

Advanced MPI

(with MPI function summary)

- **MPI_Send**

- Returns only after application buffer in sending task is free for reuse
- MPI standard permits use of system buffer but it's not mandatory

- **MPI_Recv**

- Blocks until requested data is available in application buffer in receiving task

- **MPI_Ssend**

- Blocks until application buffer in sending task is free for reuse and destination process has started to receive the message

- **MPI_Bsend**

- **Blocks until data from application buffer (in program) is copied to user allocated send *message buffer***
 - *Caters to problems associated with insufficient system buffer space*
- **Must be used with the MPI_Buffer_attach routine**

- **MPI_Buffer_attach**

- MPI_Buffer_detach**

- **Used to allocate/deallocate *message buffer* space to be used by the MPI_Bsend routine**
- **Only one buffer can be attached to a process at a time**

- **MPI_Sendrecv**

- **Sends a message and posts a receive before blocking**
- **Blocks until sending application buffer is free for reuse and until receiving application buffer contains the received message**

- **MPI_Probe**

- **Results in blocking wait until a message is available for receipt**
- **Wildcards `MPI_ANY_SOURCE` and `MPI_ANY_TAG` may be used to test for a message from any source or with any tag**
 - **Actual source and tag will be returned in C status structure as `status.MPI_SOURCE` and `status.MPI_TAG`**

- **MPI_Isend**

- **Processing continues immediately without waiting for the message to be copied out from the application buffer**
 - Request handle is returned to check the pending message status
- **Program should not modify application buffer until subsequent calls to MPI_Wait/MPI_Test indicate completion of non-blocking send**

- **MPI_Irecv**

- **Processing continues immediately without waiting for the message to be received and copied into the application buffer**
 - Request handle is returned to check the pending message status
- **Program must use MPI_Wait/MPI_Test to check completion of non-blocking receive operation**
 - So that requested message is available in application buffer

- **MPI_Issend**

- **Non-blocking synchronous send**
- **Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when destination process has received the message**

- **MPI_IbSEND**

- **Non-blocking buffered send; should be used with MPI_Buffer_attach routine**
- **Similar to MPI_Bsend(), except MPI_Wait() or MPI_Test() indicates when destination process has received the message**

- **MPI_Test, MPI_Testany, MPI_Testall, MPI_Testsome**
 - MPI_Test checks status of specified non-blocking send or receive operation (indicated by flag)
 - For multiple non-blocking operations, the programmer can specify any, all, or some completions

- **MPI_Wait, MPI_Waitany, MPI_Waitall, MPI_Waitsome**
 - MPI_Wait blocks until specified non-blocking send/receive operation has completed
 - For multiple non-blocking operations, the programmer can specify any, all, or some completions

- **MPI_Iprobe**
 - Performs non-blocking test for a message (indicated by flag)
 - Wildcards MPI_ANY_SOURCE and MPI_ANY_TAG may be used to test for a message from any source or with any tag
 - Actual source and tag will be returned in C status structure as status.MPI_SOURCE and status.MPI_TAG

- `int MPI_Waitsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[]);`

- **Parameters**
 - *incount*: length of *array_of_requests*
 - *array_of_requests*: array of handles
 - *outcount*: no. of completed requests
 - *array_of_indices*: array of indices of completed operations
 - *array_of_statuses*: array of status objects for completed operations

- **Remarks**
 - Waits until at least one of the operations associated with active handles in the list has completed
 - Returns in *outcount* the no. of requests that have completed
 - Returns in first *outcount* locations of *array_of_indices* the indices of completed operations

- `int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status);`

- **Parameters**
 - *count*: list length
 - *array_of_requests*: array of handles
 - *index*: index of handle for operation that completed
 - *status*: status object

- **Remarks**
 - Blocks until one of the operations associated with the active requests in the array has completed
 - Returns in *index* the index of that request in the array and returns in *status* the status of the completing communication
 - If *array_of_requests* contains no active handles then call returns immediately with *index* = `MPI_UNDEFINED`, and a empty status

- `int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[]);`
- **Parameters**
 - *incount*: length of *array_of_requests* (integer)
 - *array_of_requests*: array of handles
 - *outcount*: no. of completed requests (integer)
 - *array_of_indices*: indices of completed operations
 - *array_of_statuses*: array of status objects
- **Remarks**
 - Behaves like `MPI_WAITSSOME`, except that it returns immediately
 - If no operation has completed it returns `outcount = 0`
 - If there is no active handle in the list it returns `outcount = MPI_UNDEFINED`

```
• int MPI_Testany(int count, MPI_Request array_of_requests[],
  int *index, int *flag, MPI_Status *status );
```

• Parameters

- *count*: list length
- *array_of_requests*: Array of handles
- *index*: Index of completed operation that completed
- *flag*: true if one of the operations is complete
- *status*: status object

• Remarks

- If array of requests has no active handles, call returns immediately with *flag=true*, *index=MPI_UNDEFINED*, *status=empty*
- Otherwise, MPI_TESTANY performs MPI_TEST on each request handle in arbitrary order, until one call returns *flag=true*, or all fail
 - *index* is set to the last value of *i*

- `MPI_Test(*request, *flag, *status)`
- `MPI_Testany(count, *array_of_requests, *index, *flag, *status)`
- `MPI_Testall(count, *array_of_requests, *flag, *array_of_statuses)`
- `MPI_Testsome (incount, *array_of_requests, *outcount, *array_of_offsets, *array_of_statuses)`

- `MPI_Wait (*request, *status)`
- `MPI_Waitany(count, *array_of_requests, *index, *status)`
- `MPI_Waitall(count, *array_of_requests, *array_of_statuses)`
- `MPI_Waitsome(incount, *array_of_requests, *outcount, *array_of_offsets, *array_of_statuses)`

```

#include "mpi.h"
#include <stdio.h>
int main(argc,argv)
int argc; char *argv[];
{
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI_Request reqs[4];
    MPI_Status stats[4];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    prev = rank-1;
    next = rank+1;
    if (rank==0)prev = numtasks - 1;
    if (rank==(numtasks-1))next = 0;

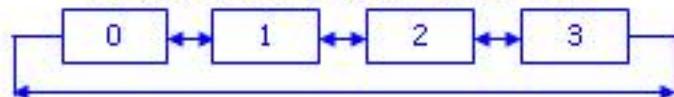
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

    MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    MPI_Waitall(4, reqs, stats);
    MPI_Finalize();
}

```

Each process sends message to two of its neighbors
and also receives the same from them



- **A mechanism for naming/arranging processes in a communicator that best reflects the actual communication pattern**
 - **Makes subsequent code simpler**
 - **Highlights main communication patterns in a communicator**
 - **May enable run-time system to optimize communication**
 - **Computation and set of interacting processes are naturally identified by coordinates in the topology**

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Row-major
mapping

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Column-major
mapping

0	3	4	5
1	2	7	6
14	13	8	9
15	12	11	10

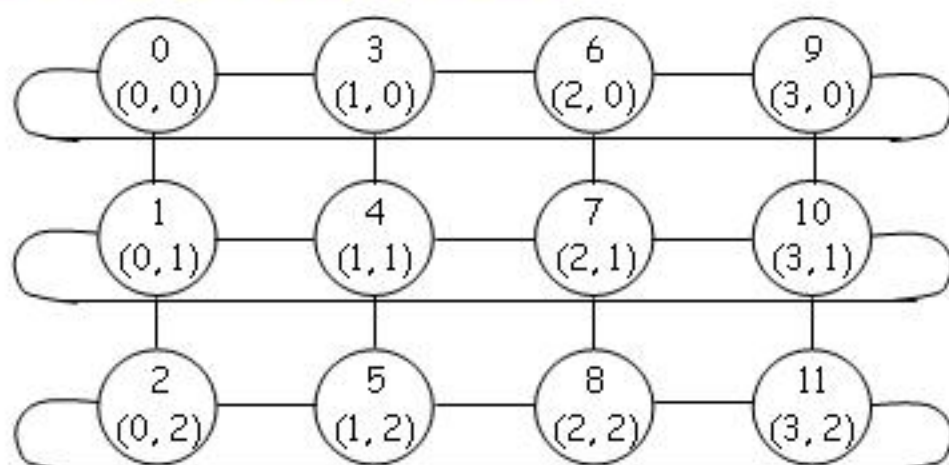
Space-filling curve
mapping

0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

Hpyercube
mapping

Different ways to map processes to two-dimensional grid

- **Access to convenient routines**
 - **Compute rank of any process given its coordinates in the grid**
 - Taking proper account of boundary conditions i.e. returning `MPI_NULL_PROC` if you go outside the grid
 - **Routines to compute the ranks of nearest neighbors**
 - The rank can then be used as an argument to `MPI_SEND`, `MPI_RECV`, `MPI_SENDRECV` etc.



es. The lines denote main communication fashion similar to a two-dimensional grid

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                   int *periods, int reorder, MPI_Comm *comm_cart)
```

- **The parameters**

- **comm_cart** identifies new virtual topology that is created
- **ndims** specifies number of dimensions of the topology
- **dims** specify size along each dimensions
- **periods** specify presence of wraparound connections
 - **True value means the dimension has a cyclic boundary**
- **reorder** specifies reordering criteria
 - **False value means rank of each process in the new group is identical to its rank in the old group**

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int
rank)
```

- Takes coordinates of the process and returns its rank

```
int MPI_Cart_coord(MPI_Comm comm_cart, int rank, int
maxdims, int *coords)
```

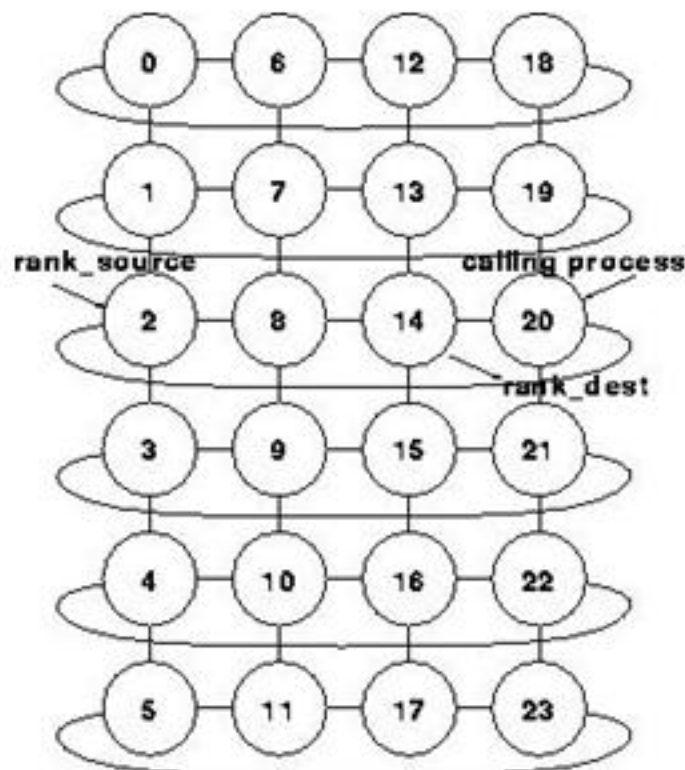
- Takes rank of the process and its Cartesian coordinates

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir,
  int s_step, int *rank_source, int *rank_dest)
```

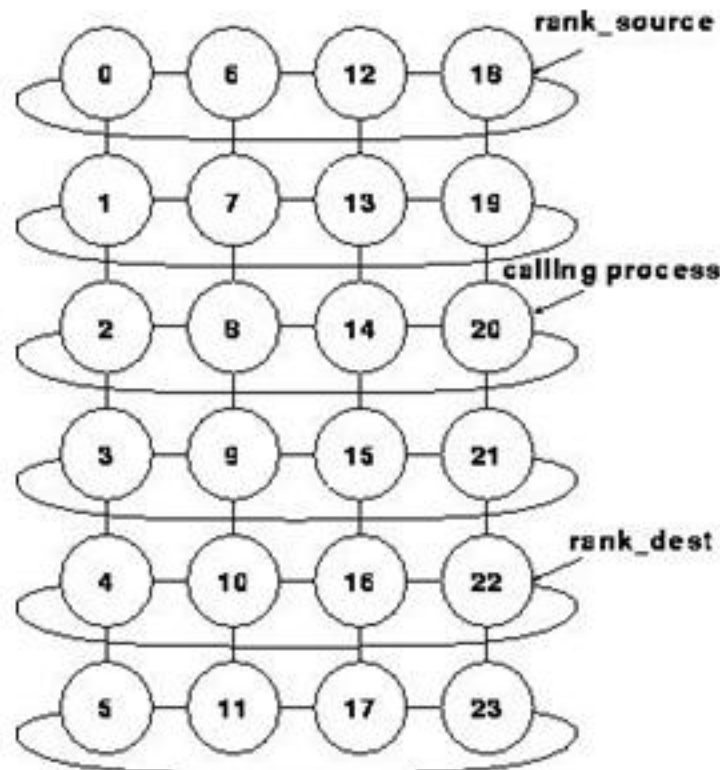
- **Computes rank of the source and destination processes**
- **dir (value 0 to dims-1) specifies direction of the shift**
- **Displacement s_step specifies number of process coordinates in direction of shift (a +ve or -ve number)**

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir,
                  int s_step, int *rank_source, int *rank_dest)
```

- **Source and destination ranks returned in calling process**
 - **Along rows (dir=0)**
 - **displacement of +1 (s_step=1), source is rank left to calling rank and destination is rank to the right**
 - **displacement of -1 (s_step=-1), source is rank right to calling rank and destination is rank to the left**
 - **Along columns (dir=1)**
 - **displacement of +1 (s_step=1), source is rank above calling rank and destination is rank below it**
 - **displacement of -2 (s_step=-2), source is two ranks below calling rank and destination is two ranks above it**



Called on process 20 with dir=0 and s_step = -1



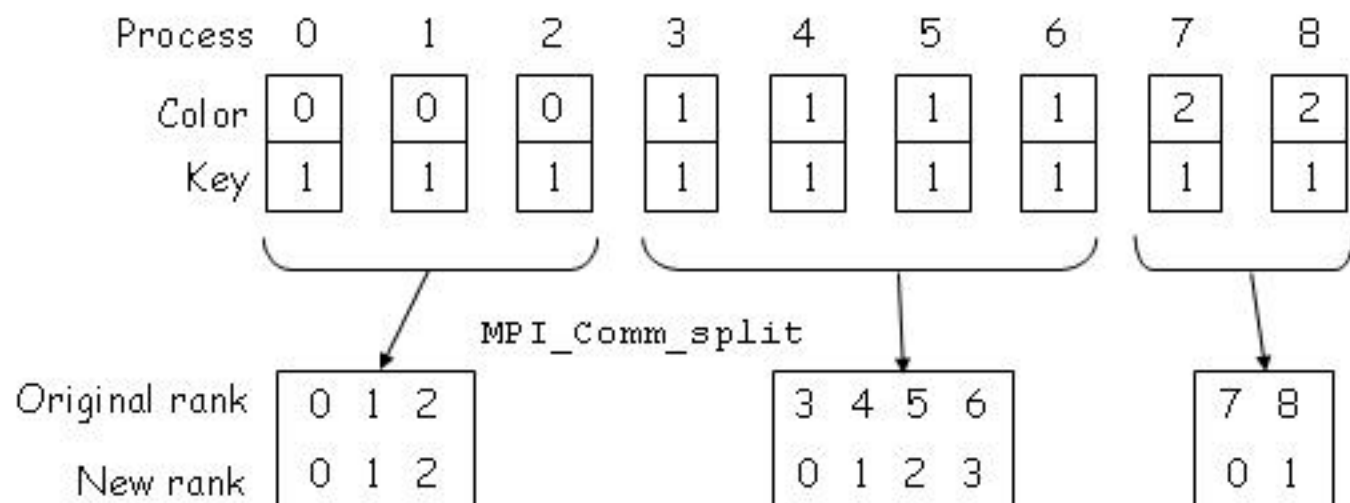
Called on process 20 with dir=1 and s_step = 2

- Communication operations may sometimes need to be restricted to certain subset of processes
- A MPI collective function to partition a group into subgroups is

```
int MPI_Comm_split(MPI_Comm comm, int color,
                  int key, MPI_Comm *newcomm)
```

- Each subgroup comprises processes having same `color`
- The processes in each subgroup are ranked by `key` parameter, with ties broken according to their rank in old communicator `comm`
- New communicator for each subgroup is returned in `newcomm`

- If each process calls `MPI_Comm_split` using values of `color` and `key` as shown below, three communicators will be created

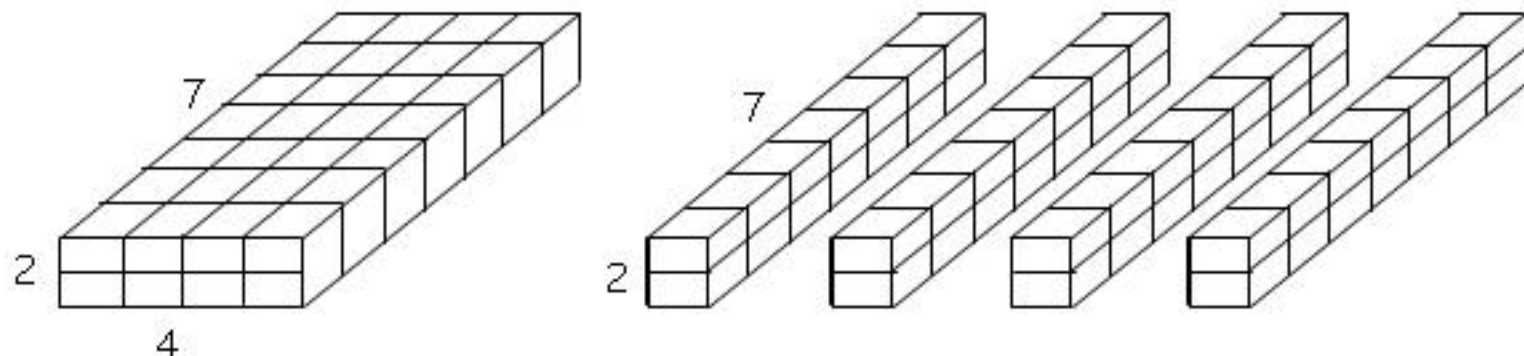


- Sometimes communications in virtual grid need to be performed only on subset of the grid (rows or columns)
- MPI has `MPI_Cart_sub` function to create new communicators for sub-grids or "slices" of a grid

```
int MPI_Cart_sub(MPI_Comm comm, int *keep_dims,
                MPI_Comm *comm_subcart)
```

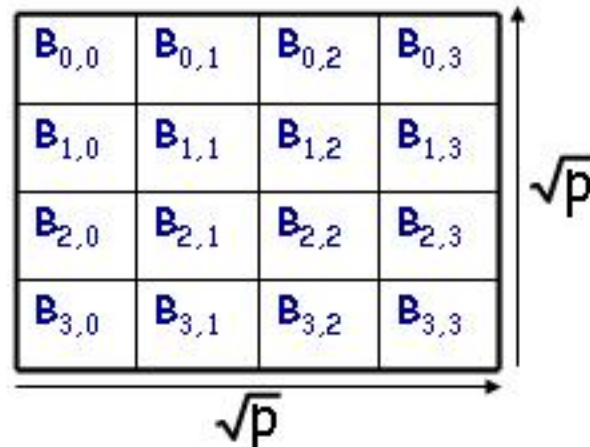
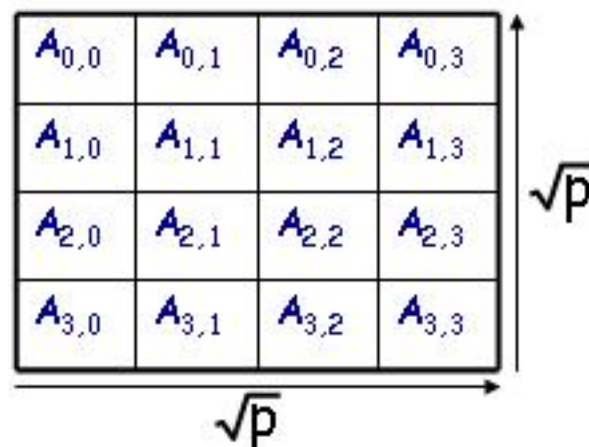
- **Array `keep_dims` specifies how the topology is partitioned**
 - **If `keep_dims[i]` is true, then i^{th} dimension is retained in new sub-topology, else it is split**
- **Only one communicator containing the calling process is returned**

- If `comm` defines a $2 \times 4 \times 7$ grid, and `keep_dims = (true, false, true)`, then `MPI_Cart_sub(...)` will create four two-dimensional sub-topologies of size 2×7 as shown below

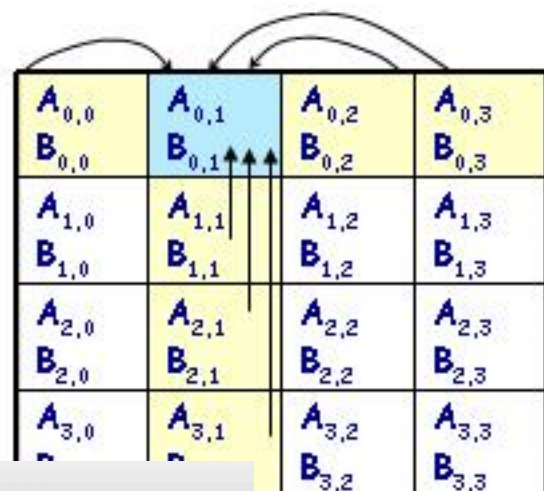


- If `keep_dims = (false, false, true)`, then `MPI_Cart_sub(...)` will create eight one-dimensional sub-topologies of size 7

- Consider two $n \times n$ matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ ($0 \leq i, j < \sqrt{p}$) of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each
 - The blocks are mapped onto a mesh of $\sqrt{p} \times \sqrt{p}$ processes

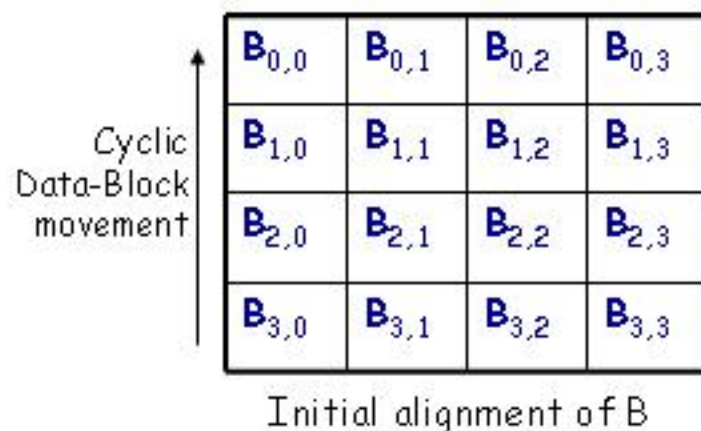
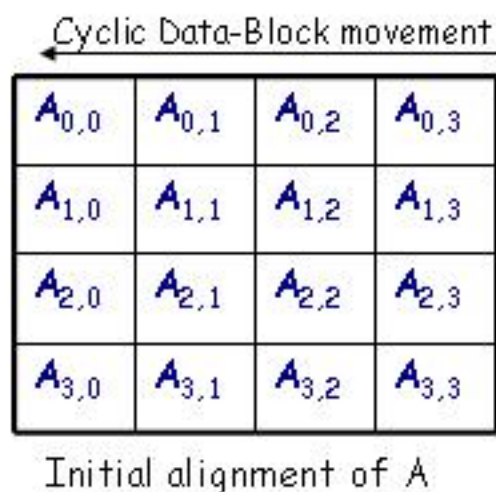


- Each process computes $C_{i,j}$, which requires $A_{i,k}$ and $B_{k,j}$ ($0 \leq k < \sqrt{p}$)
- To acquire all the required blocks
 - An All-to-All broadcast of matrix A's blocks is performed in each row of processes
 - Similarly, an All-to-All broadcast of matrix B's blocks is performed in each column
- After acquiring the required blocks, $P_{i,j}$ performs the sub-matrix multiplication and addition to produce $C_{i,j}$, for all i, j

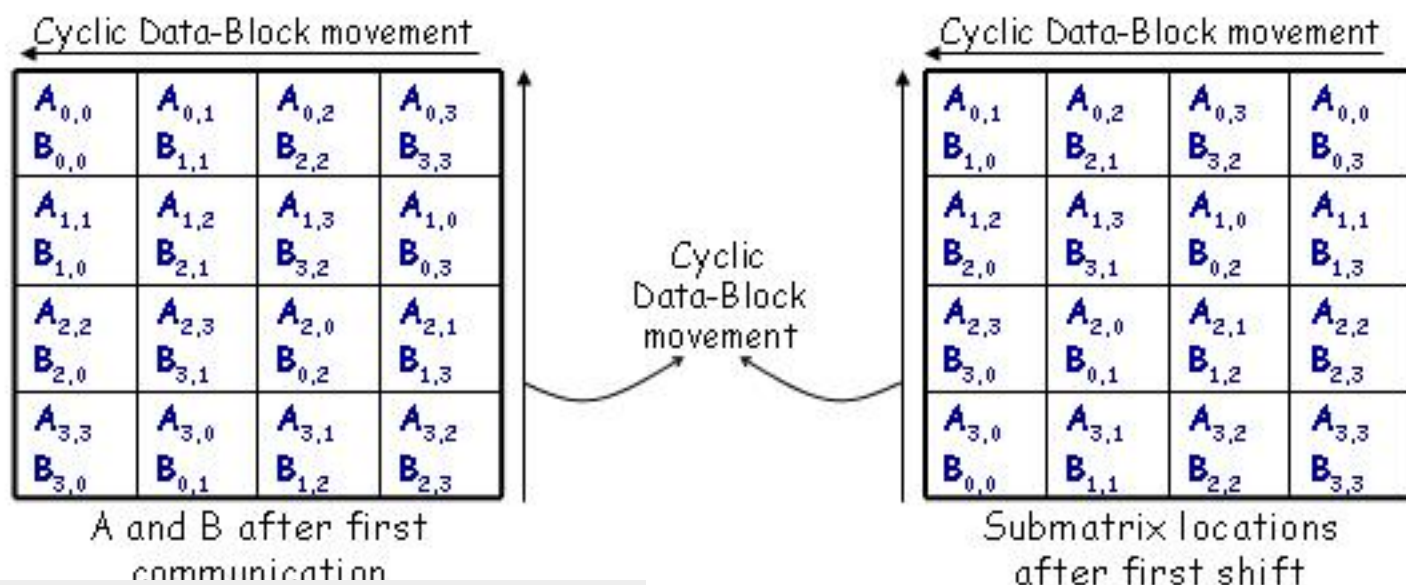


Note: It's a memory inefficient algorithm

- A memory-efficient version of the previous algorithm
- Partitioning of the matrices A and B into p square blocks is same as in previous algorithm
- However, blocks get rotated among processes after every sub-matrix multiplication, both along rows & columns



- First communication step aligns submatrices by shifting $A_{i,j}$ by i steps and $B_{i,j}$ by j steps (left figure)
- After a submatrix multiplication, each block of A moves one step left and each block of B moves one step up (with wraparound)



- A sequence of \sqrt{p} such submatrix multiplications and single-step shifts pairs up $A_{i,k}$ and $B_{k,j}$ for k ($0 \leq k < \sqrt{p}$) at $P_{i,j}$
- This completes the multiplication of matrices A and B

Cyclic Data-Block movement

$A_{0,2}$ $B_{2,0}$	$A_{0,3}$ $B_{3,1}$	$A_{0,0}$ $B_{0,2}$	$A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$	$A_{1,0}$ $B_{0,1}$	$A_{1,1}$ $B_{1,2}$	$A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$	$A_{2,1}$ $B_{1,1}$	$A_{2,2}$ $B_{2,2}$	$A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$	$A_{3,2}$ $B_{2,1}$	$A_{3,3}$ $B_{3,2}$	$A_{3,0}$ $B_{0,3}$

Submatrix locations
after second shift

Cyclic
Data-Block
movement

Cyclic Data-Block movement

$A_{0,3}$ $B_{3,0}$	$A_{0,0}$ $B_{0,1}$	$A_{0,1}$ $B_{1,2}$	$A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$	$A_{1,1}$ $B_{1,1}$	$A_{1,2}$ $B_{2,2}$	$A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$	$A_{2,2}$ $B_{2,1}$	$A_{2,3}$ $B_{3,2}$	$A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$	$A_{3,3}$ $B_{3,1}$	$A_{3,0}$ $B_{0,2}$	$A_{3,1}$ $B_{1,3}$

Submatrix locations
after third shift

. . . Declarations, etc . . .

/* Get communicator related information */

MPI_Comm_size(comm, &npes);

MPI_Comm_rank(comm, &myrank);

/* Set up Cartesian topology */

dims[0] = dims[1] = sqrt(npes);

/* Set the periods for wraparound connections */

periods[0] = periods[1] = 1;

/* Create Cartesian topology with rank reordering */

MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);

/*Get rank and coordinates with respect to new topology */

MPI_Comm_rank(comm_2d, &my2drank);

MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

```
int i, nlocal, npes, dims[2], periods[2];
int myrank, my2drank, mycoords[2];
int uprank, downrank, leftrank, rightrank;
int coords[2], shiftsource, shiftdest;
MPI_Status status;
MPI_Comm comm_2d;
```

```

/*Compute ranks of the up and left shifts */
MPI_Cart_shift(comm_2d,0,-1,&rightrank,&leftrank);
MPI_Cart_shift(comm_2d,1,-1,&downrank,&uprank);

/*Determine dimensions of the local matrix block */
nlocal = n/dims[0];

/*Perform initial matrix alignment. First for A, then for B*/
MPI_Cart_shift(comm_2d,0,-mycoords[0],&shiftsource,&shiftdest);
MPI_Sendrecv_replace(a,nlocal*nlocal,MPI_DOUBLE,shiftdest,
    1,shiftsource,1,comm_2d,&status);

MPI_Cart_shift(comm_2d,1,-mycoords[1],&shiftsource,&shiftdest);
MPI_Sendrecv_replace(b,nlocal*nlocal,MPI_DOUBLE,shiftdest,
    1,shiftsource,1,comm_2d,&status);

```

```

/* Get into the main computation loop */
for(i=0; i<dims[0]; i++) {
    MatrixMultiply(nlocal,a,b,c); /* c = c + a*b */

    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a,nlocal*nlocal,MPI_DOUBLE,
        leftrank,1,rightrank,1,comm_2d,&status);

    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b,nlocal*nlocal,MPI_DOUBLE,
        uprank,1,downrank,1,comm_2d,&status);
}

/* Restore original distribution of a and b */
... Similar shifting code but with +ve mycoords[0/1] ...
MPI_Comm_free(&comm_2d); /* Free up communicator */
}

```

```

/* Function to perform matrix-matrix multiplication c=a*b */
MatrixMultiply(int n, double *a, double *b, double *c)
{
    int i,j,k;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            for(k=0; k<n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j];
}

```