

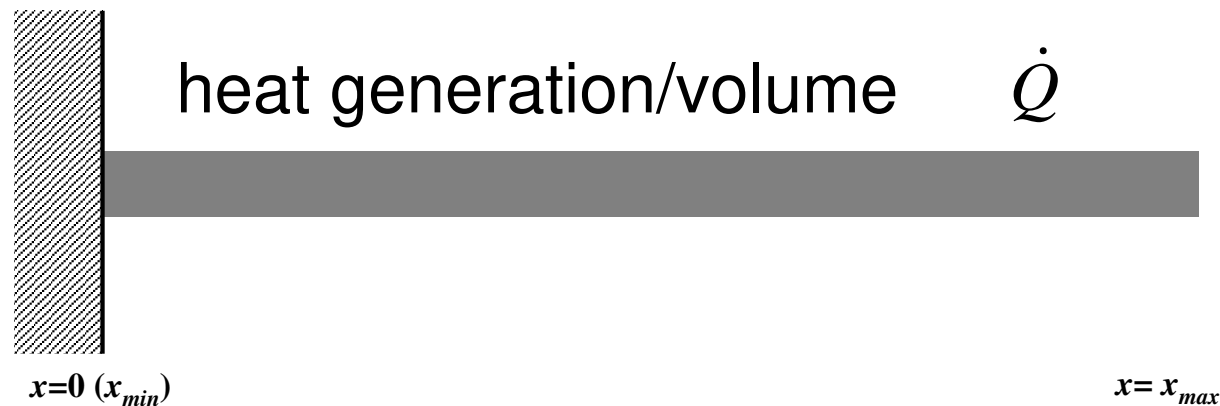
Report S2

Fortran

Kengo Nakajima
Information Technology Center
The University of Tokyo

- Overview
- Distributed Local Data
- Program
- Results

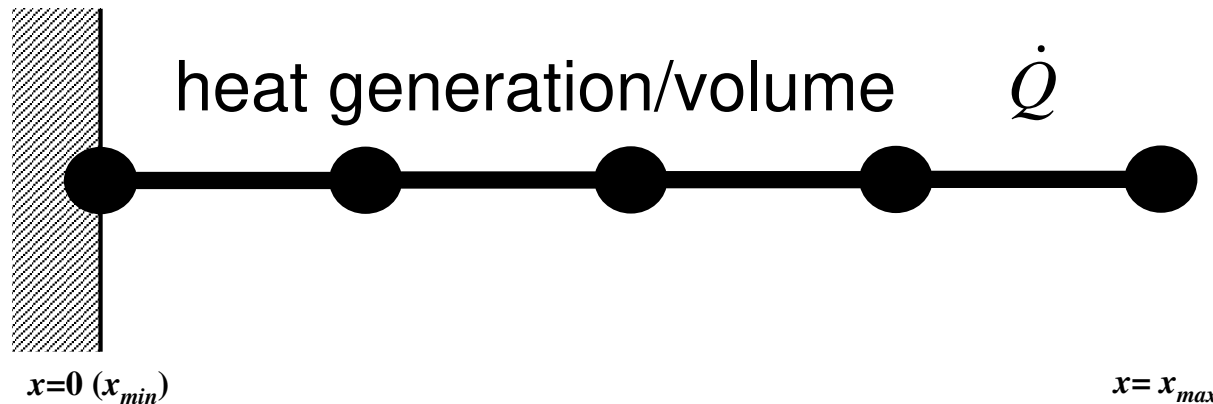
1D Steady State Heat Conduction



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

- **Uniform: Sectional Area: A , Thermal Conductivity: λ**
- Heat Generation Rate/Volume/Time [$QL^{-3}T^{-1}$] $\dot{Q}$
- Boundary Conditions
 - $x=0$: $T=0$ (Fixed Temperature)
 - $x=x_{max}$: $\frac{\partial T}{\partial x} = 0$ (Insulated)

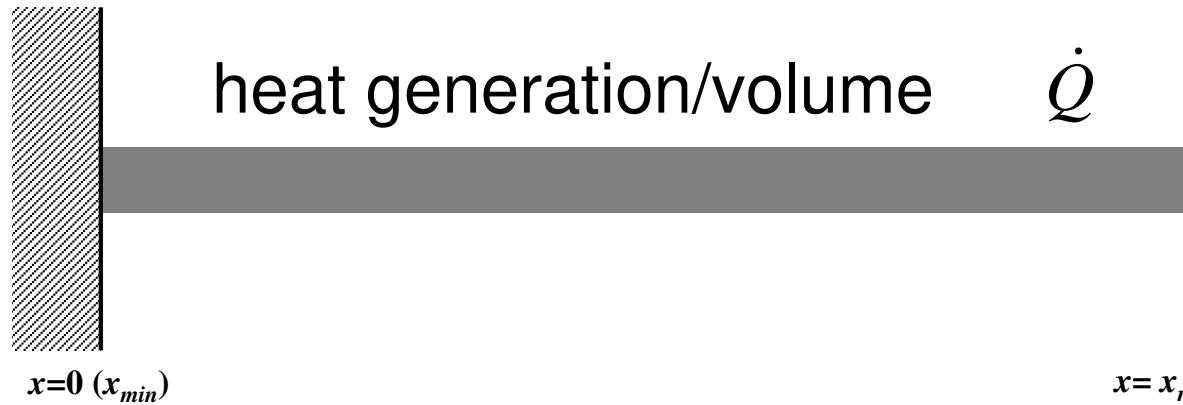
1D Steady State Heat Conduction



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

- **Uniform: Sectional Area: A , Thermal Conductivity: λ**
- Heat Generation Rate/Volume/Time [$QL^{-3}T^{-1}$] $\dot{Q}$
- Boundary Conditions
 - $x=0$: $T=0$ (Fixed Temperature)
 - $x=x_{max}$: $\frac{\partial T}{\partial x} = 0$ (Insulated)

Analytical Solution



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

$$T = 0 @ x = 0$$

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

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

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

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

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

Report S2 (1/2)

- Parallelize 1D code (1d.f) using MPI
- Read entire element number, and decompose into sub-domains in your program
- Validate the results
 - Answer of Original Code = Answer of Parallel Code
 - Explain why number of iterations does not change, as number of MPI processes changes.
- Measure parallel performance

Report S2 (2/2)

- Deadline: January 26th (Wed), 2022, 17:00@ITC-LMS
- Problem
 - Apply “Generalized Communication Table”
 - Read entire elem. #, decompose into sub-domains in your program
 - Evaluate parallel performance
 - You need huge number of elements, to get excellent performance.
 - Fix number of iterations (e.g. 100), if computations cannot be completed.
- Report
 - Cover Page: Name, ID, and Problem ID (S2) must be written.
 - Less than eight pages including figures and tables (A4).
 - Strategy, Structure of the Program, Remarks
 - Source list of the program (if you have bugs)
 - Output list (as small as possible)

Copy and Compile

Control File: input.dat

Control Data input.dat

10000000

1.0 1.0 1.0 1.0

100

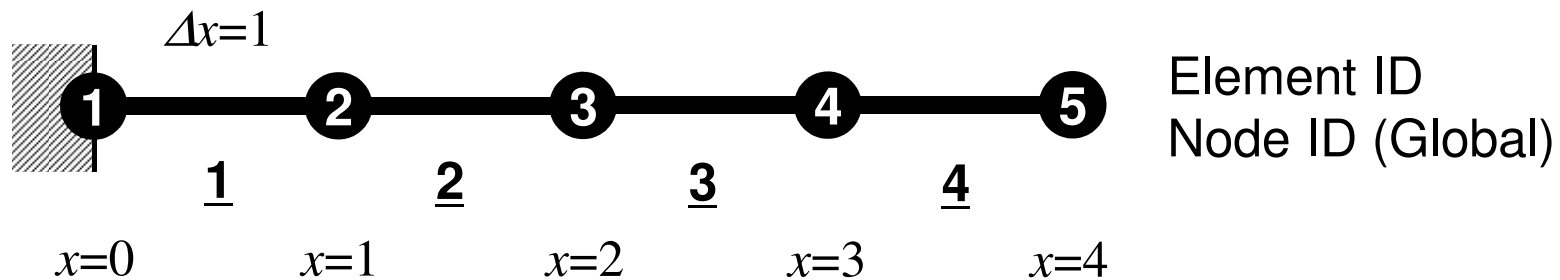
1.e-8

NE (Number of Elements)

Δx (Length of Each Elem.: L) , Q , A , λ

Number of MAX. Iterations for CG Solver

Convergence Criteria for CG Solver



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

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=8
#PJM -mpi proc=256
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst
```

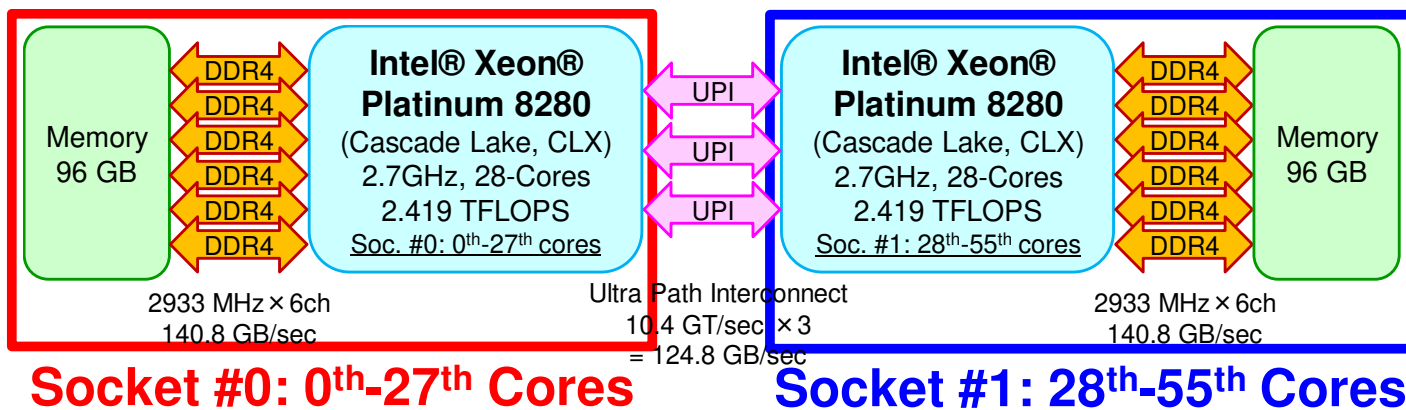
```
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
export I_MPI_PIN_PROCESSOR_LIST=0-15,28-43
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
```

256/8= 32 cores/node

1da: -O3 + AVX512
1db: -O3 Only

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

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



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

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=8
#PJM --mpi proc=384
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst
```

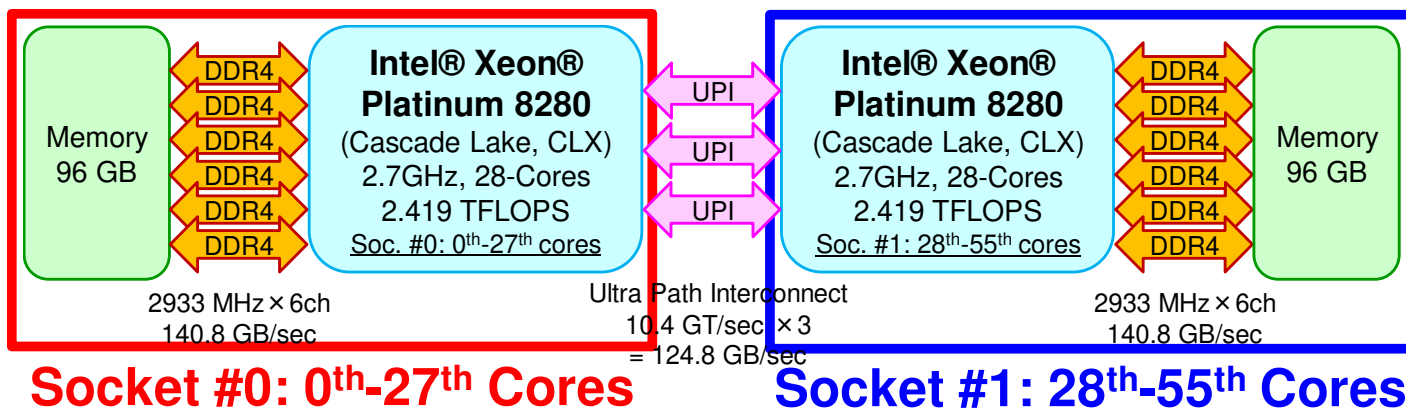
```
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
export I_MPI_PIN_PROCESSOR_LIST=0-23,28-51
mpiexec.hydra -n ${PJM_MPI_PROC} ./1da
mpiexec.hydra -n ${PJM_MPI_PROC} ./1db
```

384/8= 48 cores/node

1da: -O3 + AVX512
1db: -O3 Only

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

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



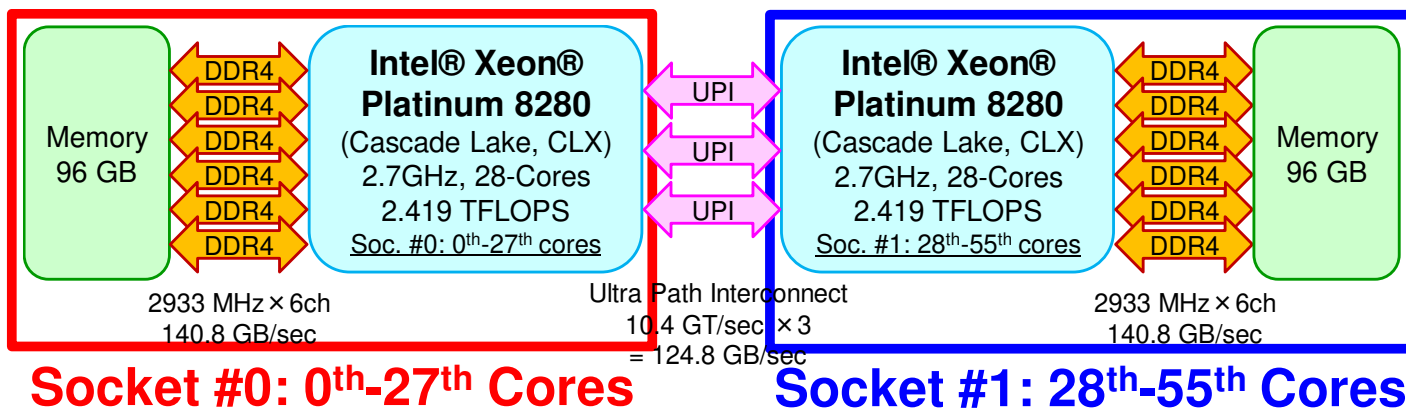
g28.sh: 8-nodes, 448-cores, 28x2x8

```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=8
#PJM --mpi proc=448
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst
```

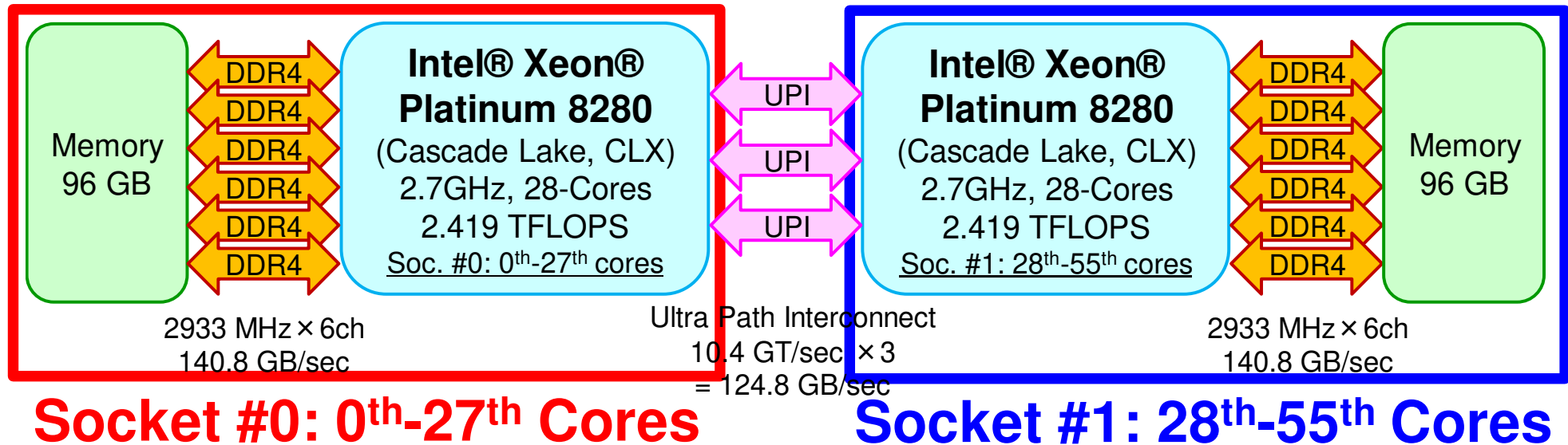
448/8= 56 cores/node

1da: -O3 + AVX512
1db: -O3 Only

```
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



- Each Node of Oakbridge-CX (OBCX)
 - 2 Sockets (CPU's) of Intel CLX
 - Each socket has 28 cores
- Each core of a socket can access to the memory on the other socket : NUMA (Non-Uniform Memory Access)
 - `numactl -l` : local memory to be used
 - Sometimes (not always), more stable with this option

Procedures for Parallel FEM

- Reading control file, entire element number etc.
- Creating “distributed local data” in the program
- Assembling local and global matrices for linear solvers
- Solving linear equations by CG
- Not so different from those of original code

- Overview
- **Distributed Local Data**
- Program
- Results

Finite Element Procedures

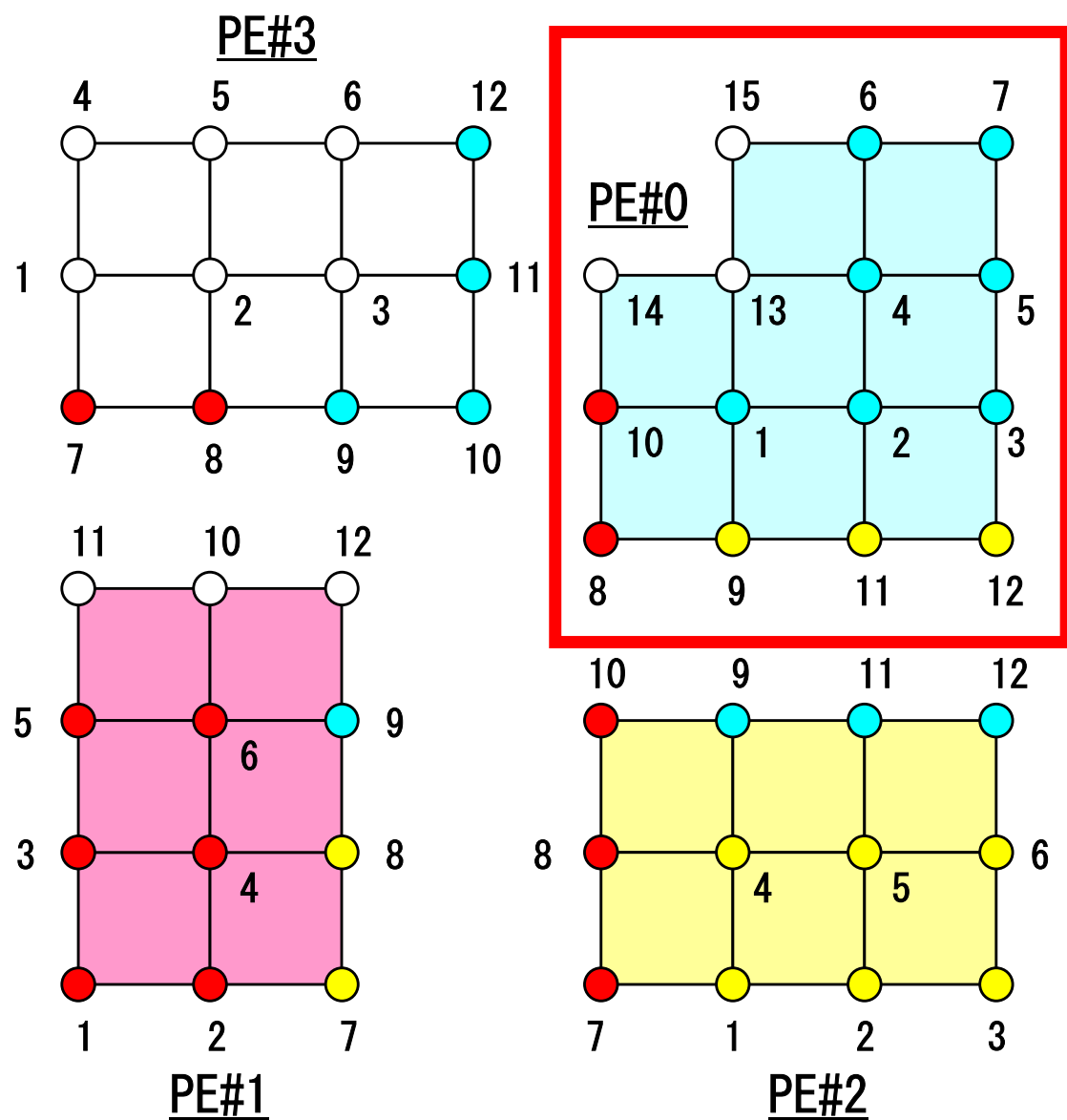
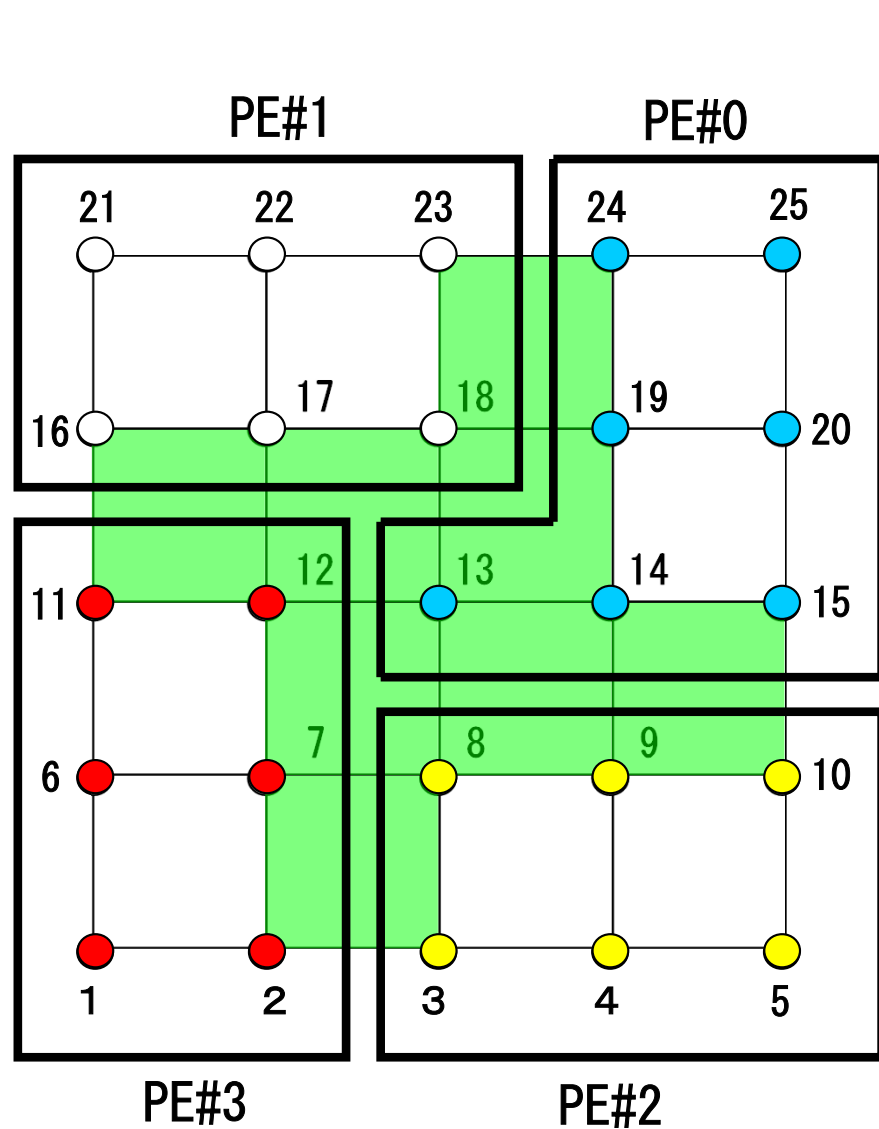
- Initialization
 - Control Data
 - Node, Connectivity of Elements (N: Node#, NE: Elem#)
 - Initialization of Arrays (Global/Element Matrices)
 - Element-Global Matrix Mapping (Index, Item)
- Generation of Matrix
 - Element-by-Element Operations (do icel= 1, NE)
 - Element matrices
 - Accumulation to global matrix
 - Boundary Conditions
- Linear Solver
 - Conjugate Gradient Method

Distributed Local Data Structure for Parallel FEM

- **Node-based partitioning**
- Local data includes:
 - Nodes originally assigned to the domain/PE/partition
 - Elements which include above nodes
 - Nodes which are included above elements, and originally NOT-assigned to the domain/PE/partition
- 3 categories for nodes
 - **Internal nodes** Nodes originally assigned to the domain/PE/partition
 - **External nodes** Nodes originally NOT-assigned to the domain/PE/partition
 - **Boundary nodes** External nodes of other domains/PE's/partitions
- Communication tables
- Global info. is not needed except relationship between domains
 - Property of FEM: local element-by-element operations

Node-based Partitioning

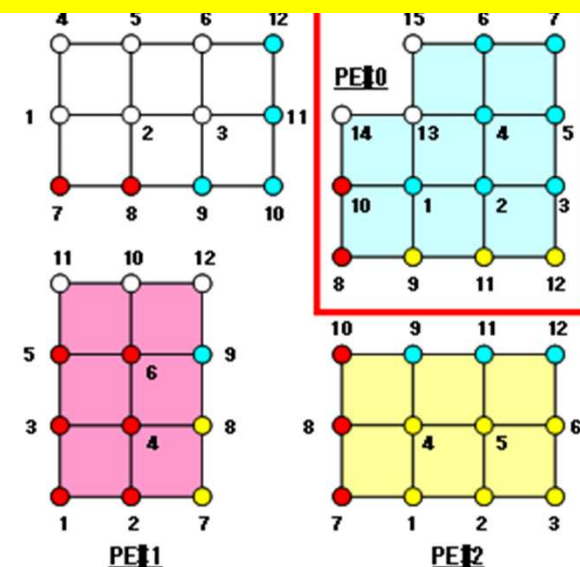
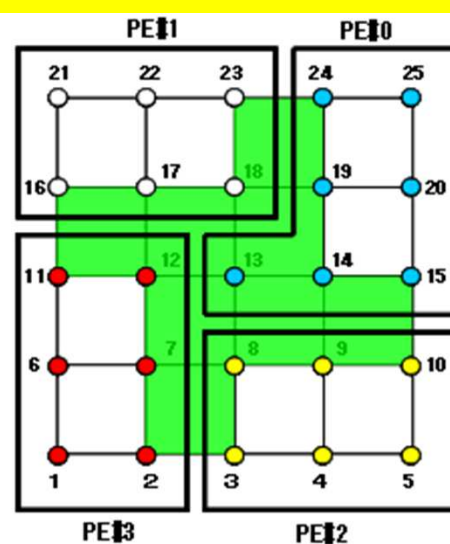
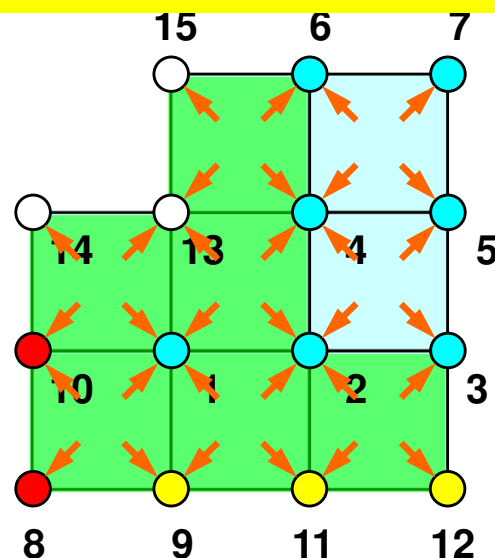
internal nodes - elements - external nodes



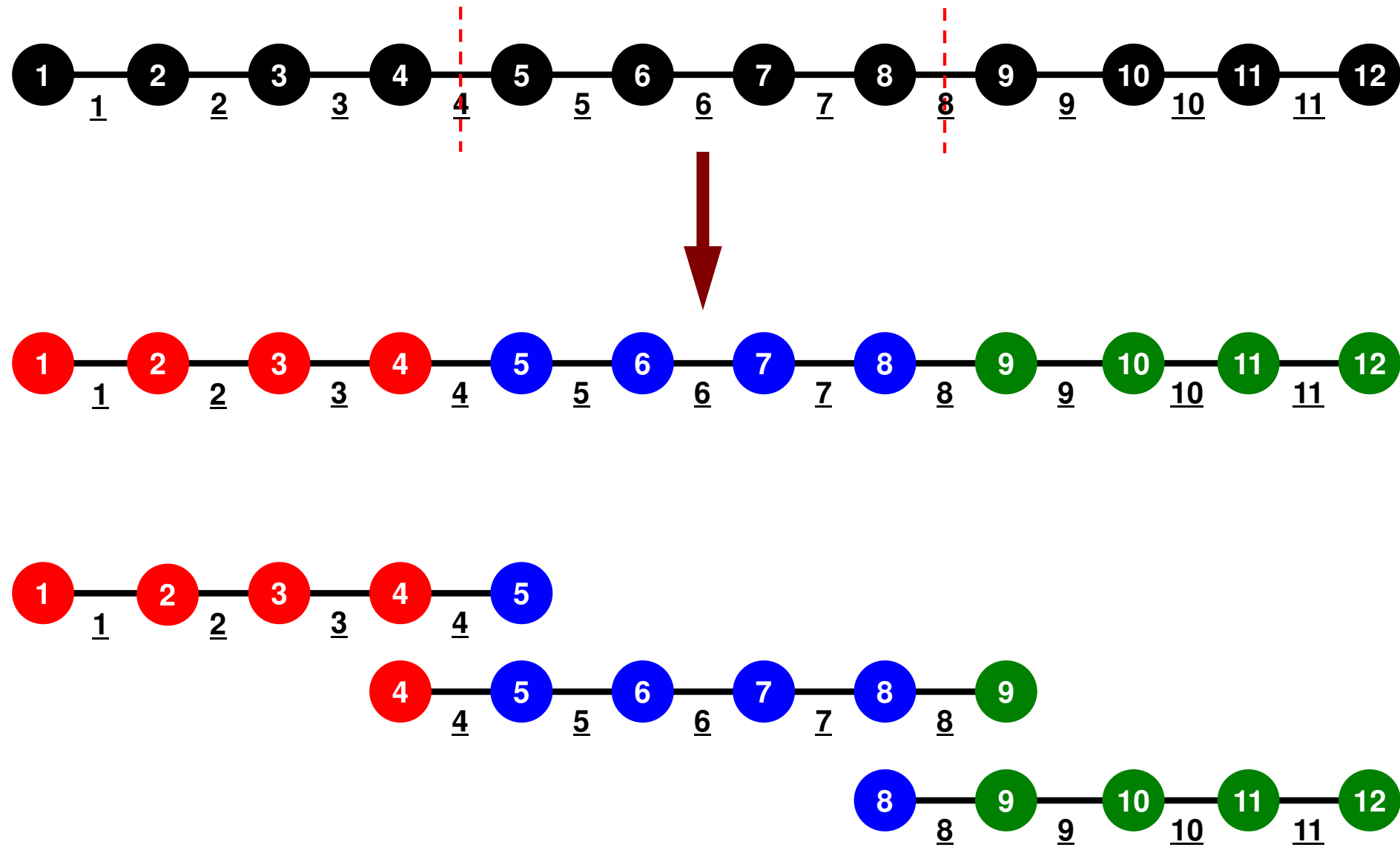
Node-based Partitioning

internal nodes - elements - external nodes

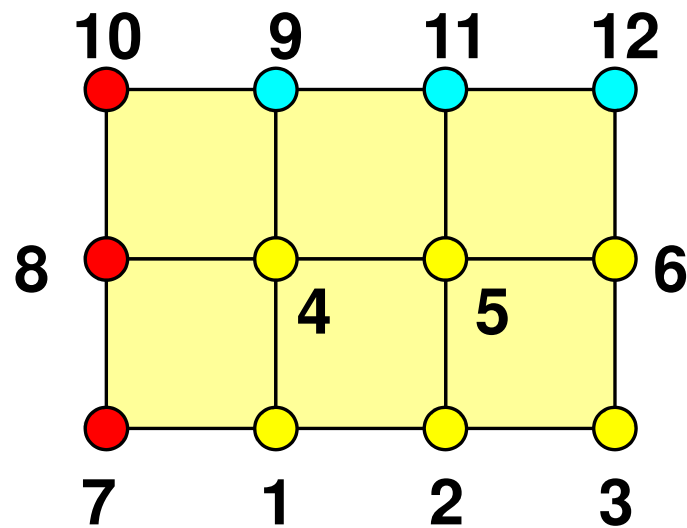
- Partitioned nodes themselves (Internal Nodes) 内点
- Elements which include Internal Nodes 内点を含む要素
- External Nodes included in the Elements 外点
in overlapped region among partitions.
- Info of External Nodes are required for completely local element-based operations on each processor.



1D FEM: 12 nodes/11 elem's/3 domains



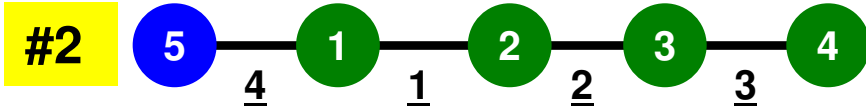
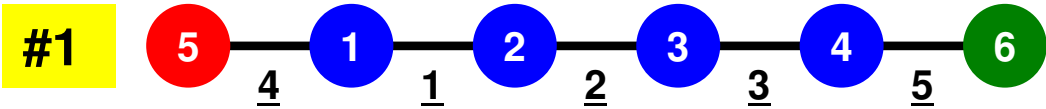
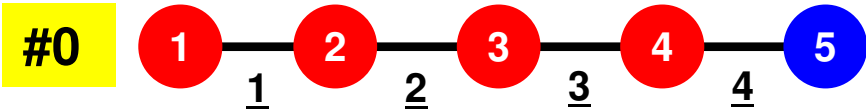
Description of Distributed Local Data



- Internal/External Points
 - Numbering: Starting from internal pts, then external pts after that
- Neighbors
 - Shares overlapped meshes
 - Number and ID of neighbors
- External Points
 - From where, how many, and which external points are received/imported ?
- Boundary Points
 - To where, how many and which boundary points are sent/exported ?

1D FEM: 12 nodes/11 elem's/3 domains

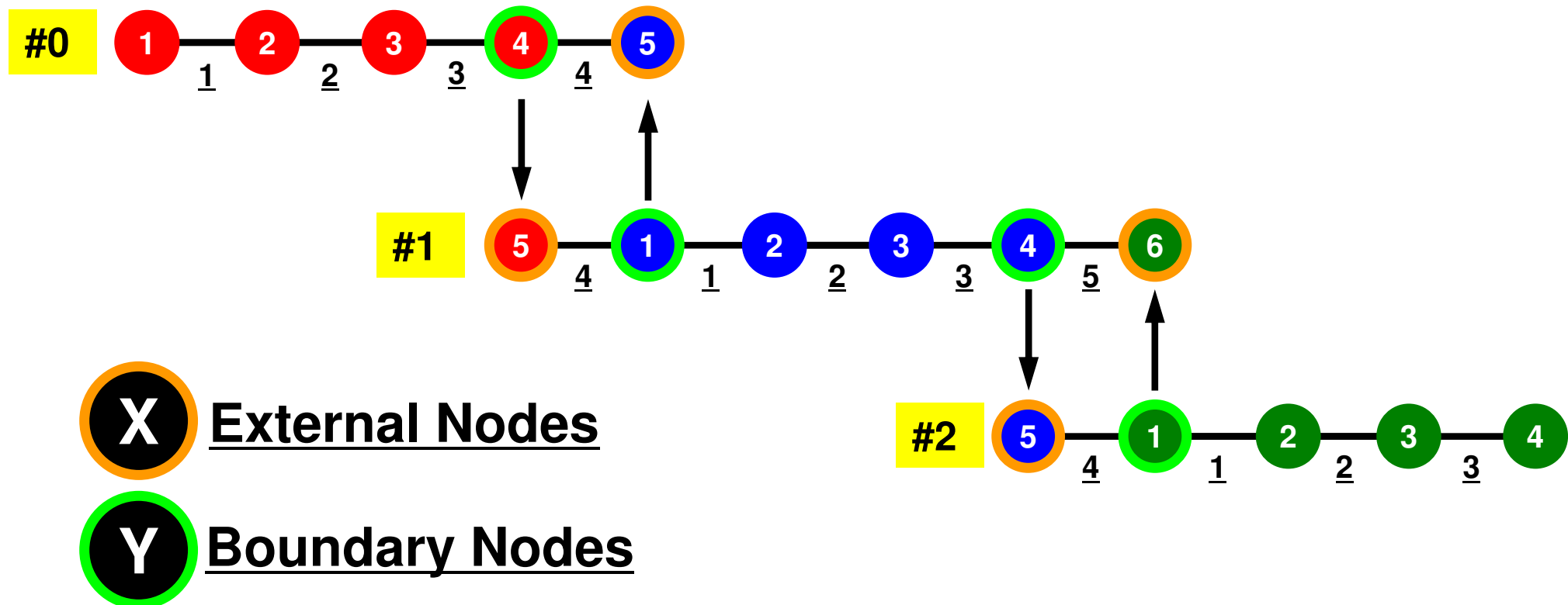
Local ID: Starting from 1 for node and elem at each domain



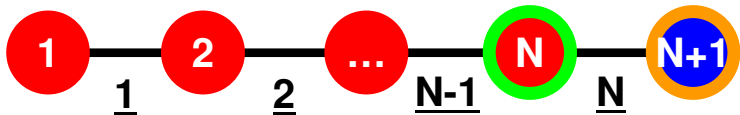
1D FEM: 12 nodes/11 elem's/3 domains

Internal/External/Boundary Nodes

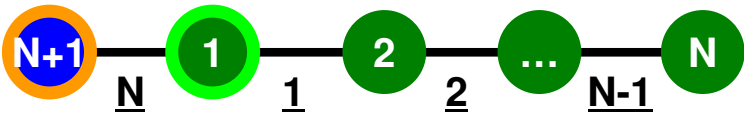
Boundary Nodes: Part of Internal Nodes, and External Nodes of Other Domains



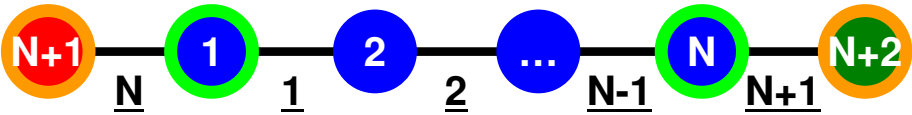
1D FEM: Numbering of Local ID



#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements

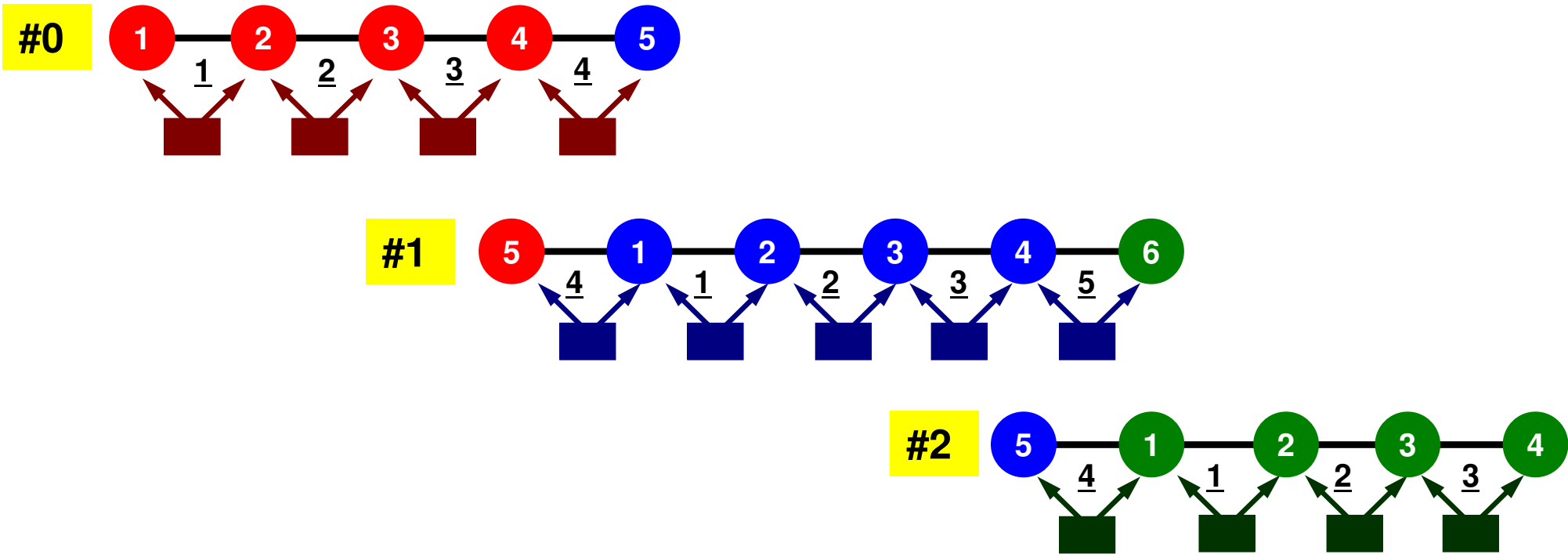


Others (General):
N+2 nodes
N+1 elements

1D FEM: 12 nodes/11 elem's/3 domains

Integration on each element, element matrix -> global matrix

Operations can be done by info. of internal/external nodes and elements which include these nodes



Preconditioned Conjugate Gradient Method (CG)

```

Compute  $\mathbf{r}^{(0)} = \mathbf{b} - [\mathbf{A}]\mathbf{x}^{(0)}$ 
for i= 1, 2, ...
    solve  $[\mathbf{M}]\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$ 
     $\rho_{i-1} = \mathbf{r}^{(i-1)} \cdot \mathbf{z}^{(i-1)}$ 
    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

```

Preconditioning:
Diagonal Scaling
(or Point Jacobi)

Preconditioning, DAXPY

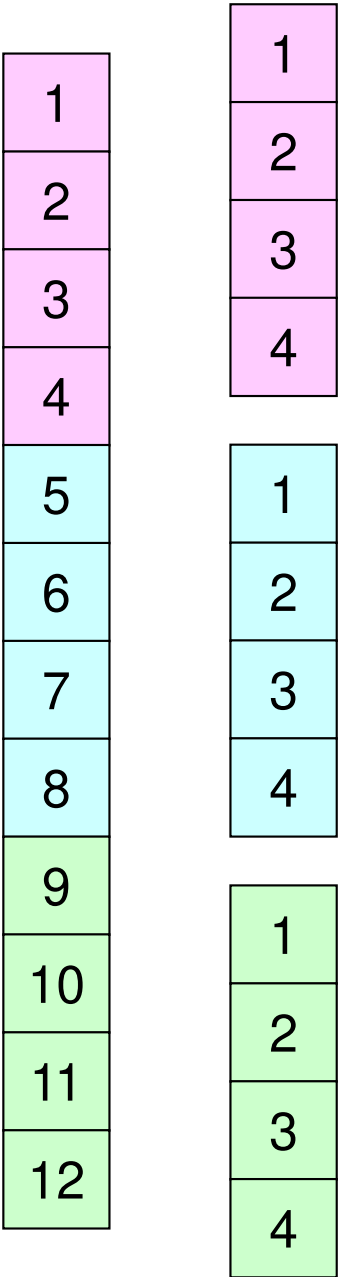
Local Operations by Only Internal Points: Parallel Processing is possible

```
!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
```

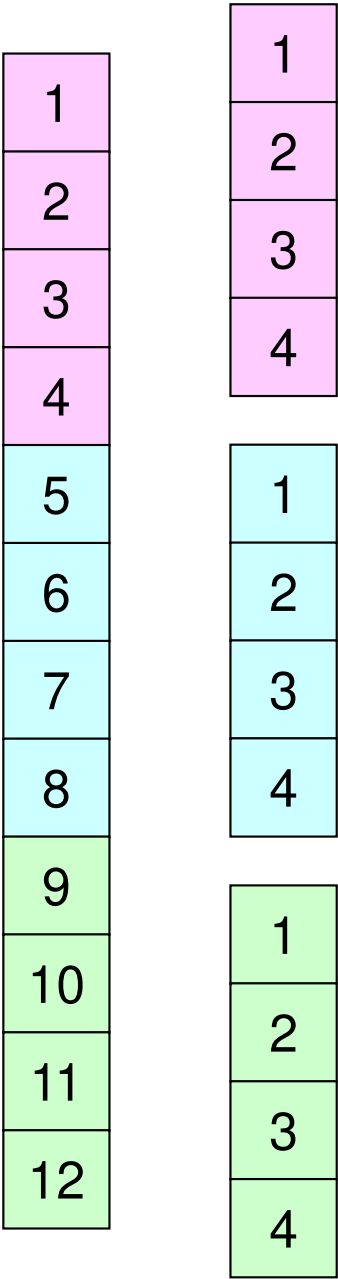


Dot Products

Global Summation needed: Communication ?

```
!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
```



Matrix-Vector Products

Values at External Points: P-to-P Communication

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

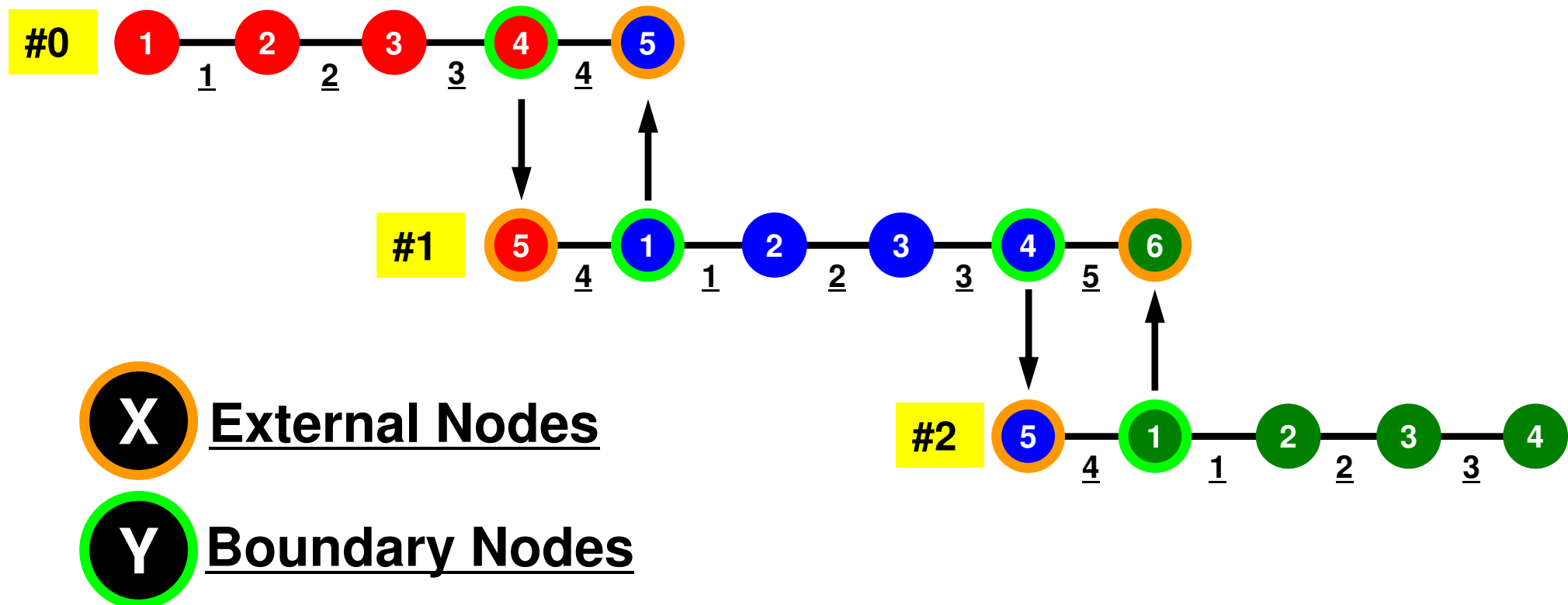
do i= 1, N
  W(i, Q) = DIAG(i)*W(i, P)
  do j= INDEX(i-1)+1, INDEX(i)
    W(i, Q) = W(i, Q) + AMAT(j)*W(ITEM(j), P)
  enddo
enddo
```



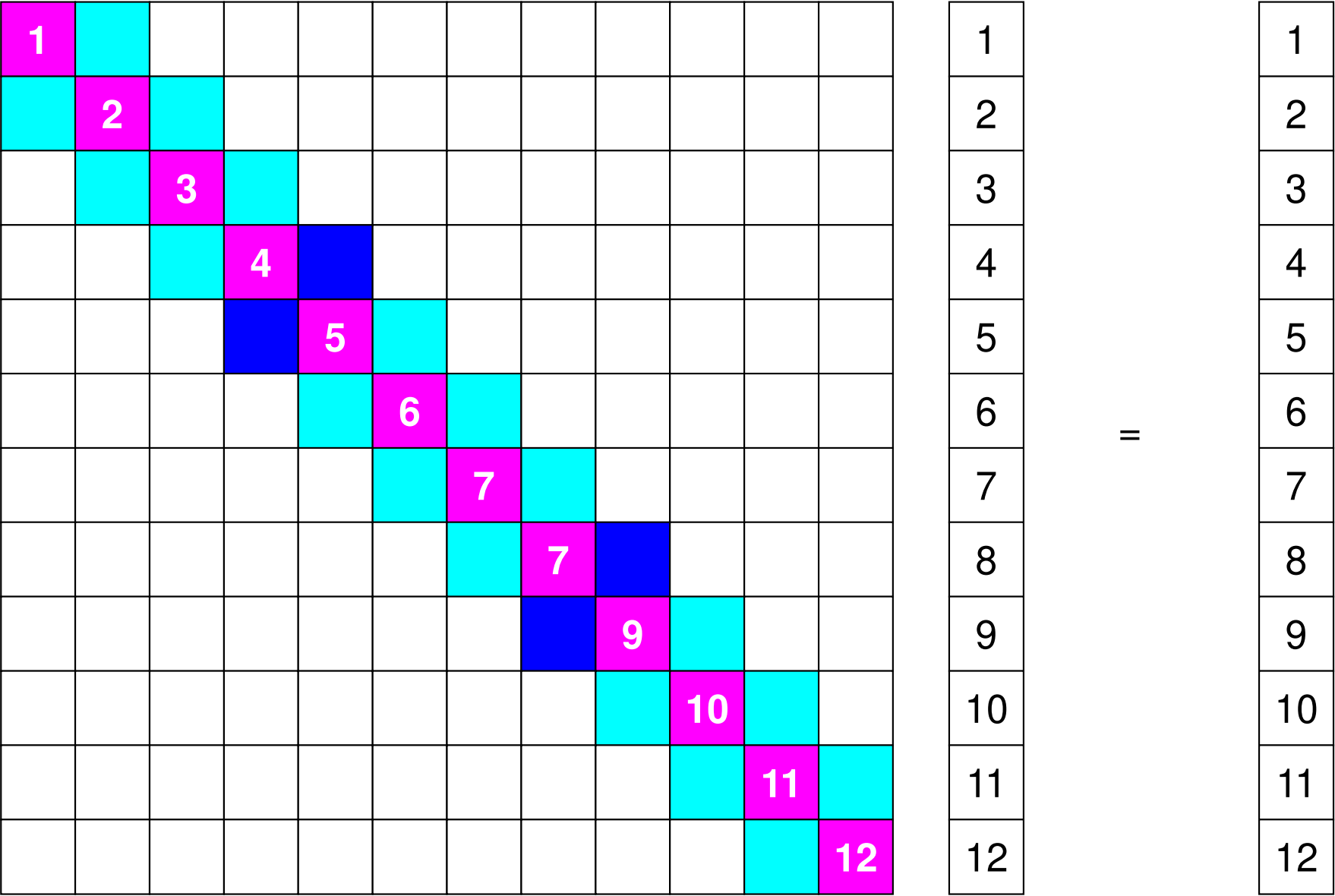
1D FEM: 12 nodes/11 elem's/3 domains

Internal/External/Boundary Nodes

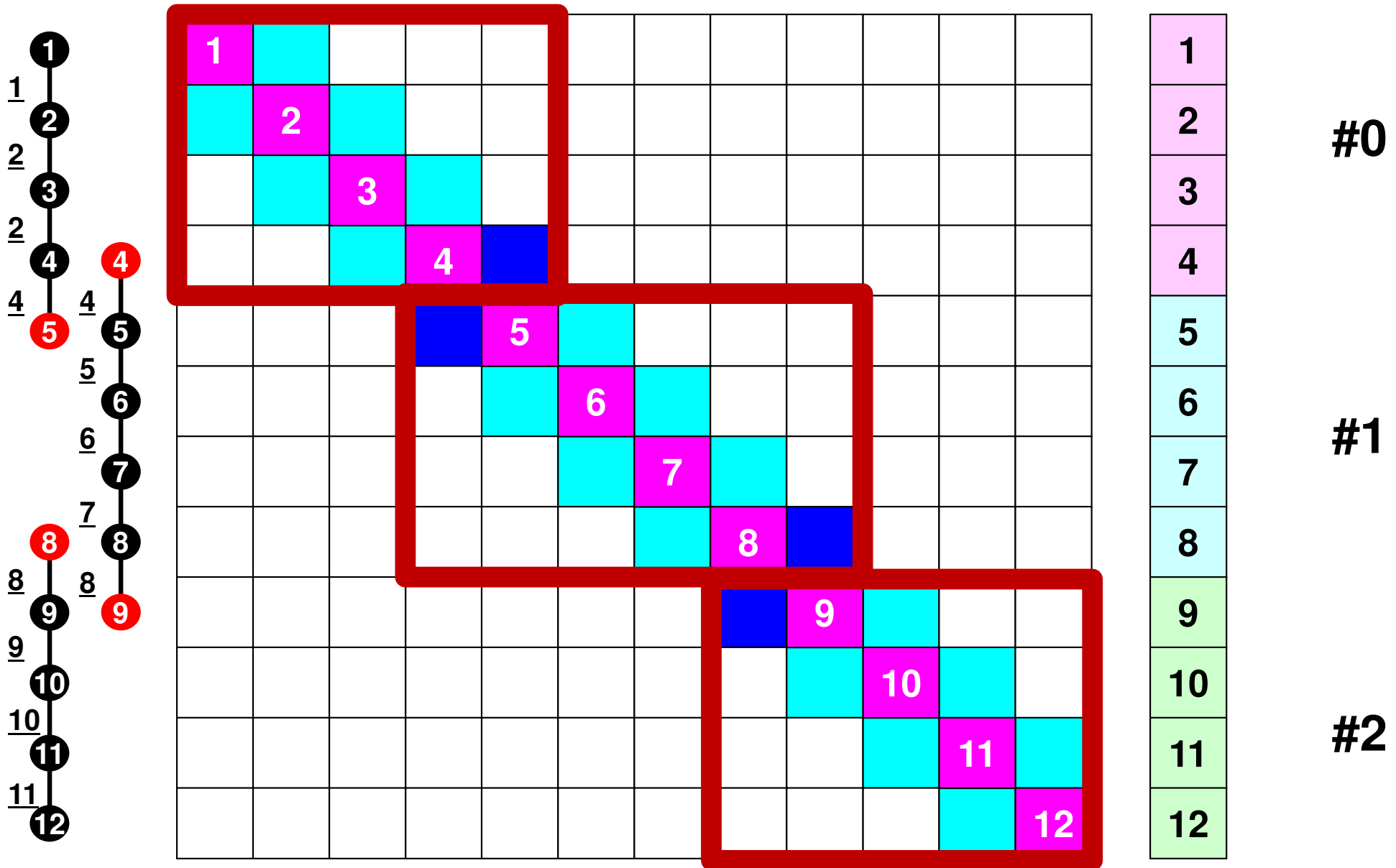
Boundary Nodes: Part of Internal Nodes, and External Nodes of Other Domains



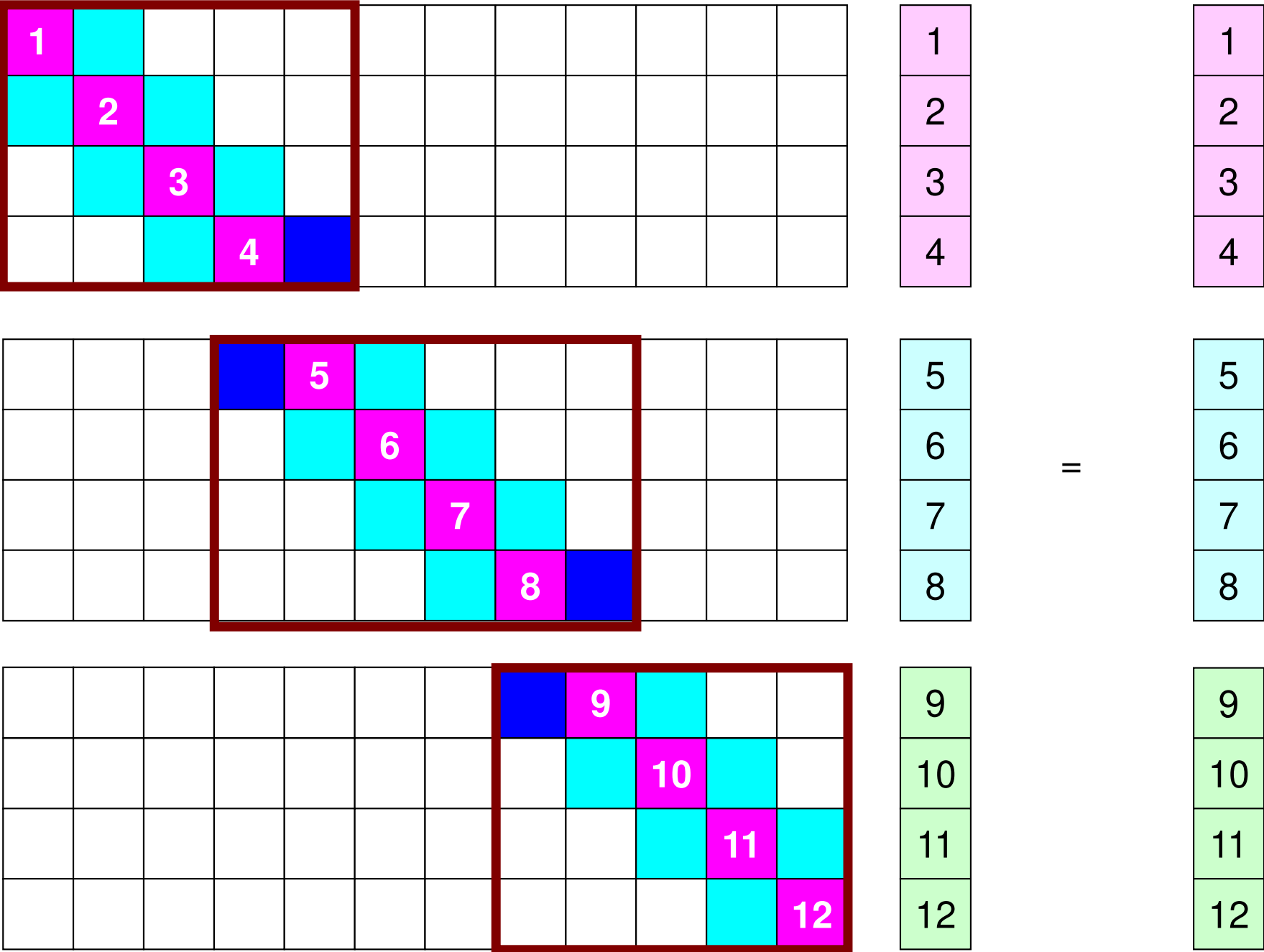
Mat-Vec Products: Local Op. Possible



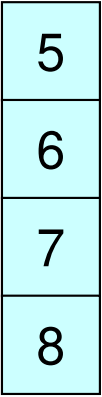
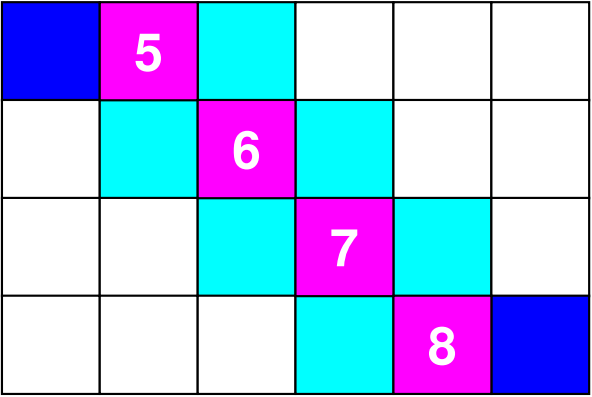
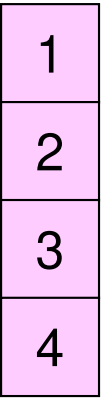
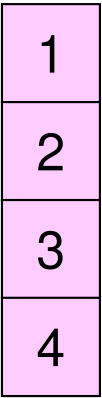
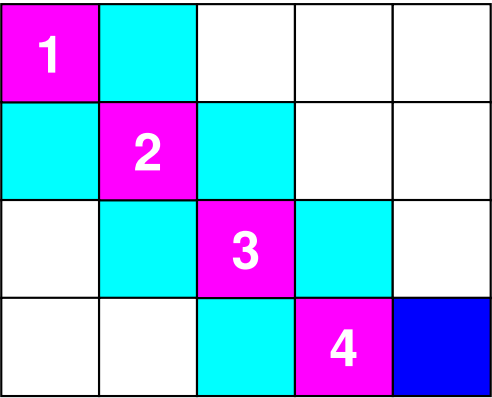
Because the matrix is sparse, the union of the local matrices forms the global matrix !



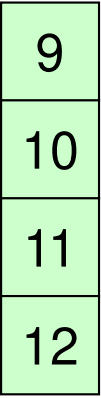
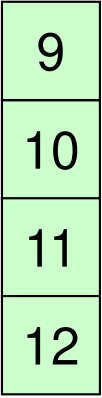
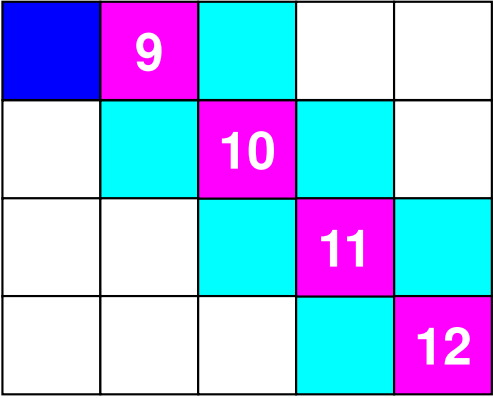
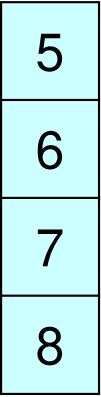
Mat-Vec Products: Local Op. Possible



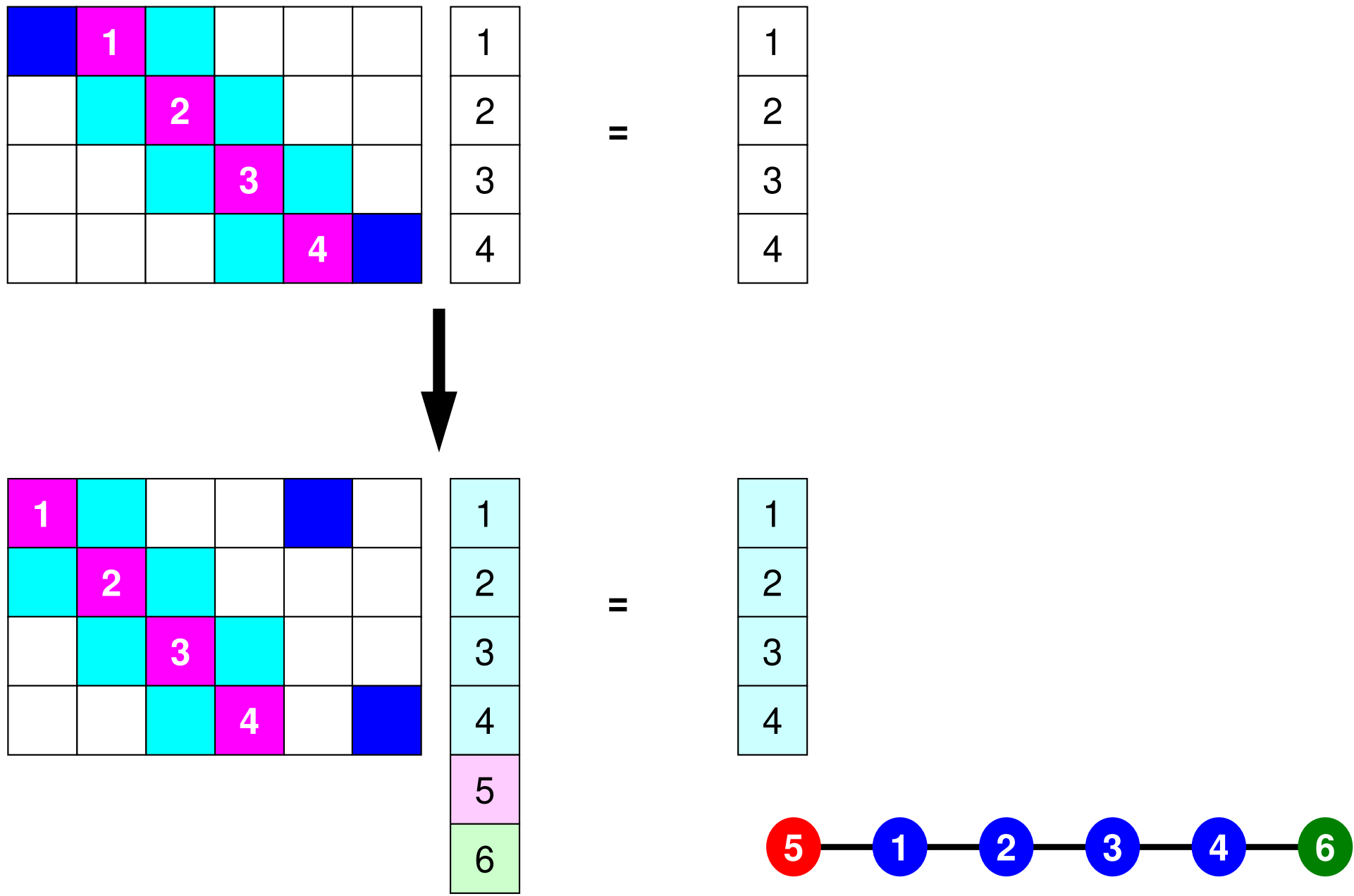
Mat-Vec Products: Local Op. Possible



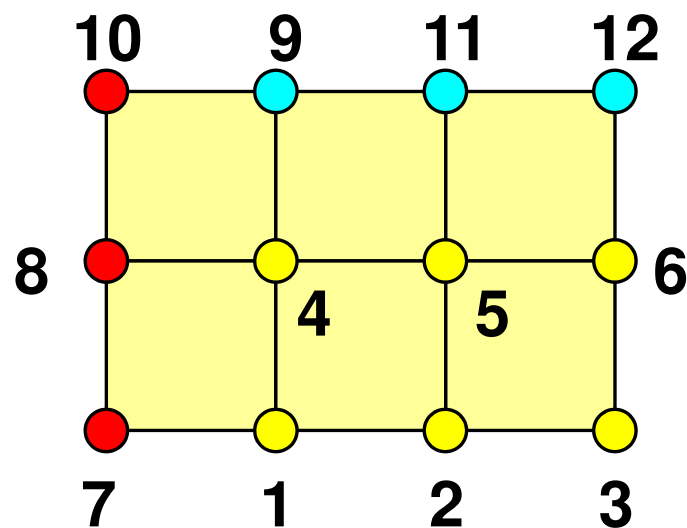
=



Mat-Vec Products: Local Op. #1



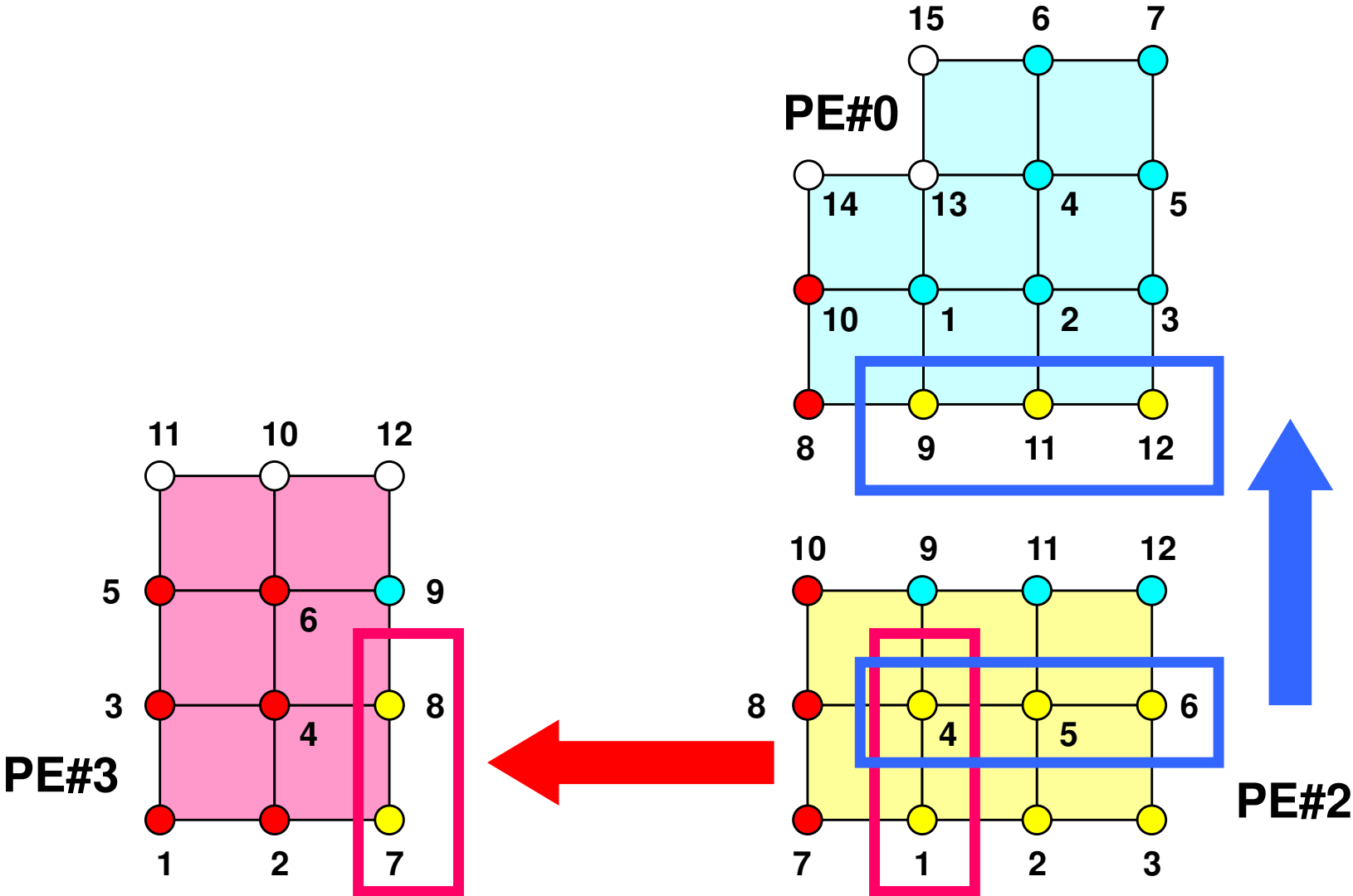
Description of Distributed Local Data



- Internal/External Points
 - Numbering: Starting from internal pts, then external pts after that
- Neighbors
 - Shares overlapped meshes
 - Number and ID of neighbors
- External Points
 - From where, how many, and which external points are received/imported ?
- Boundary Points
 - To where, how many and which boundary points are sent/exported ?

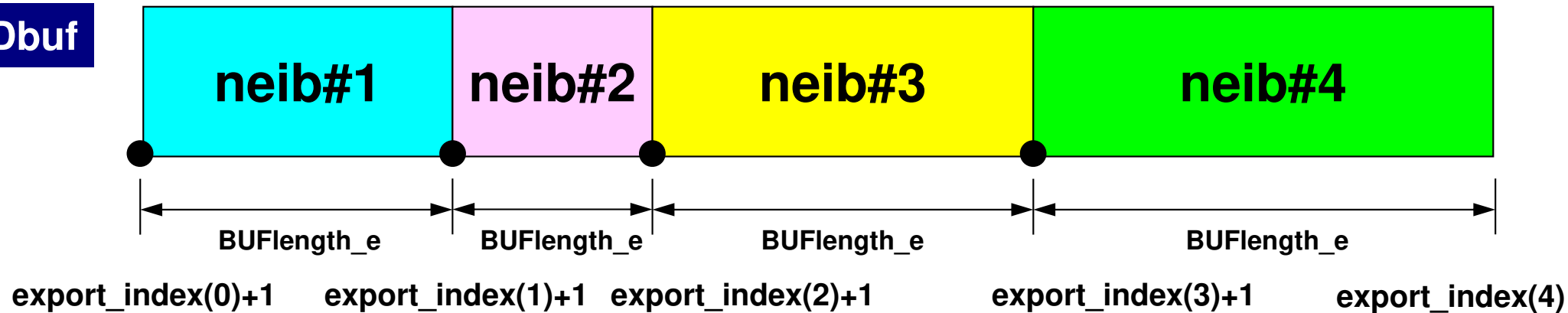
Boundary Nodes (境界点) : SEND

PE#2 : send information on “boundary nodes”



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

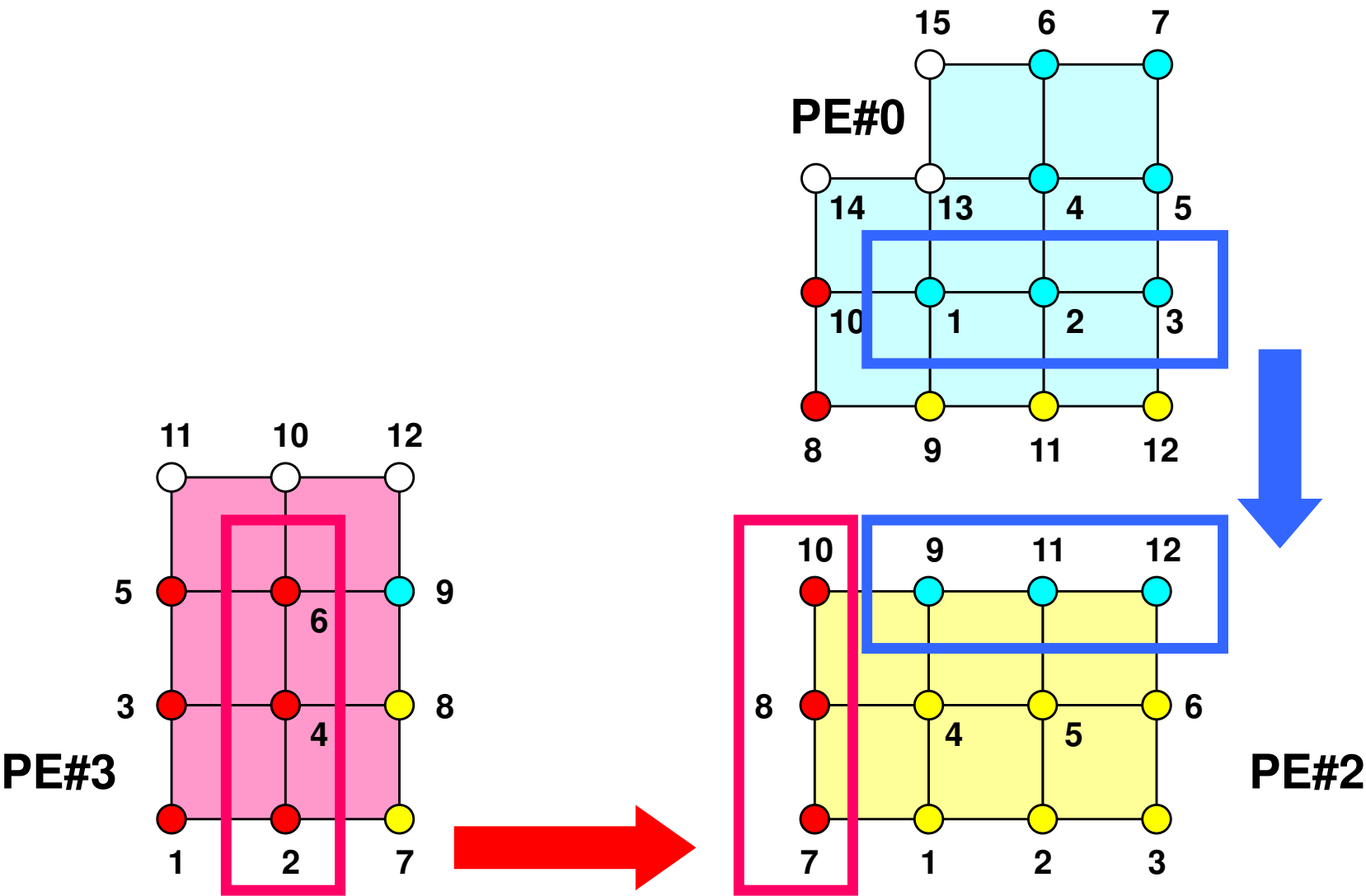
```
do neib= 1, NEIBPETOT
  iS_e= export_index(neib-1) + 1
  iE_e= export_index(neib )
  BUFlength_e= iE_e + 1 - iS_e

  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 : receive information for “external nodes”



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

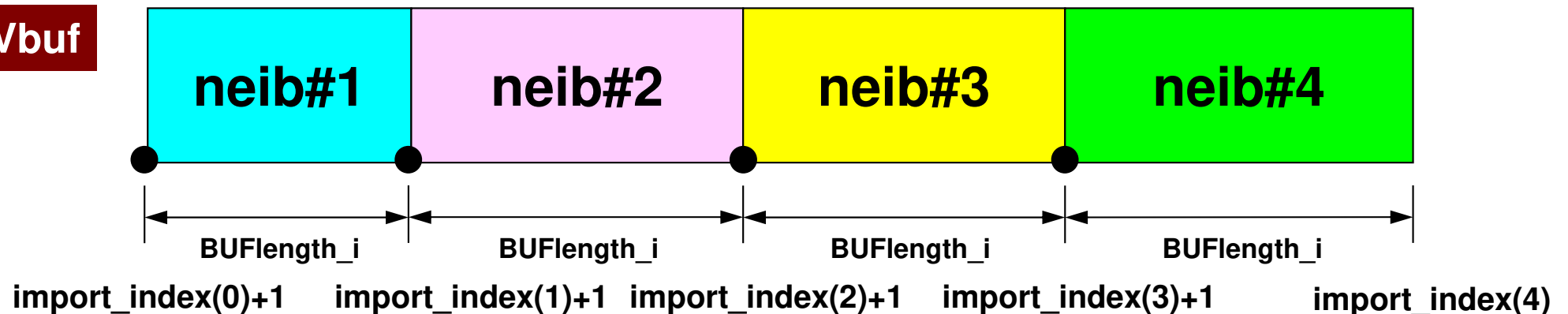
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

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



- Overview
- Distributed Local Data
- **Program**
- Results

Program: 1d.f (1/11)

Variables

```
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
```

Program: 1d.f (2/11)

Control Data

```
!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)
```

Initialization
Entire Process #: PETOT
Rank ID (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)
```

Program: 1d.f (2/11)

Control Data

```
!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)
```

Initialization
Entire Process #: PETOT
Rank ID (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
```

Reading control file if my_rank=0

Neg: Global Number of Elements

```
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)
```

Program: 1d.f (2/11)

Control Data

```
!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 )
```

Initialization
Entire Process #: PETOT
Rank ID (0-PETot-1): my_rank

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

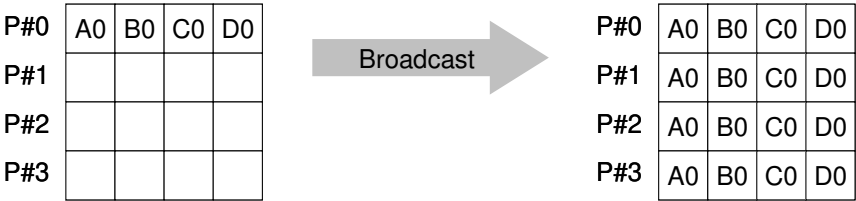
```
if (my_rank.eq.0) then
  open  (11, file='input.dat', status='unknown')
  read  (11,*) Neg
  read  (11,*) dX, QV, AREA, COND
  read  (11,*) ITERmax
  read  (11,*) EPS
  close (11)
endif
```

Reading control file if my_rank=0

Neg: Global Number of Elements

```
call MPI_Bcast (Neg      , 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr) Parameters are sent to each proces
call MPI_Bcast (ITERmax, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr) from Process #0.
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



- Broadcasts a message from the process with rank "root" to all other processes of the communicator
- call MPI_BCAST (buffer, count, datatype, root, comm, ierr)**
 - buffer** choice I/O starting address of buffer
 type is defined by "datatype"
 - count** I I number of elements in send/recv buffer
 - datatype** I I data type of elements of send/recv buffer
FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
 - root** I I rank of root process
 - comm** I I communicator
 - ierr** I O completion code

Program: 1d.f (3/11)

Distributed Local Mesh

```

!C
!C-- Local Mesh Size

Ng= NEg + 1
N = Ng / PETOT

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

mod(Ng, PETOT) .ne. 0

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

```

Program: 1d.f (3/11)

Distributed Local Mesh, Uniform Elements

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

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

Global Number of Nodes
Local Number of Nodes

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

$\text{mod}(Ng, PETOT) \neq 0$

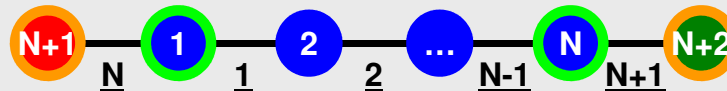
```
NE= N - 1 + 2
NP= N + 2
```

Number of Elements (Local)
Total Number of Nodes (Local) (Internal + External Nodes)

```
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
```



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

```
!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
```

Program: 1d.f (3/11)

Distributed Local Mesh, Uniform Elements

```
!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
```

Global Number of Nodes
Local Number of Nodes

```
mod(Ng, PETOT) .ne. 0
```

Number of Elements (Local)
Total Number of Nodes (Local) (Internal + External Nodes)

```
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:
 $N+1$ nodes
 N elements

```
!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
```

Program: 1d.f (3/11)

Distributed Local Mesh, Uniform Elem

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

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

Global Number of Nodes
Local Number of Nodes

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

$\text{mod}(Ng, PETOT) \neq 0$

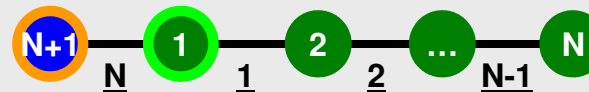
```
NE= N - 1 + 2
NP= N + 2
```

Number of Elements (Local)
Total Number of Nodes (Local) (Internal + External Nodes)

```
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 nodes
N elements

```
!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
```

Program: 1d.f (3/11)

Distributed Local Mesh, Uniform Elements

```

!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

```

Global Number of Nodes
Local Number of Nodes
mod(Ng, PETOT) .ne. 0
Number of Elements (Local)
Total Number of Nodes (Local) (Internal + External Nodes)
Size of arrays is "NP", not "N"

Program: 1d.f (4/11)

Initialization of Arrays, Elements-Nodes

```
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
```

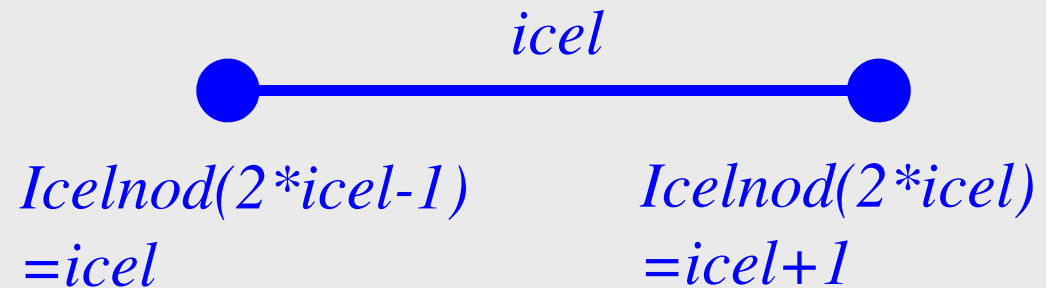
```
  icel= NE
```

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

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

```
endif
```

```
endif
```



Program: 1d.f (4/11)

Initialization of Arrays, Elements-Nodes

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

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

e.g. Element-1 includes node-1 and node-2

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



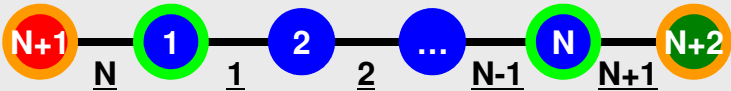
#0:
N+1 nodes
N elements

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



#PETot-1:
N+1 nodes
N elements

```
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
```



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

```
endif
endif
```

Program: 1d.f (5/11)

"Index"

```
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
!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
```

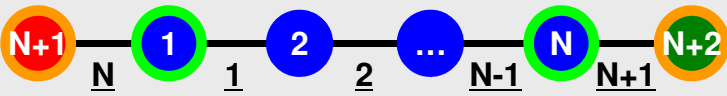
```
NPLU= INDEX(NP)
ITEM= 0
```



#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements



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

Program: 1d.f (6/11)

"Item"

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



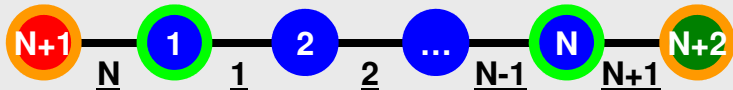
#0:
N+1 nodes
N elements

```
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 nodes
N elements

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



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

Program: 1d.f (7/11)

Communication Tables

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

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

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

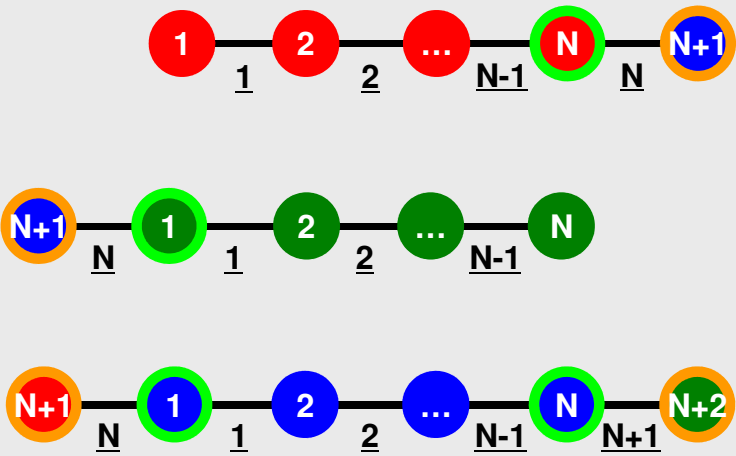
```
BUFlength= 1
```

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

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

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

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



#0:
N+1 nodes
N elements

#PETot-1:
N+1 nodes
N elements

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

MPI_ISEND

- Begins a non-blocking send
 - Send the contents of sending buffer (starting from **sendbuf**, number of messages: **count**) to **dest** with **tag** .
 - Contents of sending buffer cannot be modified before calling corresponding **MPI_Waitall**.
- **call MPI_ISEND**
(sendbuf, count, datatype, dest, tag, comm, request, ierr)
 - **sendbuf** choice I starting address of sending buffer
 - **count** I I number of elements sent to each process
 - **datatype** I I data type of elements of sending buffer
 - **dest** I I rank of destination
 - **tag** I I message tag

This integer can be used by the application to distinguish messages. Communication occurs if tag's of MPI_Isend and MPI_Irecv are matched. Usually tag is set to be "0" (in this class),
 - **comm** I I communicator
 - **request** I O communication request array used in MPI_Waitall
 - **ierr** I O completion code

MPI_Irecv

- Begins a non-blocking receive
 - Receiving the contents of receiving buffer (starting from **recvbuf**, number of messages: **count**) from **source** with **tag** .
 - Contents of receiving buffer cannot be used before calling corresponding **MPI_Waitall**.

- **call MPI_Irecv**

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

– <u>recvbuf</u>	choice	I	starting address of receiving buffer
– <u>count</u>	I	I	number of elements in receiving buffer
– <u>datatype</u>	I	I	data type of elements of receiving buffer
– <u>source</u>	I	I	rank of source
– <u>tag</u>	I	I	message tag
This integer can be used by the application to distinguish messages. Communication occurs if tag' s of MPI_Isend and MPI_Irecv are matched.			
Usually tag is set to be “0” (in this class),			
– <u>comm</u>	I	I	communicator
– <u>request</u>	I	O	communication request used in MPI_Waitall
– <u>ierr</u>	I	O	completion code

MPI_WAITALL

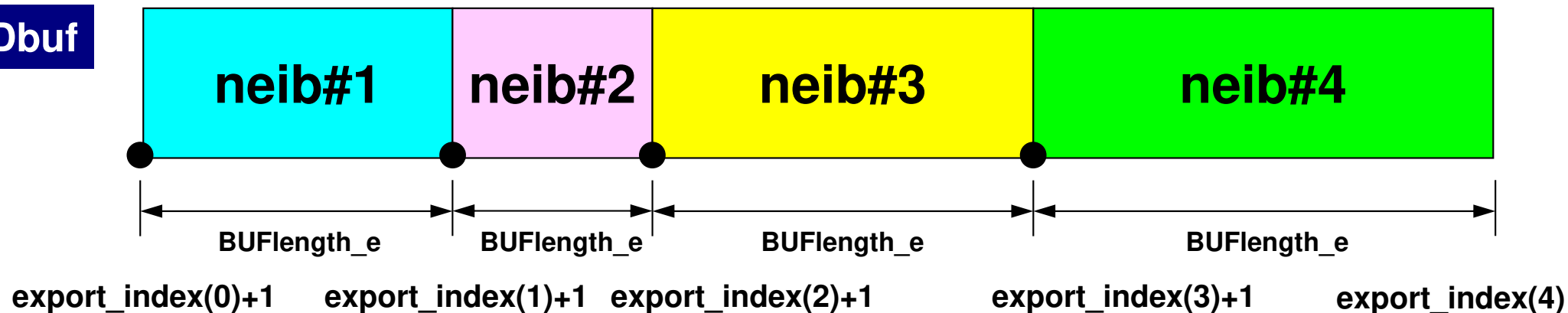
- **MPI_Waitall** blocks until all comm's, associated with request in the array, complete. It is used for synchronizing MPI_Isend and MPI_Irecv in this class.
- At sending phase, contents of sending buffer cannot be modified before calling corresponding **MPI_Waitall**. At receiving phase, contents of receiving buffer cannot be used before calling corresponding **MPI_Waitall**.
- MPI_Isend and MPI_Irecv can be synchronized simultaneously with a single **MPI_Waitall** if it is consistent.
 - Same request should be used in MPI_Isend and MPI_Irecv.
- Its operation is similar to that of **MPI_Barrier** but, **MPI_Waitall** can not be replaced by **MPI_Barrier**.
 - Possible troubles using **MPI_Barrier** instead of **MPI_Waitall**: Contents of **request** and **status** are not updated properly, very slow operations etc.
- **call MPI_WAITALL (count, request, status, ierr)**
 - count I I number of processes to be synchronized
 - request I I/O comm. request used in MPI_Waitall (array size: count)
 - status I O array of status objects
MPI_STATUS_SIZE: defined in 'mpif.h', 'mpi.h'
 - ierr I O completion code

Generalized Comm. Table: Send

- Neighbors
 - NEIBPETOT, NEIBPE(neib)
- Message size for each neighbor
 - export_index(neib), neib= 0, NEIBPETOT
- ID of **boundary** points
 - export_item(k), k= 1, export_index(NEIBPETOT)
- Messages to each neighbor
 - SENDbuf(k), k= 1, export_index(NEIBPETOT)

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
```

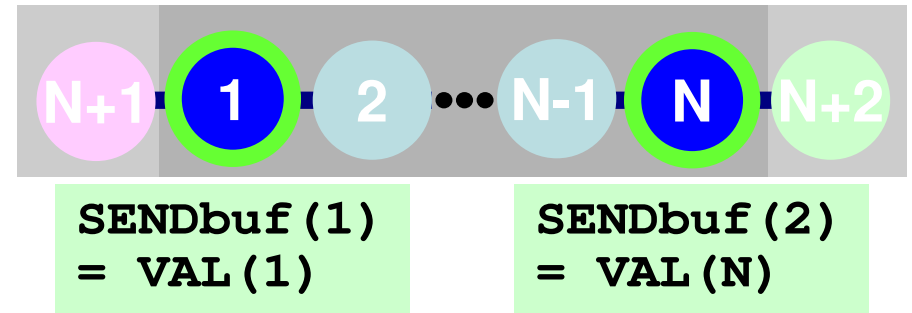
Copied to sending buffers

```
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)
```

SEND/Export: 1D Problem



- Neighbors
 - NEIBPETOT, NEIBPE(neib)
 - $\text{NEIBPETOT}=2$, $\text{NEIB}(1)=\text{my_rank}-1$, $\text{NEIB}(2)=\text{my_rank}+1$
- Message size for each neighbor
 - export_index(neib), neib= 0, NEIBPETOT
 - $\text{export_index}(0)=0$, $\text{export_index}(1)= 1$, $\text{export_index}(2)= 2$
- ID of **boundary** points
 - export_item(k), k= 1, export_index(NEIBPETOT)
 - $\text{export_item}(1)= 1$, $\text{export_item}(2)= N$
- Messages to each neighbor
 - SENDbuf(k), k= 1, export_index(NEIBPETOT)
 - $\text{SENDbuf}(1)= \text{BUF}(1)$, $\text{SENDbuf}(2)= \text{BUF}(N)$

Generalized Comm. Table: Receive

- Neighbors
 - NEIBPETOT, NEIBPE(neib)
- Message size for each neighbor
 - import_index(neib), neib= 0, NEIBPETOT
- ID of **external** points
 - import_item(k), k= 1, import_index(NEIBPETOT)
- Messages from each neighbor
 - RECVbuf(k), k= 1, import_index(NEIBPETOT)

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

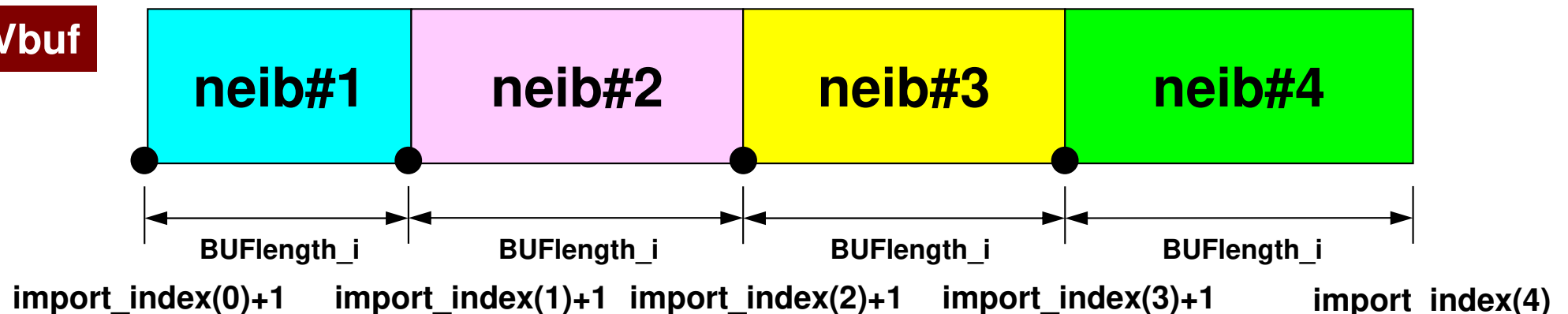
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

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

```

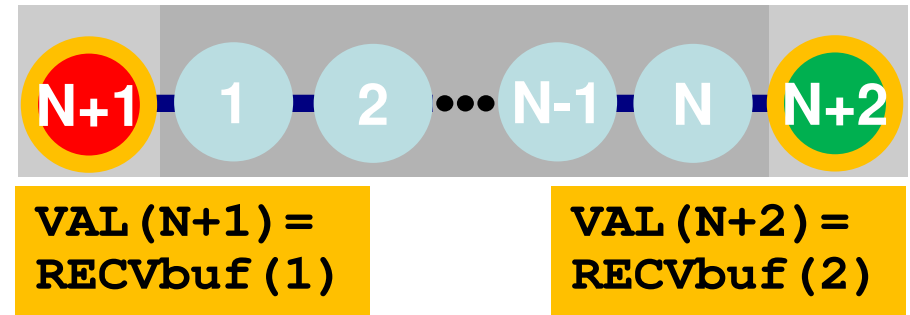
Copied from receiving buffer

RECVbuf

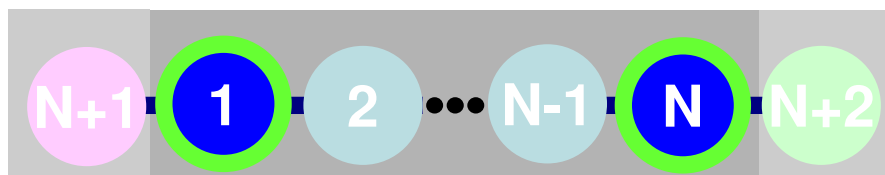


RECV/Import: 1D Problem

- Neighbors
 - NEIBPETOT, NEIBPE(neib)
 - $NEIBPETOT=2$, $NEIB(1)=my_rank-1$, $NEIB(2)=my_rank+1$
- Message size for each neighbor
 - import_index(neib), neib= 0, NEIBPETOT
 - $import_index(0)=0$, $import_index(1)=1$, $import_index(2)=2$
- ID of external points
 - import_item(k), k= 1, import_index(NEIBPETOT)
 - $import_item(1)=N+1$, $import_item(2)=N+2$
- Messages from each neighbor
 - RECVbuf(k), k= 1, import_index(NEIBPETOT)
 - $BUF(N+1)=RECVbuf(1)$, $BUF(N+2)=RECVbuf(2)$

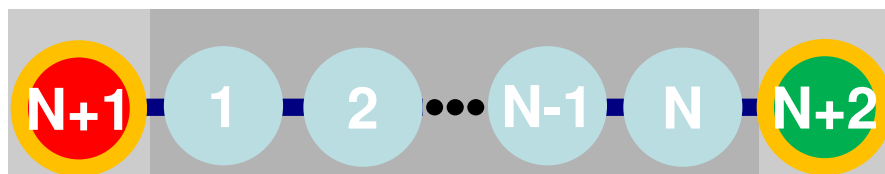


Generalized Comm. Table: Fortran



SENDbuf (1)
= VAL (1)

SENDbuf (2)
= VAL (N)



VAL (N+1) =
RECVbuf (1)

VAL (N+2) =
RECVbuf (2)

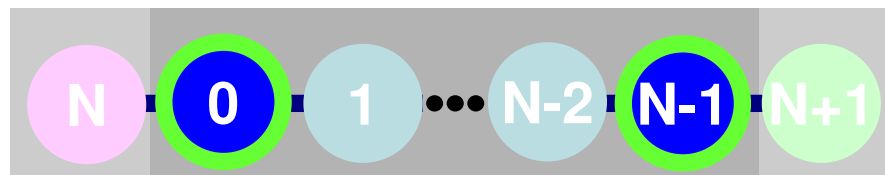
```
NEIBPETOT= 2
NEIBPE (1)= my_rank - 1
NEIBPE (2)= my_rank + 1
```

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

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

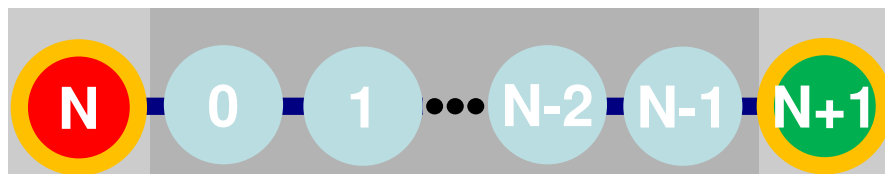
```
if (my_rank.eq.0) then
  import_item (1)= N+1
  export_item (1)= N
  NEIBPE (1)= my_rank+1
endif
```

Generalized Comm. Table: C



SENDbuf[0]
= VAL[0]

SENDbuf[1]
= VAL[N-1]



VAL[N]=
RECVbuf[0]

VAL[N+1]=
RECVbuf[1]

```
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
```

Program: 1d.f (8/11)

Matrix Assembling, NO changes from 1-CPU co

```
!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)
```

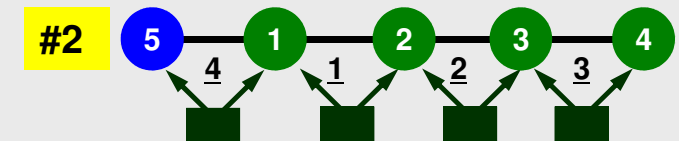
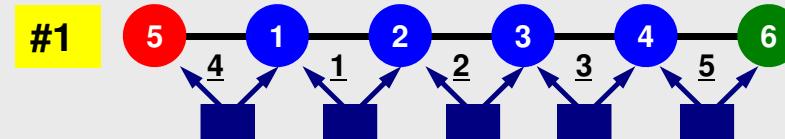
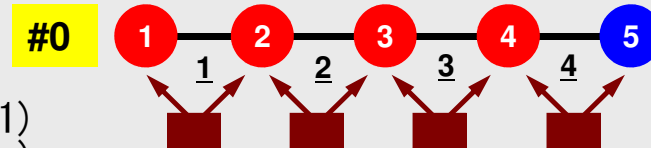
```
if (my_rank.eq.0.and.icel.eq.1) then
  k1= INDEX(in1-1) + 1
else
  k1= INDEX(in1-1) + 2
endif
k2= INDEX(in2-1) + 1
```

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

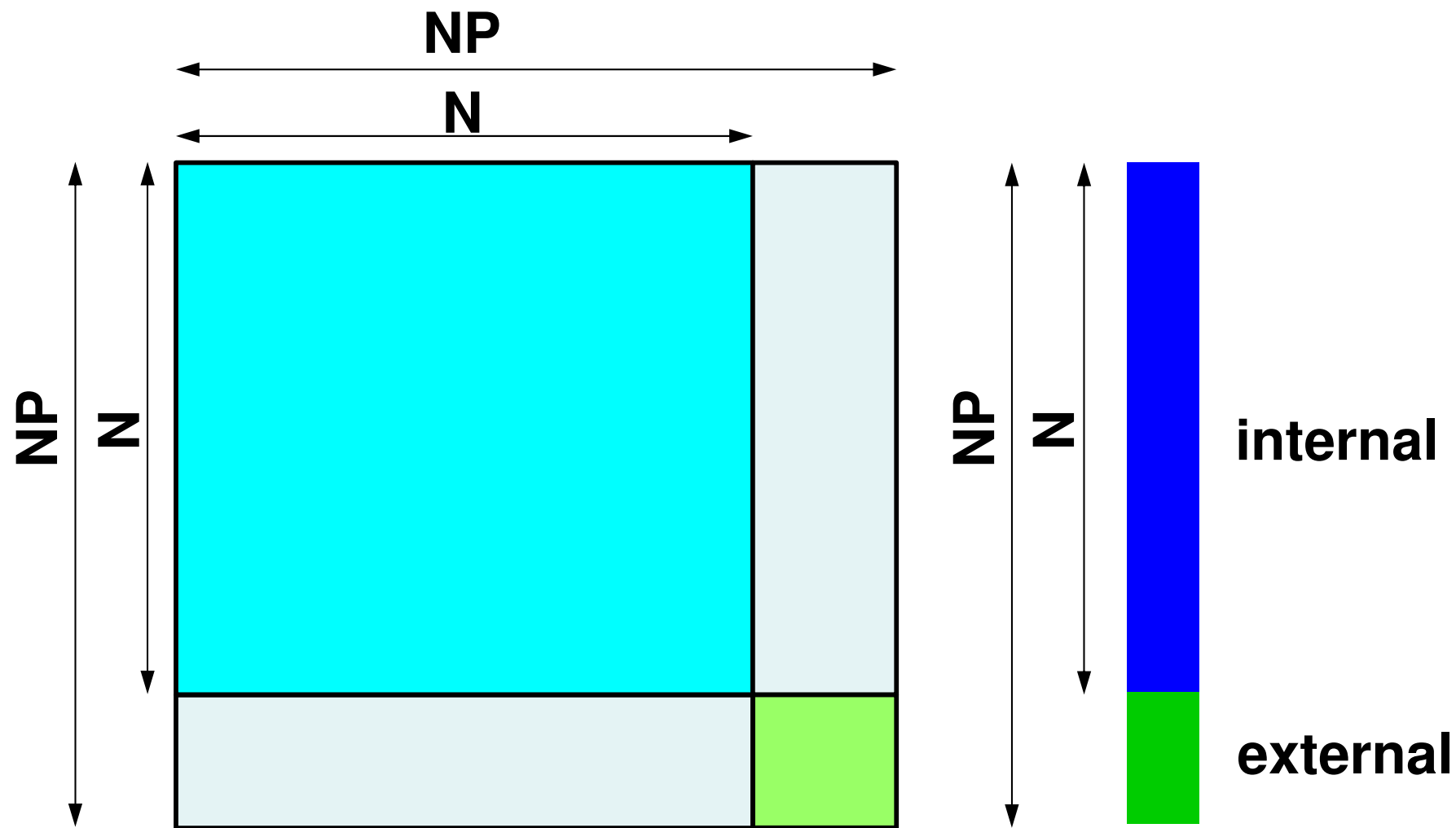
```
QN= 0.5d0*QV*AREA*DL
RHS(in1)= RHS(in1) + QN
RHS(in2)= RHS(in2) + QN
```

```
enddo
```

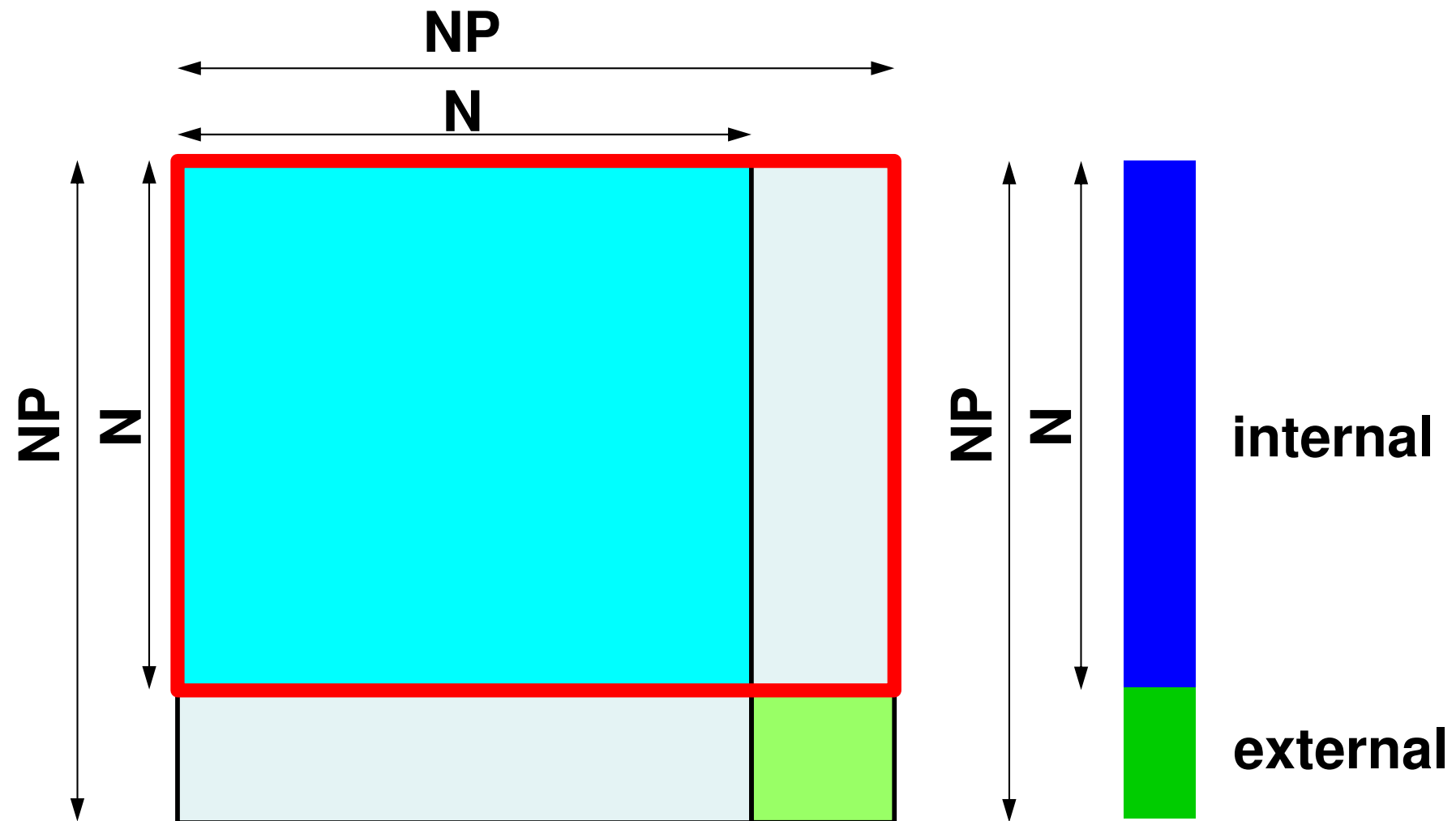
```
!C===
```



Local Matrix



We really need these parts:



MAT_ASS_MAIN: Overview

```

do kpn= 1, 2      Gaussian Quad. points in  $\zeta$ -direction
  do jpn= 1, 2    Gaussian Quad. points in  $\eta$ -direction
    do ipn= 1, 2  Gaussian Quad. Pointe in  $\xi$ -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)

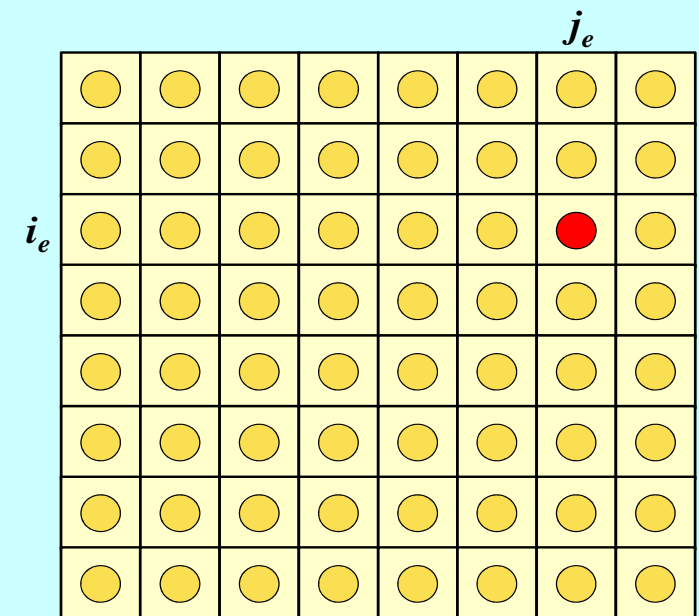
```

```

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

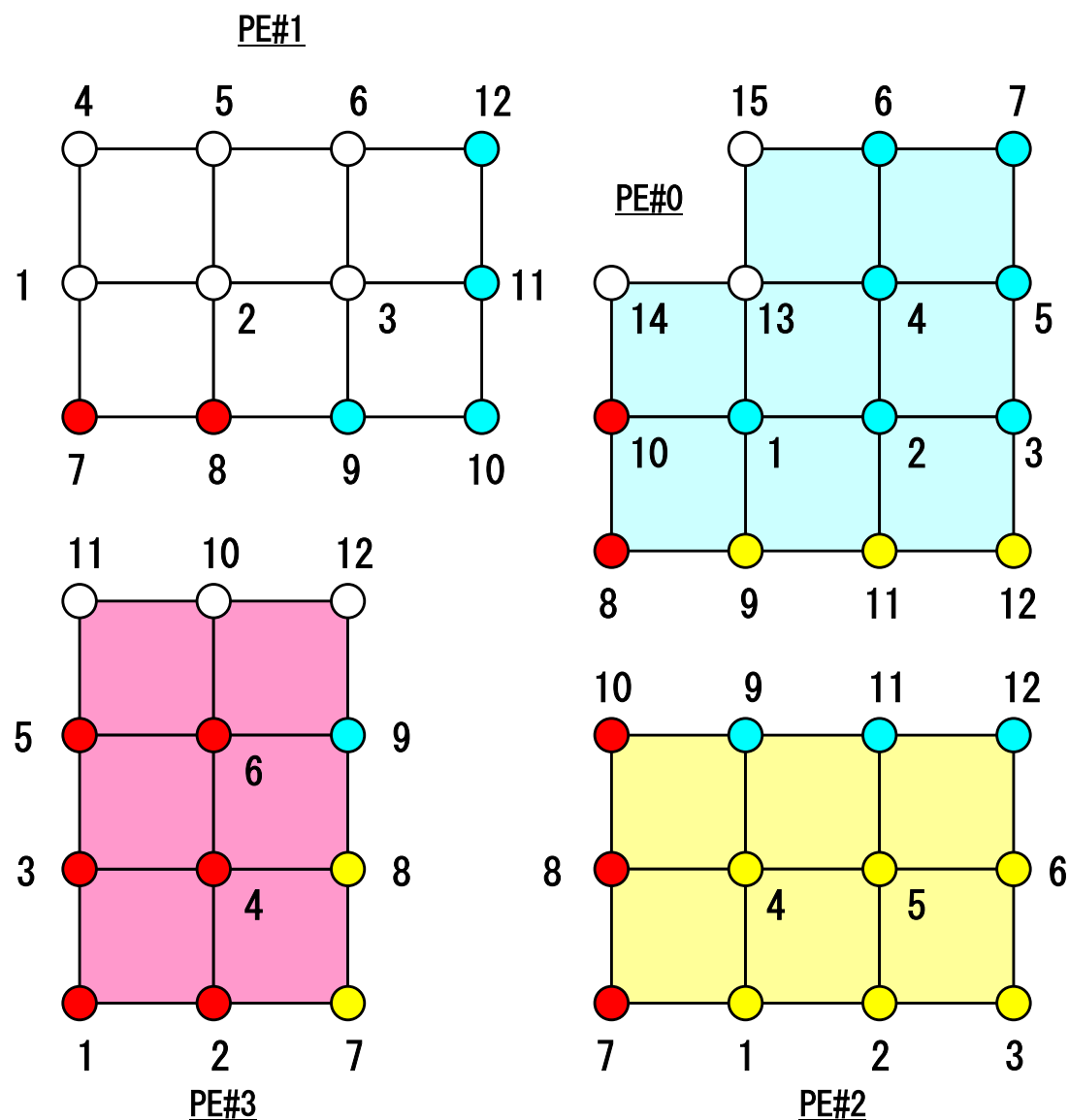
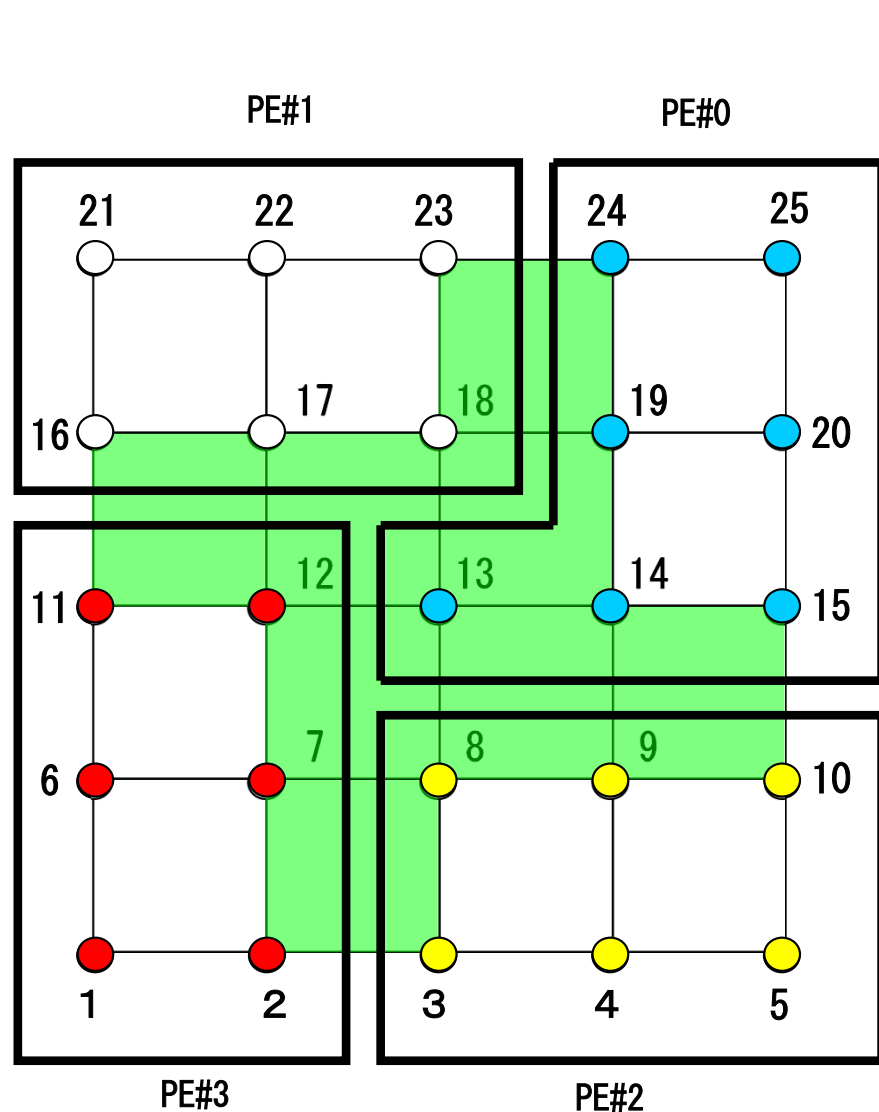
    do kpn= 1, 2      Gaussian Quad. points in  $\zeta$ -direction
      do jpn= 1, 2    Gaussian Quad. points in  $\eta$ -direction
        do ipn= 1, 2  Gaussian Quad. points in  $\xi$ -direction
          integration on each element
          coefficients of element matrices
          accumulation to global matrix
        enddo
      enddo
    enddo
  enddo
enddo
enddo
enddo

```

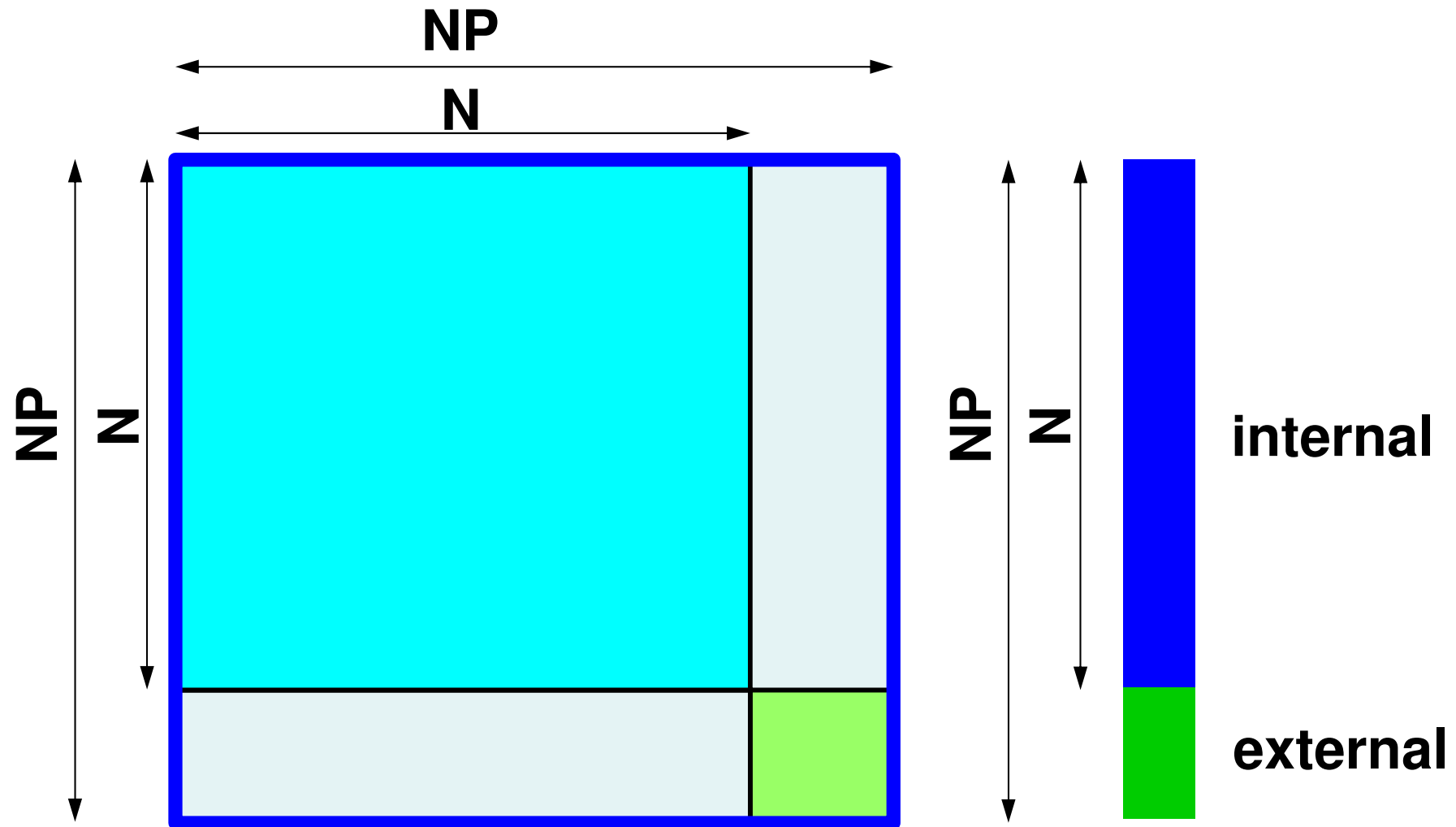


MAT_ASS_MAIN visits all elements

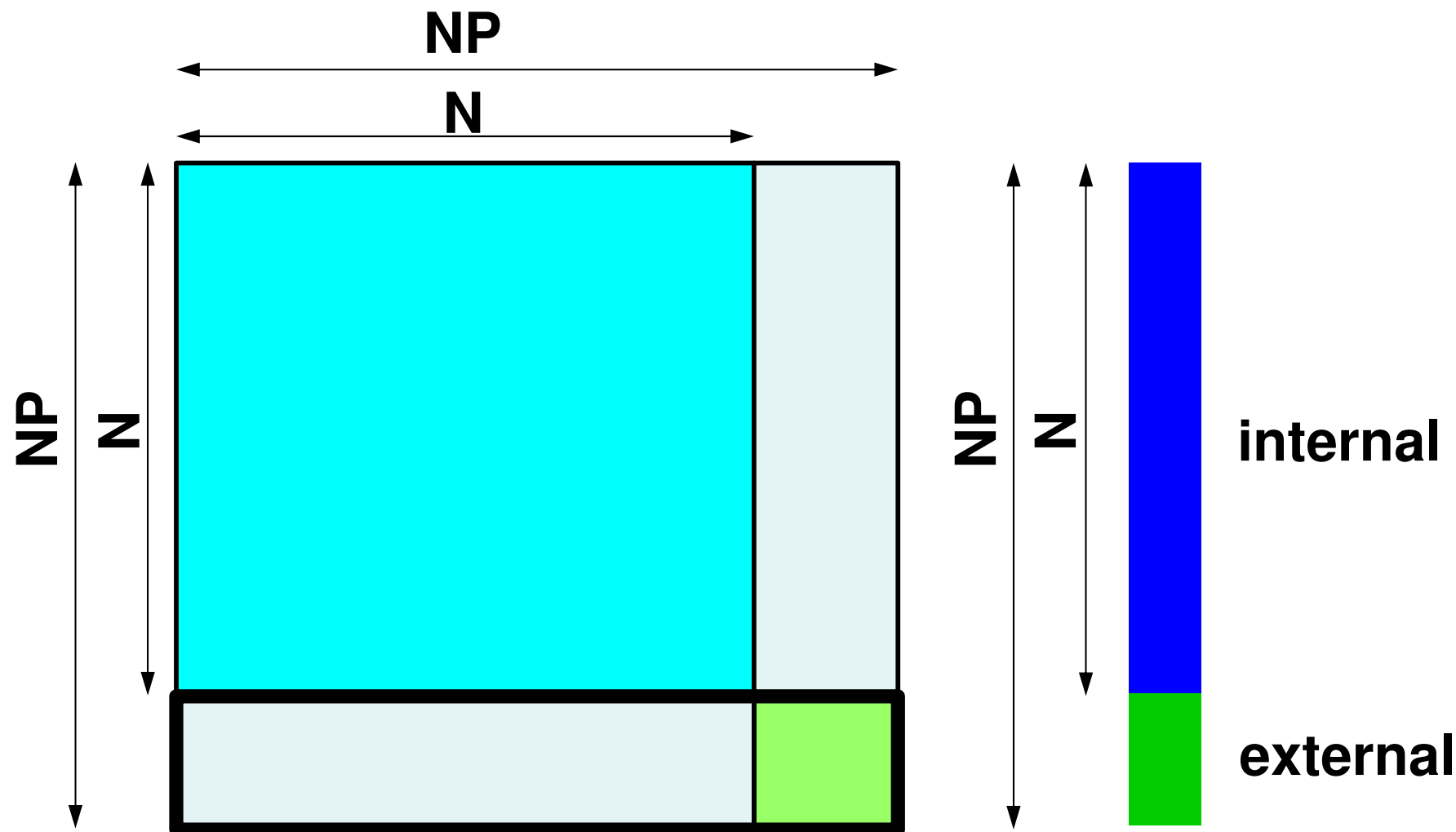
including overlapped elements with external nodes



Therefore, we have this matrix



But components of this part are not complete, and not used in computation



Program: 1d.f (9/11)

Boundary Cond., ALMOST NO changes from 1-CPU code

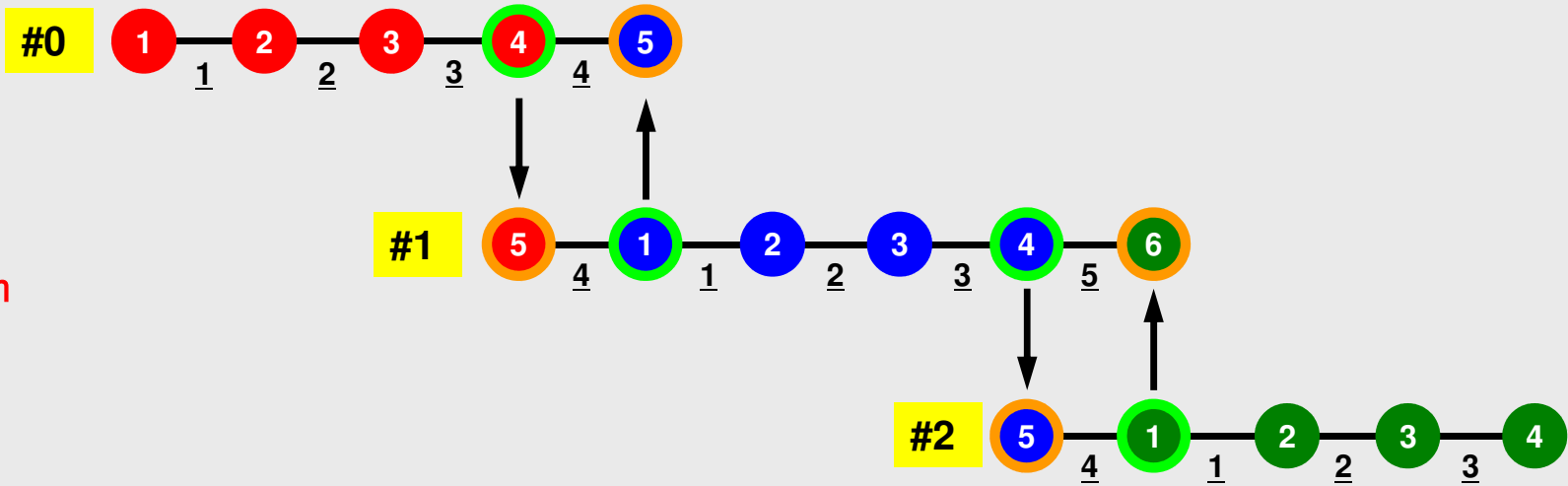
```
!C
!C +-----+
!C | BOUNDARY CONDITIONS |
!C +-----+
!C==
```

```
!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==
```



Program: 1d.c(10/11)

Conjugate Gradient Method

```

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

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

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

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

```

```

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

```

Conjugate Gradient Method (CG)

- Matrix-Vector Multiply
- Dot Product
- Preconditioning: in the same way as 1CPU code
- DAXPY: in the same way as 1CPU code

Preconditioning, 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
```

Matrix-Vector Multiply (1/2)

Using Comm. Table, {p} is updated before computation

```
!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
```

Matrix-Vector Multiply (2/2)

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

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

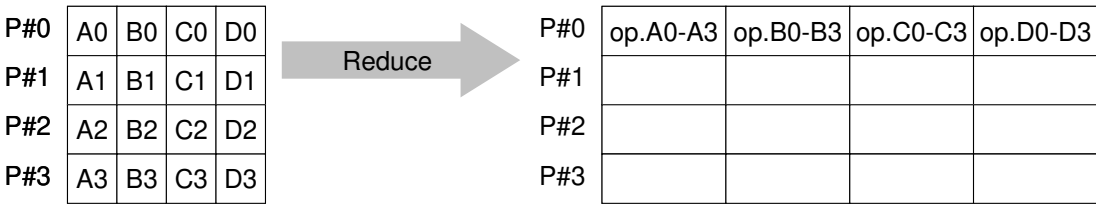
```
do i= 1, N  
  W(i,Q) = DIAG(i)*W(i,P)  
  do j= INDEX(i-1)+1, INDEX(i)  
    W(i,Q) = W(i,Q) + AMAT(j)*W(ITEM(j),P)  
  enddo  
enddo
```

Global Summation by MPI_Allreduce

```
!C
!C-- RH0= {r} {z}

      RH00= 0. d0
      do i= 1, N
        RH00= RH00 + W(i, R)*W(i, Z)
      enddo
      call MPI_Allreduce (RH00, RH0, 1, MPI_DOUBLE_PRECISION,
& MPI_SUM, MPI_COMM_WORLD, ierr)
```

MPI_REDUCE



- Reduces values on all processes to a single value
 - Summation, Product, Max, Min etc.
- call **MPI_REDUCE**
(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
 - sendbuf** choice I starting address of send buffer
 - recvbuf** choice O starting address receive buffer
 - type is defined by "**datatype**"
 - count** I I number of elements in send/receive buffer
 - datatype** I I data type of elements of send/recive buffer
 - FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
 - C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc
 - op** I I reduce operation
 - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
 - Users can define operations by **MPI_OP_CREATE**
 - root** I I rank of root process
 - comm** I I communicator
 - ierr** I O completion code

Send/Receive Buffer (Sending/Receiving)

- Arrays of “send (sending) buffer” and “receive (receiving) buffer” often appear in MPI.
- Addresses of “send (sending) buffer” and “receive (receiving) buffer” must be different.

Example of MPI_Reduce (1/2)

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

```
real(kind=8):: X0, X1  
  
call MPI_REDUCE  
(X0, X1, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0, <comm>, ierr)
```

```
real(kind=8):: X0(4), XMAX(4)  
  
call MPI_REDUCE  
(X0, XMAX, 4, MPI_DOUBLE_PRECISION, MPI_MAX, 0, <comm>, ierr)
```

Global Max values of X0(i) go to XMAX(i) on #0 process (i=1~4)

Example of MPI_Reduce (2/2)

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

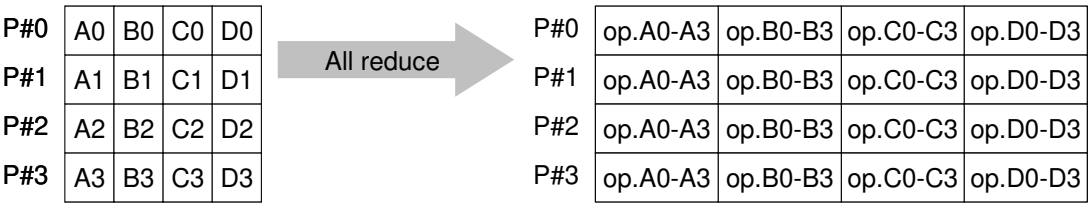
```
real(kind=8) :: X0, XSUM  
  
call MPI_REDUCE  
(X0, XSUM, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

Global summation of X0 goes to XSUM on #0 process.

```
real(kind=8) :: X0(4)  
  
call MPI_REDUCE  
(X0(1), X0(3), 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

- Global summation of X0(1) goes to X0(3) on #0 process.
- Global summation of X0(2) goes to X0(4) on #0 process.

MPI_ALLREDUCE



- MPI_Reduce + MPI_Bcast
- Summation (of dot products) and MAX/MIN values are likely to utilized in each process
- call MPI_ALLREDUCE
(sendbuf, recvbuf, count, datatype, op, comm, ierr)
 - sendbuf choice I starting address of send buffer
 - recvbuf choice O starting address receive buffer
type is defined by "datatype"
 - count I I number of elements in send/recv buffer
 - datatype I I data type of elements in send/recv buffer
 - op I I reduce operation
 - comm I I commuicator
 - ierr I O completion code

CG method (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 method (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, ...
  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 method (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, ...
  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) + αi p(i)
  r(i) = r(i-1) - αi q(i)
  check convergence |r|

end

```

CG method (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(k)
  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 method (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
RH01 = 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

Program: 1d.f (11/11)

Output by Each Proces

```
!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
```

- Overview
- Distributed Local Data
- Program
- **Results**

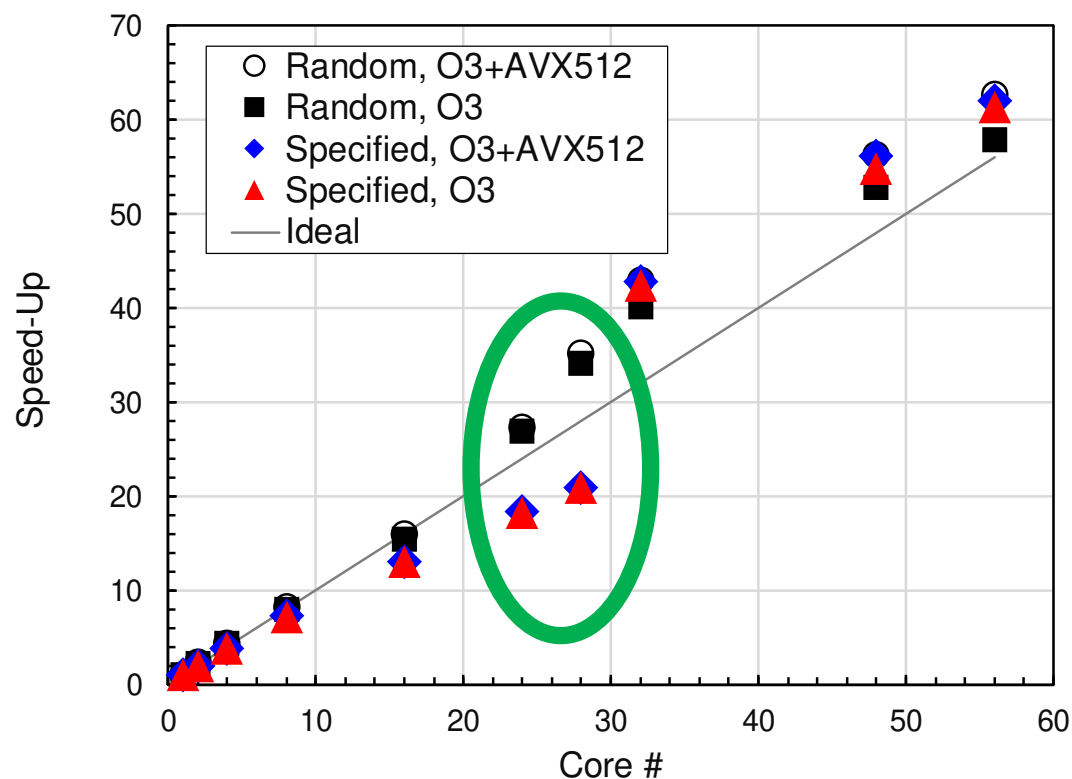
Results: Time for CG Solver, $N=10^6$

Time for 1,000 iterations, Strong Scaling

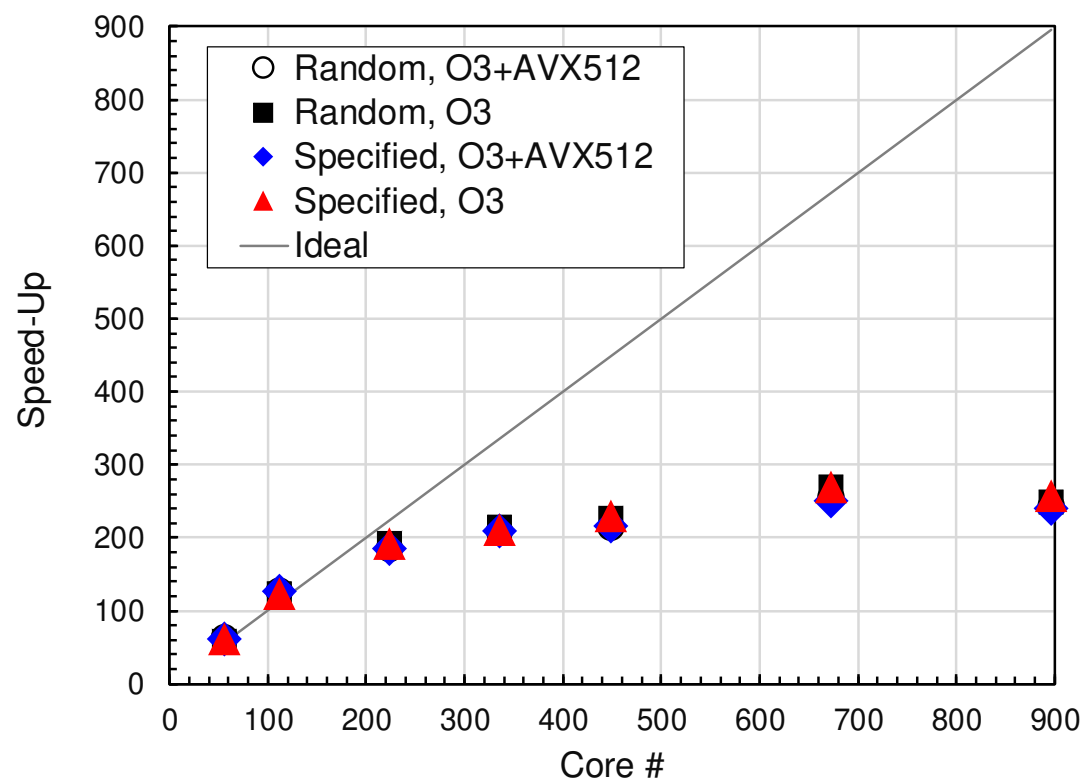
All 56 cores are used if number of nodes is more than 1

Perf. at a single core= 1.00, 5 measurements, Best Case

up to 1 node,
56cores



up to 16 nodes,
896 cores



1-node, 24-cores

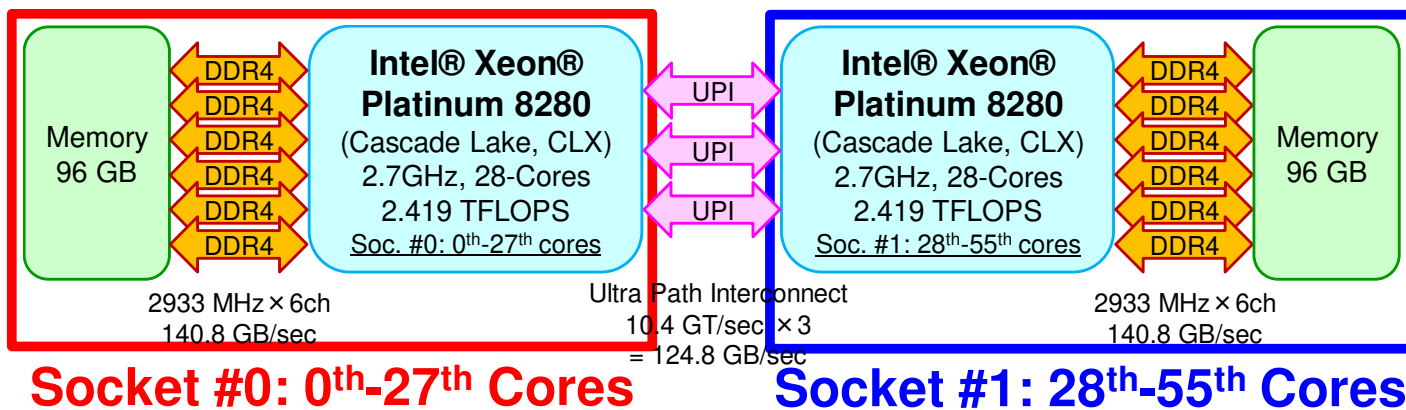
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=24
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst
```

```
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
```

1da: -O3 + AVX512
1db: -O3 Only

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

**24-cores on Socket #0
 are assigned.**
SPECIFIED



1-node, 28-cores

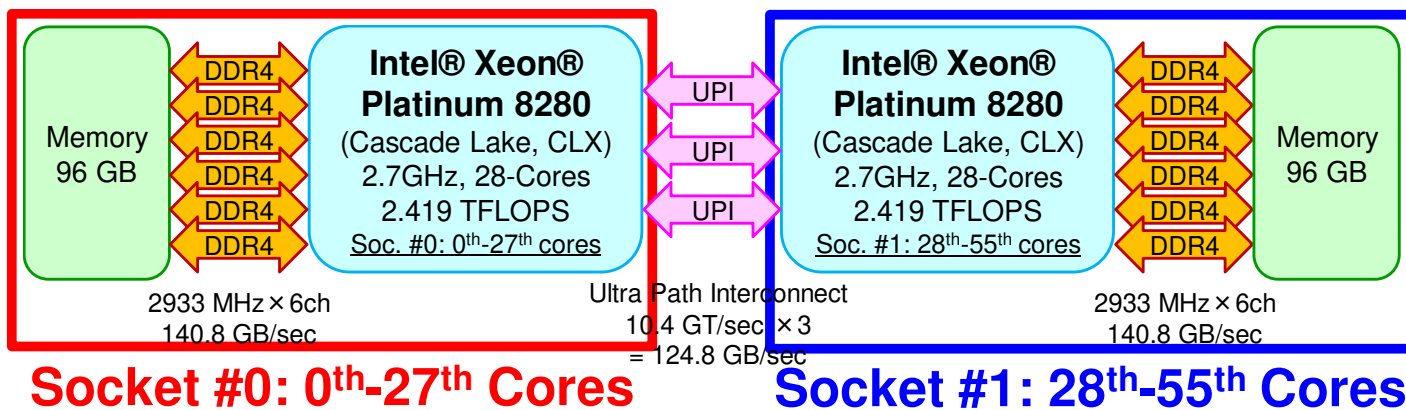
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=28
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o test.lst
```

```
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
```

1da: -O3 + AVX512
1db: -O3 Only

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

**28-cores on Socket #0
are assigned.**
SPECIFIED



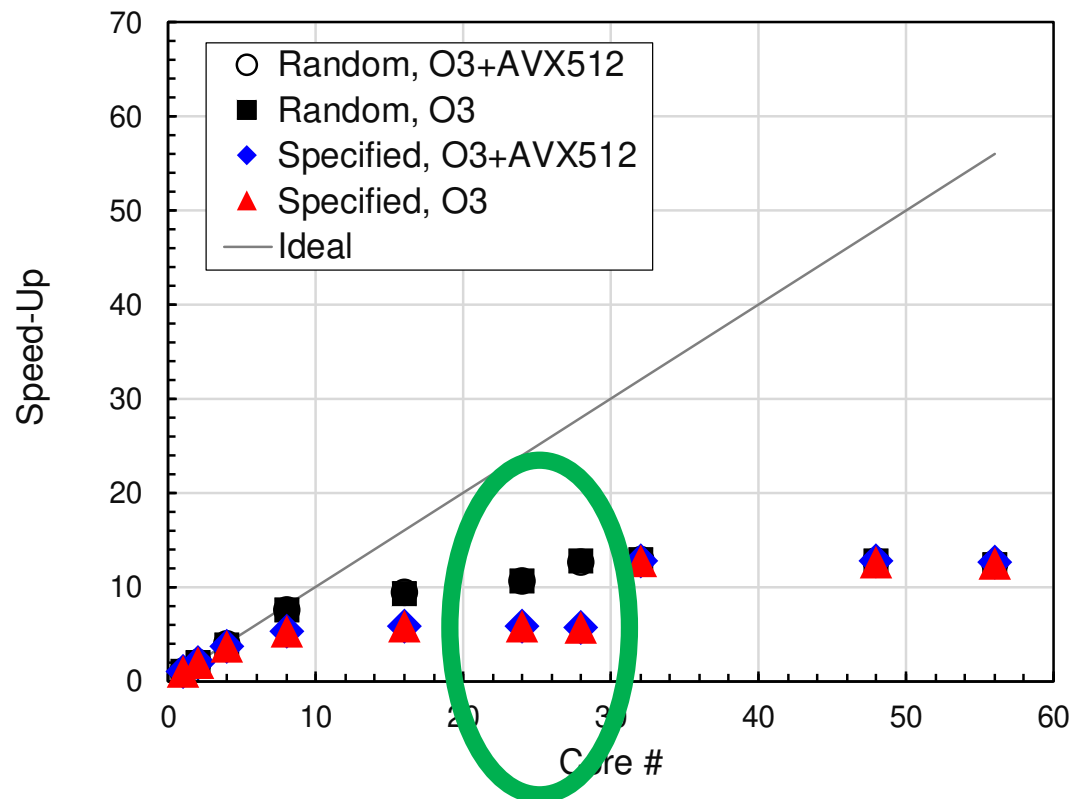
Results: Time for CG Solver, $N=10^7$

Time for 200 iterations, Strong Scaling

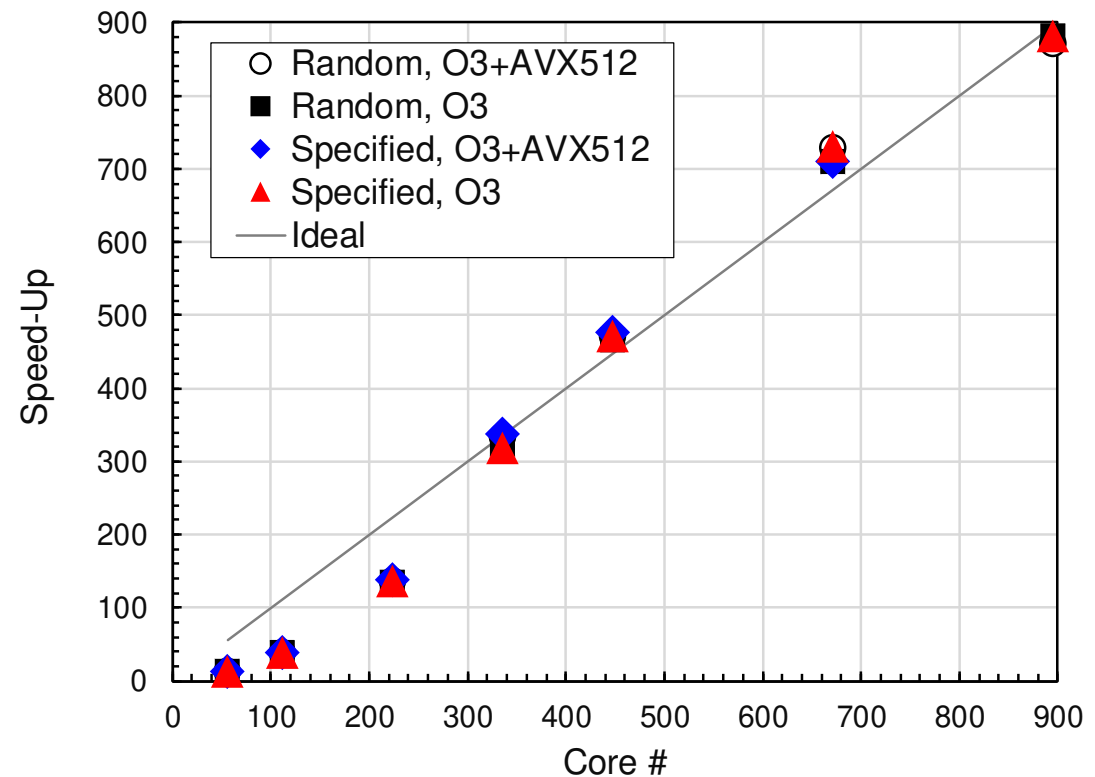
All 56 cores are used if number of nodes is more than 1

Perf. at a single core= 1.00, 5 measurements, Best Case

up to 1 node,
56cores



up to 16 nodes,
896 cores

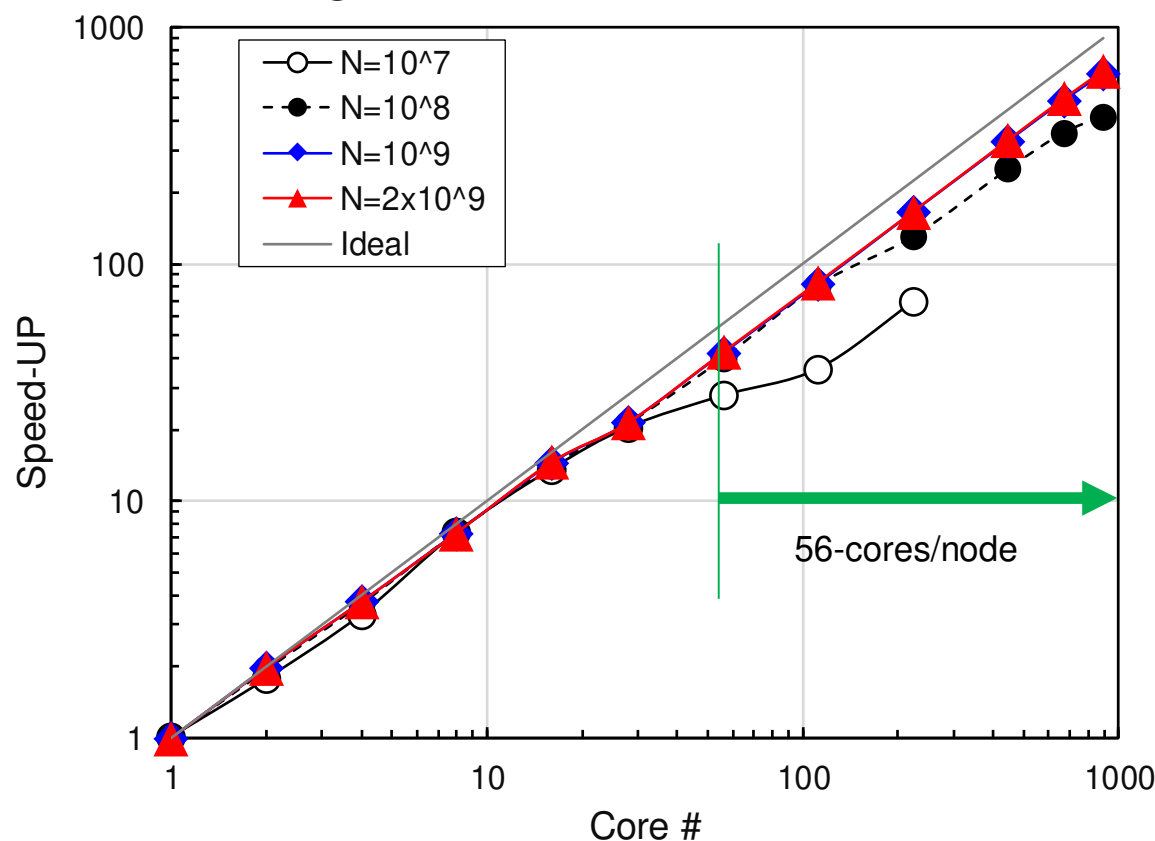


Performance is lower than ideal one

- Time for MPI communication
 - Time for sending data
 - Communication bandwidth between nodes
 - Time is proportional to size of sending/receiving buffers
- Time for starting MPI
 - latency
 - does not depend on size of buffers
 - depends on number of calling, increases according to process #
 - $O(10^0)$ - $O(10^1)$ μsec .
- Synchronization of MPI
 - Increases according to number of processes

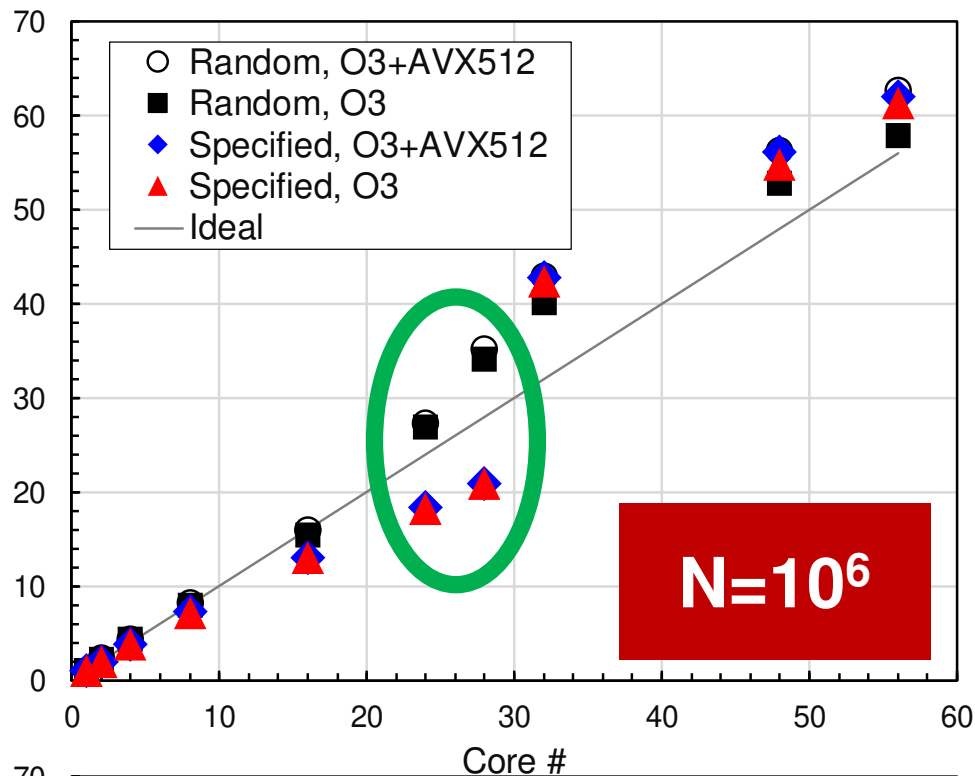
Performance is lower than ideal one (cont.): S1-3

- If computation time is relatively small (N is small in S1-3), these effects are not negligible.
 - If the size of messages is small, effect of “latency” is significant.

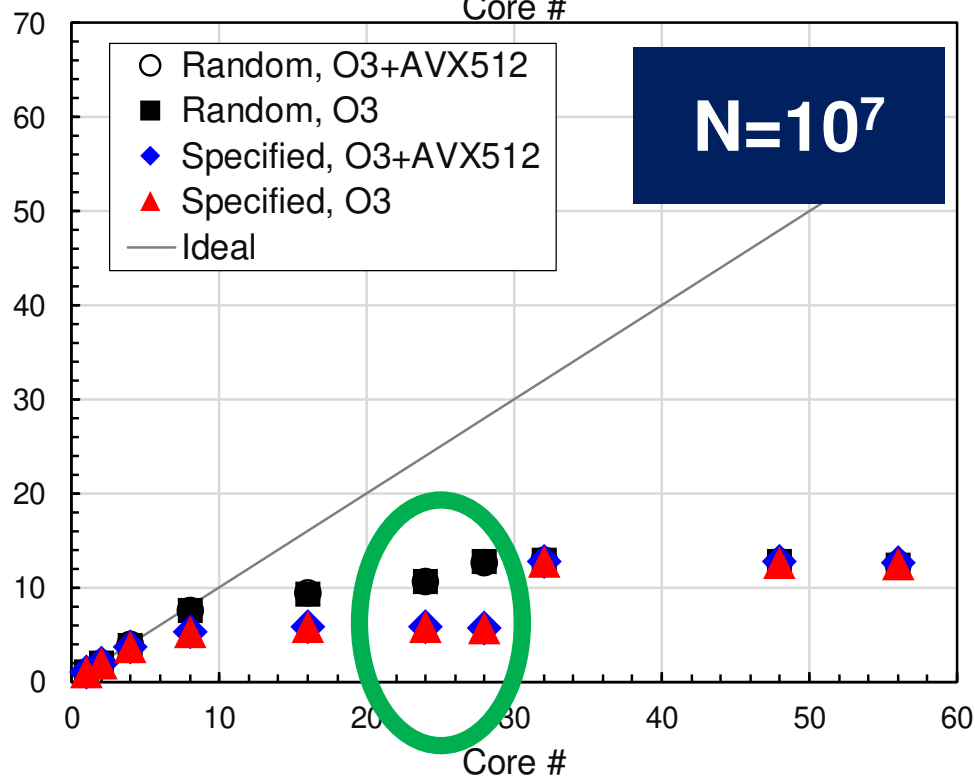


Up to 56 cores (single node)

Speed-Up



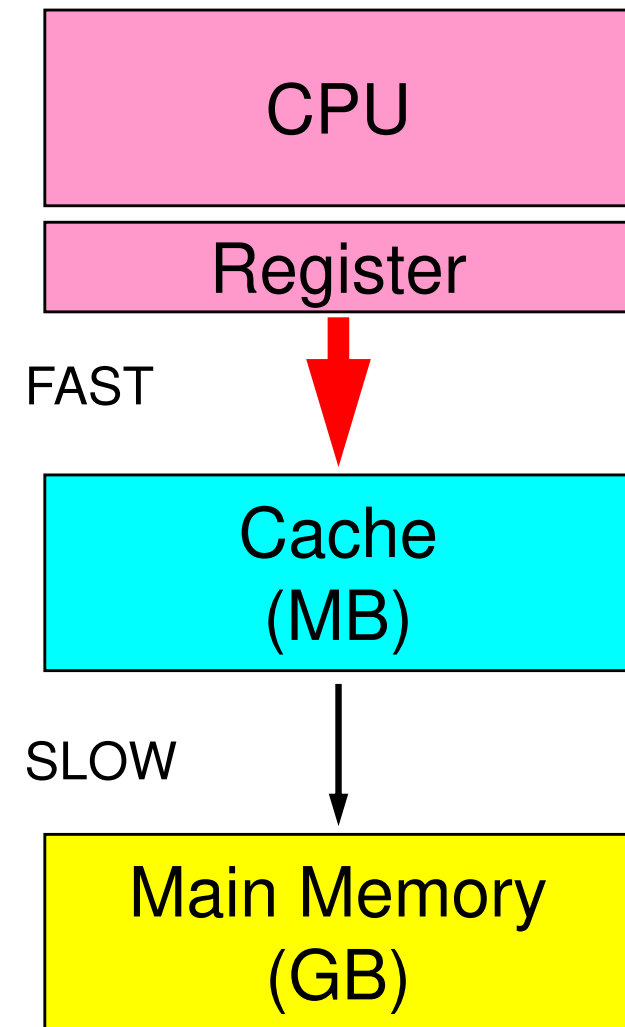
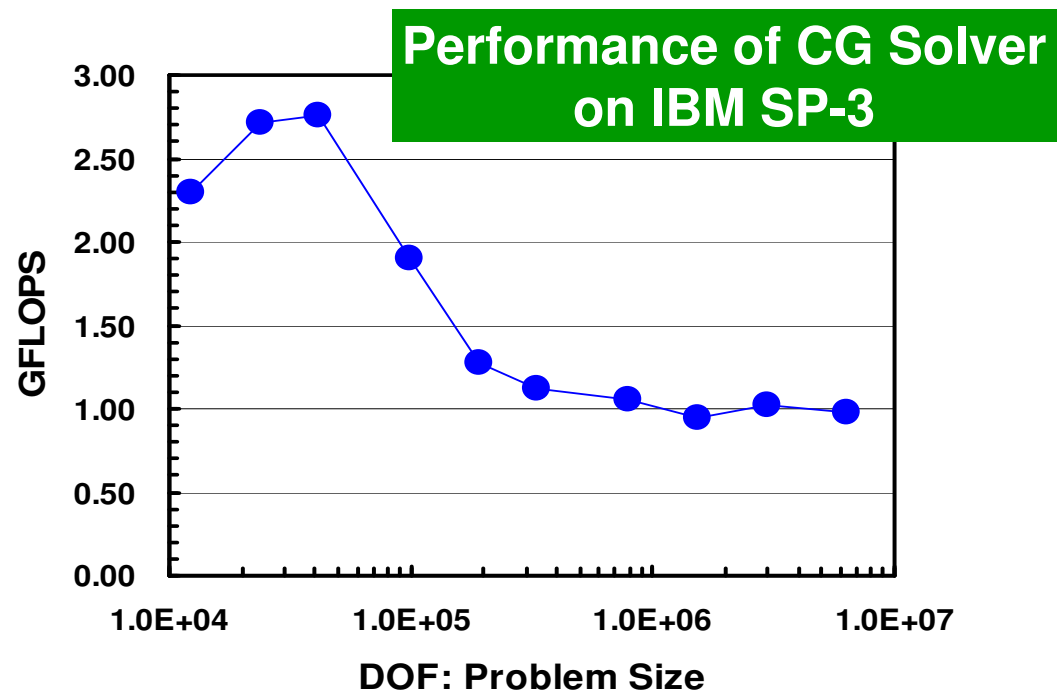
Speed-Up



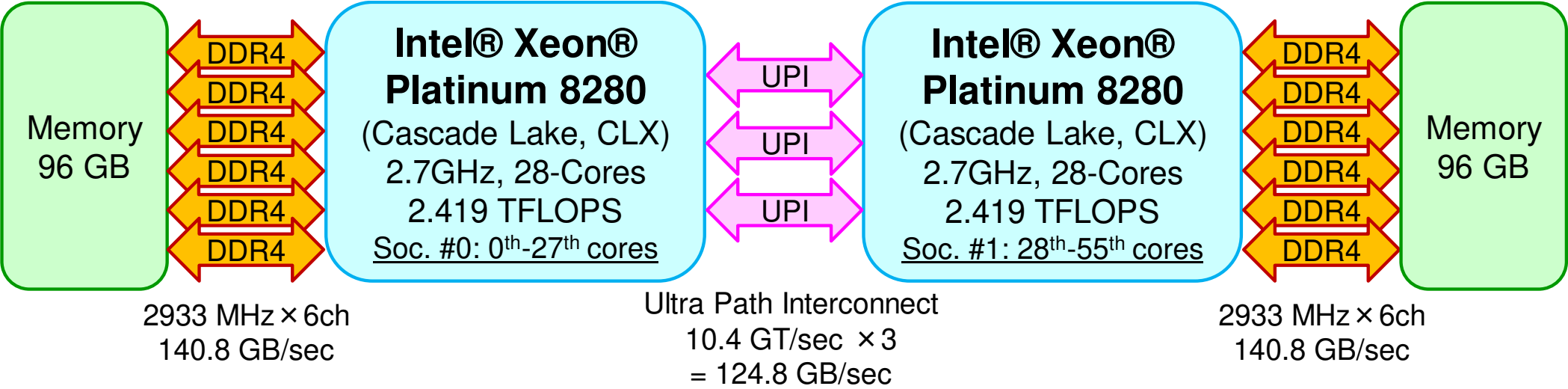
- in a single node (< 56 cores), performance of small cases ($N=10^6$) are rather better.
- Effect of memory contention/saturation (not communication)
- Memory throughput on each soc. is constant for 8+ cores (Stream)
- If problem size is small, cache can be well-utilized. Therefore, effect of memory bandwidth is small.
- Memory/cache of two sockets may be more efficiently utilized in “Random”

If problem size is small, cache can be well-utilized

- In scalar processors, performance for smaller problem is generally better.
 - Cache is well-utilized.

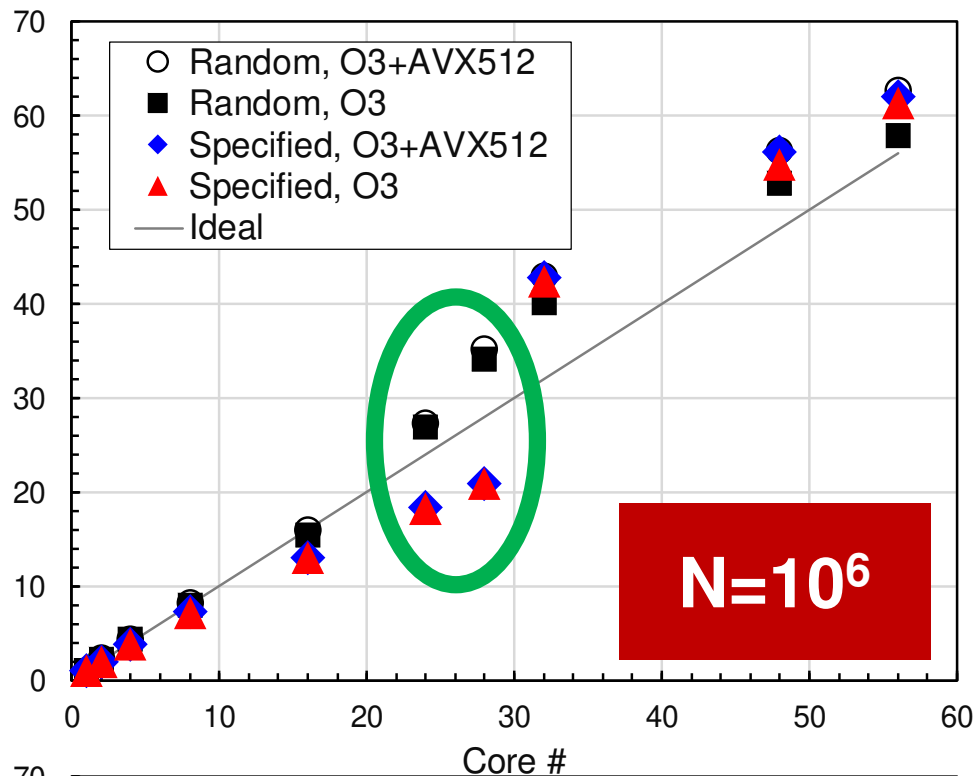


Category	Capacity	X-Way Set Associative	Cache Line
L1\$Data	32 KB/core	8-Way	64B
L1\$Instruction	32 KB/core	8-Way	64B
L2	1.00 MB/core	16-Way	64B
L3	38.5 MB/socket	11-Way	64B



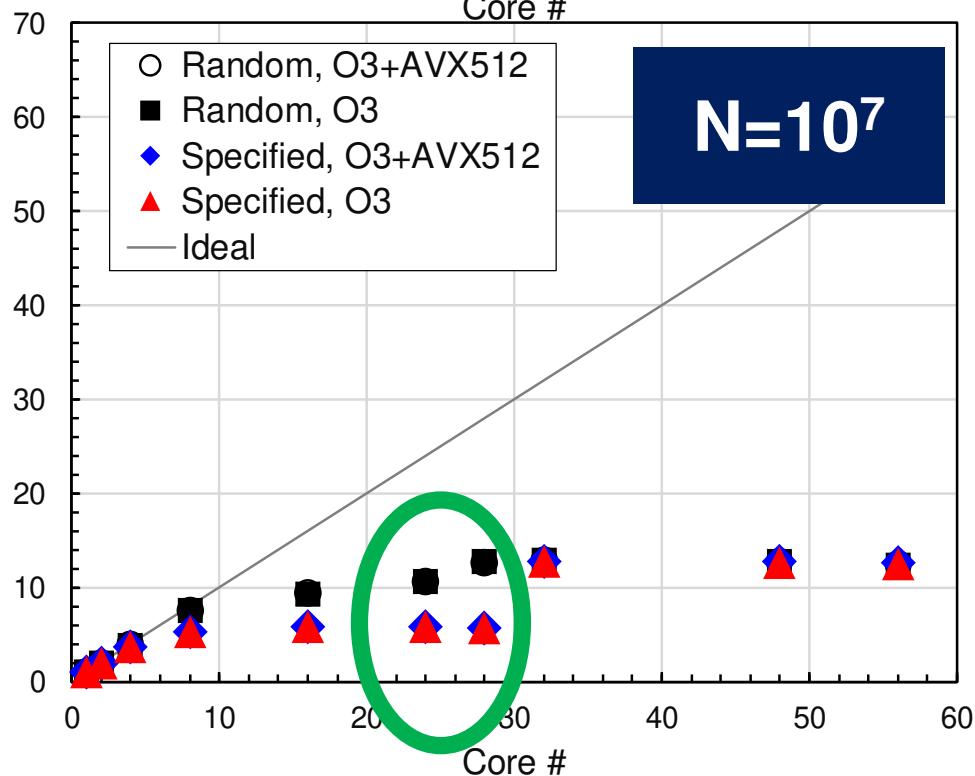
Up to 56 cores (single node)

Speed-Up



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

Speed-Up



- 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: $c(i) = a(i) + s * b(i)$

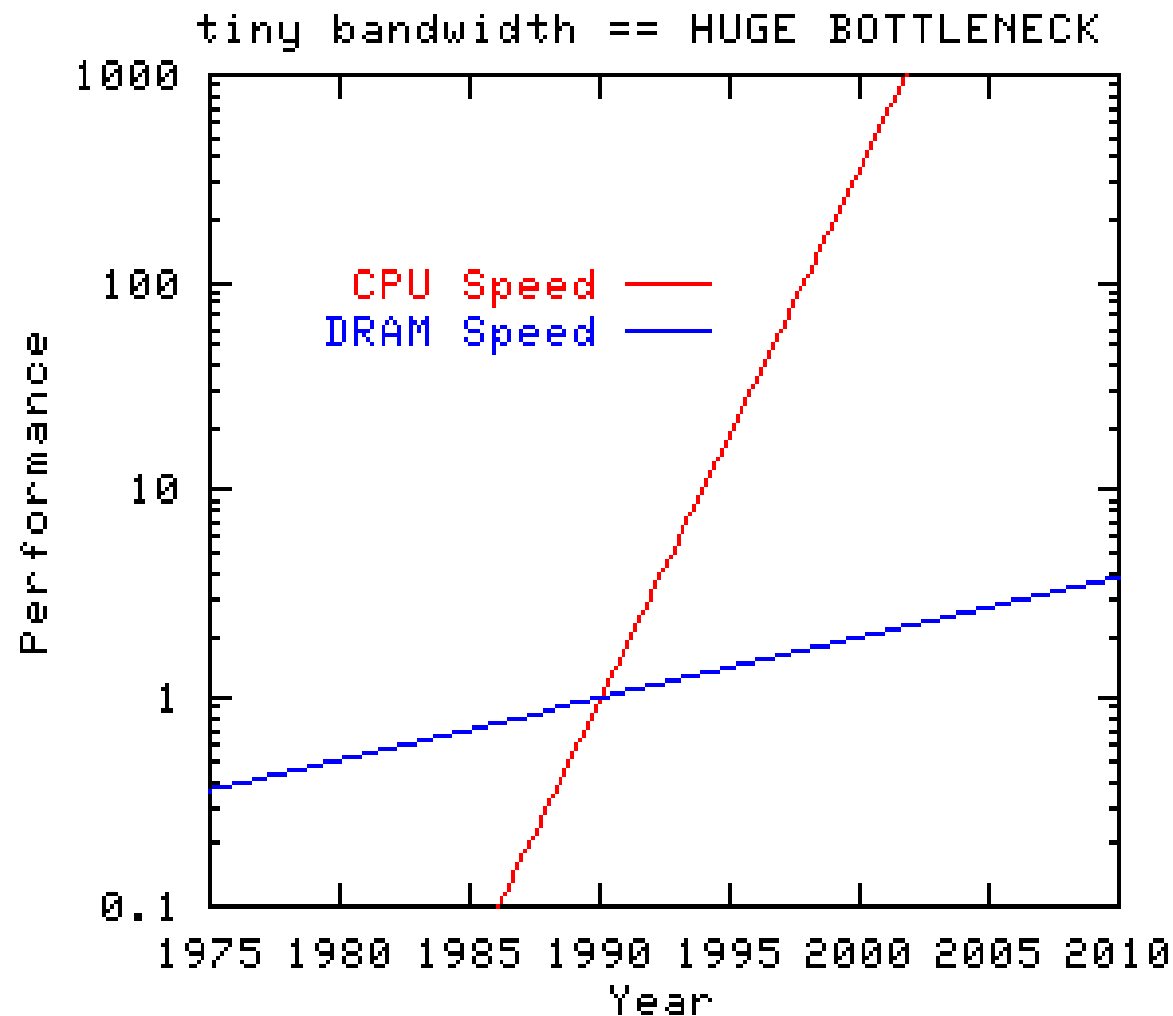
Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word

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

Gap between performance of CPU and Memory



Sparse/Dense Matrices

```
do i= 1, N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    Y(i)= Y(i) + AMAT(k)*X(item(k))
  enddo
enddo
```

```
do j= 1, N
  do i= 1, N
    Y(j)= Y(j) + A(i, j)*X(i)
  enddo
enddo
```

- “X” in RHS
 - Dense: continuous on memory, easy to utilize cache
 - Sparse: continuity is not assured (in-direct access), difficult to utilize cache
 - more “memory-bound”

GeoFEM Benchmark

ICCG in FEM for Solid Mechanics

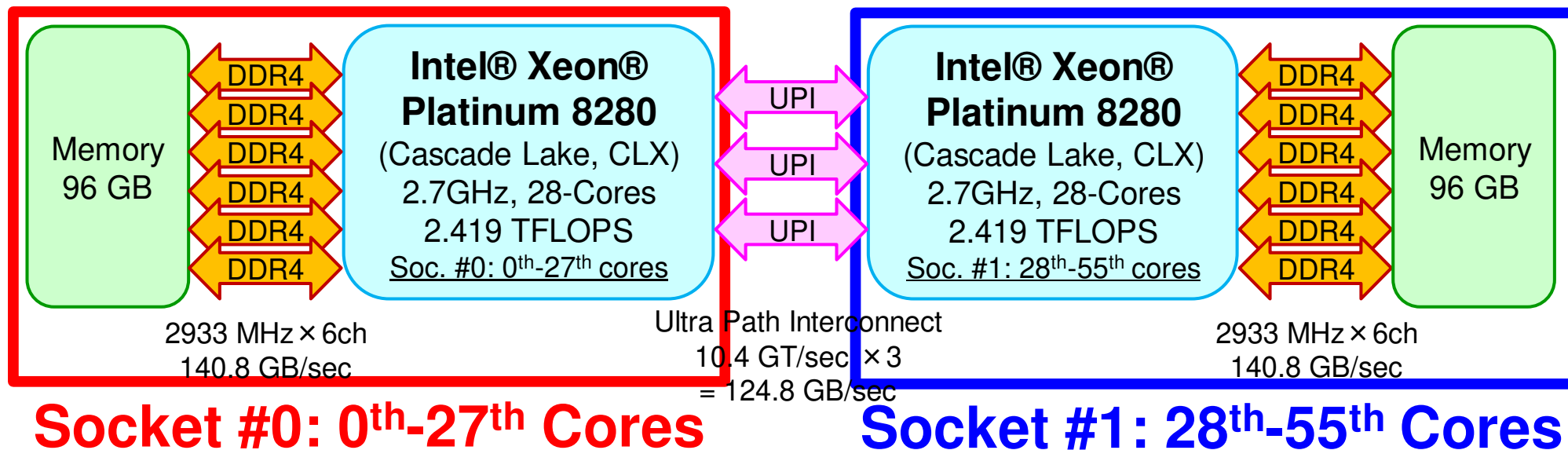
	SR11K/J2	SR16K/M1	T2K	FX10	京
Core #/Node	16	32	16	16	8
Peak Performance (GFLOPS)	147.2	980.5	147.2	236.5	128.0
STREAM Triad (GB/s)	101.0	264.2	20.0	64.7	43.3
B/F	0.686	0.269	0.136	0.274	0.338
GeoFEM (GFLOPS)	19.0	72.7	4.69	16.0	11.0
% to Peak	12.9	7.41	3.18	6.77	8.59
LLC/core (MB)	18.0	4.00	2.00	0.75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

Sparse Linear Solver: Memory-Bound

Copy, Compile and Run

```
>$ cd /work/gt73/t73XXX/pFEM
>$ cp /work/gt73/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=lecture3
#PJM -L node=1
#PJM --mpi proc=1
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o s01.lst
```

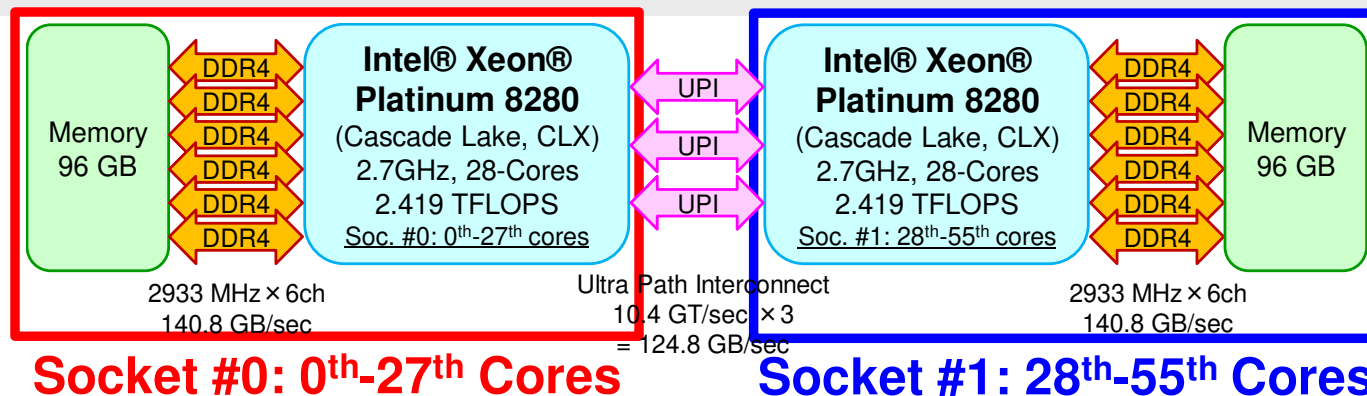
```
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

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=lecture3
#PJM -L node=1
#PJM --mpi proc=16
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o s16.lst
```

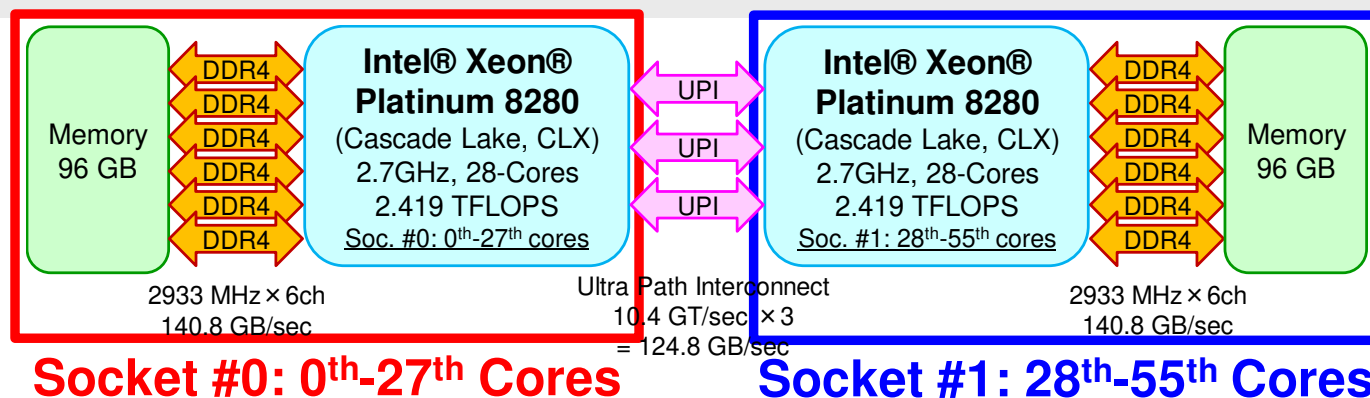
```
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

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

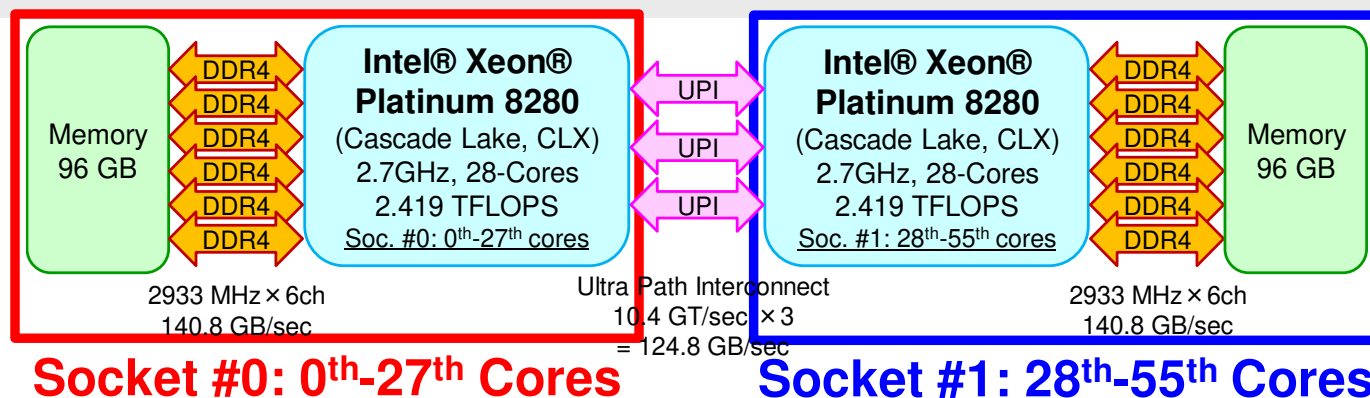
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=32
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o s32.lst
```

```
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

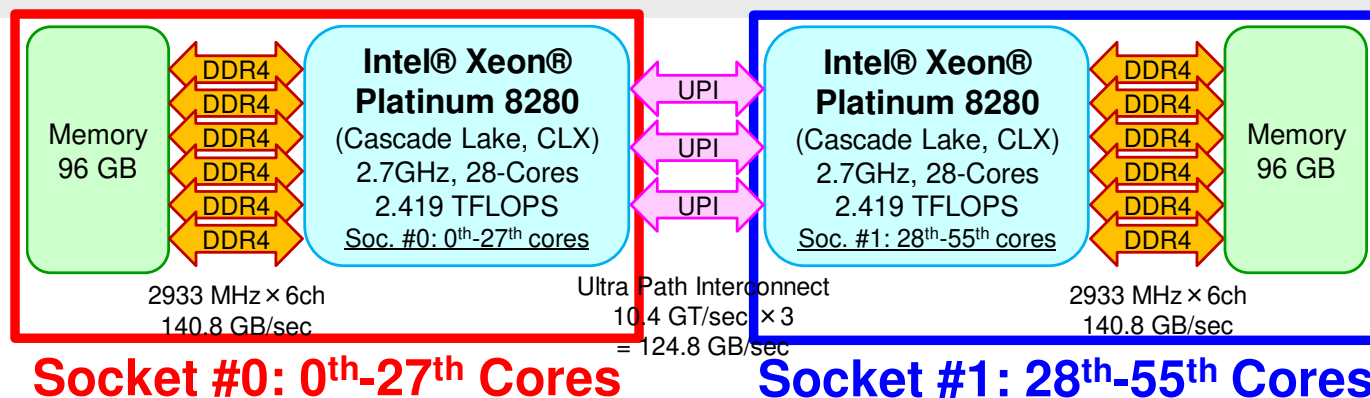
```
#!/bin/sh
#PJM -N "test"
#PJM -L rscgrp=lecture3
#PJM -L node=1
#PJM --mpi proc=48
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o s48.lst
```

```
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=lecture3
#PJM -L node=1
#PJM --mpi proc=56
#PJM -L elapse=00:15:00
#PJM -g gt73
#PJM -j
#PJM -e err
#PJM -o s56.lst
```

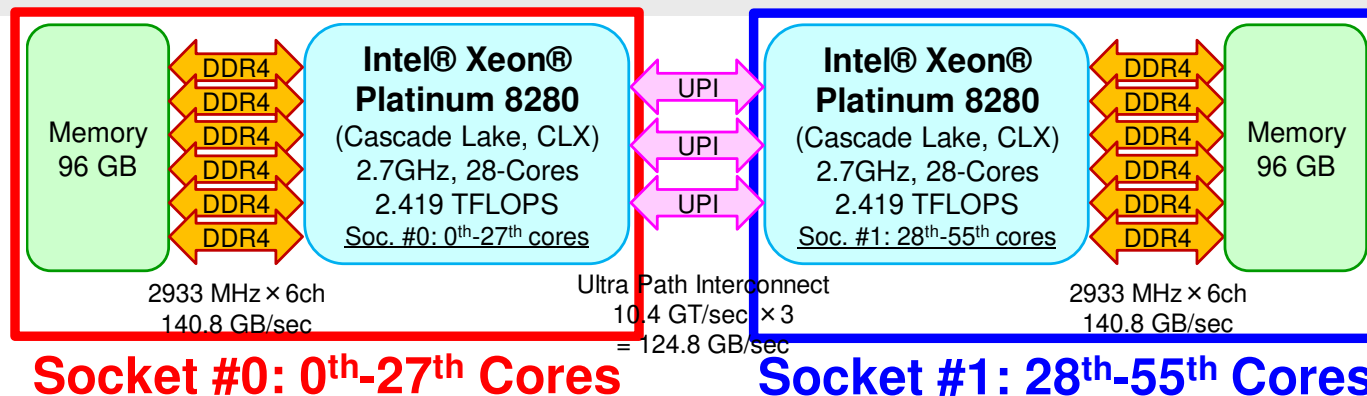
```
mpiexec.hydra -n ${PJM_MPI_PROC} numactl -l ./stream
mpiexec.hydra -n ${PJM_MPI_PROC} ./stream
```

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
```

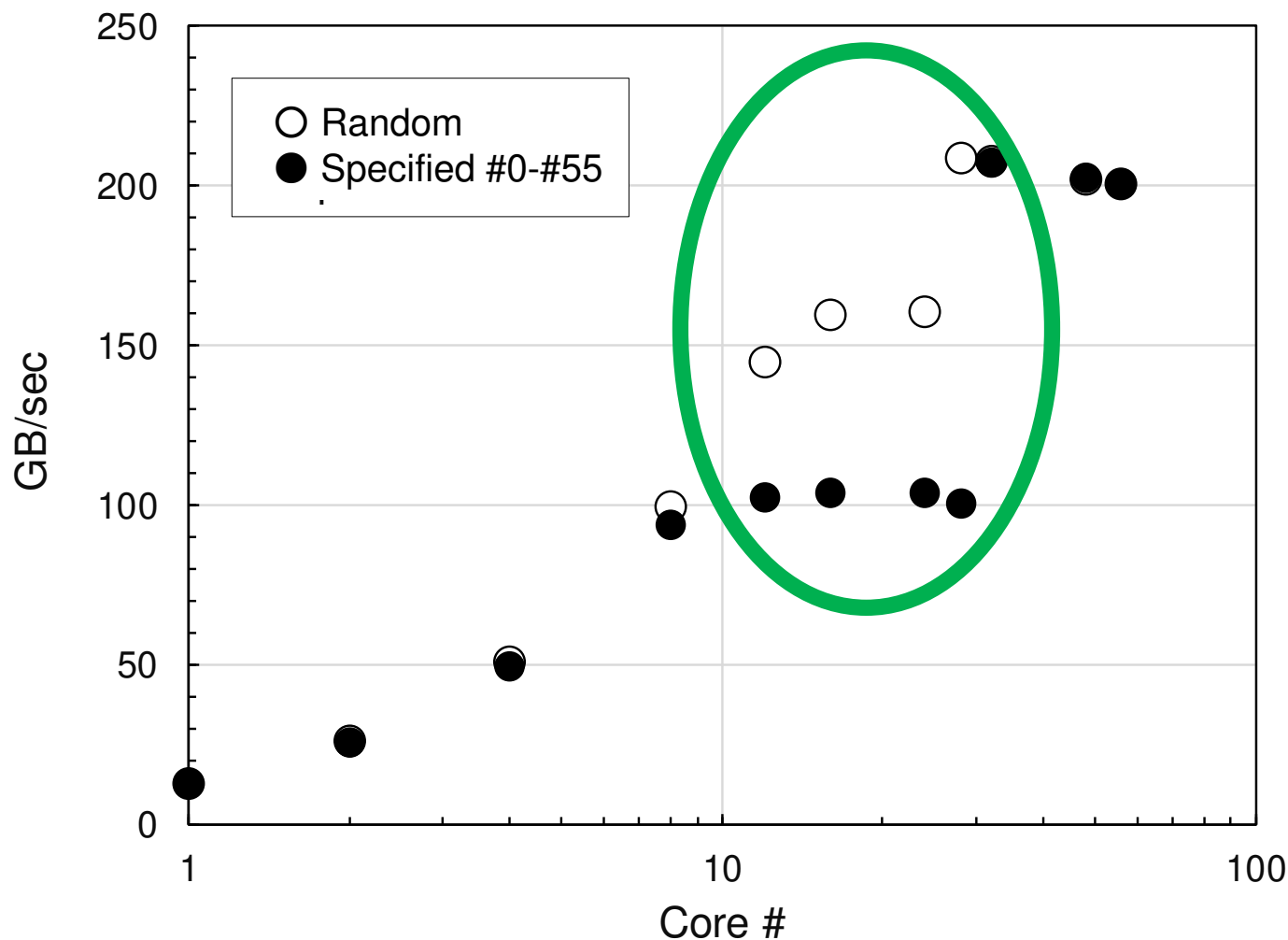
Core are specified



Triad on a Single Node of OBCX

Peak is 281.57 GB/sec.

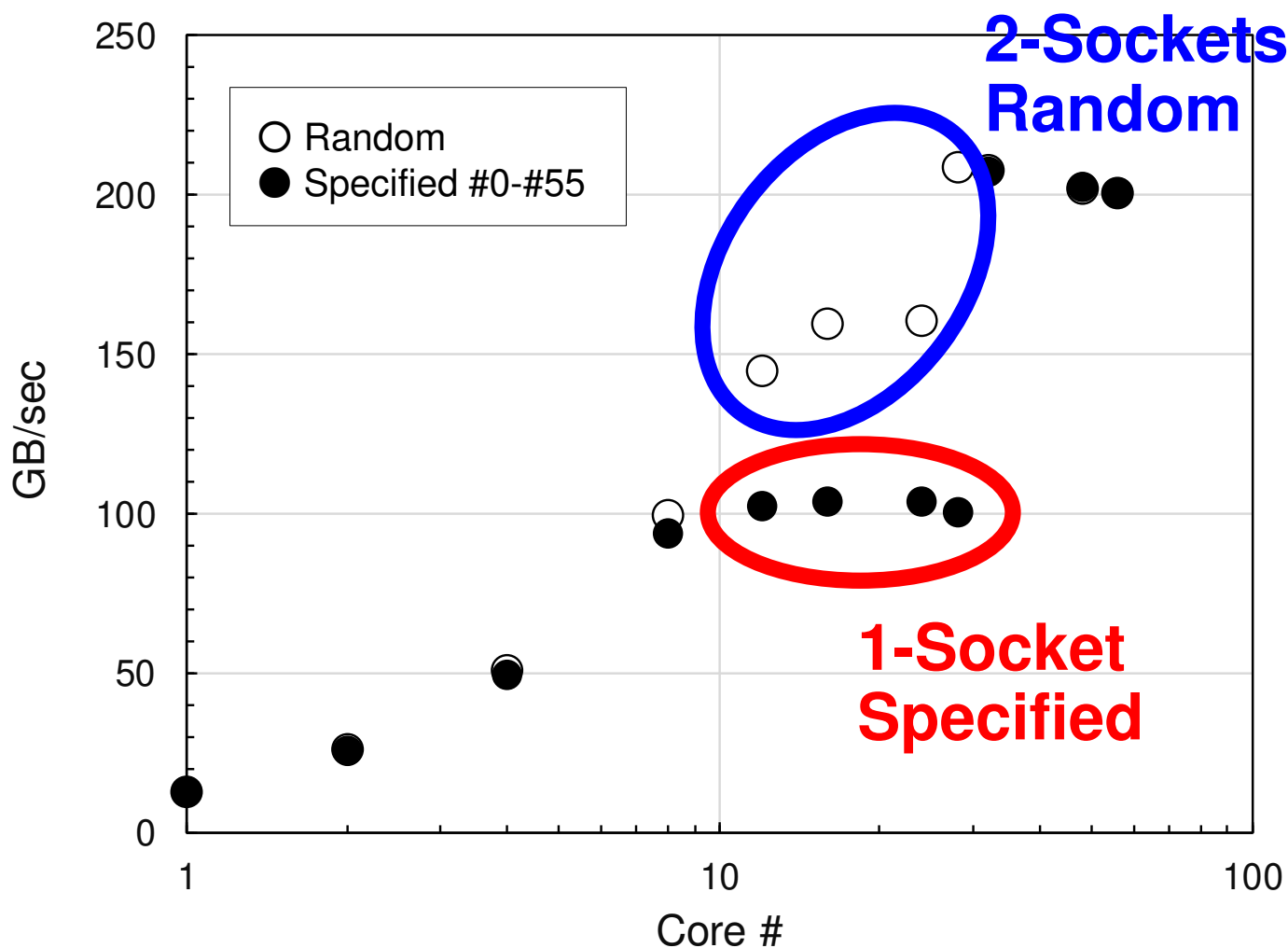
Memory/cache may be more efficiently utilized in “Random”
(12-28 cores)

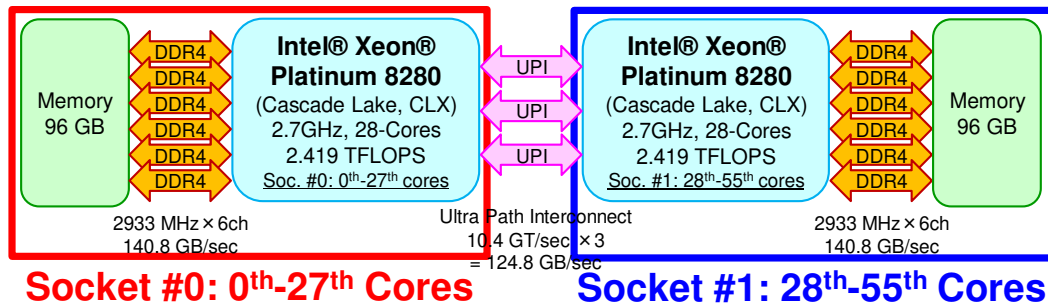


Triad on a Single Node of OBCX

Peak is 281.57 GB/sec.

Memory/cache may be more efficiently utilized in “Random”
(12-28 cores)





Triad on a Single Node of OBCX

Speed-Up based on Performance with 1 core

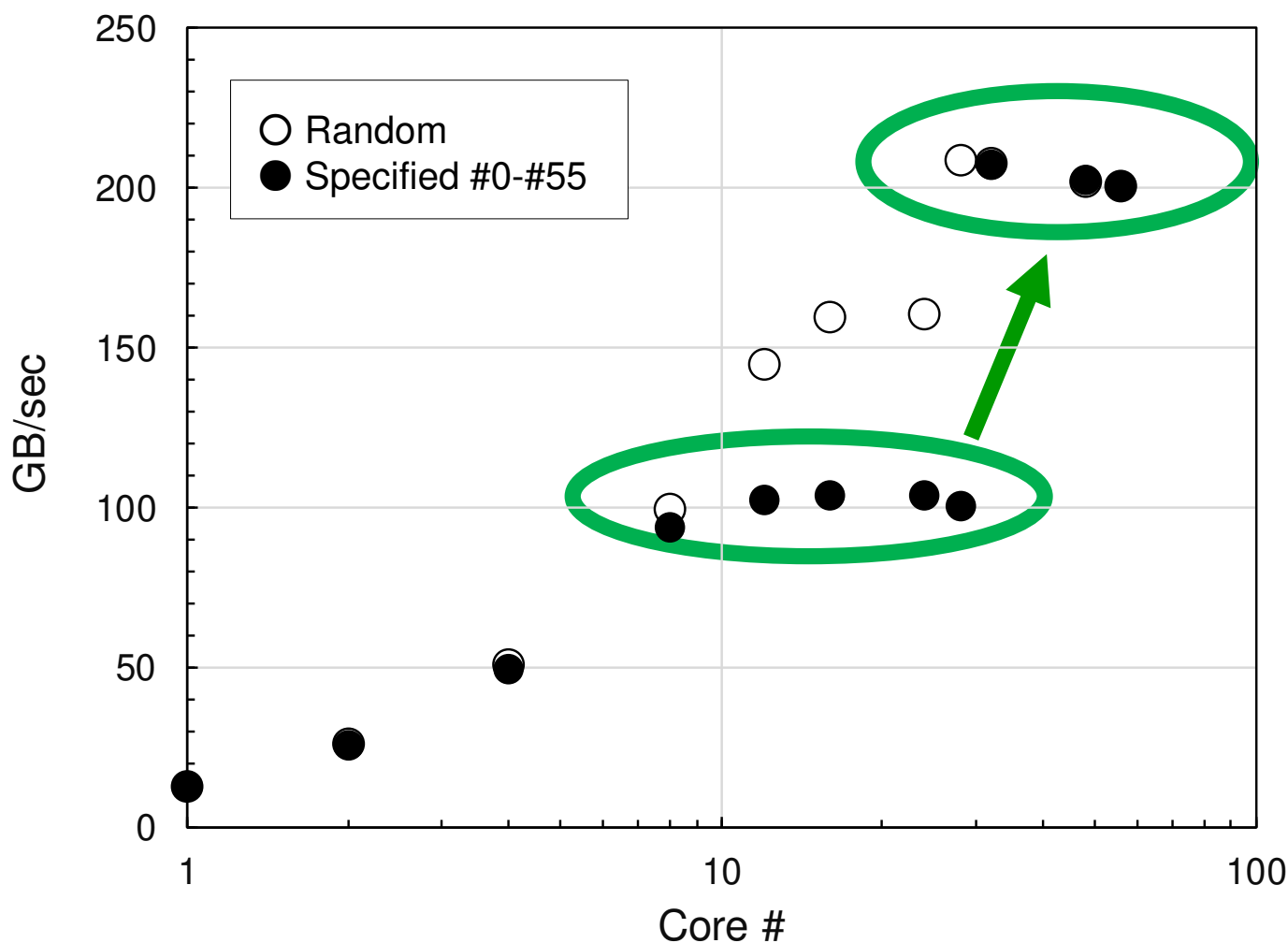
Socket #	Core #	Random	Specified
1	1	1.000	1.000
	2	1.998	1.963
	4	3.912	3.809
	6	5.792	5.682
	8	7.615	7.177
	12	11.089	7.844
	16	12.214	7.968
	24	12.278	7.945
2	28	15.962	7.705
	32	15.901	15.862
	48	15.452	15.497
	56	15.370	15.358

- Number of Memory Channels per Socket = 6
 - 6 memory chips can be loaded on each socket
 - Linear speed-up up to 6 cores (processes) on each socket

Triad on a Single Node of OBCX

Peak is 281.57 GB/sec.

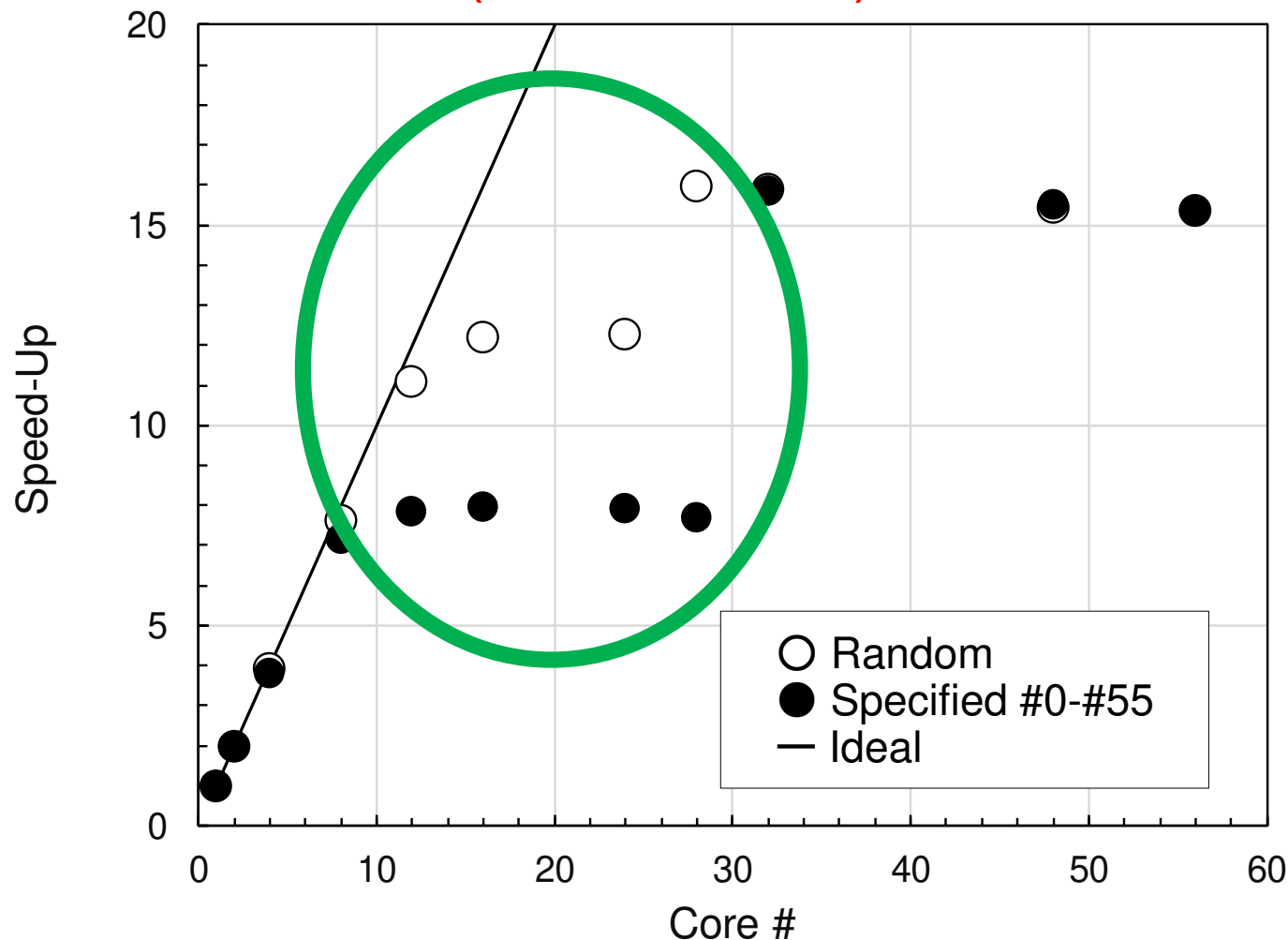
● : Memory BW is constant with 8-28 cores (saturated),
Doubled with 32-56 cores



Triad on a Single Node of OBCX

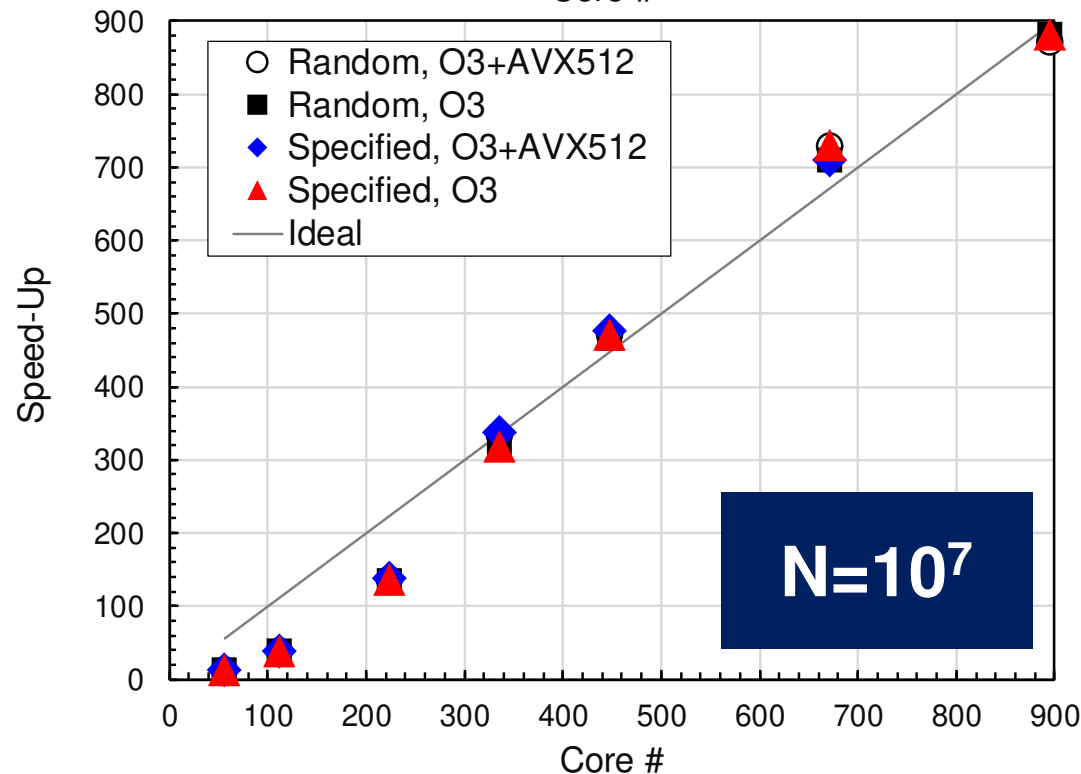
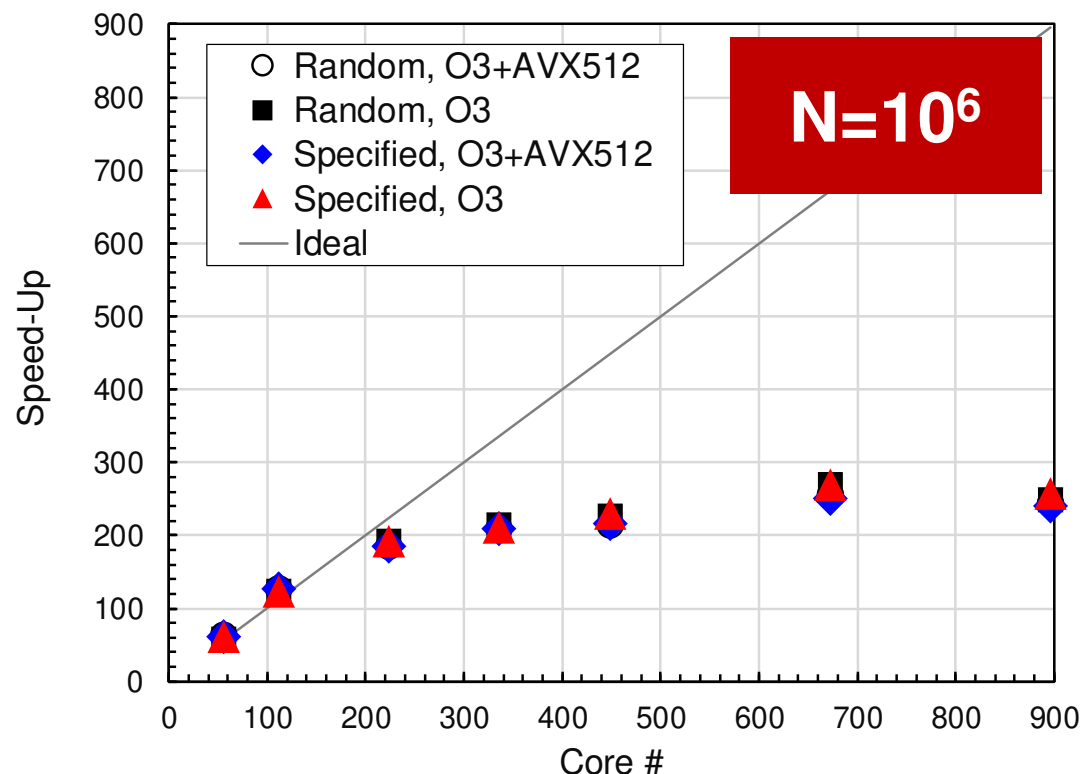
Peak is 281.57 GB/sec.

Memory/cache may be more efficiently utilized in “Random”
(12-28 cores)



Exercises

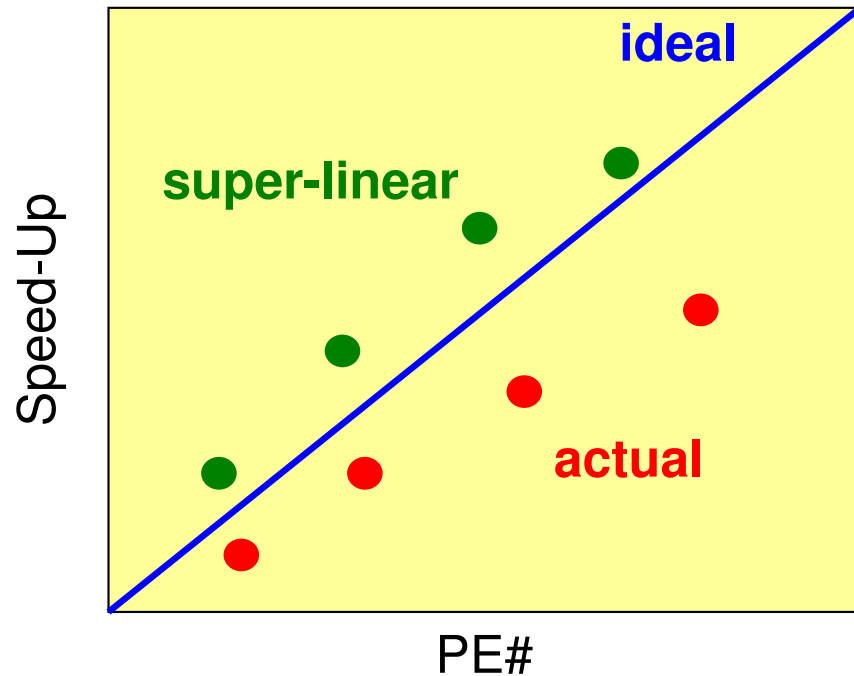
- Running the code
- Try various number of processes (1-56)
- OpenMP-version and Single PE version are available
 - Fortran, C
 - Web-site of STREAM
 - <http://www.cs.virginia.edu/stream/>



Up to 896 cores (16-nodes)

- Performance at a Single Core= 1.00
- Performance of $N=10^6$ case decreases, as node# increases.
- Performance of $N=10^7$ becomes close to ideal one gradually, and **superlinear situation occurs at 672 cores (12 nodes).**

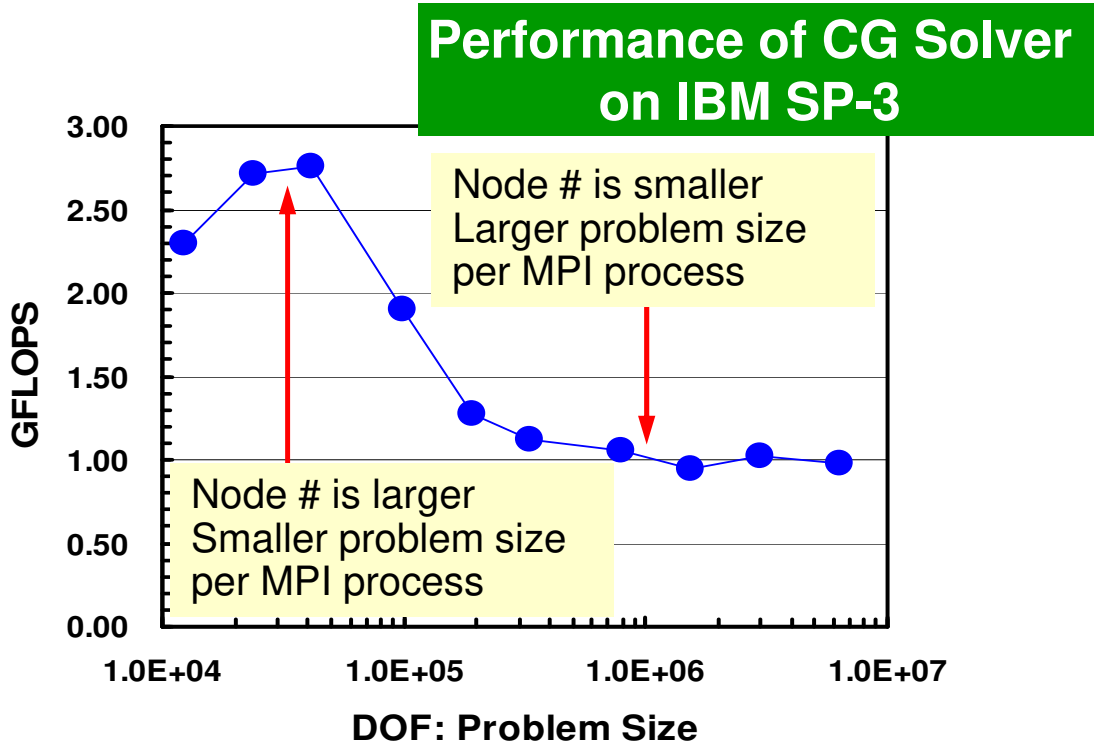
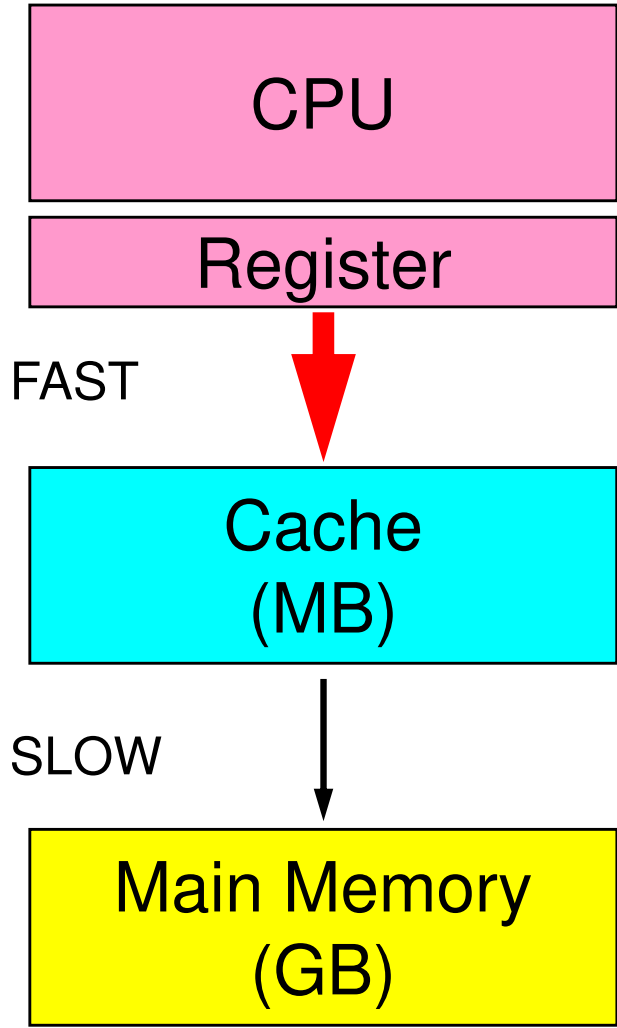
Super-Linear in Strong Scaling



- In strong scaling case where entire problem size is fixed, performance is generally lower than the ideal one due to communication overhead.
- But sometimes, actual performance may be better than the ideal one. This is called “super-linear”

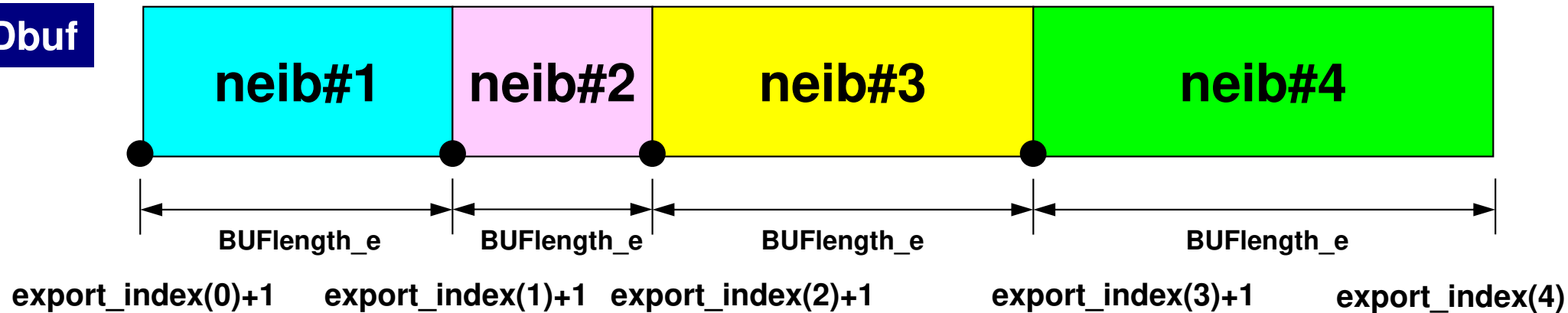
Why does “Super-Linear” happen ?

- Effect of Cache
- In scalar processors, performance for smaller problem is generally better.
 - Cache is well-utilized.



Memory Copy is expensive (1/2)

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)
```

Memory Copy is expensive (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

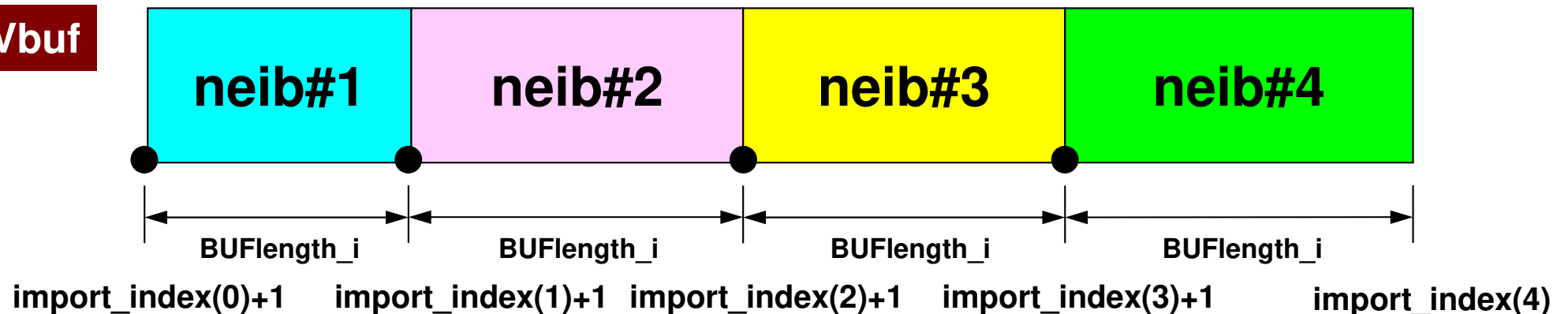
call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

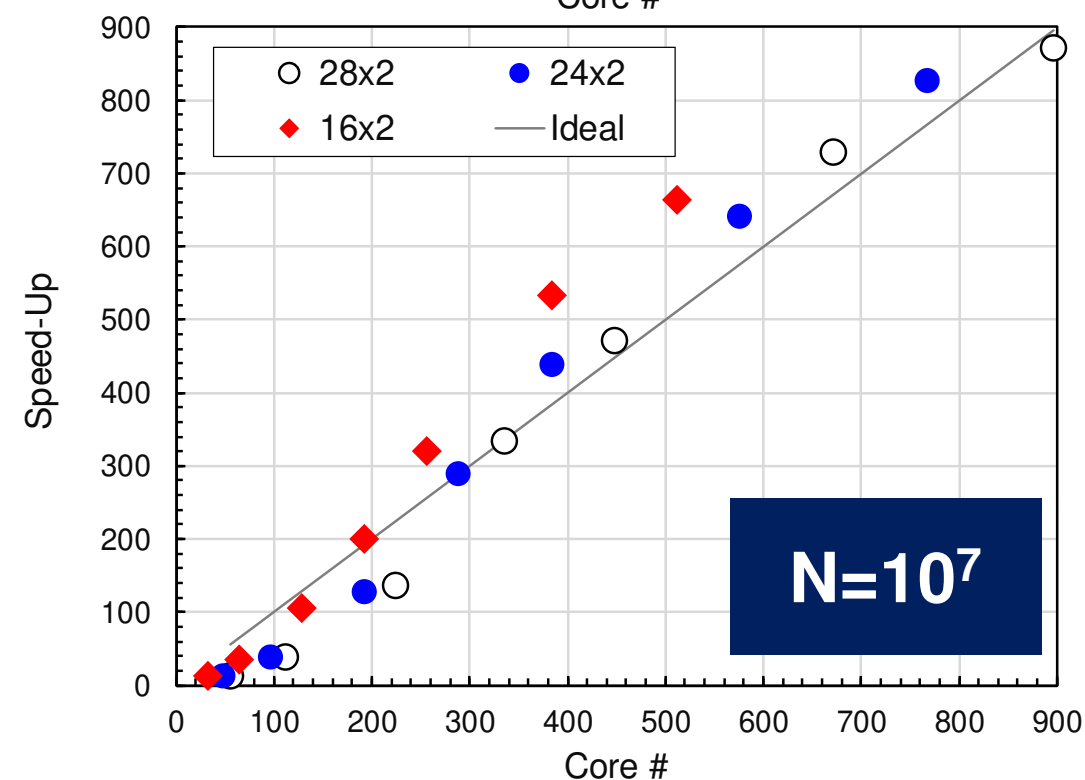
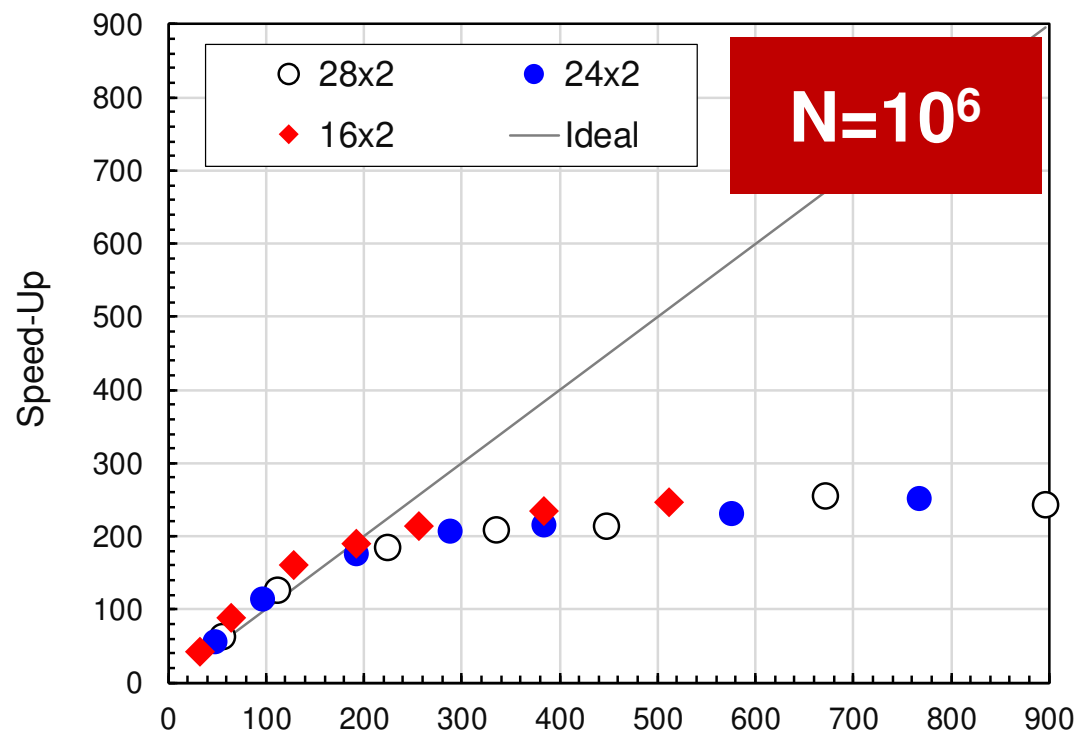
do neib= 1, NEIBPETOT
  do k= import_index(neib-1)+1, import_index(neib)
    kk= import_item(k)
    VAL(kk)= RECVbuf(k)
  enddo
enddo

```

Copied from receiving buffer

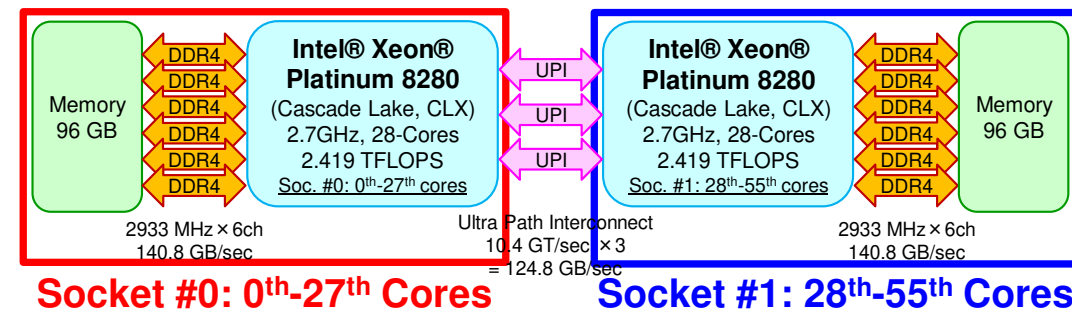
RECVbuf

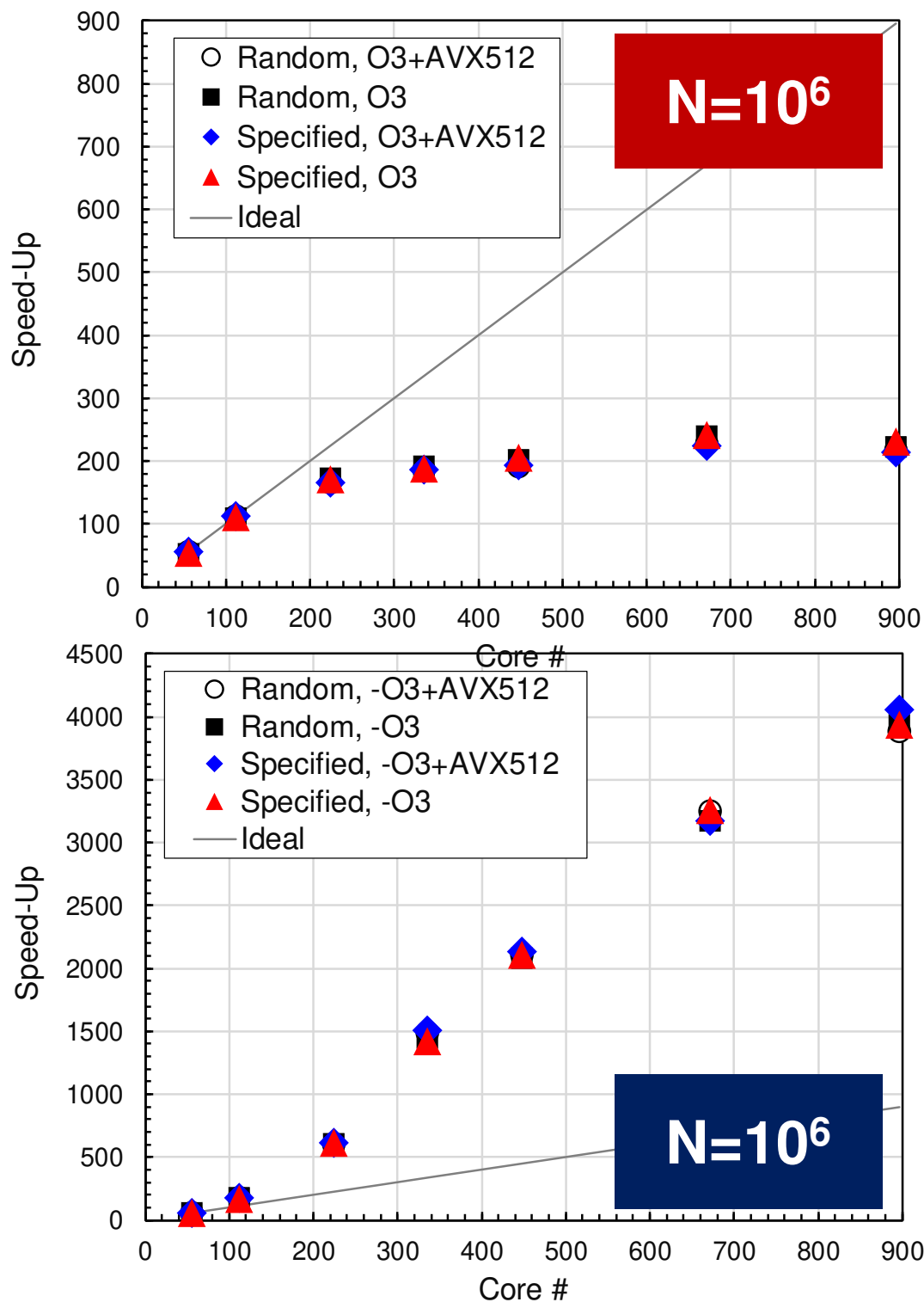




Up to 16-nodes Random, O3+AVX512

- MPI Processes per Node
 - 16x2, 24x2, 28x2
- $N=10^6$
 - No diff. if core# is larger
- $N=10^7$
 - At same core#, 16x2 is the best: Memory is more efficiently utilized

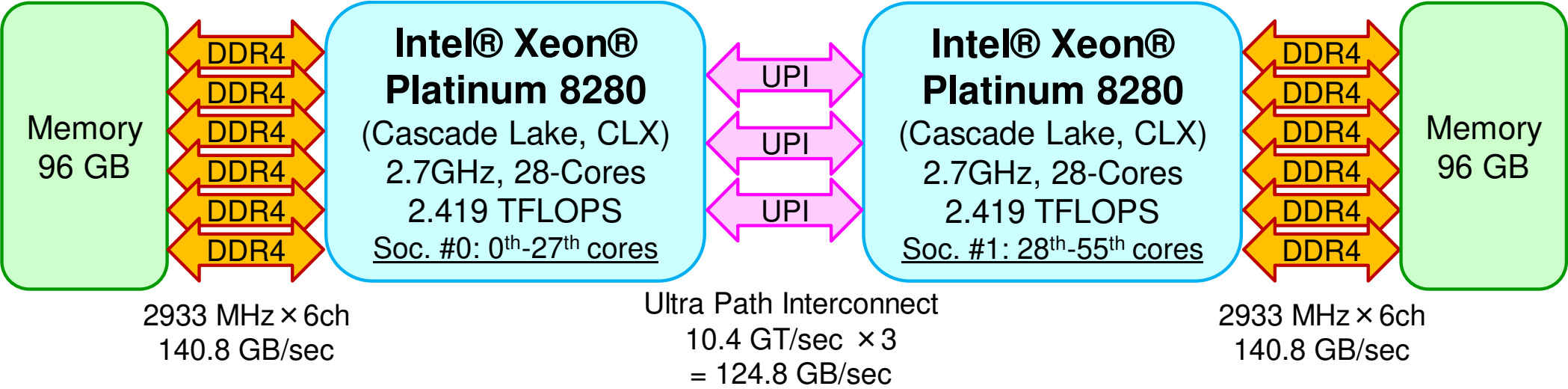




Up to 896 cores (16-nodes)

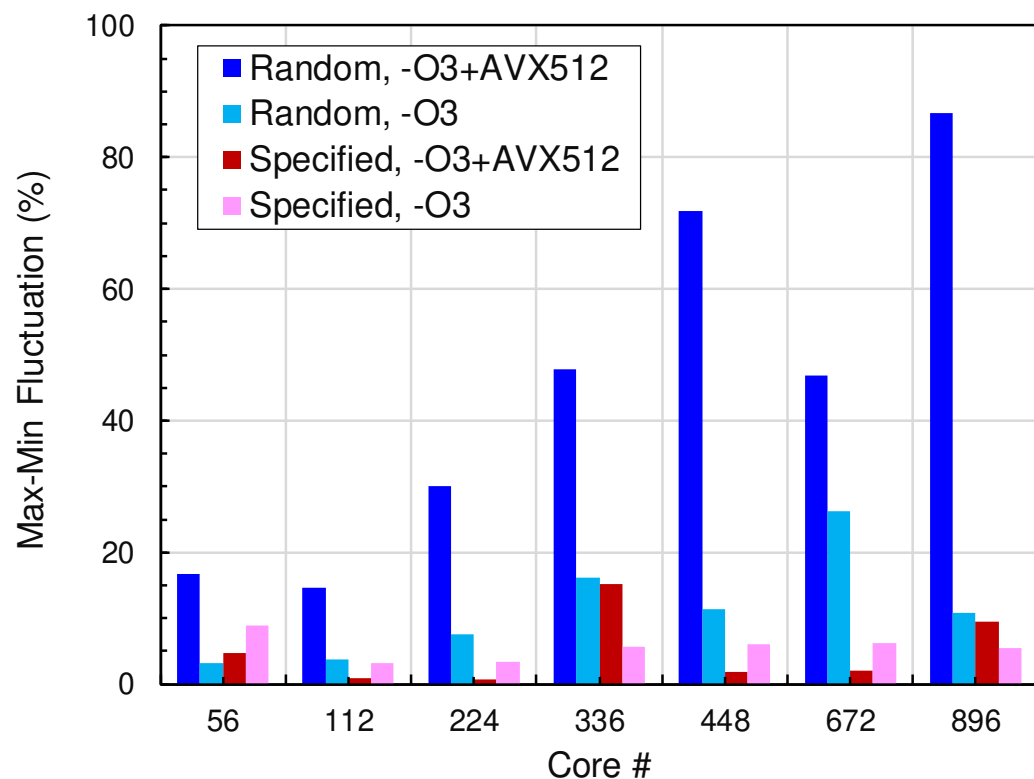
- Performance at 56 cores (= single node) = 56.0
- It is reasonable to evaluate strong scalability for multiple nodes based on the performance by a node with 56 cores
- L2 cache: 1MB/core
- L3: 38.5MB/socket (shared)
- $N=10^6$
 - Significant Comm. Overhead
- $N=10^7$
 - Very Superlinear

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

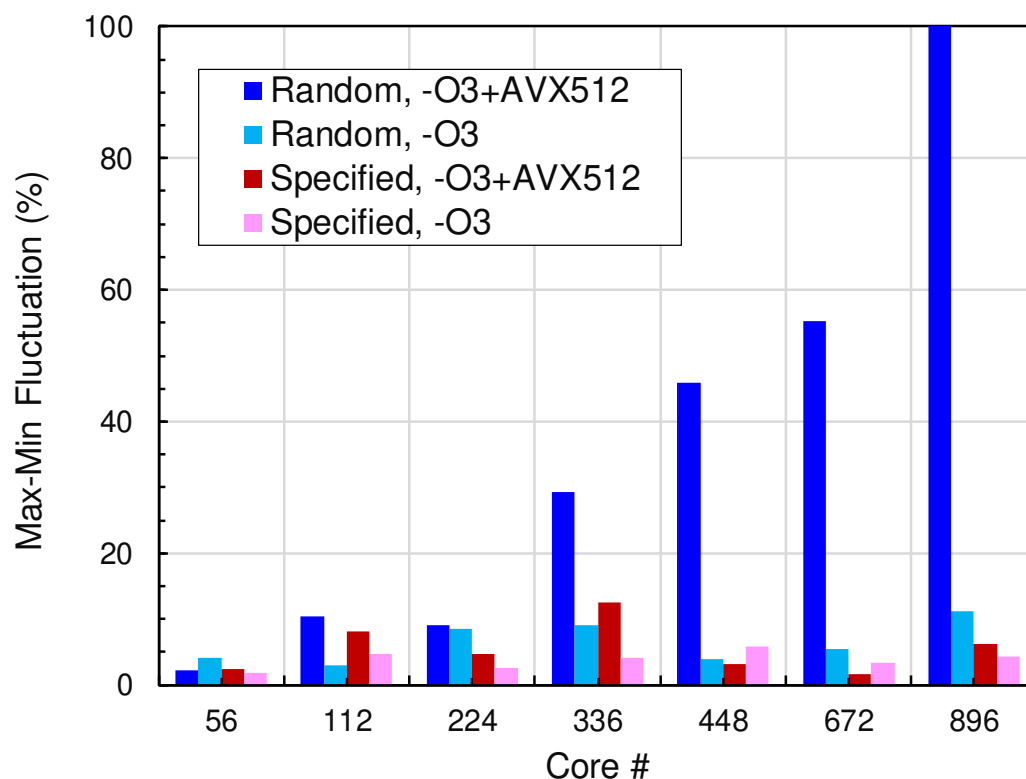


Fluctuation of Computation Time for 5 measurements

$N=10^6$



$N=10^6$



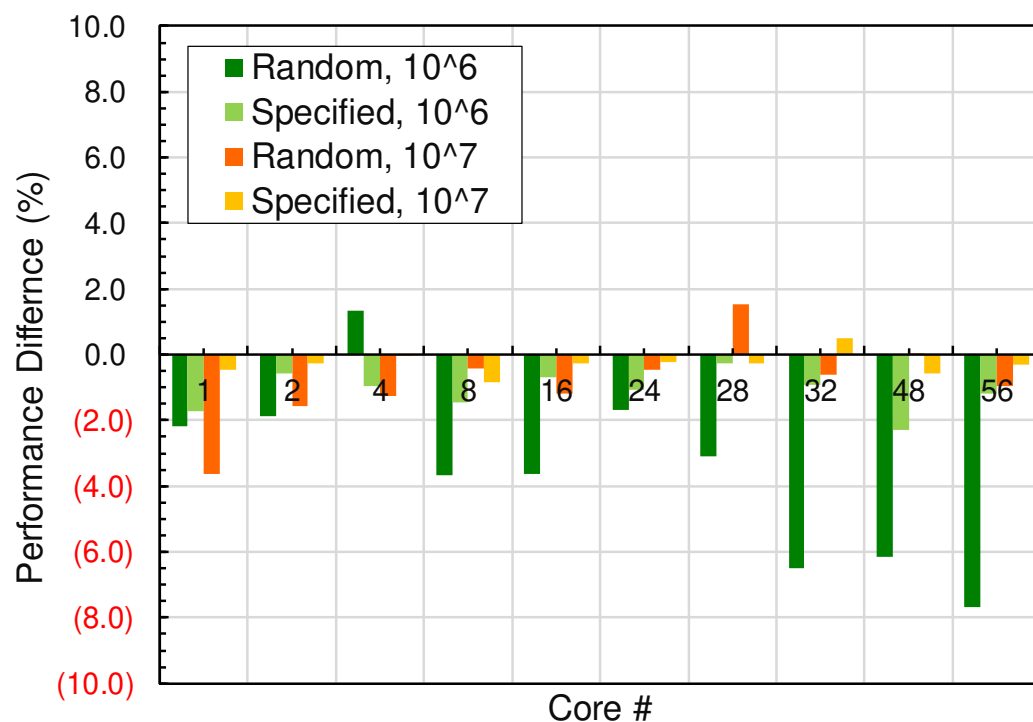
“-O3+AVX512” and “-O3”

Best Case of 5 Measurements

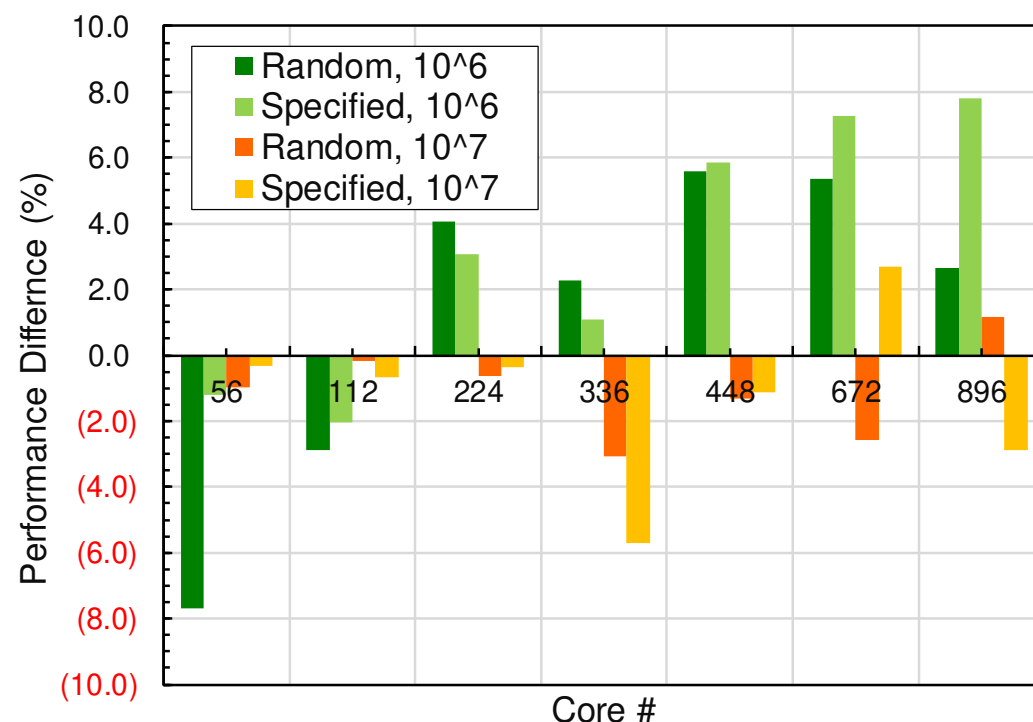
+: -O3 is better, -: -O3 is worse

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

**up to 1 node,
56cores**



**up to 16 nodes,
896 cores**



Summary: Parallel FEM

- Proper design of data structure of distributed local meshes.
- Open Technical Issues
 - Parallel Mesh Generation, Parallel Visualization
 - Parallel Preconditioner for Ill-Conditioned Problems
 - Large-Scale I/O

Distributed Local Data Structure for Parallel Computation

- Distributed local data structure for domain-to-domain communications has been introduced, which is appropriate for such applications with sparse coefficient matrices (e.g. FDM, FEM, FVM etc.).
 - SPMD
 - Local Numbering: Internal pts to External pts
 - Generalized communication table
- Everything is easy, if proper data structure is defined:
 - Values at boundary pts are copied into sending buffers
 - Send/Recv
 - Values at external pts are updated through receiving buffers

If numbering of external nodes is continuous in each neighboring process ...

	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.): No Copy for RECV

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)

```

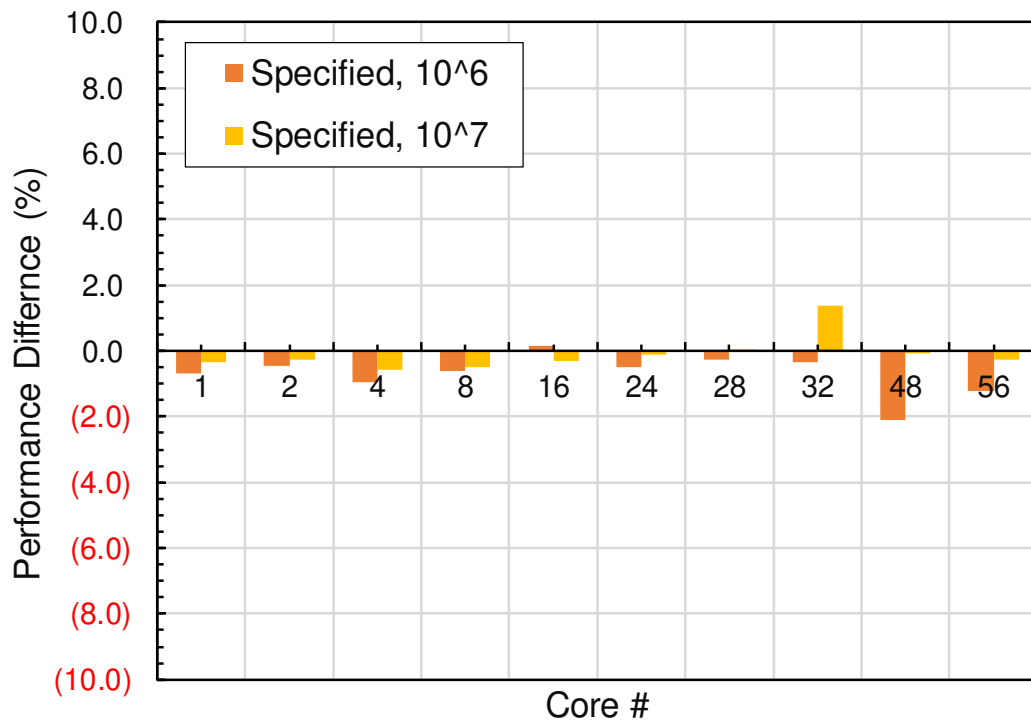
“Original” and “Modified”

Best Case of 5 Measurements

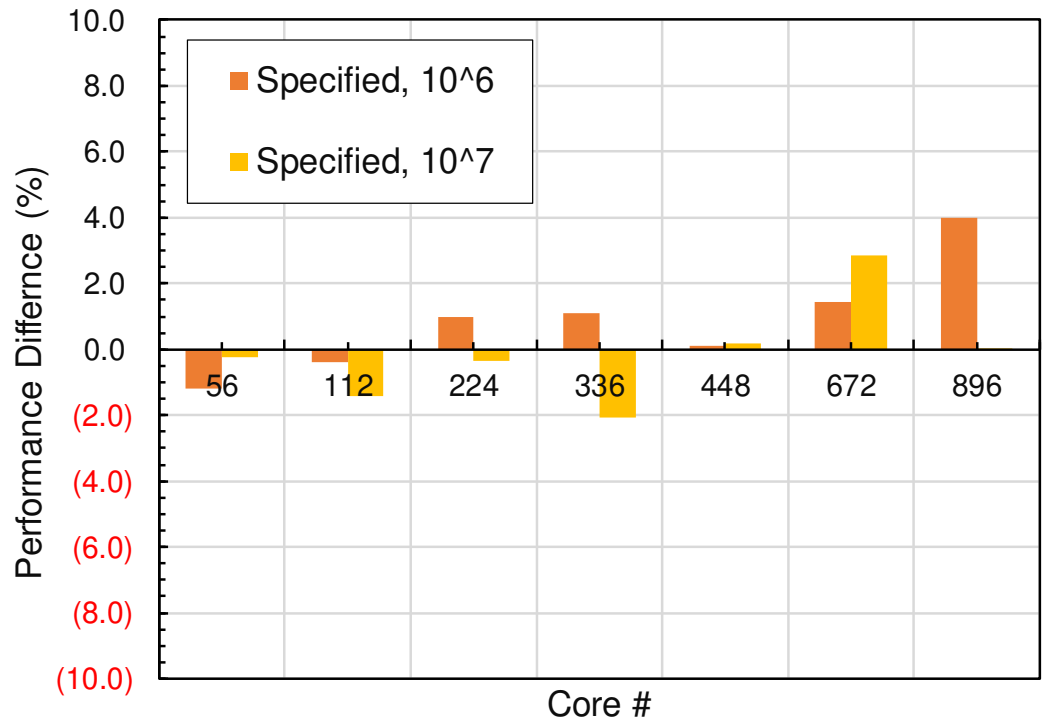
+: “Modified” is better, -: “Modified” is worse

“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



Further Investigations

- Compare with/without NUMA control

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

- MPI_Send/MPI_Recv can be used for 1D case
 - Only limited number of neighbors
- Explain why number of iterations does not change, as number of MPI processes changes.