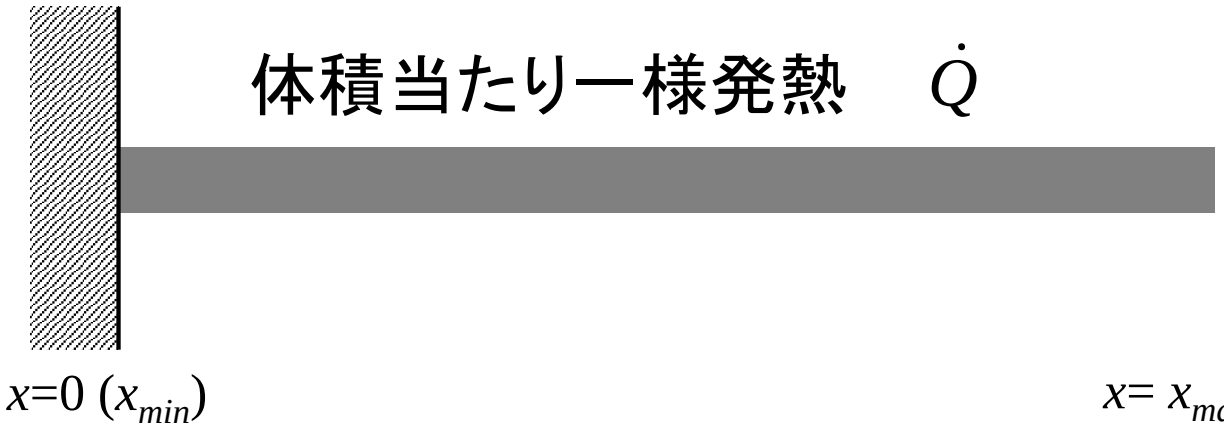
並列有限要素法による
一次元定常熱伝導解析プログラム
C言語編

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
- プログラムの説明
- 計算例

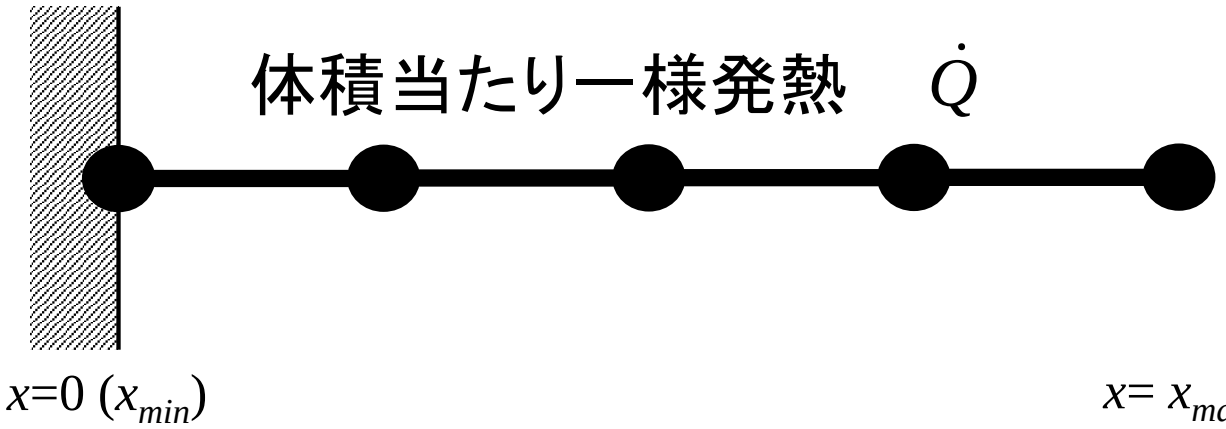
対象とする問題：一次元熱伝導問題



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

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

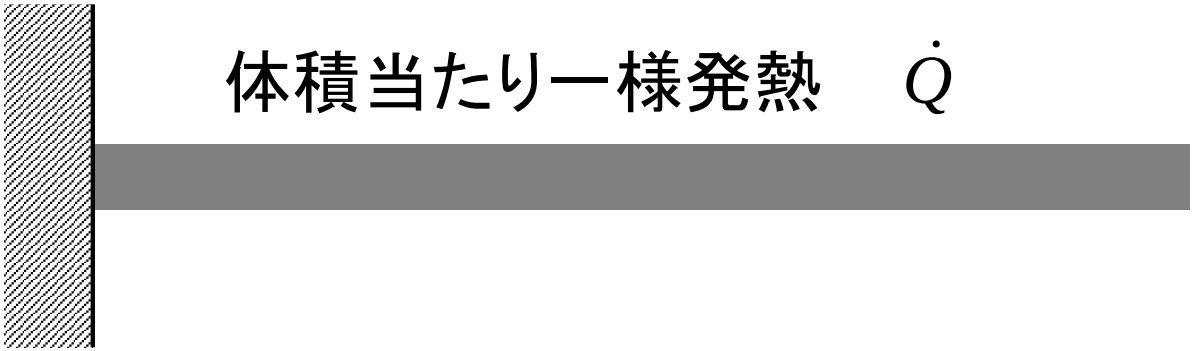
対象とする問題：一次元熱伝導問題



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

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

解析解



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

x=0 (x_{min})

$$T=0 @ x=0$$

x= x_{max}

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

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

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

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

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

ファイルコピー，コンパイル

FORTRANユーザー

```
>$ cd /work/gt00/t00XXX/pFEM
>$ cp /work/gt00/z30088/class_eps/F/s2r-f.tar .
>$ tar xvf s2r-f.tar
```

Cユーザー

```
>$ cd /work/gt00/t00XXX/pFEM
>$ cp /work/gt00/z30088/class_eps/C/s2r-c.tar .
>$ tar xvf s2r-c.tar
```

ディレクトリ確認・コンパイル

```
>$ cd mpi/S2-ref
>$ mpiifort -O3 -xCORE-AVX2 -align array32byte 1d.f
>$ mpicc -O3 -xCORE-AVX2 -align 1d.c
```

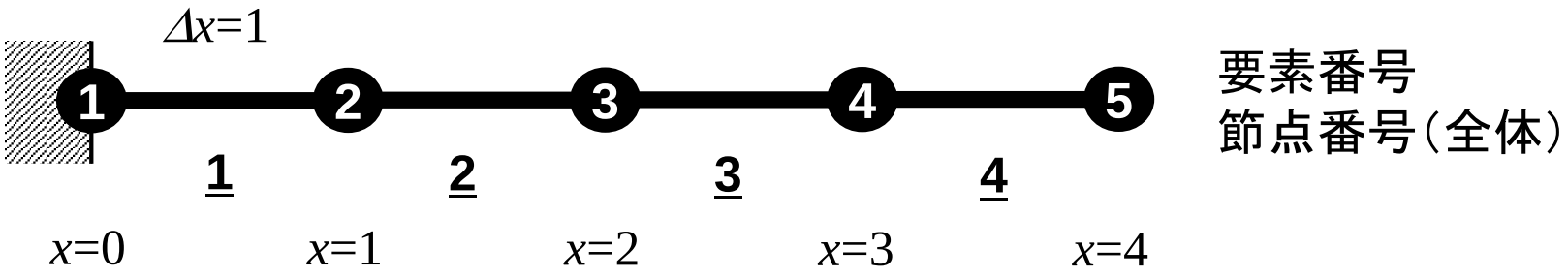
このディレクトリを本講義では <$O-S2r> と呼ぶ。

<$O-S2r> = <$O-TOP>/mpi/S2-ref

制御ファイル：input.dat

制御ファイル　**input.dat**

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



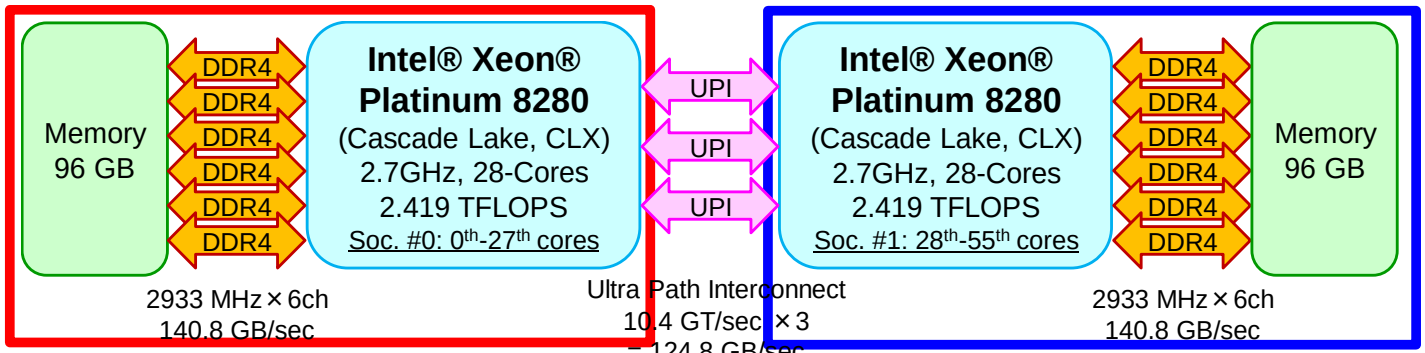
g16.sh: 8-nodes, 256-cores, 16x2x8

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=256
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
export I_MPI_PIN_PROCESSOR_LIST=0-15,28-43
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
```

256/8= 32 cores/node

32-cores are randomly selected from 56-cores on the node

32-cores on each socket are assigned. A little bit more stable



Socket #0: 0th-27th Cores

Socket #1: 28th-55th Cores

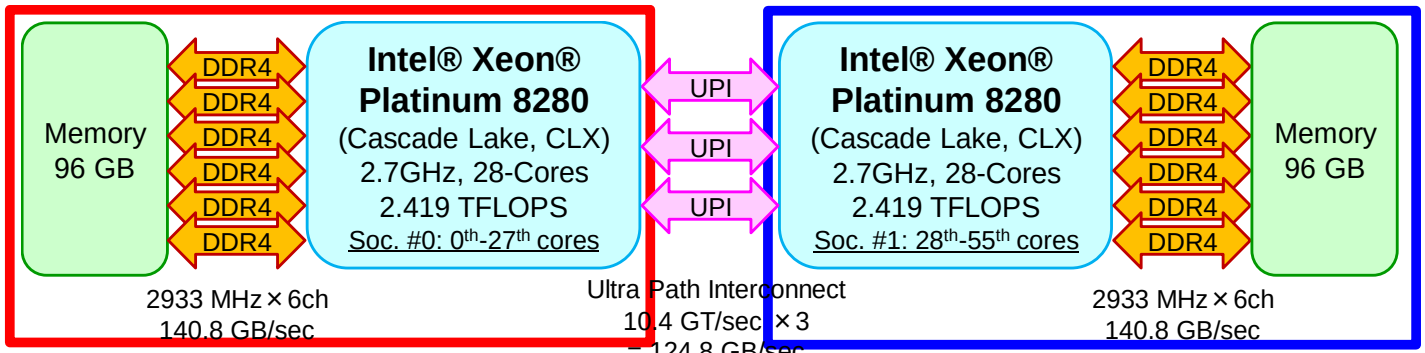
g24.sh: 8-nodes, 384-cores, 24x2x8

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=384
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
```

384/8= 48 cores/node

48-cores are randomly selected from 56-cores on the node

24-cores on each socket are assigned. A little bit more stable



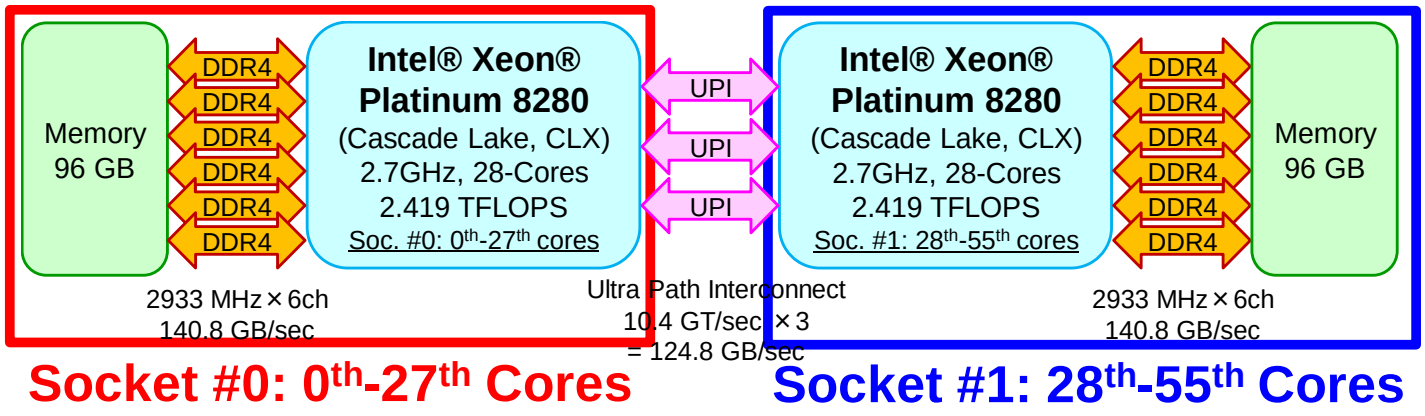
Socket #0: 0th-27th Cores

Socket #1: 28th-55th Cores

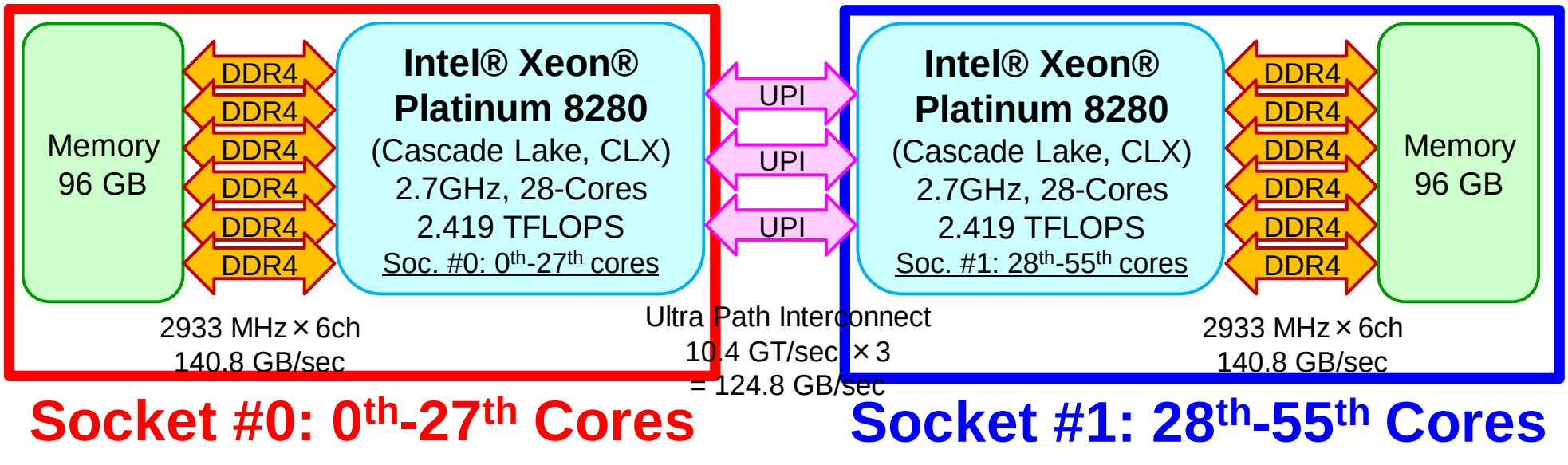
g28.sh: 8-nodes, 448-cores, 28x2x8

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=448
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
export I_MPI_PIN_PROCESSOR_LIST=0-55
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
```

448/8= 56 cores/node



NUMA Architecture



- Oakbridge-CX (OBCX)
  - 2 Sockets (CPU's) of Intel CLX
  - 各ソケットは28コア，合計56コア
- NUMAアーキテクチャ（Non-Uniform Memory Access）
  - メモリは各CPUに搭載されていて独立，異なるCPUのローカルメモリ上のデータをアクセスすることは可能
  - ローカルメモリ上のデータを使って計算するのが効率的
    - `numactl -l` ：ローカルメモリ使用，本オプションによる遅くなる場合有り

「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
- プログラムの説明
- 計算例

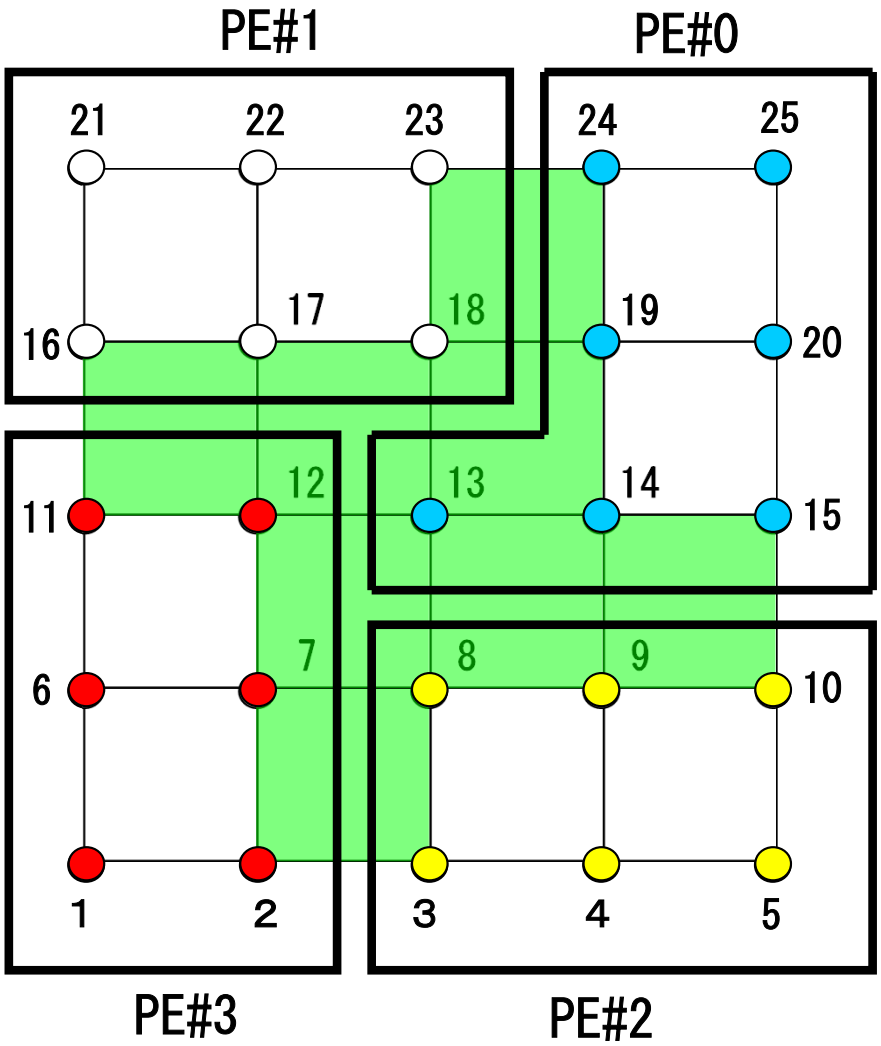
並列有限要素法の局所データ構造

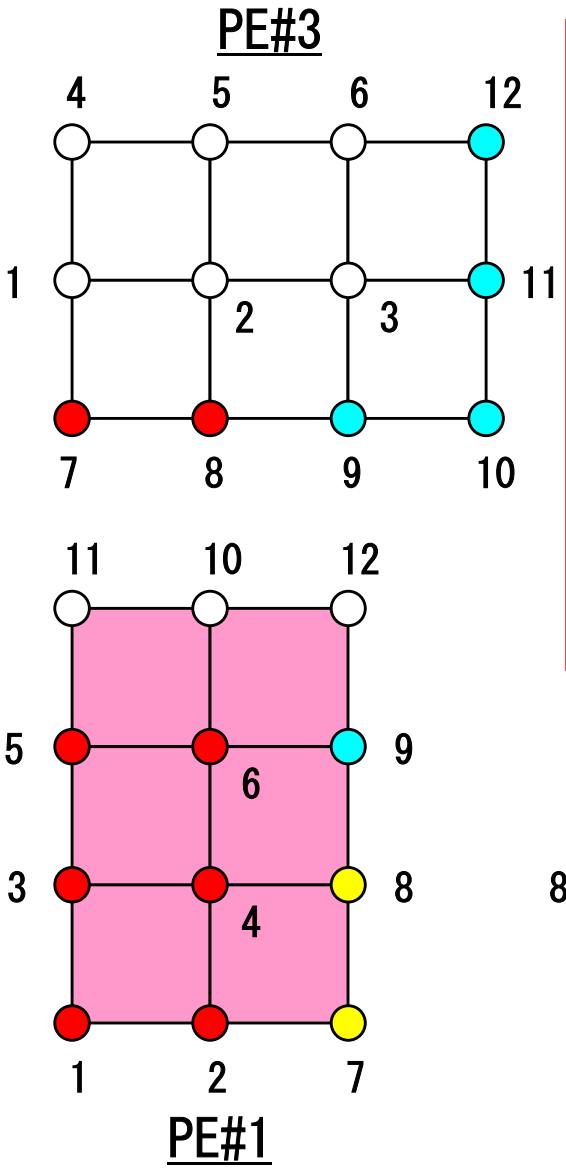
- 節点ベース：Node-based partitioning
- 局所データに含まれるもの：
  - その領域に本来含まれる節点
  - それらの節点を含む要素
  - 本来領域外であるが，それらの要素に含まれる節点
- 節点は以下の3種類に分類
  - 内点：Internal nodes　　その領域に本来含まれる節点
  - 外点：External nodes　　本来領域外であるがマトリクス生成に必要な節点
  - 境界点：Boundary nodes　他の領域の「外点」となっている節点
- 領域間の通信テーブル

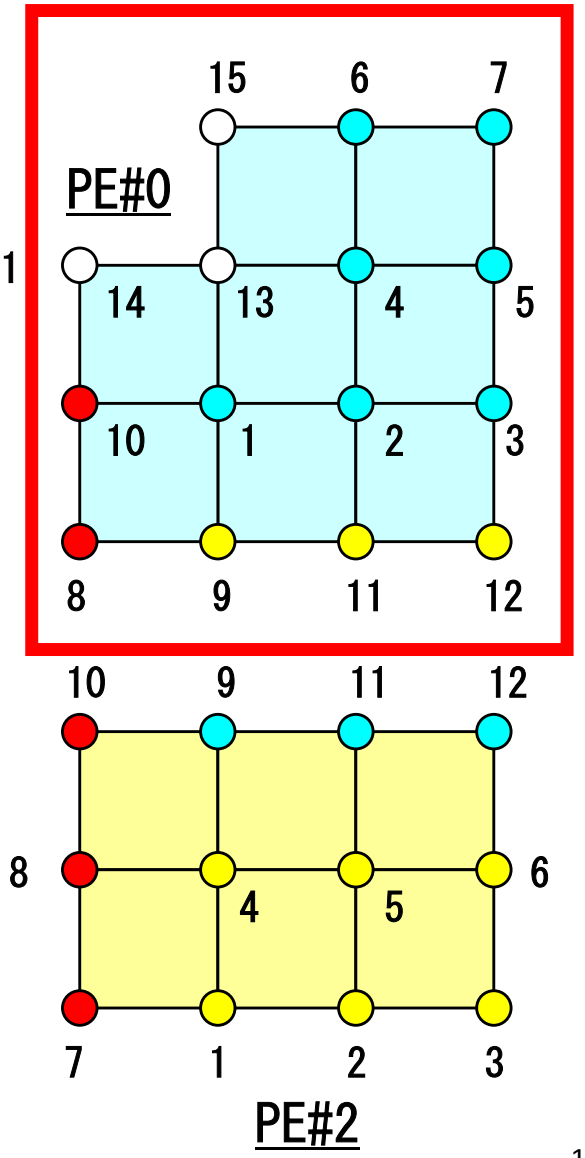
- 領域間の接続をのぞくと，大域的な情報は不要
  - 有限要素法の特性：要素で閉じた計算

Node-based Partitioning

internal nodes - elements - external nodes



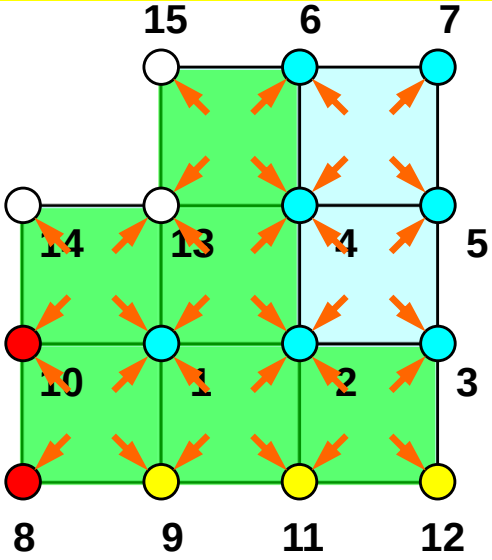


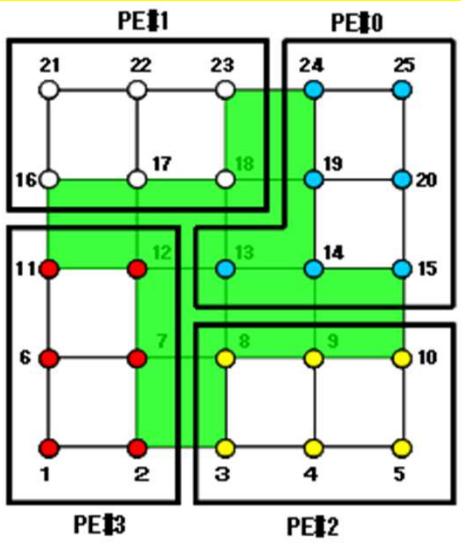


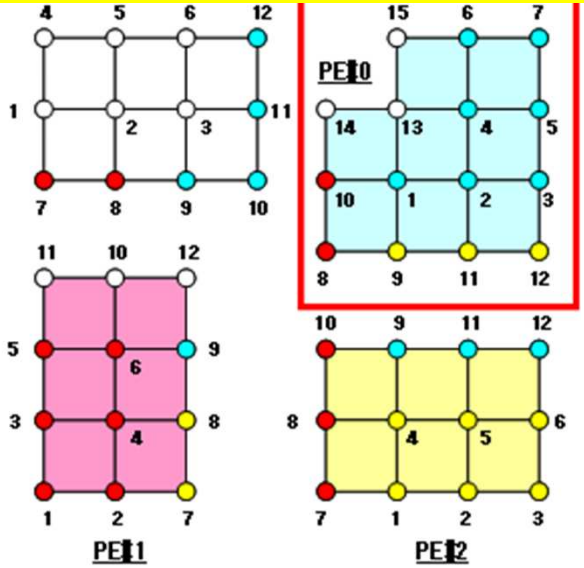
Node-based Partitioning

internal nodes - elements - external nodes

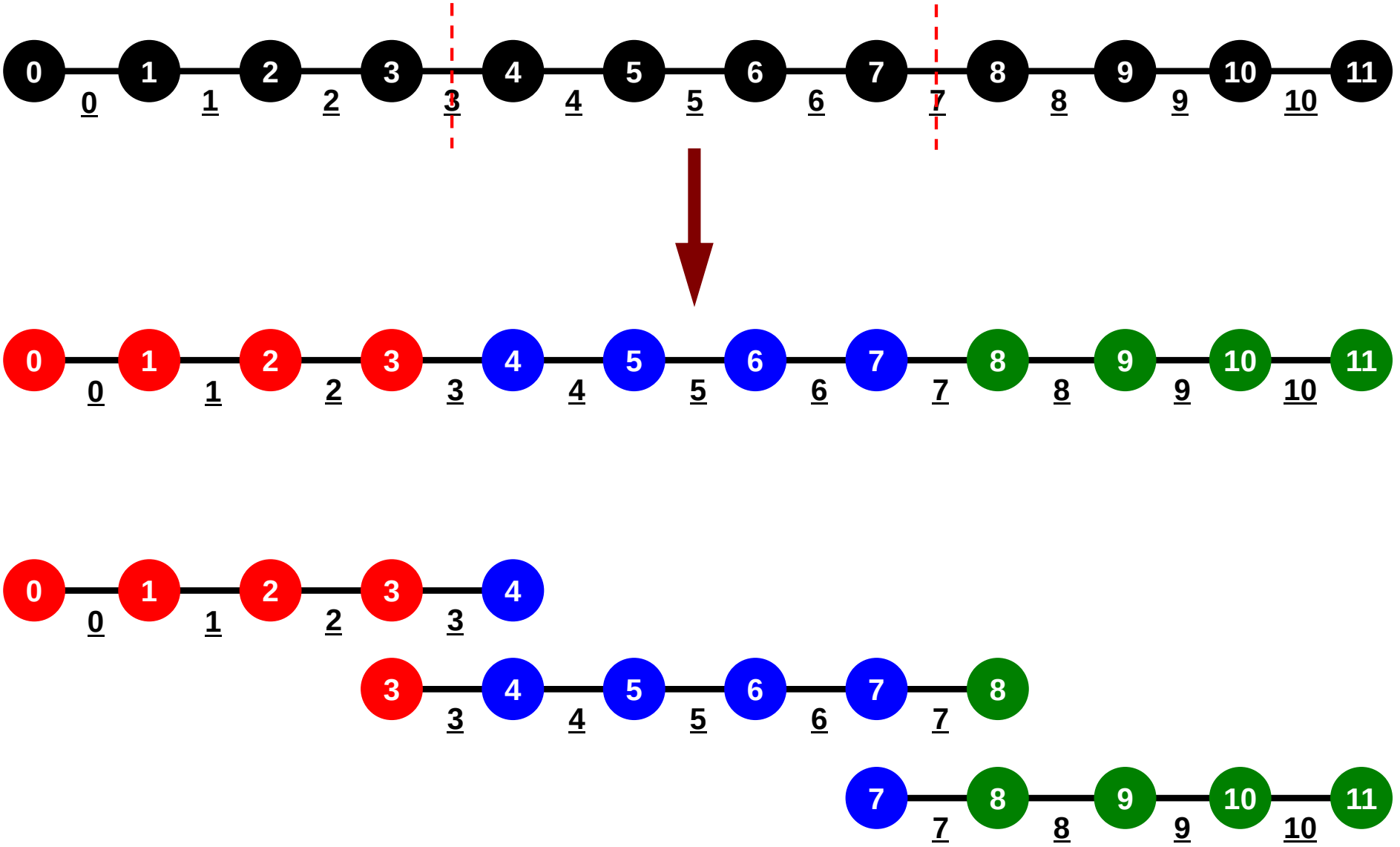
- Partitioned nodes themselves (Internal Nodes)　内点
- Elements which include Internal Nodes　内点を含む要素
- External Nodes included in the Elements　外点
  - in overlapped region among partitions.
- 外点の情報は各プロセスで独立に要素マトリクス生成を実施するのに必要⇒マトリクス生成時の通信は不要となる



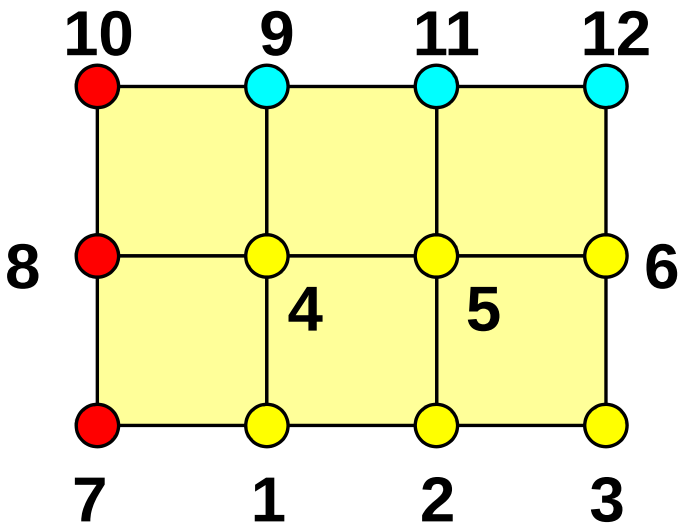




一次元問題：11要素，12節点，3領域



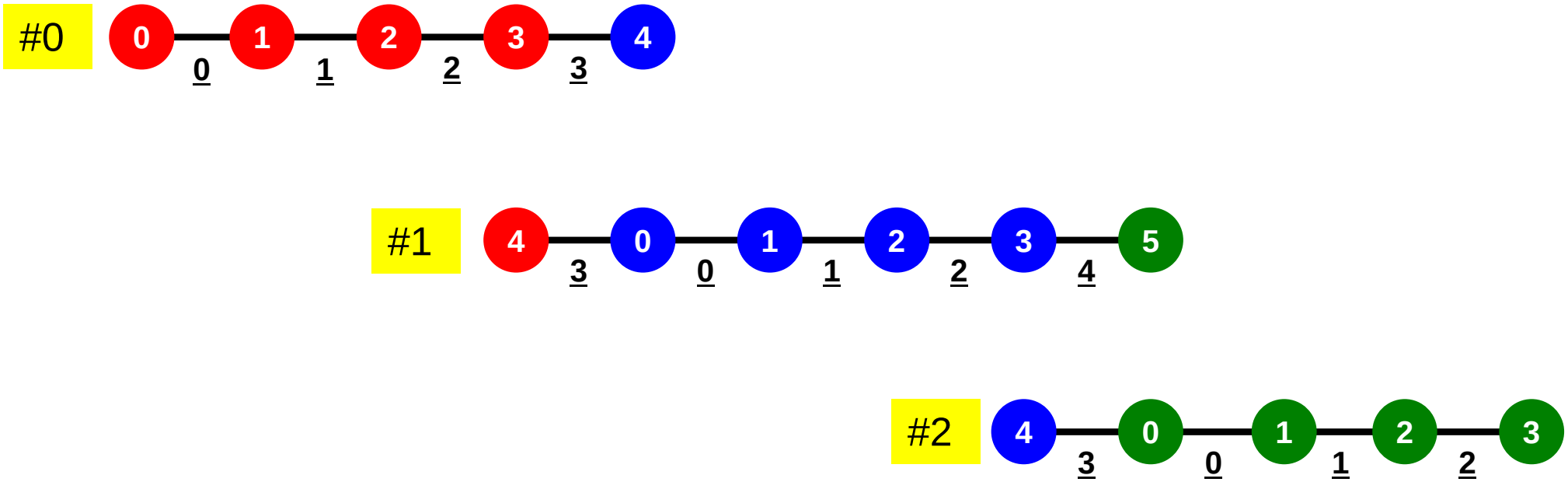
各領域データ（局所データ）仕様



- 内点，外点（internal/external nodes）
  - 内点～外点となるように局所番号をつける
- 隣接領域情報
  - オーバーラップ要素を共有する領域
  - 隣接領域数，番号
- 外点情報
  - どの領域から，何個の，どの外点の情報を「受信：import」するか
- 境界点情報
  - 何個の，どの境界点の情報を，どの領域に「送信：export」するか

SPMD向け局所番号付け

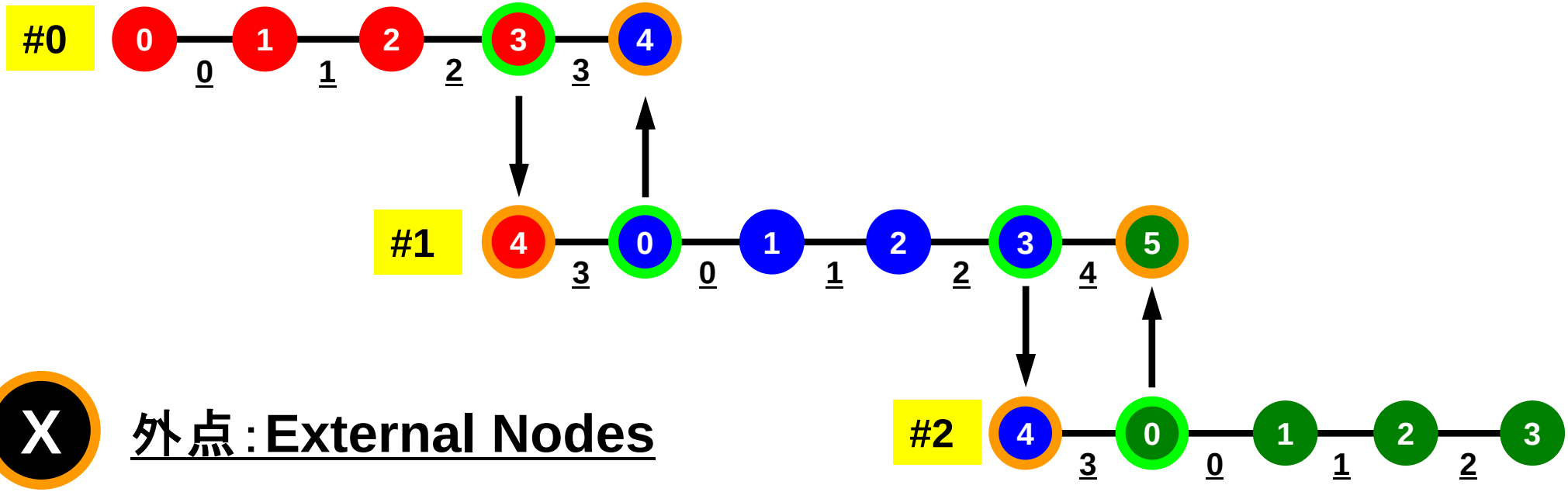
局所節点・要素番号は「0」からスタート



一次元問題：11要素，12節点，3領域

内点・外点・境界点

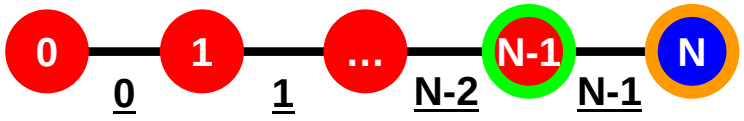
境界点は，他の領域の外点



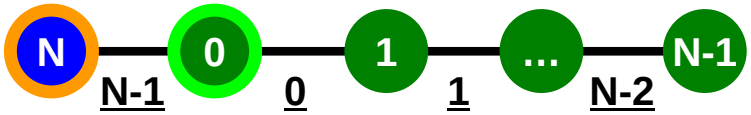
X 外点：External Nodes

Y 境界点：Boundary Nodes

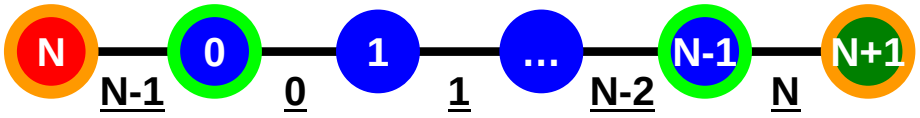
1D FEM: Numbering of Local ID



#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements

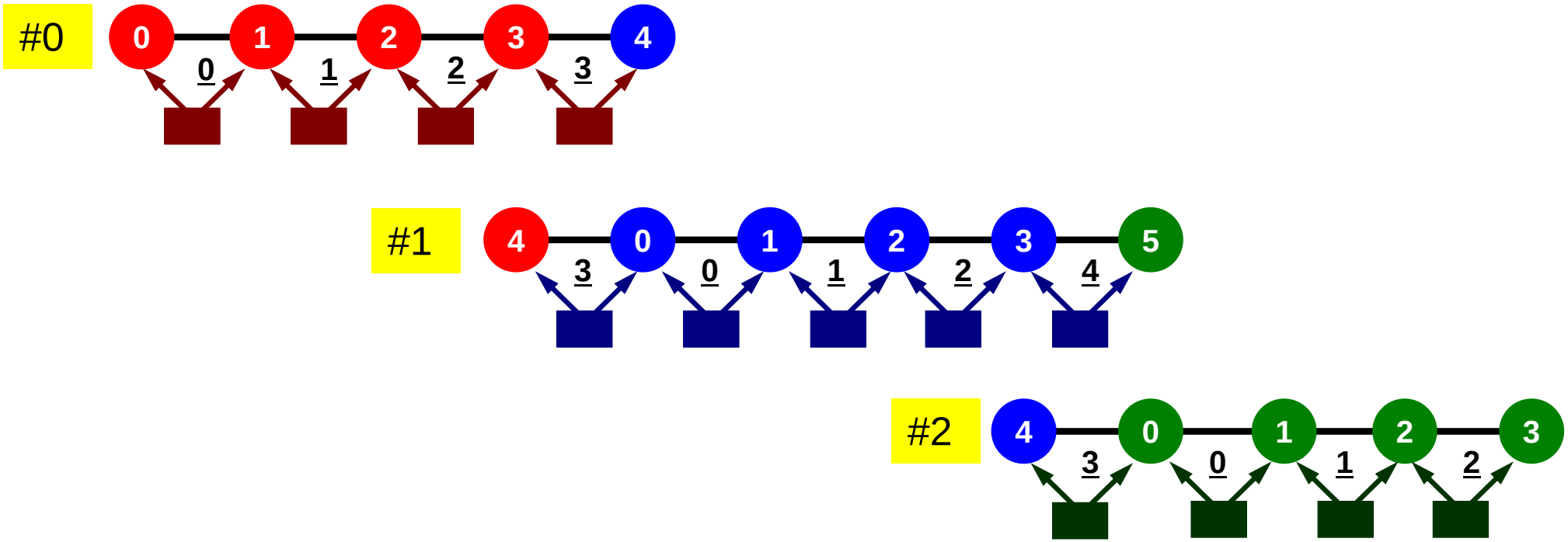


Others (General):
N+2 nodes
N+1 elements

並列有限要素法：マトリクス生成

各要素での積分，要素マトリクス⇒（局所）全体マトリクス

生成の計算は，内点・外点を含む要素の情報があれば可能



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

```
Compute r(0)= b-[A]x(0)
for i= 1, 2, …
    solve [M]z(i-1)= r(i-1)
    ρi-1= r(i-1) z(i-1)
    if i=1
      p(1)= z(0)
     else
      βi-1= ρi-1/ρi-2
      p(i)= z(i-1) + βi-1 p(i-1)
    endif
    q(i)= [A]p(i)
    αi  = ρi-1/p(i)q(i)
    x(i)= x(i-1) + αip(i)
    r(i)= r(i-1) - αiq(i)
    check convergence |r|
end
```

- 前処理
  - 対角スケーリング
- 並列処理が必要なプロセス
  - 内積
  - 行列ベクトル積

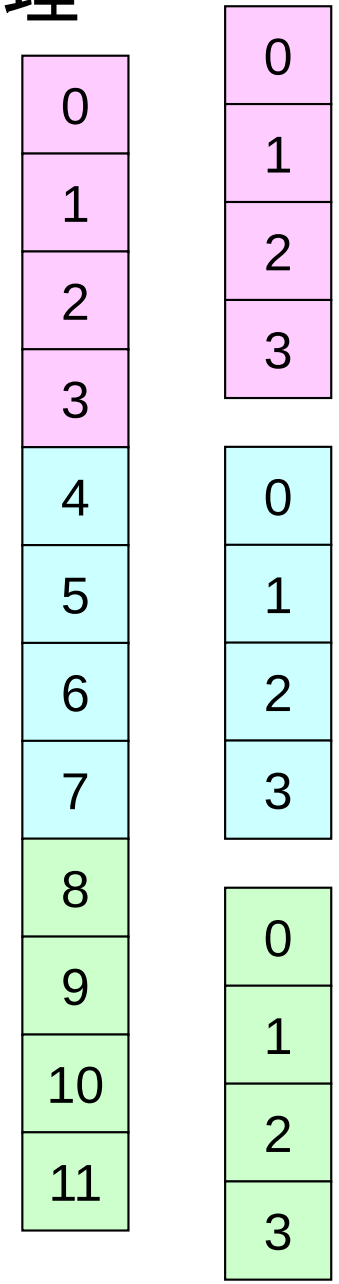
$$[M]=\begin{bmatrix} D_1 & 0 & \cdots & 0 & 0 \\ 0 & D_2 & & 0 & 0 \\ \cdots & & \cdots & & \cdots \\ 0 & 0 & & D_{N-1} & 0 \\ 0 & 0 & \cdots & 0 & D_N \end{bmatrix}$$

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];
    }
```

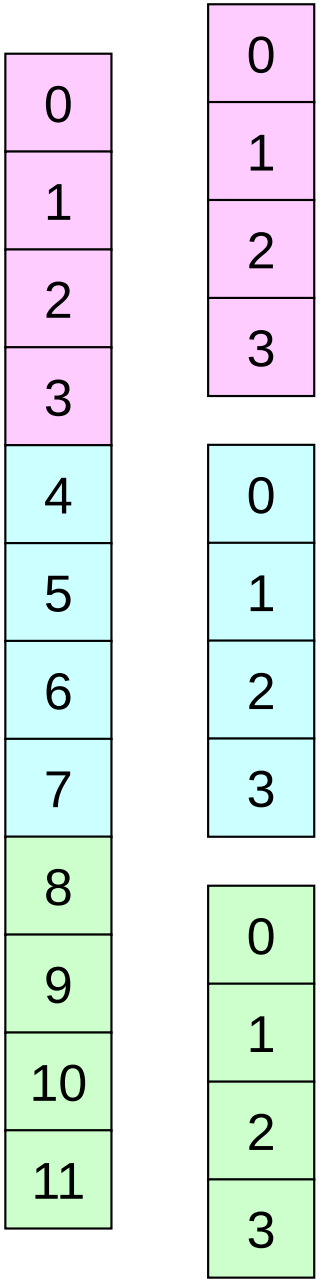
```
/*
//-- {x}= {x} + ALPHA*{p}
//   {r}= {r} - ALPHA*{q}
*/
  for(i=0;i<N;i++){
      U[i]    += Alpha * W[P][i];
      W[R][i] -= Alpha * W[Q][i];
  }
```



内積

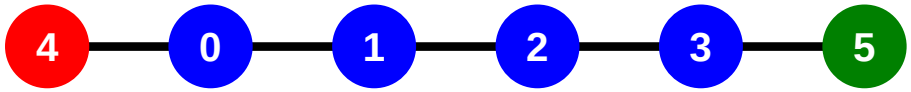
全体で和をとる必要がある⇒集団通信

```
/*
//-- 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

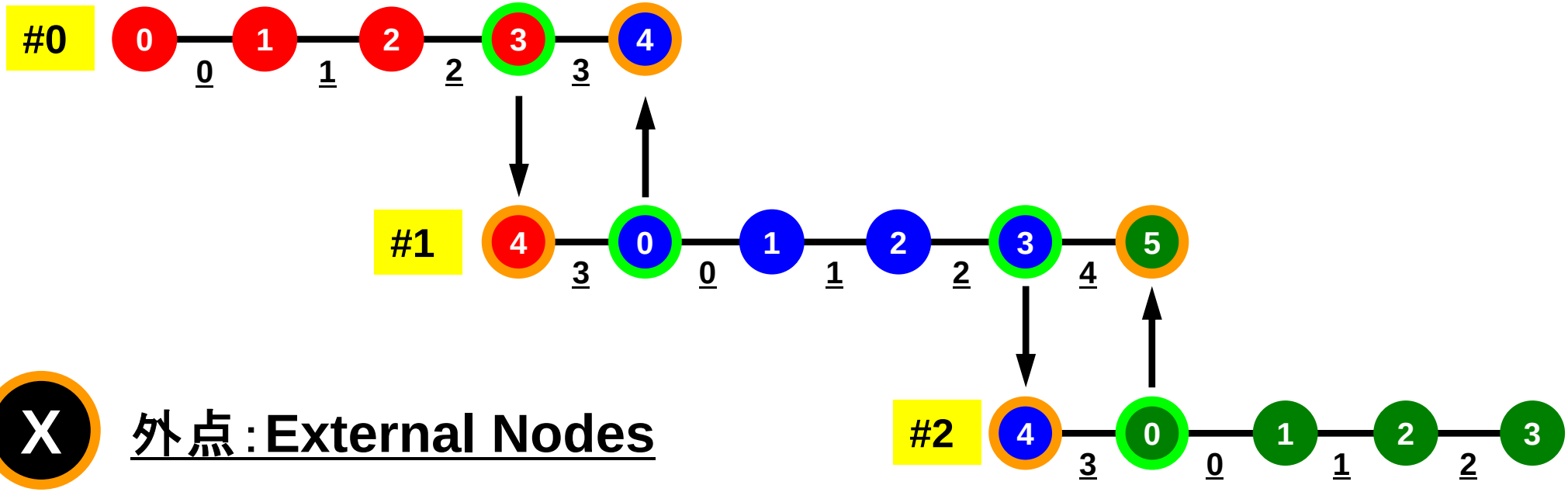
```
/*
//-- {q}= [A]{p}
*/
  for(i=0;i<N;i++){
      W[Q][i] = Diag[i] * W[P][i];
      for(j=Index[i];j<Index[i+1];j++){
          W[Q][i] += AMat[j]*W[P][Item[j]];
          }
      }
```



一次元問題：11要素，12節点，3領域

内点・外点・境界点

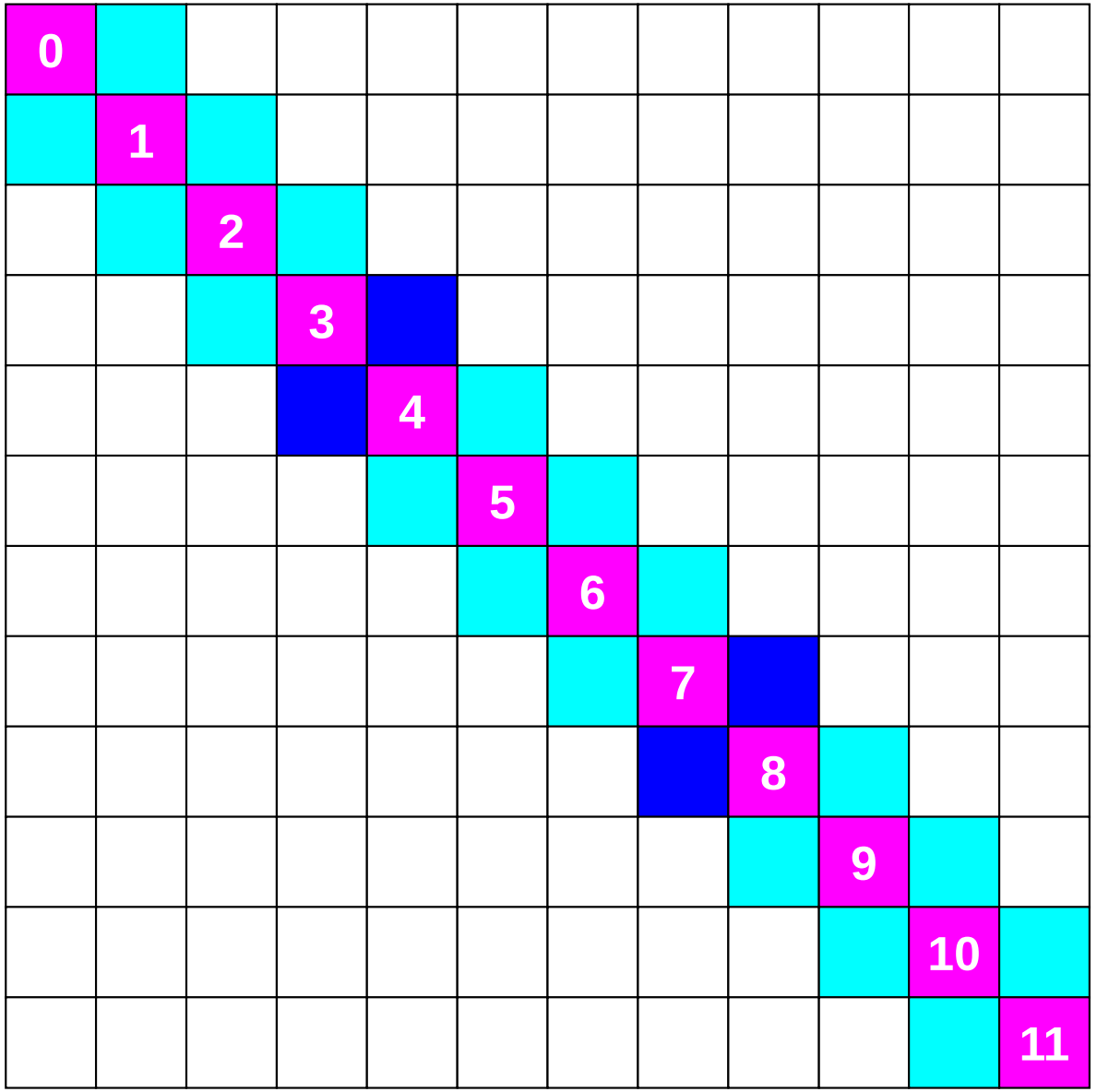
境界点は，他の領域の外点



X 外点：External Nodes

Y 境界点：Boundary Nodes

行列ベクトル積：ローカルに計算実施可能

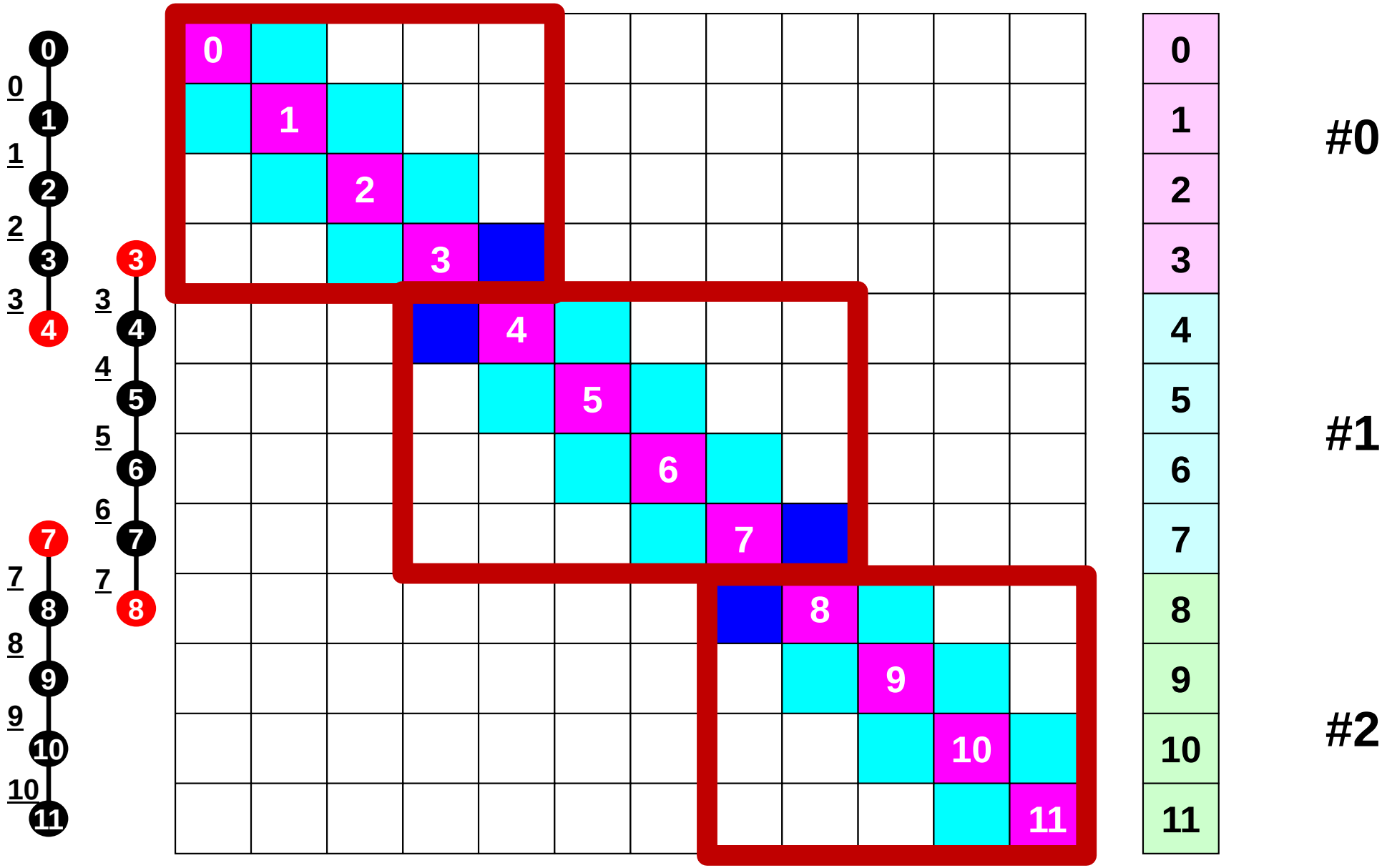


0
1
2
3
4
5
6
7
8
9
10
11

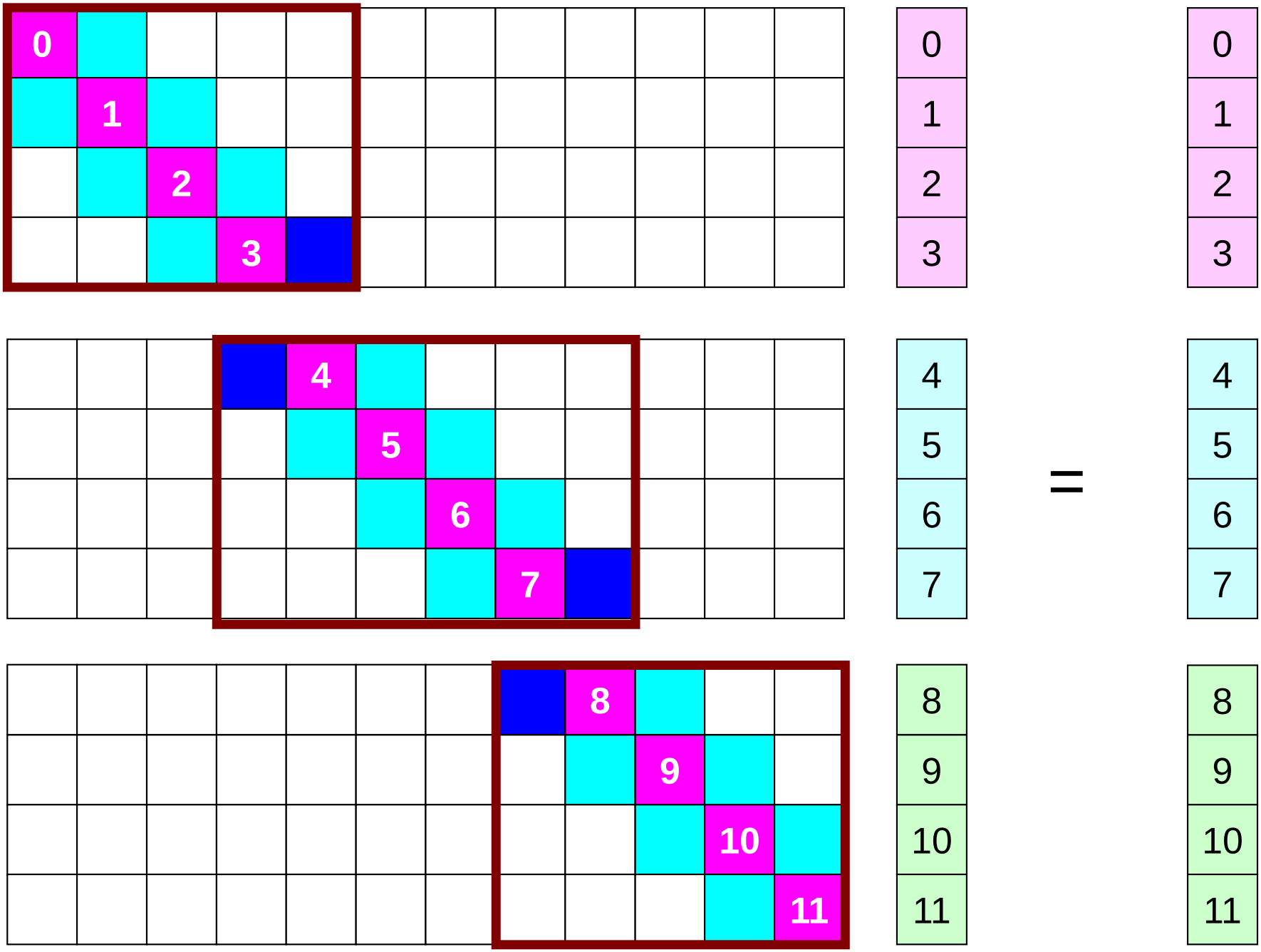
=

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

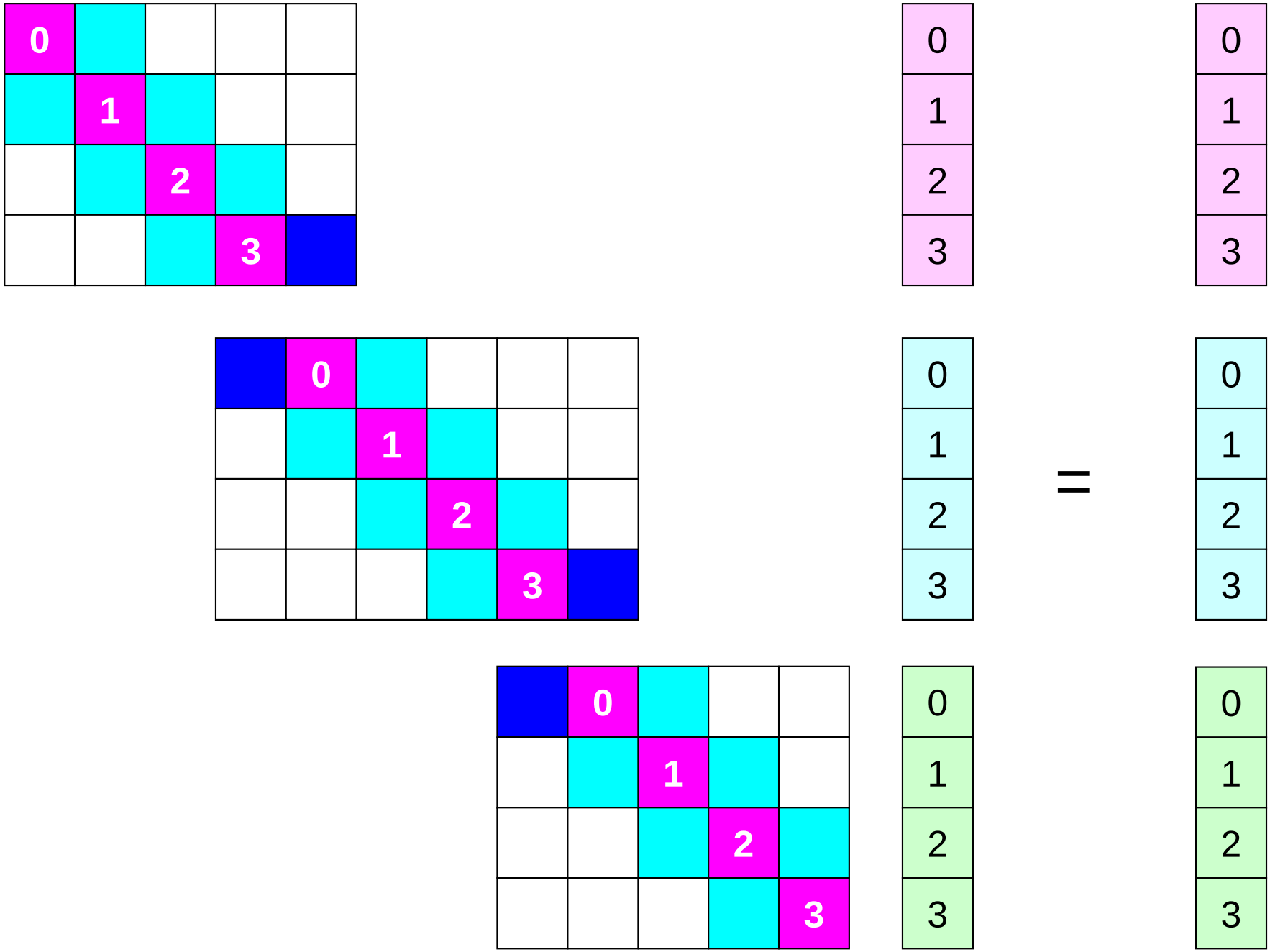
係数行列が疎であるため，局所行列の和集合が全体行列となる



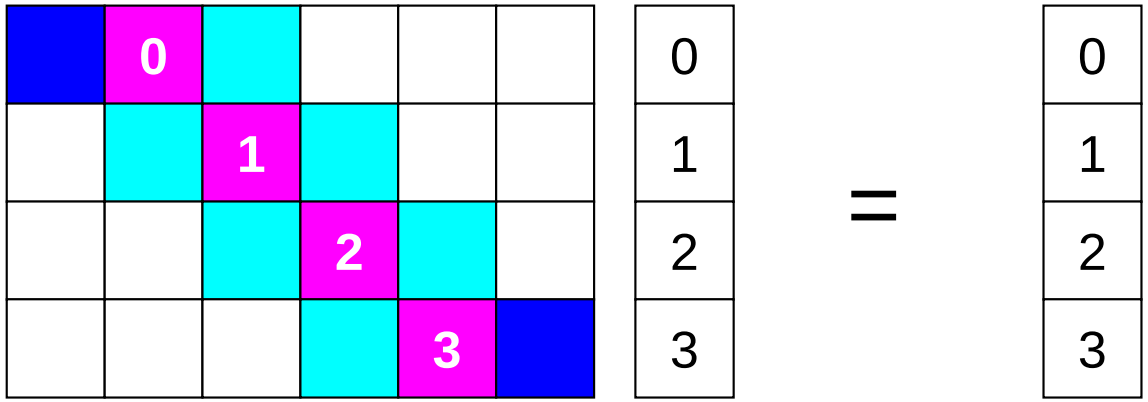
行列ベクトル積：ローカルに計算実施可能

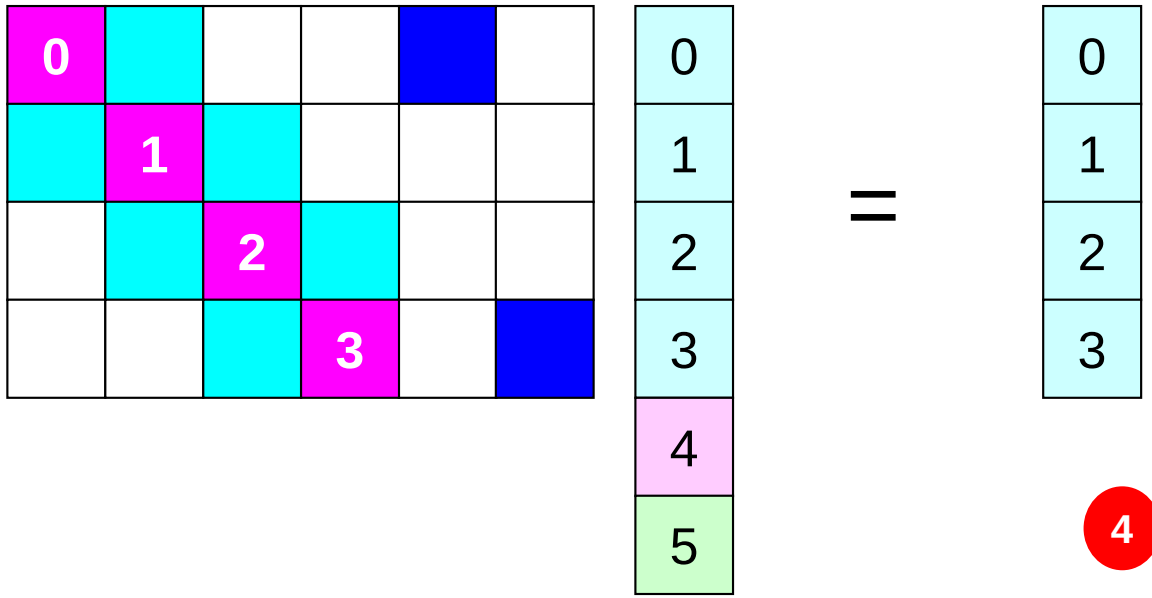


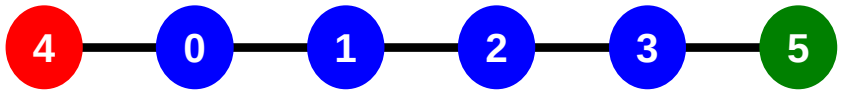
行列ベクトル積：ローカルに計算実施可能



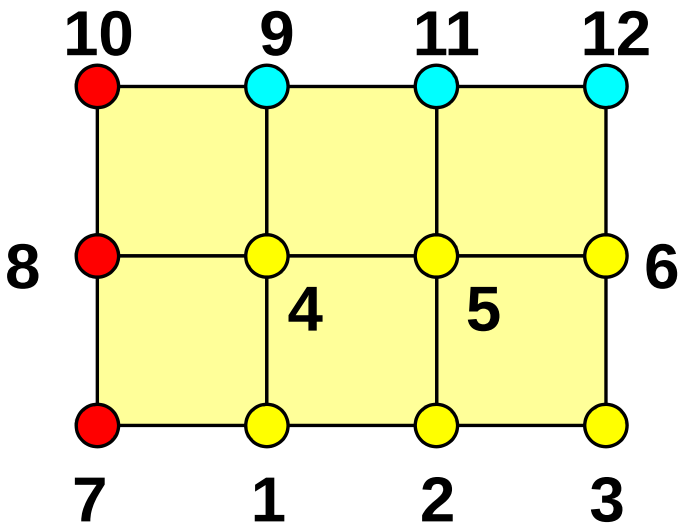
行列ベクトル積：ローカル計算 #1







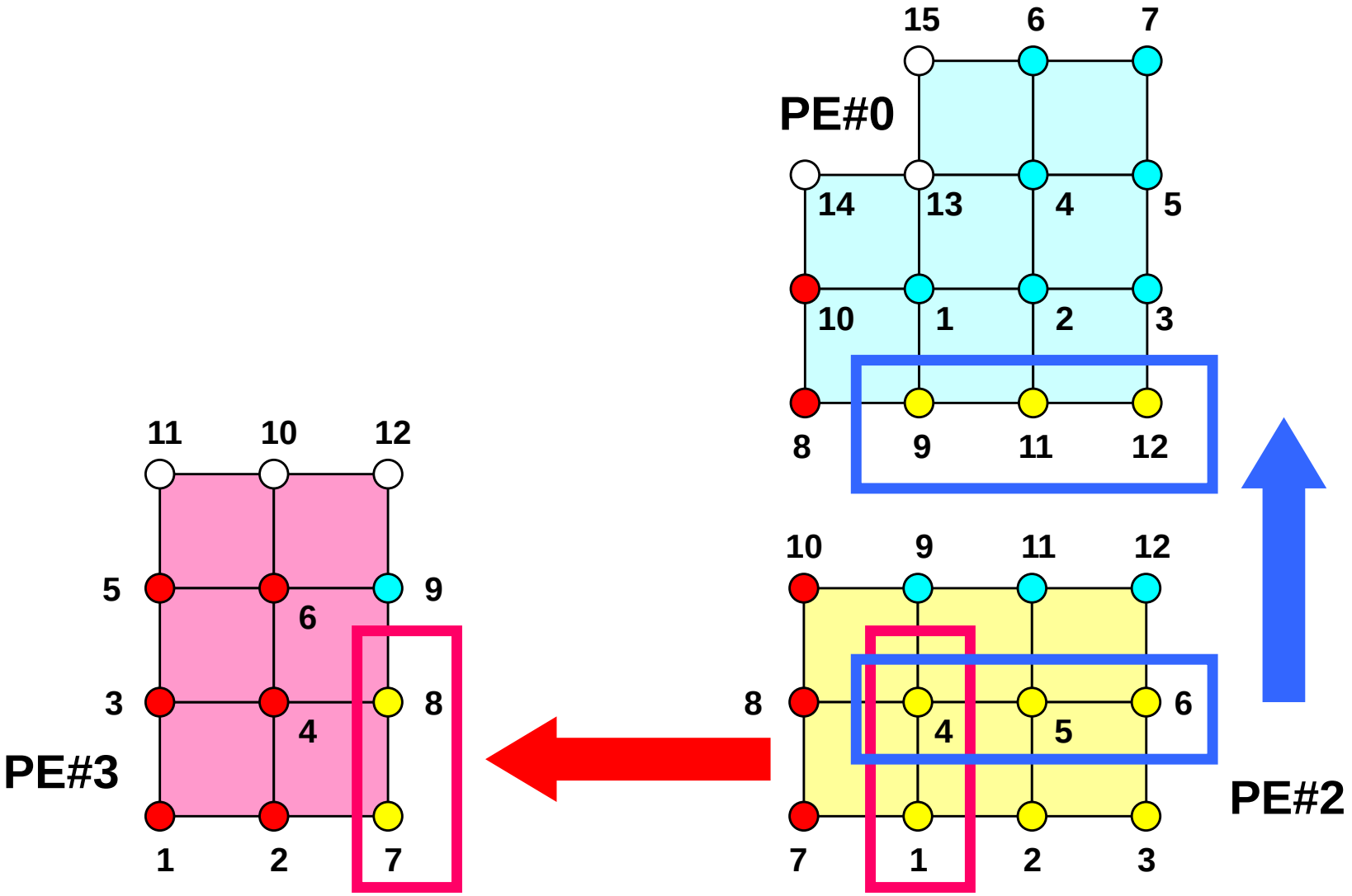
各領域データ（局所データ）仕様



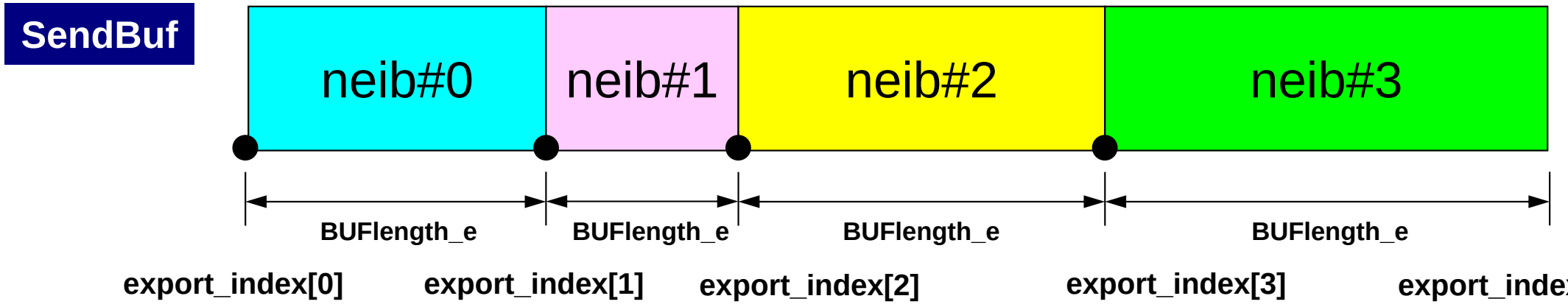
- 内点，外点（internal/external nodes）
  - 内点～外点となるように局所番号をつける
- 隣接領域情報
  - オーバーラップ要素を共有する領域
  - 隣接領域数，番号
- 外点情報
  - どの領域から，何個の，どの外点の情報を「受信：import」するか
- 境界点情報
  - 何個の，どの境界点の情報を，どの領域に「送信：export」するか

Boundary Nodes（境界点）：SEND

PE#2：境界点（Boundary Nodes）の値を送信



SEND: MPI_Isend/Irecv/Waitall



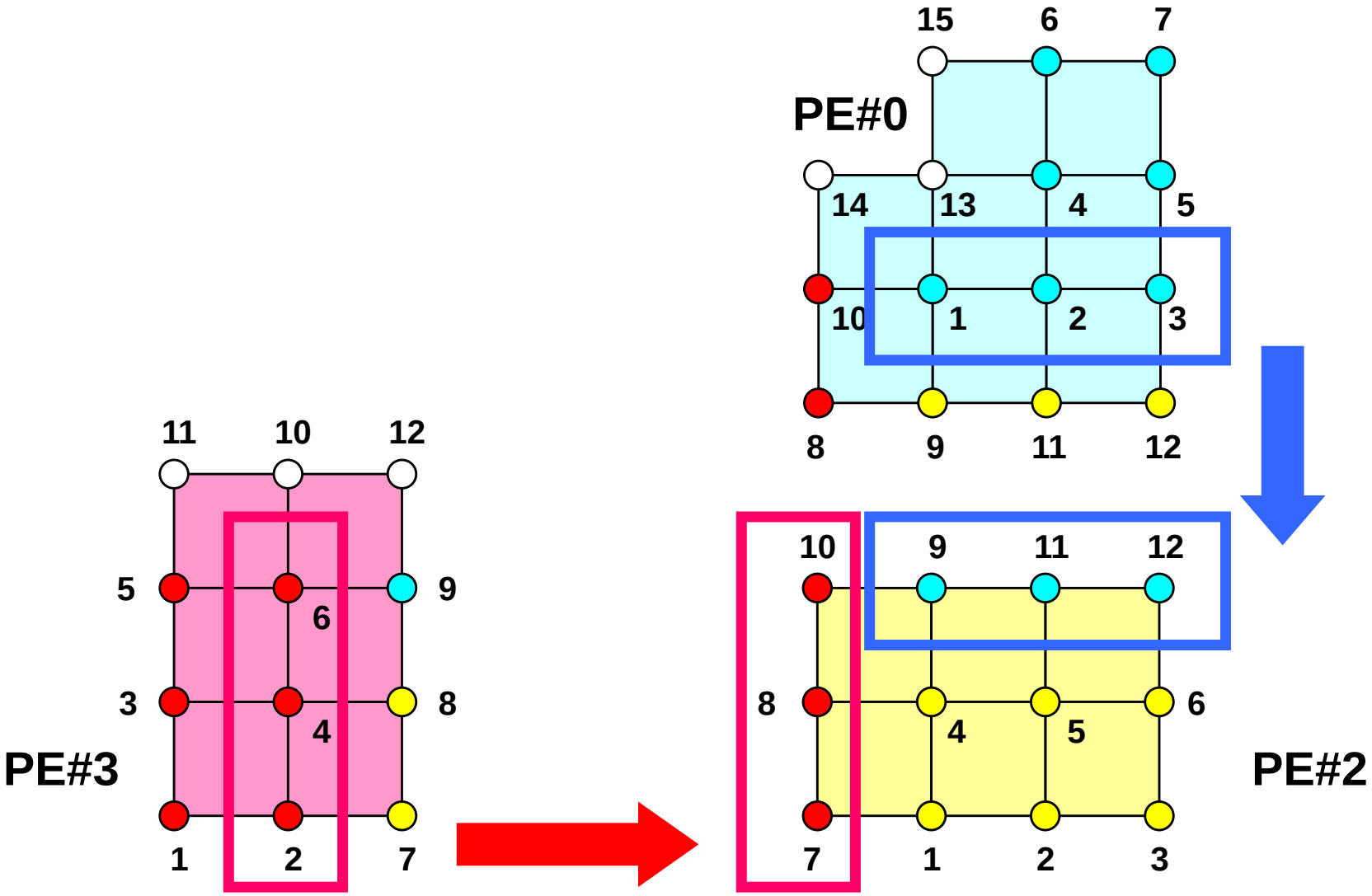
export_item (export_index[neib]:export_index[neib+1]-1) are sent to neib-th neighbor

```
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);
```

Copied to sending buffers

External Nodes（外点）：RECEIVE

PE#2：外点（External Nodes）の値を受信

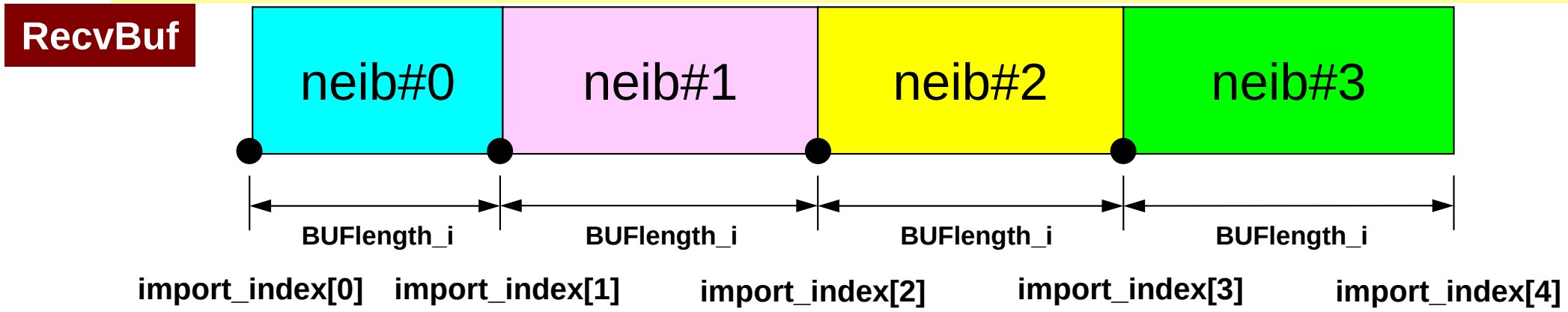


RECV: MPI_Isend/Irecv/Waitall

```
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];
    }
}
```

Copied from receiving buffer

import_item (import_index[neib]:import_index[neib+1]-1) are received from neib-th neighbor



プログラム:1d.c(1/11)
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, QV, COND, QN;
  double X1, X2, DL, Ck;  double *PHI, *Rhs, *X, *Diag, *AMat;
  double *R, *Z, *Q, *P, *DD;
  int *Index, *Item, *IceInod;
  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);                          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, &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);
```

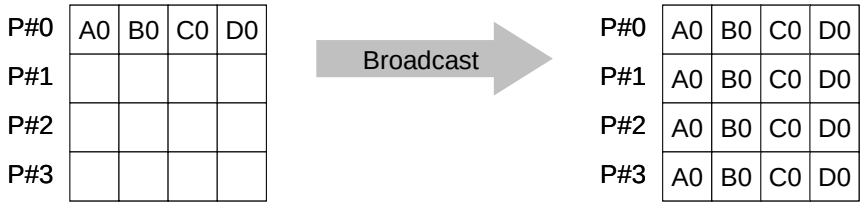
プログラム：1d.c（2/11）
制御データ読み込み

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

/*
//-- CONTROL data
*/

    ierr = MPI_Init(&argc, &argv);                          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, &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);                                  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, &QV, &Area, &COND);
      fscanf(fp, "%d", &IterMax);
      fscanf(fp, "%lf", &Eps);
      fclose(fp);
    }

    ierr = MPI_Bcast(&NEg    , 1, MPI_INT, 0, MPI_COMM_WORLD);  0番プロセスから各プロセスにデータ送信
    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);
```

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.
```

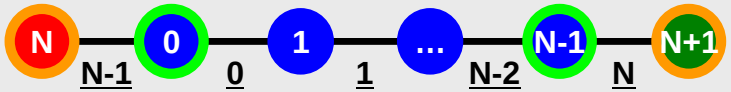
  - **root** 整数 I 送信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータ（通信グループ）を指定する

プログラム:1d.c(3/11)
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

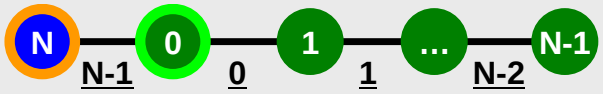
プログラム：1d.c（3/11）
局所分散メッシュデータ，各要素→一様

```
/*
//-- 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

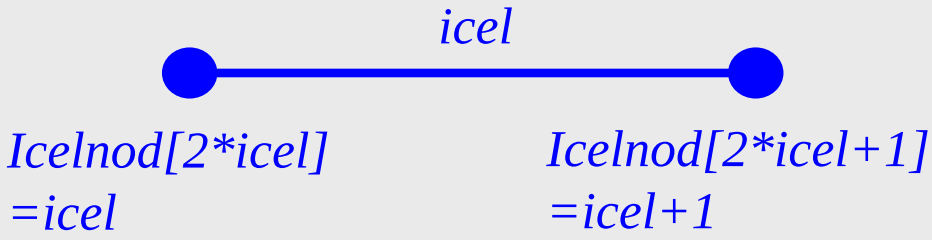
```
/*
//-- LOCAL MESH size
*/
    Ng= NEg + 1;                Ng：総節点数
    N = Ng / PETot;             N ：局所節点数（内点）

    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));
    IceInod= 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）

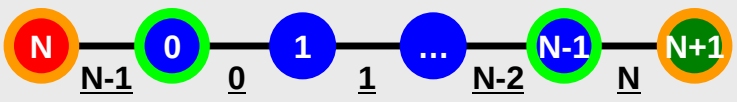
配列初期化，要素～節点

```
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

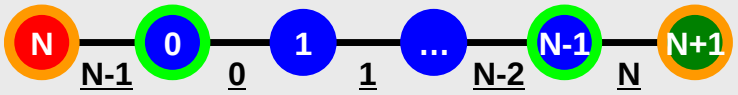
プログラム：1d.c（5/11）

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



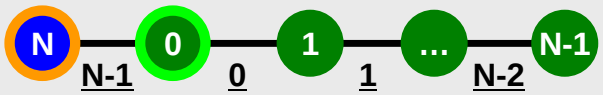
一般の領域：
N+2節点，N+1要素

プログラム:1d.c(6/11)

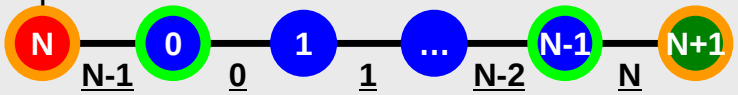
Item定義

```
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;}
  }
}

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

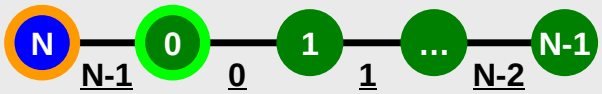


一般の領域:
N+2節点，N+1要素

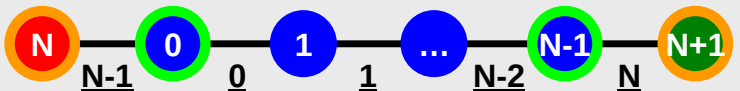
プログラム:1d.c(7/11)
通信テーブル定義

```
/*
//-- 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;

    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））

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））

MPI_Waitall

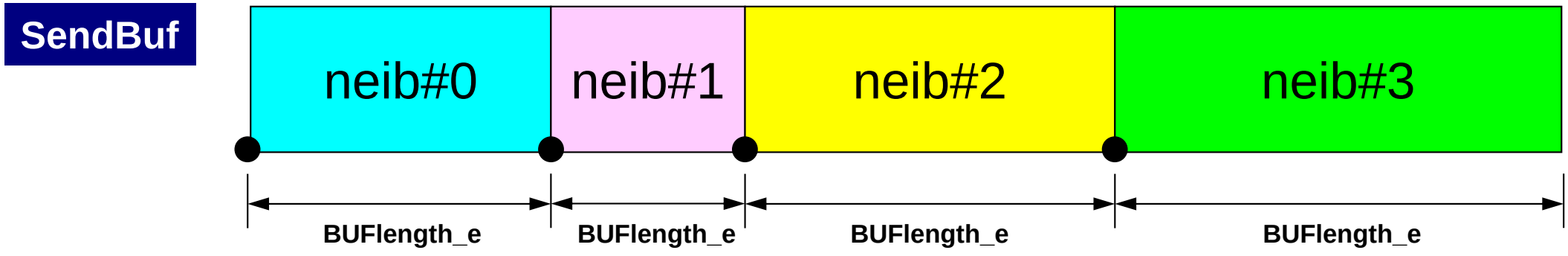
- 1対1非ブロッキング通信関数である「MPI_Isend」と「MPI_Irecv」を使用した場合，プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_Waitall」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_Waitall」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば，「MPI_Isend」と「MPI_Irecv」を同時に同期してもよい。
  - 「MPI_Isend/Irecv」で同じ通信識別子を使用すること
- 「MPI_Barrier」と同じような機能であるが，代用はできない。
  - 実装にもよるが，「request」，「status」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

一般化された通信テーブル:送信

- 送信相手
  - NeibPETot, NeibPE[NeibPETot]
- それぞれの送信相手に送るメッセージサイズ
  - export_index[NeibPETot+1]
- 「境界点」番号
  - export_item[export_index[NeibPETot+1]]
- それぞれの送信相手に送るメッセージ
  - SendBuf[export_index[NeibPETot+1]]

送信（MPI_Isend/Irecv/Waitall）

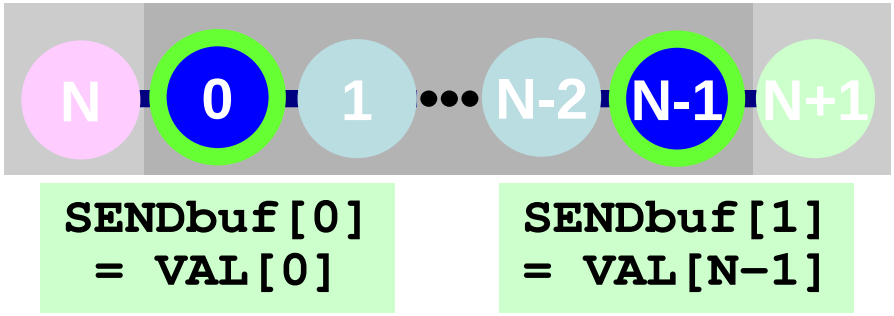


export_index[neib]～export_index[neib+1]-1番目のexport_itemがneib番目の隣接領域に送信される

```
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[NeibPETot]
    - 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[NeibPETot+1]
    - export_index[0]=0, export_index[1]= 1, export_index[2]= 2
- 「境界点」番号
  - export_item[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[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

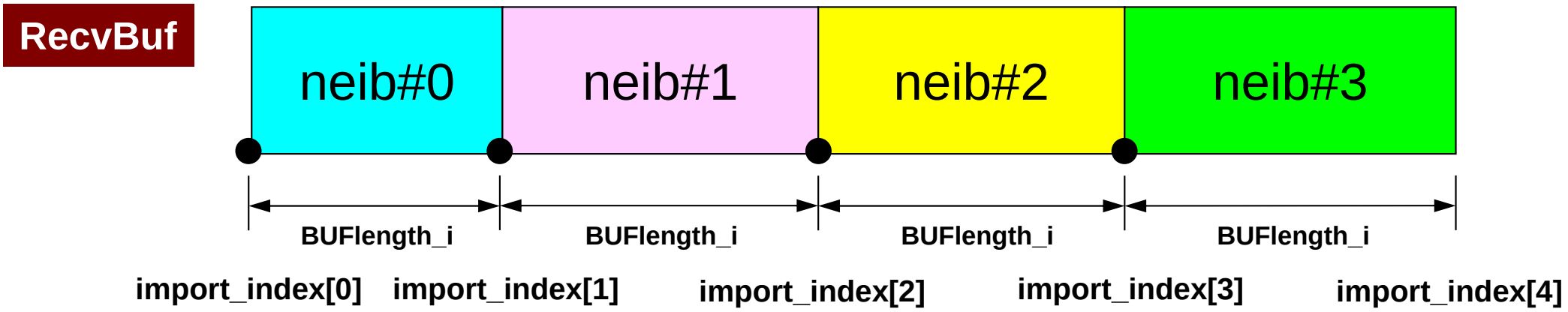
- 受信相手
  - NeibPETot, NeibPE[NeibPETot]
- それぞれの受信相手から受け取るメッセージサイズ
  - export_index[NeibPETot+1]
- 「外点」番号
  - export_item[export_index[NeibPETot+1]]
- それぞれの受信相手から受け取るメッセージ
  - SendBuf[export_index[NeibPETot+1]]

受信（MPI_Isend/Irecv/Waitall）

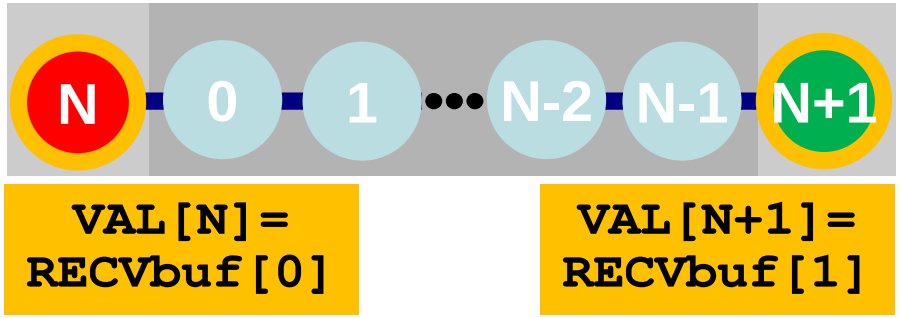
```
for (neib=0; neib<NeibPETot; neib++){
   tag= 0;
   iS_i= import_index[neib];
   iE_i= import_index[neib+1];
   BUFlength_i= iE_i - iS_i

   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

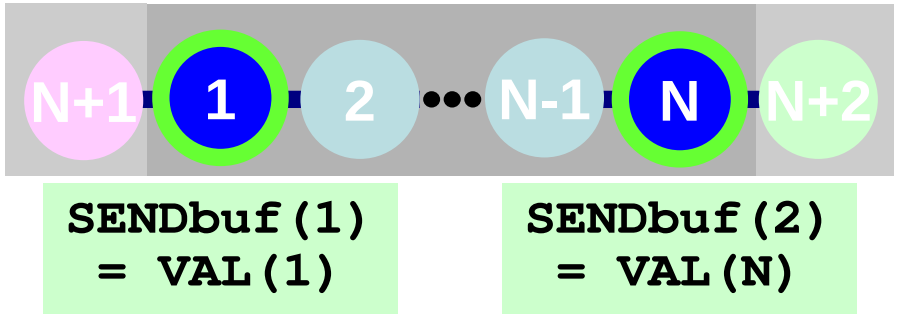


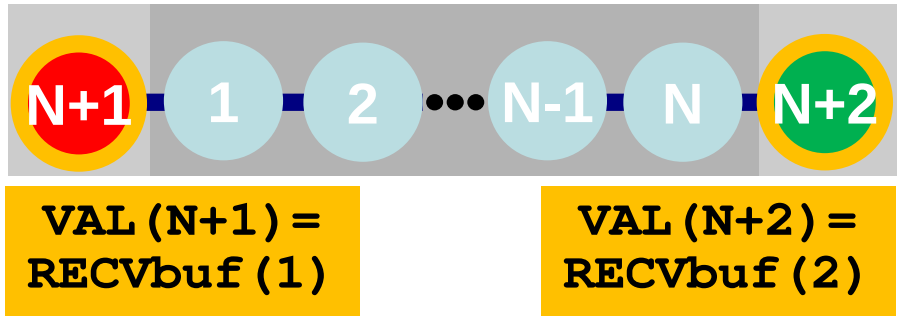
受信:一次元問題



- 受信相手
  - NeibPETot, NeibPE[NeibPETot]
    - 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 [NeibPETot+1]
    - import_index[0]=0, import_index[1]= 1, import_index[2]= 2
- 「外点」番号
  - import_item [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
  - import_item [import_index[NeibPETot+1]]
    - VAL[N]=RecvBuf[0], VAL[N+1]=RecvBuf[1]

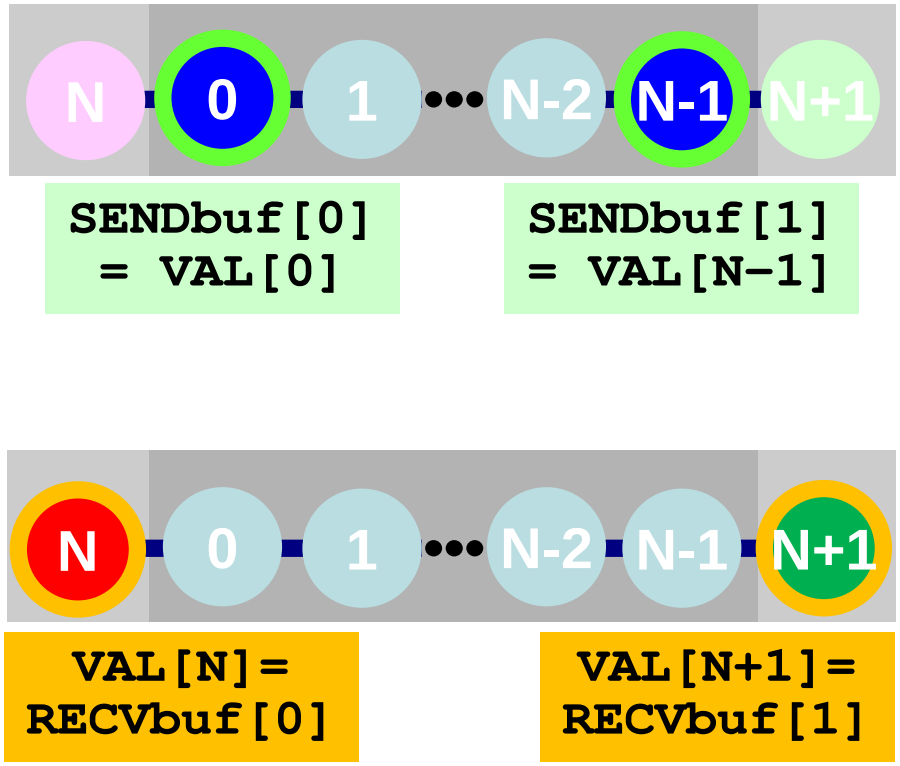
一般化された通信テーブル：Fortran






```
      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

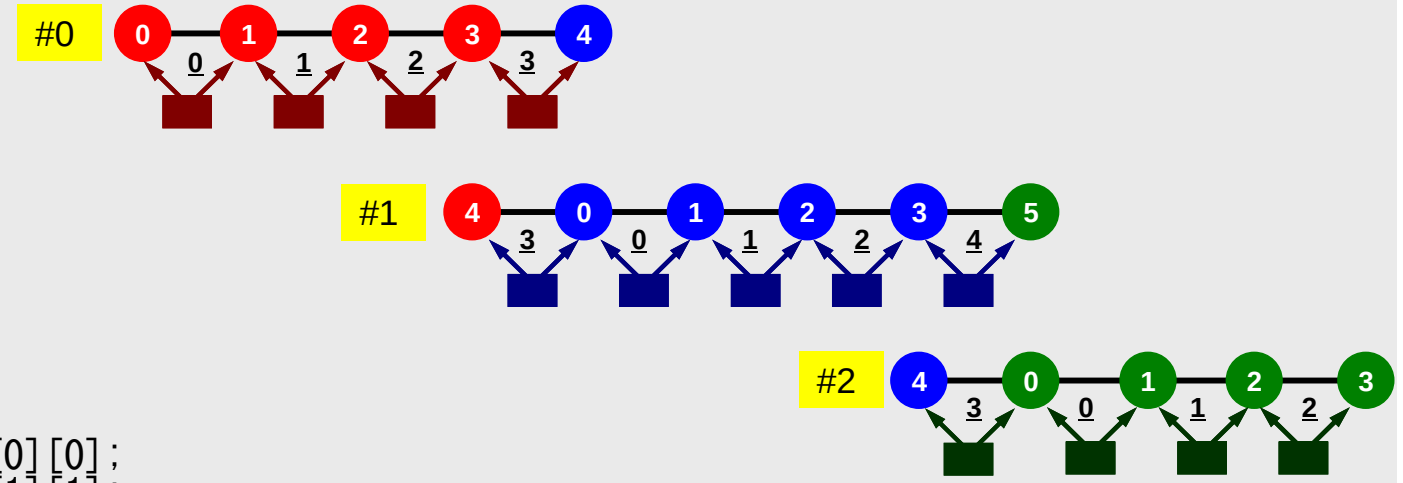



```
      NEIBPETOT= 2
NEIBPE[0]= my_rank - 1
NEIBPE[1]= my_rank + 1

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

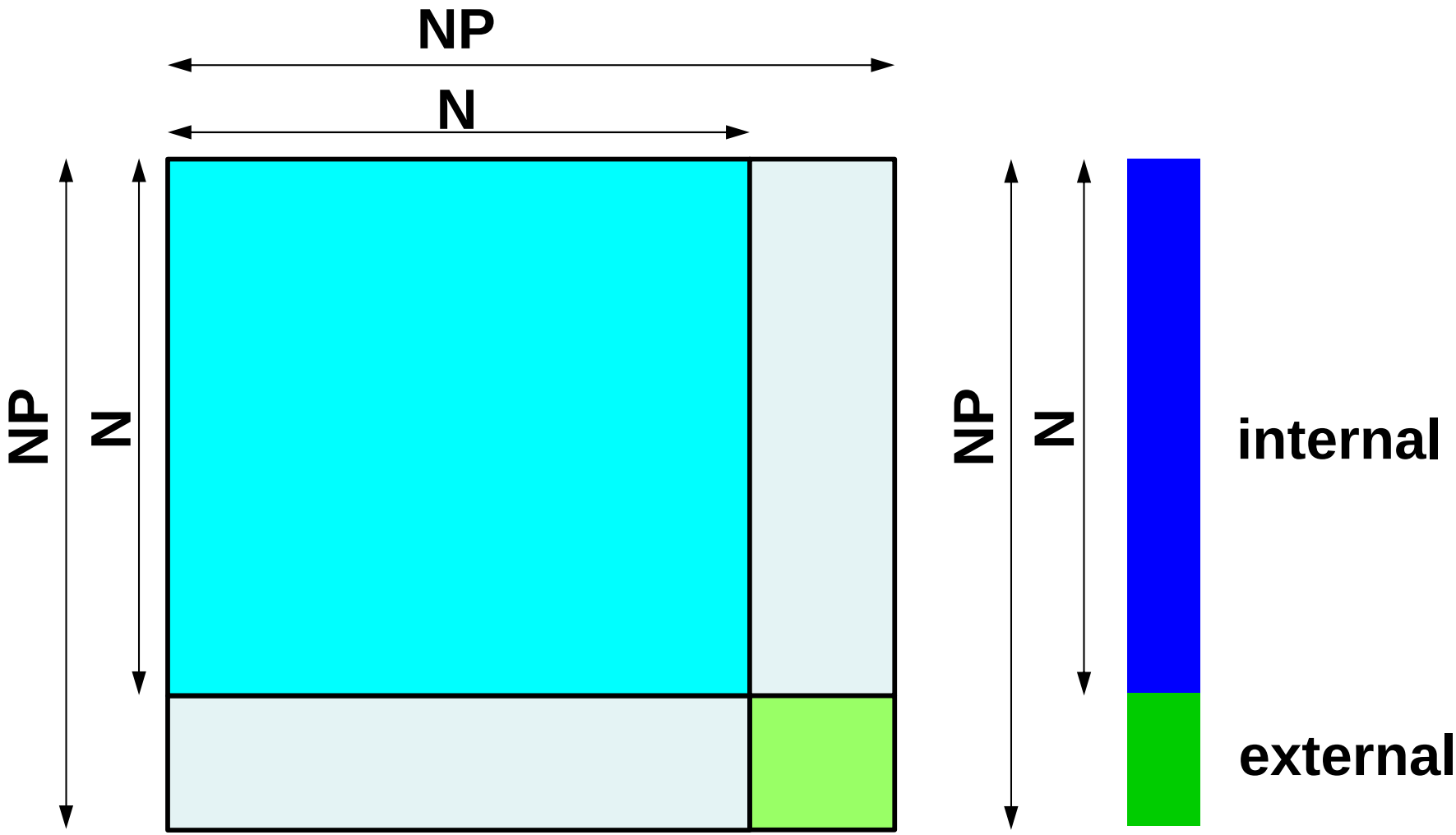
プログラム：1d.c（8/11）

全体マトリクス生成：1CPUのときと全く同じ：各要素→一様

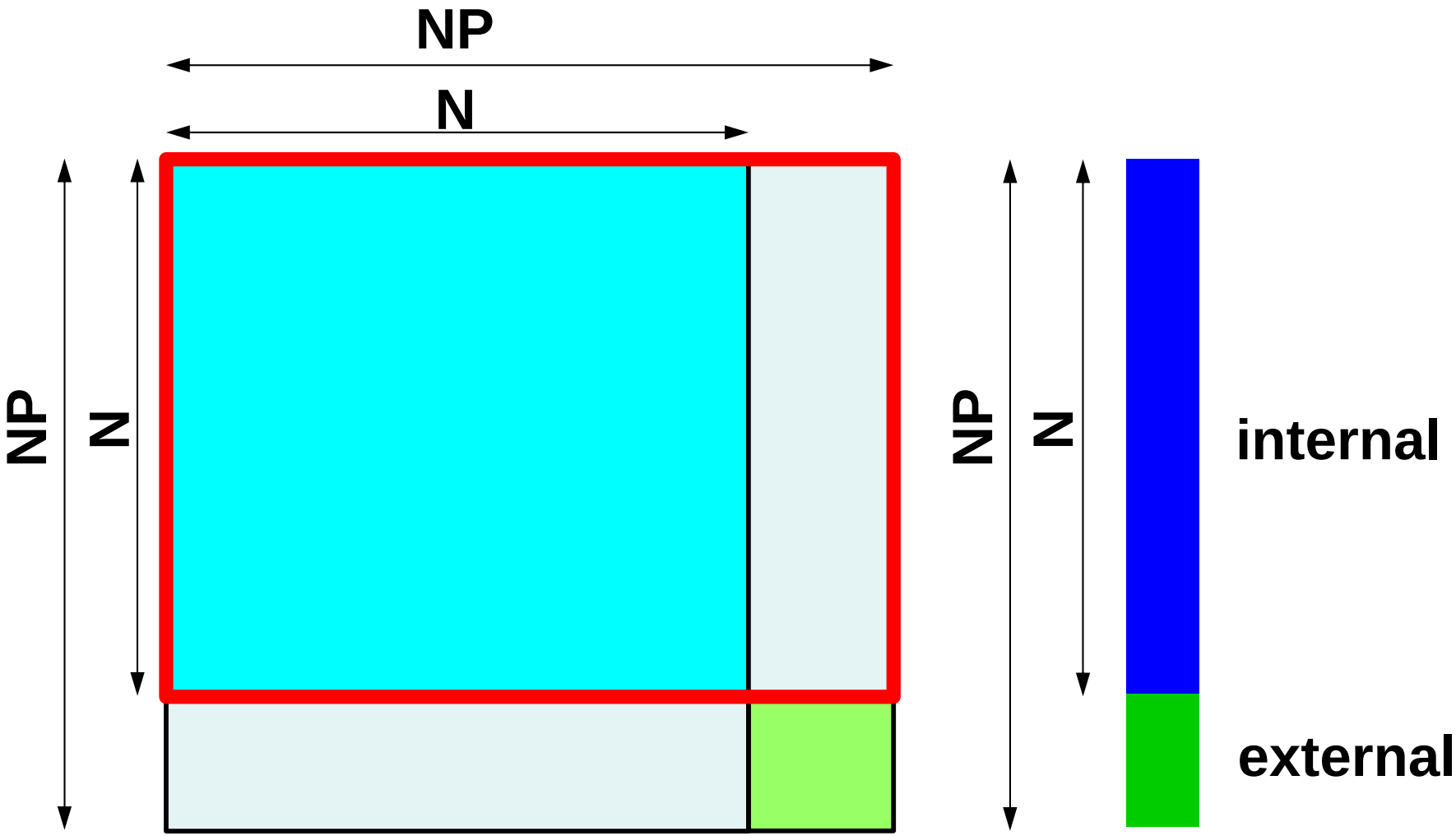
```
/*
// +-----------------+
// | MATRIX assemble |
// +-----------------+
*/
    for(icel=0;icel<NE;icel++){
      in1= Icelnod[2*icel];
      in2= Icelnod[2*icel+1];
      DL = dX;

      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



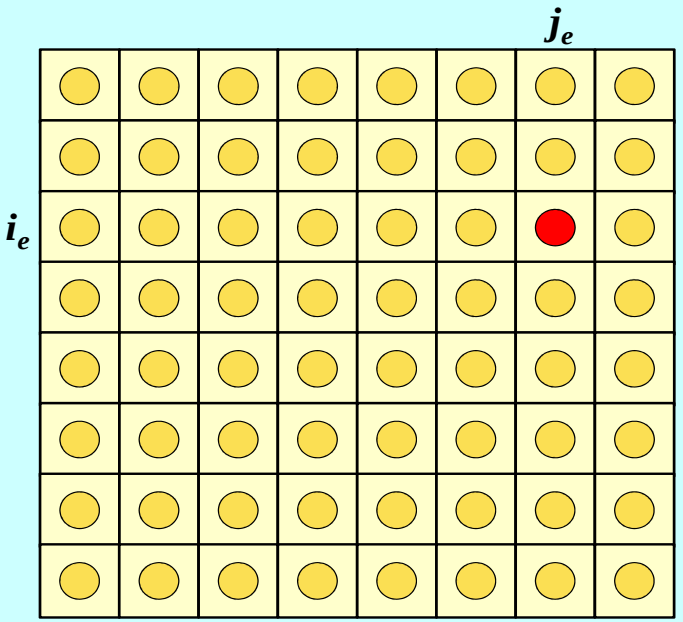
本当に必要なのはこの部分



MAT_ASS_MAIN: Overview

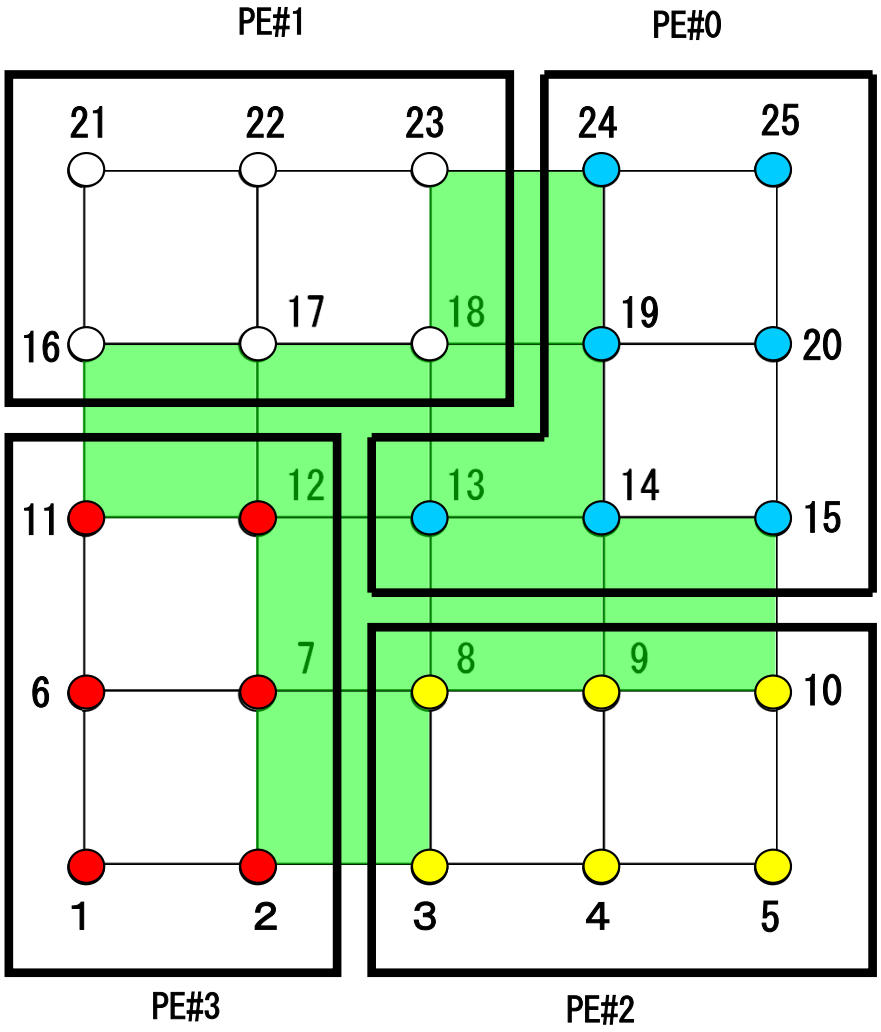
```
do kpn= 1, 2          Gaussian Quad. points in ζ-direction
  do jpn= 1, 2        Gaussian Quad. points in η-direction
    do ipn= 1, 2      Gaussian Quad. Pointe in ξ-direction
      Define Shape Function at Gaussian Quad. Points (8-points)
      Its derivative on natural/local coordinate is also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

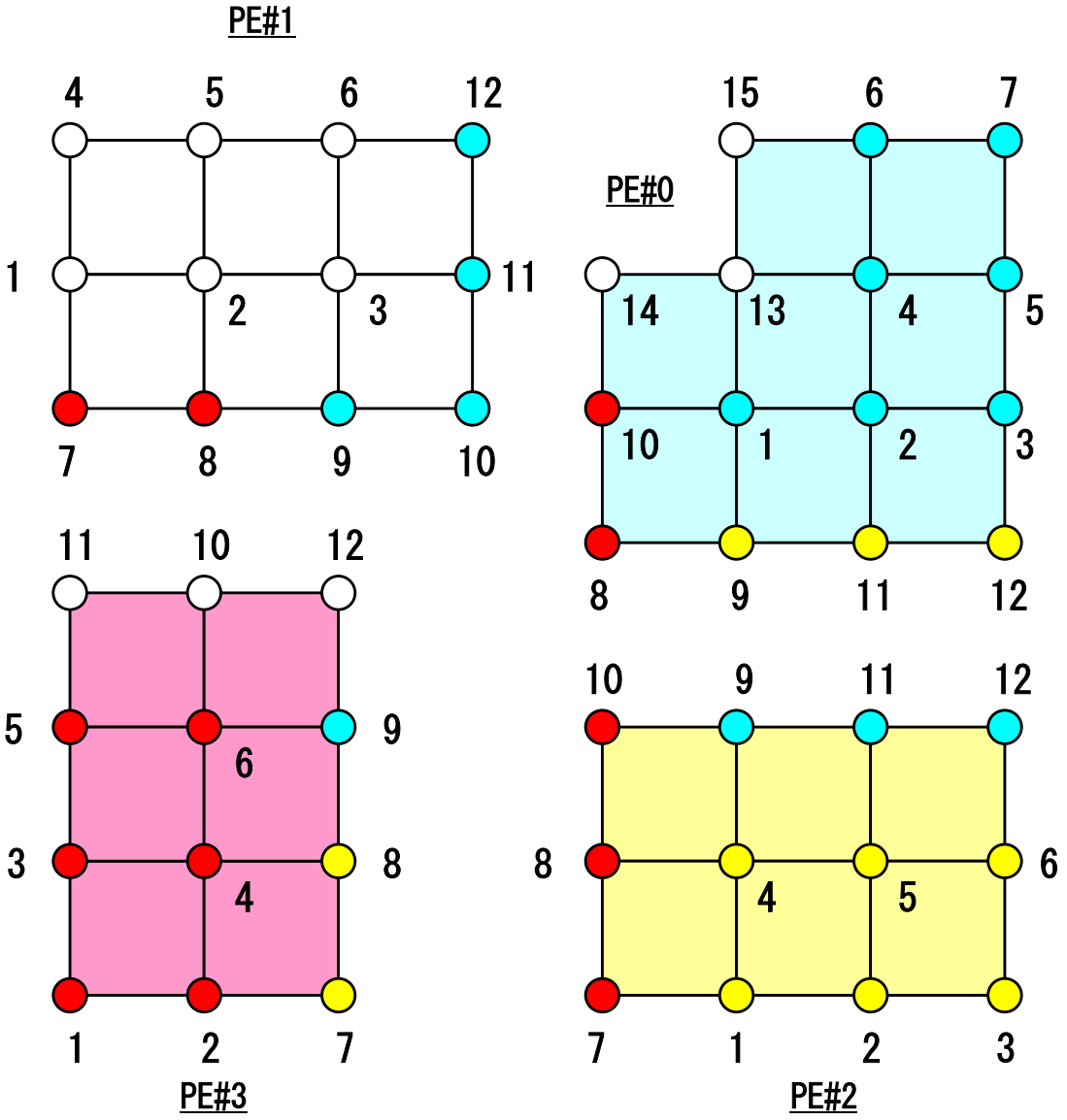
```
do icel= 1, ICELTOT    Loop for Element
  Jacobian and derivative on global coordinate of shape functions at
  Gaussian Quad. Points are defined according to coordinates of 8 nodes. (JACOBI)

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

      do kpn= 1, 2       Gaussian Quad. points in ζ-direction
        do jpn= 1, 2     Gaussian Quad. points in η-direction
          do ipn= 1, 2   Gaussian Quad. points in ξ-direction
            integration on each element
            coefficients of element matrices
            accumulation to global matrix
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```



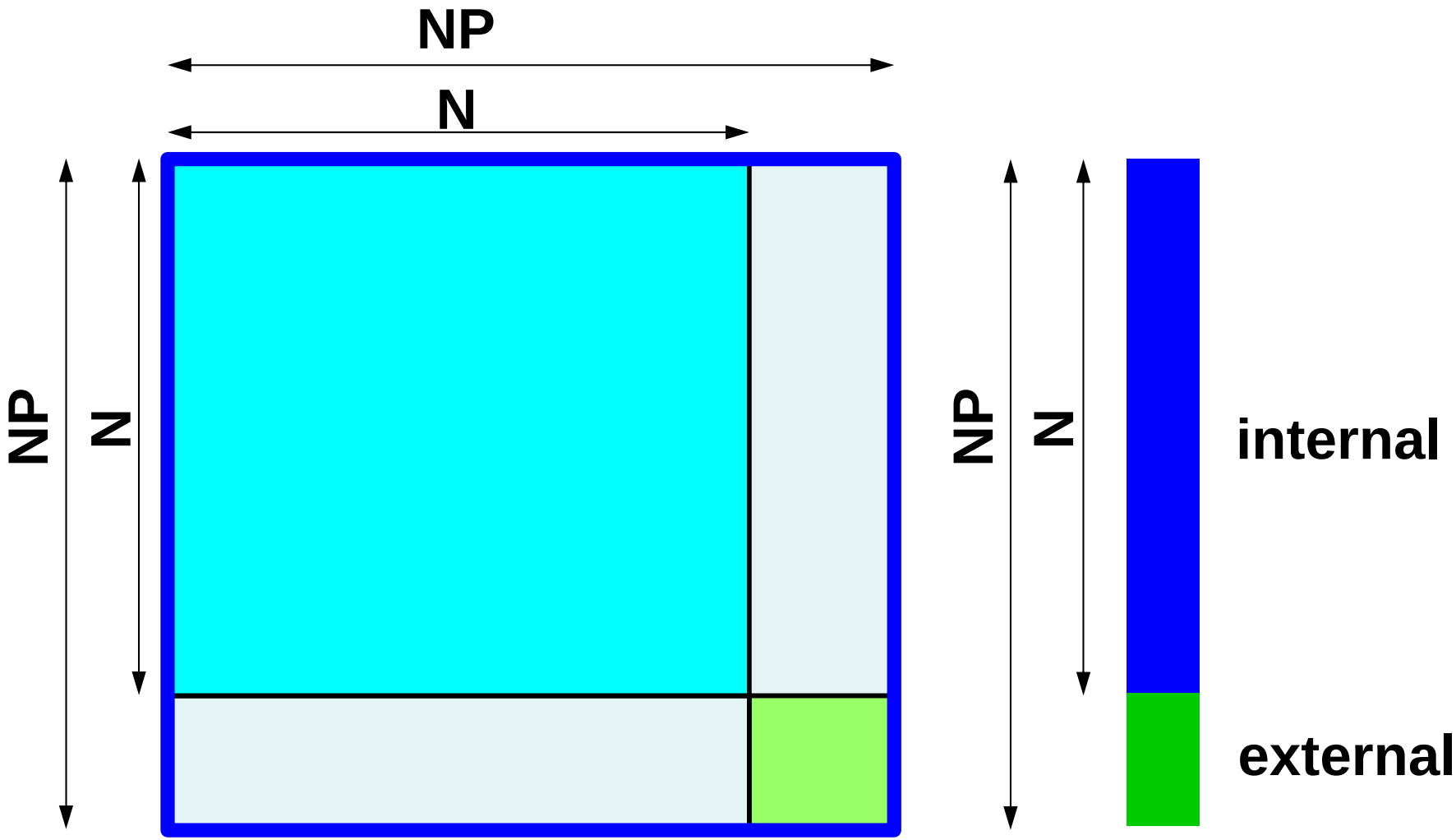
全ての要素の計算を実施する

外点を含むオーバーラップ領域の要素の計算も実施

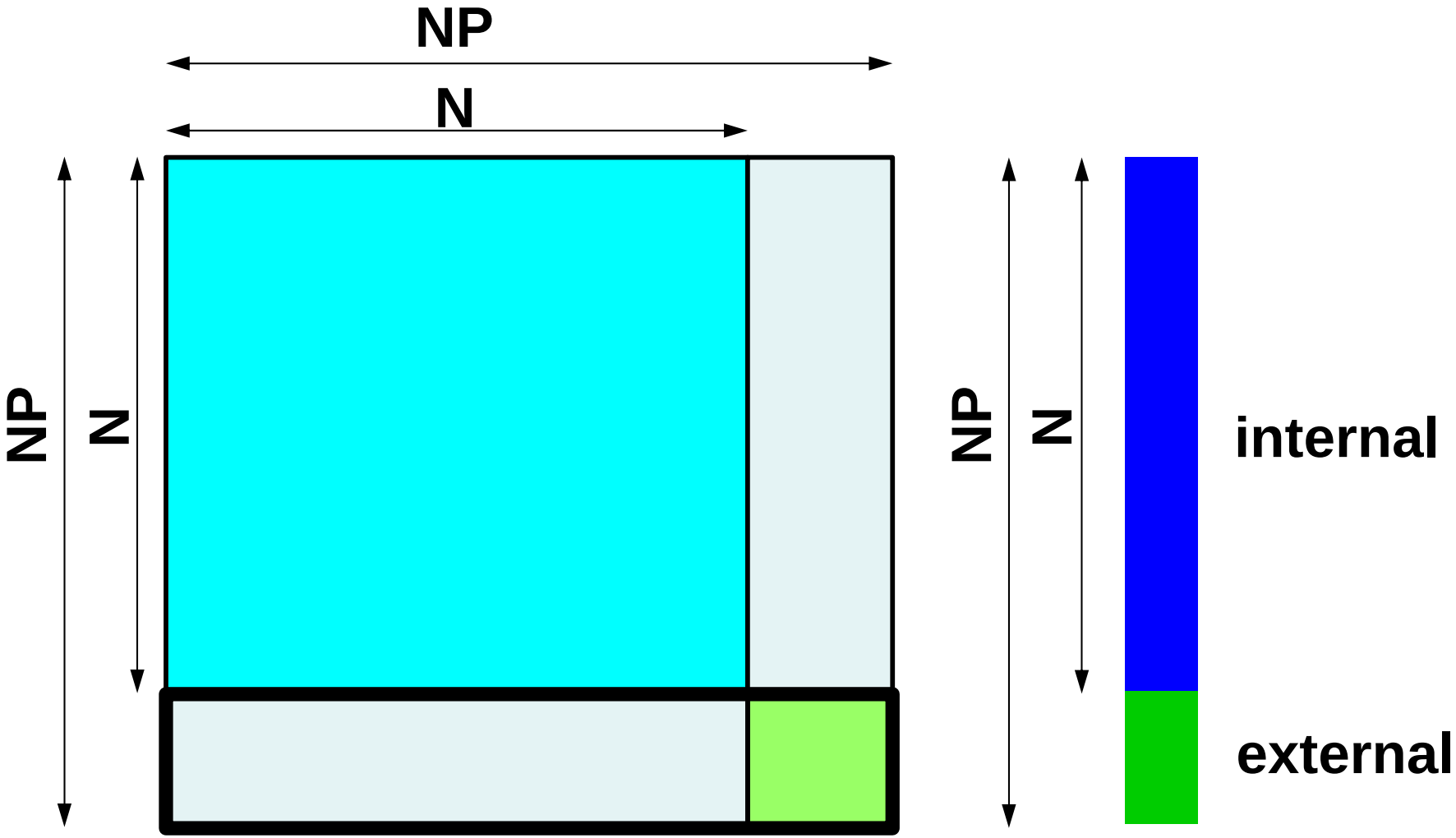




従って結果的にはこのような行列を得るが



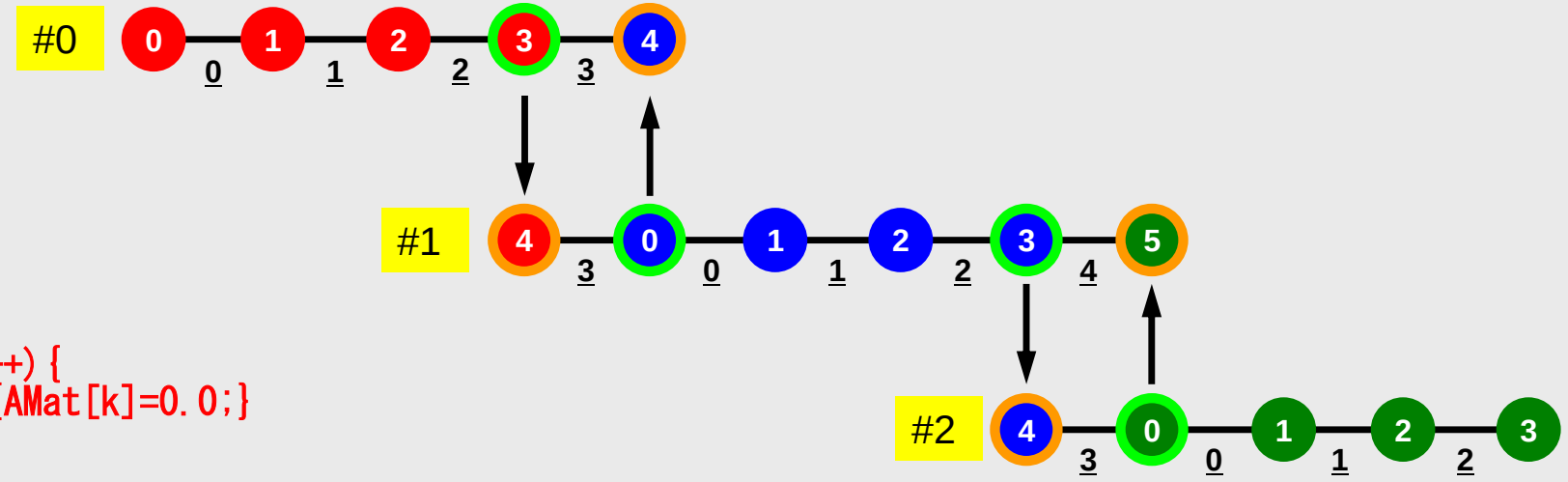
黒枠で囲んだ部分の行列は不完全
しかし，計算には使用しないのでこれで良い



プログラム：1d.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, …
  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}

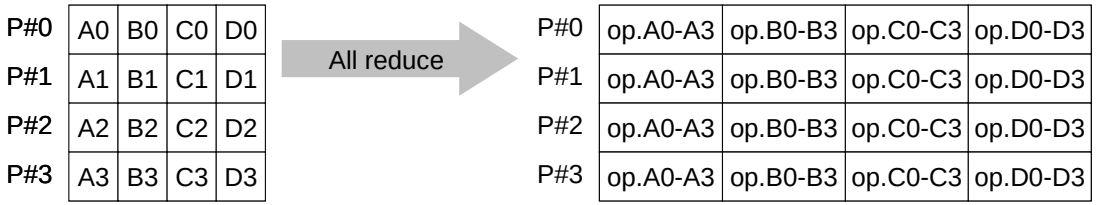
```
MPI_Waitall(NeibPETot, RequestSend, StatSend);

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_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

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, …

- 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, …
  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, …**

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

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

  if i=1

    $\mathbf{p^{(1)}= z^{(0)}}$

  else

    $\boldsymbol{\beta_{i-1}= \rho_{i-1}/\rho_{i-2}}$

    $\mathbf{p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}}$

  endif

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

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

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

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

  check convergence $|r|$

end

CG法(4/5)

```
    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, …**

- 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, …**

- 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
- プログラムの説明
- 計算例

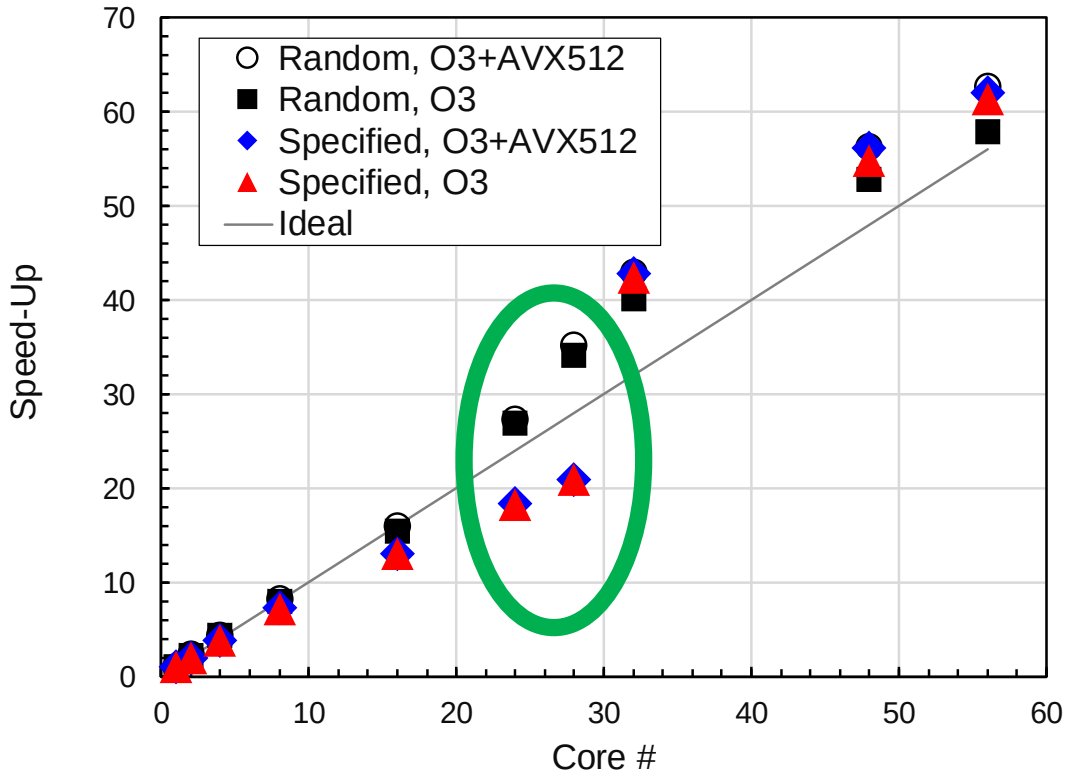
計算結果（1次元）：CG法部分，10^6

1,000回反復に要する時間，Strong Scaling

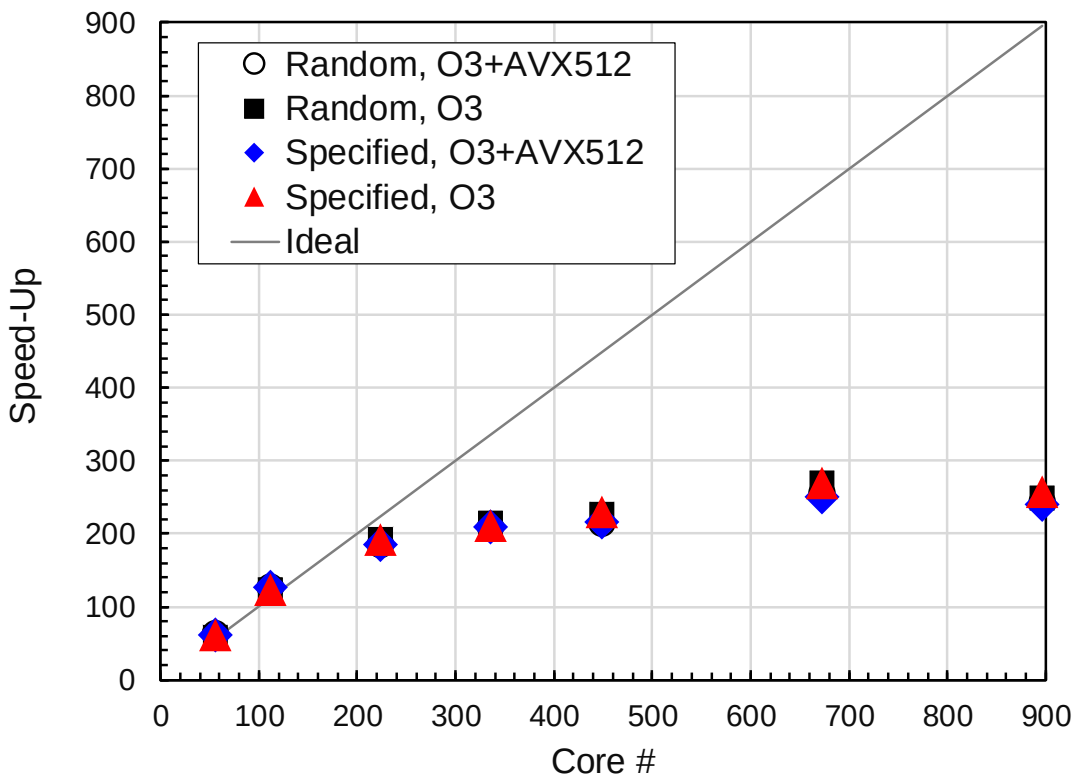
1コア計算時間を基準，2ノード以上では56コア/ノード使用

5回計測，最速のものを採択

up to 1 node, 56cores



up to 16 nodes, 896 cores

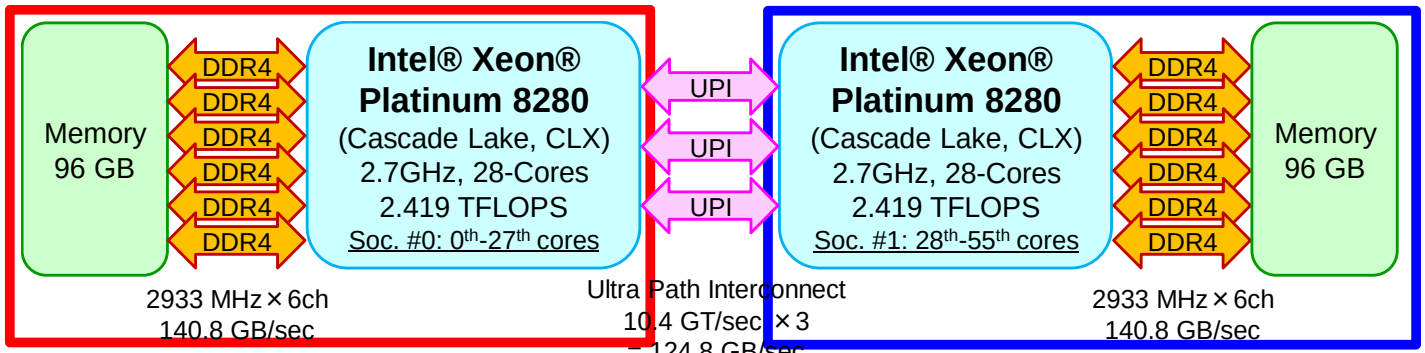


1-node, 24-cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=24
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
export I_MPI_PIN_PROCESSOR_LIST=0-23
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
```

24-cores are randomly selected from 56-cores on the node RANDOM

24-cores on Socket #0 are assigned. SPECIFIED



Socket #0: 0th-27th Cores

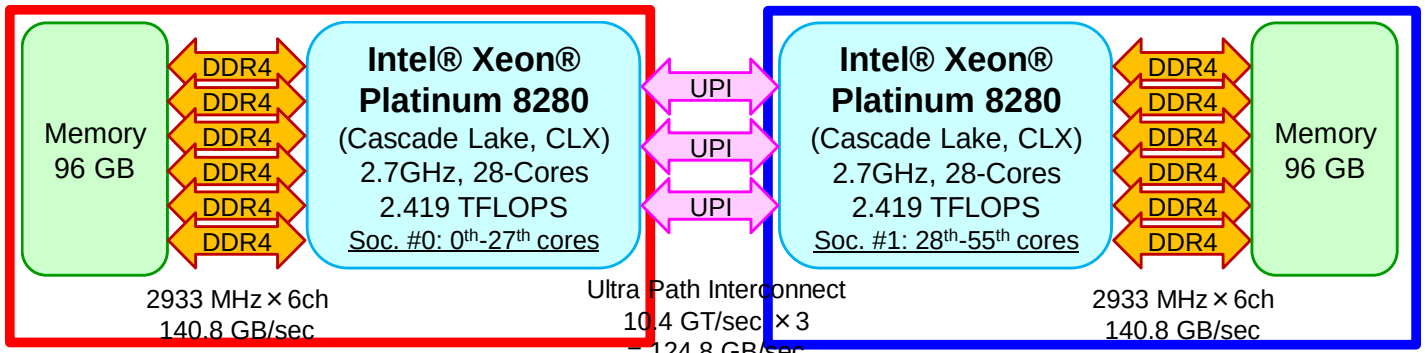
Socket #1: 28th-55th Cores

1-node, 28-cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=28
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst

mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
export I_MPI_PIN_PROCESSOR_LIST=0-27
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
```

28-cores are randomly selected from 56-cores on the node RANDOM

28-cores on Socket #0 are assigned. SPECIFIED



Socket #0: 0th-27th Cores

Socket #1: 28th-55th Cores

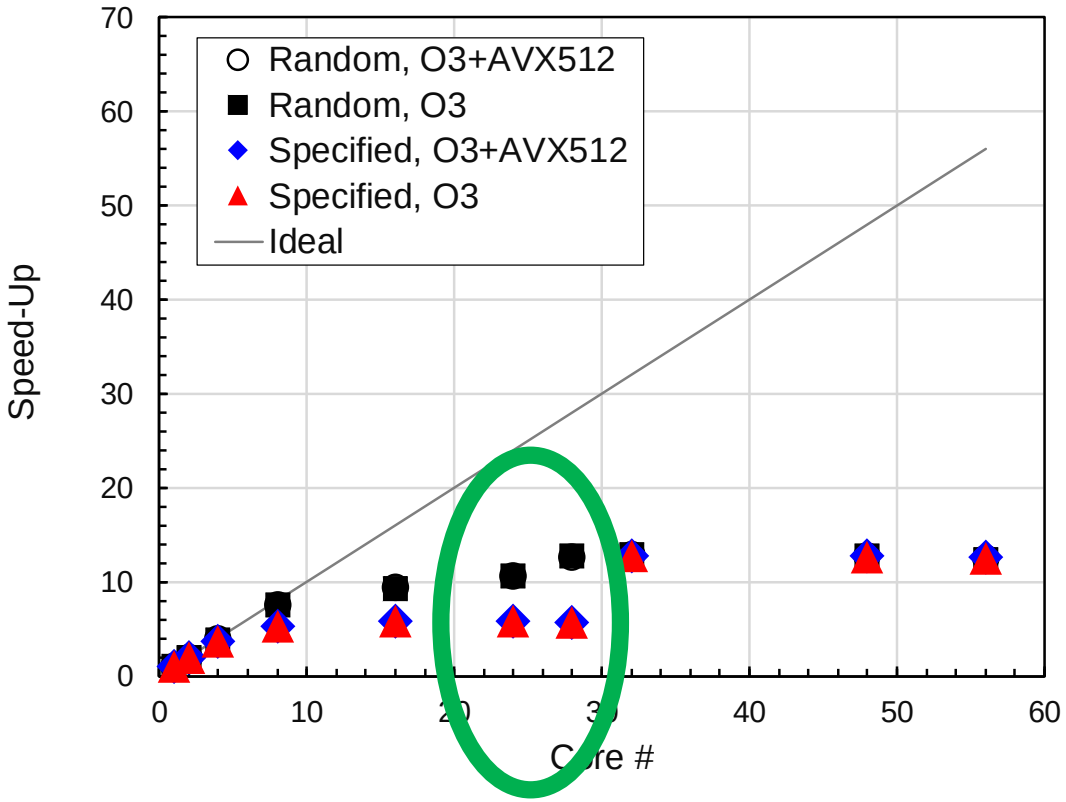
計算結果（1次元）：CG法部分，10^7

1,000回反復に要する時間，Strong Scaling

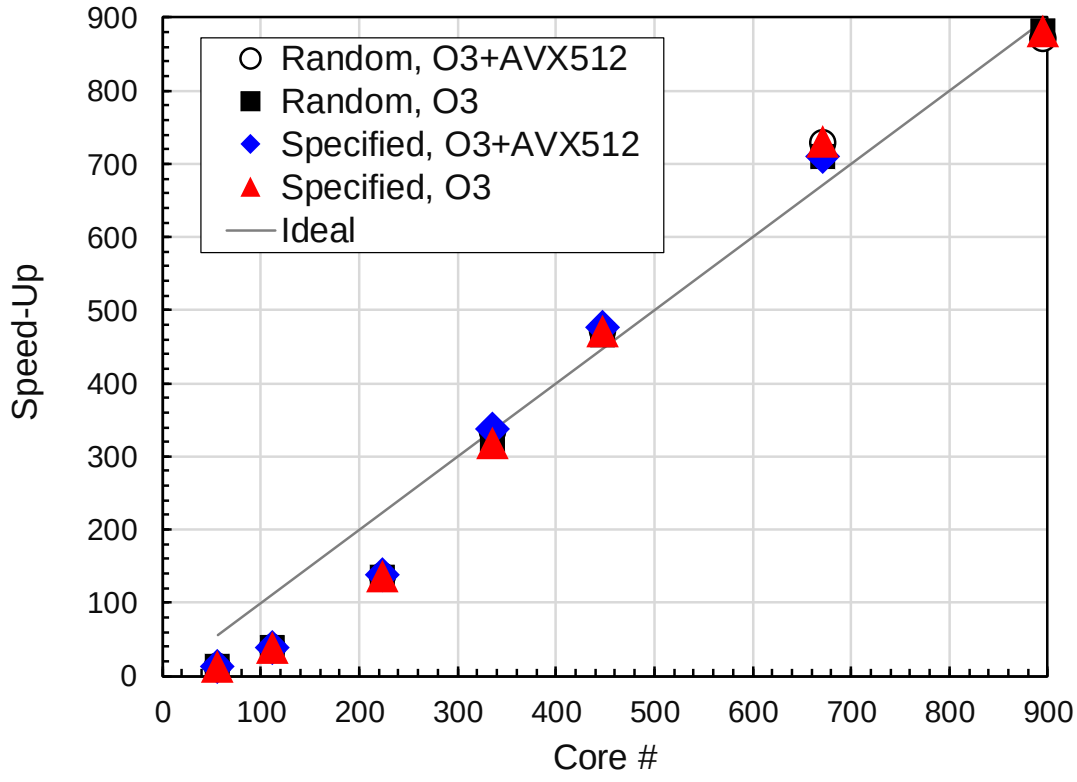
1コア計算時間を基準，2ノード以上では56コア/ノード使用

5回計測，最速のものを採択

up to 1 node, 56cores



up to 16 nodes, 896 cores

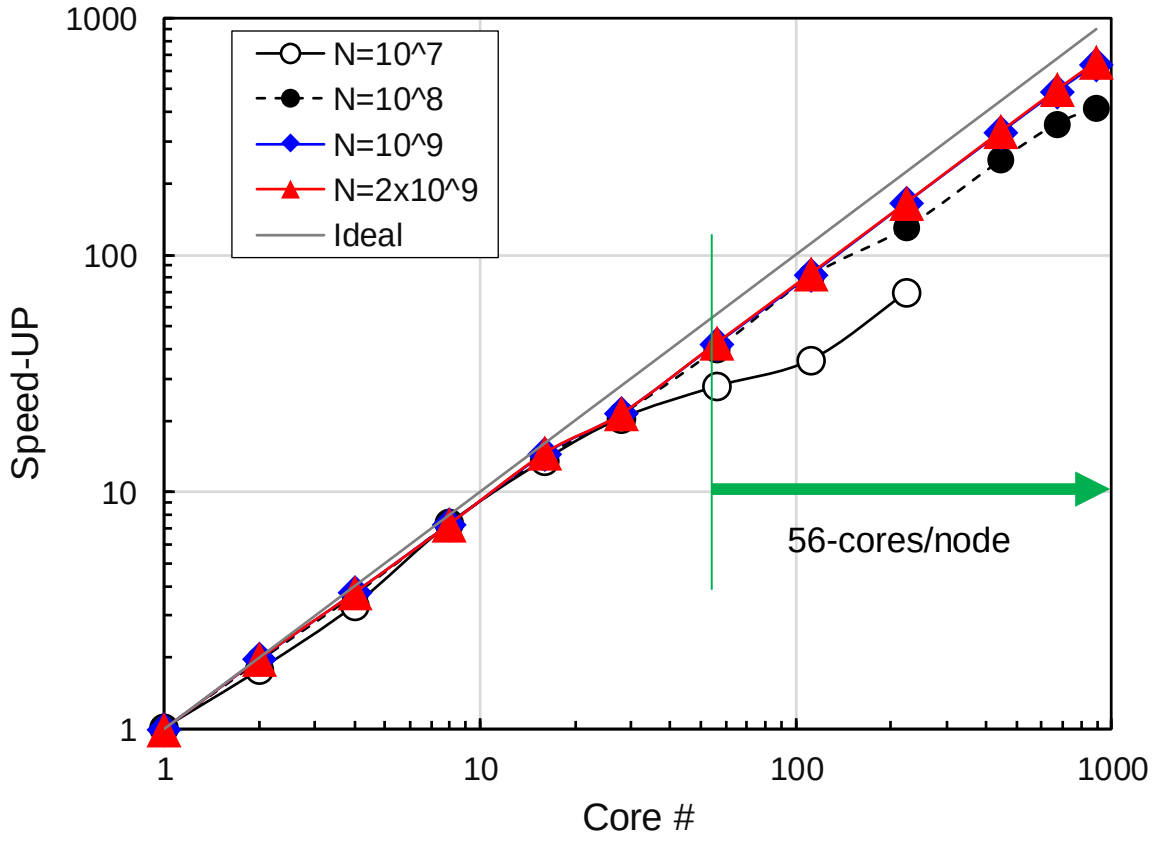


大規模並列計算：理想値からのずれ

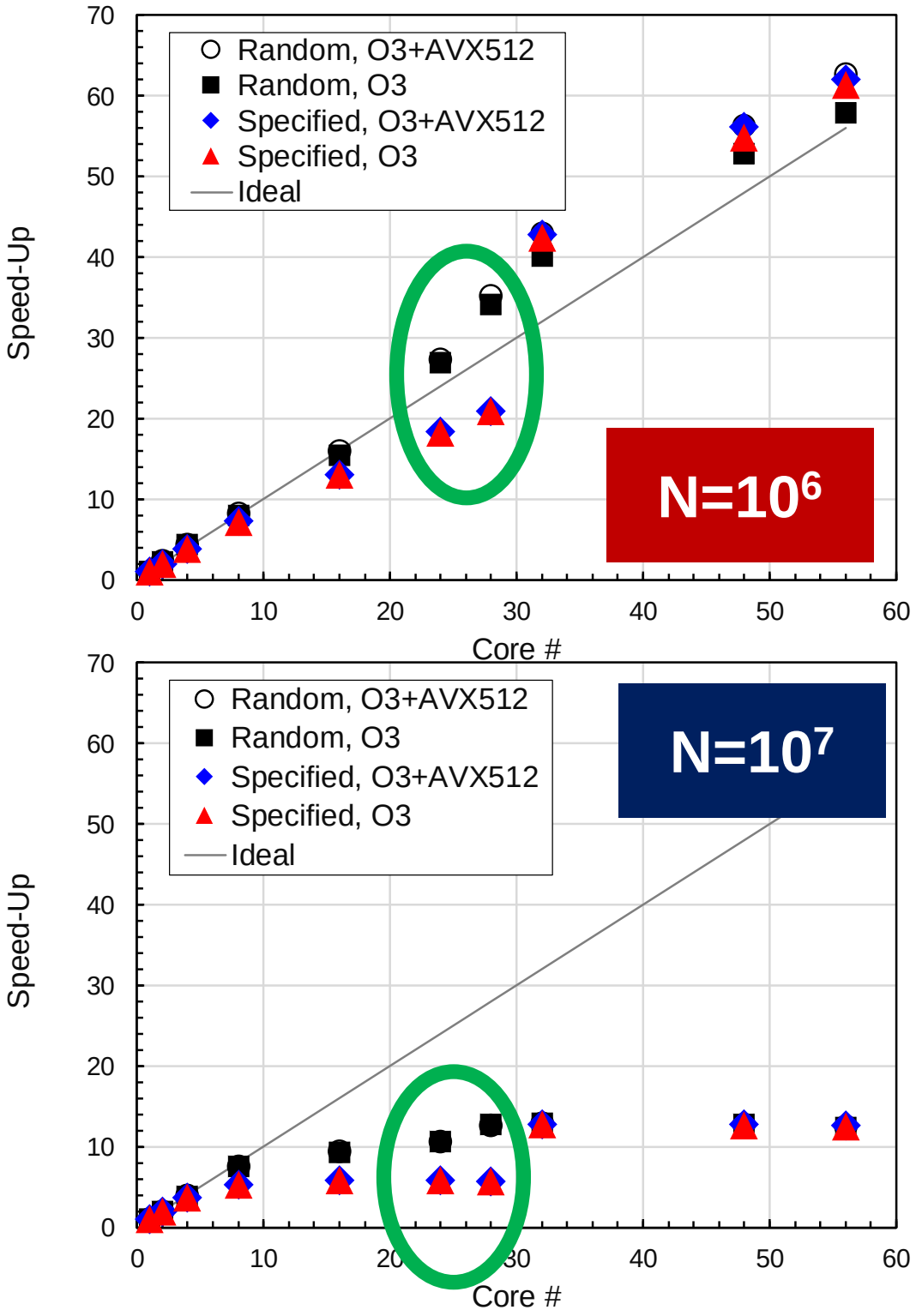
- MPI通信そのものに要する時間
  - データを送付している時間
  - ノード間においては通信バンド幅によって決まる
    - Gigabit Ethernetでは 1Gbit/sec.（理想値）
  - 通信時間は送受信バッファのサイズに比例
- MPIの立ち上がり時間
  - latency
  - 送受信バッファのサイズによらない
    - 呼び出し回数依存，プロセス数が増加すると増加する傾向
  - 通常，数～数十μsecのオーダー
- MPIの同期のための時間
  - プロセス数が増加すると増加する傾向
- 計算時間が小さい場合はこれらの効果を無視できない。
  - 特に，送信メッセージ数が小さい場合は，「Latency」が効く。

大規模並列計算：理想値からのずれ

- 問題サイズが比較的小さいと，この効果は無視できない
  - 通信メッセージのサイズが小さいと，Latencyの効果大



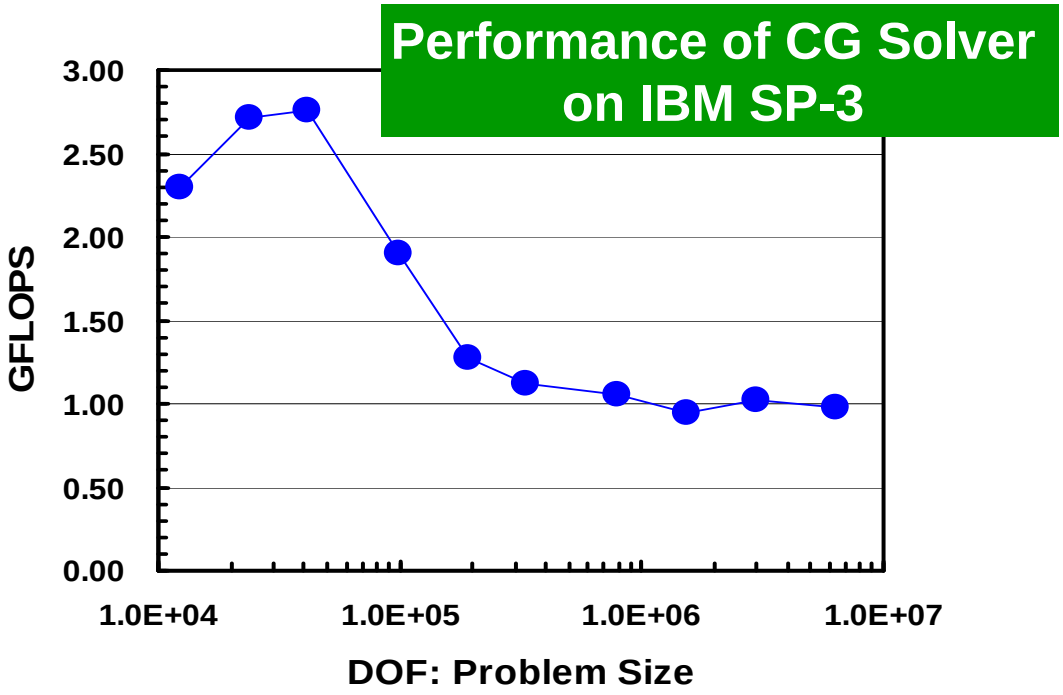
Up to 56 cores (single node)

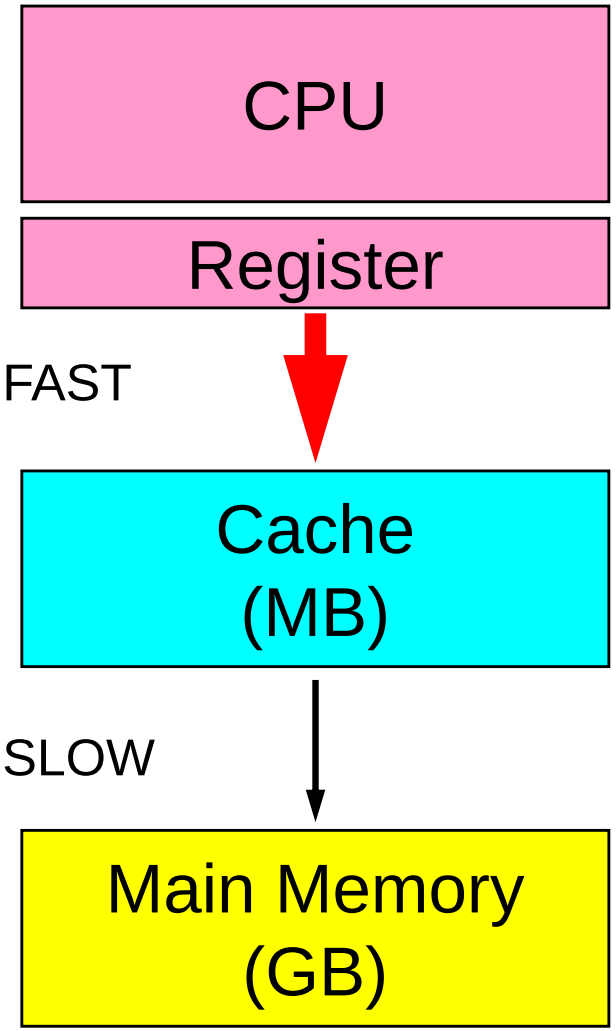


- ノード内（56コアまで）では問題規模が小さいと性能良い
- メモリ競合（通信の影響ではない）
- ノード内メモリ転送性能は8コア以上ではほとんど変化しない（Stream）
- 問題サイズが小さいとキャッシュを有効利用できるため，メモリ転送性能の影響少ない
- 2ソケット利用時のメモリ・キャッシュは「Random」の方が有効に利用されている

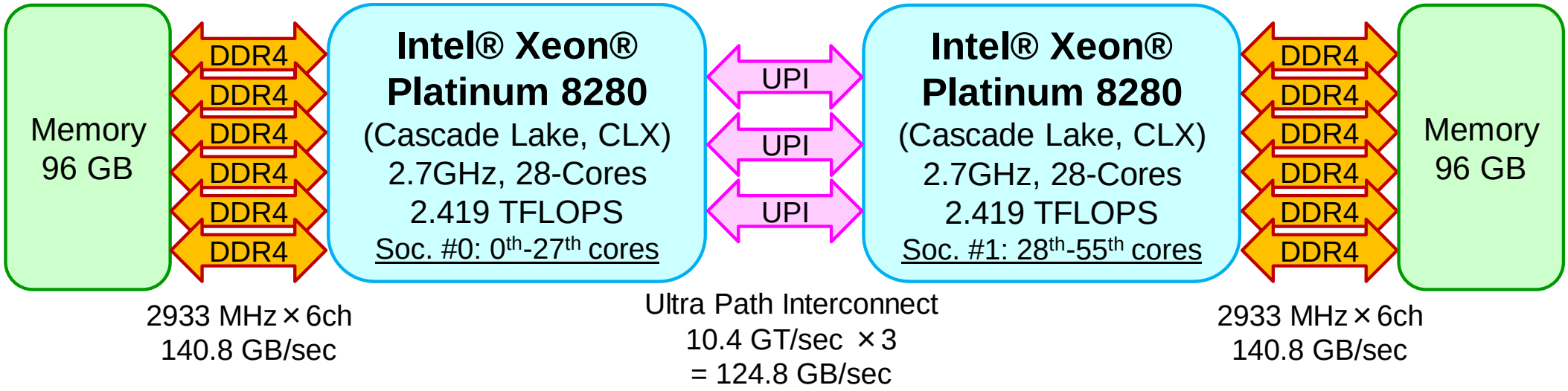
問題サイズが小さいとキャッシュを有効利用できる

- 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

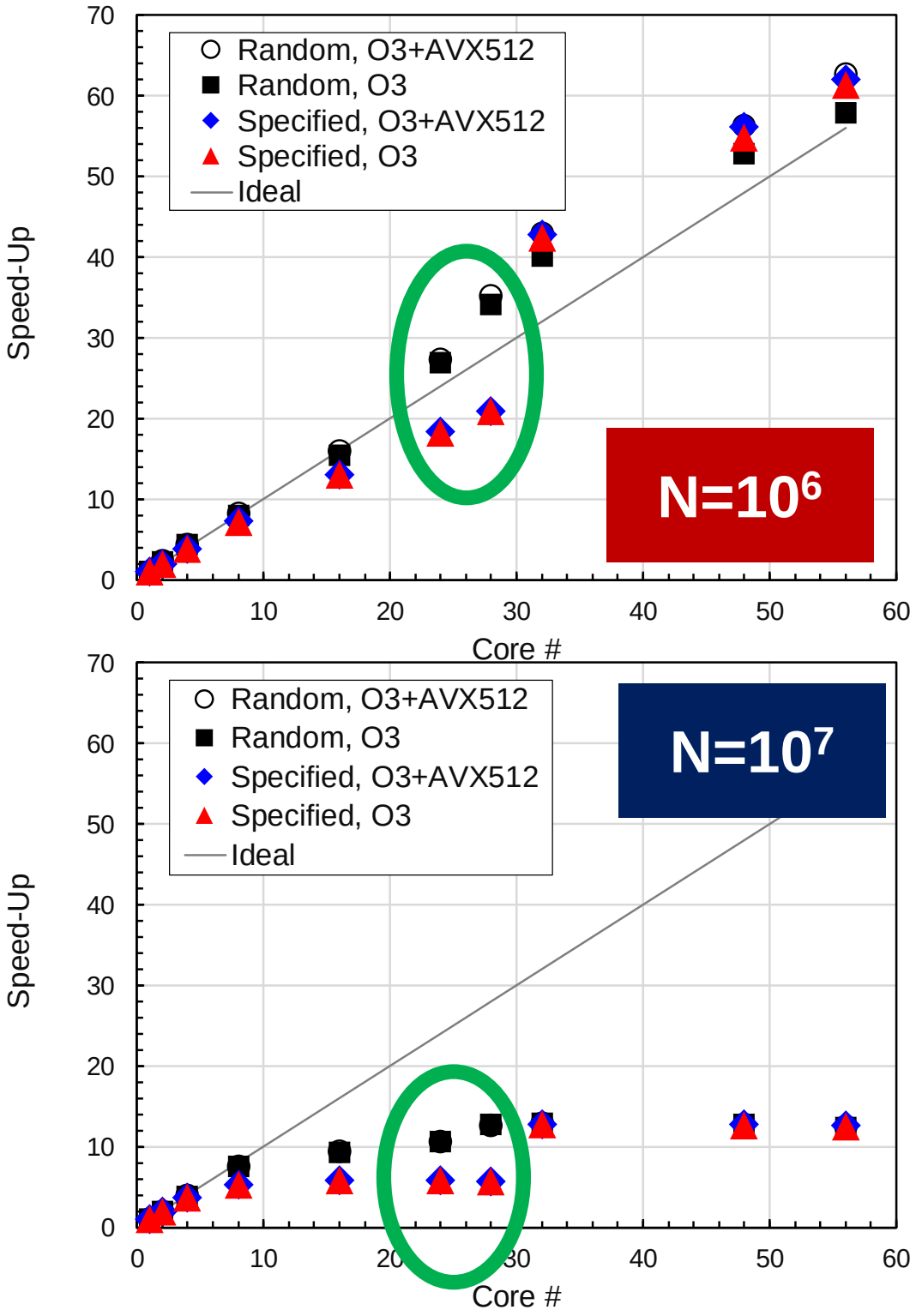




Category	Capacity	X-Way Set Associative	Cache Line
L1$Data	32 KB/core	8-Way	64B
L1$Instruction	32 KB/core	8-Way	64B
L2	1.00 MB/core	16-Way	64B
L3	38.5 MB/socket	11-Way	64B



Up to 56 cores (single node)



- Required Memory
- $N=10^6$: 80MB
  - Very Close to Cache Size
- $N=10^7$: 800MB

- Memory throughput on each socket is constant for 8+ cores (Stream)
- If problem size is small, cache can be well-utilized. Therefore, effect of memory bandwidth is small.

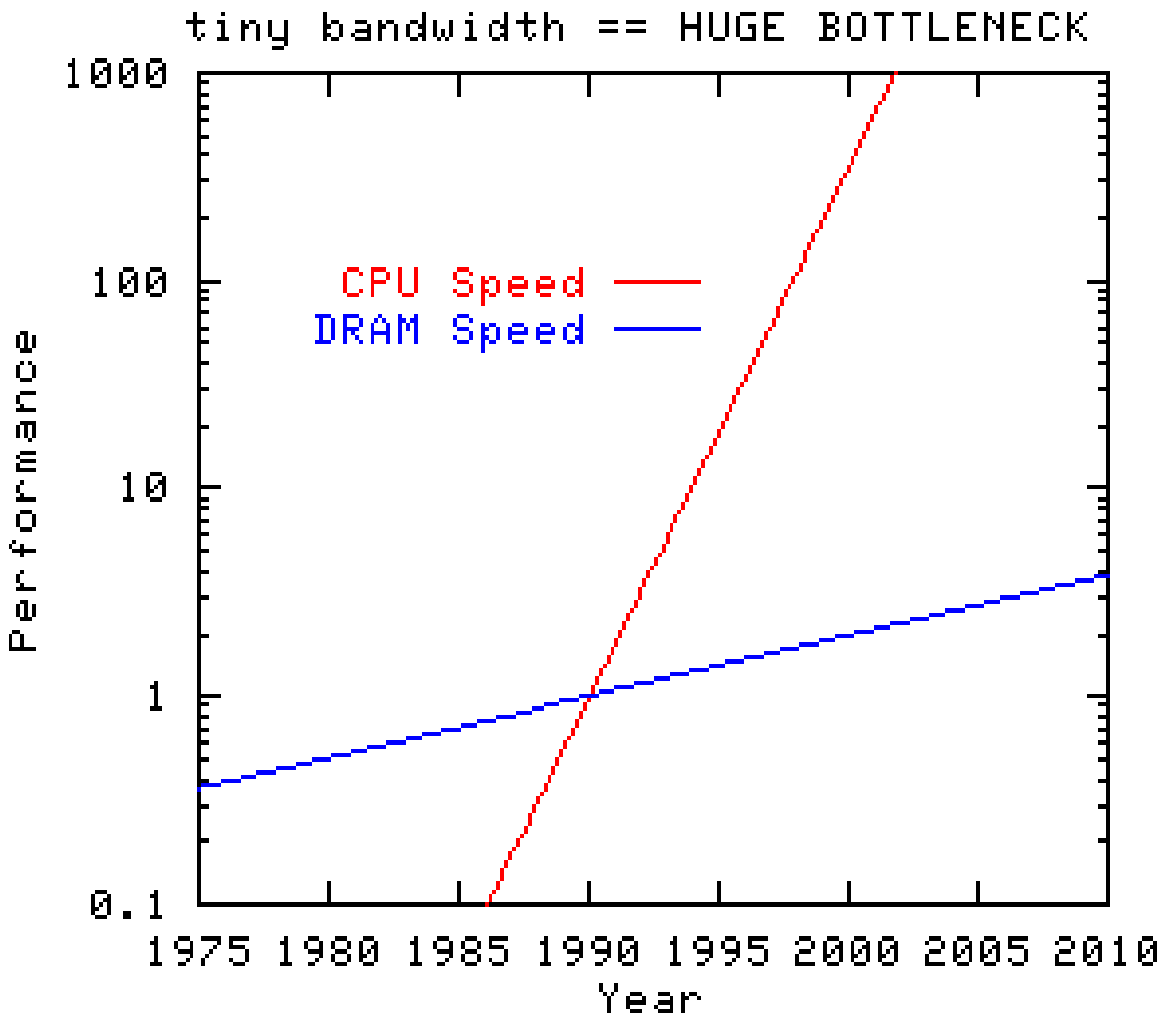
STREAM benchmark

http://www.cs.virginia.edu/stream/

- Benchmarks for Memory Bandwidth
  - Copy: c(i)= a(i)
  - Scale: c(i)= s*b(i)
  - Add: c(i)= a(i) + b(i)
  - Triad: c(i)= a(i) + s*b(i)

```
-------------------------------------------------------------
Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word
-------------------------------------------------------------
Number of processors =           16
Array size =    2000000
Offset     =          0
The total memory requirement is     732.4 MB
(     45.8MB/task)
You are running each test  10 times
--
The *best* time for each test is used
*EXCLUDING* the first and last iterations
-------------------------------------------------------------
-------------------------------------------------------------
Function     Rate (MB/s)  Avg time    Min time  Max time
Copy:       18334.1898       0.0280       0.0279       0.0280
Scale:      18035.1690       0.0284       0.0284       0.0285
Add:        18649.4455       0.0412       0.0412       0.0413
Triad:      19603.8455       0.0394       0.0392       0.0398
```

マイクロプロセッサの動向
CPU性能，メモリバンド幅のギャップ



http://www.cs.virginia.edu/stream/

Sparse/Dense Matrices（疎・密）

```
for (i=0;i<N;i++ {
  Y[i]= D[i]*X[i];
for (k=index[i];k<index[i+1];k++) {
    Y[k]~ += AMAT[k]*X[item[k]];
              }
            }
```

```
for (i=0;i<N;i++){
 for (j=0;j<N;j++) {
 Y[i] += A[i][j]*X[j];
          }
        }
```

- “X” in RHS
  - 密行列：連続アクセス，キャッシュを有効利用
  - 疎行列：連続性は保証されていない，キャッシュ有効利用は困難
    - より”memory-bound”

GeoFEM Benchmark

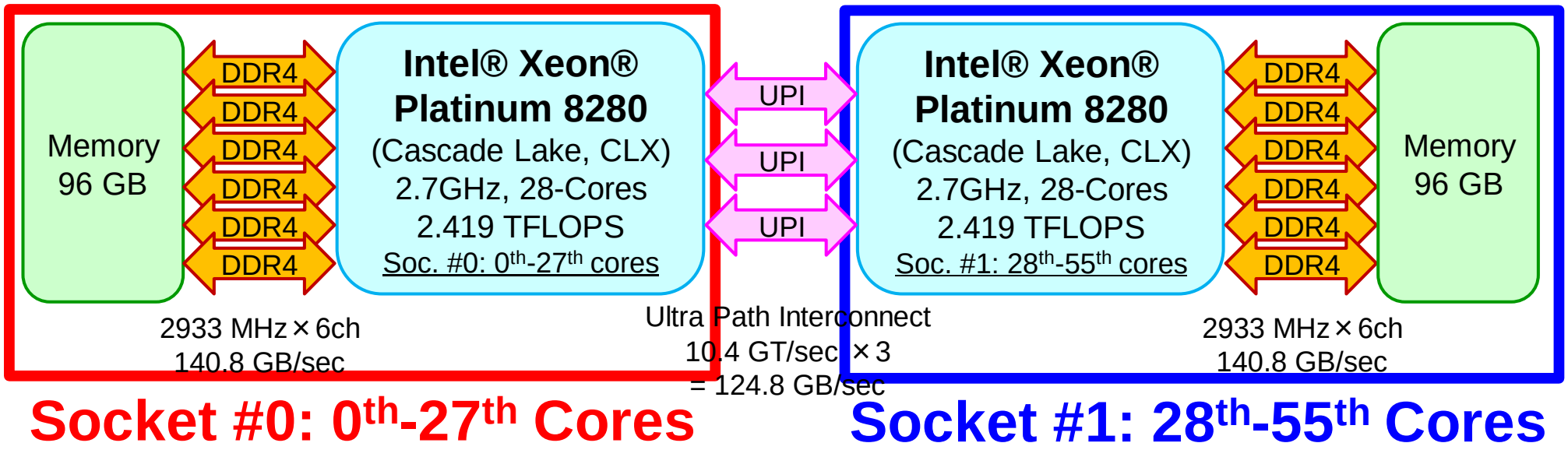
ICCG in FEM for Solid Mechanics

	SR11K/J2	SR16K/M1	T2K	FX10	京
Core #/Node	16	32	16	16	8
Peak Performance (GFLOPS)	147.2	980.5	147.2	236.5	128.0
STREAM Triad (GB/s)	101.0	264.2	20.0	64.7	43.3
B/F	0.686	0.269	0.136	0.274	0.338
GeoFEM (GFLOPS)	19.0	72.7	4.69	16.0	11.0
% to Peak	12.9	7.41	3.18	6.77	8.59
LLC/core (MB)	18.0	4.00	2.00	0.75	0.75

Sparse Linear Solver: Memory-Bound

Copy, Compile and Run

```
>$ cd /work/gt00/t00XXX/pFEM
>$ cp /work/gt00/z30088/pFEM/F/stream.tar .
>$ tar xvf stream.tar
>$ cd mpi/stream

>$ mpiifort -align array64byte -O3 -axCORE-AVX512 stream.f -o stream
>$ pjsub XXX.sh
```

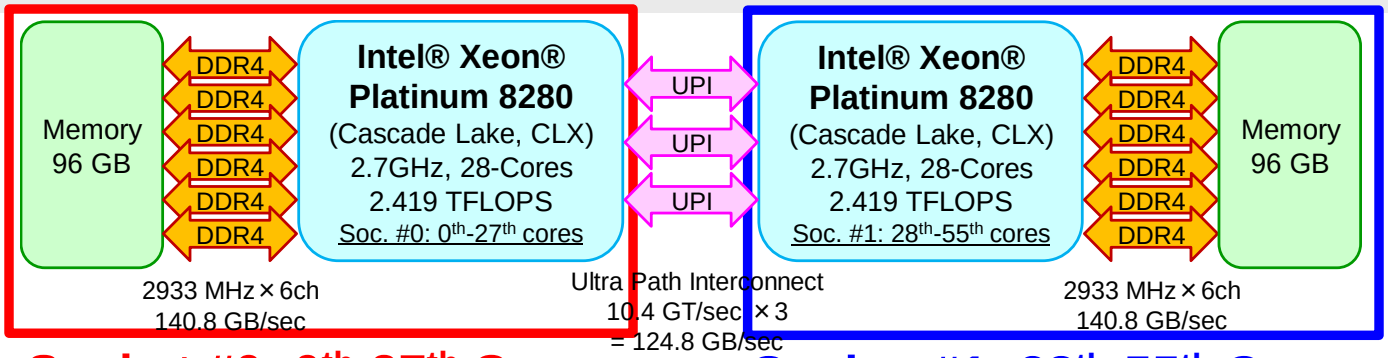


s01.sh: Use 1 core

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s01.lst

mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

Core are specified



Socket #0: 0th-27th Cores

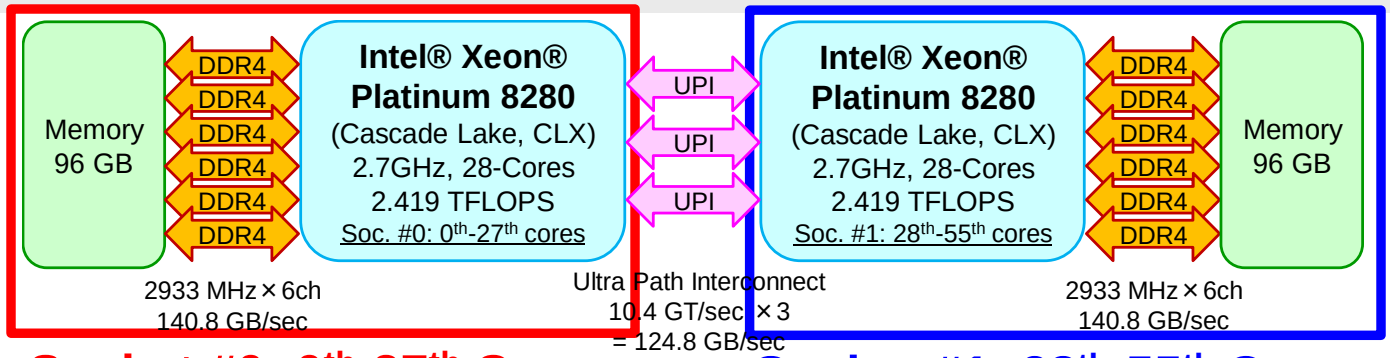
Socket #1: 28th-55th Cores

s16.sh: Use 16 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=16
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s16.lst

mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-15
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

Core are specified



Socket #0: 0th-27th Cores

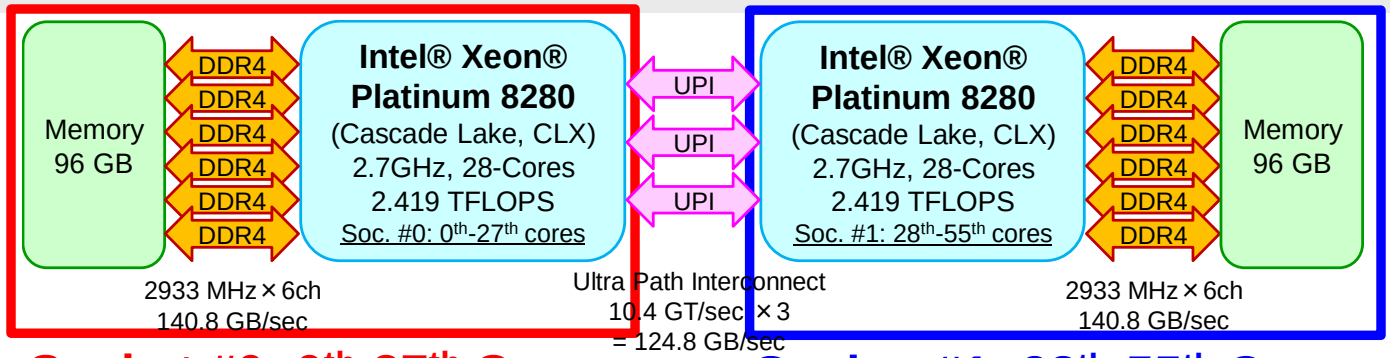
Socket #1: 28th-55th Cores

s32.sh: Use 32 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=32
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s32.lst

mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-15,28-43
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

Core are specified

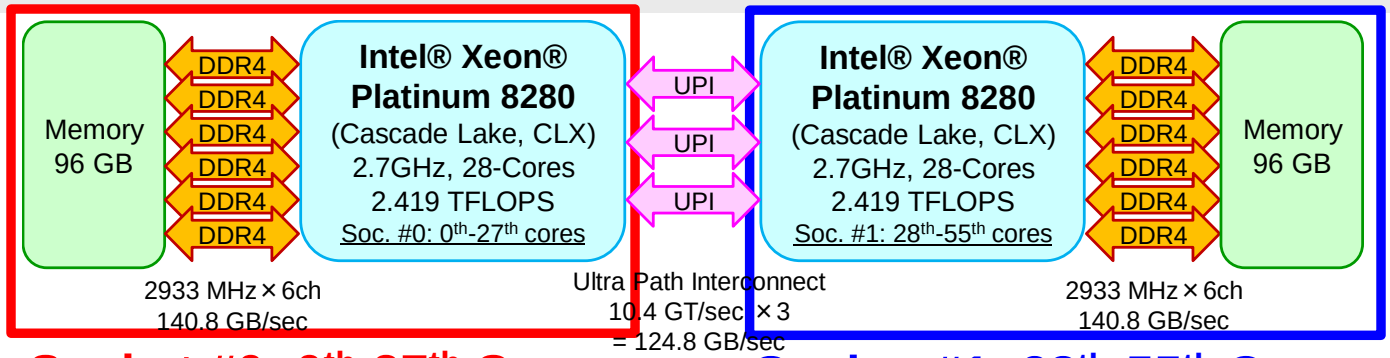


s48.sh: Use 48 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=48
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s48.lst

mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

Core are specified

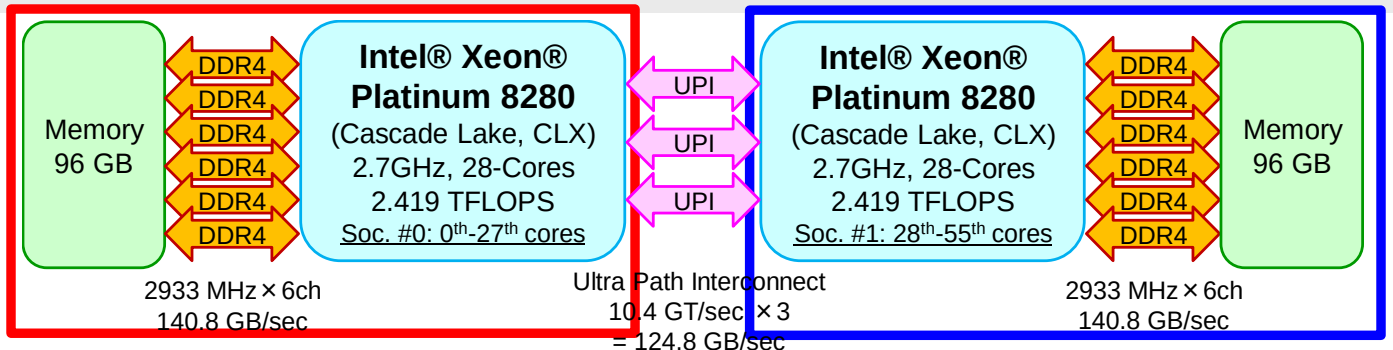


s56.sh: Use 56 cores

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=56
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s56.lst

mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream

export I_MPI_PIN_PROCESSOR_LIST=0-55
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

Core are specified

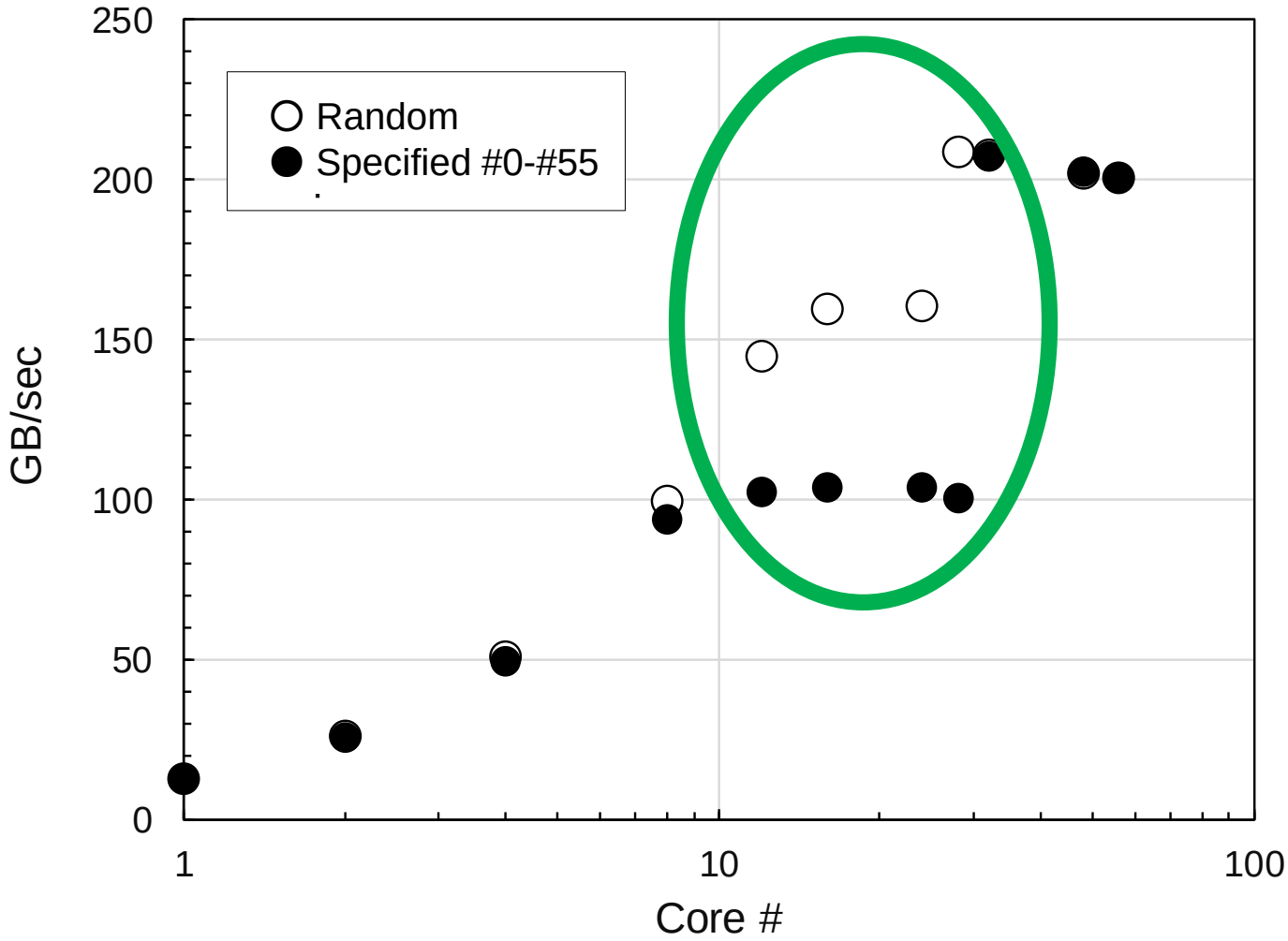


Socket #0: 0th-27th Cores

Socket #1: 28th-55th Cores

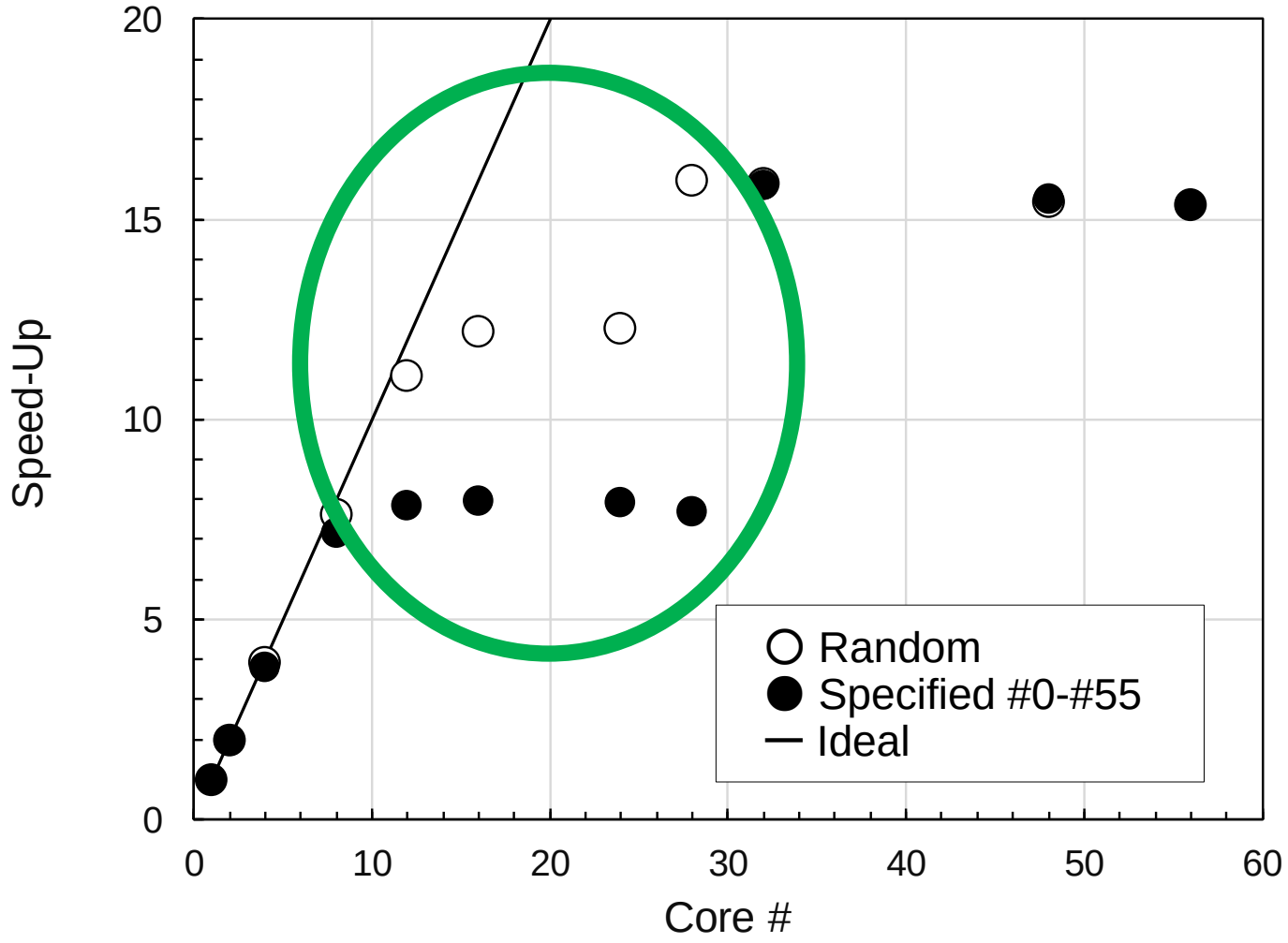
Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

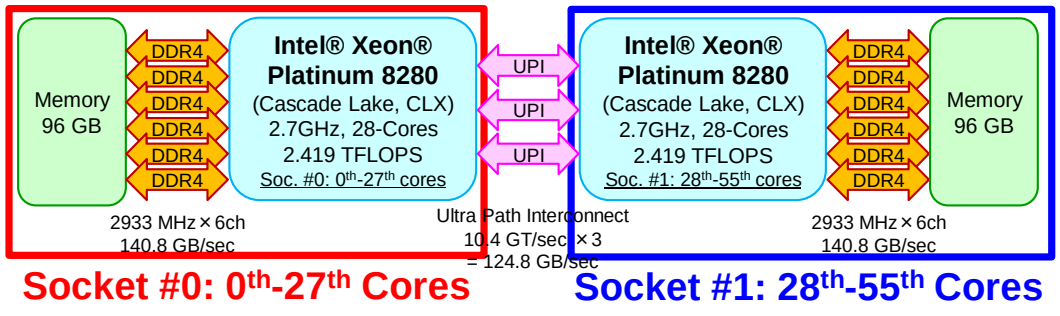
メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）



Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）





Socket #	Core #	Random	Specified
1	1	1.000	1.000
	2	1.998	1.963
	4	3.912	3.809
	6	5.792	5.682
	8	7.615	7.177
	12	11.089	7.844
	16	12.214	7.968
	24	12.278	7.945
	28	15.962	7.705
2	32	15.901	15.862
	48	15.452	15.497
	56	15.370	15.358

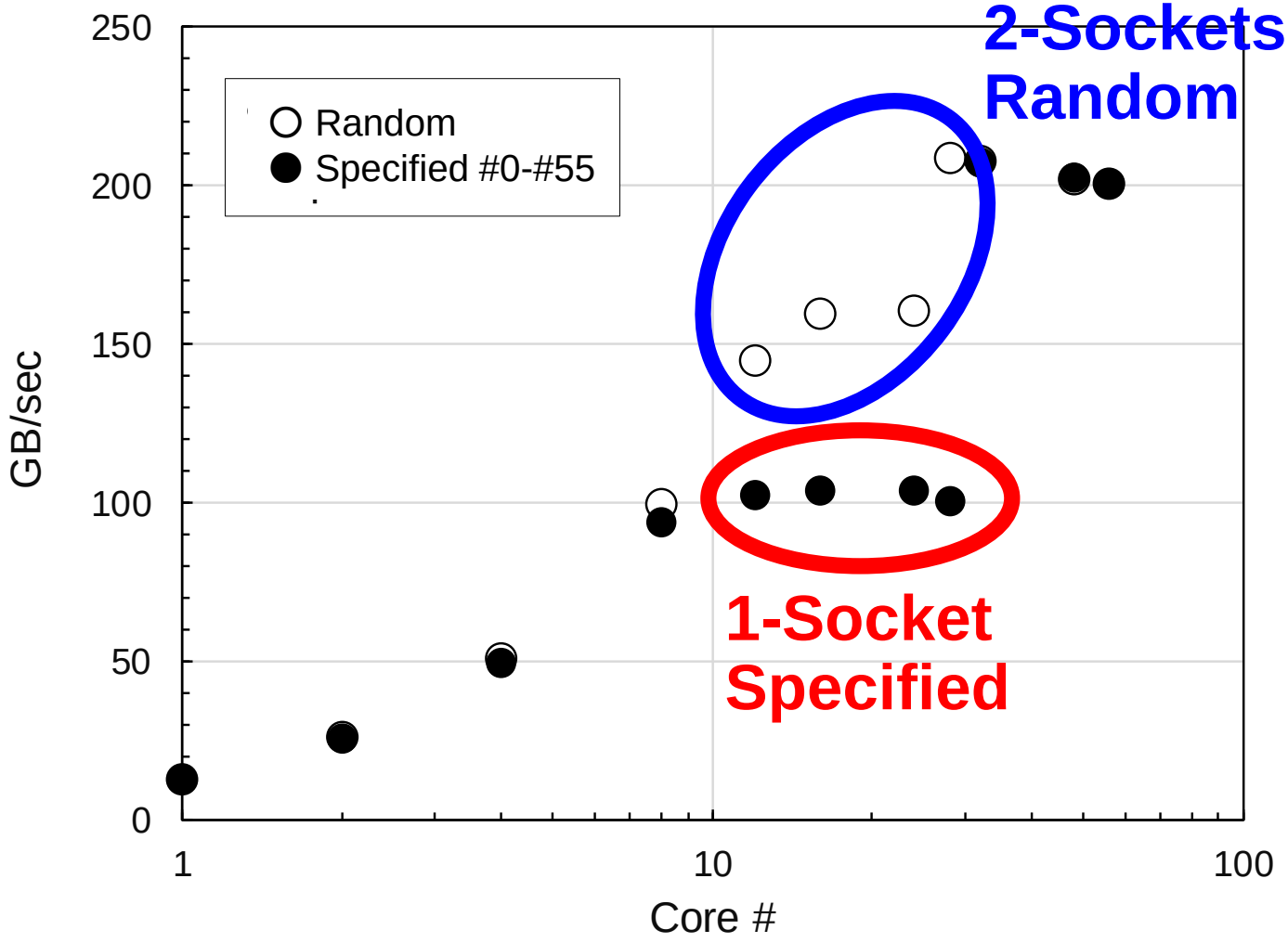
Triad on a Single Node of OBCX

1コアに対する性能向上（Speed-Up）

- ソケット当たりメモリチャンネル数=6
  - 各ソケットに6枚のメモリを装着できる
  - 1ソケット6コア（6プロセス）まではほぼ比例
- 12-28 cores
  - **Random**：2ソケット利用
  - **Specified/Compact**：1ソケット

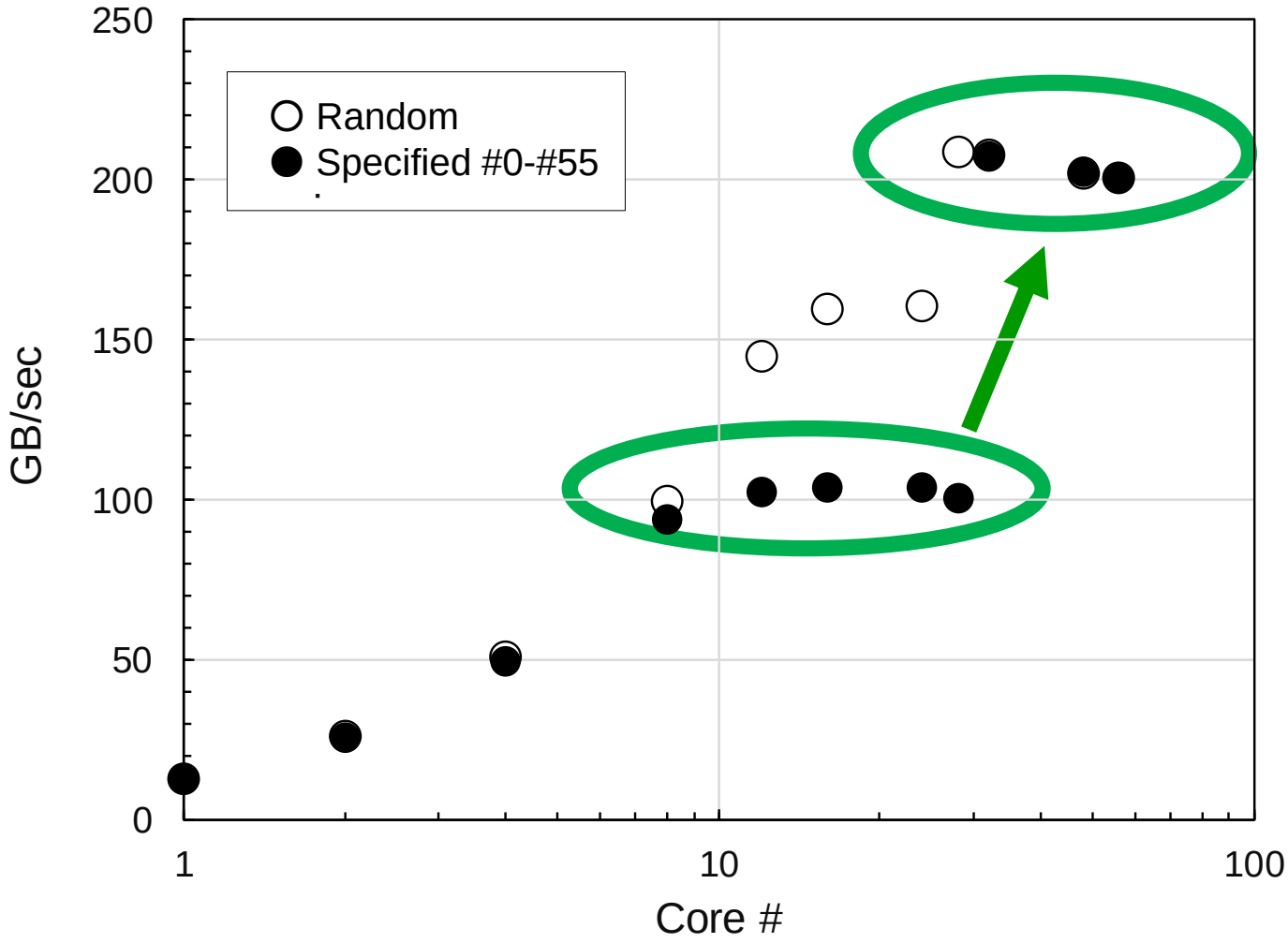
Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）



Triad on a Single Node of OBCX Peak is 281.57 GB/sec.

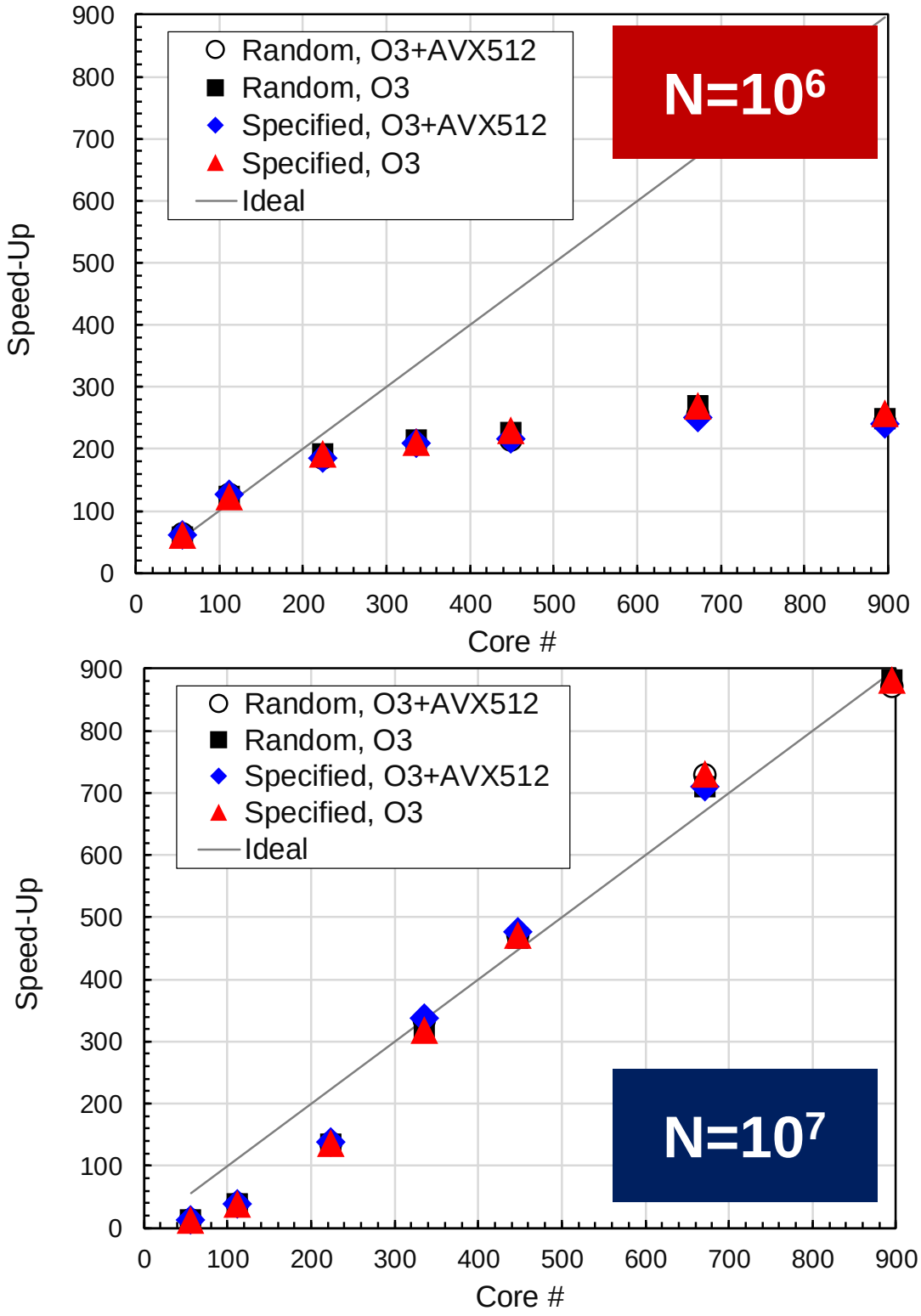
●：メモリバンド幅は8-28コアでほぼ一定（飽和）
32-56コアでは倍になる



Exercises

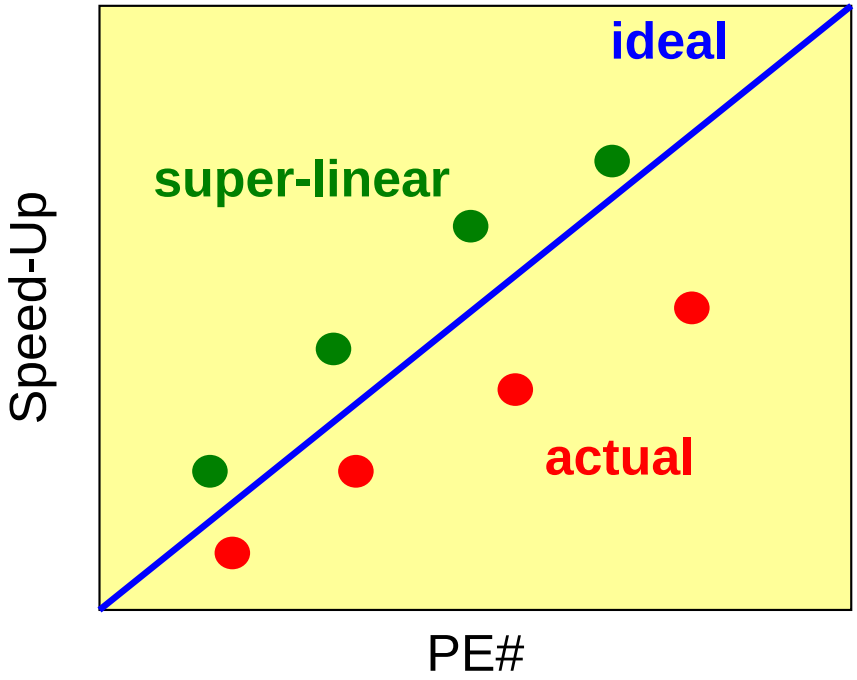
- Running the code
- Try various number of processes (1-56)
- OpenMP-version and Single PE version are available
  - Fortran, C
  - Web-site of STREAM
  - http://www.cs.virginia.edu/stream/

Up to 896 cores (16-nodes)



- Performance at a Single Core= 1.00
- ノード数が増すとN=10^6のケースは効率低下（台形積分と同様）
- N=10^7のケースは徐々にidealに近づき，672コア（12ノード）のケースではsuper linearになる

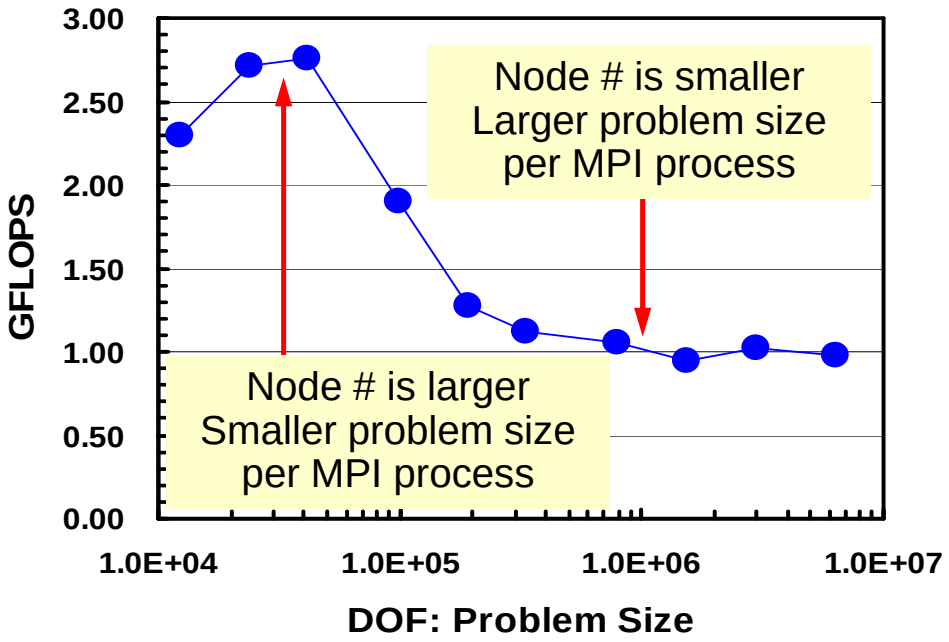
Strong-Scalingにおける「Super-Linear」

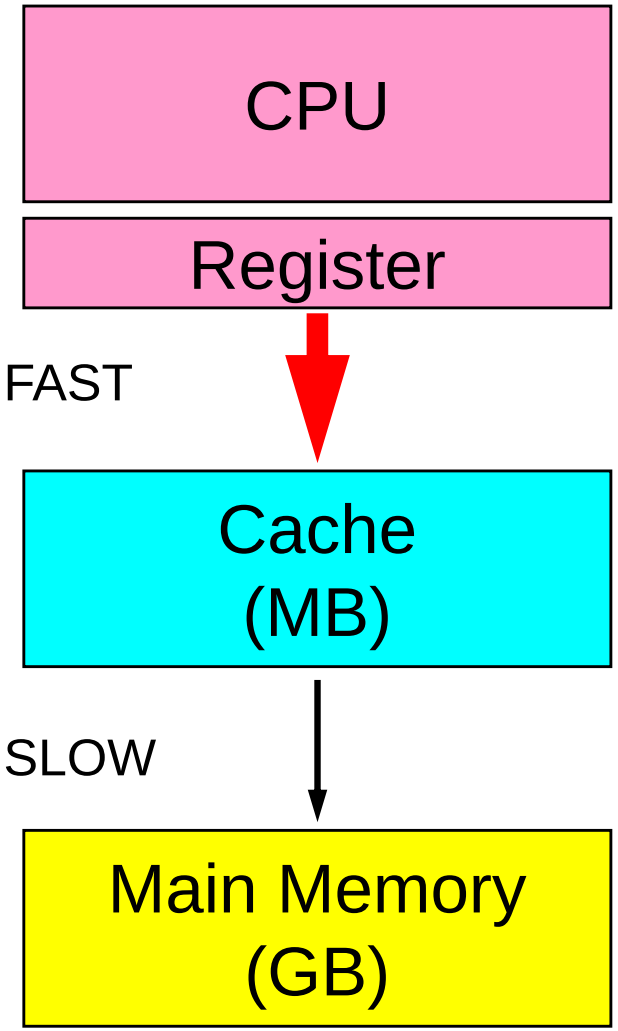


- 問題規模を固定して，使用PE数を増加させて行った場合，通常は通信の影響のために，効率は理想値（m個のPEを使用した場合，理想的にはm倍の性能になる）よりも低くなるのが普通である。
- しかし，スカラープロセッサ（PC等）の場合，逆に理想値よりも，高い性能が出る場合がある。このような現象を「Super-Linear」と呼ぶ。
  - ベクトル計算機では起こらない。

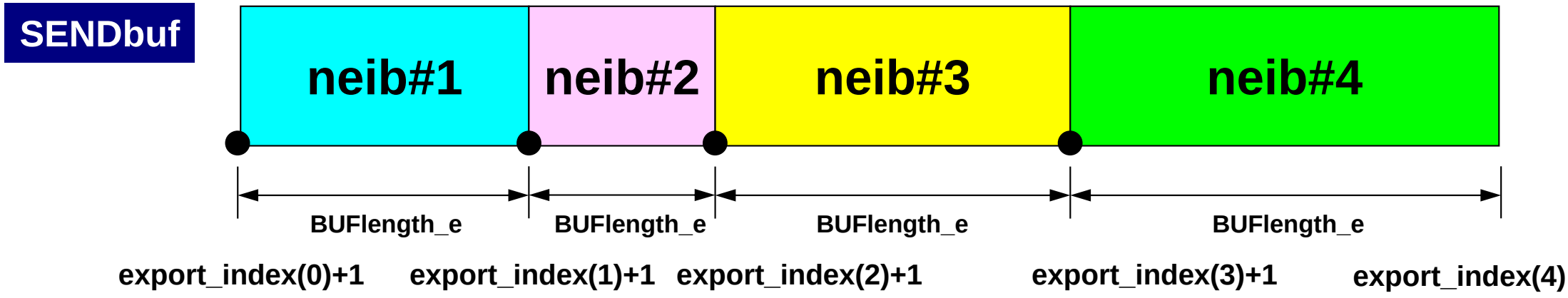
Super-Linearの生じる理由

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





メモリーコピーも意外に時間かかる（1/2）



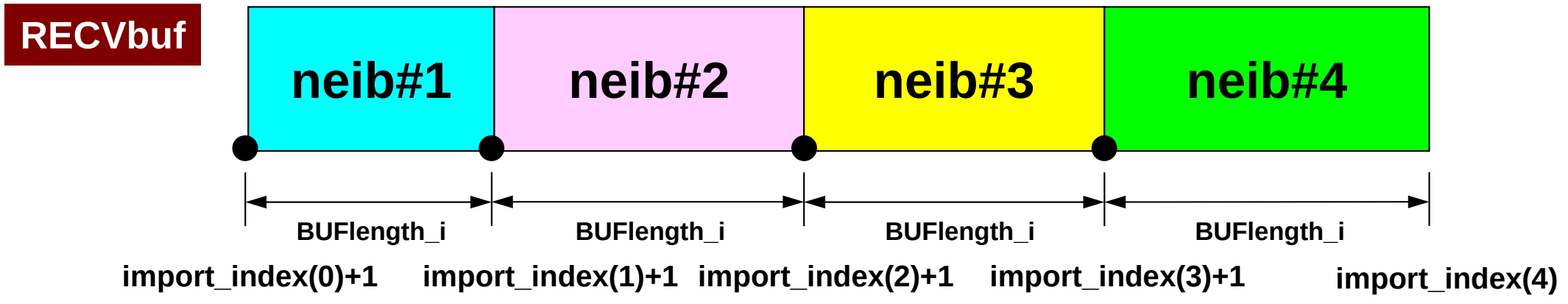

```
 do neib= 1, NEIBPETOT
   do k= export_index(neib-1)+1, export_index(neib)
     kk= export_item(k)
     SENDbuf(k)= VAL(kk)
   enddo
 enddo

 do neib= 1, NEIBPETOT
   iS_e= export_index(neib-1) + 1
   iE_e= export_index(neib  )
   BUFlength_e= iE_e + 1 - iS_e

   call MPI_ISEND                                                  &
&          (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&           MPI_COMM_WORLD, request_send(neib), ierr)
 enddo

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

Copied to sending buffers

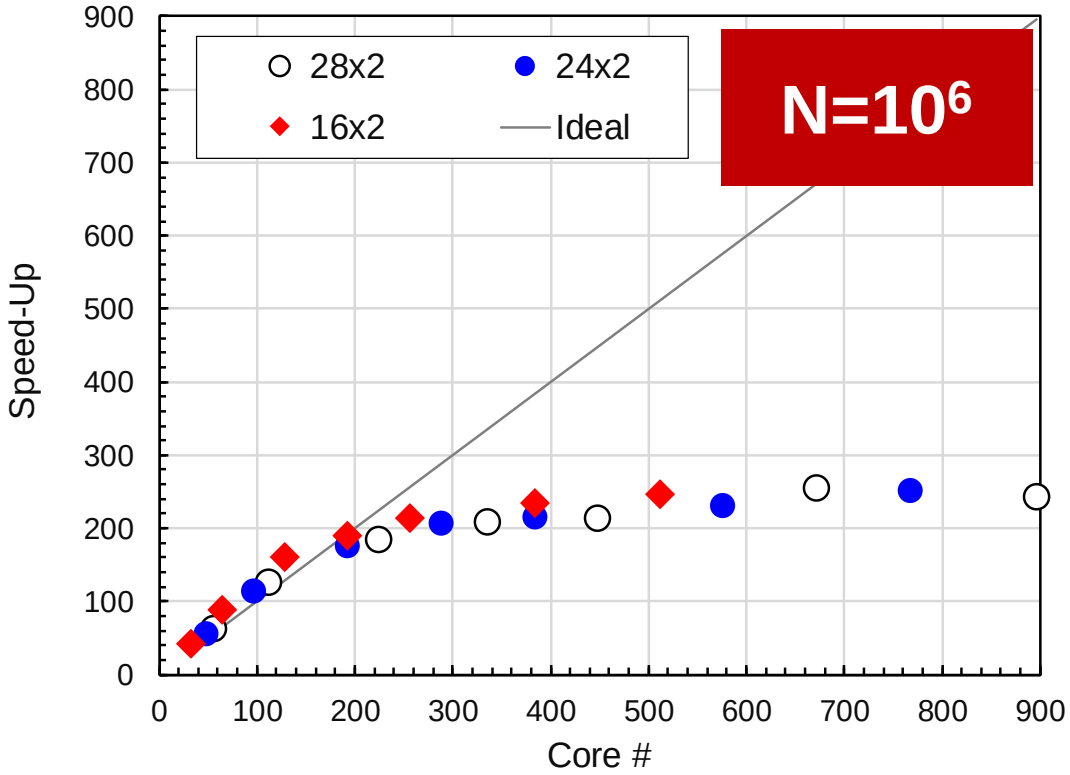
メモリーコピーも意外に時間かかる（2/2）

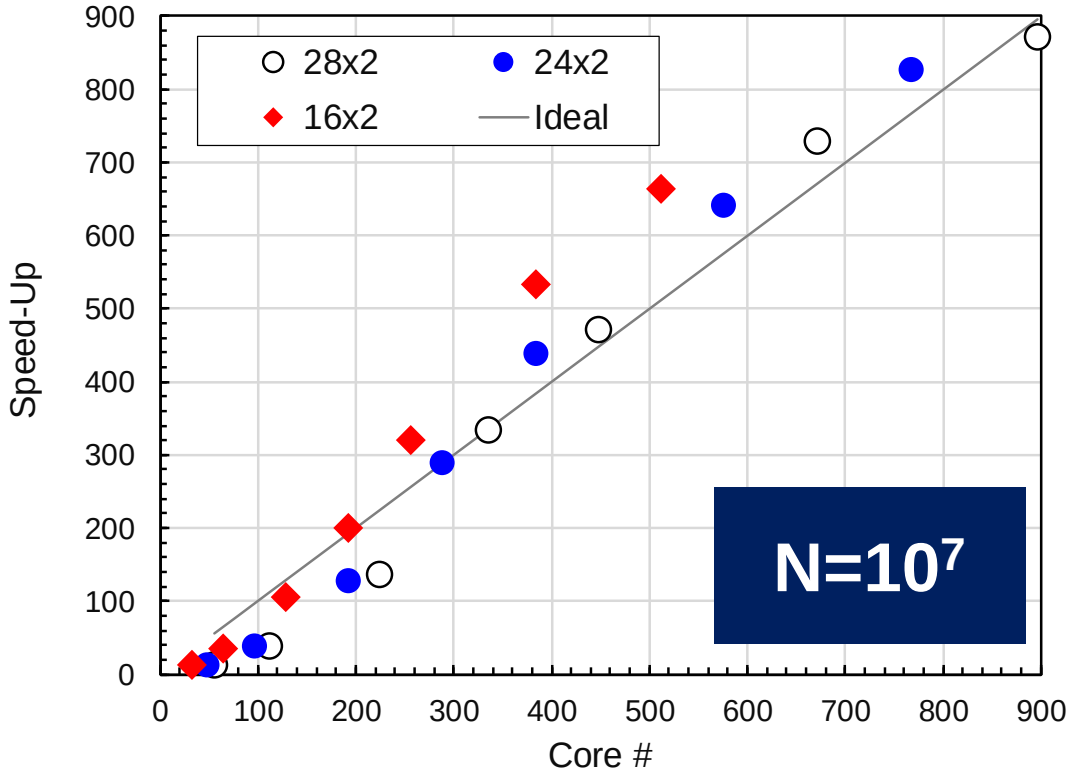
```
 do neib= 1, NEIBPETOT
   iS_i= import_index(neib-1) + 1
   iE_i= import_index(neib  )
   BUFlength_i= iE_i + 1 - iS_i

   call MPI_IRECV                                                  &
&          (RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&           MPI_COMM_WORLD, request_recv(neib), ierr)
  enddo

  call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

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

Copied from receiving buffer

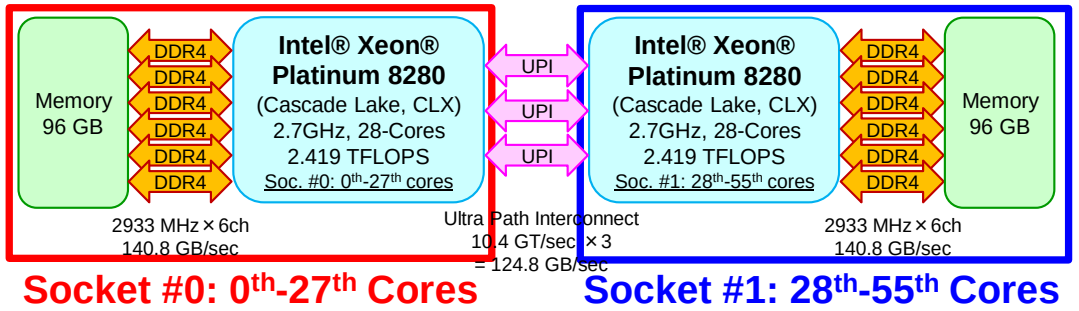


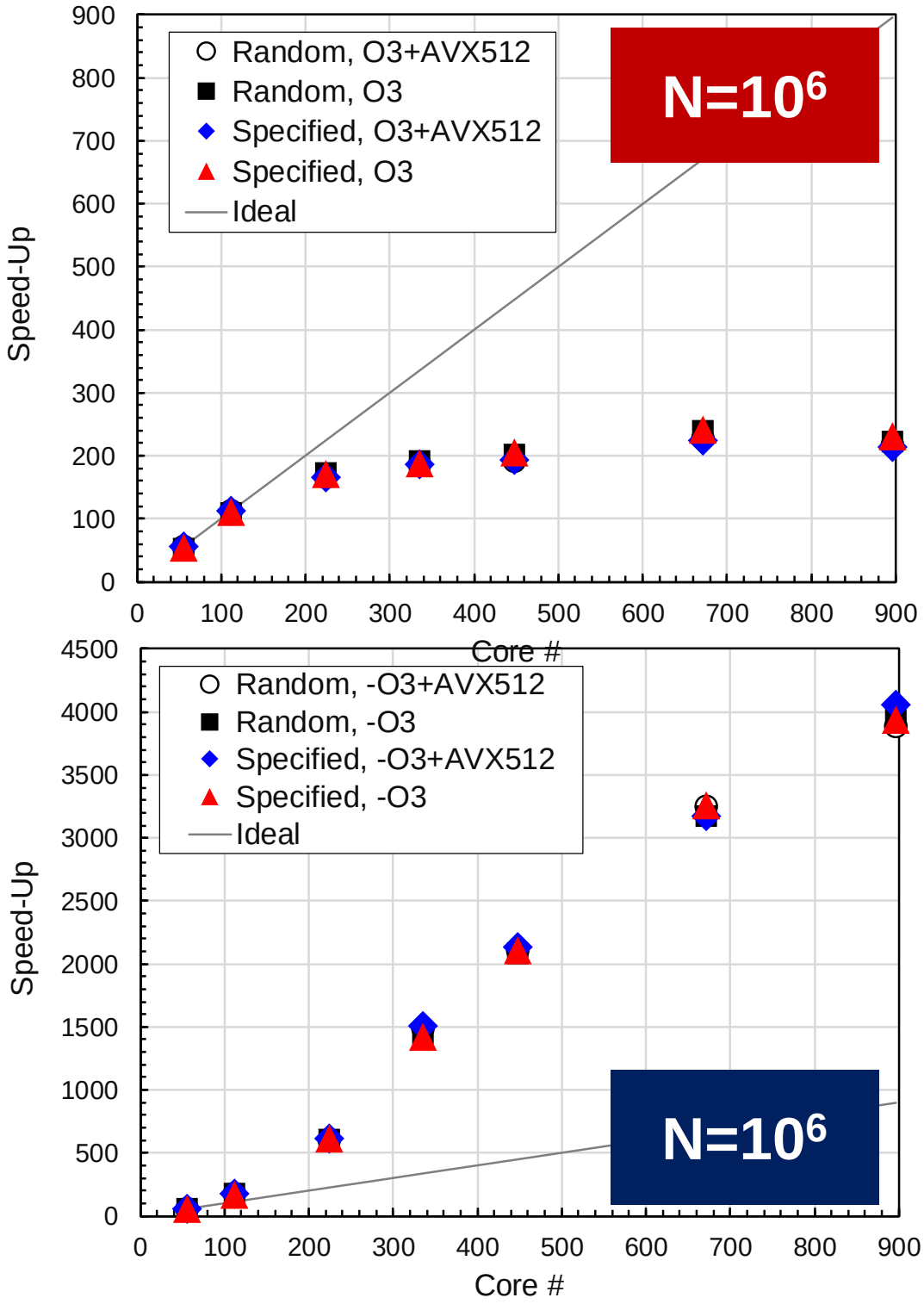
Up to 16-nodes
Random, O3+AVX512





- MPI Processes per Node
  - 16x2, 24x2, 28x2
- N=10^6
  - ノード数・コア数が増えると差異少ない
- N=10^7
  - コア数が同じであれば16x2が最良，メモリをより効率的に利用

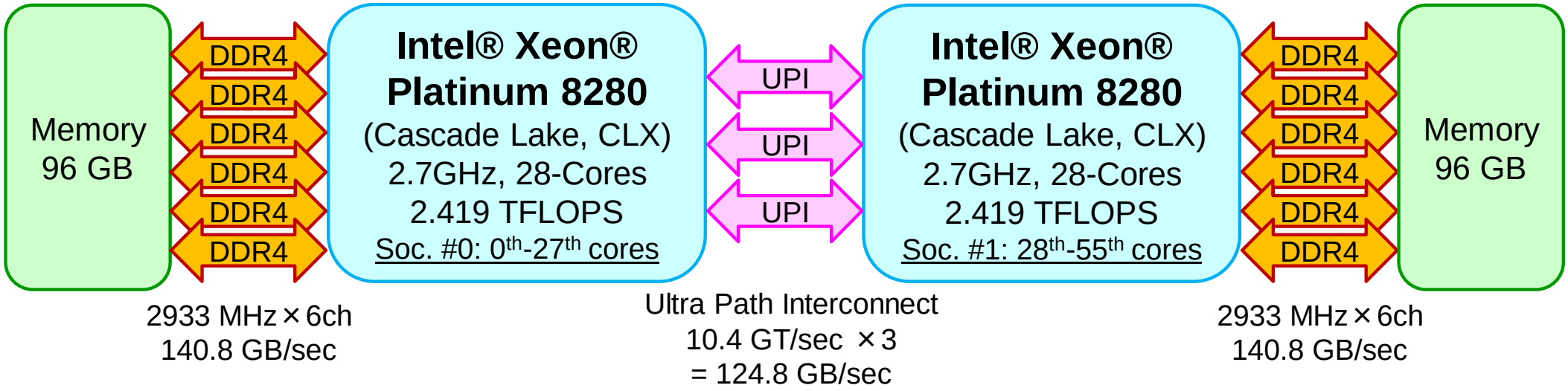




Up to 896 cores (16-nodes)

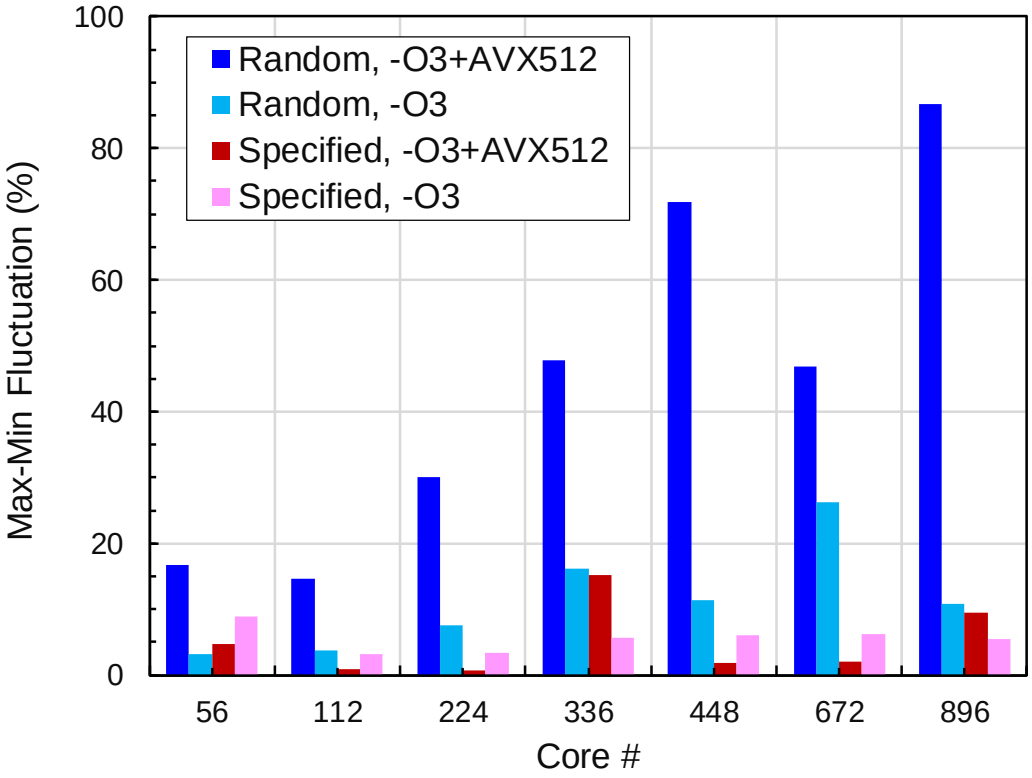
- 56コア（1ノード）の性能を56.0とする
- 複数ノードの場合には，1ノード（56コア）での値を基準とするのがreasonable
  - L2 cache: 1MB/core
  - L3: 38.5MB/socket (shared)
- $N=10^6$
  - 通信オーバーヘッド大
- $N=10^7$
  - Superlinear強い

Category	Capacity	X-Way Set Associative	Cache Line
L1$Data	32 KB/core	8-Way	64B
L1$Instruction	32 KB/core	8-Way	64B
L2	1.00 MB/core	16-Way	64B
L3	38.5 MB/socket	11-Way	64B

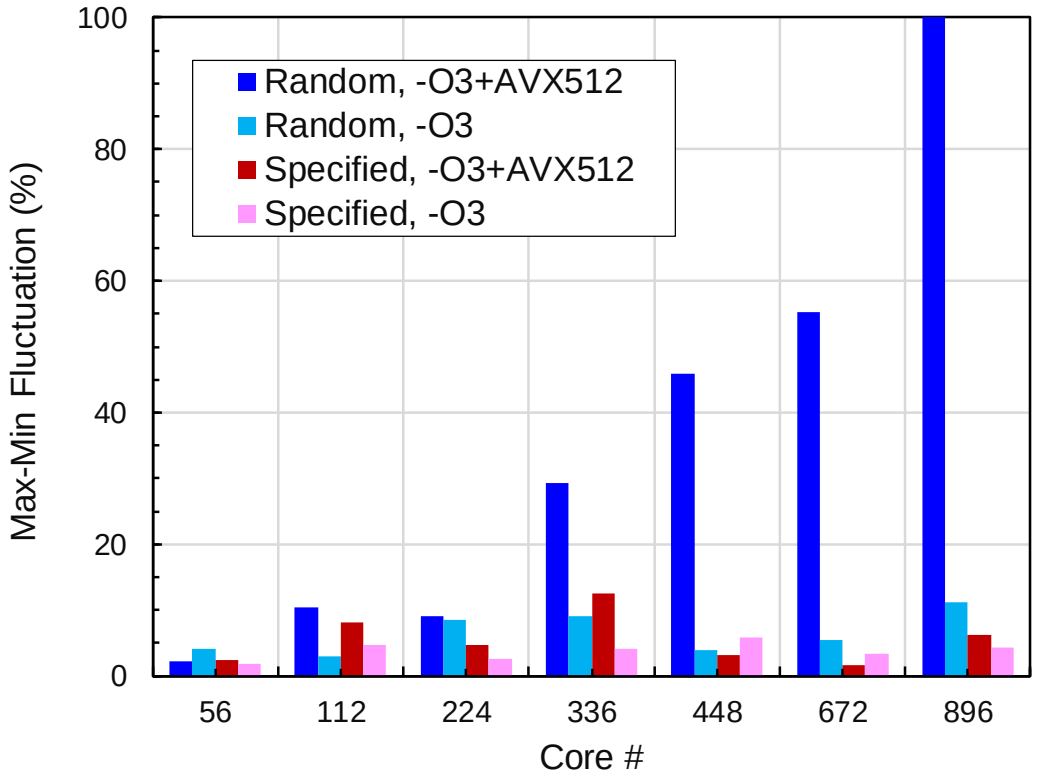


5回の計測における変動

N=10^6



N=10^6

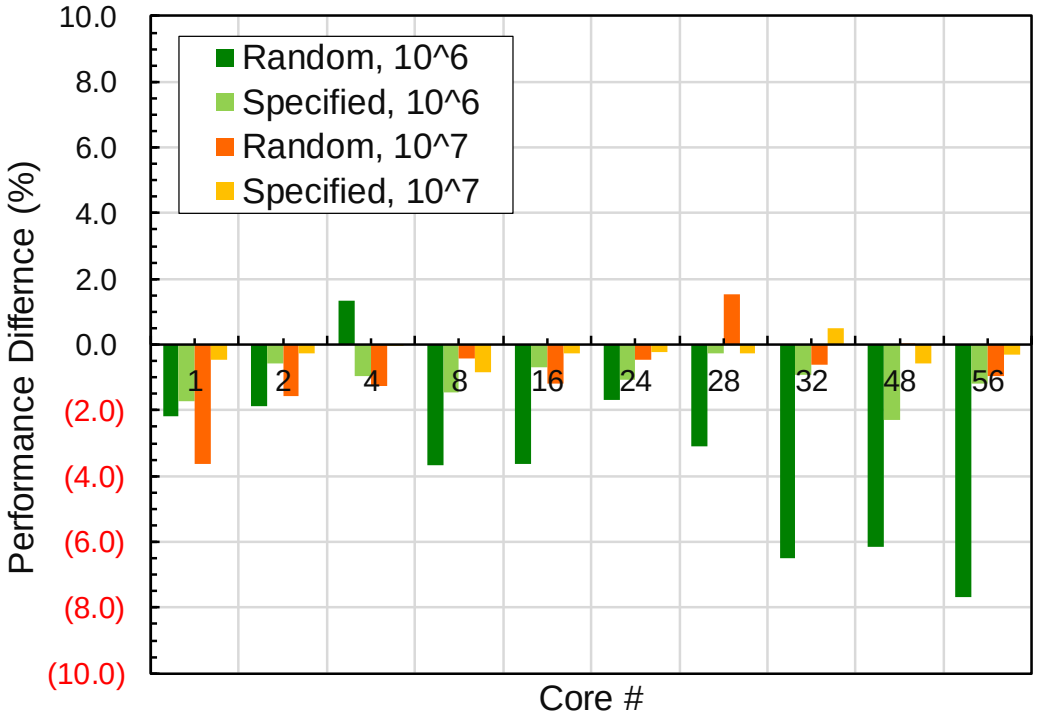


“-O3+AVX512” and “-O3”
Best Case of 5 Measurements

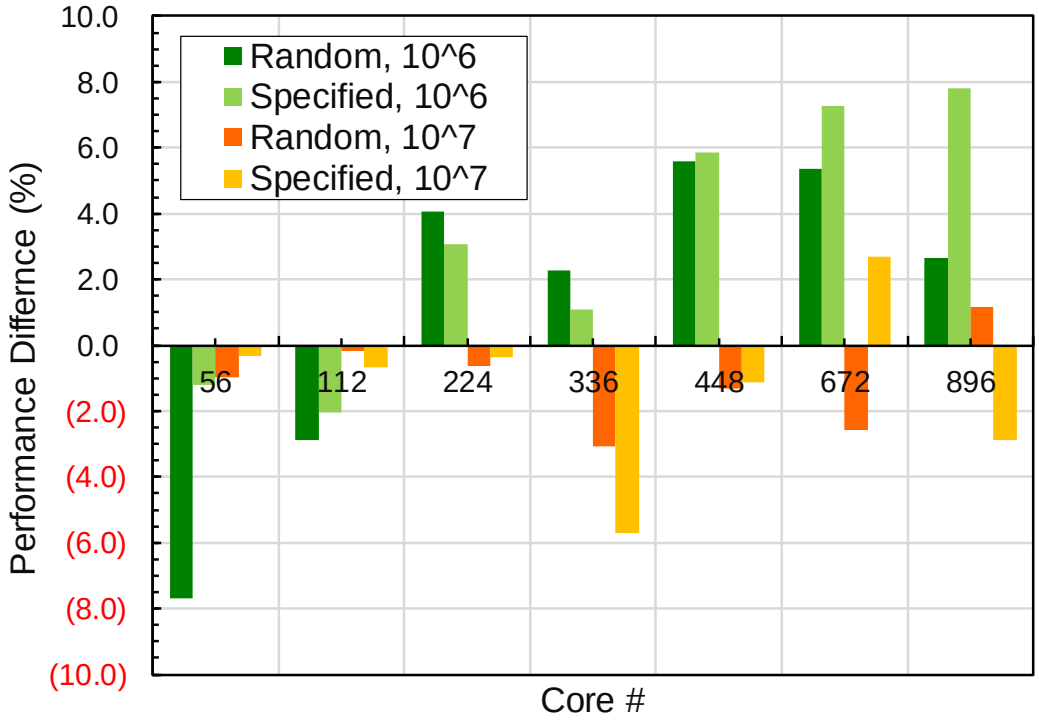
+: -O3 is better, -: -O3 is worse

“-O3” is rather faster in 10^6 cases with many cores

up to 1 node, 56cores



up to 16 nodes, 896 cores



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

もし各プロセスにおいて，各隣接プロセスに対応した外点の番号が連続していたら

	84	81	85	82	83	86	88	87	
96	57	58	59	60	61	62	63	64	73
95	49	50	51	52	53	54	55	56	74
94	41	42	43	44	45	46	47	48	80
93	33	34	35	36	37	38	39	40	79
92	25	26	27	28	29	30	31	32	78
91	17	18	19	20	21	22	23	24	77
90	9	10	11	12	13	14	15	16	76
89	1	2	3	4	5	6	7	8	75
	65	66	67	68	69	70	71	72	

[A]{p}= {q} (Original): 1d.c

```
StatSend = malloc(sizeof(MPI_Status) * NeibPETot);
StatRecv = malloc(sizeof(MPI_Status) * NeibPETot);
RequestSend = malloc(sizeof(MPI_Request) * NeibPETot);
RequestRecv = malloc(sizeof(MPI_Request) * NeibPETot);

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

[A]{p}= {q} (Mod.): No Copy for RECV: 1d2.c

```
StatSend = malloc(sizeof(MPI_Status) * 2 * NeibPETot);
RequestSend = malloc(sizeof(MPI_Request) * 2 * NeibPETot);

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(&P[ir+N], len_r, MPI_DOUBLE, NeibPE[neib],
            0, MPI_COMM_WORLD, &RequestSend[neib+NeibPETot]);
}

MPI_Waitall(2*NeibPETot, RequestSend, StatSend);
```

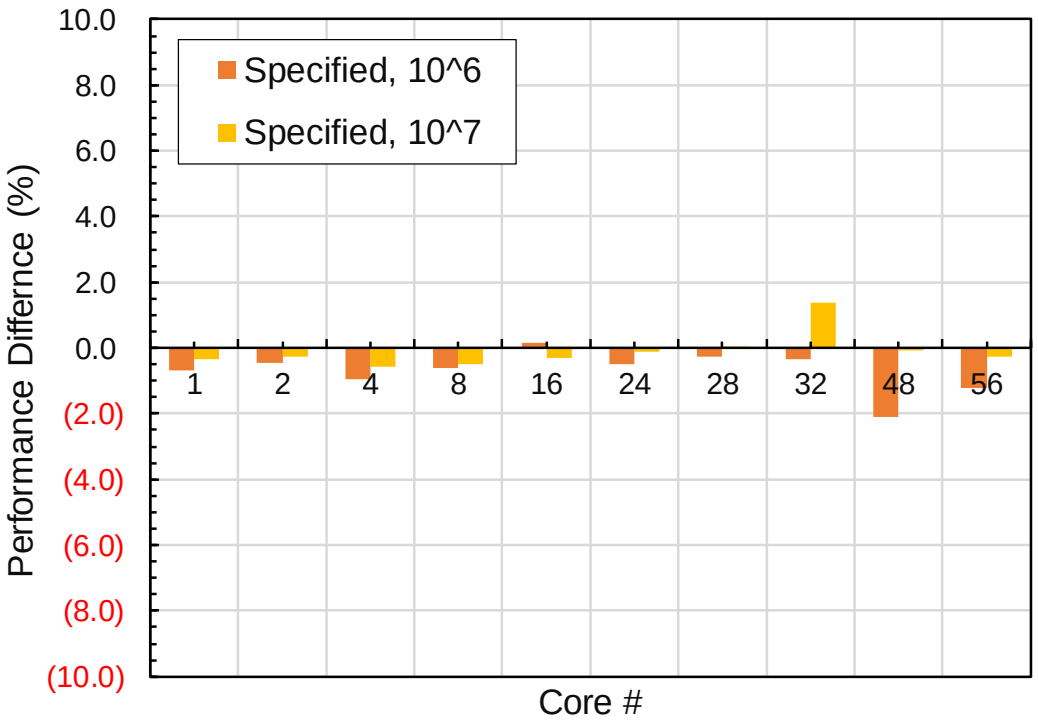
“Original” and “Modified” Best Case of 5 Measurements

+: “Modified” is better, -: “Modified” is worse

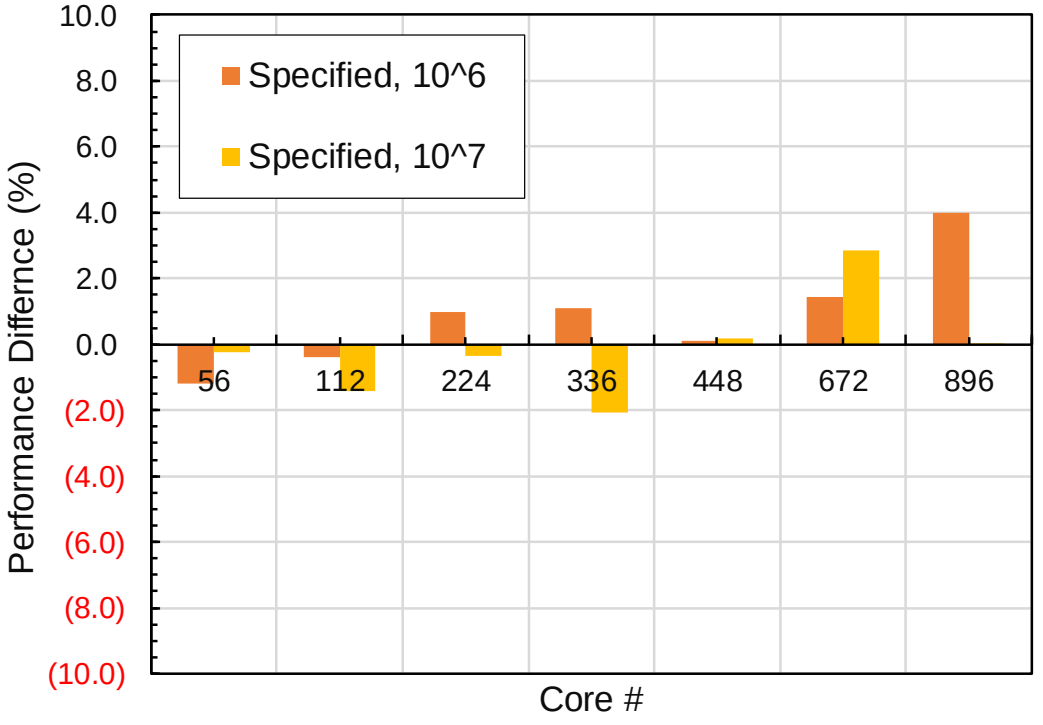
“Modified” is faster in 10^6 cases with many cores

But differences are small

up to 1 node, 56cores



up to 16 nodes, 896 cores



更なる検討

- NUMA controlの有無による比較

```
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./a.out
mpiexec.hydra -n ${PJM_MPI_PROC} ./a.out
```

- MPIプロセス数が増えても反復回数が変わらない理由