並列有限要素法入門
一次元弾性解析：課題S2解説

2011年12月

中島研吾

科学技術計算Ⅱ（4820-1028）・コンピュータ科学特別講義（4810-1205）
（並列有限要素法）

- 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：一次元弾性体（トラス）

- x方向にのみ自由度（変位u）
  - 一様な：断面積A，ヤング率E
  - 境界条件
    - $x=0$　：$u=0$（固定）
    - $x=x_{max}$：大きさFの力（軸力）
- 自重によるたわみ等はナシ：バネと同じ

ファイルコピー，コンパイル

ファイルコピー

```
>$ cd <$T-fem2>          各自作成したディレクトリ
>$ cp /home/t00000/fem2/s2r.tar .
>$ tar xvf s2r.tar
```

直下に「mpi/S2-ref」というディレクトリができている。
<$T-fem2>/mpi/S2-refを<$T-S2r>と呼ぶ。

コンパイル

```
>$ cd <$T-S2r>
>$ mpicc –Os –noparallel 1d.c
```

制御ファイル：input.dat

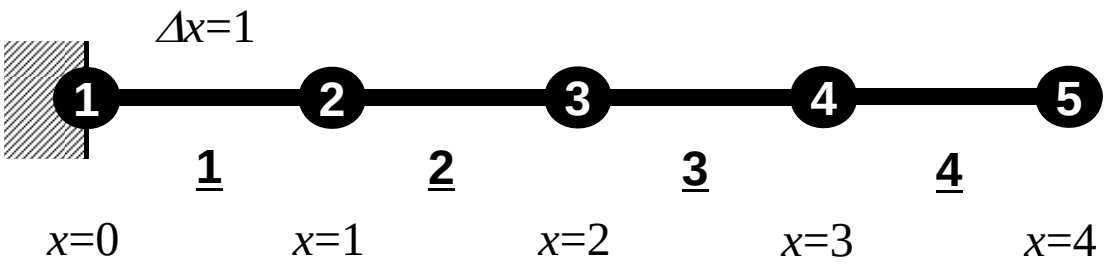
制御ファイル　input.dat

```
4                                   NE（要素数）（実際はNE=40）
1.0  1.0  1.0  1.0                  Δx（要素長さL），F，A，E
100                                 反復回数（CG法後述）
1.e-8                               CG法の反復打切誤差
```

$$\sigma = \frac{F}{A} = \frac{1}{1} = 1$$

$$\frac{du}{dx} = \varepsilon = \frac{\sigma}{E} = \frac{1}{1} = 1$$

ひずみ＝100%：2倍の長さに伸びる



要素番号
節点番号（全体）

ジョブスプリクト：go2.sh

```
#@$-r go1              実行ジョブ名（qstatで表示）
#@$-q lecture1         実行キュー名
#@$-N 1                使用ノード数（最大4）
#@$-J T8               ノードあたりMPIプロセス数（領域数）（T1~T16）
#@$-e err              標準エラー出力ファイル名
#@$-o t08.lst          標準出力ファイル名
#@$-lM 28GB            1ノードあたりメモリ使用量（固定）
#@$-lT 00:05:00        実行時間（上限15分，この場合は5分）
#@$

cd $PBS_O_WORKDIR      実行ディレクトリ移動
mpirun ./numarun.sh ./a.out          mpirun
```

並列計算:-Jの値を調整
1コア～16コア

```
#@$-r go1              実行ジョブ名（qstatで表示）
#@$-q lecture1         実行キュー名
#@$-N 1                使用ノード数（最大4）
#@$-J T8               ノードあたりMPIプロセス数（領域数）（T1~T16）
#@$-e err              標準エラー出力ファイル名
#@$-o t08.lst          標準出力ファイル名
#@$-lM 28GB            1ノードあたりメモリ使用量（固定）
#@$-lT 00:05:00        実行時間（上限15分，この場合は5分）
#@$

cd $PBS_O_WORKDIR      実行ディレクトリ移動
mpirun ./numarun.sh ./a.out          mpirun
```

「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

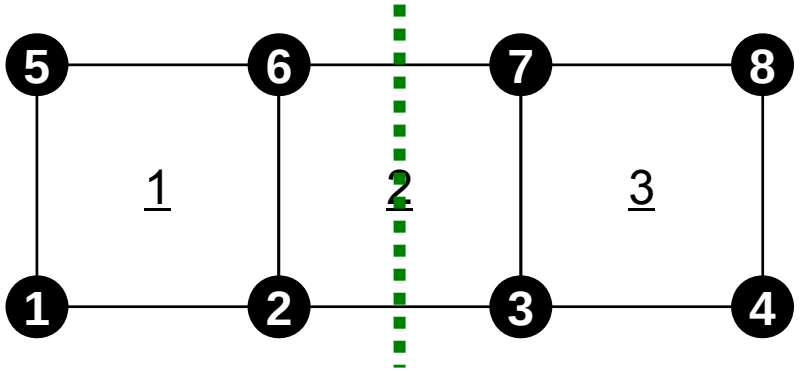
- 元のプログラムとほとんど変わらない

- 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）
- 応力計算

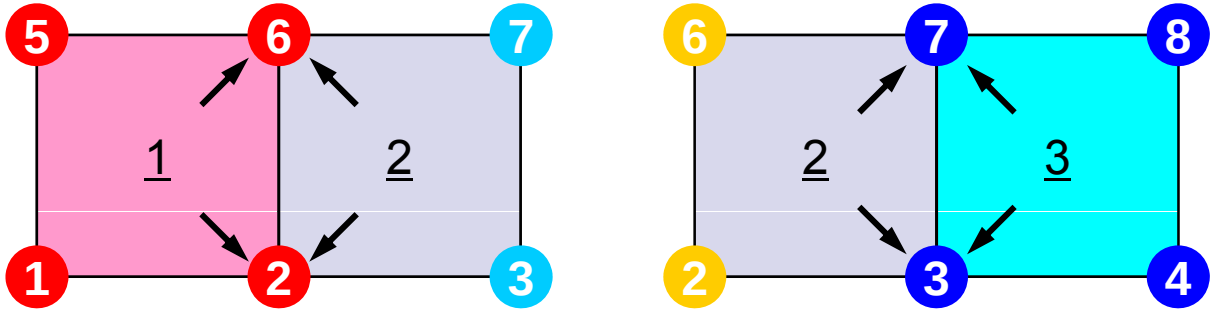
四角形要素



「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

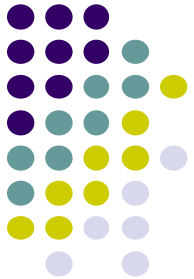
これではマトリクス生成に必要な情報は不十分



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

応力計算も同じプロセス

並列有限要素法の局所データ構造

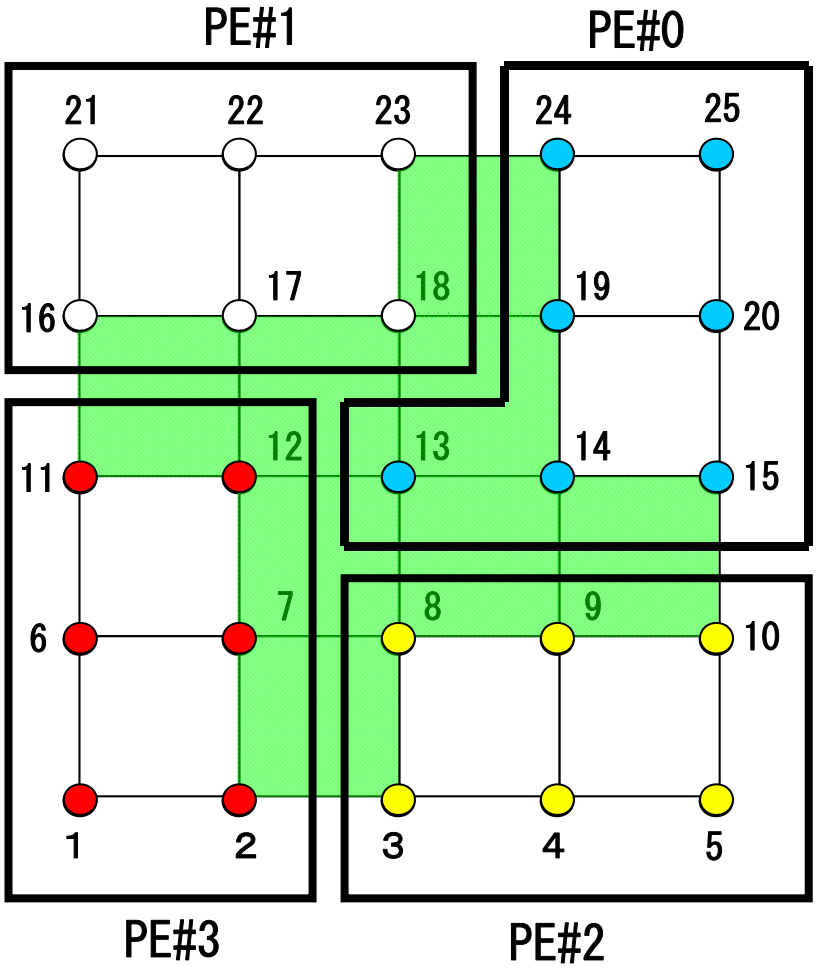


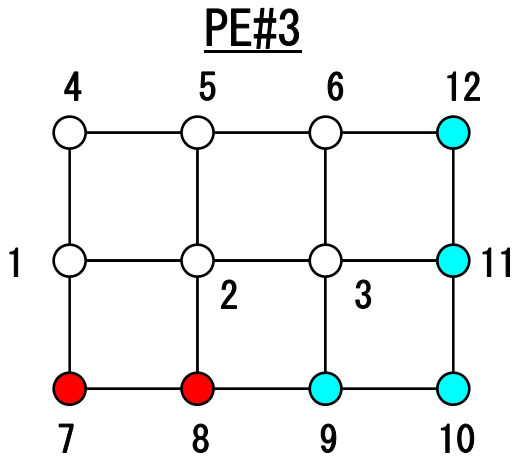
- 節点ベース：Node-based partitioning
- 局所データに含まれるもの：
  - その領域に本来含まれる節点
  - それらの節点を含む要素
  - 本来領域外であるが，それらの要素に含まれる節点
- 節点は以下の3種類に分類
  - 内点：Internal nodes　　その領域に本来含まれる節点
  - 外点：External nodes　　本来領域外であるがマトリクス生成に必要な節点
  - 境界点：Boundary nodes　　他の領域の「外点」となっている節点
- 領域間の通信テーブル

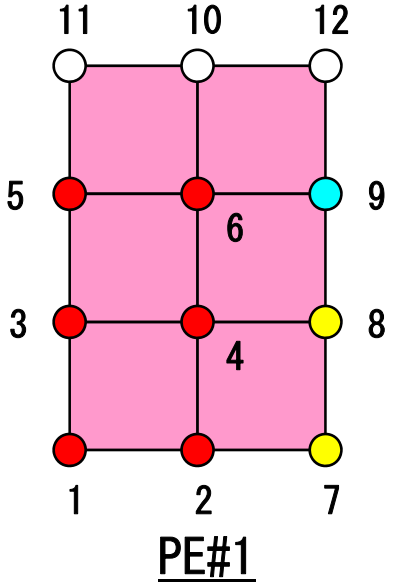
- 領域間の接続をのぞくと，大域的な情報は不要
  - 有限要素法の特性：要素で閉じた計算

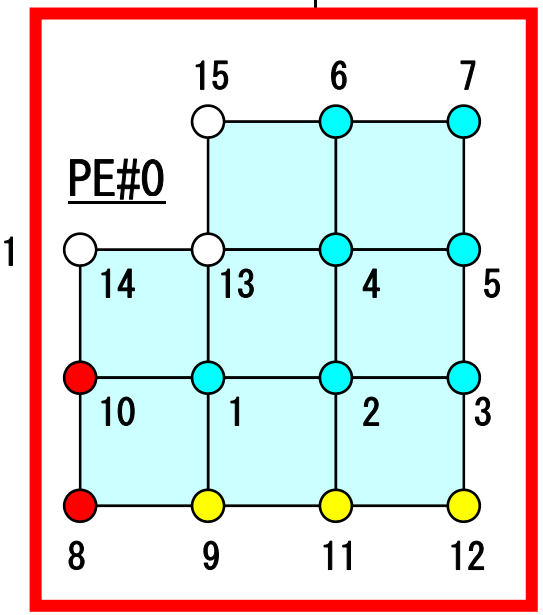
Node-based Partitioning

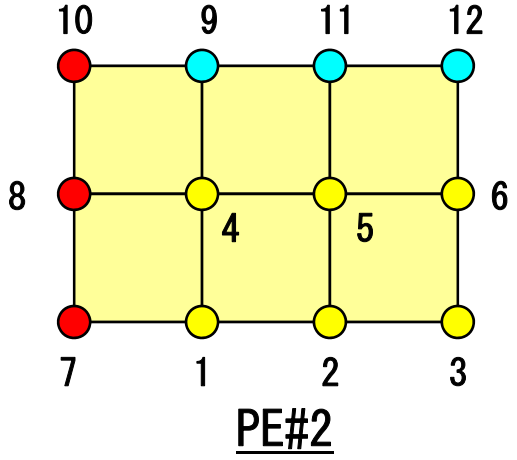
internal nodes - elements - external nodes





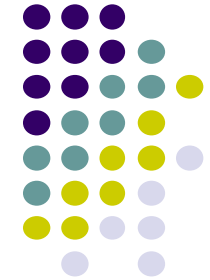




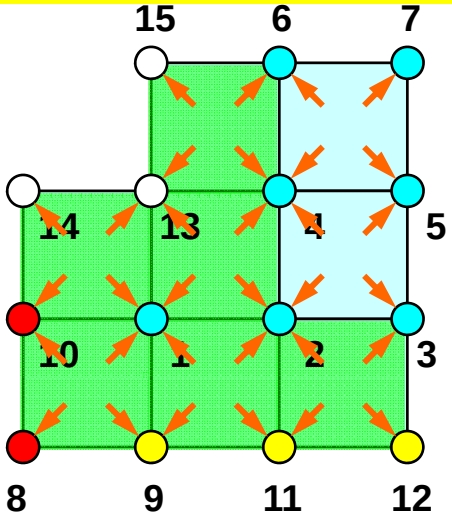


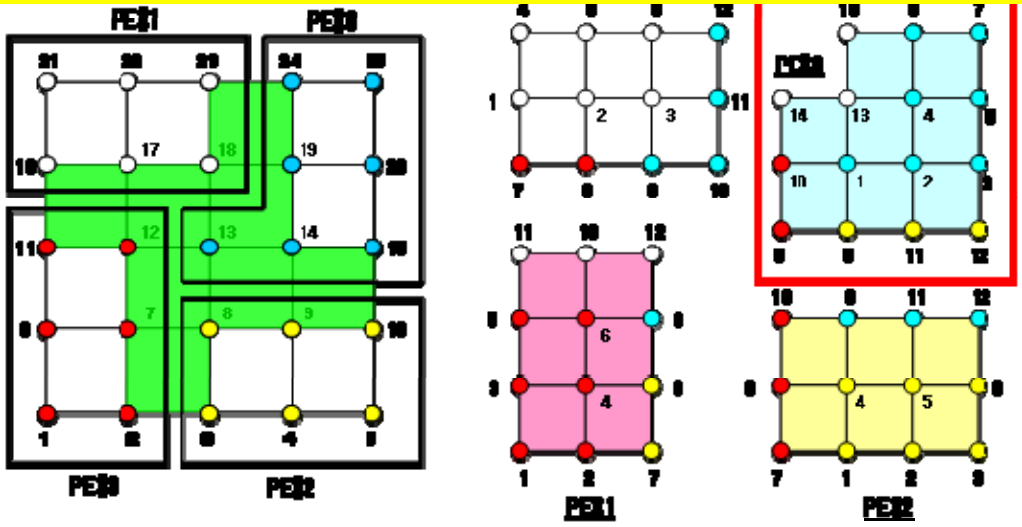
Node-based Partitioning

internal nodes - elements - external nodes

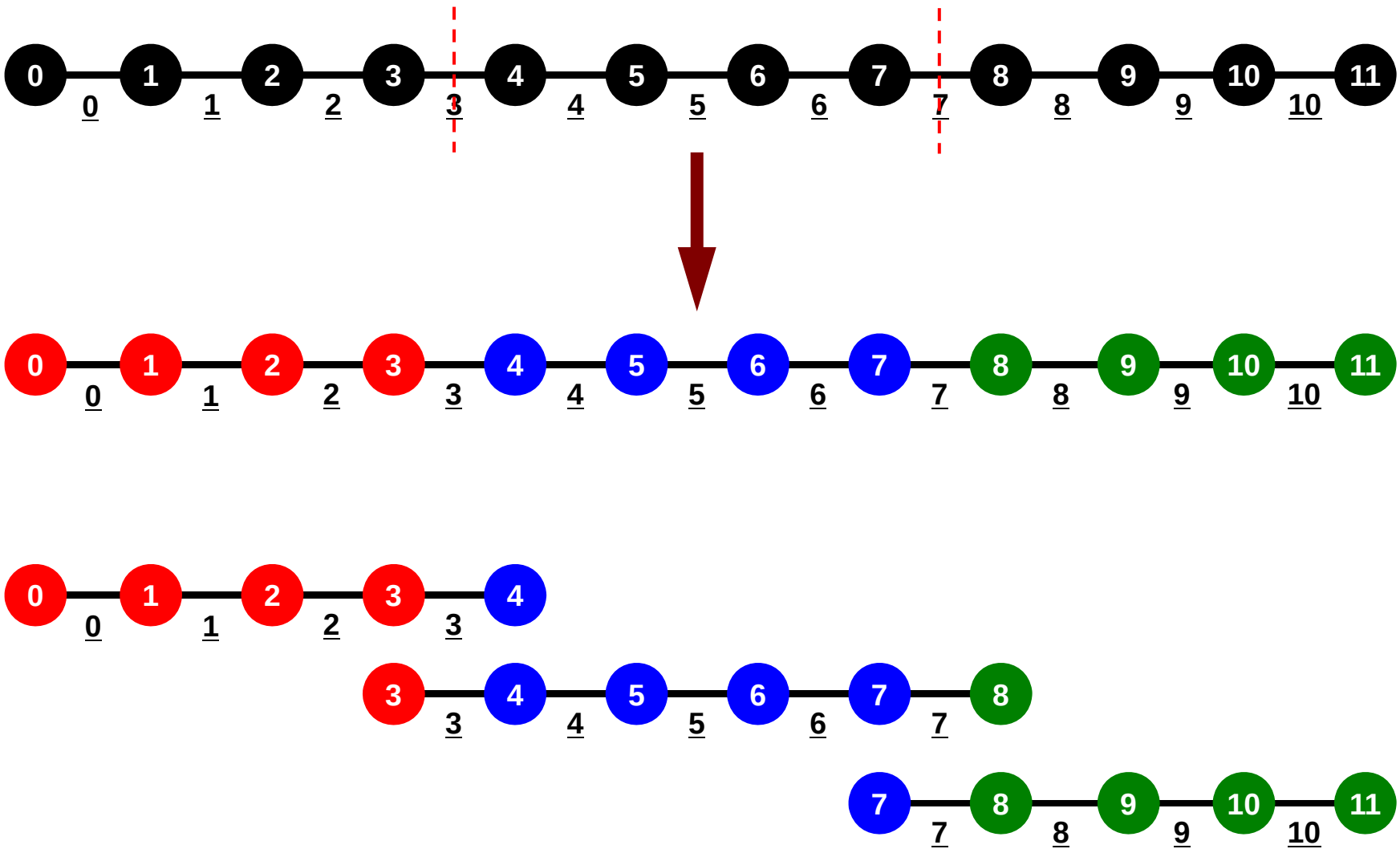


- Partitioned nodes themselves (Internal Nodes) 内点
- Elements which include Internal Nodes 内点を含む要素
- External Nodes included in the Elements 外点
  in overlapped region among partitions.
- Info of External Nodes are required for completely local element–based operations on each processor.



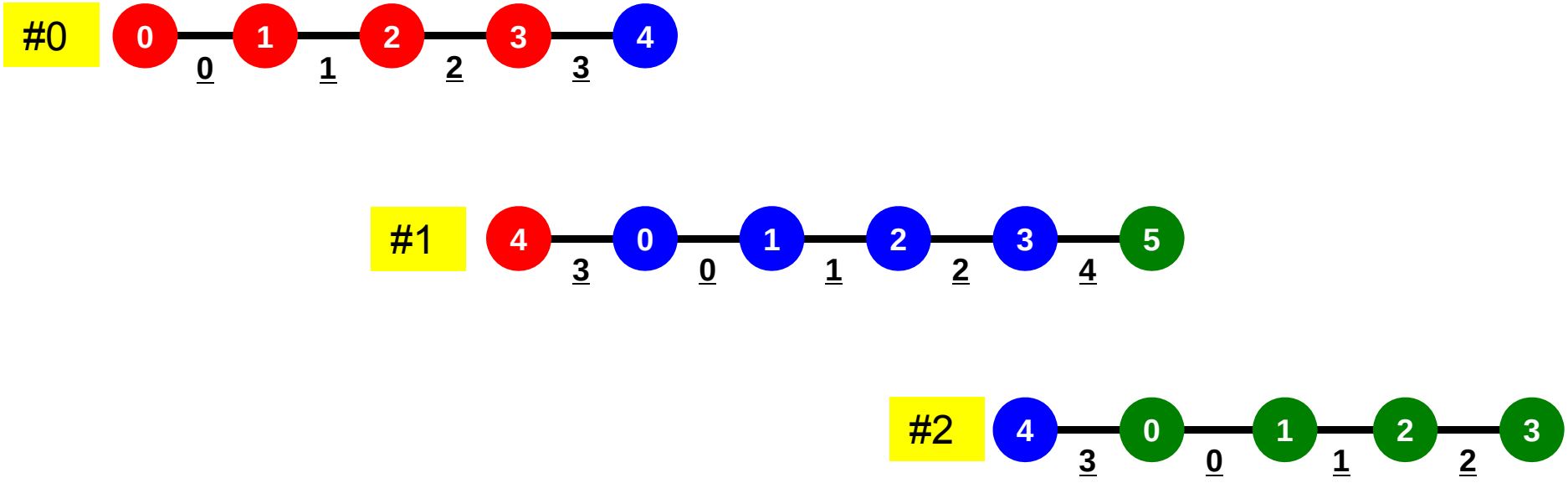


一次元問題:11要素，12節点，3領域



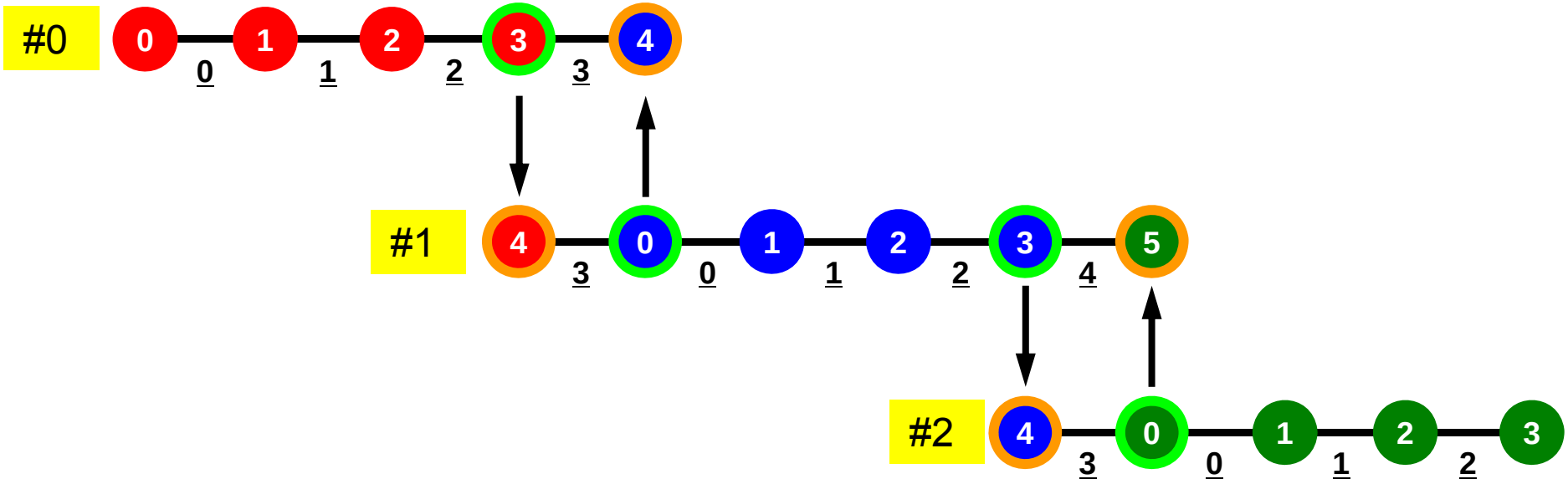
一次元問題：11要素，12節点，3領域

局所番号：節点・要素とも0からふる

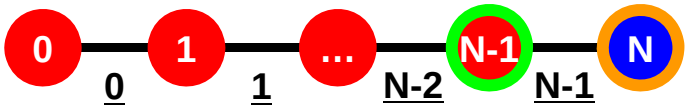


一次元問題:11要素，12節点，3領域

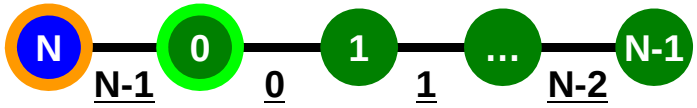
外点･境界点



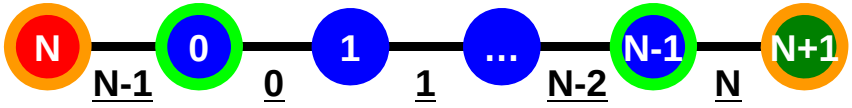
一次元問題：一般的な局所番号の付け方



#0: N+1節点，N要素



#PETot-1: N+1節点，N要素

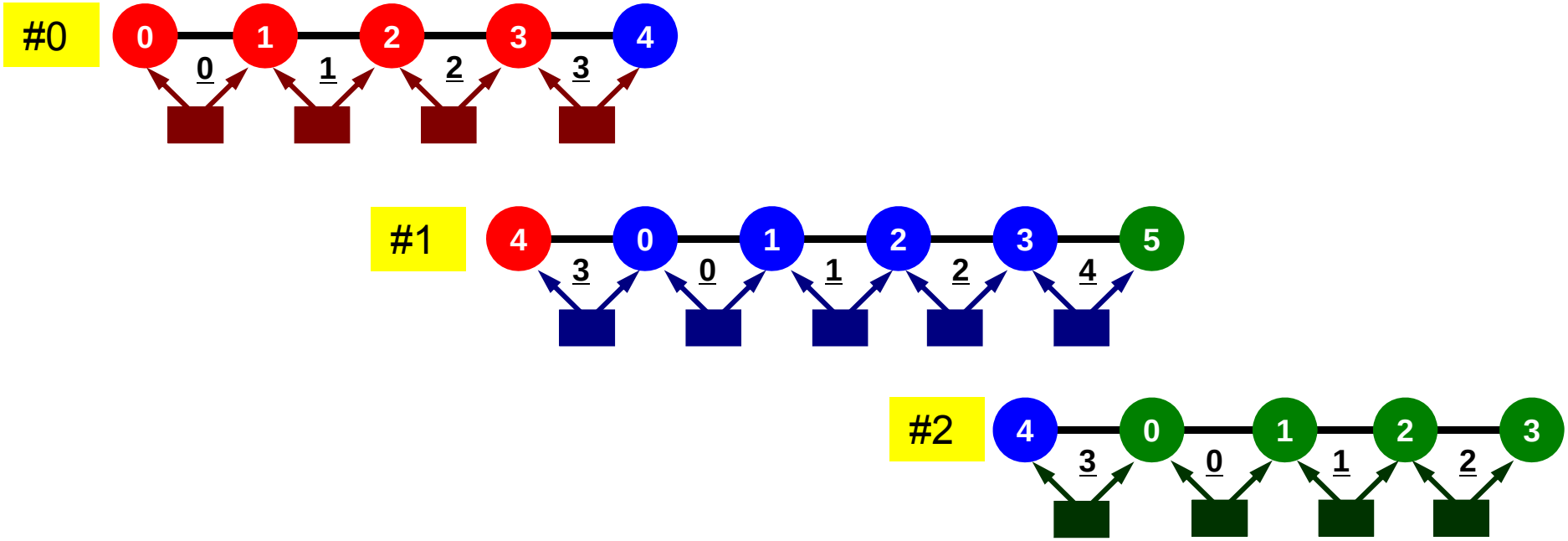


一般の領域：
N+2節点，N+1要素

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

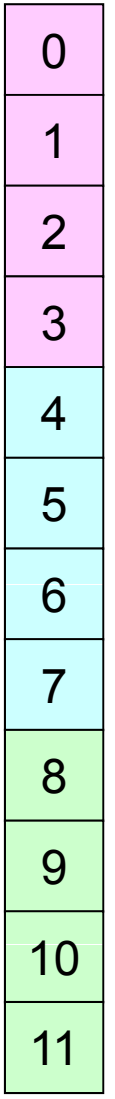
前処理：対角スケーリング

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];
    }
```

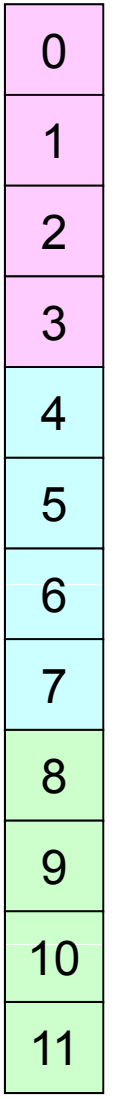
```
/*
//-- {x}= {x} + ALPHA*{p}
//   {r}= {r} - ALPHA*{q}
*/
  for(i=0;i<N;i++){
      U[i]    += Alpha * W[P][i];
      W[R][i] -= Alpha * W[Q][i];
  }
```



内積

全体で和をとる必要がある⇒通信?

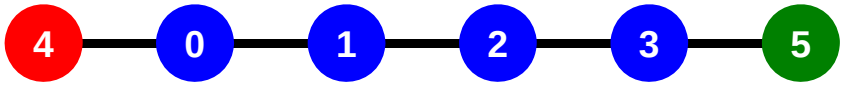
```
/*
//-- ALPHA= RHO / {p}{q}
*/
  C1 = 0.0;
  for(i=0;i<N;i++){
      C1 += W[P][i] * W[Q][i];
  }

  Alpha = Rho / C1;
```



行列ベクトル積

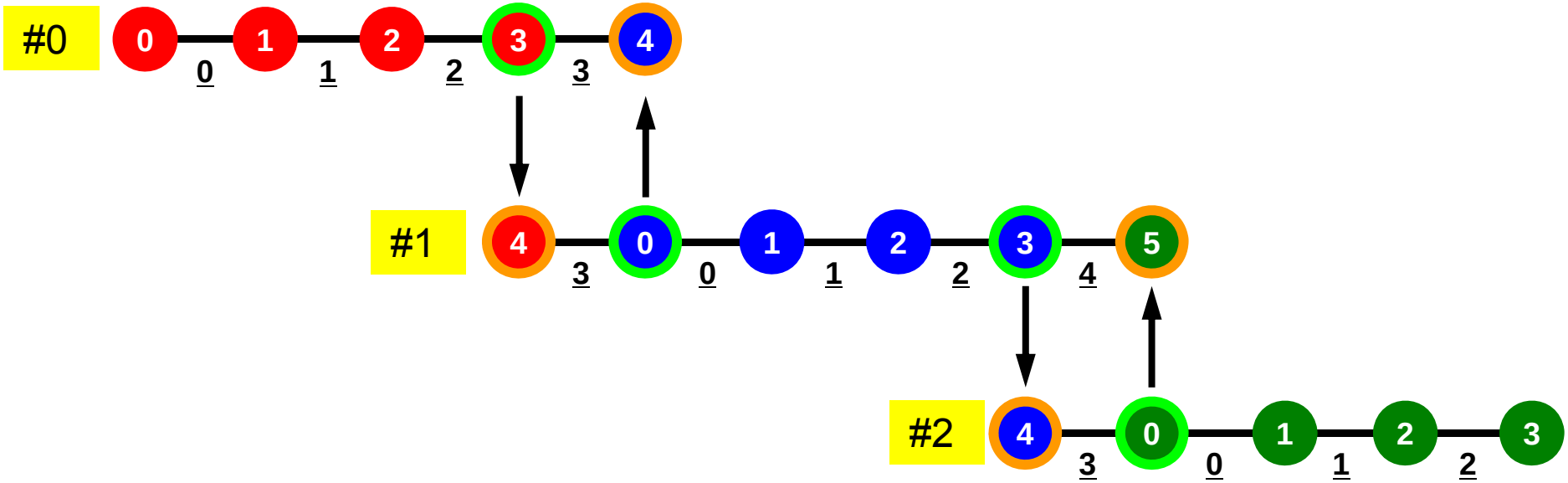
外点の値(最新のp)が必要⇒1対1通信

```
/*
//-- {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]];
          }
      }
```

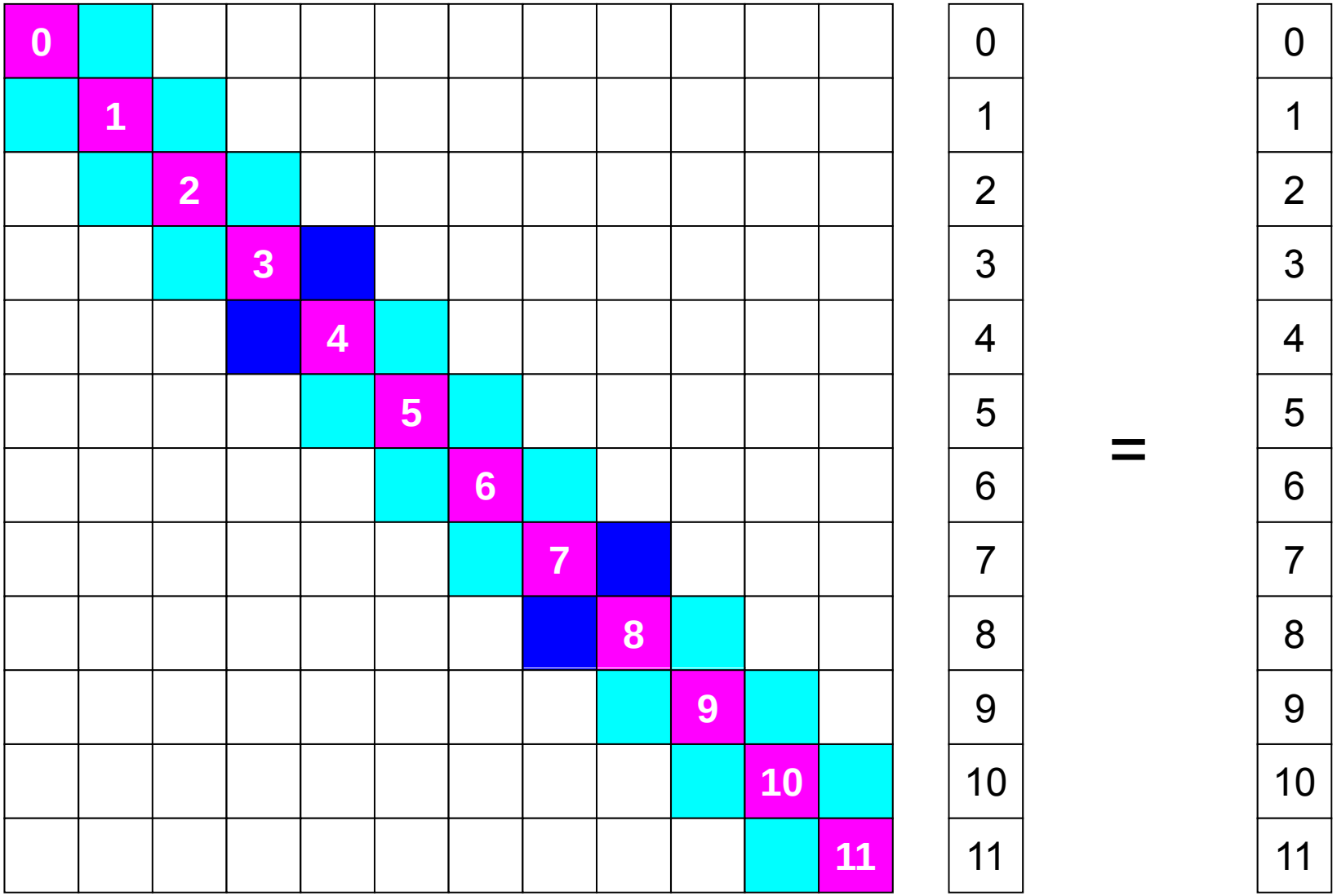


一次元問題:11要素，12節点，3領域

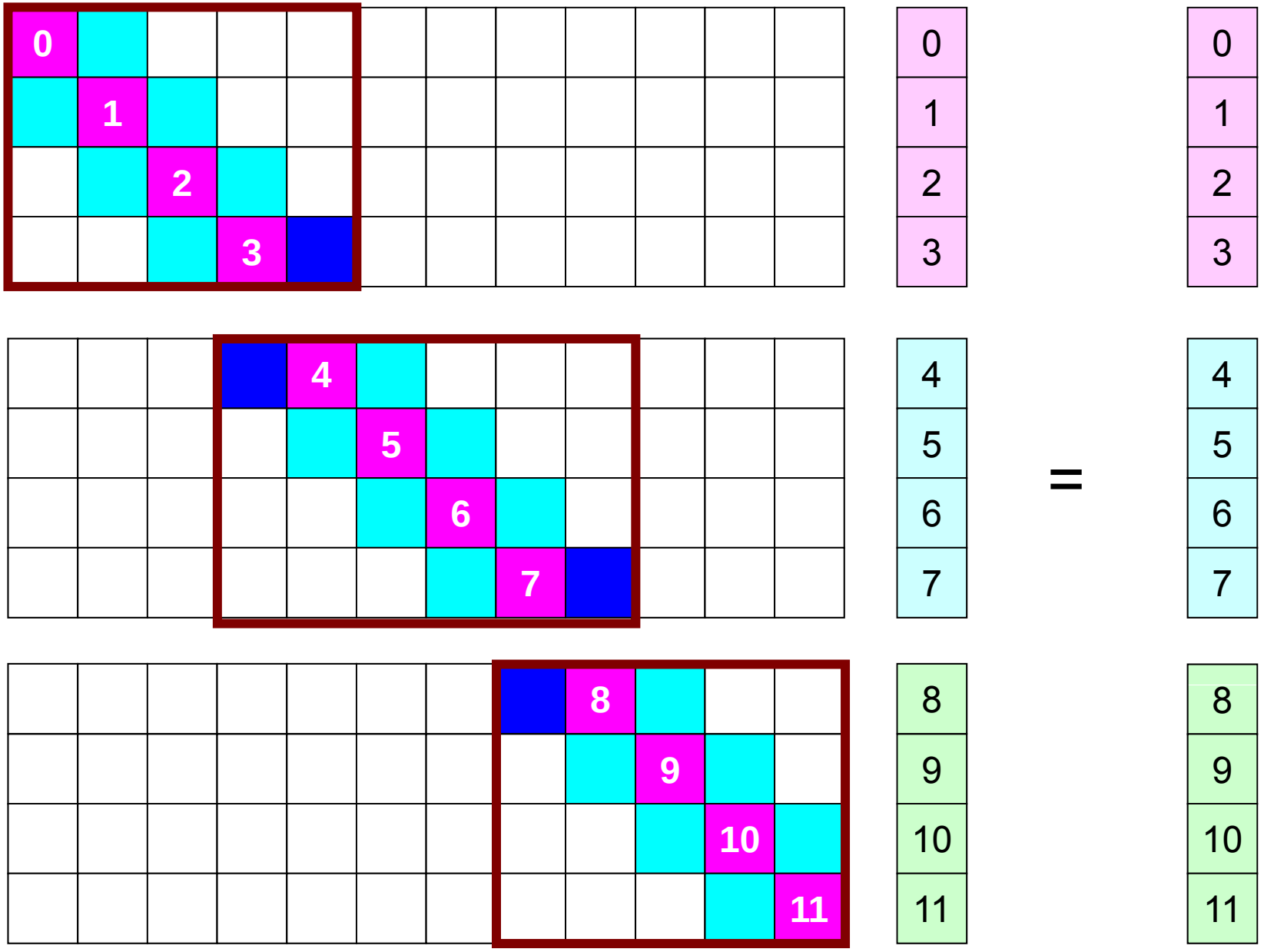
外点･境界点



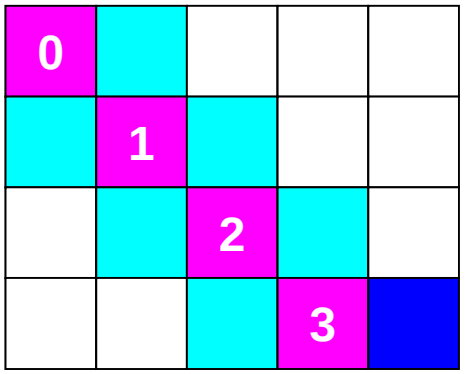
行列ベクトル積：ローカルに計算実施可能

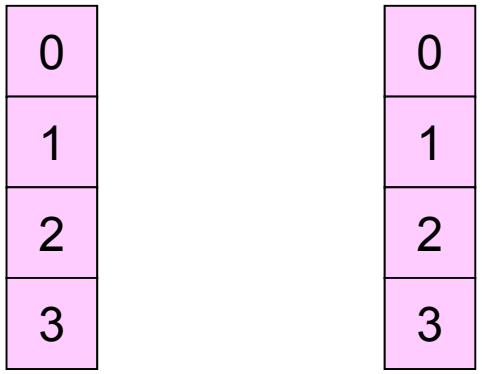


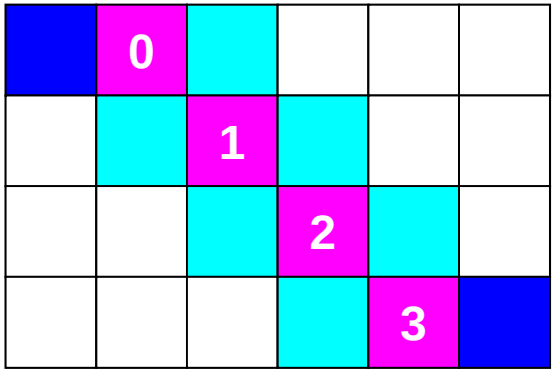
行列ベクトル積：ローカルに計算実施可能

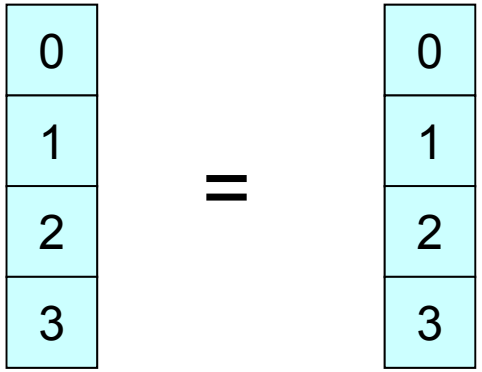


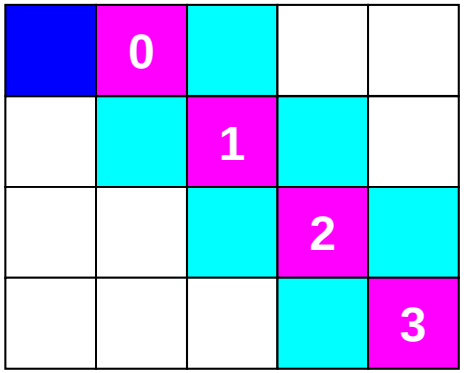
行列ベクトル積：ローカルに計算実施可能

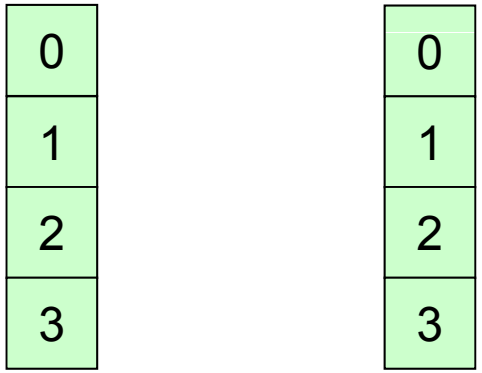




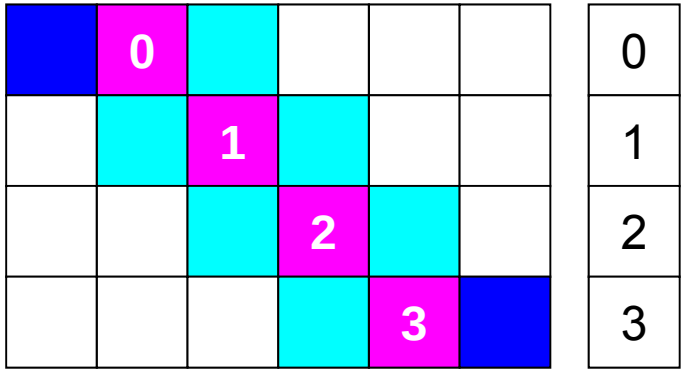




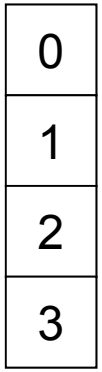


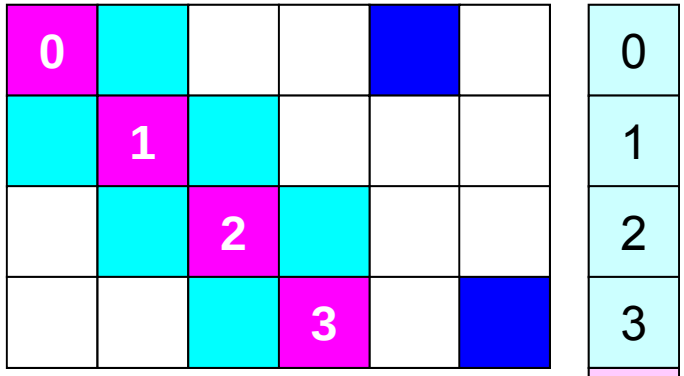


行列ベクトル積：ローカル計算 #1

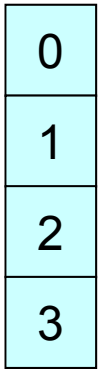


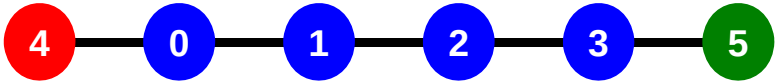
=



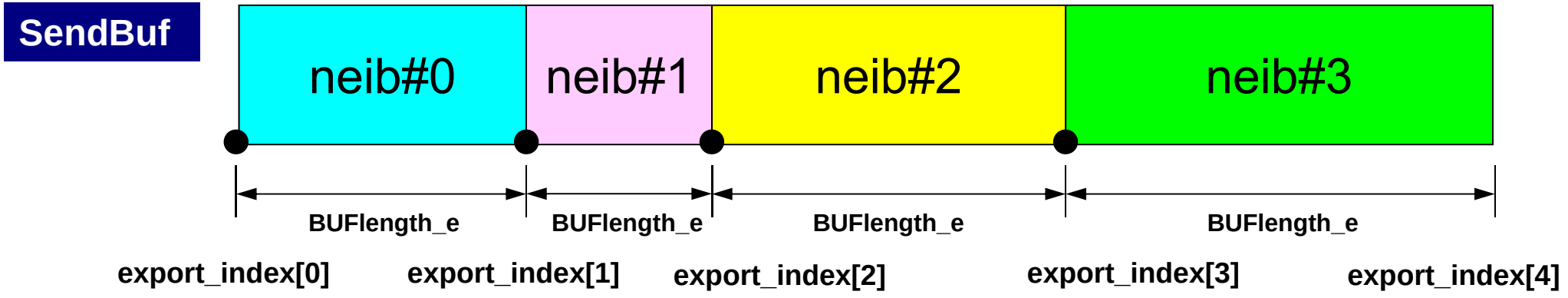


=





送信（MPI_Isend/Irecv/Waitall）



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];
   }
}

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);
```

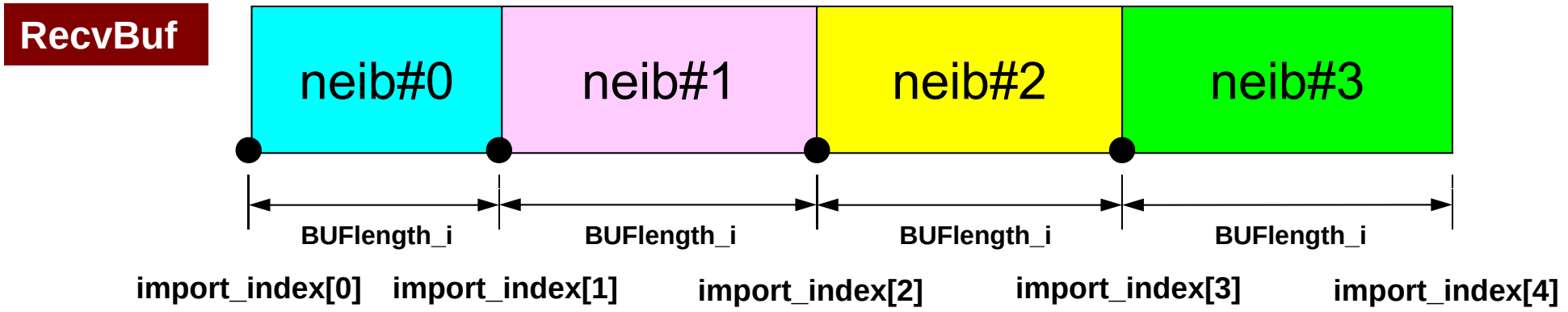
送信バッファへの代入

受信（MPI_Isend/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



- 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（1/10）
諸変数

MPIを使用するときの「おまじない」

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

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

  int NE, N, NP, NPLU, IterMax, NEg, Ng, errno;
  double dX, Resid, Eps, Area, F, Young;
  double X1, X2, U1, U2, DL, Strain, Sigma, Ck;
  double *U, *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(2/10)

制御データ読み込み

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

/*
//-- CONTROL data
*/

    ierr = MPI_Init(&argc, &argv);                          MPI初期化：必須
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);           全プロセス数：PETot
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);          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, &F, &Area, &Young);
      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(&F      , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ierr = MPI_Bcast(&Area   , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ierr = MPI_Bcast(&Young  , 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(2/10)

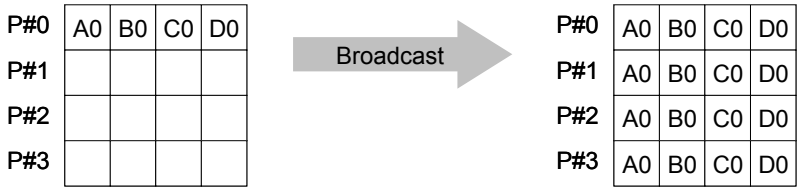
制御データ読み込み

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

/*
//-- CONTROL data
*/

    ierr = MPI_Init(&argc, &argv);                          MPI初期化：必須
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);           全プロセス数：PETot
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);          自分のランク番号（0～PETot-1）：MyRank

    if(MyRank == 0){                                        MyRank=0のとき制御データを読み込む
      fp = fopen("input.dat", "r");
      assert(fp != NULL);
      fscanf(fp, "%d", &NEg);
      fscanf(fp, "%lf %lf %lf %lf", &dX, &F, &Area, &Young);
      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(&F      , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ierr = MPI_Bcast(&Area   , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ierr = MPI_Bcast(&Young  , 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(2/10)
制御データ読み込み

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

/*
//-- CONTROL data
*/

    ierr = MPI_Init(&argc, &argv);                          MPI初期化：必須
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);           全プロセス数：PETot
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);          自分のランク番号（0～PETot-1）：MyRank

    if(MyRank == 0){                                        MyRank=0のとき制御データを読み込む
      fp = fopen("input.dat", "r");
      assert(fp != NULL);
      fscanf(fp, "%d", &NEg);                               Neg：「全」要素数
      fscanf(fp, "%lf %lf %lf %lf", &dX, &F, &Area, &Young);
      fscanf(fp, "%d", &IterMax);
      fscanf(fp, "%lf", &Eps);
      fclose(fp);
    }

    ierr = MPI_Bcast(&NEg    , 1, MPI_INT, 0, MPI_COMM_WORLD);  0番ランクから各プロセスに値を送る
    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(&F      , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ierr = MPI_Bcast(&Area   , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ierr = MPI_Bcast(&Young  , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ierr = MPI_Bcast(&Eps    , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

MPI_Bcast



- グループ（コミュニケータ）「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（通信グループ）を指定する

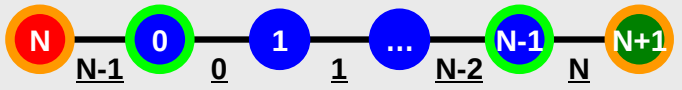
プログラム：1d.c(3/10)
局所分散メッシュデータ

```
/*
//-- 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
*/
    U    = 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(3/10)

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
*/
    U    = 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));
```

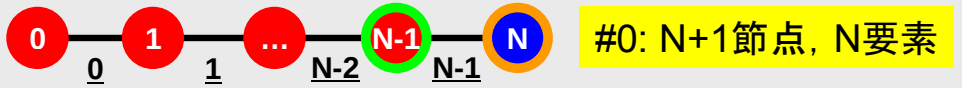


一般の領域：
N+2節点，N+1要素

プログラム：1d.c(3/10)

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
*/
    U    = 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(3/10)

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
*/
    U    = 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));
```

#PETot-1: N+1節点，N要素

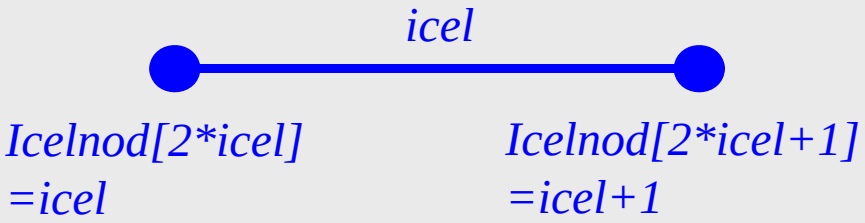
プログラム：1d.c(3/10)
局所分散メッシュデータ

```
/*
//-- 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
*/
    U    = calloc(NP, sizeof(double));          NでなくNPで配列を定義している点に注意
    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(4/10)

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;
  }
}
```



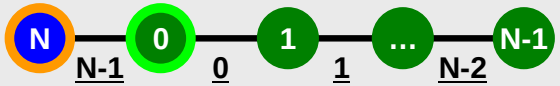
プログラム：1d.c(4/10)

配列初期化，要素～節点

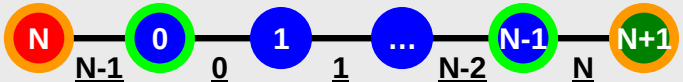
```
for(i=0;i<NP;i++)     U[i] = 0.0;
for(i=0;i<NP;i++)  Diag[i] = 0.0;
for(i=0;i<NP;i++)   Rhs[i] = 0.0;
for(k=0;k<2*NP-2;k++)  AMat[k] = 0.0;

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

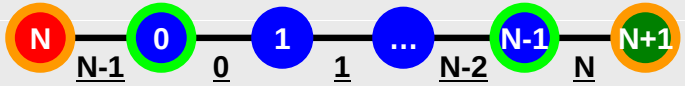
プログラム：1d.c（5/10）

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

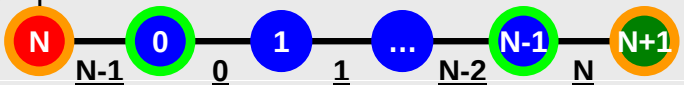
プログラム：1d.c（6/10）

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;}
  }
}

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

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

#0: N+1節点，N要素

#PETot-1: N+1節点，N要素



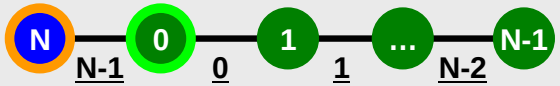
一般の領域：
N+2節点，N+1要素

プログラム：1d.c(7/10)

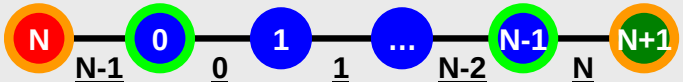
通信テーブル定義

```
/*
//-- 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;

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

    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;

    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);
```

#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。

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

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

MPI_Waitall

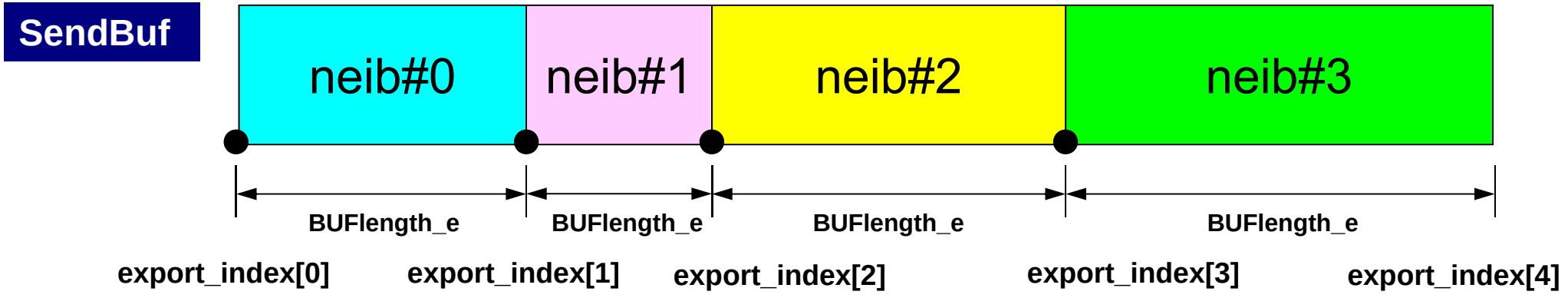
- 1対1非ブロッキング通信関数である「MPI_Isend」と「MPI_Irecv」を使用した場合，プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_Waitall」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_Waitall」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば，「MPI_Isend」と「MPI_Irecv」を同時に同期してもよい。
  - 「MPI_Isend/Irecv」で同じ通信識別子を使用すること
- 「MPI_Barrier」と同じような機能であるが，代用はできない。
  - 実装にもよるが，「request」，「status」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 O 状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE,count））
    MPI_STATUS_SIZE：“mpif.h”,”mpi.h”で定められるパラメータ：C言語については後述

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）

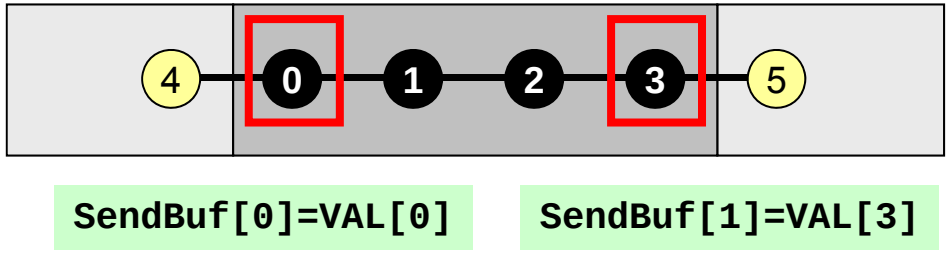


export_index[neib]～export_index[neib+1]-1番目のexport_itemがneib番目の隣接領域に送信される

```
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];
   }
}

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);
```

送信バッファへの代入

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

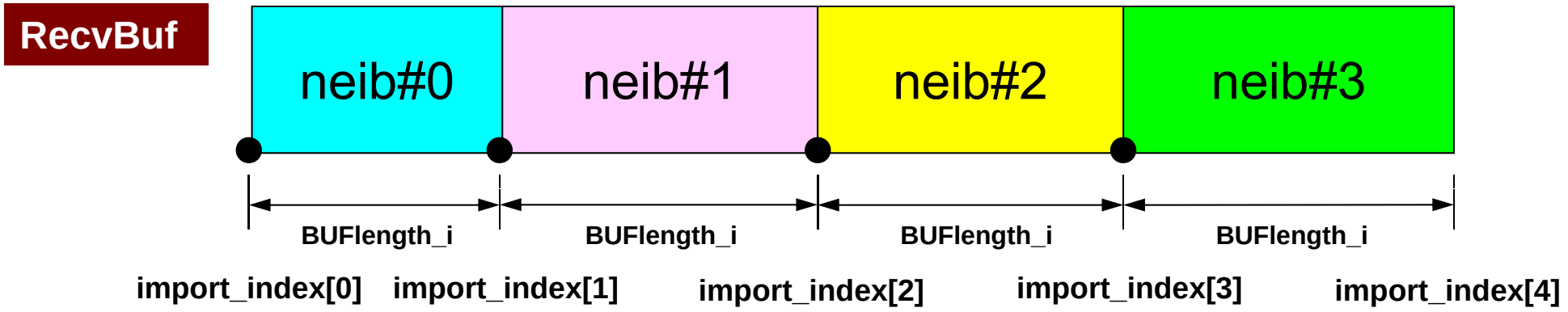
- 受信相手
  - NeibPETot，NeibPE[neib]
- それぞれの受信相手から受け取るメッセージサイズ
  - import_index[neib], neib= 0, NeibPETot-1
- 「外点」番号
  - import_item[k], k= 0, import_index[NeibPETot]-1
- それぞれの受信相手から受け取るメッセージ
  - RecvBuf[k], k= 0, import_index[NeibPETot]-1

受信（MPI_Isend/Irecv/Waitall）

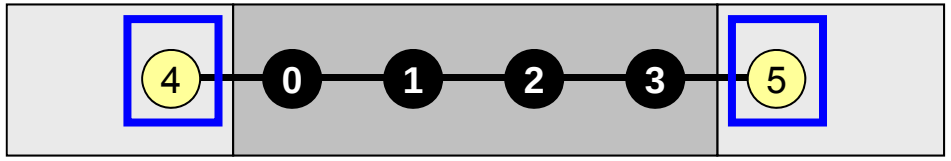
```
for (neib=0; neib<NeibPETot; neib++){
   tag= 0;
   iS_i= import_index[neib];
   iE_i= import_index[neib+1];
   BUFlength_i= iE_i - iS_i

   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



受信:一次元問題

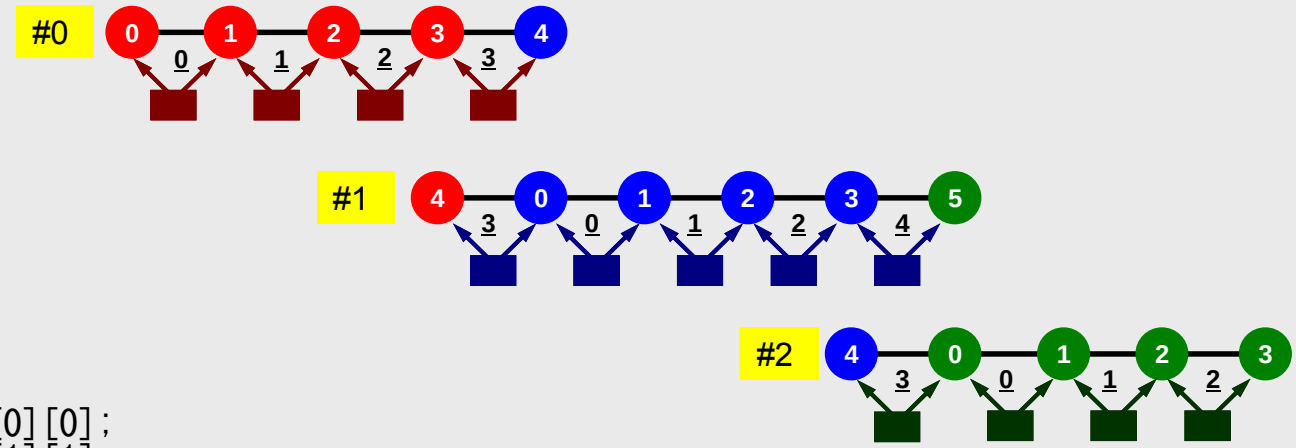


- 受信相手
  - NeibPETot, NeibPE[neib]
    - NeibPETot=2, NeibPE[0]= my_rank-1, NeibPE[1]= my_rank+1
- それぞれの受信相手から受け取るメッセージサイズ
  - import_index[neib], neib= 0, NeibPETot-1
    - import_index[0]=0, import_index[1]= 1, import_index[2]= 2
- 「外点」番号
  - import_item[k], k= 0, import_index[NeibPETot]-1
    - import_item[0]= 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]

プログラム：1d.c（8/10）

全体マトリクス生成：1CPUのときと全く同じ：各要素→一様

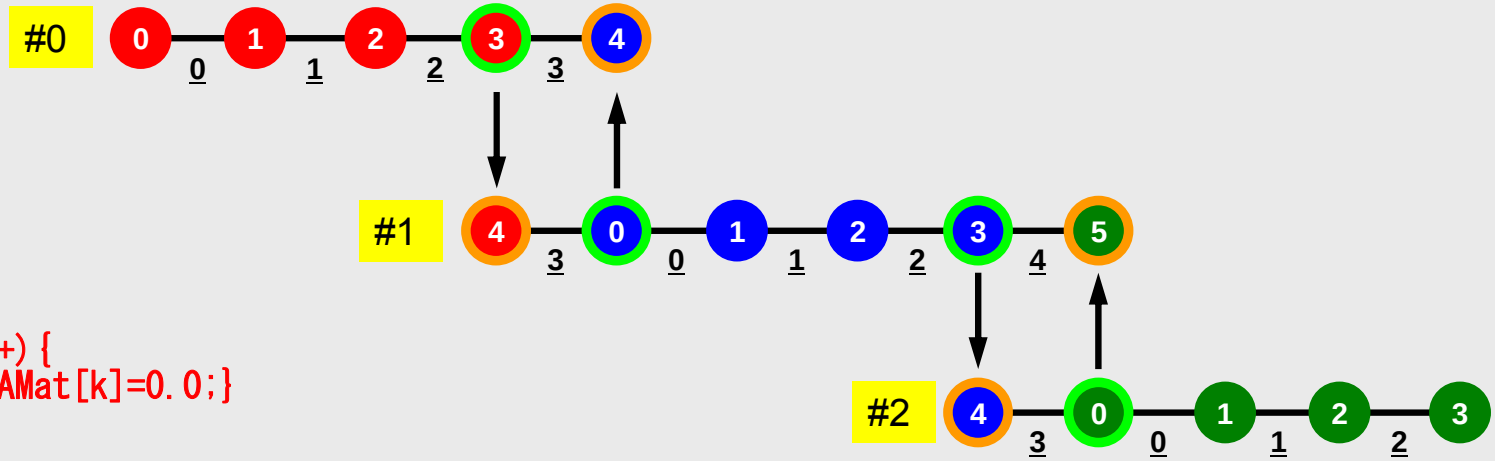
```
/*
// +-----------------+
// | MATRIX assemble |
// +-----------------+
*/
    for(icel=0;icel<NE;icel++){
      in1= IcelNod[2*icel];
      in2= IcelNod[2*icel+1];
      DL = dX;

      Ck= Area*Young/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];
    }
```



プログラム：1d.c（9/10）

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;}
      }
    }

/* X=Xmax */
    if (MyRank==PETot-1){
      i=N-1;
      Rhs[i]= F;
    }
```



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, …
  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

```
/*
//-- {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_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];
           P[kk]=RecvBuf[k];
         }
       }
```

行列ベクトル積(2/2)

{q}= [A]{p}

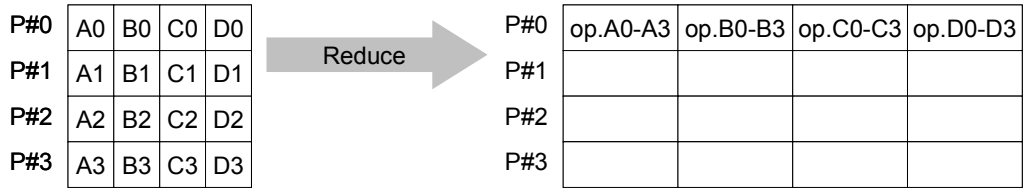
```
MPI_Waitall(NeibPETot, RequestSend, StatSend);

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「recbuf」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）

```
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)
```

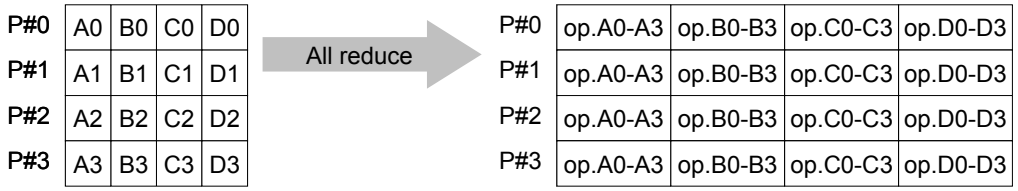
```
real(kind=8):: X0, XSUM

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，最大値等を計算して，全プロセスに配信

- **MPI_Allreduce (sendbuf,recvbuf,count,datatype,op, 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
  - **comm** 整数 I コミュニケータを指定する

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", "### DISPLACEMENT");
    for(i=0;i<N;i++){
      printf("%3d%8d%16.6E¥n", MyRank, i+1, U[i]);
    }

    ierr = MPI_Finalize();
    return ierr;
}
```

書き出し:順番通りには出てこない

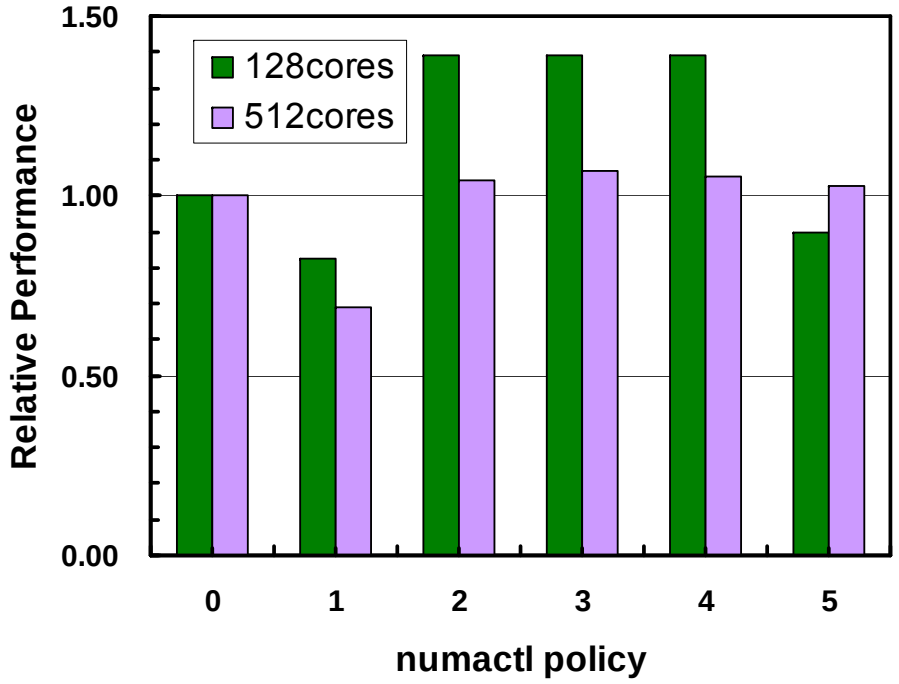
```
### DISPLACEMENT
  0        1    0.000000E+00
  0        2    1.000000E-01
  0        3    2.000000E-01
  0        4    3.000000E-01
  0        5    4.000000E-01
  0        6    5.000000E-01
  0        7    6.000000E-01
  0        8    7.000000E-01
  0        9    8.000000E-01
  0       10    9.000000E-01
  0       11    1.000000E-00

### DISPLACEMENT
  1        1    1.100000E+00
  1        2    1.200000E+00
  1        3    1.300000E+00
  1        4    1.400000E+00
  1        5    1.500000E+00
  1        6    1.600000E+00
  1        7    1.700000E+00
  1        8    1.800000E+00
  1        9    1.900000E+00
  1       10    2.000000E+00
```

```
### DISPLACEMENT
  3        1    3.100000E+00
  3        2    3.200000E+00
  3        3    3.300000E+00
  3        4    3.400000E+00
  3        5    3.500000E+00
  3        6    3.600000E+00
  3        7    3.700000E+00
  3        8    3.800000E+00
  3        9    3.900000E+00
  3       10    4.000000E+00

### DISPLACEMENT
  2        1    2.100000E+00
  2        2    2.200000E+00
  2        3    2.300000E+00
  2        4    2.400000E+00
  2        5    2.500000E+00
  2        6    2.600000E+00
  2        7    2.700000E+00
  2        8    2.800000E+00
  2        9    2.900000E+00
  2       10    3.000000E+00
```

- 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**

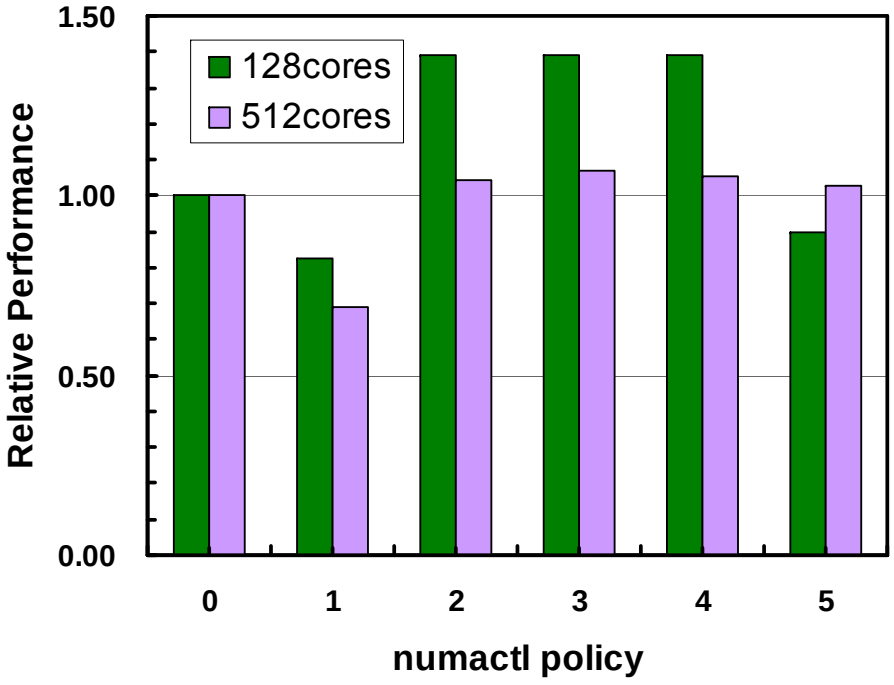
numactlの影響

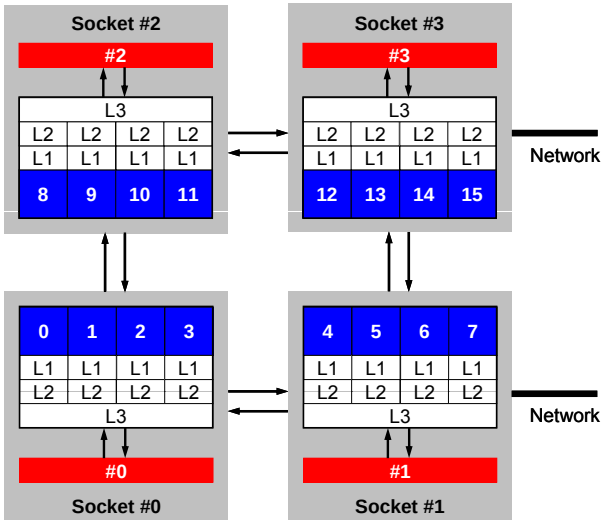
- T2K，有限要素法アプリケーション，Strong Scaling
- 128コア：1コアあたりの問題サイズは512コアの4倍
  - メモリへの負担大
- POLICY=0：何も指定しない場合
- 相対性能：大きいほど良い




```
#@$-r HID-org
#@$-q h08nkl32
#@$-N 24
#@$-J T16
#@$-e err
#@$-o x384-40-1-a.lst
#@$-lM 27GB
#@$-lE 03:00:00
#@$-s /bin/sh
#@$

cd /XXX
mpirun ./numarun.sh ./sol
exit
```

numarun.sh（シェル・スクリプト）の中身





Policy:1
```
#!/bin/bash
MYRANK=$MXMPI_ID MPIのプロセス番号
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --interleave=all $@
```

Policy:2
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --interleave=$SOCKET $@
```

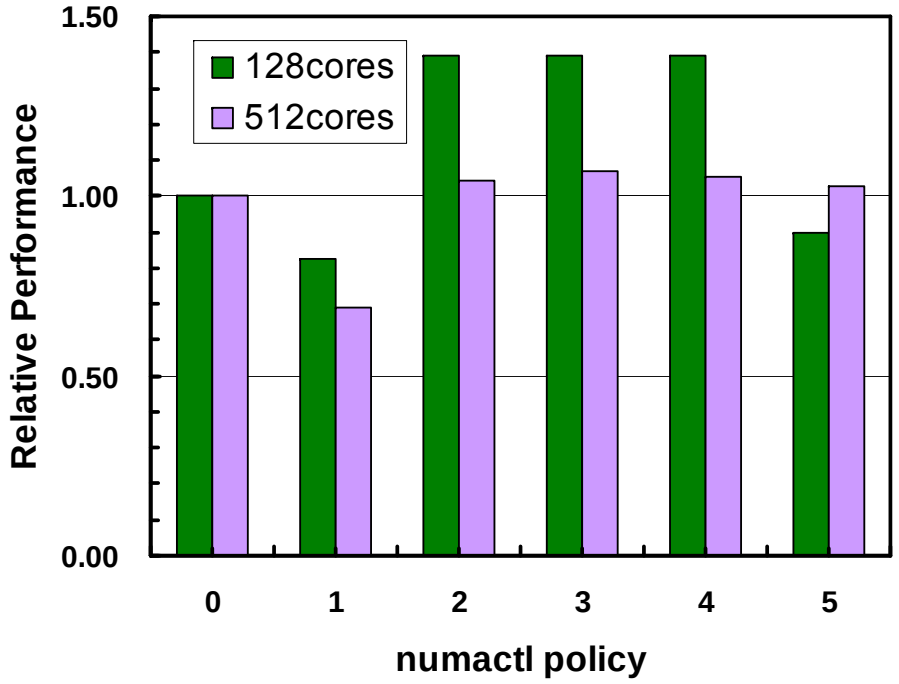
Policy:3
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --membind=$SOCKET $@
```

Policy:4
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --cpunodebind=$SOCKET --localalloc $@
```

Policy:5
```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
SOCKET=$(expr $MYVAL % 4)
numactl --localalloc $@
```

numactlの影響

- 128コア：1コアあたりの問題サイズは512コアの4倍
  - メモリへの負担大
- メモリへの負担が大きい場合，特にnumactl指定の影響は大




```
#@$-r HID-org
#@$-q h08nkl32
#@$-N 24
#@$-J T16
#@$-e err
#@$-o x384-40-1-a.lst
#@$-lM 27GB
#@$-lE 03:00:00
#@$-s /bin/sh
#@$

cd /XXX
mpirun ./numarun.sh ./sol
exit
```

本講義での例:<$T-S2r>

go2.sh-numarun.sh (policy:4相当)

```
#!/bin/bash
MYRANK=$MXMPI_ID
MYVAL=$(expr $MYRANK / 4)
NODE=$(expr $MYVAL % 4)
numactl --cpunodebind=$NODE --localalloc $@
```

go.sh (policy:5相当):直接書き込んである

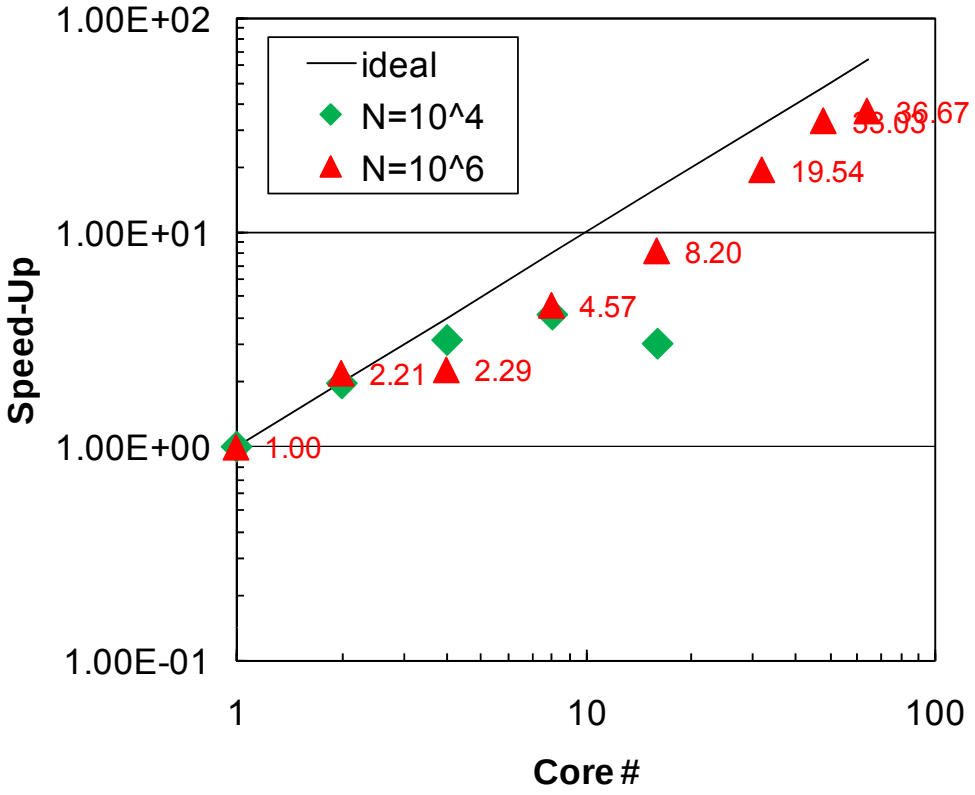
```
#!/bin/bash
numactl --localalloc $@
```

計算結果(1次元):CG法部分

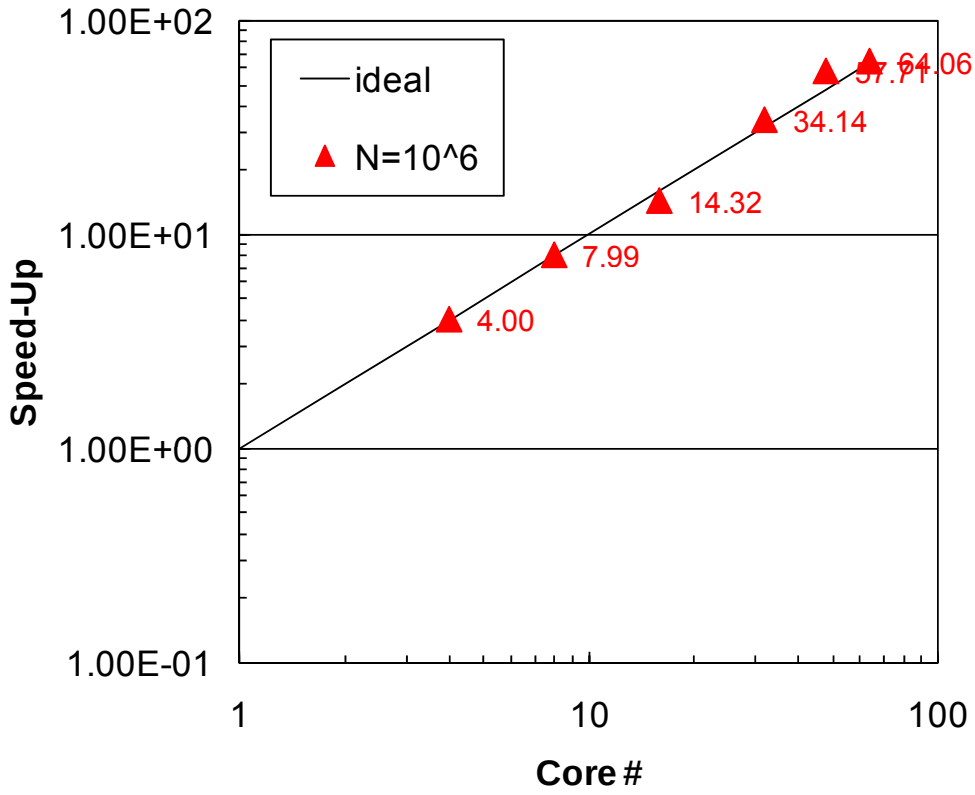
go2.sh+numarun.sh(policy-4)

N=10^6の場合は1,000回反復に要する時間

1コアを基準



4コアを基準



計算結果(3次元):NX=NY=NZ=128，16コア

FORTRAN，SSOR，iterPREmax=2，KMETIS

numactl	反復回数	計算時間（秒）
無し	245	372.9
Policy-4	245	244.3
Policy-5	245	308.9

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