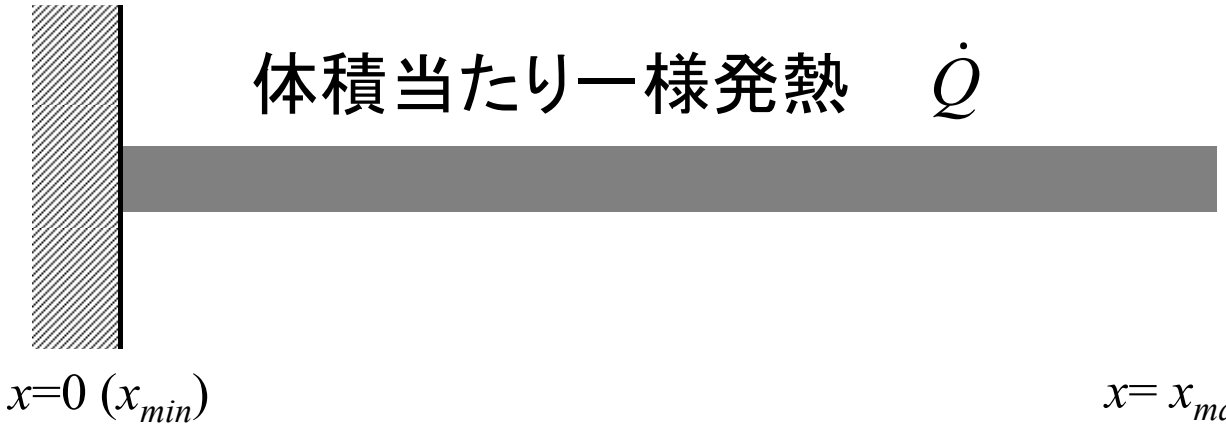
並列有限要素法による
一次元定常熱伝導解析プログラム
Fortran編

中島 研吾
東京大学情報基盤センター

- 問題の概要，実行方法
- 局所分散データの考え方
- プログラムの説明
- 計算例

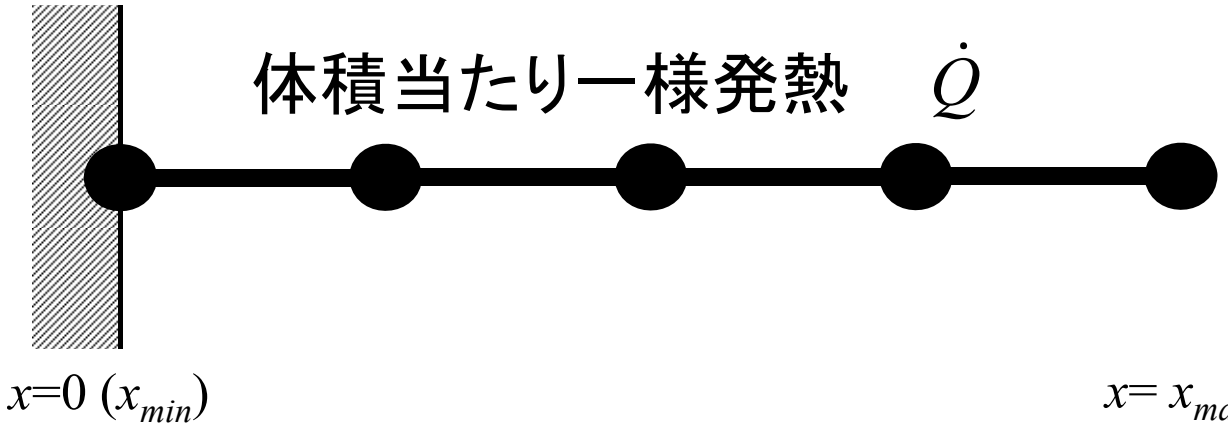
対象とする問題：一次元熱伝導問題



$$\frac{\partial}{\partial x}\left(\lambda\frac{\partial T}{\partial x}\right)+\dot{Q}=0$$

- 一様な：断面積A，熱伝導率λ
- 体積当たり一様発熱（時間当たり）〔$QL^{-3}T^{-1}$〕　$\dot{Q}$
- 境界条件
  - x=0　：T= 0　（固定）
  - x=x_{max} ：　$\dfrac{\partial T}{\partial x}=0$（断熱）

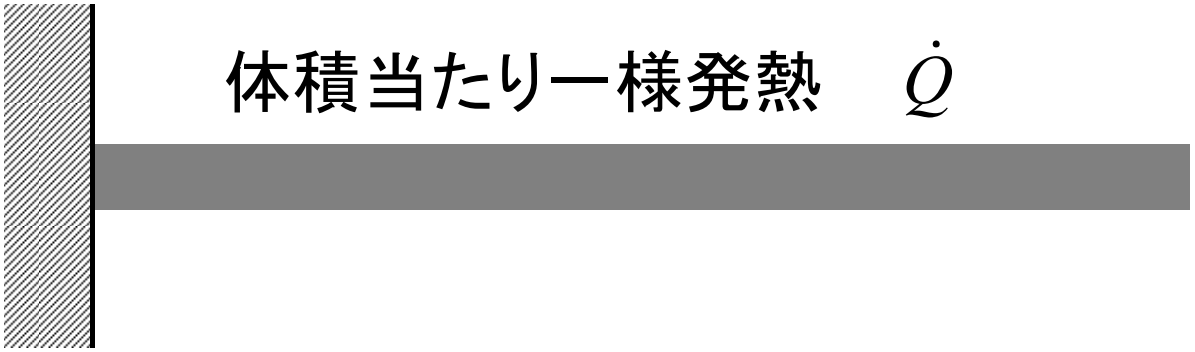
対象とする問題：一次元熱伝導問題



$$\frac{\partial}{\partial x}\left(\lambda\frac{\partial T}{\partial x}\right)+\dot{Q}=0$$

- 一様な：断面積A，熱伝導率λ
- 体積当たり一様発熱（時間当たり）〔$QL^{-3}T^{-1}$〕　$\dot{Q}$
- 境界条件
  - x=0　：T= 0　（固定）
  - x=x_{max} ：　$\dfrac{\partial T}{\partial x}=0$（断熱）

解析解



$$\frac{\partial}{\partial x}\left(\lambda\frac{\partial T}{\partial x}\right)+\dot{Q}=0$$

$$T=0 @ x=0$$

$$\frac{\partial T}{\partial x}=0 @ x=x_{\max}$$

$$\lambda T''=-\dot{Q}$$

$$\lambda T'=-\dot{Q}x+C_1 \Rightarrow C_1=\dot{Q}x_{\max}, \quad T'=0 @ x=x_{\max}$$

$$\lambda T=-\frac{1}{2}\dot{Q}x^2+C_1x+C_2 \Rightarrow C_2=0, \quad T=0 @ x=0$$

$$\therefore \quad T=-\frac{1}{2\lambda}\dot{Q}x^2+\frac{\dot{Q}x_{\max}}{\lambda}x$$

ファイルコピー，コンパイル（1/2）

ディレクトリ生成

```
>$ cd
>$ mkdir pFEM
>$ cd pFEM
```

FORTRANユーザー

```
>$ cd ~/pFEM
>$ cp /home/ss/aics60/2014Summer/F/1d.tar .
>$ tar xvf 1d.tar
```

Cユーザー

```
>$ cd ~/pFEM
>$ cp /home/ss/aics60/2014Summer/C/1d.tar .
>$ tar xvf 1d.tar
```

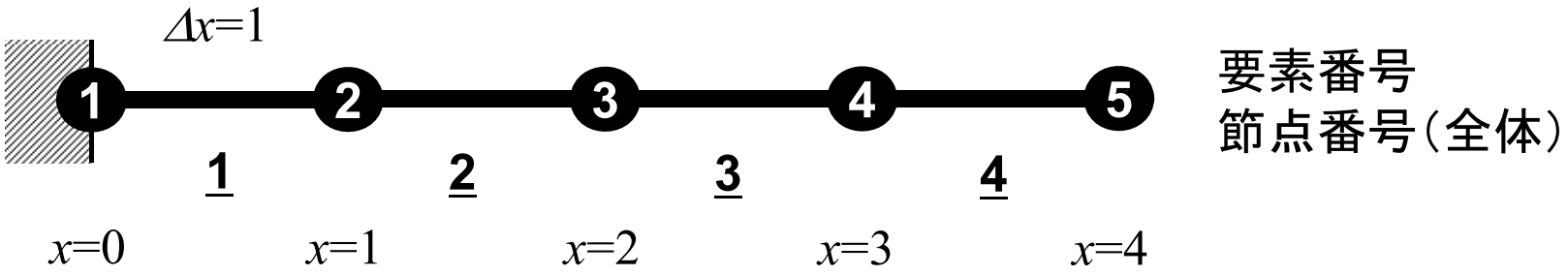
ファイルコピー，コンパイル（2/2）

ディレクトリ確認・コンパイル

```
>$ cd ~/pEFM/1d
>$ mpifrtpx -Kfast 1d.f
>$ mpifccpx -Kfast 1d.c
```

制御ファイル：input.dat

```
制御ファイル　input.dat

4                                   NE（要素数）
1.0   1.0   1.0   1.0   Δx（要素長さL），Q，A，λ
100                                 反復回数（CG法後述）
1.e-8                               CG法の反復打切誤差
```



ジョブスプリクト：go.sh

```
#!/bin/sh
#PJM -L "node=4"
#PJM -L "elapse=00:10:00"
#PJM -L "rscgrp=school"
#PJM -j
#PJM -o "test.lst"
#PJM --mpi "proc=64"

mpiexec ./a.out
```

8分割	16分割	32分割	64分割	192分割
“node=1“ “proc=8”	“node=1“ “proc=16”	“node=2“ “proc=32”	“node=4“ “proc=64”	“node=12“ “proc=192”

「並列計算」の手順

- 制御ファイル，「全要素数」を読み込む
- 内部で「局所分散メッシュデータ」を生成する
- マトリクス生成
- 共役勾配法によりマトリクスを解く

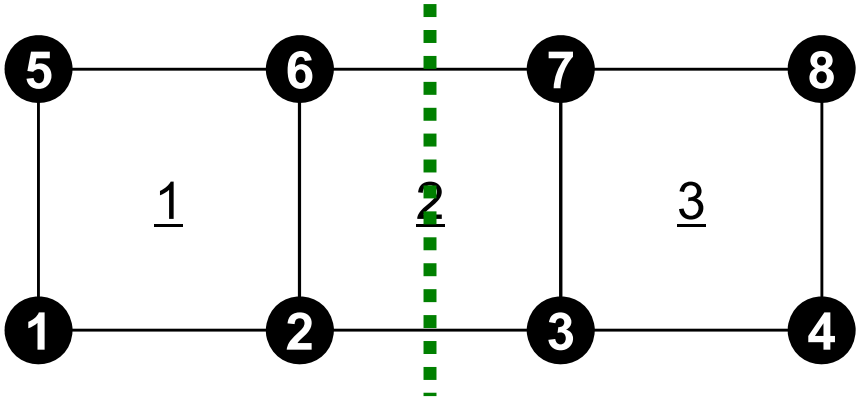
- 元のプログラムとほとんど変わらない

- 問題の概要，実行方法
- **局所分散データの考え方**
- プログラムの説明
- 計算例

有限要素法の処理：プログラム

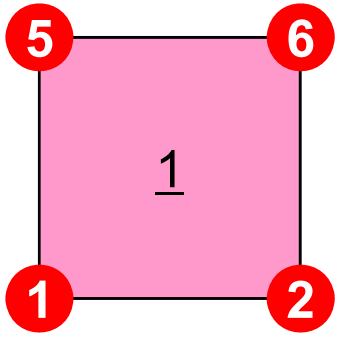
- 初期化
  - 制御変数読み込み
  - 座標読み込み⇒要素生成（N:節点数，NE：要素数）
  - 配列初期化（全体マトリクス，要素マトリクス）
  - 要素⇒全体マトリクスマッピング（Index，Item）
- マトリクス生成
  - 要素単位の処理（do icel= 1, NE）
    - 要素マトリクス計算
    - 全体マトリクスへの重ね合わせ
  - 境界条件の処理
- 連立一次方程式
  - 共役勾配法（CG）

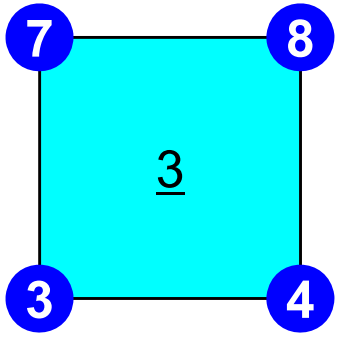
四角形要素



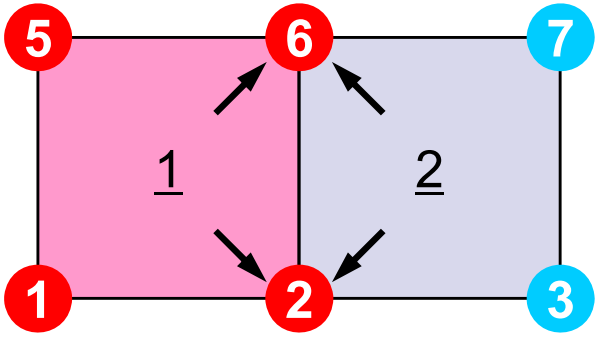
「節点ベース（領域ごとの節点数がバランスする）」の分割

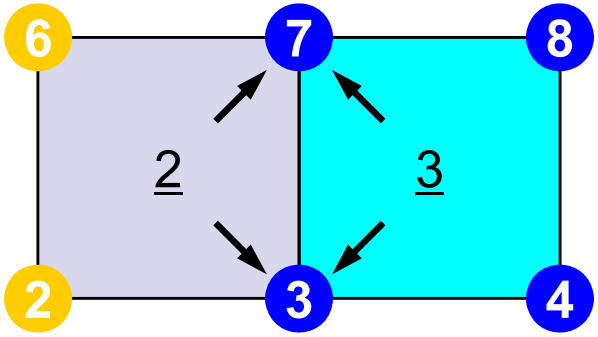
自由度：節点上で定義





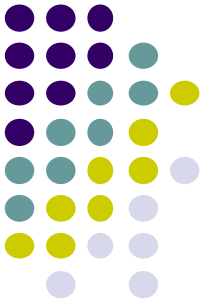
これではマトリクス生成に必要な情報は不十分





マトリクス生成のためには，オーバーラップ部分の要素と節点の情報が必要

並列有限要素法の局所データ構造

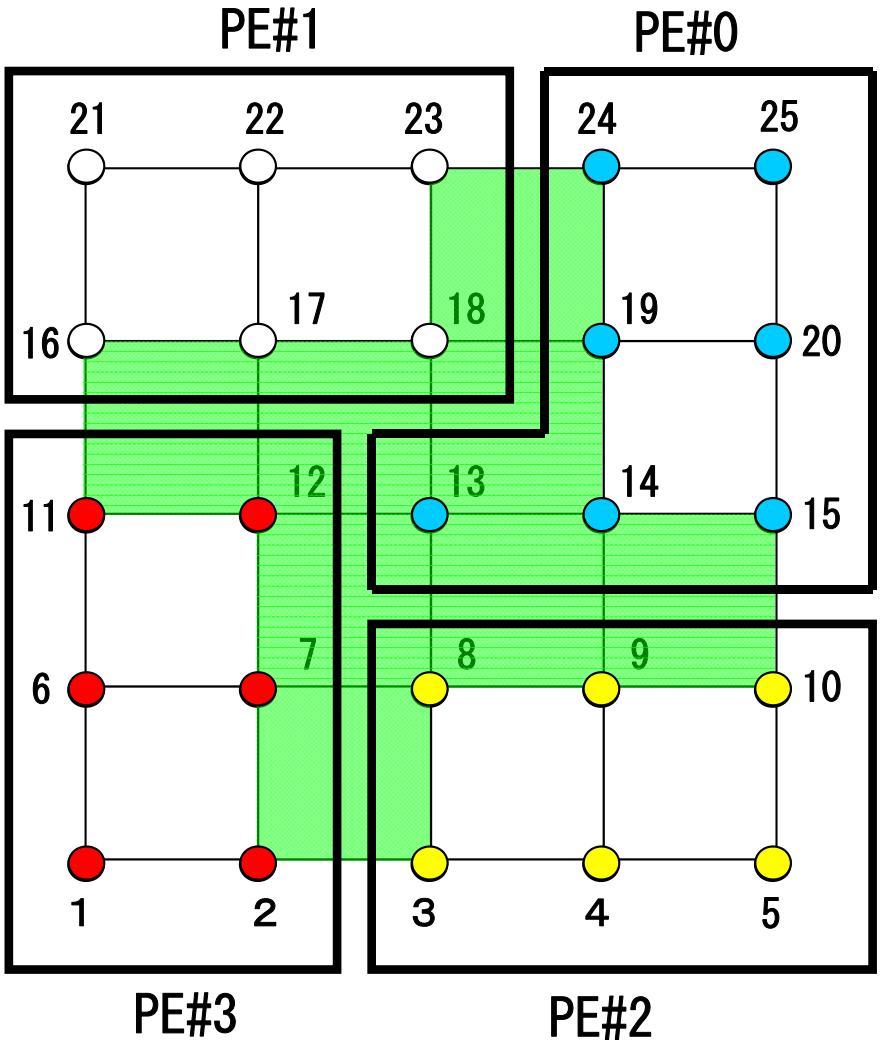


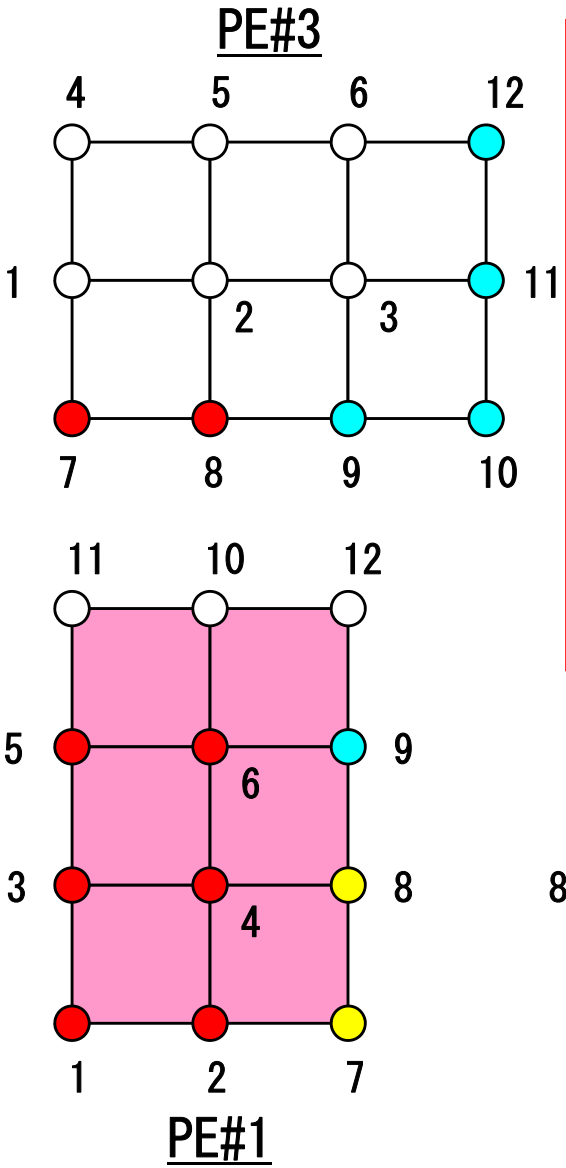
- 節点ベース：Node-based partitioning
- 局所データに含まれるもの：
  - その領域に本来含まれる節点
  - それらの節点を含む要素
  - 本来領域外であるが，それらの要素に含まれる節点
- 節点は以下の3種類に分類
  - 内点：Internal nodes　　その領域に本来含まれる節点
  - 外点：External nodes　　本来領域外であるがマトリクス生成に必要な節点
  - 境界点：Boundary nodes　　他の領域の「外点」となっている節点
- 領域間の通信テーブル

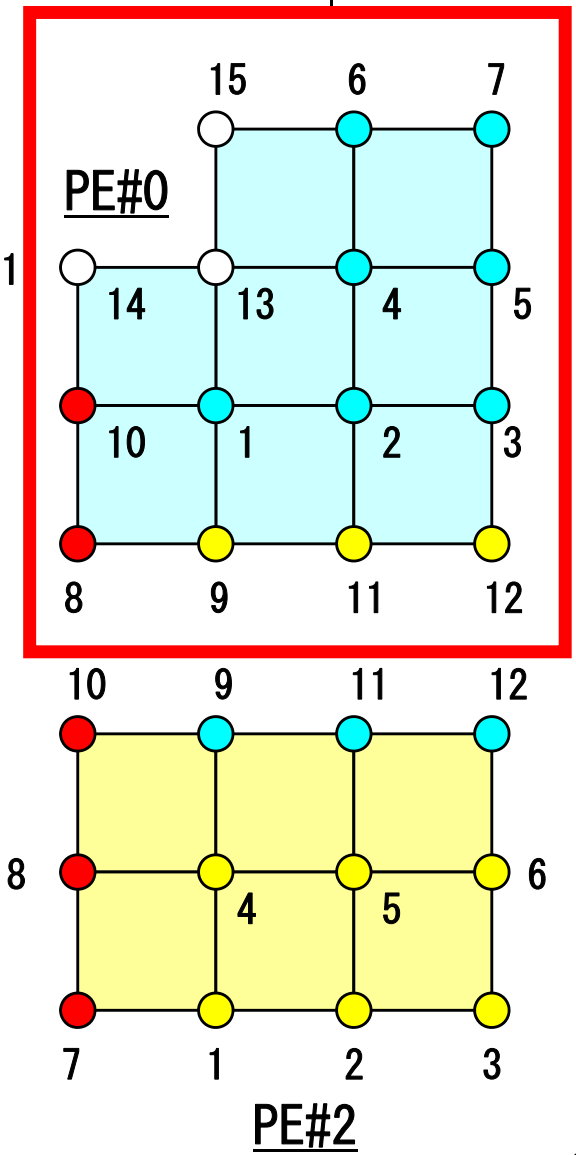
- 領域間の接続をのぞくと，大域的な情報は不要
  - 有限要素法の特性：要素で閉じた計算

Node-based Partitioning

internal nodes - elements - external nodes



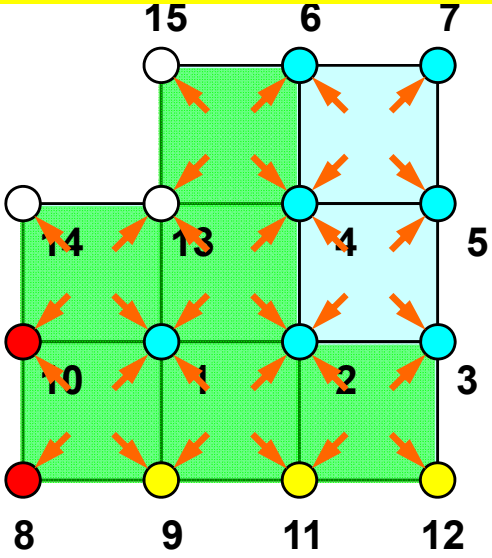


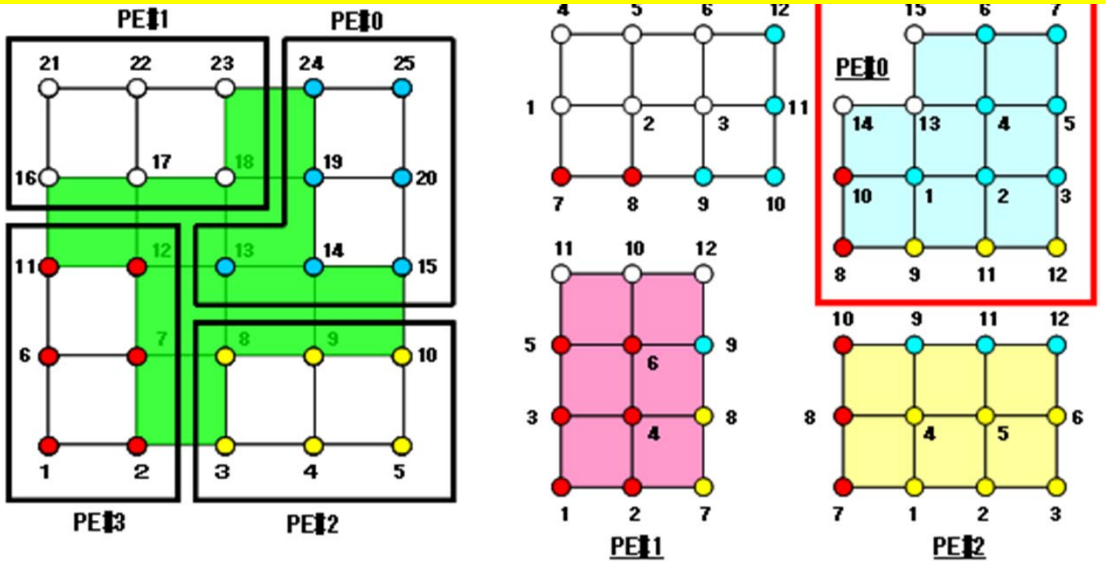


Node-based Partitioning

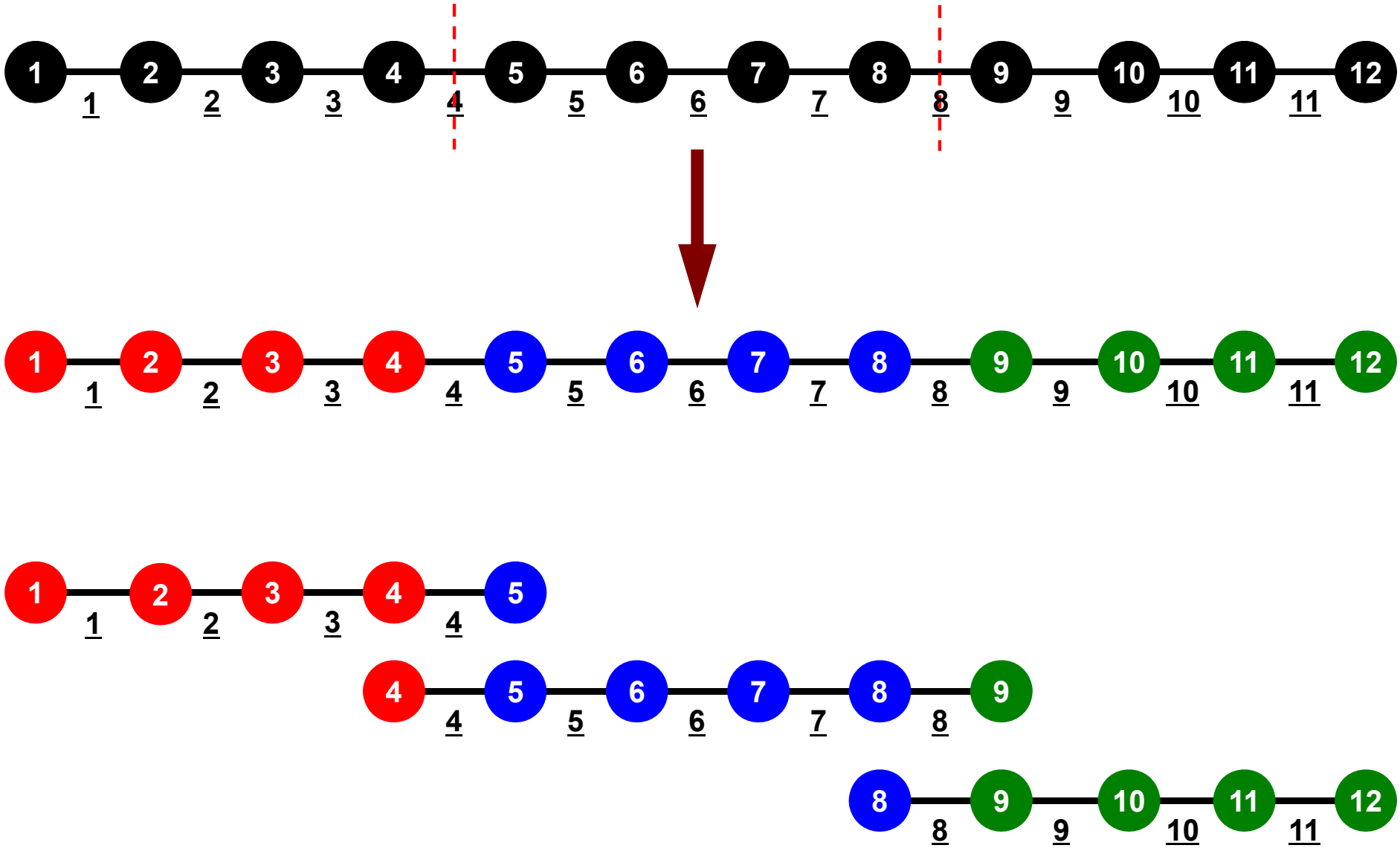
internal nodes - elements - external nodes

- Partitioned nodes themselves (Internal Nodes) 内点
- Elements which include Internal Nodes 内点を含む要素
- External Nodes included in the Elements 外点
  in overlapped region among partitions.
- Info of External Nodes are required for completely local element–based operations on each processor.



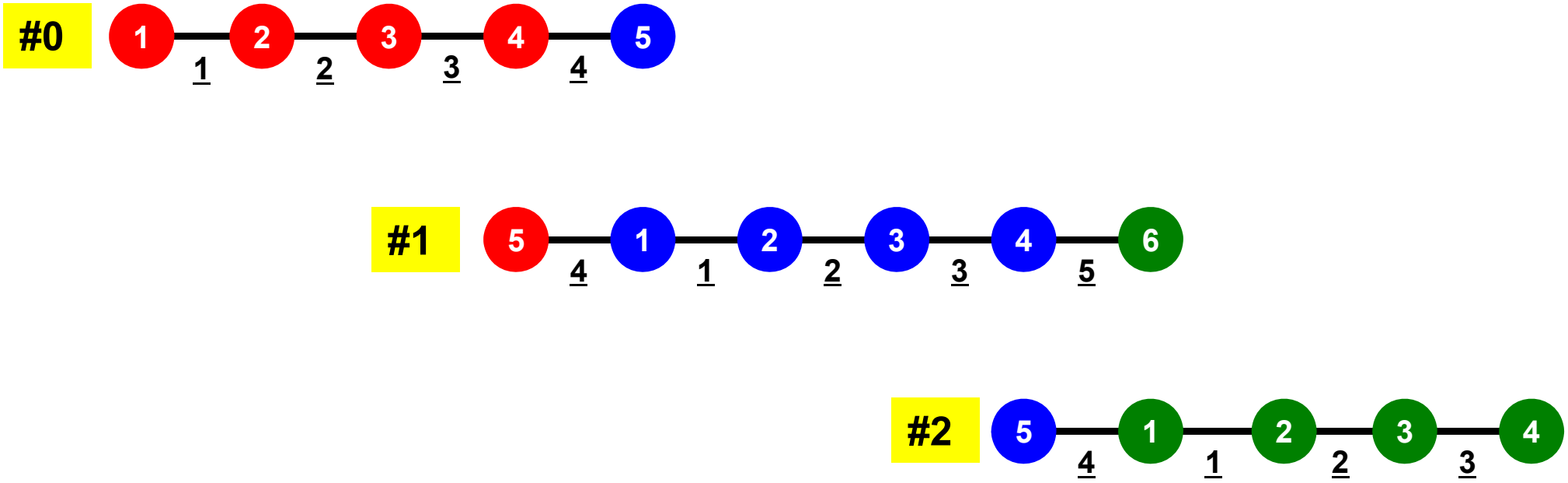


一次元問題:11要素，12節点，3領域



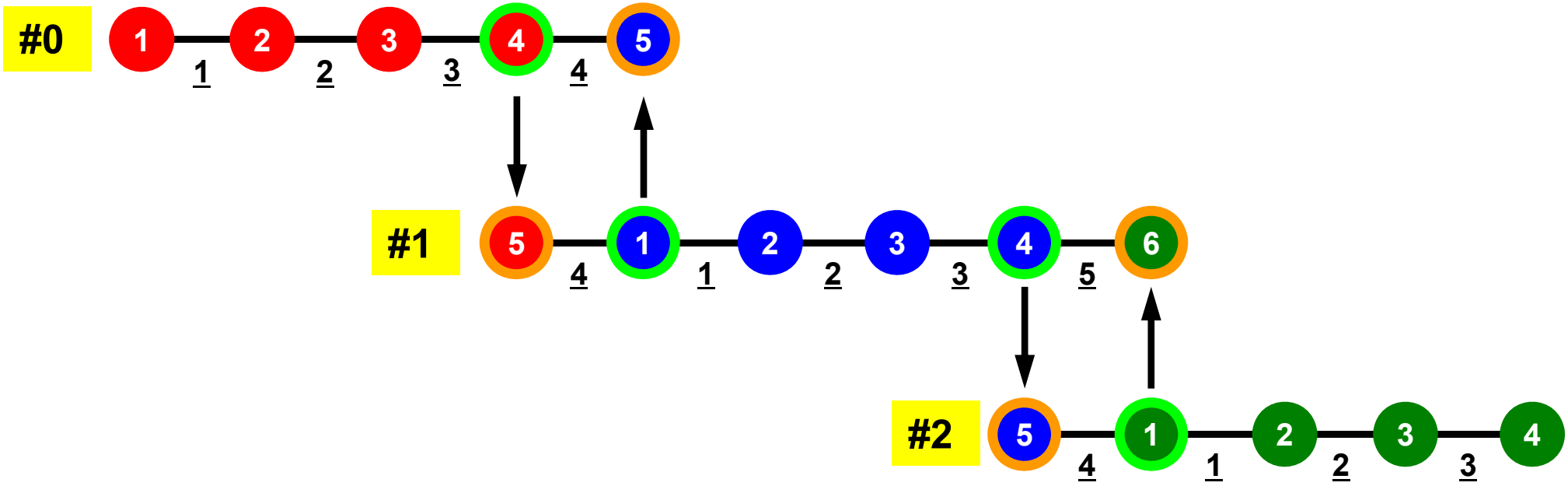
一次元問題:11要素，12節点，3領域

局所番号:節点・要素とも1からふる

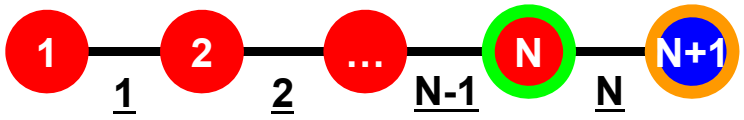


一次元問題:11要素，12節点，3領域

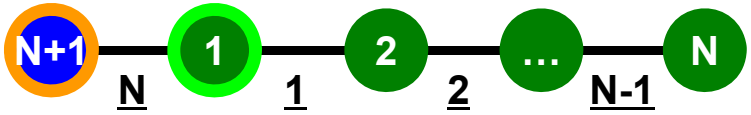
外点･境界点



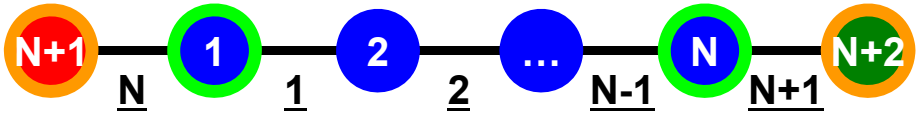
一次元問題：一般的な局所番号の付け方



#0: N+1節点，N要素



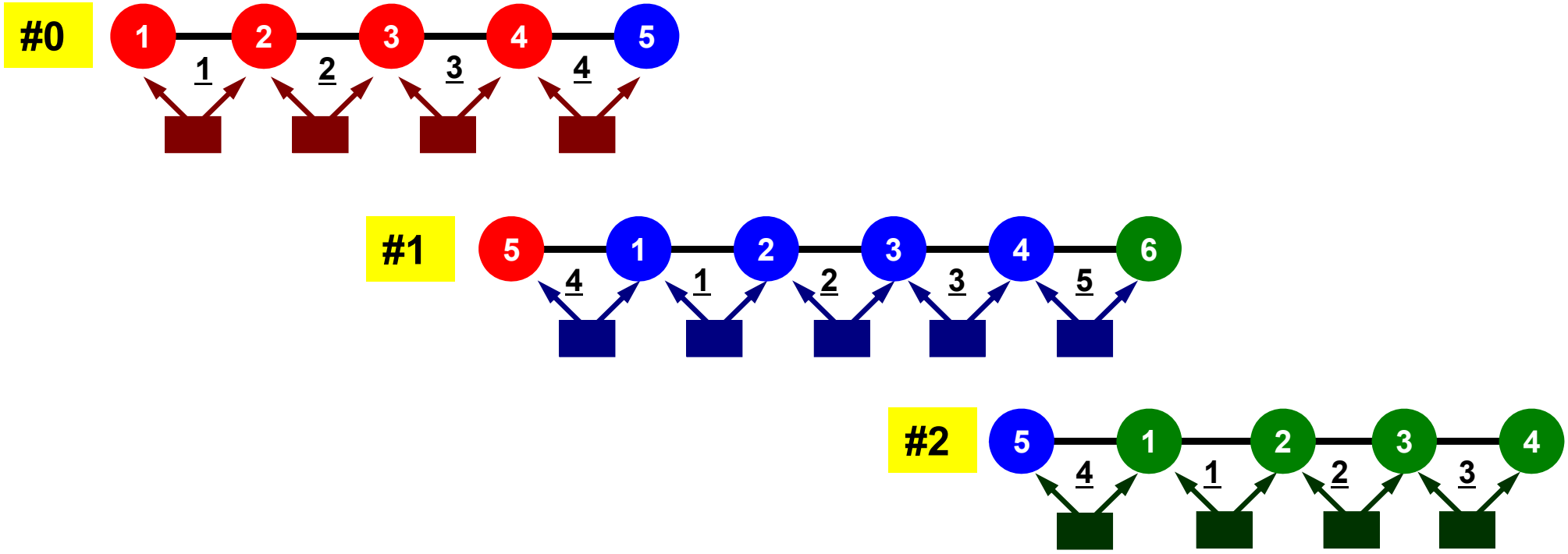
#PETot-1: N+1節点，N要素



一般の領域：
N+2節点，N+1要素

一次元問題:11要素，12節点，3領域

要素積分，要素マトリクス⇒全体マトリクス
内点，それを含む要素，外点で可能



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

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

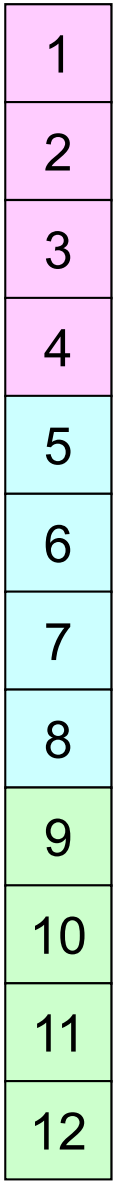
前処理：対角スケーリング

前処理，ベクトル定数倍の加減

局所的な計算（内点のみ）が可能⇒並列処理

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

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

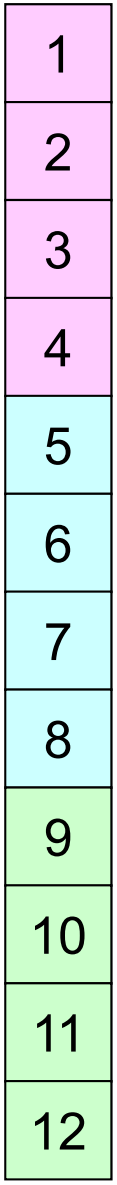
```
!C
!C-- {x}= {x} + ALPHA*{p}
!C   {r}= {r} - ALPHA*{q}

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



内積

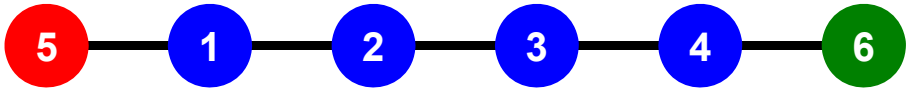
全体で和をとる必要がある⇒通信?

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

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



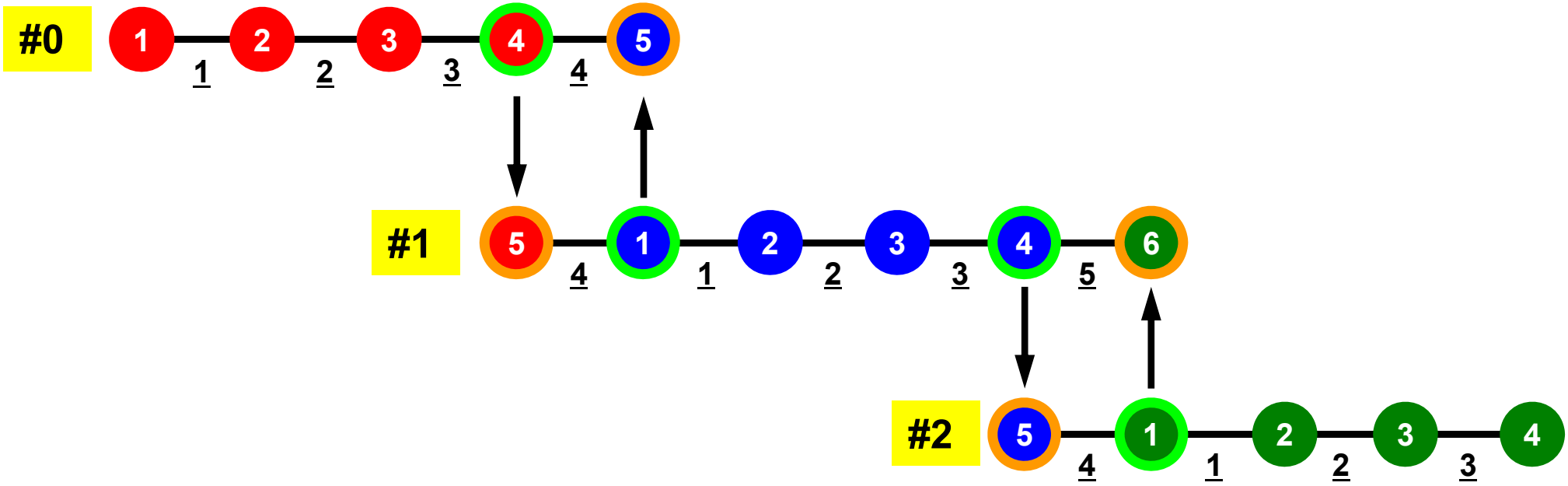
行列ベクトル積
外点の値(最新のp)が必要⇒1対1通信

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

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

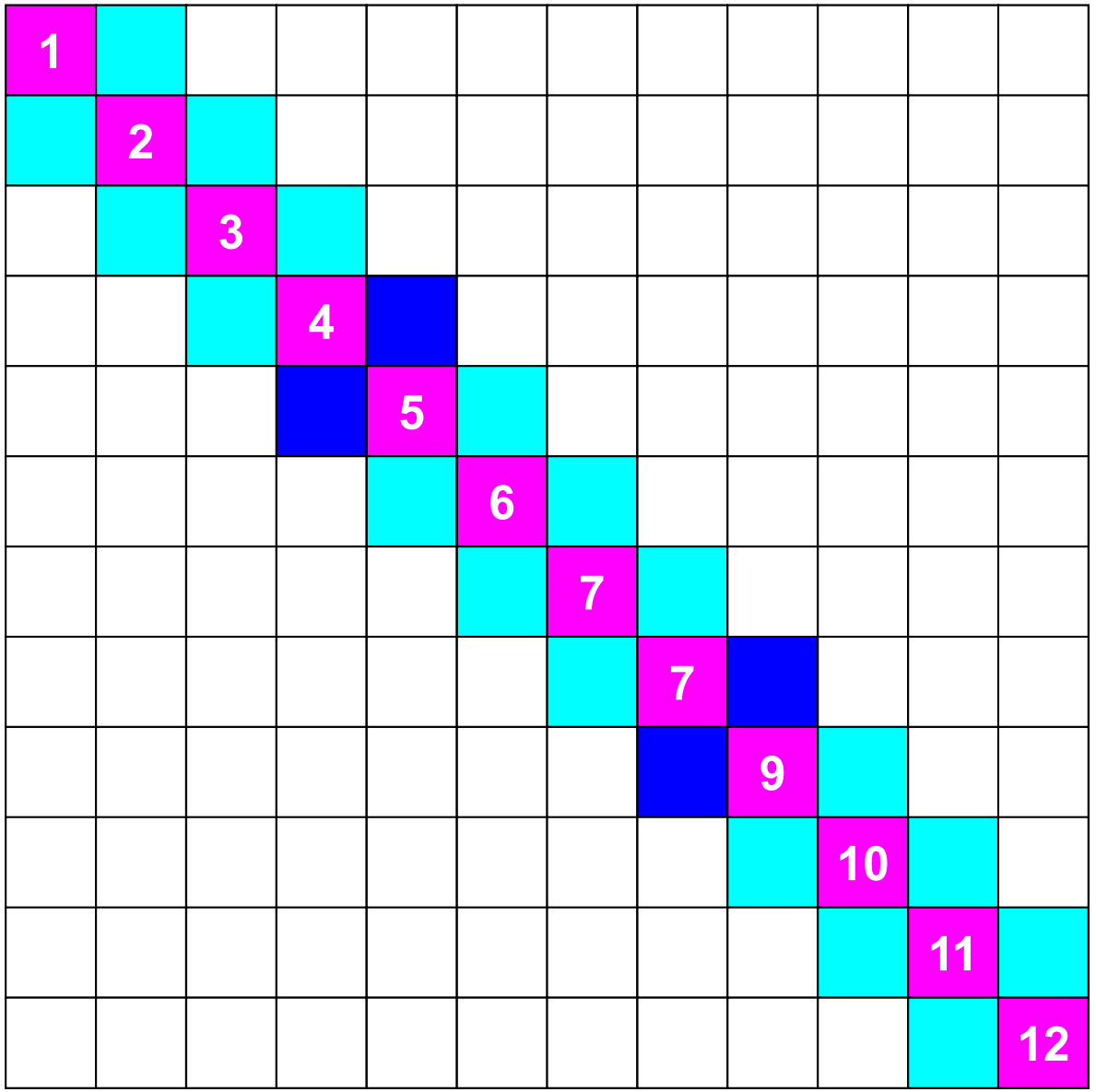


一次元問題:11要素，12節点，3領域

外点･境界点



行列ベクトル積：ローカルに計算実施可能

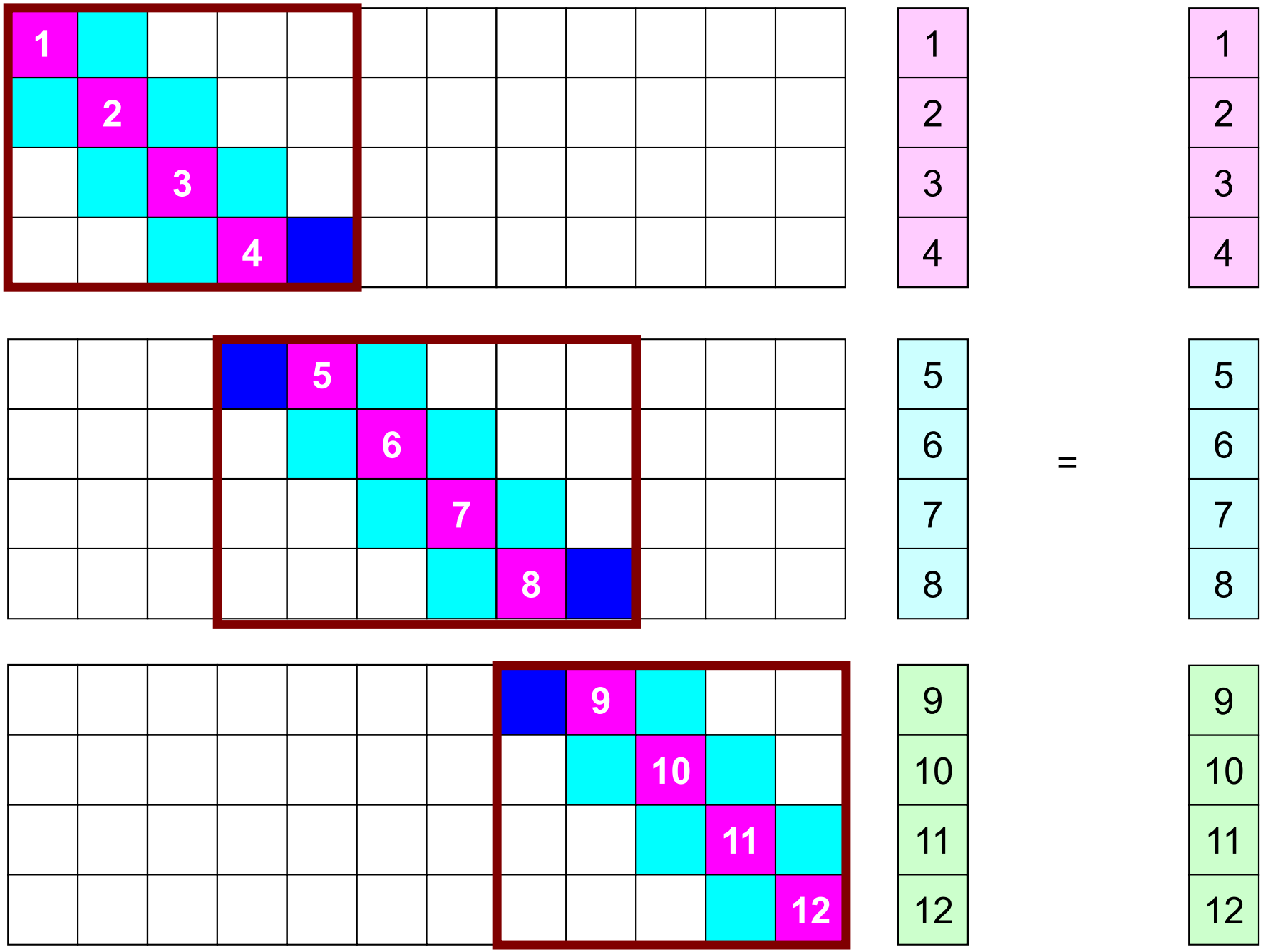


1
2
3
4
5
6
7
8
9
10
11
12

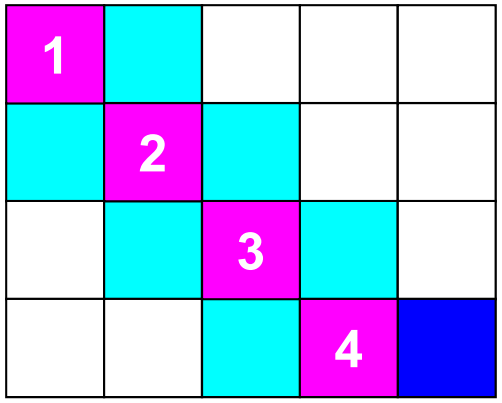
=

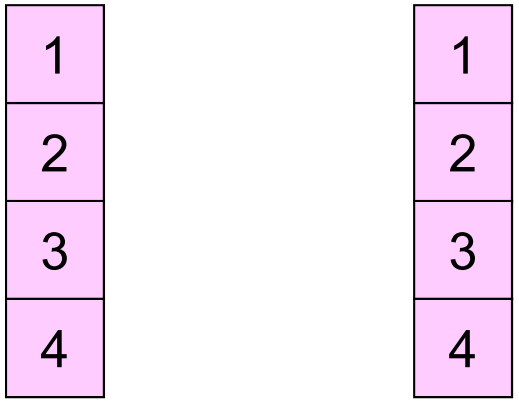
1
2
3
4
5
6
7
8
9
10
11
12

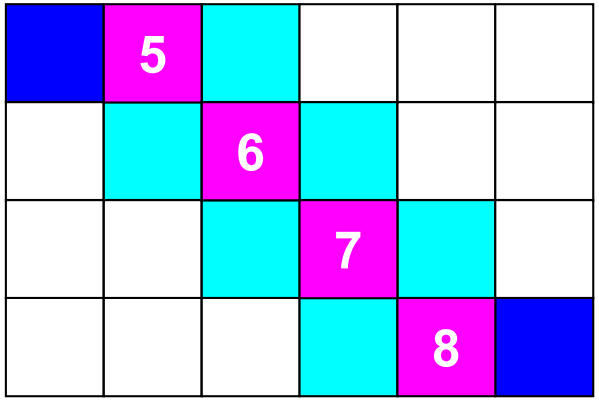
行列ベクトル積：ローカルに計算実施可能

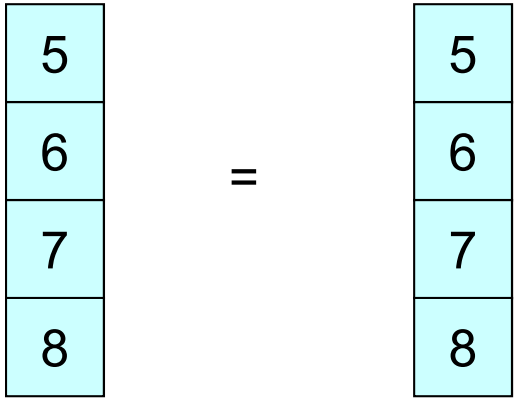


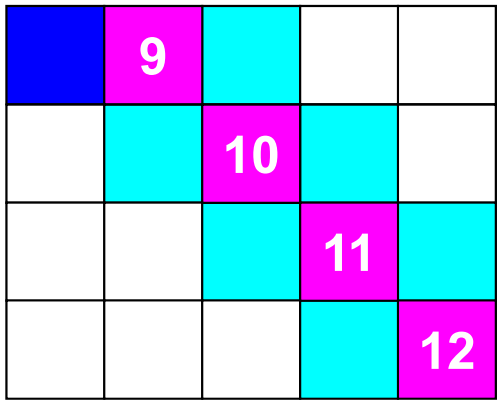
行列ベクトル積：ローカルに計算実施可能

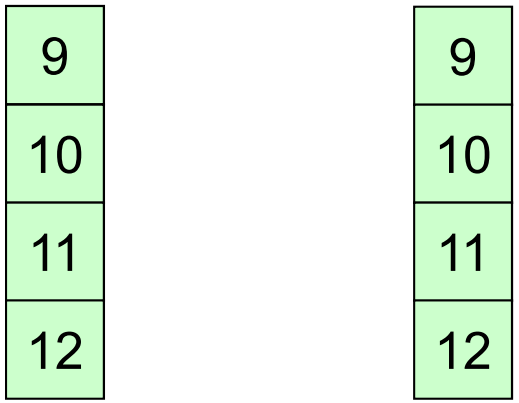




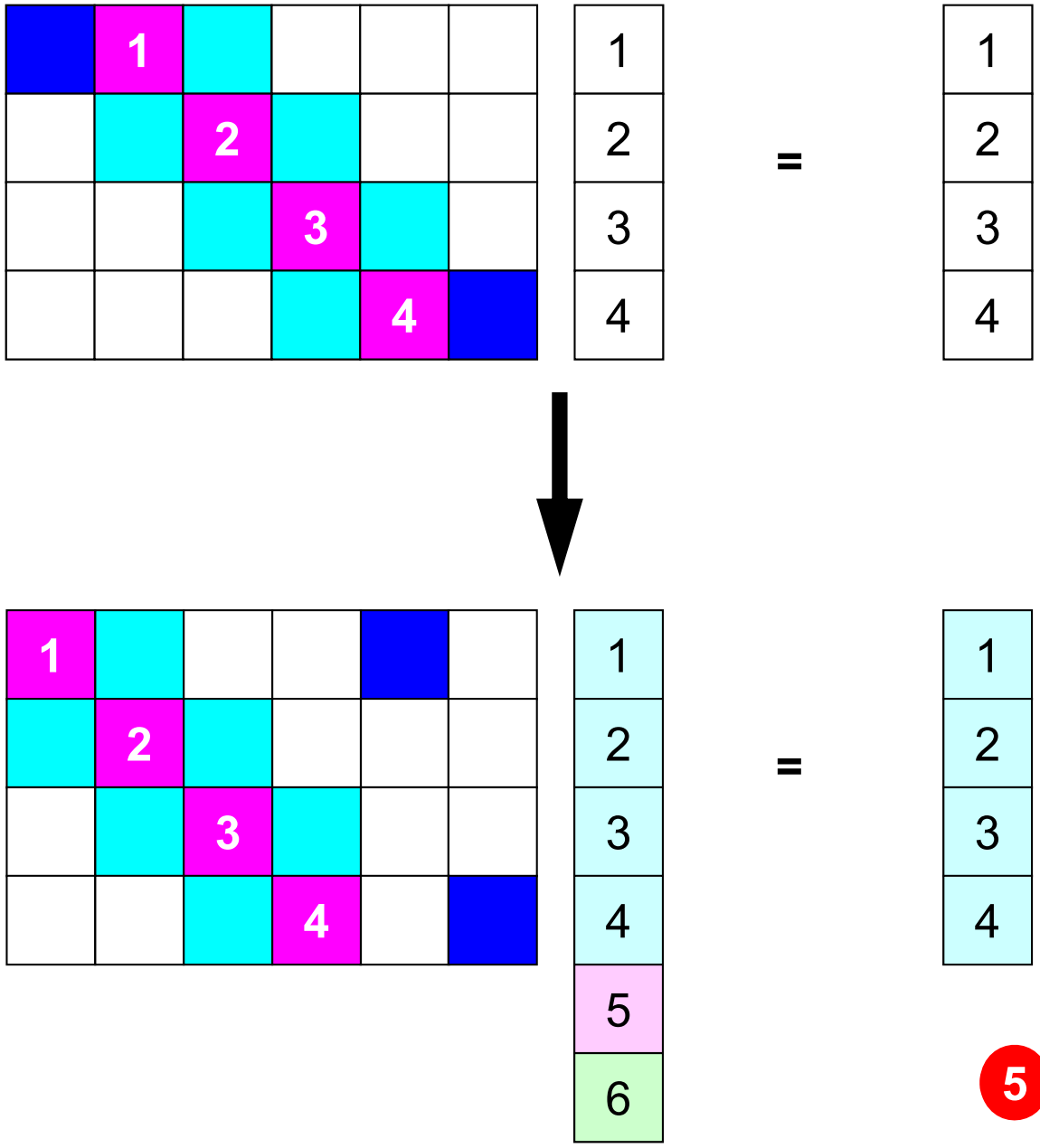




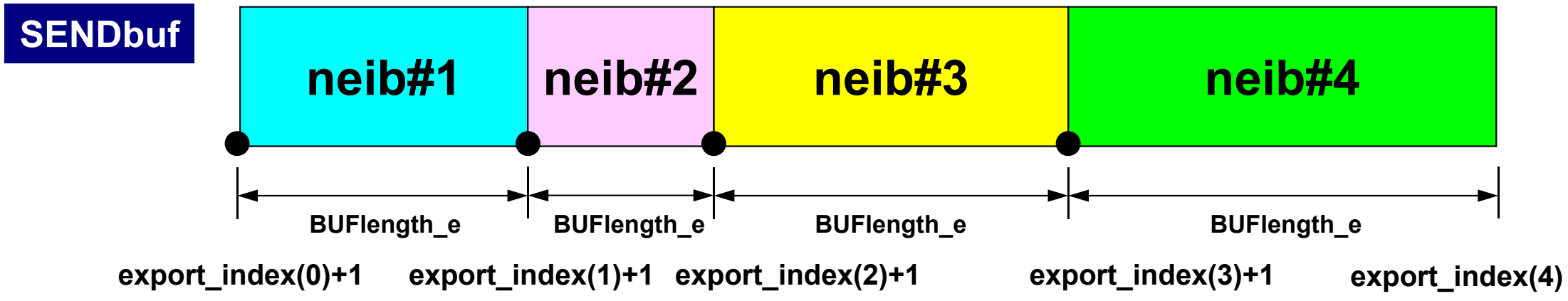




行列ベクトル積：ローカル計算 #1



送信（MPI_Isend/Irecv/Waitall）

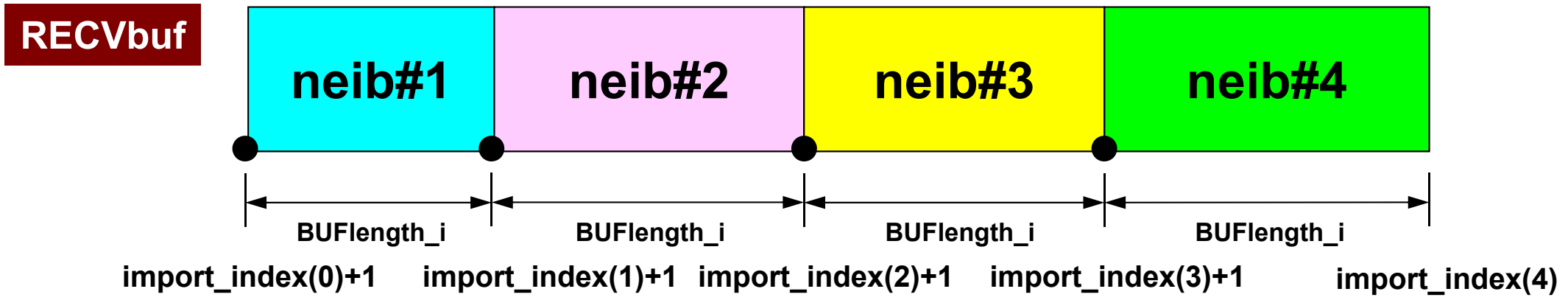



```
 do neib= 1, NEIBPETOT
   do k= export_index(neib-1)+1, export_index(neib)
     kk= export_item(k)
     SENDbuf(k)= VAL(kk)
   enddo
 enddo

 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。

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
```



- 問題の概要，実行方法
- 局所分散データの考え方
- **プログラムの説明**
- 計算例

プログラム:1d.f(1/11)

諸変数

```
      program heat1Dp
      implicit REAL*8 (A-H,O-Z)
      include 'mpif.h'

      integer :: N, NPLU, ITERmax
      integer :: R, Z, P, Q, DD

      real(kind=8) :: dX, RESID, EPS
      real(kind=8) :: AREA, QV, COND
      real(kind=8), dimension(:), allocatable :: PHI, RHS
      real(kind=8), dimension(:  ), allocatable :: DIAG, AMAT
      real(kind=8), dimension(:,:), allocatable :: W

      real(kind=8), dimension(2,2) :: KMAT, EMAT

      integer, dimension(:), allocatable :: ICELNOD
      integer, dimension(:), allocatable :: INDEX, ITEM
      integer(kind=4) :: NEIBPETOT, BUFlength, PETOT
      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

      real(kind=8), dimension(2) :: SENDbuf, RECVbuf

      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
```

プログラム：1d.f(2/11)
制御データ読み込み

```
!C
!C +-------+
!C | INIT. |
!C +-------+
!C===
!C
!C-- MPI init.

      call MPI_Init      (ierr)                                  MPI初期化：必須
      call MPI_Comm_size (MPI_COMM_WORLD, PETOT, ierr )          全プロセス数：PETOT
      call MPI_Comm_rank (MPI_COMM_WORLD, my_rank, ierr )        自分のランク番号（0～PETOT-1）：my_rank

!C
!C-- CTRL data
      if (my_rank.eq.0) then
        open  (11, file='input.dat', status='unknown')
         read (11,*) NEg
         read (11,*) dX, QV, AREA, COND
         read (11,*) ITERmax
         read (11,*) EPS
        close (11)
      endif

      call MPI_Bcast (NEg    , 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (ITERmax, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (dX     , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (QV     , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (AREA   , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (COND   , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (EPS    , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
```

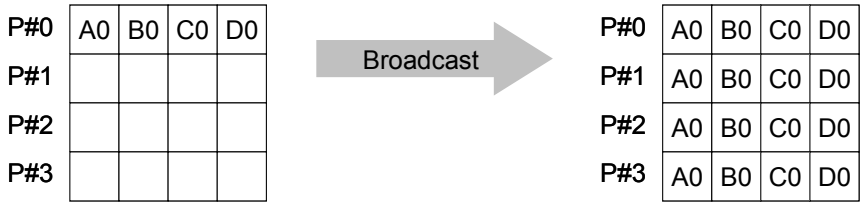
プログラム：1d.f(2/11)
制御データ読み込み

```
!C
!C +-------+
!C | INIT. |
!C +-------+
!C===
!C
!C-- MPI init.

      call MPI_Init      (ierr)                                  MPI初期化：必須
      call MPI_Comm_size (MPI_COMM_WORLD, PETOT, ierr )          全プロセス数：PETOT
      call MPI_Comm_rank (MPI_COMM_WORLD, my_rank, ierr )        自分のランク番号（0～PETOT-1）：my_rank

!C
!C-- CTRL data
      if (my_rank.eq.0) then                                     my_rank=0 のとき制御データを読み込む
        open  (11, file='input.dat', status='unknown')
         read (11,*) Neg                                         Neg: 「全」要素数
         read (11,*) dX, QV, AREA, COND
         read (11,*) ITERmax
         read (11,*) EPS
        close (11)
      endif

      call MPI_Bcast (NEg    , 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (ITERmax, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (dX     , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (QV     , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (AREA   , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (COND   , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (EPS    , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
```

プログラム：1d.f(2/11)
制御データ読み込み

```
!C
!C +-------+
!C | INIT. |
!C +-------+
!C===
!C
!C-- MPI init.

      call MPI_Init      (ierr)                                    MPI初期化：必須
      call MPI_Comm_size (MPI_COMM_WORLD, PETOT, ierr )            全プロセス数：PETOT
      call MPI_Comm_rank (MPI_COMM_WORLD, my_rank, ierr )          自分のランク番号（0～PETOT-1）：my_rank

!C
!C-- CTRL data
      if (my_rank.eq.0) then                                       my_rank=0 のとき制御データを読み込む
        open  (11, file='input.dat', status='unknown')
         read (11,*) Neg                                           Neg: 「全」要素数
         read (11,*) dX, QV, AREA, COND
         read (11,*) ITERmax
         read (11,*) EPS
        close (11)
      endif

      call MPI_Bcast (NEg    , 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr) 0番から各プロセスにデータ送信
      call MPI_Bcast (ITERmax, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (dX     , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (QV     , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (AREA   , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (COND   , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
      call MPI_Bcast (EPS    , 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
```

MPI_BCAST



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

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

    FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.

    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
  - **root** 整数 I 送信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

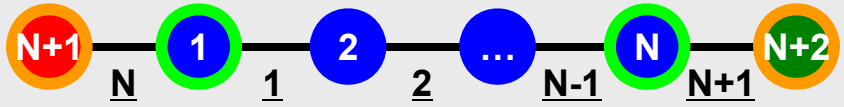
プログラム：1d.f(3/11)
局所分散メッシュデータ

```
!C
!C-- Local Mesh Size

      Ng= NEg + 1                          総節点数
      N = Ng / PETOT                       局所節点数

      nr = Ng - N*PETOT                    NgがPETOTで割り切れない場合
      if (my_rank.lt.nr) N= N+1

      NE= N - 1 + 2
      NP= N + 2

      if (my_rank.eq.0) NE= N - 1 + 1
      if (my_rank.eq.0) NP= N + 1

      if (my_rank.eq.PETOT-1) NE= N - 1 + 1
      if (my_rank.eq.PETOT-1) NP= N + 1

      if (PETOT.eq.1) NE= N-1
      if (PETOT.eq.1) NP= N

!C
!C- ARRAYs

      allocate (PHI(NP), DIAG(NP), AMAT(2*NP-2), RHS(NP))
      allocate (ICELNOD(2*NE))
      allocate (INDEX(0:NP), ITEM(2*NP-2), W(NP,4))
       PHI= 0.d0
      AMAT= 0.d0
      DIAG= 0.d0
       RHS= 0.d0
```

プログラム：1d.f(3/11)

局所分散メッシュデータ，各要素→一様

```
!C
!C-- Local Mesh Size

      Ng= NEg + 1                          総節点数
      N = Ng / PETOT                       局所節点数

      nr = Ng - N*PETOT                    NgがPETOTで割り切れない場合
      if (my_rank.lt.nr) N= N+1

      NE= N - 1 + 2                        NE：局所要素数
      NP= N + 2                            NP：局所節点数（内点＋外点）

      if (my_rank.eq.0) NE= N - 1 + 1
      if (my_rank.eq.0) NP= N + 1

      if (my_rank.eq.PETOT-1) NE= N - 1 + 1
      if (my_rank.eq.PETOT-1) NP= N + 1

      if (PETOT.eq.1) NE= N-1
      if (PETOT.eq.1) NP= N
```



一般の領域：
N+2節点，N+1要素

```
!C
!C- ARRAYs

      allocate (PHI(NP), DIAG(NP), AMAT(2*NP-2), RHS(NP))
      allocate (ICELNOD(2*NE))
      allocate (INDEX(0:NP), ITEM(2*NP-2), W(NP,4))
       PHI= 0.d0
      AMAT= 0.d0
      DIAG= 0.d0
       RHS= 0.d0
```

プログラム：1d.f(3/11)

局所分散メッシュデータ，各要素→一様

```
!C
!C-- Local Mesh Size

      Ng= NEg + 1                          総節点数
      N = Ng / PETOT                       局所節点数

      nr = Ng - N*PETOT                    NgがPETOTで割り切れない場合
      if (my_rank.lt.nr) N= N+1

      NE= N - 1 + 2                        NE：局所要素数
      NP= N + 2                            NP：局所節点数（内点＋外点）

      if (my_rank.eq.0) NE= N - 1 + 1
      if (my_rank.eq.0) NP= N + 1

      if (my_rank.eq.PETOT-1) NE= N - 1 + 1
      if (my_rank.eq.PETOT-1) NP= N + 1

      if (PETOT.eq.1) NE= N-1
      if (PETOT.eq.1) NP= N

!C
!C- ARRAYs

      allocate (PHI(NP), DIAG(NP), AMAT(2*NP-2), RHS(NP))
      allocate (ICELNOD(2*NE))
      allocate (INDEX(0:NP), ITEM(2*NP-2), W(NP,4))
       PHI= 0.d0
      AMAT= 0.d0
      DIAG= 0.d0
       RHS= 0.d0
```

#0: N+1節点，N要素

プログラム：1d.f(3/11)

局所分散メッシュデータ，各要素→一様

```
!C
!C-- Local Mesh Size

      Ng= NEg + 1                          総節点数
      N = Ng / PETOT                       局所節点数

      nr = Ng - N*PETOT                    NgがPETOTで割り切れない場合
      if (my_rank.lt.nr) N= N+1

      NE= N - 1 + 2                        NE：局所要素数
      NP= N + 2                            NP：局所節点数（内点＋外点）

      if (my_rank.eq.0) NE= N - 1 + 1
      if (my_rank.eq.0) NP= N + 1

      if (my_rank.eq.PETOT-1) NE= N - 1 + 1
      if (my_rank.eq.PETOT-1) NP= N + 1

      if (PETOT.eq.1) NE= N-1
      if (PETOT.eq.1) NP= N
```

#PETot-1: N+1節点，N要素

```
!C
!C- ARRAYs

      allocate (PHI(NP), DIAG(NP), AMAT(2*NP-2), RHS(NP))
      allocate (ICELNOD(2*NE))
      allocate (INDEX(0:NP), ITEM(2*NP-2), W(NP,4))
       PHI= 0.d0
      AMAT= 0.d0
      DIAG= 0.d0
       RHS= 0.d0
```

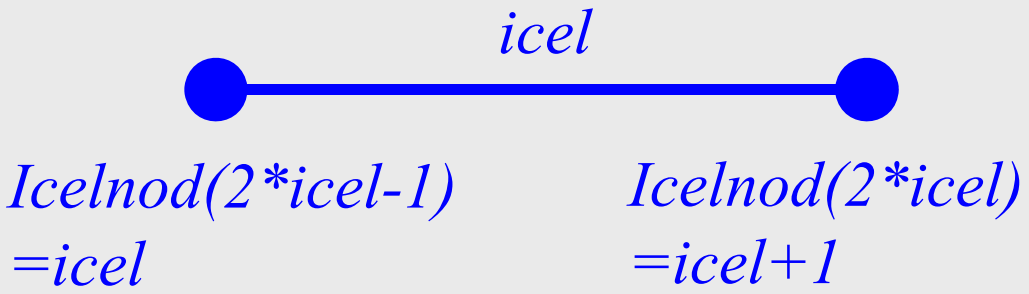
プログラム：1d.f(3/11)
局所分散メッシュデータ

```
!C
!C-- Local Mesh Size

      Ng= NEg + 1                          総節点数
      N = Ng / PETOT                       局所節点数

      nr = Ng - N*PETOT                    NgがPETOTで割り切れない場合
      if (my_rank.lt.nr) N= N+1

      NE= N - 1 + 2                        NE：局所要素数
      NP= N + 2                            NP：局所節点数（内点＋外点）

      if (my_rank.eq.0) NE= N - 1 + 1
      if (my_rank.eq.0) NP= N + 1

      if (my_rank.eq.PETOT-1) NE= N - 1 + 1
      if (my_rank.eq.PETOT-1) NP= N + 1

      if (PETOT.eq.1) NE= N-1
      if (PETOT.eq.1) NP= N

!C
!C- ARRAYs

      allocate (PHI(NP), DIAG(NP), AMAT(2*NP-2), RHS(NP))     NでなくNPで配列を定義している点に注意
      allocate (ICELNOD(2*NE))
      allocate (INDEX(0:NP), ITEM(2*NP-2), W(NP,4))
       PHI= 0.d0
      AMAT= 0.d0
      DIAG= 0.d0
       RHS= 0.d0
```

プログラム：1d.f(4/11)

配列初期化，要素～節点

```
do icel= 1, NE
  ICELNOD(2*icel-1)= icel
  ICELNOD(2*icel  )= icel + 1
enddo

if (PETOT.gt.1) then

if (my_rank.eq.0) then
  icel= NE
  ICELNOD(2*icel-1)= N
  ICELNOD(2*icel  )= N + 1


 else if (my_rank.eq.PETOT-1) then
  icel= NE
  ICELNOD(2*icel-1)= N + 1
  ICELNOD(2*icel  )= 1


 else
  icel= NE - 1
  ICELNOD(2*icel-1)= N + 1
  ICELNOD(2*icel  )= 1
  icel= NE
  ICELNOD(2*icel-1)= N
  ICELNOD(2*icel  )= N + 2

endif
endif
```

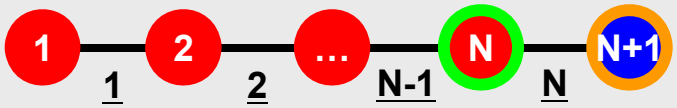


プログラム：1d.f(4/11)

配列初期化，要素～節点

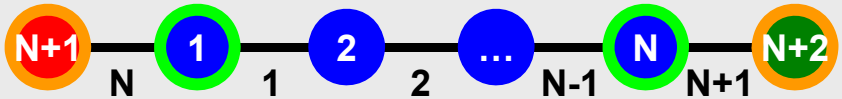
```
do icel= 1, NE
  ICELNOD(2*icel-1)= icel
  ICELNOD(2*icel  )= icel + 1
enddo

if (PETOT.gt.1) then

if (my_rank.eq.0) then
  icel= NE
  ICELNOD(2*icel-1)= N
  ICELNOD(2*icel  )= N + 1

 else if (my_rank.eq.PETOT-1) then
  icel= NE
  ICELNOD(2*icel-1)= N + 1
  ICELNOD(2*icel  )= 1

 else
  icel= NE - 1
  ICELNOD(2*icel-1)= N + 1
  ICELNOD(2*icel  )= 1
  icel= NE
  ICELNOD(2*icel-1)= N
  ICELNOD(2*icel  )= N + 2

endif
endif
```

「1-2」の要素を「1」とする



#0: N+1節点，N要素

#PETot-1: N+1節点，N要素



一般の領域：
N+2節点，N+1要素

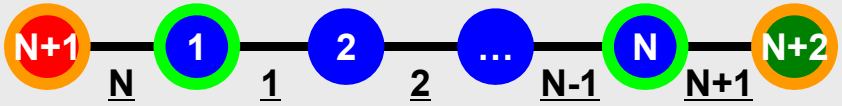
プログラム：1d.f(5/11)

Index定義

```
 KMAT(1,1)= +1.d0
      KMAT(1,2)= -1.d0
      KMAT(2,1)= -1.d0
      KMAT(2,2)= +1.d0
!C===

!C
!C +-------------+
!C | CONNECTIVITY |
!C +-------------+
!C===
      INDEX   = 2

      INDEX(0)= 0

      INDEX(N+1)= 1
      INDEX(NP )= 1

      if (my_rank.eq.0)        INDEX(1)= 1
      if (my_rank.eq.PETOT-1)  INDEX(N)= 1

      do i= 1, NP
        INDEX(i)= INDEX(i) + INDEX(i-1)
      enddo

      NPLU= INDEX(NP)
      ITEM= 0
```

#0: N+1節点，N要素

#PETot-1: N+1節点，N要素

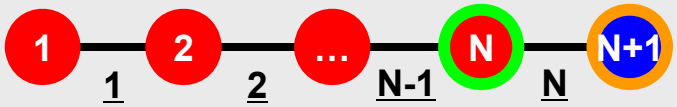


一般の領域：
N+2節点，N+1要素

プログラム：1d.f(6/11)

Item定義

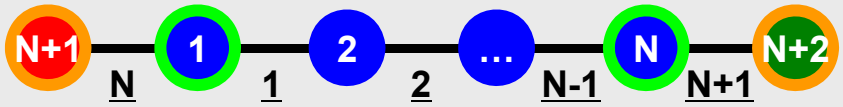
```
do i= 1, N
  jS= INDEX(i-1)
  if (my_rank.eq.0.and.i.eq.1) then
    ITEM(jS+1)= i+1
   else if (my_rank.eq.PETOT-1.and.i.eq.N) then
    ITEM(jS+1)= i-1
   else
    ITEM(jS+1)= i-1
    ITEM(jS+2)= i+1
    if (i.eq.1) ITEM(jS+1)= N + 1
    if (i.eq.N) ITEM(jS+2)= N + 2
    if (my_rank.eq.0.and.i.eq.N) ITEM(jS+2)= N + 1
  endif
enddo
```



#0: N+1節点，N要素

```
i = N + 1
jS= INDEX(i-1)
if (my_rank.eq.0) then
  ITEM(jS+1)= N
 else
  ITEM(jS+1)= 1
endif
```

#PETot-1: N+1節点，N要素



一般の領域：
N+2節点，N+1要素

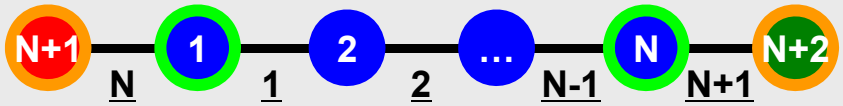
```
i = N + 2
if (my_rank.ne.0.and.my_rank.ne.PETOT-1) then
  jS= INDEX(i-1)
  ITEM(jS+1)= N
endif
```

プログラム：1d.f(7/11)
通信テーブル定義

```
!C
!C-- COMMUNICATION
      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

      BUFlength= 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
      endif
!C
!C-- INIT. arrays for MPI_Waitall
      allocate (stat_send(MPI_STATUS_SIZE,NEIBPETOT), stat_recv(MPI_STATUS_SIZE,NEIBPETOT))
      allocate (request_send(NEIBPETOT), request_recv(NEIBPETOT))
```

#0: N+1節点，N要素

#PETot-1: N+1節点，N要素



一般の領域：
N+2節点，N+1要素

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））
  - **ierr** 整数 O 完了コード

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））
  - **ierr** 整数 O 完了コード

MPI_WAITALL

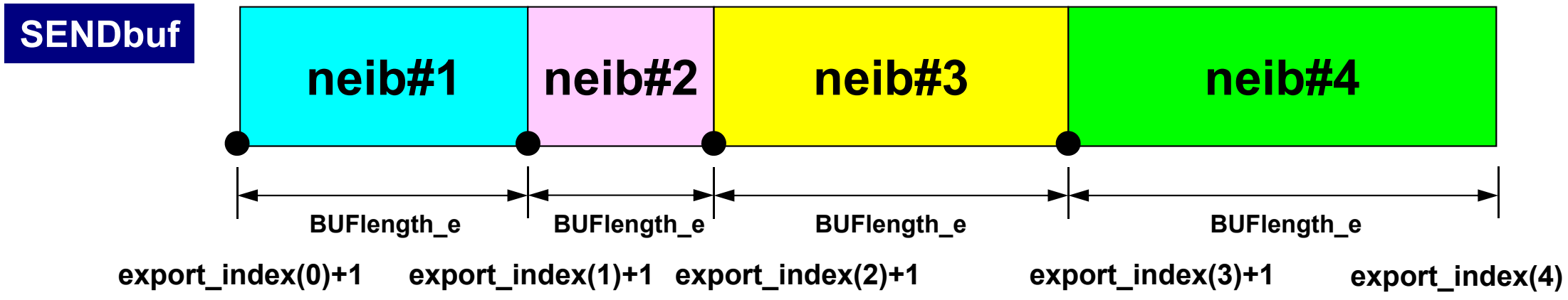
- 1対1非ブロッキング通信サブルーチンである「MPI_ISEND」と「MPI_IRECV」を使用した場合，プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_WAITALL」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_WAITALL」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば，「MPI_ISEND」と「MPI_IRECV」を同時に同期してもよい。
  - 「MPI_ISEND/IRECV」で同じ通信識別子を使用すること
- 「MPI_BARRIER」と同じような機能であるが，代用はできない。
  - 実装にもよるが，「request」，「status」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
  - **ierr** 整数 O 完了コード

一般化された通信テーブル：送信

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

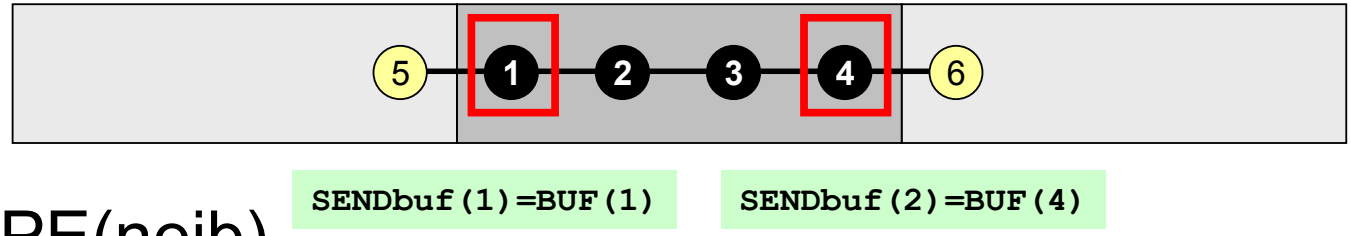
送信（MPI_Isend/Irecv/Waitall）




```
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:一次元問題



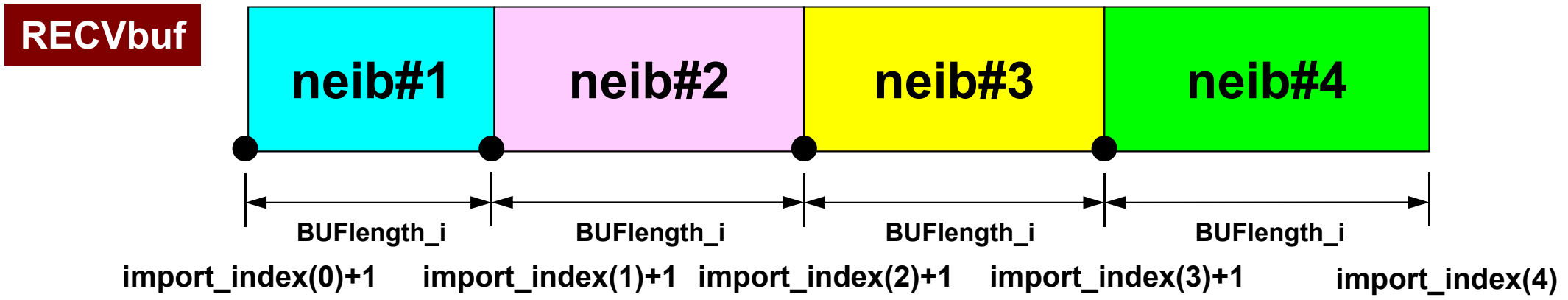
- 送信相手
  - NEIBPETOT, NEIBPE(neib)
    - NEIBPETOT=2, NEIB(1)= my_rank-1, NEIB(2)= my_rank+1
- それぞれの送信相手に送るメッセージサイズ
  - export_index(neib), neib= 0, NEIBPETOT
    - export_index(0)=0, export_index(1)= 1, export_index(2)= 2
- 「境界点」番号
  - export_item(k), k= 1, export_index(NEIBPETOT)
    - export_item(1)=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

一般化された通信テーブル：受信

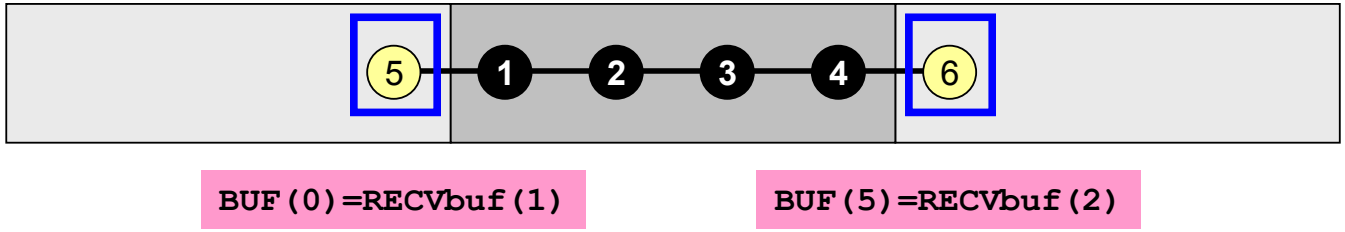
- 受信相手
  - NEIBPETOT，NEIBPE(neib)
- それぞれの受信相手から受け取るメッセージサイズ
  - import_index(neib), neib= 0, NEIBPETOT
- 「外点」番号
  - import_item(k), k= 1, import_index(NEIBPETOT)
- それぞれの受信相手から受け取るメッセージ
  - RECVbuf(k), k= 1, import_index(NEIBPETOT)

受信（MPI_Isend/Irecv/Waitall）

```
 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
```

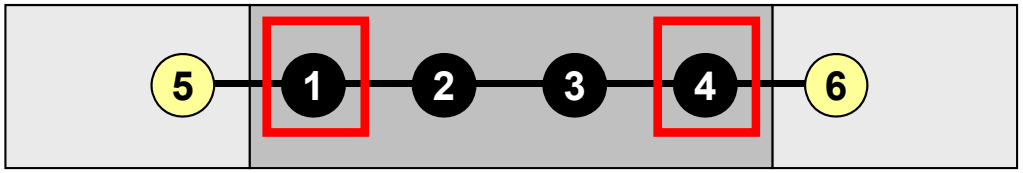


受信:一次元問題

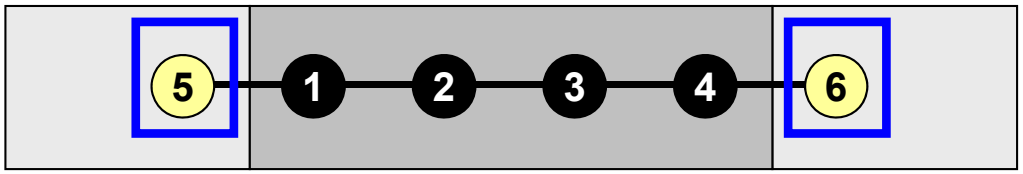


- 受信相手
  - NEIBPETOT，NEIBPE(neib)
    - NEIBPETOT=2, NEIB(1)= my_rank-1, NEIB(2)= my_rank+1
- それぞれの受信相手から受け取るメッセージサイズ
  - import_index(neib), neib= 0, NEIBPETOT
    - import_index(0)=0, import_index(1)= 1, import_index(2)= 2
- 「外点」番号
  - import_item(k), k= 1, import_index(NEIBPETOT)
    - import_item(1)=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



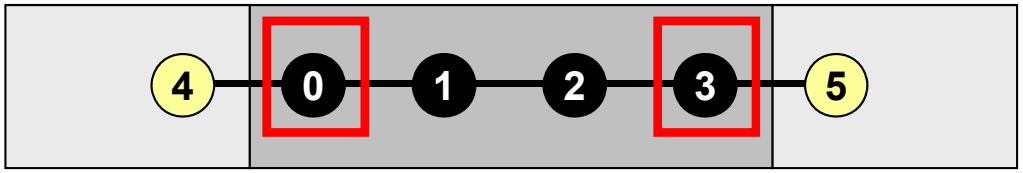
SENDbuf(1)=BUF(1)　　SENDbuf(2)=BUF(4)



BUF(5)=RECVbuf(1)　　BUF(6)=RECVbuf(2)

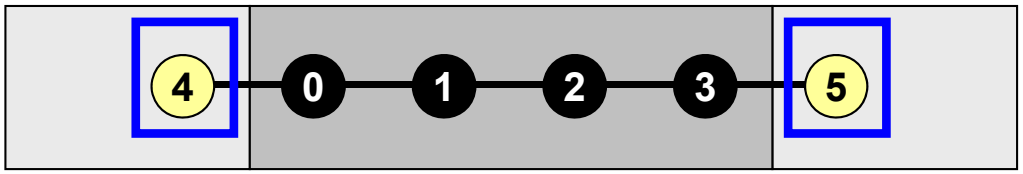
```
NEIBPETOT= 2
NEIBPE(1)= my_rank - 1
NEIBPE(2)= my_rank + 1

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]

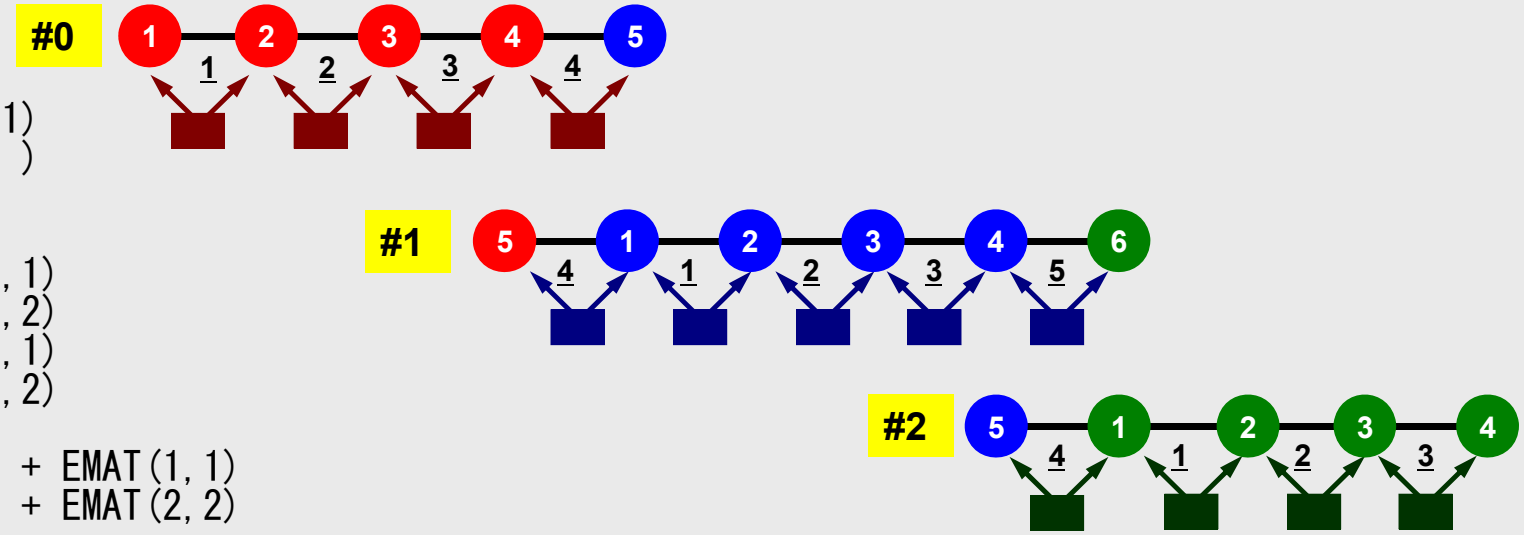
BUF[5]=RECVbuf[1]

```
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
```

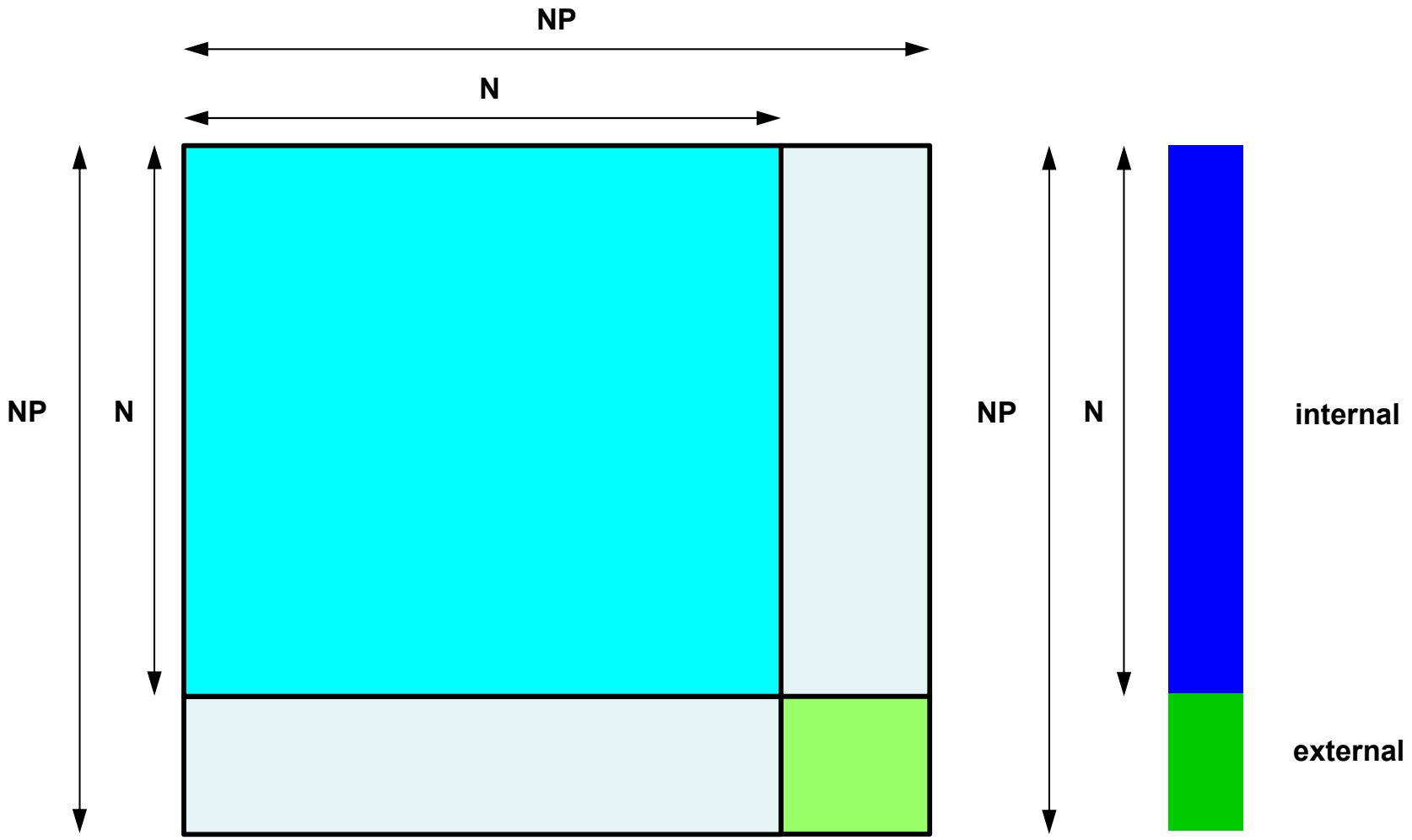
プログラム：1d.f(8/11)

全体マトリクス生成：1CPUのときと全く同じ：各要素→一様

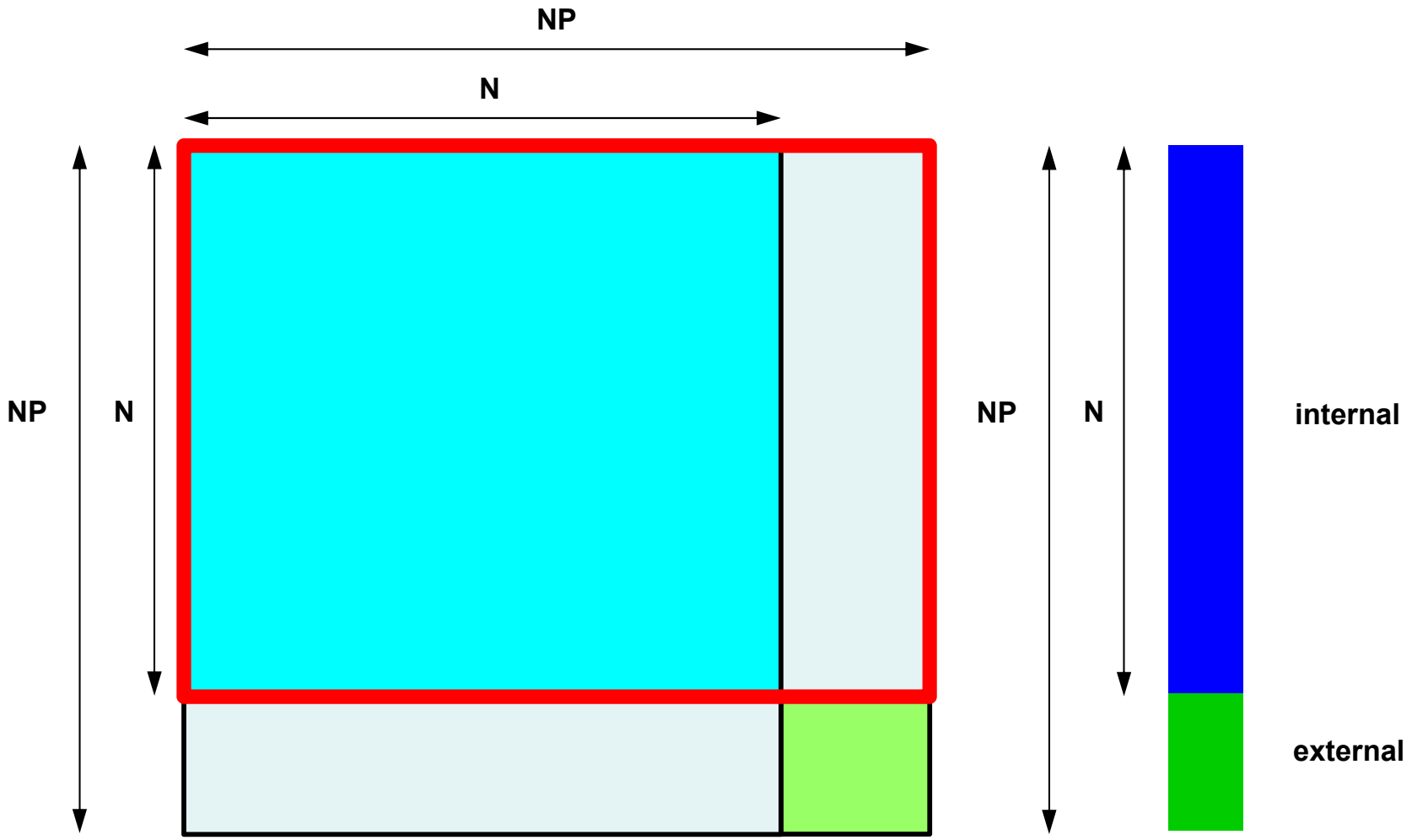
```
!C
!C +----------------+
!C | MATRIX ASSEMBLE |
!C +----------------+
!C===
      do icel= 1, NE
        in1= ICELNOD(2*icel-1)
        in2= ICELNOD(2*icel  )
        DL = dX
        cK= AREA*COND/DL
        EMAT(1,1)= Ck*KMAT(1,1)
        EMAT(1,2)= Ck*KMAT(1,2)
        EMAT(2,1)= Ck*KMAT(2,1)
        EMAT(2,2)= Ck*KMAT(2,2)

        DIAG(in1)= DIAG(in1) + EMAT(1,1)
        DIAG(in2)= DIAG(in2) + EMAT(2,2)

        if (my_rank.eq.0.and.icel.eq.1) then
          k1= INDEX(in1-1) + 1
         else
          k1= INDEX(in1-1) + 2
        endif
        k2= INDEX(in2-1) + 1

        AMAT(k1)= AMAT(k1) + EMAT(1,2)
        AMAT(k2)= AMAT(k2) + EMAT(2,1)

        QN= 0.50d0*QV*AREA*DL
        RHS(in1)= RHS(in1) + QN
        RHS(in2)= RHS(in2) + QN
      enddo
!C===
```



Local Matrix：各プロセスにおける係数行列



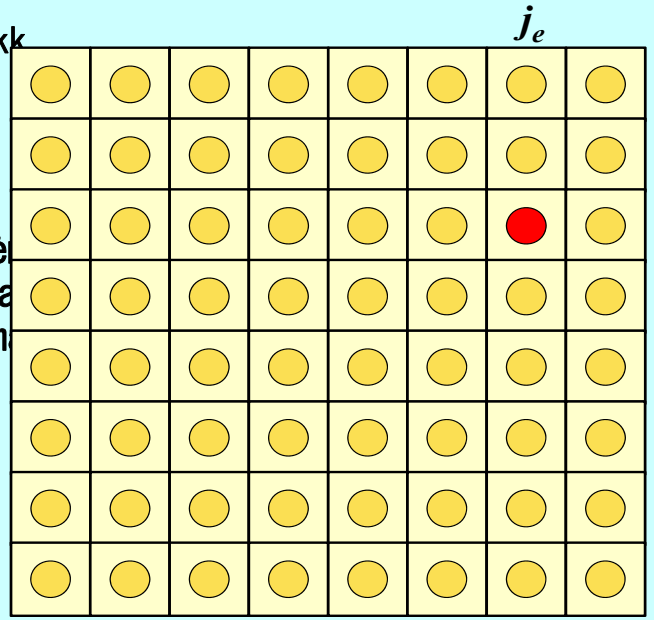
本当に必要なのはこの部分



MAT_ASS_MAIN: Overview

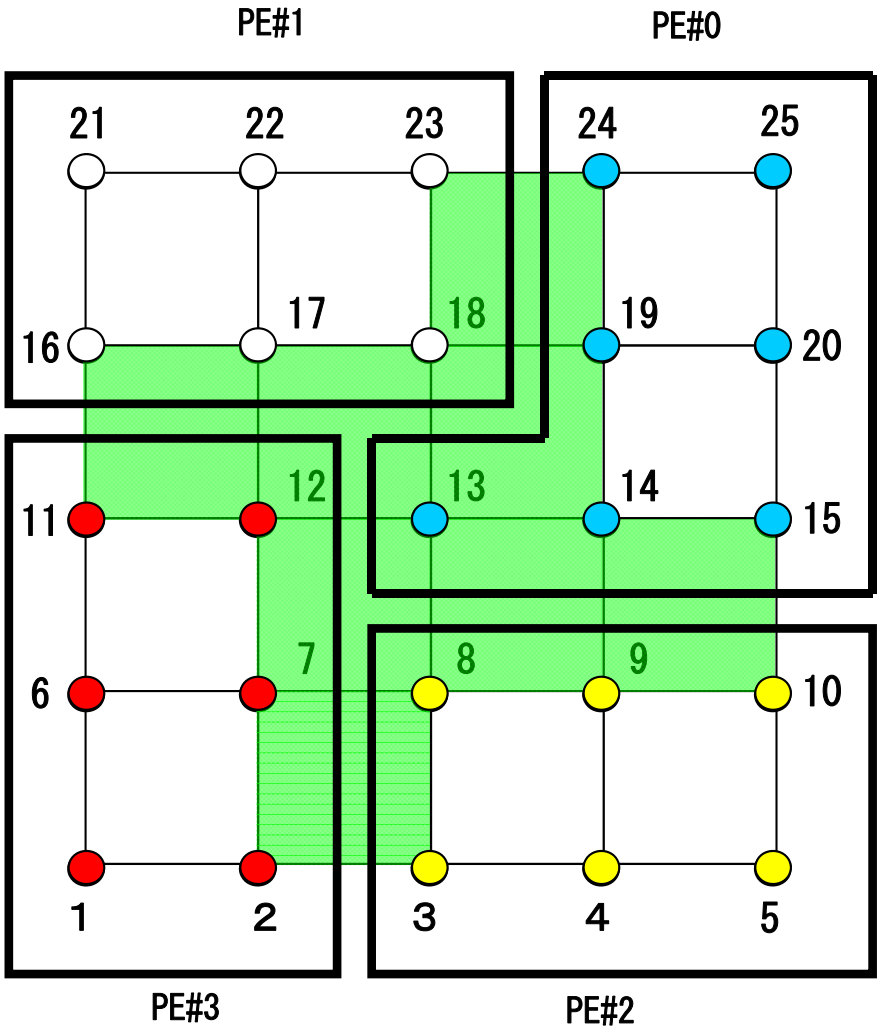
```
do kpn= 1, 2          Gaussian Quad. points in ζ-direction
  do jpn= 1, 2        Gaussian Quad. points in η-direction
    do ipn= 1, 2      Gaussian Quad. Pointe in ξ-direction
     Define Shape Function at Gaussian Quad. Points (8-points)
    Its derivative on natural/local coordinate is also defined.
                        enddo
                      enddo
                    enddo
```

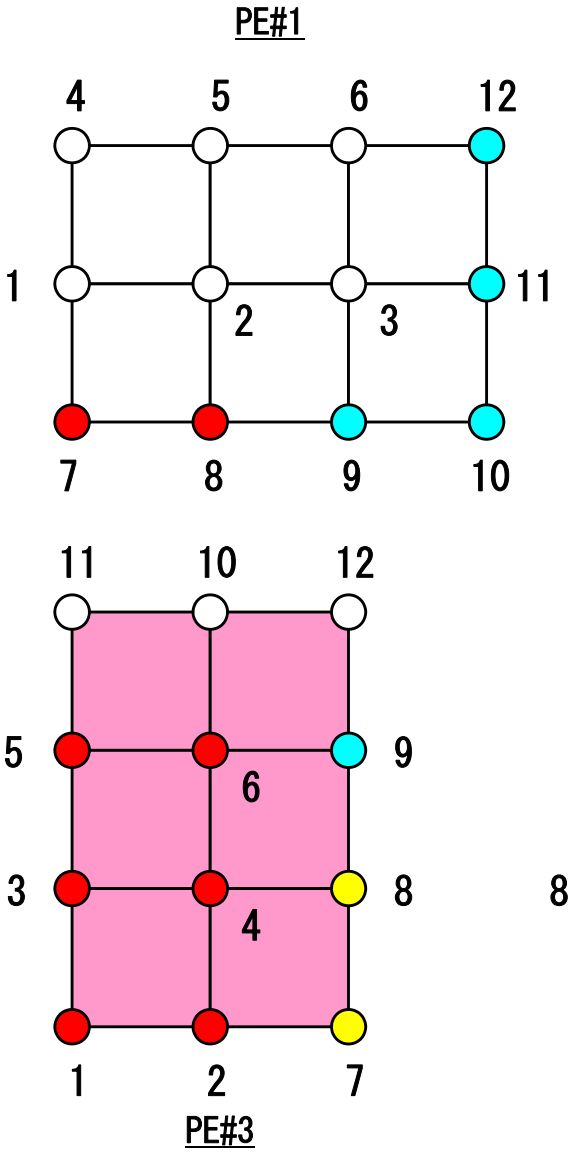
```
do icel= 1, ICELTOT    Loop for Element
Jacobian and derivative on global coordinate of shape functions at
Gaussian Quad. Points are defined according to coordinates of 8 nodes. (JACOBI)

                  do ie= 1, 8          Local Node ID
                    do je= 1, 8        Local Node ID
                          Global Node ID: ip, jp
                        Address of A_ip,jp in "item" : kk

          do kpn= 1, 2        Gaussian Quad. points
            do jpn= 1, 2      Gaussian Quad. points
              do ipn= 1, 2    Gaussian Quad. points
                        integration on each ele
                      coefficients of element ma
                       accumulation to global m
                              enddo
                            enddo
                          enddo
                        enddo
                      enddo
                    enddo
```



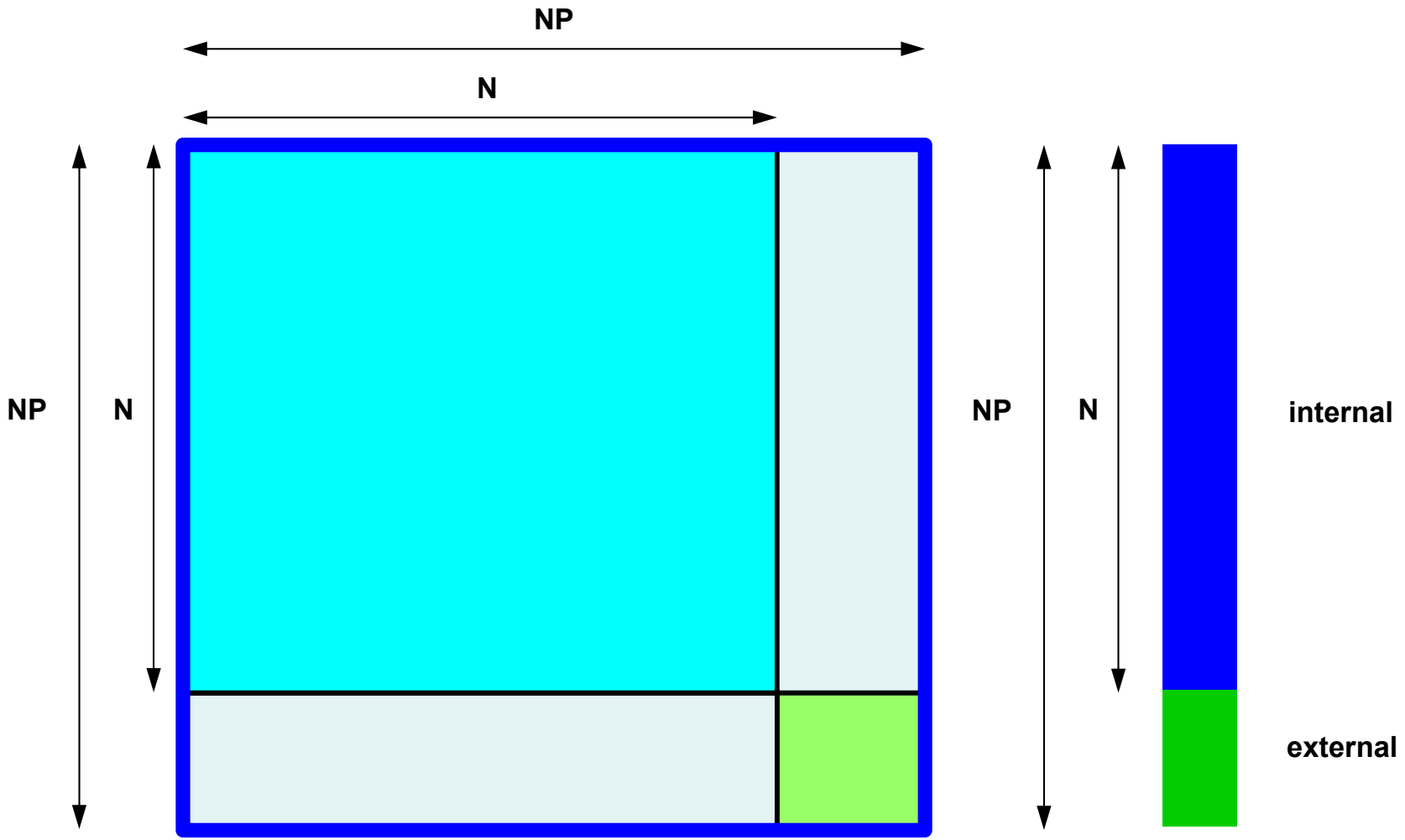
全ての要素の計算を実施する

外点を含むオーバーラップ領域の要素の計算も実施

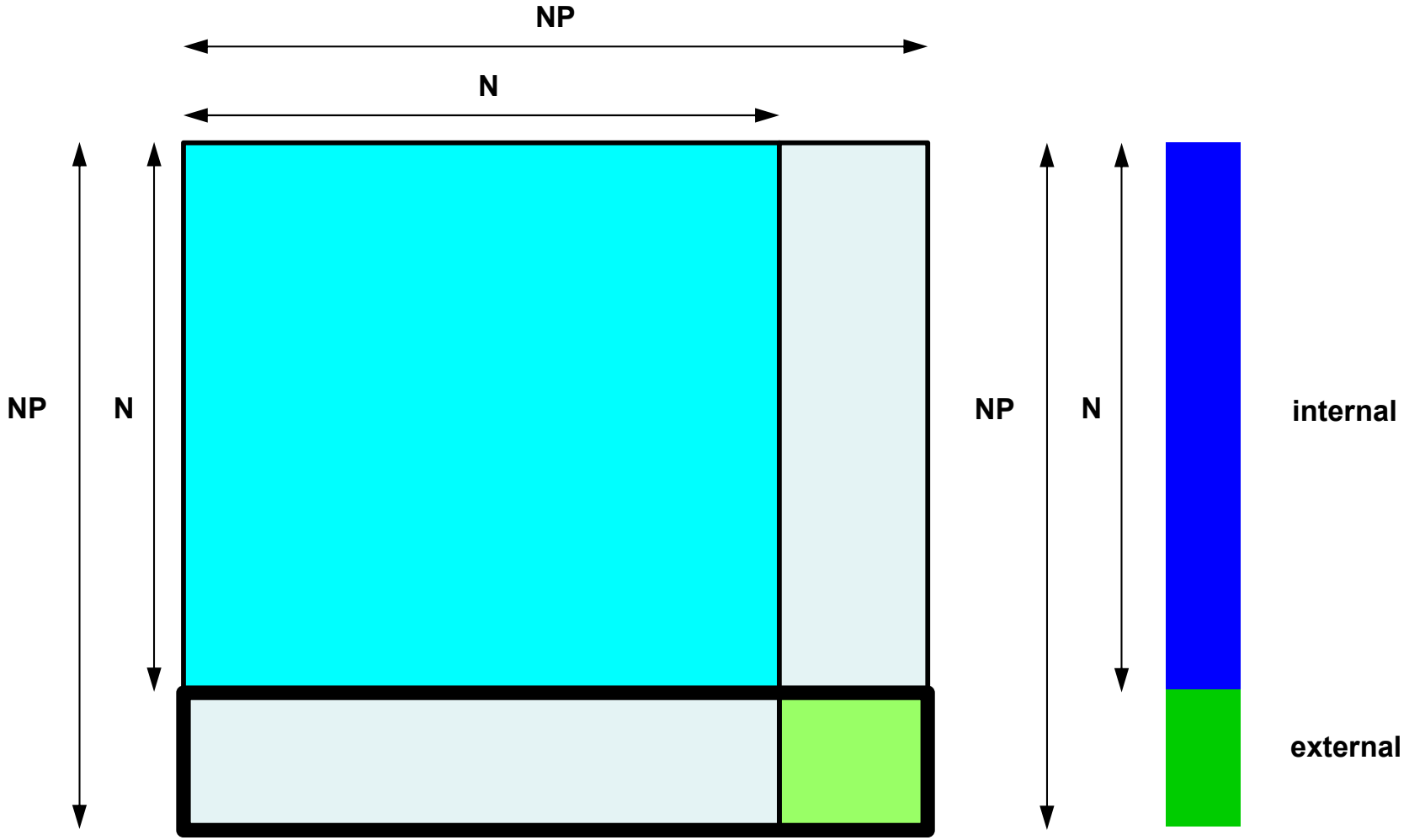




従って結果的にはこのような行列を得るが



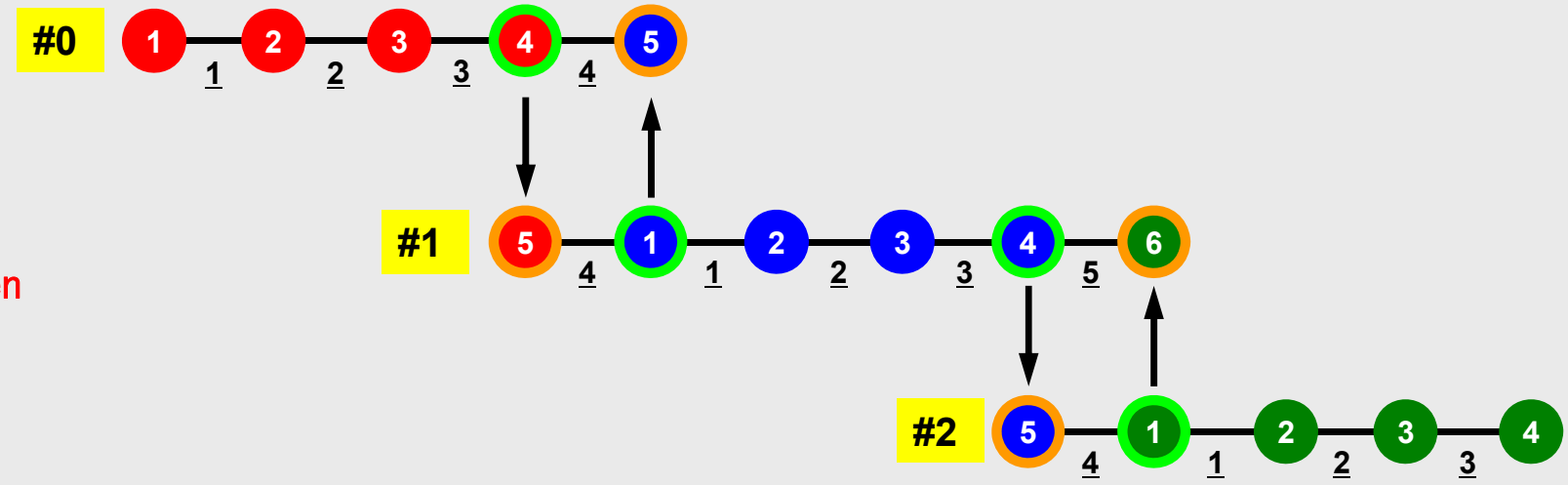
黒枠で囲んだ部分の行列は不完全
しかし，計算には使用しないのでこれで良い



プログラム：1d.f(9/11)

境界条件：1CPUのときとほとんど同じ

```
!C
!C +--------------------+
!C | BOUNDARY CONDITIONS |
!C +--------------------+
!C===

!C
!C-- X=Xmin

      if (my_rank.eq.0) then
        i = 1
        jS= INDEX(i-1)

        AMAT(jS+1)= 0.d0
        DIAG(i)= 1.d0
        RHS (i)= 0.d0
        do k= 1, NPLU
          if (ITEM(k).eq.1) AMAT(k)= 0.d0
        enddo
      endif

!C===
```



プログラム：1d.f(10/11)
共役勾配法

```
!C
!C +--------------+
!C | CG iterations |
!C +--------------+
!C===
      R = 1
      Z = 2
      Q = 2
      P = 3
      DD= 4

      do i= 1, N
        W(i,DD)= 1.0D0 / DIAG(i)
      enddo

!C
!C-- {r0}= {b} - [A]{xini} |

!C-    init

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

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

共役勾配法

- 行列ベクトル積
- 内積
- 前処理：1CPUのときと同じ
- DAXPY：1CPUのときと同じ

前処理，DAXPY

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

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

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

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

行列ベクトル積(1/2)

通信テーブル使用，{p}の最新値を計算前に取得

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

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

      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_DOUBLE_PRECISION,        &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, request_send(neib), ierr)
      enddo

      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_DOUBLE_PRECISION,        &
     &                  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)
          W(kk,P)= RECVbuf(k)
        enddo
      enddo
```

行列ベクトル積(2/2)

{q}= [A]{p}

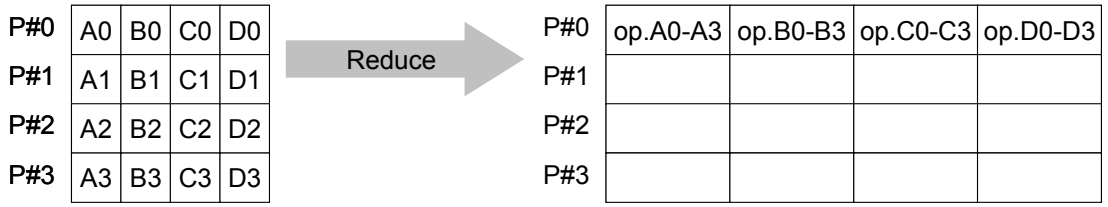
```
call MPI_Waitall (NEIBPETOT, request_send, stat_send, ierr)

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
```

内積

各プロセスで計算した値を，MPI_Allreduceで合計

```
!C
!C-- RHO= {r}{z}

      RHO0= 0.d0
      do i= 1, N
        RHO0= RHO0 + W(i,R)*W(i,Z)
      enddo
      call MPI_Allreduce (RHO0, RHO, 1, MPI_DOUBLE_PRECISION,          &
     &                    MPI_SUM, MPI_COMM_WORLD, ierr)
```

MPI_REDUCE



- コミュニケーター「comm」内の，各プロセスの送信バッファ「sendbuf」について，演算「op」を実施し，その結果を1つの受信プロセス「root」の受信バッファ「recbuf」に格納する。
  - 総和，積，最大，最小 他

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

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

  - **op** 整数 I 計算の種類

    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
    ユーザーによる定義も可能：MPI_OP_CREATE
  - **root** 整数 I 受信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

送信バッファと受信バッファ

- MPIでは「送信バッファ」，「受信バッファ」という変数がしばしば登場する。

- 送信バッファと受信バッファは必ずしも異なった名称の配列である必要はないが，必ずアドレスが異なっていなければならない。

MPI_REDUCEの例（1/2）

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

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

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

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

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

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

MPI_REDUCEの例(2/2)

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

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

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

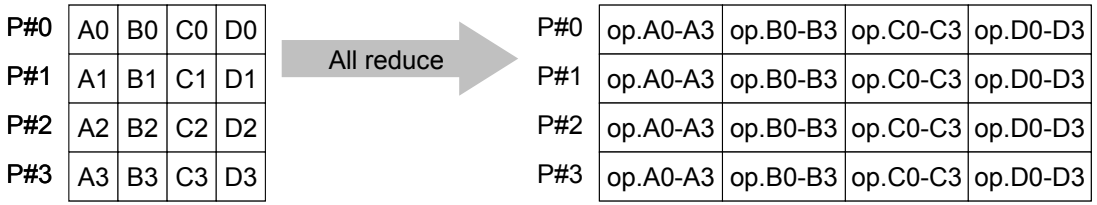
各プロセスにおける，X0の総和が0番PEのXSUMに入る。

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

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

各プロセスにおける，

- X0(1)の総和が0番プロセスのX0(3)に入る。
- X0(2)の総和が0番プロセスのX0(4)に入る。

MPI_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

CG法(1/5)

```
!C
!C-- {r0}= {b} - [A]{xini}
    do neib= 1, NEIBPETOT
      do k= export_index(neib-1)+1, export_index(neib)
        kk= export_item(k)
        SENDbuf(k)= PHI(kk)
      enddo
    enddo

    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_DOUBLE_PRECISION,
     &                NEIBPE(neib), 0, MPI_COMM_WORLD,
     &                request_send(neib), ierr)
    enddo

    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_DOUBLE_PRECISION,
     &                NEIBPE(neib), 0, MPI_COMM_WORLD,
     &                request_recv(neib), ierr)
    enddo
    call MPI_Waitall (NEIBPETOT, request_recv, stat_recv, ier

    do neib= 1, NEIBPETOT
      do k= import_index(neib-1)+1, import_index(neib)
        kk= import_item(k)
        PHI(kk)= RECVbuf(k)
      enddo
    enddo
    call MPI_Waitall (NEIBPETOT, request_send, stat_send, ierr)
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

- solve $[M]z^{(i-1)}= r^{(i-1)}$
- $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
- 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
- $q^{(i)}= [A]p^{(i)}$
- $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
- $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
- $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
- check convergence $|r|$

end

CG法(2/5)

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

      BNRM20= 0.0D0
      do i= 1, N
        BNRM20 = BNRM20  + RHS(i)  **2
        W(i,R) = RHS(i) - W(i,R)
      enddo
      call MPI_Allreduce (BNRM20, BNRM2, 1,
     &                    MPI_DOUBLE_PRECISION,
     &                    MPI_SUM, MPI_COMM_WORLD, ierr)

!C***********************************************************
      do iter= 1, ITERmax

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

      do i= 1, N
        W(i,Z)= W(i,DD) * W(i,R)
      enddo
!C
!C-- RHO= {r}{z}

      RHO0= 0.d0
      do i= 1, N
        RHO0= RHO0 + W(i,R)*W(i,Z)
      enddo
      call MPI_Allreduce (RHO0, RHO, 1, MPI_DOUBLE_PRECISION,
     &                    MPI_SUM, MPI_COMM_WORLD, ierr)
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for **i= 1, 2, …**

  solve $[M]z^{(i-1)}= r^{(i-1)}$

  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$

  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

  $q^{(i)}= [A]p^{(i)}$

  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$

  check convergence |r|

end

CG法(3/5)

```
!C
!C-- {p} = {z} if      ITER=1
!C   BETA= RHO / RHO1  otherwise

      if ( iter.eq.1 ) then
        do i= 1, N
          W(i,P)= W(i,Z)
        enddo
       else
        BETA= RHO / RHO1
        do i= 1, N
          W(i,P)= W(i,Z) + BETA*W(i,P)
        enddo
      endif


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

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

      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_DOUBLE_PRECISION,       &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD,                &
     &                  request_send(neib), ierr)
       enddo
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for **i= 1, 2, …**

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

end

CG法(4/5)

```
  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_DOUBLE_PRECISION,
&                   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)
      W(kk,P)= RECVbuf(kk)
    enddo
  enddo
  call MPI_Waitall (NEIBPETOT, request_send, stat_send, ierr)

  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

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

  C10= 0.d0
  do i= 1, N
    C10= C10 + W(i,P)*W(i,Q)
  enddo
  call MPI_Allreduce (C10, C1, 1, MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, ierr)
  ALPHA= RHO / C1
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

- solve $[M]z^{(i-1)}= r^{(i-1)}$
- $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
- 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
- **$q^{(i)}= [A]p^{(i)}$**
- **$\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$**
- $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
- $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
- check convergence |r|

end

CG法(5/5)

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

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

      DNRM20 = 0.0
      do i= 1, N
        DNRM20= DNRM20 + W(i,R)**2
      enddo

      call MPI_Allreduce (DNRM20, DNRM2, 1,
     &                    MPI_DOUBLE_PRECISION,
     &                    MPI_SUM, MPI_COMM_WORLD, ierr)

        RESID= dsqrt(DNRM2/BNRM2)

        if (my_rank.eq.0.and.mod(iter,1000).eq.0) then
          write (*, '(i8,1pe16.6)') iter, RESID
        endif

        if ( RESID.le.EPS) goto 900
        RHO1 = RHO

      enddo
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

- solve $[M]z^{(i-1)}= r^{(i-1)}$
- $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
- 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
- $q^{(i)}= [A]p^{(i)}$
- $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
- **$x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$**
- **$r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$**
- **check convergence |r|**

end

プログラム：1d.f(11/11)

結果書き出し：各プロセスごとに実施

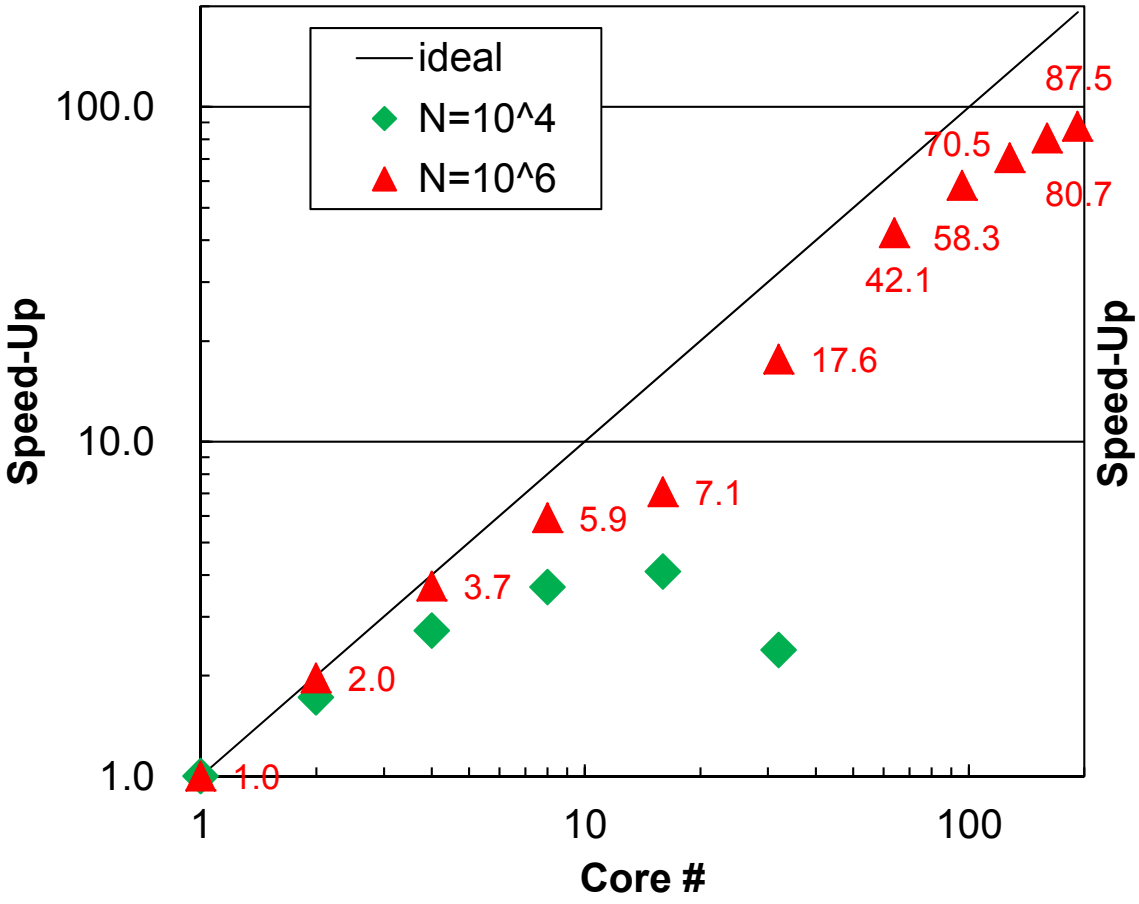
```
!C
!C-- OUTPUT
        if (my_rank.eq.0) then
          write (*,'(2(1pe16.6))') E1Time-S1Time, E2Time-E1Time
        endif

      write (*,'(/a)') '### TEMPERATURE'
      do i= 1, N
        write (*,'(2i8, 2(1pe16.6))') my_rank, i, PHI(i)
      enddo

      call MPI_FINALIZE (ierr)
      end program heat1Dp
```

- 問題の概要，実行方法
- 局所分散データの考え方
- プログラムの説明
- **計算例**

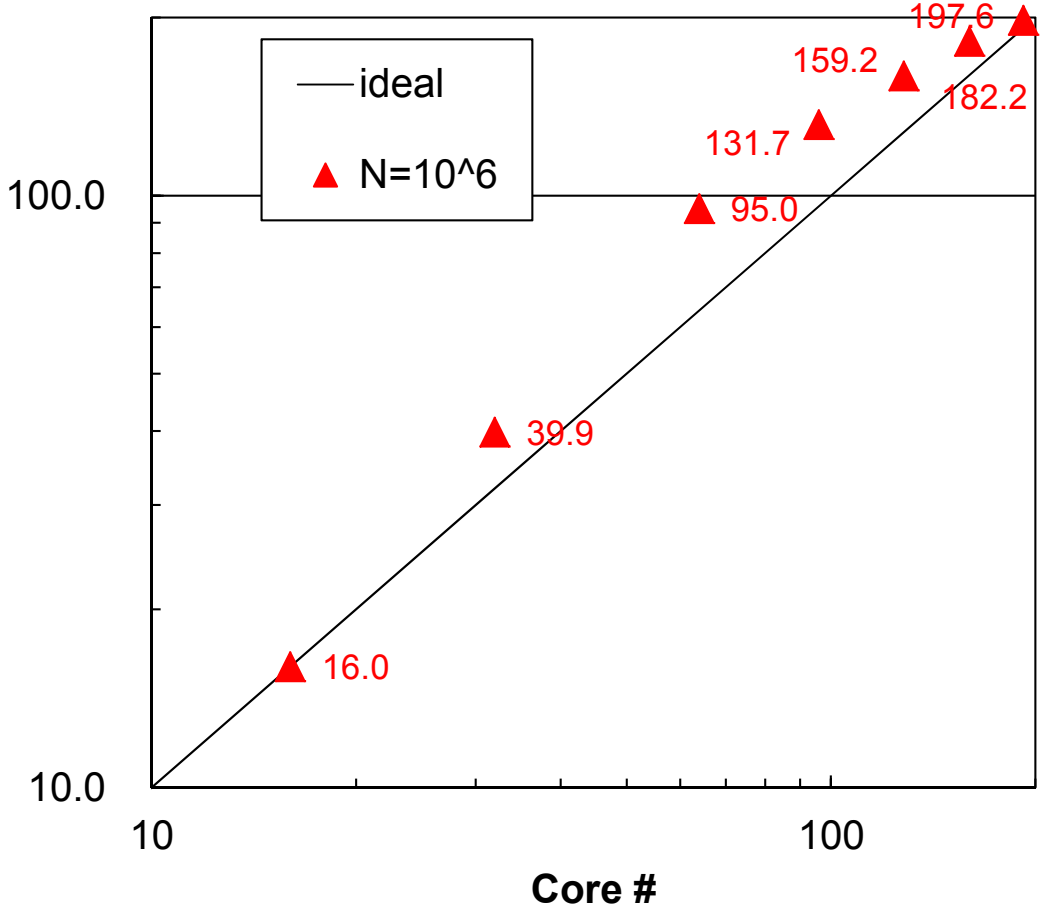
計算結果(1次元):CG法部分

N=10^6の場合は100回反復に要する時間

1コアを基準



1ノード・16コアを基準



理想値からのずれ

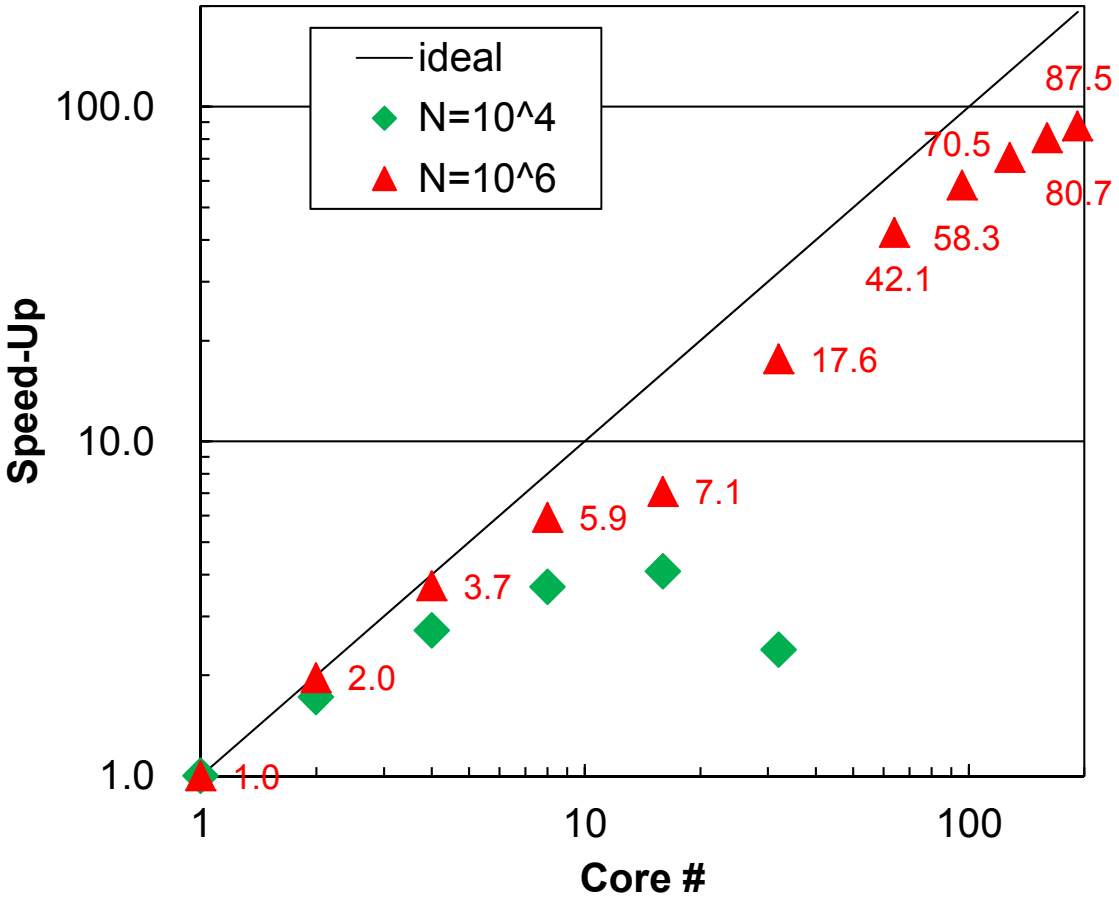
- MPI通信そのものに要する時間
  - データを送付している時間
  - ノード間においては通信バンド幅によって決まる
    - Gigabit Ethernetでは 1Gbit/sec.（理想値）
  - 通信時間は送受信バッファのサイズに比例
- MPIの立ち上がり時間
  - latency
  - 送受信バッファのサイズによらない
    - 呼び出し回数依存，プロセス数が増加すると増加する傾向
  - 通常，数～数十μsecのオーダー
- MPIの同期のための時間
  - プロセス数が増加すると増加する傾向

理想値からのずれ（続き）

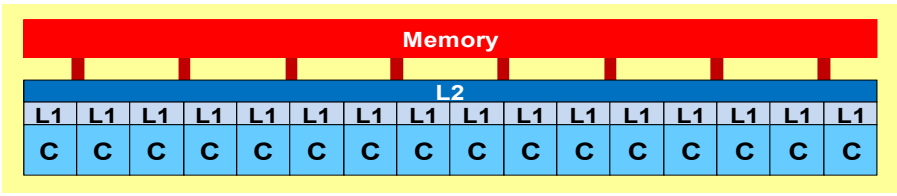
- 計算時間が小さい場合（S1-3ではNが小さい場合）はこれらの効果を無視できない。
  - 特に，送信メッセージ数が小さい場合は，「Latency」が効く。

1コア～16コアであまり性能が出ていない件

1コアを基準



- 16コアで1コアの7.1倍程度の性能にしかなっていないのは，メモリ競合のため。
  - STREAMのケース
  - 通信が原因ではない

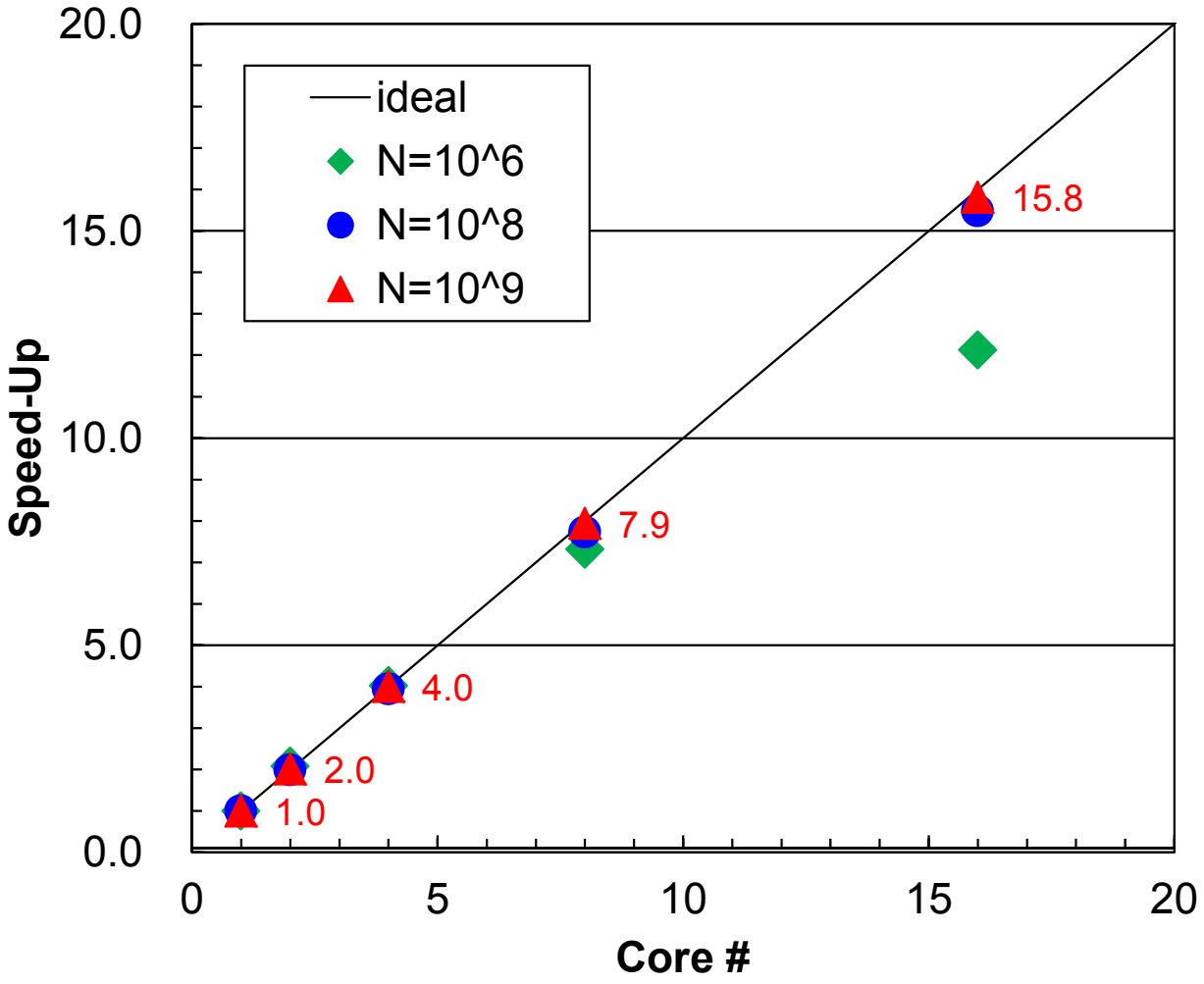


台形積分ではあまり影響が無い

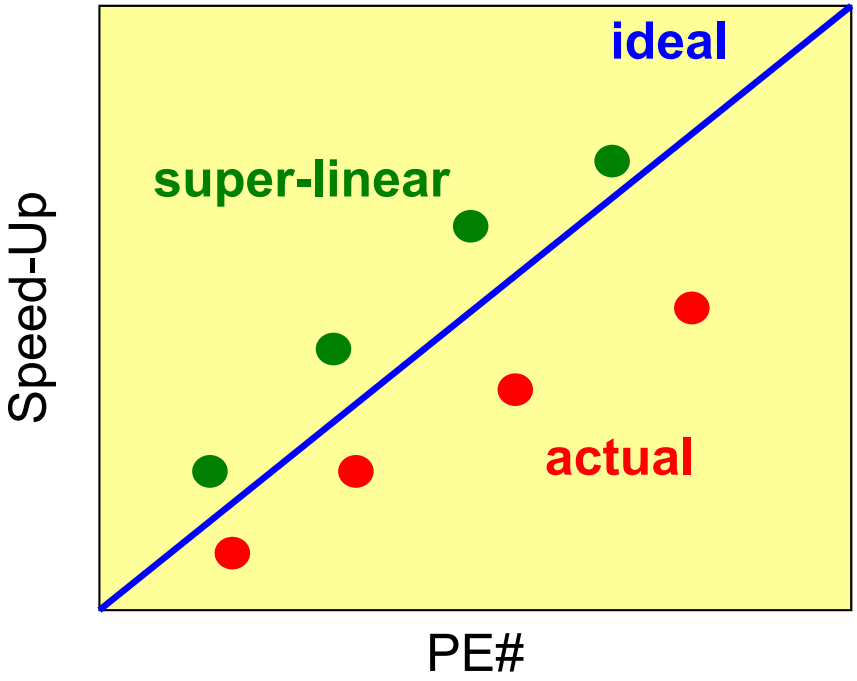
- ◆:N=10^6, ●:10^8, ▲:10^9, ―:理想値
- 1コアにおける計測結果(sec.)からそれぞれ算出

- 台形積分:ほとんどメモリを使わない，メモリに負担のかからないアプリケーション
- 1データ(スカラー)をAllreduceするだけ

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

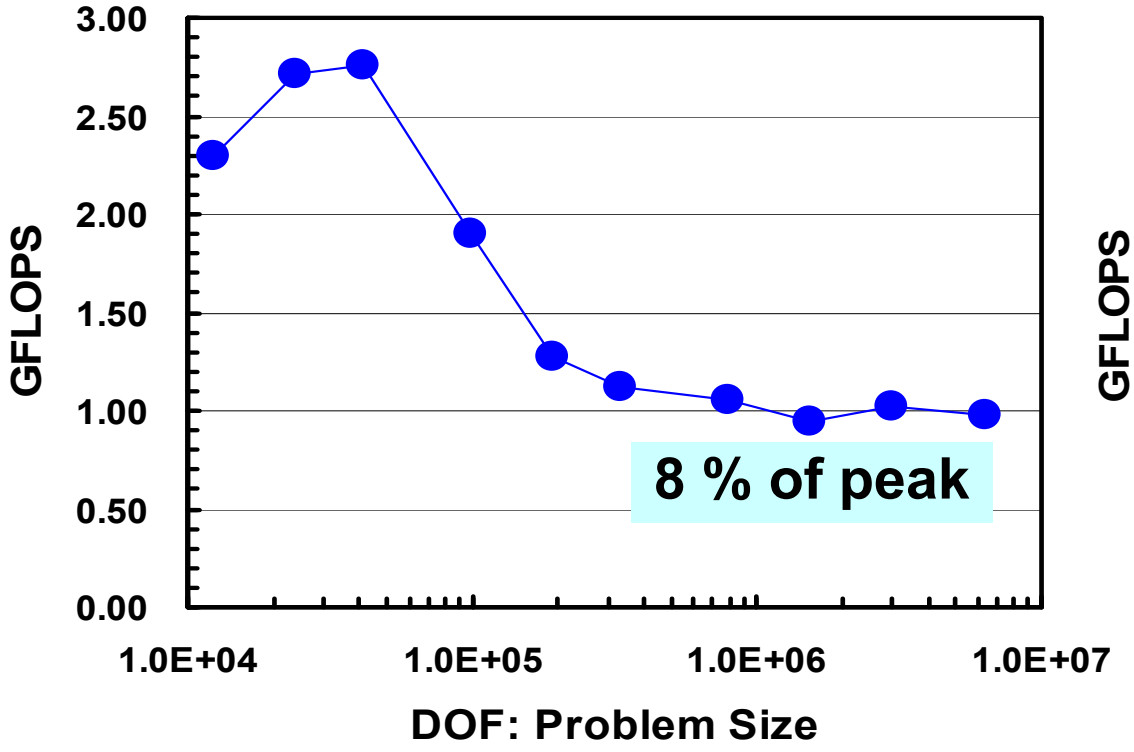


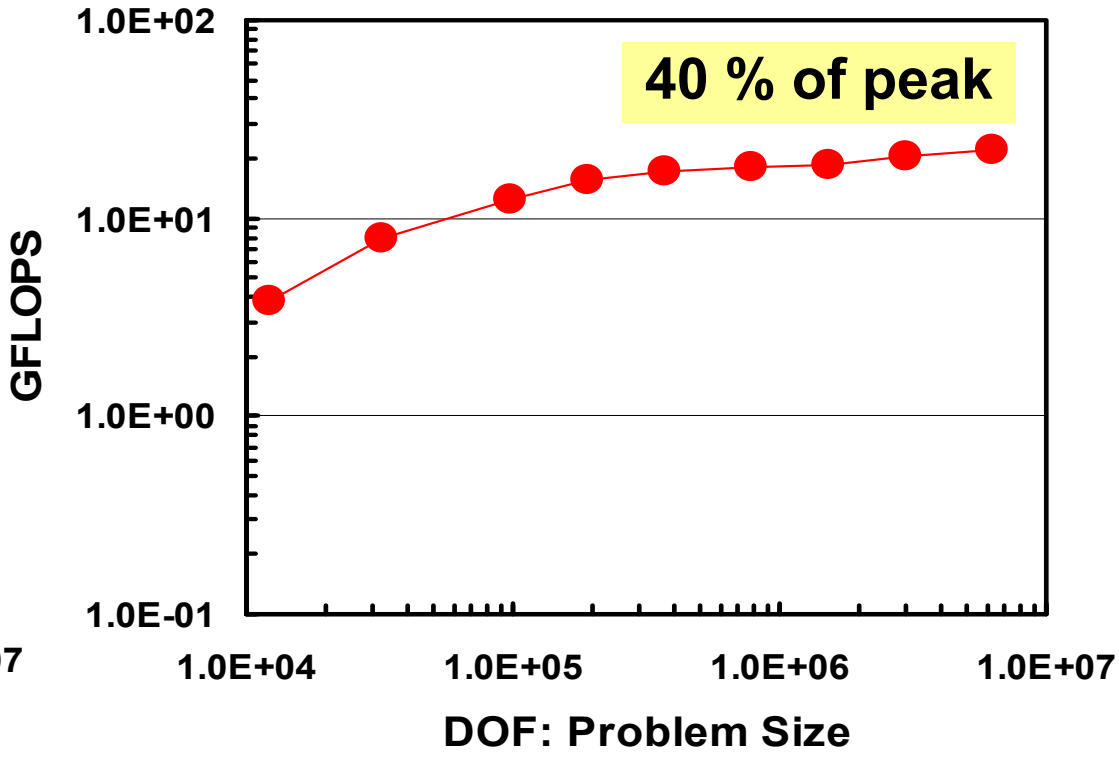
Strong-Scalingにおける「Super-Linear」



- 問題規模を固定して，使用PE数を増加させて行った場合，通常は通信の影響のために，効率は理想値（m個のPEを使用した場合，理想的にはm倍の性能になる）よりも低くなるのが普通である。
- しかし，スカラープロセッサ（PC等）の場合，逆に理想値よりも，高い性能が出る場合がある。このような現象を「Super-Linear」と呼ぶ。
  - ベクトル計算機では起こらない。

典型的な挙動

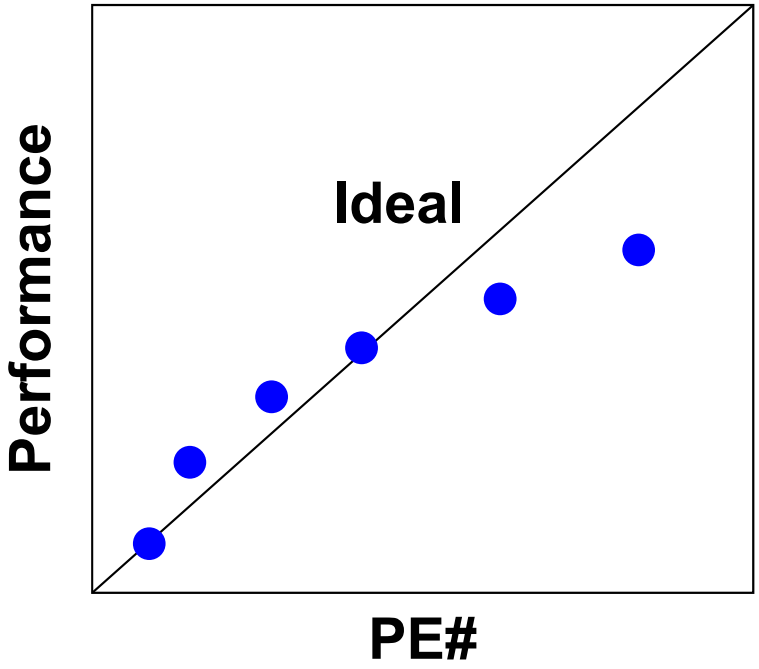


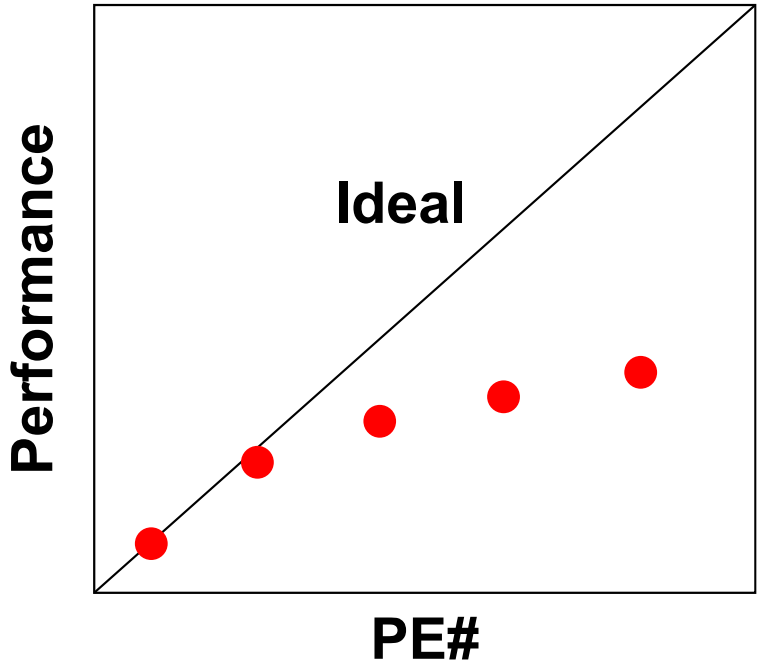


IBM-SP3:
問題サイズが小さい場合はキャッシュの影響のため性能が良い

Earth Simulator:
大規模な問題ほどベクトル長が長くなり，性能が高い

並列計算
Strong Scaling（全体問題規模固定）



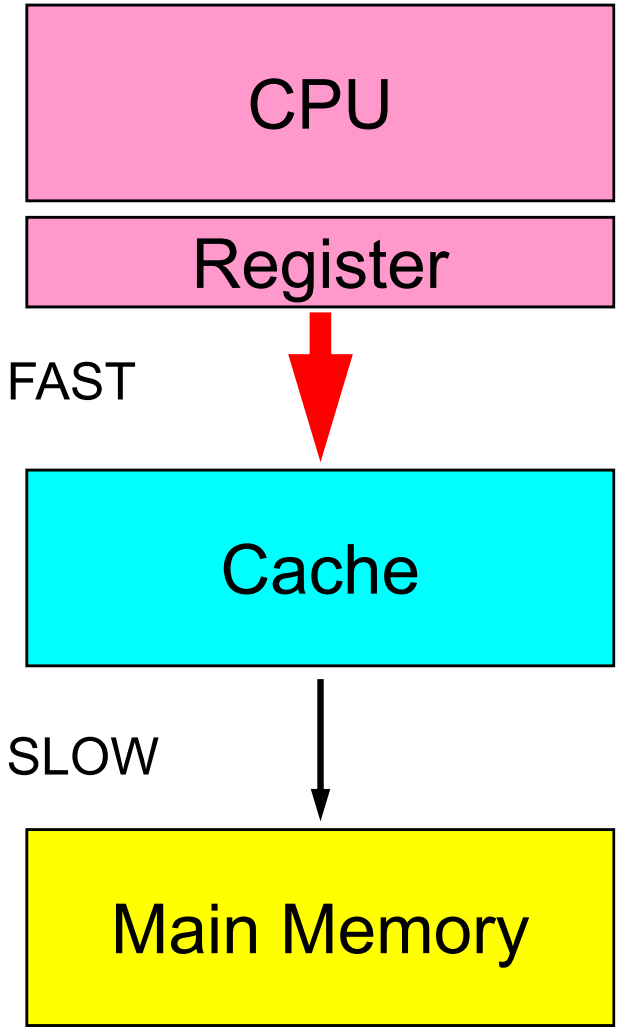


IBM-SP3:
PE（Processing Element）数が少ない場合はいわゆるスーパースカラー。
PE数が増加すると通信オーバーヘッドのため性能低下。

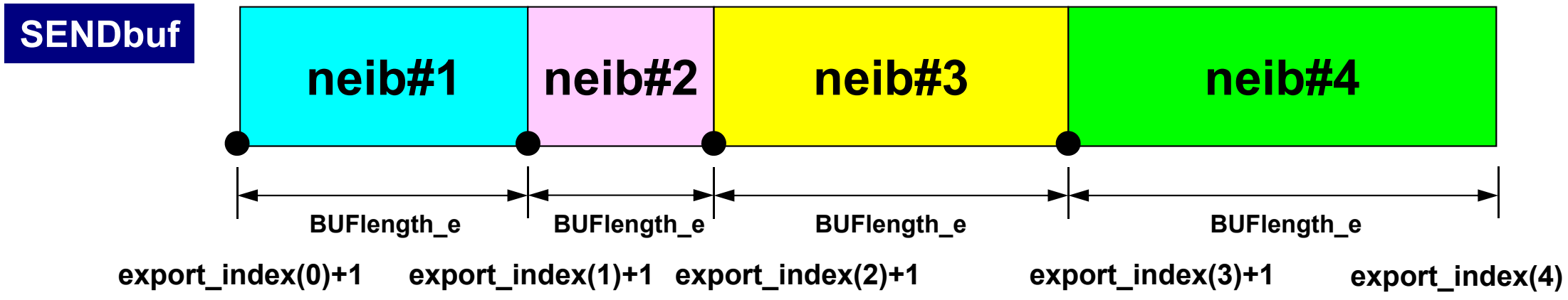
Earth Simulator:
PE数が増加すると，通信オーバーヘッドに加え，PEあたりの問題規模が小さくなるため性能低下。

Super-Linearの生じる理由

- キャッシュの影響
- スカラープロセッサでは，全般に問題規模が小さいほど性能が高い。
  - キャッシュの有効利用



メモリーコピーも意外に時間かかる(1/2)

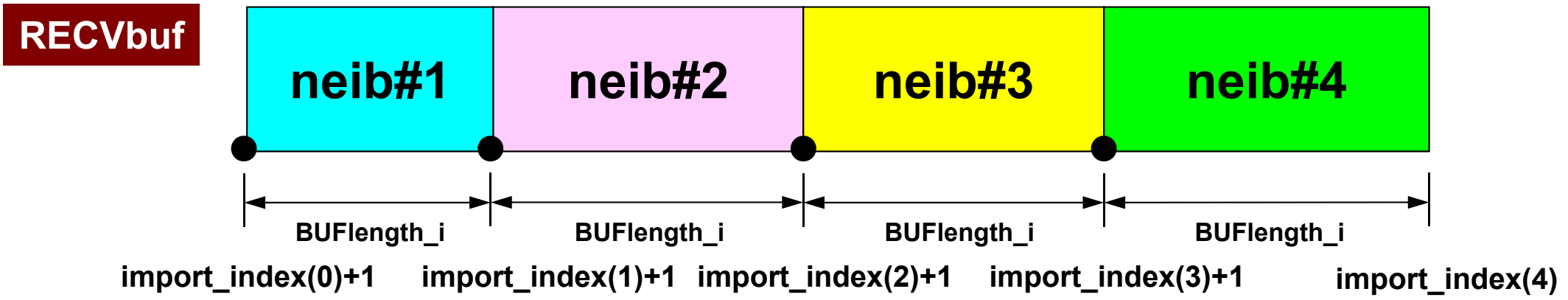



```
 do neib= 1, NEIBPETOT
   do k= export_index(neib-1)+1, export_index(neib)
     kk= export_item(k)
     SENDbuf(k)= VAL(kk)
   enddo
 enddo

 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。

メモリーコピーも意外に時間かかる(2/2)

```
 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
```



並列有限要素法:まとめ

- 「局所分散データ構造の適切な設計」に尽きる

- 問題点
  - 並列メッシュ生成，並列可視化
  - 悪条件問題における並列前処理手法
  - 大規模I/O

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