MPIによるプログラミング概要 その2
課題S1出題

2007年5月2日

中島研吾

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

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　Collective Communication
  - 1対1通信　　Point-to-Point Communication

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

前回の復習

- MPIとは
- PC Cluster “CENJU” について

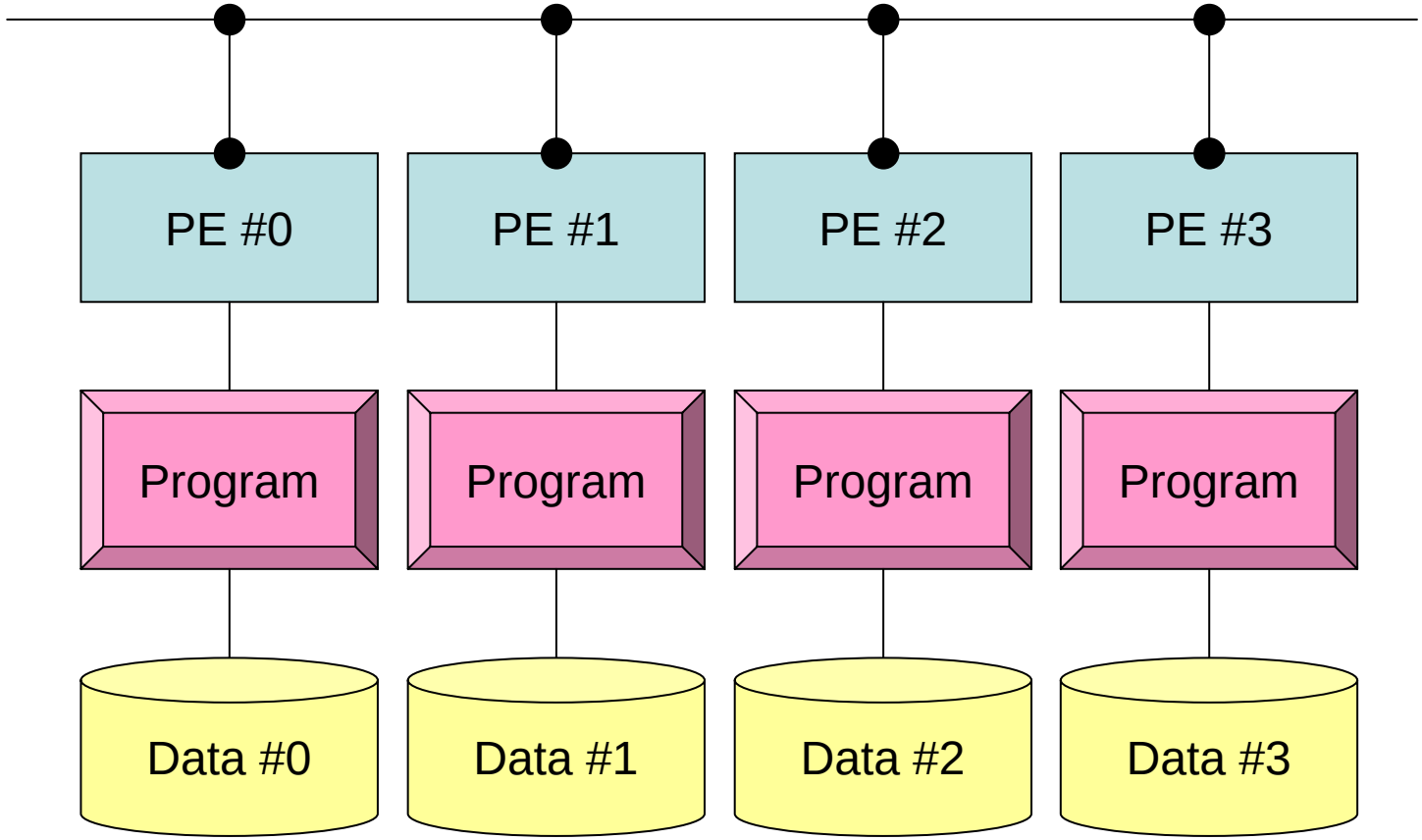
- MPIの基礎：Hello World

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

データ構造とアルゴリズム

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

SPMDに適したデータ構造とは？

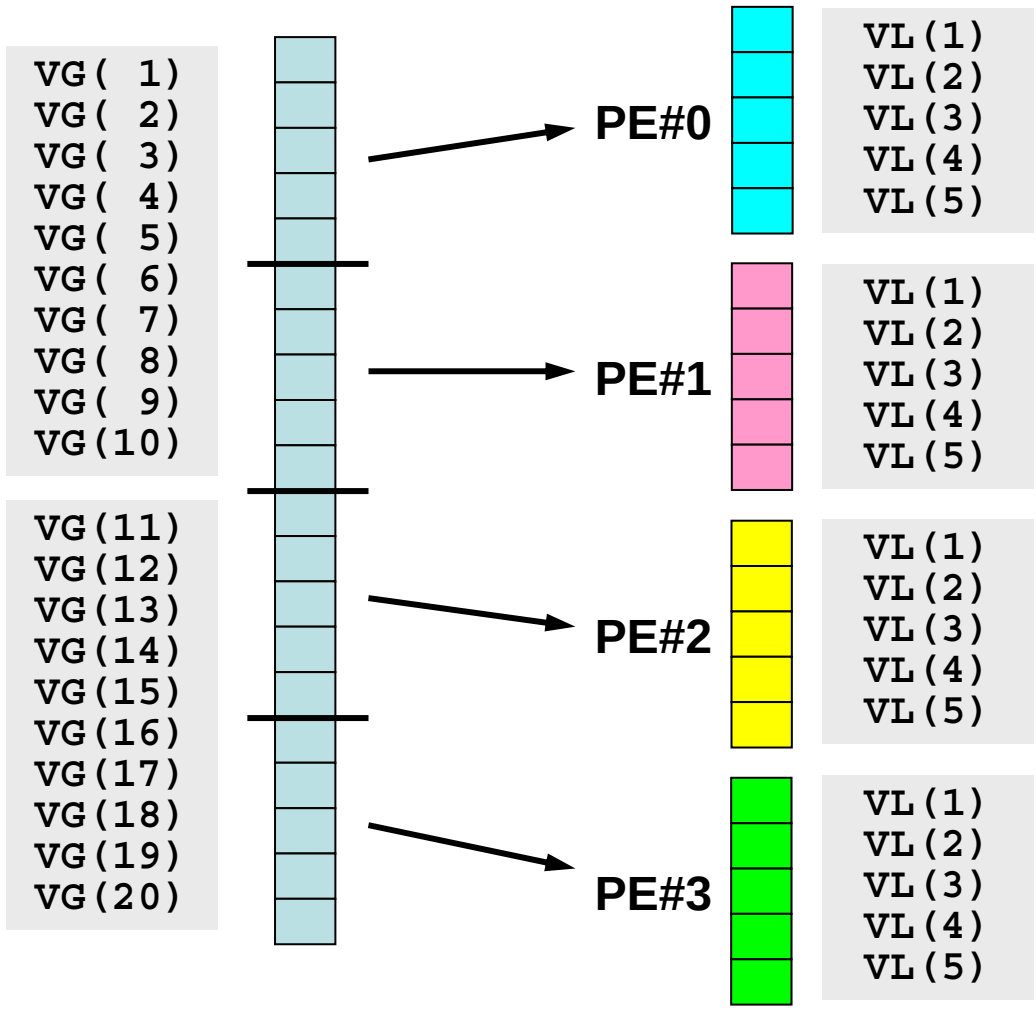


局所データの考え方

「全体データ」**VG**の：

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

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



全体データと局所データ

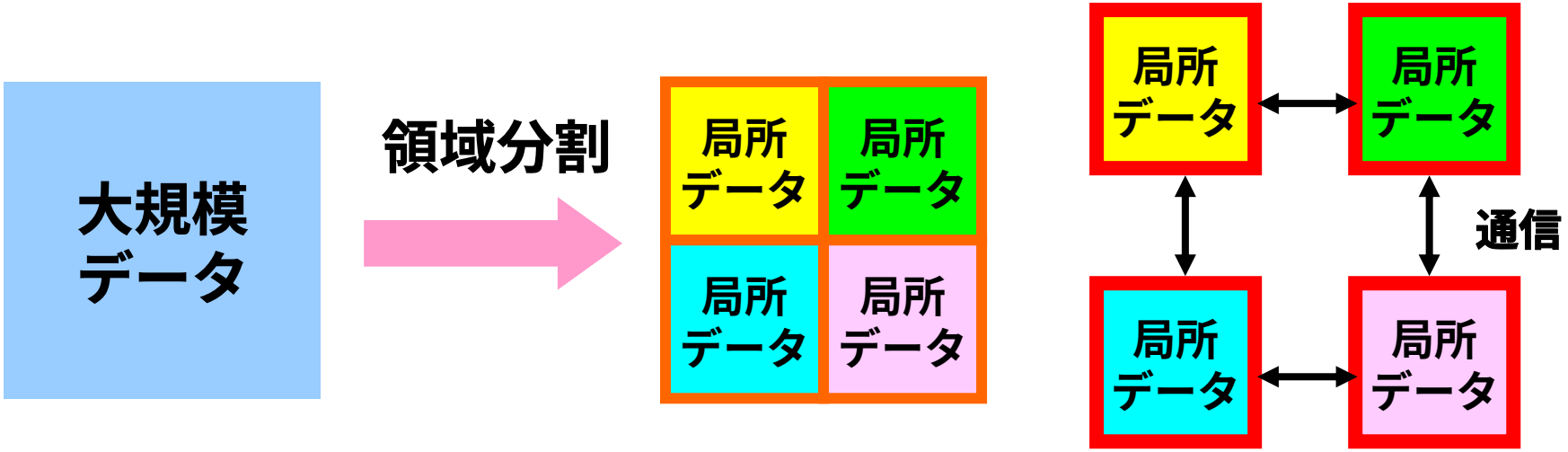
- VG
  - 領域全体
  - 1番から20番までの「全体番号」を持つ「全体データ（Global Data）」
- VL
  - 各プロセッサ
  - 1番から5番までの「局所番号」を持つ「局所データ（Local Data）」
- この講義で常に注意してほしいこと
  - VG（全体データ）からVL（局所データ）をどのように生成するか。
  - VGからVL，VLからVGへデータの中身をどのようにマッピングするか。
  - VLがプロセスごとに独立して計算できない場合はどうするか。
  - できる限り「局所性」を高めた処理を実施する⇒高い並列性能
    - そのための「データ構造」，「ア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

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

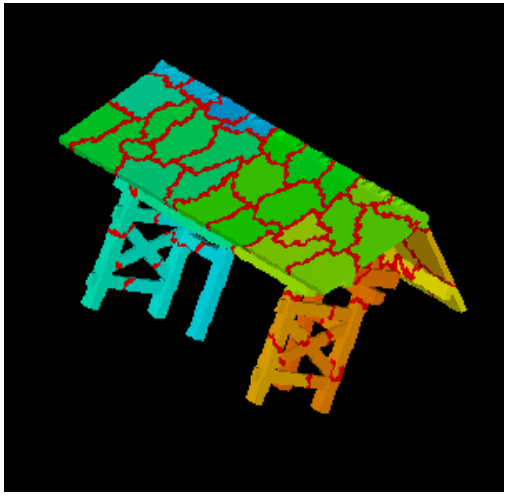
領域分割

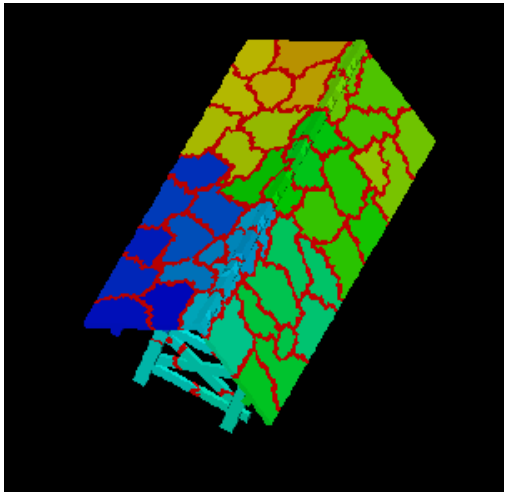
- 1GB程度のPC → 10^6メッシュが限界：FEM
  - 1000km×1000km×1000kmの領域（西南日本）を1kmメッシュで切ると10^9メッシュになる
- 大規模データ → 領域分割，局所データ並列処理
- 全体系計算 → 領域間の通信が必要



局所データ構造

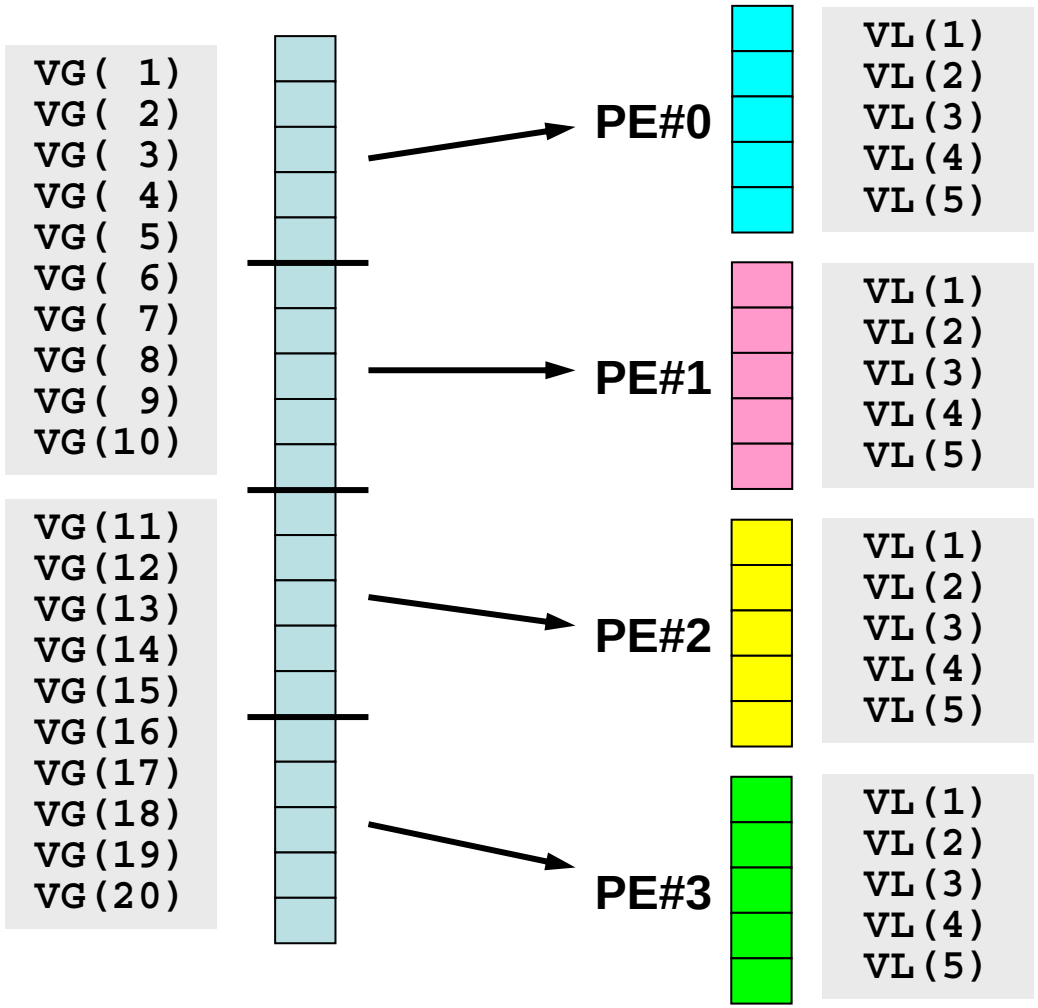
- 対象とする計算（のアルゴリズム）に適した局所データ構造を定めることが重要
  - アルゴリズム＝データ構造
- この講義の主たる目的の一つと言ってよい





局所データ構造（続き）

- といいつつ，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本日の後半に紹介する。



グループ通信による計算例

- ベクトルの内積
- Scatter/Gather
- 分散ファイルの読み込み
- MPI_Allgatherv
  - 前回やり残した部分

MPI_GATHERV，MPI_SCATTERV

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

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

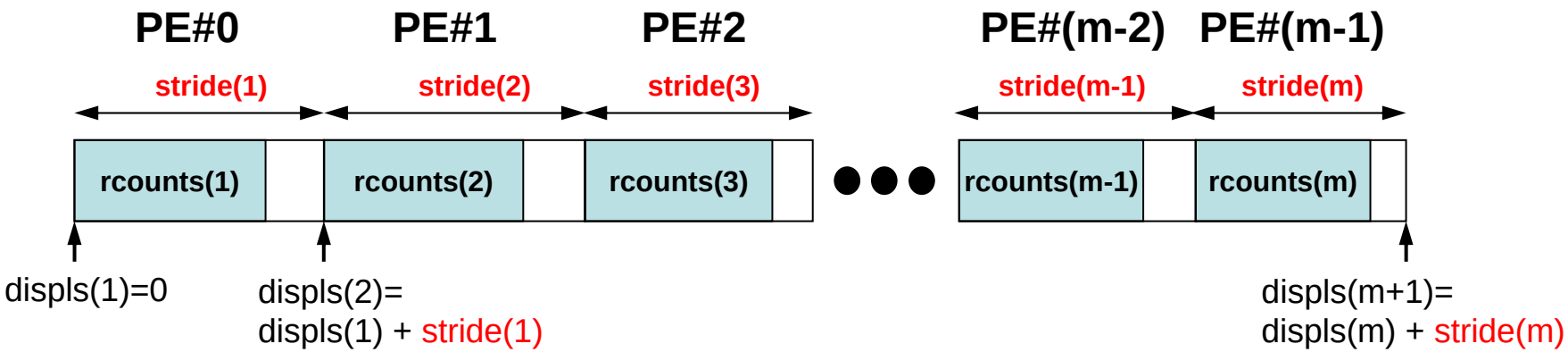
MPI_ALLGATHERV

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

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

MPI_ALLGATHERV（続き）

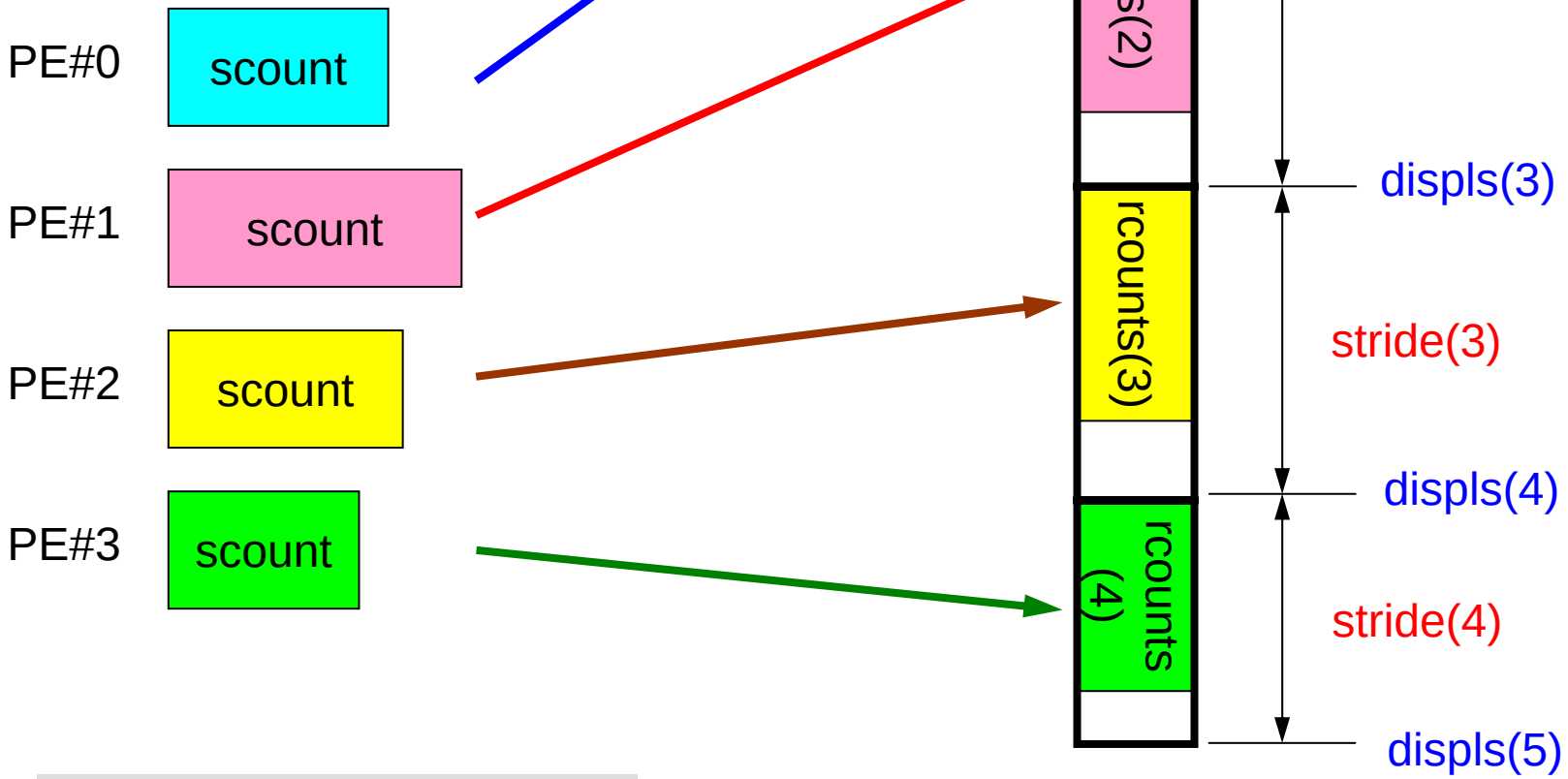
- `call MPI_ALLGATHERV (sendbuf, scount, sendtype, recvbuf, rcounts, displs, recvtype, comm, ierr)`
  - `rcounts` 整数 I 受信メッセージのサイズ（配列：サイズ＝PETOT）
  - `displs` 整数 I 受信メッセージのインデックス（配列：サイズ＝PETOT+1）
  - この2つの配列は，最終的に生成される「全体データ」のサイズに関する配列であるため，各プロセスで配列の全ての値が必要になる：
    - もちろん各プロセスで共通の値を持つ必要がある。
  - 通常は`stride(i)=rcounts(i)`



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

MPI_ALLGATHERV でやっていること

局所データから全体データを生成する

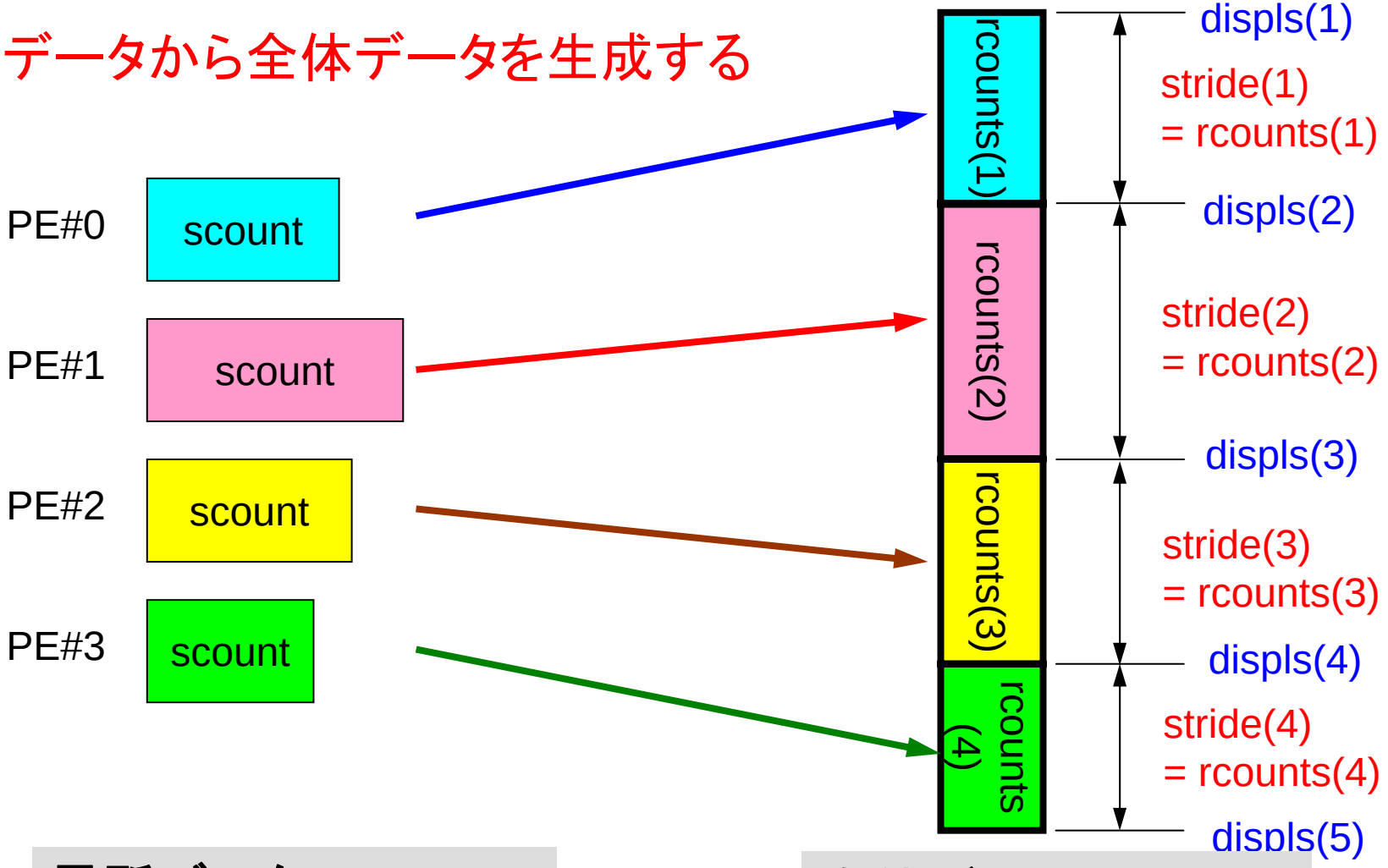


局所データ：sendbuf

全体データ：recvbuf

MPI_ALLGATHERV でやっていること

局所データから全体データを生成する



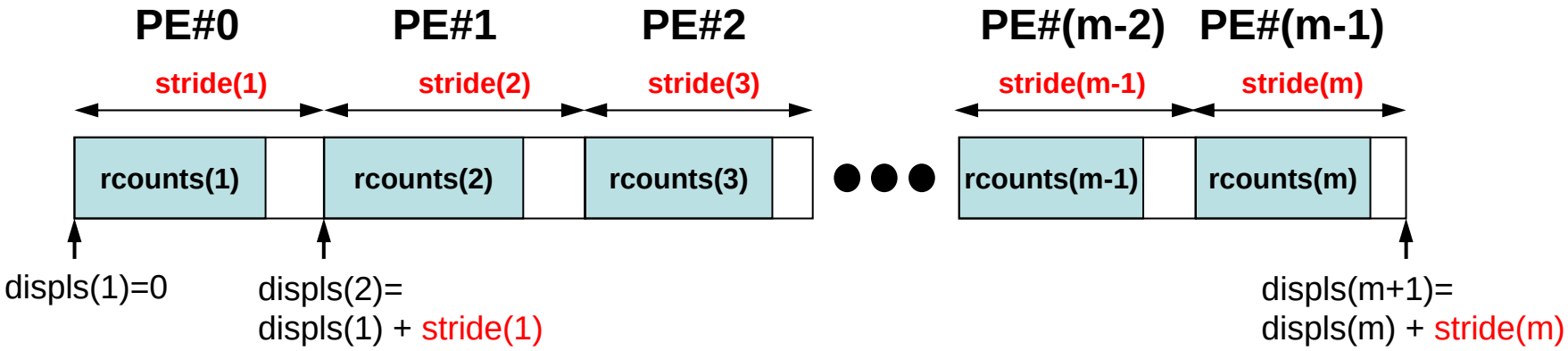
局所データ：sendbuf

全体データ：recvbuf

MPI_ALLGATHERV詳細(1/2)

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

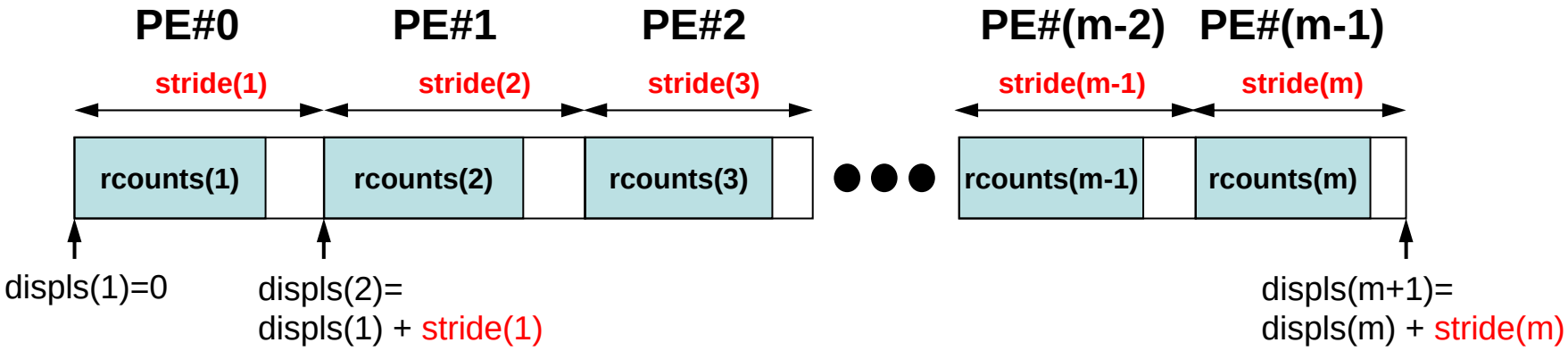
- `rcounts`
  - 各PEにおけるメッセージサイズ:局所データのサイズ
- `displs`
  - 各局所データの全体データにおけるインデックス
  - `displs(PETOT+1)`が全体データのサイズ



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

MPI_ALLGATHERV詳細(2/2)

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



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

MPI_ALLGATHERV使用準備
例題：<$S1>/agv.f，<$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)

<$S1>/agv.f

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

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

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

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

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

N(NL)の値が各PEで異なることに注意

MPI_ALLGATHERV 使用準備(2/4)

<$S1>/agv.f

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

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

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

各PEにrcountsを生成

PE#0 N=8

PE#1 N=5

PE#2 N=7

PE#3 N=3

MPI_Allgather

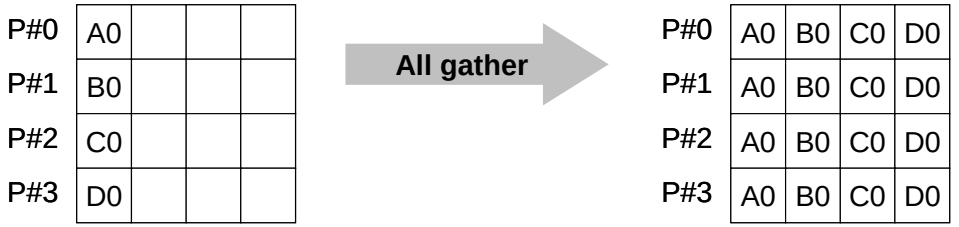
rcounts(1:4)= {8, 5, 7, 3}

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

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

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

MPI_Allgather



- MPI_Allgather = MPI_Gather + MPI_Bcast

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

MPI_ALLGATHERV 使用準備(2/4)

<$S1>/agv.f

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

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

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

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

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

 call MPI_FINALIZE (ierr)

 stop
 end
```

各PEにrcountsを生成

各PEでdisplsを生成

MPI_ALLGATHERV 使用準備(3/4)

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

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：2007年9月19日（水）1700
- 内容
  - 「<$S1>/a1.0～a1.3」，「 <$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）。
    - <$S1>file.f，<$S1>file2.fをそれぞれ参考にする。
  - 「<$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）

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　　Collective Communication
  - 1対1通信　　Point-to-Point Communication

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

1対1通信

- 差分法による一次元熱伝導方程式ソルバー
  - 概要
  - 並列化にあたって:データ構造
- 1対1通信とは
- 1対1通信の実装例
  - 一次元問題
  - 二次元問題
- 差分法による一次元熱伝導方程式ソルバーの並列化

ファイルコピー

```
>$ cd <$07S>
```
前回講義で各自作成したディレクトリ

FORTRAN
```
>$ cp /home/nakajima/class/2007summer/F/s2-f.tar .
>$ tar xvf s2-f.tar
```

C
```
>$ cp /home/nakajima/class/2007summer/C/s2-c.tar .
>$ tar xvf s2-c.tar
```

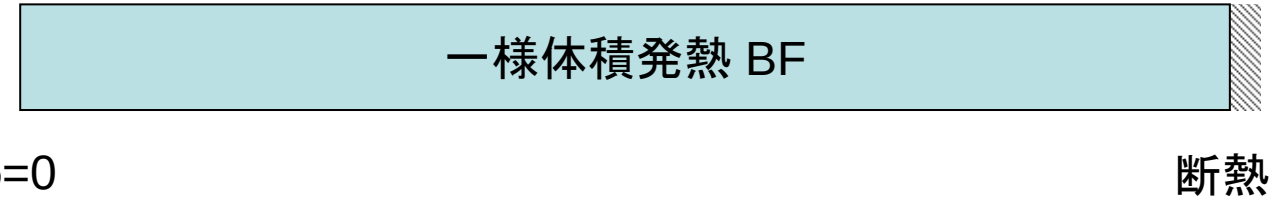
直下に「S2」というディレクトリができている。
<$07S>/S2を<$S2>と呼ぶ。

一次元熱伝導方程式(1/7)

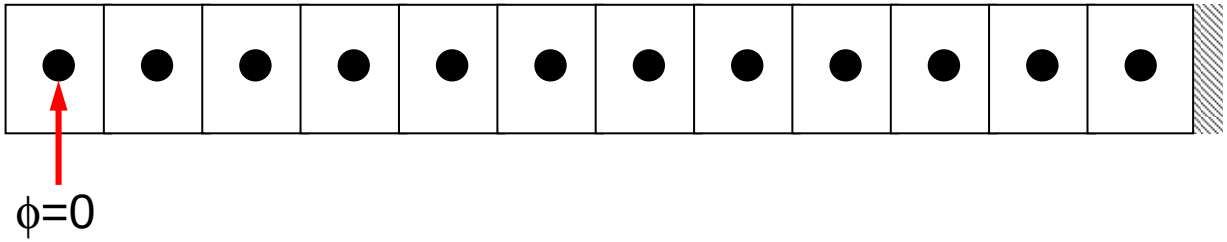
支配方程式:熱伝導率＝1(一様)

$$\frac{d^2\phi}{dx^2} + BF = 0, \quad \phi = 0 @ x = 0, \frac{d\phi}{dx} = 0 @ x = x_{\max}$$

$$\phi = -\frac{1}{2} BF\, x^2 + BF\, x_{\max}\, x$$

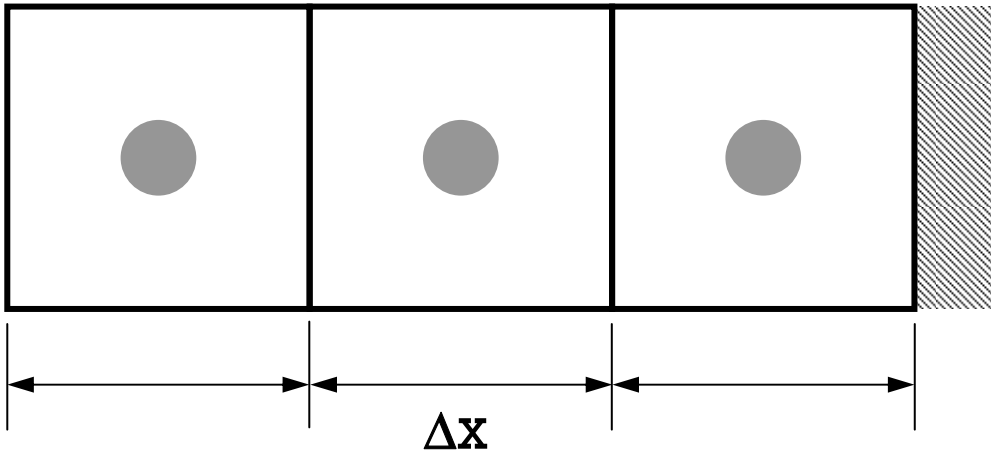


以下のような離散化(要素中心で従属変数を定義)をしているので注意が必要



断熱となっているのはこの面,しかし温度は計算されない

差分格子:要素中心で従属変数を定義

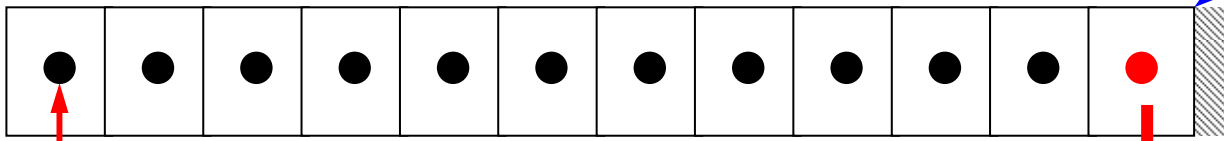


一次元熱伝導方程式(2/7)

解析解

$$\phi = -\frac{1}{2} BF\, x^2 + BF\, x_{\max}\, x$$

断熱となって
いるのはこの面,
しかし温度は計算
されない(X=X_{max})。



ϕ =0
X=0

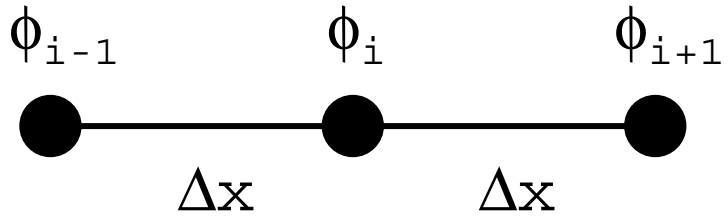
Δx=1.d0, メッシュ数=50, とすると, X_{max}=49.5,
●の点のX座標は49.0となる。BF=1.0d0とすると●での温度は:

$$\phi = -\frac{1}{2} 49^2 + 49.5 \times 49 = -1200.5 + 9850.5 = 1225$$

念のため・・・差分について

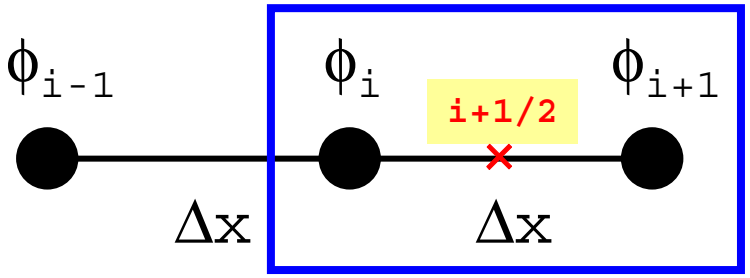
- 差分法：Finite Difference Method
- マクロな微分
  - 微分係数を数値的に近似する手法

- 以下のような一次元系を考える



直感的･･･というか安易な定義

- ×（iとi+1の中点）における微分係数



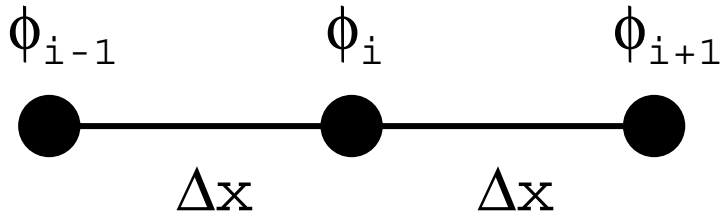
$$\left(\frac{d\phi}{dx}\right)_{i+1/2} \approx \frac{\phi_{i+1}-\phi_i}{\Delta x}$$

Δx→0となると微分係数の定義そのもの

- iにおける二階微分係数

$$\left(\frac{d^2\phi}{dx^2}\right)_i \approx \frac{\left(\frac{d\phi}{dx}\right)_{i+1/2} - \left(\frac{d\phi}{dx}\right)_{i-1/2}}{\Delta x} = \frac{\frac{\phi_{i+1}-\phi_i}{\Delta x} - \frac{\phi_i-\phi_{i-1}}{\Delta x}}{\Delta x} = \frac{\phi_{i+1}-2\phi_i+\phi_{i-1}}{\Delta x^2}$$

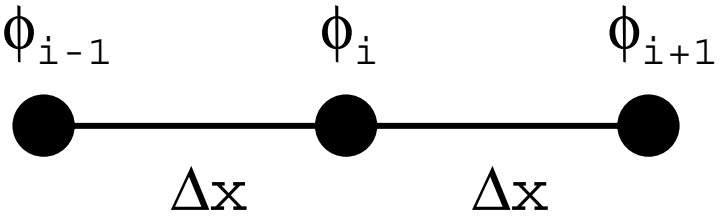
厳密な定義:Taylor展開(1/3)



$$\phi_{i+1} = \phi_i + \Delta x\left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)^2}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i + \frac{(\Delta x)^3}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

$$\phi_{i-1} = \phi_i - \Delta x\left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)^2}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i - \frac{(\Delta x)^3}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

厳密な定義：Taylor展開（2/3）



前進差分

$$\phi_{i+1} = \phi_i + \Delta x\left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)^2}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i + \frac{(\Delta x)^3}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

$$\frac{\phi_{i+1} - \phi_i}{\Delta x} = \left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i + \frac{(\Delta x)^2}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

打ち切り誤差が
Δxのオーダー
（一次精度）

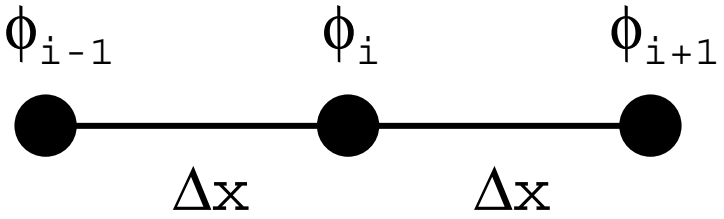
後退差分

$$\phi_{i-1} = \phi_i - \Delta x\left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)^2}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i - \frac{(\Delta x)^3}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

$$\frac{\phi_i - \phi_{i-1}}{\Delta x} = \left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i + \frac{(\Delta x)^2}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

打ち切り誤差が
Δxのオーダー
（一次精度）

厳密な定義：Taylor展開（3/3）



中央差分，中心差分

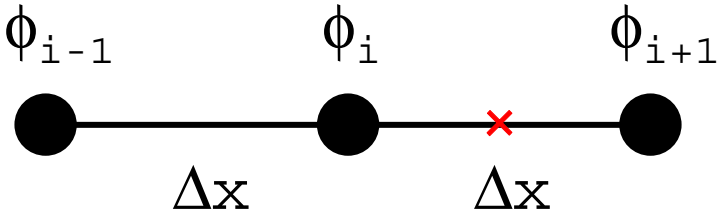
$$\phi_{i+1} = \phi_i + \Delta x\left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)^2}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i + \frac{(\Delta x)^3}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

$$\phi_{i-1} = \phi_i - \Delta x\left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(\Delta x)^2}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i - \frac{(\Delta x)^3}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

$$\frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} = \left(\frac{\partial \phi}{\partial x}\right)_i + \frac{2\times(\Delta x)^2}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i \cdots$$

打ち切り誤差が
$(\Delta x)^2$のオーダー
（二次精度）

安易な定義：実は二次精度だった



$$\phi_{i+1} = \phi_{i+1/2} + \Delta x/2\left(\frac{\partial\phi}{\partial x}\right)_{i+1/2} + \frac{(\Delta x/2)^2}{2!}\left(\frac{\partial^2\phi}{\partial x^2}\right)_{i+1/2} + \frac{(\Delta x/2)^3}{3!}\left(\frac{\partial^3\phi}{\partial x^3}\right)_{i+1/2} \cdots$$

$$\phi_{i} = \phi_{i+1/2} - \Delta x/2\left(\frac{\partial\phi}{\partial x}\right)_{i+1/2} + \frac{(\Delta x/2)^2}{2!}\left(\frac{\partial^2\phi}{\partial x^2}\right)_{i+1/2} - \frac{(\Delta x/2)^3}{3!}\left(\frac{\partial^3\phi}{\partial x^3}\right)_{i+1/2} \cdots$$

$$\frac{\phi_{i+1}-\phi_i}{\Delta x} = \left(\frac{\partial\phi}{\partial x}\right)_{i+1/2} + \frac{2\times(\Delta x/2)^2}{3!}\left(\frac{\partial^3\phi}{\partial x^3}\right)_{i+1/2} \cdots$$

打ち切り誤差が
$(\Delta x)^2$のオーダー
（二次精度）

二点間の中点で二次精度，それ以外の点では一次精度・・・ということもできる。
Δxが均一でない場合も同様のことが起こる。

一次元熱伝導方程式(3/7)

要素単位の線形方程式

- 差分法による離散化

$$\left(\frac{d^2\phi}{dx^2}\right)_i \approx \frac{\left(\frac{d\phi}{dx}\right)_{i+1/2} - \left(\frac{d\phi}{dx}\right)_{i-1/2}}{\Delta x} = \frac{\frac{\phi_{i+1}-\phi_i}{\Delta x} - \frac{\phi_i-\phi_{i-1}}{\Delta x}}{\Delta x} = \frac{\phi_{i+1}-2\phi_i+\phi_{i-1}}{\Delta x^2}$$

- 各要素における線形方程式は以下のような形になる

$$\frac{d^2\phi}{dx^2} + BF = 0 \quad \rightarrow$$

$$\frac{\phi_{i+1}-2\phi_i+\phi_{i-1}}{\Delta x^2} + BF(i) = 0 \quad (1 \le i \le N)$$

$$\frac{1}{\Delta x^2}\phi_{i+1} - \frac{2}{\Delta x^2}\phi_i + \frac{1}{\Delta x^2}\phi_{i-1} + BF(i) = 0 \quad (1 \le i \le N)$$

$$A_L(i)\times\phi_{i-1} + A_D(i)\times\phi_i + A_R(i)\times\phi_{i+1} = BF(i) \quad (1 \le i \le N)$$

$$A_L(i) = \frac{1}{\Delta x^2},\ A_D(i) = -\frac{2}{\Delta x^2},\ A_R(i) = \frac{1}{\Delta x^2}$$

N=8の場合：連立一次方程式

自分とその周囲のみに非ゼロ成分：疎行列

	1	2	3	4	5	6	7	8			
1	$A_D(1)$	$A_R(1)$							×	$\phi(1)$	= $BF(1)$
2	$A_L(2)$	$A_D(2)$	$A_R(2)$							$\phi(2)$	$BF(2)$
3		$A_L(3)$	$A_D(3)$	$A_R(3)$						$\phi(3)$	$BF(3)$
4			$A_L(4)$	$A_D(4)$	$A_R(4)$					$\phi(4)$	$BF(4)$
5				$A_L(5)$	$A_D(5)$	$A_R(5)$				$\phi(5)$	$BF(5)$
6					$A_L(6)$	$A_D(6)$	$A_R(6)$			$\phi(6)$	$BF(6)$
7						$A_L(7)$	$A_D(7)$	$A_R(7)$		$\phi(7)$	$BF(7)$
8							$A_L(8)$	$A_D(8)$		$\phi(8)$	$BF(8)$

$$A_D(1)\times\phi(1)+A_R(1)\times\phi(2)=BF(1)$$

$$A_L(2)\times\phi(1)+A_D(2)\times\phi(2)+A_R(2)\times\phi(3)=BF(2)$$

$$A_L(3)\times\phi(2)+A_D(3)\times\phi(3)+A_R(3)\times\phi(4)=BF(3)$$

$$A_L(4)\times\phi(3)+A_D(4)\times\phi(4)+A_R(4)\times\phi(5)=BF(4)$$

$$A_L(5)\times\phi(4)+A_D(5)\times\phi(5)+A_R(5)\times\phi(6)=BF(5)$$

$$A_L(6)\times\phi(5)+A_D(6)\times\phi(6)+A_R(6)\times\phi(7)=BF(6)$$

$$A_L(7)\times\phi(6)+A_D(7)\times\phi(7)+A_R(7)\times\phi(8)=BF(7)$$

$$A_L(8)\times\phi(7)+A_D(8)\times\phi(8)=BF(8)$$

$$A_L(i)\times\phi(i-1)+A_D(i)\times\phi(i)+A_R(i)\times\phi(i+1)=BF(i)$$

$$A_L(i)=\frac{1}{\Delta x^2},\ A_D(i)=-\frac{2}{\Delta x^2},\ A_R(i)=\frac{1}{\Delta x^2}$$

一次元熱伝導方程式(4/7)

連立一次方程式の解法:古典的反復法(1)

各要素における線形方程式

$$A_L(i)\times\phi_{i-1}+A_D(i)\times\phi_i+A_R(i)\times\phi_{i+1}=BF(i)\quad(1\le i\le N)$$

$$A_L(i)=\frac{1}{\Delta x^2},\ A_D(i)=-\frac{2}{\Delta x^2},\ A_R(i)=\frac{1}{\Delta x^2}$$

解

$$A_D(i)\times\phi(i)=BF(i)-A_L(i)\times\phi_{i-1}-A_R(i)\times\phi_{i+1}$$

$$\phi(i)=\frac{BF(i)-A_L(i)\times\phi_{i-1}-A_R(i)\times\phi_{i+1}}{A_D(i)}\quad(1\le i\le N)$$

- 各点においてφ(i)の値の変化が無くなるまで反復を繰り返す。
- 前の反復における解を φ⁰(i)とする
- 逆行列を使って解く方法(ガウスの消去法等)もある

一次元熱伝導方程式(5/7)
連立一次方程式の解法:古典的反復法(2)

- ヤコビ法(Jacobi)
  - ϕ_{i-1}およびϕ_{i+1}の値として前の反復における $\phi^0(i-1), \phi^0(i+1)$ を使用する。
  - 収束は遅い。

$$\phi(i) = \frac{BF(i) - A_L(i) \times \phi^0(i-1) - A_R(i) \times \phi^0(i+1)}{A_D(i)}$$

- ガウス＝ザイデル(Gauss-Seidel)法
  - 計算の終了した値 ϕ_{i-1}については最新の値 $\phi(i-1)$, ϕ_{i+1}の値としては前の反復における値 $\phi^0(i+1)$を使用する。
  - Jacobi法より速い。

$$\phi(i) = \frac{BF(i) - A_L(i) \times \phi(i-1) - A_R(i) \times \phi^0(i+1)}{A_D(i)}$$

一次元熱伝導方程式(6/7)

連立一次方程式の解法:古典的反復法(3)

- SOR(Successive-Over Relaxation)法
  - Gauss-Seidel法によって求められた解を ϕ_{GS} とすると, $\phi(i) = \phi^0(i) + \omega\ (\phi_{GS} - \phi^0(i))$

$$\phi_{GS}(i) = \frac{BF(i) - A_L(i) \times \phi(i-1) - A_R(i) \times \phi^0(i+1)}{A_D(i)}$$

$$\phi_{SOR}(i) = \phi^0(i) + \omega \times \left[\phi_{GS}(i) - \phi^0(i)\right]$$

一次元熱伝導方程式(7/7)
連立一次方程式の解法:古典的反復法(4)

- SOR(Successive-Over Relaxation)法(続き)
  - ω=1の場合はGauss-Seidelと同じ，ω>1の場合を過緩和(over relaxation)，ω<1の場合を不足緩和(under relaxation)と呼ぶ。通常 1<ω<2 の値を使用する。
  - ω>1とすることによって収束が加速される場合がある。
    - 値が大きすぎると発散する場合がある。
  - 一次元線形問題における最適値(メッシュ数=N)，Nが大きくなると2に近づく:境界条件等によって変わる

$$\varpi_{opt} \approx \frac{2}{1+\sin\left(\pi / N\right)}$$

サンプルコード：一次元熱伝導方程式

```
$ cd <$S2>
$ pgf90 -O3 heat_jacobi.f -o ja
$ pgf90 -O3 heat_gs.f -o gs
$ pgf90 -O3 heat_sor.f -o sor

$ ./ja, ./sor, ./gs
```

```
$ cd <$S2>
$ pgcc -O3 heat_jacobi.c -o ja
$ pgcc -O3 heat_gs.c -o gs
$ pgcc -O3 heat_sor.c -o sor

$ ./ja, ./sor, ./gs
```

input.dat, cinput.dat

50	N	メッシュ数
1.d0 1.d0	dx, BF	メッシュ幅，体積発熱量
50000	ITERmax	最大反復回数
1.d-7 -1.00	EPS, OMEGA	打切誤差， SORのω

実行例：Jacobi法（./ja）

```
1000     iters,   RESID=   3.911949E-01      PHI(N)=  4.724513E+02
2000     iters,   RESID=   2.350935E-01      PHI(N)=  7.746137E+02
3000     iters,   RESID=   1.406316E-01      PHI(N)=  9.555996E+02
...
29000    iters,   RESID=   2.213721E-07      PHI(N)=  1.225000E+03
30000    iters,   RESID=   1.324140E-07      PHI(N)=  1.225000E+03
30548    iters,   RESID=   9.991504E-08      PHI(N)=  1.225000E+03

1        0.000000E+00      0.000000E+00
2        4.899999E+01      4.900000E+01
3        9.699999E+01      9.700000E+01
4        1.440000E+02      1.440000E+02
5        1.900000E+02      1.900000E+02
...

41       1.180000E+03      1.180000E+03
42       1.189000E+03      1.189000E+03
43       1.197000E+03      1.197000E+03
44       1.204000E+03      1.204000E+03
45       1.210000E+03      1.210000E+03
46       1.215000E+03      1.215000E+03
47       1.219000E+03      1.219000E+03
48       1.222000E+03      1.222000E+03
49       1.224000E+03      1.224000E+03
50       1.225000E+03      1.225000E+03
```

反復回数
最大残差
$\phi(50)$

数値解，解析解

$$\phi = -\frac{1}{2}49^2 + 49.5 \times 49 = -1200.5 + 9850.5 = 1225$$

実行例：Gauss-Seidel法（./gs）

```
1000     iters,   RESID=   4.591084E-01      PHI(N)=  7.785284E+02
2000     iters,   RESID=   1.642708E-01      PHI(N)=  1.065259E+03
3000     iters,   RESID=   5.877313E-02      PHI(N)=  1.167848E+03
...
14000    iters,   RESID=   7.227382E-07      PHI(N)=  1.224999E+03
15000    iters,   RESID=   2.585828E-07      PHI(N)=  1.225000E+03
15925    iters,   RESID=   9.993005E-08      PHI(N)=  1.225000E+03

1        0.000000E+00      0.000000E+00
2        4.899999E+01      4.900000E+01
3        9.699999E+01      9.700000E+01
4        1.440000E+02      1.440000E+02
5        1.900000E+02      1.900000E+02
...

41       1.180000E+03      1.180000E+03
42       1.189000E+03      1.189000E+03
43       1.197000E+03      1.197000E+03
44       1.204000E+03      1.204000E+03
45       1.210000E+03      1.210000E+03
46       1.215000E+03      1.215000E+03
47       1.219000E+03      1.219000E+03
48       1.222000E+03      1.222000E+03
49       1.224000E+03      1.224000E+03
50       1.225000E+03      1.225000E+03
```

反復回数
最大残差
$\phi(50)$

数値解，解析解

実行例：SOR法（./sor）

```
### OMEGA=     1.881838E+00
1000 iters, RESID=     4.921991E-07 PHI(N)=      1.225000E+03
1091 iters, RESID=     9.923362E-08 PHI(N)=      1.225000E+03

1         0.000000E+00      0.000000E+00
2         4.899999E+01      4.900000E+01
3         9.699999E+01      9.700000E+01
4         1.440000E+02      1.440000E+02
5         1.900000E+02      1.900000E+02
...

41        1.180000E+03      1.180000E+03
42        1.189000E+03      1.189000E+03
43        1.197000E+03      1.197000E+03
44        1.204000E+03      1.204000E+03
45        1.210000E+03      1.210000E+03
46        1.215000E+03      1.215000E+03
47        1.219000E+03      1.219000E+03
48        1.222000E+03      1.222000E+03
49        1.224000E+03      1.224000E+03
50        1.225000E+03      1.225000E+03
```

数値解，解析解

OMEGA（この場合は最適値）

反復回数
最大残差
$\phi(50)$

一次元熱伝導方程式：Jacobi法 heat_jacobi.f（1/3）

```
!C
!C   1D Poisson Equation Solver by
!C   Jacobi Method
!C
!C   d/dx(dPHI/dx) + BF = 0
!C   PHI=0@x=0
!C
      program JACOBI_poi
      implicit REAL*8 (A-H,O-Z)

      integer :: N, ITERmax
      real(kind=8) :: dx, RESID, dPHI, dPHImax, BF, EPS
      real(kind=8), dimension(:), allocatable :: PHI, RHS, PHI0
      real(kind=8), dimension(:), allocatable :: rAD, AR, AL

!C
!C-- INIT.
      open  (11, file='input.dat', status='unknown')
       read (11,*) N
       read (11,*) dX, BF
       read (11,*) ITERmax
       read (11,*) EPS
      close (11)
```

一次元熱伝導方程式:Jacobi法 heat_jacobi.f（2/3）

```
      allocate (PHI(N+1), rAD(N), AR(N), AL(N), RHS(N), PHI0(N))

       PHI = 0.d0
       PHI0= 0.d0

       AR = 1.d0/dX
       AL = 1.d0/dX
      rAD = 1.d0/(-2.d0/dX)
      RHS=  -BF * dX

       AL (1)= 0.d0
       AR (1)= 0.d0
      rAD (1)= 1.d0
       RHS(1)= 0.d0

       AR (N)= 0.d0
      rAD (N)= 1.d0/(-1.d0/dX)
```

$$\frac{d^2\phi}{dx^2} + BF = 0, \quad \phi = 0 @ x = 0, \frac{d\phi}{dx} = 0 @ x = x_{\max}$$

一次元熱伝導方程式:Jacobi法 heat_jacobi.f（2/3）

```
      allocate (PHI(N+1), rAD(N), AR(N), AL(N), RHS(N), PHI0(N))

       PHI = 0.d0
       PHI0= 0.d0

       AR = 1.d0/dX
       AL = 1.d0/dX
      rAD = 1.d0/(-2.d0/dX)
      RHS=  -BF * dX

       AL (1)= 0.d0
       AR (1)= 0.d0
      rAD (1)= 1.d0
       RHS(1)= 0.d0

       AR (N)= 0.d0
      rAD (N)= 1.d0/(-1.d0/dX)
```

$$\left(\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \right) \times V + BF \times V = 0$$

$$\left(\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \right) \times \Delta x + BF \times \Delta x = 0$$

一次元熱伝導方程式：Jacobi法 heat_jacobi.f（2/3）

```
      allocate (PHI(N+1), rAD(N), AR(N), AL(N), RHS(N), PHI0(N))

       PHI = 0.d0
       PHI0= 0.d0

       AR = 1.d0/dX
       AL = 1.d0/dX
      rAD = 1.d0/(-2.d0/dX)
      RHS=  -BF * dX

       AL (1)= 0.d0
       AR (1)= 0.d0
      rAD (1)= 1.d0
       RHS(1)= 0.d0

       AR (N)= 0.d0
      rAD (N)= 1.d0/(-1.d0/dX)
```

$$rA_D(i) = \frac{1}{A_D(i)}$$

割り算は計算時間がかかるため

$$\left(\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \right) \times \Delta x + BF \times \Delta x = 0$$

$$\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x} = \left(\frac{1}{\Delta x} \right) \phi_{i-1} - \left(\frac{2}{\Delta x} \right) \phi_i + \left(\frac{1}{\Delta x} \right) \phi_{i+1} = -BF \times \Delta x$$

AL **AD** **AR** **RHS**

境界条件の処理：i=1

```
AL (1)= 0.d0
AR (1)= 0.d0
rAD (1)= 1.d
RHS (1)= 0.d0
```

$$\phi = 0 @ x = 0 \Rightarrow \phi_1 = 0$$

$$\Rightarrow (0)\phi_2 + (1)\phi_1 + (0)\phi_0 = 0$$

AR AD AL RHS

境界条件の処理：i=N

```
AL (N)= 1.d0/dX
AR (N)= 0.d0
rAD(N)= 1.d0/(-1.d0/dX)
```

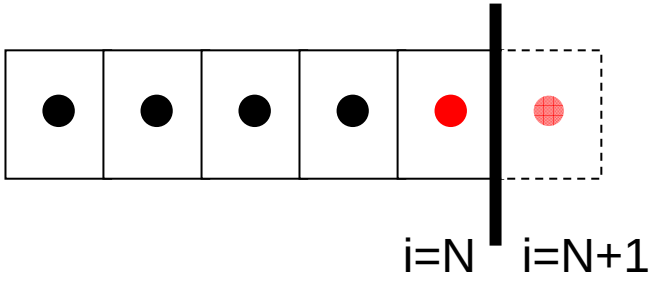
$$\frac{d\phi}{dx} = 0 @ x = x_{\max} \Rightarrow \frac{\phi_{N+1} - \phi_N}{\Delta x} = 0$$

$$\left(\frac{\phi_{N+1} - 2\phi_N + \phi_{N-1}}{\Delta x^2} \right) \times \Delta x + BF \times \Delta x = 0$$

$$\Rightarrow \left(\frac{-\phi_N + \phi_{N-1}}{\Delta x^2} \right) \times \Delta x + BF \times \Delta x = 0$$

$$\Rightarrow (0)\phi_{N+1} + \left(\frac{-1}{\Delta x} \right) \phi_N + \left(\frac{1}{\Delta x} \right) \phi_{N-1} = -BF \times \Delta x$$

AR **AD** **AL** **RHS**



境界面で断熱条件が成立するためには，$\phi_{N+1}=\phi_N$を満たすような仮想的な要素があると都合が良い

一次元熱伝導方程式：Jacobi法 heat_jacobi.f（3/3）

```
!C
!C-- ITERATIONs
      do iter= 1, ITERmax
        dPHImax= -1.d0
        do i= 1, N
          RESID  = RHS(i) - AL(i)*PHI0(i-1) - AR(i)*PHI0(i+1)
          dPHI   = RESID*rAD(i) - PHI0(i)
          dPHImax= dmax1 (dabs(dPHI), dPHImax)
          PHI(i) = PHI0(i) + dPHI
        enddo

        do i= 1, N
          PHI0(i) = PHI(i)
        enddo

        if (dPHImax.lt.EPS) exit
      enddo

!C
!C-- OUTPUT
      Xmax= (dfloat(N-1)+0.5d0)*dX
      do i= 1, N
        XX= dfloat(i-1)*dX
        T = -0.5d0*BF*(XX**2) + BF*Xmax*XX
        write (*,'(i8, 2(1pe16.6))') i, PHI(i), T
      enddo

      end program JACOBI_poi
```

$$\phi(i) = \frac{RHS(i) - A_L(i)\times\phi^0(i-1) - A_R(i)\times\phi^0(i+1)}{A_D(i)}$$

$$= \left[\frac{RHS(i) - A_L(i)\times\phi^0 i-1) - A_R(i)\times\phi^0(i+1)}{A_D(i)} - \phi 0(i) \right] + \phi^0(i)$$

$$T = \phi(analy.) = -\frac{1}{2} BF\, x^2 + BF\, x_{\max}\, x$$

一次元熱伝導方程式：SOR法 heat_sor.f（1/3）

```
!C
!C   1D Poisson Equation Solver by
!C   SOR (Successive Over Relaxation) Method
!C
!C   d/dx(dPHI/dx) + BF = 0
!C   PHI=0@x=0
!C
      program SOR_poi
      implicit REAL*8 (A-H,O-Z)

      integer :: N, ITERmax
      real(kind=8) :: dx, RESID, dPHI, dPHImax, BF, EPS
      real(kind=8), dimension(:), allocatable :: PHI, RHS, PHI0
      real(kind=8), dimension(:), allocatable :: rAD, AR, AL

!C
!C-- INIT.
      open  (11, file='input.dat', status='unknown')
       read (11,*) N
       read (11,*) dX, BF
       read (11,*) ITERmax
       read (11,*) EPS, OMEGA
      close (11)

      if (OMEGA.le.0.d0) then
        PI= 4.d0 * atan(1.d0)
        OMEGA= 2.d0/(1.d0+dsin(PI/dfloat(N)))
      endif
```

一次元熱伝導方程式：SOR法 heat_sor.f（1/3）

```
!C
!C   1D Poisson Equation Solver by
!C   SOR (Successive Over Relaxation) Method
!C
!C   d/dx(dPHI/dx) + BF = 0
!C   PHI=0@x=0
!C
      program SOR_poi
      implicit REAL*8 (A-H,O-Z)

      integer :: N, ITERmax
      real(kind=8) :: dx, OMEGA, RESID, dPHI, dPHImax, BF, EPS
      real(kind=8), dimension(:), allocatable :: PHI, RHS
      real(kind=8), dimension(:), allocatable :: rAD, AR, AL

!C
!C-- INIT.
      open  (11, file='input.dat', status='unknown')
       read (11,*) N
       read (11,*) dX, BF
       read (11,*) ITERmax
       read (11,*) EPS, OMEGA
      close (11)

      if (OMEGA.le.0.d0) then
        PI= 4.d0 * atan(1.d0)
        OMEGA= 2.d0/(1.d0+dsin(PI/dfloat(N)))
      endif
```

ωの最適値
（<0.0が入力された場合）

一次元熱伝導方程式:SOR法 heat_sor.f（2/3）

```
      allocate (PHI(N+1), rAD(N), AR(N), AL(N), RHS(N), PHI0(N))

       PHI = 0.d0
       PHI0= 0.d0

       AR = 1.d0/dX
       AL = 1.d0/dX
      rAD = 1.d0/(-2.d0/dX)
      RHS=  -BF * dX

       AL (1)= 0.d0
       AR (1)= 0.d0
      rAD (1)= 1.d0
       RHS(1)= 0.d0

       AR (N)= 0.d0
      rAD (N)= 1.d0/(-1.d0/dX)
```

$$\frac{d^2\phi}{dx^2} + BF = 0, \quad \phi = 0 @ x = 0, \frac{d\phi}{dx} = 0 @ x = x_{\max}$$

一次元熱伝導方程式:SOR法 heat_sor.f（2/3）

```
      allocate (PHI(N+1), rAD(N), AR(N), AL(N), RHS(N), PHI0(N))

       PHI = 0.d0
       PHI0= 0.d0

       AR = 1.d0/dX
       AL = 1.d0/dX
      rAD = 1.d0/(-2.d0/dX)
      RHS=  -BF * dX

       AL (1)= 0.d0
       AR (1)= 0.d0
      rAD (1)= 1.d0
       RHS(1)= 0.d0

       AR (N)= 0.d0
      rAD (N)= 1.d0/(-1.d0/dX)
```

$$\left(\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \right) \times V + BF \times V = 0$$

$$\left(\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \right) \times \Delta x + BF \times \Delta x = 0$$

一次元熱伝導方程式:SOR法 heat_sor.f（2/3）

```
      allocate (PHI(N+1), rAD(N), AR(N), AL(N), RHS(N), PHI0(N))

       PHI = 0.d0
       PHI0= 0.d0

       AR = 1.d0/dX
       AL = 1.d0/dX
      rAD = 1.d0/(-2.d0/dX)
      RHS=  -BF * dX

       AL (1)= 0.d0
       AR (1)= 0.d0
      rAD (1)= 1.d0
       RHS(1)= 0.d0

       AR (N)= 0.d0
      rAD (N)= 1.d0/(-1.d0/dX)
```

$$rA_D(i) = \frac{1}{A_D(i)}$$

割り算は計算時間がかかるため

$$\left(\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} \right) \times \Delta x + BF \times \Delta x = 0$$

$$\frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x} = \left(\frac{1}{\Delta x} \right) \phi_{i-1} - \left(\frac{2}{\Delta x} \right) \phi_i + \left(\frac{1}{\Delta x} \right) \phi_{i+1} = -BF \times \Delta x$$

AL　AD　AR　RHS

一次元熱伝導方程式：SOR法 heat_sor.f（3/3）

```
!C
!C-- ITERATIONs
      do iter= 1, ITERmax
        dPHImax= -1.d0
        do i= 1, N
          RESID  = RHS(i) - AL(i)*PHI(i-1) - AR(i)*PHI(i+1)
          dPHI   = OMEGA * (RESID*rAD(i)-PHI(i))
          dPHImax= dmax1 (dabs(dPHI), dPHImax)
          PHI(i) = PHI(i) + dPHI
        enddo

        if (dPHImax.lt.EPS) exit
      enddo

!C
!C-- OUTPUT
      Xmax= (dfloat(N-1)+0.5d0)*dX
      do i= 1, N
        XX= dfloat(i-1)*dX
        T = -0.5d0*BF*(XX**2) + BF*Xmax*XX
        write (*,'(i8, 2(1pe16.6))') i, PHI(i), T
      enddo

      end program SOR_poi
```

$$\phi_{GS}(i) = \frac{BF(i) - A_L(i)\times\phi(i-1) - A_R(i)\times\phi^0(i+1)}{A_D(i)}$$

$$\phi_{SOR}(i) = \phi^0(i) + \omega\times\left[\phi_{GS}(i) - \phi^0(i)\right]$$

JacobiとSOR

JACOBI

```
      do iter= 1, ITERmax
        do i= 1, N
          RESID  = RHS(i) - AL(i)*PHI0(i-1) - AR(i)*PHI0(i+1)
          dPHI   = RESID*rAD(i) - PHI0(i)
          PHI(i) = PHI0(i) + dPHI
        enddo

        do i= 2, N
          PHI0(i) = PHI(i)
        enddo
      enddo
```

反復の間ϕの値は不変(ϕ^0)

SOR

```
      do iter= 1, ITERmax
        do i= 1, N
          RESID  = RHS(i) - AL(i)*PHI(i-1) - AR(i)*PHI(i+1)
          dPHI   = OMEGA * (RESID*rAD(i)-PHI(i))
          PHI(i) = PHI(i) + dPHI
        enddo
      enddo
```

反復の間ϕの値は常に
最新値を使用

動作確認

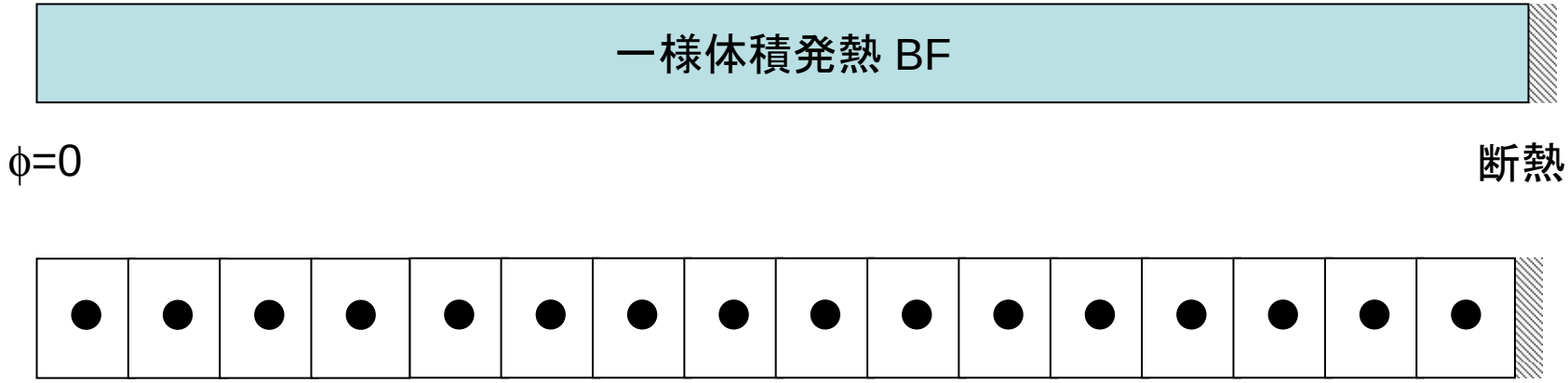
- ./ja，./gs，./sor をランさせる。
- 「input.dat」,「cinput.dat」を変更して ./sor 実行
  - OMEGA=1.00とした場合の収束回数が ./gs の場合と一致することを確認
  - OMEGAを1.00から増加させると，収束回数が減少することを確認
  - OMEGAが最適値（OMEGA<0とした場合）を上回る場合の収束回数
    - 実際はOMEGA=1.94程度が最適値

- 差分法による一次元熱伝導方程式ソルバー
  - 概要
  - 並列化にあたって:データ構造
- 1対1通信とは
- 1対1通信の実装例
  - 一次元問題
  - 二次元問題
- 差分法による一次元熱伝導方程式ソルバーの並列化

一次元熱伝導方程式ソルバーの並列化

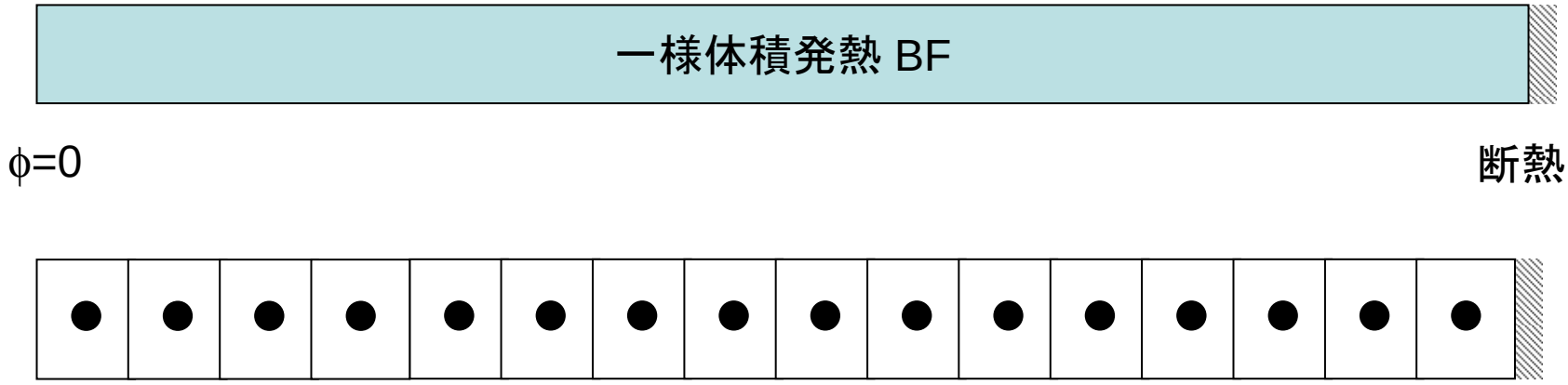
$$\frac{d^2\phi}{dx^2} + BF = 0, \quad \phi = 0 @ x = 0, \frac{d\phi}{dx} = 0 @ x = x_{\max}$$

$$\phi = -\frac{1}{2} BF\, x^2 + BF\, x_{\max}\, x$$



一次元熱伝導方程式ソルバーの並列化

- 全体データと局所データ
- 例えば，NG=16の問題を4PEで実行する場合，各PEにおいてはNL=16/4=4となる

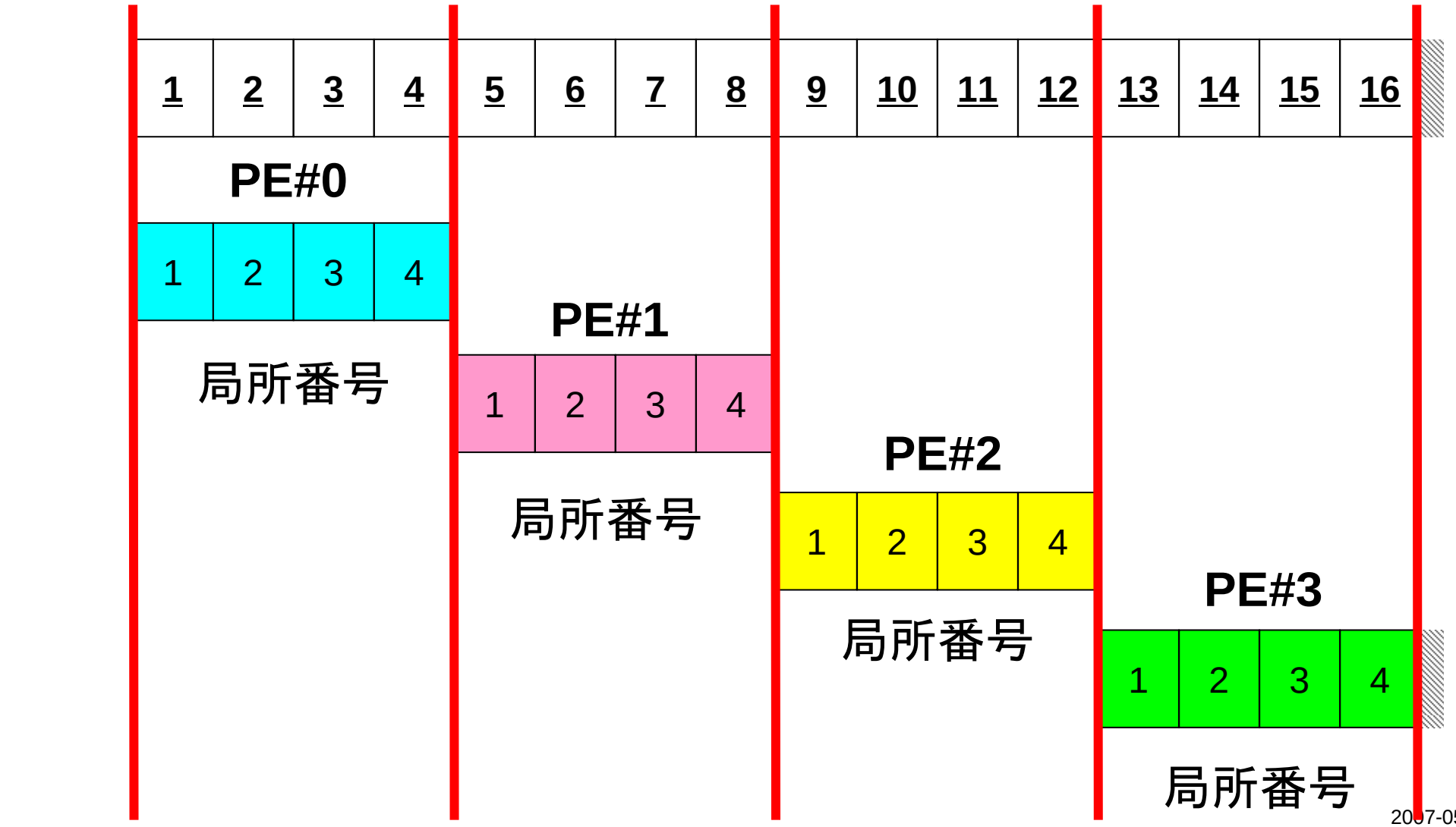


全体データと局所データ
NG=16, PE#=4

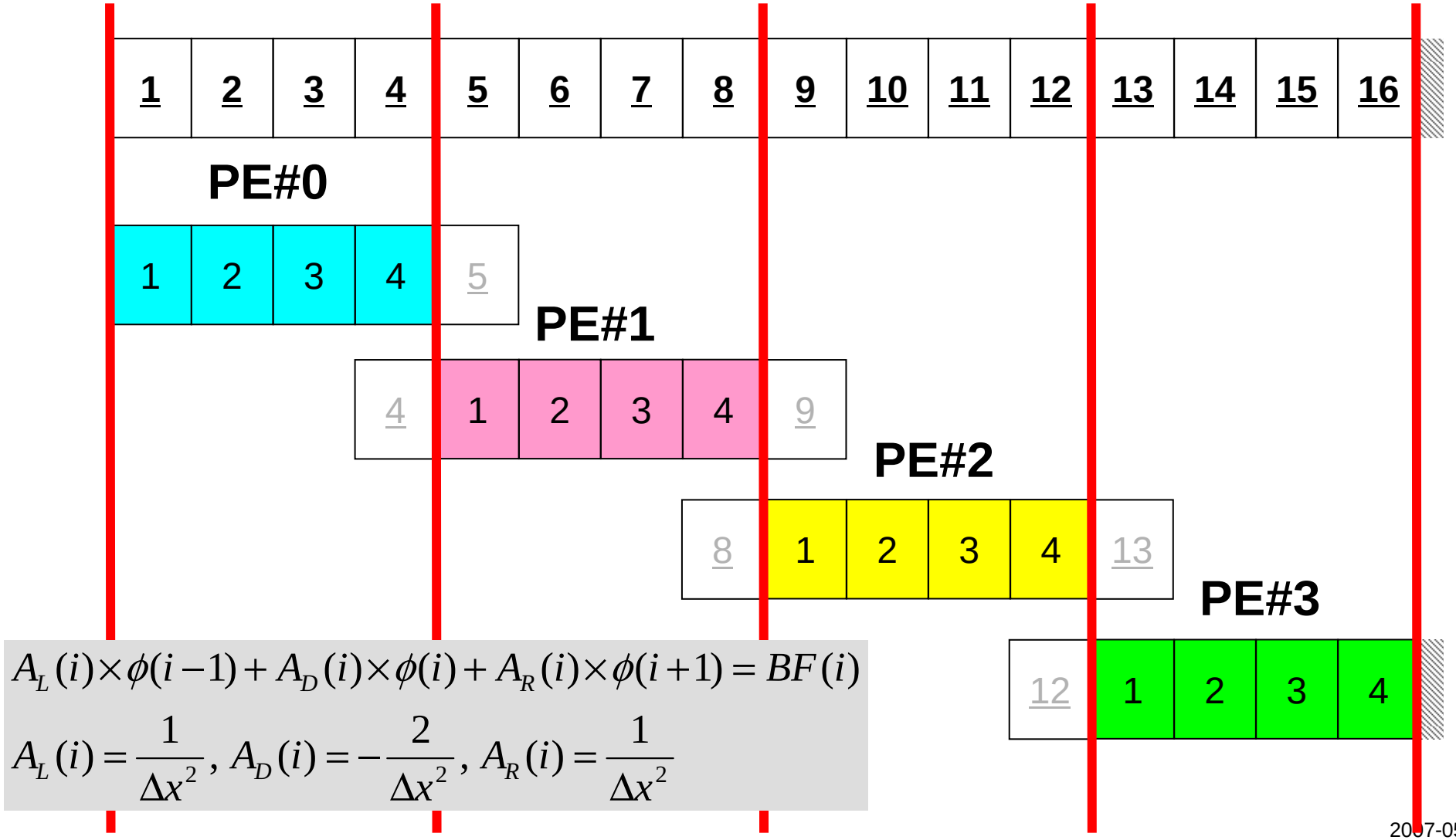
全体番号

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

全体データと局所データ
NG=16, PE#=4



差分法のオペレーションを実施するためには隣接要素の情報が必要



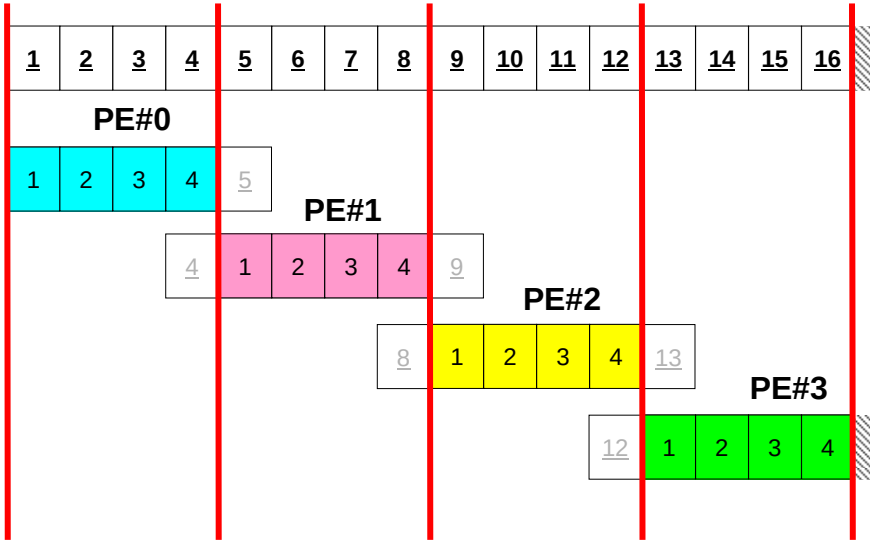
$$A_L(i)\times\phi(i-1)+A_D(i)\times\phi(i)+A_R(i)\times\phi(i+1)=BF(i)$$

$$A_L(i)=\frac{1}{\Delta x^2},\ A_D(i)=-\frac{2}{\Delta x^2},\ A_R(i)=\frac{1}{\Delta x^2}$$

一次元熱伝導方程式ソルバーの並列化

- 全体データと局所データ
- 差分法のオペレーションには隣接要素の値が必要
  - 領域（PE）の境界では隣接領域に属する要素からの情報が必要
  - 例えば，PE#1での計算を完結させるためには，PE#0の「4」番，PE#2の「1」番要素の情報が必要

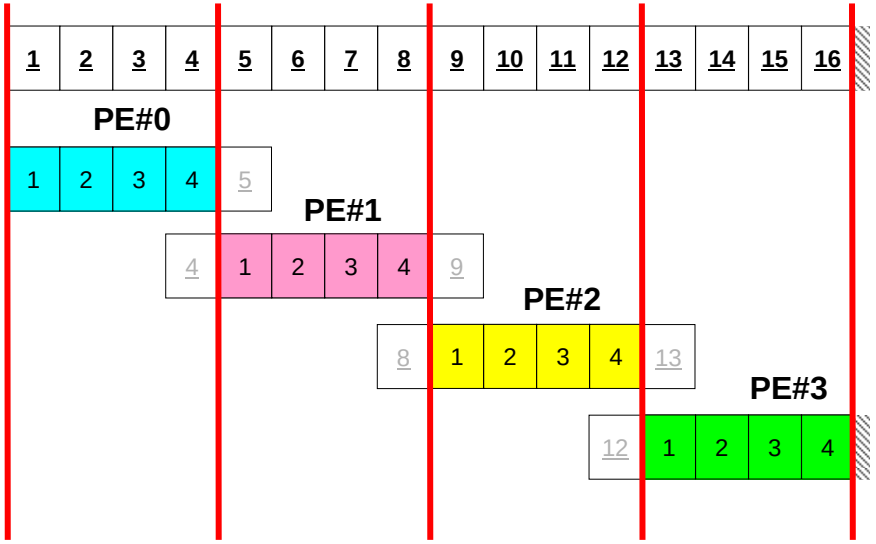
- 隣接領域からの情報を考慮可能なデータ構造が必要
  - それほど単純では無い・・・



- 差分法による一次元熱伝導方程式ソルバー
  - 概要
  - 並列化にあたって:データ構造
- 1対1通信とは
- 1対1通信の実装例
  - 一次元問題
  - 二次元問題
- 差分法による一次元熱伝導方程式ソルバーの並列化

1対1通信とは？

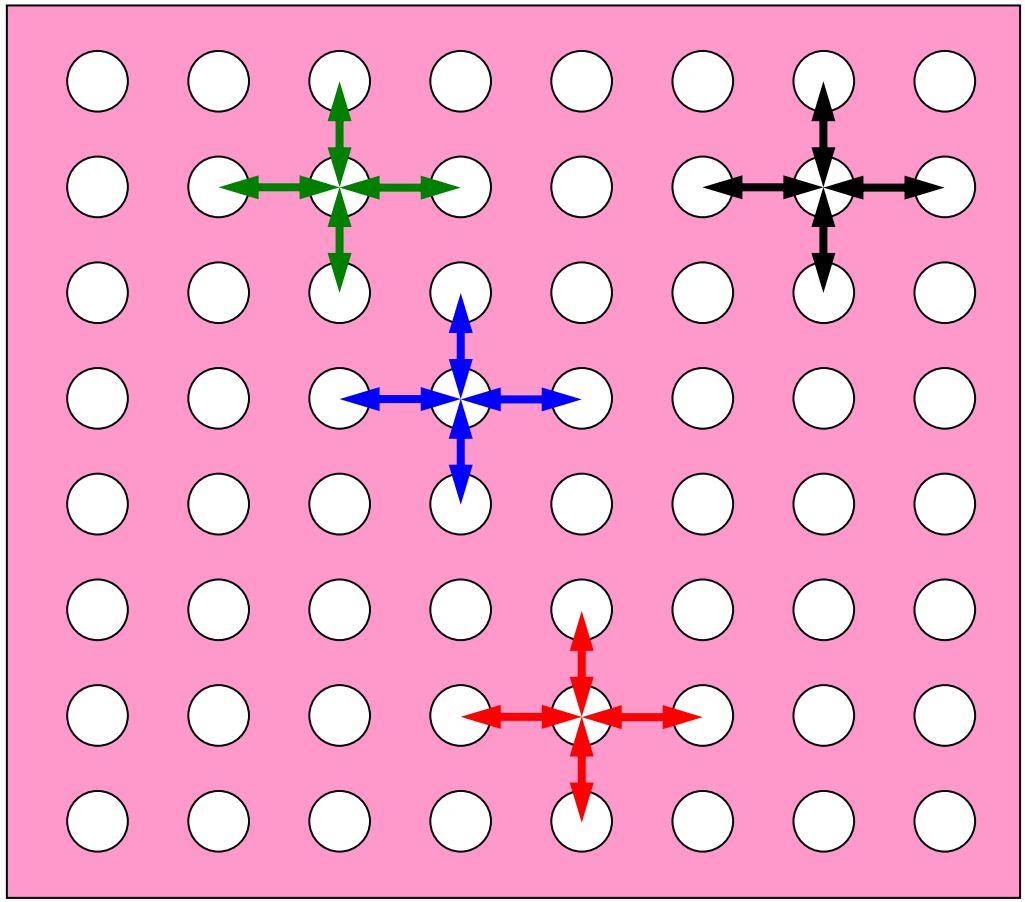
- グループ通信：Collective Communication
  - MPI_Reduce, MPI_Scatter/Gather など
  - 同じコミュニケータ内の全プロセスと通信する
  - 適用分野
    - 境界要素法，スペクトル法，分子動力学等グローバルな相互作用のある手法
    - 内積，最大値などのオペレーション
- 1対1通信：Point-to-Point
  - MPI_Send, MPI_Receive
  - 特定のプロセスとのみ通信がある
    - 隣接領域
  - 適用分野
    - 差分法，有限要素法などローカルな情報を使う手法



グループ通信，1対1通信

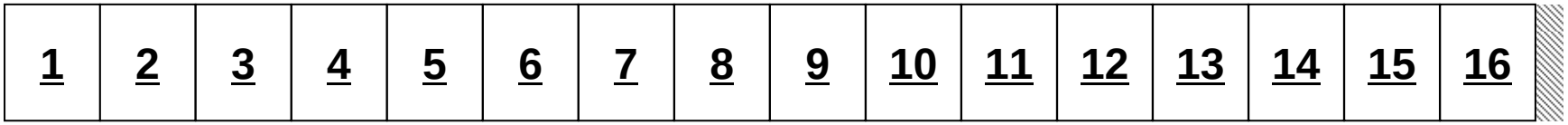
近接PE（領域）のみとの相互作用

差分法，有限要素法



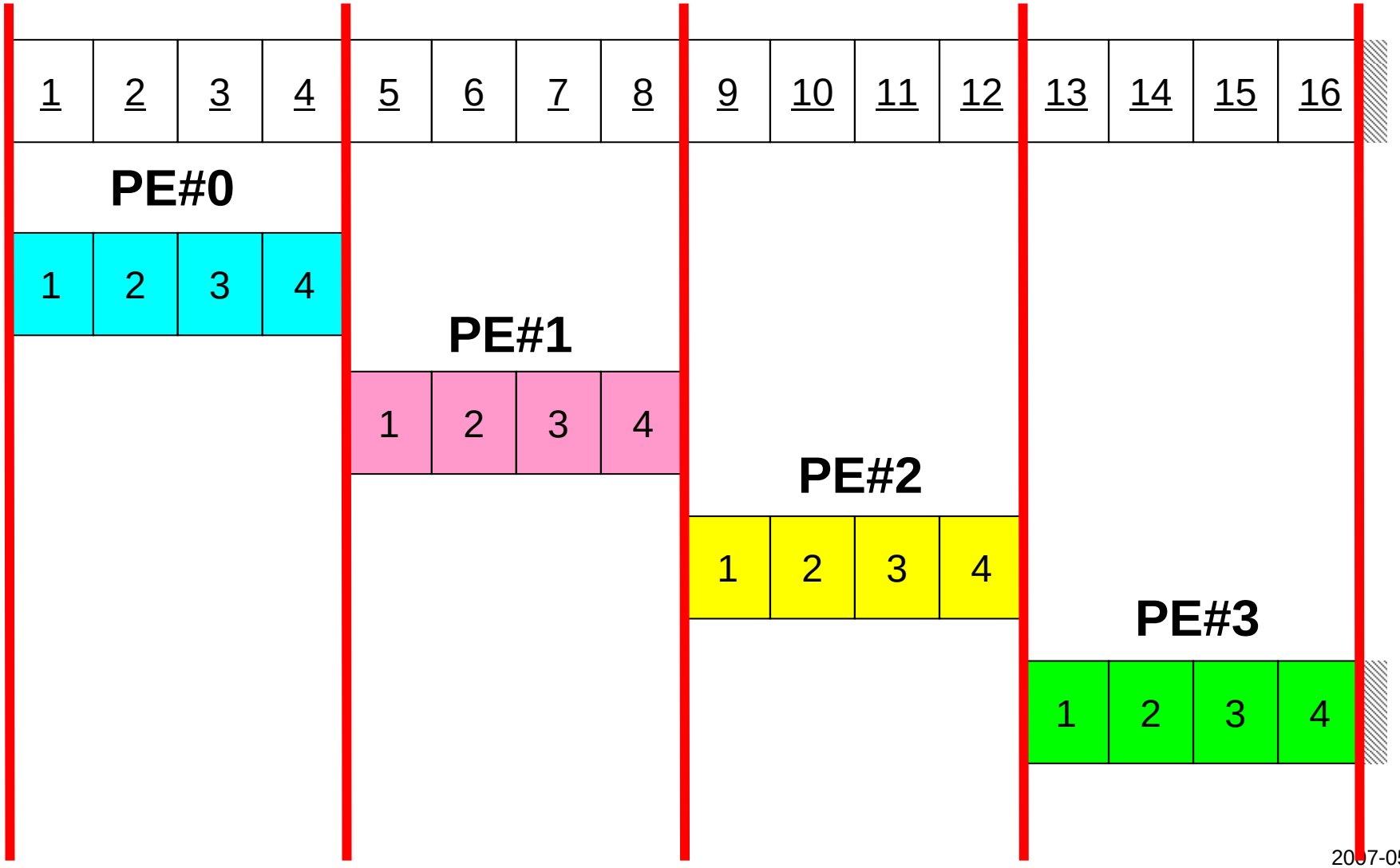
1対1通信が必要になる場面：1D中央差分

NG=16, PE#=4



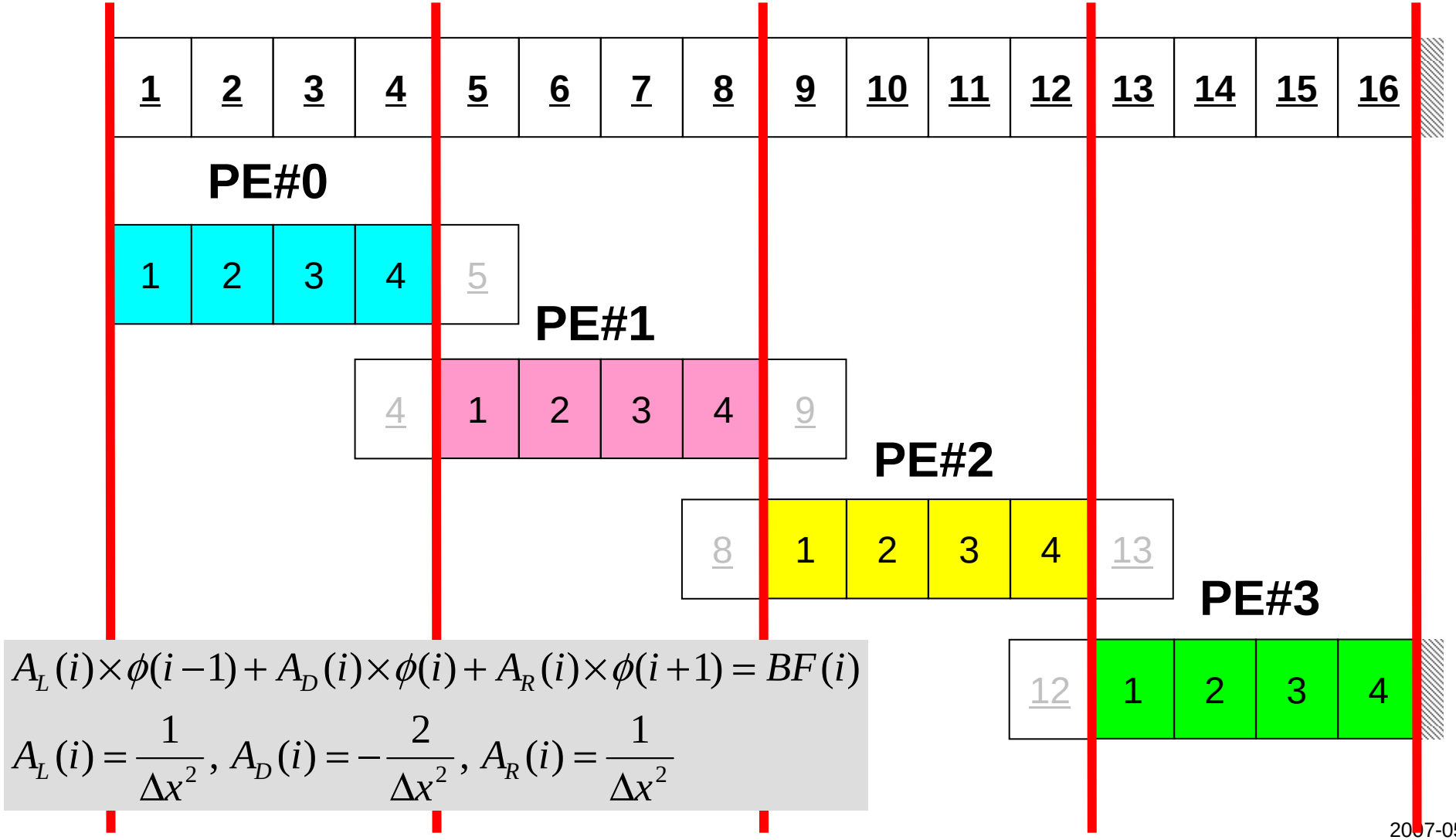
1対1通信が必要になる場面：1D中央差分

NG=16, PE#=4



1対1通信が必要になる場面：1D中央差分

差分法のオペレーションのためには隣接要素の情報が必要

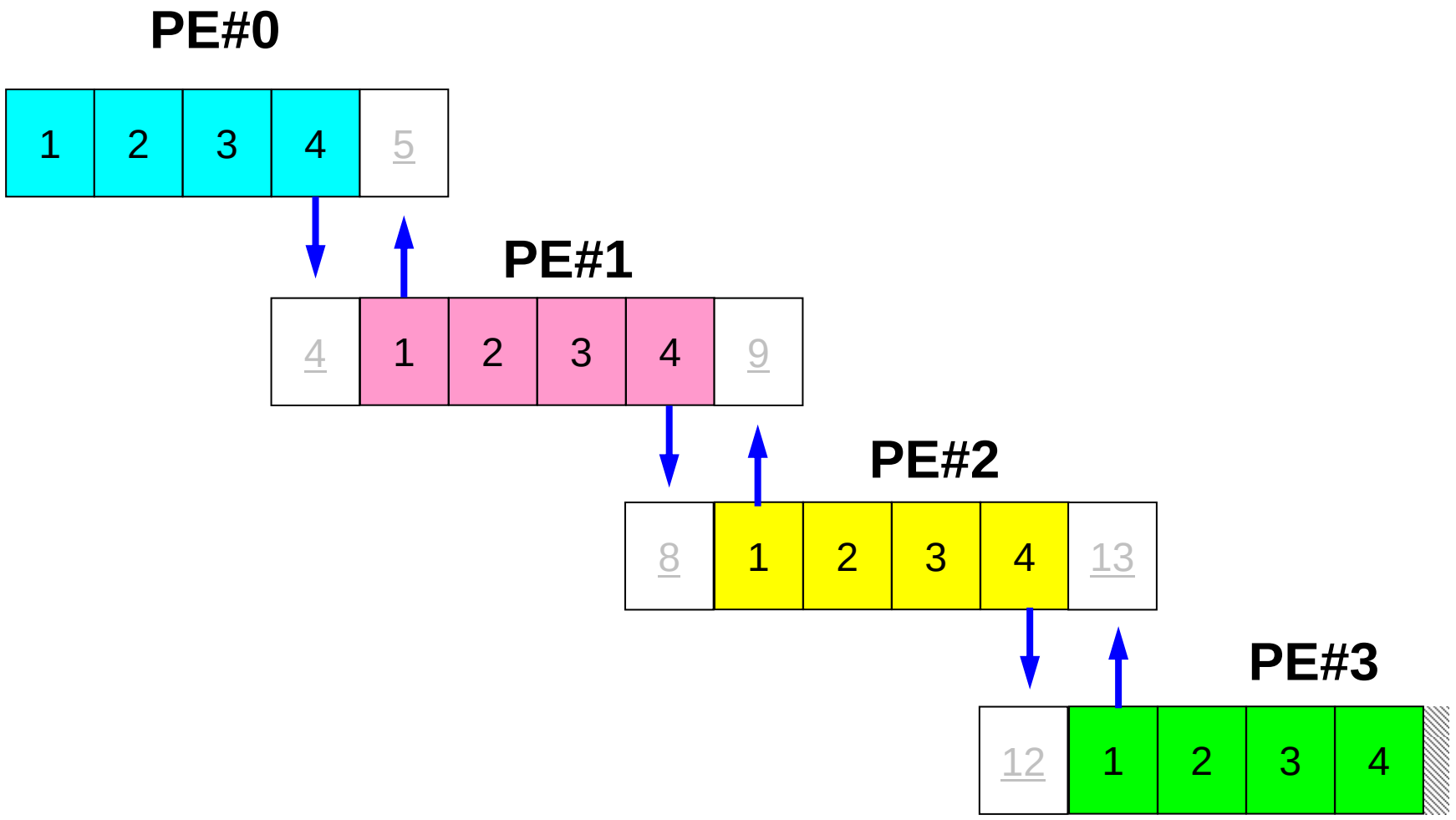


$$A_L(i)\times\phi(i-1)+A_D(i)\times\phi(i)+A_R(i)\times\phi(i+1)=BF(i)$$

$$A_L(i)=\frac{1}{\Delta x^2},\ A_D(i)=-\frac{2}{\Delta x^2},\ A_R(i)=\frac{1}{\Delta x^2}$$

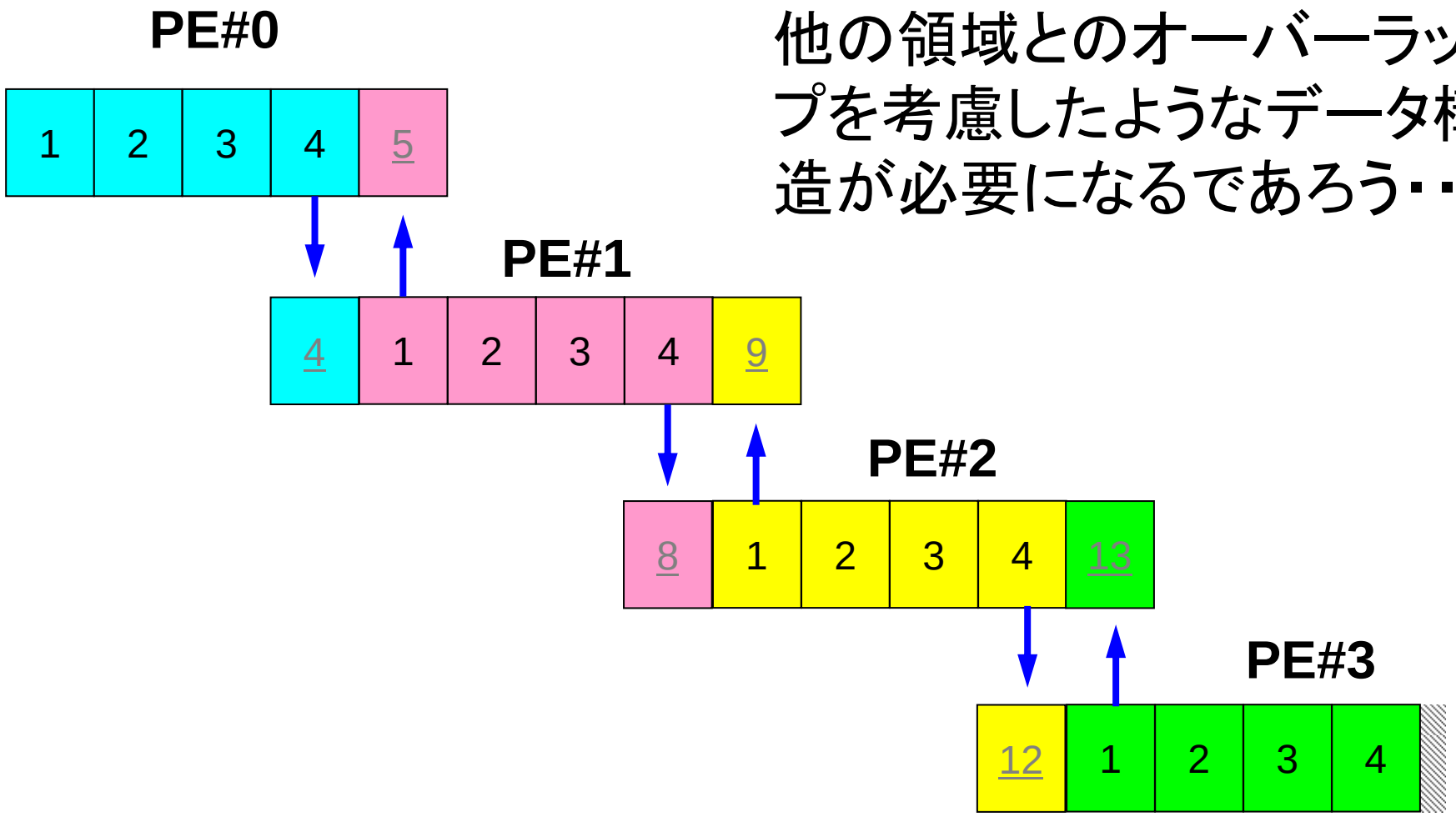
1対1通信が必要になる場面：1D中央差分

差分法のオペレーションのためには隣接要素の情報が必要



1対1通信が必要になる場面：1D中央差分

差分法のオペレーションのためには隣接要素の情報が必要

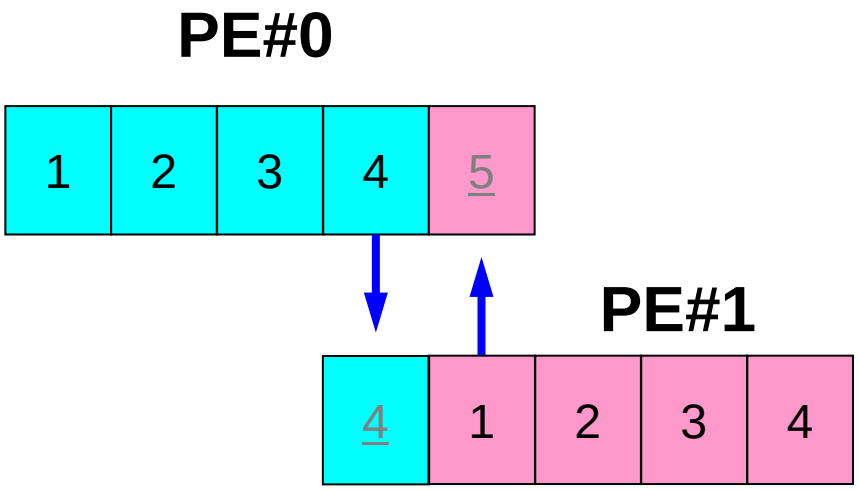


他の領域とのオーバーラップを考慮したようなデータ構造が必要になるであろう・・・

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。

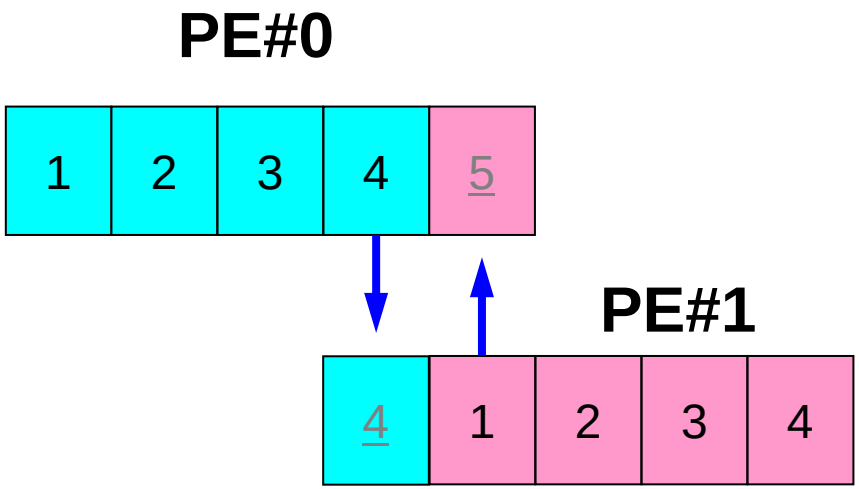
MPI_SEND/MPI_RECV




```
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：実はcenjuではOK

MPI_SEND/MPI_RECV（続き）

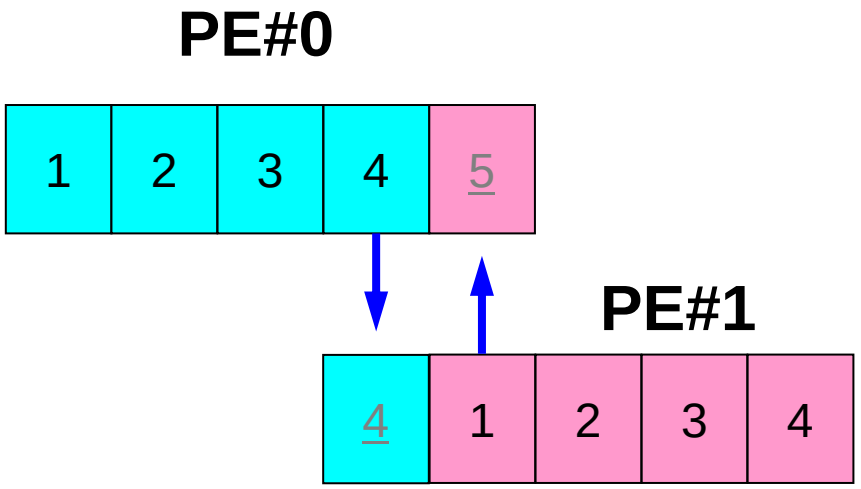



```
if (my_rank.eq.0) NEIB_ID=1
if (my_rank.eq.1) NEIB_ID=0

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

通信識別子（request handle）：request

- **call MPI_ISEND**
  (sendbuf,count,datatype,dest,tag,comm,request, ierr)
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，送信メッセージの種類を区別するときに使用。通常は「0」でよい。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_WAITALLで使用。（配列：サイズは同期する必要のある「MPI_ISEND」呼び出し数（通常は隣接プロセス数など））
  - **ierr** 整数 O 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)
  - **recvbuf** 任意 I 受信バッファの先頭アドレス，
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，受信メッセージの種類を区別するときに使用。通常は「0」でよい。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_WAITALLで使用。（配列：サイズは同期する必要のある「MPI_IRECV」呼び出し数（通常は隣接プロセス数など））：C言語については後述
  - **ierr** 整数 O 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 O 状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE,count））
    MPI_STATUS_SIZE: “mpif.h”,”mpi.h”で定められるパラメータ：C言語については後述
  - **ierr** 整数 O 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 O 状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE,count））
    MPI_STATUS_SIZE: “mpif.h”,”mpi.h”で定められるパラメータ
  - **ierr** 整数 O 完了コード

- 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

- **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）
  - **sendtag** 整数 I 受信用メッセージタグ，送信メッセージの種類を区別するときに使用。
    通常は「0」でよい。
  - **comm** 整数 I コミュニケータを指定する
  - **status** 整数 O 状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE））
    MPI_STATUS_SIZE: “mpif.h”で定められるパラメータ
    C言語については後述
  - **ierr** 整数 O 完了コード

通信識別子，状況オブジェクト配列の定義の仕方（FORTRAN）

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

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

- **MPI_Sendrecv: status**

```
integer status (MPI_STATUS_SIZE)
```

通信識別子，状況オブジェクト配列の定義の仕方（C）：特殊な変数の型がある

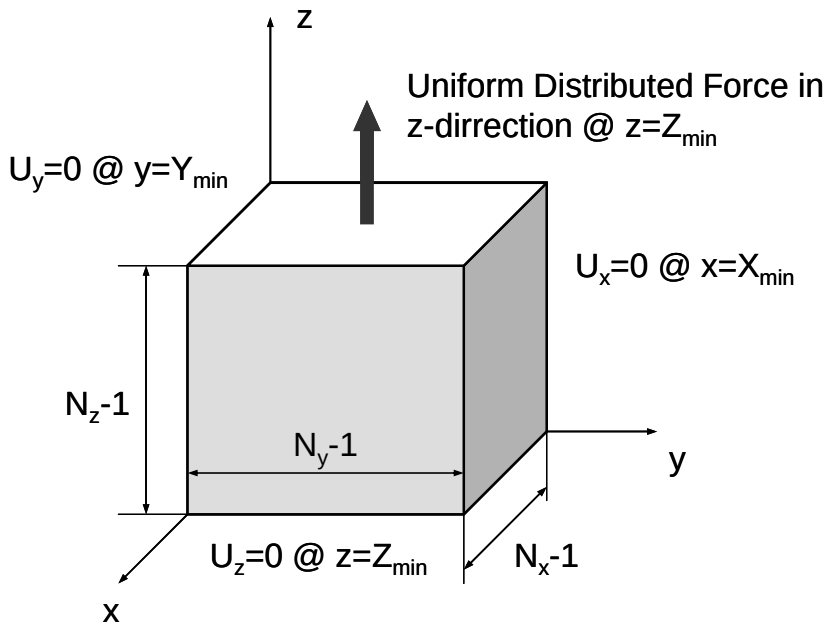
- MPI_Isend: request
- MPI_Irecv: request
- MPI_Waitall: request, status

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

- MPI_Sendrecv: status

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

ISEND-IRECV vs. SENDRECV



3×128×128×64= 3,145,728 DOF
一様物性，境界条件
32 PE

ISEND-IRECV

```
### elapsed    :      6.867188E+01
### comm.       :      1.945313E+00
### work ratio :      9.716724E+01
```

SENDRECV

```
### elapsed    :      7.352344E+01
### comm.       :      5.371094E+00
### work ratio :      9.269472E+01
```

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

```
$> mpif90 -O3 ex1-1.f
$> mpirun -np 2 a.out
```

```
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), 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 C

Isend/Irecv/Waitall

```
$> mpicc -O3 ex1-1.c
$> mpirun -np 2 a.out
```

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

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

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

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

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

利用例(1):スカラー送受信 FORTRAN

SendRecv

```
$> mpif90 -O3 ex1-2.f
$> mpirun -np 2 a.out
```

```
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(1):スカラー送受信 C

SendRecv

```
$> mpicc -O3 ex1-2.c
$> mpirun -np 2 a.out
```

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

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

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

        StatSR = malloc(sizeof(MPI_Status));

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

        MPI_Finalize();
        return 0;
}
```

利用例(2):配列の送受信(1/4)

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

- 演習:プログラムを作成して見よう！

演　習

- VEC(:)の初期状態を以下のようにする:
  - PE#0　VEC(1-36)= 100
  - PE#1　VEC(1-36)= 200

- 次ページのような結果になることを確認せよ

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

予測される結果

```
0 #BEFORE#  1      100.
0 #BEFORE#  2      100.
0 #BEFORE#  3      100.
0 #BEFORE#  4      100.
0 #BEFORE#  5      100.
0 #BEFORE#  6      100.
0 #BEFORE#  7      100.
0 #BEFORE#  8      100.
0 #BEFORE#  9      100.
0 #BEFORE# 10      100.
0 #BEFORE# 11      100.
0 #BEFORE# 12      100.
0 #BEFORE# 13      100.
0 #BEFORE# 14      100.
0 #BEFORE# 15      100.
0 #BEFORE# 16      100.
0 #BEFORE# 17      100.
0 #BEFORE# 18      100.
0 #BEFORE# 19      100.
0 #BEFORE# 20      100.
0 #BEFORE# 21      100.
0 #BEFORE# 22      100.
0 #BEFORE# 23      100.
0 #BEFORE# 24      100.
0 #BEFORE# 25      100.
0 #BEFORE# 26      100.
0 #BEFORE# 27      100.
0 #BEFORE# 28      100.
0 #BEFORE# 29      100.
0 #BEFORE# 30      100.
0 #BEFORE# 31      100.
0 #BEFORE# 32      100.
0 #BEFORE# 33      100.
0 #BEFORE# 34      100.
0 #BEFORE# 35      100.
0 #BEFORE# 36      100.
```

```
0 #AFTER #  1      100.
0 #AFTER #  2      100.
0 #AFTER #  3      100.
0 #AFTER #  4      100.
0 #AFTER #  5      100.
0 #AFTER #  6      100.
0 #AFTER #  7      100.
0 #AFTER #  8      100.
0 #AFTER #  9      100.
0 #AFTER # 10      100.
0 #AFTER # 11      100.
0 #AFTER # 12      200.
0 #AFTER # 13      200.
0 #AFTER # 14      200.
0 #AFTER # 15      200.
0 #AFTER # 16      200.
0 #AFTER # 17      200.
0 #AFTER # 18      200.
0 #AFTER # 19      200.
0 #AFTER # 20      200.
0 #AFTER # 21      200.
0 #AFTER # 22      200.
0 #AFTER # 23      200.
0 #AFTER # 24      200.
0 #AFTER # 25      200.
0 #AFTER # 26      200.
0 #AFTER # 27      200.
0 #AFTER # 28      200.
0 #AFTER # 29      200.
0 #AFTER # 30      200.
0 #AFTER # 31      200.
0 #AFTER # 32      200.
0 #AFTER # 33      200.
0 #AFTER # 34      200.
0 #AFTER # 35      200.
0 #AFTER # 36      200.
```

```
1 #BEFORE#  1      200.
1 #BEFORE#  2      200.
1 #BEFORE#  3      200.
1 #BEFORE#  4      200.
1 #BEFORE#  5      200.
1 #BEFORE#  6      200.
1 #BEFORE#  7      200.
1 #BEFORE#  8      200.
1 #BEFORE#  9      200.
1 #BEFORE# 10      200.
1 #BEFORE# 11      200.
1 #BEFORE# 12      200.
1 #BEFORE# 13      200.
1 #BEFORE# 14      200.
1 #BEFORE# 15      200.
1 #BEFORE# 16      200.
1 #BEFORE# 17      200.
1 #BEFORE# 18      200.
1 #BEFORE# 19      200.
1 #BEFORE# 20      200.
1 #BEFORE# 21      200.
1 #BEFORE# 22      200.
1 #BEFORE# 23      200.
1 #BEFORE# 24      200.
1 #BEFORE# 25      200.
1 #BEFORE# 26      200.
1 #BEFORE# 27      200.
1 #BEFORE# 28      200.
1 #BEFORE# 29      200.
1 #BEFORE# 30      200.
1 #BEFORE# 31      200.
1 #BEFORE# 32      200.
1 #BEFORE# 33      200.
1 #BEFORE# 34      200.
1 #BEFORE# 35      200.
1 #BEFORE# 36      200.
```

```
1 #AFTER #  1      200.
1 #AFTER #  2      200.
1 #AFTER #  3      200.
1 #AFTER #  4      200.
1 #AFTER #  5      200.
1 #AFTER #  6      200.
1 #AFTER #  7      200.
1 #AFTER #  8      200.
1 #AFTER #  9      200.
1 #AFTER # 10      200.
1 #AFTER # 11      200.
1 #AFTER # 12      200.
1 #AFTER # 13      200.
1 #AFTER # 14      200.
1 #AFTER # 15      200.
1 #AFTER # 16      200.
1 #AFTER # 17      200.
1 #AFTER # 18      200.
1 #AFTER # 19      200.
1 #AFTER # 20      200.
1 #AFTER # 21      200.
1 #AFTER # 22      200.
1 #AFTER # 23      200.
1 #AFTER # 24      200.
1 #AFTER # 25      200.
1 #AFTER # 26      100.
1 #AFTER # 27      100.
1 #AFTER # 28      100.
1 #AFTER # 29      100.
1 #AFTER # 30      100.
1 #AFTER # 31      100.
1 #AFTER # 32      100.
1 #AFTER # 33      100.
1 #AFTER # 34      100.
1 #AFTER # 35      100.
1 #AFTER # 36      100.
```

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,0,…,req_send,…)
  call MPI_Irecv (VEC(12),25,MPI_DOUBLE_PRECISION,0,…,req_recv,…)
endif

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

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

これでも良いが，操作が煩雑
SPMDらしくない
汎用性が無い

利用例(2):配列の送受信(3/4)

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

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

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

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

一気にSPMDらしくなる

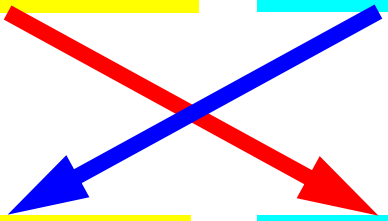
利用例(2):配列の送受信(4/4)

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

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

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

配列の送受信:注意

```
#PE0
  send:
    VEC(start_send)~
    VEC(start_send+length_send-1)
```

```
#PE1
  send:
    VEC(start_send)~
    VEC(start_send+length_send-1)
```



```
#PE0
  recv:
    VEC(start_recv)~
    VEC(start_recv+length_recv-1)
```

```
#PE1
  recv:
    VEC(start_recv)~
    VEC(start_recv+length_recv-1)
```

- 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

次回の予定

- 差分法による一次元熱伝導方程式ソルバー
  - 概要
  - 並列化にあたって:データ構造
- 1対1通信とは
- 1対1通信の実装例
  - 一次元問題
  - 二次元問題
- 差分法による一次元熱伝導方程式ソルバーの並列化
  - 課題S2