MPIによるプログラミング概要（Ⅱ）
課題S2出題

2011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）

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

- CE04～CE08
  - 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
  - MPI
  - 一次元問題
  - 二次元問題
- 1対1通信の実装例
- 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 <$T-EPS> 各自作成したディレクトリ

FORTRAN
>$ cp /home/t00000/EPSsummer/F/s2-f.tar .
>$ tar xvf s2-f.tar

C
>$ cp /home/t00000/EPSsummer/C/s2-c.tar .
>$ tar xvf s2-c.tar
```

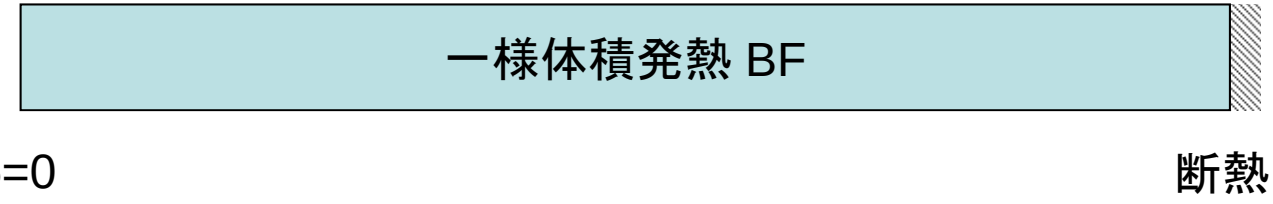
直下に「S2」というディレクトリができている。
<$T-EPS>/S2を<$T-S2>と呼ぶ。

一次元熱伝導方程式

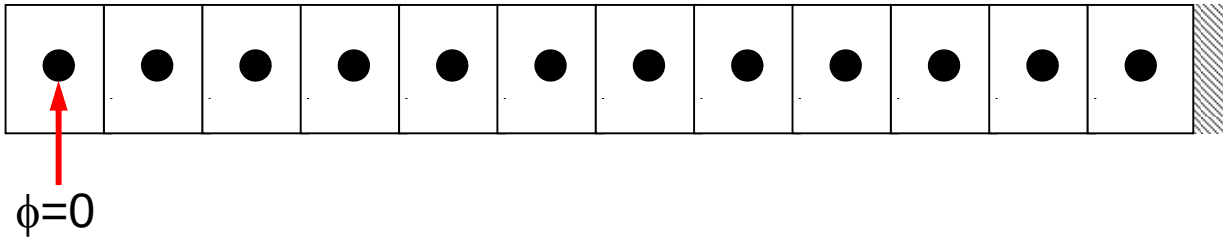
支配方程式：熱伝導率＝1（一様）

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

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



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



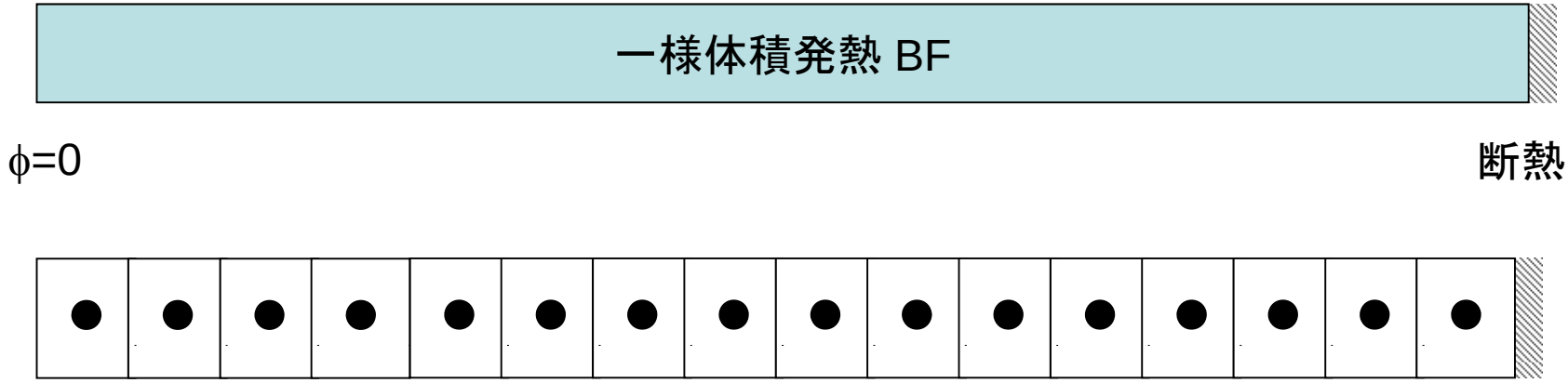
断熱となっているのはこの面，しかし温度は計算されない

- 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
  - MPI
  - 一次元問題
  - 二次元問題
- 1対1通信の実装例
- 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

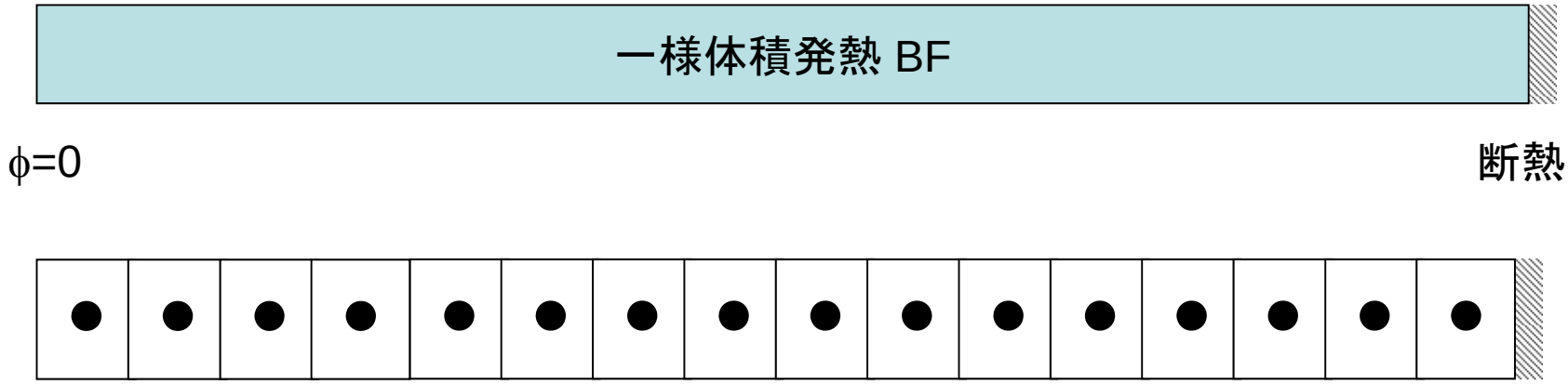
$$\frac{d^2\phi}{dx^2} + BF = 0, \quad \phi = 0 @ x = 0, \frac{d\phi}{dx} = 0 @ x = x_{\max}$$

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



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

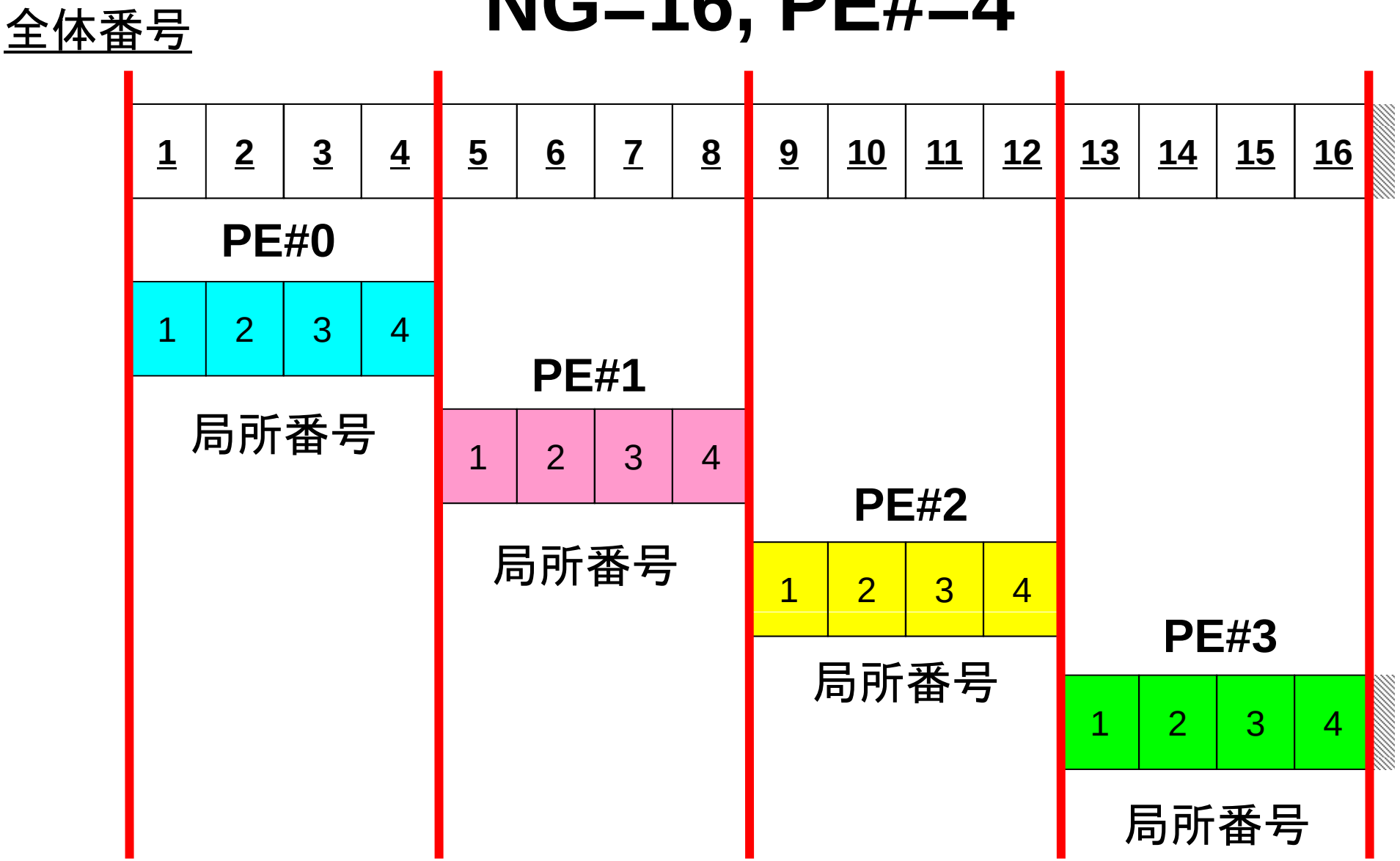


全体データと局所データ
NG=16, PE#=4

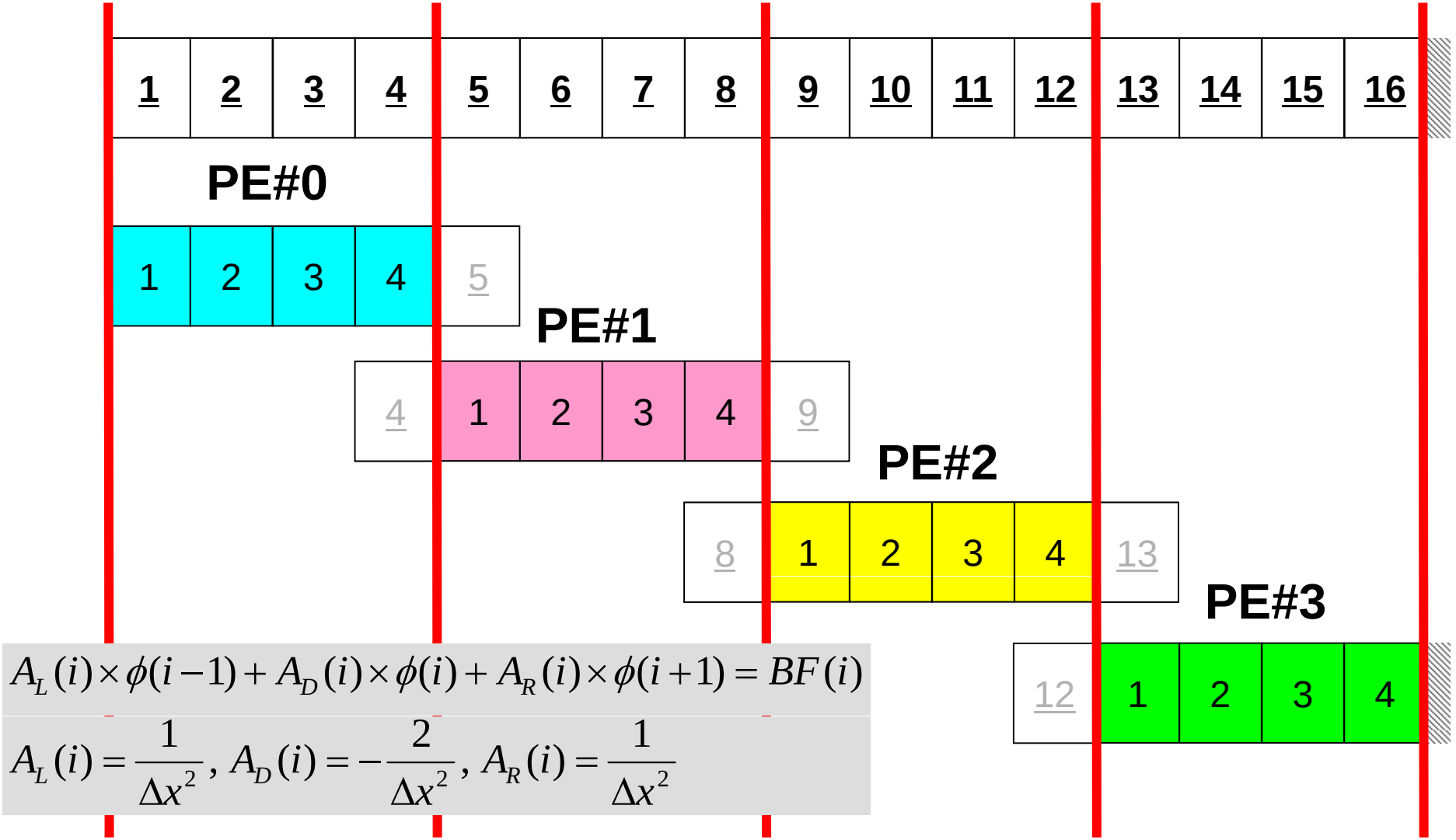
全体番号

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



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

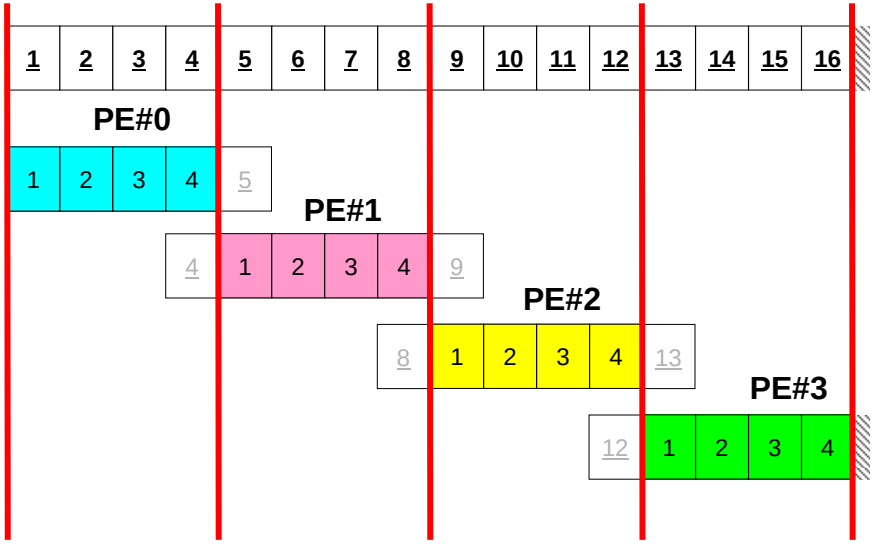
$$A_L(i)=\frac{1}{\Delta x^2},\ A_D(i)=-\frac{2}{\Delta x^2},\ A_R(i)=\frac{1}{\Delta x^2}$$

- 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
  - MPI
  - 一次元問題
  - 二次元問題
- 1対1通信の実装例
- 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

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

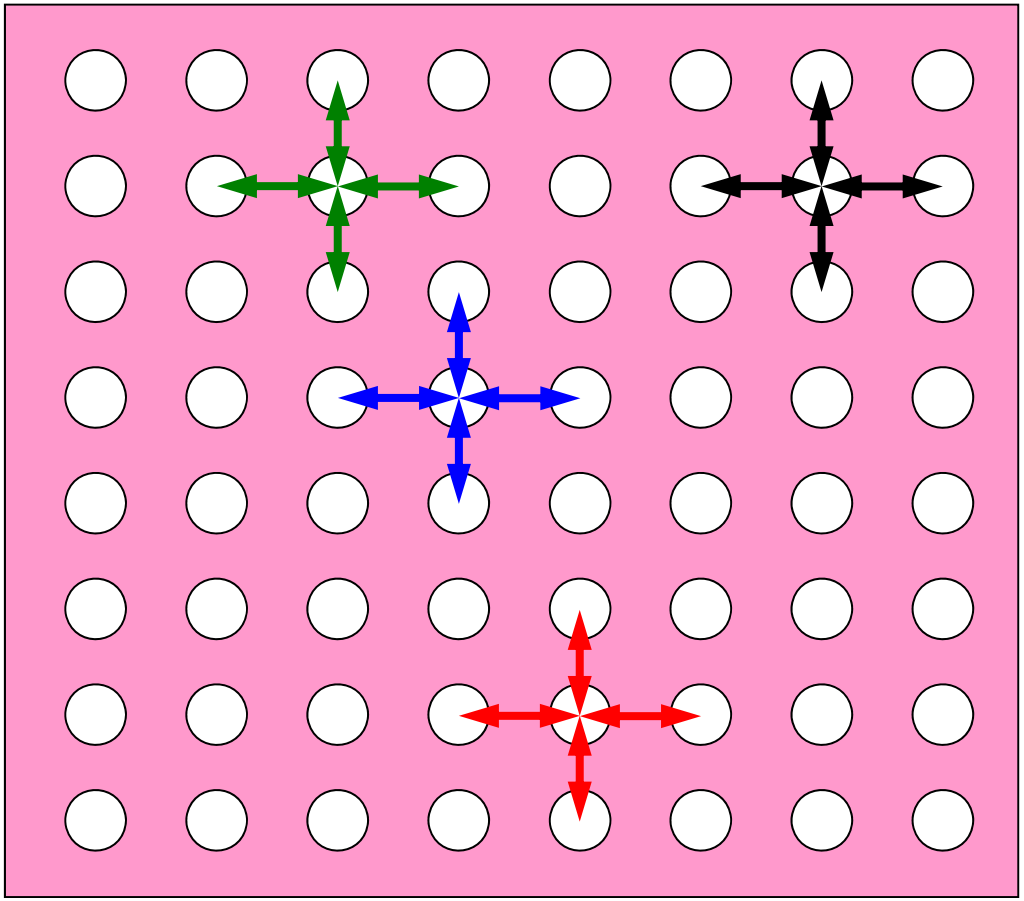
- 1対1通信：Point-to-Point
  - MPI_Send, MPI_Receive
  - 特定のプロセスとのみ通信がある
    - 隣接領域
  - 適用分野
    - 差分法，有限要素法などローカルな情報を使う手法



グループ通信，1対1通信

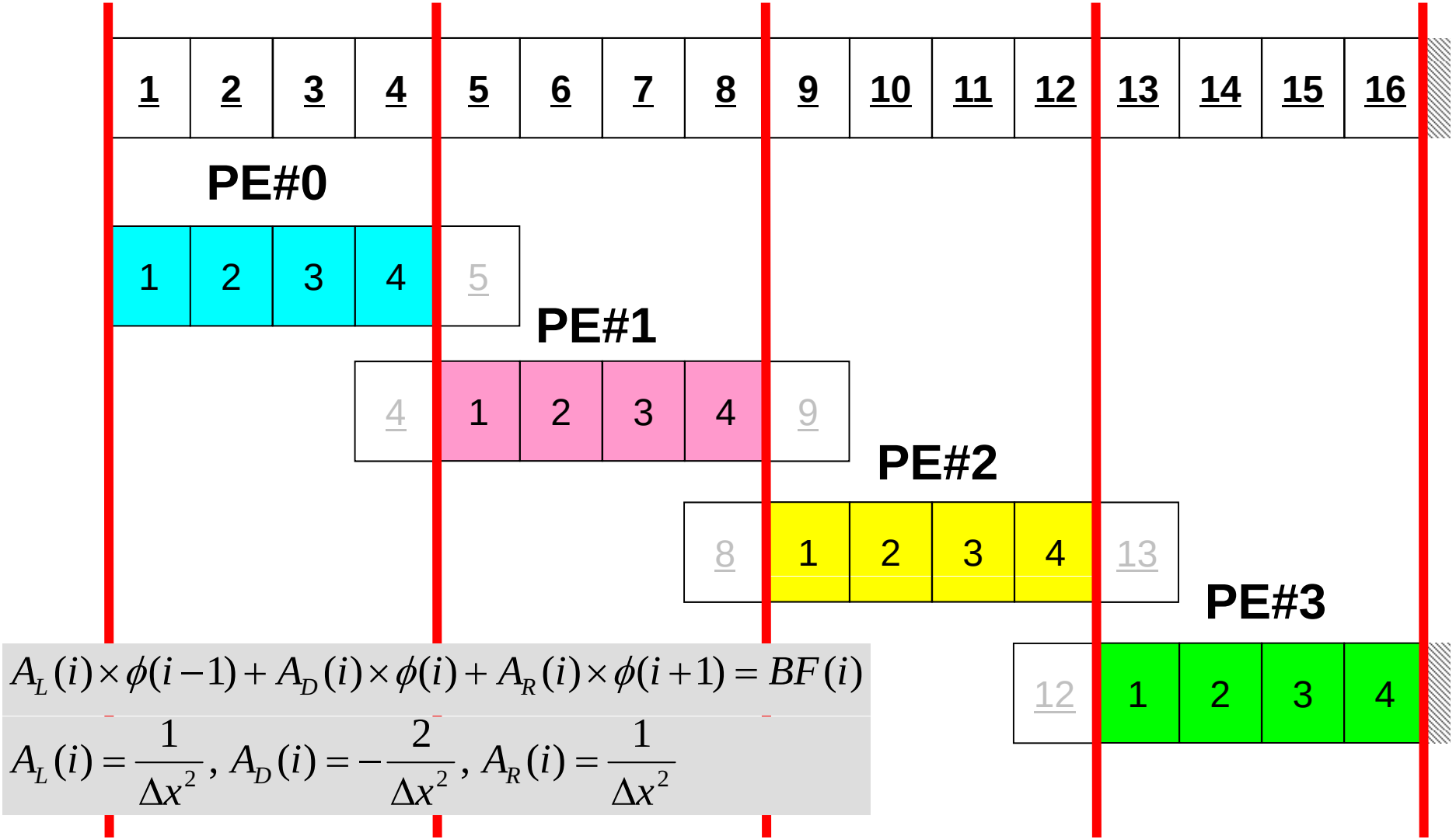
近接PE（領域）のみとの相互作用

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

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

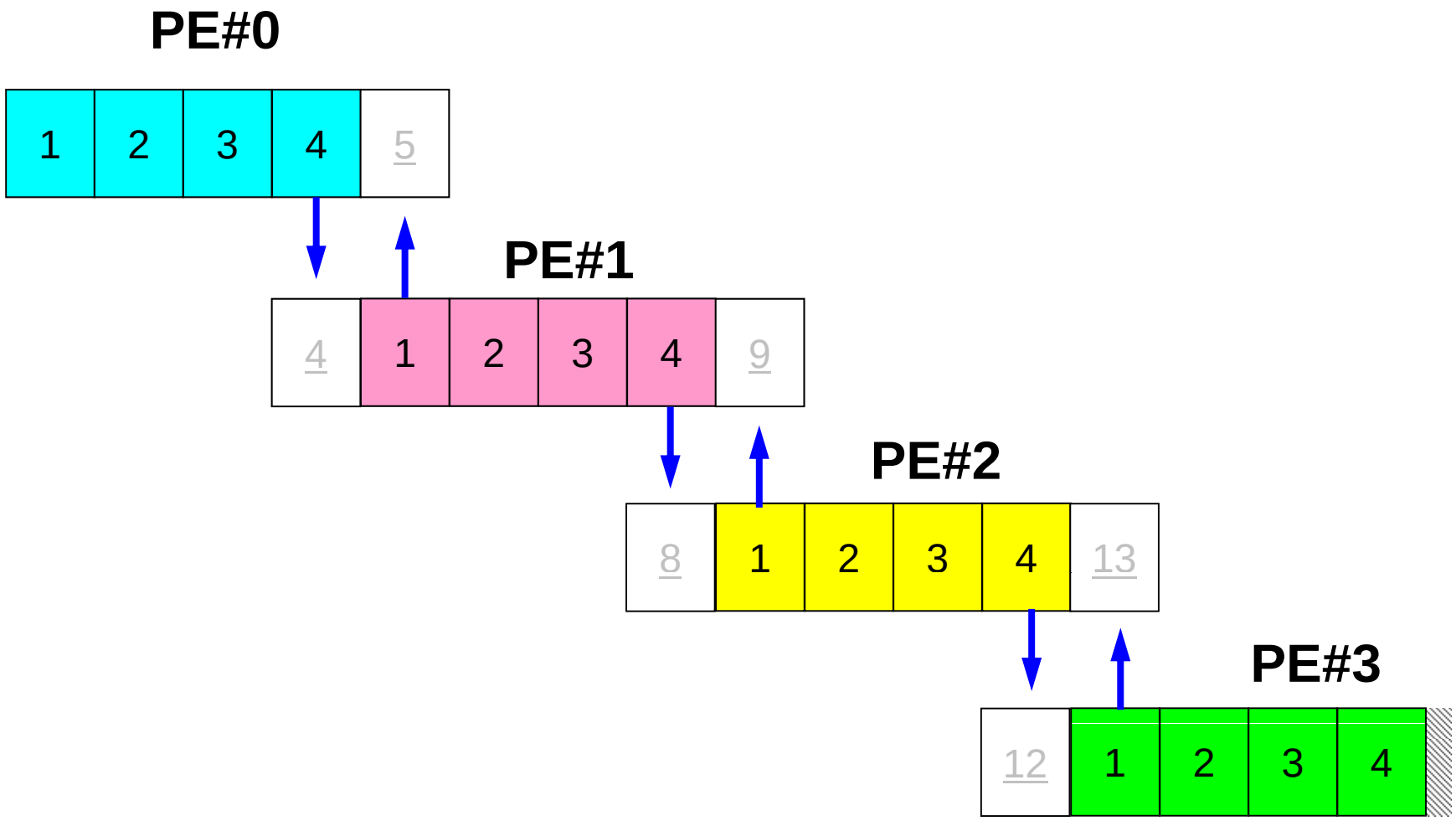


$$A_L(i)\times\phi(i-1)+A_D(i)\times\phi(i)+A_R(i)\times\phi(i+1)=BF(i)$$

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

1対1通信が必要になる場面：1D中央差分

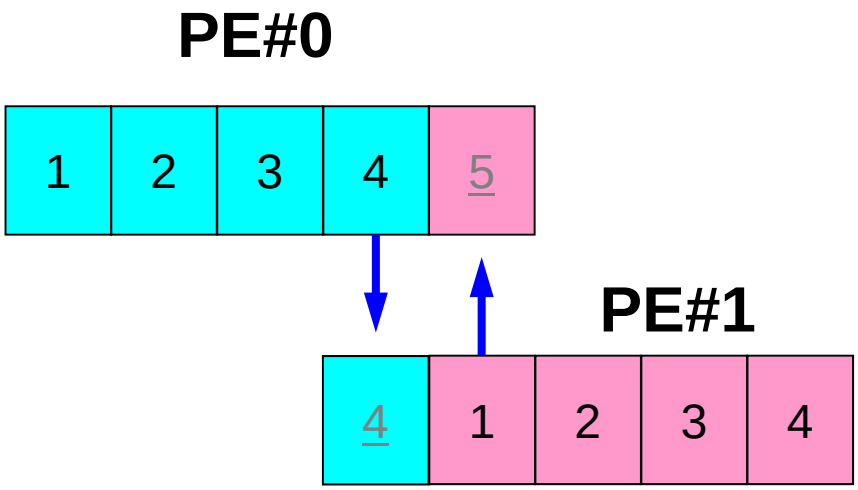
差分法のオペレーションのためには隣接要素の情報が必要



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。

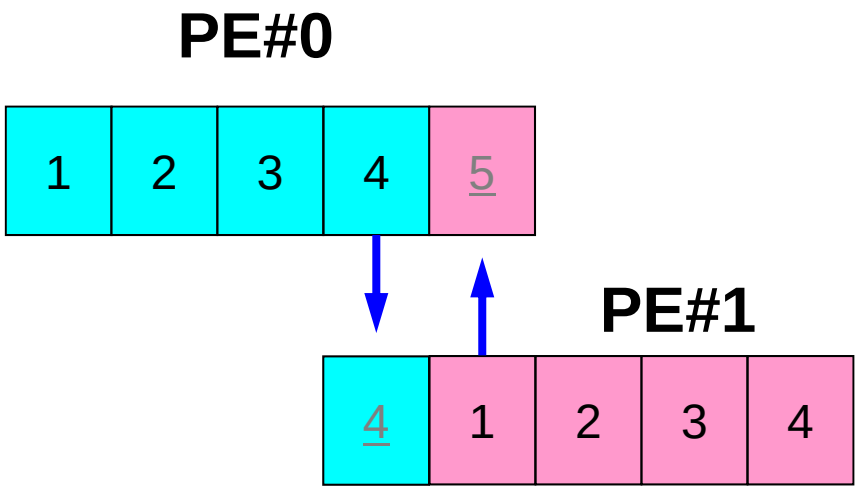
MPI_Send/MPI_Recv




```
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）

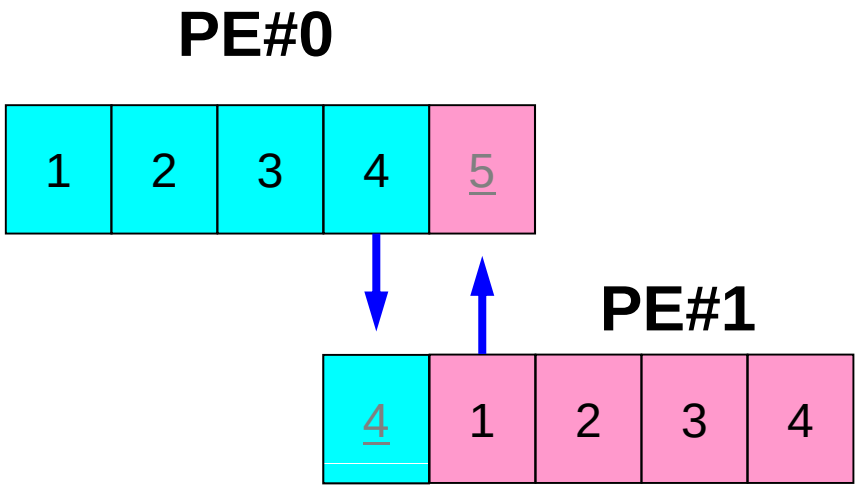



```
if (my_rank.eq.0) NEIB_ID=1
if (my_rank.eq.1) NEIB_ID=0

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

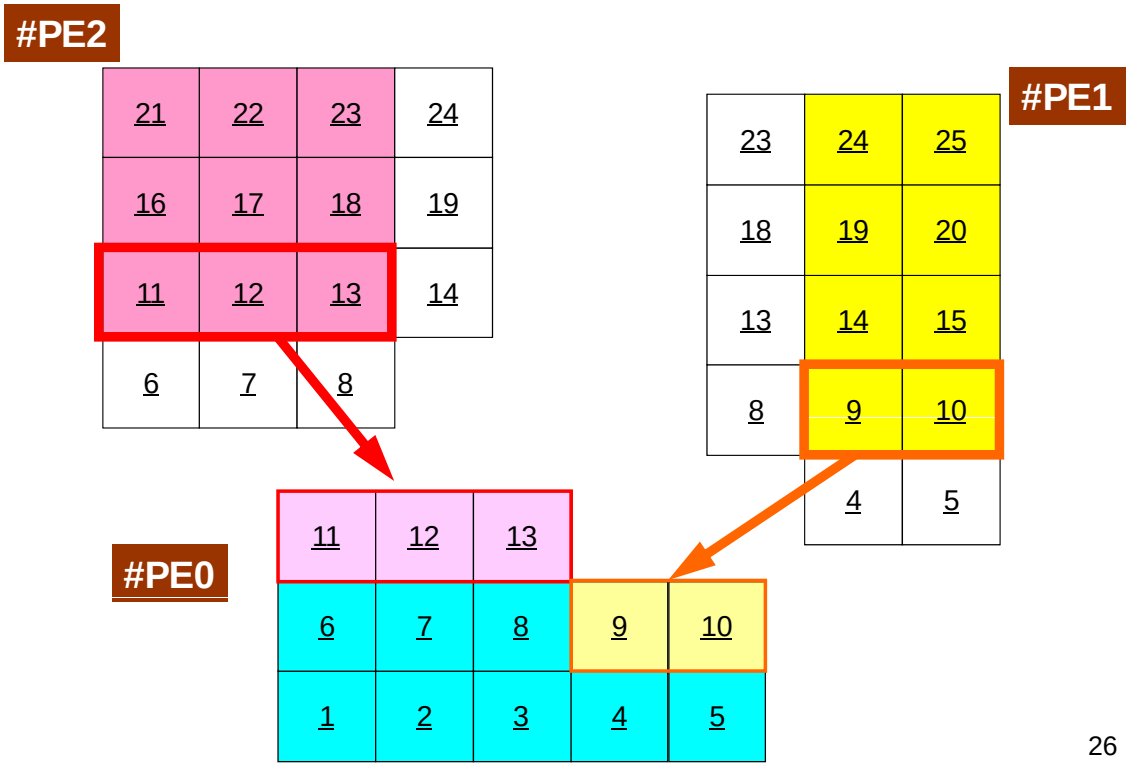
- MPI_SEND+MPI_RECV

- **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。同じメッセージタグ番号同士で通信。
  - **comm** 整数 I コミュニケータを指定する
  - **status** 整数 O 状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE））
    MPI_STATUS_SIZE: “mpif.h”で定められるパラメータ
    C言語については後述
  - **ierr** 整数 O 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

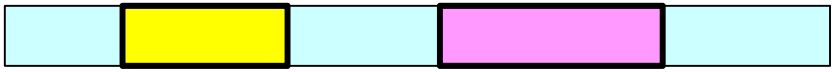
- **MPI_Irecv**
  (recvbuf,count,datatype,dest,tag,comm,request)
  - **recvbuf** 任意 I 受信バッファの先頭アドレス,
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **dest** 整数 I 宛先プロセスのアドレス(ランク)

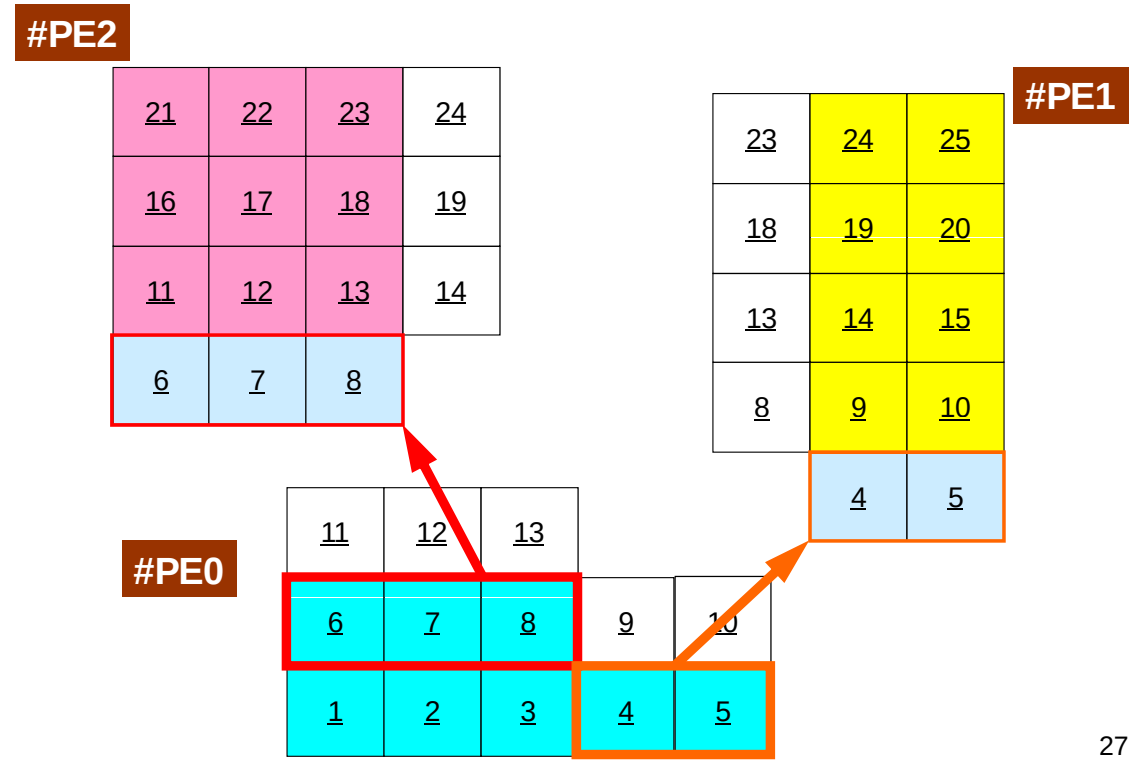


SEND（送信）：境界点の送信

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)
```

通信識別子，状況オブジェクト配列の定義の仕方（C）：特殊な変数の型がある

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

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

- **MPI_Sendrecv: status**

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

利用例(1):スカラー送受信

- PE#0，PE#1間 で8バイト実数VALの値を交換する。

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

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

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

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

受信バッファ名を「VAL」にしても動く場合はあるが，お勧めはしない。

利用例(1):スカラー送受信 FORTRAN

Isend/Irecv/Waitall

```
$> cd <$T-S2>
$> mpif90 –Oss -noparallel ex1-1.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), 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

```
$> cd <$T-S2>
$> mpicc –Os -noparallel ex1-1.c
$> バッチジョブ実行(2プロセス)
```

```
#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

```
$> cd <$T-S2>
$> mpif90 –Oss -noparallel 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(1):スカラー送受信 C

SendRecv

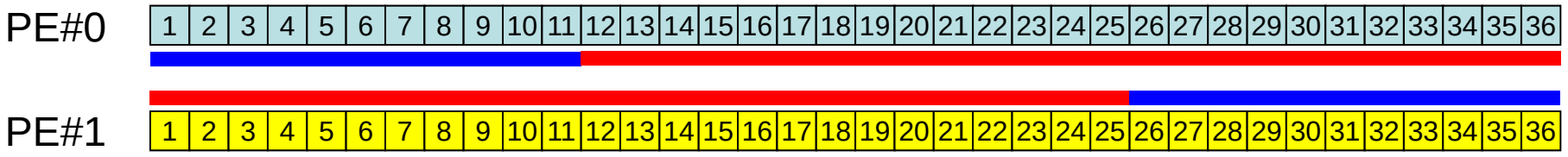
```
$> cd <$T-S2>
$> mpicc –Os -noparallel ex1-2.c
$> バッチジョブ実行(2プロセス)
```

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

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

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

        StatSR = malloc(sizeof(MPI_Status));

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

        MPI_Finalize();
        return 0;
}
```

利用例(2):配列の送受信(1/4)

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

- 演習:プログラムを作成して見よう！



演　習

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

- 次ページのような結果になることを確認せよ

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

予測される結果

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

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

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

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

利用例(2):配列の送受信(2/4)

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

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

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

これでも良いが，操作が煩雑
SPMDらしくない
汎用性が無い

利用例(2):配列の送受信(3/4)

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

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

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

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

一気にSPMDらしくなる

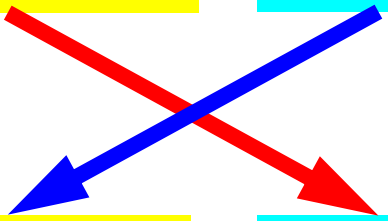
利用例(2):配列の送受信(4/4)

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

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

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

配列の送受信:注意

```
#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)
```

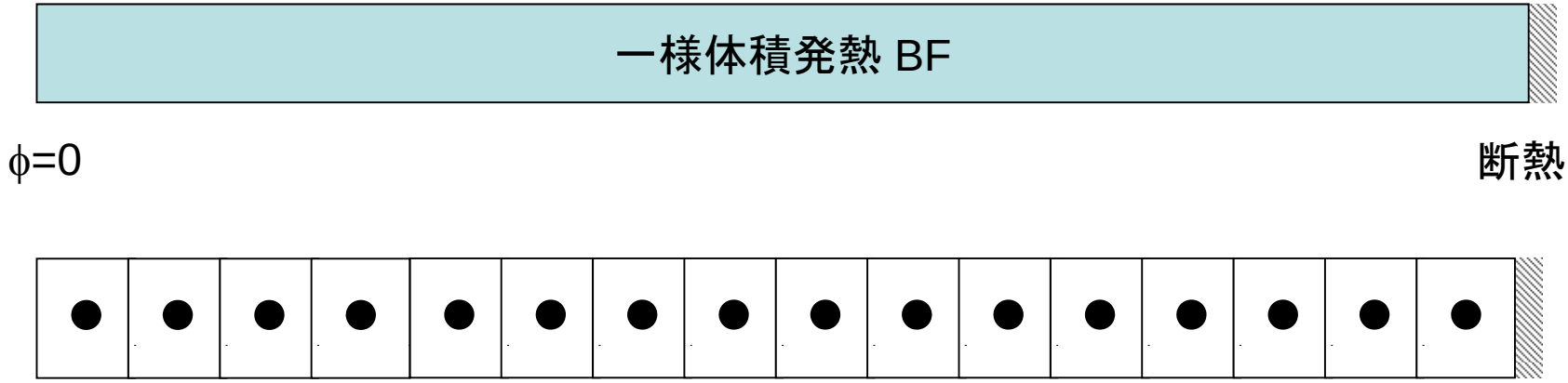
- 送信側の「length_send」と受信側の「length_recv」は一致している必要がある。
  - PE#0⇒PE#1，PE#1⇒PE#0
- 「送信バッファ」と「受信バッファ」は別のアドレス

- 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
  - MPI
  - 一次元問題
  - 二次元問題
- 1対1通信の実装例
- 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

一次元熱伝導方程式ソルバーの並列化

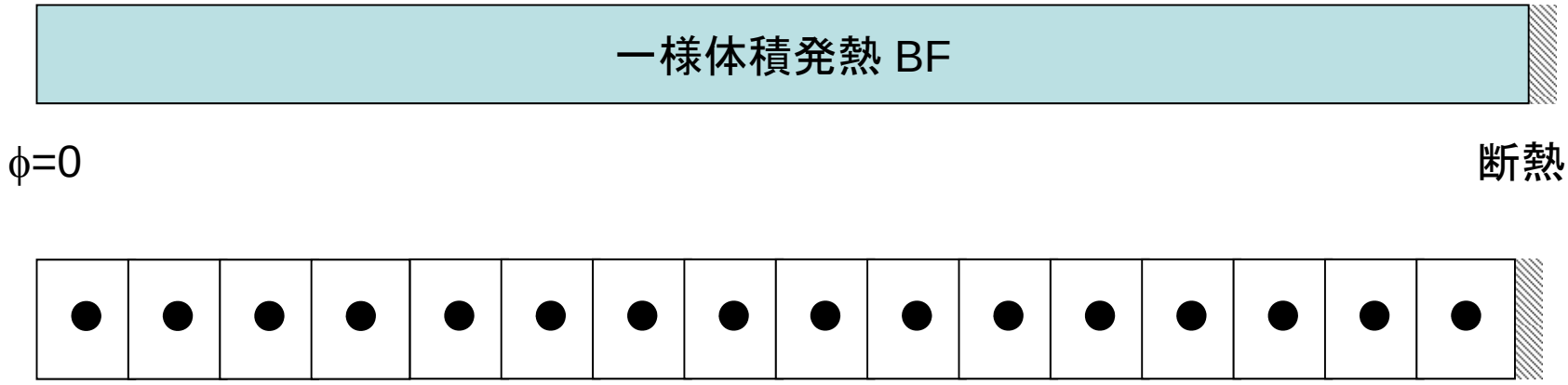
$$\frac{d^2\phi}{dx^2} + BF = 0, \quad \phi = 0 @ x = 0, \frac{d\phi}{dx} = 0 @ x = x_{\max}$$

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



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

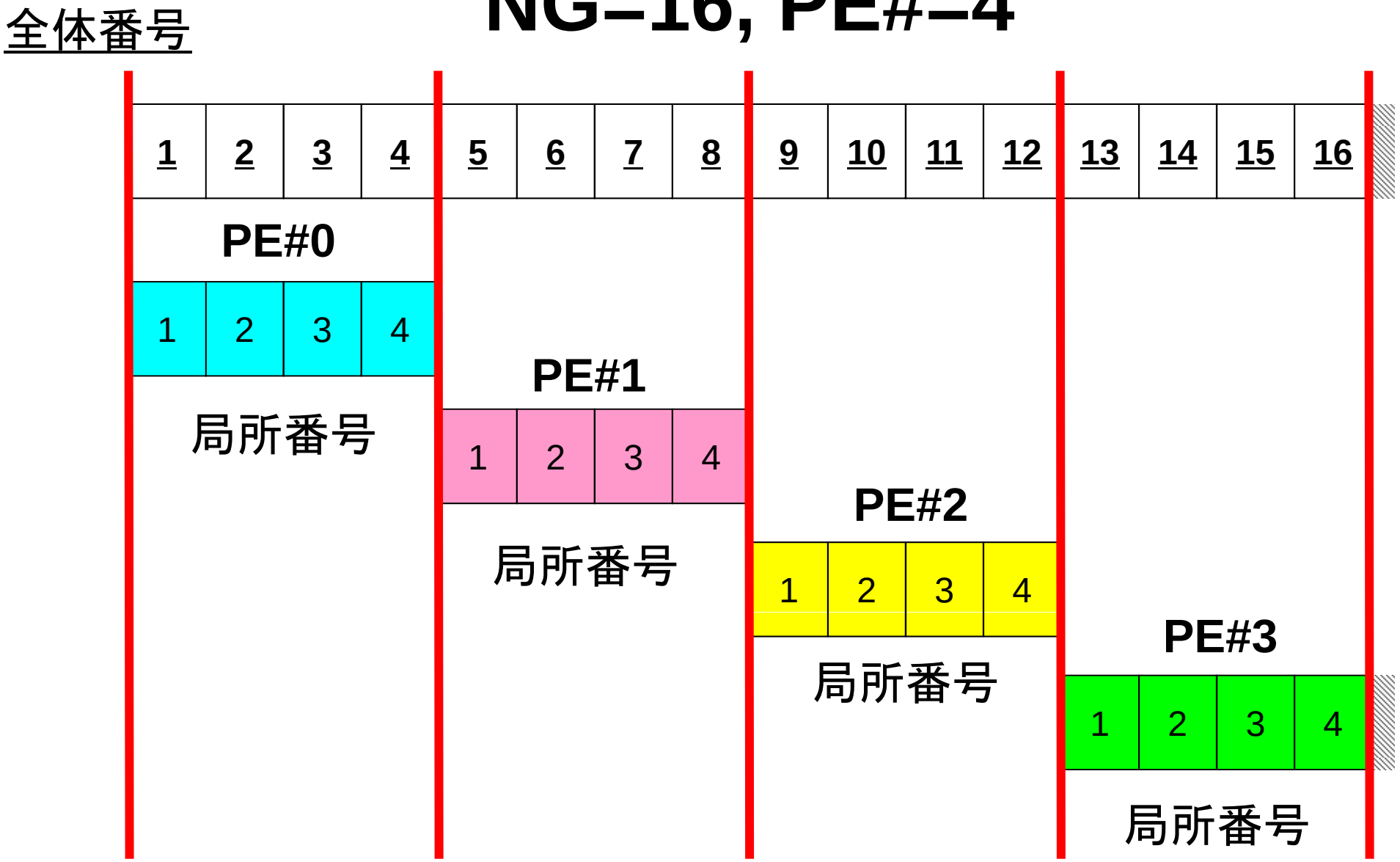


全体データと局所データ
NG=16, PE#=4

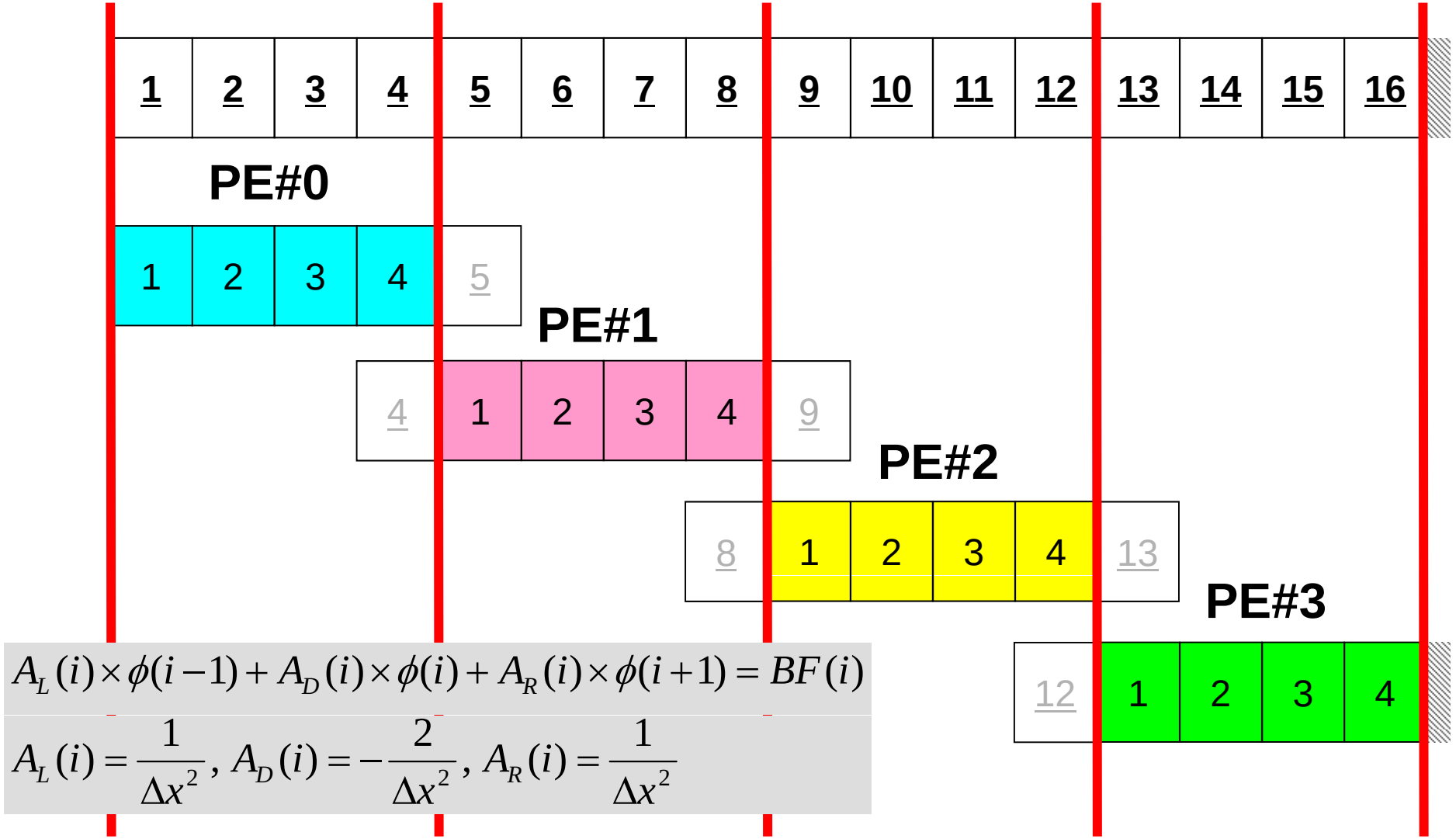
全体番号

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



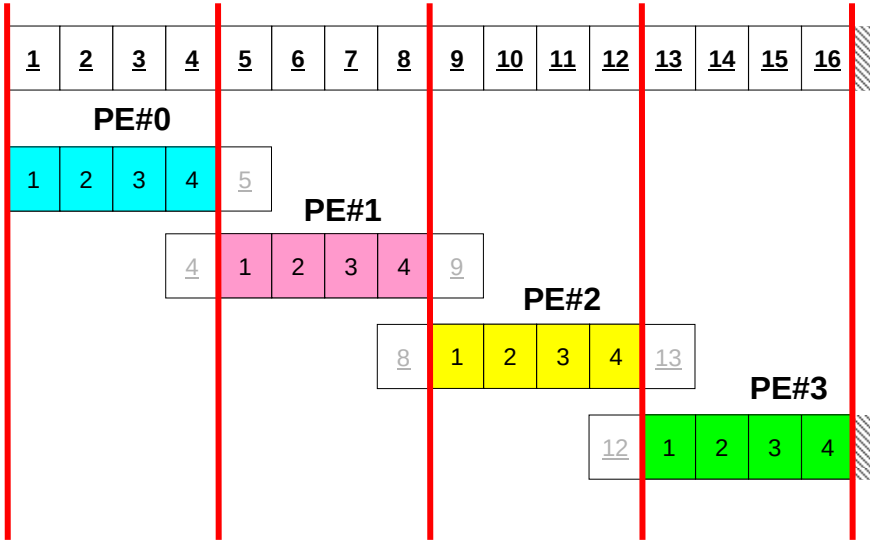
$$A_L(i)\times\phi(i-1)+A_D(i)\times\phi(i)+A_R(i)\times\phi(i+1)=BF(i)$$

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

一次元熱伝導方程式ソルバーの並列化

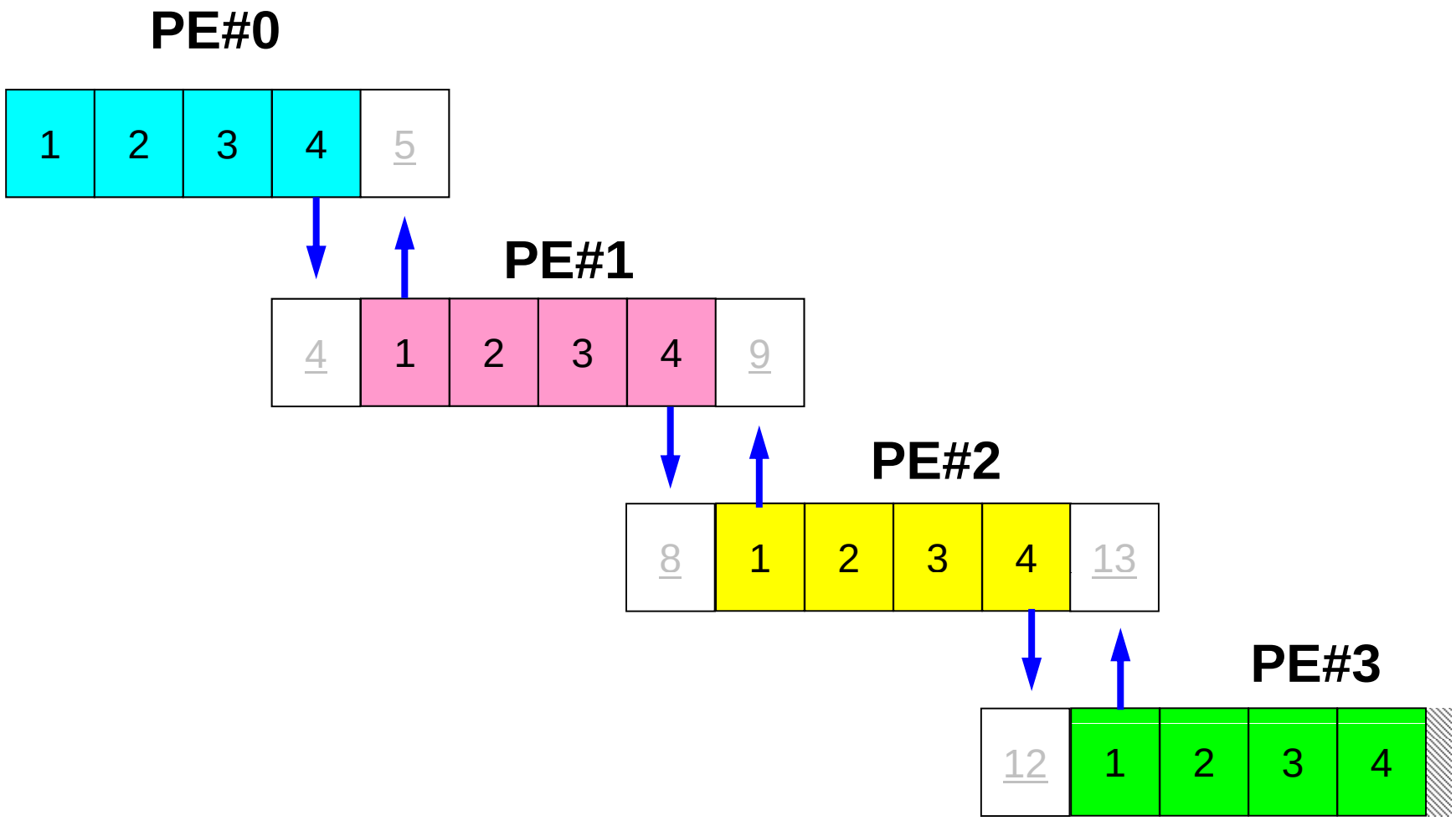
- 全体データと局所データ
- 差分法のオペレーションには隣接要素の値が必要
  - 領域（PE）の境界では隣接領域に属する要素からの情報が必要
  - 例えば，PE#1での計算を完結させるためには，PE#0の「4」番，PE#2の「1」番要素の情報が必要

- 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・・・



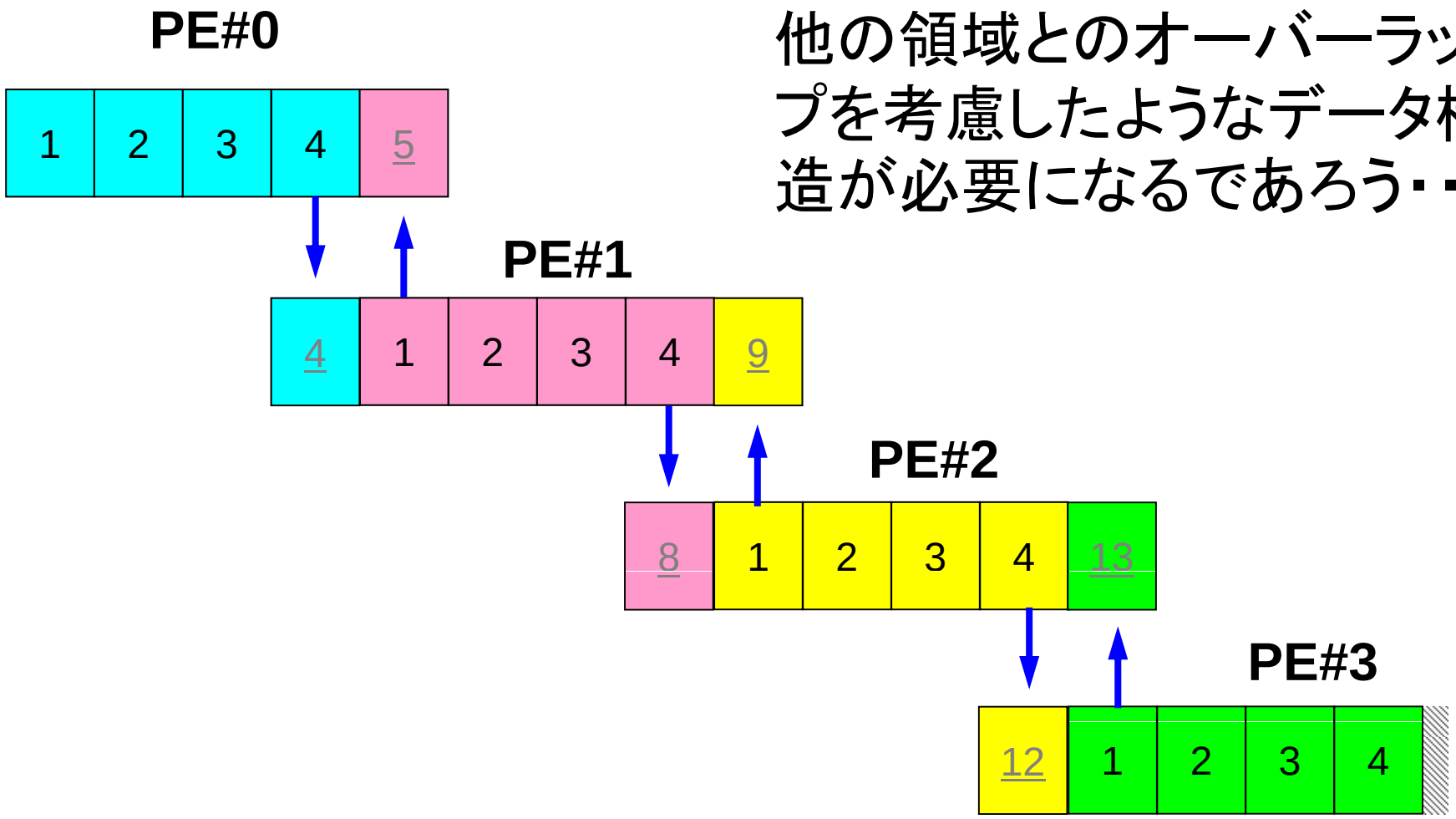
1対1通信が必要になる場面：1D中央差分

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

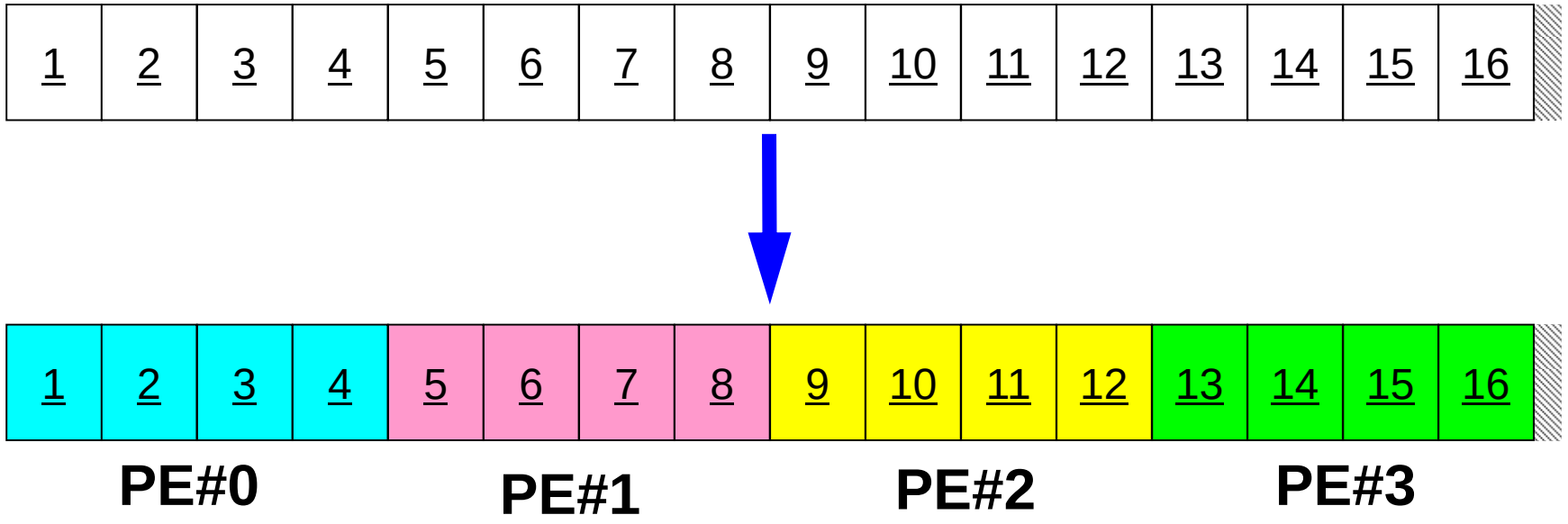


他の領域とのオーバーラップを考慮したようなデータ構造が必要になるであろう・・・

一次元差分法モデル局所データ構造(1/5)

NG=16の一次元領域

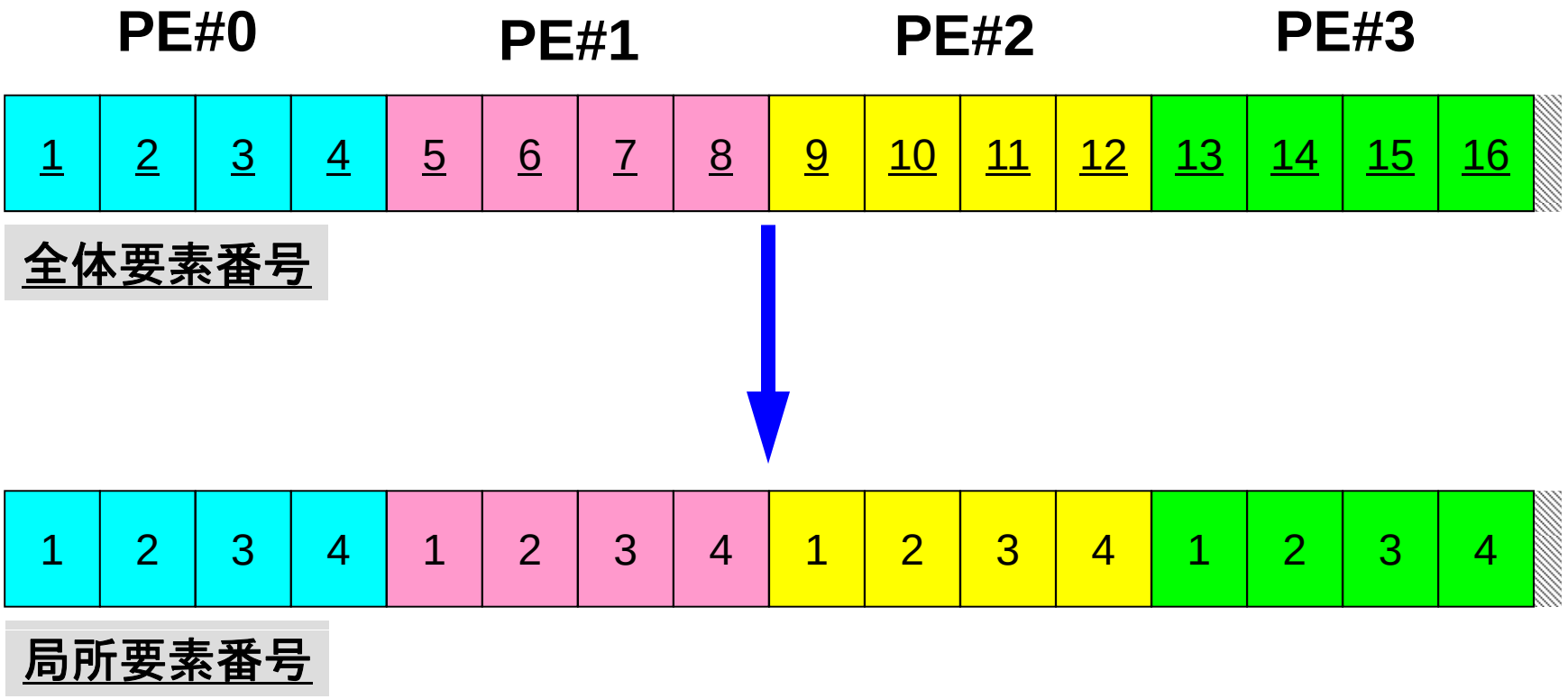
- NG=16の一次元領域を4領域(プロセッサ)に分割して並列に差分法(二次精度)の計算を実施するために必要なデータ構造を考える。



一次元差分法モデル局所データ構造(2/5)

局所要素番号

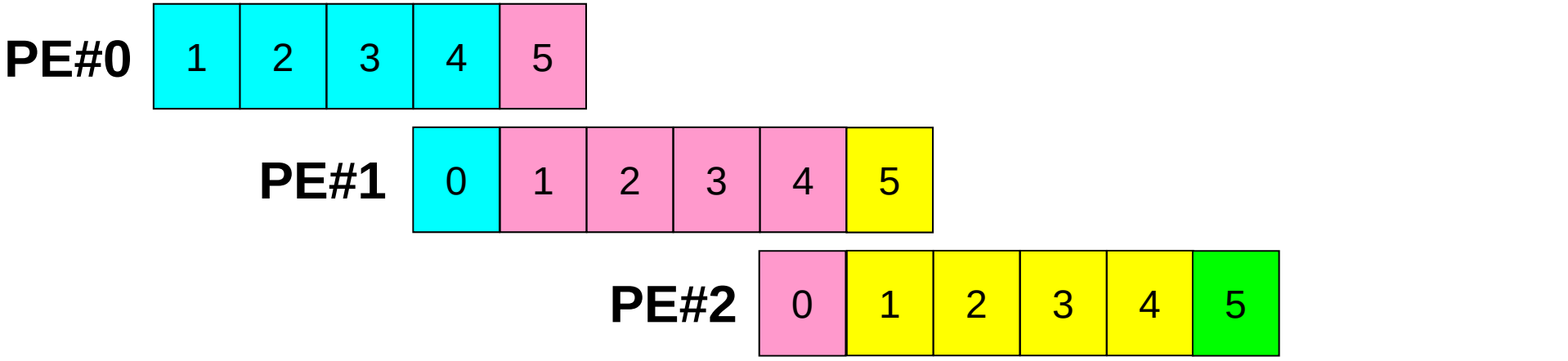
- 各領域(プロセッサ)において，N=4の局所データを扱う



一次元差分法モデル局所データ構造(3/5)

領域外の要素

- 差分法(二次精度中央差分)の計算を並列に実施するためには，本来領域外の要素の影響を考慮する必要がある。
  - 領域間オーバーラップ
- 要素番号が0，N+1(=5)の要素を加えた，局所データ構造



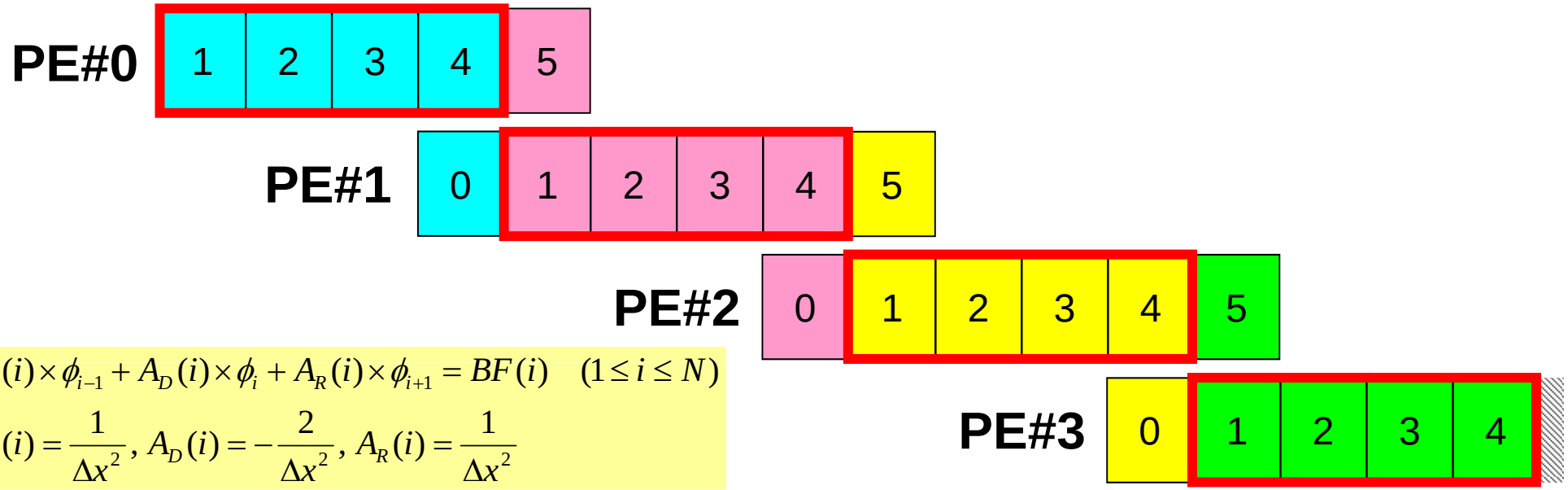
$$A_L(i)\times\phi_{i-1}+A_D(i)\times\phi_i+A_R(i)\times\phi_{i+1}=BF(i)\quad(1\le i\le N)$$

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

一次元差分法モデル局所データ構造(4/5)

必要な情報:内点

- 領域内の要素，オーバーラップ(本来領域外)要素の区別
  - 領域内要素(i=1~N)　　:内点(Interior/Internal Points)



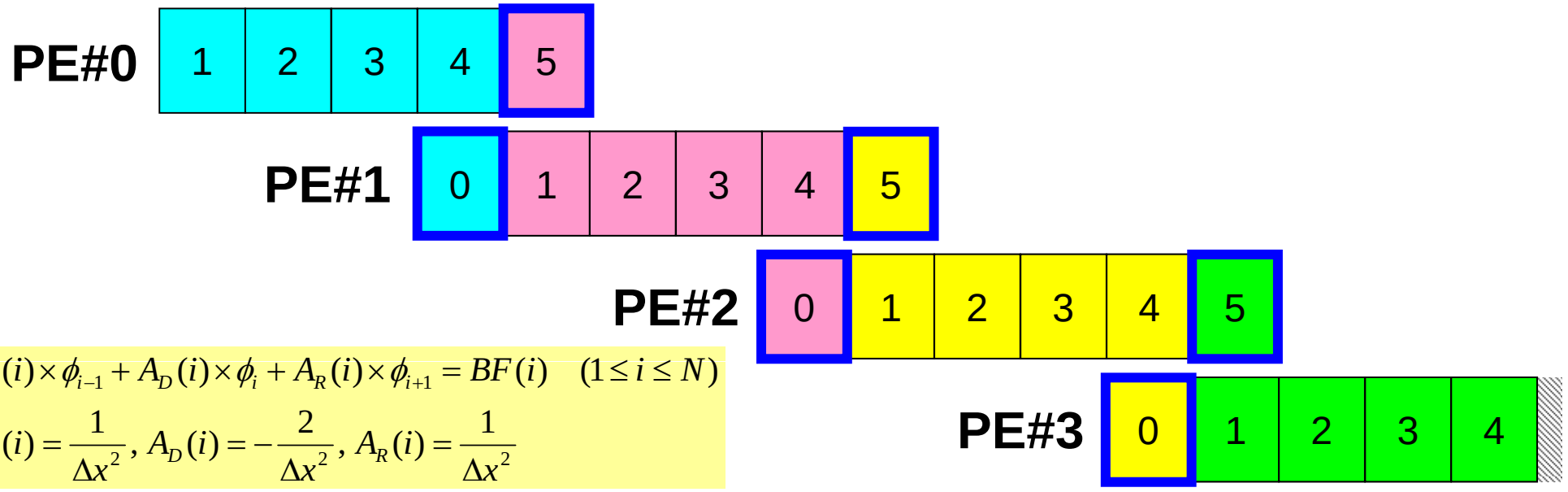
$$A_L(i)\times\phi_{i-1}+A_D(i)\times\phi_i+A_R(i)\times\phi_{i+1}=BF(i)\quad(1\le i\le N)$$

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

一次元差分法モデル局所データ構造(4/5)

必要な情報:外点

- 領域内要素，オーバーラップ(本来領域外)要素の区別
  - 領域内要素(i=1~N)　　:内点(Interior/Internal Points)
  - オーバーラップ要素(i=0, N+1):外点(Exterior/External Points)
    - i=0，i=N+1の要素が存在しない場合もある



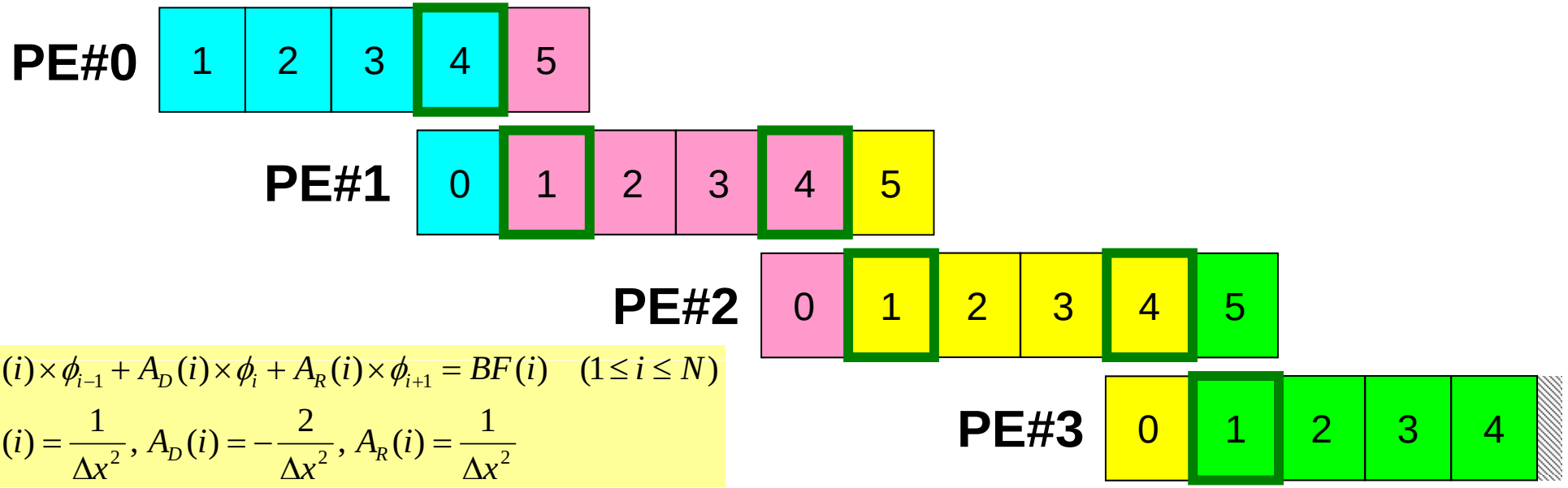
$$A_L(i)\times\phi_{i-1}+A_D(i)\times\phi_i+A_R(i)\times\phi_{i+1}=BF(i)\quad(1\le i\le N)$$

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

一次元差分法モデル局所データ構造(4/5)

必要な情報:境界点

- 領域内要素，オーバーラップ(本来領域外)要素の区別
  - 領域内要素(i=1~N)　　:内点(Interior/Internal Points)
  - オーバーラップ要素(i=0, N+1):外点(Exterior/External Points)
    - i=0，i=N+1の要素が存在しない場合もある
  - 「内点」のうち他領域の「外点」となっている要素:境界点(Boundary Points)



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

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

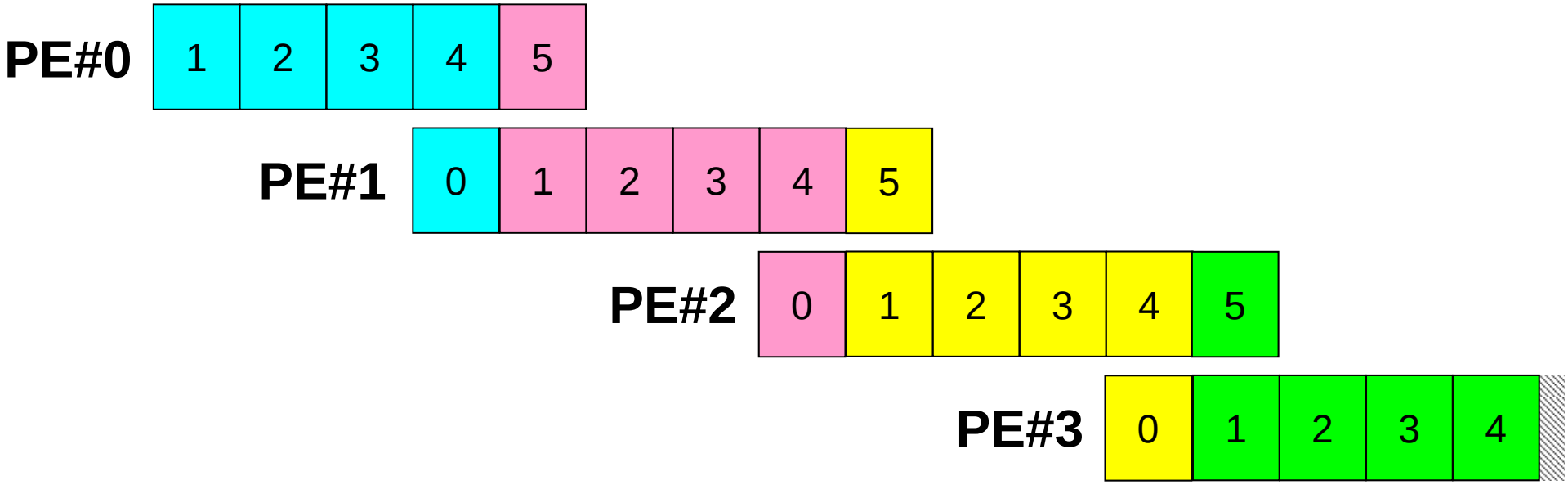
一次元差分法モデル局所データ構造(5/5)

必要な情報(続き)

- 「領域間オーバーラップ」を考慮した並列計算を実現するためには？
  - 各領域(プロセッサ)における「境界点」の情報を，各隣接領域に「外点」の情報として配信する必要がある。
  - そのための局所データ構造が必要
- 隣接している領域数，隣接している領域番号
  - 1対1通信の場合重要
  - オーバーラップ要素を共有している領域(プロセッサ)
- 隣接領域との「通信テーブル」
  - 境界点：隣接領域への「送信」(send)
  - 外点　：隣接領域からの「受信」(recv)

これには問題がある

- C言語:「-1」から始まる配列は不可
- 二次元



- 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
  - MPI
  - 一次元問題
  - 二次元問題
- 1対1通信の実装例
- 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

例えばこのような二次元系の場合

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

例えばこのような二次元系の場合

33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

「外点」の局所番号はどうする?

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

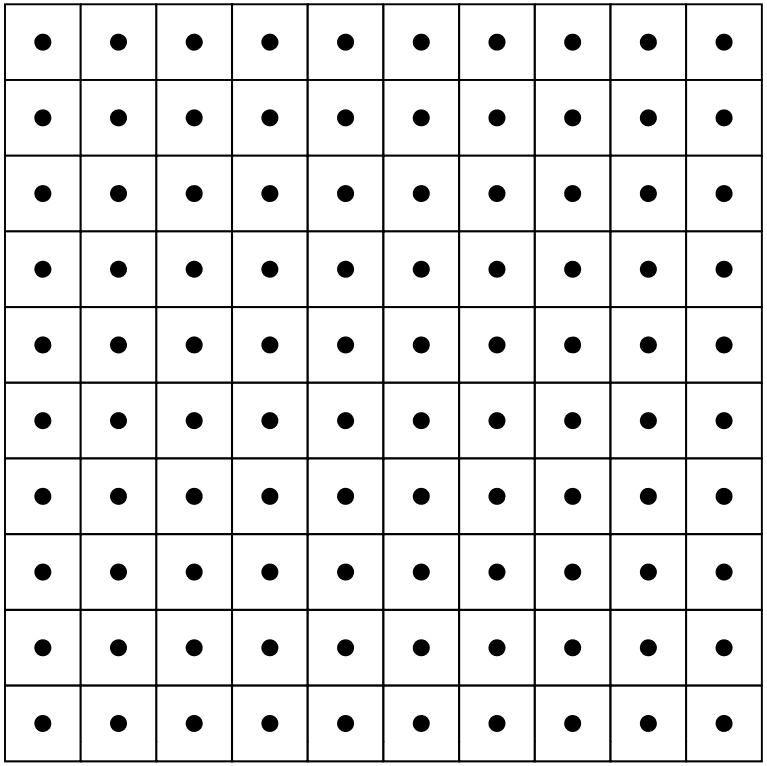
?	9	10	11	12	?
?	5	6	7	8	?
?	1	2	3	4	?

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

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

二次元差分法（1/5）

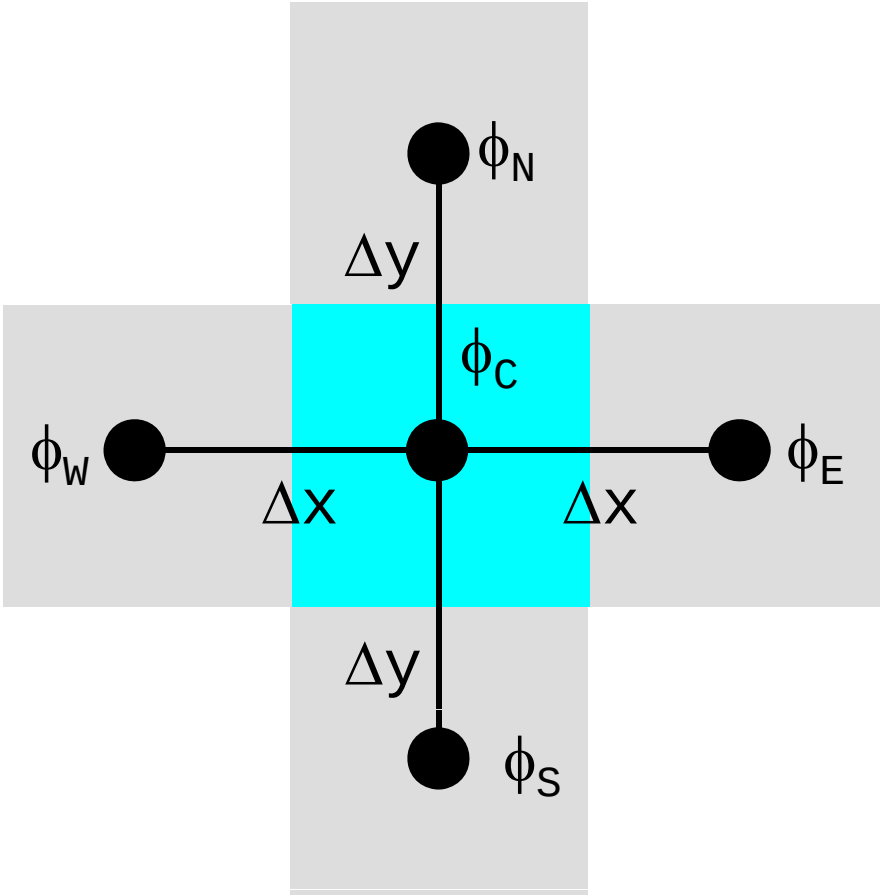
全体メッシュ



二次元中央差分法(5点差分法)の定式化

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

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



二次元系の場合：4領域に分割

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

4領域に分割：全体番号

PE#3

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

PE#2

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

PE#0

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

PE#1

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

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

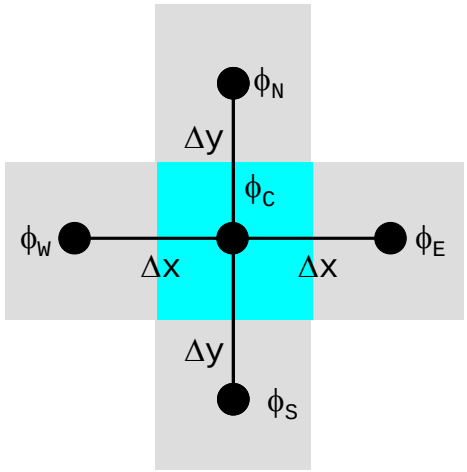
PE#0

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



PE#3

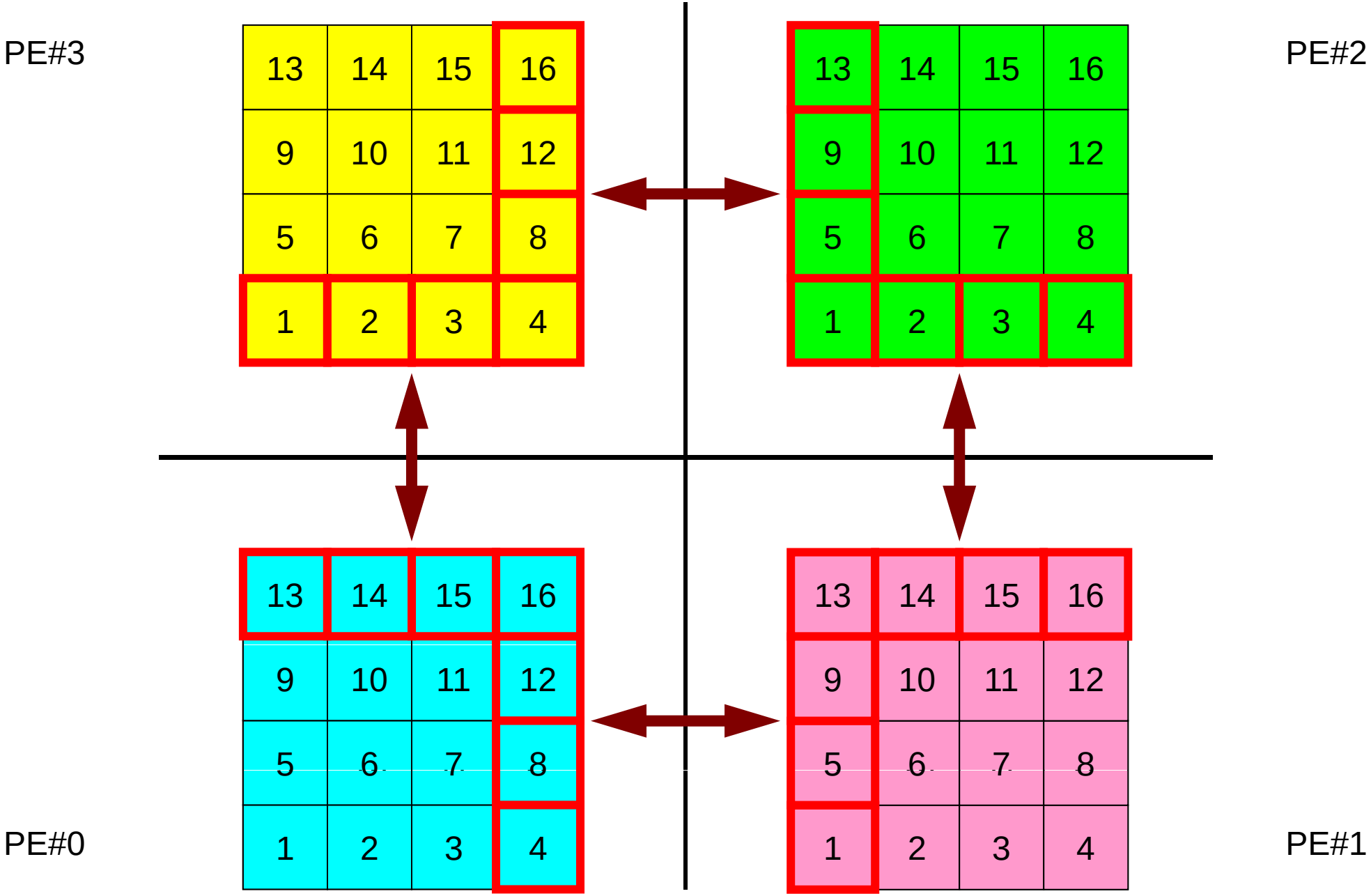
PE#2

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

PE#0

PE#1

オーバーラップ領域の値が必要



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
	?	?	?	?

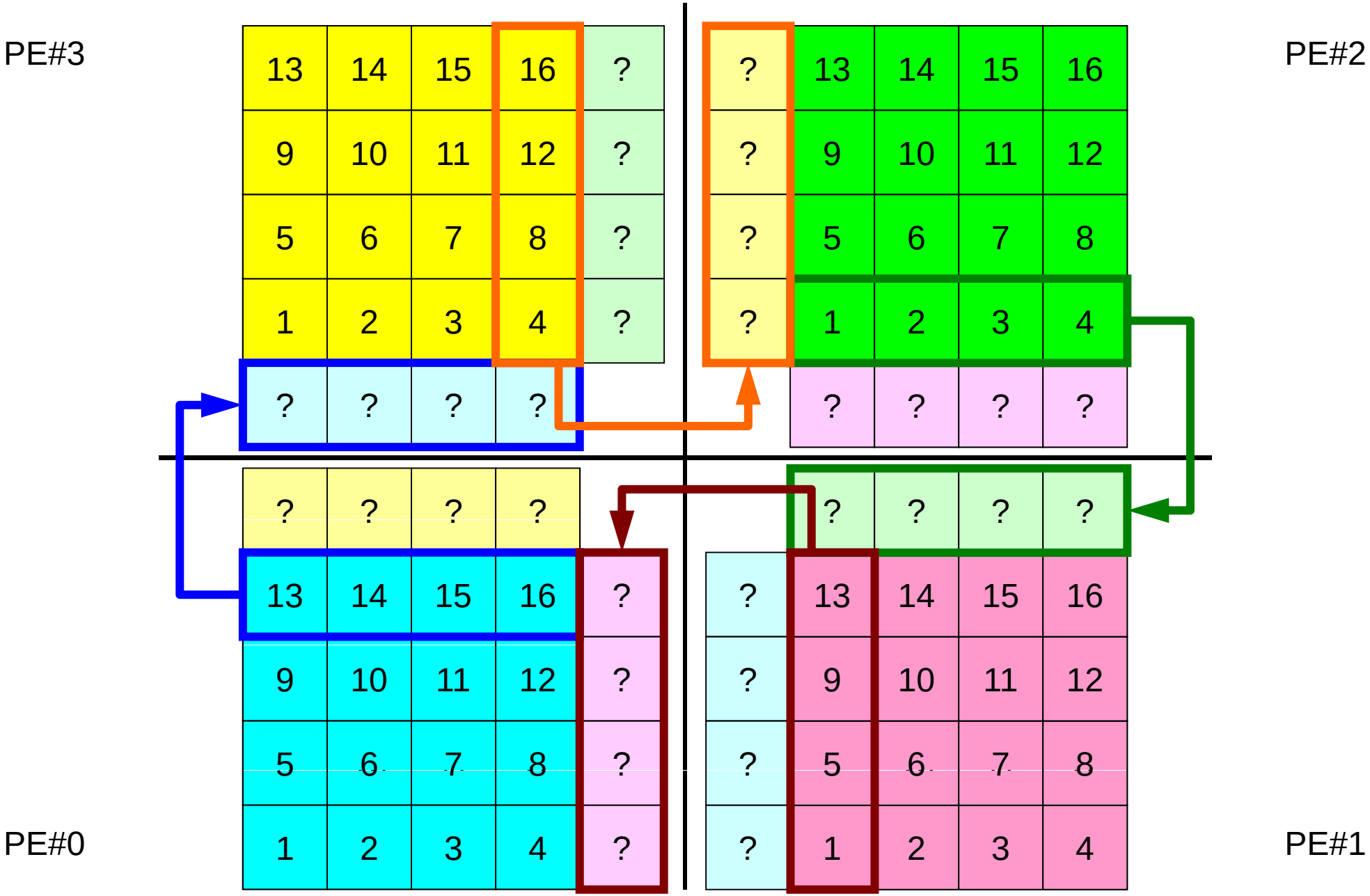
PE#0

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

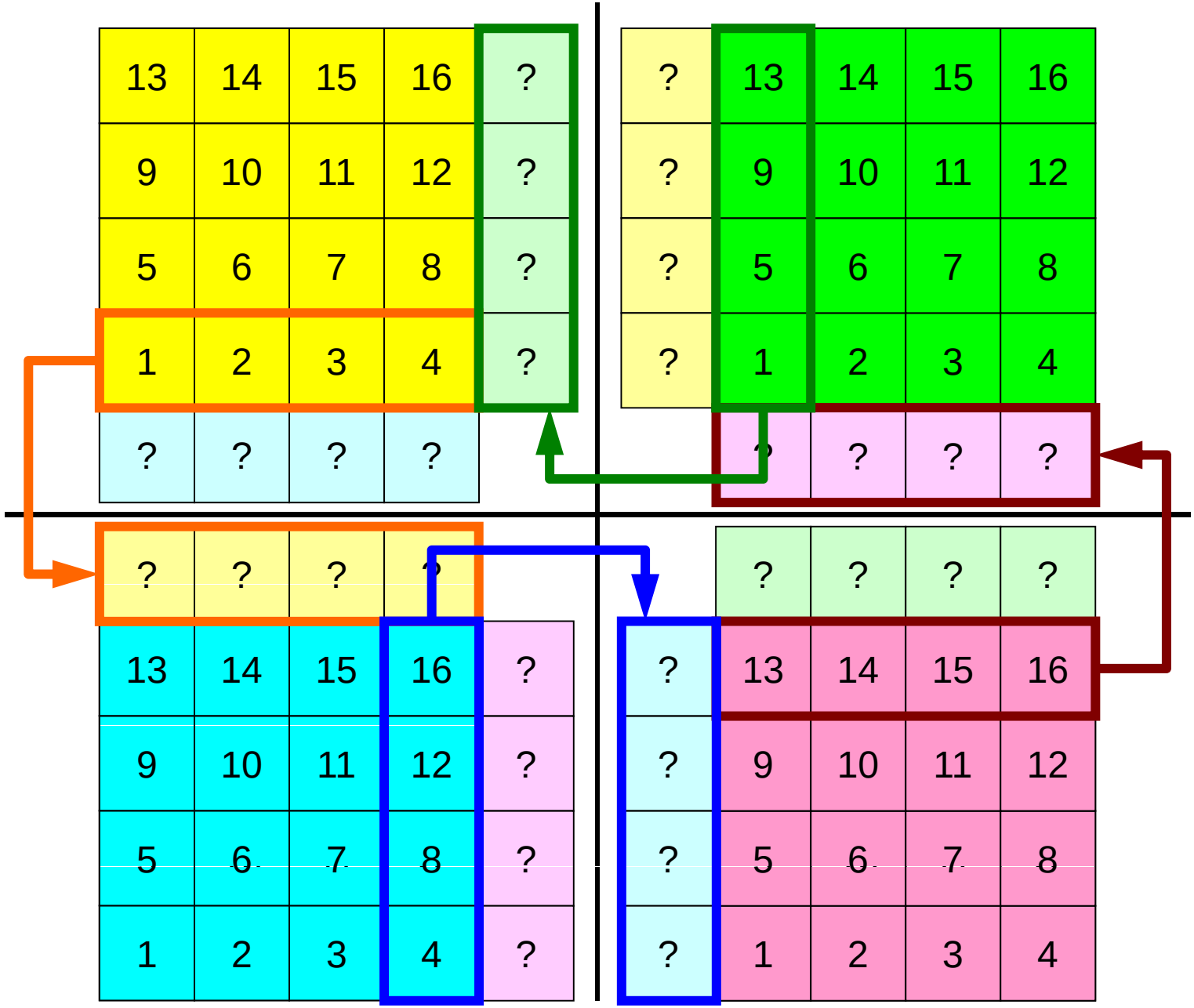
PE#1

	?	?	?	?
?	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



オーバーラップ領域の値が必要



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

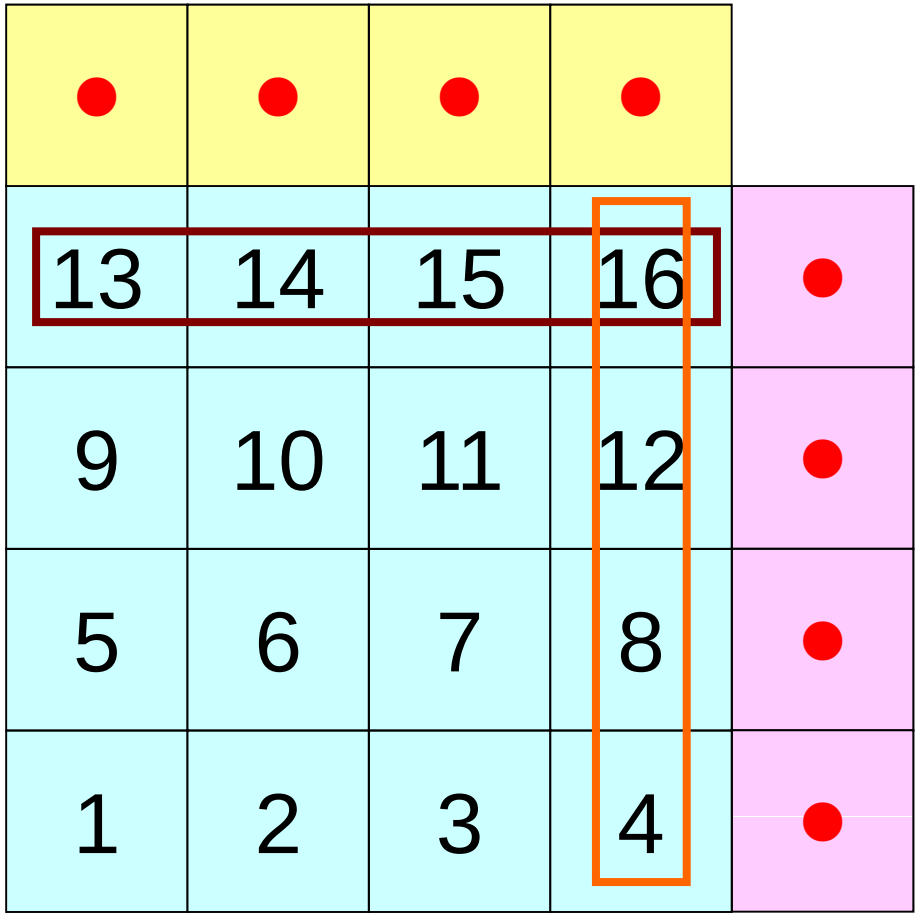
内点（Internal Points）
その領域にアサインされた要素

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



PE#1

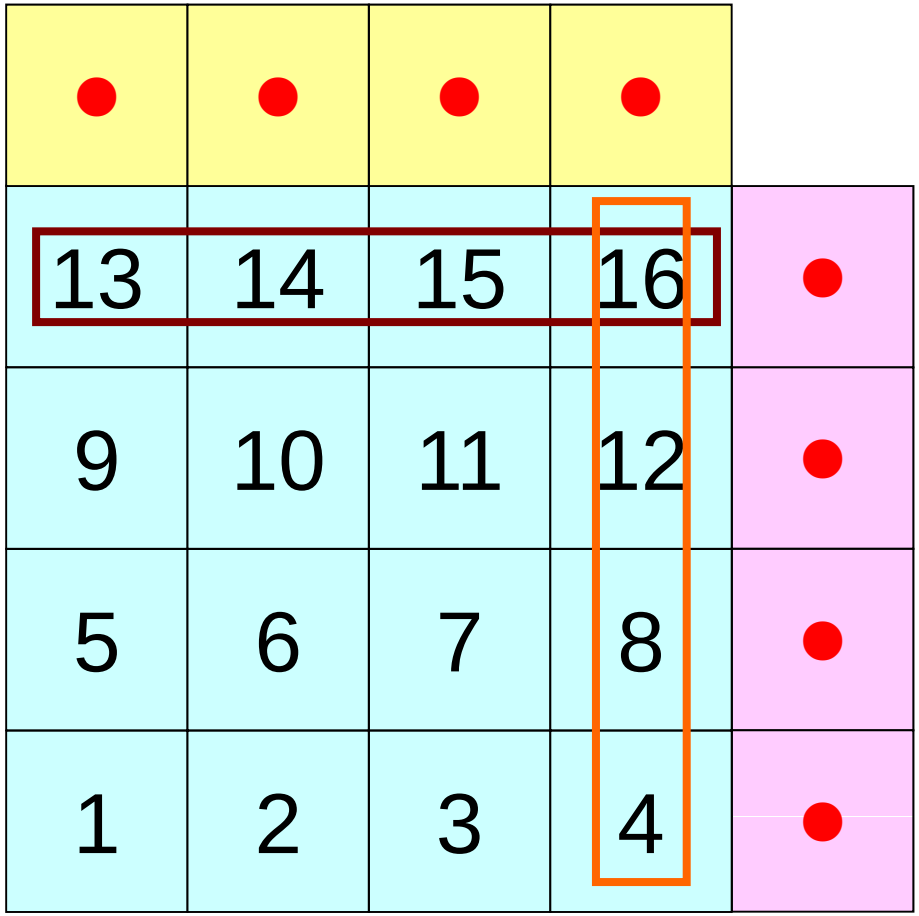
内点（Internal Points）
その領域にアサインされた要素

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



PE#1

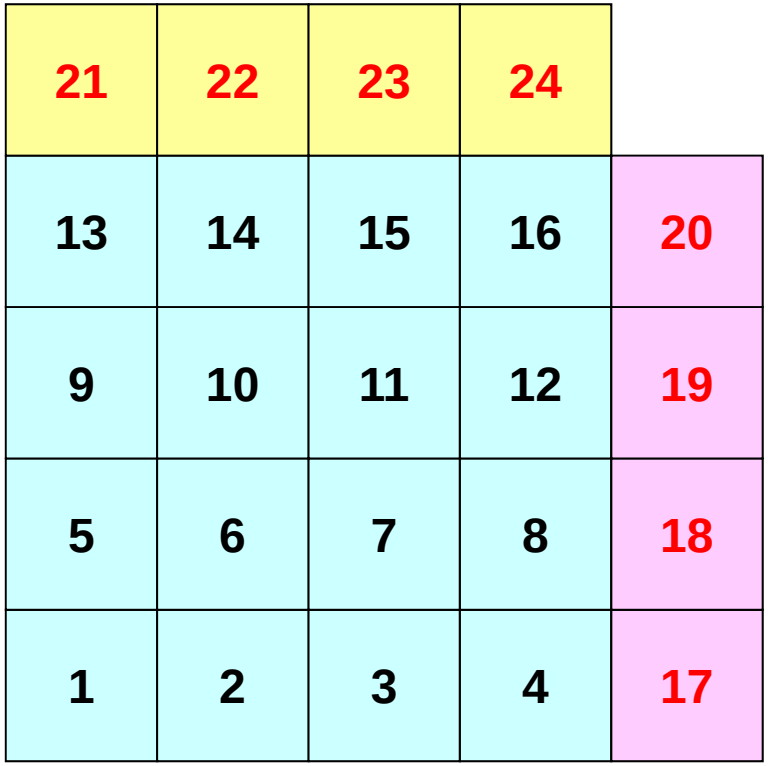
内点（Internal Points）
その領域にアサインされた要素

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



- 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

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)

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)

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)

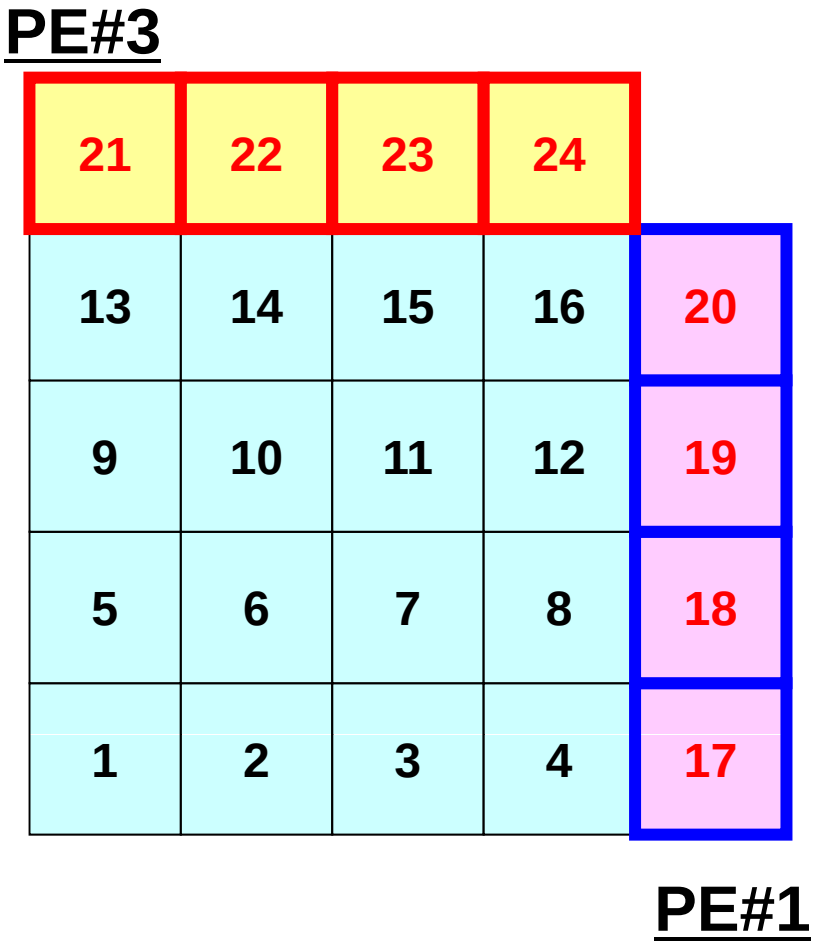
PE#3

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
2
#NEIBPE
1  3
#NODE
24 16                内点＋外点，内点
#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)

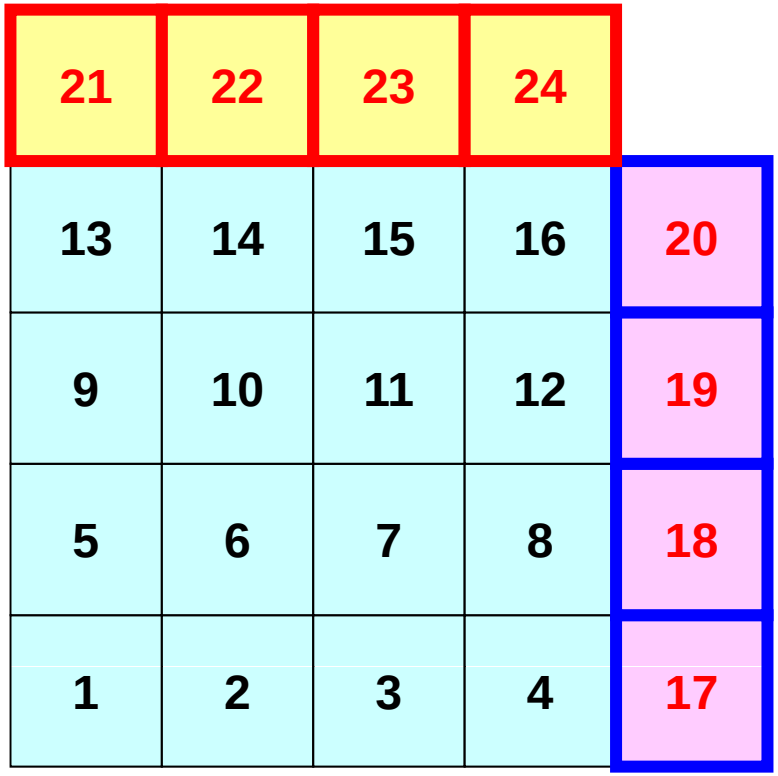



```
#NEIBPEtot
2
#NEIBPE
1  3
#NODE
24 16
#IMPORT_index
4  8
#IMPORT_items
17
18
19
20
21
22
23
24
#EXPORT_index
4  8
#EXPORT_items
 4
 8
12
16
13
14
15
16
```

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

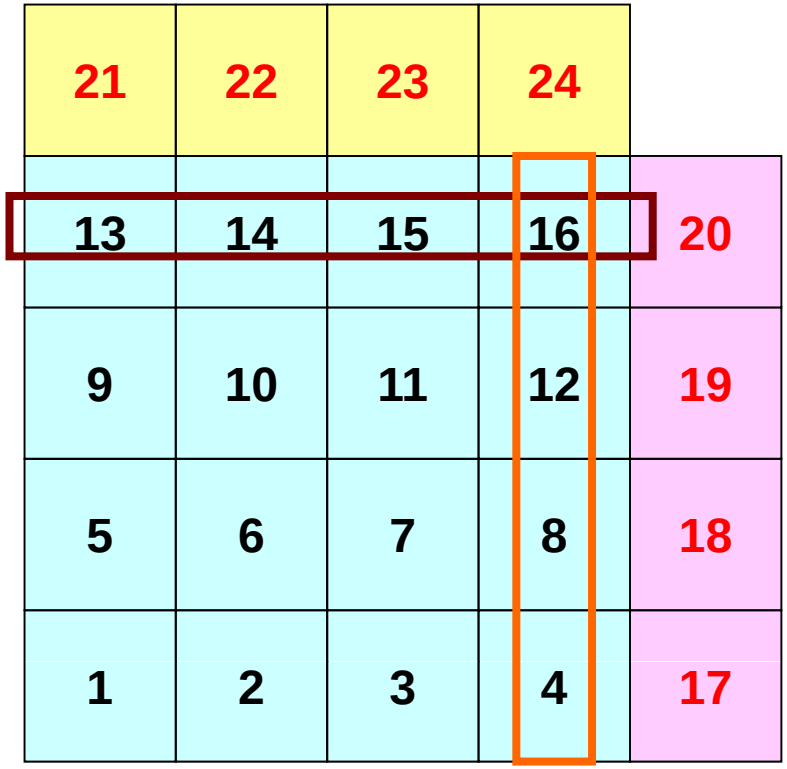


PE#1

```
#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
22          隣接領域2(#3)から
23          「import」する要素(5~8)
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

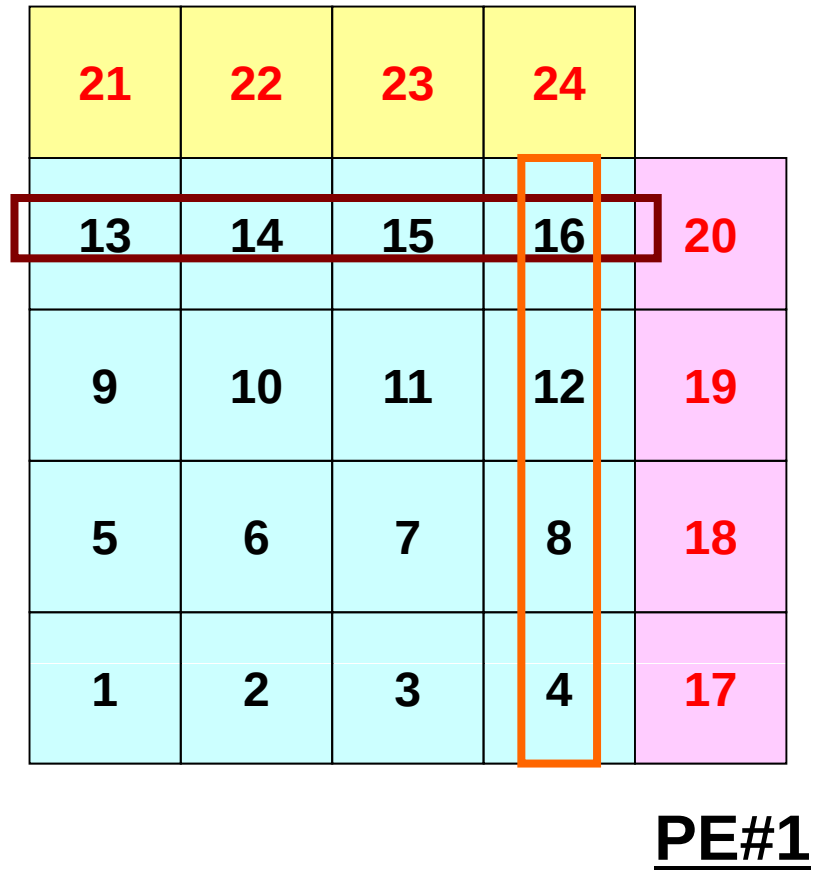


PE#1

```
#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)



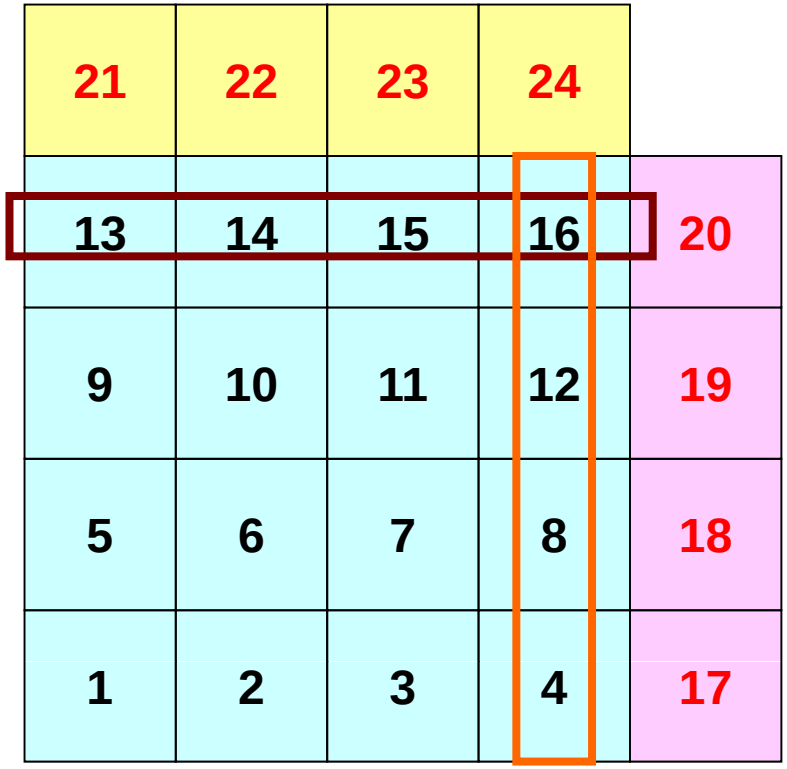

```
#NEIBPEtot
2
#NEIBPE
1  3
#NODE
24 16
#IMPORT_index
4  8
#IMPORT_items
17
18
19
20
21
22
23
24
#EXPORT_index
4  8
#EXPORT_items
 4
 8
12
16
13
14
15
16
```

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

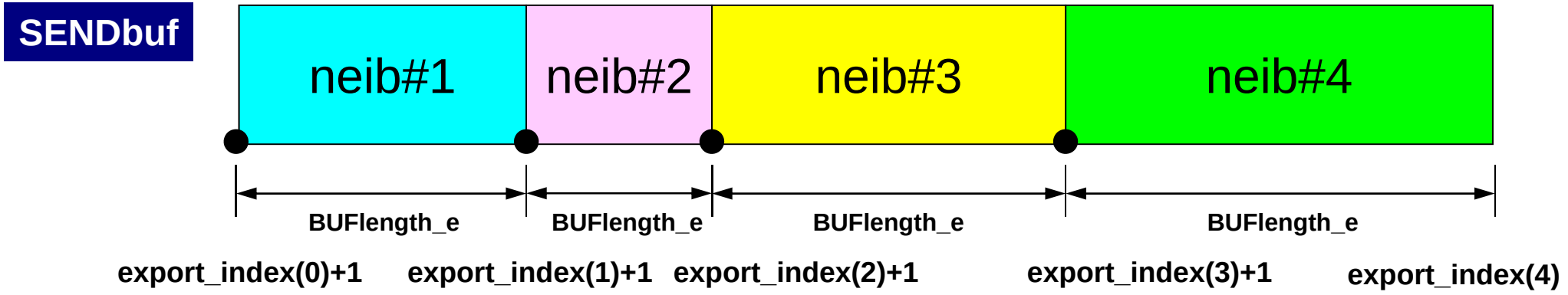


PE#1

「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)。

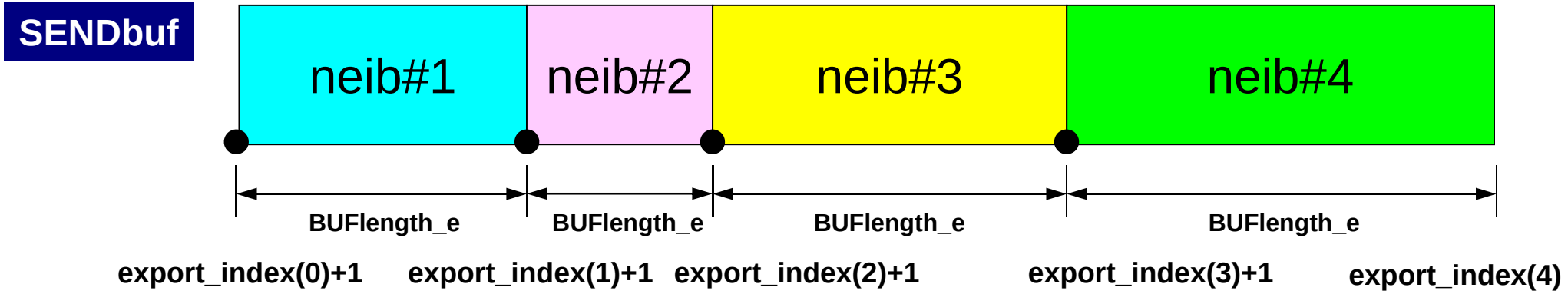
送信（MPI_Isend/Irecv/Waitall）




```
 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。

送信（MPI_Sendrecv）

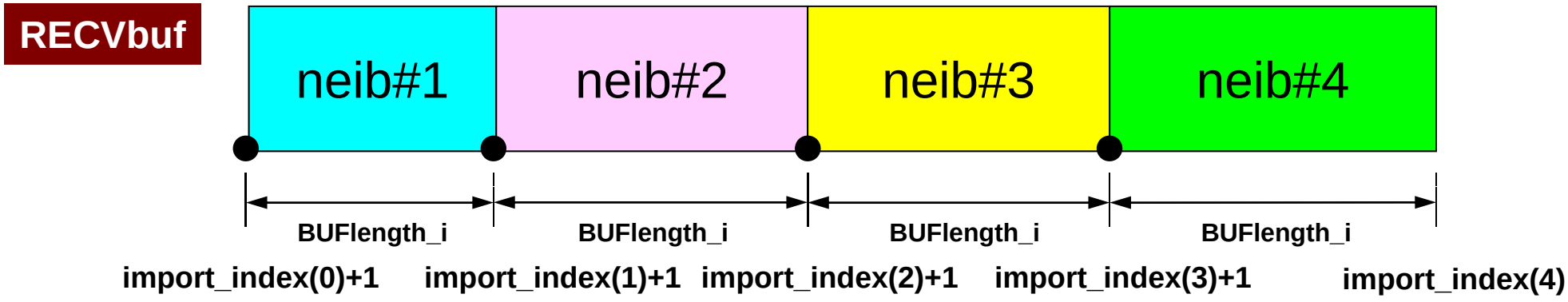



```
 do neib= 1, NEIBPETOT
   do k= export_index(neib-1)+1, export_index(neib)
     kk= export_item(k)
     SENDbuf(k)= VAL(kk)
   enddo
 enddo

 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_SENDRECV                                              &
&         (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&          RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&          MPI_COMM_WORLD, stat_sr, ierr)
  enddo
```

送信バッファへの代入

受信（MPI_Isend/Irecv/Waitall）

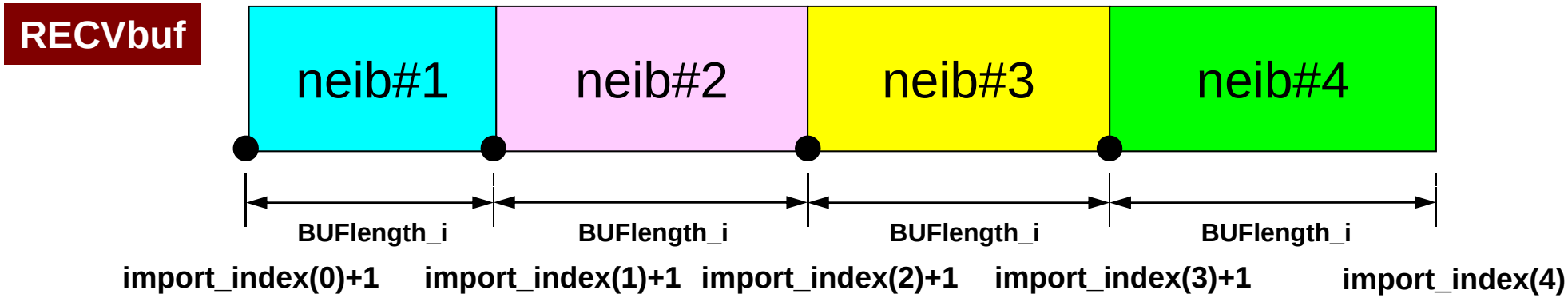
```
 do neib= 1, NEIBPETOT
   iS_i= import_index(neib-1) + 1
   iE_i= import_index(neib  )
   BUFlength_i= iE_i + 1 - iS_i

   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)                                      受信バッファから代入
     enddo
  enddo
```



受信（MPI_Sendrecv）

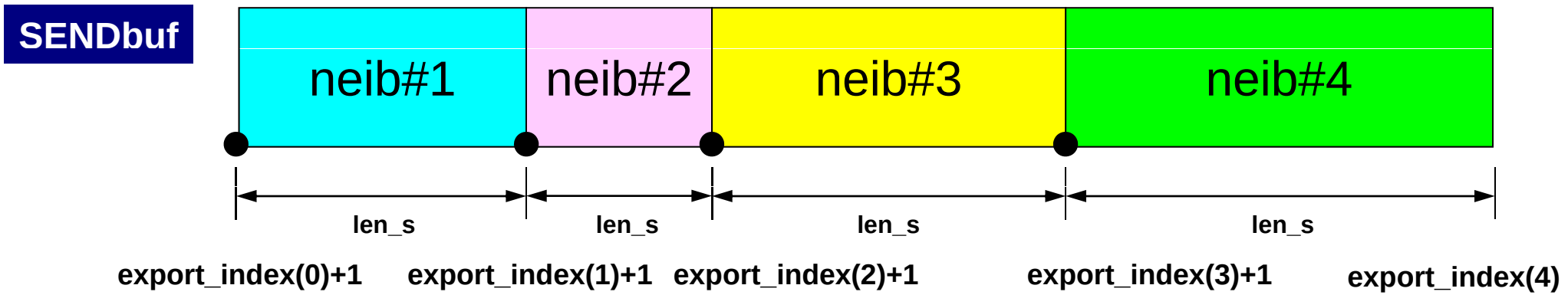
```
 do neib= 1, NEIBPETOT
   iS_i= import_index(neib-1) + 1
   iE_i= import_index(neib  )
   BUFlength_i= iE_i + 1 - iS_i

   call MPI_SENDRECV                                                  &
&         (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&          RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&          MPI_COMM_WORLD, stat_sr, ierr)
  enddo

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

受信バッファからの代入



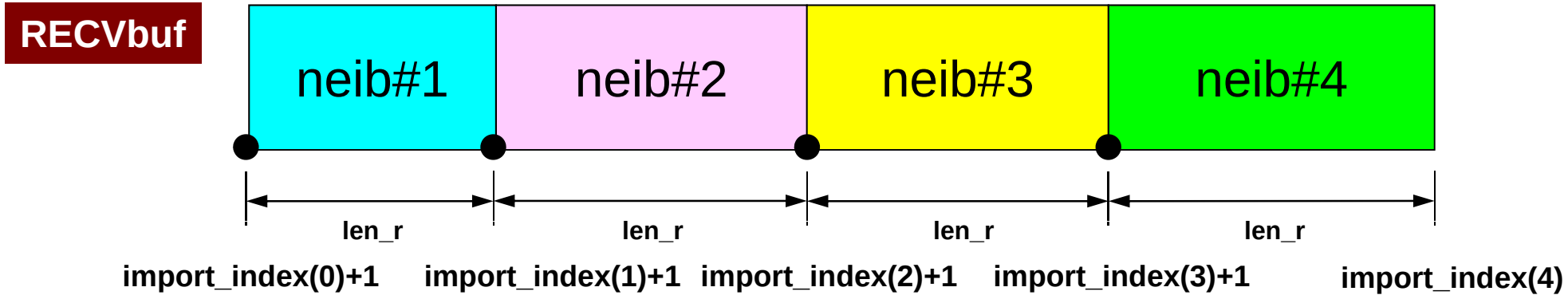
一般化された通信テーブル：送信

```
!C
!C-- PREPARE send  buffer
      do neib= 1, NEIBPETOT
        do k= export_index(neib-1)+1, export_index(neib)
          kk= export_item(k)
          SENDbuf(k)= BUF(kk)
        enddo
      enddo

!C
!C-- SEND
      do neib= 1, NEIBPETOT
        is   = export_index(neib-1) + 1
        len_s= export_index(neib) - export_index(neib-1)
        call MPI_ISEND (SENDbuf(is), len_s, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_send(neib), ierr)
      enddo
```



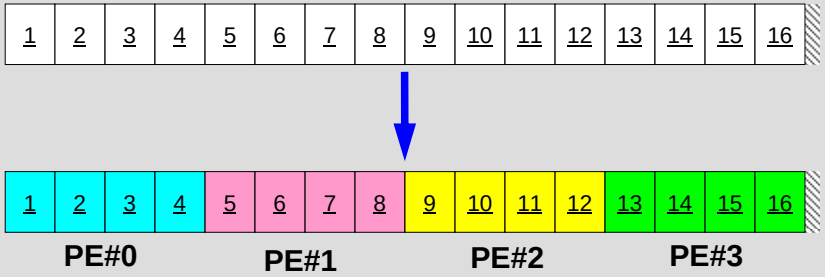
一般化された通信テーブル：受信

```
!C
!C-- RECV
      do neib= 1, NEIBPETOT
        ir   = import_index(neib-1) + 1
        len_r= import_index(neib) - import_index(neib-1)
        call MPI_IRECV (RECVbuf(ir), len_r, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_recv(neib), ierr)
      enddo

!C
!C-- WAITall for RECV
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

!C
!C-- update array
      do neib= 1, NEIBPETOT
        do k= import_index(neib-1)+1, import_index(neib)
          kk= import_item(k)
          BUF(kk)= RECVbuf(k)
        enddo
      enddo
```



実際何をやっているのか

```
!C
!C +------+
!C | SEND |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_ISEND (SENDbuf(neib), len_s, MPI_INTEGER,           &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,             &
     &                  request_send(neib), ierr)
      enddo
!C===


!C
!C +------+
!C | RECV |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_IRECV (RECVbuf(neib), len_r, MPI_INTEGER,           &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,             &
     &                  request_recv(neib), ierr)
      enddo
!C===
```



- 例えば，PE#1からはPE#0，PE#2に送信して，PE#0，PE#2から受信する。
- PE#0はPE#1から，PE#2はPE#1とPE#3から受信する。
- IRECVの部分では，相手のPE（NEIBPE(neib)）がISENDのループで自分に向けて送ったメッセージを受信する。
- 送信側のlen_s，受信側のlen_rは一致していなければならない。
  - NEIBPE(neib)がマッチしたときに通信が発生する。

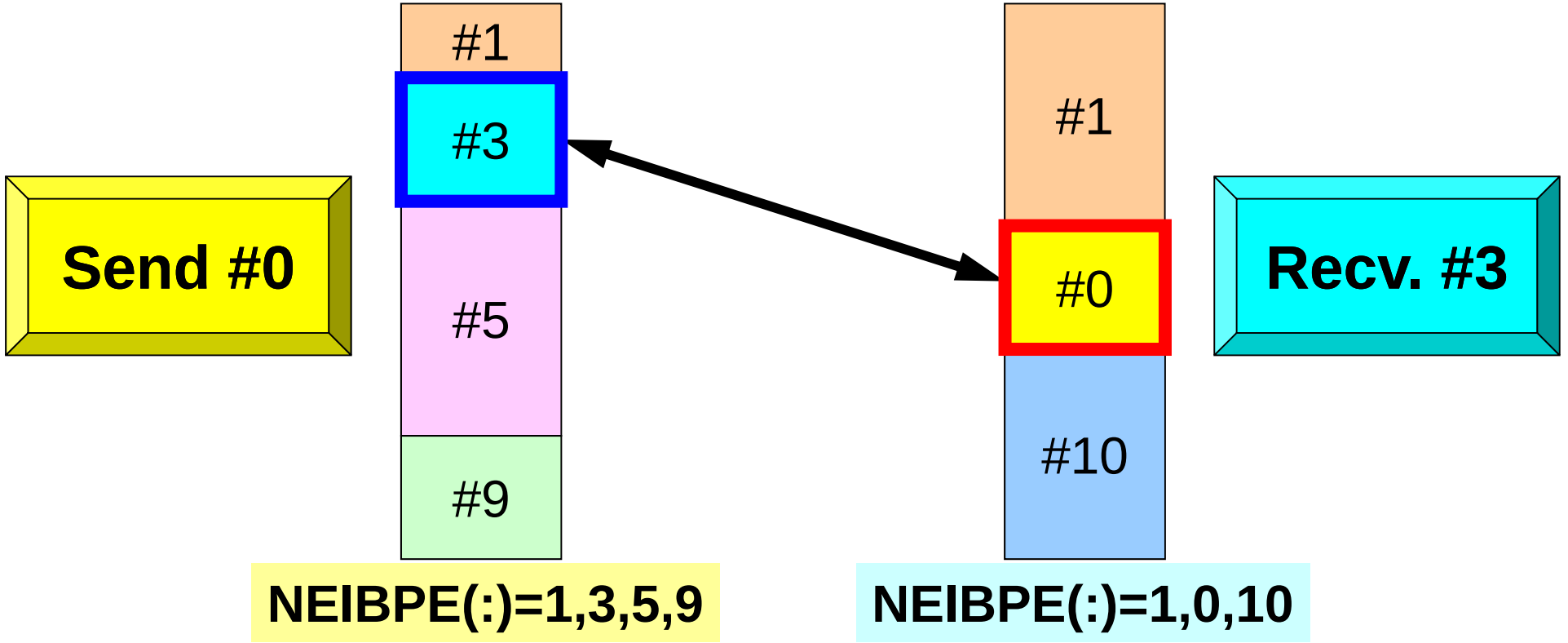
送信と受信の関係

```
 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
```

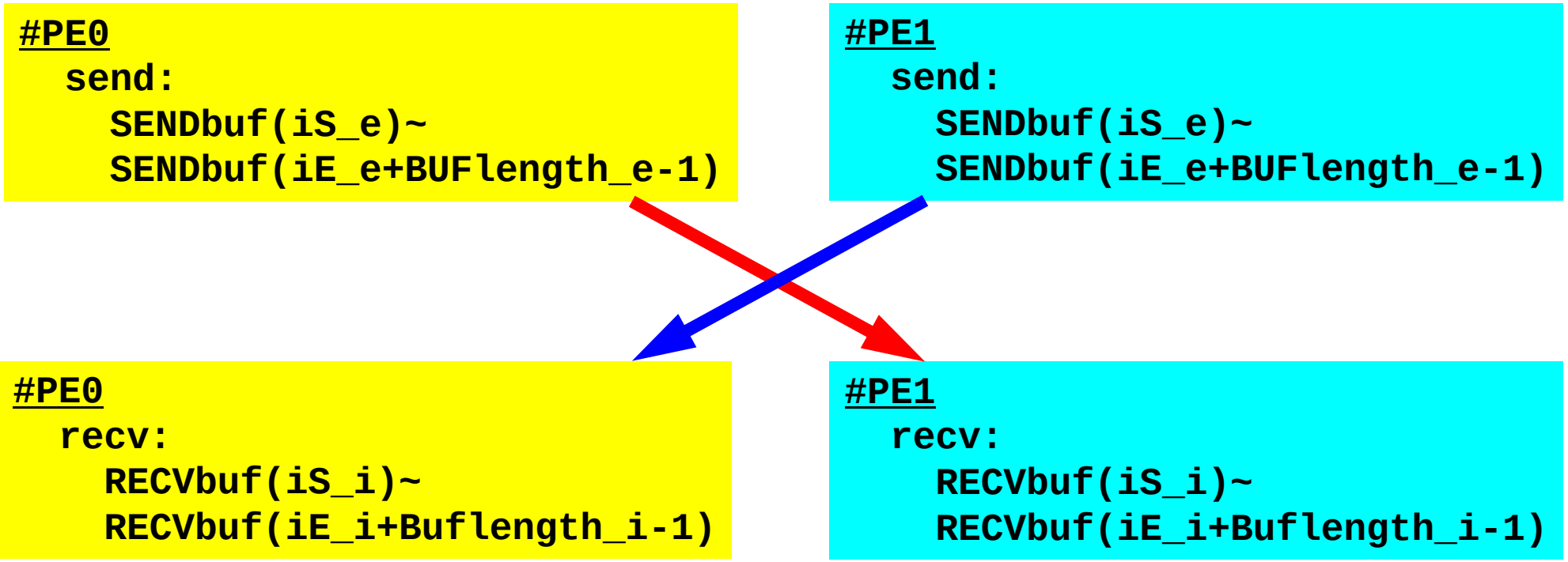
- 送信元・受信先プロセス番号，メッセージサイズ，内容の整合性 ！
- NEIBPE（neib）がマッチしたときに通信が起こる。

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。

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

なぜ SENDbuf，RECVbuf を用意するか?

- BUFを直接使っても良いのであるが･･･

```
!C
!C +------+
!C | SEND |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_ISEND (SENDbuf(neib), len_s, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_send(neib), ierr)
      enddo
!C===

!C
!C +------+
!C | RECV |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_IRECV (RECVbuf(neib), len_r, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_recv(neib), ierr)
      enddo
!C===
```

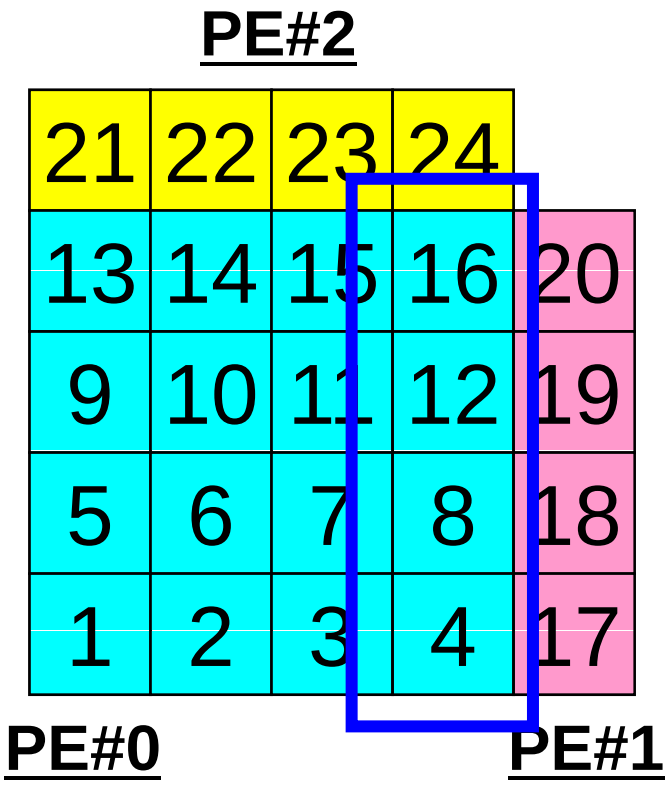
なぜ SENDbuf，RECVbuf を用意するか?

- BUFを直接使っても良いのであるが・・・
  - こんな感じになって使いにくい

```
!C
!C +------+
!C | SEND |
!C +------+
!C===
      do neib= 1, NEIBPETOT

        if (neib.eq.1) then
          ikk= 1
         else
          ikk= N
        endif

        if (my_rank.eq.0 .and. neib.eq.1) then
          ikk= N
        endif

        call MPI_ISEND (BUF(ikk), len_s, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,             &
     &                  request_send(neib), ierr)
      enddo
!C===
```

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
```

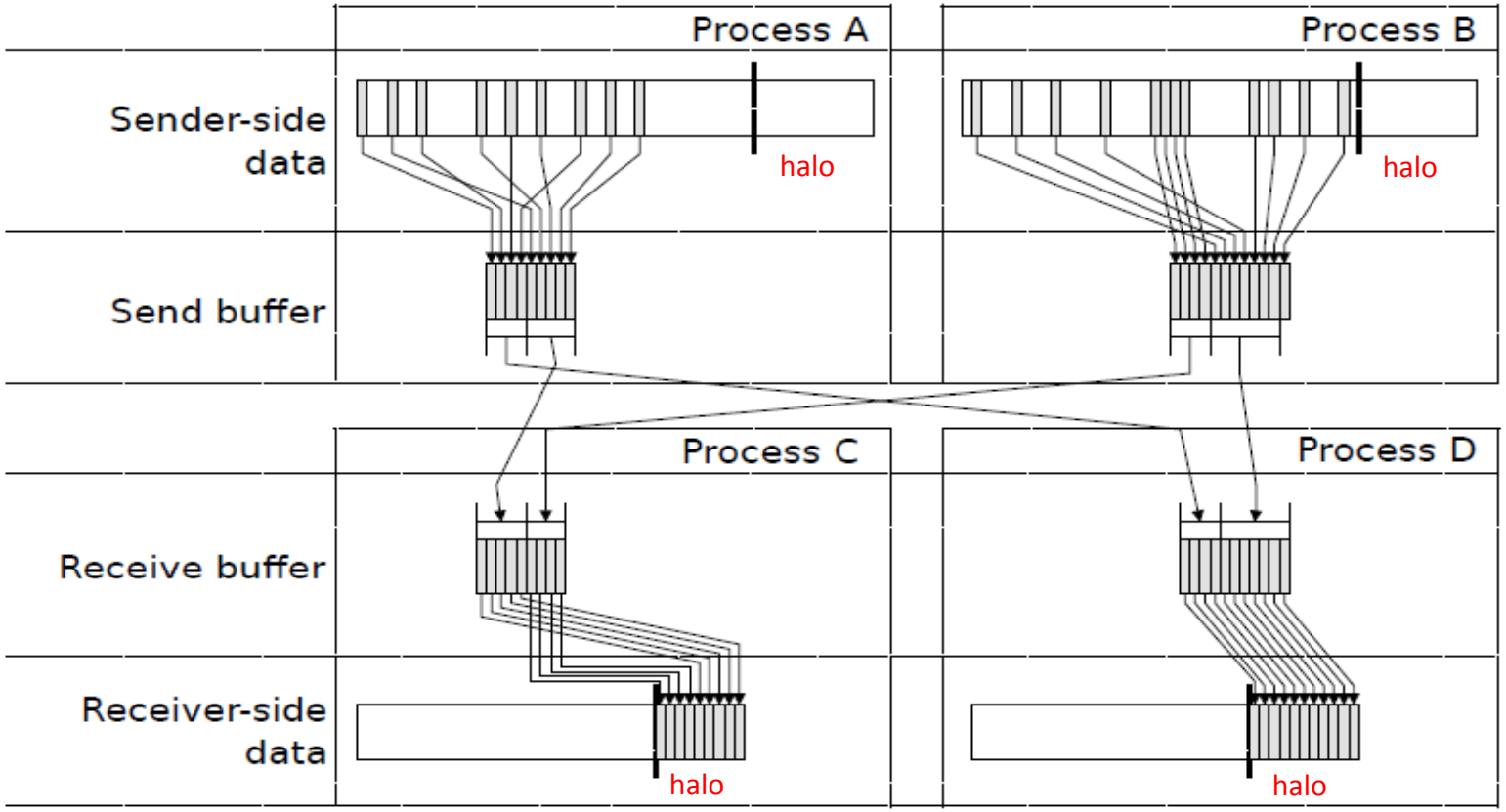


たとえば，この境界点は連続していないので，

- 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

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

演習

初期全体メッシュ

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

3領域に分割

演習

#PE2

21	22	23	24
16	17	18	19
11	12	13	14
6	7	8	

#PE1

23	24	25
18	19	20
13	14	15
8	9	10
	4	5

#PE0

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

3領域に分割

演習

#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）
○の部分をうめよ!

演習

#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）
○の部分をうめよ!

演習

#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）
○の部分をうめよ!

演習

#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

#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

演習

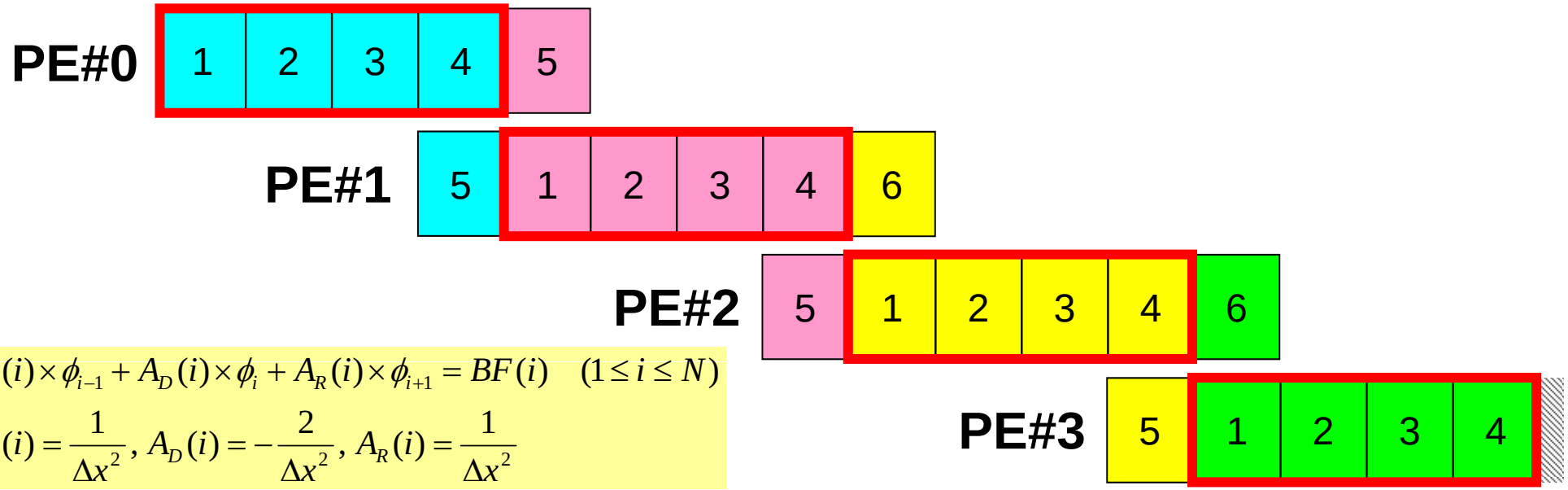
手 順

- 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

- 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
  - MPI
  - 再び一次元問題
  - 二次元問題
- 1対1通信の実装例
- 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

一次元差分法モデル局所データ構造
新しい番号付け

- 領域内の要素，オーバーラップ（本来領域外）要素の区別
  - 領域内要素（i=1~N）　　：内点（Interior/Internal Points）

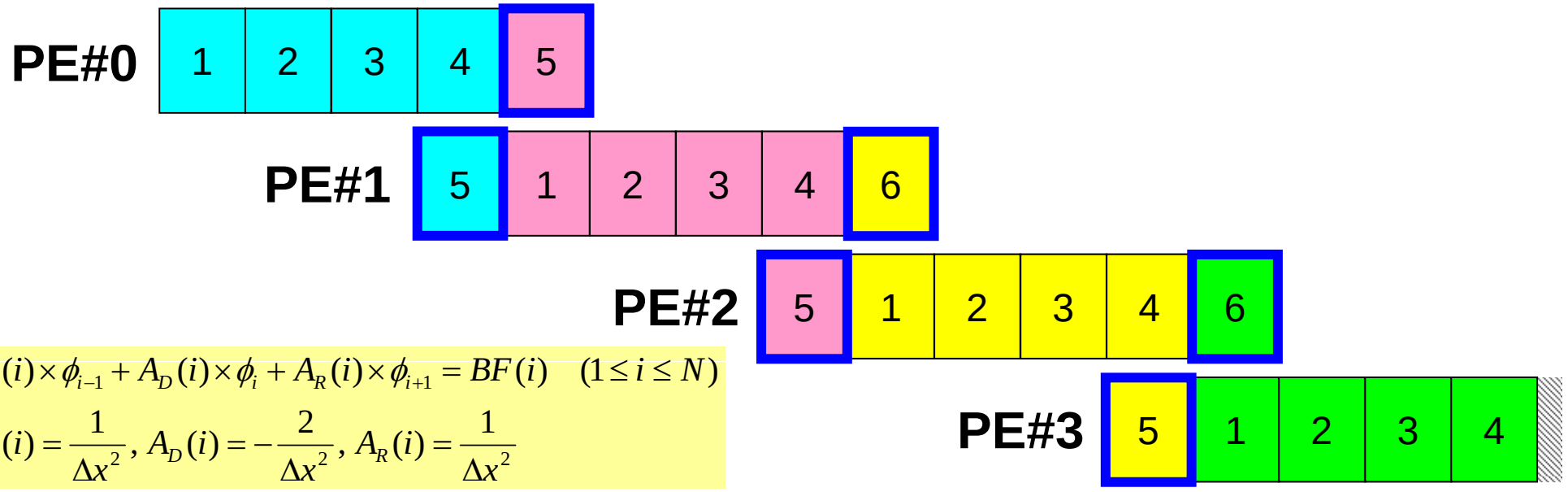


$$A_L(i)\times\phi_{i-1}+A_D(i)\times\phi_i+A_R(i)\times\phi_{i+1}=BF(i)\quad(1\le i\le N)$$

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

一次元差分法モデル局所データ構造
新しい番号付け

- 領域内要素，オーバーラップ（本来領域外）要素の区別
  - 領域内要素（i=1~N）　　：内点（Interior/Internal Points）
  - オーバーラップ要素（i=N+1, N+2）：外点（Exterior/External Points）
    - i=N+1，i=N+2の要素が存在しない場合もある

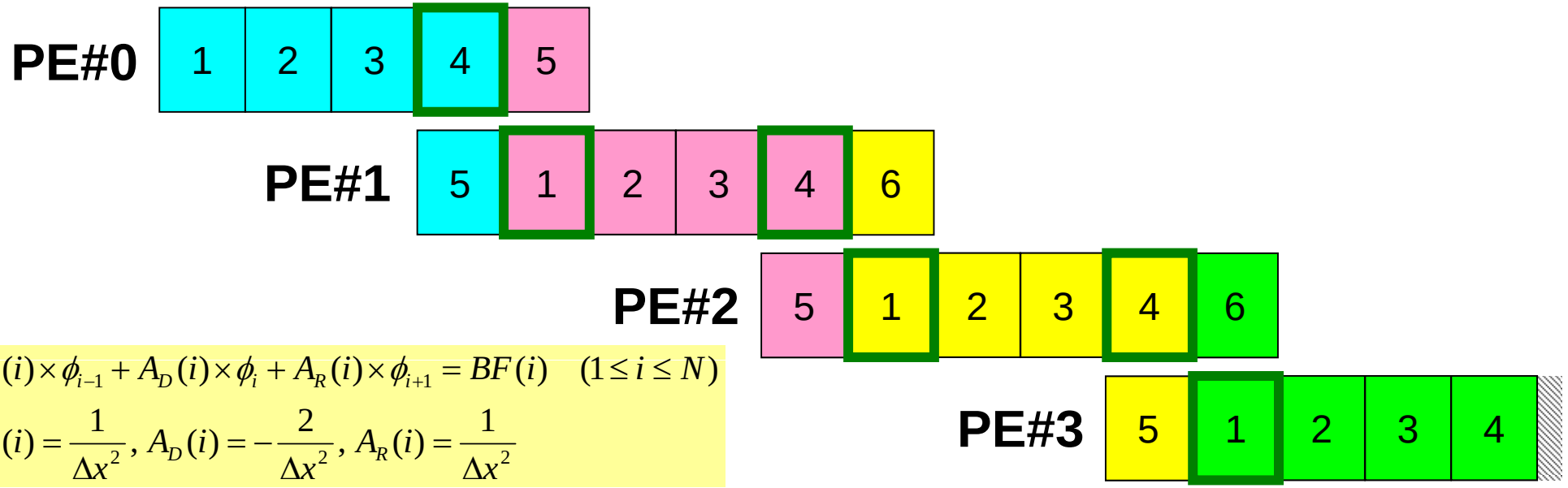


$$A_L(i)\times\phi_{i-1}+A_D(i)\times\phi_i+A_R(i)\times\phi_{i+1}=BF(i)\quad(1\le i\le N)$$

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

一次元差分法モデル局所データ構造
新しい番号付け

- 領域内要素，オーバーラップ（本来領域外）要素の区別
  - 領域内要素（i=1~N）　　：内点（Interior/Internal Points）
  - オーバーラップ要素（i=N+1, N+2）：外点（Exterior/External Points）
    - i=N+1，i=N+2の要素が存在しない場合もある
  - 「内点」のうち他領域の「外点」となっている要素：境界点（Boundary Points）

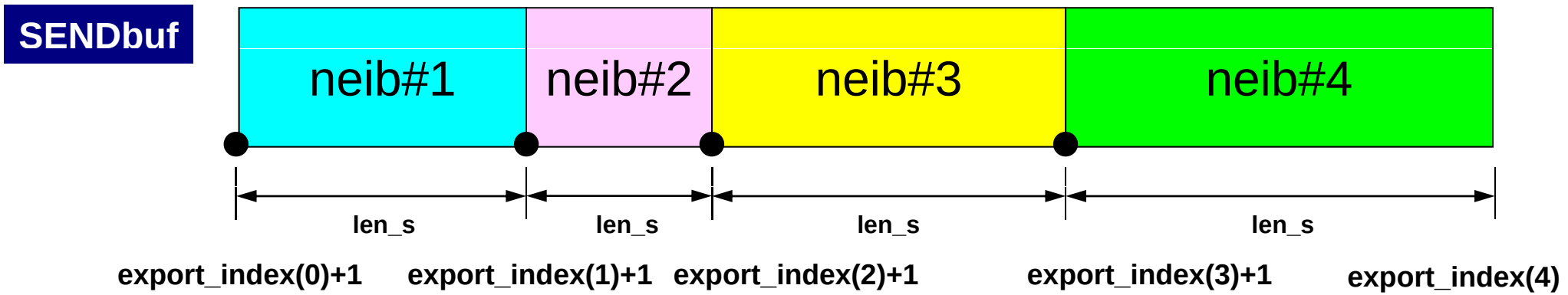


$$A_L(i)\times\phi_{i-1}+A_D(i)\times\phi_i+A_R(i)\times\phi_{i+1}=BF(i)\quad(1\le i\le N)$$

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

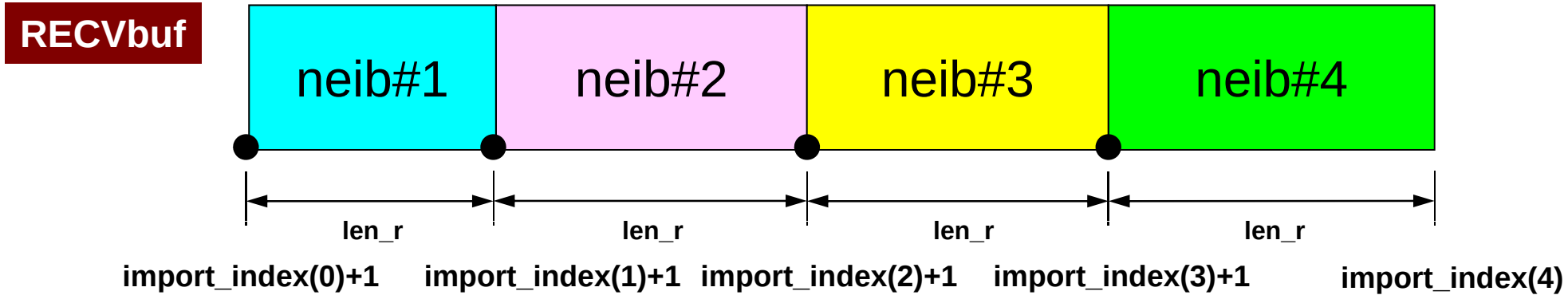
一般化された通信テーブル：送信

```
!C
!C-- PREPARE send  buffer
      do neib= 1, NEIBPETOT
        do k= export_index(neib-1)+1, export_index(neib)
          kk= export_item(k)
          SENDbuf(k)= BUF(kk)
        enddo
      enddo

!C
!C-- SEND
      do neib= 1, NEIBPETOT
        is   = export_index(neib-1) + 1
        len_s= export_index(neib) - export_index(neib-1)
        call MPI_ISEND (SENDbuf(is), len_s, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_send(neib), ierr)
      enddo
```

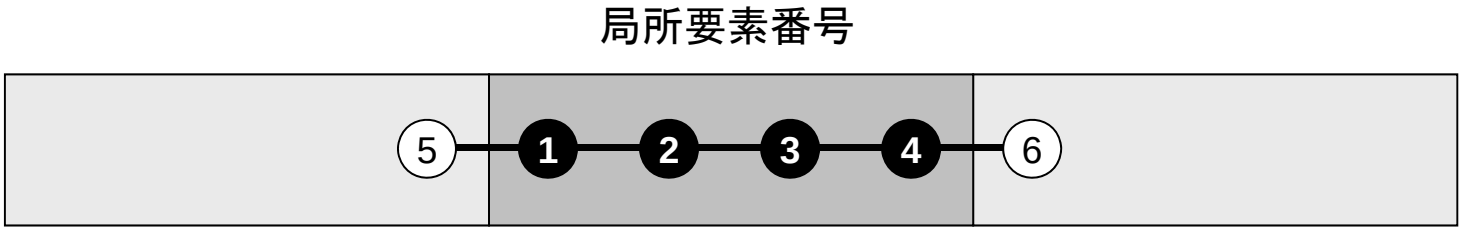


一般化された通信テーブル：受信

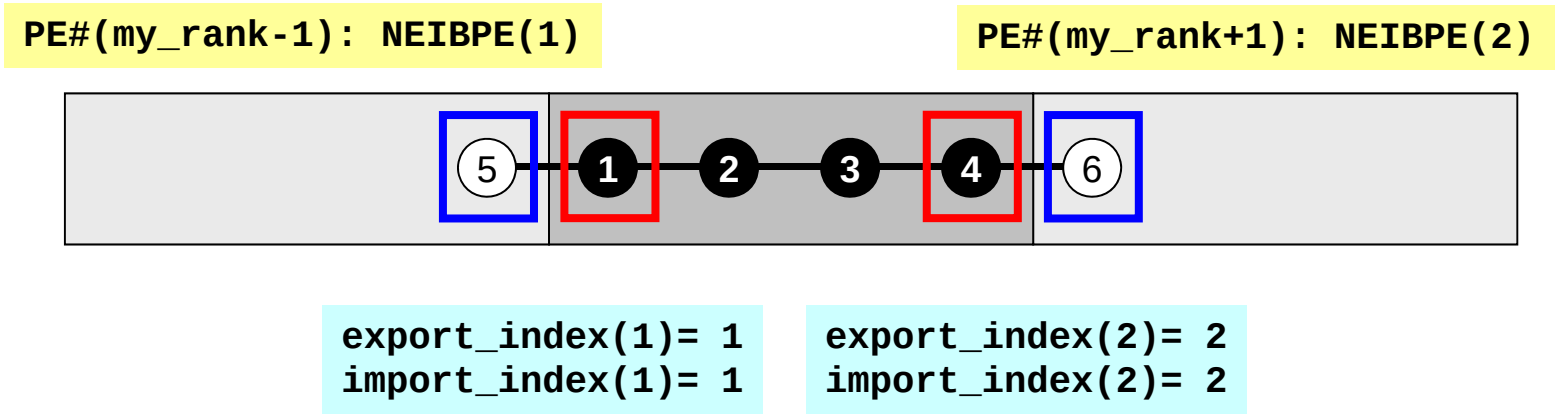
```
!C
!C-- RECV
      do neib= 1, NEIBPETOT
        ir   = import_index(neib-1) + 1
        len_r= import_index(neib) - import_index(neib-1)
        call MPI_IRECV (RECVbuf(ir), len_r, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_recv(neib), ierr)
      enddo

!C
!C-- WAITall for RECV
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

!C
!C-- update array
      do neib= 1, NEIBPETOT
        do k= import_index(neib-1)+1, import_index(neib)
          kk= import_item(k)
          BUF(kk)= RECVbuf(k)
        enddo
      enddo
```



1D差分法の並列計算に必要な情報（1/4）
内点・外点・（境界点）



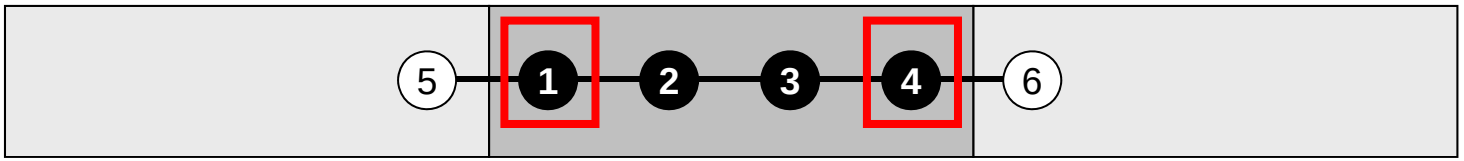
1D差分法の並列計算に必要な情報(2/4)
隣接プロセッサ情報



この領域から2つの隣接領域へ合計2つの要素(境界点□)を送信し,
2つの要素(外点□)を受信する

1D差分法の並列計算に必要な情報(3/4) 通信テーブル

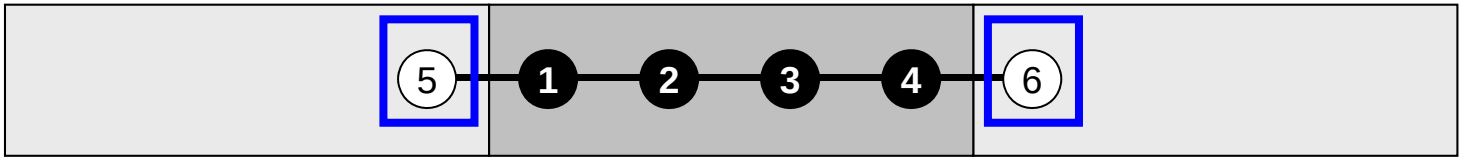
隣接PEへの送信(境界点)




```
export_index(1)= 1
export_item (1)= 1
```

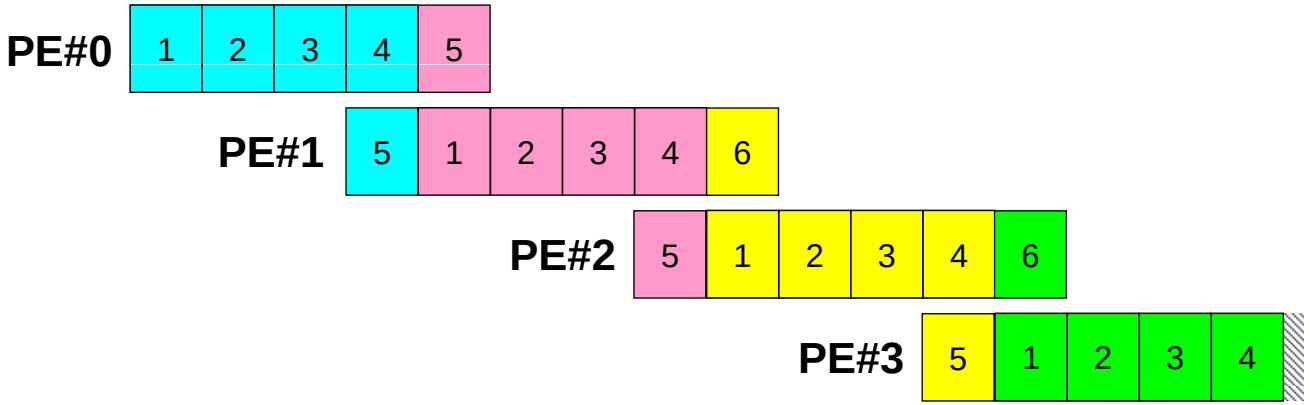
```
export_index(2)= 2
export_item(2) = 4
```

隣接PEからの受信(外点)



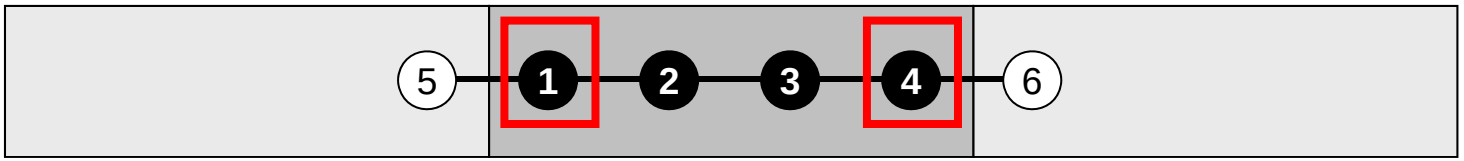

```
import_index(1)= 1
import_item (1)= 5
```

```
import_index(2)= 2
import_item (2)= 6
```



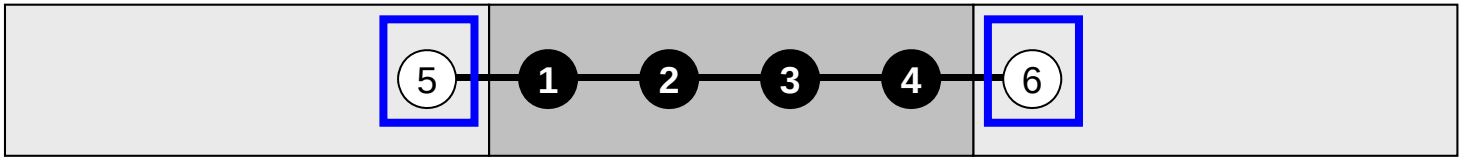
1D差分法の並列計算に必要な情報(4/4) 通信テーブル

隣接PEへの送信(境界点)

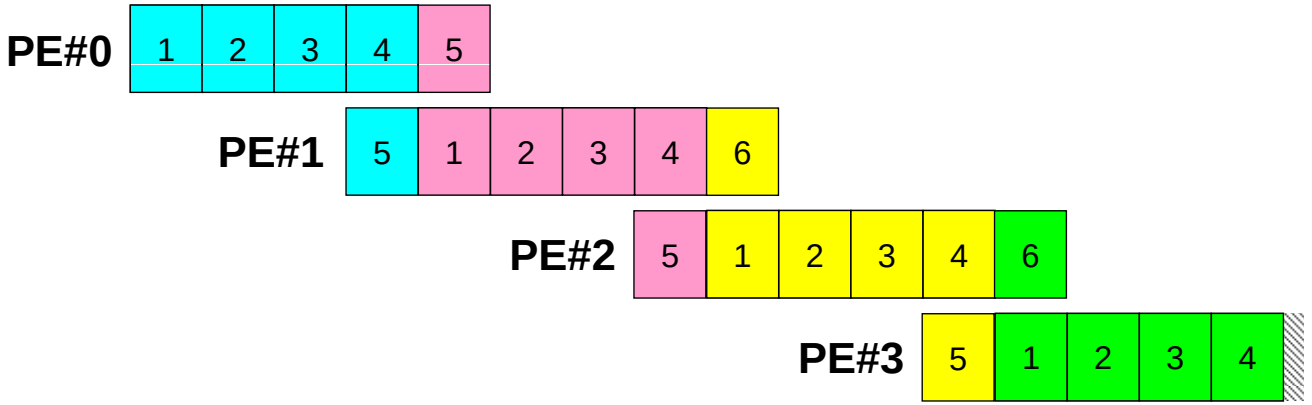



```
SENDbuf(1)=BUF(1)
SENDbuf(2)=BUF(4)
```

隣接PEからの受信(外点)

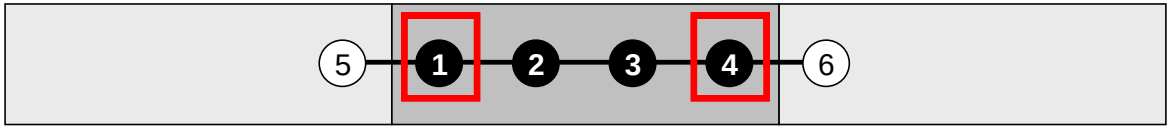



```
BUF(5)=RECVbuf(1)
BUF(6)=RECVbuf(2)
```



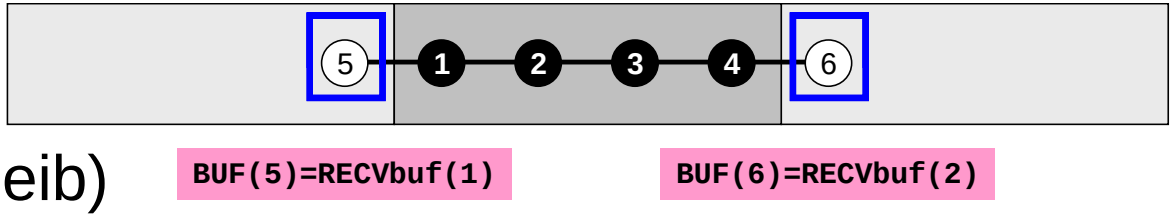
送信：一次元問題

- 送信相手
  - NEIBPETOT，NEIBPE(neib)



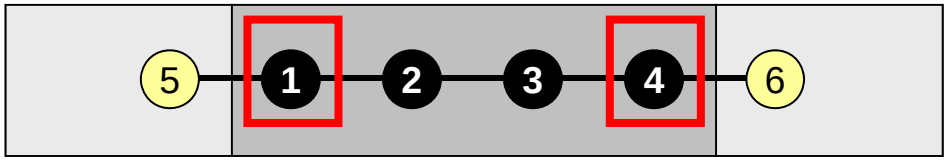
    - NEIBPETOT=2, NEIB(1)= my_rank-1, NEIB(2)= my_rank+1
- それぞれの送信相手に送るメッセージサイズ
  - export_index(neib), neib= 0, NEIBPETOT
    - export_index(0)=0, export_index(1)= 1, export_index(2)= 2
- 「境界点」番号
  - export_item(k), k= 1, export_index(NEIBPETOT)
    - export_item(1)= 1, export_item(2)= N
- それぞれの送信相手に送るメッセージ
  - SENDbuf(k), k= 1, export_index(NEIBPETOT)
    - SENDbuf(1)= BUF(1), SENDbuf(2)= BUF(N)

受信:一次元問題

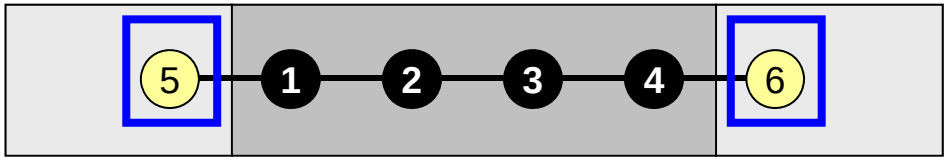


- 受信相手
  - NEIBPETOT, NEIBPE(neib)
    - NEIBPETOT=2, NEIB(1)= my_rank-1, NEIB(2)= my_rank+1
- それぞれの受信相手から受け取るメッセージサイズ
  - import_index(neib), neib= 0, NEIBPETOT
    - import_index(0)=0, import_index(1)= 1, import_index(2)= 2
- 「外点」番号
  - import_item(k), k= 1, import_index(NEIBPETOT)
    - import_item(1)= N+1, import_item(2)= N+2
- 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)
    - BUF(N+1)=RECVbuf(1), BUF(N+2)=RECVbuf(2)

一般化された通信テーブル



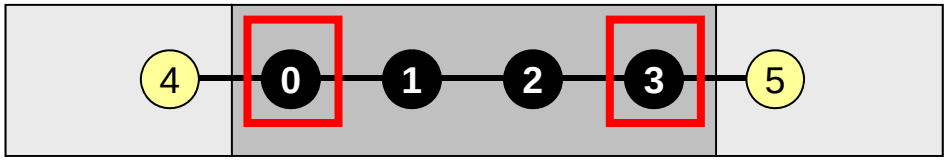
SENDbuf(1)=BUF(1)　　SENDbuf(2)=BUF(4)



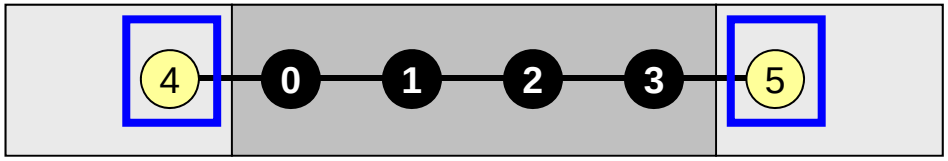
BUF(5)=RECVbuf(1)　　BUF(6)=RECVbuf(2)

```
NEIBPETOT= 2
NEIBPE(1)= my_rank - 1
NEIBPE(2)= my_rank + 1

import_index(1)= 1
import_index(2)= 2
import_item (1)= N+1
import_item (2)= N+2

export_index(1)= 1
export_index(2)= 2
export_item (1)= 1
export_item (2)= N

if (my_rank.eq.0) then
  import_item (1)= N+1
  export_item (1)= N
  NEIBPE(1)= my_rank+1
endif
```

一般化された通信テーブル:C言語



SENDbuf[0]=BUF[0]　　SENDbuf[1]=BUF[3]



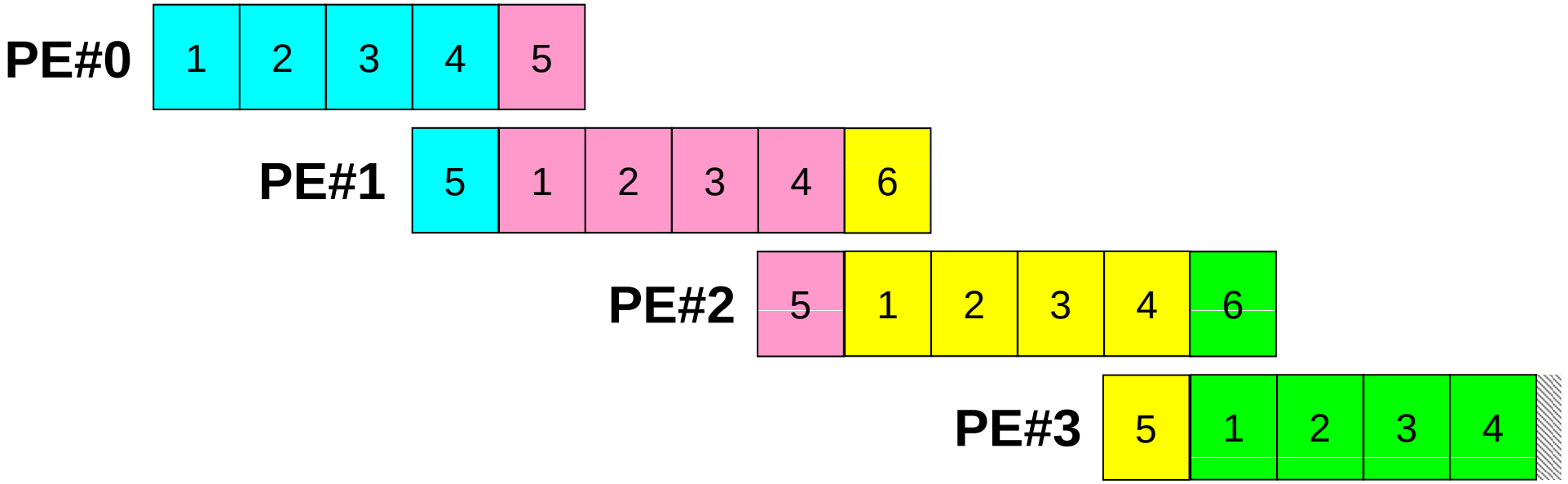
BUF[4]=RECVbuf[0]　　BUF[5]=RECVbuf[1]

```
NEIBPETOT= 2
NEIBPE[0]= my_rank - 1
NEIBPE[1]= my_rank + 1

import_index[1]= 0
import_index[2]= 1
import_item [0]= N
import_item [1]= N+1

export_index[1]= 0
export_index[2]= 1
export_item [0]= 0
export_item [1]= N-1

if (my_rank.eq.0) then
  import_item [0]= N
  export_item [0]= N-1
  NEIBPE[0]= my_rank+1
endif
```

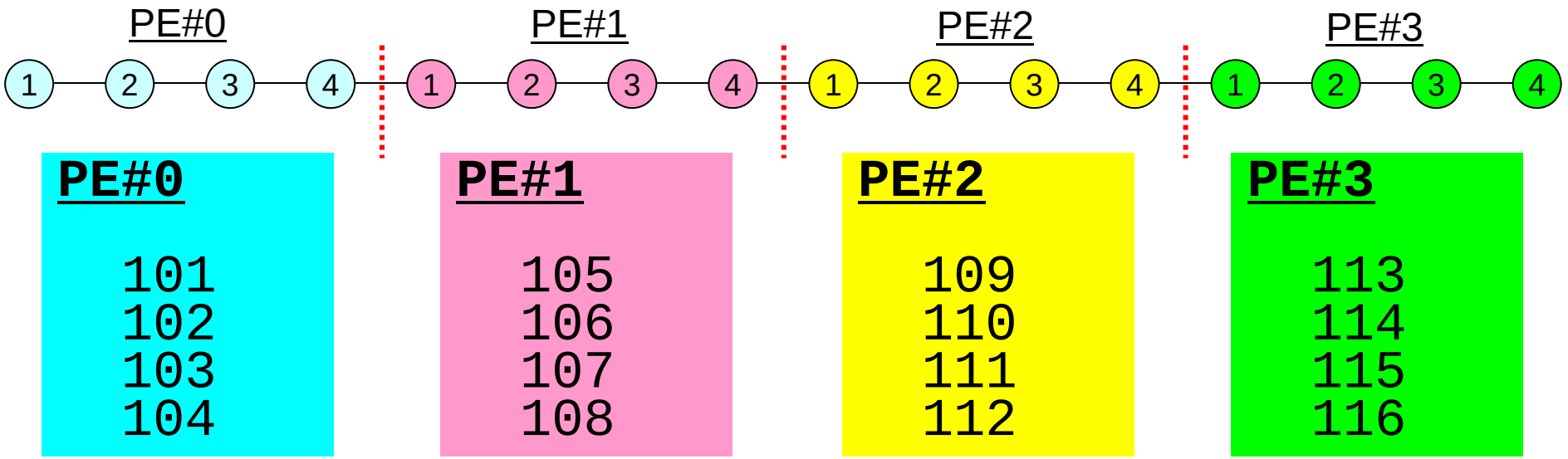
- 差分法による一次元熱伝導方程式ソルバー
  - 概要
  - 並列化にあたって：データ構造
- 1対1通信とは
  - MPI
  - 一次元問題
  - 二次元問題
- 1対1通信の実装例
- 差分法による一次元熱伝導方程式ソルバーの並列化
  - 課題S2

一次元モデル例題(1/10)

- オーバーラップのある一次元分散データ(NG=16，4領域)について，各プロセッサで「内点」に関する情報を入力し，「境界点」の情報を，各隣接領域に「外点」の情報として配信する。

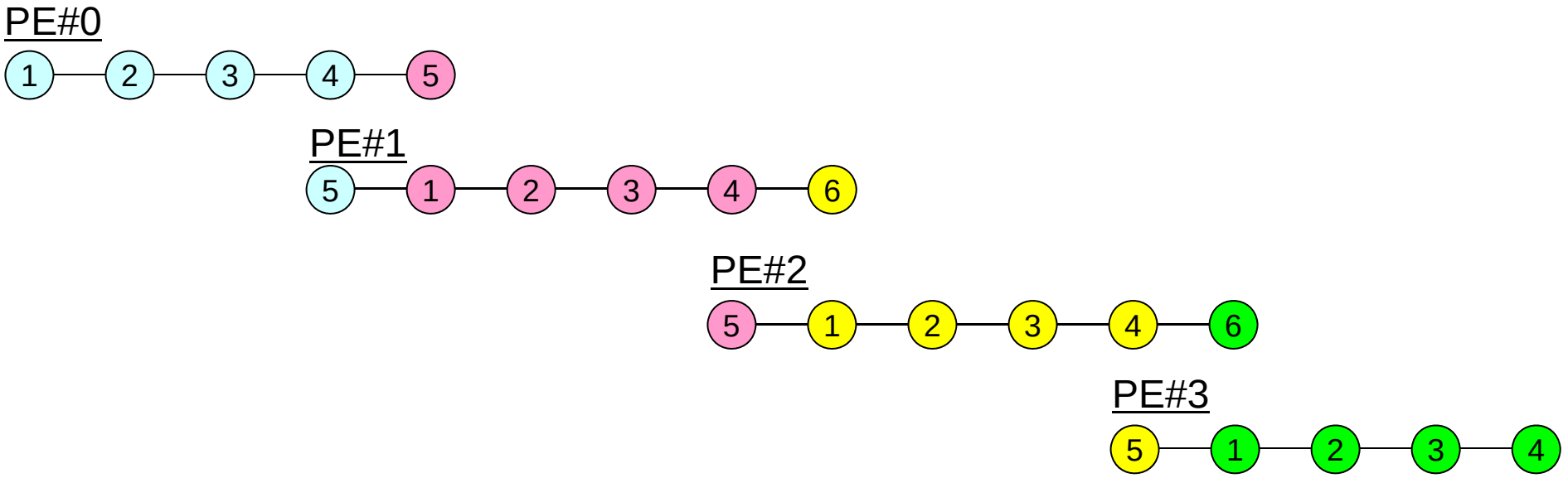


演算内容(1/3)



- 各PEの内点(i=1~N(=4))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」として配信する。

演算内容(2/3):送信，受信前



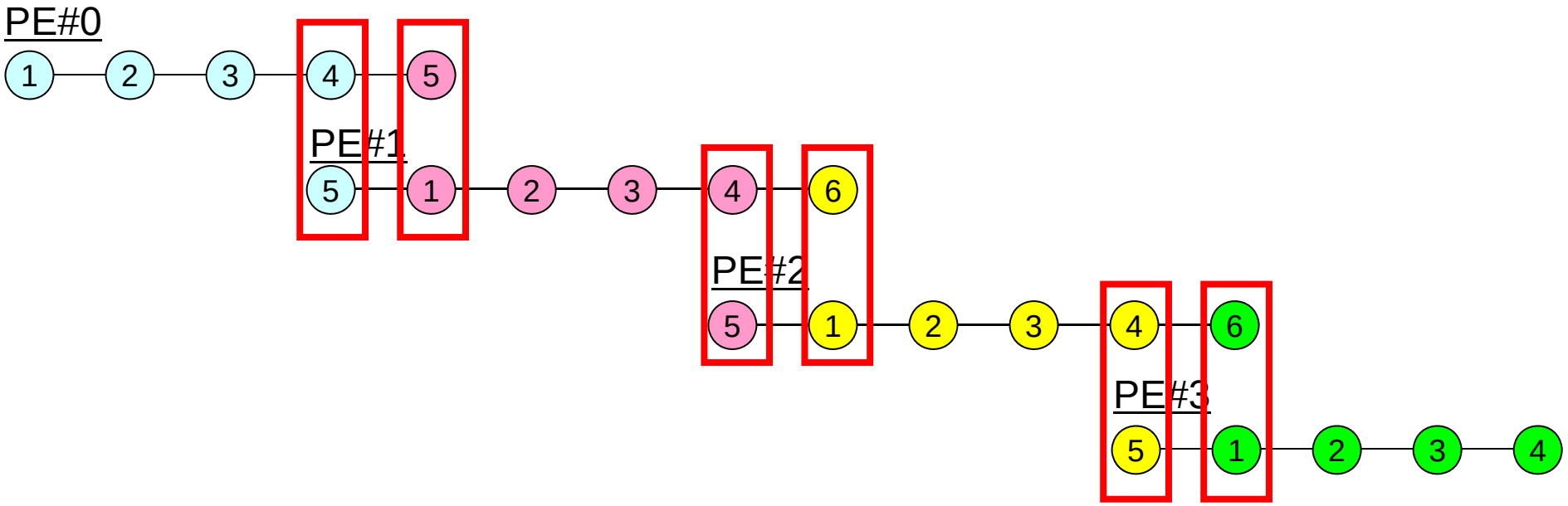
PE#0
```
i=1 101
i=2 102
i=3 103
i=4 104
i=5  ?
```

PE#1
```
i=1 105
i=2 106
i=3 107
i=4 108
i=5  ?
i=6  ?
```

PE#2
```
i=1 109
i=2 110
i=3 111
i=4 112
i=5  ?
i=6  ?
```

PE#3
```
i=1 113
i=2 114
i=3 115
i=4 116
i=5  ?
```

演算内容(3/3):送信，受信後



PE#0
i=1 101
i=2 102
i=3 103
i=4 104
i=5 105

PE#1
i=1 105
i=2 106
i=3 107
i=4 108
i=5 104
i=6 109

PE#2
i=1 109
i=2 110
i=3 111
i=4 112
i=5 108
i=6 113

PE#3
i=1 113
i=2 114
i=3 115
i=4 116
i=5 112

サンプルプログラム

FORTRAN

```
$ cd <$T-S2>
$ mpif90 –Oss -noparallel 1d-srb1.f
$ バッチ処理実行（4プロセス）
```

ISEND/IRECV

```
$ mpif90 –Oss -noparallel 1d-srb2.f
$ バッチ処理実行（4プロセス）
```

SENDRECV

C

```
$ cd <$T-S2>
$ mpicc –Os -noparallel 1d-srb1.c
$ バッチ処理実行（4プロセス）
```

ISEND/IRECV

```
$ mpicc –Os -noparallel 1d-srb2.c
$ バッチ処理実行（4プロセス）
```

SENDRECV

一次元モデル例題: 1d-srb1.f（2/10）

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

      integer(kind=4) :: my_rank, PETOT
      integer(kind=4) :: N, NEIBPETOT, BUFlength
      integer(kind=4), dimension(2) :: NEIBPE

      integer(kind=4), dimension(0:2) :: import_index, export_index
      integer(kind=4), dimension(  2) :: import_item , export_item

      integer(kind=4), dimension(2) :: SENDbuf, RECVbuf

      integer(kind=4), dimension(:), allocatable :: BUF

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

!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 )
!C===
```

一次元モデル例題: 1d-srb1.f（3/10）

各PEにおいて局所データ（内点の値）読み込み

```
!C
!C +------+
!C | INIT |
!C +------+
!C===
      N= 4
      allocate (BUF(N+2))
      BUF= 0

      if (my_rank.eq.0) open (11, file='1d.0', status='unknown')
      if (my_rank.eq.1) open (11, file='1d.1', status='unknown')
      if (my_rank.eq.2) open (11, file='1d.2', status='unknown')
      if (my_rank.eq.3) open (11, file='1d.3', status='unknown')

        do i= 1, N
          read (11,*) BUF(i)
        enddo

      close (11)
```

N: 内点数

PE#0	PE#1	PE#2	PE#3
101	105	109	113
102	106	110	114
103	107	111	115
104	108	112	116

一次元モデル例題: 1d-srb1.f（4/10）

各領域における外点

```
      iSTART= 1
      iEND  = N+2
      if (my_rank.eq.0      ) iEND  = N+1
      if (my_rank.eq.PETOT-1) iEND  = N+1
      do i= iSTART, iEND
        write (*,'(a, 3i8)') '%%% before', my_rank, i, BUF(i)
      enddo
```

PE#0
- iS= 1
- iE= N+1
- 外点数:1

```
%%% before    0     1   101
%%% before    0     2   102
%%% before    0     3   103
%%% before    0     4   104
%%% before    0     5     0
```

PE#1
- iS= 1
- iE= N+2
- 外点数:2

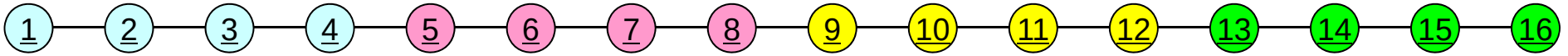
```
%%% before    1     1   105
%%% before    1     2   106
%%% before    1     3   107
%%% before    1     4   108
%%% before    1     5     0
%%% before    1     6     0
```

PE#2
- iS= 1
- iE= N+2
- 外点数:2

```
%%% before    2     1   109
%%% before    2     2   110
%%% before    2     3   111
%%% before    2     4   112
%%% before    2     5     0
%%% before    2     6     0
```

PE#3
- iS= 1
- iE= N+1
- 外点数:1

```
%%% before    3     1   113
%%% before    3     2   114
%%% before    3     3   115
%%% before    3     4   116
%%% before    3     5     0
```



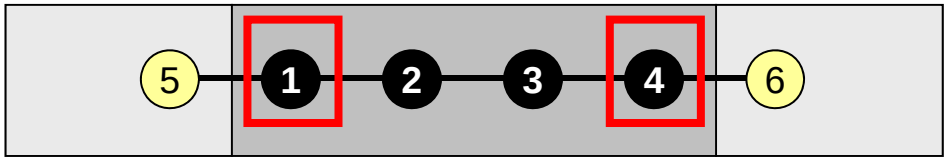
一次元モデル例題: 1d-srb1.f（5/10）

隣接領域，領域数

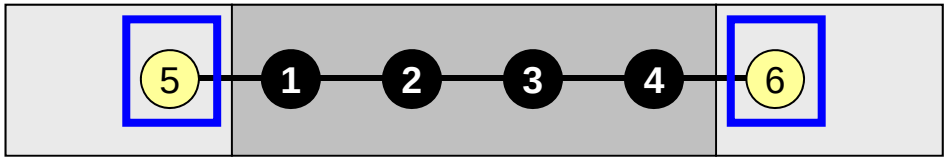
```
NEIBPETOT= 2
if (my_rank.eq.0       ) NEIBPETOT= 1
if (my_rank.eq.PETOT-1) NEIBPETOT= 1
if (PETOT.eq.1)          NEIBPETOT= 0

NEIBPE(1)= my_rank - 1
NEIBPE(2)= my_rank + 1

if (my_rank.eq.0       ) NEIBPE(1)= my_rank + 1
if (my_rank.eq.PETOT-1) NEIBPE(1)= my_rank - 1

import_index= 0
export_index= 0
import_item = 0
export_item = 0

import_index(1)= 1
import_index(2)= 2
import_item (1)= N+1
import_item (2)= N+2

export_index(1)= 1
export_index(2)= 2
export_item (1)= 1
export_item (2)= N

if (my_rank.eq.0) then
  import_item (1)= N+1
  export_item (1)= N
endif

BUFlength= 1
```

隣接PE数（NEIBPETOT），
隣接PE（NEIBPE）を決定

一般化された通信テーブル



SENDbuf(1)=BUF(1)　　SENDbuf(2)=BUF(4)



BUF(5)=RECVbuf(1)　　BUF(6)=RECVbuf(2)

```
NEIBPETOT= 2
NEIBPE(1)= my_rank - 1
NEIBPE(2)= my_rank + 1

import_index(1)= 1
import_index(2)= 2
import_item (1)= N+1
import_item (2)= N+2

export_index(1)= 1
export_index(2)= 2
export_item (1)= 1
export_item (2)= N

if (my_rank.eq.0) then
  import_item (1)= N+1
  export_item (1)= N
  NEIBPE(1)= my_rank+1
endif
```

一次元モデル例題: 1d-srb1.f（6/10）

通信識別子等の宣言，記憶領域確保

```
!C
!C +-----------------------------+
!C | INIT. arrays for MPI_WAITALL |
!C +-----------------------------+
!C===
      allocate (stat_send(MPI_STATUS_SIZE,NEIBPETOT))
      allocate (stat_recv(MPI_STATUS_SIZE,NEIBPETOT))
      allocate (request_send(NEIBPETOT))
      allocate (request_recv(NEIBPETOT))
!C===
```

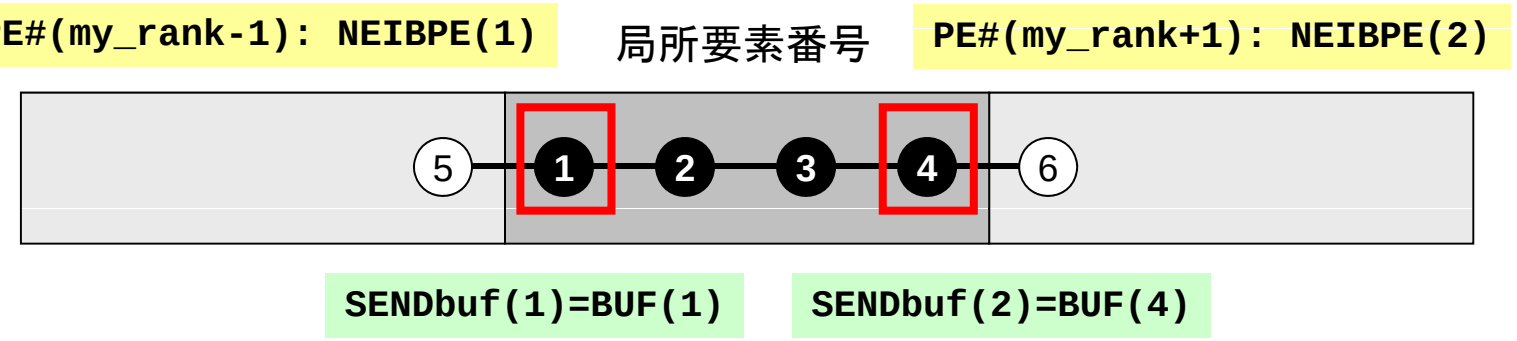
- 記憶領域を確保するだけで良い!!
- 通信識別子：request handle（通信リクエスト，とも言う）
  - 送信用：request_send，受信用：request_recv
- 状況オブジェクト配列：status object
  - 送信用：stat_send，受信用：stat_recv
- MPI_STATUS_SIZE
  - “mpif.h”，”mpi.h” で定義されている。
  - 勝手にこの数字を変更してはいけない。

一次元モデル例題: 1d-srb1.f（7/10）

境界点情報送信準備

```
!C
!C +----------------------+
!C | PREPARE send  buffer |
!C +----------------------+
!C===
      do neib= 1, NEIBPETOT
        do k= export_index(neib-1)+1, export_index(neib)
          kk= export_item(k)
          SENDbuf(k)= BUF(kk)
        enddo
      enddo
!C===
```

隣接プロセッサに BUF(1), BUF(N) を送る準備をする
境界点情報を SENDbufに代入



一次元モデル例題: 1d-srb1.f（8/10）

境界点情報の送信

```
!C
!C +------+
!C | SEND |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        is   = export_index(neib-1) + 1
        len_s= export_index(neib) - export_index(neib-1)
        call MPI_ISEND (SENDbuf(is), len_s, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_send(neib), ierr)
      enddo
!C===

!C
!C +------+
!C | RECV |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        ir   = import_index(neib-1) + 1
        len_r= import_index(neib) - import_index(neib-1)
        call MPI_IRECV (RECVbuf(ir), len_r, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_recv(neib), ierr)
      enddo
!C===

!C
!C +------------------+
!C | WAITall for RECV |
!C +------------------+
!C===
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)
!C===
```

SENDbuf(is)から始まる，長さlen_s(=1)のメッセージを隣接PE(NEIBPE(neib))に送信する。

一次元モデル例題: 1d-srb1.f（8/10）

外点情報の送信

```
!C
!C +------+
!C | SEND |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        is   = export_index(neib-1) + 1
        len_s= export_index(neib) - export_index(neib-1)
        call MPI_ISEND (SENDbuf(is), len_s, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_send(neib), ierr)
      enddo
!C===

!C
!C +------+
!C | RECV |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        ir   = import_index(neib-1) + 1
        len_r= import_index(neib) - import_index(neib-1)
        call MPI_IRECV (RECVbuf(ir), len_r, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_recv(neib), ierr)
      enddo
!C===

!C
!C +------------------+
!C | WAITall for RECV |
!C +------------------+
!C===
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)
!C===
```

RECVbuf(ir)から始まる，長さlen_r(=1)のメッセージを隣接PE(NEIBPE(neib))から受信する。

一次元モデル例題: 1d-srb1.f（8/10）

```
!C
!C +------+
!C | SEND |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_ISEND (SENDbuf(neib), len_s, MPI_INTEGER,           &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,             &
     &                  request_send(neib), ierr)
      enddo
!C===


!C
!C +------+
!C | RECV |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_IRECV (RECVbuf(neib), len_r, MPI_INTEGER,           &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,             &
     &                  request_recv(neib), ierr)
      enddo
!C===

!C
!C +------------------+
!C | WAITall for RECV |
!C +------------------+
!C===
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)
!C===
```

このメッセージのあと，RECVbufの中身を利用することが可能となる。

一次元モデル例題: 1d-srb1.f（8/10）

```
!C
!C +------+
!C | SEND |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_ISEND (SENDbuf(neib), len_s, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_send(neib), ierr)
      enddo
!C===


!C
!C +------+
!C | RECV |
!C +------+
!C===
      do neib= 1, NEIBPETOT
        call MPI_IRECV (RECVbuf(neib), len_r, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_recv(neib), ierr)
      enddo
!C===

!C
!C +------------------+
!C | WAITall for RECV |
!C +------------------+
!C===
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)
!C===
```

通信識別子（受信）

このメッセージのあと，RECVbufの中身を利用することが可能となる。

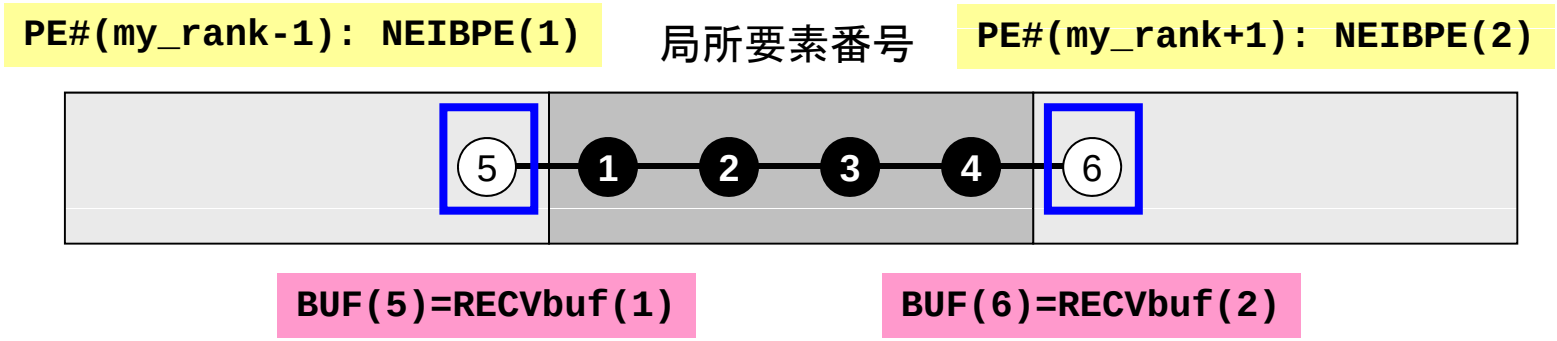
通信識別子　状況オブジェクト配列

一次元モデル例題: 1d-srb1.f（9/10）

外点情報代入

```
!C
!C +--------------+
!C | update array |
!C +--------------+
!C===
      do neib= 1, NEIBPETOT
        do k= import_index(neib-1)+1, import_index(neib)
          kk= import_item(k)
          BUF(kk)= RECVbuf(k)
        enddo
      enddo

      do i= iSTART, iEND
        write (*,'(a, 3i8)') '### after', my_rank, i, BUF(i)
      enddo
!C===
```

隣接プロセッサから受け取った値をBUF(N+1), BUF(N+2)に代入する。



一次元モデル例題: 1d-srb1.f（10/10）

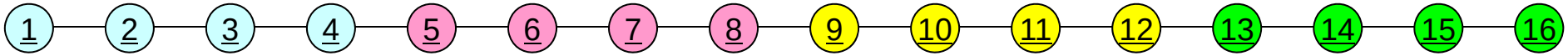
```
!C
!C +--------------+
!C | update array |
!C +--------------+
!C===
      do neib= 1, NEIBPETOT
        do k= import_index(neib-1)+1, import_index(neib)
          kk= import_item(k)
          BUF(kk)= RECVbuf(k)
        enddo
      enddo

      do i= iSTART, iEND
        write (*,'(a, 3i8)') '### after', my_rank, i, BUF(i)
      enddo
!C===

!C
!C +------------------+
!C | WAITall for SEND |
!C +------------------+
!C===
      call MPI_WAITALL (NEIBPETOT, request_send, stat_send, ierr)
!C===
```

```
%%% before   0   1   101
%%% before   0   2   102
%%% before   0   3   103
%%% before   0   4   104
%%% before   0   5   105
```

```
%%% before   1   1   105
%%% before   1   2   106
%%% before   1   3   107
%%% before   1   4   108
%%% before   1   5   104
%%% before   1   6   109
```

```
%%% before   2   1   109
%%% before   2   2   110
%%% before   2   3   111
%%% before   2   4   112
%%% before   2   5   108
%%% before   2   6   113
```

```
%%% before   3   1   113
%%% before   3   2   114
%%% before   3   3   115
%%% before   3   4   116
%%% before   3   5   112
```



一次元モデル例題: 1d-sr1.f（10/10）

```
!C
!C +--------------+
!C | update array |
!C +--------------+
!C===
      do neib= 1, NEIBPETOT
        do k= import_index(neib-1)+1, import_index(neib)
          kk= import_item(k)
          BUF(kk)= RECVbuf(k)
        enddo
      enddo

      do i= iSTART, iEND
        write (*,'(a, 3i8)') '### after', my_rank, i, BUF(i)
      enddo
!C===

!C
!C +------------------+
!C | WAITall for SEND |
!C +------------------+
!C===
      call MPI_WAITALL (NEIBPETOT, request_send, stat_send, ierr)
!C===
```

このメッセージのあと，SENDbufの中身を変更することが可能となる。

一次元モデル例題: 1d-srb1.f（10/10）

```
!C
!C +--------------+
!C | update array |
!C +--------------+
!C===
      do neib= 1, NEIBPETOT
        do k= import_index(neib-1)+1, import_index(neib)
          kk= import_item(k)
          BUF(kk)= RECVbuf(k)
        enddo
      enddo

      do i= iSTART, iEND
        write (*,'(a, 3i8)') '### after', my_rank, i, BUF(i)
      enddo!C===

!C
!C +------------------+
!C | WAITall for SEND |
!C +------------------+
!C===
      call MPI_WAITALL (NEIBPETOT, request_send, stat_send, ierr)
!C===
```

通信識別子（送信用）

状況オブジェクト配列（送信用）

MPI_ISEND/IRECV/WAITALLと MPI_SENDRECV

1d-srb1.f: Isend/Irecv/Waitall

```
      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
        call MPI_ISEND (SENDbuf(neib), len_s, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_send(neib), ierr)
      enddo
…
      do neib= 1, NEIBPETOT
        call MPI_IRECV (RECVbuf(neib), len_r, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_recv(neib), ierr)
      enddo
…
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)
      call MPI_WAITALL (NEIBPETOT, request_send, stat_send, ierr)
```

1d-srb2.f: Sendrecv

```
      allocate (stat_sr(MPI_STATUS_SIZE))
…
      do neib= 1, NEIBPETOT
        call MPI_SENDRECV                                           &
     &       (SENDbuf(neib), len_s, MPI_INTEGER, NEIBPE(neib), 0,   &
     &        RECVbuf(neib), len_r, MPI_INTEGER, NEIBPE(neib), 0,   &
     &        MPI_COMM_WORLD, stat_sr, ierr)
      enddo
```

MPI_ISEND/IRECV/WAITALLの使い方

1d-srb1.f: Isend/Irecv/Waitall

```
      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
        call MPI_ISEND (SENDbuf(neib), len_s, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_send(neib), ierr)
      enddo
…
      do neib= 1, NEIBPETOT
        call MPI_IRECV (RECVbuf(neib), len_r, MPI_INTEGER,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,            &
     &                  request_recv(neib), ierr)
      enddo
…
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)
      call MPI_WAITALL (NEIBPETOT, request_send, stat_send, ierr)
```

- 通信識別子 request を定義：送信用，受信用
- 状況オブジェクト配列 status を定義：送信用，受信用
- MPI_Isend，MPI_Irecvにそれぞれ対応したMPI_Waitallを呼ぶ
  - request，statusで区別

MPI_SENDRECVの使い方

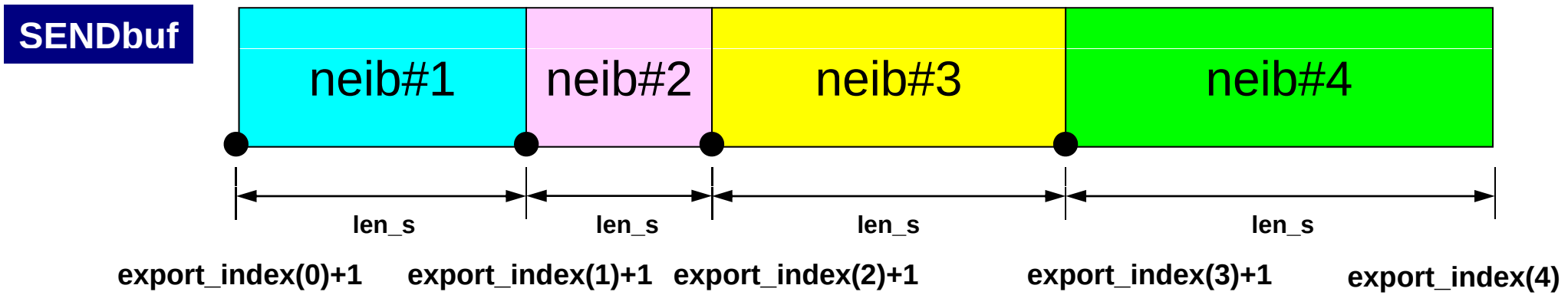
1d-srb2.f: Sendrecv

```
      allocate (stat_sr(MPI_STATUS_SIZE))
…
      do neib= 1, NEIBPETOT
        call MPI_SENDRECV                                              &
     &         (SENDbuf(neib), BUFlength, len_s, NEIBPE(neib), 0,      &
     &          RECVbuf(neib), BUFlength, len_r, NEIBPE(neib), 0,      &
     &          MPI_COMM_WORLD, stat_sr, ierr)
      enddo
```

- 状況オブジェクト配列 status
  - status(MPI_STATUS_SIZE) を定義するだけで良い
  - Isend/Irecv/Waitallのときとは statusのサイズが異なるので注意すること。

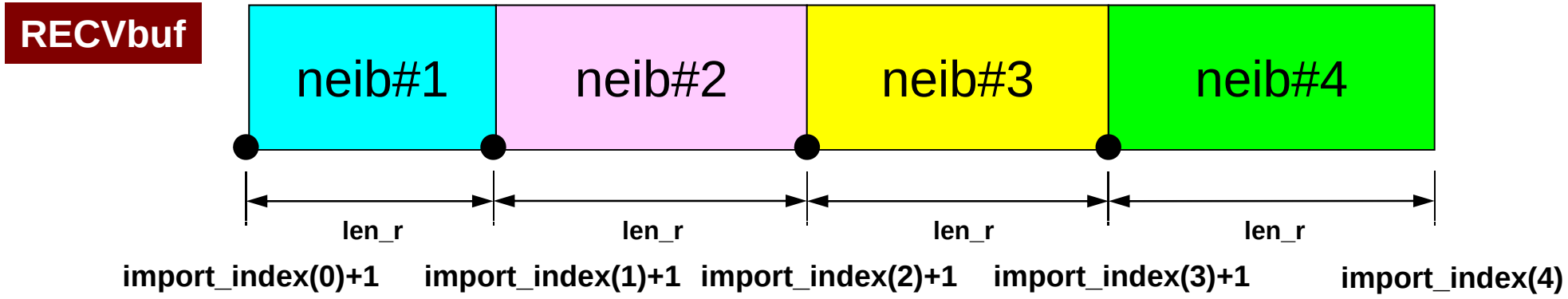
一般化された通信テーブル：送信

```
!C
!C-- PREPARE send  buffer
      do neib= 1, NEIBPETOT
        do k= export_index(neib-1)+1, export_index(neib)
          kk= export_item(k)
          SENDbuf(k)= BUF(kk)
        enddo
      enddo

!C
!C-- SEND
      do neib= 1, NEIBPETOT
        is   = export_index(neib-1) + 1
        len_s= export_index(neib) - export_index(neib-1)
        call MPI_ISEND (SENDbuf(is), len_s, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_send(neib), ierr)
      enddo
```



一般化された通信テーブル：受信

```
!C
!C-- RECV
      do neib= 1, NEIBPETOT
        ir   = import_index(neib-1) + 1
        len_r= import_index(neib) - import_index(neib-1)
        call MPI_IRECV (RECVbuf(ir), len_r, MPI_INTEGER,                &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_recv(neib), ierr)
      enddo

!C
!C-- WAITall for RECV
      call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

!C
!C-- update array
      do neib= 1, NEIBPETOT
        do k= import_index(neib-1)+1, import_index(neib)
          kk= import_item(k)
          BUF(kk)= RECVbuf(k)
        enddo
      enddo
```



- 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
  - MPI
  - 一次元問題
  - 二次元問題
- 1対1通信の実装例
- 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

課題S2：実施内容

- 提出期限：2011年10月21日（金）1700
- 「<$T-S2>/heat_cg.f」，「<$T-S2>/heat_cg.c」を参考にして，一次元熱伝導方程式をCG法により解くプログラムを並列化せよ。
  - 「一般化された通信テーブル（1d-srb1.f/c）」を使用せよ
- 実施課題
  - N=100およびN=103の場合について，1，2，4，8CPUを使用して計算してみよ。
    - 実は，N=100程度では並列化の効果は余り・・・全く得られない
  - Nを大きくして（例えば10^4，10^5），プロセッサ数を変更した場合について計算を実施してみよ。

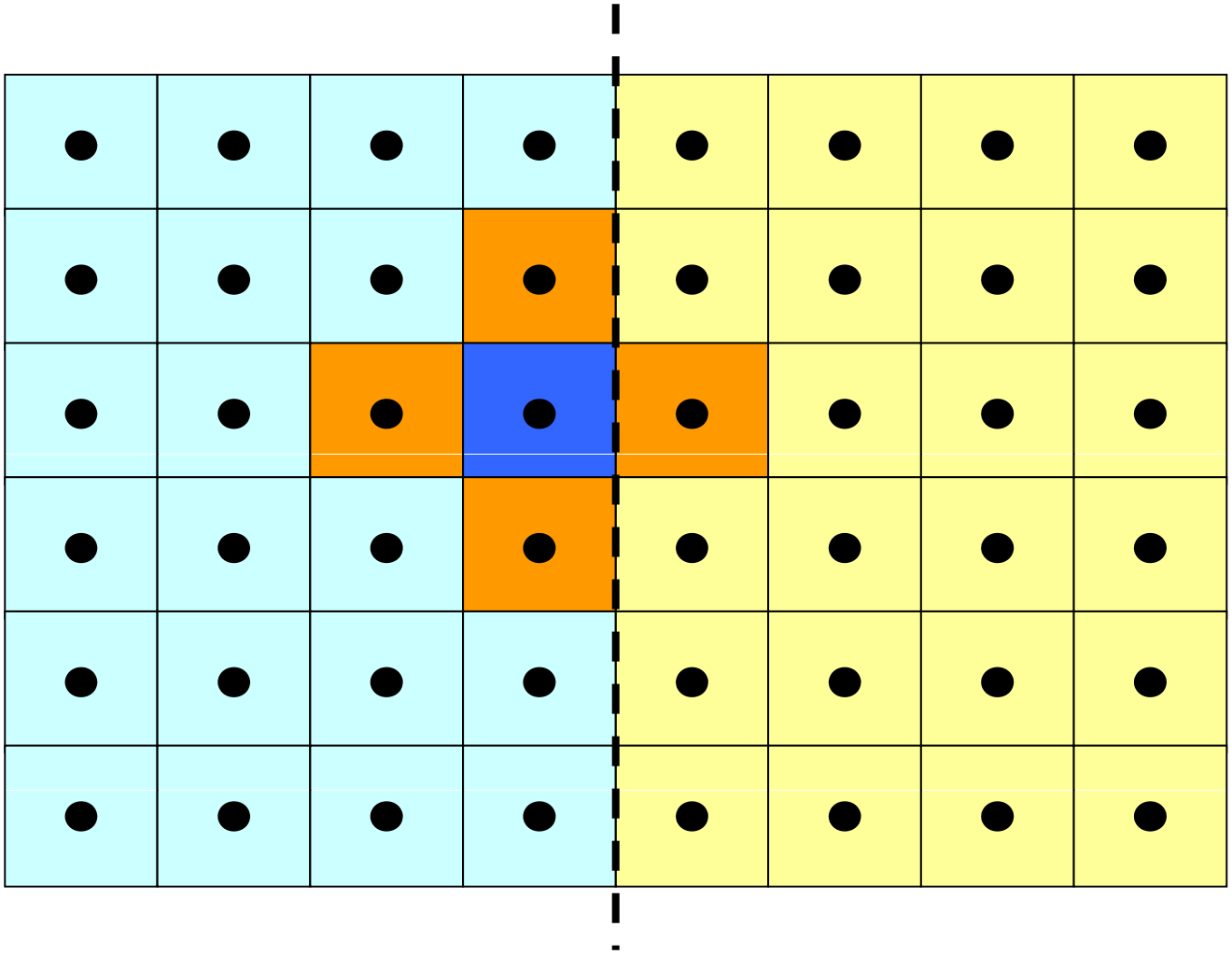
前処理付き共役勾配法
Preconditioned Conjugate Gradient Method（CG）

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

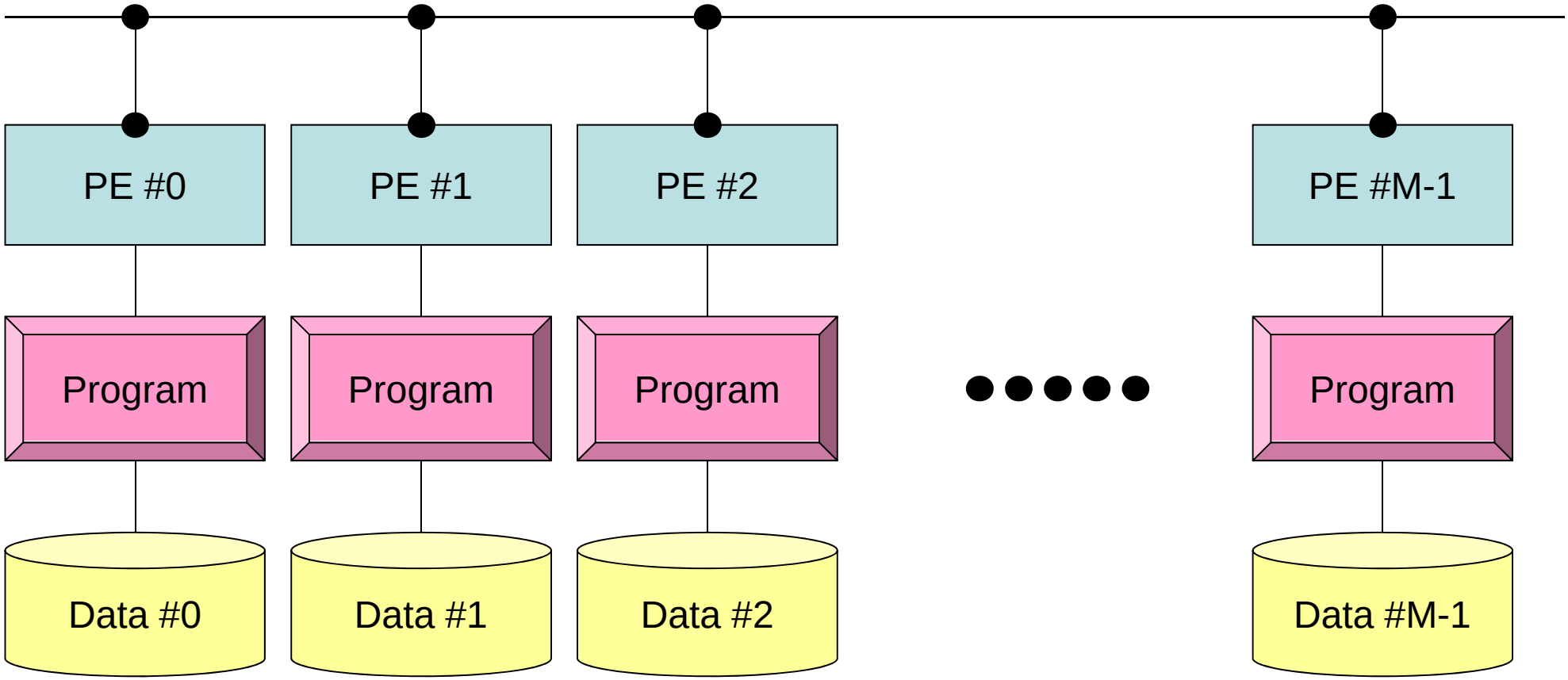
並列計算，領域間通信が必要な部分

- 行列ベクトル積
- 内積

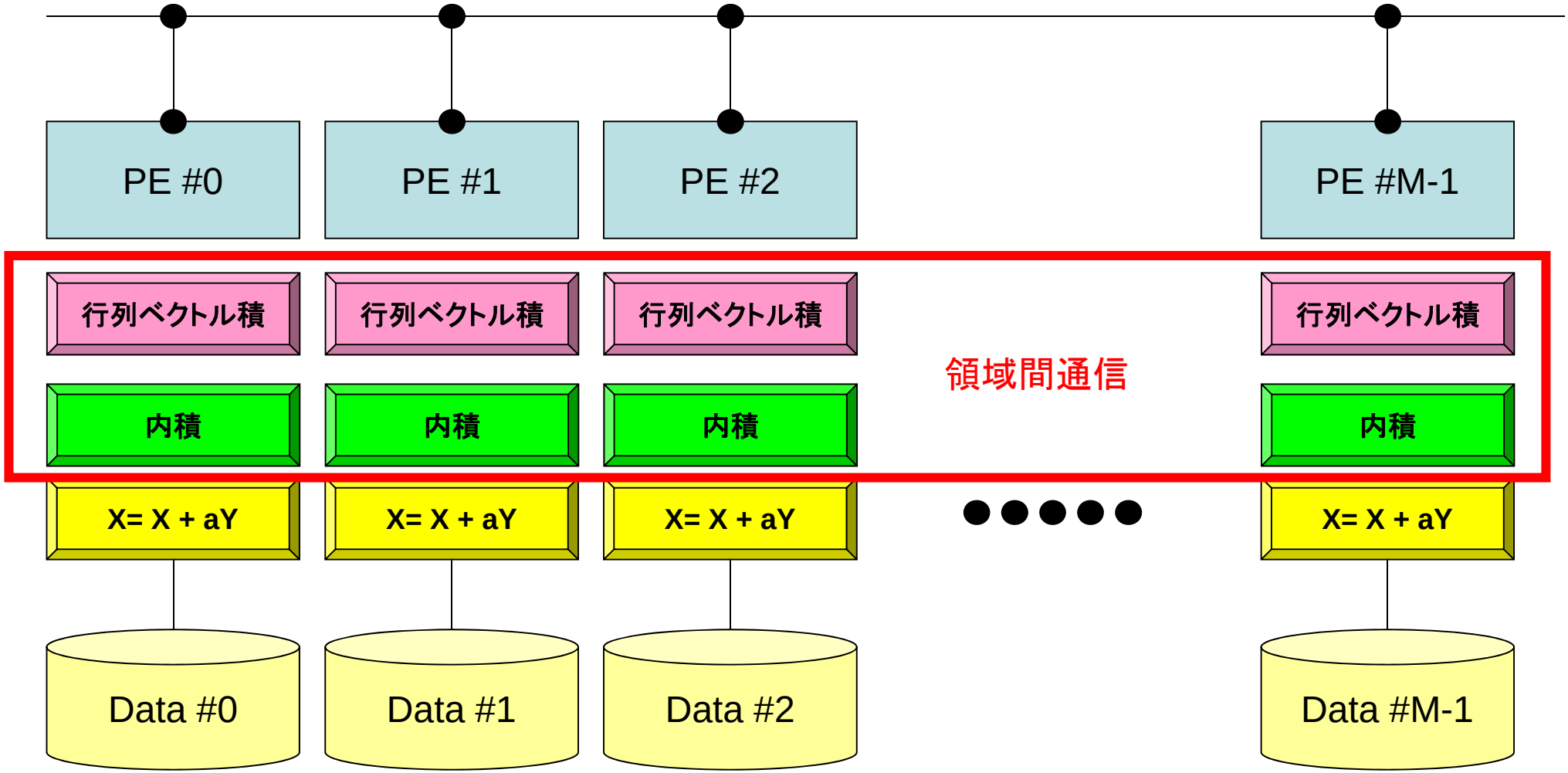
行列ベクトル積：他領域の値が必要
オーバーラップ＋通信テーブル



再びSPMD


PE #0
PE #1
PE #2
PE #M-1
Program
Program
Program
Program
Data #0
Data #1
Data #2
Data #M-1

CG法でははではこうなる



共役勾配法の「並列化」(1/2)

- 行列ベクトル積
  - 領域境界データの交換を実施して，外点のベクトル値の最新値を得ておく。

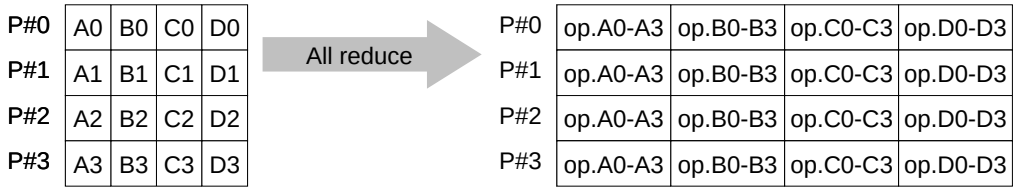
```
!C
!C-- {q}= [A]{p}

      exchange W(i,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
```

共役勾配法の「並列化」(2/2)

- 内積
  - MPI_ALLREDUCE

```
!C
!C +--------------+
!C | RHO= {r}{z} |
!C +--------------+
!C===
      RHO= 0.d0

      do i= 1, N
        RHO= RHO + W(i,R)*W(i,Z)
      enddo

      allreduce RHO

!C===
```

MPI_ALLREDUCE



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

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

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

```
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)
```