

Report S2

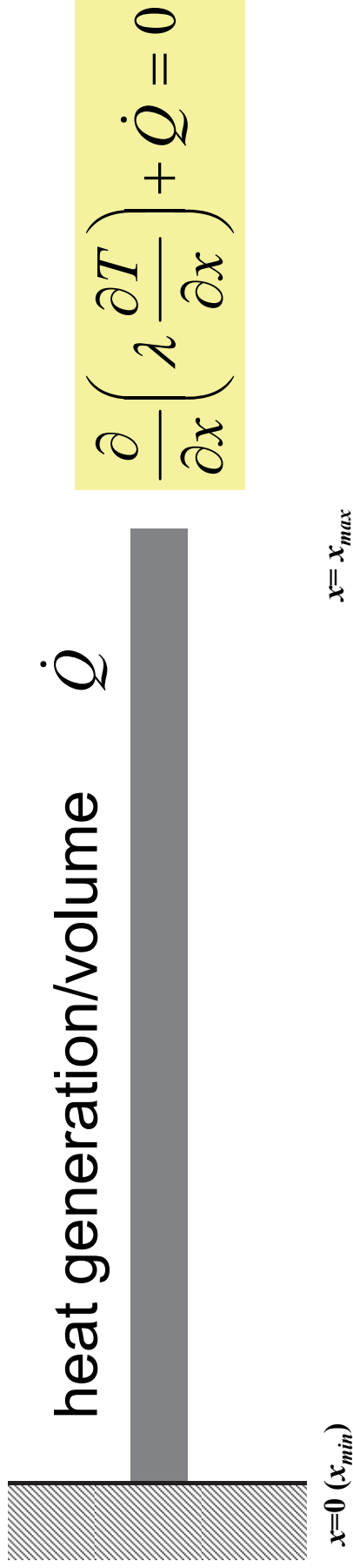
C

Kengo Nakajima

Programming for Parallel Computing (616-2057)
Seminar on Advanced Computing (616-4009)

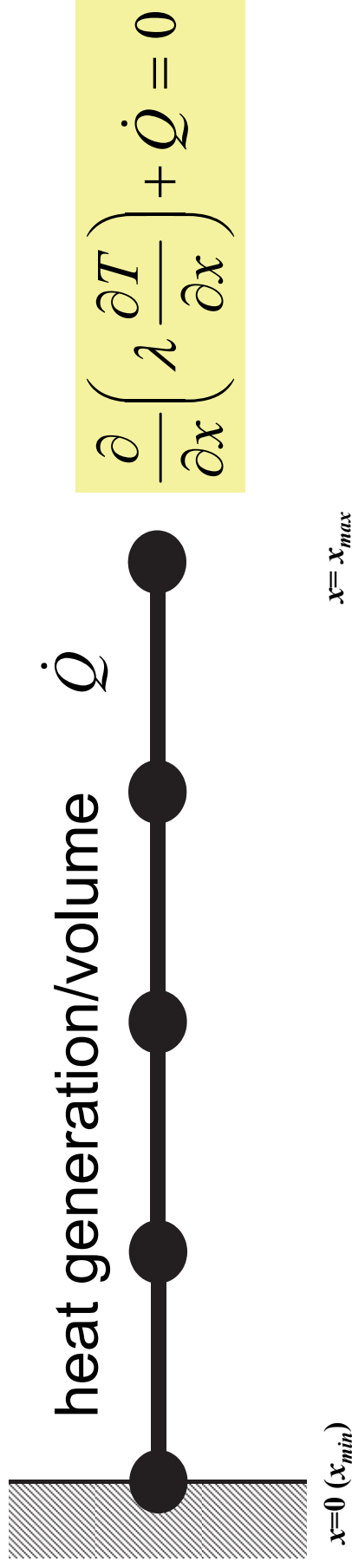
- Overview
- Distributed Local Data
- Program
- Results

1D Steady State Heat Conduction



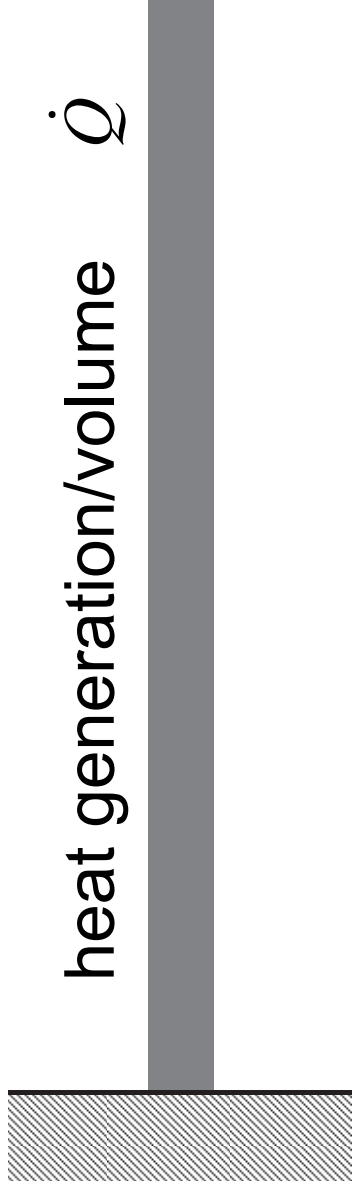
- Uniform: Sectional Area: A , Thermal Conductivity: λ
- Heat Generation Rate/Volume/Time [$\text{QL}^{-3}\text{T}^{-1}$] $\dot{Q}$
- Boundary Conditions
 - $x=0$: $T=0$ (Fixed Temperature)
 - $x=x_{max}$: $\frac{\partial T}{\partial x} = 0$ (Insulated)

1D Steady State Heat Conduction



- 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



$x=0$ (x_{min})

$x=x_{max}$

$$T = 0 @ x = 0$$

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

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

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

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

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

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

Copy and Compile

Fortran

```
>$ cd <$O-TOP>  
>$ cp /home/z30088/class_eps/F/s2r-f.tar .  
>$ tar xvf s2r-f.tar
```

C

```
>$ cd <$O-TOP>  
>$ cp /home/z30088/class_eps/C/s2r-c.tar .  
>$ tar xvf s2r-c.tar
```

Confirm/Compile

```
>$ cd mpi/S2-ref  
>$ mpifrtpx -Kfast ld.f  
>$ mpifccpx -Kfast ld.c
```

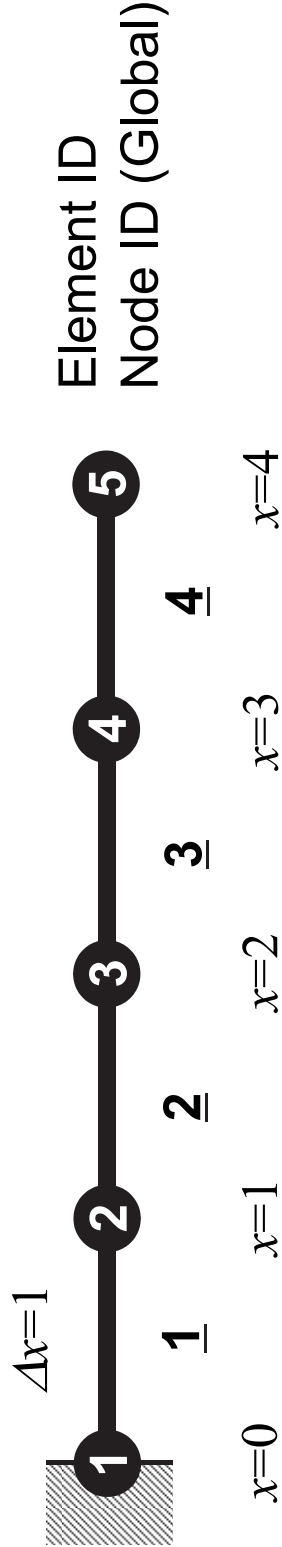
<\$O-S2r> = <\$O-TOP>/mpi/S2-ref

Control File: input.dat

```
Control Data  input.dat

4
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
```



go.sh

```
#!/bin/sh
#PJM -I "node=4" Node # (.1e.12)
#PJM -L "elapsed=00:10:00" Comp.Time (.1e.15min)
#PJM -L "rscgrp=lecture2" "Queue" (or lecture1)
#PJM -g "gt82" "Wallet"
#PJM -
#PJM -o "test.lst" Standard Output
#PJM --mpi "proc=64" MPI Process # (.1e.192)

mpiexec ./a.out
```

N=8
"node=1"
"proc=8"

N=16
"node=1"
"proc=16"

N=32
"node=2"
"proc=32"

N=64
"node=4"
"proc=64"

N=192
"node=12"
"proc=192"

Procedures for Parallel FEM

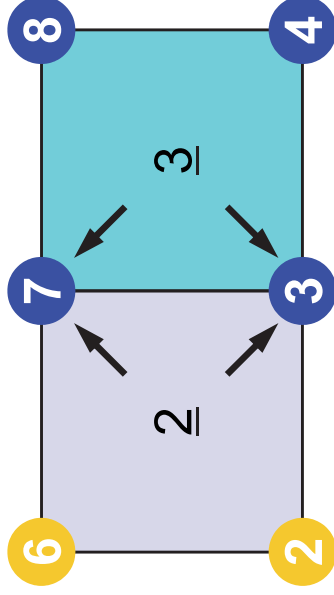
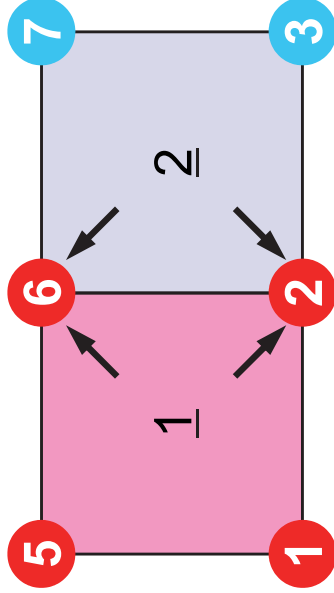
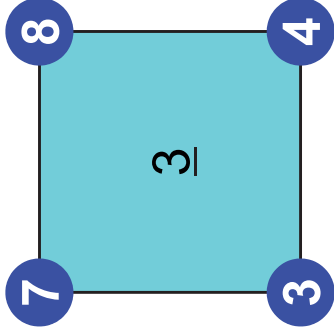
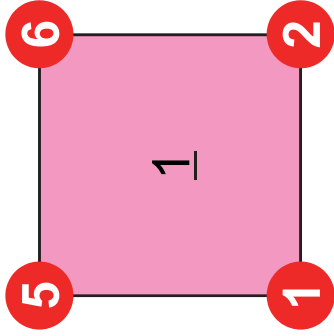
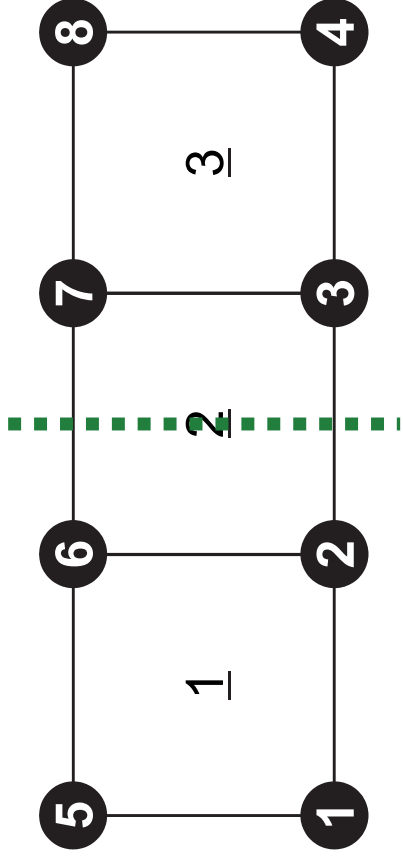
- Reading control file, entire element number etc.
- Creating “distributed local data” in the program
- Assembling local and global matrices for linear solvers
- Solving linear equations by CG
- Not so different from those of original code

- 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

Quadrilateral Elements



Node-based partitioning
Independent variables are
defined at nodes (vertices).

Information is not sufficient for
assembling coefficient matrices.

Info. of nodes and elements in
overlapped zones are required
for assembling coef. matrices.
Computation of stress
components needs same info.

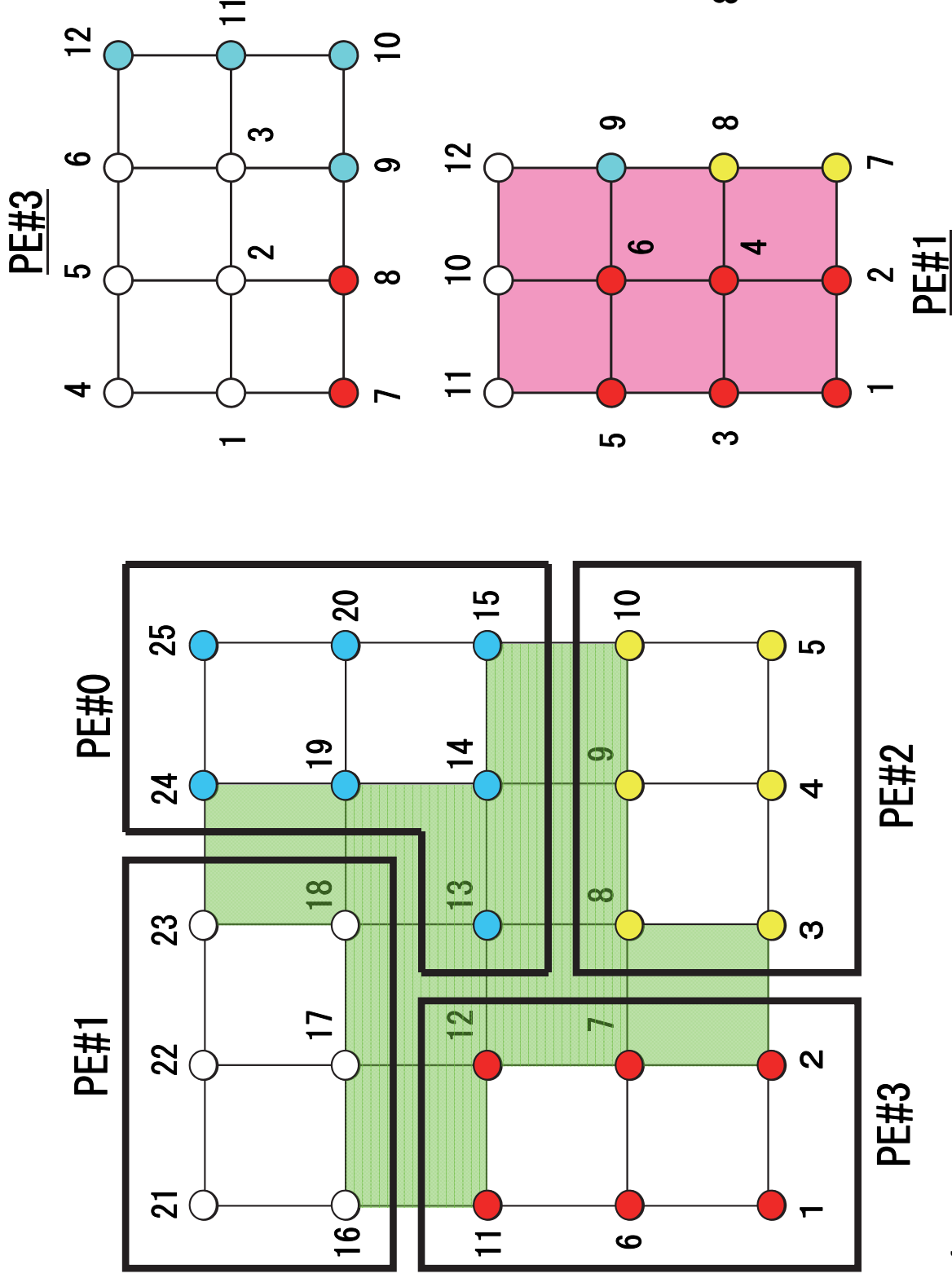
Distributed Local Data Structure for Parallel FEM



- **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

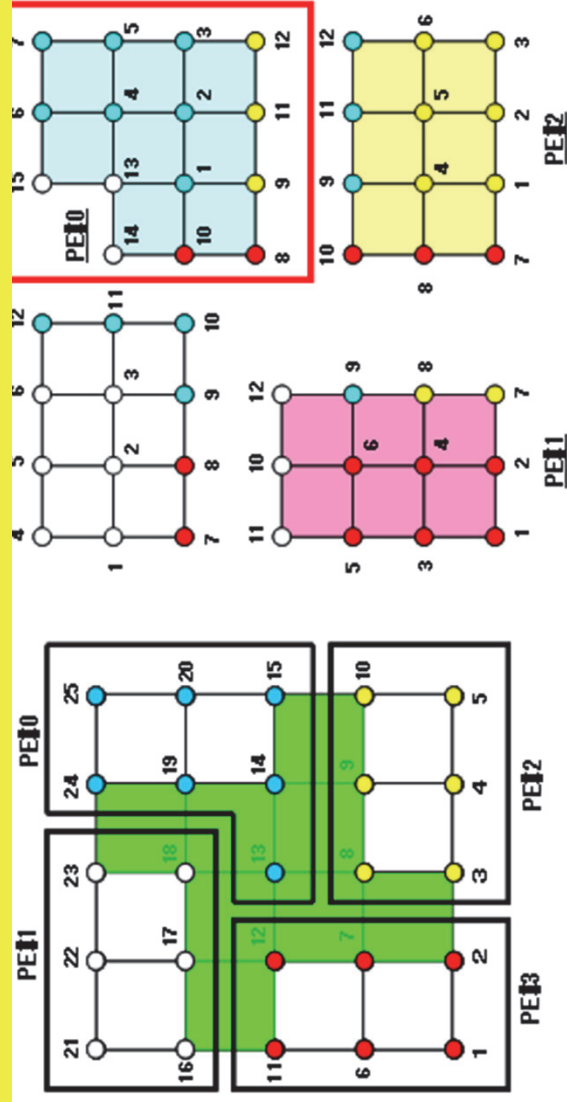
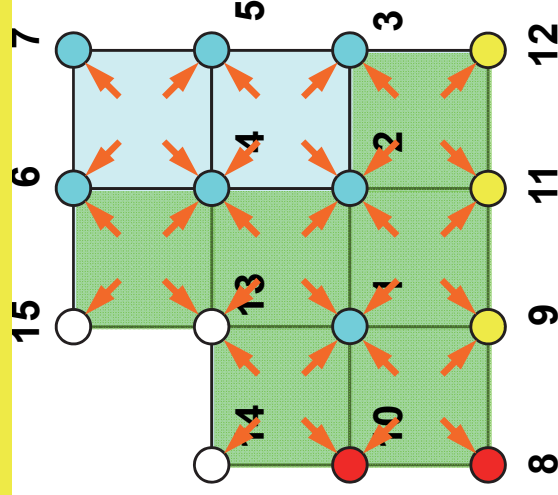


Node-based Partitioning

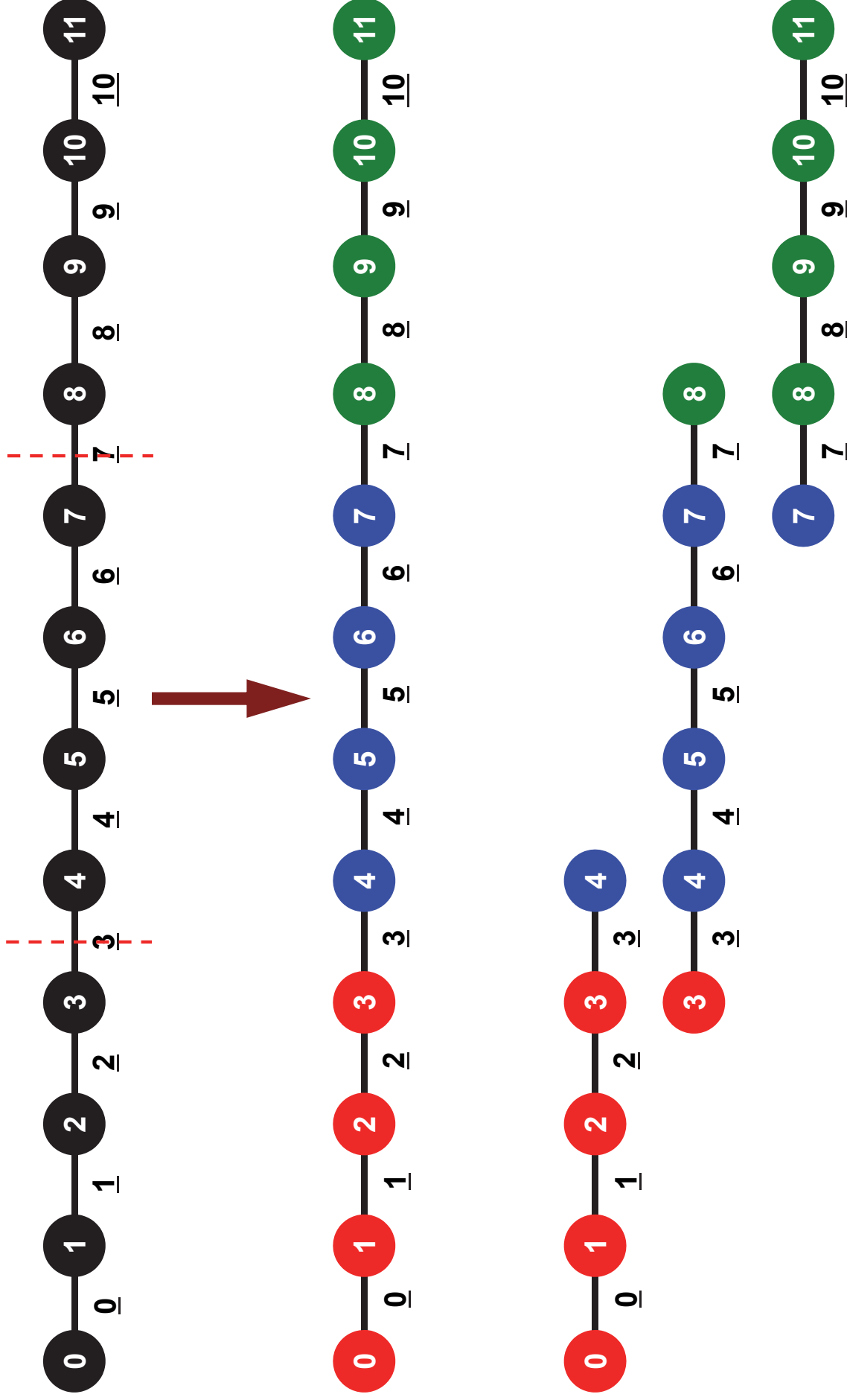
internal nodes - elements - external nodes

- 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.

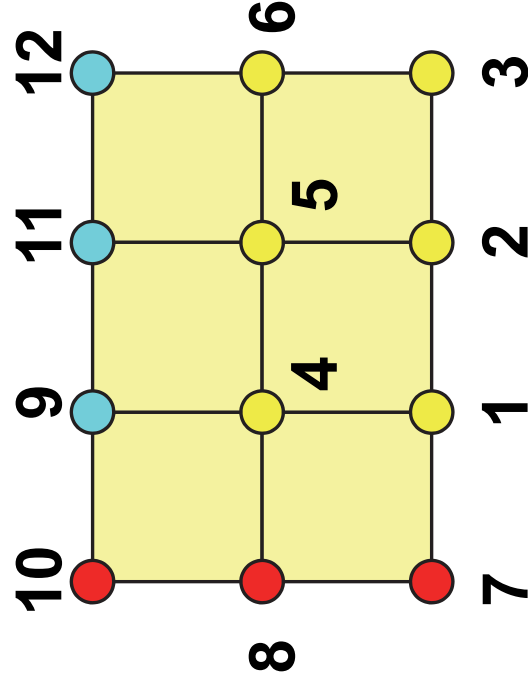
- Info of External Nodes are required for completely local element-based operations on each processor.



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



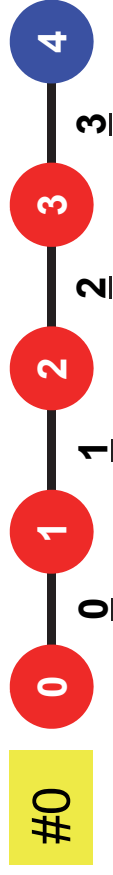
Description of Distributed Local Data



- 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 ?

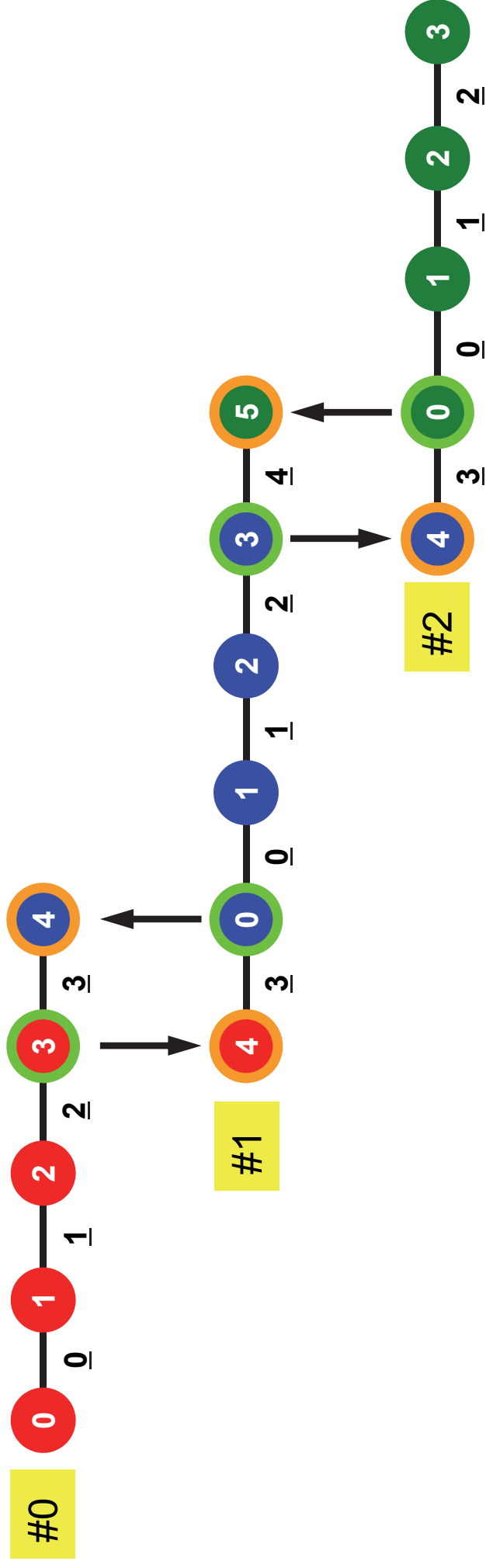
1D FEM: 12 nodes/11 elem's/3 domains

Local ID: Starting from 0 for node and elem at each domain



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

Internal/External Nodes



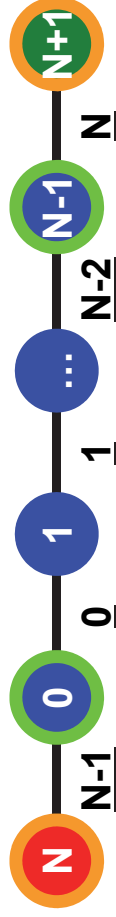
1D FEM: Numbering of Local ID



#0:
 $N+1$ nodes
 N elements



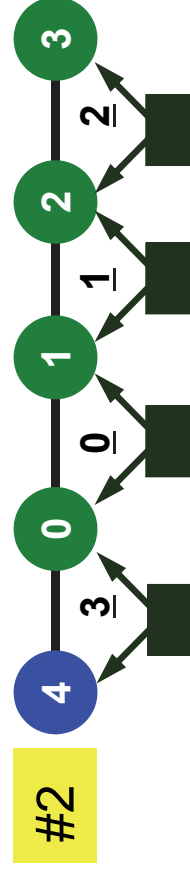
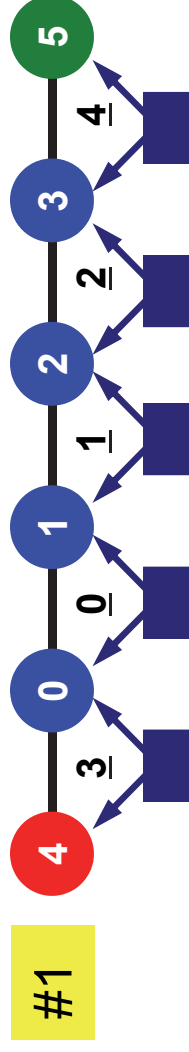
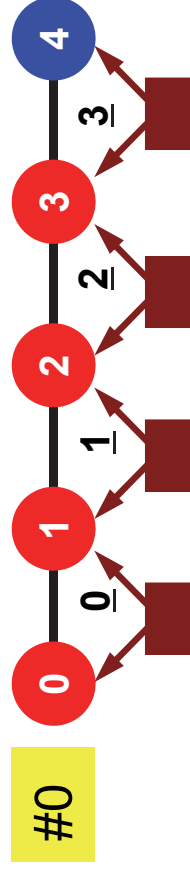
#PETot-1:
 $N+1$ nodes
 N elements



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

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

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

```

Preconditioning:
Diagonal Scaling
(or Point Jacobi)

Preconditioning, DAXPY

Local Operations by Only Internal Points: Parallel

Processing is possible

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

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

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

Dot Products

Global Summation needed: Communication ?

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

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

Matrix-Vector Products

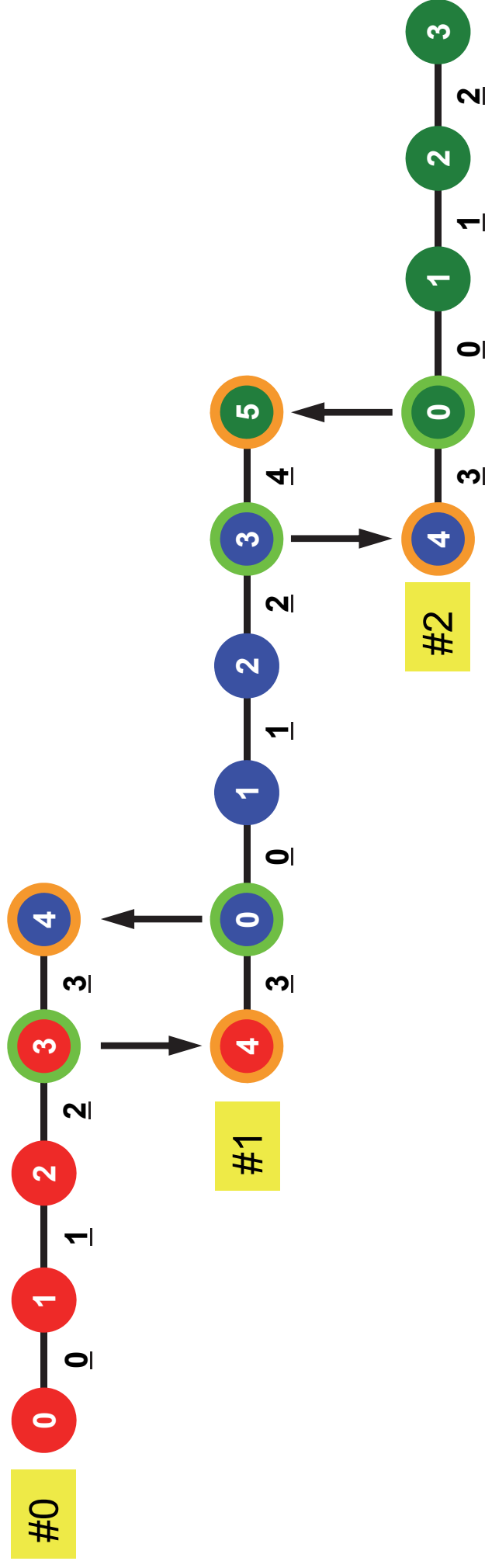
Values at External Points: P-to-P Communication

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



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

Internal/External Nodes



Mat-Vec Products: Local Op. Possible

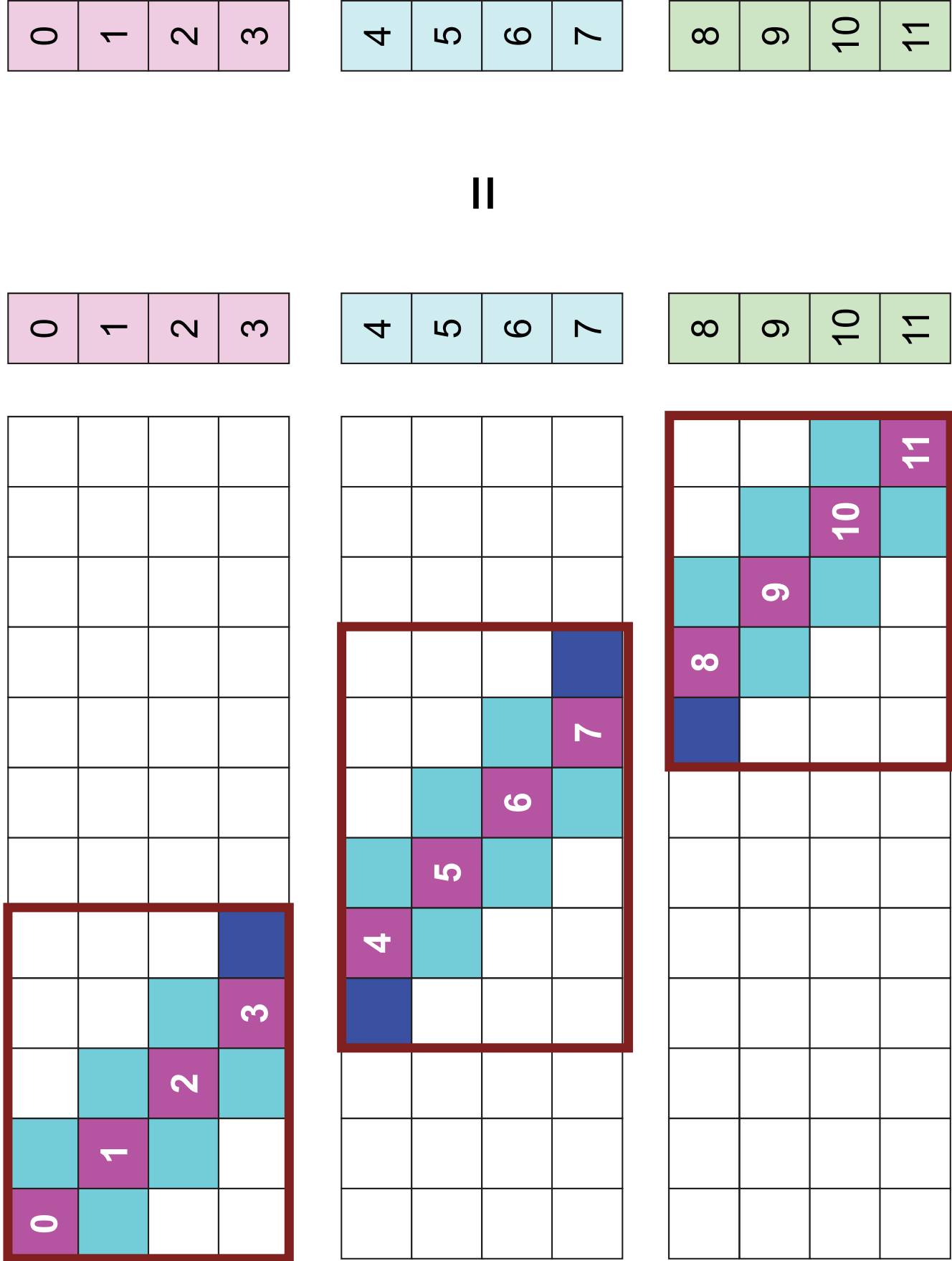
[illegible]

11

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

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

Mat-Vec Products: Local Op. Possible



Mat-Vec Products: Local Op. Possible

0			
	1		
		2	
			3

0	1	2	3
---	---	---	---

0	1	2	3
---	---	---	---

0	1	2	3
---	---	---	---

0	1	2	3
---	---	---	---

=

0	1	2	3
---	---	---	---

0	1	2	3
---	---	---	---

Mat-Vec Products: Local Op. #1

0
1
2
3

=



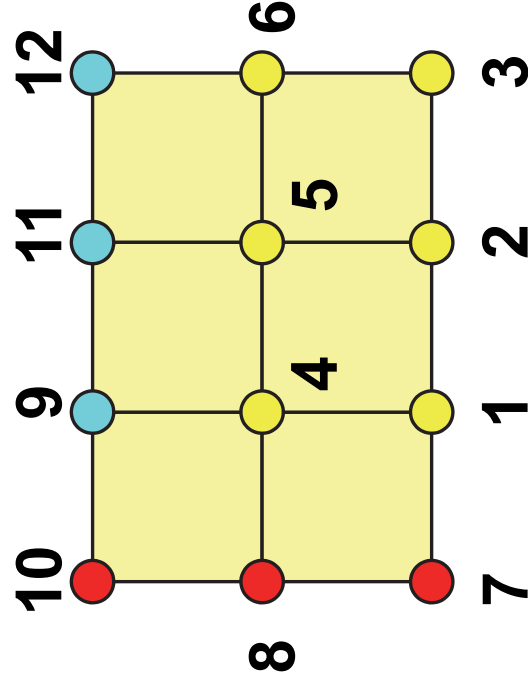
0
1
2
3

=

4
5



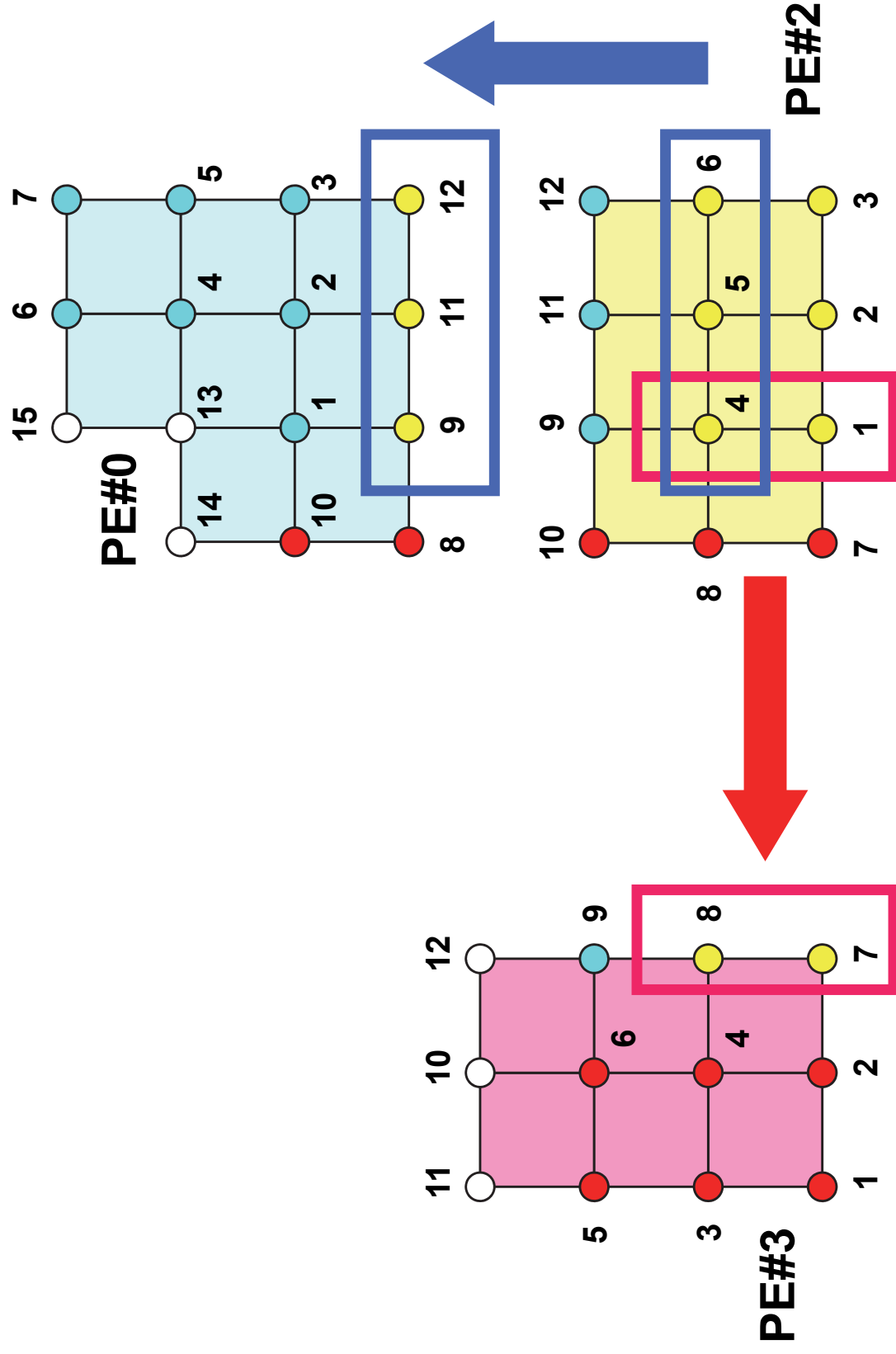
Description of Distributed Local Data



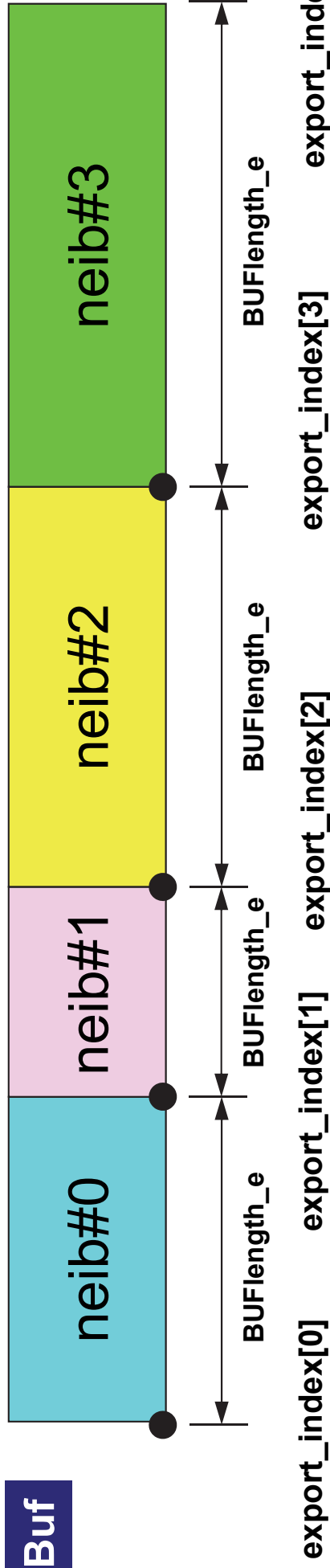
- Internal/External Points
 - Numbering: Starting from internal pts, then external pts after that
- Neighbors
 - Shares overlapped meshes
 - Number and ID of neighbors
- External Points
 - From where, how many, and which external points are received/imported ?
- Boundary Points
 - To where, how many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



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

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

Copied to sending buffers

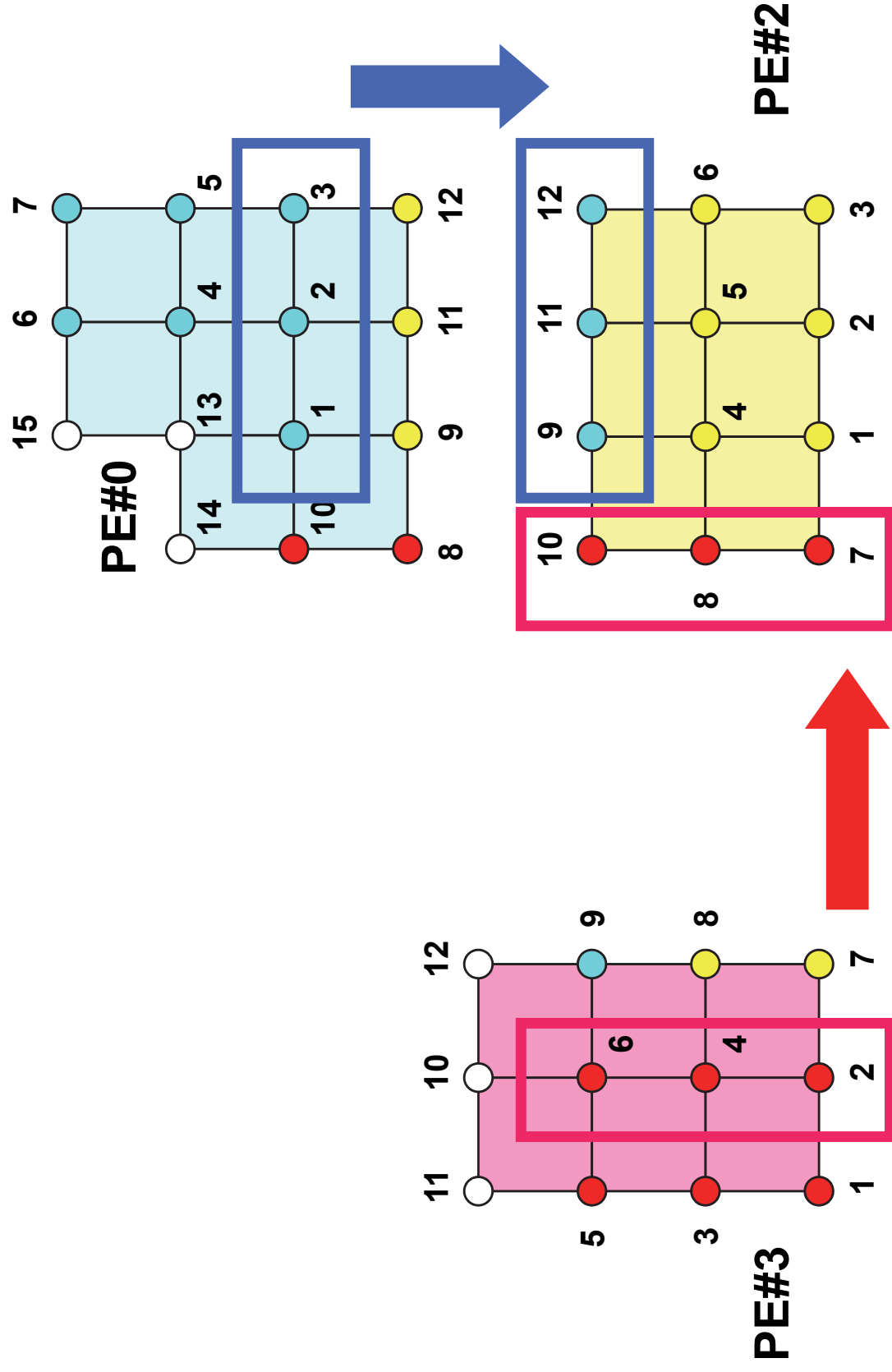
```
for (neib=0; neib<NeibPFTot; neib++){
    tag= 0;
    iS_e= export_index[neib];
    iE_e= export_index[neib+1];
    BUFLength_e= iE_e - iS_e

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

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

External Nodes (外点) : RECEIVE

PE#2 : receive information for “external nodes”



RECV: MPI_Isend/Irecv/Waitall

```

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

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

MPI_Waitall(NeibPETot, ReqRecv, StatRecv);

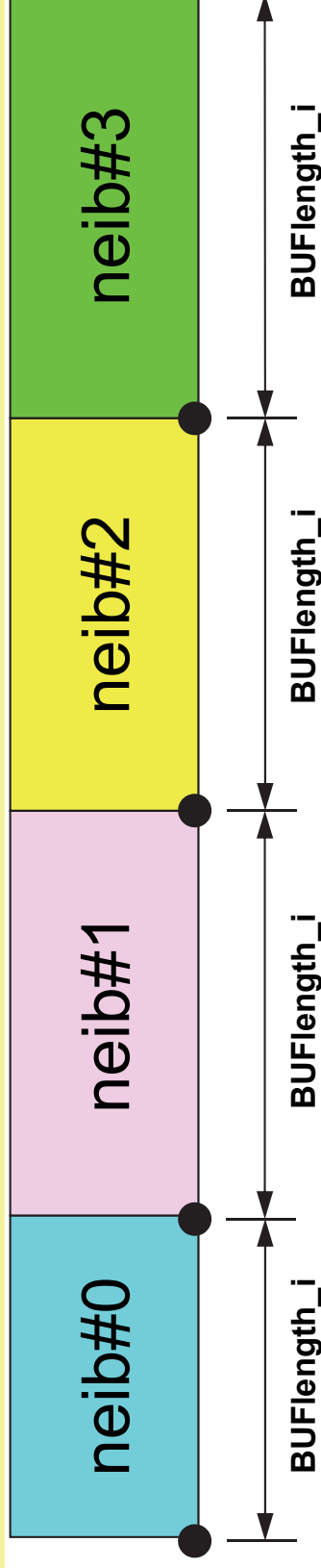
for (neib=0; neib<NeibPETot;neib++){
    for (k=import_index[neib];k<import_index[neib+1];k++){
        kk= import_item[k];
        VAL[kk]= RecvBuf[k];
    }
}

```

Copied from receiving buffer

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

RecvBuf



import_index[0] import_index[1] import_index[2] import_index[3] import_index[4]

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

Program: 1d.c (1/11)

Variables

```

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

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

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

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

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

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

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

```

Program: 1d.c (2/11)

Control Data

```

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

/*
//--- CONTROL data
*/

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

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

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

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

```

Program: 1d.c (2/11)

Control Data

```

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

/*
//--- CONTROL data
*/

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

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

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

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

```

Reading control file if MyRank=0

Neg: Global Number of Elements

Program: 1d.c (2/11)

Control Data

```

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

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

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

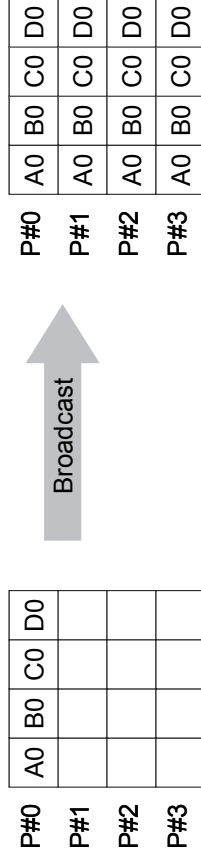
Initialization
Entire Process #: PETot
Rank ID (0-PETot-1): MyRank

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

ierr = MPI_Bcast(&NEg, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&dX, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&QV, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Area, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&COND, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
Parameters are sent to each process
from Process #0.

```

MPI_Bcast



- Broadcasts a message from the process with rank "root" to all other processes of the communicator
- **MPI_Bcast (buffer, count, datatype, root, comm)**
 - buffer choice I/O starting address of buffer
type is defined by "datatype"
 - count int I number of elements in send/receive buffer
 - datatype MPI_Datatype 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.
 - root int I rank of root process
 - comm MPI_Comm I communicator

Program: 1d.c (3/11)

Distributed Local Mesh

```

/*
  //-- LOCAL MESH size
  */

  Ng= NEg + 1;
  N = Ng / PETot;

  Ng: Number of Nodes (Global)
  N : Number of Nodes (Local)

  nr = Ng - N*PETot;
  if(MyRank < nr) N++;

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

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

/*
  //-- Arrays
  */

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

```

Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
//-- LOCAL MESH size
*/

Ng= NEg + 1;
N = Ng / PETot;

nr = Ng - N*PETot;
if (MyRank < nr) N++;

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

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

```

```

/*
//-- Arrays
*/

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

```



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

Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
//-- LOCAL MESH size
*/
Ng= NEg + 1;
N = Ng / PETot;

nr = Ng - N*PETot;
if (MyRank < nr) N++;

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

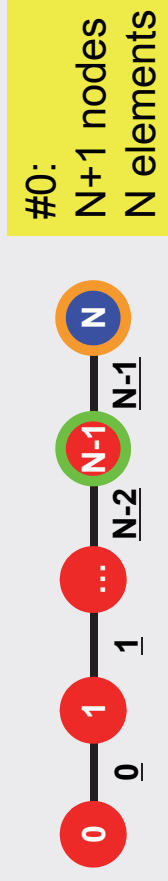
if (PETot==1) {NE=N-1;}
if (PETot==1) {NP=N ;}

```

```

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

```



Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
//-- LOCAL MESH size
*/

Ng= NEg + 1;
N = Ng / PETot;

nr = Ng - N*PETot;
if (MyRank < nr) N++;
NE= N - 1 + 2;
NP= N + 2;
if (MyRank == 0) NE= N - 1 + 1;
if (MyRank == 0) NP= N + 1;
if (MyRank == PETot-1) NE= N - 1 + 1;
if (MyRank == PETot-1) NP= N + 1;

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

/*
//-- Arrays
*/

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

```



#PETot-1:
N+1 nodes
N elements

Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
//-- LOCAL MESH size
*/

Ng= NEg + 1;
N = Ng / PETot;

      Ng: Number of Nodes (Global)
      N : Number of Nodes (Local)

nr = Ng - N*PETot;
if (MyRank < nr) N++;

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

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

/*
//-- Arrays
*/

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

```

Size of arrays is “NP” , not “N”

Program: 1d.c (4/11)

Initialization of Arrays, Elements-Nodes

```

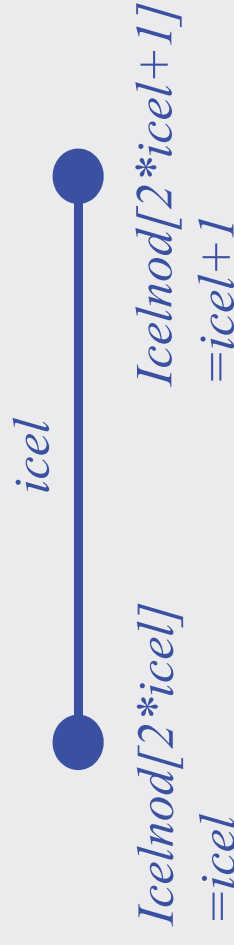
for (i=0; i<NP; i++)    U[i] = 0.0;
for (i=0; i<NP; i++)    Diag[i] = 0.0;
for (i=0; i<NP; i++)    Rhs[i] = 0.0;
for (k=0; k<2*NP-2; k++)  AMat[k] = 0.0;

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

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

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

```



Program: 1d.c (4/11)

Initialization of Arrays, Elements-Nodes

```

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

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

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

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

```

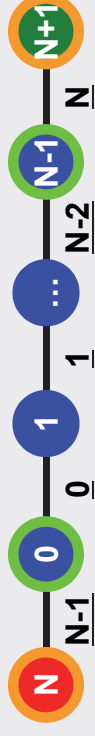
e.g. Element-0 includes node-0 and node-1



#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements



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

Program: 1d.c (5/11)

"Index"

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

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

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

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

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

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

#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements



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



Program: 1d.c (6/11)

”Item”

```

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

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

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

```

#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements



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



Program: 1d.c (7/11)

Communication Tables

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

BufLength = 1;

```

```

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

```

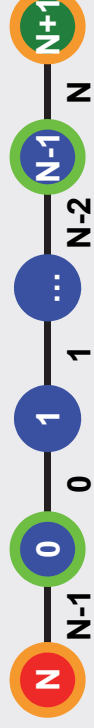
#0:
N+1 nodes
N elements



#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`.

- **MPI_Isend**

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

- `sendbuf` choice I starting address of sending buffer
- `count` int I number of elements in sending buffer
- `datatype` MPI_Datatype I datatype of each sending buffer element
- `dest` int I rank of destination
- `tag` int 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), communicator

- `comm` MPI_Comm I
- `request` MPI_Request O **communication request array used in `MPI_Waitall`**

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`.

- `MPI_Irecv`

`(recvbuf, count, datatype, source, tag, comm, request)`

- recvbuf choice I starting address of receiving buffer
- count int I number of elements in receiving buffer
- datatype MPI_Datatype I datatype of each receiving buffer element
- source int I rank of source
- tag int 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),

communicator

- comm MPI_Comm I
- request MPI_Request O

communication request array used in `MPI_Waitall`

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.

• `MPI_Waitall (count, request, status)`

- count int I number of processes to be synchronized
 - request MPI_Request I/O comm. request used in `MPI_Waitall` (array size: count)
 - status MPI_Status O array of status objects
- `MPI_STATUS_SIZE`: defined in `'mpif.h'`, `'mpi.h'`

Generalized Comm. Table: Send

- Neighbors
 - NeibPETot, NeibPE[neib]
- Message size for each neighbor
 - export_index[neib], neib= 0, NeibPETot-1
- ID of boundary points
 - export_item[k], k= 0, export_index[NeibPETot]-1
- Messages to each neighbor
 - SendBuf[k], k= 0, export_index[NeibPETot]-1

SEND: MPI_Isend/Irecv/Waitall

SendBuf



export_index[0] export_index[1] export_index[2] export_index[3] export_index[4]

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

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

Copied to sending buffers

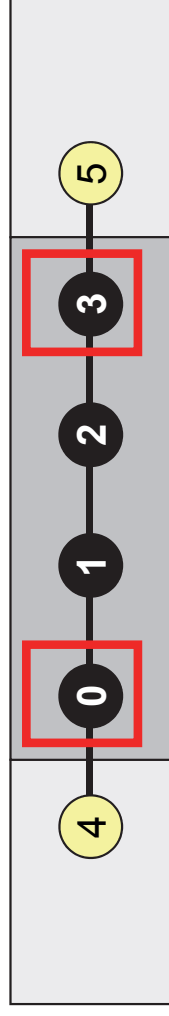
```
for (neib=0; neib<NeibPETot; neib++){
    tag= 0;
    iS_e= export_index[neib];
    iE_e= export_index[neib+1];
    BUFLength_e= iE_e - iS_e

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

    }

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

SEND/Export: 1D Problem



– NeibPETot, NeibPE[neib]

SendBuf[0]=VAL[0]

SendBuf[1]=VAL[3]

- NeibPETot=2, NeibPE[0]= my_rank-1, NeibPE[1]= my_rank+1

- Message size for each neighbor

- export_index[neib], neib= 0, NeibPETot-1
- export_index[0]=0, export_index[1]= 1, export_index[2]= 2

- ID of boundary points

- export_item[k], k= 0, export_index[NeibPETot]-1
- export_item[0]= 0, export_item[1]= N-1

- Messages to each neighbor

- SendBuf[k], k= 0, export_index[NeibPETot]-1
- SendBuf[0]= VAL[0], SendBuf[1]= VAL[N-1]

Generalized Comm. Table: Receive

- Neighbors
 - NeibPETot , NeibPE[neib]
- Message size for each neighbor
 - import_index[neib], neib= 0, NeibPETot-1
- ID of **external** points
 - import_item[k], k= 0, import_index[NeibPETot]-1
- Messages from each neighbor
 - RecvBuf[k], k= 0, import_index[NeibPETot]-1

RECV: MPI_Isend/Irecv/Waitall

```

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

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

MPI_Waitall(NeibPETot, ReqRecv, StatRecv);

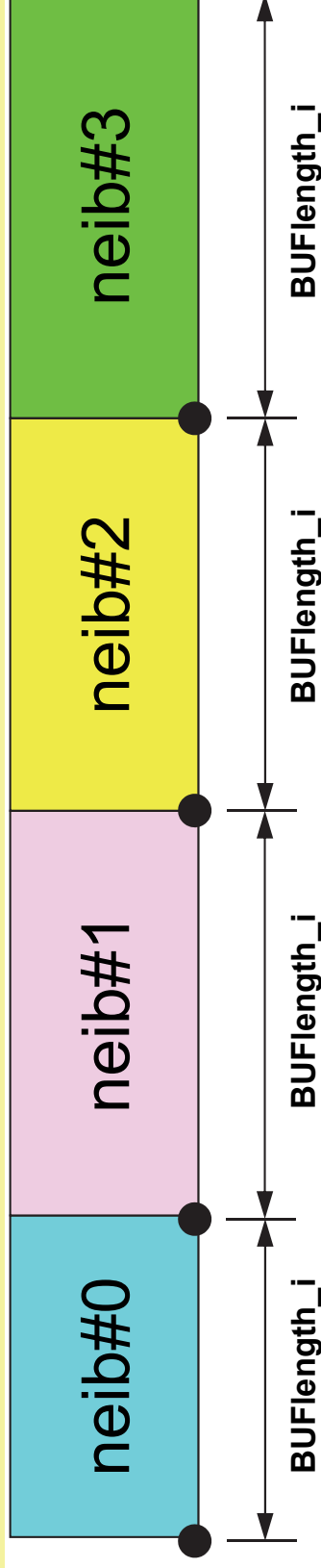
for (neib=0; neib<NeibPETot;neib++){
    for (k=import_index[neib];k<import_index[neib+1];k++){
        kk= import_item[k];
        VAL[kk]= RecvBuf[k];
    }
}

```

Copied from receiving buffer

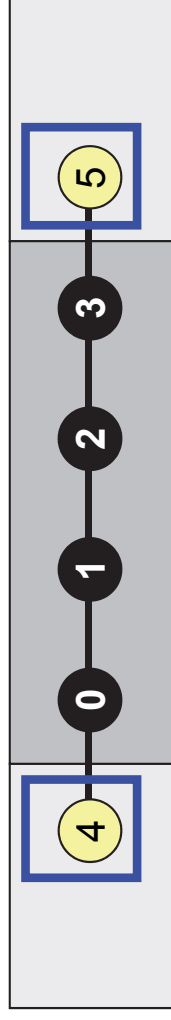
import_item (import_index[neib]:import_index[neib+1]-1) are received from neib-th neighbor

RecvBuf



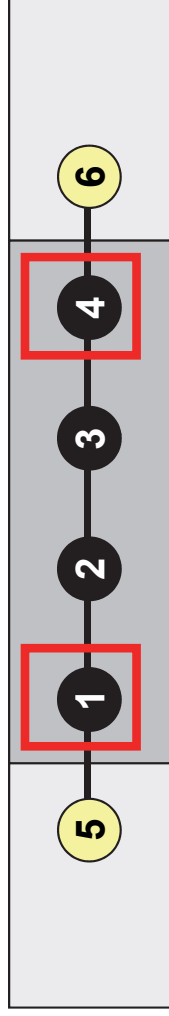
import_index[0] import_index[1] import_index[2] import_index[3] import_index[4]

RCV/Import: 1D Problem



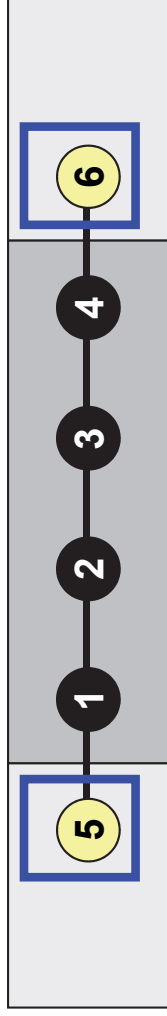
- Neighbors
 - NeibPETot, NeibPE[neib]
 - NeibPETot=2, NeibPE[0]= my_rank-1, NeibPE[1]= my_rank+1
- Message size for each neighbor
 - import_index[neib], neib= 0, NeibPETot-1
 - import_index[0]=0, import_index[1]= 1, import_index[2]= 2
- ID of external points
 - import_item[k], k= 0, import_index[NeibPETot]-1
 - import_item[0]= N, import_item[1]= N+1
- Messages from each neighbor
 - RCVbuf[k], k= 0, import_index[NeibPETot]-1
 - VAL[N]=RecvBuf[0], VAL[N+1]=RecvBuf[1]

Generalized Comm. Table: Fortran



SENDbuf (1) = BUF (1)

SENDbuf (2) = BUF (4)



BUF (5) = RECVbuf (1)

BUF (6) = RECVbuf (2)

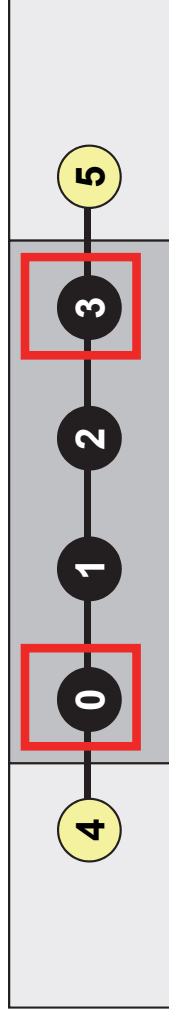
```
NEIBPETOT= 2
NEIBPE(1)= my_rank - 1
NEIBPE(2)= my_rank + 1
```

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

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

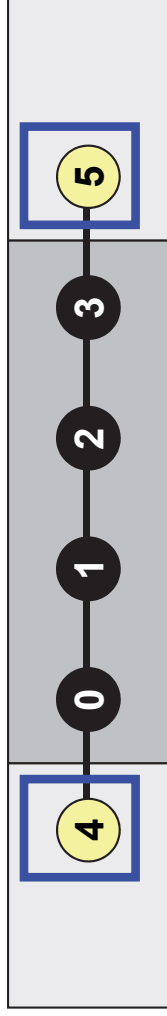
```
if (my_rank.eq.0) then
  import_item(1)= N+1
  export_item(1)= N
  NEIBPE(1)= my_rank+1
endif
```

Generalized Comm. Table: C



SENDbuf[0]=BUF[0]

SENDbuf[1]=BUF[3]



BUF[4]=RECVbuf[0]

BUF[5]=RECVbuf[1]

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

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

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

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

Program: 1d.c (8/11)

Matrix Assembling, NO changes from 1-CPU code

```

/*
//-----+
// | MATRIX assemble |
//-----+
*/

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

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

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

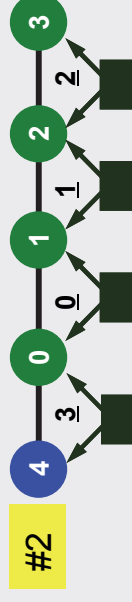
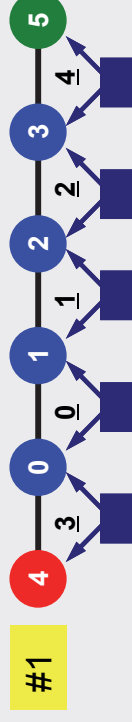
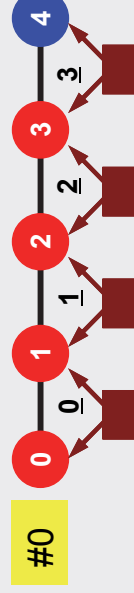
    if ((MyRank==0)&&(icel==0)) {
        k1=Index[in1];
    } else {k1=Index[in1]+1;}

    k2=Index[in2];

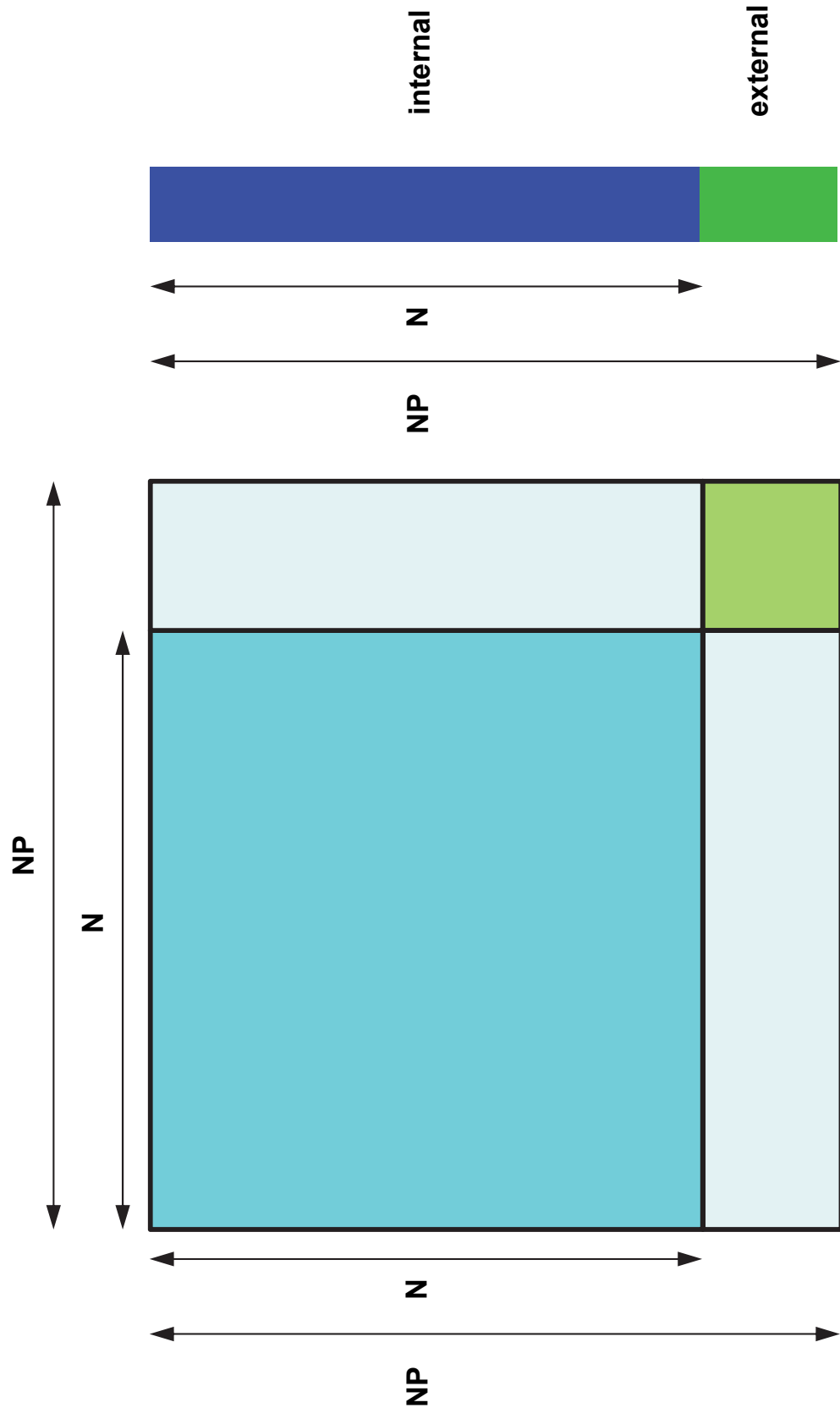
    AMat[k1]= AMat[k1] + Emat[0][1];
    AMat[k2]= AMat[k2] + Emat[1][0];

    QN= 0.5*QV*Area*dX;
    Rhc[in1]= Rhc[in1] + QN;
    Rhc[in2]= Rhc[in2] + QN;
}

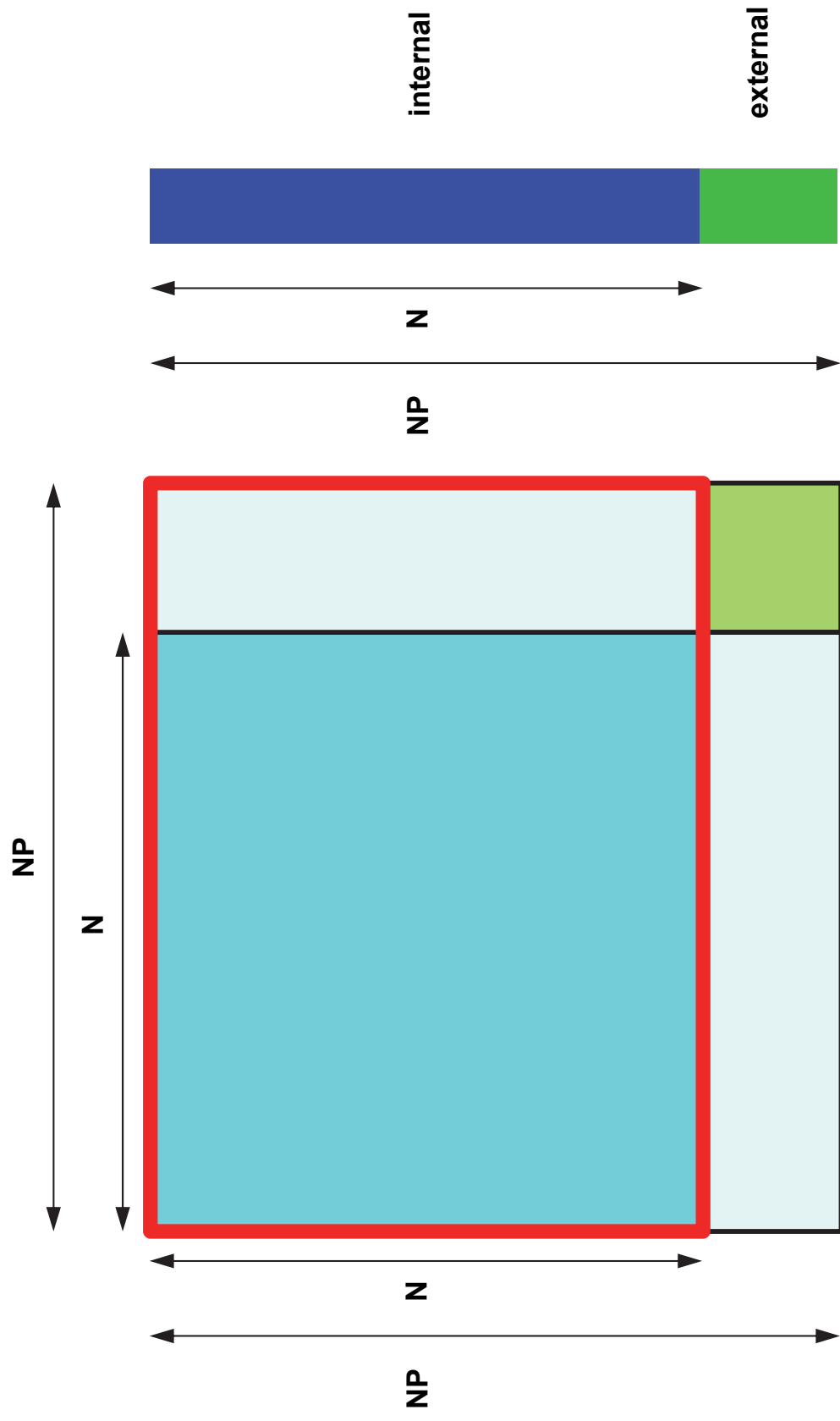
```



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. Points 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)

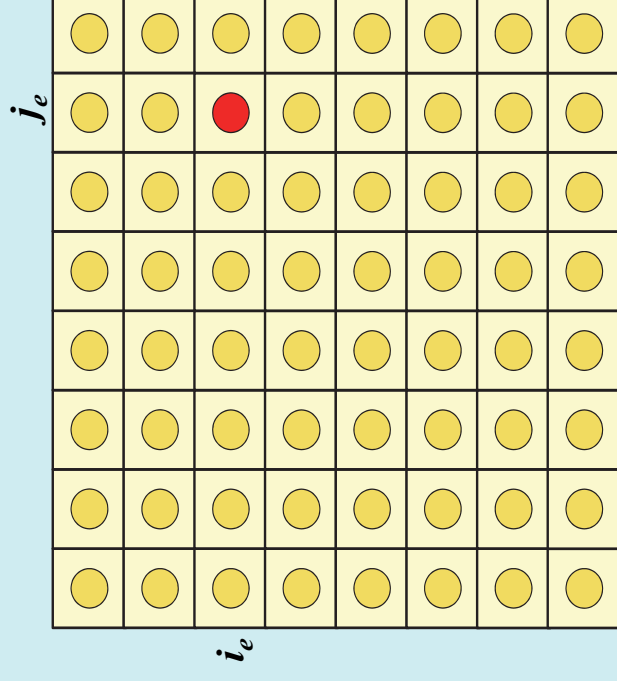
```

```

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

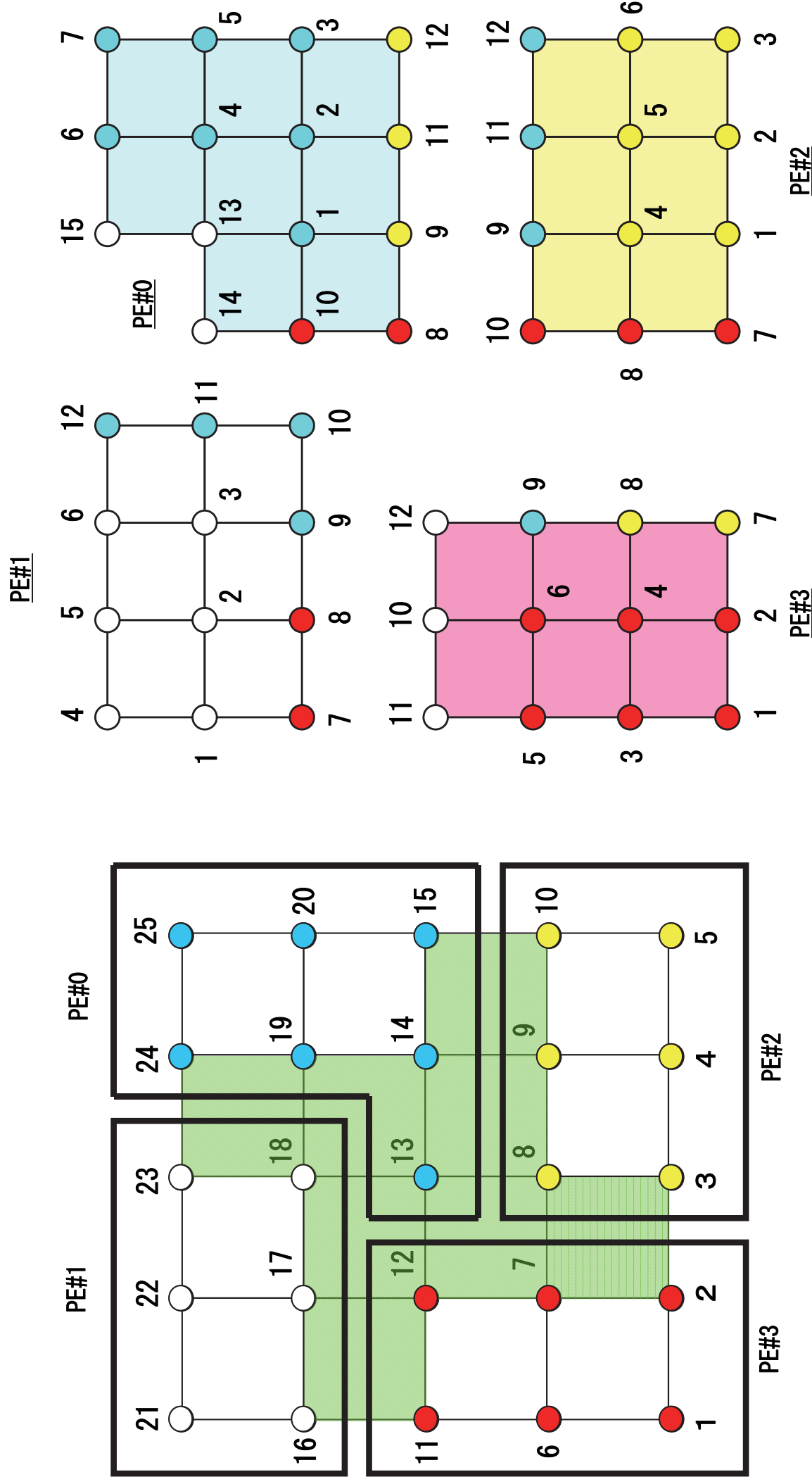
do kpn= 1, 2      Gaussian Quad. points in  $\zeta$ -direction
do jpn= 1, 2      Gaussian Quad. points in  $\eta$ -direction
do ipn= 1, 2      Gaussian Quad. points in  $\xi$ -direction
  integration on each element
  coefficients of element matrices
  accumulation to global matrix
enddo
enddo
enddo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

```

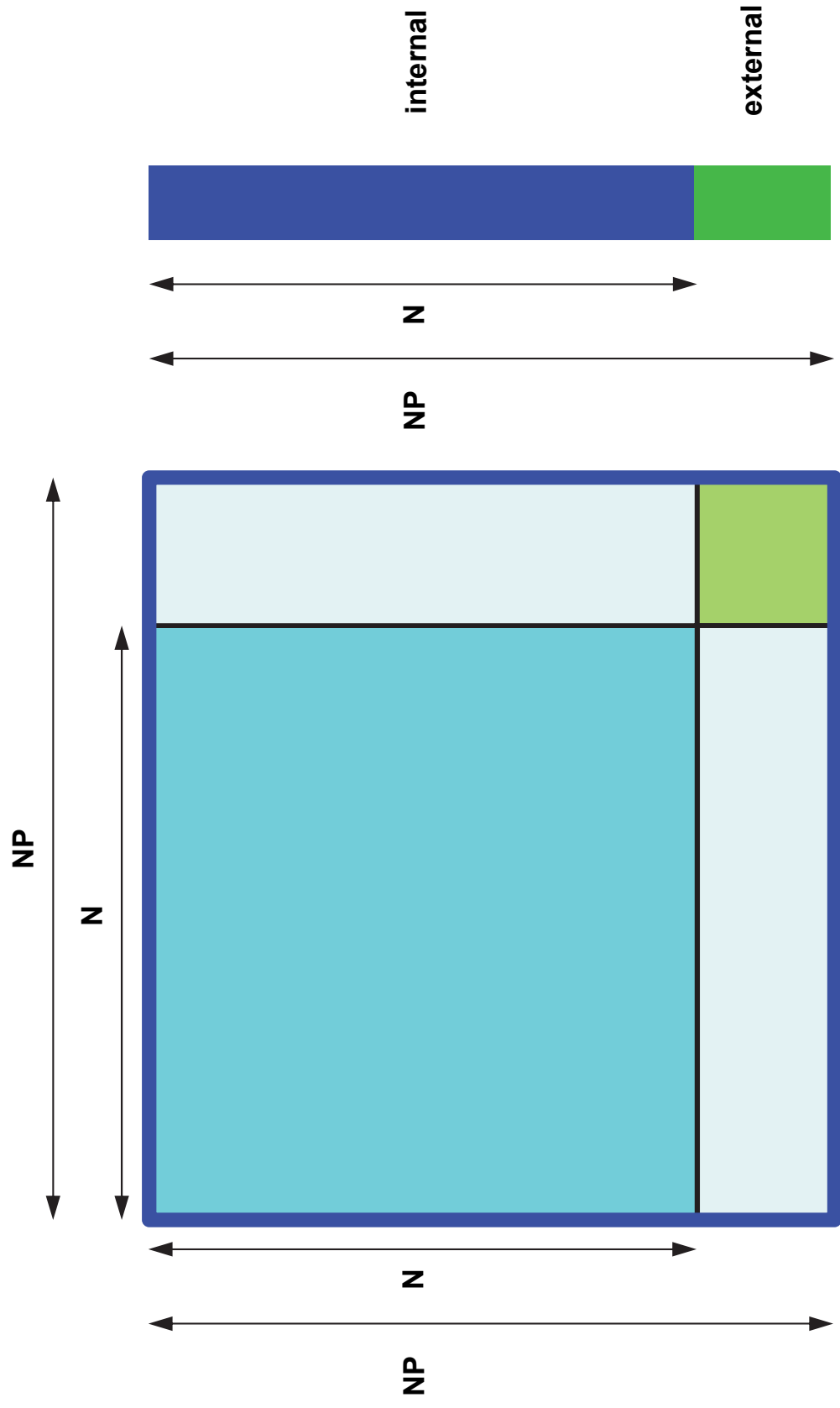


MAT_ASS_MAIN visits all elements

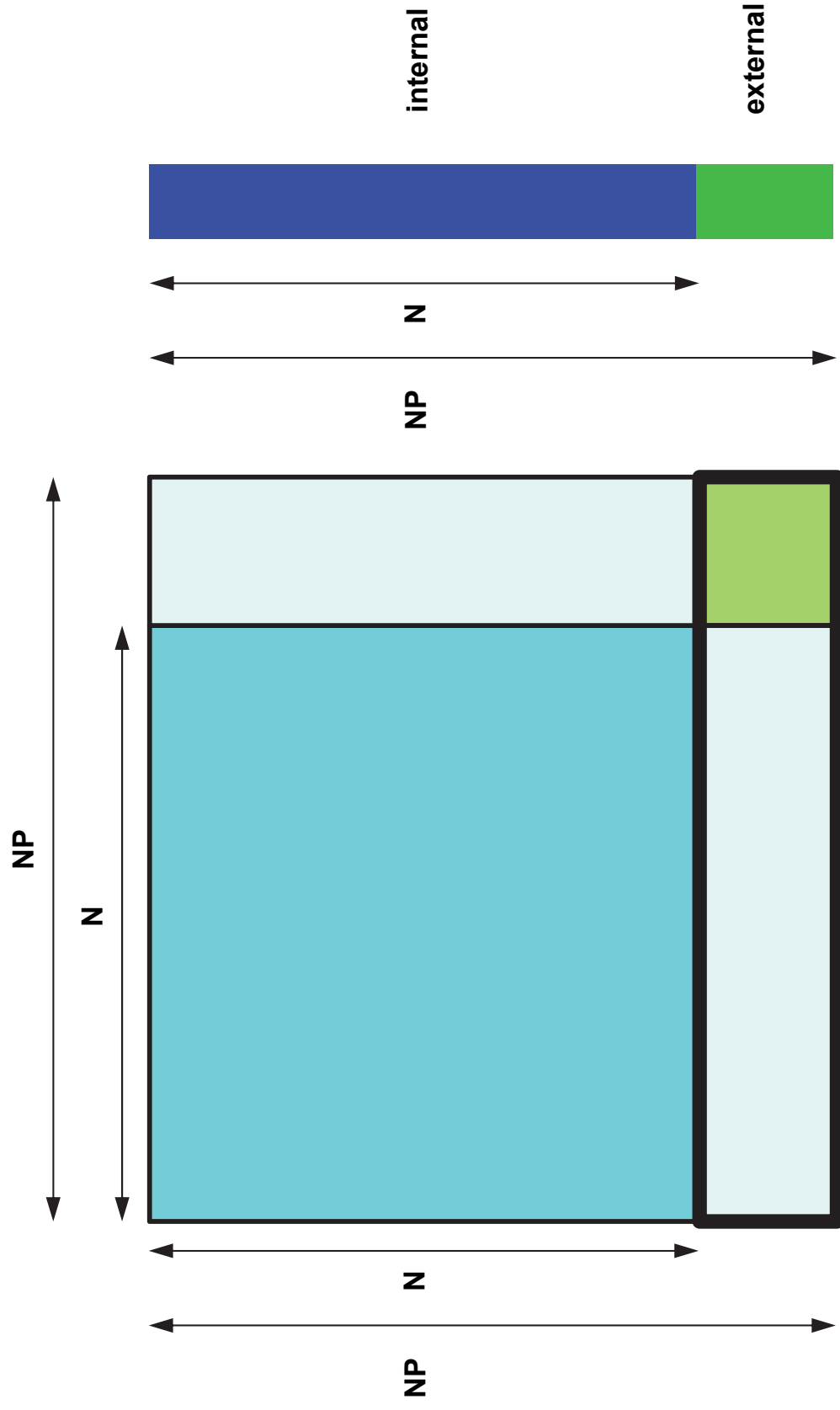
including overlapped elements with external nodes



Therefore, we have this matrix



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



Program: 1d.c (9/11)

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

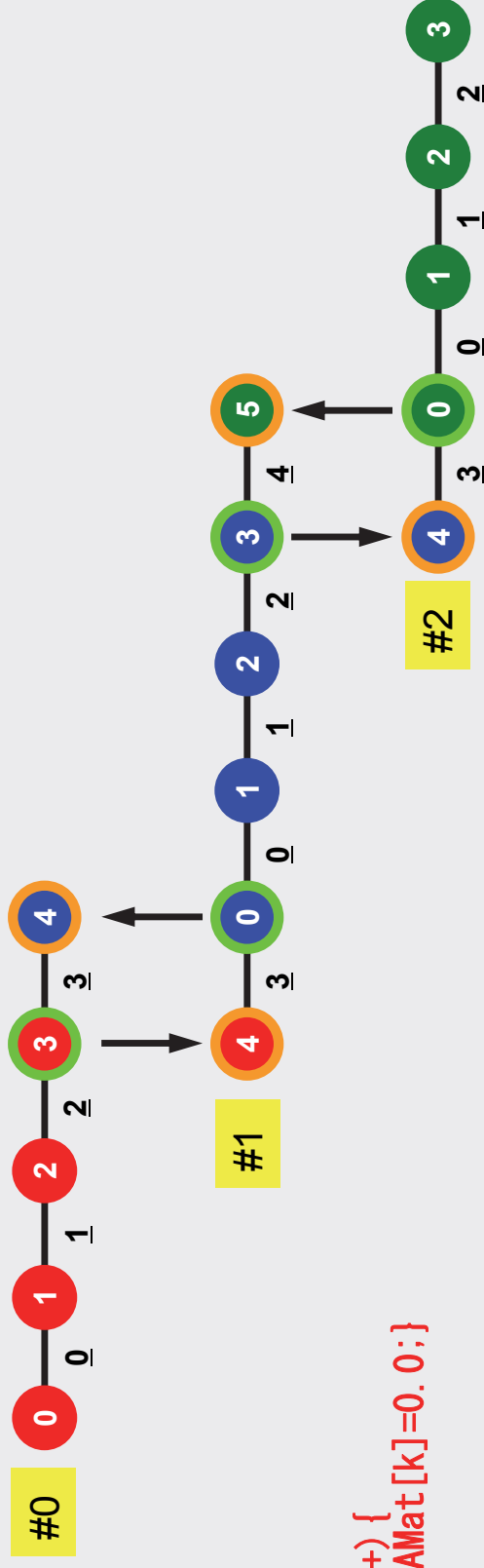
```

/*
//-----+
// | BOUNDARY conditions |
//-----+
*/

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

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

```



Program: 1d.c(10/11)

Conjugate Gradient Method

```

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

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

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

```

```

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

```

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

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

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

Matrix-Vector Multiply (1/2)

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

```

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

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

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

MPI_Waitall(NeibPETot, RequestRecv, StatRecv);

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

```

Matrix-Vector Multiply (2/2)

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

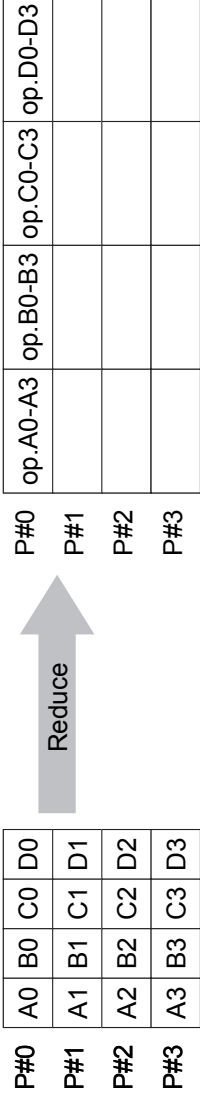
```
MPI_Waitall (NeibPETot, RequestSend, StatSend) ;  
  
for (i=0; i<N; i++) {  
    Q[i] = Diag[i] * P[i];  
    for (j=Index[i]; j<Index[i+1]; j++) {  
        Q[i] += AMat[j]*P[Item[j]];  
    }  
}
```

Dot Product

Global Summation by MPI_Allreduce

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

MPI_Reduce



- Reduces values on all processes to a single value
 - Summation, Product, Max, Min etc.
- **MPI_Reduce (sendbuf , recvbuf , count , datatype , op , root , comm)**
 - sendbuf choice I starting address of send buffer
 - recvbuf choice O starting address receive buffer
 - count int I number of elements in send/receive buffer
 - datatype MPI_Datatype 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 MPI_Op I reduce operation
 - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
 - root int I rank of root process
 - comm MPI_Comm I communicator

Users can define operations by MPI_OP_CREATE

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=0~3)

Example of MPI_Reduce (2/2)

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

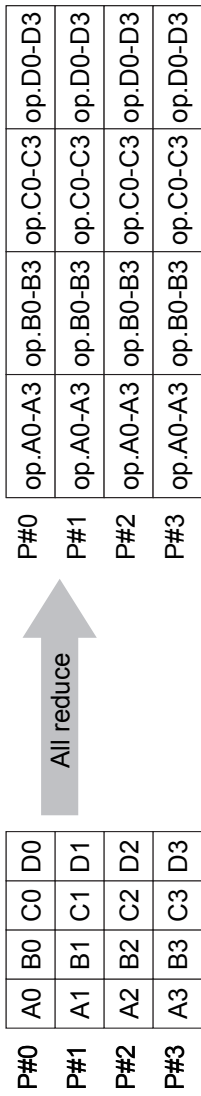
```
real(kind=8):: X0, XSUM  
  
call MPI_REDUCE  
(X0, XSUM, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

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[0] goes to X0[2] on #0 process.
- Global summation of X0[1] goes to X0[3] 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)
 - sendbuf choice I starting address of send buffer
 - recvbuf choice O starting address receive buffer

type is defined by "datatype"

- <u>count</u>	int	I	number of elements in send/receive buffer
- <u>datatype</u>	MPI_Datatype	I	data type of elements of send/recive buffer
- <u>op</u>	MPI_Op	I	reduce operation
- <u>comm</u>	MPI_Comm	I	communicator

CG method (1/5)

```

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

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

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

MPI_Waitall (NeibPETot, RequestRecv, StatRecv);

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

MPI_Waitall (NeibPETot, RequestSend, StatSend);

```

```

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

```

CG method (2/5)

```

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

```

```

BNorm20 = 0.0;
for (i=0; i<N; i++) {
    BNorm20 += Rhs[i] * Rhs[i];
    R[i] = Rhs[i] - R[i];
}

```

```

ierr = MPI_Allreduce (&BNorm20, &BNorm2, 1, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);

```

```

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

```

```

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

```

```

/*
//--- RHO= {r} {z}
*/

```

```

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

```

```

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

```

CG method (3/5)

```

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

```

```

if(iter == 1){
  for(i=0; i<N; i++){
    P[i] = Z[i];
  }
}else{
  Beta = Rho / Rho1;
  for(i=0; i<N; i++){
    P[i] = Z[i] + Beta*P[i];
  }
}

```

```

/*
//-- {q} = [A] {p}
*/

```

```

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

```

```

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

```

```

Compute  $r^{(0)} = b - [A]x^{(0)}$ 
for i= 1, 2, ...
  solve  $[M]z^{(i-1)} = r^{(i-1)}$ 
   $\rho_{i-1} = r^{(i-1)} z^{(i-1)}$ 
  if i=1
     $p^{(1)} = z^{(0)}$ 
  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 (4/5)

```

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

MPI_Waitall(NeibPETot, RequestRecv, StatRecv);

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

MPI_Waitall(NeibPETot, RequestSend, StatSend);

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

/*--- ALPHA= RHO / {p} {q}
*/
C10 = 0.0;
for (i=0;i<N;i++) {
    C10 += P[i] * Q[i];
}
ierr = MPI_Allreduce(&C10, &C1, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
Alpha = Rho / C1;

```

```

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

end

```

CG method (5/5)

```

/*-----
//
//
*/
{x} = {x} + ALPHA*{p}
{r} = {r} - ALPHA*{q}

for (i=0; i<N; i++) {
    PHI[i] += Alpha * P[i];
    R[i] -= Alpha * Q[i];
}

```

```

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

```

```

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

```

```

Resid = sqrt(DNorm2/BNorm2);

```

```

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

```

```

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

```

```

Rho1 = Rho;

```

```

}

```

```

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

```

Program: 1d.c (11/11)

Output by Each Process

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

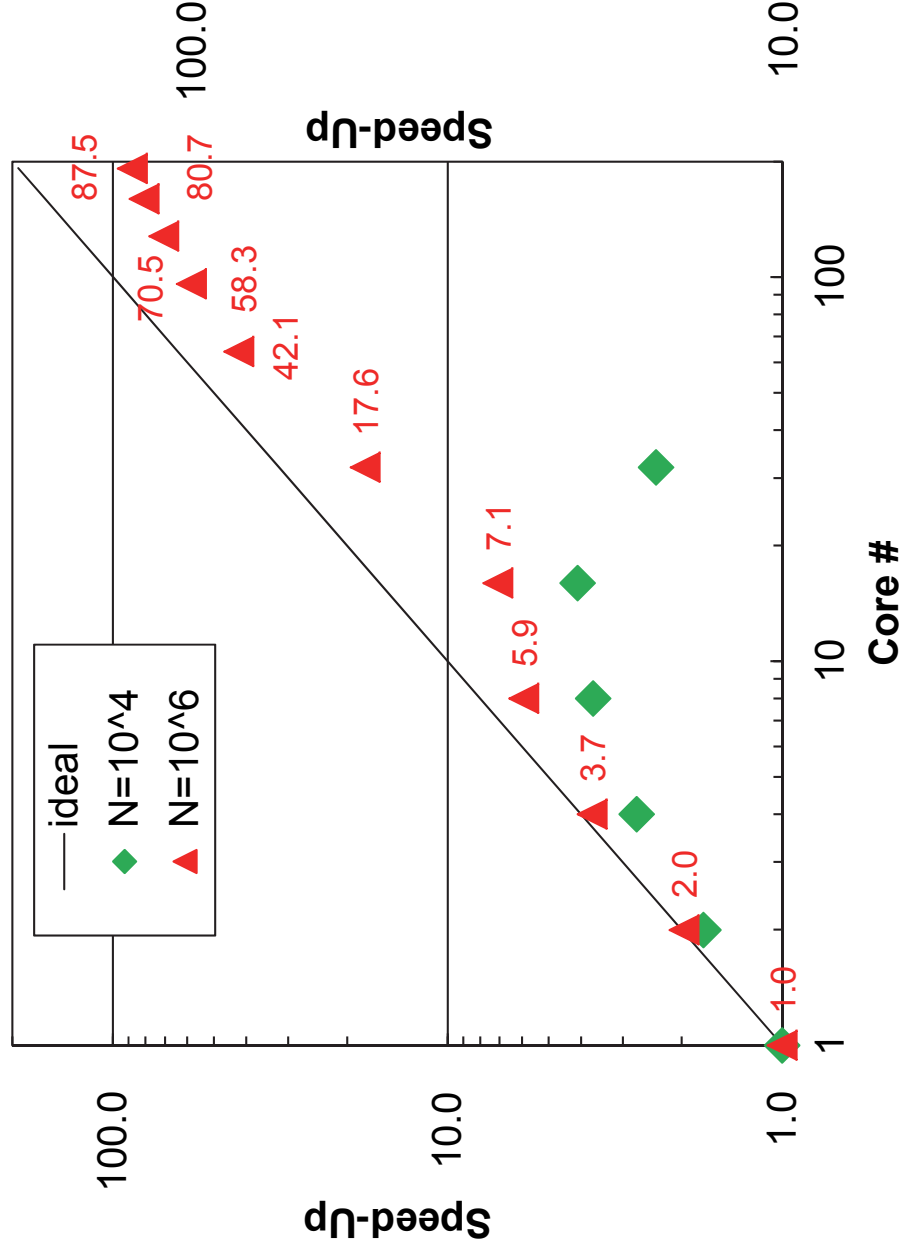
ierr = MPI_Finalize();
return ierr;
}
```

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

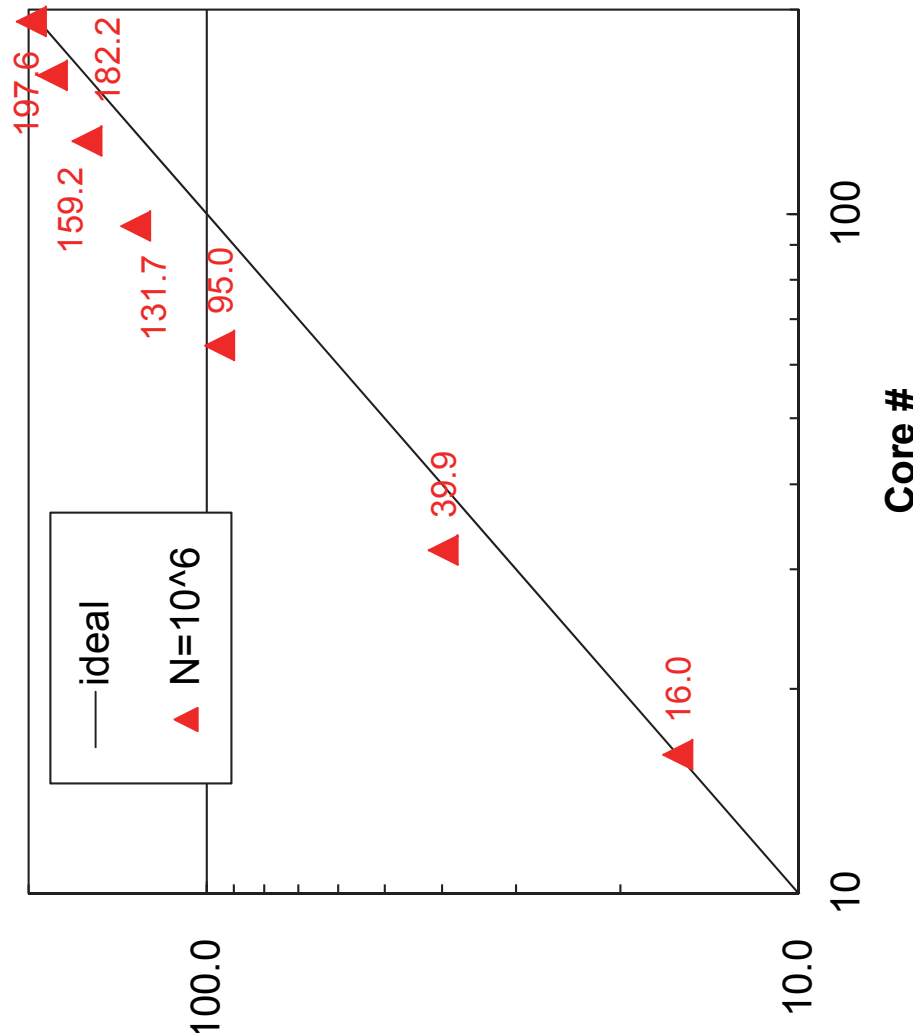
Results for CG Solver

Time for 100 Iterations in $N=10^6$ case

based on
a core



based on
a node (16 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

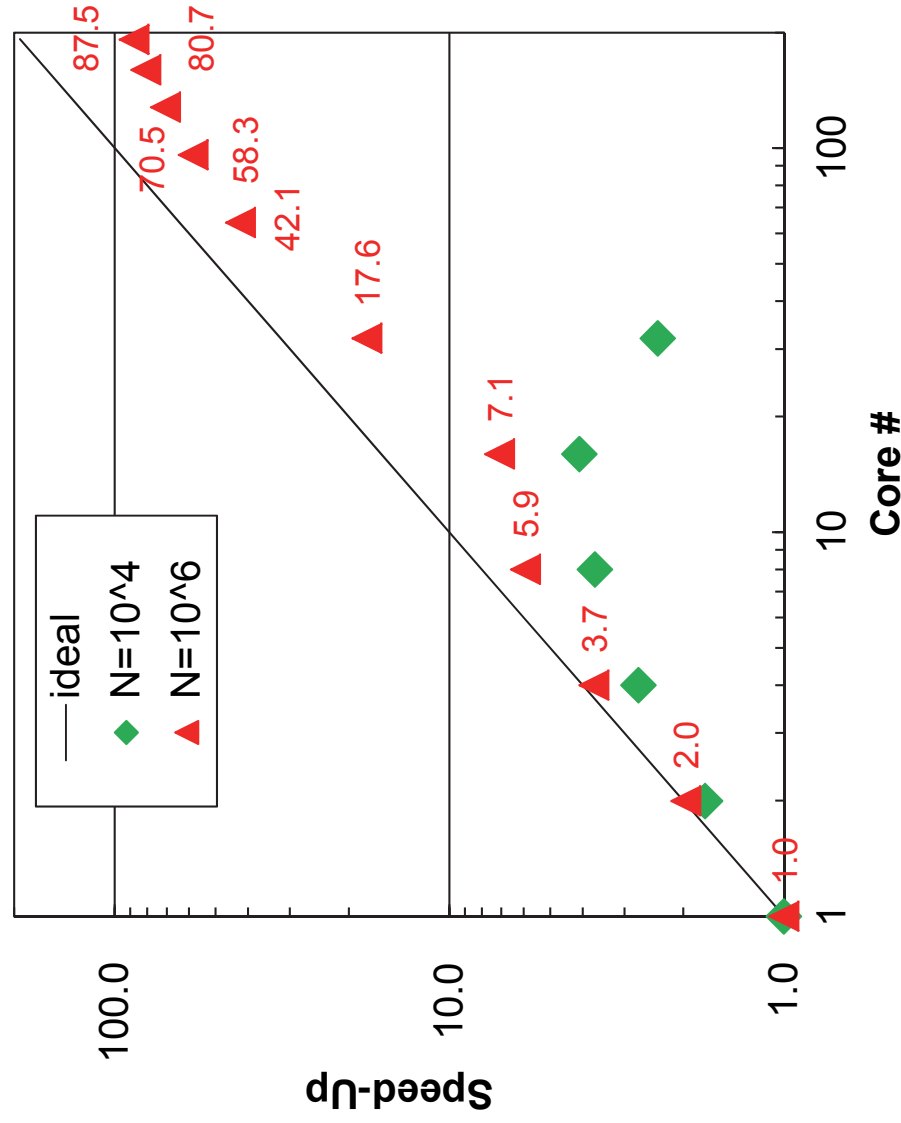
Performance is lower than ideal one

(cont.)

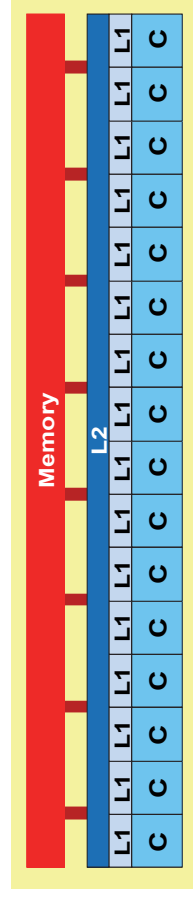
- If computation time is relatively small (N is small in S1-3), these effects are not negligible.
 - If the size of messages is small, effect of “latency” is significant.

Performance is not so good... between 1-16 cores

based on performance at a single core

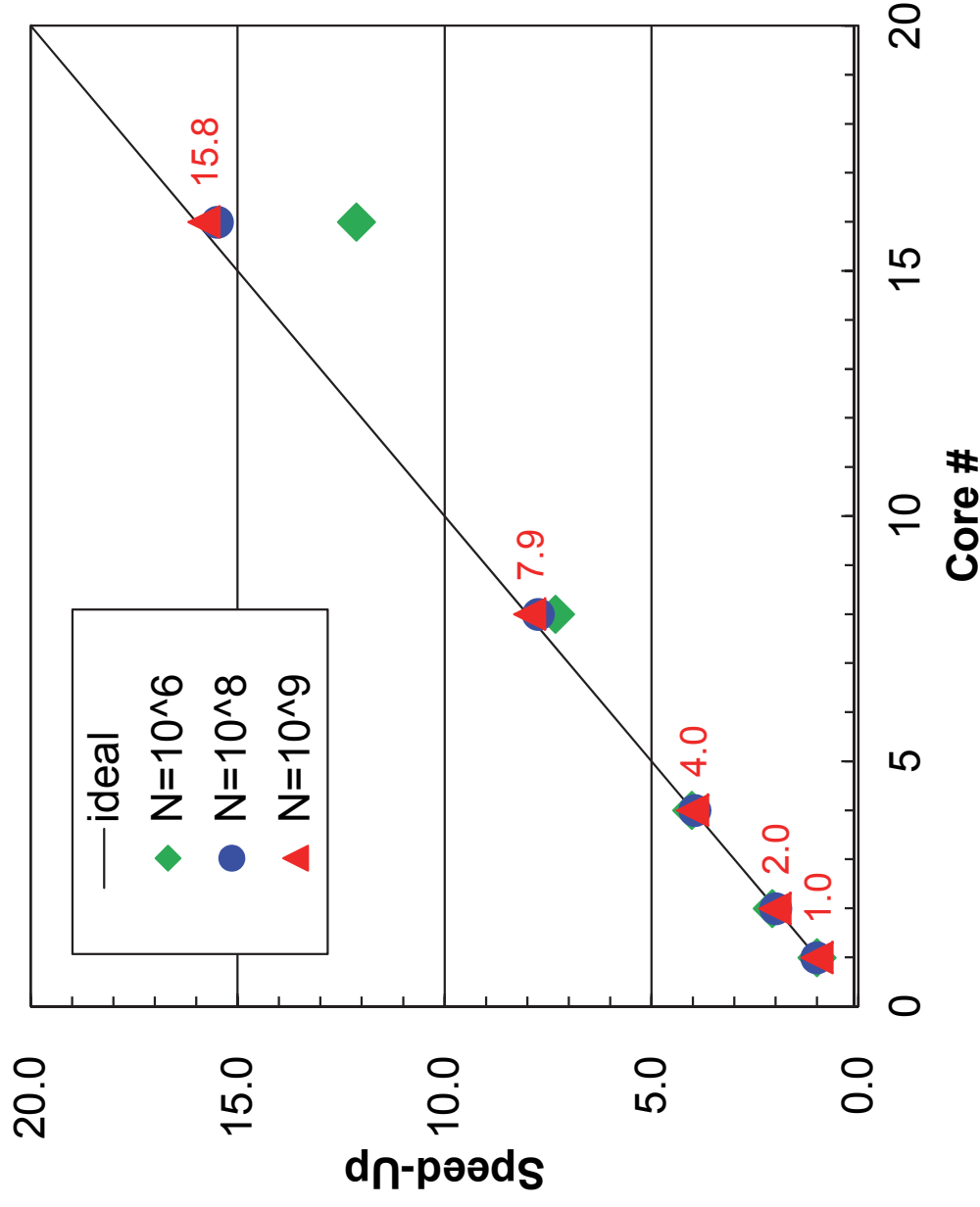


- x 7.1 at 16 cores, because of memory contention
 - STREAM
 - NOT mainly because of communication overhead

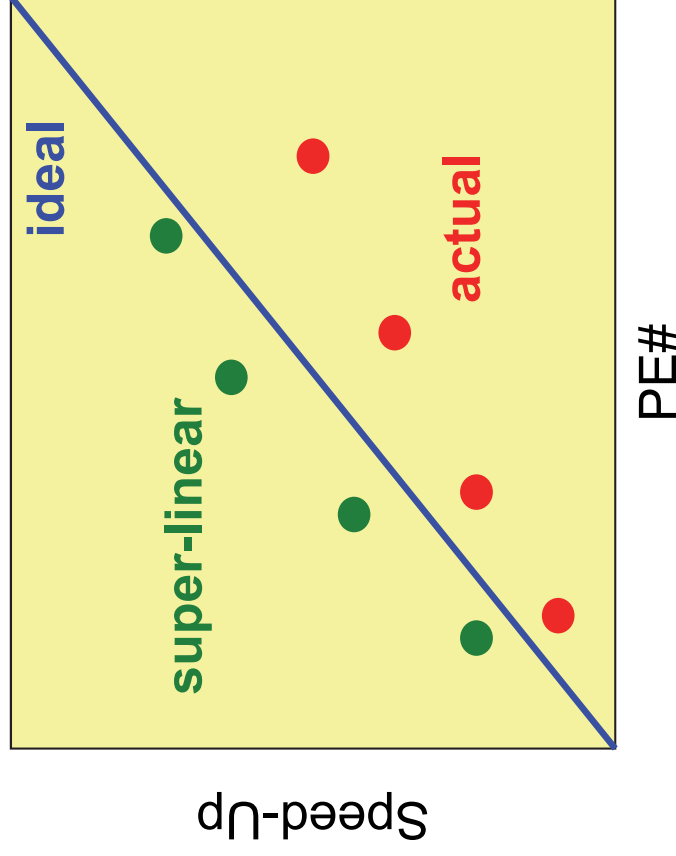


Not so significant in S1-3

- ◆ : $N=10^6$, ● : 10^8 , ▲ : 10^9 , — : Ideal
- Based on performance at a single core (sec.)
- Trapezoidal rule:
requirement for
memory is very
small (no arrays),
NON memory-
bound application

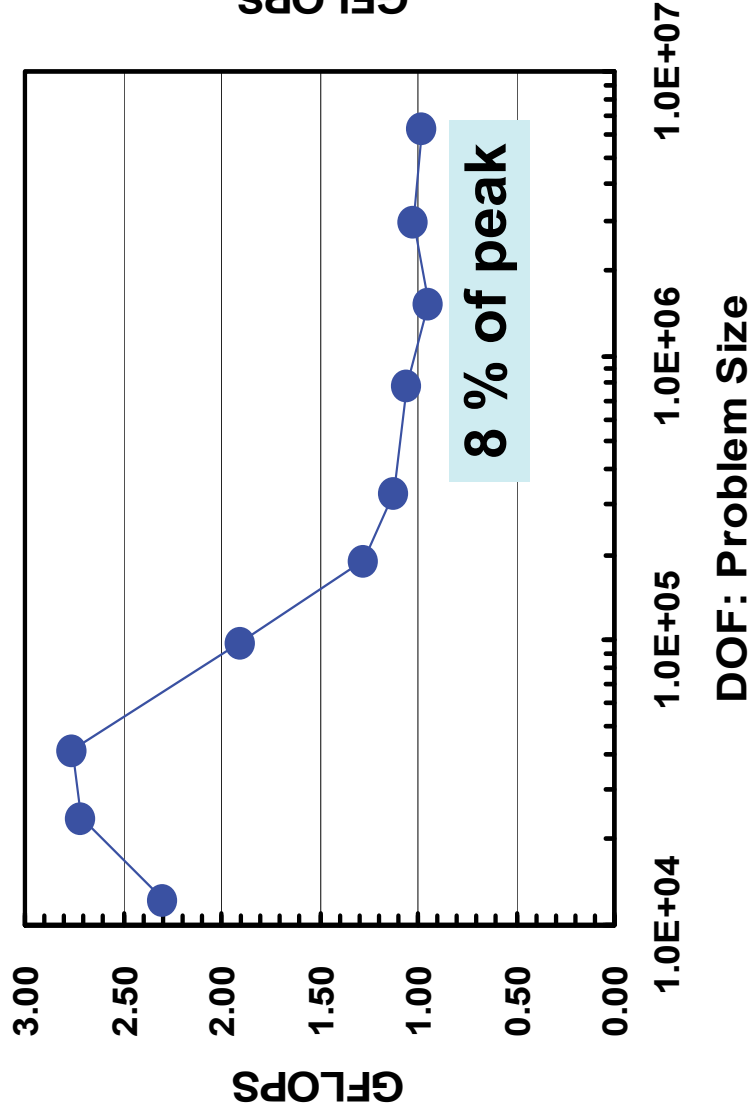


Super-Linear in Strong Scaling



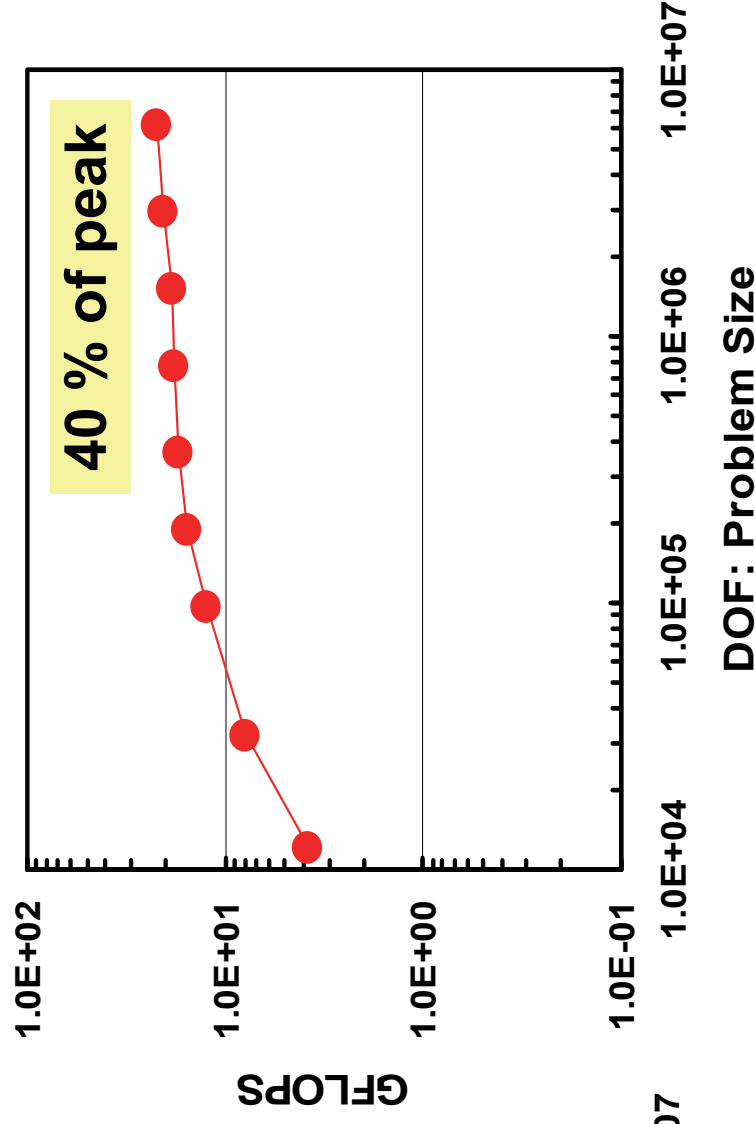
- In strong scaling case where entire problem size is fixed, performance is generally lower than the ideal one due to communication overhead.
- But sometime, actual performance may be better than the ideal one. This is called “super-linear”
 - only for scalar processors
 - does not happen in vector processors

Typical Behaviors



IBM-SP3:

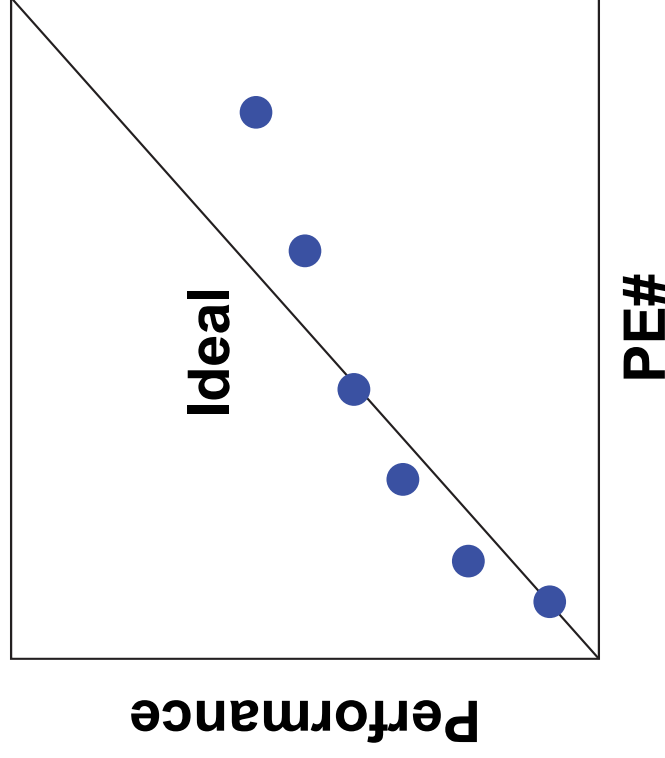
Higher performance for small problems, effect of cache



Earth Simulator:

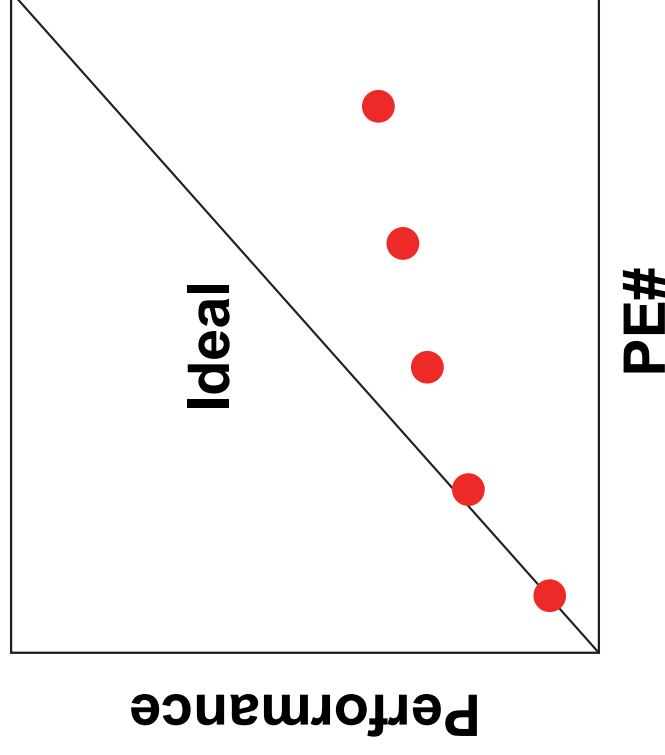
Higher performance for large-scale problems with longer loops

Strong Scaling



IBM-SP3:

“Super-linear” happens if number of PE is not so large. Performance is getting worse due to communication overhead if PE number is larger.



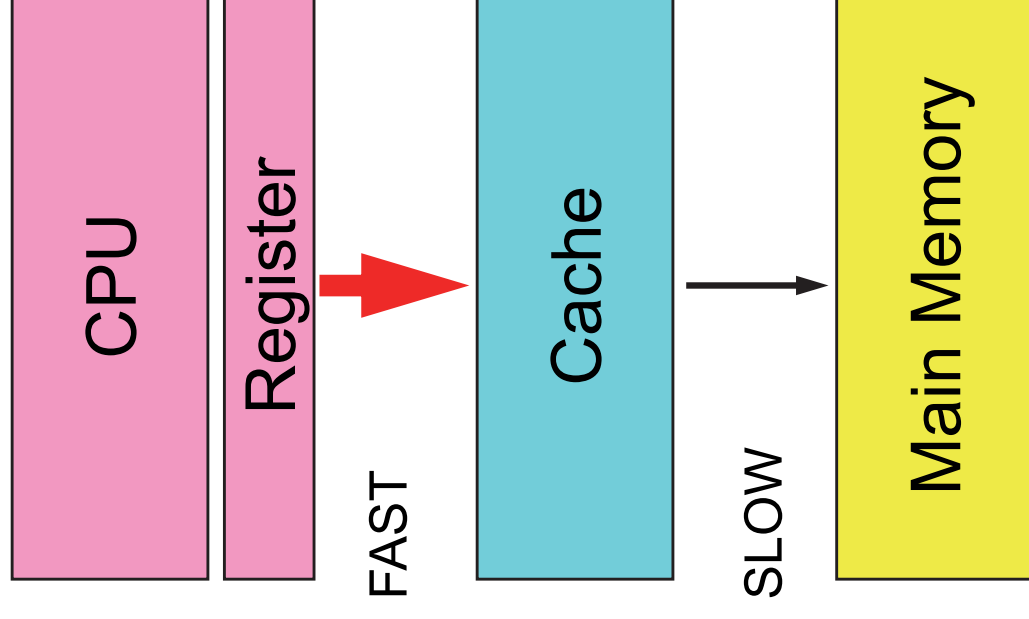
Earth Simulator:

Performance is getting worse due to communication overhead and smaller loop length if PE number is larger.



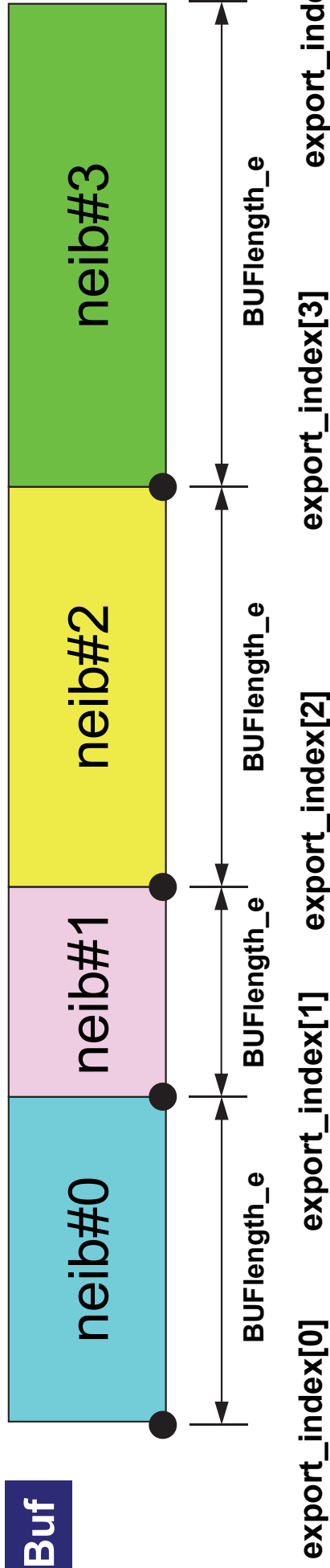
Why does “Super-Linear” happen ?

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



Memory Copy is expensive (1/2)

SendBuf



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

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

Copied to sending buffers

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

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

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

Memory Copy is expensive (1/2)

```

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

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

MPI_Waitall(NeibPETot, ReqRecv, StatRecv);

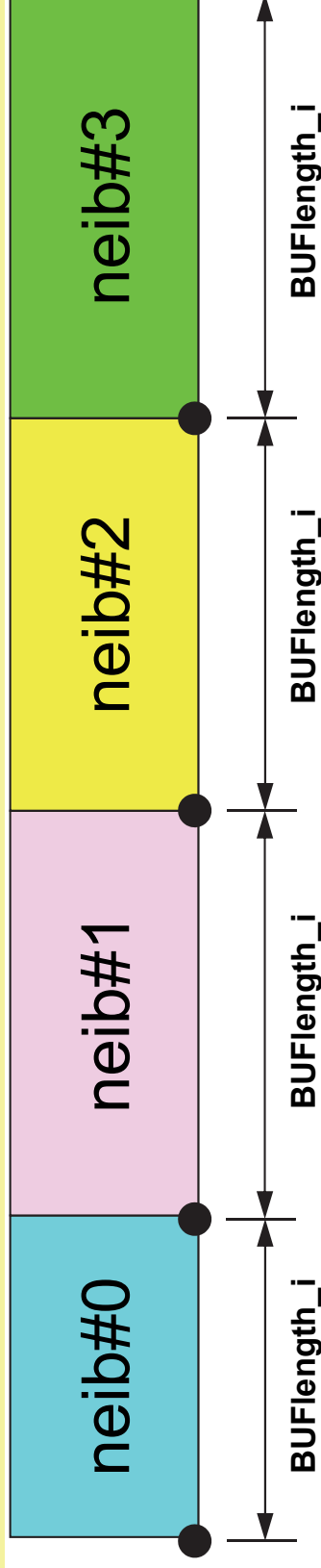
for (neib=0; neib<NeibPETot;neib++){
    for (k=import_index[neib];k<import_index[neib+1];k++){
        kk= import_item[k];
        VAL[kk]= RecvBuf[k];
    }
}

```

Copied from receiving buffer

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

RecvBuf



import_index[0] import_index[1] import_index[2] import_index[3] import_index[4]

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