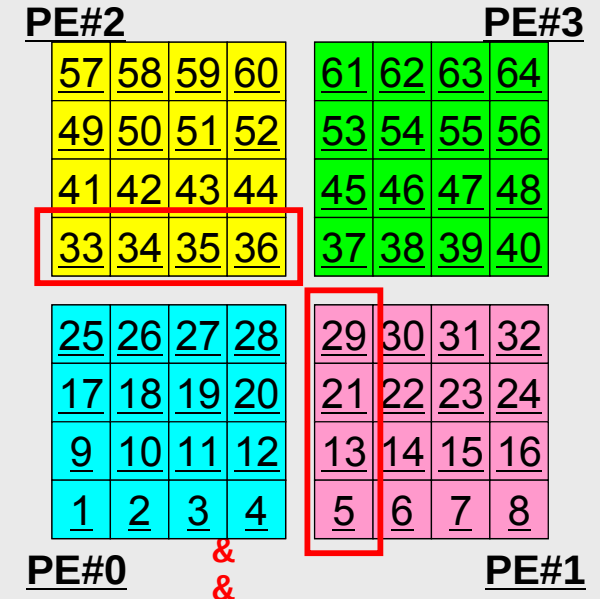
```

!C
!C +-----+
!C | SEND-RECV |
!C +-----+
!C===
      allocate (stat_send(MPI_STATUS_SIZE, NEIBPETOT))
      allocate (stat_recv(MPI_STATUS_SIZE, NEIBPETOT))
      allocate (request_send(NEIBPETOT))
      allocate (request_recv(NEIBPETOT))

      do neib= 1, NEIBPETOT
        iS= export_index(neib-1) + 1
        iE= export_index(neib )
        BUFlength= iE + 1 - iS
        call MPI_ISEND (SENDbuf(iS), BUFlength, MPI_INTEGER,
&                      NEIBPE(neib), 0, MPI_COMM_WORLD,
&                      request_send(neib), ierr)
      enddo

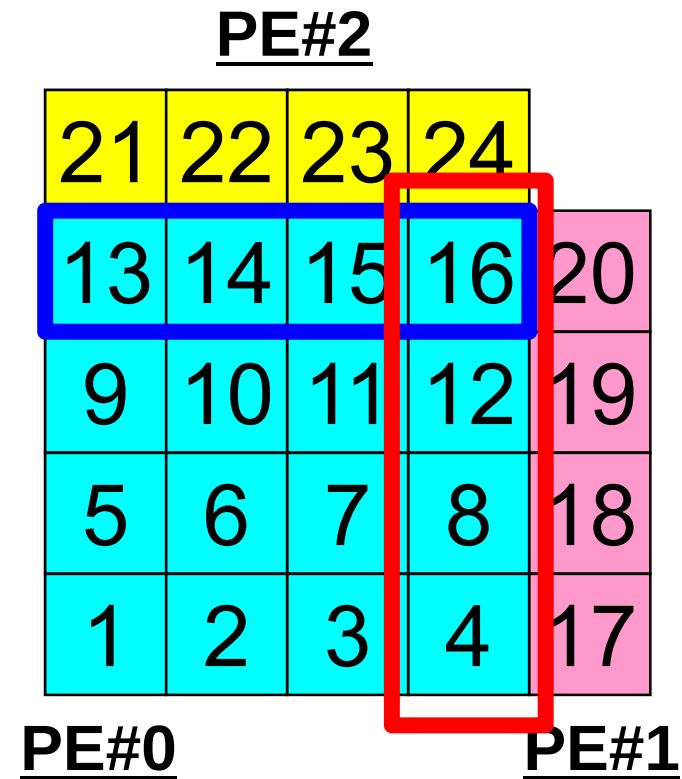
      do neib= 1, NEIBPETOT
        iS= import_index(neib-1) + 1
        iE= import_index(neib )
        BUFlength= iE + 1 - iS
        call MPI_IRECV (RECVbuf(iS), BUFlength, MPI_INTEGER,
&                      NEIBPE(neib), 0, MPI_COMM_WORLD,
&                      request_recv(neib), ierr)
      enddo

```



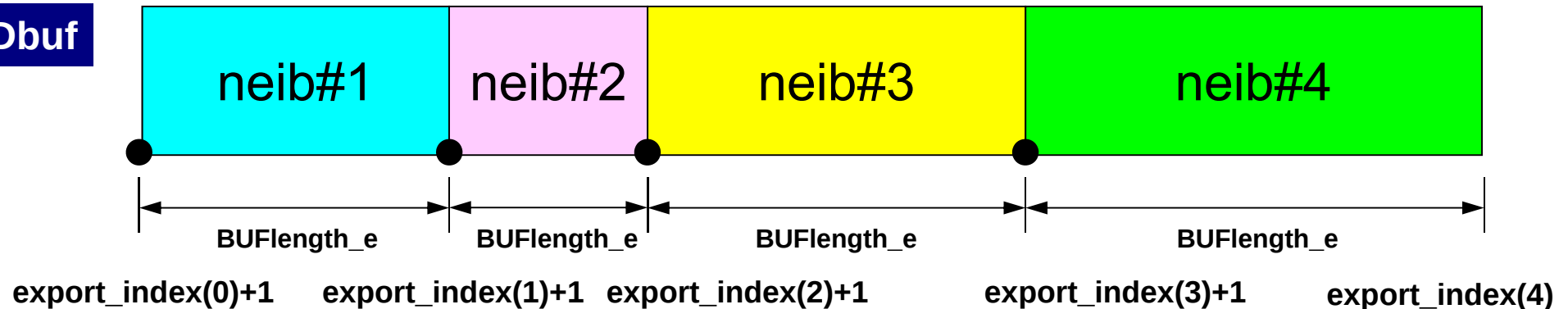
PE#0 送信

```
#NEIBPEtot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16
```



送信 (MPI_Isend/Irecv/Waitall) Fortran

SENDbuf



```

do neib= 1, NEIBPETOT
  do k= export_index(neib-1)+1, export_index(neib)
    kk= export_item(k)
    SENDbuf(k)= VAL(kk)
  enddo
enddo

do neib= 1, NEIBPETOT
  iS_e= export_index(neib-1) + 1
  iE_e= export_index(neib)
  BUFlength_e= iE_e + 1 - iS_e

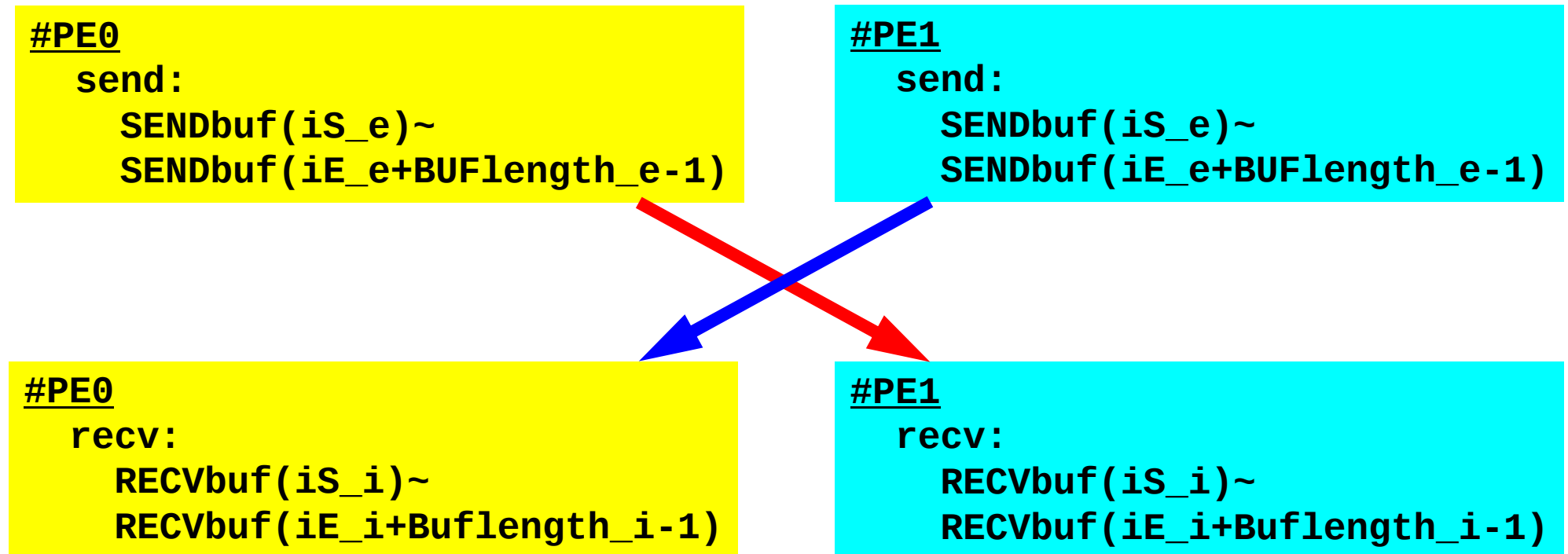
  call MPI_ISEND
&      (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&      MPI_COMM_WORLD, request_send(neib), ierr)
  enddo

call MPI_WAITALL (NEIBPETOT, request_send, stat_recv, ierr)

```

送信バッファへの代入
温度などの変数を直接送信, 受信に使うのではなく, このようなバッファへ一回代入して計算することを勧める。

配列の送受信:注意



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

送信と受信の関係

```
do neib= 1, NEIBPETOT
  iS_e= export_index(neib-1) + 1
  iE_e= export_index(neib )
  BUFlength_e= iE_e + 1 - iS_e

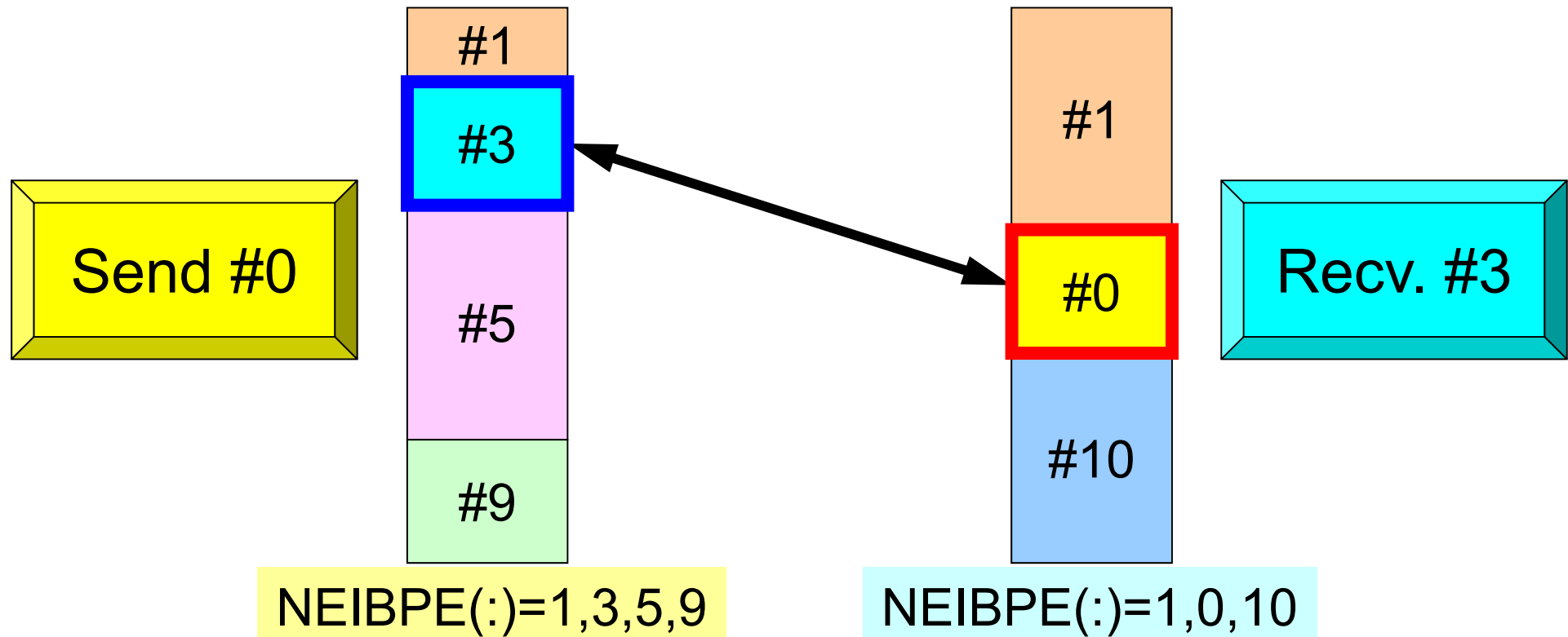
  call MPI_ISEND
&      (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&      MPI_COMM_WORLD, request_send(neib), ierr)
enddo
```

```
do neib= 1, NEIBPETOT
  iS_i= import_index(neib-1) + 1
  iE_i= import_index(neib )
  BUFlength_i= iE_i + 1 - iS_i

  call MPI_Irecv
&      (RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&      MPI_COMM_WORLD, request_recv(neib), ierr)
enddo
```

- 送信元・受信先プロセス番号, メッセージサイズ, 内容の整合性 !
- NEIBPE(neib)がマッチしたときに通信が起こる。

送信と受信の関係 (#0⇒#3)



- 送信元・受信先プロセス番号, メッセージサイズ, 内容の整合性 !
- NEIBPE(neib)がマッチしたときに通信が起こる。

プログラム例: sq-sr1.f (5/6)

Fortran

受信 (MPI_Irecv)

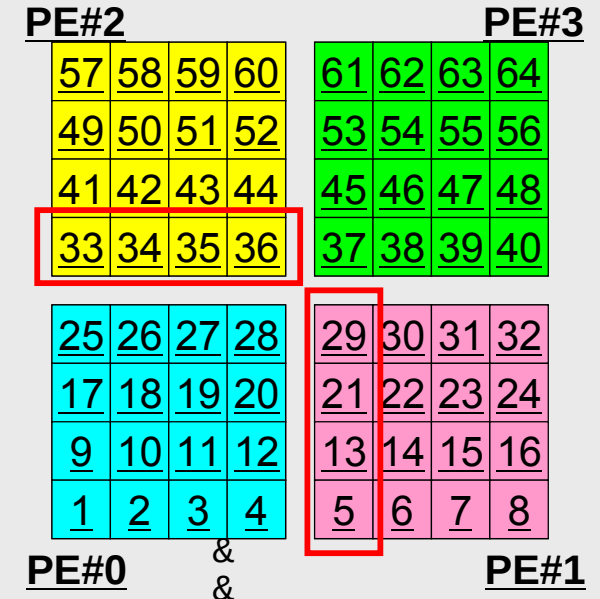
```

!C
!C +-----+
!C | SEND-RECV |
!C +-----+
!C===
      allocate (stat_send(MPI_STATUS_SIZE, NEIBPETOT))
      allocate (stat_recv(MPI_STATUS_SIZE, NEIBPETOT))
      allocate (request_send(NEIBPETOT))
      allocate (request_recv(NEIBPETOT))

      do neib= 1, NEIBPETOT
        iS= export_index(neib-1) + 1
        iE= export_index(neib )
        BUflength= iE + 1 - iS
        call MPI_ISEND (SENDbuf(iS), BUflength, MPI_INTEGER,
&                        NEIBPE(neib), 0, MPI_COMM_WORLD,
&                        request_send(neib), ierr)
      enddo

      do neib= 1, NEIBPETOT
        iS= import_index(neib-1) + 1
        iE= import_index(neib )
        BUflength= iE + 1 - iS
        call MPI_IRECV (RECVbuf(iS), BUflength, MPI_INTEGER,
&                        NEIBPE(neib), 0, MPI_COMM_WORLD,
&                        request_recv(neib), ierr)
      enddo

```



PE#0 受信

```
#NEIBPEtot
2
#NEIBPE
1 2
#NODE
24 16
#IMPORTindex
4 8
#IMPORTitems
17
18
19
20
21
22
23
24
#EXPORTindex
4 8
#EXPORTitems
4
8
12
16
13
14
15
16
```

<u>PE#2</u>				
21	22	23	24	
13	14	15	16	20
9	10	11	12	19
5	6	7	8	18
1	2	3	4	17
<u>PE#0</u>				<u>PE#1</u>

受信 (MPI_Isend/Irecv/Waitall) Fortran

```

do neib= 1, NEIBPETOT
  iS_i= import_index(neib-1) + 1
  iE_i= import_index(neib )
  BUFlength_i= iE_i + 1 - iS_i

  call MPI_Irecv
&      (RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&      MPI_COMM_WORLD, request_recv(neib), ierr)
enddo

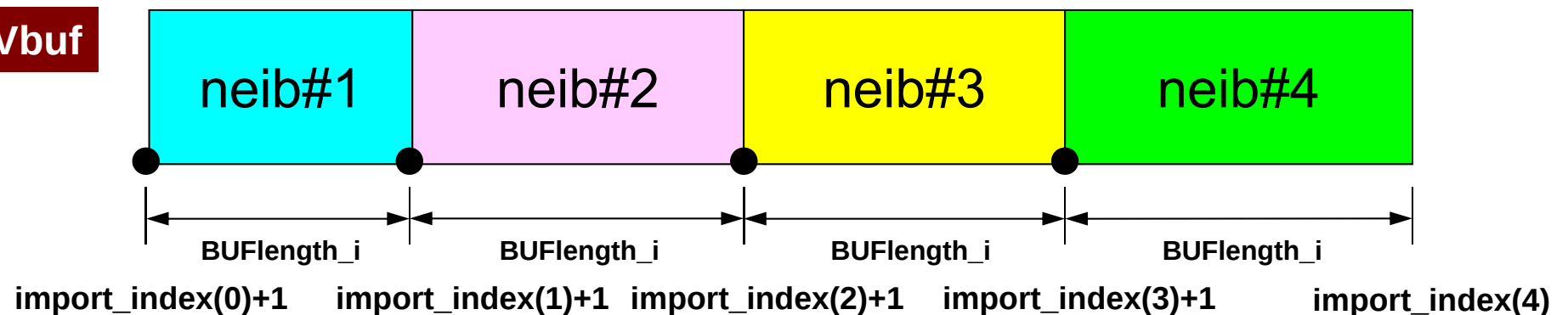
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

do neib= 1, NEIBPETOT
  do k= import_index(neib-1)+1, import_index(neib)
    kk= import_item(k)
    VAL(kk)= RECVbuf(k)
  enddo
enddo

```

受信バッファから代入

RECVbuf



プログラム例: sq-sr1.f (6/6)

Fortran

受信バッファの中身の代入

```
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)
```

```
do neib= 1, NEIBPETOT
  iS= import_index(neib-1) + 1
  iE= import_index(neib )
  do i= iS, iE
    VAL(import_item(i))= RECVbuf(i)
  enddo
enddo
```

受信バッファの中身を「外点」の値として代入する。

```
call MPI_WAITALL (NEIBPETOT, request_send, stat_send, ierr)
```

```
!C===
```

```
!C
!C +-----+
!C | OUTPUT |
!C +-----+
!C===
```

```
do neib= 1, NEIBPETOT
  iS= import_index(neib-1) + 1
  iE= import_index(neib )
  do i= iS, iE
    in= import_item(i)
    write (*, '(a, 3i8)') 'RECVbuf', my_rank, NEIBPE(neib), VAL(in)
  enddo
enddo
```

```
!C===
```

```
call MPI_FINALIZE (ierr)
stop
```

```
end
```

プログラム例: sq-sr1.f (6/6)

外点の値の書き出し

Fortran

```
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

do neib= 1, NEIBPETOT
  iS= import_index(neib-1) + 1
  iE= import_index(neib  )
  do i= iS, iE
    VAL(import_item(i))= RECVbuf(i)
  enddo
enddo

call MPI_WAITALL (NEIBPETOT, request_send, stat_send, ierr)
!C===

!C
!C +-----+
!C |  OUTPUT  |
!C +-----+
!C===
      do neib= 1, NEIBPETOT
        iS= import_index(neib-1) + 1
        iE= import_index(neib  )
        do i= iS, iE
          in= import_item(i)
          write (*,'(a, 3i8)') 'RECVbuf', my_rank, NEIBPE(neib), VAL(in)
        enddo
      enddo
!C===
call MPI_FINALIZE (ierr)
stop

end
```

実行結果 (PE#0)

PE#2

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>

PE#0

<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>

PE#3

<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>

PE#1

<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

RECVbuf	0	1	5
RECVbuf	0	1	13
RECVbuf	0	1	21
RECVbuf	0	1	29
RECVbuf	0	2	33
RECVbuf	0	2	34
RECVbuf	0	2	35
RECVbuf	0	2	36
RECVbuf	1	0	4
RECVbuf	1	0	12
RECVbuf	1	0	20
RECVbuf	1	0	28
RECVbuf	1	3	37
RECVbuf	1	3	38
RECVbuf	1	3	39
RECVbuf	1	3	40
RECVbuf	2	3	37
RECVbuf	2	3	45
RECVbuf	2	3	53
RECVbuf	2	3	61
RECVbuf	2	0	25
RECVbuf	2	0	26
RECVbuf	2	0	27
RECVbuf	2	0	28
RECVbuf	3	2	36
RECVbuf	3	2	44
RECVbuf	3	2	52
RECVbuf	3	2	60
RECVbuf	3	1	29
RECVbuf	3	1	30
RECVbuf	3	1	31
RECVbuf	3	1	32

実行結果(PE#1)

PE#2

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>

PE#3

<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>

PE#0

<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>

PE#1

<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

RECVbuf	0	1	5
RECVbuf	0	1	13
RECVbuf	0	1	21
RECVbuf	0	1	29
RECVbuf	0	2	33
RECVbuf	0	2	34
RECVbuf	0	2	35
RECVbuf	0	2	36
RECVbuf	1	0	4
RECVbuf	1	0	12
RECVbuf	1	0	20
RECVbuf	1	0	28
RECVbuf	1	3	37
RECVbuf	1	3	38
RECVbuf	1	3	39
RECVbuf	1	3	40
RECVbuf	2	3	37
RECVbuf	2	3	45
RECVbuf	2	3	53
RECVbuf	2	3	61
RECVbuf	2	0	25
RECVbuf	2	0	26
RECVbuf	2	0	27
RECVbuf	2	0	28
RECVbuf	3	2	36
RECVbuf	3	2	44
RECVbuf	3	2	52
RECVbuf	3	2	60
RECVbuf	3	1	29
RECVbuf	3	1	30
RECVbuf	3	1	31
RECVbuf	3	1	32

実行結果 (PE#2)

PE#2

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>

PE#3

<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>

<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>

PE#0

<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

PE#1

RECVbuf	0	1	5
RECVbuf	0	1	13
RECVbuf	0	1	21
RECVbuf	0	1	29
RECVbuf	0	2	33
RECVbuf	0	2	34
RECVbuf	0	2	35
RECVbuf	0	2	36
RECVbuf	1	0	4
RECVbuf	1	0	12
RECVbuf	1	0	20
RECVbuf	1	0	28
RECVbuf	1	3	37
RECVbuf	1	3	38
RECVbuf	1	3	39
RECVbuf	1	3	40
RECVbuf	2	3	37
RECVbuf	2	3	45
RECVbuf	2	3	53
RECVbuf	2	3	61
RECVbuf	2	0	25
RECVbuf	2	0	26
RECVbuf	2	0	27
RECVbuf	2	0	28
RECVbuf	3	2	36
RECVbuf	3	2	44
RECVbuf	3	2	52
RECVbuf	3	2	60
RECVbuf	3	1	29
RECVbuf	3	1	30
RECVbuf	3	1	31
RECVbuf	3	1	32

実行結果 (PE#3)

PE#2

<u>57</u>	<u>58</u>	<u>59</u>	<u>60</u>
<u>49</u>	<u>50</u>	<u>51</u>	<u>52</u>
<u>41</u>	<u>42</u>	<u>43</u>	<u>44</u>
<u>33</u>	<u>34</u>	<u>35</u>	<u>36</u>

PE#3

<u>61</u>	<u>62</u>	<u>63</u>	<u>64</u>
<u>53</u>	<u>54</u>	<u>55</u>	<u>56</u>
<u>45</u>	<u>46</u>	<u>47</u>	<u>48</u>
<u>37</u>	<u>38</u>	<u>39</u>	<u>40</u>

PE#0

<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>
<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>
<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>

PE#1

<u>29</u>	<u>30</u>	<u>31</u>	<u>32</u>
<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>
<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

RECVbuf	0	1	5
RECVbuf	0	1	13
RECVbuf	0	1	21
RECVbuf	0	1	29
RECVbuf	0	2	33
RECVbuf	0	2	34
RECVbuf	0	2	35
RECVbuf	0	2	36
RECVbuf	1	0	4
RECVbuf	1	0	12
RECVbuf	1	0	20
RECVbuf	1	0	28
RECVbuf	1	3	37
RECVbuf	1	3	38
RECVbuf	1	3	39
RECVbuf	1	3	40
RECVbuf	2	3	37
RECVbuf	2	3	45
RECVbuf	2	3	53
RECVbuf	2	3	61
RECVbuf	2	0	25
RECVbuf	2	0	26
RECVbuf	2	0	27
RECVbuf	2	0	28
RECVbuf	3	2	36
RECVbuf	3	2	44
RECVbuf	3	2	52
RECVbuf	3	2	60
RECVbuf	3	1	29
RECVbuf	3	1	30
RECVbuf	3	1	31
RECVbuf	3	1	32

並列計算向け局所(分散)データ構造

- 差分法, 有限要素法, 有限体積法等係数が疎行列のアプリケーションについては領域間通信はこのような局所(分散)データによって実施可能
 - SPMD
 - 内点～外点の順に「局所」番号付け
 - 通信テーブル: 一般化された通信テーブル
- 適切なデータ構造が定められれば, 処理は非常に簡単。
 - 送信バッファに「境界点」の値を代入
 - 送信, 受信
 - 受信バッファの値を「外点」の値として更新

初期全体メッシュ

演習t2

<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>	<u>25</u>
<u>16</u>	<u>17</u>	<u>18</u>	<u>19</u>	<u>20</u>
<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>
<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>

3領域に分割

演習t2

#PE2

<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>16</u>	<u>17</u>	<u>18</u>	<u>19</u>
<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>
<u>6</u>	<u>7</u>	<u>8</u>	

#PE1

<u>23</u>	<u>24</u>	<u>25</u>
<u>18</u>	<u>19</u>	<u>20</u>
<u>13</u>	<u>14</u>	<u>15</u>
<u>8</u>	<u>9</u>	<u>10</u>
	<u>4</u>	<u>5</u>

#PE0

<u>11</u>	<u>12</u>	<u>13</u>		
<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>

3領域に分割

演習t2

#PE2

7 21	8 22	9 23	15 24
4 16	5 17	6 18	14 19
1 11	2 12	3 13	13 14
10 6	11 7	12 8	

#PE1

14 23	7 24	8 25
13 18	5 19	6 20
12 13	3 14	4 15
11 8	1 9	2 10
	9 4	10 5

#PE0

11 11	12 12	13 13		
6 6	7 7	8 8	9 9	10 10
1 1	2 2	3 3	4 4	5 5

PE#0: 局所分散データ(sqm.0)

○の部分をつめよ!

演習t2

#PE2

7 21	8 22	9 23	15 24
4 16	5 17	6 18	14 19
1 11	2 12	3 13	13 14
10 6	11 7	12 8	

#PE1

14 23	7 24	8 25
13 18	5 19	6 20
12 13	3 14	4 15
11 8	1 9	2 10
	9 4	10 5

#PE0

11 11	12 12	13 13		
6 6	7 7	8 8	9 9	10 10
1 1	2 2	3 3	4 4	5 5

```
#NEIBPetot
2
#NEIBPE
1      2
#NODE
13      8 (内点+外点, 内点)
#IMPORTindex
○      ○
#IMPORTitems
○...
#EXPORTindex
○      ○
#EXPORTitems
○...
```

PE#1: 局所分散データ(sqm.1)

○の部分をつめよ!

演習t2

#PE2

7 21	8 22	9 23	15 24
4 16	5 17	6 18	14 19
1 11	2 12	3 13	13 14
10 6	11 7	12 8	

#PE1

14 23	7 24	8 25
13 18	5 19	6 20
12 13	3 14	4 15
11 8	1 9	2 10
	9 4	10 5

#PE0

11 11	12 12	13 13		
6 6	7 7	8 8	9 9	10 10
1 1	2 2	3 3	4 4	5 5

```
#NEIBPetot
2
#NEIBPE
0      2
#NODE
8      14 (内点, 内点+外点)
#IMPORTindex
○      ○
#IMPORTitems
○...
#EXPORTindex
○      ○
#EXPORTitems
○...
```

PE#2: 局所分散データ(sqm.2)

○の部分をつめよ!

演習t2

#PE2

7 21	8 22	9 23	15 24
4 16	5 17	6 18	14 19
1 11	2 12	3 13	13 14
10 6	11 7	12 8	

#PE1

14 23	7 24	8 25
13 18	5 19	6 20
12 13	3 14	4 15
11 8	1 9	2 10
	9 4	10 5

#PE0

11 11	12 12	13 13		
6 6	7 7	8 8	9 9	10 10
1 1	2 2	3 3	4 4	5 5

```
#NEIBPEtot
2
#NEIBPE
1 0
#NODE
9 15 (内点, 内点+外点)
#IMPORTindex
○ ○
#IMPORTitems
○...
#EXPORTindex
○ ○
#EXPORTitems
○...
```

演習t2

#PE2

7 <u>21</u>	8 <u>22</u>	9 <u>23</u>	15 <u>24</u>
4 <u>16</u>	5 <u>17</u>	6 <u>18</u>	14 <u>19</u>
1 <u>11</u>	2 <u>12</u>	3 <u>13</u>	13 <u>14</u>
10 <u>6</u>	11 <u>7</u>	12 <u>8</u>	

#PE1

14 <u>23</u>	7 <u>24</u>	8 <u>25</u>
13 <u>18</u>	5 <u>19</u>	6 <u>20</u>
12 <u>13</u>	3 <u>14</u>	4 <u>15</u>
11 <u>8</u>	1 <u>9</u>	2 <u>10</u>
	9 <u>4</u>	10 <u>5</u>

#PE0

11 <u>11</u>	12 <u>12</u>	13 <u>13</u>		
6 <u>6</u>	7 <u>7</u>	8 <u>8</u>	9 <u>9</u>	10 <u>10</u>
1 <u>1</u>	2 <u>2</u>	3 <u>3</u>	4 <u>4</u>	5 <u>5</u>

手 順

- 内点数, 外点数
- 外点がどこから来ているか?
 - IMPORTIndex, IMPORTItems
 - NEIBPEの順番
- それを逆にたどって, 境界点の送信先を調べる
 - EXPORTIndex, EXPORTItems
 - NEIBPEの順番
- <\$O-S2>/exに「sq.*」がある
- 自分で「sqm.*」を作成する
- <\$O-S2>から「sq-sr1.f/c」をコンパイルした実行形式をコピー
- pjsb go3.sh

課題S2

- 一次元弾性解析コード「1d.f, 1d.c」をMPIによって並列化せよ
- 全要素数を読み込んで、プログラム内で領域分割すること
- 並列性能はあまり出ないが測定して見よ

課題S2

- 提出期限: 2012年10月12日(金) 17:00(メール添付可)
- 内容
 - 1d.f/1d.cを「一般化された通信テーブル」を使って並列化せよ
 - 全要素数を読み込んで、プログラム内で領域分割すること
 - 並列性能について考察すること
 - 要素数はかなり多くしないと多分性能が出ない
 - 計算が終わらないようであれば反復回数を少なくして比較
- 提出物(レポート): 最高級仕様
 - 表紙: 氏名, 学籍番号, 課題番号を明記
 - 以下についてA4 8枚以内(図表含む)でまとめること
 - 基本方針(フロー図), プログラム構造・説明, 考察・課題
 - プログラムリスト
 - 結果出力リスト(最小限にとどめること)