

並列有限要素法による 一次元定常熱伝導解析プログラム Fortran編

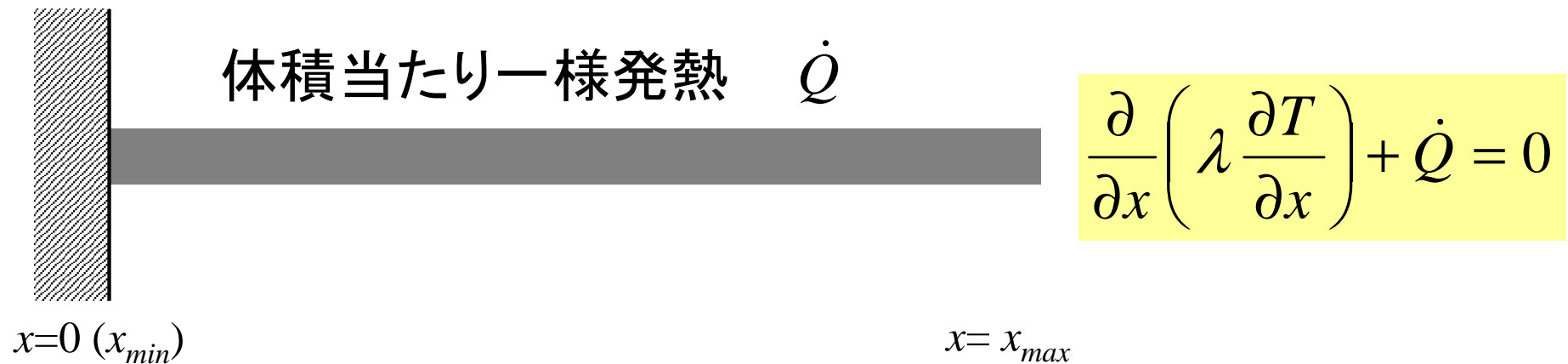
中島 研吾

東京大学情報基盤センター

お試しアカウント付き講習会「MPI応用編:並列有限要素法」

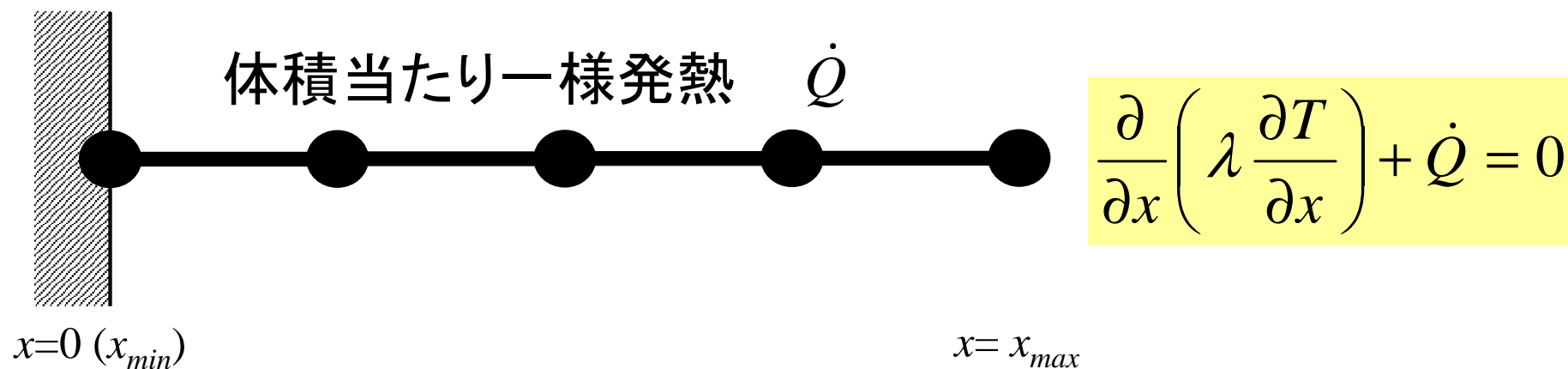
- 問題の概要, 実行方法
- プログラムの説明
- 計算例

対象とする問題：一次元熱伝導問題



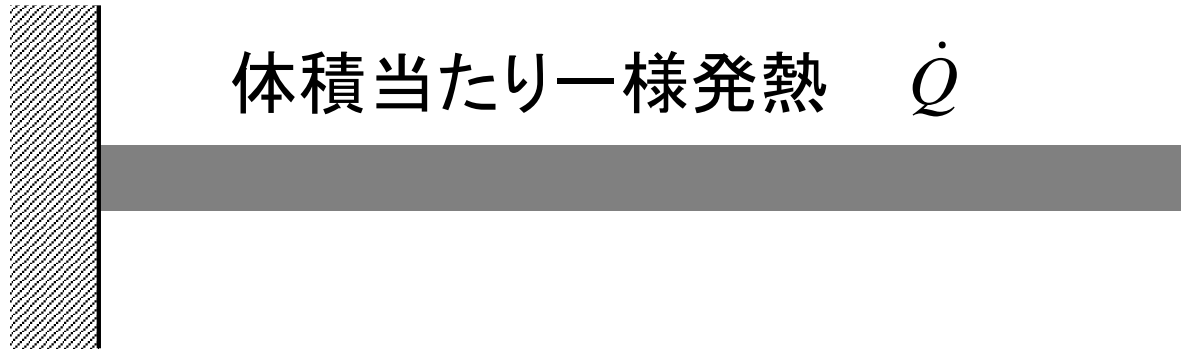
- 一様な：断面積 A ，熱伝導率 λ
- 体積当たり一様発熱（時間当たり） $[QL^{-3}T^{-1}]$ \dot{Q}
- 境界条件
 - $x=0$: $T=0$ （固定）
 - $x=x_{max}$: $\frac{\partial T}{\partial x} = 0$ (断熱)

対象とする問題：一次元熱伝導問題



- 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



体積当たり一様発熱 \dot{Q}

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

$x=0$ (x_{min})

$$T = 0 @ x = 0$$

$x = x_{max}$

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

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

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

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

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

ファイルコピー, コンパイル

Fortran

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

C

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

Confirm/Compile

```
>$ cd mpi/S2-ref  
>$ mpiifort -align array64byte -O3 -axCORE-AVX512 1d.f -o 1da  
>$ mpiifort -O3 1d.f -o 1db
```

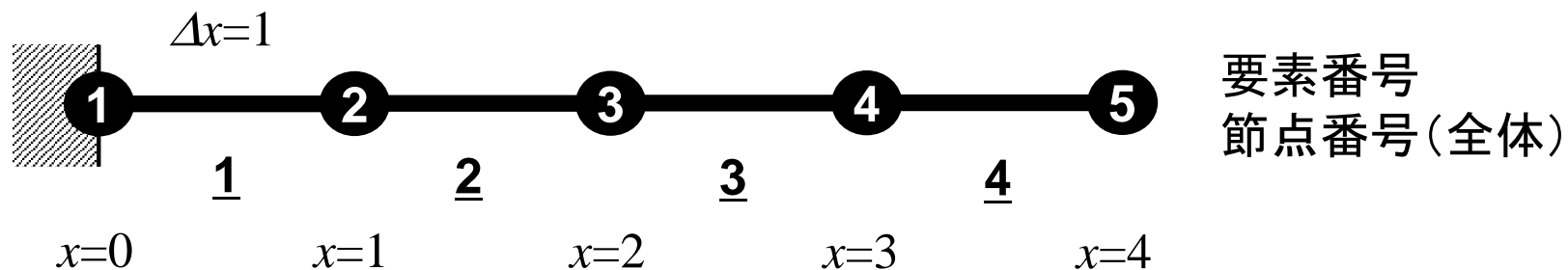
```
>$ mpiicc -align -O3 -axCORE-AVX512 1d.c -o 1da  
>$ mpiicc -O3 1d.c -o 1db
```

```
<$O-S2r> = <$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

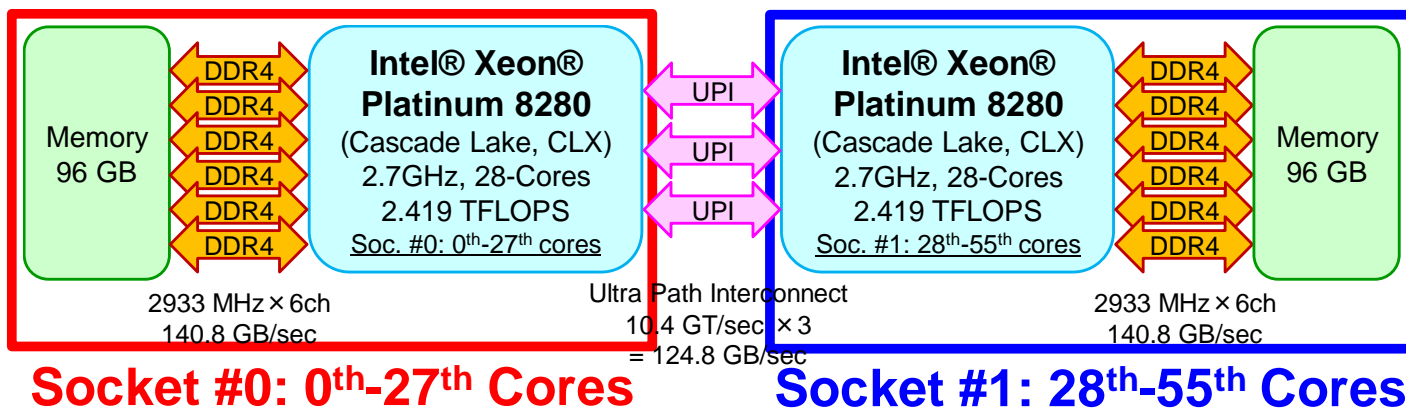
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=256
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst
```

```
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

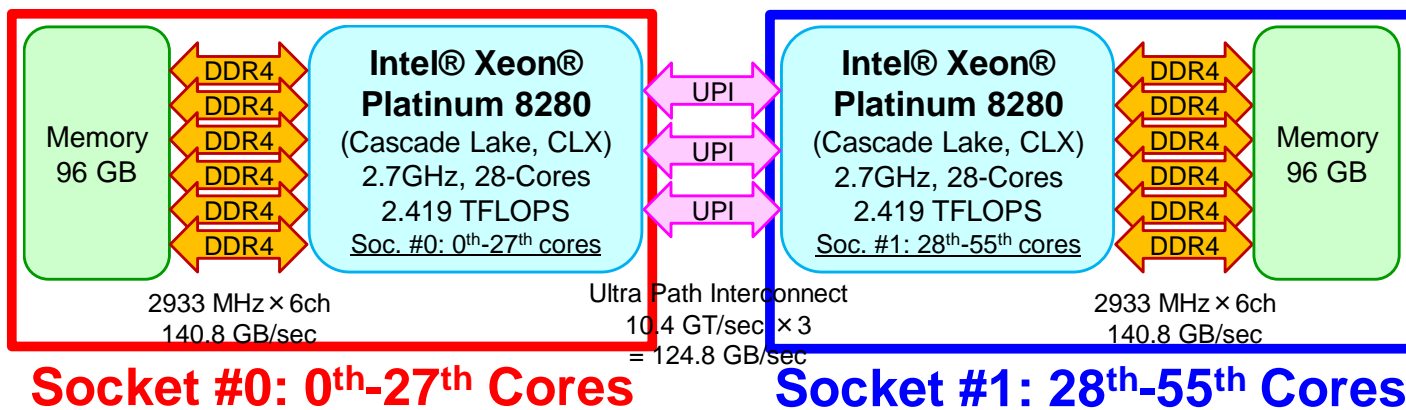
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=384
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst
```

```
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

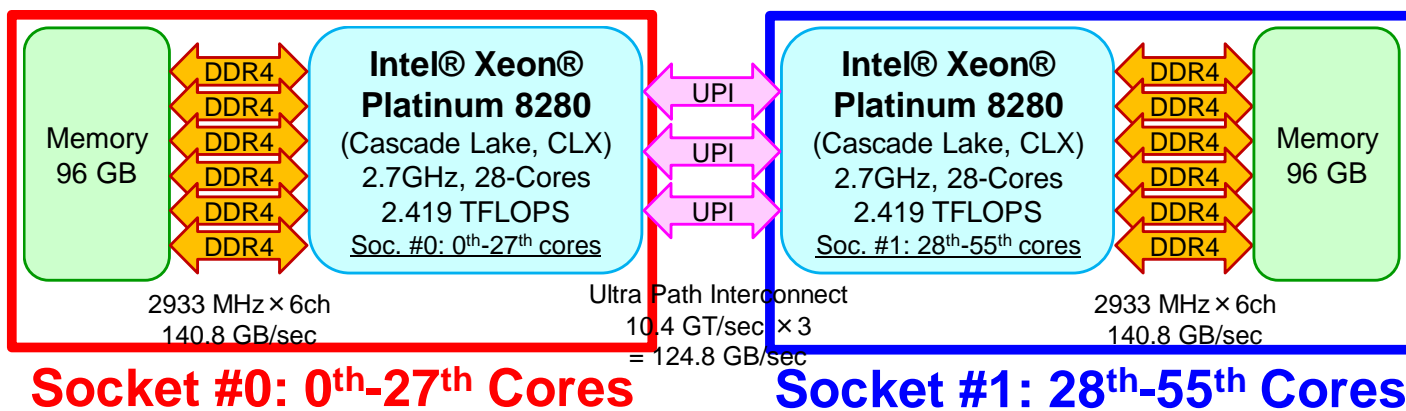


g28.sh: 8-nodes, 448-cores, 28x2x8

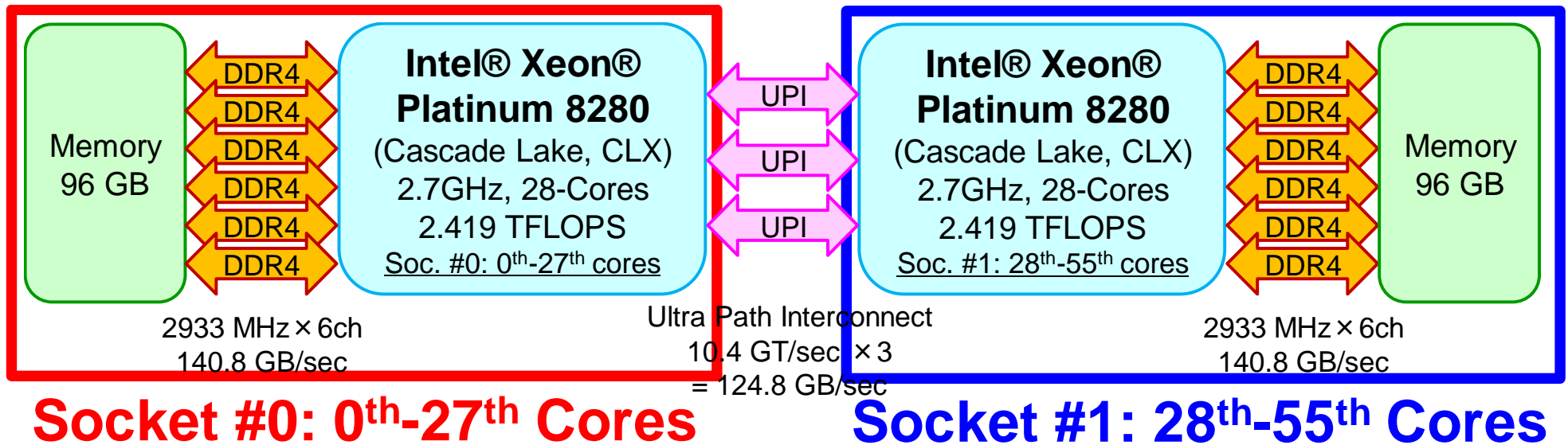
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=8
#PJM --mpi proc=448
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o test.lst
```

448/8= 56 cores/node

```
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
```



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

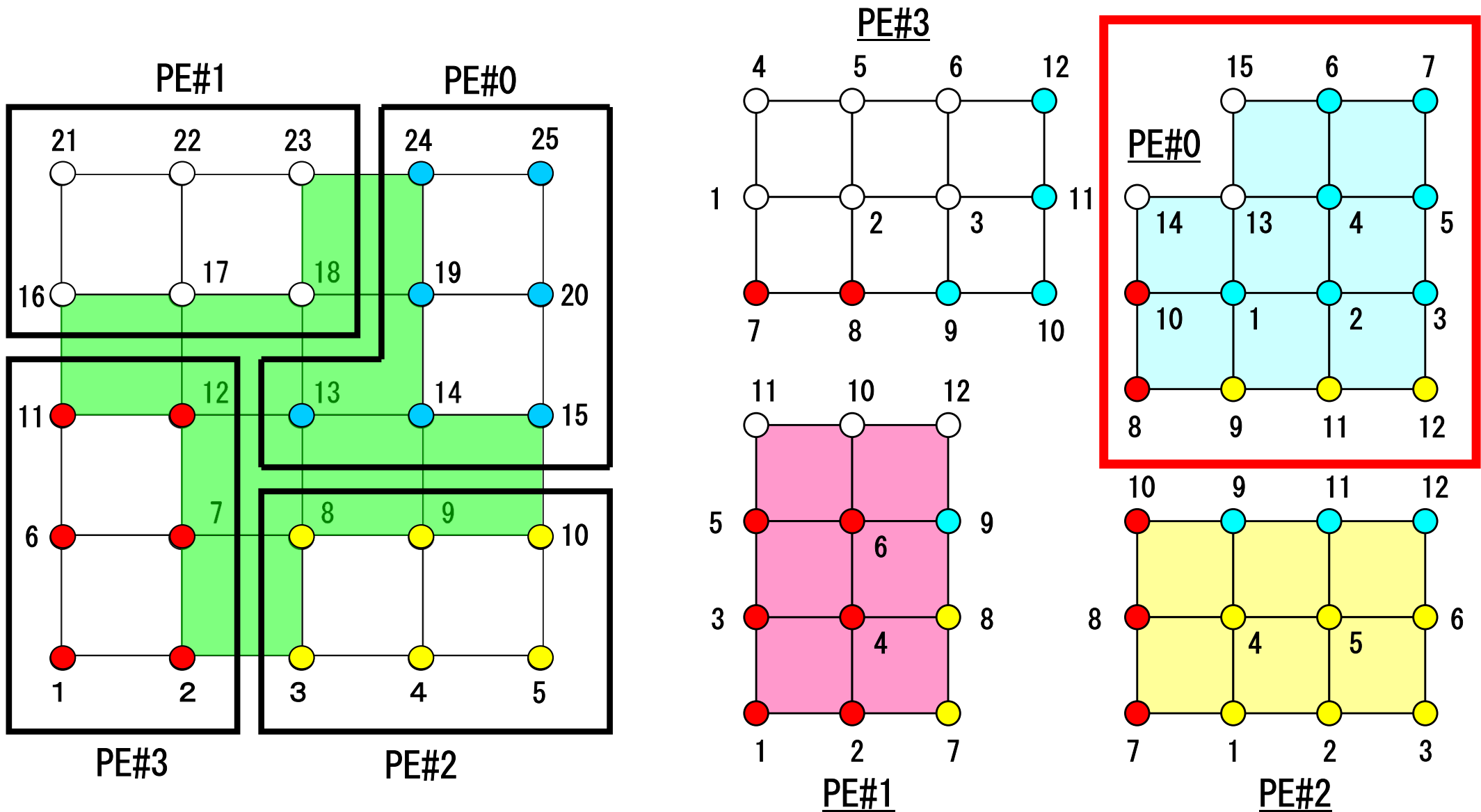
- 問題の概要, 実行方法
- プログラムの説明
- 計算例

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
 - 有限要素法の特徴 : 要素で閉じた計算

Node-based Partitioning

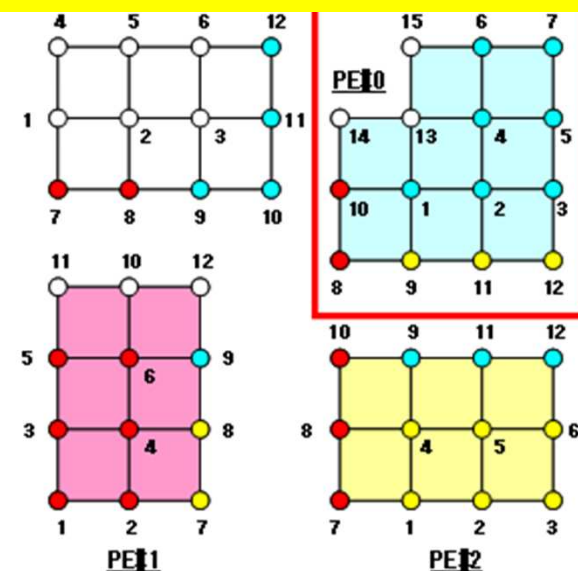
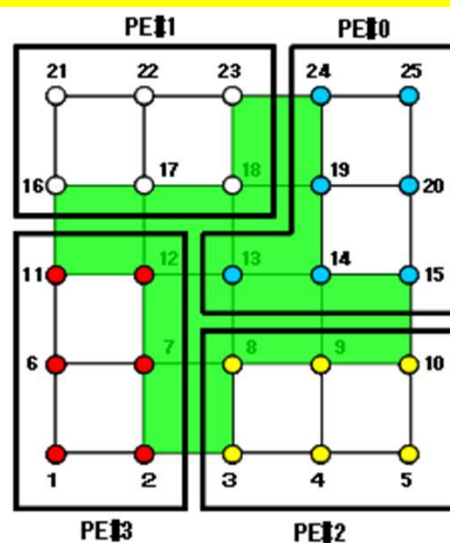
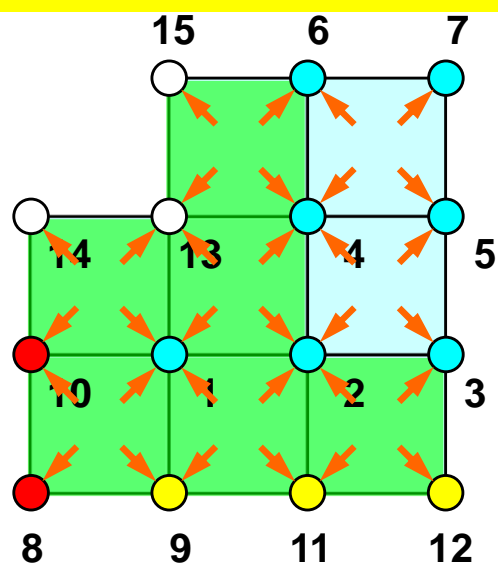
internal nodes - elements - external nodes



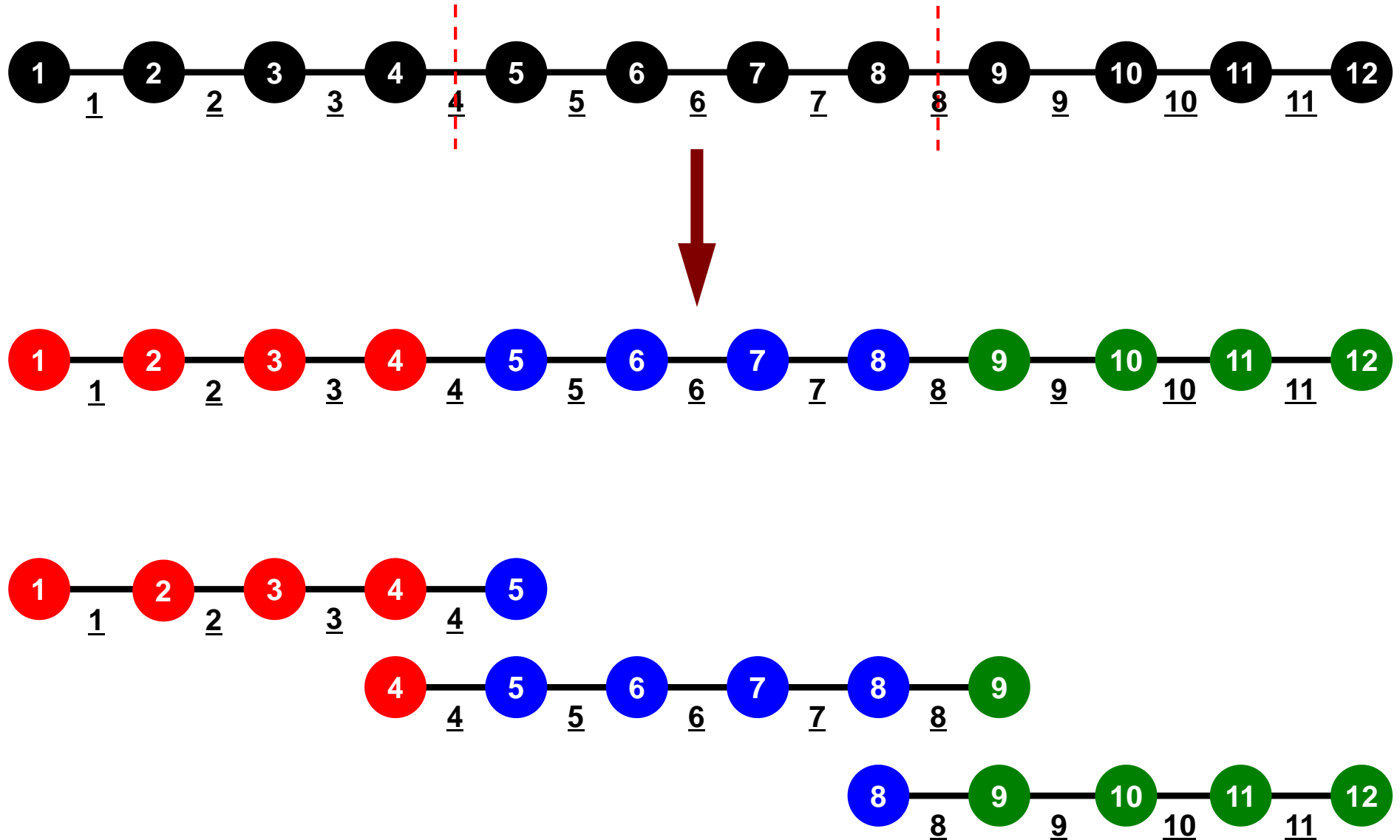
Node-based Partitioning

internal nodes - elements - external nodes

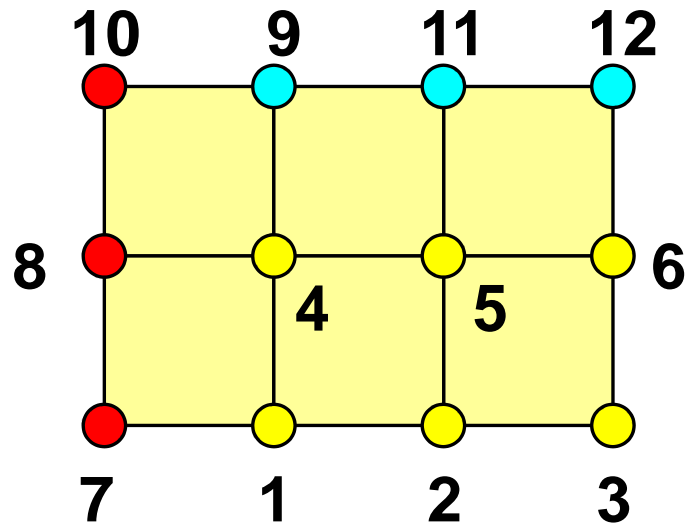
- Partitioned nodes themselves (Internal Nodes) 内点
- Elements which include Internal Nodes 内点を含む要素
- External Nodes included in the Elements 外点
in overlapped region among partitions.
- 外点の情報は各プロセスで独立に要素マトリクス生成を実施するのに必要⇒マトリクス生成時の通信は不要となる



一次元問題：11要素，12節点，3領域



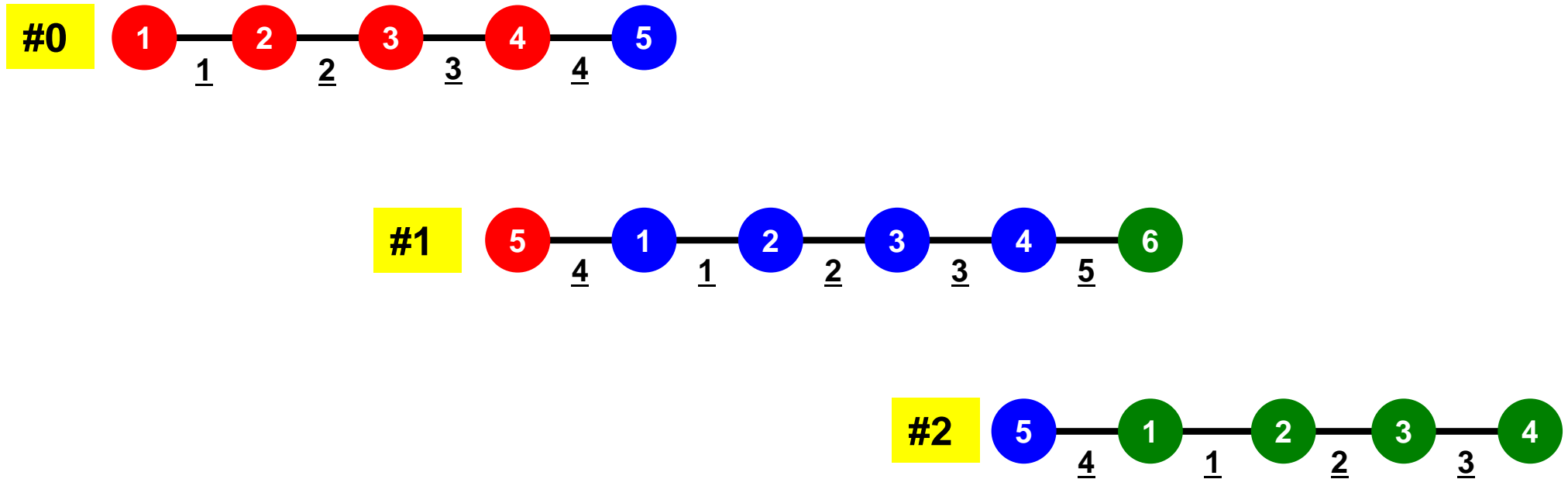
各領域データ（局所データ）仕様



- 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

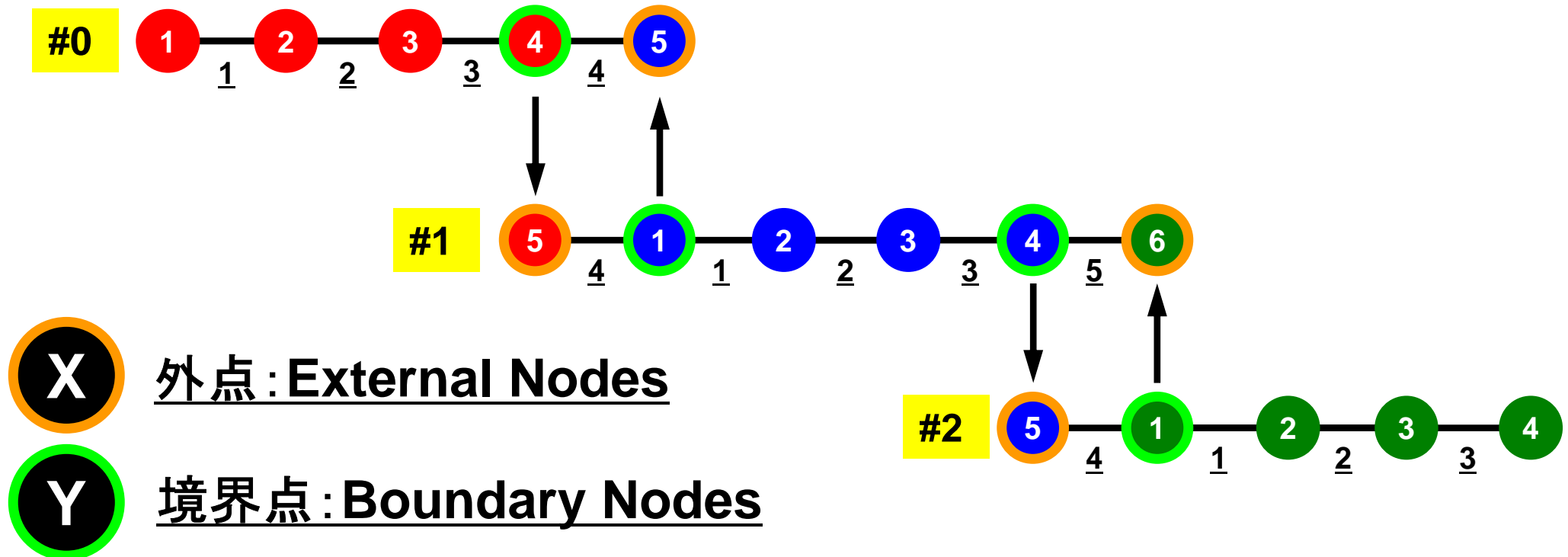
局所節点・要素番号は「1」からスタート



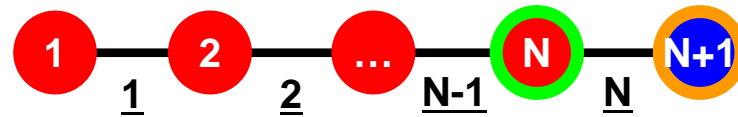
一次元問題：11要素，12節点，3領域

内点・外点・境界点

境界点は，他の領域の外点

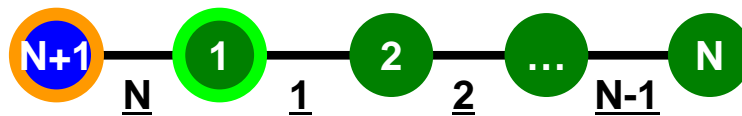


1D FEM: Numbering of Local ID



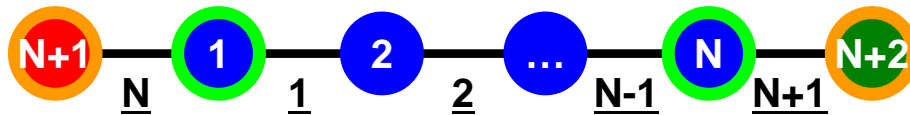
#0:

N+1 nodes
N elements



#PETot-1:

N+1 nodes
N elements

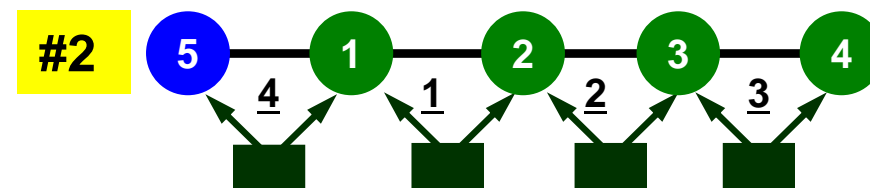
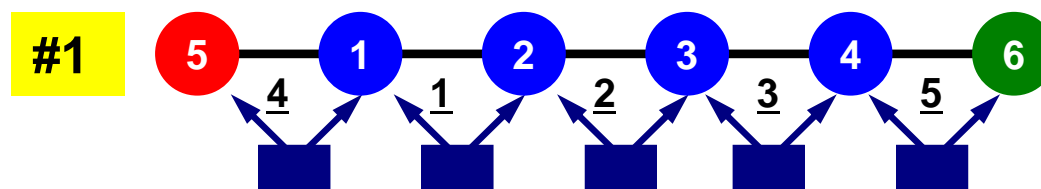
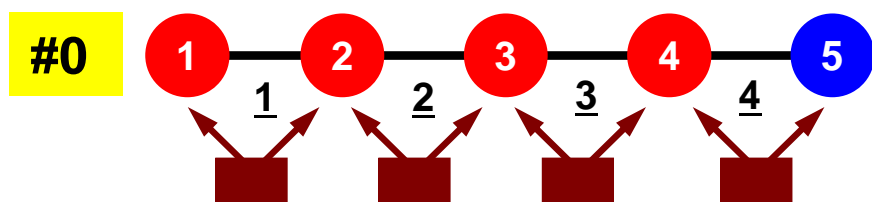


Others (General):

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

```

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

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

前処理, ベクトル定数倍の加減

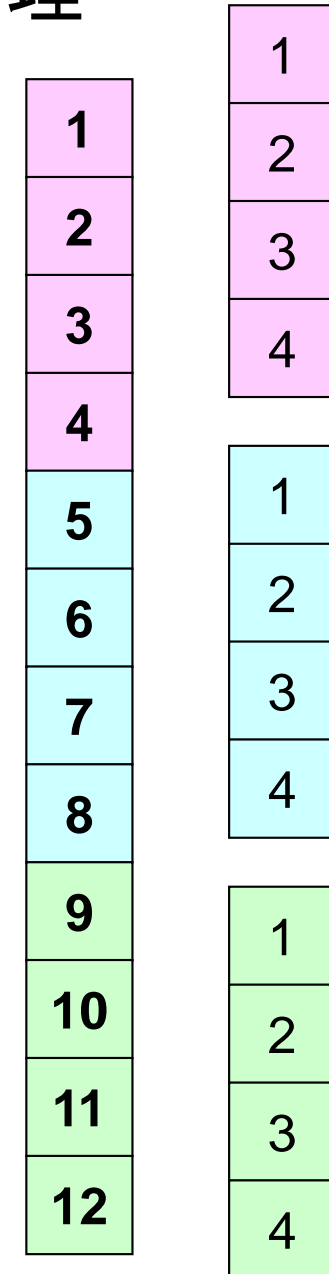
局所的な計算 (内点のみ) が可能 \Rightarrow 並列処理

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

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

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

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

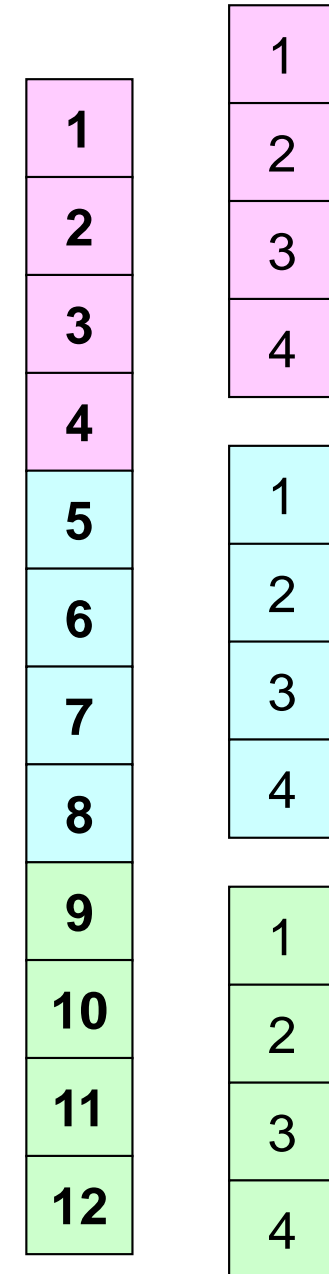


内積

全体で和をとる必要がある⇒集団通信

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

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



行列ベクトル積

外点の値が必要⇒1対1通信

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

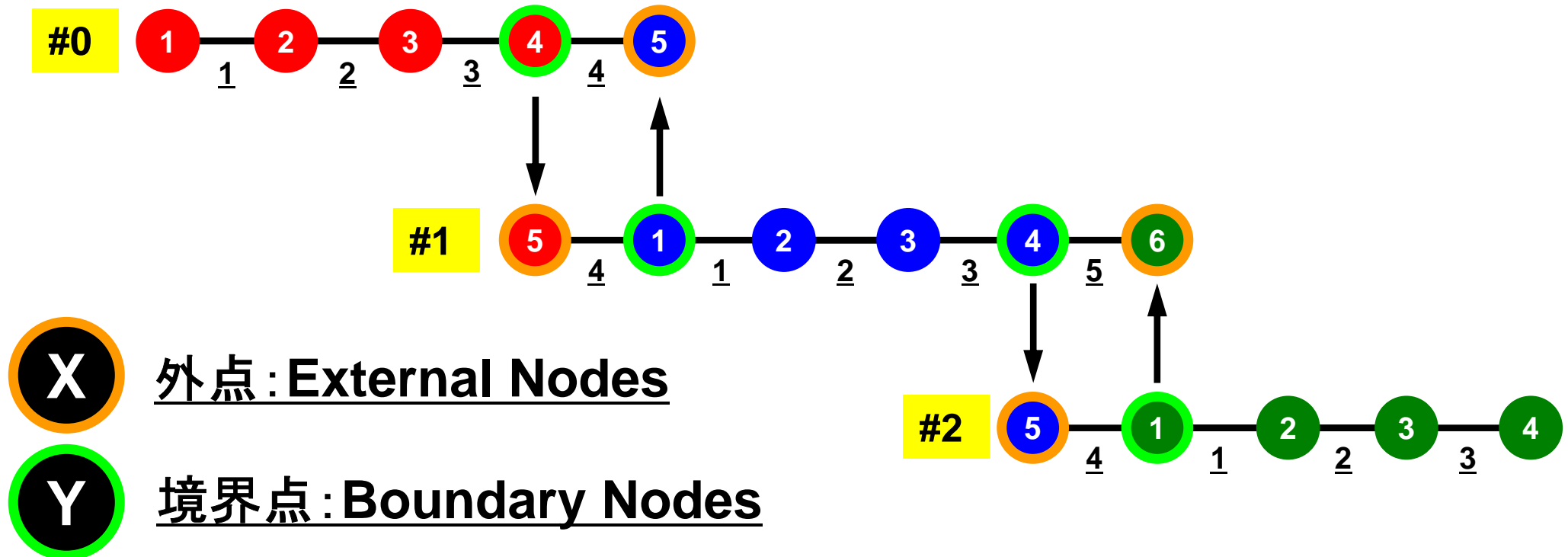
do i= 1, N
  W(i, Q) = DIAG(i)*W(i, P)
  do j= INDEX(i-1)+1, INDEX(i)
    W(i, Q) = W(i, Q) + AMAT(j)*W(ITEM(j), P)
  enddo
enddo
```



一次元問題：11要素，12節点，3領域

内点・外点・境界点

境界点は，他の領域の外点



行列ベクトル積：ローカルに計算実施可能

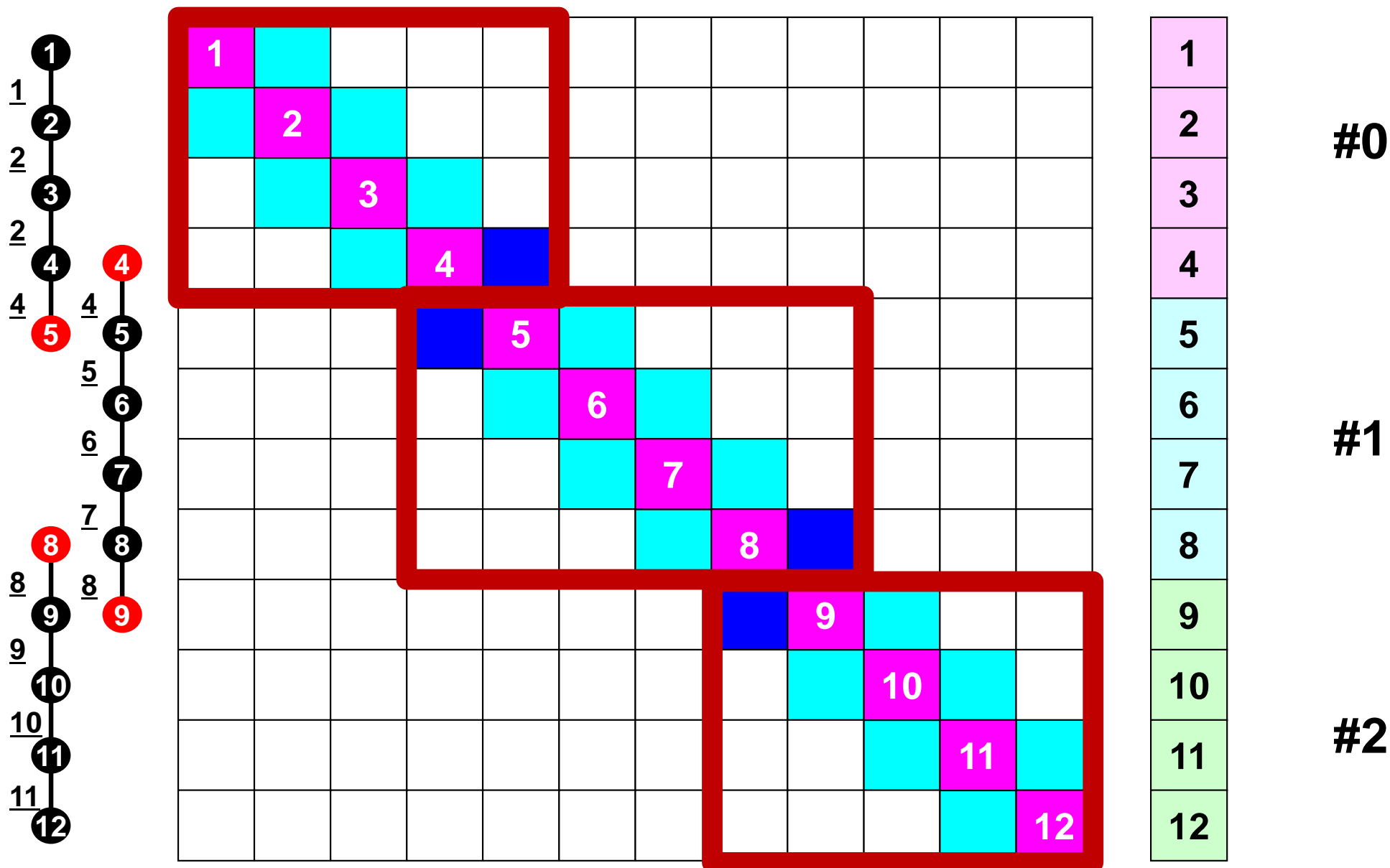
1											
	2										
		3									
			4								
				5							
					6						
						7					
							7				
								9			
									10		
										11	
											12

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

=

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

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

1										
	2									
		3								
			4							

1
2
3
4

1
2
3
4

				5						
					6					
						7				
							8			

5
6
7
8

=

5
6
7
8

								9			
									10		
										11	
											12

9
10
11
12

9
10
11
12

行列ベクトル積：ローカルに計算実施可能

1				
	2			
		3		
			4	

1
2
3
4

1
2
3
4

	5			
		6		
			7	
				8

5
6
7
8

=

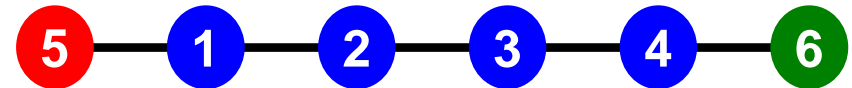
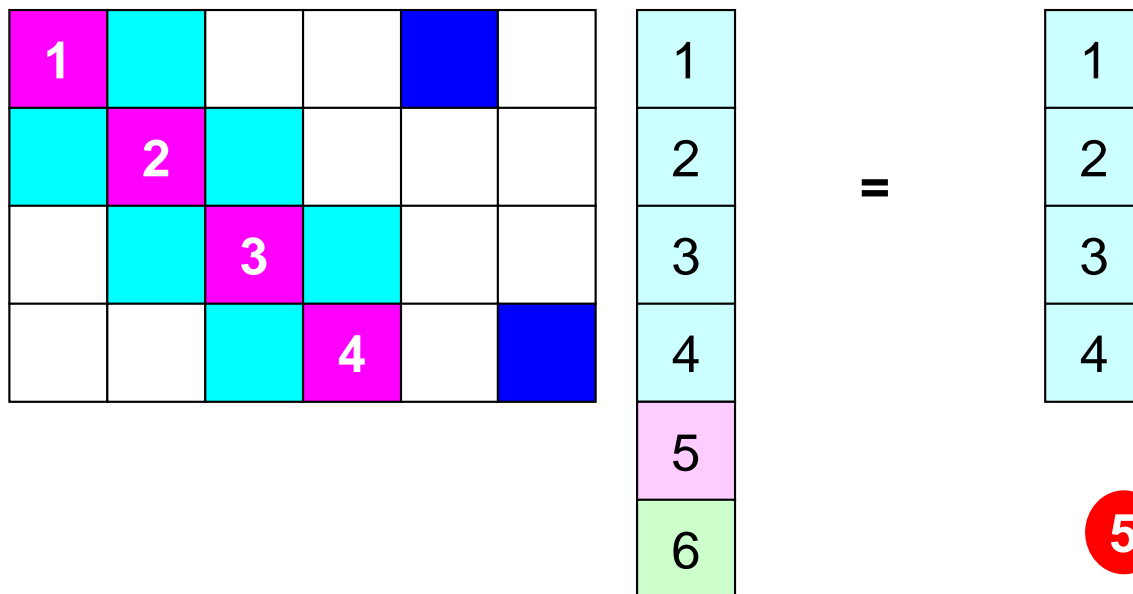
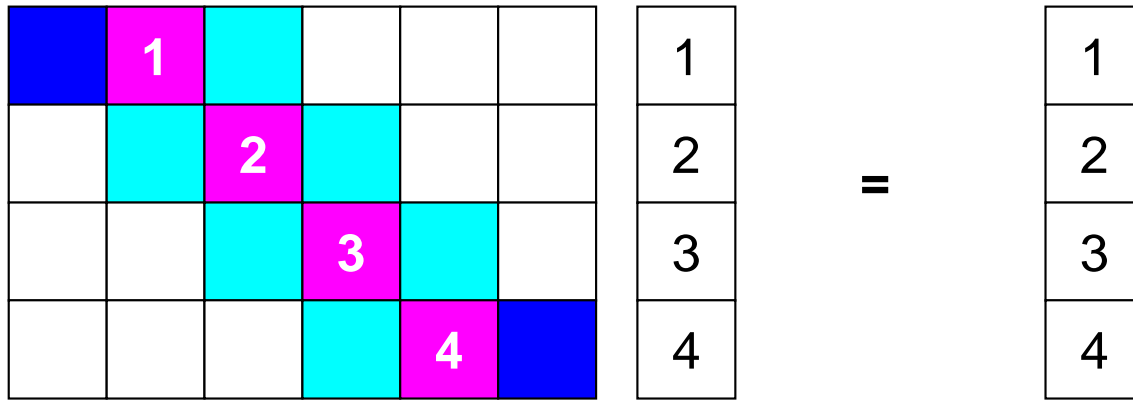
5
6
7
8

	9			
		10		
			11	
				12

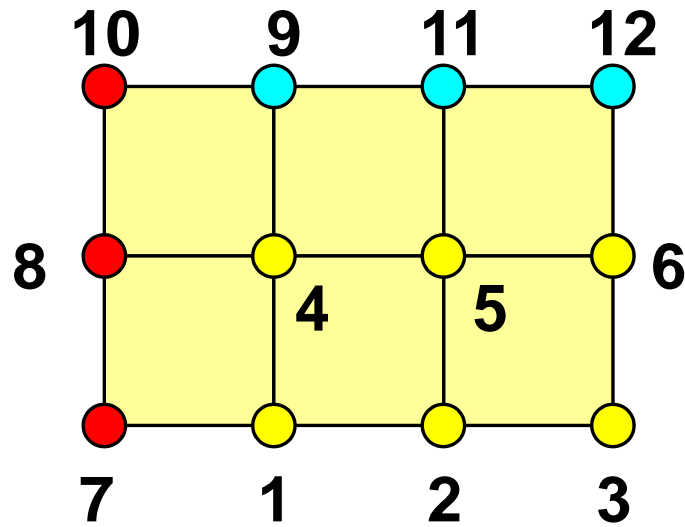
9
10
11
12

9
10
11
12

行列ベクトル積：ローカル計算 #1



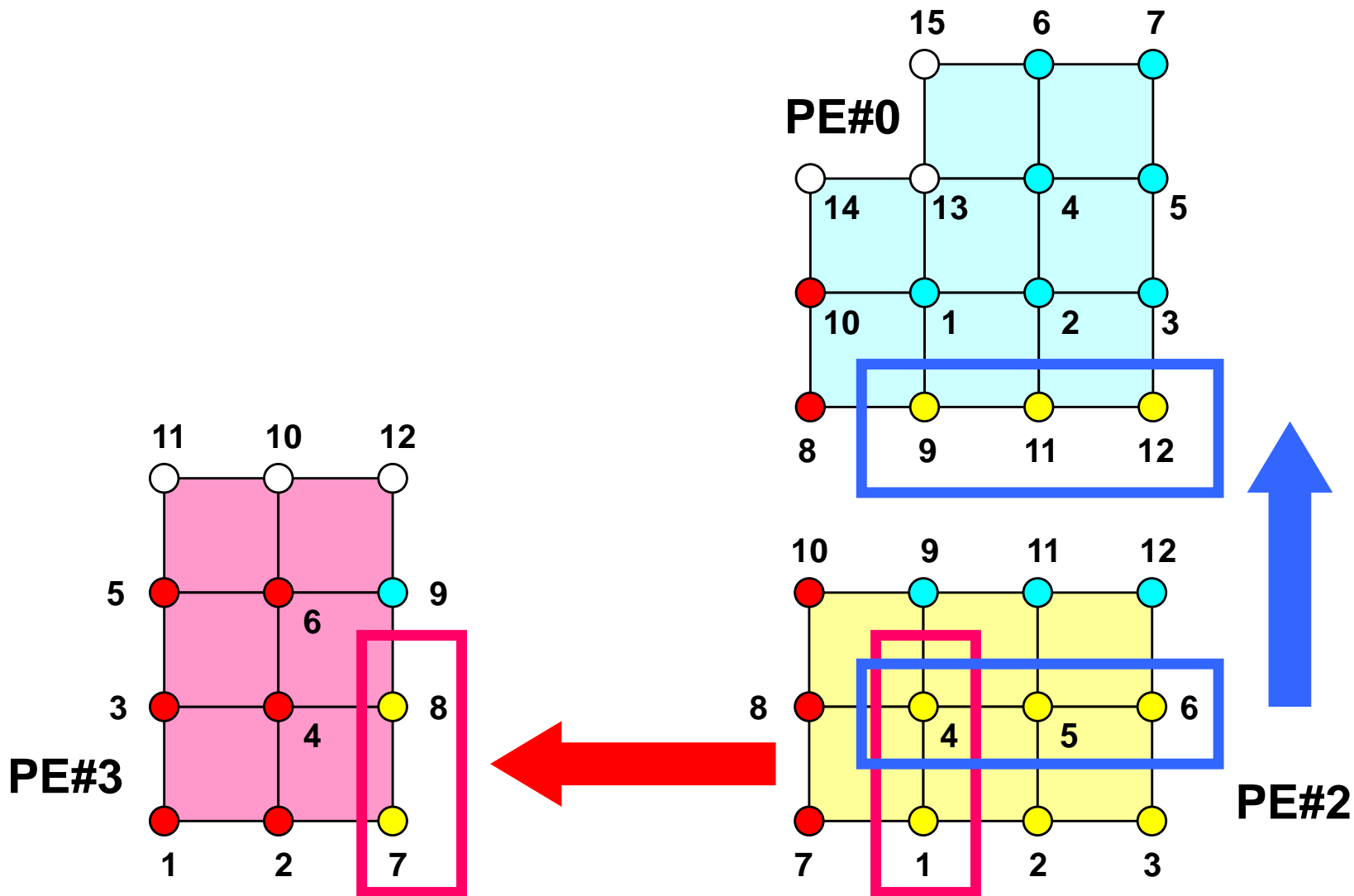
各領域データ（局所データ）仕様



- 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

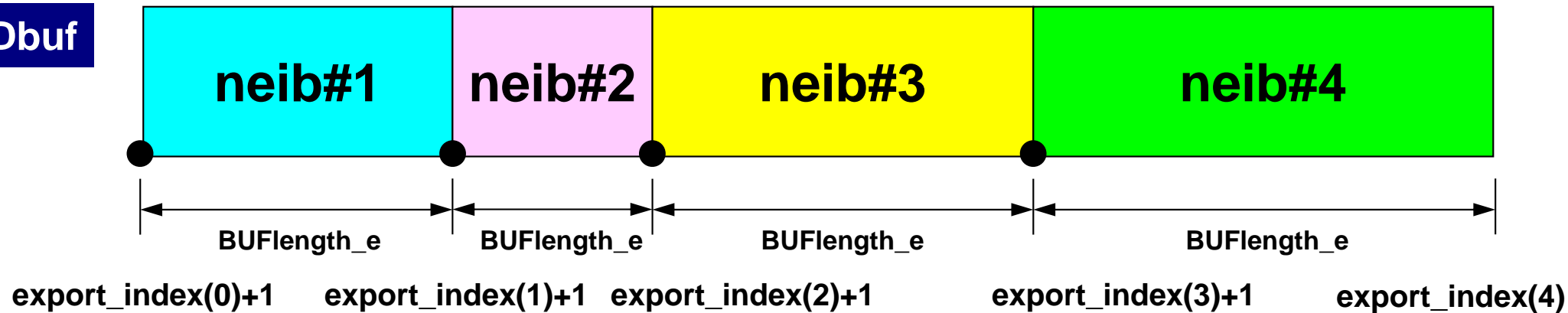
Boundary Nodes (境界点) : SEND

PE#2 : 境界点 (Boundary Nodes) の値を送信



SEND: MPI_Isend/Irecv/Waitall

SENDbuf



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

Copied to sending buffers

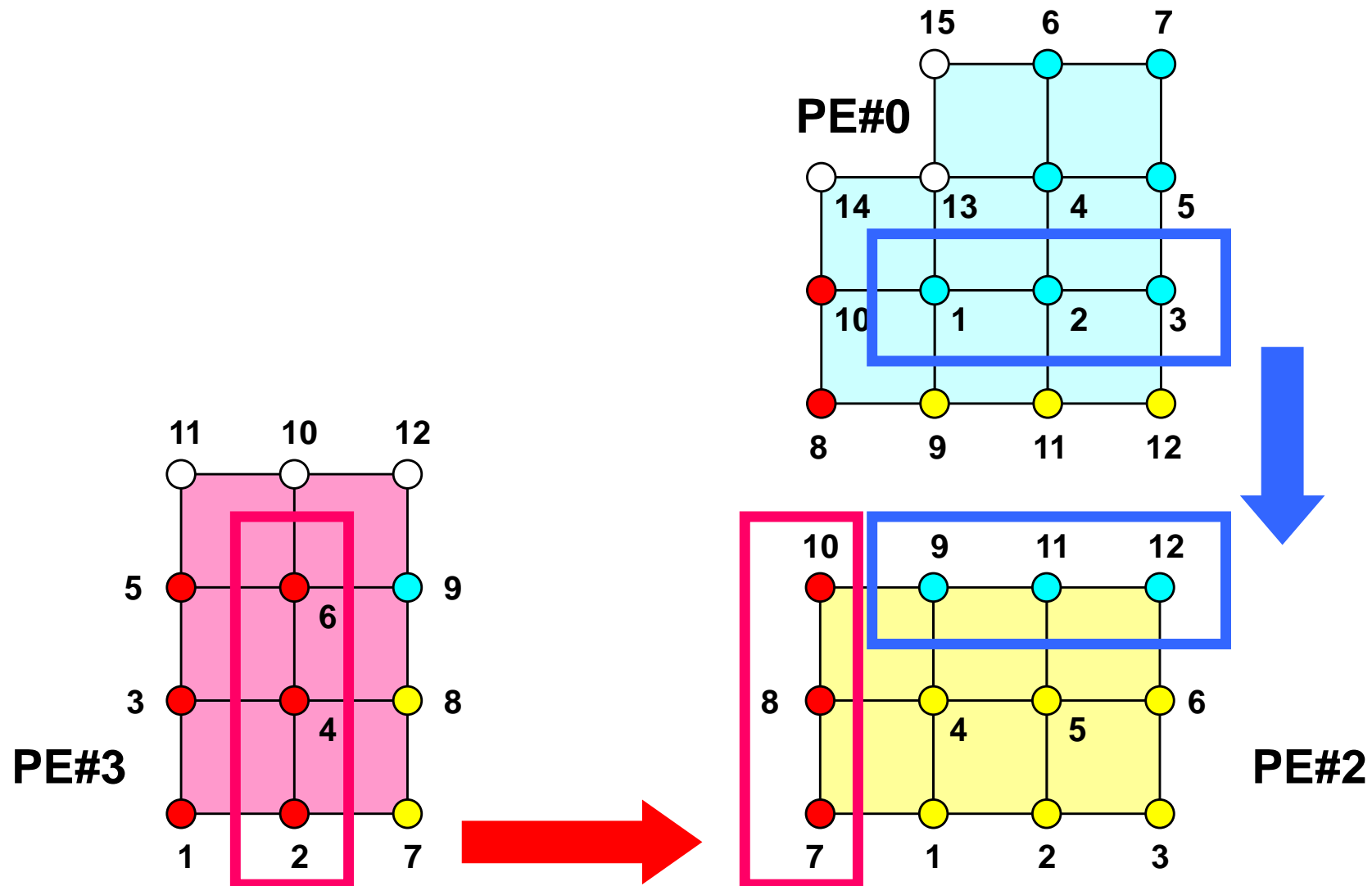
```
do neib= 1, NEIBPETOT
  iS_e= export_index(neib-1) + 1
  iE_e= export_index(neib )
  BUFlength_e= iE_e + 1 - iS_e

  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)
```

External Nodes (外点) : RECEIVE

PE#2 : 外点 (External Nodes) の値を受信



RECV: MPI_Isend/Irecv/Waitall

```

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

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

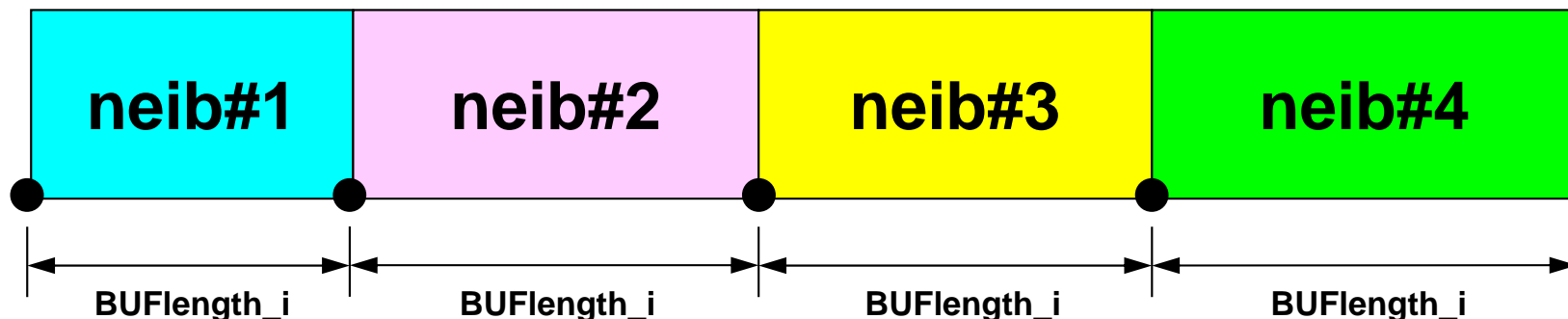
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

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

```

Copied from receiving buffer

RECVbuf



import_index(0)+1 import_index(1)+1 import_index(2)+1 import_index(3)+1 import_index(4)

プログラム: 1d.f(1/11)

諸変数

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

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

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

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

integer, dimension(:), allocatable :: ICELNOD
integer, dimension(:), allocatable :: INDEX, ITEM
integer(kind=4) :: NEIBPETOT, BUFlength, PETOT
integer(kind=4), dimension(2) :: NEIBPE

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

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

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

プログラム: 1d.f(2/11)

制御データ読み込み

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

```
call MPI_Init      (ierr)
call MPI_Comm_size (MPI_COMM_WORLD, PETOT, ierr)
call MPI_Comm_rank (MPI_COMM_WORLD, my_rank, ierr)
```

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

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

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

プログラム: 1d.f(2/11)

制御データ読み込み

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

```
call MPI_Init      (ierr)
call MPI_Comm_size (MPI_COMM_WORLD, PETOT, ierr)
call MPI_Comm_rank (MPI_COMM_WORLD, my_rank, ierr)
```

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

```
!C
!C-- CTRL data
```

```
if (my_rank.eq.0) then
  open  (11, file='input.dat', status='unknown')
  read  (11,*) Neg
  read  (11,*) dX, QV, AREA, COND
  read  (11,*) ITERmax
  read  (11,*) EPS
  close (11)
endif
```

my_rank=0 のとき制御データを読み込む

Neg: 「全」要素数

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


プログラム: 1d.f(2/11)

制御データ読み込み

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

```
call MPI_Init      (ierr)
call MPI_Comm_size (MPI_COMM_WORLD, PETOT, ierr)
call MPI_Comm_rank (MPI_COMM_WORLD, my_rank, ierr)
```

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

```
!C
!C-- CTRL data
```

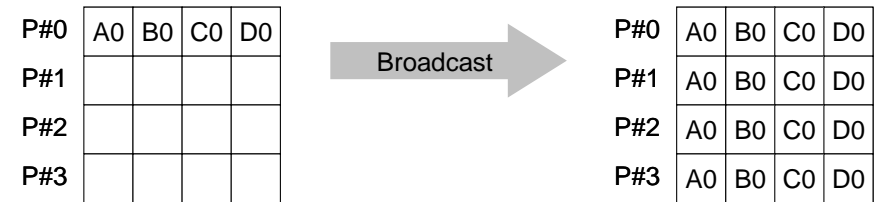
```
if (my_rank.eq.0) then
  open  (11, file='input.dat', status='unknown')
  read  (11,*) Neg
  read  (11,*) dX, QV, AREA, COND
  read  (11,*) ITERmax
  read  (11,*) EPS
  close (11)
endif
```

my_rank=0 のとき制御データを読み込む

Neg: 「全」要素数

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

MPI_BCAST



- コミュニケータ「comm」内の一つの送信元プロセス「root」のバッファ「buffer」から、その他全てのプロセスのバッファ「buffer」にメッセージを送信。
- **call MPI_BCAST (buffer, count, datatype, root, comm, ierr)**
 - **buffer** 任意 I/O バッファの先頭アドレス,
タイプは「datatype」により決定
 - **count** 整数 I メッセージのサイズ
 - **datatype** 整数 I メッセージのデータタイプ
FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
 - **root** 整数 I 送信元プロセスのID(ランク)
 - **comm** 整数 I コミュニケータを指定する
 - **ierr** 整数 0 完了コード

プログラム: 1d.f(3/11)

局所分散メッシュデータ

```

!C
!C-- Local Mesh Size

Ng= NEg + 1
N = Ng / PETOT

nr = Ng - N*PETOT
if (my_rank. lt. nr) N= N+1

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

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

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

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

!C
!C- ARRAYs

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

```

総節点数
局所節点数

NgがPETOTで割り切れない場合

プログラム: 1d.f(3/11)

局所分散メッシュデータ, 各要素→一様

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

```
Ng= NEg + 1
N = Ng / PETOT
```

総節点数
局所節点数

```
nr = Ng - N*PETOT
if (my_rank.lt.nr) N= N+1
```

NgがPETOTで割り切れない場合

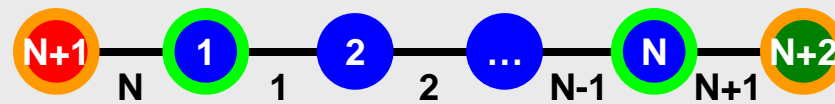
```
NE= N - 1 + 2
NP= N + 2
```

NE : 局所要素数
NP : 局所節点数 (内点+外点)

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

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

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



一般の領域:
N+2節点, N+1要素

```
!C
!C- ARRAYs
```

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

プログラム: 1d.f(3/11)

局所分散メッシュデータ, 各要素→一様

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

```
Ng= NEg + 1
N = Ng / PETOT
```

総節点数
局所節点数

```
nr = Ng - N*PETOT
if (my_rank.lt.nr) N= N+1
```

NgがPETOTで割り切れない場合

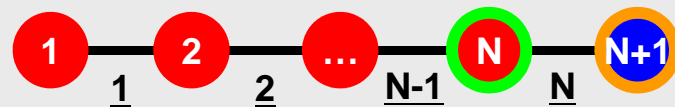
```
NE= N - 1 + 2
NP= N + 2
```

NE : 局所要素数
NP : 局所節点数 (内点+外点)

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

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

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



#0: N+1節点, N要素

```
!C
!C- ARRAYs
```

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

プログラム: 1d.f(3/11)

局所分散メッシュデータ, 各要素→一様

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

```
Ng= NEg + 1
N = Ng / PETOT
```

総節点数
局所節点数

```
nr = Ng - N*PETOT
if (my_rank.lt.nr) N= N+1
```

NgがPETOTで割り切れない場合

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

NE : 局所要素数
NP : 局所節点数 (内点+外点)

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

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

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



#PETot-1: N+1節点, N要素

```
!C
!C- ARRAYs
```

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

プログラム: 1d.f(3/11)

局所分散メッシュデータ

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

```
Ng= NEg + 1
N = Ng / PETOT
```

総節点数
局所節点数

```
nr = Ng - N*PETOT
if (my_rank. lt. nr) N= N+1
```

NgがPETOTで割り切れない場合

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

NE : 局所要素数
NP : 局所節点数 (内点+外点)

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

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

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

```
!C
!C- ARRAYs
```

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

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.f(4/11)

配列初期化, 要素~節点

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

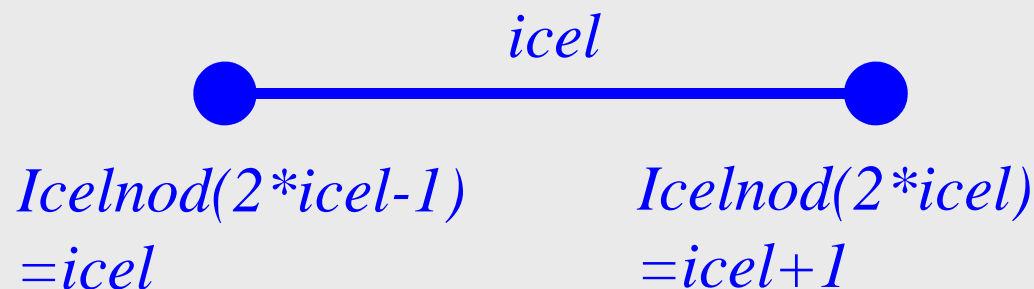
```
if (PETOT.gt.1) then
```

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

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

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

```
endif
endif
```



プログラム: 1d.f(4/11)

配列初期化, 要素~節点

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

```
if (PETOT.gt.1) then
```

「1-2」の要素を「1」とする

```
if (my_rank.eq.0) then
```

```
  icel= NE
```

```
  ICELNOD(2*icel-1)= N
```

```
  ICELNOD(2*icel )= N + 1
```



#0: N+1節点, N要素

```
else if (my_rank.eq.PETOT-1) then
```

```
  icel= NE
```

```
  ICELNOD(2*icel-1)= N + 1
```

```
  ICELNOD(2*icel )= 1
```



#PETot-1: N+1節点, N要素

```
else
```

```
  icel= NE - 1
```

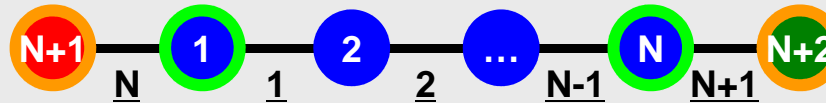
```
  ICELNOD(2*icel-1)= N + 1
```

```
  ICELNOD(2*icel )= 1
```

```
  icel= NE
```

```
  ICELNOD(2*icel-1)= N
```

```
  ICELNOD(2*icel )= N + 2
```



一般の領域:
N+2節点, N+1要素

```
endif
endif
```

プログラム: 1d.f (5/11)

Index定義

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

```
!C===
```

```
!C
!C +-----+
!C | CONNECTIVITY |
!C +-----+
!C
```

```
INDEX = 2
```

```
INDEX(0) = 0
```

```
INDEX(N+1) = 1
```

```
INDEX(NP) = 1
```

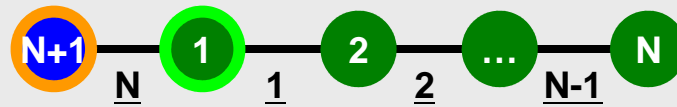
```
if (my_rank.eq.0) INDEX(1) = 1
if (my_rank.eq.PETOT-1) INDEX(N) = 1
```

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

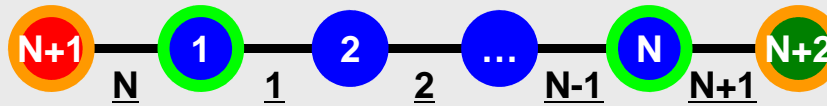
```
NPLU = INDEX(NP)
ITEM = 0
```



#0: N+1 節点, N 要素



#PETot-1: N+1 節点, N 要素



一般の領域:
N+2 節点, N+1 要素

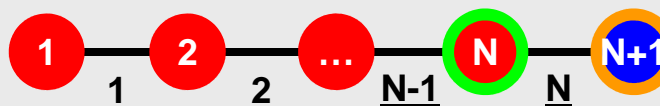
プログラム: 1d.f(6/11)

Item定義

```

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

```

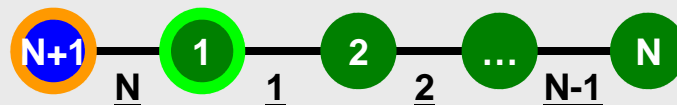


#0: N+1節点, N要素

```

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

```

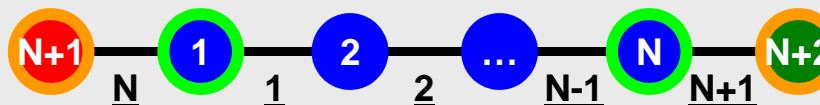


#PETot-1: N+1節点, N要素

```

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

```



一般の領域:
N+2節点, N+1要素

プログラム: 1d.f(7/11)

通信テーブル定義

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

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

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

```
BUFlength= 1
```

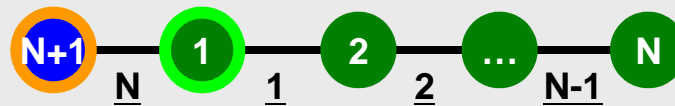
```
import_index(1)= 1
import_index(2)= 2
import_item  (1)= N+1
import_item  (2)= N+2
```

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

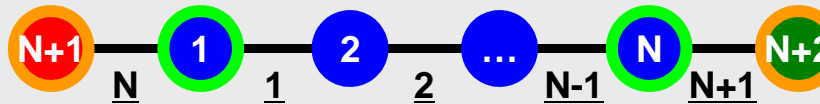
```
if (my_rank.eq.0) then
  import_item (1)= N+1
  export_item (1)= N
endif
```



#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

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

MPI_ISEND

- 送信バッファ「sendbuf」内の、連続した「count」個の送信メッセージを、タグ「tag」を付けて、コミュニケータ内の、「dest」に送信する。「MPI_WAITALL」を呼ぶまで、送信バッファの内容を更新してはならない。

- call MPI_ISEND

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

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

MPI_IRecv

- 受信バッファ「recvbuf」内の、連続した「count」個の送信メッセージを、タグ「tag」を付けて、コミュニケータ内の、「dest」から受信する。「MPI_WAITALL」を呼ぶまで、受信バッファの内容を利用した処理を実施してはならない。

- call MPI_IRecv

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

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

MPI_WAITALL

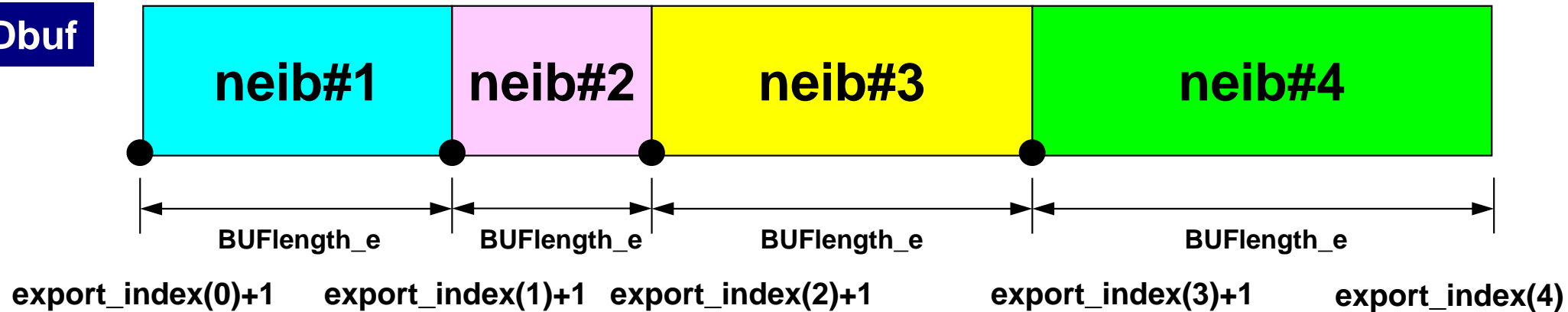
- 1対1非ブロッキング通信サブルーチンである「MPI_ISEND」と「MPI_IRecv」を使用した場合、プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_WAITALL」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_WAITALL」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば、「MPI_ISEND」と「MPI_IRecv」を同時に同期してもよい。
 - 「MPI_ISEND/IRecv」で同じ通信識別子を使用すること
- 「MPI_BARRIER」と同じような機能であるが、代用はできない。
 - 実装にもよるが、「request」、「status」の内容が正しく更新されず、何度も「MPI_ISEND/IRecv」を呼び出すと処理が遅くなる、というような経験もある。
- **call MPI_WAITALL (count, request, status, ierr)**
 - **count** 整数 I 同期する必要のある「MPI_ISEND」、「MPI_IRecv」呼び出し数。
 - **request** 整数 I/O 通信識別子。「MPI_ISEND」、「MPI_IRecv」で利用した識別子名に対応。(配列サイズ:(count))
 - **status** 整数 O 状況オブジェクト配列(配列サイズ:(MPI_STATUS_SIZE,count))
MPI_STATUS_SIZE: “mpif.h”, “mpi.h”で定められる
パラメータ
 - **ierr** 整数 O 完了コード

一般化された通信テーブル:送信

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

送信 (MPI_Isend/Irecv/Waitall)

SENDbuf



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

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

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

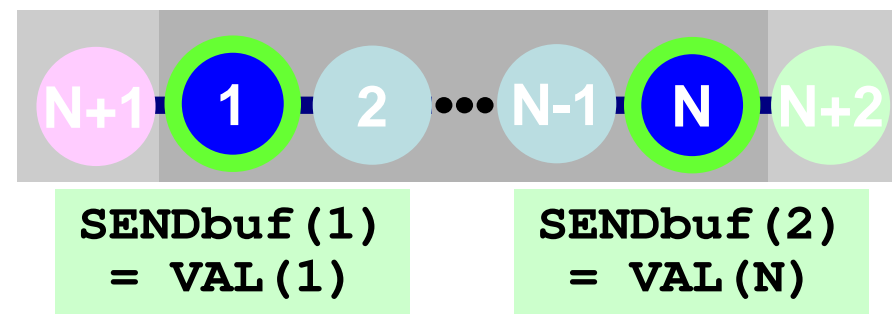
enddo

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

送信バッファへの代入

温度などの変数を直接送信, 受信に使うのではなく, このようなバッファへ一回代入して計算することを勧める。

送信：一次元問題



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

一般化された通信テーブル: 受信

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

受信 (MPI_Isend/Irecv/Waitall)

```

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

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

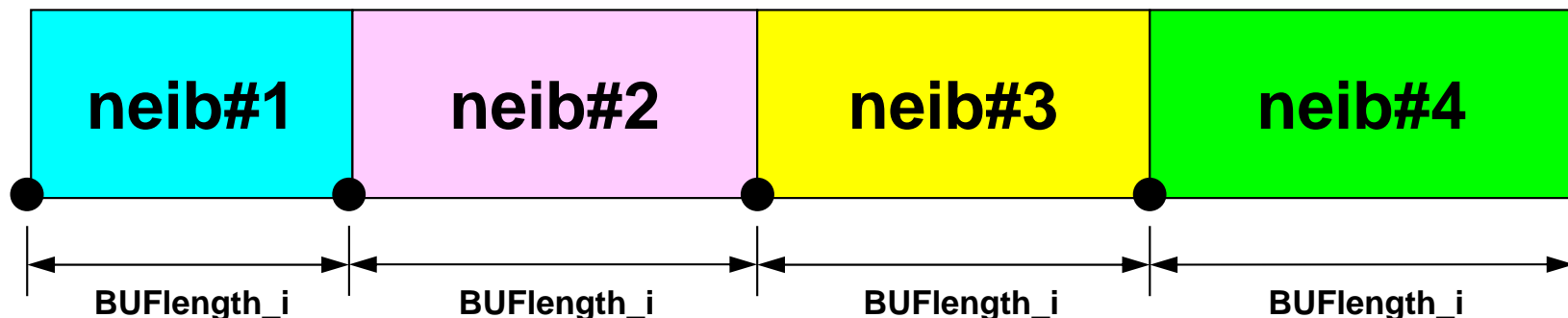
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

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

```

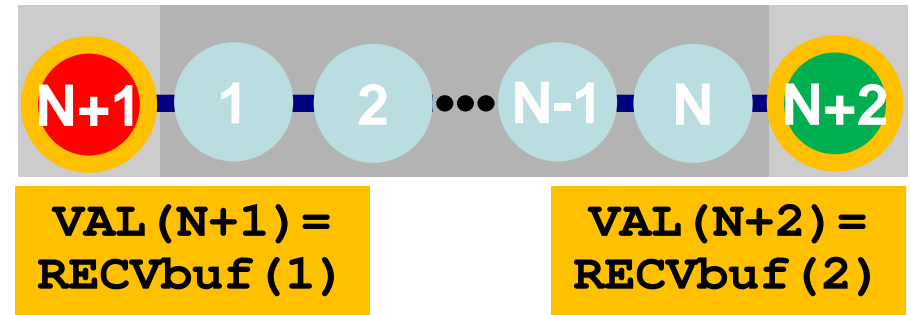
受信バッファから代入

RECVbuf



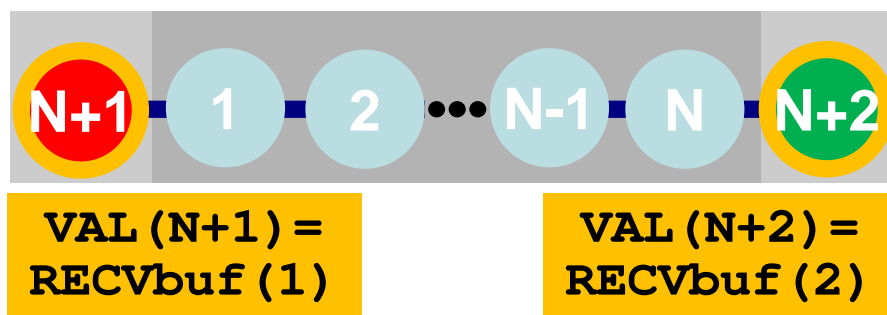
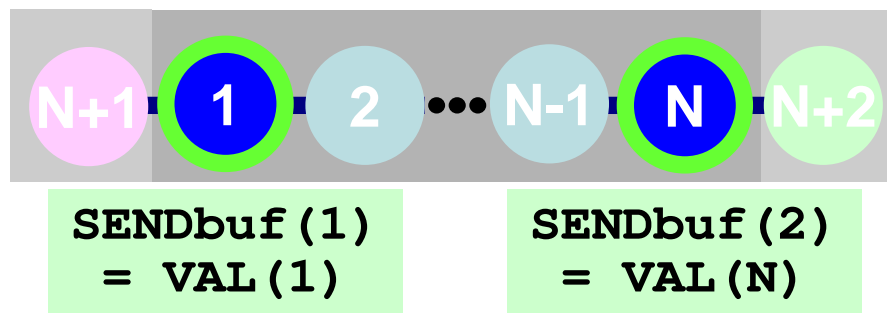
import_index(0)+1 import_index(1)+1 import_index(2)+1 import_index(3)+1 import_index(4)

受信:一次元問題



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

一般化された通信テーブル: Fortran



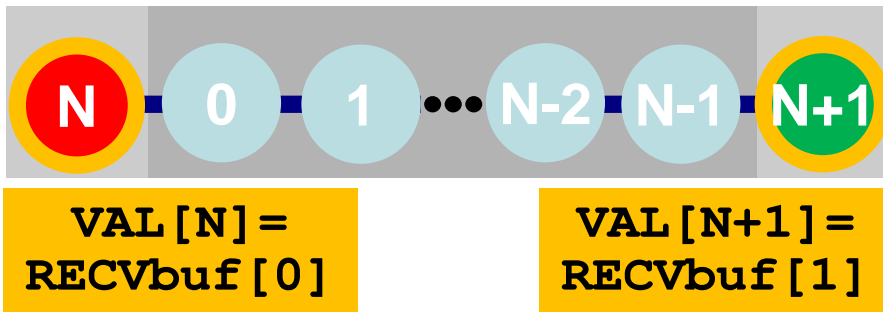
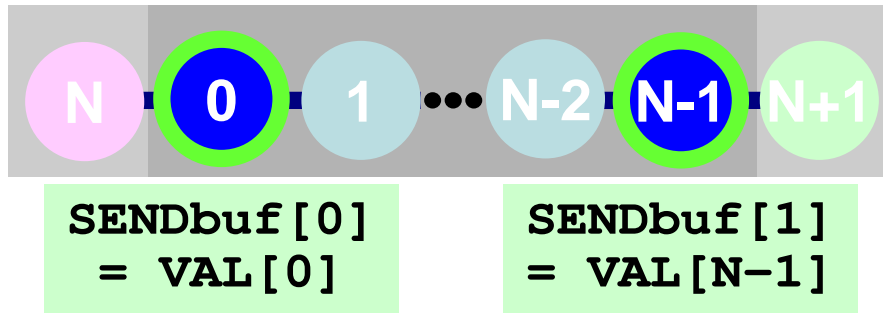
```
NEIBPETOT= 2
NEIBPE (1)= my_rank - 1
NEIBPE (2)= my_rank + 1
```

```
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.f(8/11)

全体マトリクス生成: 1CPUのときと全く同じ: 各要素→一様

```
!C
!C +-----+
!C | MATRIX ASSEMBLE |
!C +-----+
!C==
```

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

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

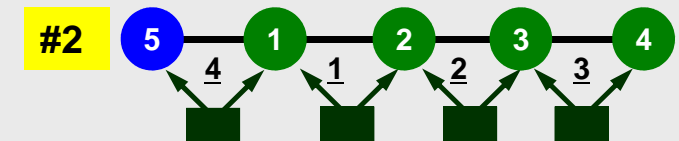
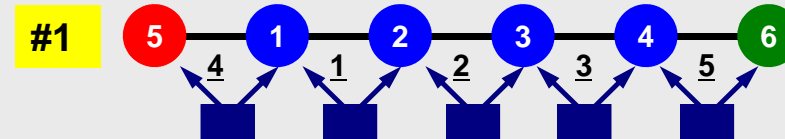
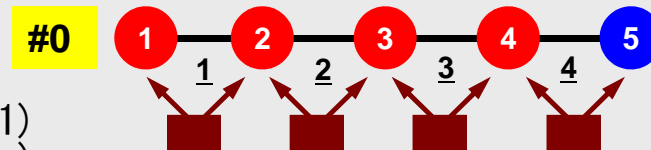
```
if (my_rank.eq.0.and.icel.eq.1) then
  k1= INDEX(in1-1) + 1
  else
  k1= INDEX(in1-1) + 2
endif
k2= INDEX(in2-1) + 1
```

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

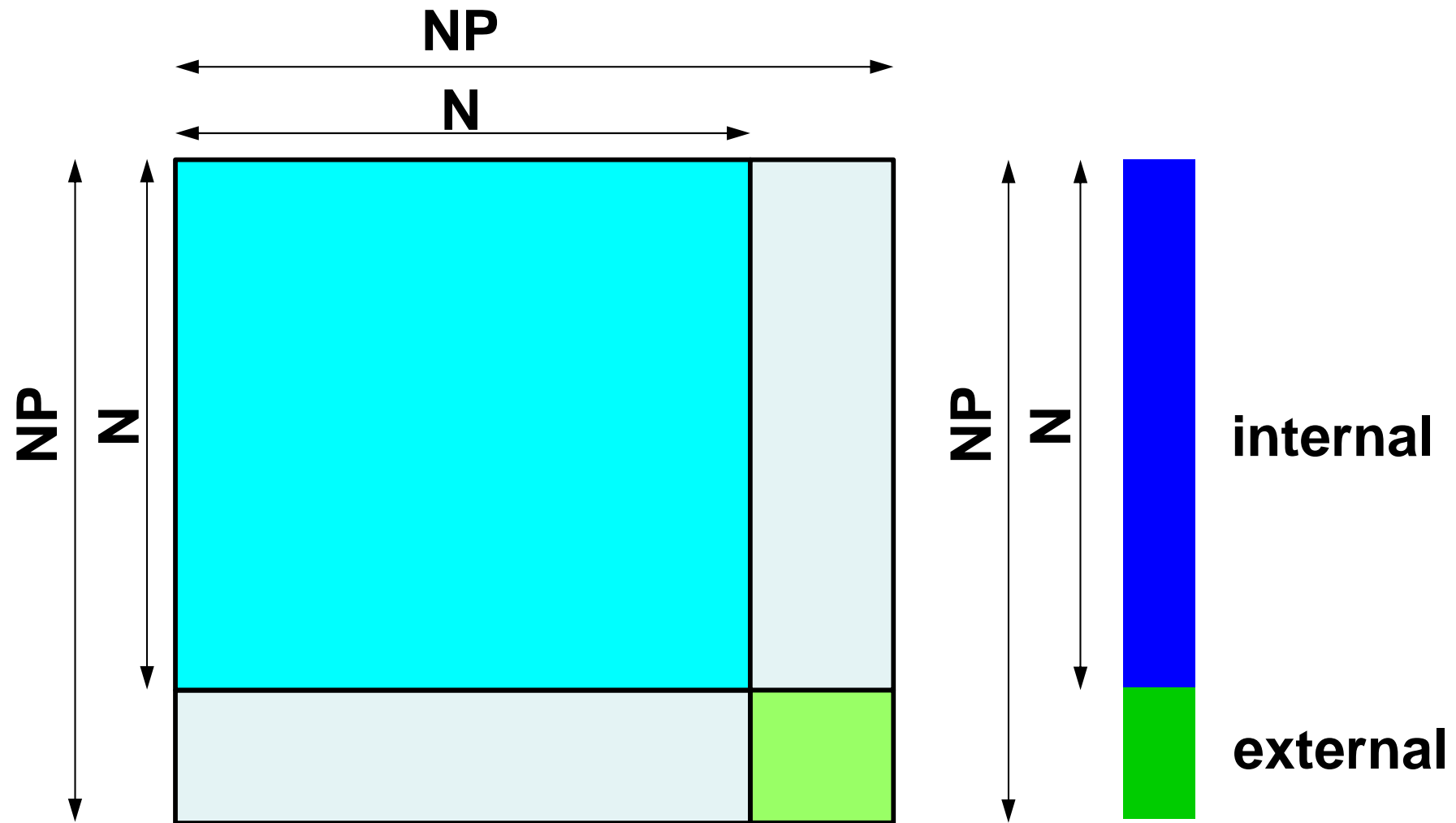
```
QN= 0.50d0*QV*AREA*DL
RHS(in1)= RHS(in1) + QN
RHS(in2)= RHS(in2) + QN
```

```
enddo
```

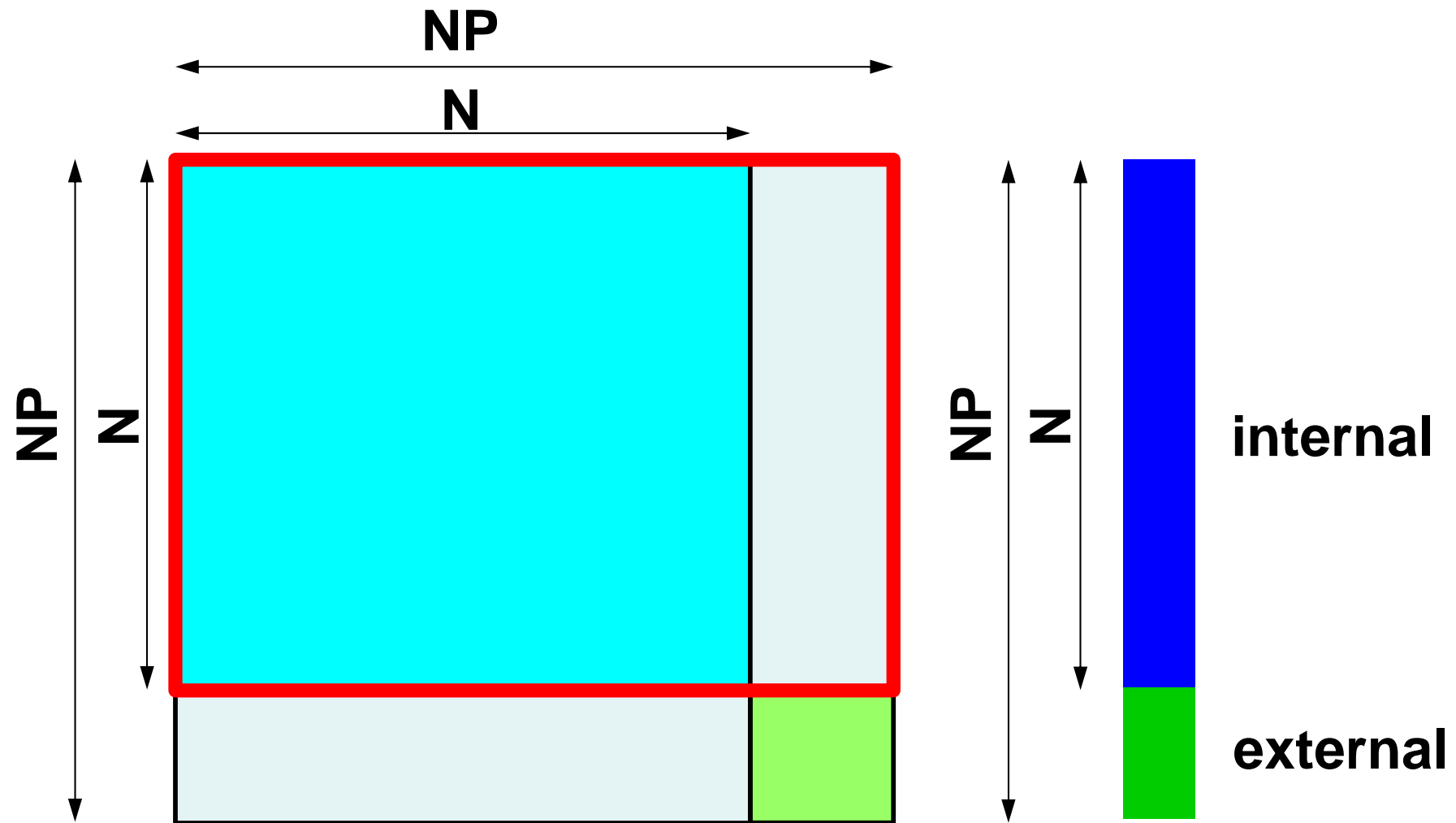
```
!C==
```



Local Matrix: 各プロセスにおける係数行列

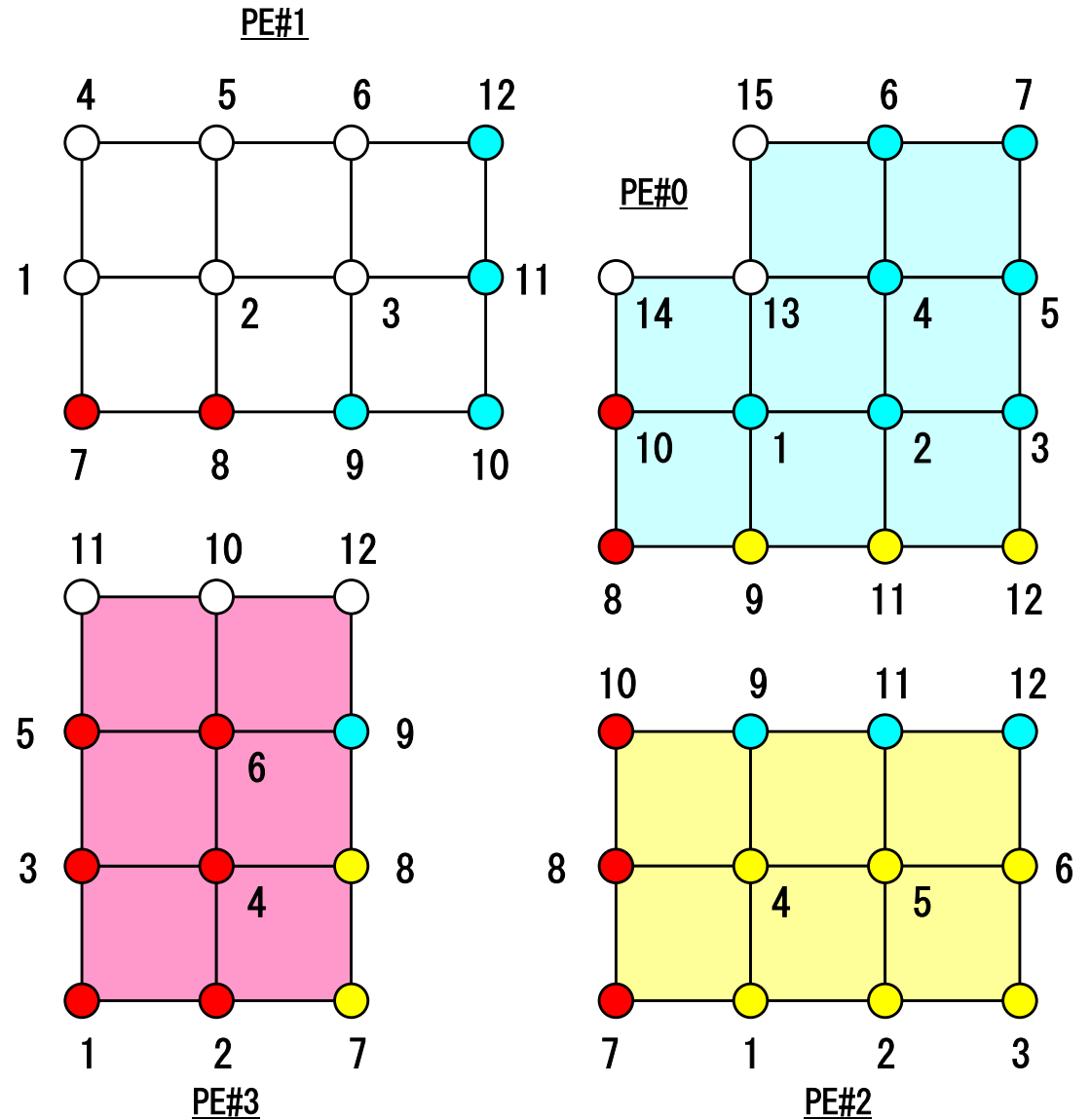
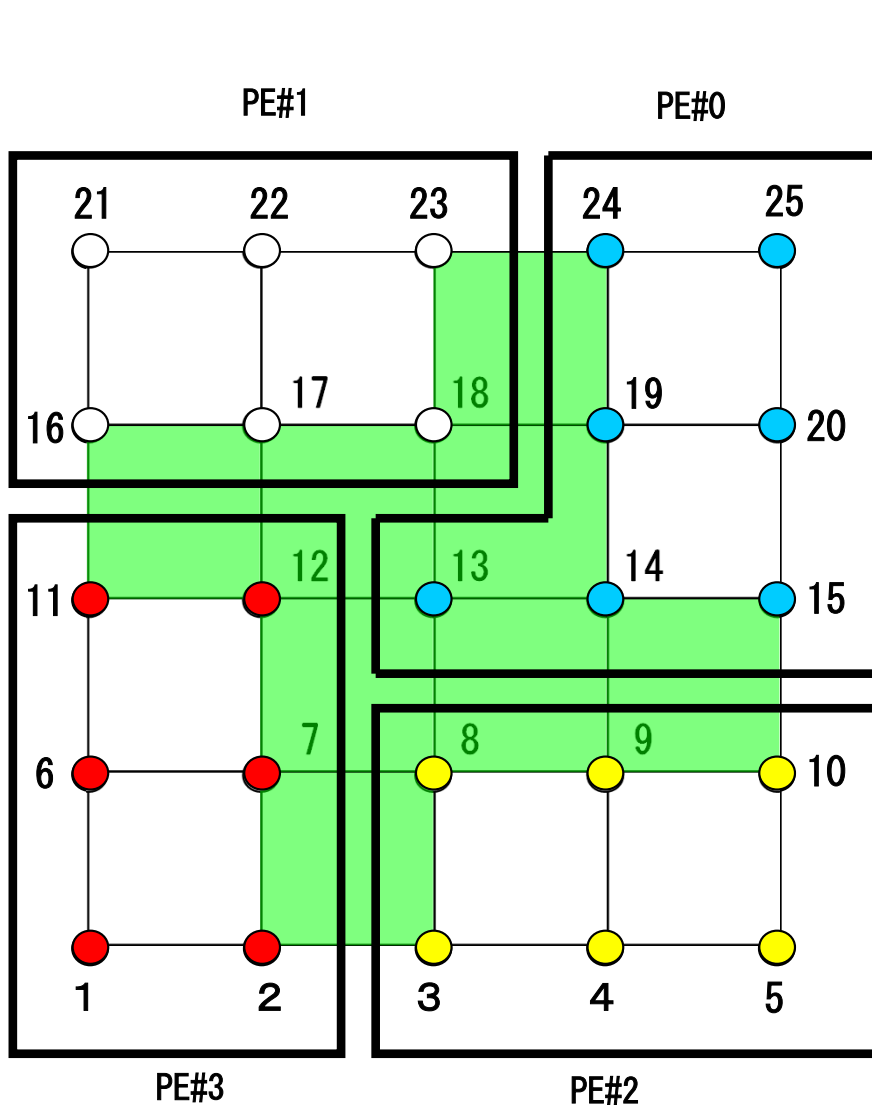


本当に必要なのはこの部分

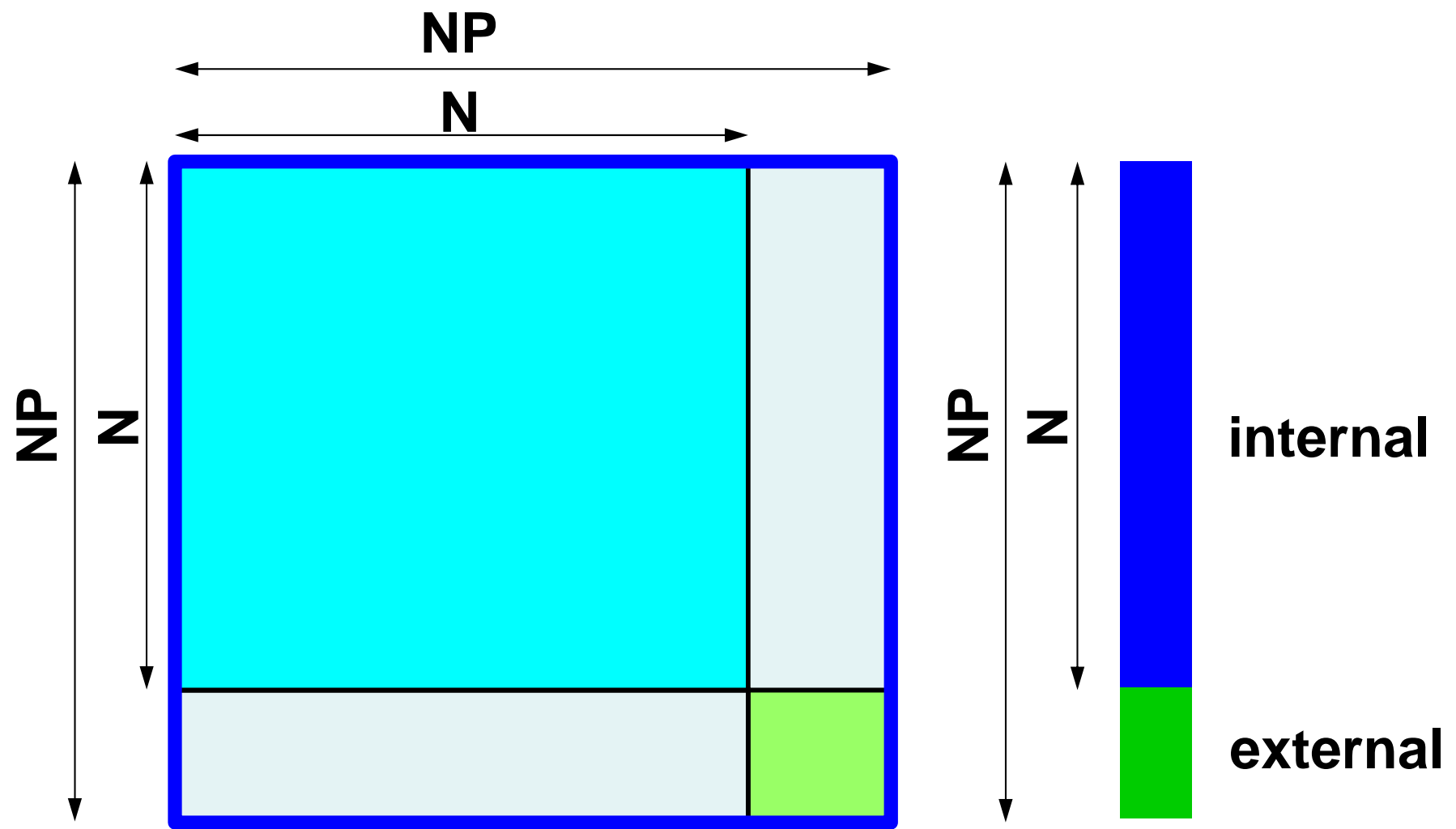


全ての要素の計算を実施する

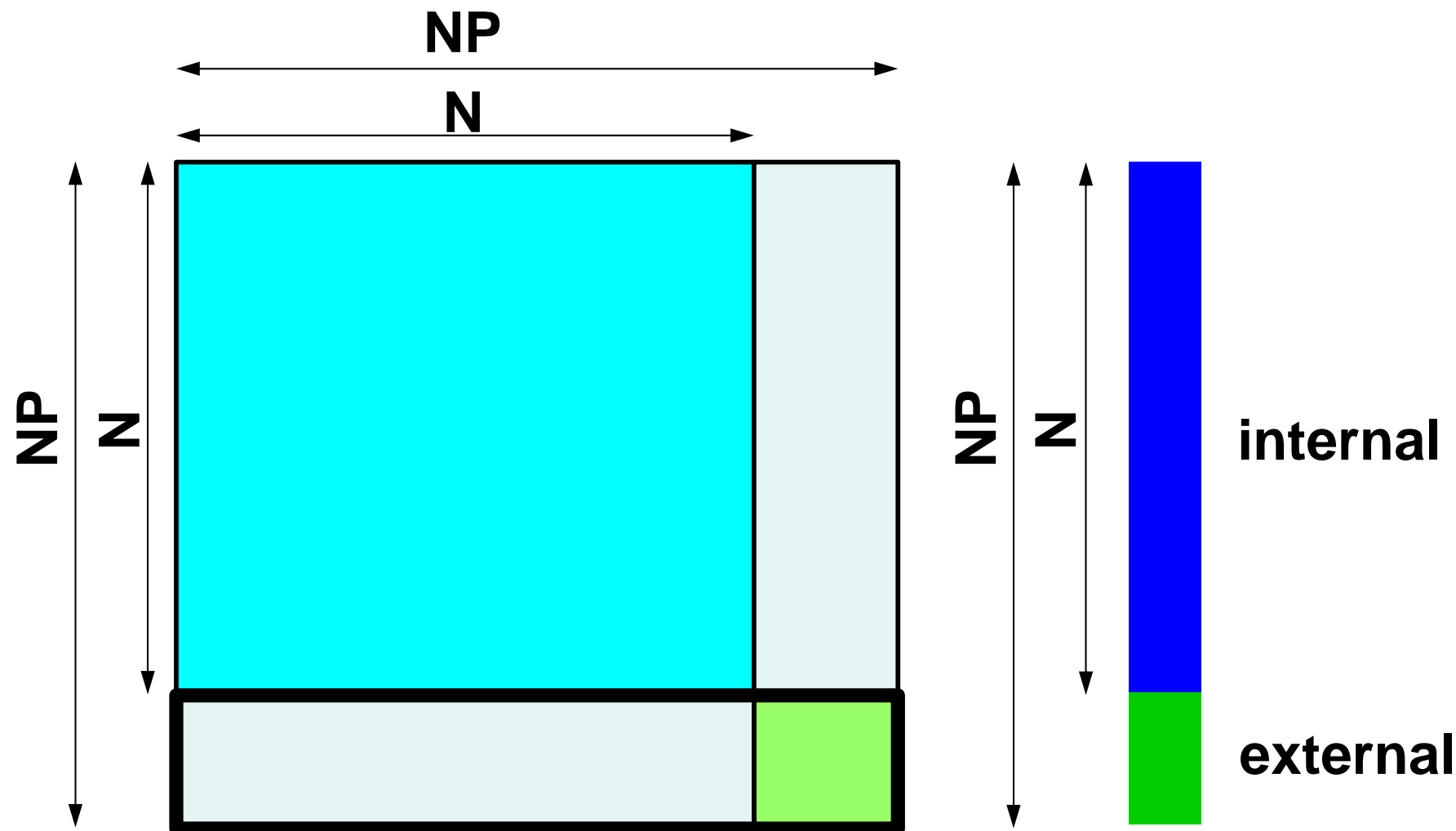
外点を含むオーバーラップ領域の要素の計算も実施



従って結果的にはこのような行列を得るが



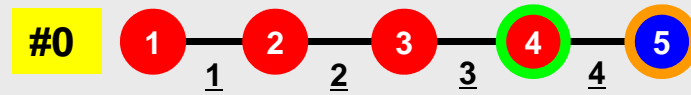
黒枠で囲んだ部分の行列は不完全
しかし、計算には使用しないのでこれで良い



プログラム: 1d.f(9/11)

境界条件: 1CPUのときとほとんど同じ

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

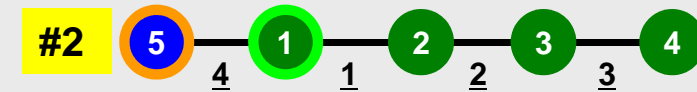


```
!C
!C-- X=Xmin
```

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

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

```
!C===
```



プログラム: 1d.f(10/11)

共役勾配法

```

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

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

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

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

```

```

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

```


共役勾配法

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

前处理, DAXPY

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

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

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

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

行列ベクトル積 (1/2)

通信テーブル使用, {p}の最新値を計算前に取得

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

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

do neib= 1, NEIBPETOT
  is = export_index(neib-1) + 1
  len_s= export_index(neib) - export_index(neib-1)
  call MPI_Isend (SENDbuf(is), len_s, MPI_DOUBLE_PRECISION, &
& NEIBPE(neib), 0, MPI_COMM_WORLD, request_send(neib), ierr)
enddo

do neib= 1, NEIBPETOT
  ir = import_index(neib-1) + 1
  len_r= import_index(neib) - import_index(neib-1)
  call MPI_Irecv (RECVbuf(ir), len_r, MPI_DOUBLE_PRECISION, &
& NEIBPE(neib), 0, MPI_COMM_WORLD, request_recv(neib), ierr)
enddo

call MPI_Waitall (NEIBPETOT, request_recv, stat_recv, ierr)

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

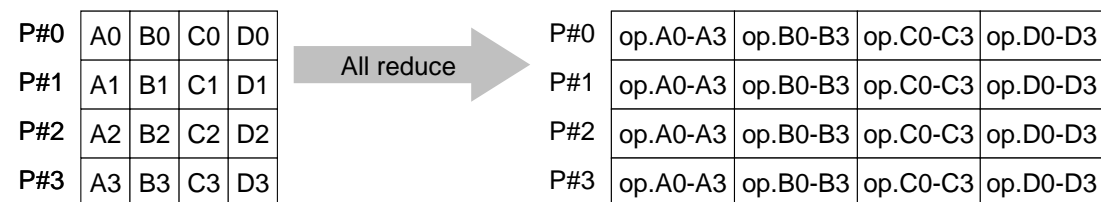
行列ベクトル積 (2/2)

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

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

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


MPI_ALLREDUCE



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

- call MPI_ALLREDUCE

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

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

CG法 (1/5)

```

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

do neib= 1, NEIBPETOT
  is = export_index(neib-1) + 1
  len_s= export_index(neib) - export_index(neib-1)
  call MPI_Isend (SENDbuf(is), len_s,
&                MPI_DOUBLE_PRECISION,
&                NEIBPE(neib), 0, MPI_COMM_WORLD,
&                request_send(neib), ierr)
enddo

do neib= 1, NEIBPETOT
  ir = import_index(neib-1) + 1
  len_r= import_index(neib) - import_index(neib-1)
  call MPI_Irecv (RECVbuf(ir), len_r,
&                MPI_DOUBLE_PRECISION,
&                NEIBPE(neib), 0, MPI_COMM_WORLD,
&                request_recv(neib), ierr)
enddo
call MPI_Waitall (NEIBPETOT, request_recv, stat_recv, ierr)

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

```

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

for $i = 1, 2, \dots$

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

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

if $i=1$

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

else

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

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

endif

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

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

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

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

check convergence $|r|$

end

CG法 (2/5)

```

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

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

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

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

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

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

  RH00= 0.d0
  do i= 1, N
    RH00= RH00 + W(i,R)*W(i,Z)
  enddo
  call MPI_Allreduce (RH00, RHO, 1, MPI_DOUBLE_PRECISION,
& MPI_SUM, MPI_COMM_WORLD, ierr)

```

```

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

end

```


CG法 (3/5)

```

!C
!C-- {p} = {z} if      ITER=1
!C  BETA= RHO / RH01 otherwise

      if ( iter.eq.1 ) then
        do i= 1, N
          W(i,P)= W(i,Z)
        enddo
      else
        BETA= RHO / RH01
        do i= 1, N
          W(i,P)= W(i,Z) + BETA*W(i,P)
        enddo
      endif

```

```

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

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

      do neib= 1, NEIBPETOT
        is = export_index(neib-1) + 1
        len_s= export_index(neib) - export_index(neib-1)
        call MPI_Isend (SENDbuf(is), len_s, MPI_DOUBLE_PRECISION,
&                      NEIBPE(neib), 0, MPI_COMM_WORLD,
&                      request_send(neib), ierr)
      enddo

```

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

for $i = 1, 2, \dots$

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

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

if $i=1$

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

else

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

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

endif

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

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

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

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

check convergence $|r|$

end

CG法 (4/5)

```

do neib= 1, NEIBPETOT
  ir  = import_index(neib-1) + 1
  len_r= import_index(neib) - import_index(neib-1)
  call MPI_Irecv (RECVbuf(ir), len_r,
&                MPI_DOUBLE_PRECISION,
&                NEIBPE(neib), 0, MPI_COMM_WORLD,
&                request_recv(neib), ierr)
enddo
call MPI_Waitall (NEIBPETOT, request_recv, stat_recv, ierr)

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

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

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

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

```

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

for $i = 1, 2, \dots$

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

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

if $i=1$

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

else

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

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

endif

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

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

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

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

 check convergence $|r|$

end

CG法 (5/5)

```

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

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

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

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

RESID = dsqrt(DNRM2/BNRM2)

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

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

enddo

```

```

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

```

プログラム: 1d.f(11/11)

結果書き出し: 各プロセスごとの実施

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

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

      call MPI_FINALIZE (ierr)
      end program heat1Dp
```

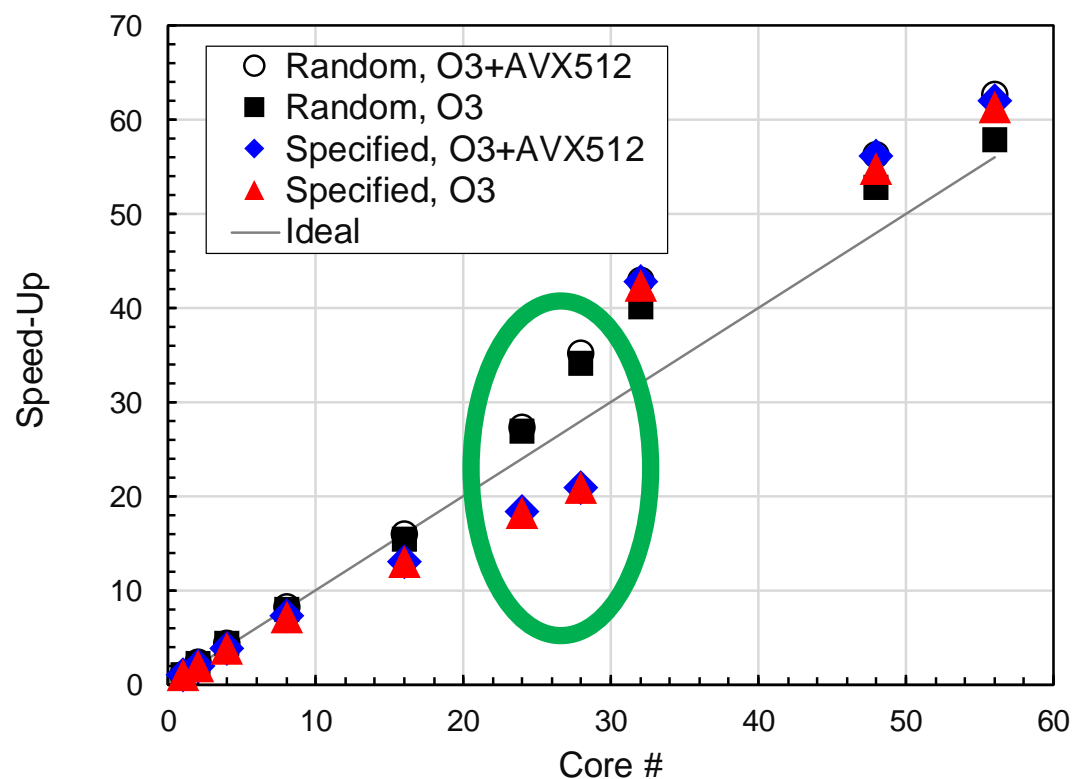
- 問題の概要, 実行方法
- プログラムの説明
- 計算例

計算結果（1次元）：CG法部分， 10^6

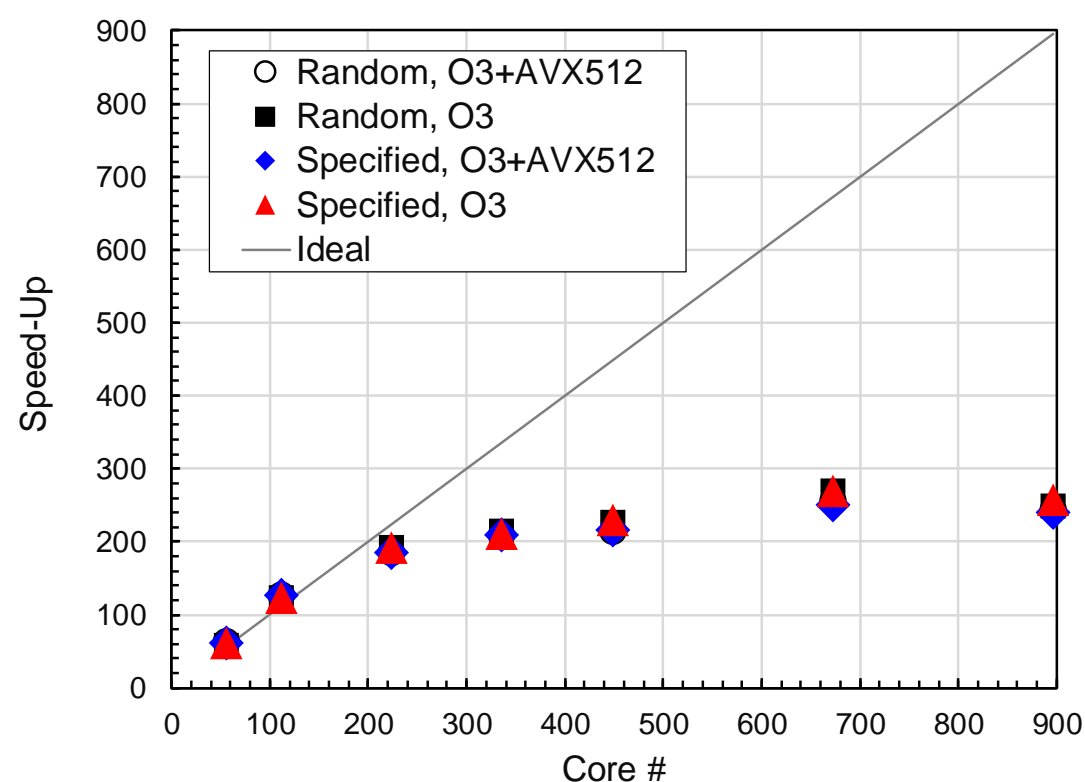
1,000回反復に要する時間， Strong Scaling

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



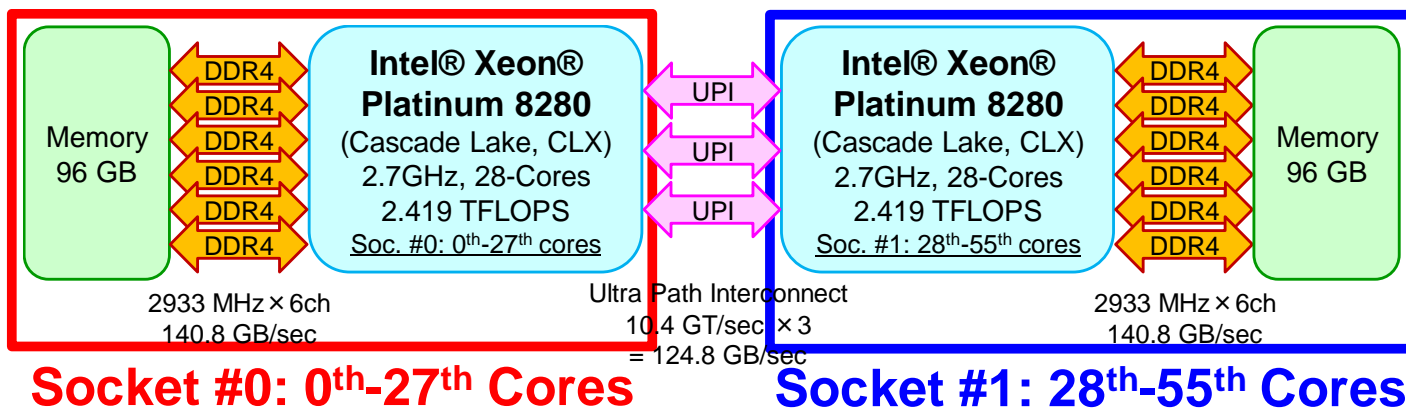
1-node, 24-cores

```
#!/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



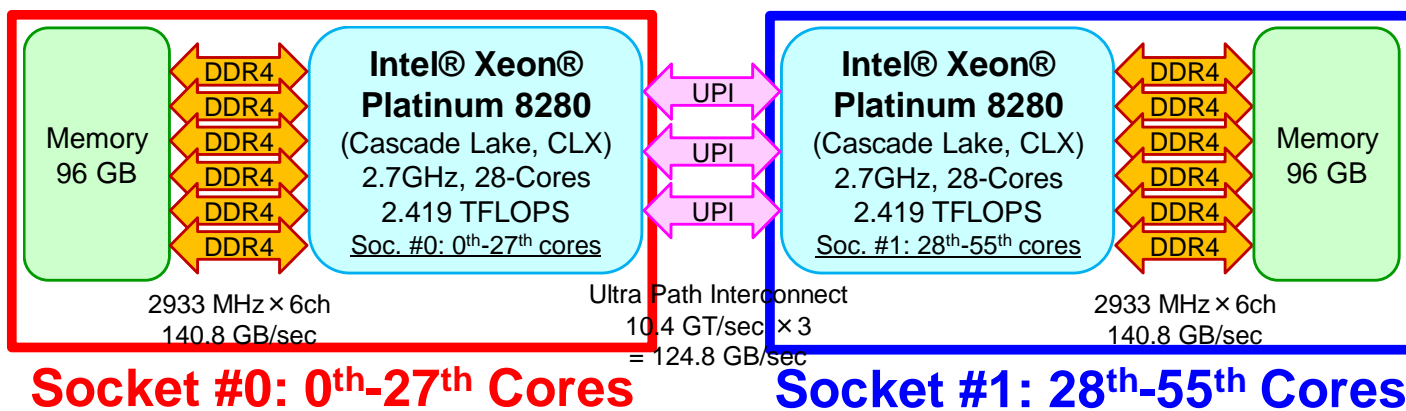
1-node, 28-cores

```
#!/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

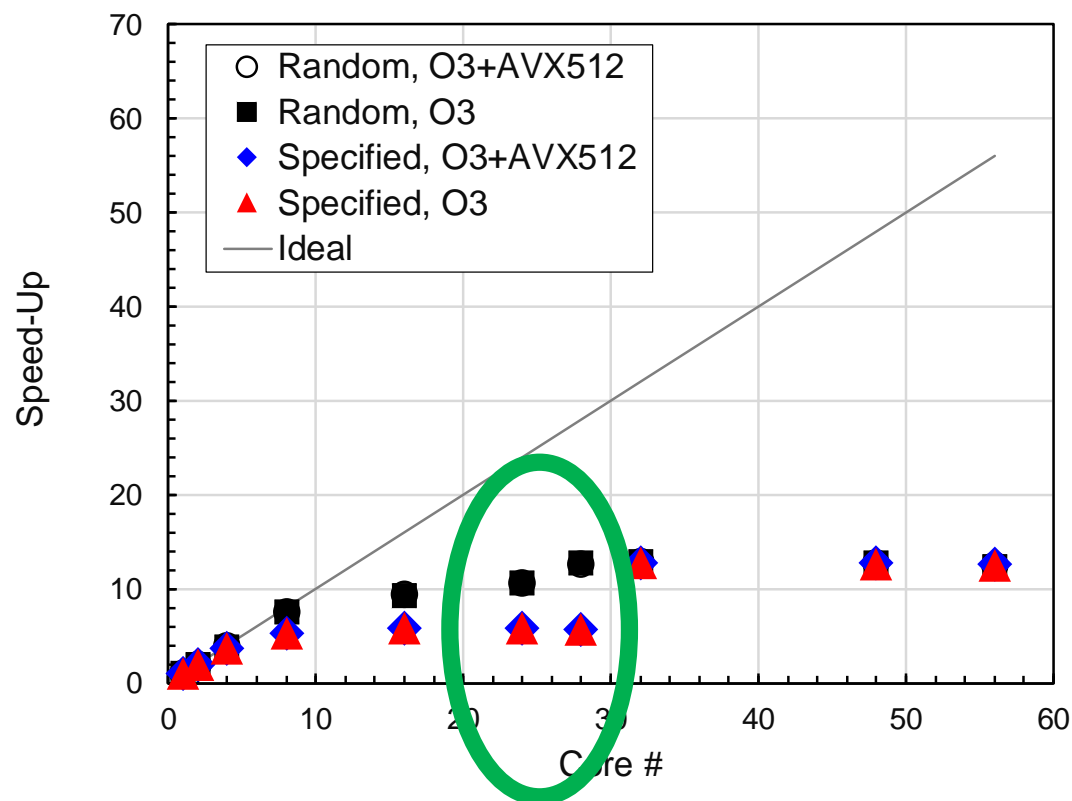


計算結果（1次元）：CG法部分， 10^7

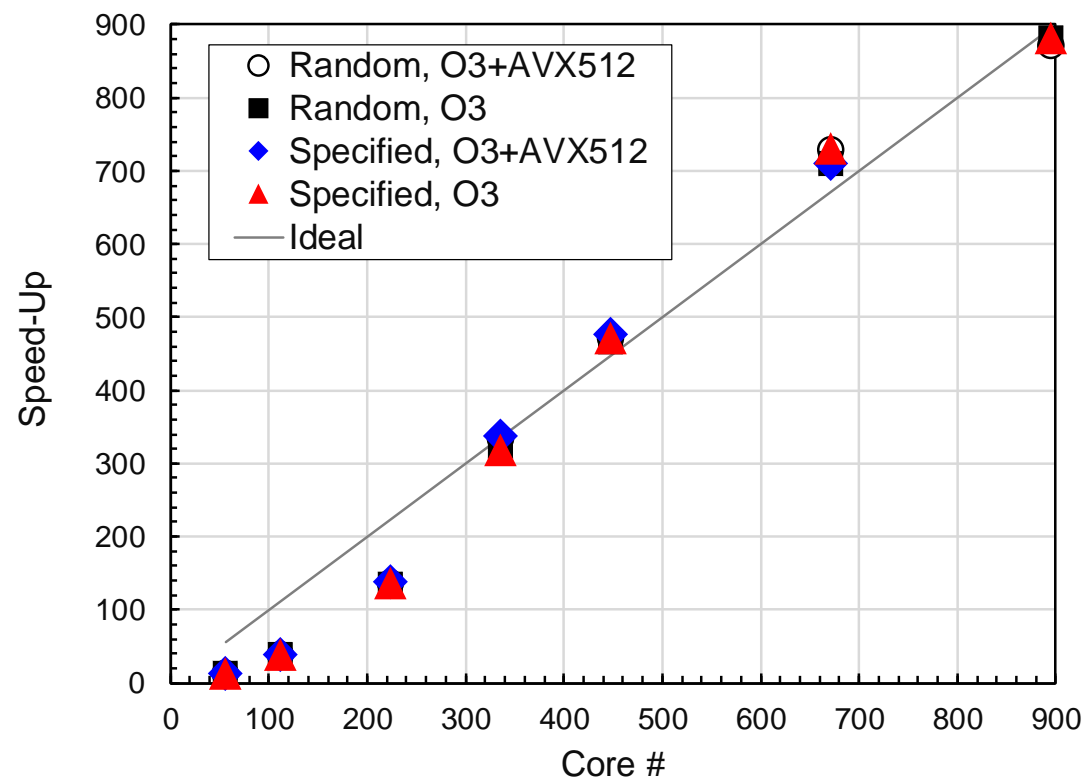
1,000回反復に要する時間， Strong Scaling

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

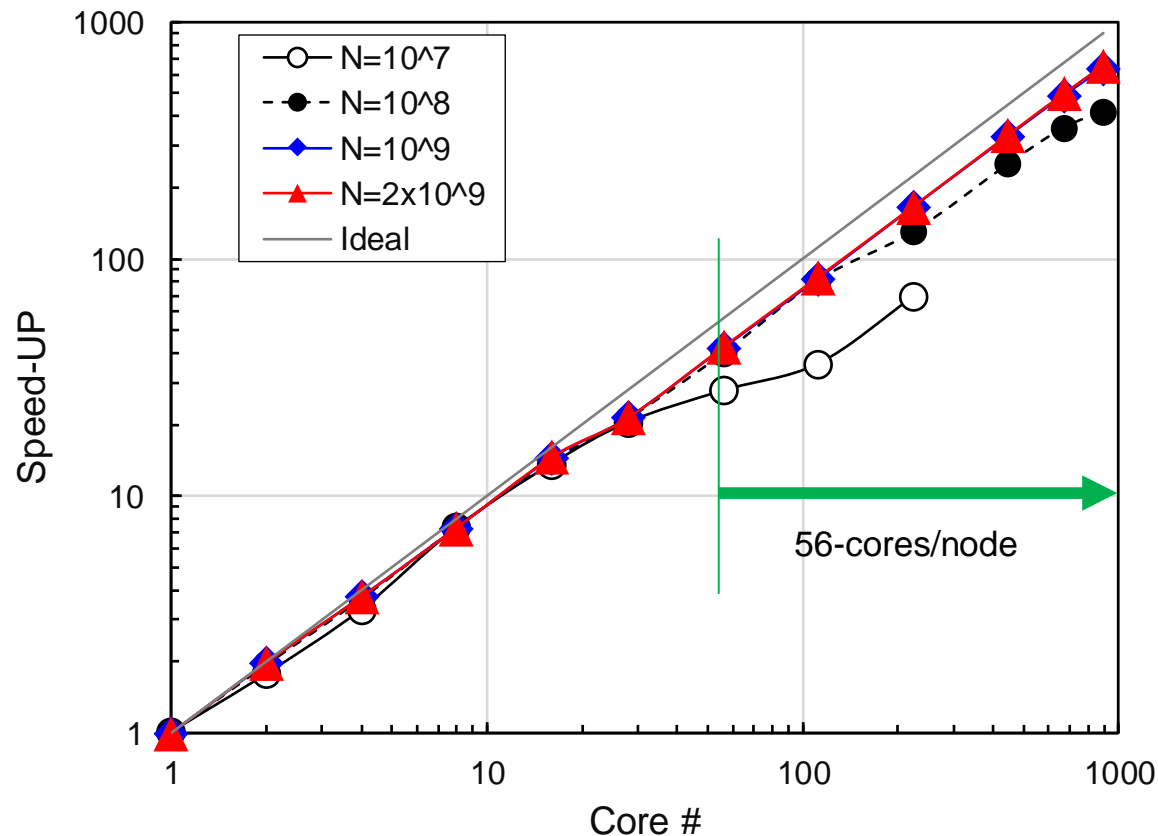


大規模並列計算：理想値からのずれ

- 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 μ secのオーダー
- MPIの同期のための時間
 - プロセス数が増加すると増加する傾向
- 計算時間が小さい場合はこれらの効果は無視できない。
 - 特に, 送信メッセージ数が小さい場合は, 「Latency」が効く。

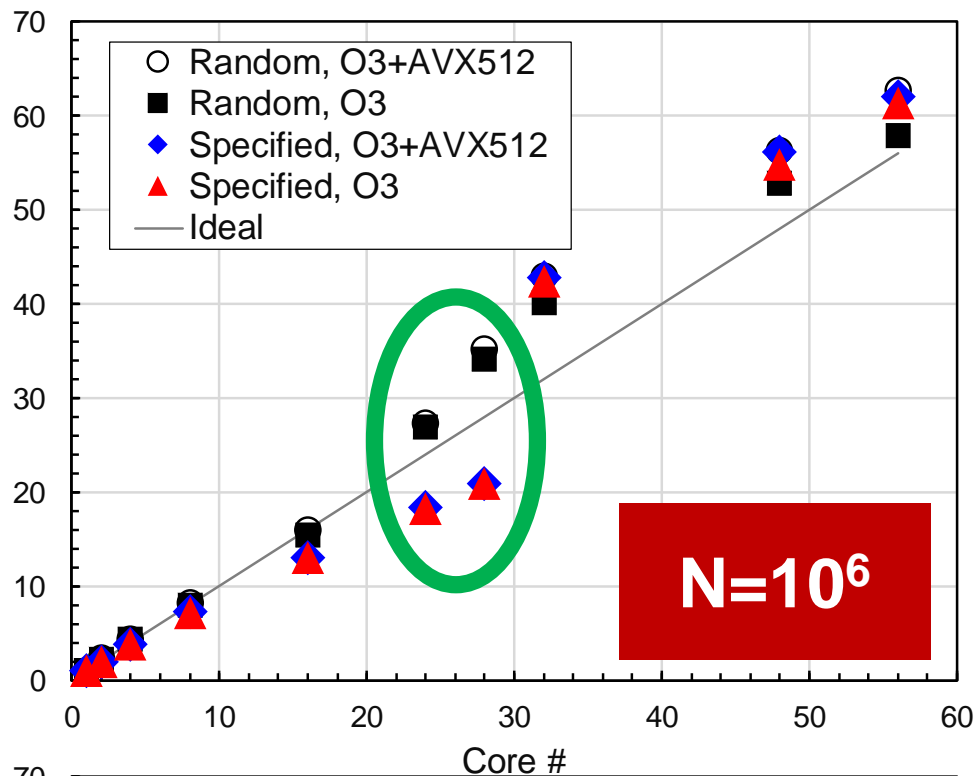
大規模並列計算：理想値からのずれ

- 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

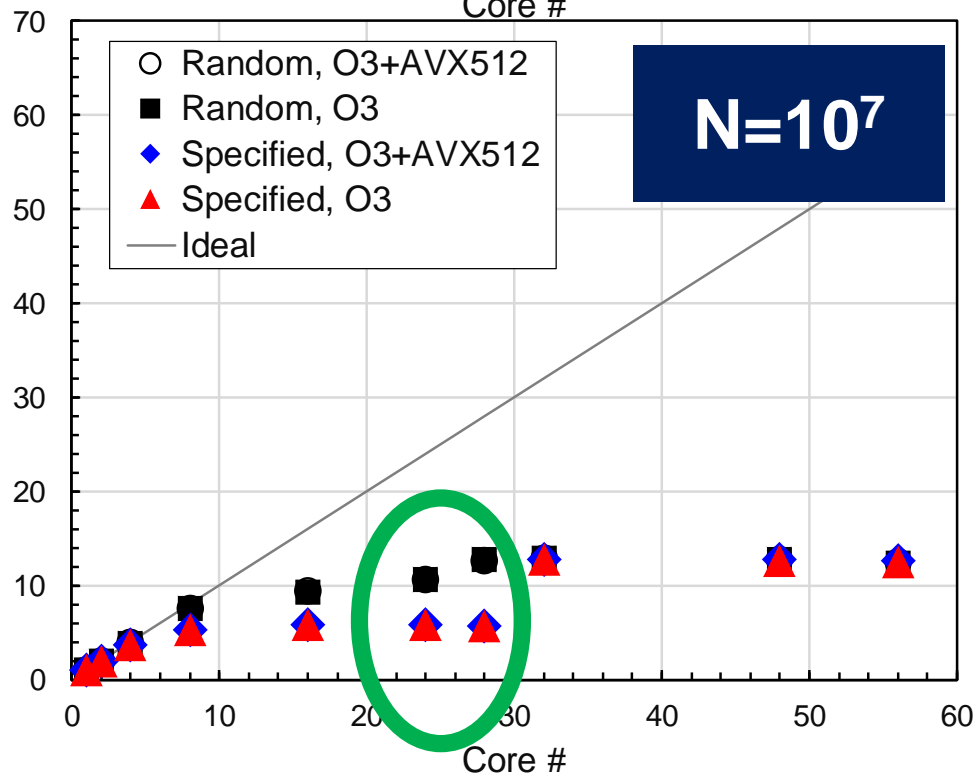


Up to 56 cores (single node)

Speed-Up



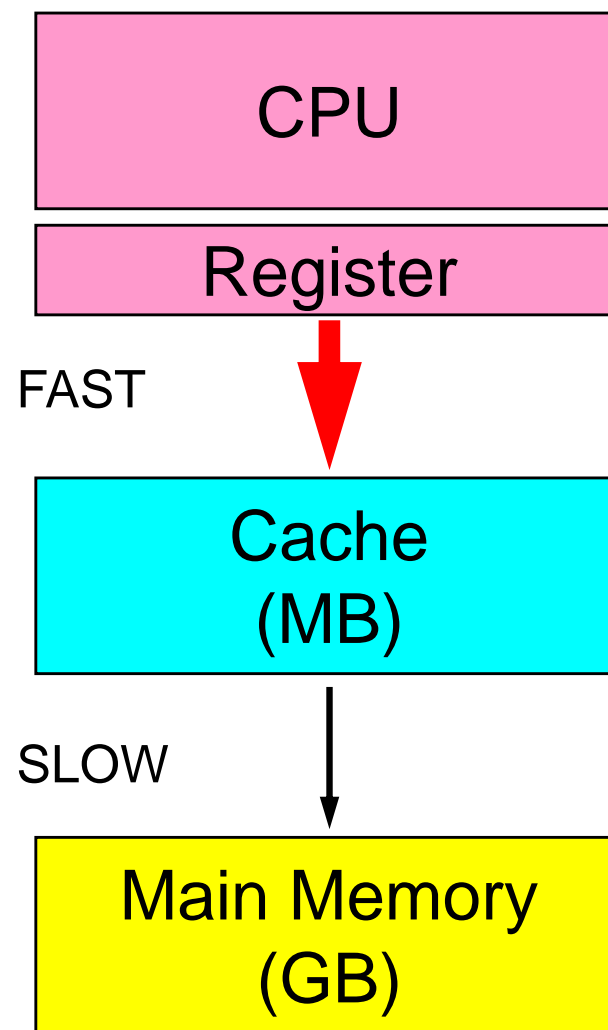
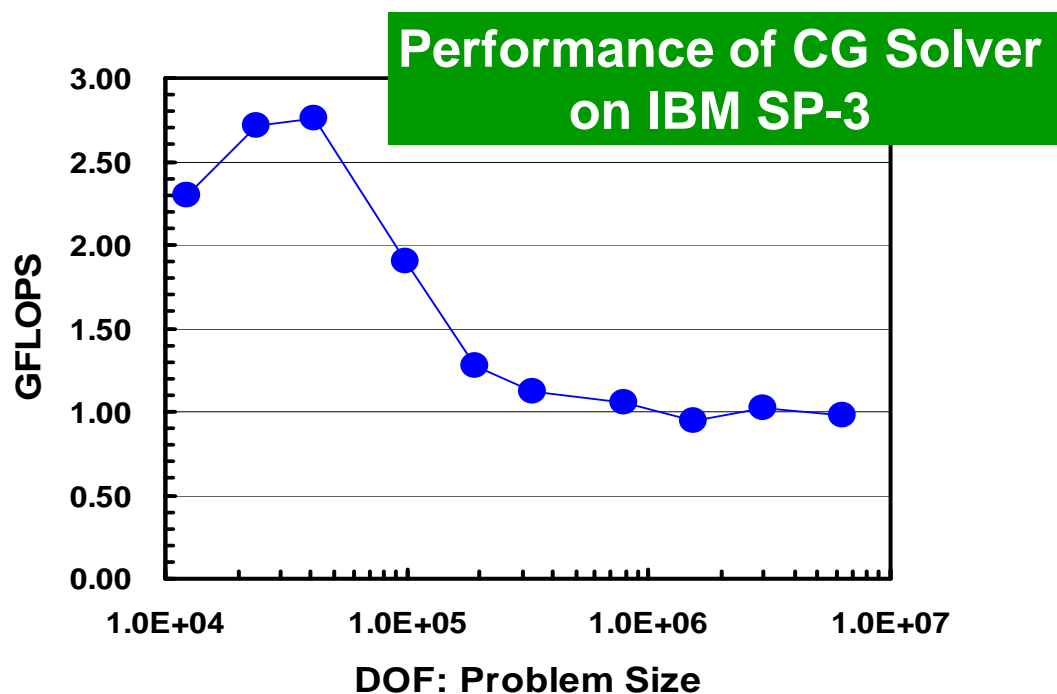
Speed-Up



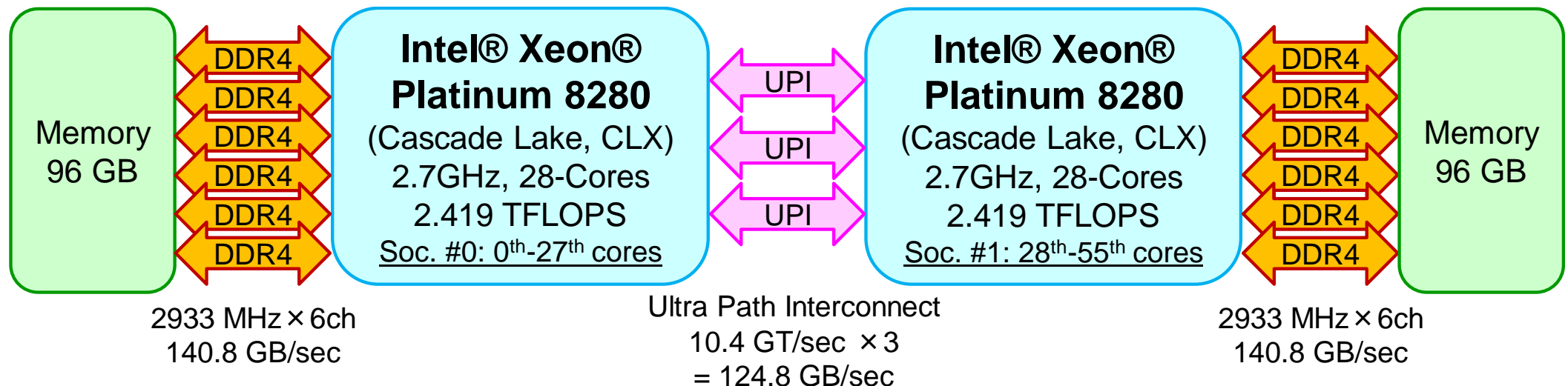
- ノード内（56コアまで）では問題規模が小さいと性能良い
- メモリ競合（通信の影響ではない）
- ノード内メモリ転送性能は8コア以上ではほとんど変化しない（Stream）
- 問題サイズが小さいとキャッシュを有効利用できるため、メモリ転送性能の影響少ない
- 2ソケット利用時のメモリ・キャッシュは「Random」の方が有効に利用されている

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、一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

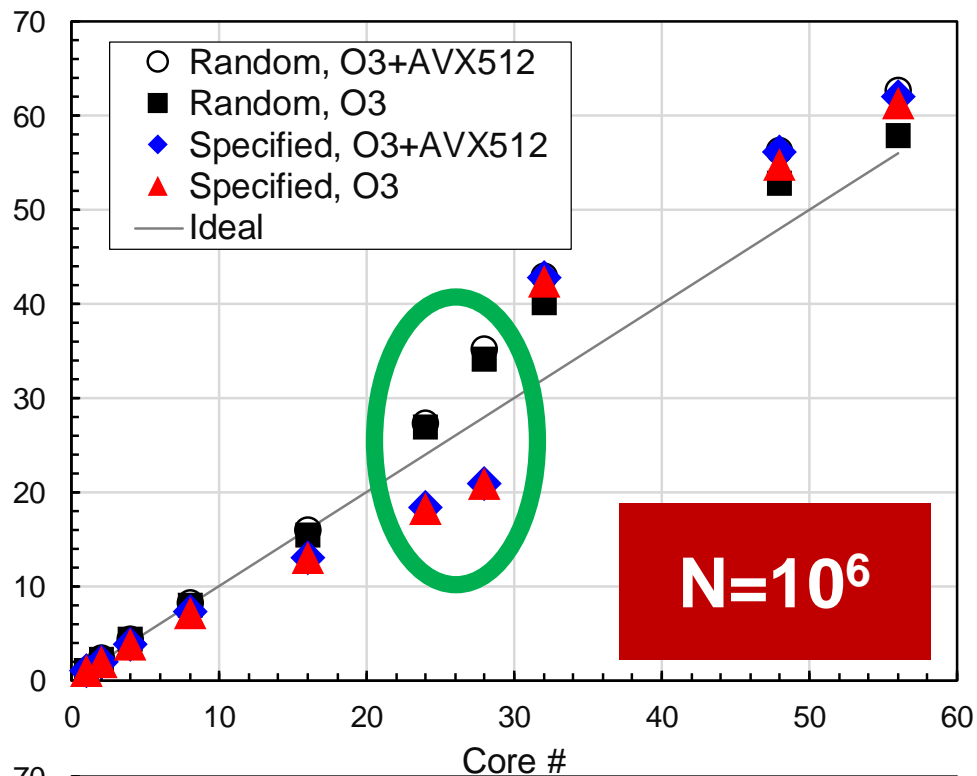


Category	Capacity	X-Way Set Associative	Cache Line
L1\$Data	32 KB/core	8-Way	64B
L1\$Instruction	32 KB/core	8-Way	64B
L2	1.00 MB/core	16-Way	64B
L3	38.5 MB/socket	11-Way	64B

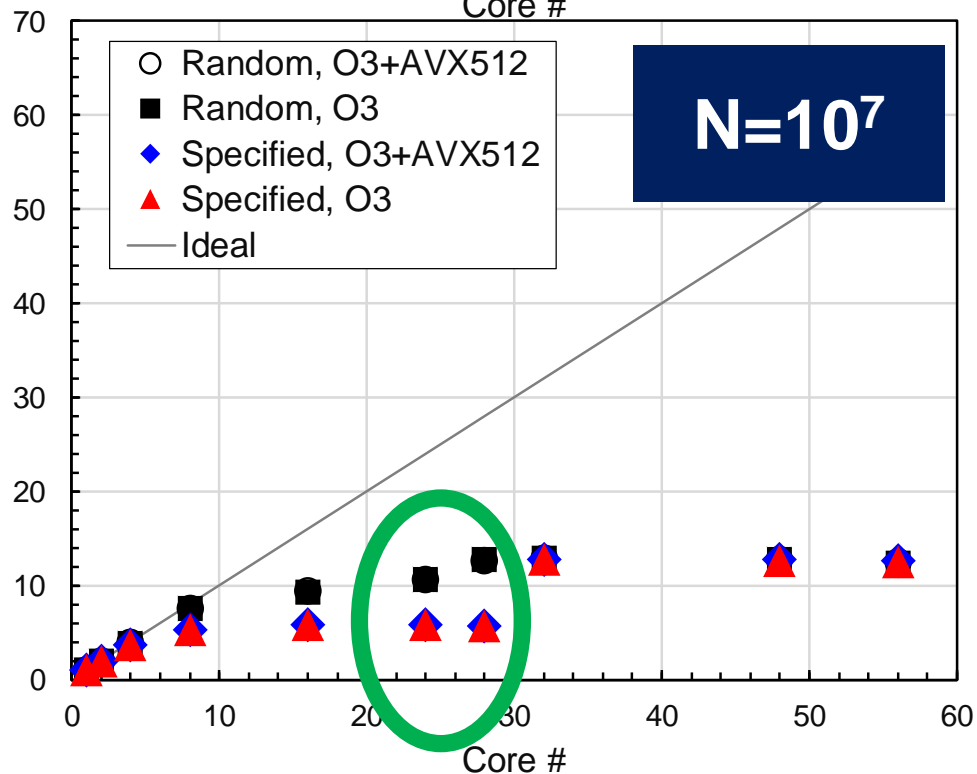


Up to 56 cores (single node)

Speed-Up



Speed-Up



- Required Memory
- N=10⁶: 80MB
 - Very Close to Cache Size
- N=10⁷: 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

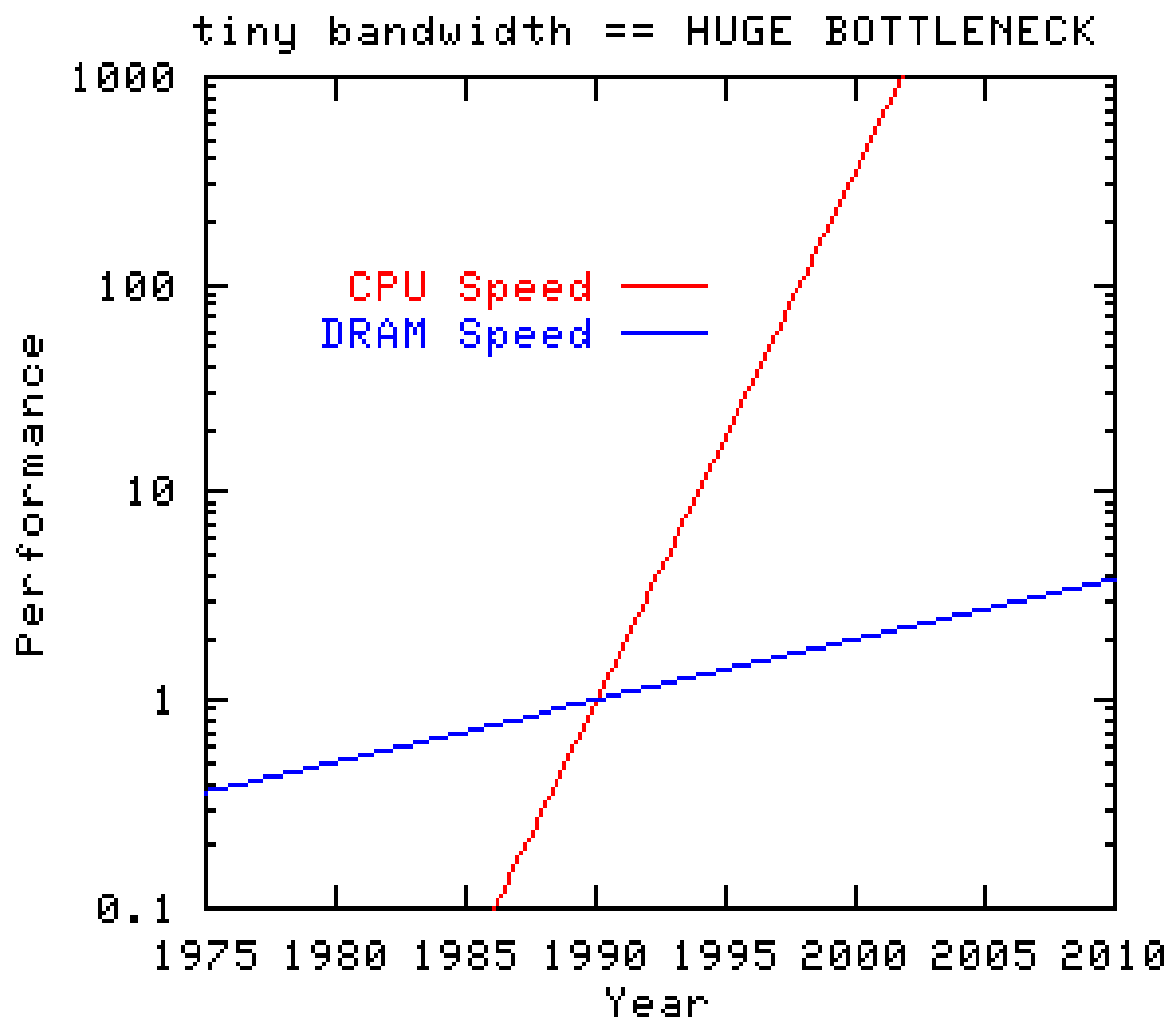
Number of processors = 16
 Array size = 2000000
 Offset = 0
 The total memory requirement is 732.4 MB
 (45.8MB/task)
 You are running each test 10 times

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



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

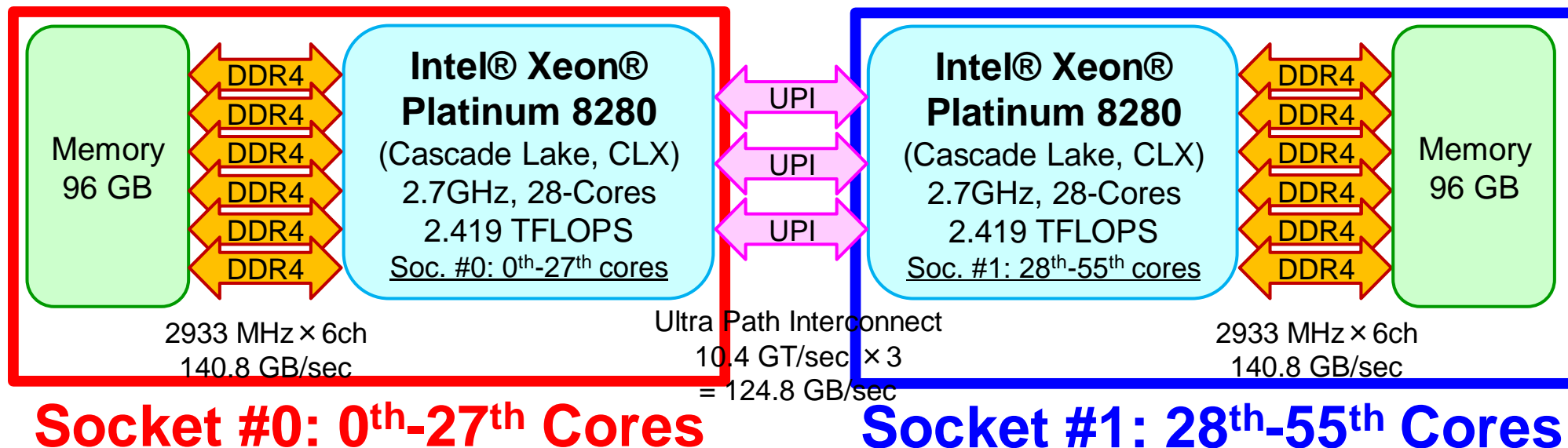
Copy, Compile and Run

```

>$ 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
```

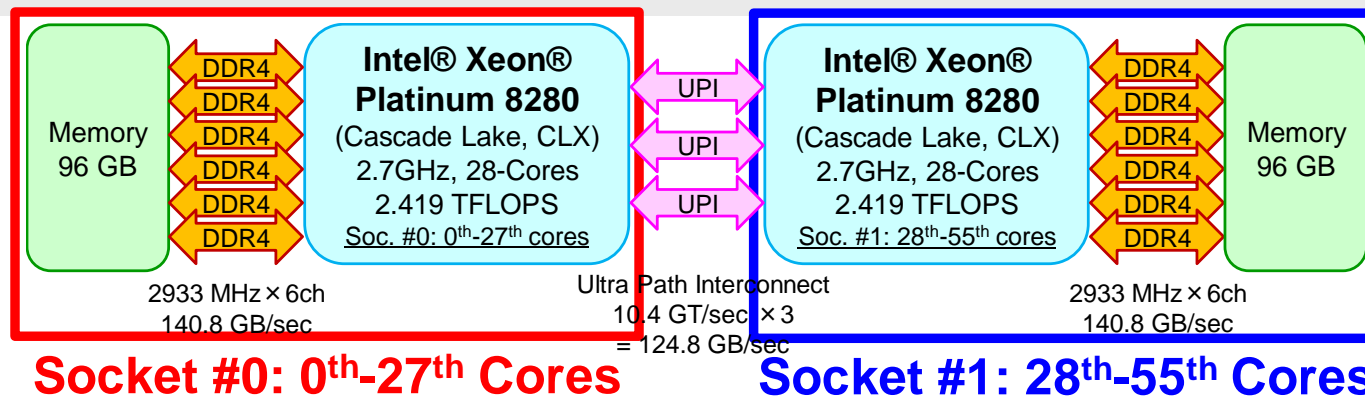
```
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

```
export I_MPI_PIN_PROCESSOR_LIST=0
```

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

Core are specified



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
```

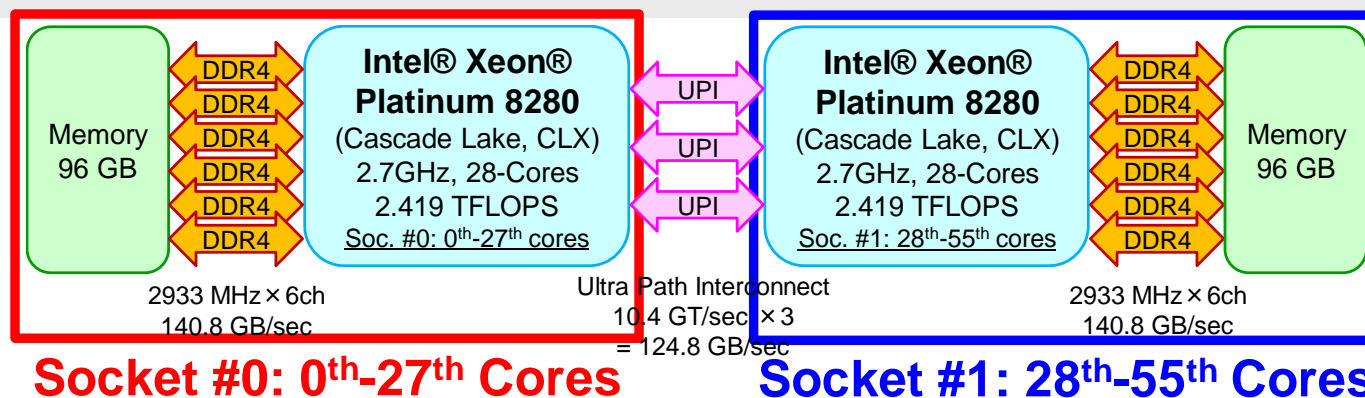
```
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

```
export I_MPI_PIN_PROCESSOR_LIST=0-15
```

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

Core are specified



s32.sh: Use 32 cores

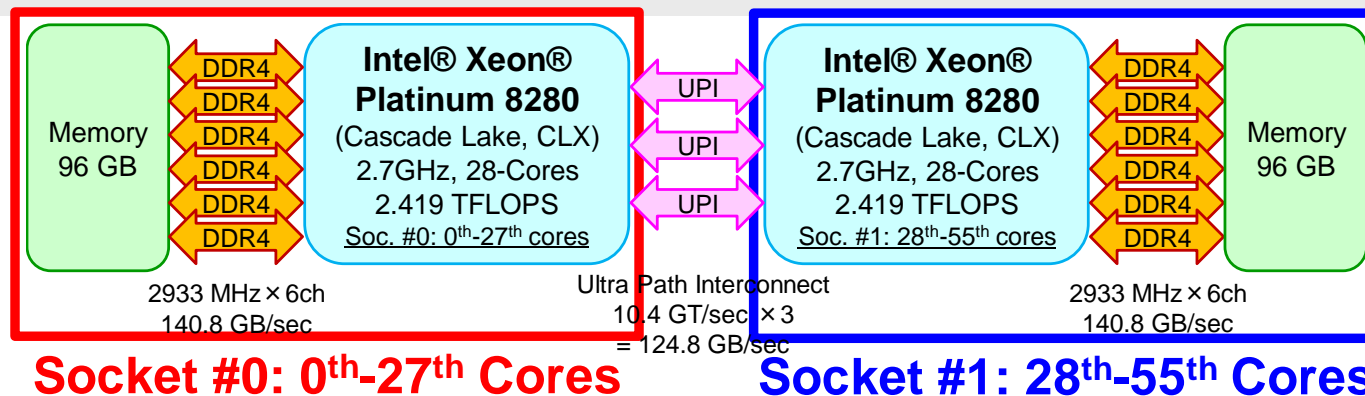
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=32
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s32.lst
```

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

Cores are randomly selected

```
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
```

Core are specified



s48.sh: Use 48 cores

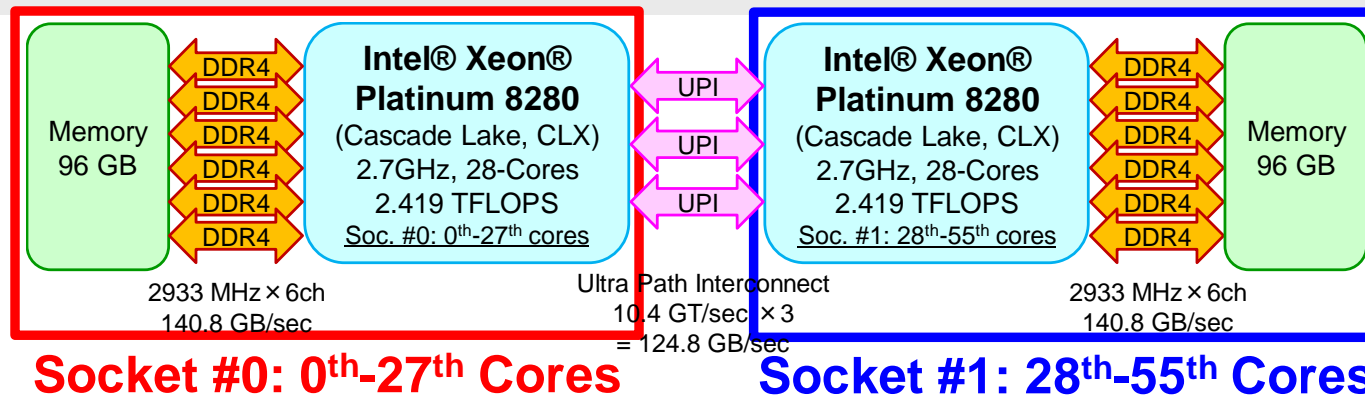
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=tutorial
#PJM -L node=1
#PJM --mpi proc=48
#PJM -L elapse=00:15:00
#PJM -g gt00
#PJM -j
#PJM -e err
#PJM -o s48.lst
```

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

Cores are randomly selected

```
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
```

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
```

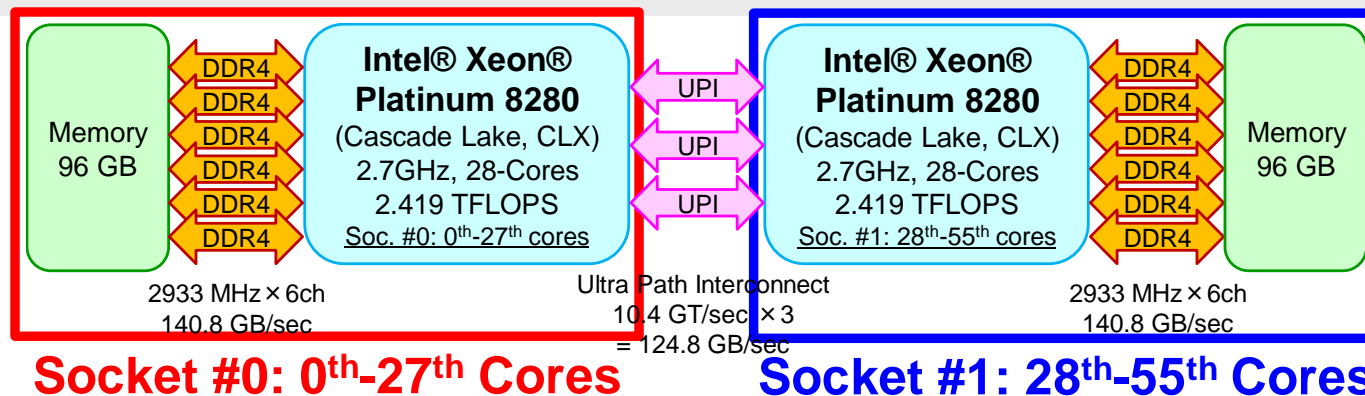
```
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

Cores are randomly selected

```
export I_MPI_PIN_PROCESSOR_LIST=0-55
```

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

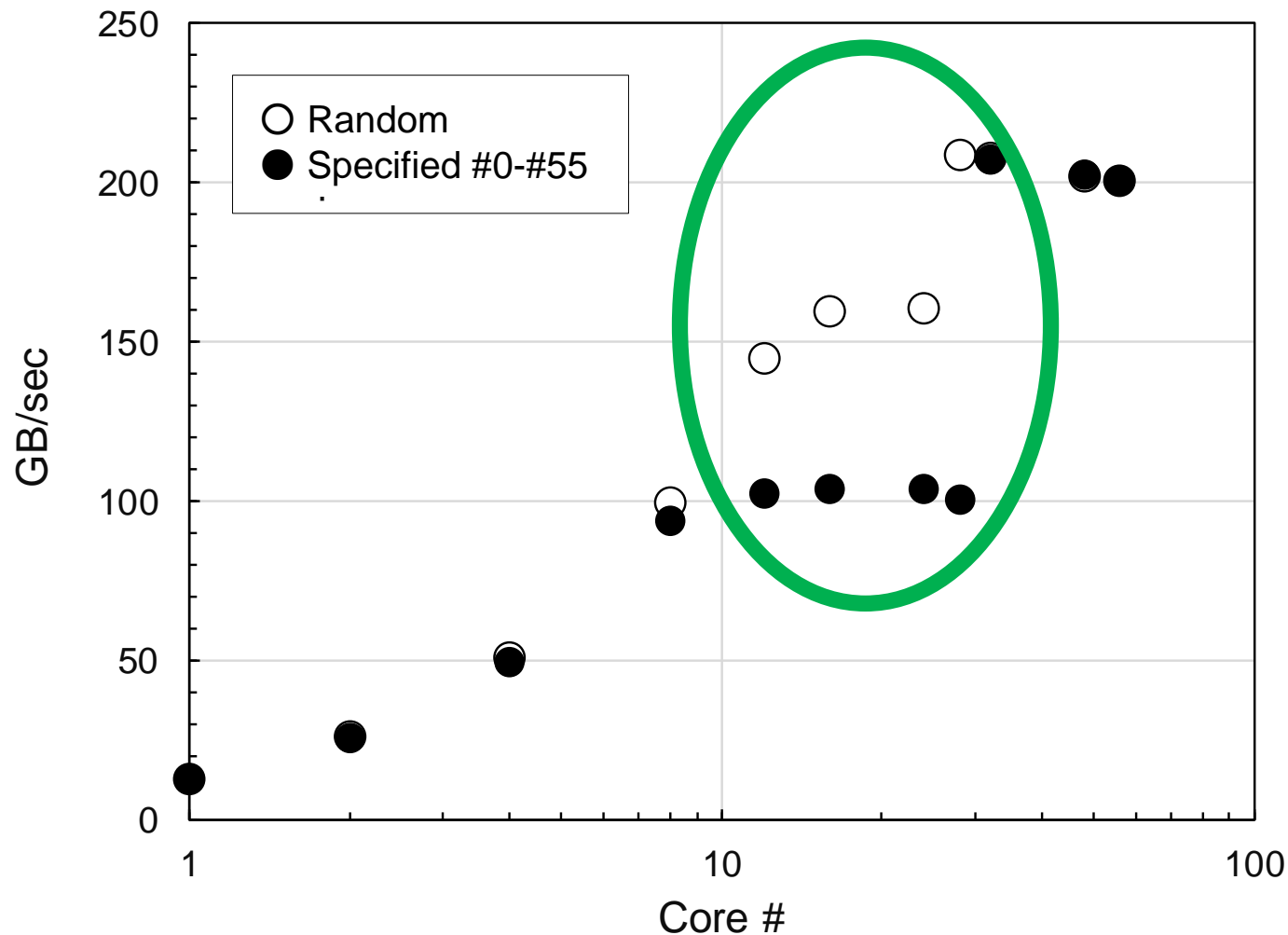
Core are specified



Triad on a Single Node of OBCX

Peak is 281.57 GB/sec.

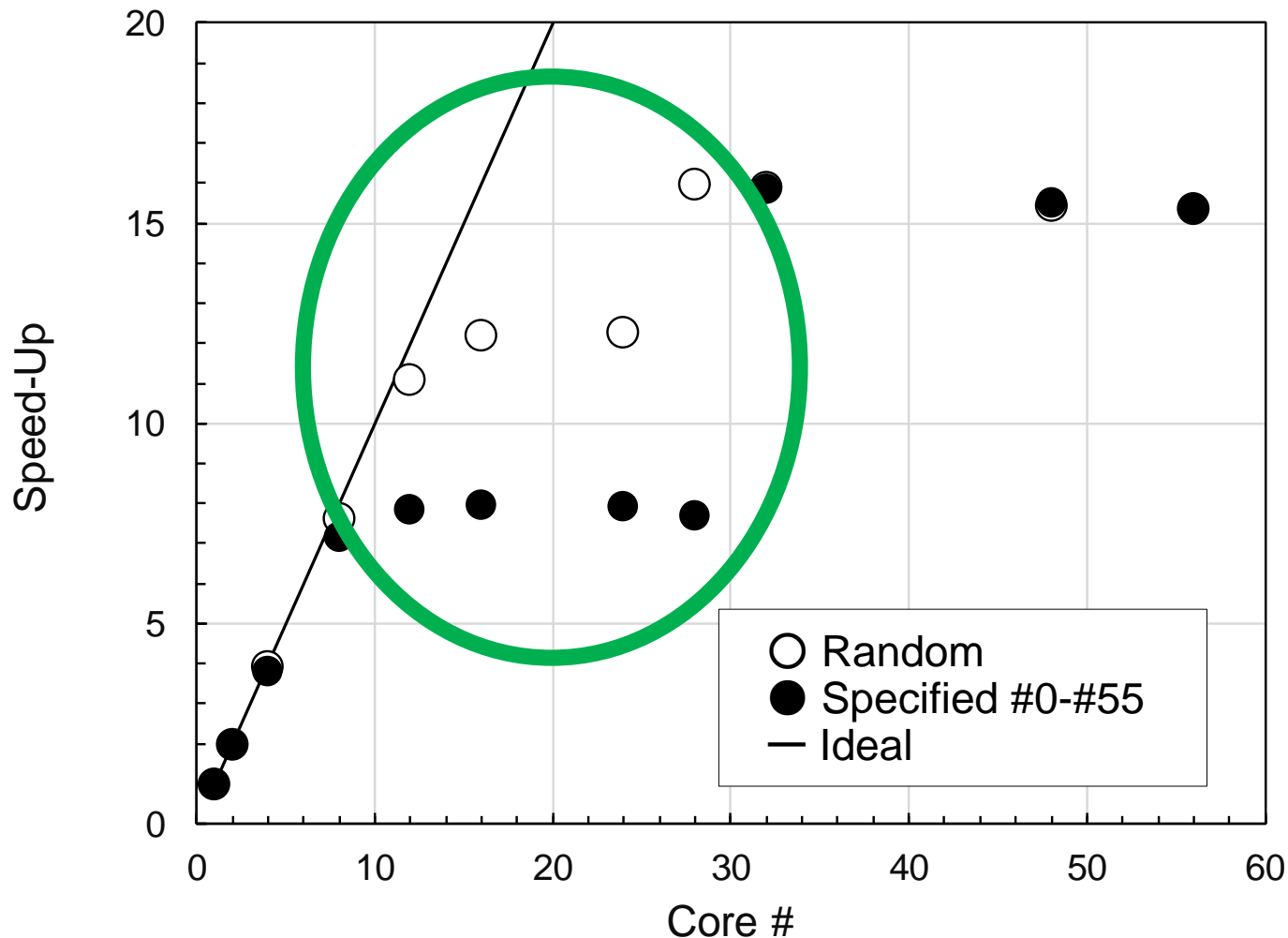
メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）



Triad on a Single Node of OBCX

Peak is 281.57 GB/sec.

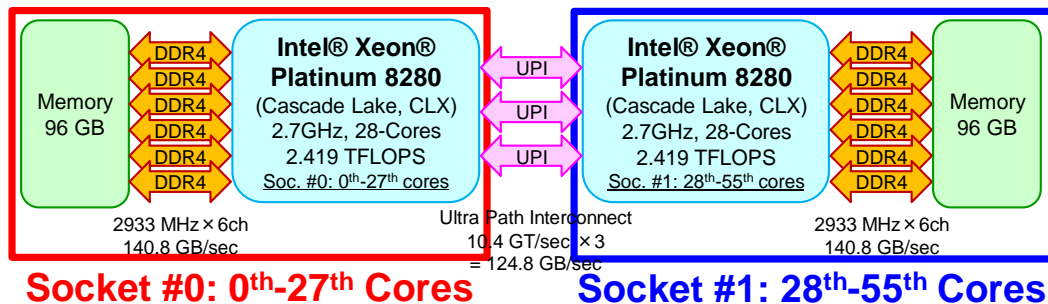
メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）



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

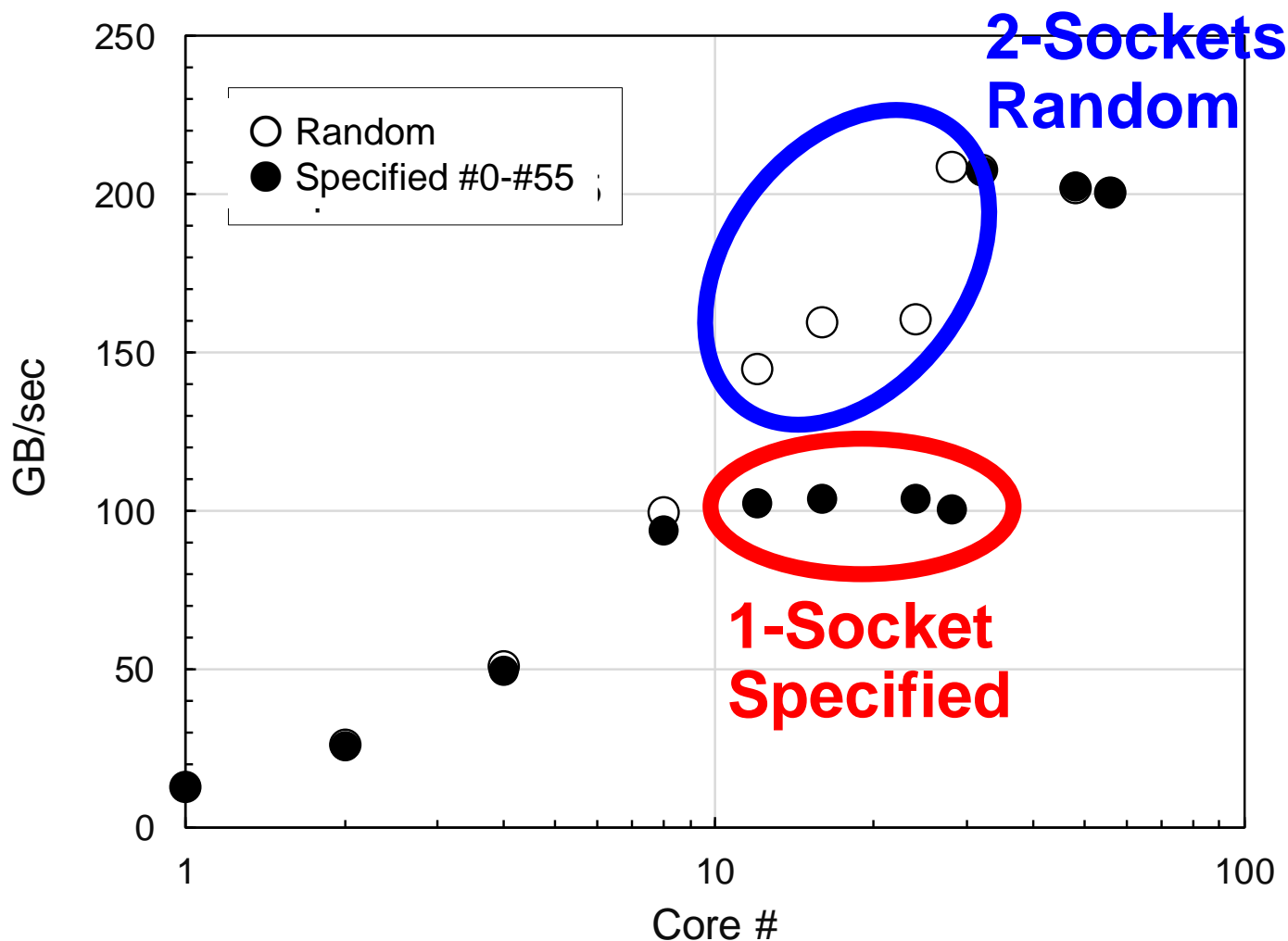


Socket #	Core #	Random	Specified
1	1	1.000	1.000
	2	1.998	1.963
	4	3.912	3.809
	6	5.792	5.682
	8	7.615	7.177
	12	11.089	7.844
	16	12.214	7.968
	24	12.278	7.945
2	28	15.962	7.705
	32	15.901	15.862
	48	15.452	15.497
	56	15.370	15.358

Triad on a Single Node of OBCX

Peak is 281.57 GB/sec.

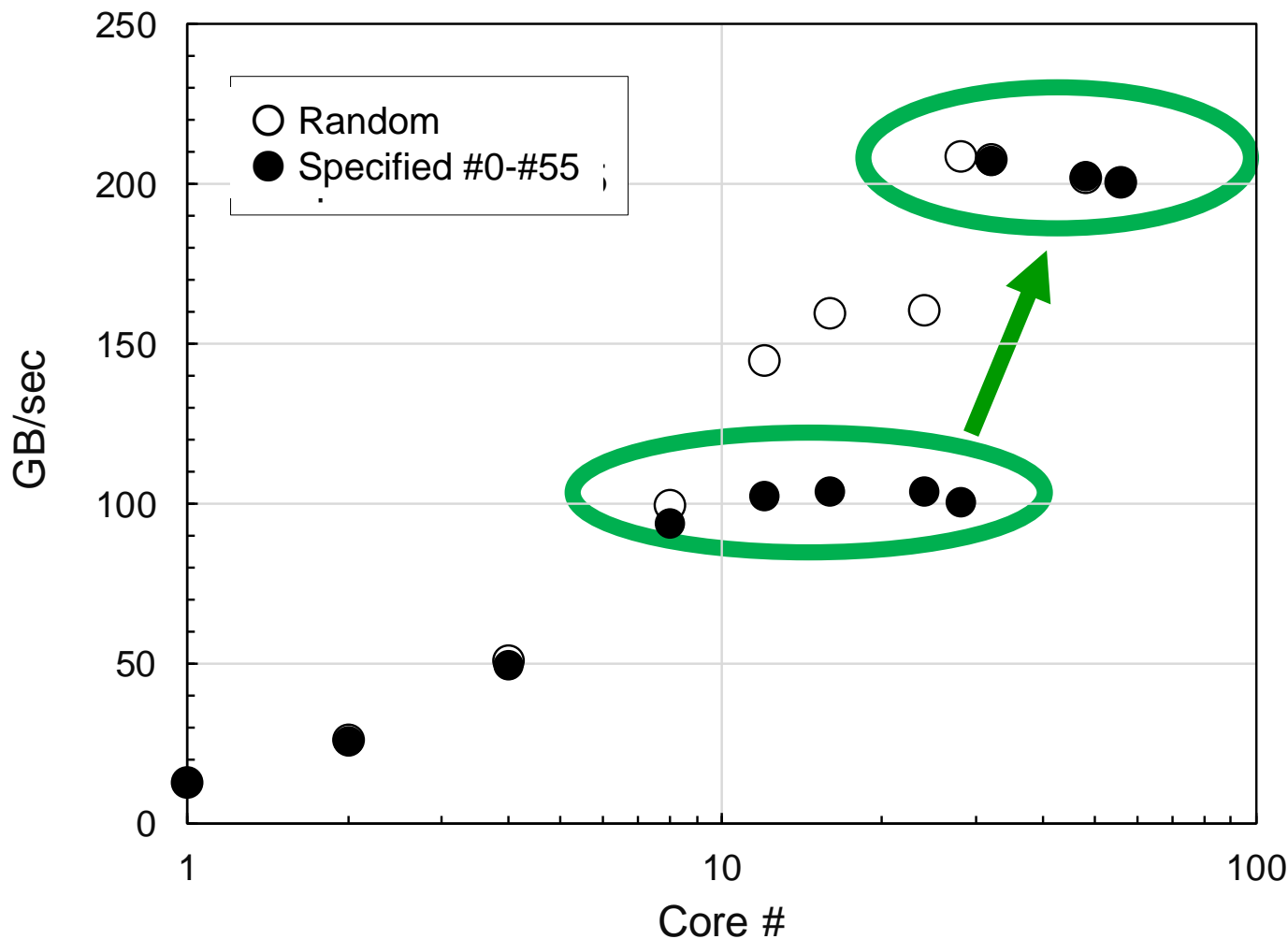
メモリ・キャッシュは「Random」の方がより効果的に利用されている（12-28コアの場合）



Triad on a Single Node of OBCX

Peak is 281.57 GB/sec.

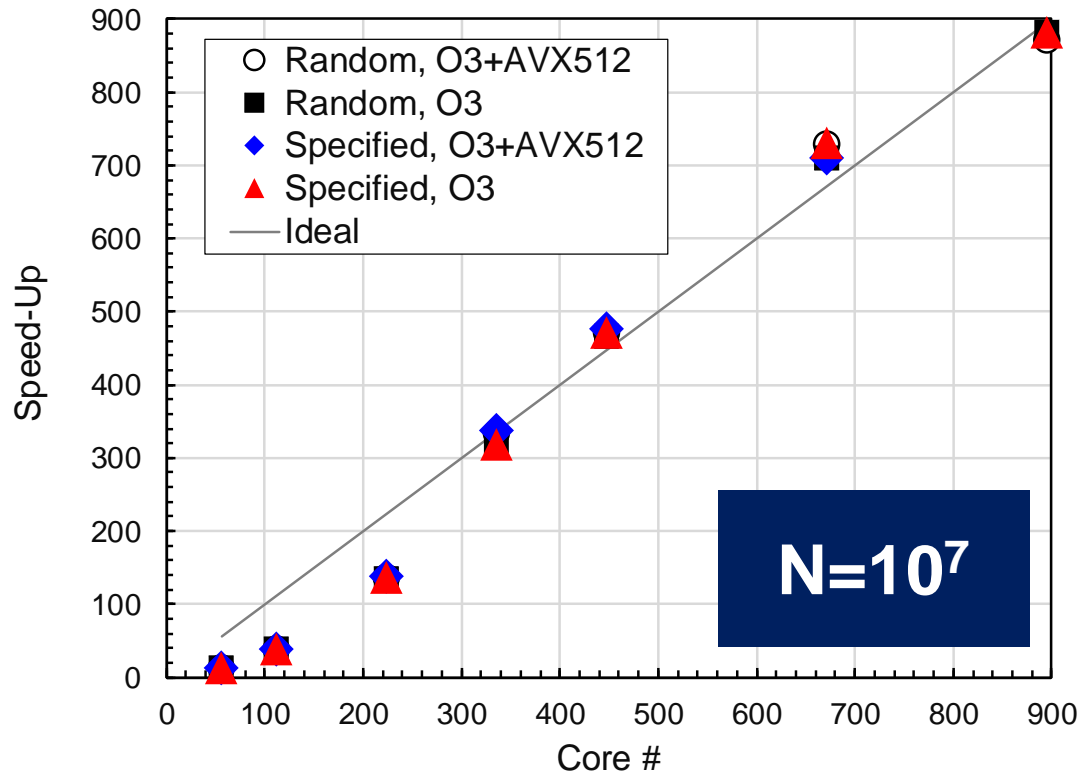
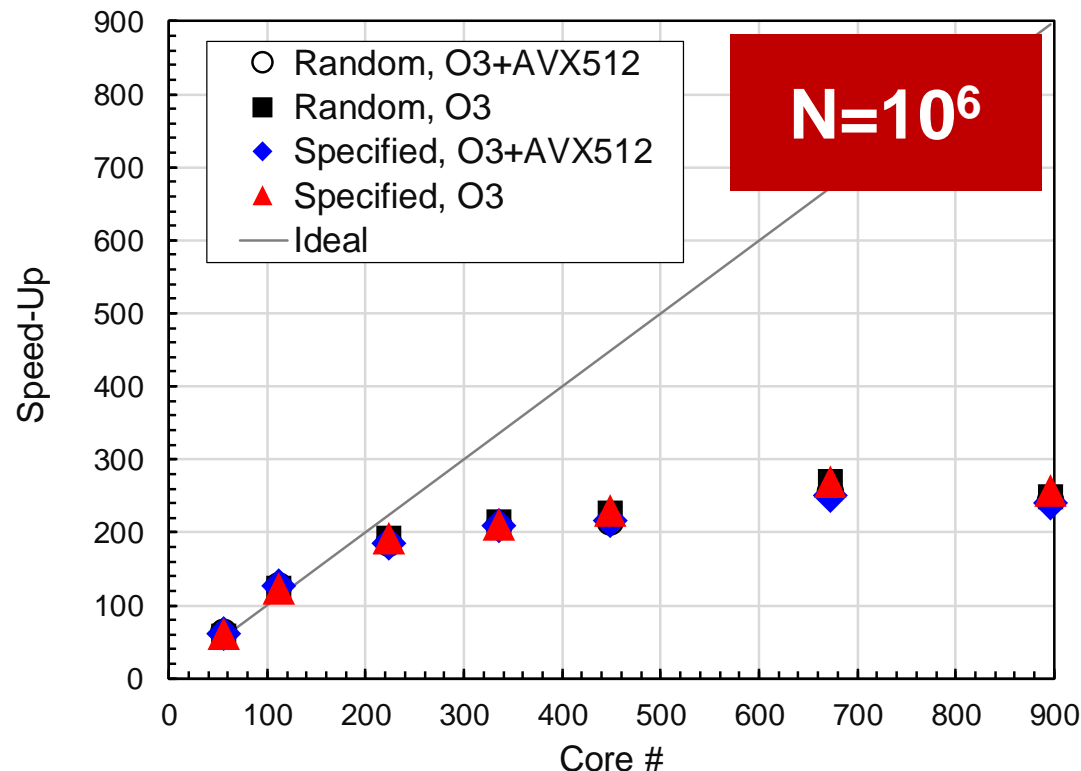
● : メモリバンド幅は8-28コアでほぼ一定（飽和）
32-56コアでは倍になる



Exercises

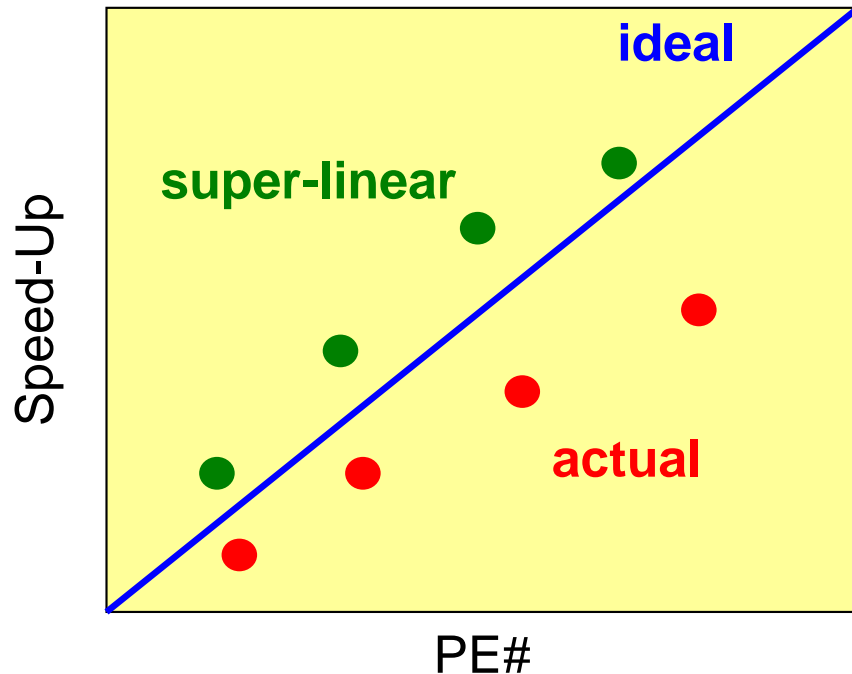
- Running the code
- Try various number of processes (1-56)
- OpenMP-version and Single PE version are available
 - Fortran, C
 - Web-site of STREAM
 - <http://www.cs.virginia.edu/stream/>

Up to 896 cores (16-nodes)



- Performance at a Single Core= 1.00
- ノード数が増すとN=10⁶のケースは効率低下（台形積分と同様）
- N=10⁷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」

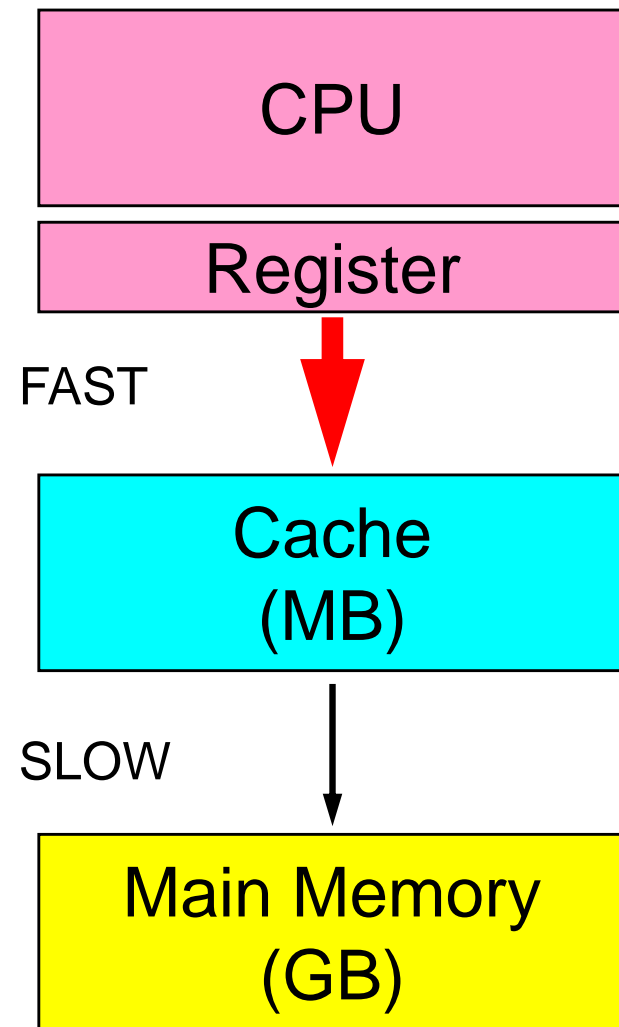
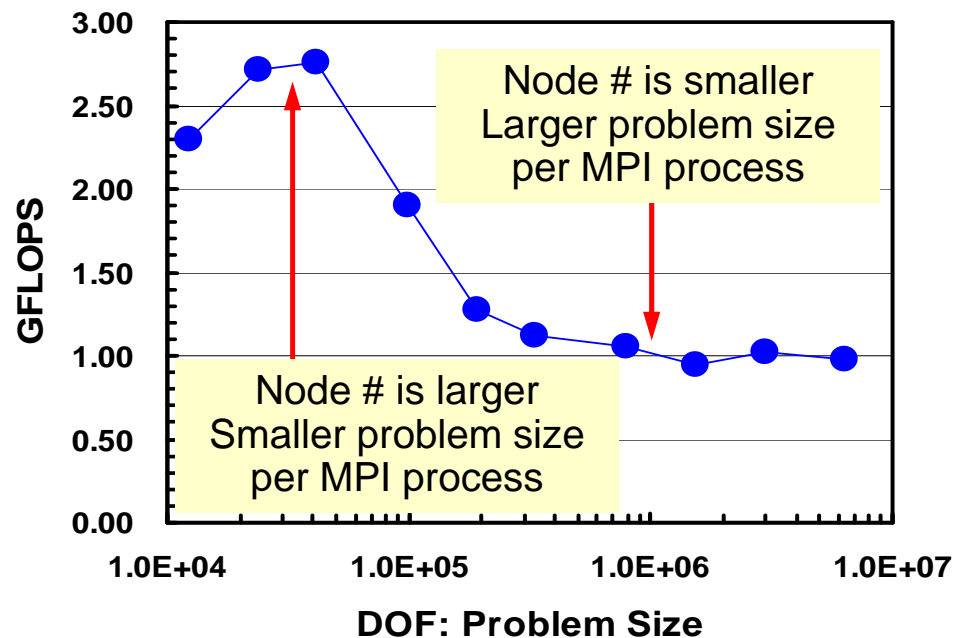


- 問題規模を固定して、使用PE数を増加させて行った場合、通常は通信の影響のために、効率は理想値（ m 個のPEを使用した場合、理想的には m 倍の性能になる）よりも低くなるのが普通である。
- しかし、スカラープロセッサ（PC等）の場合、逆に理想値よりも、高い性能が出る場合がある。このような現象を「Super-Linear」と呼ぶ。
 - ベクトル計算機では起こらない。

Super-Linearの生じる理由

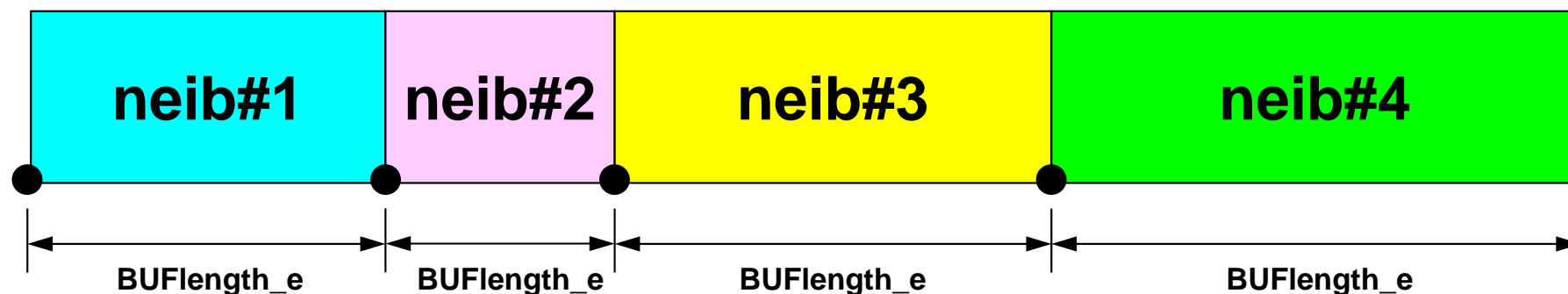
- キャッシュの影響
- スカラープロセッサでは、一般に問題規模が小さいほど性能が高い。

– キャッシュの有効利用



メモリーコピーも意外に時間かかる (1/2)

SENDbuf



export_index(0)+1 export_index(1)+1 export_index(2)+1 export_index(3)+1 export_index(4)

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

Copied to sending buffers

```
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)
```

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

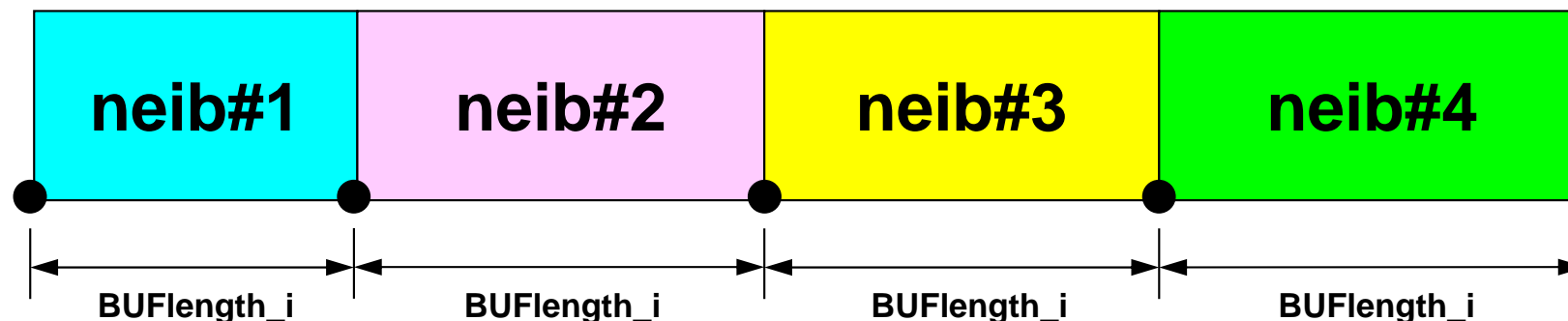
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

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

RECVbuf



import_index(0)+1 import_index(1)+1 import_index(2)+1 import_index(3)+1 import_index(4)

Up to 16-nodes Random, O3+AVX512

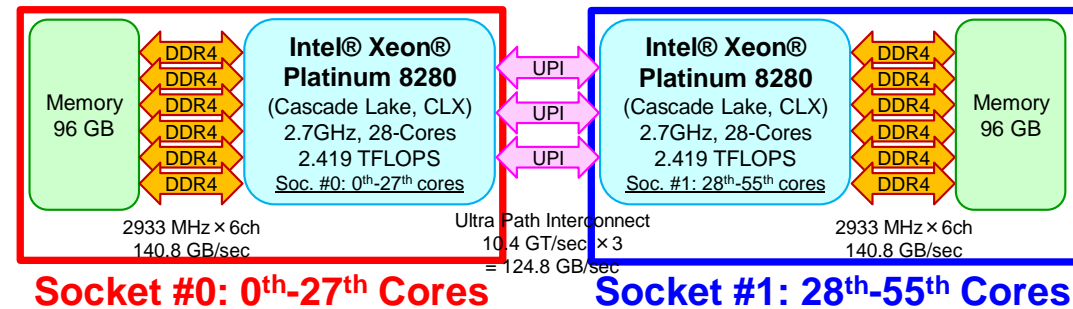
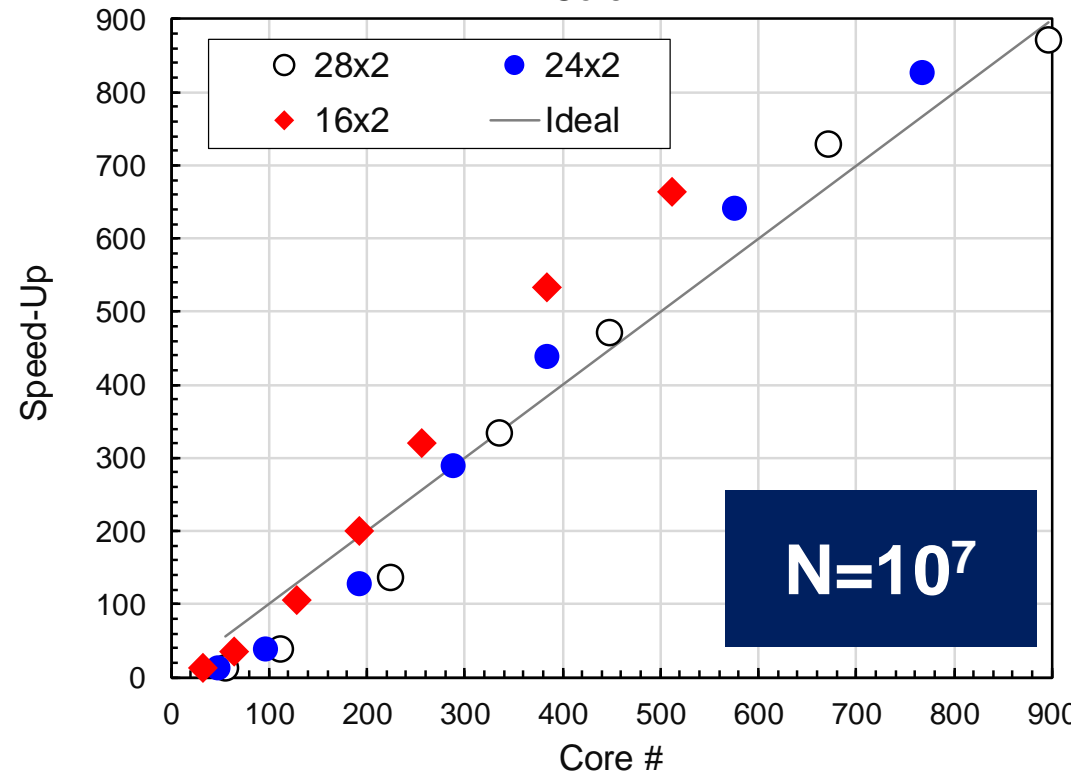
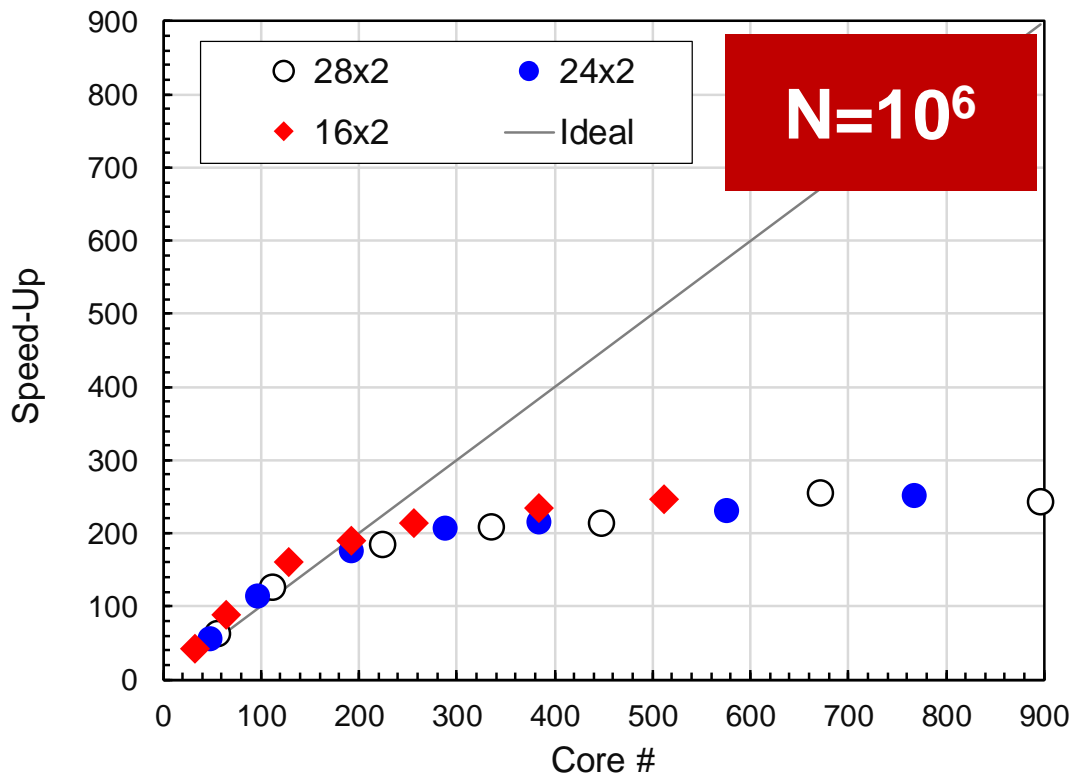
- MPI Processes per Node
 - 16x2, 24x2, 28x2

- $N=10^6$

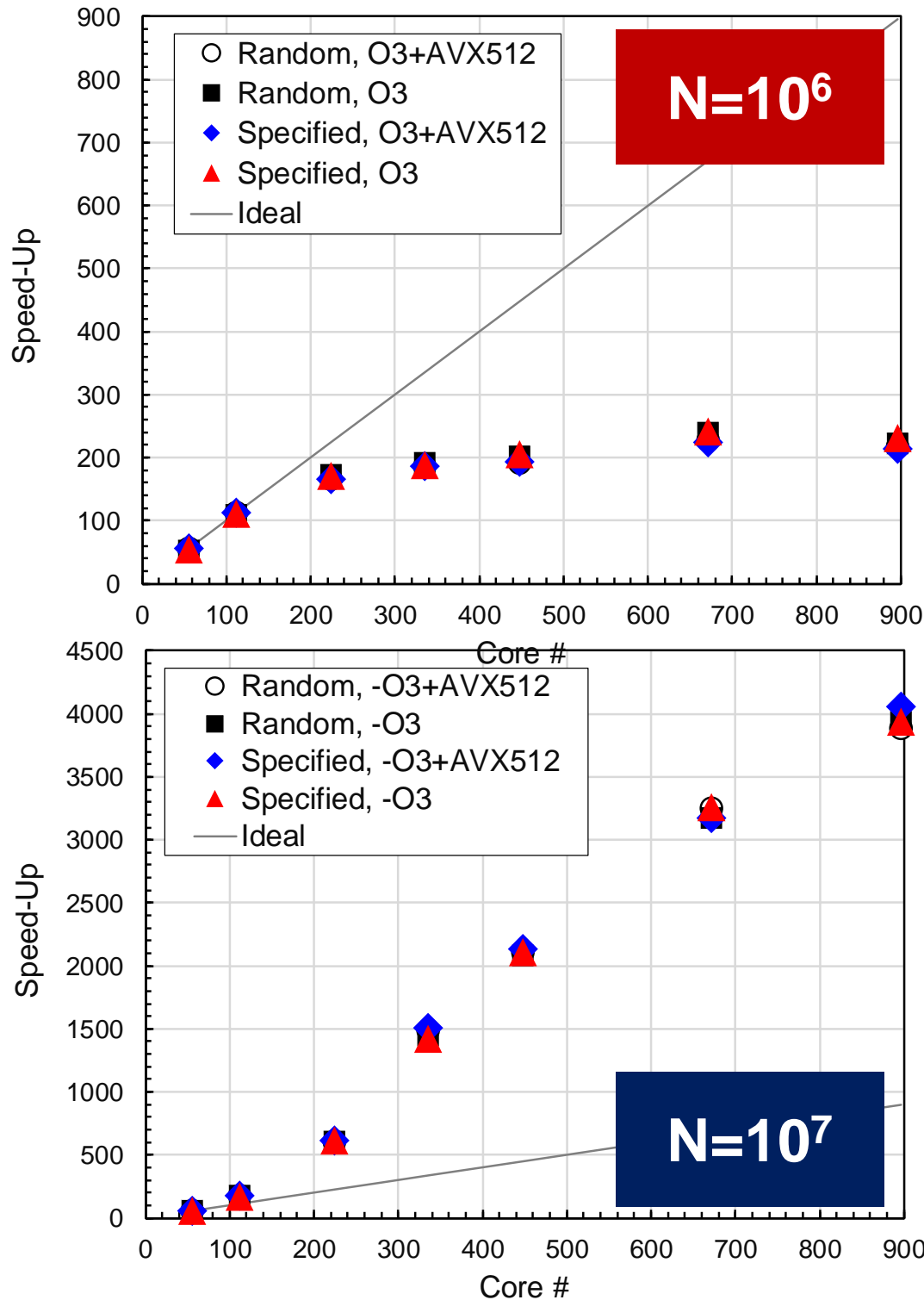
– ノード数・コア数が増えると
差異少ない

- $N=10^7$

– 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

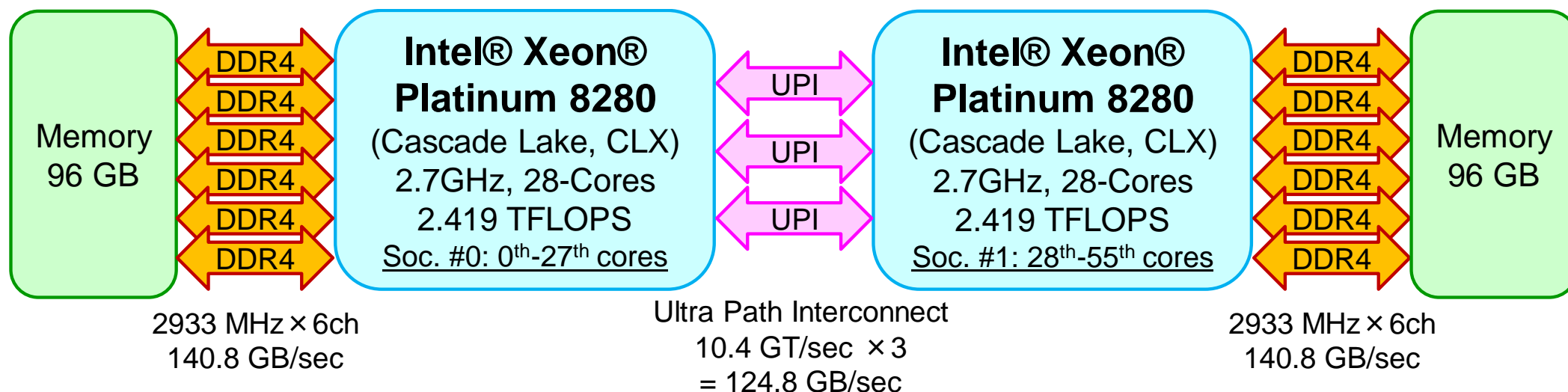


Up to 896 cores (16-nodes)



- 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=10⁶
 - 通信オーバーヘッド大
- N=10⁷
 - Superlinear強い

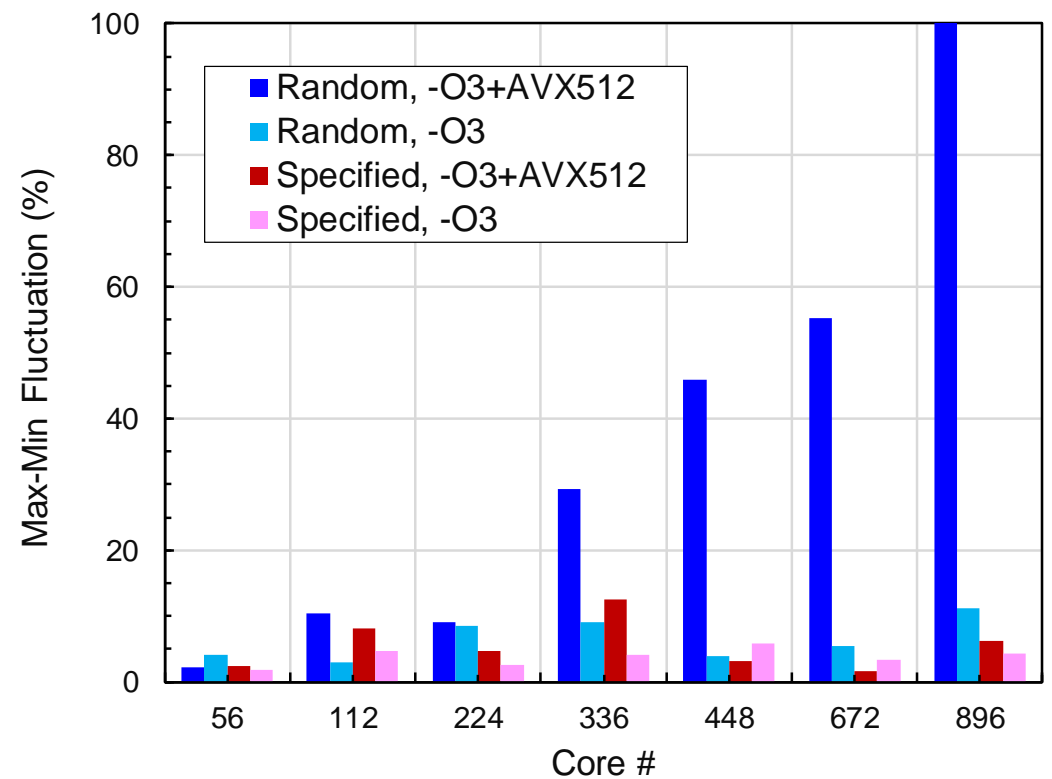
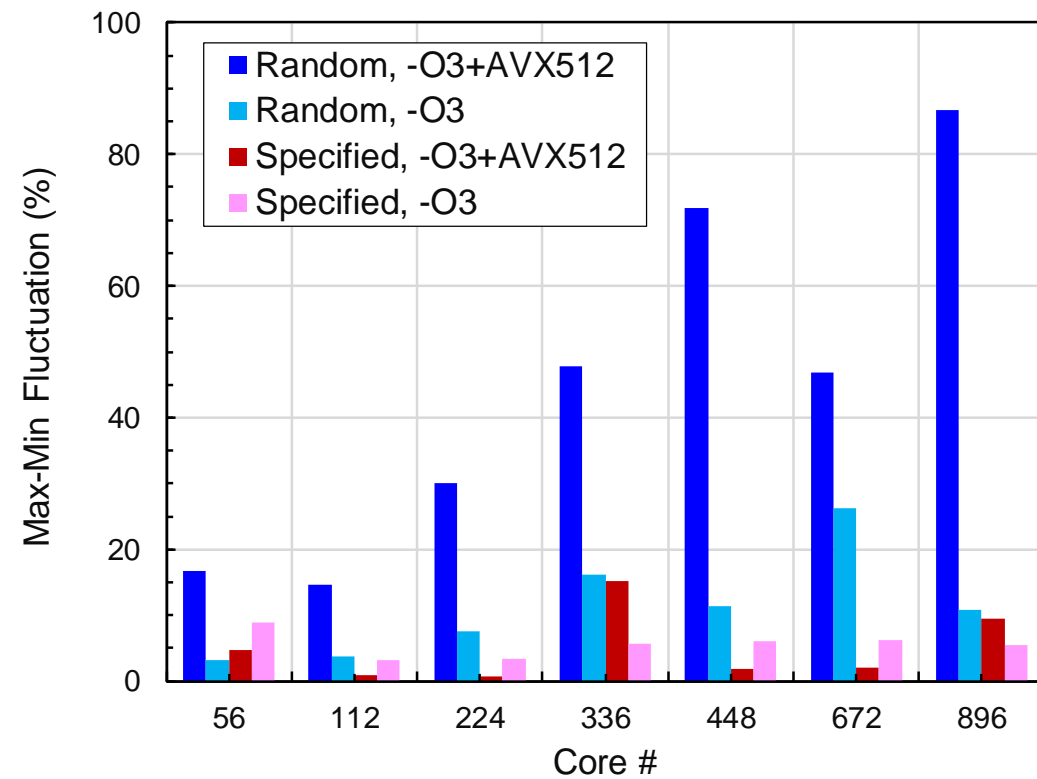
Category	Capacity	X-Way Set Associative	Cache Line
L1\$Data	32 KB/core	8-Way	64B
L1\$Instruction	32 KB/core	8-Way	64B
L2	1.00 MB/core	16-Way	64B
L3	38.5 MB/socket	11-Way	64B



5回の計測における変動

$N=10^6$

$N=10^6$



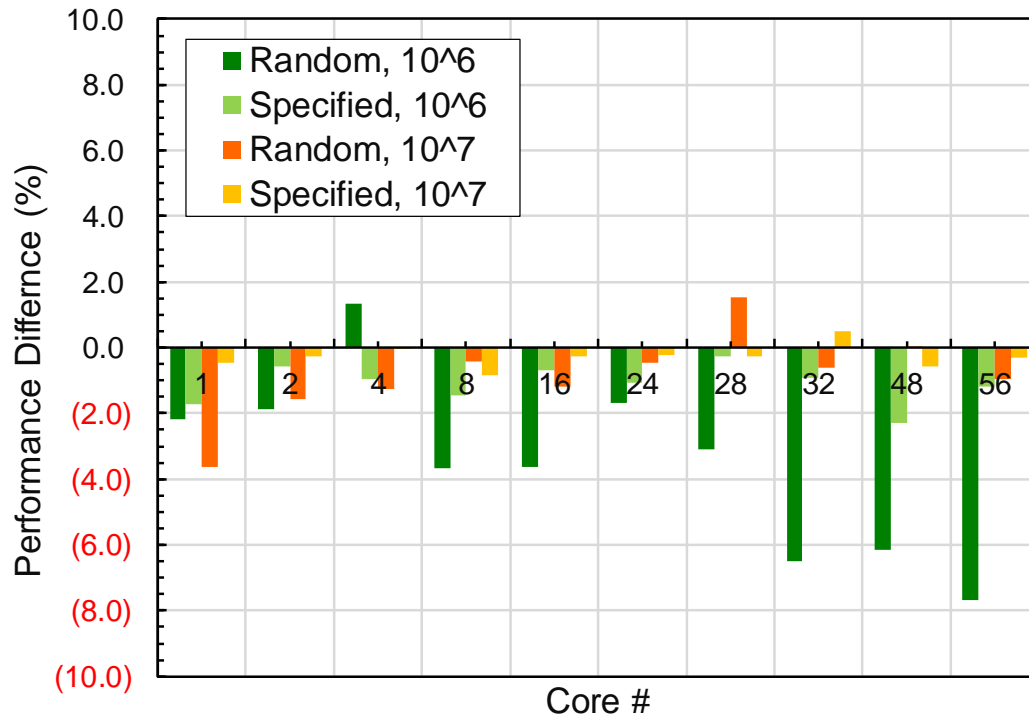
“-O3+AVX512” and “-O3”

Best Case of 5 Measurements

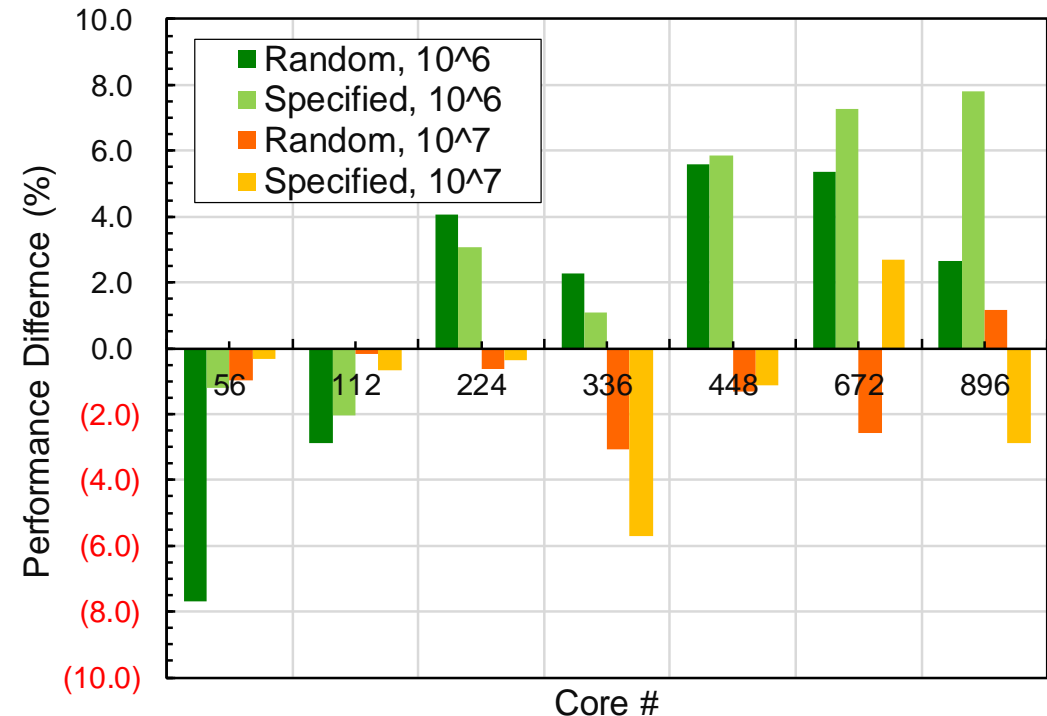
+: -O3 is better, -: -O3 is worse

“-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.f

```

allocate (stat_send(MPI_STATUS_SIZE, NEIBPETOT))
allocate (stat_recv(MPI_STATUS_SIZE, NEIBPETOT))
allocate (request_send(NEIBPETOT)); allocate (request_recv(NEIBPETOT))

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

do neib= 1, NEIBPETOT
  is = export_index(neib-1) + 1
  len_s= export_index(neib) - export_index(neib-1)
  call MPI_Isend (SENDbuf(is), len_s, MPI_DOUBLE_PRECISION,      &
&                NEIBPE(neib), 0, MPI_COMM_WORLD,                &
&                request_send(neib), ierr)
enddo

do neib= 1, NEIBPETOT
  ir = import_index(neib-1) + 1
  len_r= import_index(neib) - import_index(neib-1)
  call MPI_Irecv (RECVbuf(ir), len_r, MPI_DOUBLE_PRECISION,      &
&                NEIBPE(neib), 0, MPI_COMM_WORLD,                &
&                request_recv(neib), ierr)
enddo
call MPI_Waitall (NEIBPETOT, request_recv, stat_recv, ierr)

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

```

[A]{p} = {q} (Mod.): 受信でコピー不要 1d2.f

```

allocate (stat_send(MPI_STATUS_SIZE, 2*NEIBPETOT))
allocate (request_send(2*NEIBPETOT))

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

do neib= 1, NEIBPETOT
  is = export_index(neib-1) + 1
  len_s= export_index(neib) - export_index(neib-1)
  call MPI_Isend (SENDbuf(is), len_s, MPI_DOUBLE_PRECISION,      &
&                NEIBPE(neib), 0, MPI_COMM_WORLD,                &
&                request_send(neib), ierr)
enddo

do neib= 1, NEIBPETOT
  ir = import_index(neib-1) + 1
  len_r= import_index(neib) - import_index(neib-1)
  call MPI_Irecv (W(ir+N, P), len_r, MPI_DOUBLE_PRECISION,      &
&               NEIBPE(neib), 0, MPI_COMM_WORLD,                &
&               request_send(neib+NEIBPETOT), ierr)
enddo

call MPI_Waitall (2*NEIBPETOT, request_send, stat_send, ierr)

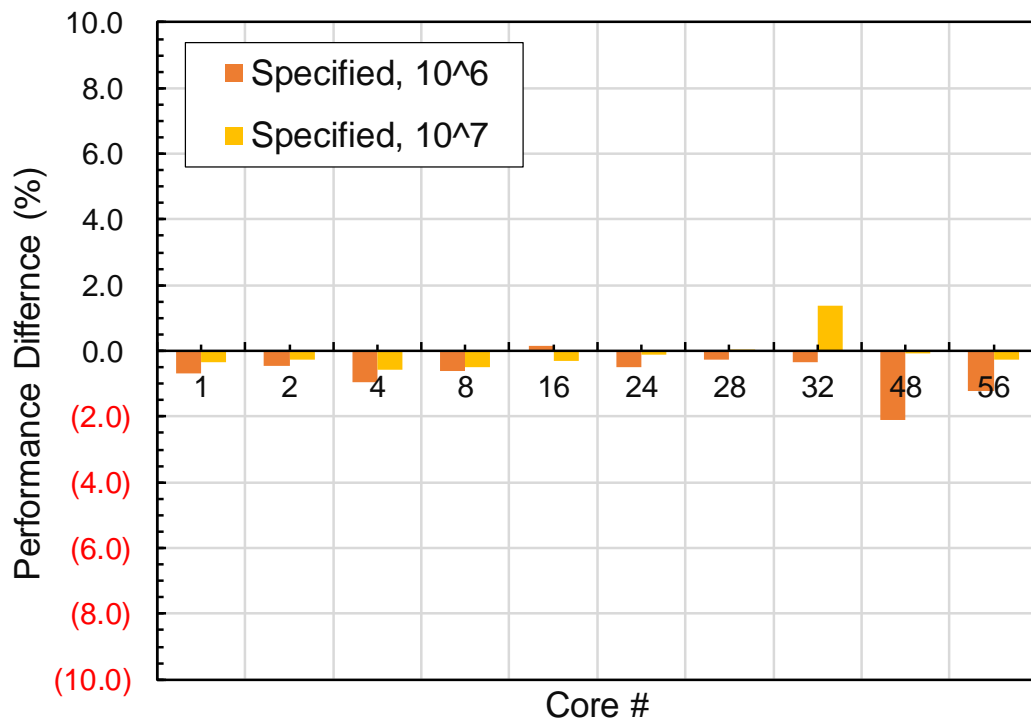
```

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

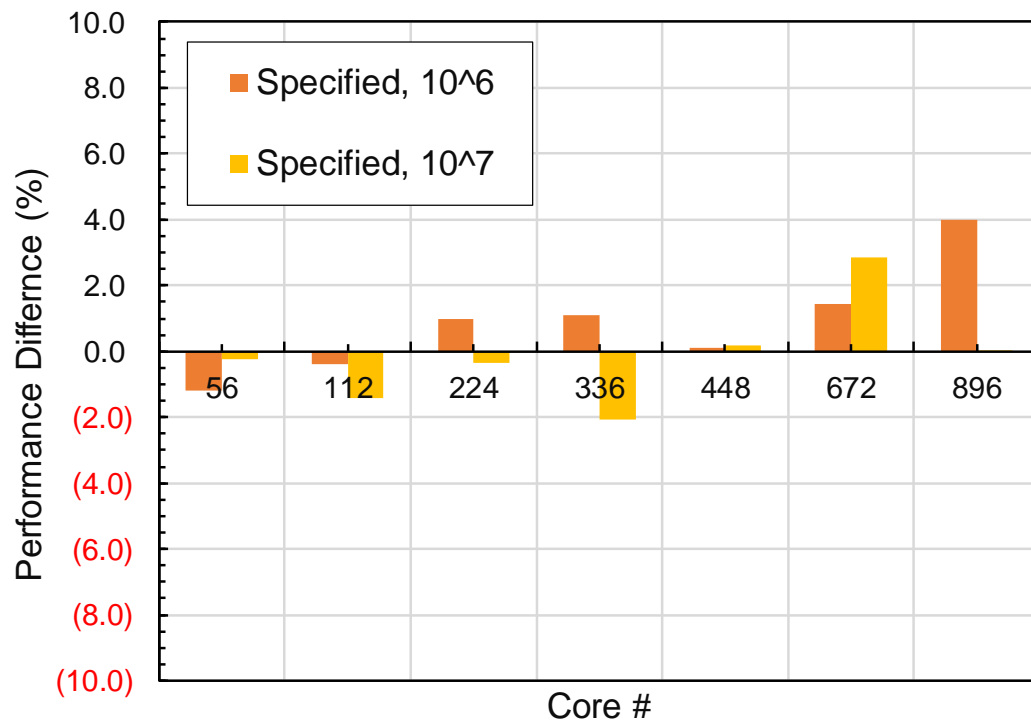
+: “Modified” is better, -: “Modified” is worse

“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