# 5240 CSE
# Parallel Processing
# Spring 2002

## Project
Matrix-Matrix Multiplication
Using
Fox's Algorithm and PBLAS Routine PDGEMM

***Due***
**March 24, 2002**

By
Abdullah I. Alkraiji
900075541

Instructor
Dr. Charles T Fulton

**Continents**



**References**


| NO | Name | Author |
|---|---|---|
| 1 | Parallel Programming with MPI | Peter S. Pacheco |
| 2 | Parallel Programming | Barry Wilkinson<br>Michael Allen |
| 3 | Introduction To Parallel Computing | Vipin Kumar<br>Ananth Grama<br>Anshul Gupta<br>George Karypis |
| 4 | Numerical Analysis | Richard Burden<br>J. Douglas Faires |

## Matrix Multiplication Introduction

Definition-1: ***A*** square matrix has the same number of rows as columns.
Definition-2: If $\boldsymbol{A} = (\boldsymbol{a_{ij}})$ and $\boldsymbol{B} = (\boldsymbol{b_{ij}})$ are square matrix of order n, then $\boldsymbol{C} = (\boldsymbol{c_{ij}}) = \boldsymbol{AB}$ is also a square matrix of order ***n***, and $\boldsymbol{c_{ij}}$ is obtained by taking the dot product of the ***i***th row of ***A*** with the ***j***th column of ***B***. That is,

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + a_{i,n-1}b_{n-1,j}.$$

## Parallel Algorithms for Square Matrices Multiplication

There are several algorithms for square matrix multiplication, some of them are not realistic; for the rest, each one has advantages and disadvantages. Following are examples of both types:

1- First Algorithm is to assign each row of ***A*** to all of ***B*** to be sent to each process. For simplicity, we assume that the number of processes is the same as the order of the matrices $\boldsymbol{p} = \boldsymbol{n}$. Here's algorithm for this method:

```
Send all of B to each process
Send ith row of A to ith process
Compute ith row of C on ith process
```

This algorithm would work assuming that there is a sufficient storage for each process to store all of B. This algorithm will not be implemented for large ***n*** since it is not storage efficient.

2- Second Algorithm, suppose that the number of processes is the same as the order of the matrices $\boldsymbol{p} = \boldsymbol{n}$, and suppose that we have distributed the matrices by rows. So process ***0*** is assigned row ***0*** of ***A***, ***B***, and ***C***; process ***1*** is assigned row ***1*** of ***A***, ***B***, and ***C***; etc. Then in order to form the dot product of the ***i***th row of ***A*** with the ***j***th column of ***B***, we will need to gather the ***j***th column of ***B*** onto process ***i***. But we will need to form the dot product of the ***j***th column with every row of ***A***. Here's algorithm for this method:

```
for  each row of  C
     for each column of  C
     {
          C [row] [column] = 0.0;
           for each element of this row of  A
                Add  A[row] [element]  *  B[element] [column] to  C[row] [column]
     }
```

Observe that a straightforward parallel implementation will be quite costly because this will involve a lot of communication. Similar reasoning shows that an algorithm that distributes the matrices by columns or that distributes each row of ***A*** and each column of ***B*** to each process will also involve large amounts of communication.

## Checkerboard

Based on the above considerations:

1- The excessive use of storage
2- The communication overhead

Most parallel matrix multiplication functions use a checkerboard distribution of the matrices. This means that the processes are viewed as a grid, and, rather than assigning entire rows or entire columns to each process, we assign small sub-matrices. For example, if we have four processes, we might assign the element of a 4 X 4 matrix as shown below, checkerboard mapping of a 4 X 4 matrix to four processes.

| **Process 0** | **Process 1** |
|---|---|
| $a_{00}$ $a_{01}$ | $a_{02}$ $a_{03}$ |
| $a_{10}$ $a_{11}$ | $a_{12}$ $a_{13}$ |
| **Process 2** | **Process 3** |
| $a_{20}$ $a_{21}$ | $a_{22}$ $a_{23}$ |
| $a_{30}$ $a_{31}$ | $a_{32}$ $a_{33}$ |

## Fox's Algorithm

Fox's algorithm is a one that distributes the matrix using a checkerboard scheme like the above. In order to simplify the discussion, let's assume that the matrices have order $n$, and the number of processes, $p$, equals $n^2$. Then a checkerboard mapping assigns $a_{ij}$, $b_{ij}$, and $c_{ij}$ to process (i,j). In a process grid like the above, the process (i,j) is the same as process p= i * n + j, or, loosely, process (i,j) using row major form in the process grid.

Fox's algorithm for matrix multiplication proceeds in n stages: one stage for each term $a_{ik}b_{kj}$ in the dot product

$$C_{ij} = a_{i0}b_{0j} + a_{i1}b_{1i} + \ldots + a_{i,n-1}b_{n-1,j}.$$

During the initial stage, each process multiplies the diagonal entry of $A$ in its process row by its element of $B$:

Stage ***0*** on process(i,j): $c_{ij}=a_{ii}b_{ij}$.

During the next stage, each process multiplies the element immediately to the right of the diagonal of $A$ by the element of $B$ directly beneath its own element of $B$:

Stage ***1*** on process(i,j): $c_{ij} = c_{ij} + a_{i,i+1}b_{i+1,j}$.

In general, during the ***k***th stage, each process multiplies the element ***k*** columns to the right of the diagonal of ***A*** by the element ***k*** rows below its own element of ***B***:

Stage ***k*** on process(i,j): $c_{ij} = c_{ij} + a_{i,i+k}b_{i+k,j}$.

Of, course, we can't just add ***k*** to a row or column subscript and expect to always get a valid row or column number. For example, if i = j = n –1, then any positive value added to ***i*** or ***j*** will result in an out-of-range subscript. One possible solution is to use subscripts module ***n***. That is, rather than use i + k for a row or column subscript, use i + k mode n. Perhaps we should say that the incomplete algorithm is correct. We still haven't said how we arrange that each process gets the appropriate values $a_{i,k}$ and $b_{k,i}$. Thus, we need to broadcast $a_{ik}$ across the ***i***th row before each multiplication. Finally, observe that during the initial stage, each process uses its own element, $b_{ij}$, of ***B***. During subsequent stages, process(i,j) will use $b_{ik}$. Thus, after each multiplication is completed, the element of ***B*** should be shifted up one row, and elements in the top row should be sent to the bottom row.

**Example-1:** illustrates the stages in Fox's algorithm for multiplying two 2 X 2 matrices distributes across four processes.

| Stages | First Step | | Second Step | | Third Step | |
|---|---|---|---|---|---|---|
| Stage 0 | $a_{00}$ | $a_{01}$ | $c_{00} = a_{00}b_{00}$ | $c_{01} = a_{00}b_{01}$ | $b_{00}$ | $b_{01}$ |
| | $a_{10}$ | $a_{11}$ | $c_{10} = a_{11}b_{10}$ | $c_{11} = a_{11}b_{11}$ | $b_{10}$ | $b_{11}$ |
| Stage 1 | $a_{00}$ | $a_{01}$ | $c_{00} = c_{00} + a_{01}b_{10}$ | $c_{01} = c_{01} + a_{01}b_{11}$ | $b_{10}$ | $b_{11}$ |
| | $a_{10}$ | $a_{11}$ | $c_{10} = c_{10} + a_{10}b_{00}$ | $c_{11} = c_{11} + a_{10}b_{01}$ | $b_{00}$ | $b_{01}$ |

**First step**: each blue element in the ***i***th row of ***A*** will be multiplied by all of the corresponding ***i***th row of ***B*** in the third step.
**Second step**: assigns the result of the multiplications to $c_{ij}$.
**Third Step**: after each stage, the elements of ***B*** should be shifted up one row, and elements in the top row should be sent to the bottom row.

**Example-2**: illustrates the stages in Fox's algorithm for multiplying two 3 X 3 matrices distributed across nine processes.

| Stages | First Step | | | Second Step | | | Third Step | | |
|---|---|---|---|---|---|---|---|---|---|
| Stage 0 | $a_{00}$ | $a_{01}$ | $a_{02}$ | $c_{00}$+=$a_{00}b_{00}$ | $c_{01}$+=$a_{00}b_{01}$ | $c_{02}$+=$a_{00}b_{02}$ | $b_{00}$ | $b_{01}$ | $b_{02}$ |
| | $a_{10}$ | $a_{11}$ | $a_{12}$ | $c_{10}$+=$a_{11}b_{10}$ | $c_{11}$+=$a_{11}b_{11}$ | $c_{12}$+=$a_{11}b_{12}$ | $b_{10}$ | $b_{11}$ | $b_{12}$ |
| | $a_{20}$ | $a_{21}$ | $a_{22}$ | $c_{20}$+=$a_{22}b_{20}$ | $c_{20}$+=$a_{22}b_{21}$ | $c_{22}$+=$a_{22}b_{22}$ | $b_{20}$ | $b_{21}$ | $b_{22}$ |
| Stage 1 | $a_{00}$ | $a_{01}$ | $a_{02}$ | $c_{00}$+=$a_{01}b_{10}$ | $c_{01}$+=$a_{01}b_{11}$ | $c_{02}$+=$a_{01}b_{12}$ | $b_{10}$ | $b_{11}$ | $b_{12}$ |
| | $a_{10}$ | $a_{11}$ | $a_{12}$ | $c_{10}$+=$a_{12}b_{20}$ | $c_{11}$+=$a_{12}b_{21}$ | $c_{12}$+=$a_{12}b_{22}$ | $b_{20}$ | $b_{21}$ | $b_{22}$ |
| | $a_{20}$ | $a_{21}$ | $a_{22}$ | $c_{20}$+=$a_{20}b_{00}$ | $c_{21}$+=$a_{20}b_{01}$ | $c_{22}$+=$a_{20}b_{02}$ | $b_{00}$ | $b_{01}$ | $b_{02}$ |
| Stage 2 | $a_{00}$ | $a_{01}$ | $a_{02}$ | $c_{00}$+=$a_{02}b_{20}$ | $c_{01}$+=$a_{02}b_{21}$ | $c_{02}$+=$a_{02}b_{22}$ | $b_{20}$ | $b_{21}$ | $b_{22}$ |
| | $a_{10}$ | $a_{11}$ | $a_{12}$ | $c_{10}$+=$a_{10}b_{00}$ | $c_{11}$+=$a_{10}b_{01}$ | $c_{12}$+=$a_{10}b_{02}$ | $b_{00}$ | $b_{01}$ | $b_{02}$ |
| | $a_{20}$ | $a_{21}$ | $a_{22}$ | $c_{20}$+=$a_{21}b_{10}$ | $c_{21}$+=$a_{21}b_{11}$ | $c_{22}$+=$a_{21}b_{12}$ | $b_{10}$ | $b_{11}$ | $b_{12}$ |

**Example-3**: illustrates the stages in Fox's algorithm for multiplying two 4 X 4 matrices distributed across four processes.

For large ***n*** it is that we can unlikely access to $n^2$ processors. So a natural solution would seem to be to store sub-matrices rather than matrix elements on each process. We use a square grid of processes, where the number of process rows or process columns, ***sqrt(p)***, evenly divides ***n***. With this assumption, each process is assigned a square

$$n/sqrt(p) \quad \mathbf{x} \quad n/sqrt(p)$$

sub-matrix of each of the three matrices. In our example p = 4, and n = 4, so the sub-matrices for matrix of A will be as following:

$$A_{00} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \qquad A_{01} = \begin{bmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{bmatrix}$$

$$A_{10} = \begin{bmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{bmatrix} \qquad A_{11} = \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix}$$

If we make similar definition of $B_{ij}$ and $C_{ij}$, assign $A_{ij}$, $B_{ij}$, and $C_{ij}$ to process (i,j), and we define ***q = sqrt(p)***, then our algorithm will compute

$$C_{ij} = A_{ii}B_{ij} + A_{i,i+1}B_{i+1,j} + \ldots + A_{i,q-1}B_{q-1,j} + A_{i0}B_{0j} + \ldots + A_{i,i-1}B_{i-1,j}.$$

The stages for computing $C_{ij}$ will be as following:

| Q | First Step | | Second Step | | Third Step | |
|---|---|---|---|---|---|---|
| q = 0 | $A_{00}$ | $A_{01}$ | $C_{00} = A_{00}B_{00}$ | $C_{01} = A_{00}B_{01}$ | $B_{00}$ | $B_{01}$ |
| | $A_{10}$ | $A_{11}$ | $C_{10} = A_{11}B_{10}$ | $C_{11} = A_{11}B_{11}$ | $B_{10}$ | $B_{11}$ |
| q = 1 | $A_{00}$ | $A_{01}$ | $C_{00} = C_{00} + A_{01}B_{10}$ | $C_{01} = C_{01} + A_{01}B_{11}$ | $B_{10}$ | $B_{11}$ |
| | $A_{10}$ | $A_{11}$ | $C_{10} = C_{10} + A_{10}B_{00}$ | $C_{11} = C_{11} + A_{10}B_{01}$ | $B_{00}$ | $B_{01}$ |

**Fox's Algorithm Psuedocode**

```
/* my process  row = i ,  my process  column = j */
q = sqrt(p);
dest = ((i – 1)  mod  q , j);
source = ((i + 1)  mod  q, j);
for (stage = 0  ; stage < q ; stage++)
{
            k_bar = (i + stage)  mod  q;
    (a)     Broadcast A[i,k_bar] across process row i;
    (b)     C[i,j] = C[i,j] + A[i,k_bar]*B[k_bar,j];
    (c)     Send B[(k_bar+1)  mod  q, j]  to dest;
            Receive B[(k_bar+1)  mod  q, j]  from source;
}
```

| Process(0,0) | Process(0,1) |
|---|---|
| 1- i = j = 0 | 1- i = 0,  j = 1 |
| 2- Assign $A_{00}$,$B_{00}$,$C_{00}$ to this process(0,0) | 2- Assign $A_{01}$,$B_{01}$,$C_{01}$ to this process(0,1) |
| 3- q = 2 | 3- q = 2 |
| 4- dest = (1,0) | 4- dest= (1,1) |
| 5- source = (1,0) | 5- source= (0,1) |
| 6- stage = 0 | 6- stage = 0 |
| 7- k_bar = 0 | 7- k_bar = 0 |
| 8- Broadcast $A_{00}$ across process row i = 0 | 8- Broadcast $A_{00}$ across process row i = 0 |
| **9- $C_{00} = C_{00} + A_{00} * B_{00}$** | **9- $C_{01} = C_{01} + A_{00} * B_{01}$** |
| 10- Send $B_{00}$ to dest = (1,0) | 10- Send $B_{11}$ to dest = (1,1) |
| 11- Receive $B_{10}$ from source = (1,0) | 11- Receive $B_{11}$ from source = (0,1) |
| 12- stage = 1 | 12- stage = 1 |
| 13- k_bar = 1 | 13- k_bar = 1 |
| 14- Broadcast $A_{01}$ across process row i = 0 | 14- Broadcast $A_{01}$ across process row i = 0 |

| | |
|---|---|
| **15- $C_{00} = C_{00} + A_{01}B_{10}$**<br>16- Send $B_{00}$ to dest = (1,0)<br>17- Receive $B_{00}$ from source = (1,0) | **15- $C_{01} = C_{01} + A_{01} * B_{11}$**<br>16- Send $B_{01}$ to dest = (1,1)<br>17- Receive $B_{01}$ from source = (0,1) |
| Process(1,0)<br><br>1- i = 1, j = 0<br>2- Assign $A_{10}$,$B_{10}$,$C_{10}$ to this process(1,0)<br>3- q = 2<br>4- dest = (0,0)<br>5- source = (0,0)<br>6- stage = 0<br>7- k_bar = 1<br>8- Broadcast $A_{11}$ across process row i = 1<br>**9- $C_{00} = C_{00} + A_{11} * B_{10}$**<br>10- Send $B_{00}$ to dest = (0,0)<br>11- Receive $B_{00}$ from source = (0,0)<br>12- stage = 1<br>13- k_bar = 0<br>14- Broadcast $A_{10}$ across process row i = 0<br>**15- $C_{00} = C_{00} + A_{10}B_{00}$**<br>16- Send $B_{10}$ to dest = (0,0)<br>17- Receive $B_{10}$ from source = (0,0) | Process(1,1)<br><br>1- i = 1, j = 1<br>2- Assign $A_{11}$,$B_{11}$,$C_{11}$ to this process(1,1)<br>3- q = 2<br>4- dest = (0,1)<br>5- source = (0,1)<br>6- stage = 0<br>7- k_bar = 1<br>8- Broadcast $A_{11}$ across process row i = 1<br>9- $C_{11} = C_{11} + A_{11} * B_{11}$<br>10- Send $B_{01}$ to dest = (0,1)<br>11- Receive $B_{01}$ from source = (0,1)<br>12- stage = 1<br>13- k_bar = 0<br>14- Broadcast $A_{10}$ across process row i = 1<br>15- $C_{11} = C_{11} + A_{10}B_{01}$<br>16- Send $B_{11}$ to dest = (0,1)<br>17- Receive $B_{11}$ from source = (0,1) |

## Implementation of Fox's Algorithm

In this section, we will describe the MPI Datatype, MPI Function Calls, and Functions that are used in our Fox's program.

### MPI Data Type

1- MPI_Comm: It a communicator that contains all of the processes.
2- MPI_Datatype: The principle behind MPI's derived datatypes is to provide all of the information except the address of the beginning of the message in a new MPI datatype. Then, when a program calls MPI_Send, MPI_Recv, etc., it simply provides the address of the first element, and the communication subsystem can determine exactly what needs to be sent or received.

3- MPI_Status: It is a structure that consists of five elements, and the descriptions for the three elements are:
   - astatus.source = process source of received data
   - status.tag = tag(integer) of received data
   - status.error = error status of reception
4- MPI_Aint: It declares the variable to be an integer type that holds any valid address.

**MPI Function Calls:**

1- MPI_Init: Before any other MPI functions can be called, the function MPI_Init must be called, and it should only be called once. Here's he syntax for this function MPI_Init(&argc, &argv); .
2- MPI_Comm_rank: It returns the rank of a process in its second parameter. The first parameter is a communicator. Essentially a communicator is a collection of processes that can send messages to each other. Its syntax as following:

```
int MPI_Comm_rank(MPI_Comm comm, int* my_rank)
```

3- MPI_Bcast: It is a collective communication in which a single process sends the same data to every process in the communicator.It simply sends a copy of the data in message on the process with rank root to each process in the communicator comm.. It should be called by all the processes in the communicator with the same argument for root and comm. The syntax of MPI_Bcast is

```
int MPI_Bcast(
        void*          message,
        int            count,
        MPI_Datatype   datatype,
        int            root,
        MPI_Comm       comm.)
```

4- MPI_Comm_size: It gives the number of processes that are involved in the execution of a program, it can call

```
int MPI_Comm_size(MPI_Comm comm, int* number_of_processes)
```

5- MPI_Cart_create: It creates a new communicator, cart_comm., by caching a Cartesian topology with old_comm. Information on the structure of the Cartesian topology is contained in the parameters number_of_dims, dim_sizes, and warp_around. The first of these, number_of_dims, contains the number of dimensions in the Cartesian coordinates system. The next two, dim_size and wrap_around, are arrays with the order equal to number_of_dims. The array dim_size specifies the order of each dimension, and wrap_around specifies whether each dimension is circular, wrap_around[i]= 1, or linear, wrap_around[i]= 0. The processes in cart_comm are linked in row-major order. That is, the first row cosists of processes 0,1,…,dim_size[0]-1; the second row consists of processes dim_size[0], dim_size[0]+1, …, 2*dim_size[0]-1; etc. The syntax of MPI_Cart_create is

```
int MPI_Cart_create(
                MPI_Comm        old_comm,
```

```
                    int             number_of_dim,
                    int             dim_size[],
                    int             wrap_around[],
                    int             reorder,
                    MPI_Comm*       cart_comm)
```

6- MPI_Cart_coords: It returns the coordinates of the process with rank in the Cartesian communicator comm. The syntax as following

```
int  MPI_Cart_coords(
                    MPI_Comm        comm,
                    int             rank,
                    int             number_of_dims,
                    int             coordinates[])
```

7- MPI_Cart_rank: It returns the rank in the Cartesian communicator comm of the process with Cartesian coordinates. So coordinates is an array with order equal to the number of dimensions in the Cartesian topology associated with comm. The syntax is

```
int MPI_Cart_rank(
                    MPI_Comm        comm,
                    int             coordinates[],
                    int*            rank)
```

8- MPI_Cart_sub: The call to MPI_Cart_sub creates q new communicators. The free_cords parameter is an array of Boolean. It specifies whether each dimension belong to the new communicator consists of the processes obtained by fixing the row coordinate and letting the column coordinate vary; i.e., the row coordinate is fixed and the column coordinate is free. MPI_Cart_sub can only be used with a communicator that has an associated Cartesian topology, and the new communicators can only be created by fixing one or more dimensions of the old communicators and letting the other dimensions vary. The syntax is

```
int MPI_Cart_sub(
                    MPI_Comm        cart_comm,
                    int             free_coords[],
                    MPI_Comm*       new_comm)
```

9- MPI_Sendrecv_replace: It performs both the send and the receive required for the circular shift of local_B: it sends the current copy of local_B to the process in col_comm with rank dest, and then receives the copy of local_B residing on the process in col_comm with rank source. It sends the contents of buffer to the process in comm with rank dest and receives in buffer data sent from the process with rank source. The send uses the tag send_tag, and the receive uses the tag recv_tag. The processes involved in the send and receive do not have to be distinct. Its syntax is

```
int MPI_Sendrecv_replace(
                    void*           buffer,
                    int             count,
                    MPI_Datatype    datatype,
                    int             dest,
                    int             send_tag,
```

```
                    int             source,
                    int             recv_tag,
                    MPI_Comm        comm,
                    MPI_Status*     status)
```

10- MPI_Send: The contents of the message are stored in a block of memory referenced by the parameter message. The next two parameters, count, and datatype, allow the system to determine how much storage is needed for the message: the message contains a sequence of count values, each having MPI type datatype. The parameter source is the rank of the receiving process. The tag is an int. The exact syntax is

```
int MPI_Send(
                    void*           message,
                    int             count,
                    MPI_Datatype    datatype,
                    Int             dest,
                    Int             tag,
                    MPI_Comm        comm.)
```

11- MPI_Recv: The syntax for this function is:

```
int MPI_Recv(
                    void*           message,
                    int             count,
                    MPI_Datatype    datatype,
                    int             source,
                    int             tag,
                    MPI_Comm        comm.,
                    MPI_status*     status)
```

The first six parameters are the same that we described with MPI_Send, but the last one status, returns information on the data that was actually received.

12- MPI_Type_contiguous: It is constructor that builds a derived type whose elements are contiguous entries in an array. In MPI_Type_contiguous, one simply specifies that the derived type new_mpi_t will consist of count contiguous elements, each of which has type old_type. The syntax is

```
int MPI_Type_contiguous(
                    int             count,
                    MPI_Datatype    old_type,
                    MPI_Datatype*   new_mpi_t)
```

13- MPI_Address: In order to compute addresses, we use MPI_Address function. It returns the byte address of location in address, and the syntax as following

```
MPI_Address(
                    void*           location,
                    MPI_Aint        address)
```

14- MPI_Type_struct: This function has five parameters. The first parameter count is the number of blocks of elements in the derived type. It is also the size of the three arrays, block_lengths, displacements, and typelist. The array block_lengths contains the number of entries in each element of the type. So if an element of the type is an array of m values, then the corresponding entry in block_length is m.

The array displacements contains the displacement of each element from the beginning of the message, and the typelist contains the MPI datatype of each entry. The parameter new_mpi_t returns a pointer to the MPI datatype created by the call to MPI_Type_struct. Its syntax is

```
int MPI_Type_struct(
                        int             count,
                        int             block_lengths[],
                        MPI_Aint        displacements[],
                        MPI_Datatype    typelist[],
                        MPI_Datatype*   new_mpi_t)
```

15- MPI_Type_commit: After the call to MPI_Type_struct, we cannot use new_mpi_t in communication functions until we call MPI_Type_commit. This is a mechanism for the system to make internal changes in the representaion of new_mpi_t that may improve the communication performance. Its syntax is simply

```
int MPI_Type_commit(
                        MPI_Datatype*       new_mpi_t)
```

**Program's Function**:

1- Setup_grid: This function creates the various communicators and associated information. Notice that since each of our communicators has associated topology, we constructed them using the topology construction functions MPI_Cart_creat and MPI_Cart_sub rather than the more general communicator construction functions MPI_Comm_create and MPI_Comm_split. The outline for this code is

```
void Setup_grid(
          GRID_INFO_T*  grid  /* out */) {
    int old_rank;
    int dimensions[2];
    int wrap_around[2];
    int coordinates[2];

    int free_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

    /* We assume p is a perfect square */
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;

    /* We want a circular shift in second dimension. */
    /* Don't care about first                        */
    wrap_around[0] = wrap_around[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
        wrap_around, 1, &(grid->comm));
    MPI_Comm_rank(grid->comm, &(grid->my_rank));
    MPI_Cart_coords(grid->comm, grid->my_rank, 2,
        coordinates);
    grid->my_row = coordinates[0];
```

```
    grid->my_col = coordinates[1];


    /* Set up row communicators */
    free_coords[0] = 0;
    free_coords[1] = 1;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->row_comm));

    /* Set up column communicators */
    free_coords[0] = 1;
    free_coords[1] = 0;

    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->col_comm));
} /* Setup_gr
```

2- Fox: This function does the actual multiplication for each process. It uses MPI_Bcast for distributing sub-matrices (A) among other processes in the same row and MPI_Sendrecv_replace for distributing sub-matrices (B) among processes in the same column. Its code is

```
void Fox(
        int               n         /* in  */,
        GRID_INFO_T*      grid      /* in  */,
        LOCAL_MATRIX_T*   local_A   /* in  */,

        LOCAL_MATRIX_T*   local_B   /* in  */,
        LOCAL_MATRIX_T*   local_C   /* out */) {

    LOCAL_MATRIX_T*   temp_A; /* Storage for the sub-    */
                              /* matrix of A used during */
                              /* the current stage       */
    int               stage;
    int               bcast_root;
    int               n_bar;  /* n/sqrt(p)               */
    int               source;

    int               dest;
    MPI_Status        status;

    n_bar = n/grid->q;
    Set_to_zero(local_C);

    /* Calculate addresses for circular shift of B */
    source = (grid->my_row + 1) % grid->q;
    dest = (grid->my_row + grid->q - 1) % grid->q;

    /* Set aside storage for the broadcast block of A */
    temp_A = Local_matrix_allocate(n_bar);

    for (stage = 0; stage < grid->q; stage++) {
        bcast_root = (grid->my_row + stage) % grid->q;
        if (bcast_root == grid->my_col) {
            MPI_Bcast(local_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(local_A, local_B,
```

```
                local_C);
        } else {
            MPI_Bcast(temp_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(temp_A, local_B,
                local_C);
        }
        MPI_Sendrecv_replace(local_B, 1, local_matrix_mpi_t,
            dest, 0, source, 0, grid->col_comm, &status);
    } /* for */

} /* Fox */
```

3- Read_matrix: This function reads and distributes matrix among the processes. Process 0 reads a block of n_bar and sends it to the appropriate process. The outline code is

```
void Read_matrix(
         char*            prompt   /* in  */,
         LOCAL_MATRIX_T*  local_A  /* out */,
         GRID_INFO_T*     grid     /* in  */,
         int              n        /* in  */) {

    int        mat_row, mat_col;
    int        grid_row, grid_col;
    int        dest;
    int        coords[2];
    float*     temp;
    MPI_Status status;

    if (grid->my_rank == 0) {

        temp = (float*)
malloc(Order(local_A)*sizeof(float));
        printf("%s\n", prompt);
        fflush(stdout);
        for (mat_row = 0;  mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_A);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q;
grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords, &dest);
                if (dest == 0) {
                    for (mat_col = 0; mat_col <
Order(local_A); mat_col++)
                        scanf("%f",
                          (local_A-
>entries)+mat_row*Order(local_A)+mat_col);
                } else {
                    for(mat_col = 0; mat_col <
Order(local_A); mat_col++)
                        scanf("%f", temp + mat_col);
                    MPI_Send(temp, Order(local_A),
MPI_FLOAT, dest, 0,
                        grid->comm);
                }
```

```
            }
        }
        free(temp);
    } else {
        for (mat_row = 0; mat_row < Order(local_A);
mat_row++)
            MPI_Recv(&Entry(local_A, mat_row, 0),
Order(local_A),
                MPI_FLOAT, 0, 0, grid->comm, &status);
    }

}  /* Read_matrix */
```

4- Local_matrix_multiply: This function gets the appropriate A, B and C sub-matrices from the fox function and does the multiplication. Its code is

```
void Local_matrix_multiply(
         LOCAL_MATRIX_T*  local_A  /* in  */,
         LOCAL_MATRIX_T*  local_B  /* in  */,
         LOCAL_MATRIX_T*  local_C  /* out */) {
    int i, j, k;

    for (i = 0; i < Order(local_A); i++)
        for (j = 0; j < Order(local_A); j++)
            for (k = 0; k < Order(local_B); k++)
                Entry(local_C,i,j) = Entry(local_C,i,j)
                    +  Entry(local_A,i,k)*Entry(local_B,k,j);

}  /* Local_matrix_multiply */
```

5- Build_matrix_type: This function builds the matrices contiguously in the storage, get the address for each one, and the number of blocks that each one has. The outline is

```
void Build_matrix_type(
         LOCAL_MATRIX_T*  local_A  /* in */) {
    MPI_Datatype  temp_mpi_t;
    int           block_lengths[2];
    MPI_Aint      displacements[2];
    MPI_Datatype  typelist[2];
    MPI_Aint      start_address;
    MPI_Aint      address;

    MPI_Type_contiguous(Order(local_A)*Order(local_A),
        MPI_FLOAT, &temp_mpi_t);

    block_lengths[0] = block_lengths[1] = 1;

    typelist[0] = MPI_INT;
    typelist[1] = temp_mpi_t;

    MPI_Address(local_A, &start_address);
    MPI_Address(&(local_A->n_bar), &address);
    displacements[0] = address - start_address;
    MPI_Address(local_A->entries, &address);
    displacements[1] = address - start_address;
```

```
    MPI_Type_struct(2, block_lengths, displacements,
        typelist, &local_matrix_mpi_t);
    MPI_Type_commit(&local_matrix_mpi_t);
}  /* Build_matrix_type */
```

## Fox's Algorithm Execution Results

The tables below show the result of executing fox algorithm (cpu_time, mflop) with
P = 4, 9, 16, 25 36 and N = 720
M-flop over all Processes = ( 2 * n ^ 3 ) / ( 10 ^ 6 * Max (cpu time overall processes))

| P = 4 and N =720 | | | |
|---|---|---|---|
| My Rank | cpu time | M-flop for each P | M-flop overall Ps |
| 0 | 3.500000e+00 seconds | 0.296229 | 178.161336 |
| 1 | 4.170000e+00 seconds | 0.248633 | |
| 2 | 4.180000e+00 seconds | 0.248038 | |
| 3 | 4.190000e+00 seconds | 0.247446 | |
| P = 9 and N =720 | | | |
| 0 | 1.480000e+00 seconds | 0.700541 | 431.500578 |
| 1 | 1.410000e+00 seconds | 0.735319 | |
| 2 | 1.620000e+00 seconds | 0.640000 | |
| 3 | 1.390000e+00 seconds | 0.745899 | |
| 4 | 1.490000e+00 seconds | 0.695839 | |
| 5 | 1.420000e+00 seconds | 0.730141 | |
| 6 | 1.440000e+00 seconds | 0.720000 | |
| 7 | 1.420000e+00 seconds | 0.730141 | |
| 8 | 1.730000e+00 seconds | 0.599306 | |
| P = 16 and N =720 | | | |
| 0 | 8.200000e-01 seconds | 1.264390 | 746.496 |
| 1 | 9.800000e-01 seconds | 1.057959 | |
| 2 | 1.020000e+00 seconds | 1.016471 | |
| 3 | 8.400000e-01 seconds | 1.234286 | |
| 4 | 1.000000e+00 seconds | 1.036800 | |
| 5 | 9.200000e-01 seconds | 1.126957 | |
| 6 | 8.200000e-01 seconds | 1.264390 | |
| 7 | 8.200000e-01 seconds | 1.264390 | |
| 8 | 8.500000e-01 seconds | 1.016471 | |
| 9 | 8.300000e-01 seconds | 1.249157 | |
| 10 | 8.200000e-01 seconds | 1.264390 | |
| 11 | 8.500000e-01 seconds | 1.219765 | |
| 12 | 8.100000e-01 seconds | 1.280000 | |
| 13 | 8.600000e-01 seconds | 1.205581 | |
| 14 | 8.600000e-01 seconds | 1.205581 | |

| | | | |
|---|---|---|---|
| 15 | 8.400000e-01 seconds | 1.234286 | |
| P = 25 and N =720 | | | |
| 0 | 5.600000e-01 seconds | 1.851429 | |
| 1 | 6.400000e-01 seconds | 1.620000 | |
| 2 | 5.900000e-01 seconds | 1.757288 | |
| 3 | 6.300000e-01 seconds | 1.645714 | |
| 4 | 6.400000e-01 seconds | 1.620000 | |
| 5 | 6.900000e-01 seconds | 1.502609 | |
| 6 | 6.700000e-01 seconds | 1.547463 | |
| 7 | 6.600000e-01 seconds | 1.570909 | |
| 8 | 6.700000e-01 seconds | 1.547463 | |
| 9 | 6.300000e-01 seconds | 1.645714 | |
| 10 | 6.200000e-01 seconds | 1.672258 | 1081.87826 |
| 11 | 6.000000e-01 seconds | 1.728000 | |
| 12 | 5.400000e-01 seconds | 1.920000 | |
| 13 | 5.800000e-01 seconds | 1.787586 | |
| 14 | 5.900000e-01 seconds | 1.757288 | |
| 15 | 5.600000e-01 seconds | 1.851429 | |
| 16 | 6.200000e-01 seconds | 1.672258 | |
| 17 | 5.700000e-01 seconds | 1.818947 | |
| 18 | 5.700000e-01 seconds | 1.818947 | |
| 19 | 5.700000e-01 seconds | 1.818947 | |
| 20 | 5.700000e-01 seconds | 1.818947 | |
| 21 | 6.000000e-01 seconds | 1.728000 | |
| 22 | 5.900000e-01 seconds | 1.757288 | |
| 23 | 5.600000e-01 seconds | 1.851429 | |
| 24 | 6.000000e-01 seconds | 1.728000 | |
| P = 36 and N =720 | | | |
| 0 | 4.200000e-01 seconds | 2.468571 | |
| 1 | 4.400000e-01 seconds | 2.356364 | |
| 2 | 4.200000e-01 seconds | 2.468571 | |
| 3 | 4.400000e-01 seconds | 2.356364 | |
| 4 | 4.300000e-01 seconds | 2.411163 | |
| 5 | 4.700000e-01 seconds | 2.205957 | |
| 6 | 4.100000e-01 seconds | 2.528780 | |
| 7 | 4.100000e-01 seconds | 2.528780 | |
| 8 | 4.200000e-01 seconds | 2.468571 | 1588.289361 |
| 9 | 4.400000e-01 seconds | 2.356364 | |
| 10 | 4.300000e-01 seconds | 2.411163 | |
| 11 | 4.500000e-01 seconds | 2.304000 | |
| 12 | 4.000000e-01 seconds | 2.592000 | |
| 13 | 4.000000e-01 seconds | 2.592000 | |
| 14 | 4.000000e-01 seconds | 2.592000 | |
| 15 | 4.000000e-01 seconds | 2.592000 | |
| 16 | 4.200000e-01 seconds | 2.468571 | |

| | | |
|---|---|---|
| 17 | 3.900000e-01 seconds | 2.658462 |
| 18 | 4.000000e-01 seconds | 2.592000 |
| 19 | 4.000000e-01 seconds | 2.592000 |
| 20 | 4.100000e-01 seconds | 2.528780 |
| 21 | 3.800000e-01 seconds | 2.728421 |
| 22 | 3.900000e-01 seconds | 2.658462 |
| 23 | 4.000000e-01 seconds | 2.592000 |
| 24 | 4.100000e-01 seconds | 2.528780 |
| 25 | 4.000000e-01 seconds | 2.592000 |
| 26 | 4.100000e-01 seconds | 2.528780 |
| 27 | 3.600000e-01 seconds | 2.880000 |
| 28 | 4.000000e-01 seconds | 2.592000 |
| 29 | 4.000000e-01 seconds | 2.592000 |
| 30 | 4.200000e-01 seconds | 2.468571 |
| 31 | 4.000000e-01 seconds | 2.592000 |
| 32 | 4.100000e-01 seconds | 2.528780 |
| 33 | 3.900000e-01 seconds | 2.658462 |
| 34 | 4.100000e-01 seconds | 2.528780 |
| 35 | 4.000000e-01 seconds | 2.592000 |

## ScaLAPACK Matrix-Matrix Multiplication PBLAS routine (PDGEMM)

ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing computers and networks of workstations supporting PVM and MPI. The library is currently written in Fortran 77 in a Single Program Multiple Data (SPMD) style using explicit message passing for interprocessor communication. The ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are distributed-memory versions of the Level 1, Level 2, and Level 3 BLAS, called the Parallel BLAS or PBLAS, and a set of Basic Linear Algebra Communication subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, the majority of interprocessor communication occurs within the PBLAS. ScaLAPACK contains driver routines for solving standard types of problems, computational routines to perform a distinct computational task, and auxiliary routines to perform a certain subtask or common low-level computation. Each driver routine typically calls a sequence of computation routines. Four basics steps are required to call a ScaLAPACK routine:

I- Initialize the process grid
II- Distribute the matrix on the process grid
III- Call ScaLAPACK routine
IV- Release the process grid

Every ScaLAPACK routine call is referenced to PBLAS.dat that has some parameters, which are needed, in order to accomplish the function. The parameters for this file are described below

M : The number of rows in the matrices A and C.
N : The number of columns in the matrices B and C.
K : The number of rows of B and the number of columns of A.
NB :The size of the square blocks the matrices A, B and C are split into.
P :The number of process rows.
Q :he number of process columns.

In our case (Square Matrix-Matrix Multiplication), M, N, and K are the same values.The tables below show the result of executing PBLAS routine (cpu_time, mflop) with
P = 4, 9, 16, 25 36 and N = 720
M-flop over all Processes = ( 2 * n ^ 3 ) / ( 10 ^ 6 * Max (cpu time overall processes))

| P = 4, N =720, and NB = 360 | | | |
|---|---|---|---|
| My Rank | cpu time | M-flop for each P | M-flop overall P's |
| 0 | 0.3800000 | 2.728421 | 1964.463157 |
| 1 | 0.3600000 | 2.880000 | |
| 2 | 0.3500000 | 2.962286 | |
| 3 | 0.3800000 | 2.728421 | |
| P = 9, N =720, and NB = 240 | | | |
| 0 | 0.1800000 | 5.760000 | 3393.6363 |
| 1 | 0.2100000 | 4.937143 | |
| 2 | 0.1700000 | 6.098824 | |
| 3 | 0.2000000 | 5.184000 | |
| 4 | 0.1800000 | 5.760000 | |
| 5 | 0.1900000 | 5.456842 | |
| 6 | 0.2000000 | 5.184000 | |
| 7 | 0.2000000 | 5.184000 | |
| 8 | 0.2200000 | 4.712727 | |
| P = 16, N =720, and NB = 180 | | | |
| 0 | 0.1000000 | 10.36800 | 4147.2 |
| 1 | 0.1400000 | 7.405715 | |
| 2 | 0.1300000 | 7.975385 | |
| 3 | 0.1600000 | 6.480000 | |
| 4 | 0.1300000 | 7.975385 | |
| 5 | 0.1200000 | 8.639999 | |
| 6 | 9.9999994E-02 | 10.36800 | |
| 7 | 0.1300000 | 7.975384 | |
| 8 | 0.1800000 | 5.760000 | |
| 9 | 0.1500000 | 6.912000 | |
| 10 | 0.1300000 | 7.975384 | |
| 11 | 0.1100000 | 9.425454 | |
| 12 | 0.1200000 | 8.639999 | |
| 13 | 0.1300000 | 7.975385 | |
| 14 | 0.1400000 | 7.405715 | |

| | | | |
|---|---|---|---|
| 15 | 0.1200000 | 8.640000 | |
| P = 5, N =720, and NB = 144 | | | |
| 0 | 9.0000004E-02 | 11.52000 | |
| 1 | 6.9999993E-02 | 14.81143 | |
| 2 | 9.0000004E-02 | 11.52000 | |
| 3 | 6.9999993E-02 | 14.81143 | |
| 4 | 0.1000000 | 10.36800 | |
| 5 | 9.0000004E-02 | 11.52000 | |
| 6 | 9.0000004E-02 | 11.52000 | |
| 7 | 5.0000012E-02 | 20.73599 | |
| 8 | 0.1000000 | 10.36800 | |
| 9 | 9.9999994E-02 | 10.36800 | |
| 10 | 0.1300000 | 7.975385 | |
| 11 | 6.9999993E-02 | 14.81143 | |
| 12 | 9.0000004E-02 | 11.52000 | 5742.27692 |
| 13 | 8.9999974E-02 | 11.52000 | |
| 14 | 6.9999993E-02 | 14.81143 | |
| 15 | 6.9999993E-02 | 14.81143 | |
| 16 | 9.9999994E-02 | 10.36800 | |
| 17 | 0.1100000 | 9.425454 | |
| 18 | 7.0000008E-02 | 14.81143 | |
| 19 | 8.9999989E-02 | 11.52000 | |
| 20 | 7.9999998E-02 | 12.96000 | |
| 21 | 0.1100000 | 9.425454 | |
| 22 | 0.1200000 | 8.640000 | |
| 23 | 8.9999996E-02 | 11.52000 | |
| 24 | 9.9999994E-02 | 10.36800 | |
| P = 36, N =720, and NB = 120 | | | |
| 0 | 7.0000000E-02 | 14.81143 | |
| 1 | 0.2700000 | 3.839999 | |
| 2 | 6.9999993E-02 | 14.81143 | |
| 3 | 0.1000000 | 10.36800 | |
| 4 | 9.0000033E-02 | 11.52000 | |
| 5 | 8.9999974E-02 | 11.52000 | |
| 6 | 6.0000002E-02 | 17.28000 | |
| 7 | 8.9999974E-02 | 11.52000 | |
| 8 | 6.0000002E-02 | 17.28000 | |
| 9 | 7.0000023E-02 | 14.81142 | 7464.96 |
| 10 | 5.9999943E-02 | 17.28002 | |
| 11 | 6.9999993E-02 | 14.81143 | |
| 12 | 6.9999993E-02 | 14.81143 | |
| 13 | 6.0000032E-02 | 17.27999 | |
| 14 | 9.9999964E-02 | 10.36800 | |
| 15 | 0.1000000 | 10.36800 | |
| 16 | 6.0000002E-02 | 17.28000 | |

| | | |
|---|---|---|
| 17 | 5.0000012E-02 | 20.73599 |
| 18 | 9.0000004E-02 | 11.52000 |
| 19 | 7.9999983E-02 | 12.96000 |
| 20 | 5.0000012E-02 | 20.73599 |
| 21 | 4.9999982E-02 | 20.73601 |
| 22 | 6.9999993E-02 | 14.81143 |
| 23 | 8.0000013E-02 | 12.96000 |
| 24 | 0.1000000 | 10.36800 |
| 25 | 6.0000002E-02 | 17.28000 |
| 26 | 7.0000008E-02 | 14.81143 |
| 27 | 6.9999993E-02 | 14.81143 |
| 28 | 6.9999993E-02 | 14.81143 |
| 29 | 7.9999998E-02 | 12.96000 |
| 30 | 3.0000001E-02 | 34.56000 |
| 31 | 4.9999997E-02 | 20.73600 |
| 32 | 5.9999987E-02 | 17.28000 |
| 33 | 7.9999998E-02 | 12.96000 |
| 34 | 6.9999993E-02 | 14.81143 |
| 35 | 6.9999993E-02 | 14.81143 |

## Comparing Fox's Algorithm With PBLAS Routine (PDGEMM)

The table below shows the speedup table for both Fox's Algorithm and PBLAS Routine, where n = 720

| Fox | | | | | PBLAS | | | |
|---|---|---|---|---|---|---|---|---|
| P | Time on Process 0 | Speed up | M-flop on Process 0 | | P | Time on Process 0 | Speed up | M-flop on Process 0 |
| **4** | **3.500000e+00** | **-** | **0.296229** | | **4** | **0.3800000** | **-** | **2.728421** |
| **9** | **1.480000e+00** | **2.3648** | **0.700541** | | **9** | **0.1800000** | **2.11** | **5.760000** |
| **16** | **8.200000e-01** | **4.2683** | **1.264390** | | **16** | **0.1000000** | **3.80** | **10.36800** |
| **25** | **5.600000e-01** | **6.25** | **1.851429** | | **25** | **9.0000004E-02** | **4.22** | **11.52000** |
| **36** | **4.200000e-01** | **8.33** | **2.468571** | | **36** | **7.0000000e-02** | **5.42857** | **14.81143** |

The speedup table using the M-flop rate overall processes:

M-flop over all Processes = ( 2 * n ^ 3 ) / ( 10 ^ 6 * Max (cpu time overall processes))

| Fox | | | | | PBLAS | | | |
|---|---|---|---|---|---|---|---|---|
| P | Largest cpu time | Speed up | M-flop overall Ps | | P | Largest cpu time | Speed up | M-flop overall Ps |
| **4** | **4.180000** | **-** | **178.16133** | | **4** | **0.380000** | **-** | **1964.4631** |
| **9** | **1.730000** | **2.416** | **431.50057** | | **9** | **0.220000** | **1.727** | **3393.6363** |
| **16** | **1.020000** | **4.098** | **746.496** | | **16** | **0.180000** | **2.111** | **4147.2** |
| **25** | **0.690000** | **6.057** | **1081.8782** | | **25** | **0.130000** | **2.923** | **5742.2769** |
| **36** | **0.470000** | **8.893** | **1588.2893** | | **36** | **0.100000** | **3.800** | **7464.96** |

From the above tables we can result that PBLAS Routine (PDGEMM) performance is more efficient than Fox's Algorithm.

## Conclusion

I- The number of processes using Fox's Algorithm must be a perfect square.

II- The number of processes using Fox's Algorithm depends on the order of n; i.e., P = K processes, then N must be chosen, so the N/sqrt(P) is integer.

III- We can rewrite the Fox's Algorithm to use Bcast data between columns instead of using send and receive as following

```
/* my process  row = i ,  my process  column = j */
     q = sqrt(p);
    dest = ((i – 1)  mod  q , j);
    source = ((i + 1)  mod  q, j);
    for (stage = 0  ; stage < q ; stage++)
    {
        k_bar = (i + stage)  mod  q;
        1.  Broadcast A[i,k_bar] across process row i;
        2.  Broadcast B[k_bar,j] across process column j;
        3.  C[i,j] = C[i,j] + A[i,k_bar]*B[k_bar,j];
    }
```

Let's take example-3 to run it on this algorithm to show how it works.

| Process(0,0) | Process(0,1) |
|---|---|
| i = j = 0<br>Assign $A_{00,}B_{00},C_{00}$ to this process(0,0)<br>q = 2<br>dest = (1,0)<br>source = (1,0)<br>stage = 0<br>k_bar = 0<br>Broadcast $A_{00}$ across process row i = 0<br>Broadcast $B_{00}$ across process column j =0<br>$C_{00} = C_{00} + A_{00} * B_{00}$<br>stage = 1<br>k_bar = 1<br>Broadcast $A_{01}$ across process row i = 0<br>Broadcast $B_{10}$ across process column j = 0<br>$C_{00} = C_{00} + A_{01} * B_{10}$ | i = 0, j = 1<br>assign $A_{01,}B_{01},C_{01}$ to this process(0,1)<br>q = 2<br>dest= (1,1)<br>source= (0,1)<br>stage = 0<br>k_bar = 0<br>Broadcast $A_{00}$ across process row i = 0<br>Broadcast $B_{01}$ across process column j =1<br>$C_{01} = C_{01} + A_{00} * B_{01}$<br>stage = 1<br>k_bar = 1<br>Broadcast $A_{01}$ across process row i=0<br>Broadcast $B_{11}$ across process column j = 1<br>$C_{01} = C_{01} + A_{01} * B_{11}$ |
| Process(1,0) | Process(1,1) |
| i = 1, j = 0<br>Assign $A_{10,}B_{10},C_{10}$ to this process(1,0)<br>q = 2<br>dest = (0,0)<br>source = (0,0)<br>stage = 0<br>k_bar = 1<br>Broadcast $A_{11}$ across process row i = 1<br>Broadcast $B_{10}$ across process column i =1<br>$C_{10} = C_{10} + A_{11} * B_{10}$<br>stage = 1<br>k_bar = 0<br>Broadcast $A_{10}$ across process row i = 1<br>Broadcast $B_{00}$ across process column i = 1<br>$C_{10} = C_{10} + A_{10} * B_{00}$ | i = 1, j = 1<br>Assign $A_{11,}B_{11},C_{11}$ to this process(1,1)<br>q = 2<br>dest = (0,1)<br>source = (0,1)<br>stage = 0<br>k_bar = 1<br>Broadcast $A_{11}$ across process row i = 1<br>Broadcast $B_{11}$ across process column i =1<br>$C_{11} = C_{11} + A_{11} * B_{11}$<br>stage = 1<br>k_bar = 0<br>Broadcast $A_{10}$ across process row i = 1<br>Broadcast $B_{01}$ across process column i = 1<br>$C_{11} = C_{11} + A_{10} * B_{01}$ |

IV. Applying ScaLAPACK PBLAS Routine (PDGEMM) for Matrix-Matrix Multiplication is much faster than using Fox's Algorithm.