

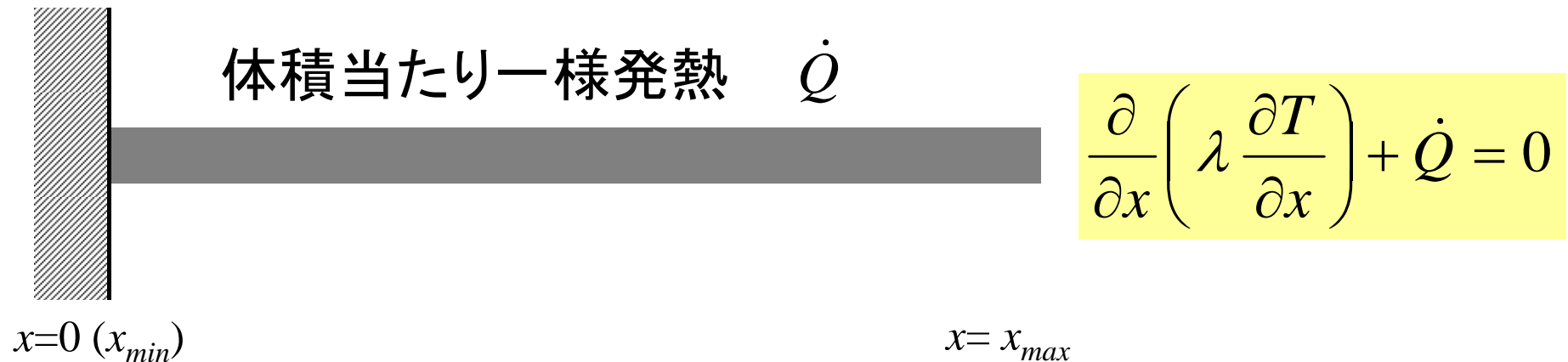
# 並列有限要素法による 一次元定常熱伝導解析プログラム C言語編

中島 研吾

東京大学情報基盤センター

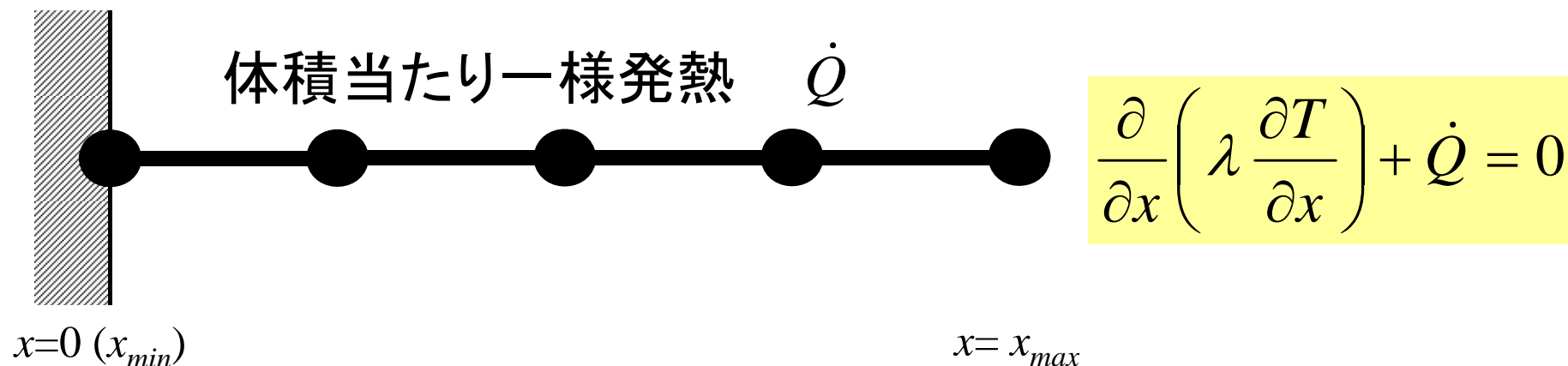
- 問題の概要, 実行方法
- 局所分散データの考え方
- プログラムの説明
- 計算例

# 対象とする問題：一次元熱伝導問題



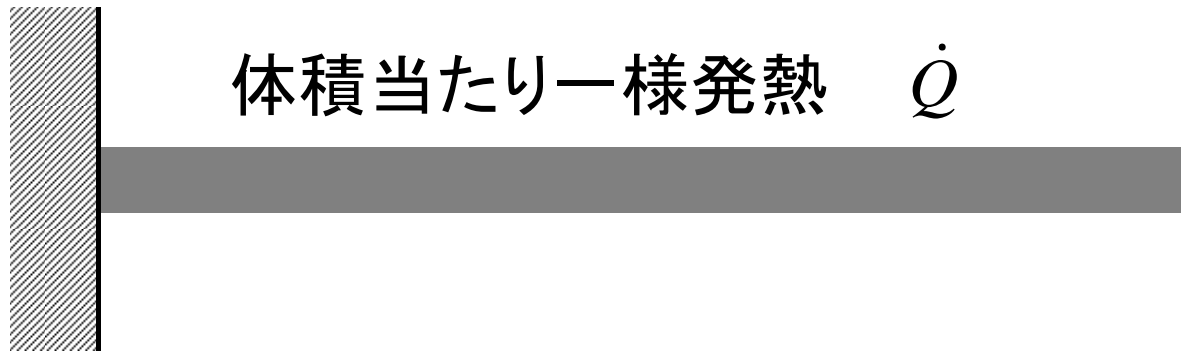
- 一様な：断面積 $A$ ，熱伝導率 $\lambda$
- 体積当たり一様発熱（時間当たり） $[QL^{-3}T^{-1}]$   $\dot{Q}$
- 境界条件
  - $x=0$  :  $T=0$  （固定）
  - $x=x_{max}$  :  $\frac{\partial T}{\partial x} = 0$  (断熱)

# 対象とする問題：一次元熱伝導問題



- 一様な：断面積 $A$ ，熱伝導率 $\lambda$
- 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）
  - $x=x_{max}$  :  $\frac{\partial T}{\partial x} = 0$  (断熱)

# 解析解



体積当たり一様発熱  $\dot{Q}$

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

$x=0$  ( $x_{min}$ )

$$T = 0 @ x = 0$$

$x = x_{max}$

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

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

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

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

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

# 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
```

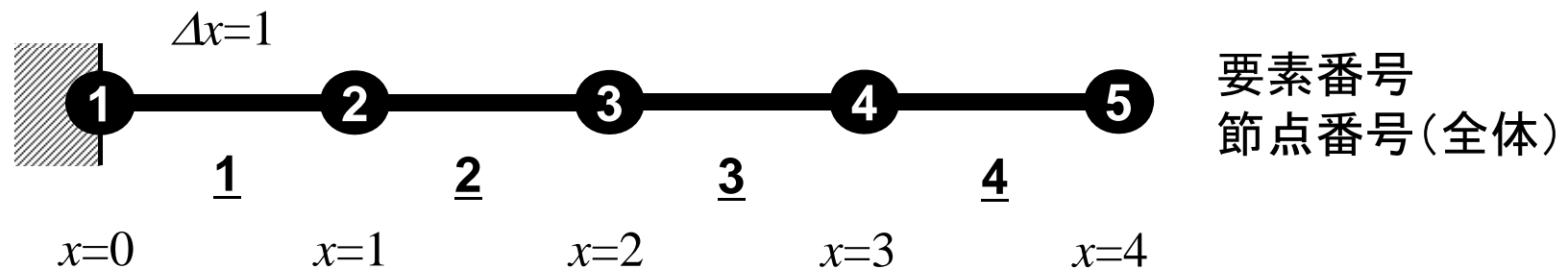
```
>$ mpifrtpx -Kfast 1d.f
```

```
>$ 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	$\Delta x$ (要素長さL), Q, A, $\lambda$
100	反復回数 (CG法後述)
1.e-8	CG法の反復打切誤差





# ジョブスクリプト: 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

"node=1"  
"proc=8"

16分割

"node=1"  
"proc=16"

32分割

"node=2"  
"proc=32"

64分割

"node=4"  
"proc=64"

192分割

"node=12"  
"proc=192"

# 「並列計算」の手順

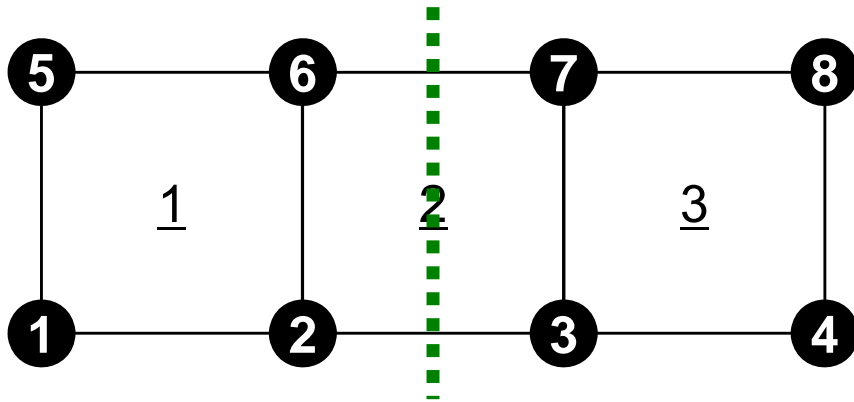
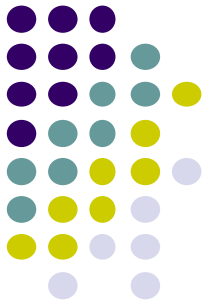
- 制御ファイル, 「全要素数」を読み込む
- 内部で「局所分散メッシュデータ」を生成する
- マトリクス生成
- 共役勾配法によりマトリクスを解く
  
- 元のプログラムとほとんど変わらない

- 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

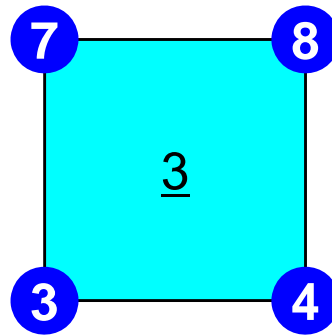
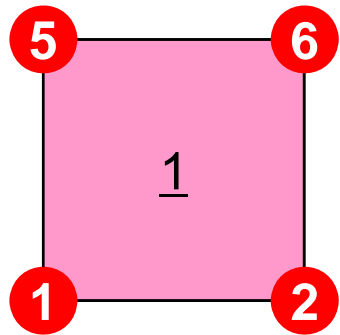
# 有限要素法の処理：プログラム

- 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)

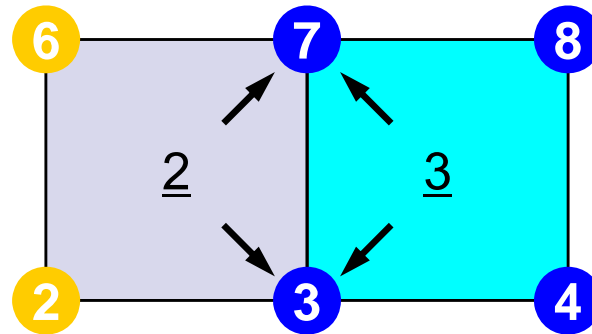
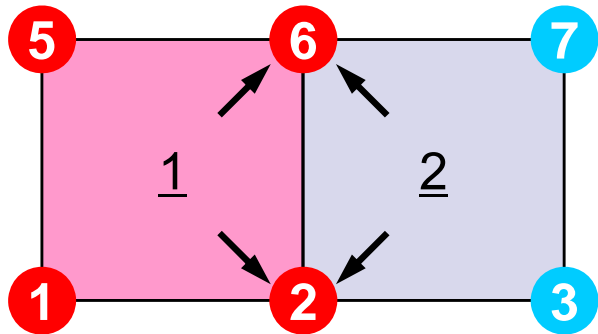
# 四角形要素



「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

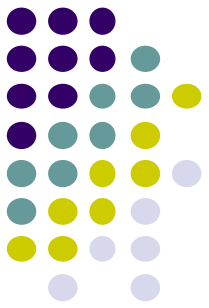


これではマトリクス生成に必要な情報は不十分



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

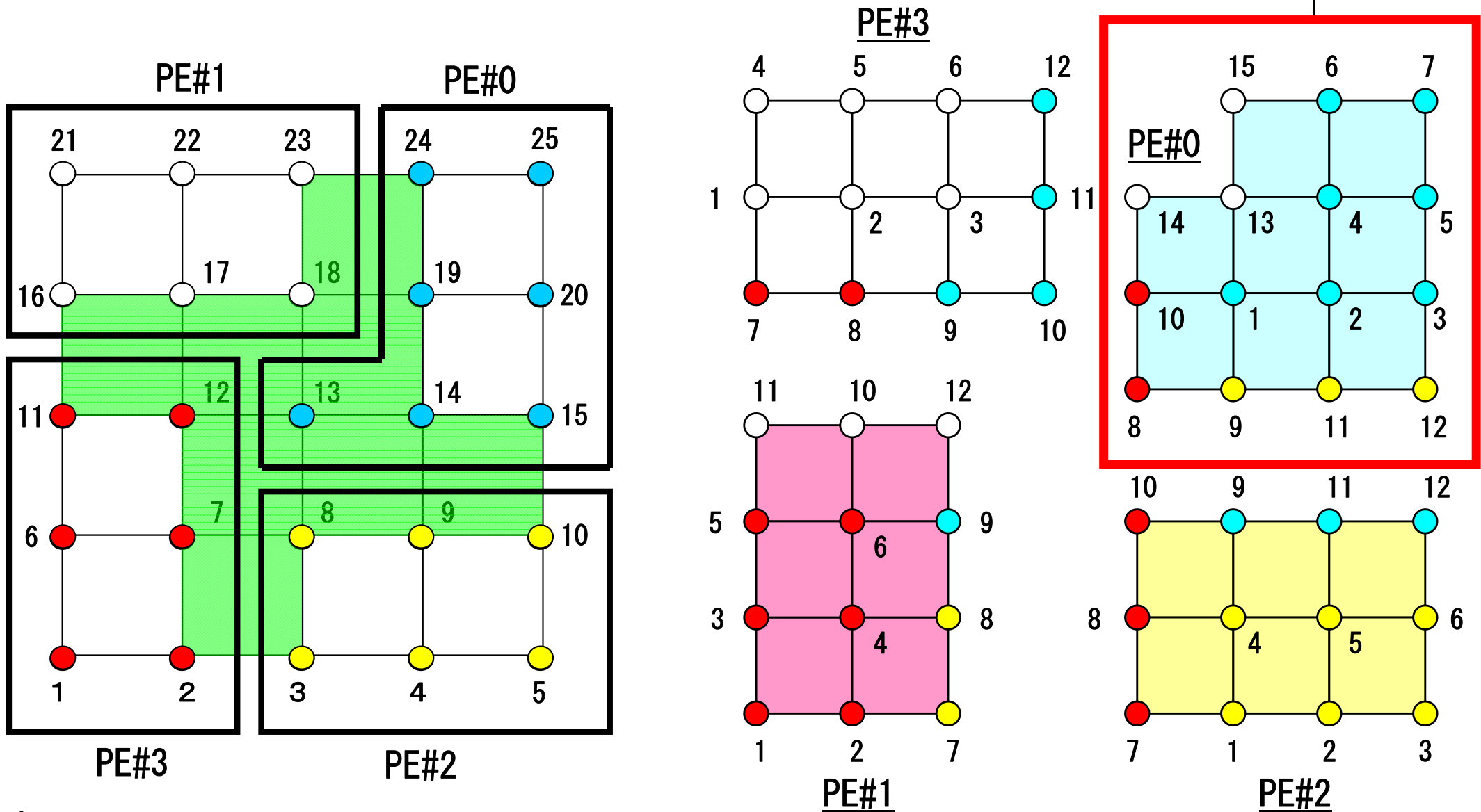
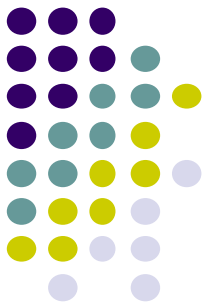
# 並列有限要素法の局所データ構造



- **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
  - 有限要素法の特長 : 要素で閉じた計算

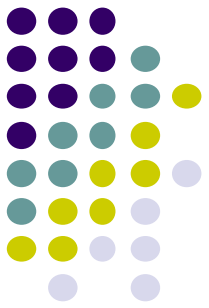
# Node-based Partitioning

internal nodes - elements - external nodes

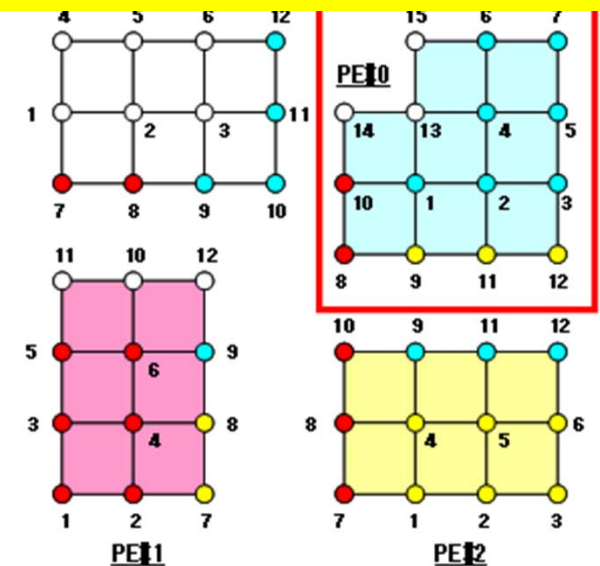
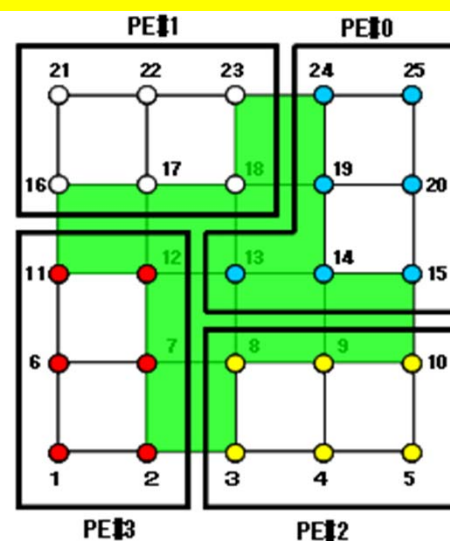
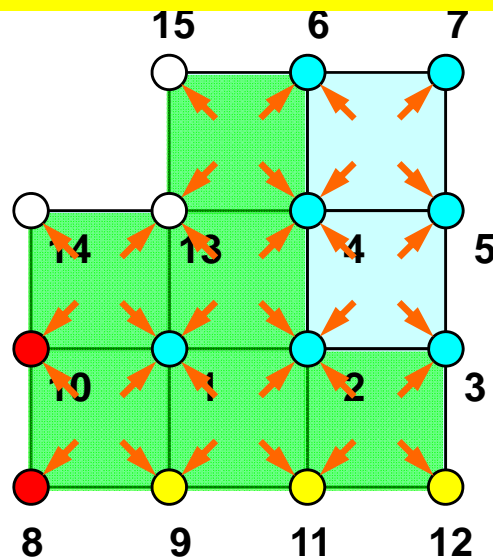


# Node-based Partitioning

internal nodes - elements - external nodes

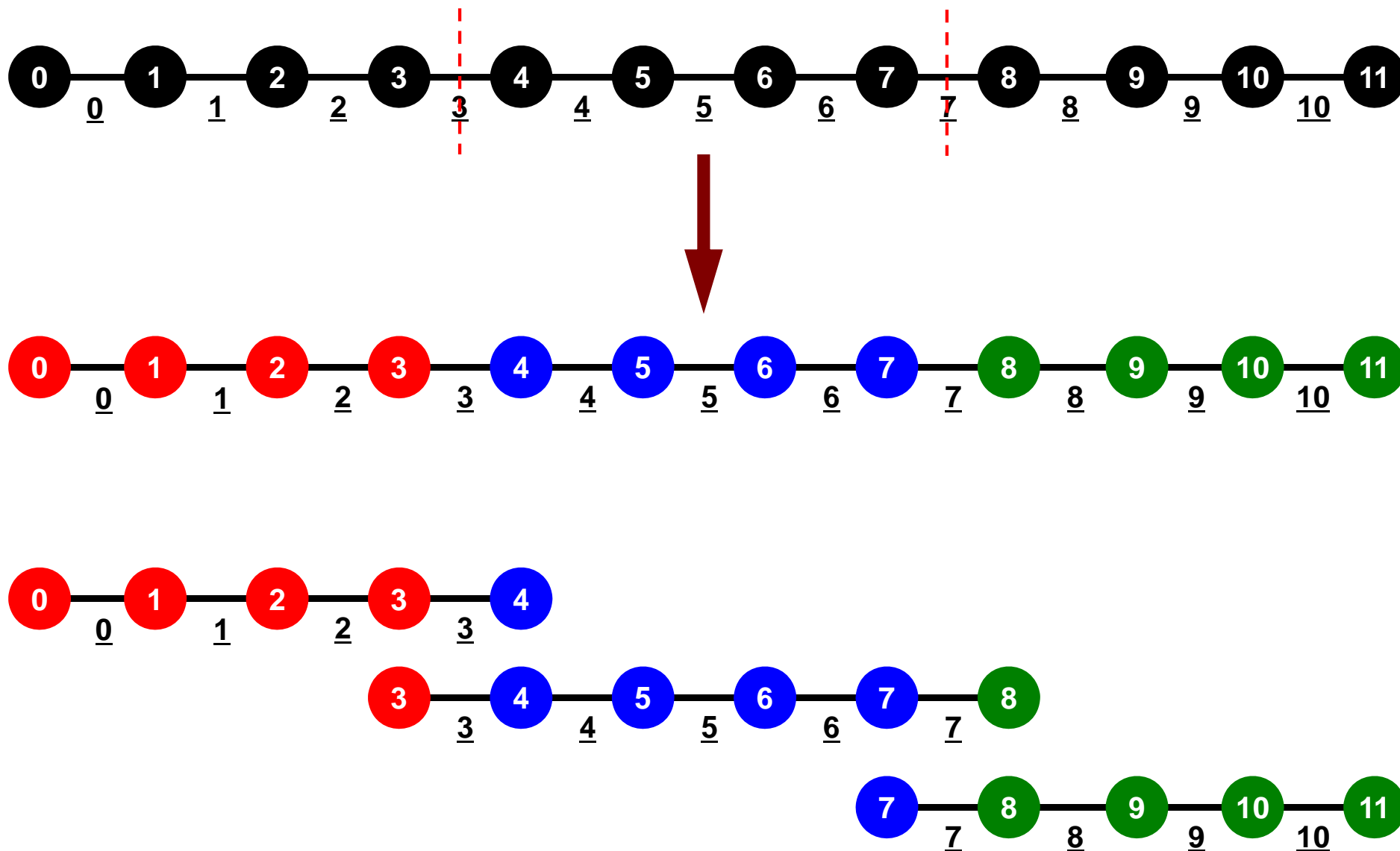


- Partitioned nodes themselves (Internal Nodes) 内点
- Elements which include Internal Nodes 内点を含む要素
- External Nodes included in the Elements 外点  
in overlapped region among partitions.
- Info of External Nodes are required for completely local element-based operations on each processor.



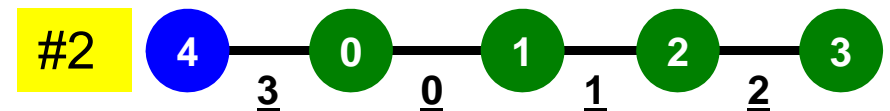
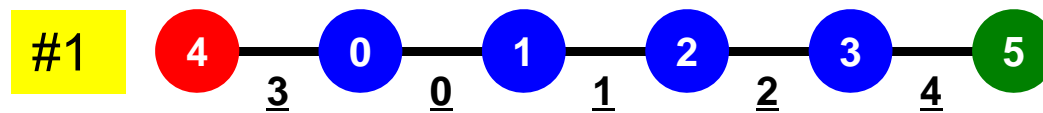
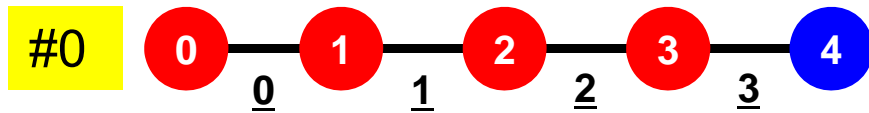


# 一次元問題: 11要素, 12節点, 3領域



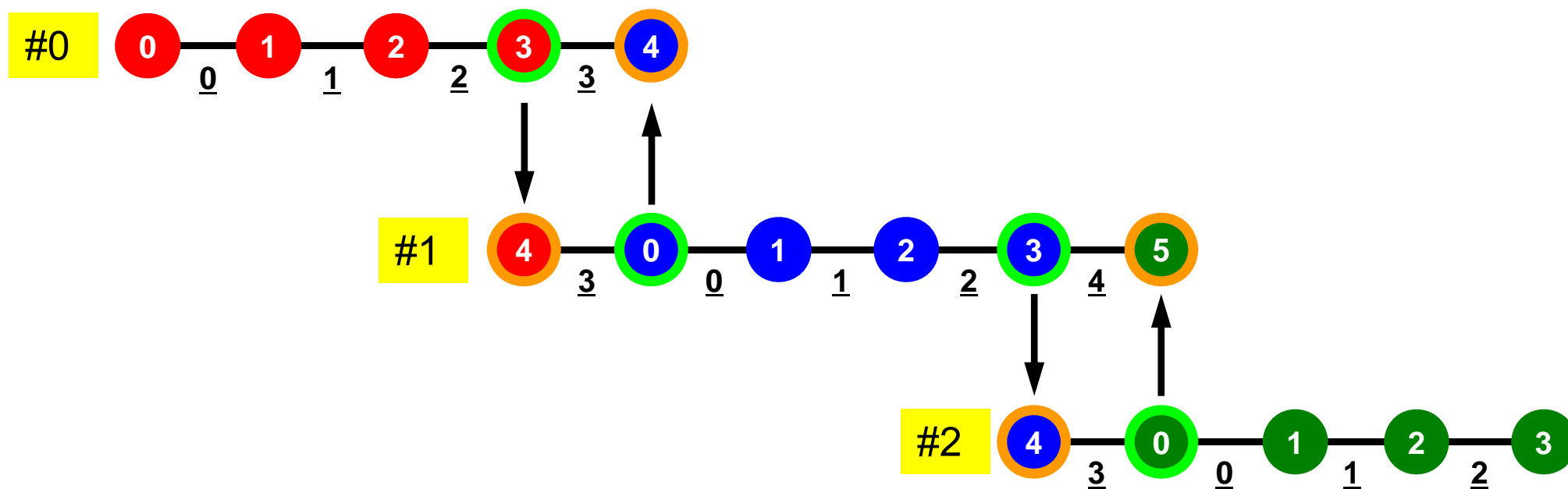
# 一次元問題: 11要素, 12節点, 3領域

局所番号: 節点・要素とも0からふる

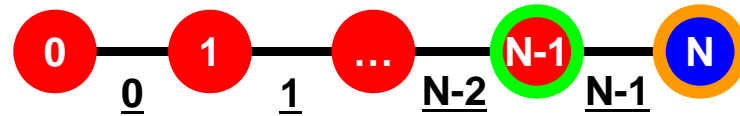


# 一次元問題：11要素，12節点，3領域

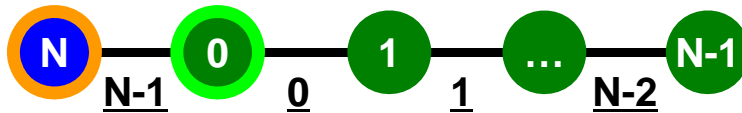
外点・境界点



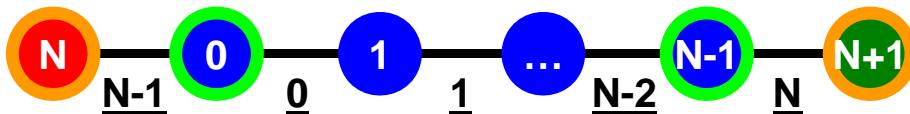
# 一次元問題：一般的な局所番号の付け方



#0: N+1節点, N要素



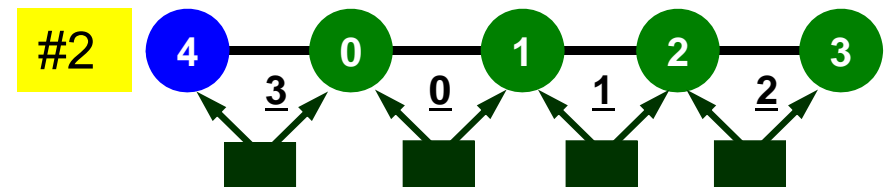
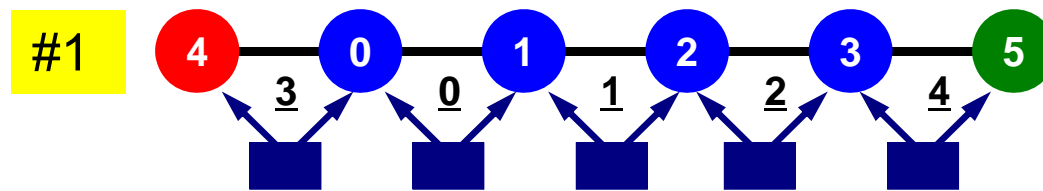
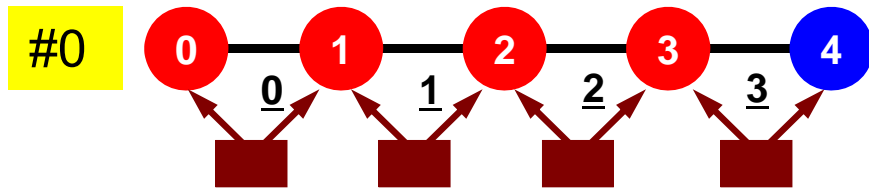
#PETot-1: N+1節点, N要素



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

前処理: 対角スケーリング

# 前処理, ベクトル定数倍の加減

## 局所的な計算(内点のみ)が可能⇒並列処理

```
/*  
/-- {z} = [Minv]{r}  
*/  
for (i=0; i<N; i++) {  
    W[Z][i] = W[DD][i] * W[R][i];  
}
```

```
/*  
/-- {x} = {x} + ALPHA*{p}  
// {r} = {r} - ALPHA*{q}  
*/  
for (i=0; i<N; i++) {  
    U[i] += Alpha * W[P][i];  
    W[R][i] -= Alpha * W[Q][i];  
}
```

0
1
2
3
4
5
6
7
8
9
10
11

# 内積

全体で和をとる必要がある⇒通信？

```
/*  
/-- ALPHA= RHO / {p} {q}  
*/  
C1 = 0.0;  
for (i=0; i<N; i++) {  
    C1 += W[P][i] * W[Q][i];  
}  
  
Alpha = Rho / C1;
```

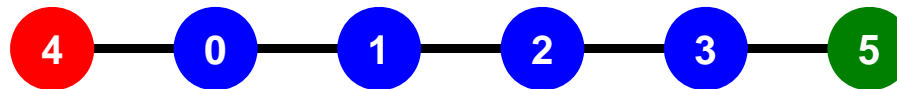
0
1
2
3
4
5
6
7
8
9
10
11



# 行列ベクトル積

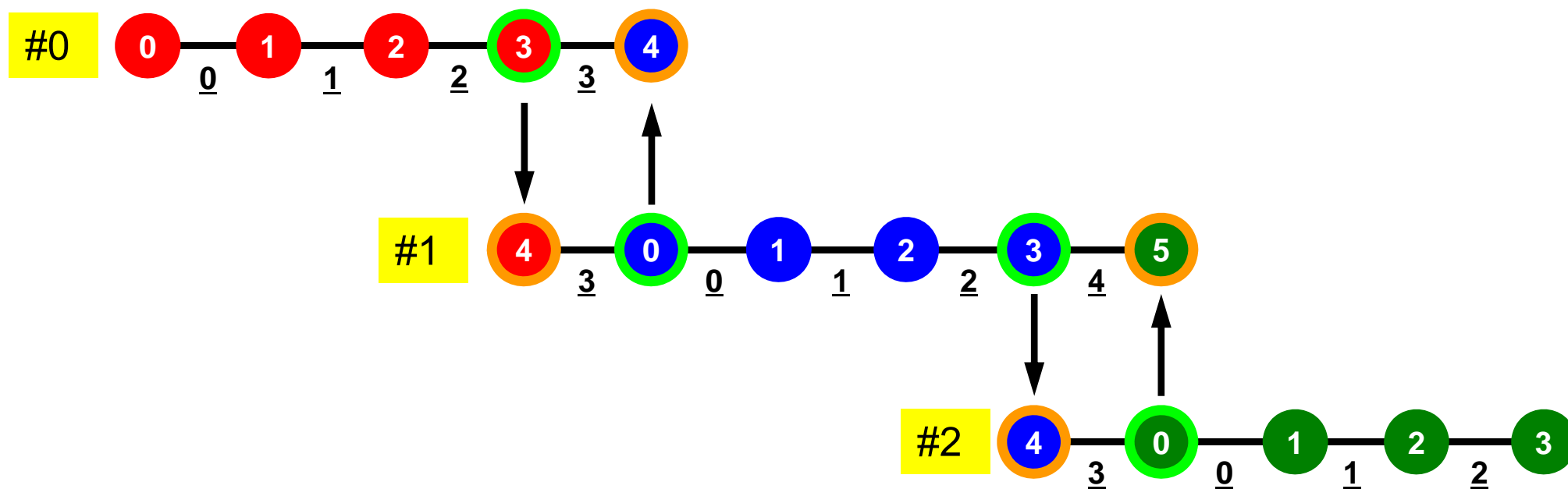
外点の値(最新のp)が必要 $\Rightarrow$ 1対1通信

```
/*  
/-- {q} = [A] {p}  
*/  
for (i=0; i<N; i++) {  
    W[Q][i] = Diag[i] * W[P][i];  
    for (j=Index[i]; j<Index[i+1]; j++) {  
        W[Q][i] += AMat[j]*W[P][Item[j]];  
    }  
}
```



# 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

0																			
	1																		
		2																	
			3																

0
1
2
3

0
1
2
3

				4															
					5														
						6													
							7												

4
5
6
7

=

4
5
6
7

								8											
									9										
										10									
											11								

8
9
10
11

8
9
10
11

# 行列ベクトル積：ローカルに計算実施可能

0				
	1			
		2		
			3	

0
1
2
3

0
1
2
3

	0			
		1		
			2	
				3

0
1
2
3

=

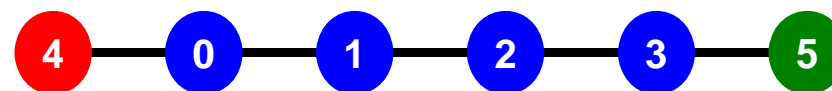
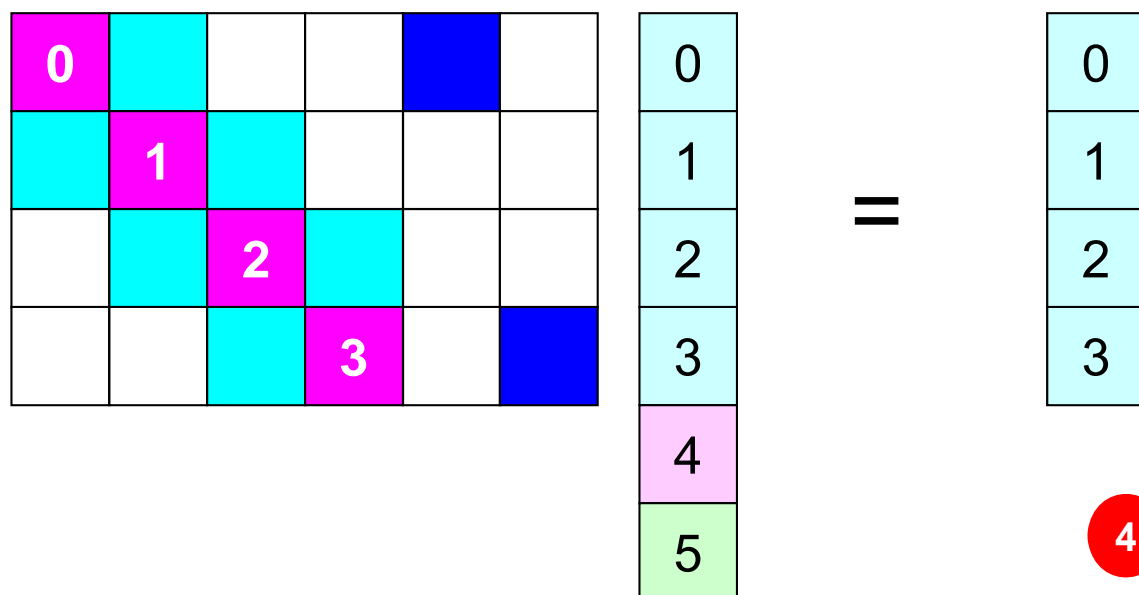
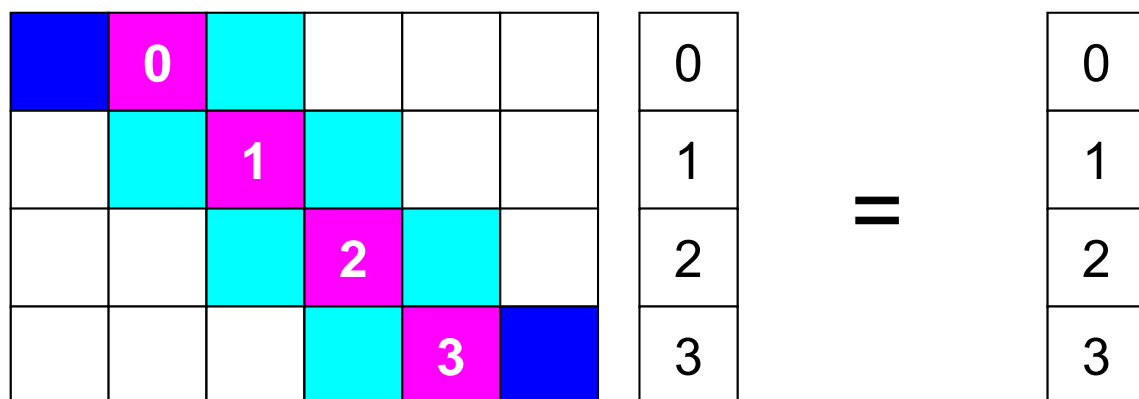
0
1
2
3

	0			
		1		
			2	
				3

0
1
2
3

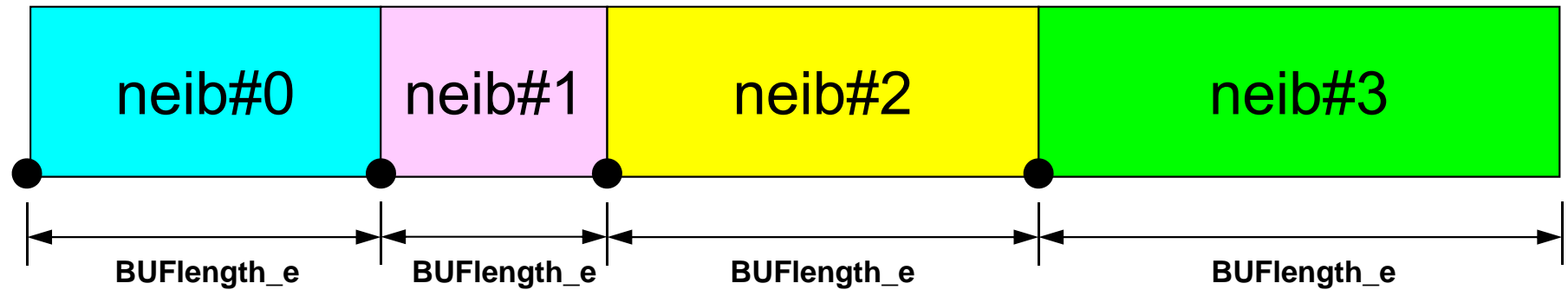
0
1
2
3

# 行列ベクトル積：ローカル計算 #1



# 送信 (MPI\_Isend/Irecv/Waitall)

SendBuf



export\_index[0]      export\_index[1]      export\_index[2]      export\_index[3]      export\_index[4]

export\_index[neib] ~ export\_index[neib+1]-1 番目の export\_item が neib 番目の隣接領域に送信される

```
for (neib=0; neib<NeibPETot;neib++){
  for (k=export_index[neib];k<export_index[neib+1];k++){
    kk= export_item[k];
    SendBuf[k]= VAL[kk];           送信バッファへの代入
  }
}
```

```
for (neib=0; neib<NeibPETot; neib++){
  tag= 0;
  iS_e= export_index[neib];
  iE_e= export_index[neib+1];
  BUFlength_e= iE_e - iS_e

  ierr= MPI_Isend
    (&SendBuf[iS_e], BUFlength_e, MPI_DOUBLE, NeibPE[neib], 0,
     MPI_COMM_WORLD, &ReqSend[neib])
}
```

```
MPI_Waitall(NeibPETot, ReqSend, StatSend);
```

# 受信 (MPI\_Irecv/Irecv/Waitall)

```

for (neib=0; neib<NeibPETot; neib++){
  tag= 0;
  iS_i= import_index[neib];
  iE_i= import_index[neib+1];
  BUFlength_i= iE_i - iS_i

  ierr= MPI_Irecv
    (&RecvBuf[iS_i], BUFlength_i, MPI_DOUBLE, NeibPE[neib], 0,
     MPI_COMM_WORLD, &ReqRecv[neib])
}

```

```
MPI_Waitall(NeibPETot, ReqRecv, StatRecv);
```

```

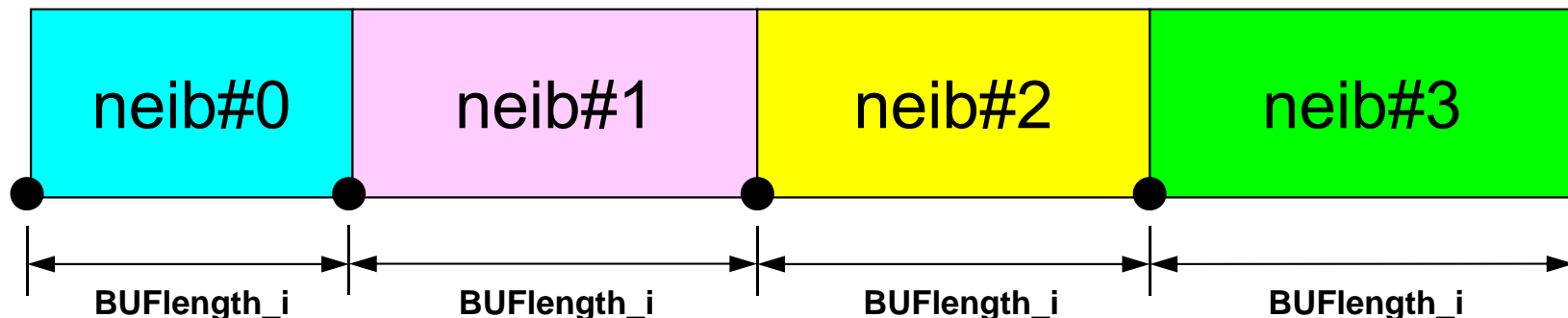
for (neib=0; neib<NeibPETot;neib++){
  for (k=import_index[neib];k<import_index[neib+1];k++){
    kk= import_item[k];
    VAL[kk]= RecvBuf[k];
  }
}

```

受信バッファからの代入

import\_index[neib] ~ import\_index[neib+1]-1番目のimport\_itemがneib番目の隣接領域から受信される

RecvBuf



import\_index[0]   import\_index[1]   import\_index[2]   import\_index[3]   import\_index[4]



- 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.c (1/11)

## 諸変数

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

```
int main(int argc, char **argv) {
```

MPIを使用するときの「おまじない」

```
int NE, N, NP, NPLU, IterMax, NEg, Ng, errno;
double dX, Resid, Eps, Area, QV, COND, QN;
double X1, X2, DL, Ck; double *PHI, *Rhs, *X, *Diag, *AMat;
double *R, *Z, *Q, *P, *DD;
int *Index, *Item, *Icelnod;
double Kmat[2][2], Emat[2][2];
```

```
int i, j, in1, in2, k, icel, k1, k2, jS;
int iter, nr, neib;
FILE *fp;
double BNorm2, Rho, Rho1=0.0, C1, Alpha, Beta, DNorm2;
int PETot, MyRank, kk, is, ir, len_s, len_r, tag;
int NeibPETot, BufLength, NeibPE[2];
```

```
int import_index[3], import_item[2];
int export_index[3], export_item[2];
double SendBuf[2], RecvBuf[2];
```

```
double BNorm20, Rho0, C10, DNorm20;
double StartTime, EndTime;
int ierr = 1;
```

```
MPI_Status *StatSend, *StatRecv;
MPI_Request *RequestSend, *RequestRecv;
```

# プログラム: 1d.c (2/11)

## 制御データ読み込み

```
/*
// +-----+
// | INIT. |
// +-----+
//=== */
```

```
/*
//-- CONTROL data
*/
```

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

MPI初期化 : 必須  
 全プロセス数 : PETot  
 自分のランク番号 (0~PETot-1) : MyRank

```
if(MyRank == 0) {
  fp = fopen("input.dat", "r");
  assert(fp != NULL);
  fscanf(fp, "%d", &NEg);
  fscanf(fp, "%lf %lf %lf %lf", &dX, &QV, &Area, &COND);
  fscanf(fp, "%d", &IterMax);
  fscanf(fp, "%lf", &Eps);
  fclose(fp);
}
```

```
ierr = MPI_Bcast(&NEg, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&dX, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&QV, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Area, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&COND, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

# プログラム: 1d.c (2/11)

## 制御データ読み込み

```
/*
// +-----+
// | INIT. |
// +-----+
//=== */
```

```
/*
//-- CONTROL data
*/
```

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

MPI初期化 : 必須  
 全プロセス数 : PETot  
 自分のランク番号 (0~PETot-1) : MyRank

```
if (MyRank == 0) {
  fp = fopen("input.dat", "r");
  assert(fp != NULL);
  fscanf(fp, "%d", &NEg);
  fscanf(fp, "%lf %lf %lf %lf", &dX, &QV, &Area, &COND);
  fscanf(fp, "%d", &IterMax);
  fscanf(fp, "%lf", &Eps);
  fclose(fp);
}
```

MyRank=0のとき制御データを読み込む

NEg : 「全」要素数

```
ierr = MPI_Bcast(&NEg, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&dX, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&QV, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Area, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&COND, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

# プログラム: 1d.c (2/11)

## 制御データ読み込み

```
/*
// +-----+
// |  INIT.  |
// +-----+
//=== */
```

```
/*
//-- CONTROL data
*/
```

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

MPI初期化 : 必須  
 全プロセス数 : PETot  
 自分のランク番号 (0~PETot-1) : MyRank

```
if(MyRank == 0) {
  fp = fopen("input.dat", "r");
  assert(fp != NULL);
  fscanf(fp, "%d", &NEg);
  fscanf(fp, "%lf %lf %lf %lf", &dX, &QV, &Area, &COND);
  fscanf(fp, "%d", &IterMax);
  fscanf(fp, "%lf", &Eps);
  fclose(fp);
}
```

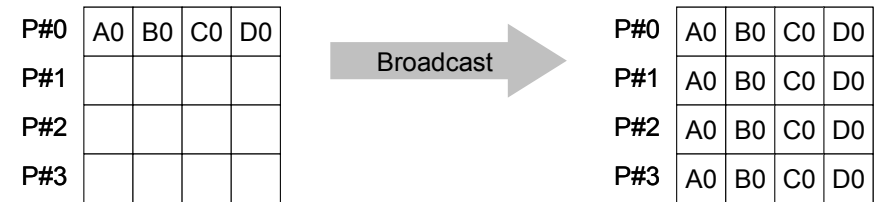
MyRank=0のとき制御データを読み込む

Neg : 「全」要素数

```
ierr = MPI_Bcast(&NEg, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&dX, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&QV, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Area, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&COND, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

0番プロセスから各プロセスにデータ送信

# MPI\_Bcast



- グループ(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。
- **MPI\_Bcast (buffer, count, datatype, root, comm)**
  - **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 コミュニケーター(通信グループ)を指定する

# プログラム: 1d.c (3/11)

## 局所分散メッシュデータ

```
/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng : 総節点数
N = Ng / PETot;       N  : 局所節点数

nr = Ng - N*PETot;    NgがPETotで割り切れない場合
if(MyRank < nr) N++;

NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N  ;}

/*
/-- Arrays
*/
PHI  = calloc(NP, sizeof(double));
Diag = calloc(NP, sizeof(double));
AMat = calloc(2*NP-2, sizeof(double));
Rhs  = calloc(NP, sizeof(double));
Index= calloc(NP+1, sizeof(int));
Item = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));
```

# プログラム: 1d.c (3/11)

## 局所分散メッシュデータ, 各要素→一様

```

/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng : 総節点数
N = Ng / PETot;       N : 局所節点数 (内点)

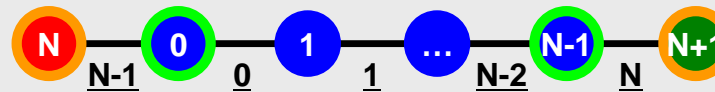
nr = Ng - N*PETot;    NgがPETotで割り切れない場合
if(MyRank < nr) N++;

NE= N - 1 + 2;        局所要素数
NP= N + 2;            内点+外点 (局所総節点数)
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N ;}

/*
/-- Arrays
*/
PHI  = calloc(NP, sizeof(double));
Diag = calloc(NP, sizeof(double));
AMat = calloc(2*NP-2, sizeof(double));
Rhs  = calloc(NP, sizeof(double));
Index= calloc(NP+1, sizeof(int));
Item = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```



一般の領域:  
N+2節点, N+1要素



# プログラム: 1d.c (3/11)

## 局所分散メッシュデータ, 各要素→一様

```

/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng : 総節点数
N = Ng / PETot;       N : 局所節点数 (内点)

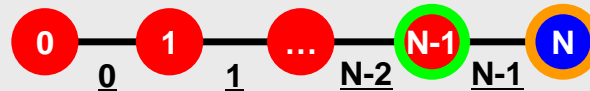
nr = Ng - N*PETot;    NgがPETotで割り切れない場合
if(MyRank < nr) N++;

NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N ;}

/*
/-- Arrays
*/
PHI = calloc(NP, sizeof(double));
Diag = calloc(NP, sizeof(double));
AMat = calloc(2*NP-2, sizeof(double));
Rhs = calloc(NP, sizeof(double));
Index= calloc(NP+1, sizeof(int));
Item = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```



#0: N+1節点, N要素

# プログラム: 1d.c (3/11)

## 局所分散メッシュデータ, 各要素→一様

```

/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng : 総節点数
N = Ng / PETot;       N  : 局所節点数 (内点)

nr = Ng - N*PETot;    NgがPETotで割り切れない場合
if(MyRank < nr) N++;

NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N  ;}

```



#PETot-1: N+1節点, N要素

```

/*
/-- Arrays
*/
PHI  = calloc(NP, sizeof(double));
Diag = calloc(NP, sizeof(double));
AMat = calloc(2*NP-2, sizeof(double));
Rhs  = calloc(NP, sizeof(double));
Index= calloc(NP+1, sizeof(int));
Item = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```

# プログラム: 1d.c (3/11)

## 局所分散メッシュデータ

```

/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng : 総節点数
N = Ng / PETot;       N  : 局所節点数 (内点)

nr = Ng - N*PETot;    NgがPETotで割り切れない場合
if(MyRank < nr) N++;

NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N  ;}

```

```

/*
/-- Arrays
*/

```

```

PHI  = calloc(NP, sizeof(double));
Diag = calloc(NP, sizeof(double));
AMat = calloc(2*NP-2, sizeof(double));
Rhs  = calloc(NP, sizeof(double));
Index= calloc(NP+1, sizeof(int));
Item = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```

NでなくNPで配列を定義している点に注意

# プログラム: 1d.c(4/11)

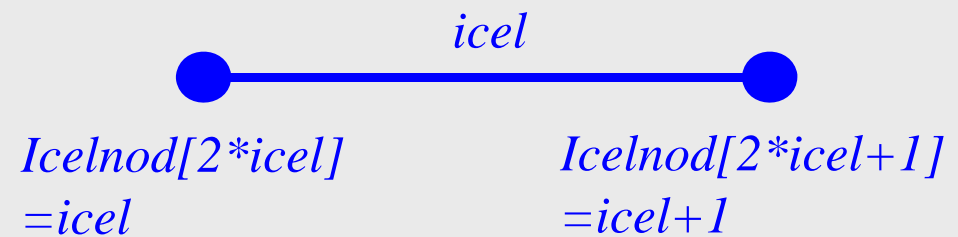
## 配列初期化, 要素~節点

```
for (i=0; i<NP; i++)    U[i] = 0.0;
for (i=0; i<NP; i++)    Diag[i] = 0.0;
for (i=0; i<NP; i++)    Rhs[i] = 0.0;
for (k=0; k<2*NP-2; k++) AMat[k] = 0.0;
```

```
for (i=0; i<3; i++) import_index[i]= 0;
for (i=0; i<3; i++) export_index[i]= 0;
for (i=0; i<2; i++) import_item[i]= 0;
for (i=0; i<2; i++) export_item[i]= 0;
```

```
for (icel=0; icel<NE; icel++) {
    Icelnod[2*icel] = icel;
    Icelnod[2*icel+1] = icel+1;
}
```

```
if (PETot>1) {
    if (MyRank==0) {
        icel = NE-1;
        Icelnod[2*icel] = N-1;
        Icelnod[2*icel+1] = N;
    } else if (MyRank==PETot-1) {
        icel = NE-1;
        Icelnod[2*icel] = N;
        Icelnod[2*icel+1] = 0;
    } else {
        icel = NE-2;
        Icelnod[2*icel] = N;
        Icelnod[2*icel+1] = 0;
        icel = NE-1;
        Icelnod[2*icel] = N-1;
        Icelnod[2*icel+1] = N+1;
    }
}
```



# プログラム: 1d.c(4/11)

## 配列初期化, 要素~節点

```
for (i=0; i<NP; i++)    U[i] = 0.0;
for (i=0; i<NP; i++)    Diag[i] = 0.0;
for (i=0; i<NP; i++)    Rhs[i] = 0.0;
for (k=0; k<2*NP-2; k++) AMat[k] = 0.0;
```

```
for (i=0; i<3; i++) import_index[i]= 0;
for (i=0; i<3; i++) export_index[i]= 0;
for (i=0; i<2; i++) import_item[i]= 0;
for (i=0; i<2; i++) export_item[i]= 0;
```

```
for (icel=0; icel<NE; icel++) {
    Icelnod[2*icel  ]= icel;
    Icelnod[2*icel+1]= icel+1;
}
```

```
if (PETot>1) {
    if (MyRank==0) {
        icel= NE-1;
        Icelnod[2*icel  ]= N-1;
        Icelnod[2*icel+1]= N;
    } else if (MyRank==PETot-1) {
        icel= NE-1;
        Icelnod[2*icel  ]= N;
        Icelnod[2*icel+1]= 0;
    } else {
        icel= NE-2;
        Icelnod[2*icel  ]= N;
        Icelnod[2*icel+1]= 0;
        icel= NE-1;
        Icelnod[2*icel  ]= N-1;
        Icelnod[2*icel+1]= N+1;
    }
}
```

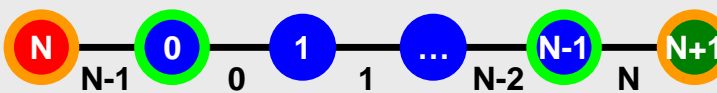
「0-1」の要素を「0」とする



#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.c (5/11)

## Index定義

```
Kmat[0][0] = +1.0;
Kmat[0][1] = -1.0;
Kmat[1][0] = -1.0;
Kmat[1][1] = +1.0;
```

```
/*
// |-----|
// | CONNECTIVITY |
// |-----|
*/
```

```
for (i=0; i<N+1; i++) Index[i] = 2;
for (i=N+1; i<NP+1; i++) Index[i] = 1;
```

```
Index[0] = 0;
if (MyRank == 0) Index[1] = 1;
if (MyRank == PETot-1) Index[N] = 1;
```

```
for (i=0; i<NP; i++) {
  Index[i+1] = Index[i+1] + Index[i];
}
```

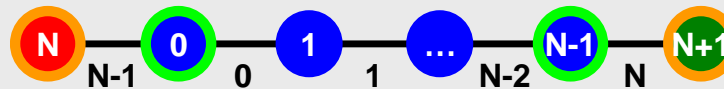
```
NPLU = Index[NP];
```



#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.c (6/11)

## Item定義

```

for (i=0; i<N; i++) {
  jS = Index[i];
  if ((MyRank==0) && (i==0)) {
    Item[jS] = i+1;
  } else if ((MyRank==PETot-1) && (i==N-1)) {
    Item[jS] = i-1;
  } else {
    Item[jS] = i-1;
    Item[jS+1] = i+1;
    if (i==0) { Item[jS] = N; }
    if (i==N-1) { Item[jS+1] = N+1; }
    if ((MyRank==0) && (i==N-1)) { Item[jS+1] = N; }
  }
}

```



#0: N+1節点, N要素

```

i =N;
jS= Index[i];
if (MyRank==0) {
  Item[jS]= N-1;
} else {
  Item[jS]= 0;
}

```

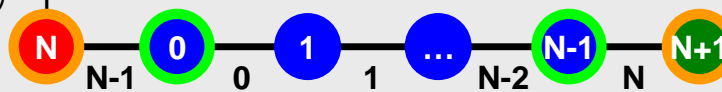


#PETot-1: N+1節点, N要素

```

i =N+1;
jS= Index[i];
if ((MyRank!=0) && (MyRank!=PETot-1)) {
  Item[jS]= N-1;
}

```



一般の領域:  
N+2節点, N+1要素

# プログラム: 1d.c (7/11)

## 通信テーブル定義

```

/*
//-- COMMUNICATION
*/
NeibPETot = 2;
if(MyRank == 0) NeibPETot = 1;
if(MyRank == PETot-1) NeibPETot = 1;
if(PETot == 1) NeibPETot = 0;

```

```

NeibPE[0] = MyRank - 1;
NeibPE[1] = MyRank + 1;

```

```

if(MyRank == 0) NeibPE[0] = MyRank + 1;
if(MyRank == PETot-1) NeibPE[0] = MyRank - 1;

```



#0: N+1節点, N要素

```

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

```



#PETot-1: N+1節点, N要素

```

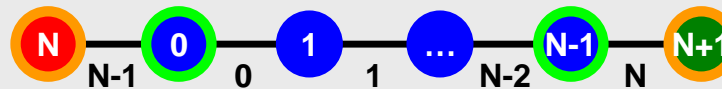
export_index[1]=1;
export_index[2]=2;
export_item[0]= 0;
export_item[1]= N-1;

```

```

if(MyRank == 0) import_item[0]=N;
if(MyRank == 0) export_item[0]=N-1;

```



一般の領域:  
N+2節点, N+1要素

```

BufLength = 1;

```

```

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\_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。

- MPI\_Isend**

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

- |                   |    |   |  |
|-------------------|----|---|--|
| - <u>sendbuf</u>  | 任意 | I | 送信バッファの先頭アドレス,   |
| - <u>count</u>    | 整数 | I | メッセージのサイズ  |
| - <u>datatype</u> | 整数 | I | メッセージのデータタイプ   |
| - <u>dest</u>     | 整数 | I | 宛先プロセスのアドレス(ランク)   |
| - <u>tag</u>      | 整数 | I | メッセージタグ, 送信メッセージの種類を区別するときに使用。<br>通常は「0」でよい。同じメッセージタグ番号同士で通信。              |
| - <u>comm</u>     | 整数 | I | コミュニケータを指定する   |
| - <u>request</u>  | 整数 | O | 通信識別子。MPI_Waitallで使用。<br>(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)) |

# 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。

- MPI\_Irecv**

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

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

# MPI\_Waitall

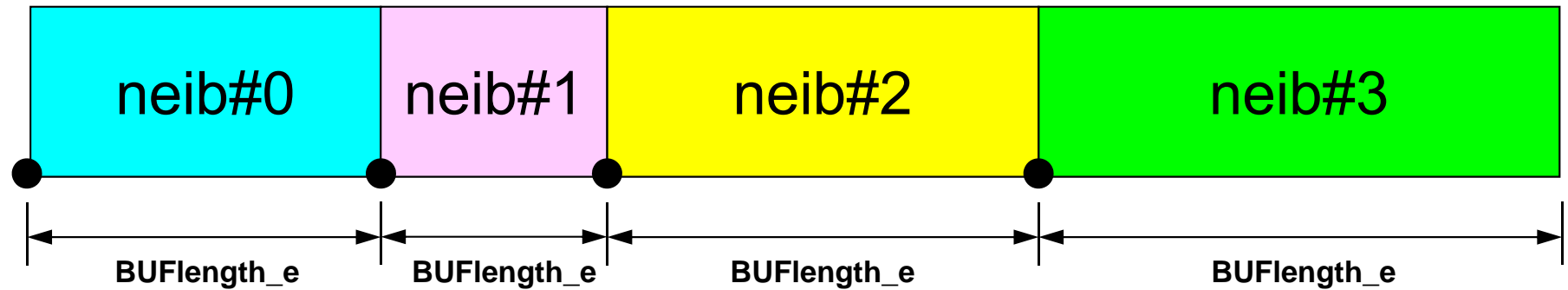
- 1対1非ブロッキング通信関数である「MPI\_Isend」と「MPI\_Irecv」を使用した場合、プロセスの同期を取るのに使用する。
- 送信時はこの「MPI\_Waitall」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI\_Waitall」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば、「MPI\_Isend」と「MPI\_Irecv」を同時に同期してもよい。
  - 「MPI\_Isend/Irecv」で同じ通信識別子を使用すること
- 「MPI\_Barrier」と同じような機能であるが、代用はできない。
  - 実装にもよるが、「request」、「status」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。
- **MPI\_Waitall (count, request, status)**
  - **count**      整数      I      同期する必要のある「MPI\_ISEND」, 「MPI\_RECV」呼び出し数。
  - **request**    整数      I/O    通信識別子。「MPI\_ISEND」, 「MPI\_Irecv」で利用した識別子名に対応。(配列サイズ: (count))
  - **status**      整数      0      状況オブジェクト配列(配列サイズ: (MPI\_STATUS\_SIZE, count))  
MPI\_STATUS\_SIZE: “mpif.h”, “mpi.h”で定められる  
パラメータ

# 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-1
- 「境界点」番号
  - export\_item[k], k= 0, export\_index[NeibPETot]-1
- それぞれの送信相手に送るメッセージ
  - SendBuf[k], k= 0, export\_index[NeibPETot]-1

# 送信 (MPI\_Isend/Irecv/Waitall)

SendBuf



export\_index[0]      export\_index[1]      export\_index[2]      export\_index[3]      export\_index[4]

export\_index[neib] ~ export\_index[neib+1]-1 番目の export\_item が neib 番目の隣接領域に送信される

```
for (neib=0; neib<NeibPETot;neib++){
  for (k=export_index[neib];k<export_index[neib+1];k++){
    kk= export_item[k];
    SendBuf[k]= VAL[kk];
  }
}
```

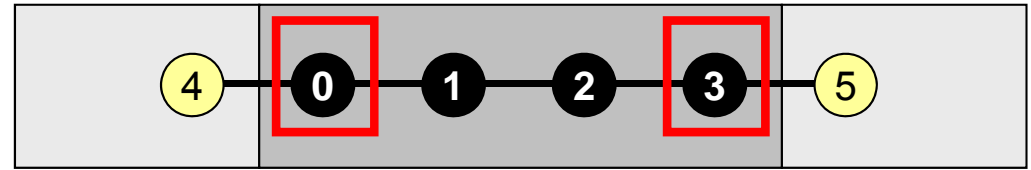
送信バッファへの代入

```
for (neib=0; neib<NeibPETot; neib++){
  tag= 0;
  iS_e= export_index[neib];
  iE_e= export_index[neib+1];
  BUFlength_e= iE_e - iS_e

  ierr= MPI_Isend
    (&SendBuf[iS_e], BUFlength_e, MPI_DOUBLE, NeibPE[neib], 0,
     MPI_COMM_WORLD, &ReqSend[neib])
}
```

```
MPI_Waitall(NeibPETot, ReqSend, StatSend);
```

# 送信：一次元問題



- 受信相手
  - NeibPETot, NeibPE[neib]
    - NeibPETot=2, NeibPE[0]= my\_rank-1, NeibPE[1]= 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-1
    - export\_index[0]=0, export\_index[1]= 1, export\_index[2]= 2
- 「境界点」番号
  - export\_item[k], k= 0, export\_index[NeibPETot]-1
    - export\_item[0]= 0, export\_item[1]= N-1
- それぞれの送信相手に送るメッセージ
  - SendBuf[k], k= 0, export\_index[NeibPETot]-1
    - SendBuf[0]= VAL[0], SendBuf[1]= VAL[N-1]

# 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-1
- 「外点」番号
  - import\_item[k], k= 0, import\_index[NeibPETot]-1
- それぞれの受信相手から受け取るメッセージ
  - RecvBuf[k], k= 0, import\_index[NeibPETot]-1

# 受信 (MPI\_Irecv/Irecv/Waitall)

```

for (neib=0; neib<NeibPETot; neib++){
  tag= 0;
  iS_i= import_index[neib];
  iE_i= import_index[neib+1];
  BUFlength_i= iE_i - iS_i

  ierr= MPI_Irecv
    (&RecvBuf[iS_i], BUFlength_i, MPI_DOUBLE, NeibPE[neib], 0,
     MPI_COMM_WORLD, &ReqRecv[neib])
}

```

```
MPI_Waitall(NeibPETot, ReqRecv, StatRecv);
```

```

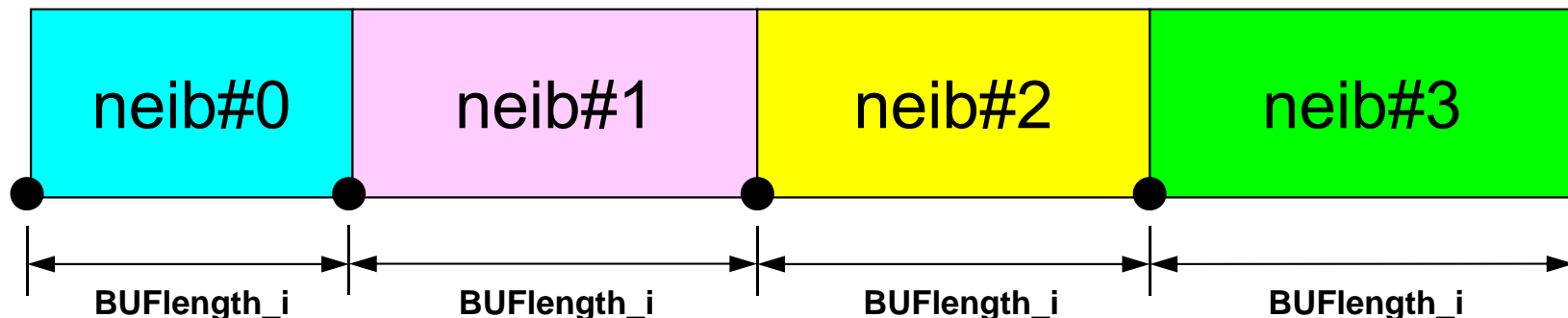
for (neib=0; neib<NeibPETot;neib++){
  for (k=import_index[neib];k<import_index[neib+1];k++){
    kk= import_item[k];
    VAL[kk]= RecvBuf[k];
  }
}

```

受信バッファからの代入

import\_index[neib] ~ import\_index[neib+1]-1番目のimport\_itemがneib番目の隣接領域から受信される

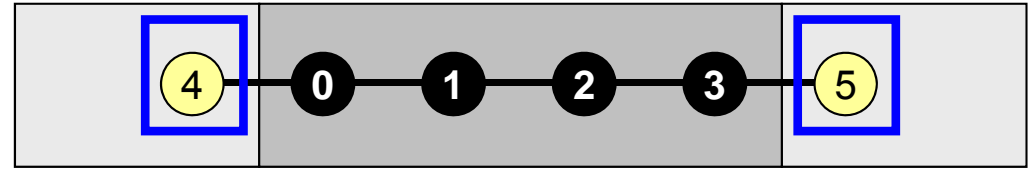
RecvBuf



import\_index[0]   import\_index[1]   import\_index[2]   import\_index[3]   import\_index[4]



# 受信:一次元問題

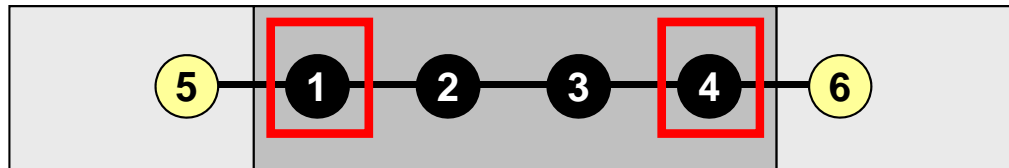


- 受信相手
  - NeibPETot, NeibPE[neib]
    - NeibPETot=2, NeibPE[0]= my\_rank-1, NeibPE[1]= my\_rank+1
- それぞれの受信相手から受け取るメッセージサイズ
  - import\_index[neib], neib= 0, NeibPETot-1
    - import\_index[0]=0, import\_index[1]= 1, import\_index[2]= 2
- 「外点」番号
  - import\_item[k], k= 0, import\_index[NeibPETot]-1
    - import\_item[0]= N, import\_item[1]= N+1
- それぞれの受信相手から受け取るメッセージ
  - RECVbuf[k], k= 0, import\_index[NeibPETot]-1
    - VAL[N]=RecvBuf[0], VAL[N+1]=RecvBuf[1]

VAL[4]=RecvBuf[0]

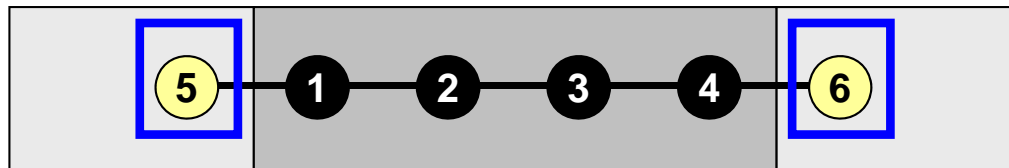
VAL[5]=RecvBuf[1]

# 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)

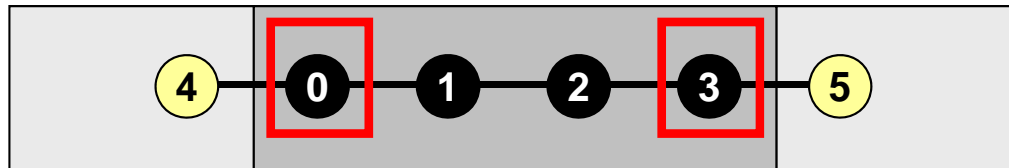
```
NEIBPETOT= 2
NEIBPE(1)= my_rank - 1
NEIBPE(2)= my_rank + 1
```

```
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
```

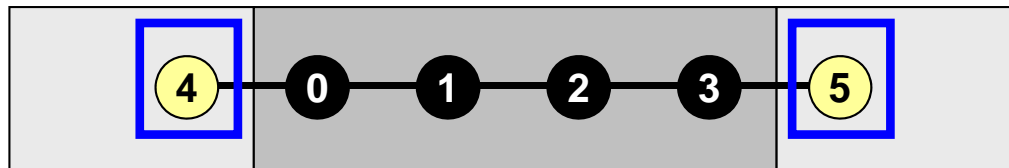
```
if (my_rank.eq.0) then
  import_item (1)= N+1
  export_item (1)= N
  NEIBPE(1)= my_rank+1
endif
```

# 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
```

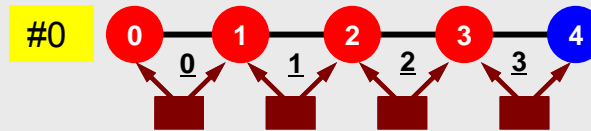
```
if (my_rank.eq.0) then
  import_item [0]= N
  export_item [0]= N-1
  NEIBPE[0]= my_rank+1
endif
```

# プログラム: 1d.c (8/11)

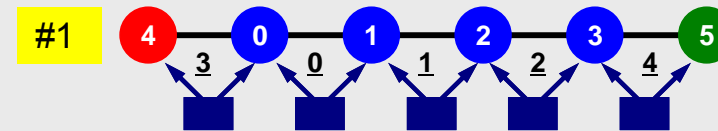
全体マトリクス生成: 1CPUのときと全く同じ: 各要素→一様

```
/*
//
// | MATRIX assemble |
//
*/
```

```
for (icel=0; icel<NE; icel++) {
  in1= Icelnod[2*icel];
  in2= Icelnod[2*icel+1];
  DL = dX;
```



```
  Ck= Area*COND/DL;
  Emat[0][0]= Ck*Kmat[0][0];
  Emat[0][1]= Ck*Kmat[0][1];
  Emat[1][0]= Ck*Kmat[1][0];
  Emat[1][1]= Ck*Kmat[1][1];
```



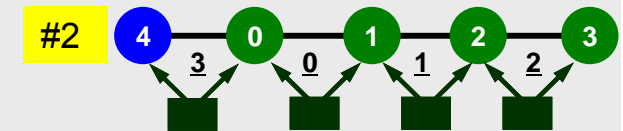
```
  Diag[in1]= Diag[in1] + Emat[0][0];
  Diag[in2]= Diag[in2] + Emat[1][1];
```

```
  if ((MyRank==0)&&(icel==0)) {
    k1=Index[in1];
  }else {k1=Index[in1]+1;}
```

```
  k2=Index[in2];
```

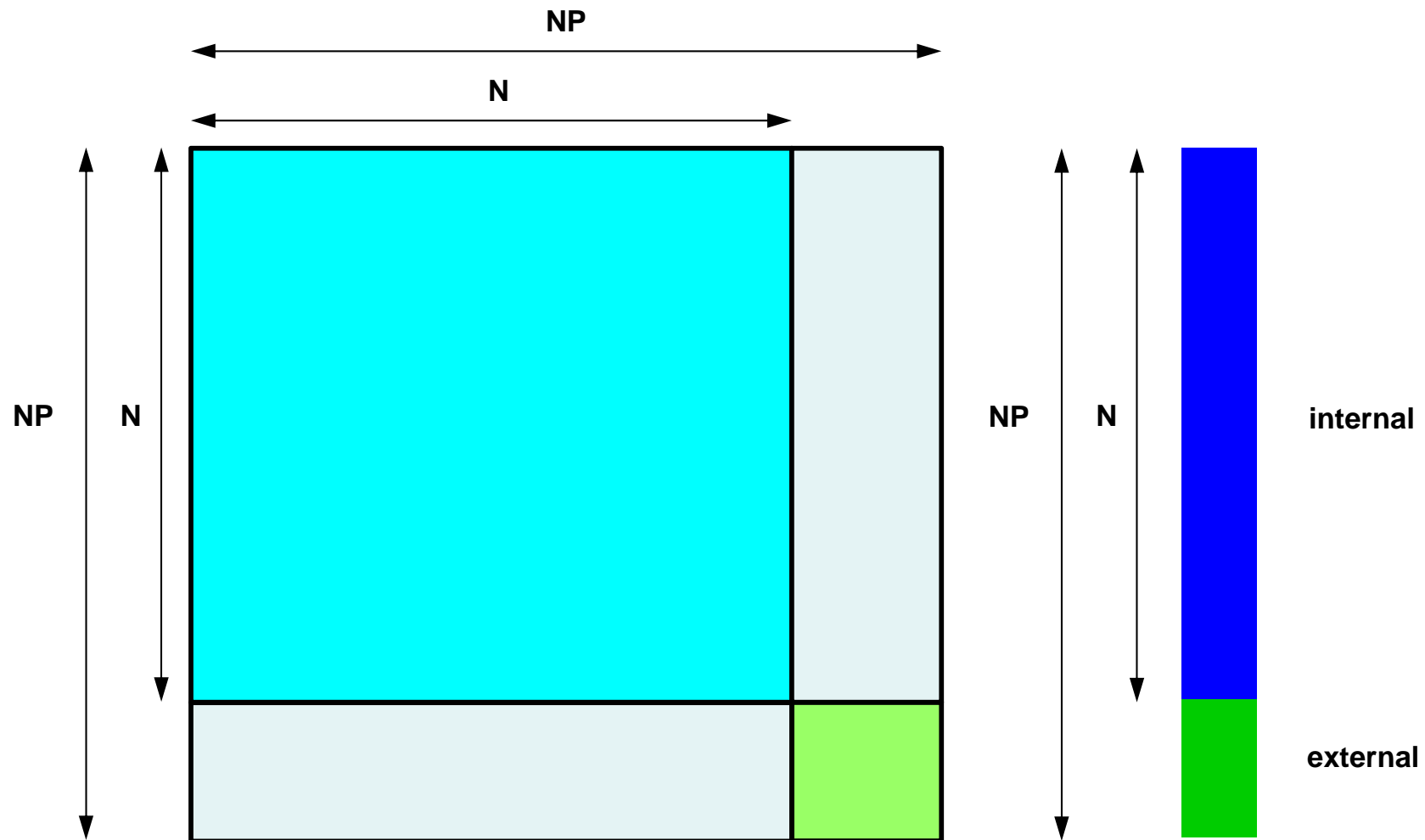
```
  AMat[k1]= AMat[k1] + Emat[0][1];
  AMat[k2]= AMat[k2] + Emat[1][0];
```

```
  QN= 0.5*QV*Area*dX;
  Rhs[in1]= Rhs[in1] + QN;
  Rhs[in2]= Rhs[in2] + QN;
```

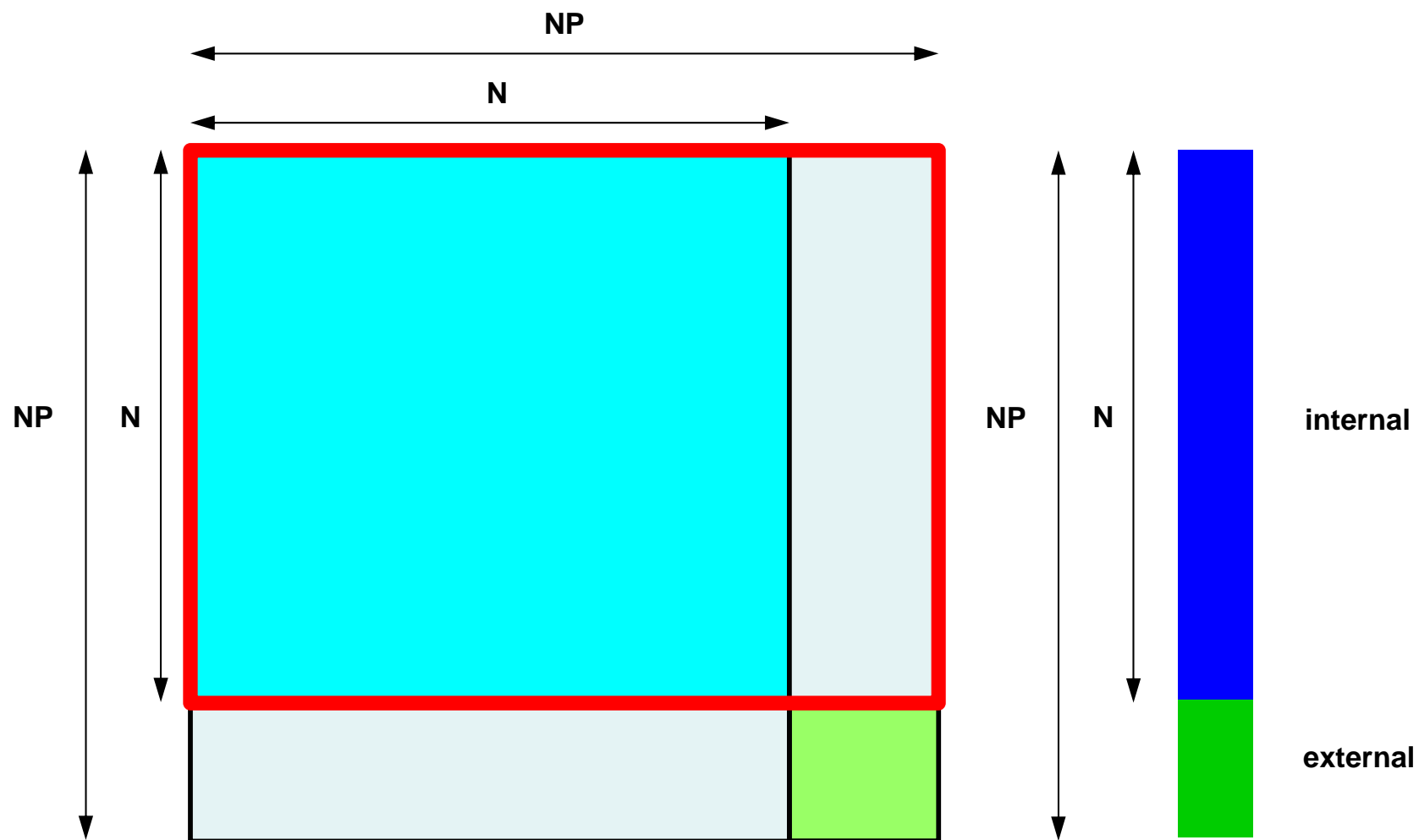


```
}
```

# Local Matrix: 各プロセスにおける係数行列



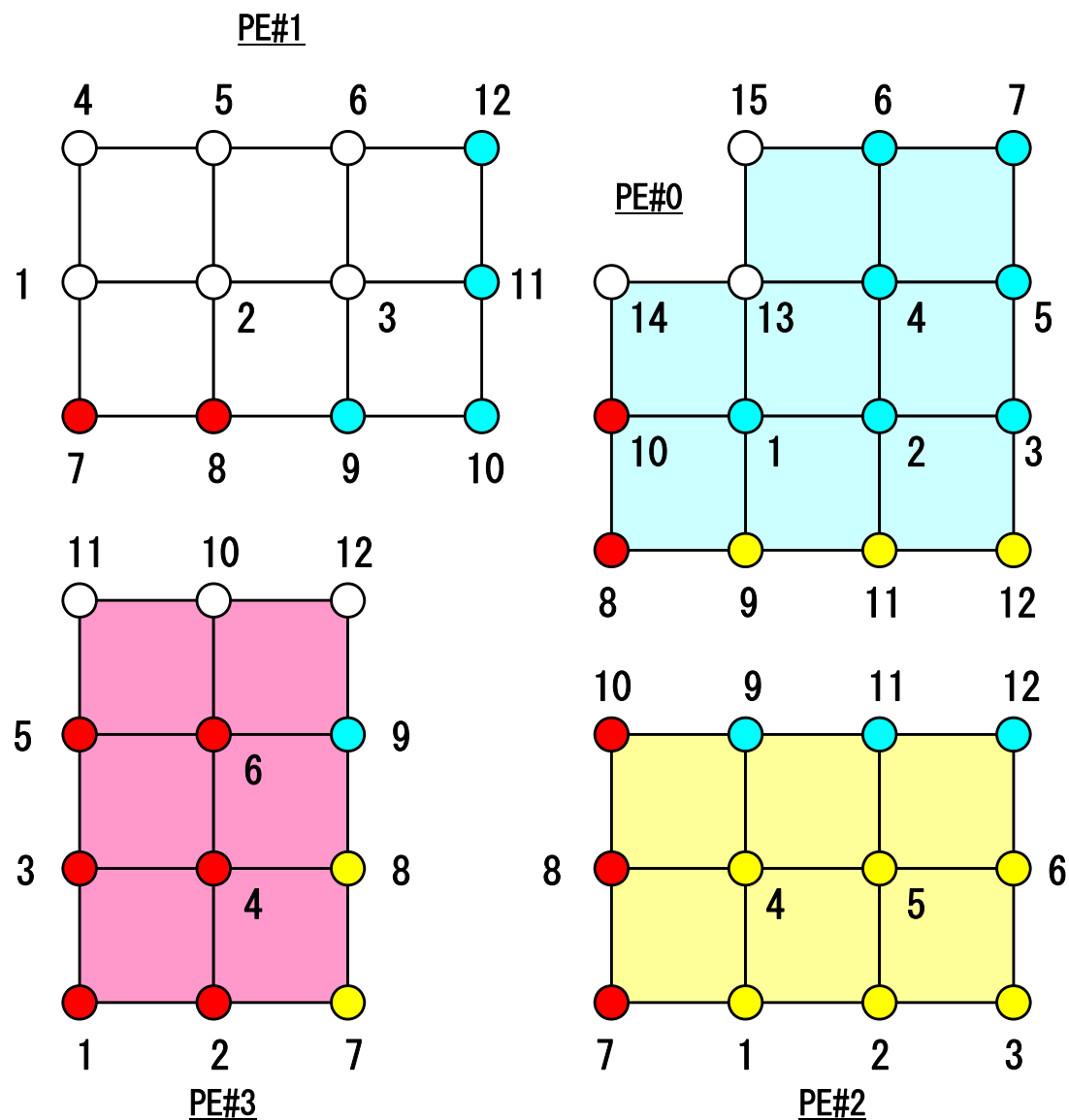
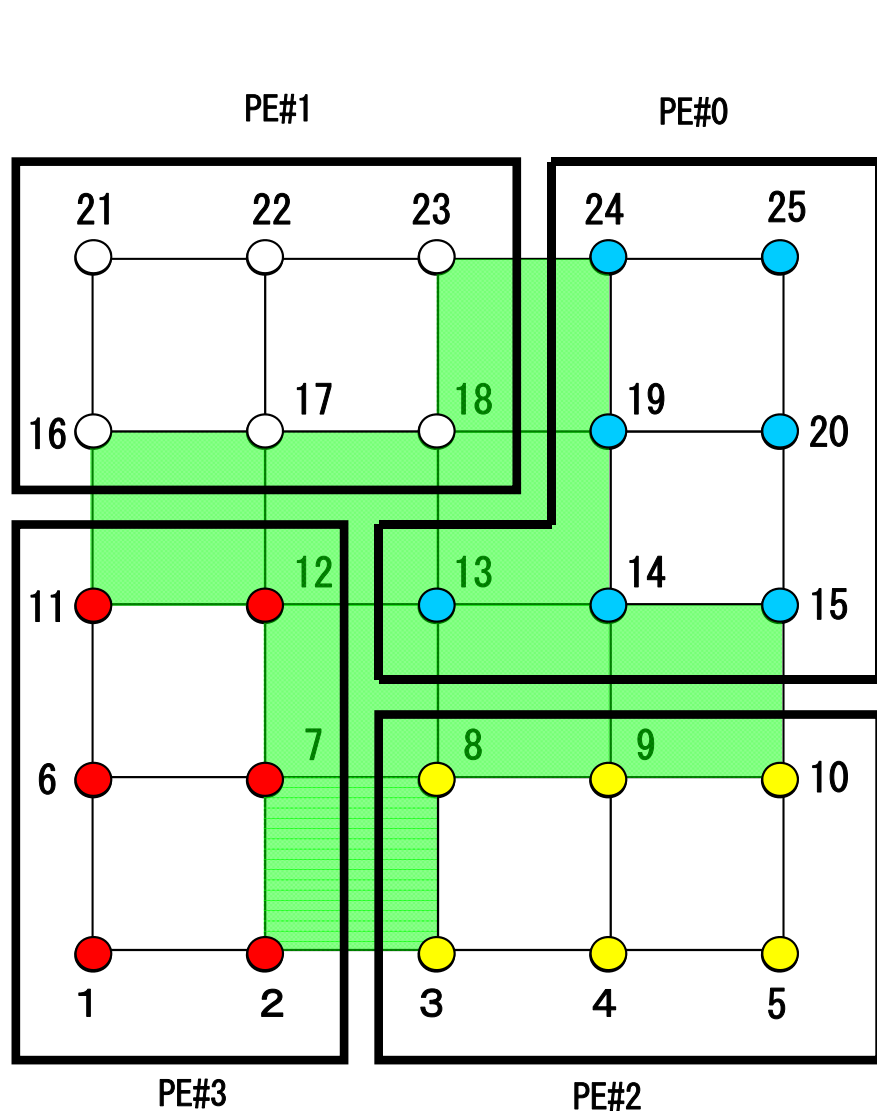
# 本当に必要なのはこの部分





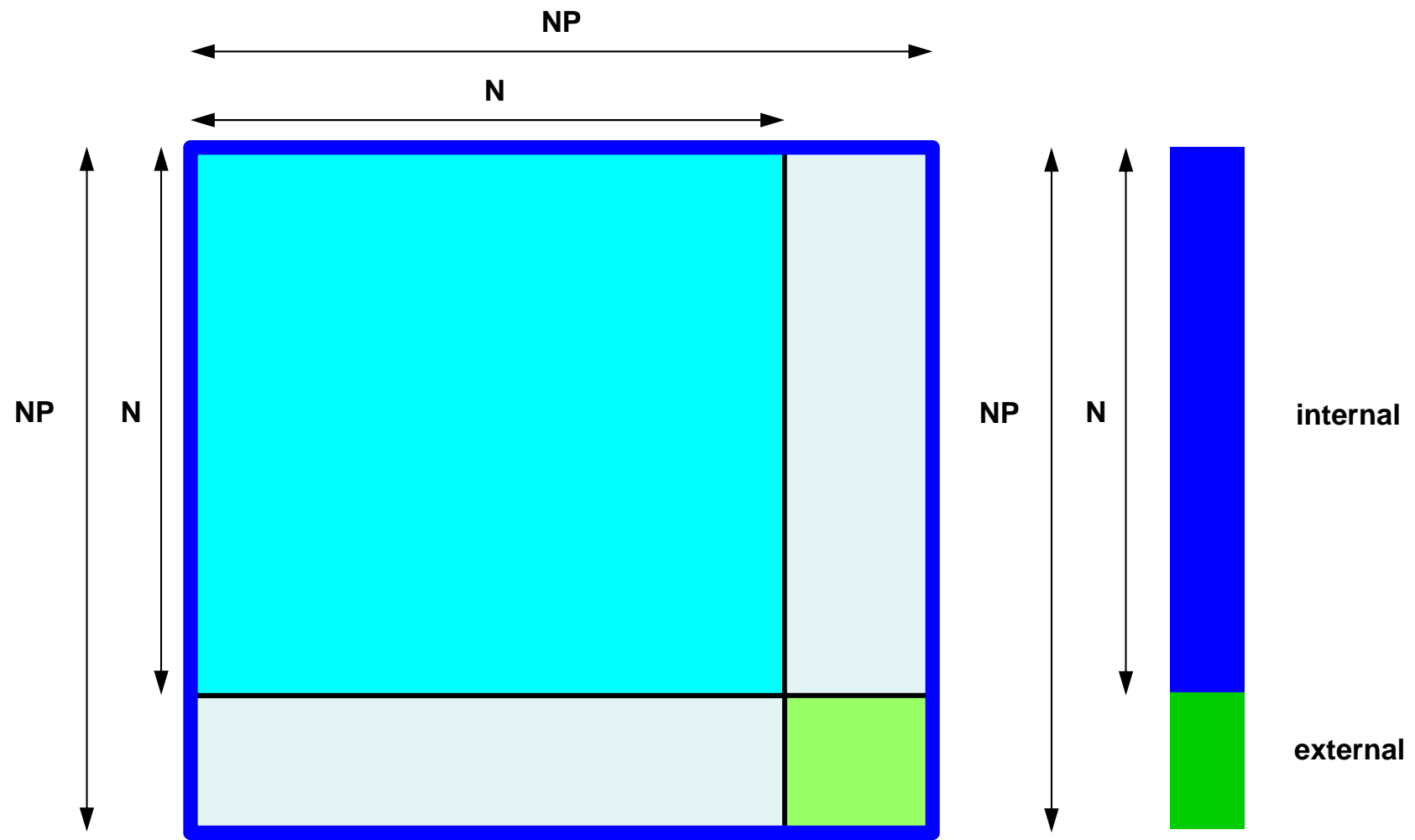
# 全ての要素の計算を実施する

## 外点を含むオーバーラップ領域の要素の計算も実施

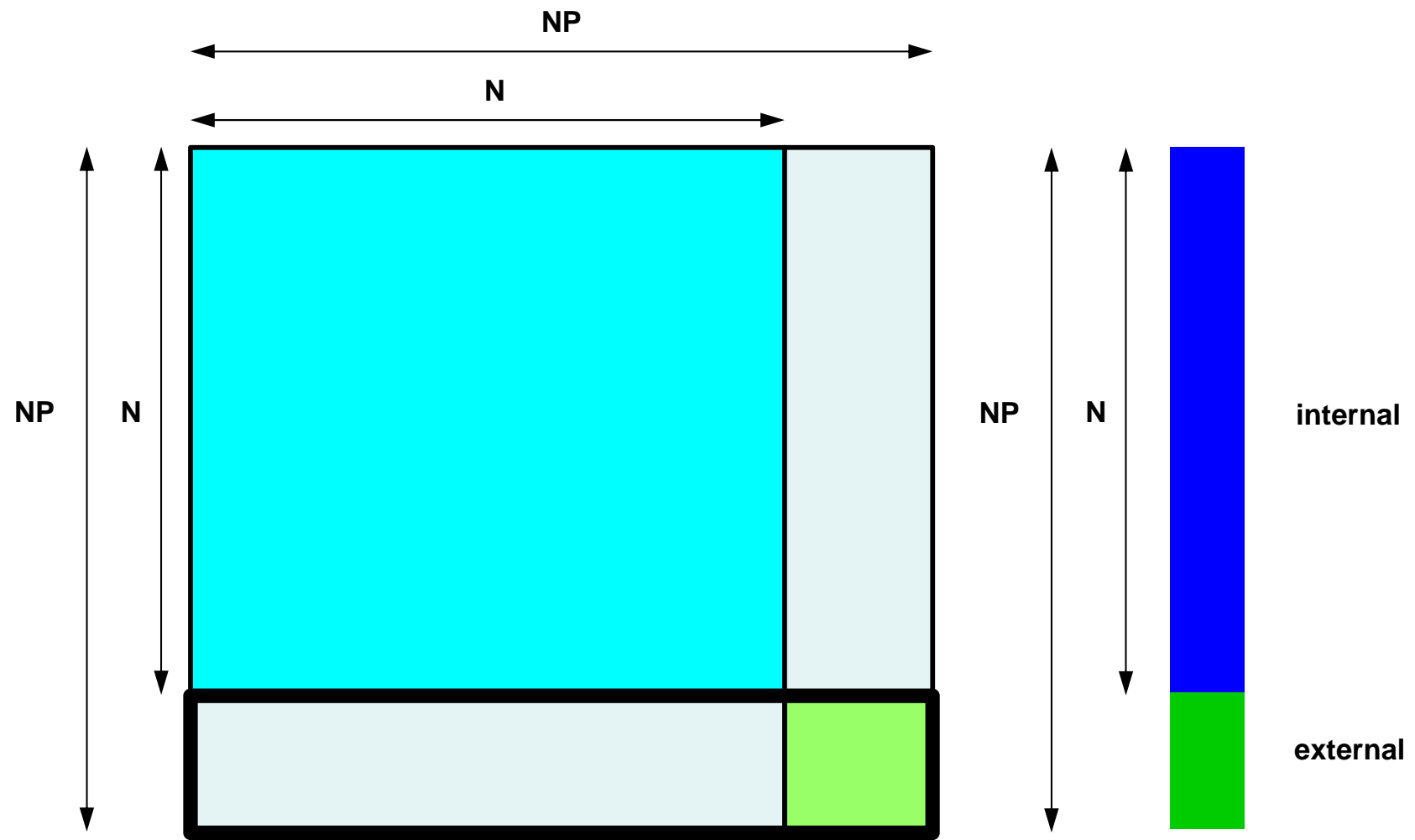




従って結果的にはこのような行列を得るが



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



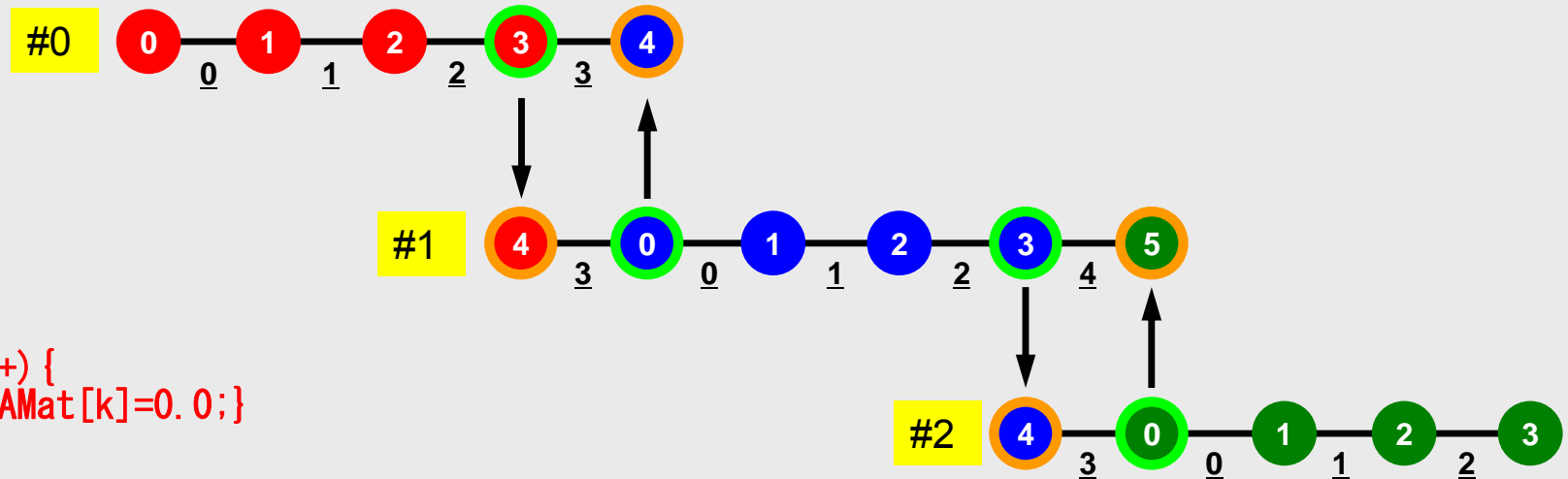
# プログラム: 1d.c(9/11)

## 境界条件: 1CPUのときとほとんど同じ

```
/*
//
// |-----|
// | BOUNDARY conditions |
// |-----|
*/
```

```
/* X=Xmin */
if (MyRank==0) {
  i=0;
  jS= Index[i];
  AMat[jS]= 0.0;
  Diag[i ]= 1.0;
  Rhs [i ]= 0.0;

  for (k=0;k<NPLU;k++) {
    if (Item[k]==0) {AMat[k]=0.0;}
  }
}
```



# プログラム: 1d.c (10/11)

## 共役勾配法

```

/*
// +-----+
// | CG iterations |
// +-----+
//=== */
R = calloc(NP, sizeof(double));
Z = calloc(NP, sizeof(double));
P = calloc(NP, sizeof(double));
Q = calloc(NP, sizeof(double));
DD= calloc(NP, sizeof(double));

for (i=0; i<N; i++) {
    DD[i]= 1.0 / Diag[i];
}

/*
//-- {r0}= {b} - [A]{xini} |
*/
for (neib=0; neib<NeibPETot; neib++) {
    for (k=export_index[neib]; k<export_index[neib+1]; k++) {
        kk= export_item[k];
        SendBuf[k]= U[kk];
    }
}

```

```

Compute  $r^{(0)} = b - [A]x^{(0)}$ 
for  $i = 1, 2, \dots$ 
    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

# 前处理, DAXPY

```
/*  
/-- {z} = [Minv]{r}  
*/  
  for (i=0; i<N; i++) {  
    Z[i] = DD[i] * R[i];  
  }
```

```
/*  
/-- {x} = {x} + ALPHA*{p}  
  -- {r} = {r} - ALPHA*{q}  
*/  
  for (i=0; i<N; i++) {  
    U[i] += Alpha * P[i];  
    R[i] -= Alpha * Q[i];  
  }
```

# 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

```
/*  
/-- {q} = [A] {p}  
*/  
for (neib=0; neib<NeibPETot; neib++) {  
    for (k=export_index[neib]; k<export_index[neib+1]; k++) {  
        kk= export_item[k];  
        SendBuf[k]= P[kk];  
    }  
}  
  
for (neib=0; neib<NeibPETot; neib++) {  
    is = export_index[neib];  
    len_s= export_index[neib+1] - export_index[neib];  
    MPI_Isend(&SendBuf[is], len_s, MPI_DDOUBLE, NeibPE[neib],  
             0, MPI_COMM_WORLD, &RequestSend[neib]);  
}  
  
for (neib=0; neib<NeibPETot; neib++) {  
    ir = import_index[neib];  
    len_r= import_index[neib+1] - import_index[neib];  
    MPI_Irecv(&RecvBuf[ir], len_r, MPI_DDOUBLE, NeibPE[neib],  
            0, MPI_COMM_WORLD, &RequestRecv[neib]);  
}  
  
MPI_Waitall(NeibPETot, RequestRecv, StatRecv);  
  
for (neib=0; neib<NeibPETot; neib++) {  
    for (k=import_index[neib]; k<import_index[neib+1]; k++) {  
        kk= import_item[k];  
        P[kk]=RecvBuf[k];  
    }  
}
```

# 行列ベクトル積 (2/2)

$$\{q\} = [A]\{p\}$$

```
MPI Waitall(NeibPETot, RequestSend, StatSend);
```

```
for (i=0; i<N; i++) {  
    Q[i] = Diag[i] * P[i];  
    for (j=Index[i]; j<Index[i+1]; j++) {  
        Q[i] += AMat[j]*P[Item[j]];  
    }  
}
```

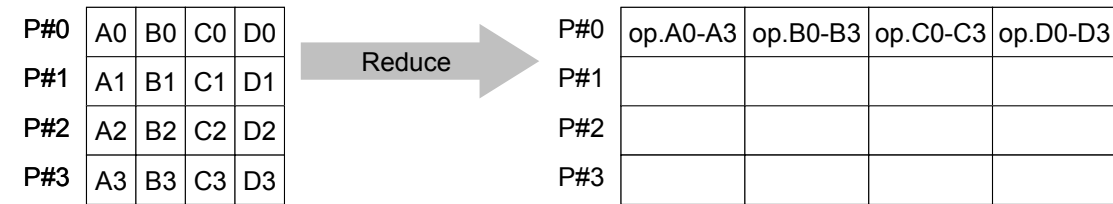


# 内積

各プロセスで計算した値を, MPI\_Allreduceで合計

```
/*  
//-- RHO= {r} {z}  
*/  
    Rho0= 0.0;  
    for (i=0; i<N; i++) {  
        Rho0 += R[i] * Z[i];  
    }  
  
    ierr = MPI_Allreduce(&Rho0, &Rho, 1, MPI_DOUBLE,  
                        MPI_SUM, MPI_COMM_WORLD);
```

# MPI\_Reduce



- 「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「recvbuf」に格納する。

- 総和, 積, 最大, 最小 他

- MPI\_Reduce

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

- 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

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

# 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) C

`MPI_Reduce`

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

```
double x0, x1;
```

`MPI_Reduce`

```
(&x0, &x1, 1, MPI_DOUBLE, MPI_MAX, 0, <comm>);
```

```
double x0[4], xmax[4];
```

`MPI_Reduce`

```
(x0, xmax, 4, MPI_DOUBLE, MPI_MAX, 0, <comm>);
```

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( $i=0\sim 3$ )

# MPI\_Reduceの例(2/2) C

## MPI\_Reduce

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

```
double X0, XSUM;
```

## MPI\_Reduce

```
(&X0, &XSUM, 1, MPI_DOUBLE, MPI_SUM, 0, <comm>)
```

各プロセスにおける, X0の総和が0番PEのXSUMに入る。

```
double X0[4];
```

## MPI\_Reduce

```
(&X0[0], &X0[2], 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>)
```

各プロセスにおける,

- ・ X0[0]の総和が0番プロセスのX0[2]に入る。
- ・ X0[1]の総和が0番プロセスのX0[3]に入る。



# CG法 (1/5)

```

/*
//-- {r0} = {b} - [A]{xini} |
*/
for (neib=0; neib < NeibPETot; neib++) {
    for (k=export_index[neib]; k < export_index[neib+1]; k++) {
        kk= export_item[k];
        SendBuf[k]= PHI[kk];
    }
}

for (neib=0; neib < NeibPETot; neib++) {
    is = export_index[neib];
    len_s= export_index[neib+1] - export_index[neib];
    MPI_Isend(&SendBuf[is], len_s, MPI_DOUBLE, NeibPE[neib],
             0, MPI_COMM_WORLD, &RequestSend[neib]);
}

for (neib=0; neib < NeibPETot; neib++) {
    ir = import_index[neib];
    len_r= import_index[neib+1] - import_index[neib];
    MPI_Irecv(&RecvBuf[ir], len_r, MPI_DOUBLE, NeibPE[neib],
             0, MPI_COMM_WORLD, &RequestRecv[neib]);
}

MPI_Waitall(NeibPETot, RequestRecv, StatRecv);

for (neib=0; neib < NeibPETot; neib++) {
    for (k=import_index[neib]; k < import_index[neib+1]; k++) {
        kk= import_item[k];
        PHI[kk]=RecvBuf[k];
    }
}

MPI_Waitall(NeibPETot, RequestSend, StatSend);

```

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

**for**  $i = 1, 2, \dots$

    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)

```

for (i=0; i<N; i++) {
    R[i] = Diag[i]*PHI[i];
    for (j=Index[i]; j<Index[i+1]; j++) {
        R[i] += AMat[j]*PHI[Item[j]];
    }
}

BNorm20 = 0.0;
for (i=0; i<N; i++) {
    BNorm20 += Rhs[i] * Rhs[i];
    R[i] = Rhs[i] - R[i];
}
ierr = MPI_Allreduce(&BNorm20, &BNorm2, 1, MPI_DOUBLE,
                    MPI_SUM, MPI_COMM_WORLD);

for (iter=1; iter<=IterMax; iter++) {

/*
//-- {z} = [Minv]{r}
*/
    for (i=0; i<N; i++) {
        Z[i] = DD[i] * R[i];
    }

/*
//-- RHO = {r}{z}
*/
    Rho0 = 0.0;
    for (i=0; i<N; i++) {
        Rho0 += R[i] * Z[i];
    }
    ierr = MPI_Allreduce(&Rho0, &Rho, 1, MPI_DOUBLE,
                        MPI_SUM, MPI_COMM_WORLD);
}

```

```

Compute  $r^{(0)} = b - [A]x^{(0)}$ 
for  $i = 1, 2, \dots$ 
    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)

```

/*
//-- {p} = {z} if ITER=1
// BETA= RHO / RHO1 otherwise
*/
if(iter == 1) {
  for(i=0; i<N; i++) {
    P[i] = Z[i];
  }
} else {
  Beta = Rho / Rho1;
  for(i=0; i<N; i++) {
    P[i] = Z[i] + Beta*P[i];
  }
}

/*
//-- {q} = [A] {p}
*/
for (neib=0; neib<NeibPETot; neib++) {
  for (k=export_index[neib]; k<export_index[neib+1]; k++) {
    kk= export_item[k];
    SendBuf[k]= P[kk];
  }
}

for (neib=0; neib<NeibPETot; neib++) {
  is = export_index[neib];
  len_s= export_index[neib+1] - export_index[neib];
  MPI_Isend(&SendBuf[is], len_s, MPI_DOUBLE, NeibPE[neib],
            0, MPI_COMM_WORLD, &RequestSend[neib]);
}

```

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

**for**  $i = 1, 2, \dots$

  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 (4/5)

```

for (neib=0;neib<NeibPETot;neib++) {
  ir   = import_index[neib];
  len_r= import_index[neib+1] - import_index[neib];
  MPI_Irecv(&RecvBuf[ir], len_r, MPI_DOUBLE, NeibPE[neib]
           0, MPI_COMM_WORLD, &RequestRecv[neib]);
}

```

```
MPI_Waitall(NeibPETot, RequestRecv, StatRecv);
```

```

for (neib=0;neib<NeibPETot;neib++) {
  for (k=import_index[neib];k<import_index[neib+1];k++) {
    kk= import_item[k];
    P[kk]=RecvBuf[k];
  }
}

```

```
MPI_Waitall(NeibPETot, RequestSend, StatSend);
```

```

for (i=0;i<N;i++) {
  Q[i] = Diag[i] * P[i];
  for (j=Index[i];j<Index[i+1];j++) {
    Q[i] += AMat[j]*P[Item[j]];
  }
}

```

```

/*
//-- ALPHA= RHO / {p} {q}
*/

```

```

C10 = 0.0;
for (i=0;i<N;i++) {
  C10 += P[i] * Q[i];
}

```

```

ierr = MPI_Allreduce(&C10, &C1, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
Alpha = Rho / C1;

```

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

**for**  $i = 1, 2, \dots$

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)

```

/*
//-- {x} = {x} + ALPHA*{p}
//   {r} = {r} - ALPHA*{q}
*/
for (i=0; i<N; i++) {
    PHI[i] += Alpha * P[i];
    R[i] -= Alpha * Q[i];
}

DNorm20 = 0.0;
for (i=0; i<N; i++) {
    DNorm20 += R[i] * R[i];
}

ierr = MPI_Allreduce(&DNorm20, &DNorm2, 1, MPI_DOUBLE,
                    MPI_SUM, MPI_COMM_WORLD);

Resid = sqrt(DNorm2/BNorm2);
if (MyRank==0)
    printf("%8d%s%16.6e\n", iter, " iters, RESID=", Resid);

if (Resid <= Eps) {
    ierr = 0;
    break;
}

Rho1 = Rho;
}

```

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

for  $i = 1, 2, \dots$

    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.c (11/11)

結果書き出し: 各プロセスごとに実施

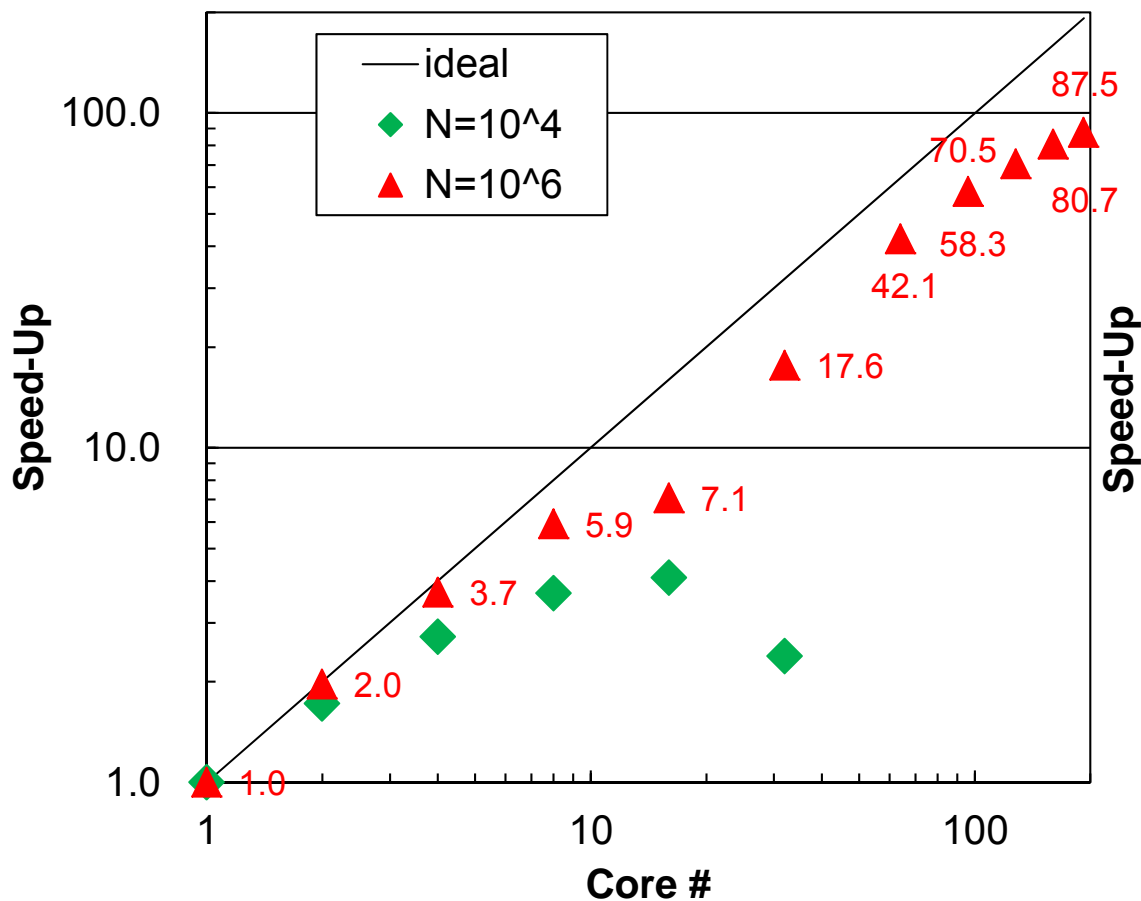
```
/*  
//-- OUTPUT  
*/  
printf("\n%s\n", "### TEMPERATURE");  
for (i=0; i<N; i++) {  
    printf("%3d%8d%16.6E\n", MyRank, i+1, PHI[i]);  
}  
  
ierr = MPI_Finalize();  
return ierr;  
}
```

- 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**

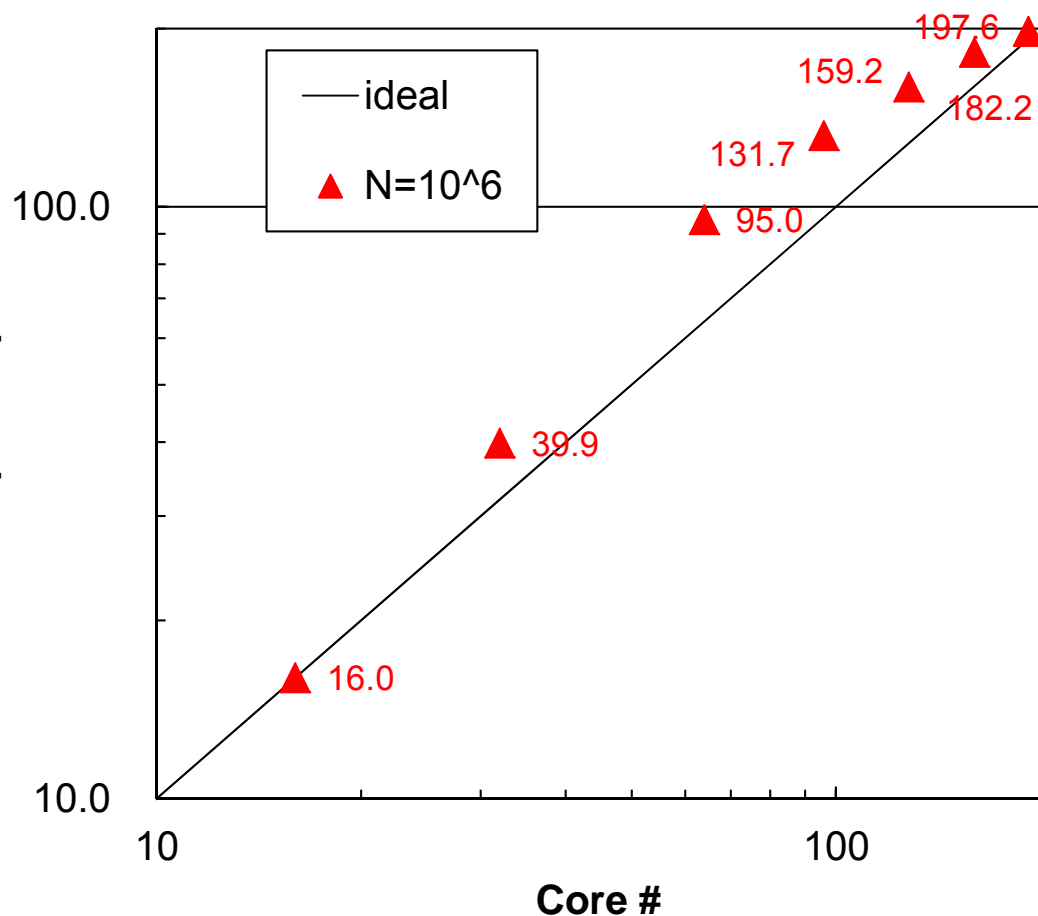
# 計算結果(1次元):CG法部分

$N=10^6$ 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 $\mu$ sec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

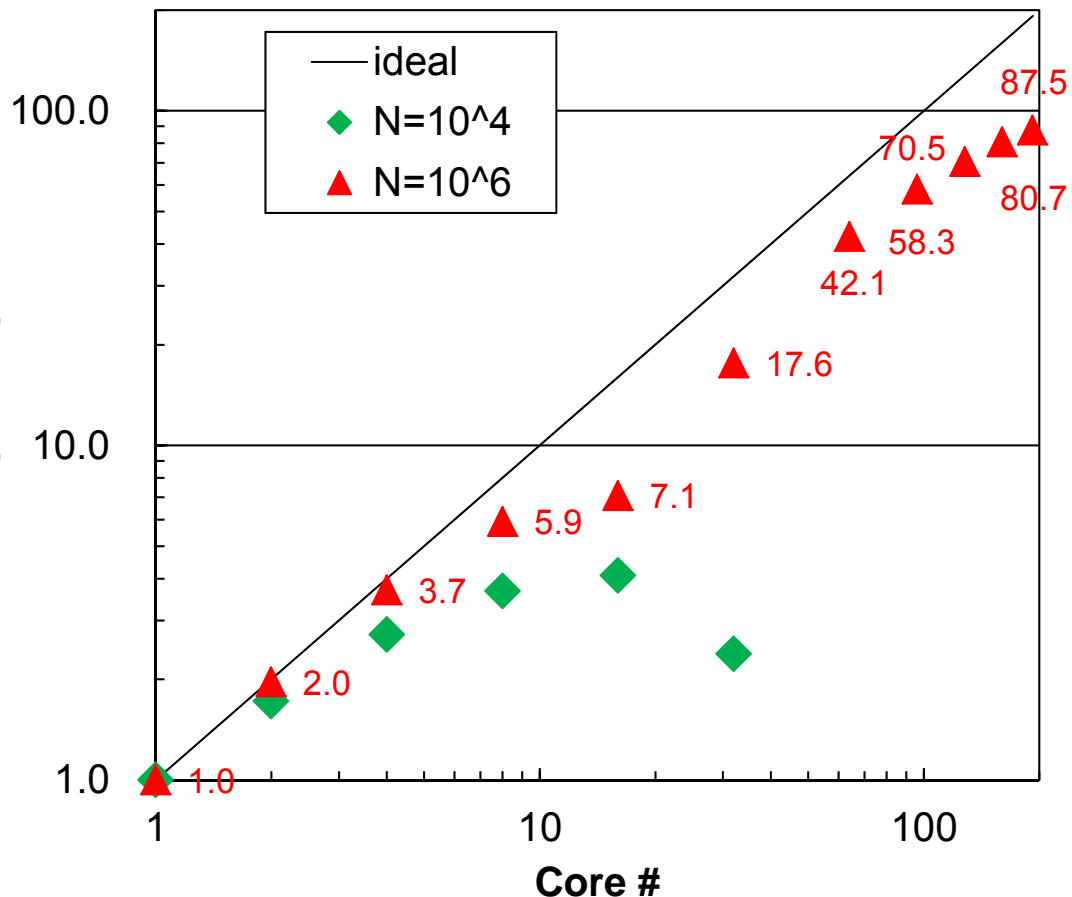
# 理想値からのずれ（続き）

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

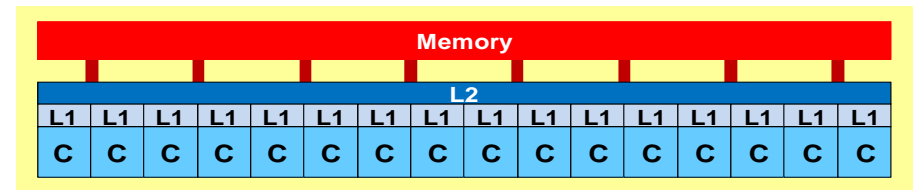


# 1コア～16コアであまり性能が出ていない件

## 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

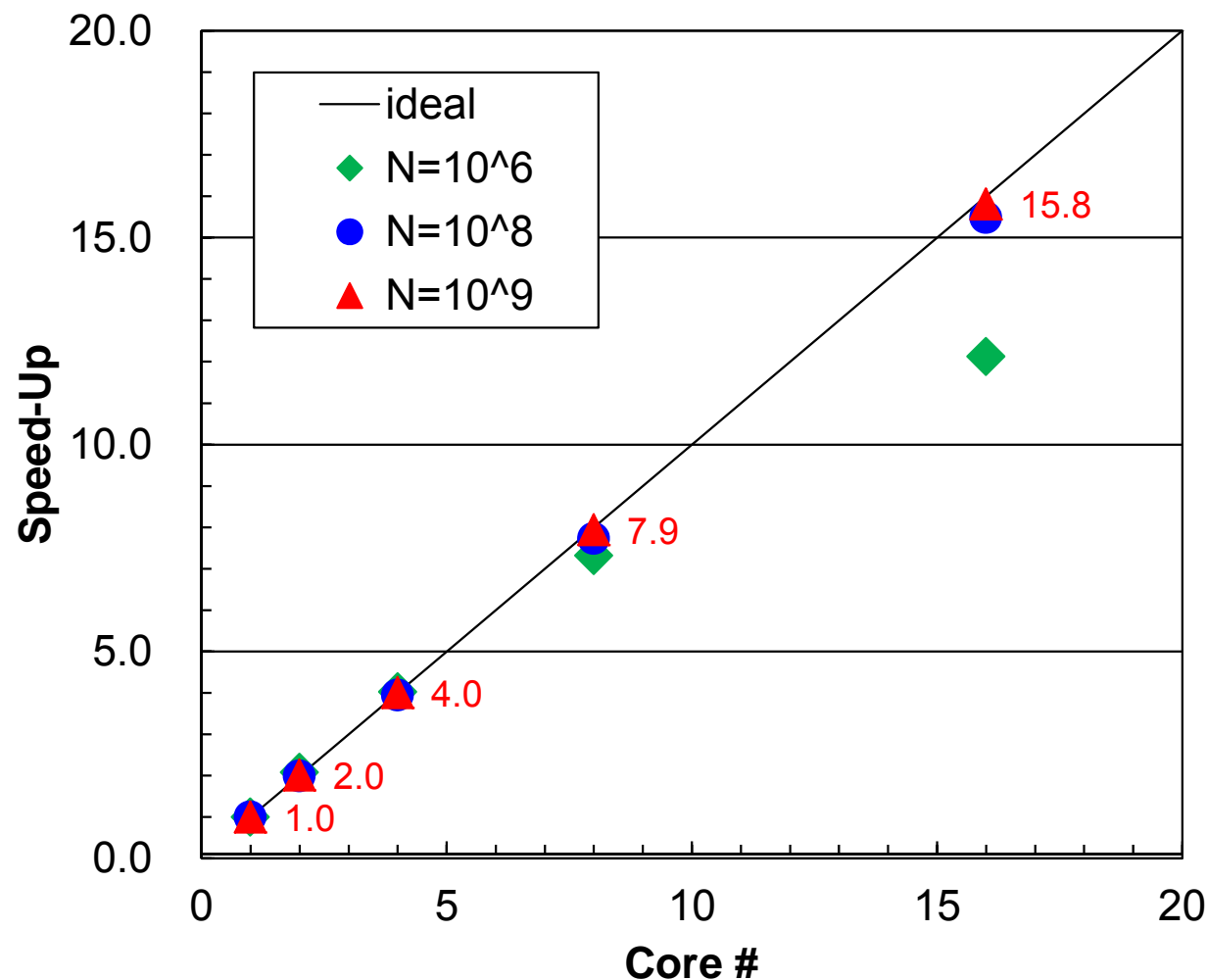


# 台形積分ではあまり影響が無い

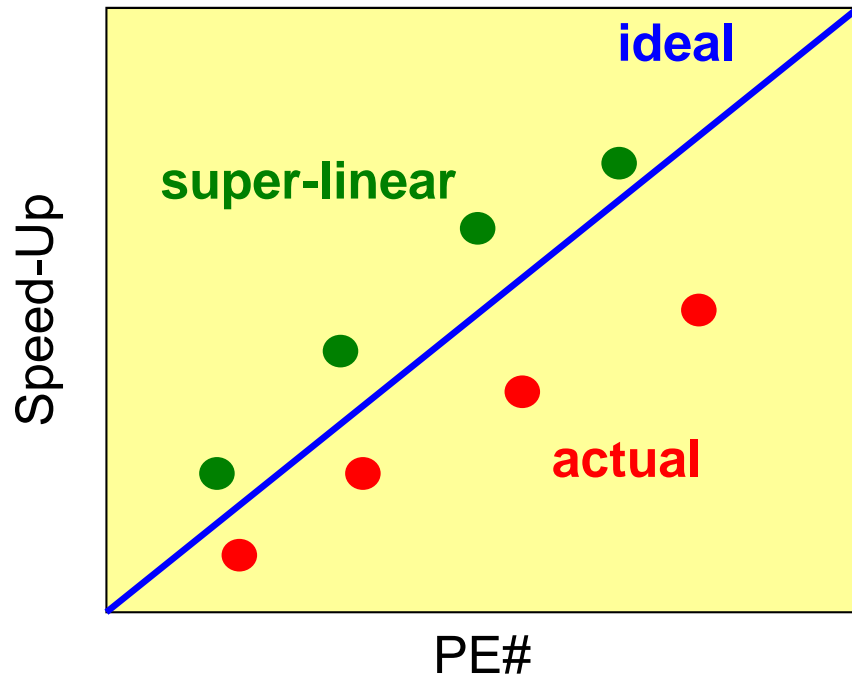
- ◆ :  $N=10^6$ , ● :  $10^8$ , ▲ :  $10^9$ , — : 理想値
- 1コアにおける計測結果(sec.)からそれぞれ算出

- 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

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

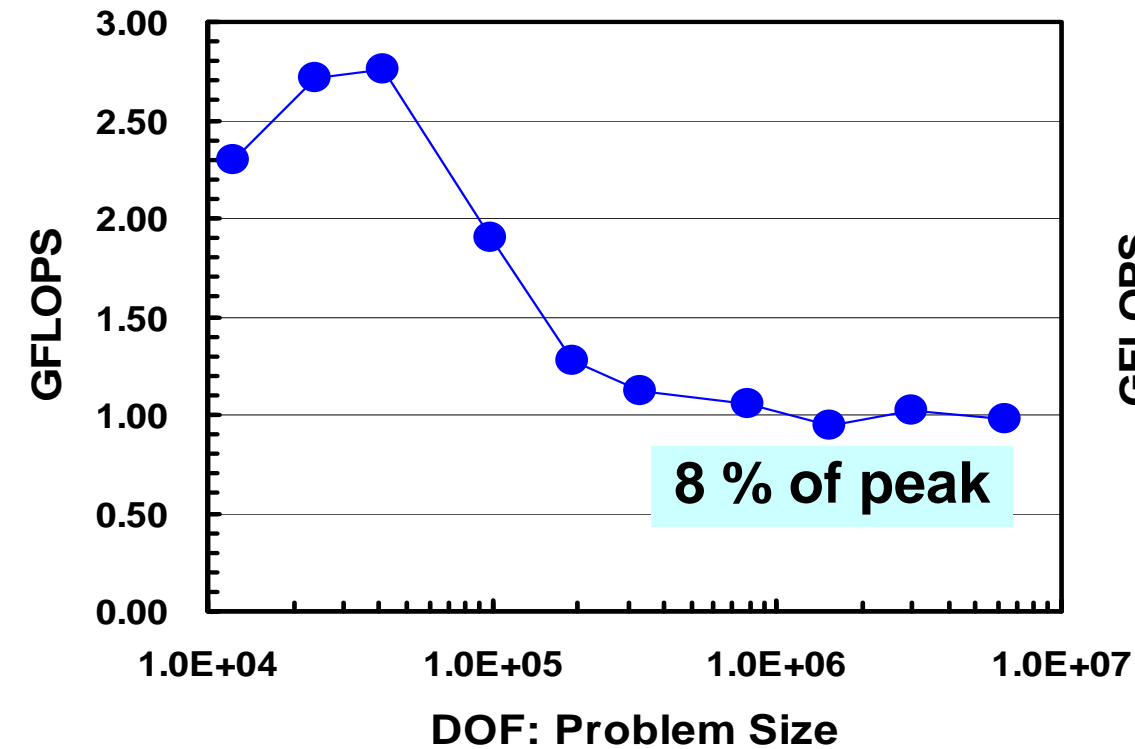
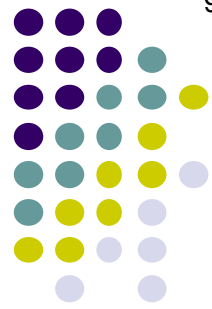


# Strong-Scalingにおける「Super-Linear」



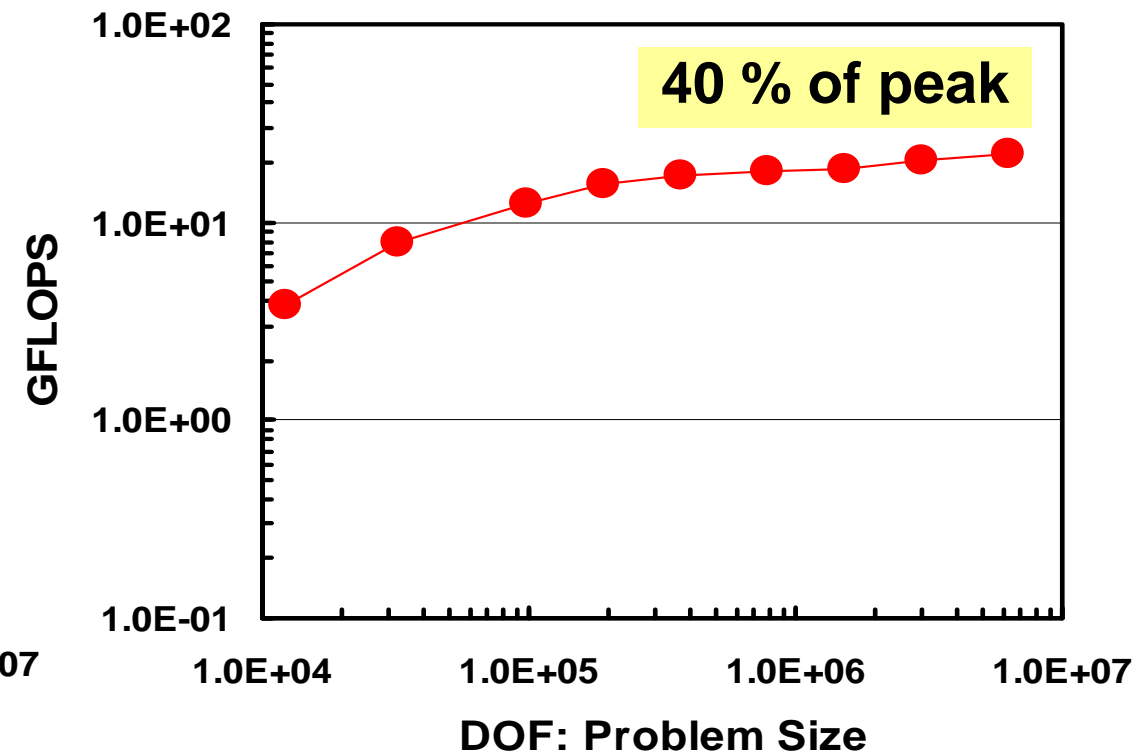
- 問題規模を固定して、使用PE数を増加させて行った場合、通常は通信の影響のために、効率は理想値（ $m$ 個のPEを使用した場合、理想的には $m$ 倍の性能になる）よりも低くなるのが普通である。
- しかし、スカラープロセッサ（PC等）の場合、逆に理想値よりも、高い性能が出る場合がある。このような現象を「Super-Linear」と呼ぶ。
  - ベクトル計算機では起こらない。

# 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

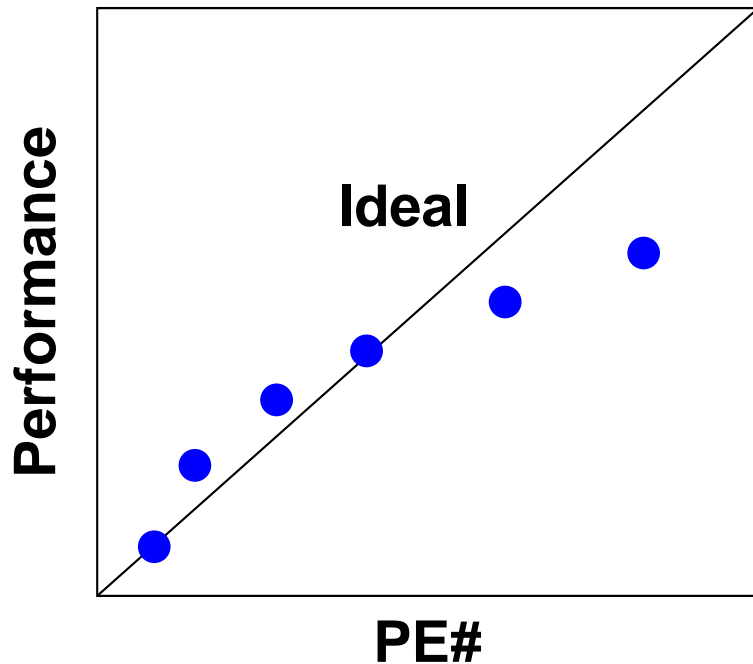
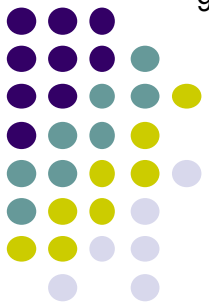


## Earth Simulator:

大規模な問題ほどベクトル長が長くなり、性能が高い

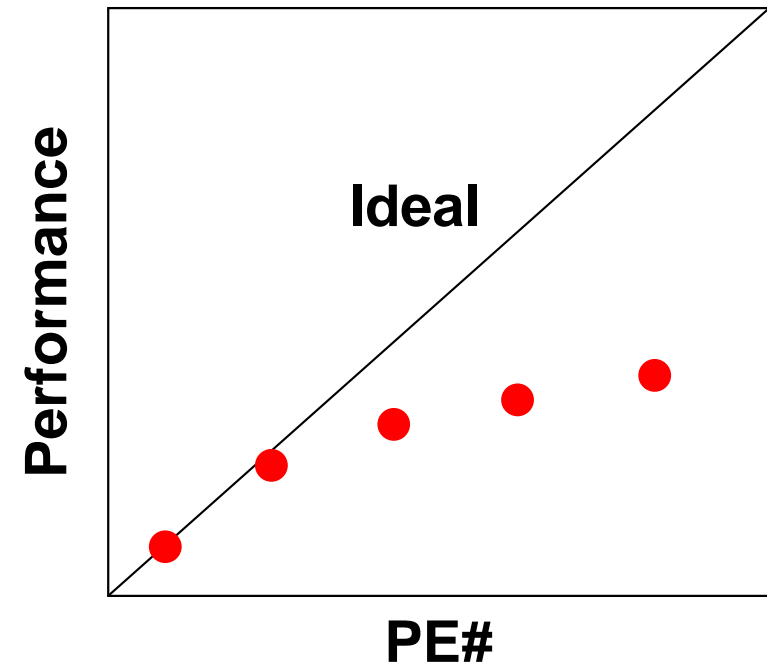
# 並列計算

## 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。

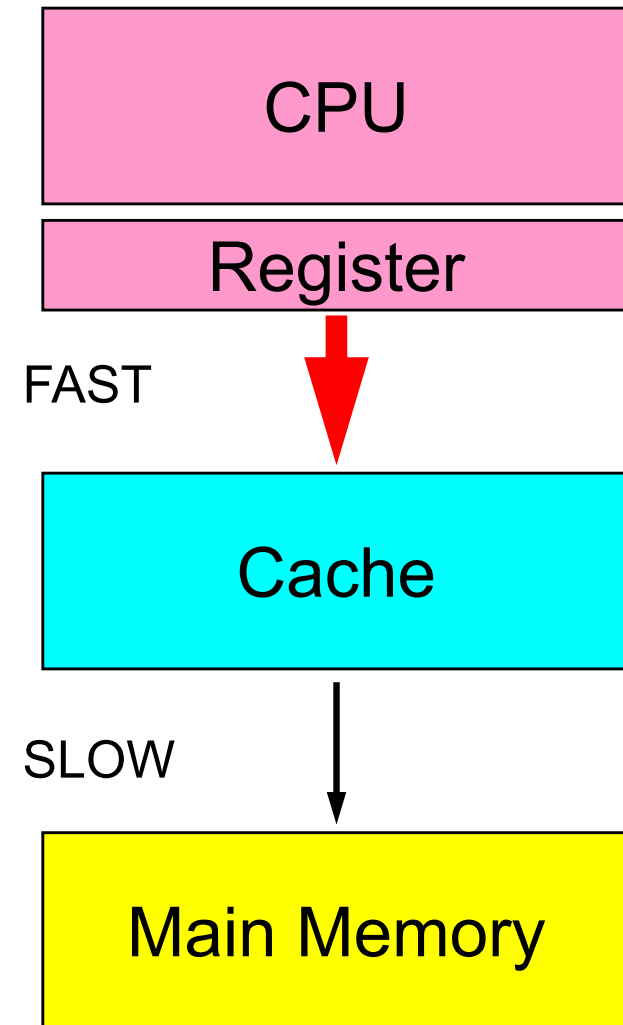


### **Earth Simulator:**

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。

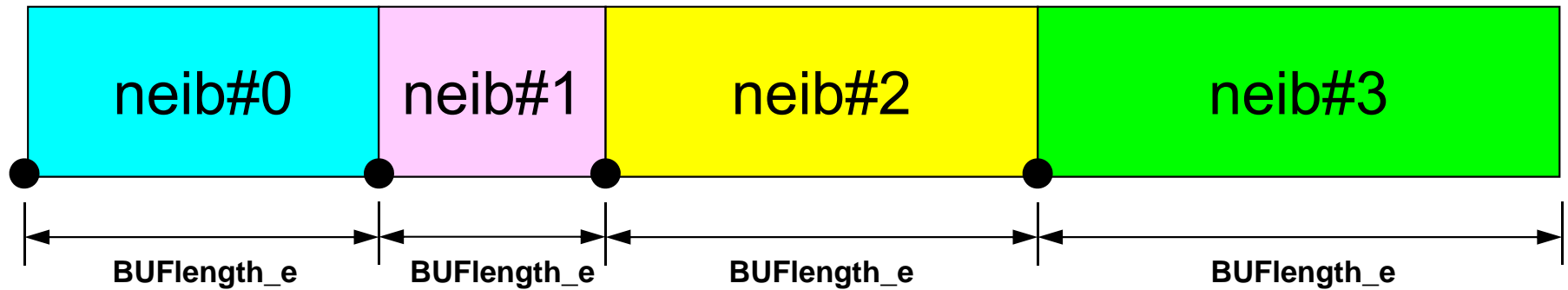
# Super-Linearの生じる理由

- 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)

SendBuf



export\_index[0]      export\_index[1]      export\_index[2]      export\_index[3]      export\_index[4]

export\_index[neib] ~ export\_index[neib+1]-1 番目の export\_item が neib 番目の隣接領域に送信される

```
for (neib=0; neib<NeibPETot;neib++){
  for (k=export_index[neib];k<export_index[neib+1];k++){
    kk= export_item[k];
    SendBuf[k]= VAL[kk];
  }
}
```

送信バッファへの代入

```
for (neib=0; neib<NeibPETot; neib++){
  tag= 0;
  iS_e= export_index[neib];
  iE_e= export_index[neib+1];
  BUFlength_e= iE_e - iS_e

  ierr= MPI_Isend
    (&SendBuf[iS_e], BUFlength_e, MPI_DOUBLE, NeibPE[neib], 0,
     MPI_COMM_WORLD, &ReqSend[neib])
}
```

```
MPI_Waitall(NeibPETot, ReqSend, StatSend);
```

# メモリーコピーも意外に時間かかる(2/2)

```

for (neib=0; neib<NeibPETot; neib++){
  tag= 0;
  iS_i= import_index[neib];
  iE_i= import_index[neib+1];
  BUFlength_i= iE_i - iS_i

  ierr= MPI_Irecv
    (&RecvBuf[iS_i], BUFlength_i, MPI_DOUBLE, NeibPE[neib], 0,
     MPI_COMM_WORLD, &ReqRecv[neib])
}

```

```
MPI_Waitall(NeibPETot, ReqRecv, StatRecv);
```

```

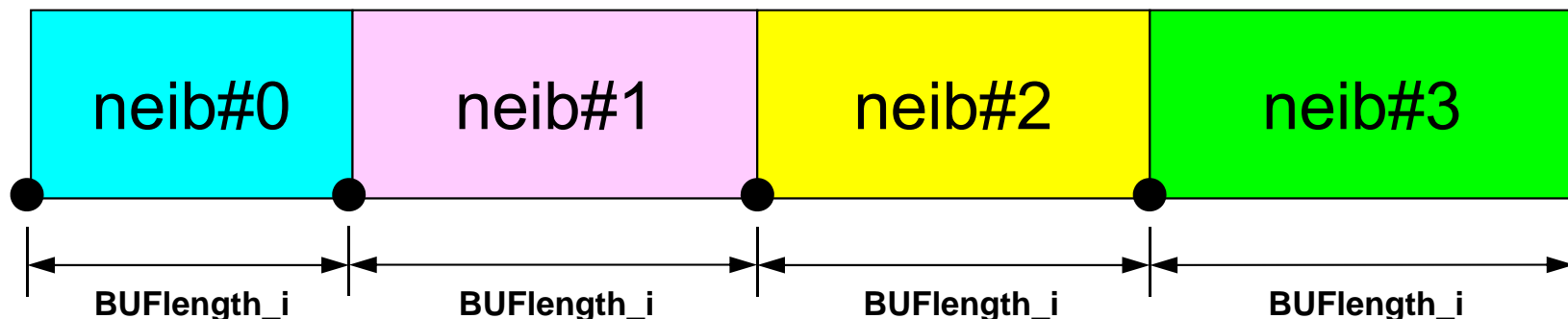
for (neib=0; neib<NeibPETot;neib++){
  for (k=import_index[neib];k<import_index[neib+1];k++){
    kk= import_item[k];
    VAL[kk]= RecvBuf[k];
  }
}

```

受信バッファからの代入

import\_index[neib] ~ import\_index[neib+1]-1番目のimport\_itemがneib番目の隣接領域から受信される

RecvBuf



import\_index[0]   import\_index[1]   import\_index[2]   import\_index[3]   import\_index[4]



# 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