Parallelization of Fully Distributed dense Matrix-Matrix Multiplication (2)

名古屋大学情報基盤中心　教授　片桐孝洋

Takahiro Katagiri, Professor,
Information Technology Center, Nagoya University

台大数学科学中心　科学計算冬季学校

Agenda

1. Execute sample program of fully distributed matrix-matrix multiplication
2. Explanation of sample program
3. Homework 5: fully distributed matrix-matrix multiplication
4. Hints of parallelization

Execute sample program (Matrix-matrix Multiplication (2))

Note: sample program of matrix-matrix multiplication

- Common file name of C/Fortran languages:

  Mat-Mat-d-fx.tar

- Modify queue name from lecture to lecture7 in job script file mat-mat-d.bash. Then type "pjsub".
  - lecture : Queue in out of time of this lecture.
  - lecture7: Queue in time of this lecture.

Execute sample program of dense matrix-matrix multiplication (2)

- Type the follows in command line:

```
$  cp  /home/z30082/Mat-Mat-d-fx.tar  ./
$  tar  xvf  Mat-Mat-d-fx.tar
$  cd  Mat-Mat-d
```

- Choose the follows:

```
$  cd  C   : For C language.
$  cd  F   : For Fortran language.
```

- The follows are common:

```
$  make
$  pjsub  mat-mat-d.bash
```

- After finishing the job, type the follow:

```
$  cat  mat-mat-d.bash.oXXXXXX
```

Output of sample program of matrix-matrix multiplication (C Language)

- You can see the followings if it runs successfully.

```
N  = 384
Mat-Mat time  = 0.000135 [sec.]
 841973.194818 [MFLOPS]
 Error! in ( 0 , 2 )-th argument in PE 0
 Error! in ( 0 , 2 )-th argument in PE 61
 Error! in ( 0 , 2 )-th argument in PE 51
 Error! in ( 0 , 2 )-th argument in PE 59
 Error! in ( 0 , 2 )-th argument in PE 50
 Error! in ( 0 , 2 )-th argument in PE 58
….
```

It is true execution for printing errors, because it does not finish parallelization.

Output of sample program of matrix-matrix multiplication (Fortran Language)

- You can see the followings if it runs successfully.

```
NN  =  384
Mat-Mat time  =  1.295508991461247E-03
MFLOPS =  87414.45135502046
 Error! in ( 1 , 3 )-th argument in PE  0
 Error! in ( 1 , 3 )-th argument in PE  61
 Error! in ( 1 , 3 )-th argument in PE  51
 Error! in ( 1 , 3 )-th argument in PE  58
 Error! in ( 1 , 3 )-th argument in PE  55
 Error! in ( 1 , 3 )-th argument in PE  63
 Error! in ( 1 , 3 )-th argument in PE  60
```

It is true execution for printing errors, because it does not finish parallelization.

Explanation of sample program (C Language)

- #define N 384
  - You can change size of matrix by varying the number.
- #define DEBUG 1
  - Results of matrix-matrix multiplication can be verified by setting "1".
- Specification of MyMatMat function
  - Return result of matrix-matrix multiplication of A with (N/NPROCS) x N of **double** times B with N x (N/NPROCS) of **double,** in C with (N/NPROCS) x N of **double**.

Explanation of sample program (Fortran Language)

- The declaration of size of N can be found in the following name of file: mat-mat-d.inc
- The size of matrix is defined as variable NN as follows:

```
integer  NN
parameter (NN=384)
```

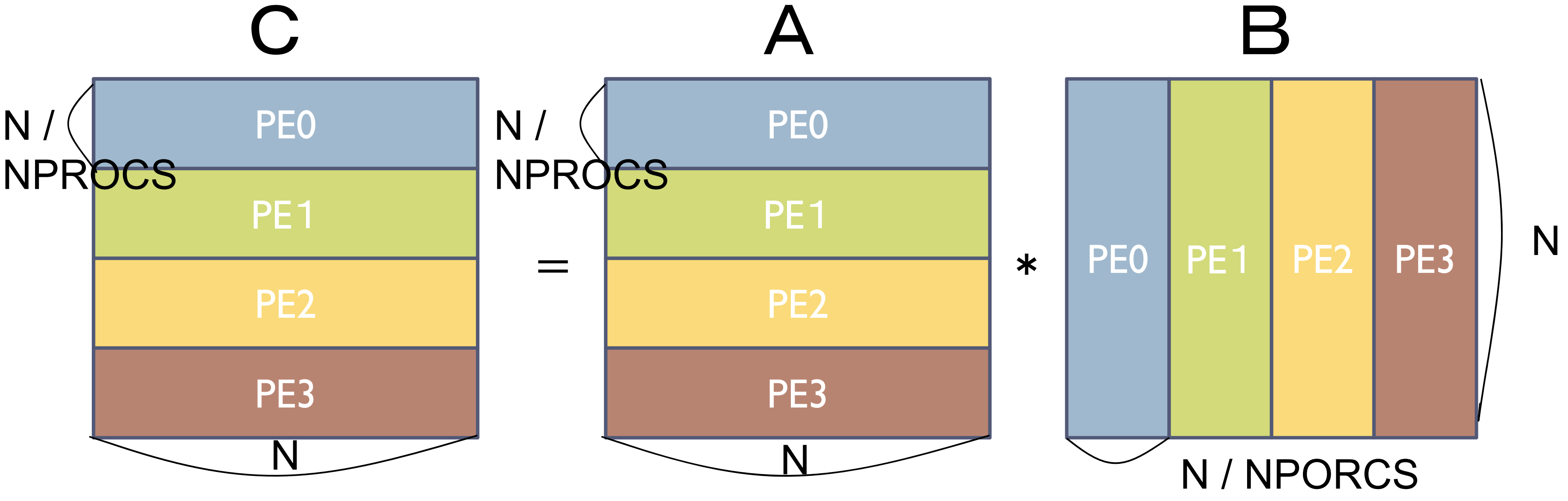
Homework 5

- Parallelize MyMatMat function (procedure):
  - For debugging, use:
    #define N 384

- To parallelize this, please take care initial data distributions for A, B and C.

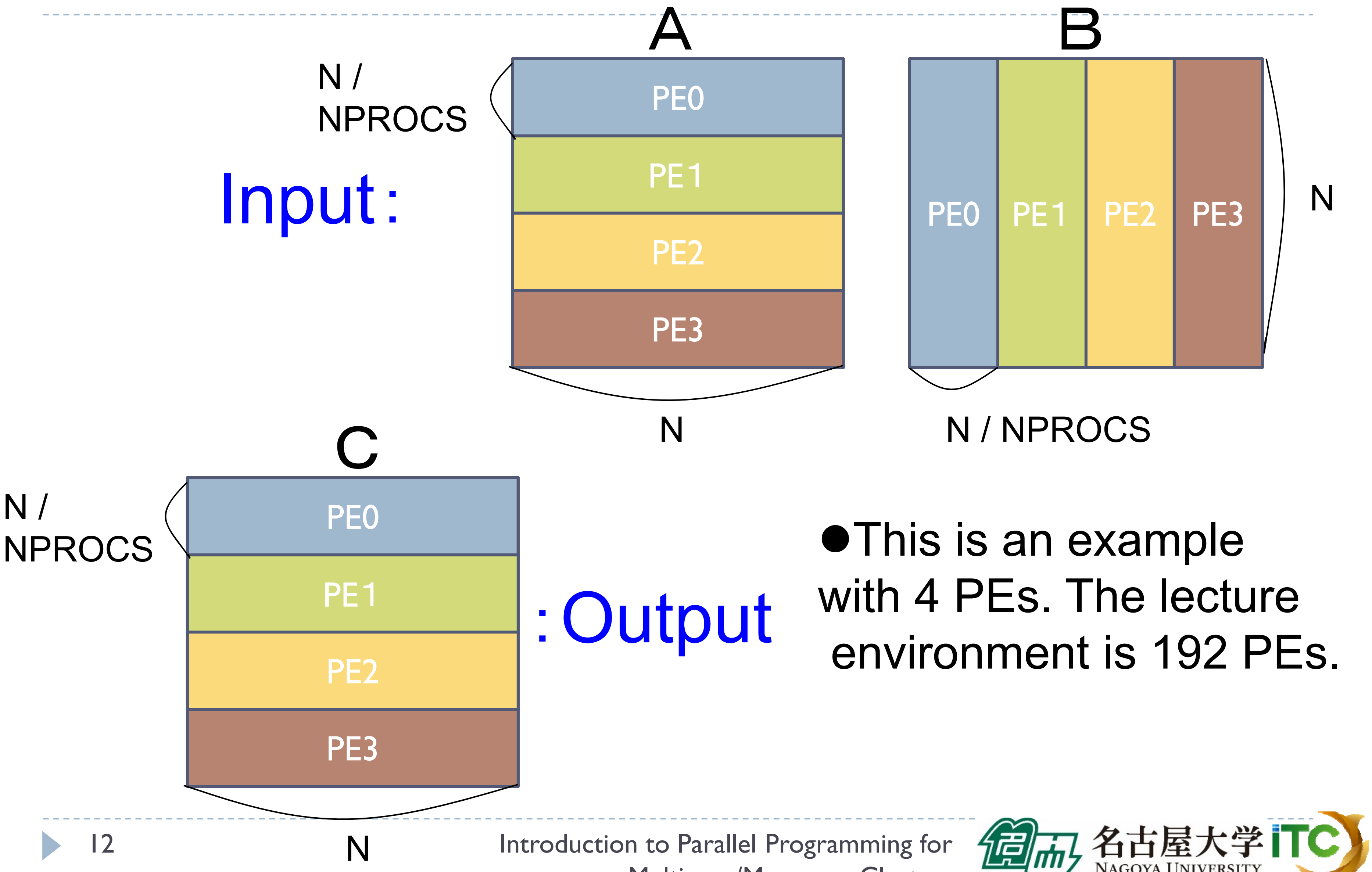
Initial data distributions for A, B and C

- The follows is recommended initial distribution for A, B and C. The following is an example with 4 PEs. The lecture environment is 192 PEs.



- It needs 1-to-1 communications.
- It needs a receive buffer in addition to arrays for matrices A, B, and C.

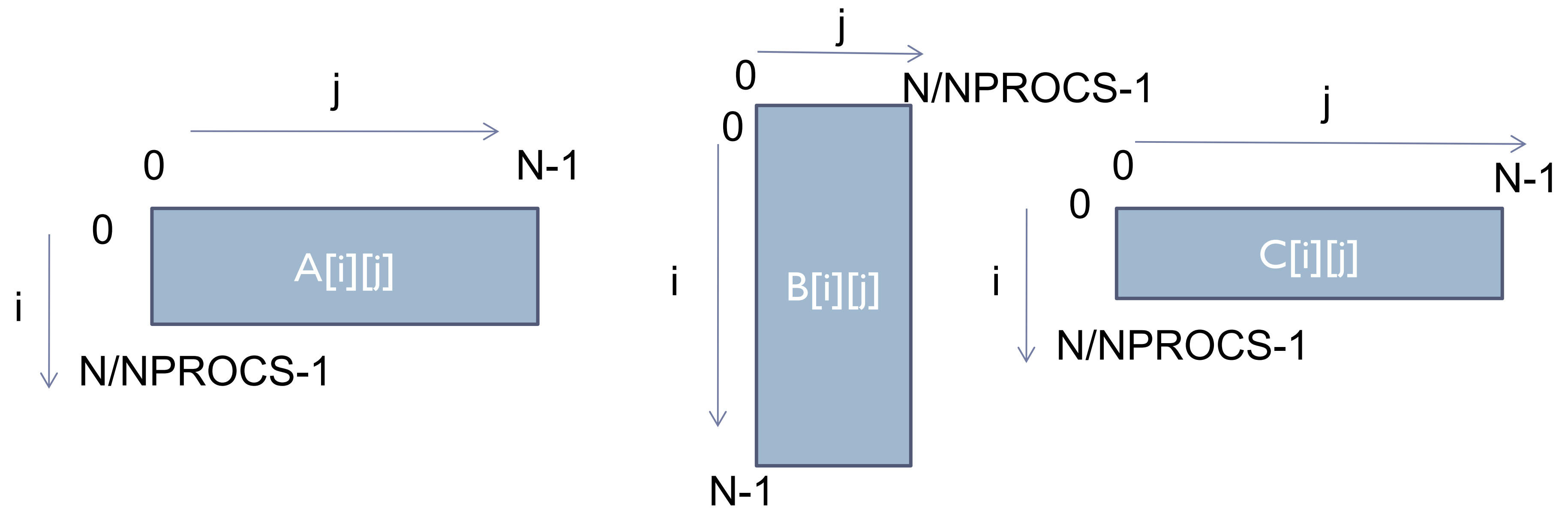
Specification of input and output



●This is an example with 4 PEs. The lecture environment is 192 PEs.

Note: Parallelization (C Language)

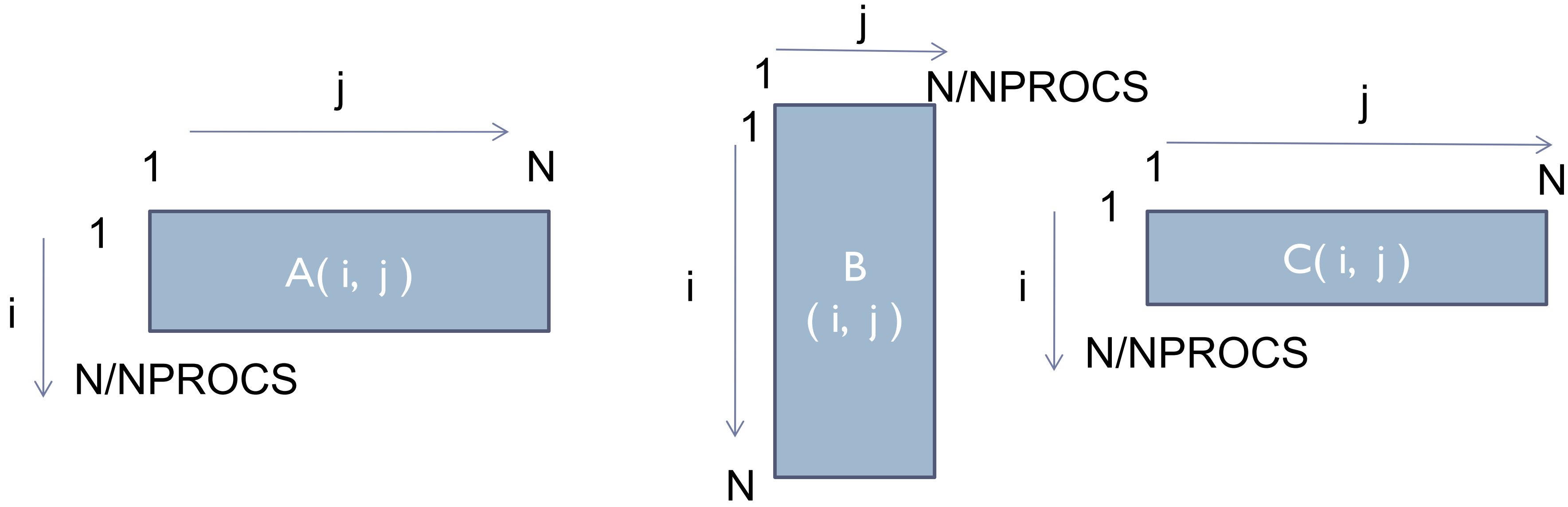
- Each element of array is totally distributed.
- In each PE, indexes of array are as follows:



- Take care of specification for local indexes in each PE for the matrix-matrix multiplication.

Note: Parallelization (Fortran Language)

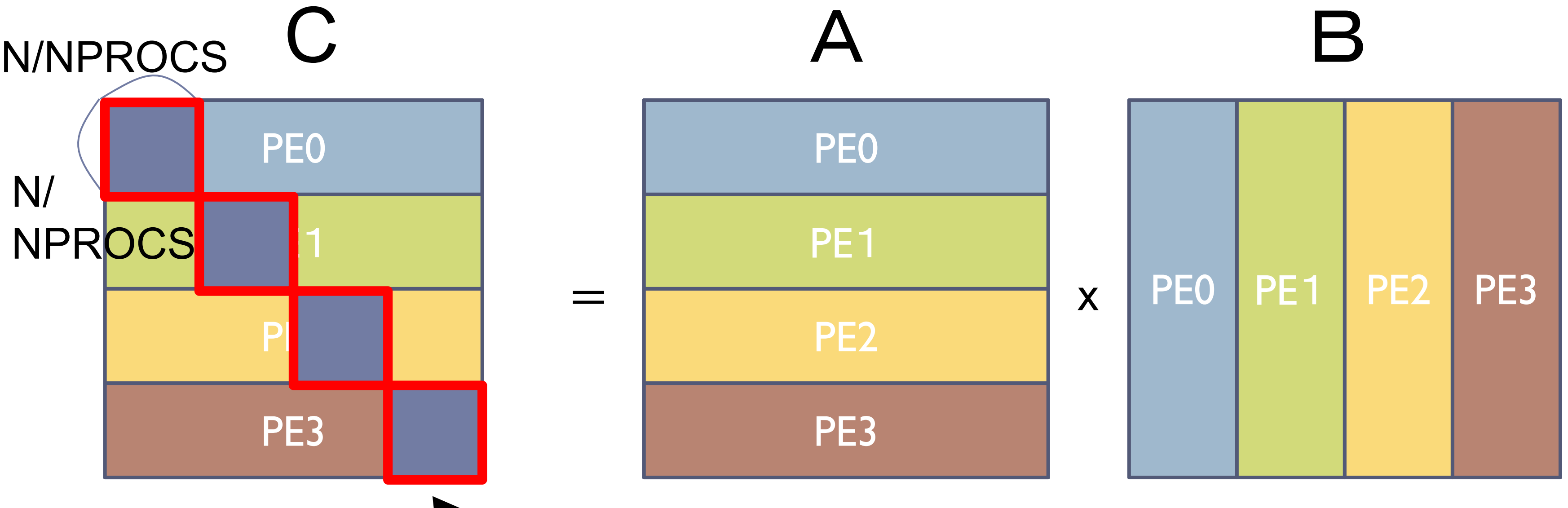
- Each element of array is totally distributed.
- In each PE, indexes of array are as follows:



- Take care of specification for local indexes in each PE for the matrix-matrix multiplication.

Hints of Parallelization

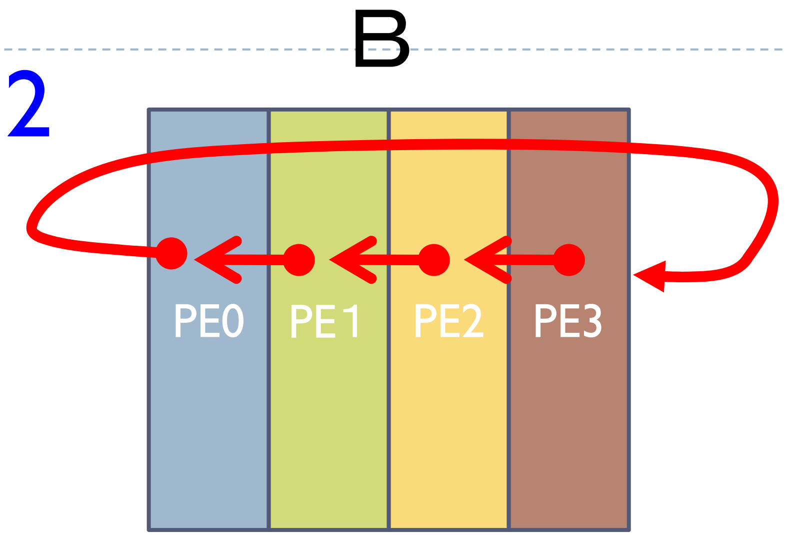
- A communication is needed for data of matrix B, since whole elements are not allocated in each PE to do matrix-matrix multiplication. One of parallel algorithms can be described as:
- Step 1



Results of partial matrix-matrix multiplication by using local data.

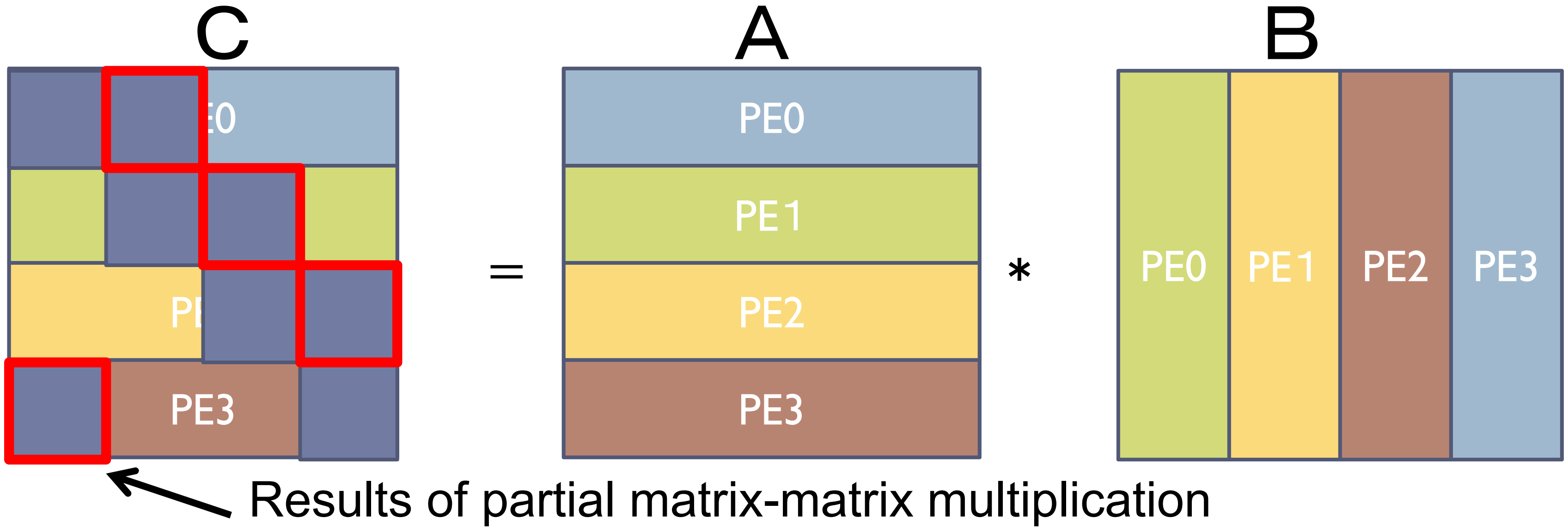
Hints of Parallelization

- Step 2



Send data to left-hand side PE (PE0 sends to PE3)

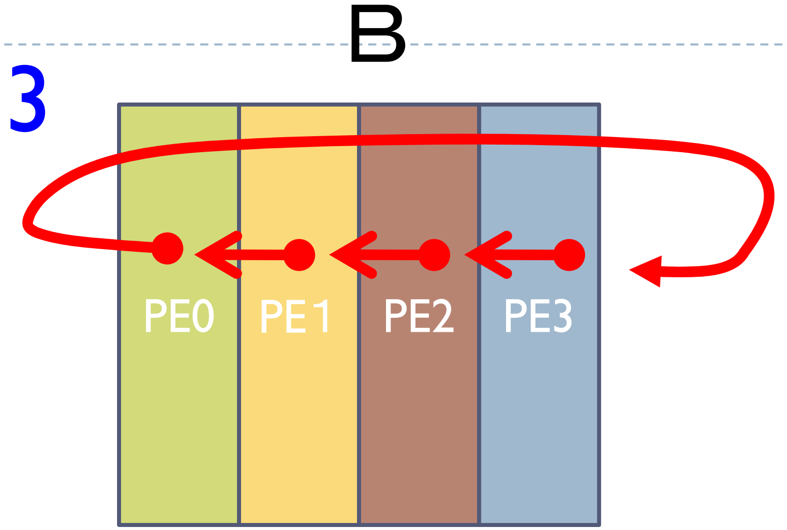
[cyclic left-hand side shift sending]



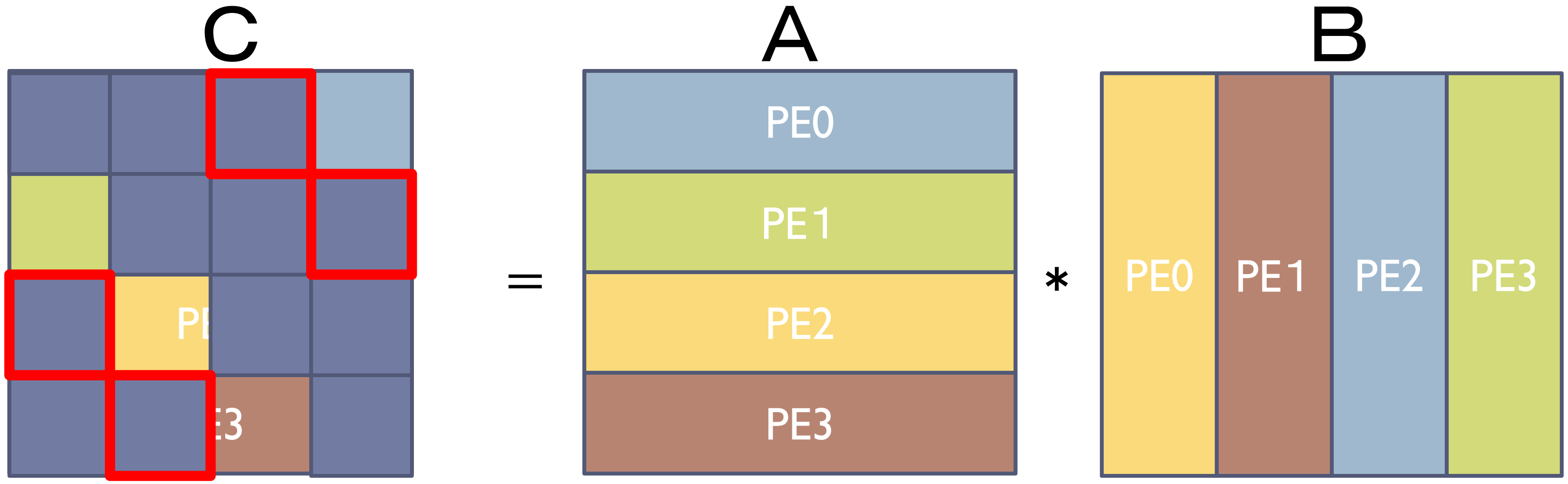
Results of partial matrix-matrix multiplication by using local data.

Hints of Parallelization

- Step 3



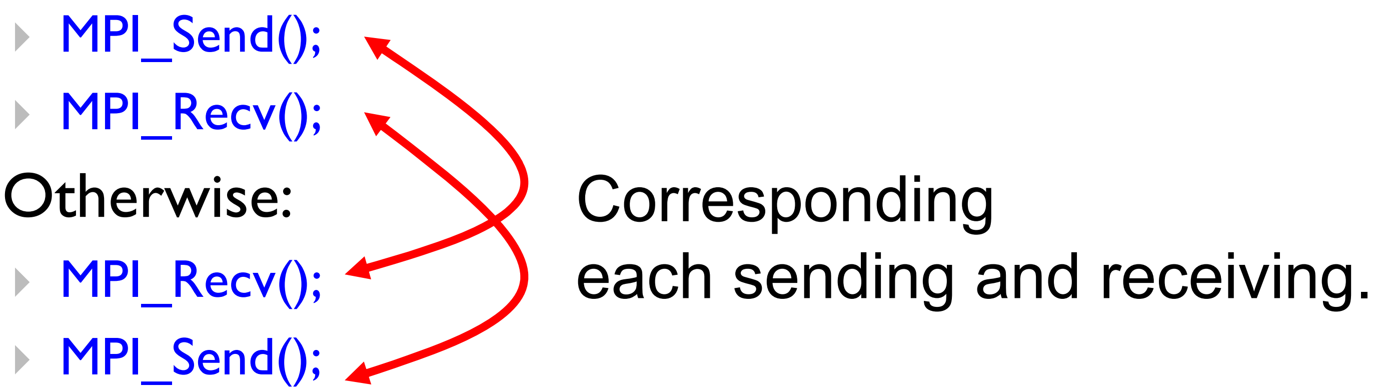
Send data to left-hand side PE (PE0 sends to PE3)
[cyclic left-hand side shift sending]



Results of partial matrix-matrix multiplication by using local data.

Note: Cyclic left-hand side shift sending

- If all PEs send data by using MPI_Send to implement cyclic left-hand side shift sending, process is stopped in that point. (Sometimes is working, but sometime is not working.)
  - When large message is sending in MPI_Send, system buffer is all used.
  - Waiting until the system buffer to be reused. (Spin waiting)
  - However never reuse it. (since there is no MPI_Recv in this world!).
- To avoid this, use the following implementation:

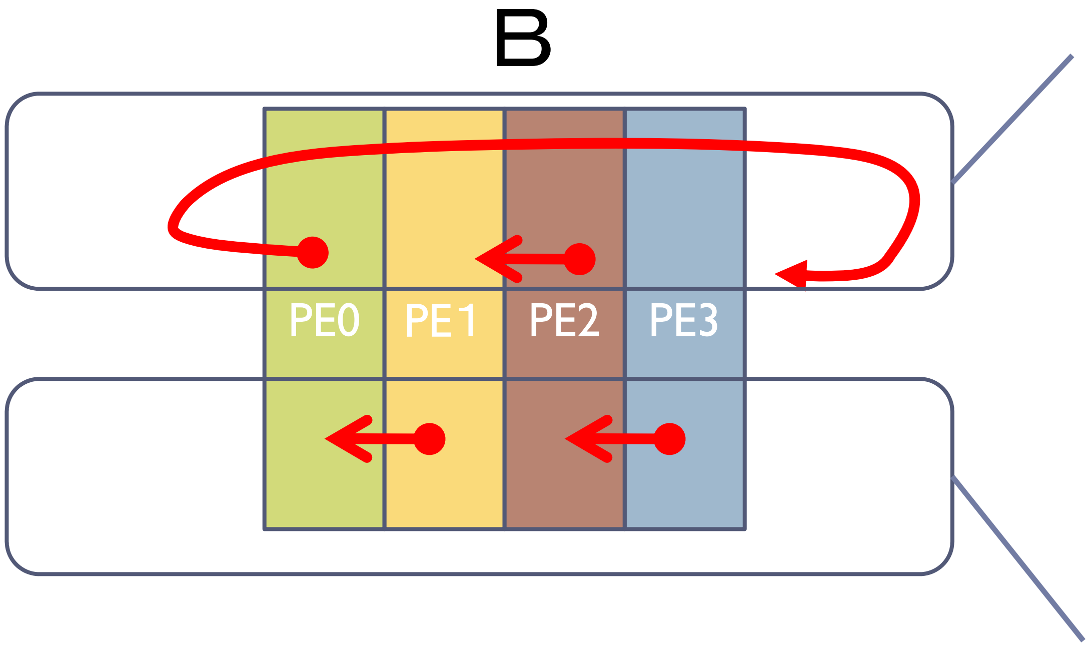


  - If rank number can be devisable by 2:
    - MPI_Send();
    - MPI_Recv();
  - Otherwise:
    - MPI_Recv();
    - MPI_Send();

Corresponding
each sending and receiving.

Note: Parallelization

- This means implementing the cyclic left-hand side shift sending with the following 2 steps.



Step 1:
Send data from PE which has rank number that can be dividable by 2.

Step 2:
Send data from PE which has rank number that can not be dividable by 2.

Basic Communication Function —MPI_Send

- ierr = MPI_Send(sendbuf, icount, idatatype, idest, itag, icomm);
  - sendbuf : Specify first address of sending area.
  - icount : Integer type. Specify number of elements for sending area.
  - idatatype : Integer type. Specify data type of sending area.
  - idest : Integer type. Specify rank number in communicator icomm.
  - itag : Integer type. Specify tags for receiving message.
  - icomm : Integer type. Specify communicator.
  - ierr (return value) : Integer type. An error code returns.

Basic Communication Function —MPI_Recv (1/2)

- ierr = MPI_Recv(recvbuf, icount, idatatype, isource,
  itag, icomm, istatus);

- recvbuf : Specify first address of receiving area.
- icount : Integer type. Specify number of elements for receiving area.
- idatatype : Integer type. Specify data type of receiving area.
  - MPI_CHAR (Character type) , MPI_INT (Integer type), MPI_FLOAT (float type), MPI_DOUBLE(double type)
- isource : Integer type. Specify rank number for receiving message.
  - If you want to receive any ranks, specify "MPI_ANY_SOURCE ".

Basic Communication Function —MPI_Recv (2/2)

- itag : Integer type. Specify tag number for receiving message.
  - If you want to receive any tag number, specify "MPI_ANY_TAG".
- icomm : Integer type. Specify communicator.
  - Normally, specify "MPI_COMM_WORLD "
- istatus : MPI_Status Type (Array of integer type.) Return status of receiving.
  - Declare an integer array with elements of MPI_STATUS_SIZE.
  - Number of rank that is sending message is stored in istatus[MPI_SOURCE], its tag is stored in istatus[MPI_TAG].
- ierr (return value) : Integer type. Return an error code.

Note: Implementation of Tags

- How to describe Tag (itag)?
  - Tag (itag) can be specified any values with **int** type for MPI_Send() and MPI_Recv().
  - However it is reasonable to specify different values of tag in each communication to know errors for the communications.
  - In this implementation, there are two pairs of MPI_Send() and MPI_Recv(). Hence we can describe different values of tags in each step.
  - For example, we use value of the outer loop induction variable, say iloop, for one communication in this algorithm. The other can be specified with iloop+NPROCS.

Additional Hints

Answer codes are shown.

Summary of the parallel implementation

1. The times of cyclic left-hand side shift sending is [total number of processes -1].
2. To receive data of array B[][], we need a buffer array B_T[][].
3. Copy the received B_T[][] to B[][] to do local matrix-matrix multiplication.
4. Initial indexes of diagonal blocks for the local matrix-matrix multiplication: Length of block * myid. The indexes are added with the length of block, but it should be set to 0 if it exceeds N.

Hints of parallelization (Almost answer code, C Language)

- The follows are overview of code.

```
ib = n/numprocs;
for (iloop=0;  iloop<NPROCS;  iloop++ )  {
    A local matrix-matrix multiplication  C  =  A * B;
    if (iloop != (numprocs-1) ) {
      if (myid % 2 == 0 ) {
          MPI_Send(B, ib*n, MPI_DOUBLE, isendPE,
                iloop, MPI_COMM_WORLD);
          MPI_Recv(B_T, ib*n, MPI_DOUBLE, irecvPE,
                iloop+numprocs, MPI_COMM_WORLD, &istatus);
      } else {
          MPI_Recv(B_T, ib*n, MPI_DOUBLE, irecvPE,
                iloop, MPI_COMM_WORLD, &istatus);
          MPI_Send(B, ib*n, MPI_DOUBLE, isendPE,
                iloop+numprocs, MPI_COMM_WORLD);
      }
      Copy B_T[ ][ ] to B[ ][ ];
    }
}
```

Hints of parallelization (Almost answer code, C Language)

- The follows are local matrix-matrix multiplication.

```
jstart=ib*( (myid+iloop)%NPROCS );
for (i=0;  i<ib;  i++) {
   for(j=0;  j<ib;  j++) {
       for(k=0;  k<n;  k++) {
          C[ i ][ jstart + j ] += A[ i ][ k ] * B[ k ][ j ];
         }
      }
 }
```

Hints of parallelization (Almost answer code, Fortran Language)

- The follows are overview of code.

```
ib = n/numprocs
do iloop=0,  NPROCS-1
   A local matrix-matrix multiplication C  =  A * B
   if (iloop .ne. (numprocs-1) ) then
     if (mod(myid, 2)  .eq. 0 ) then
        call MPI_SEND(B, ib*n, MPI_DOUBLE_PRECISION,  isendPE,
&             iloop, MPI_COMM_WORLD,  ierr)
        call MPI_RECV(B_T,  ib*n,  MPI_DOUBLE_PRECISION,  irecvPE,
&             iloop+numprocs, MPI_COMM_WORLD,  istatus,  ierr)
     else
        call MPI_RECV(B_T,  ib*n,  MPI_DOUBLE_PRECISION,  irecvPE,
&             iloop,  MPI_COMM_WORLD,  istatus,  ierr)
        call MPI_SEND(B,  ib*n,  MPI_DOUBLE_PRECISION,  isendPE,
&             iloop+numprocs,  MPI_COMM_WORLD,  ierr)
     endif
     Copy B_T to B
   endif
 enddo
```

Hints of parallelization (Almost answer code, Fortran Language)

- The follows are local matrix-matrix multiplication.

```
imod = mod( (myid+iloop), NPROCS )
jstart = ib* imod
do i=1, ib
  do j=1, ib
     do k=1, n
       C( i , jstart + j ) = C( i , jstart + j ) +  A( i , k ) * B( k , j )
      enddo
   enddo
 enddo
```