MPI – Groups and Communicators

• We now learn techniques to modularize communications at a more local level so that messages do not interfere with each other
• We do this by setting up Groups of processes
• We then create a new communicator based on the subgroup structure
• All MPI communication commands involve a communicator, so far we have only used MPI_COMM_WORLD; we can now use other communicators based on group structures
MPI Group Commands

- MPI_Group_size returns number of processes in group
- MPI_Group_rank returns rank of calling process in group
- MPI_Group_compare compares group members and group order
- MPI_Group_translate_ranks translates ranks of processes in one group to those in another group
- MPI_Comm_group returns the group associated with a communicator
- MPI_Group_union creates a group by combining two groups
- MPI_Group_intersection creates a group from the intersection of two groups
- MPI_Group_difference creates a group from the difference between two groups
- MPI_Group_incl creates a group from listed members of an existing group
- MPI_Group_excl creates a group excluding listed members of an existing group
- MPI_Group_range_incl creates a group according to first rank, stride, last rank
- MPI_Group_range_excl creates a group by deleting according to first rank, stride, last rank
- MPI_Group_free marks a group for deallocation

These are the most common of the commands
Details on the parameters will be given as part of the exercises
MPI Communicator Commands

- **MPI_Comm_size** returns number of processes in communicator's group
- **MPI_Comm_rank** returns rank of calling process in communicator's group
- **MPI_Comm_compare** compares two communicators
- **MPI_Comm_dup** duplicates a communicator
- **MPI_Comm_create** creates a new communicator for a group
- **MPI_Comm_split** splits a communicator into multiple, non-overlapping communicators
- **MPI_Comm_free** marks a communicator for deallocation

- These are the most common of the commands
- Details on the parameters will be given as part of the exercises
Example: Eight Subgroups of Size 4

• Assume 32 processes are available

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

• Here are the commands to create these groups

```c
MPI_Group orig_group, new_group; MPI_Comm new_comm;
/* initialize, get size, get ranks, etc. here */
/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
/* Divide tasks into eight distinct groups based upon rank */
if (rank < NPROCS/8) {
    MPI_Group_incl(orig_group, NPROCS/8, ranks1, &new_group);
} else .......... /* other seven cases */
/* Create new new communicator */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
```
In-Class Lab

• You will use eight processes
  – Process 0, 1, 2, 3 will be assigned to the first group
  – Process 4, 5, 6, 7 will be assigned to the second group
  – Each process will know its original rank and its new rank within the new groups
  – You will create one new communicator based on these groups

• Using the new communicator
  – Have each process sum up the original ranks in its group using reduce all
  – The result for the first group is 6, the result for the second group is 22

• The lab sheet describes this lab in detail; check off with the lab assistant when finished
MPI_COMM_DUP and MPI_COMM_FREE

• A program may create a new communicator comprising the same process group
  – Uses a new context to ensure that communications performed for different purposes are not confused.
  – This mechanism supports sequential composition.

\[\text{MPI\_COMM\_DUP}\text{(comm, newcomm)}\]
Create new communicator: same group, new context.

  IN  comm  communicator (handle)
  OUT newcomm  communicator (handle)

• A communicator is discarded after it is done

\[\text{MPI\_COMM\_FREE}\text{(comm)}\]
Destroy a communicator.

  IN  comm  communicator (handle)
A Simple Example

• We want to transpose a matrix using a local communicator that will not interfere with other communications

integer comm, newcomm, ierr ! Handles are integers ...
call MPI_COMM_DUP(comm, newcomm, ierr) ! Create new context
call transpose(newcomm, A) ! Pass matrix A to library
call MPI_COMM_FREE(newcomm, ierr) ! Free new context

This example is in Fortran, but you get the idea
MPI_COMM_SPLIT

- A program may create a new communicator comprising just a subset of a given group of processes.
  - These processes can then communicate among themselves without fear of conflict with other concurrent computations.
  - This mechanism supports parallel composition.

\[
\text{MPI\_COMM\_SPLIT}(\text{comm, color, key, newcomm}) \\
\text{Partition group into disjoint subgroups.}
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>comm</td>
</tr>
<tr>
<td>IN</td>
<td>color</td>
</tr>
<tr>
<td>IN</td>
<td>key</td>
</tr>
<tr>
<td>OUT</td>
<td>newcomm</td>
</tr>
<tr>
<td></td>
<td>communicator (handle)</td>
</tr>
<tr>
<td></td>
<td>subgroup control (integer)</td>
</tr>
<tr>
<td></td>
<td>process id control (integer)</td>
</tr>
<tr>
<td></td>
<td>communicator (handle)</td>
</tr>
</tbody>
</table>
A Simple Example

- The goal is to split eight processes into three communicators with the ID of each process defined by $\text{myid mod 3}$

```c
MPI_Comm comm, newcomm; int myid, color;
MPI_Comm_rank(comm, &myid);
color = myid % 3;
MPI_Comm_split(comm, color, myid, &newcomm);
```

Notice that in the subcommunicators the numbering starts at 0 and increases by 1.
Program 12

• What will be printed by this program? (assume –n 8)

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    MPI_Comm MY_COMM;
    int world_rank, my_rank, my_size;
    static int value[12] = {0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3};
    int color;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    color = value[world_rank];
    MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &MY_COMM);

    MPI_Comm_rank(MY_COMM, &my_rank);
    MPI_Comm_size(MY_COMM, &my_size);

    printf("world_rank=%d my_rank=%d color=%d my_size=%d\n", 
            world_rank, my_rank, color, my_size);

    MPI_Finalize();
}
```
[blk@virtual prog12]$ mpirun -n 8 ./prog12
world_rank=0 my_rank=0 color=0 my_size=2
world_rank=2 my_rank=0 color=1 my_size=3
world_rank=6 my_rank=1 color=2 my_size=3
world_rank=5 my_rank=0 color=2 my_size=3
world_rank=1 my_rank=1 color=0 my_size=2
world_rank=4 my_rank=2 color=1 my_size=3
world_rank=3 my_rank=1 color=1 my_size=3
world_rank=7 my_rank=2 color=2 my_size=3
[blk@virtual prog12]$
MPI_INTERCOM_CREATE

• A program may construct an intercommunicator
  – This links processes in two groups.
  – This mechanism supports parallel composition.

MPI_INTERCOMM_CREATE(comm, leader, peer, rleader, tag, inter)
Create an intercommunicator.

  IN    comm    local intracommunicator (handle)
  IN    leader  local leader (integer)
  IN    peer    peer intracommunicator (handle)
  IN    rleader process id of remote leader in peer (integer)
  IN    tag     tag for communicator set up (integer)
  OUT   inter   new intercommunicator (handle)
A Simple Example

• Suppose MPI_COMM_SPLIT has taken 8 processes and created two distinct communicators: the first contains the even elements and the second contains the odd elements

• The goal is the set up intercommunications between these two communicators
The Program Code

```fortran
integer comm, intercomm, ierr, status(MPI_STATUS_SIZE)

C For simplicity, we require an even number of processes
   call MPI_COMM_SIZE(MPI_COMM_WORLD, count, ierr)
   if(mod(count,2) .ne. 0) stop

C Split processes into two groups: odd and even numbered
   call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
   call MPI_COMM_SPLIT(MPI_COMM_WORLD, mod(myid,2), myid,
                        comm, ierr)

C Determine process id in new group
   call MPI_COMM_RANK(comm, newid, ierr)
   if(mod(newid,2) .eq. 0) then

   C Group 0: create intercommunicator and send message
   C Arguments: 0=local leader; 1=remote leader; 99=tag
      call MPI_INTERCOMM_CREATE(comm, 0, MPI_COMM_WORLD, 1, 99,
                                  intercomm, ierr)
      call MPI_SEND(msg, 1, type, newid, 0, intercomm, ierr)
   else

   C Group 1: create intercommunicator and receive message
   C Note that remote leader has id 0 in MPI_COMM_WORLD
      call MPI_INTERCOMM_CREATE(comm, 0, MPI_COMM_WORLD, 0, 99,
                                  intercomm, ierr)
      call MPI_RECV(msg, 1, type, newid, 0, intercomm,
                    status, ierr)
   endif

C Free communicators created during this operation
   call MPI_COMM_FREE(intercomm, ierr)
   call MPI_COMM_FREE(comm, ierr)
```

this example is in FORTRAN, the intent should be clear
#include <stdio.h>
#include <mpi.h>

main(int argc, char* argv[]) {
    MPI_Comm MY_COMM;
    MPI_Comm THE_INTER;
    int world_rank, local_rank, inter_rank;
    int color, i;
    int arr[5];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&world_rank);
    color=world_rank%2;
    MPI_Comm_split(MPI_COMM_WORLD,color,0,&MY_COMM);
    if (color==0) {
        MPI_Intercomm_create(MY_COMM,0,MPI_COMM_WORLD,1,33,&THE_INTER);
        MPI_Comm_rank(MY_COMM, &local_rank);
        MPI_Comm_rank(THE_INTER, &inter_rank);
        if(local_rank==0){
            printf("color=%d inter_rank=%d local rank %d: ", color, inter_rank, local_rank);
            for(i = 0; i < 5; i++){
                printf("%d ", arr[i]);
            }printf("\n");
            MPI_Send(arr, 5, MPI_INT, 0, 0, THE_INTER);
        }
    }
}
else if (color==1) {
    MPI_Intercomm_create(MY_COMM,0,MPI_COMM_WORLD,0,33,&THE_INTER);
    MPI_Comm_rank(MY_COMM, &local_rank);
    MPI_Comm_rank(THE_INTER, &inter_rank);
    if(local_rank == 0){
        printf("About to receive. color=%d inter_rank=%d P%d: ",
            color, inter_rank, local_rank);
        MPI_Recv(arr, 5, MPI_INT, 0, 0, THE_INTER, &status);
        for(i = 0; i<5;i++){
            printf("%d ", arr[i]);
        }
        printf("\n");
    }
    MPI_Comm_free(&THE_INTER);
    MPI_Finalize();
}

What is printed by this program?
[blk@virtual prog13]$ mpirun -n 2 ./prog13
color=0 inter_rank=0 local rank 0: 0 1 2 3 4
About to recieve. color=1 inter_rank=0 P0: 0 1 2 3 4
[blk@virtual prog13]$
Exercise: Using Intercommunicators

- Split the processes as shown: 0,2,4,6 in one group and 1,3,5,7 in one group
- Set up intercommunicator between corresponding elements in subgroups
- Use send and receive between paired processors to exchange their world ranks; print the sums
- For the original ranks 0..7 the results are 1, 1, 5, 5, 9, 9, 13, 13; the order will vary due to parallelism
- Have the lab assistant check off your program when completed
MPI_Cart_create

Makes a new communicator to which topology information has been attached

- **Synopsis**
  ```c
  #include "mpi.h"
  int MPI_Cart_create ( MPI_Comm comm_old, int ndims,
   int *dims, int *periods, int reorder, MPI_Comm
   *comm_cart )
  ```

- **Input Parameters**
  - `comm_old` input communicator (handle)
  - `ndims` number of dimensions of cartesian grid (integer)
  - `dims` integer array of size ndims specifying the number of processes in each dimension
  - `periods` logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
  - `reorder` ranking may be reordered (true) or not (false) (logical)

- **Output Parameter**
  - `comm_cart` communicator with new cartesian topology (handle)
MPI_Cart_coords

Determines process coordinates in cartesian topology given rank in group

• **Synopsis**
  
  `#include "mpi.h" int MPI_Cart_coords ( MPI_Comm comm, int rank, int maxdims, int *coords )`

• **Input Parameters**
  
  - `comm` communicator with cartesian structure (handle)
  - `rank` rank of a process within group of comm (integer)
  - `maxdims` length of vector coords in the calling program (integer)

• **Output Parameter**
  
  - `coords` integer array (of size ndims) containing the Cartesian coordinates of specified process (integer)
MPI_Cart_sub

Partitions a communicator into subgroups which form lower-dimensional cartesian subgrids

• **Synopsis**
  ```c
  #include "mpi.h"
  int MPI_Cart_sub ( MPI_Comm comm, int *remain_dims, MPI_Comm *comm_new )
  ```

• **Input Parameters**
  - `comm` communicator with cartesian structure (handle)
  - `remain_dims` the ith entry of remain_dims specifies whether the ith dimension is kept in the subgrid (true) or is dropped (false) (logical vector)

• **Output Parameter**
  - `newcomm` communicator containing the subgrid that includes the calling process (handle)
MPI_Cart_rank

Determines process rank in communicator given Cartesian location

- **Synopsis**
  
  ```
  #include "mpi.h"
  int MPI_Cart_rank ( MPI_Comm comm, int *coords, int *rank )
  ```

- **Input Parameters**
  
  - `comm` communicator with cartesian structure (handle)
  - `coords` integer array (of size ndims) specifying the cartesian coordinates of a process

- **Output Parameter**
  
  - `rank` rank of specified process (integer)
Matrix Multiplication

The classic sequential algorithm

```java
for (int i = 0 ; i < ROWS ; i++) {
    for (int j = 0 ; j < COLUMNS ; j++) {
        double sum = 0.0 ;
        for (int k = 0 ; k < COLUMNS ; k++) {
            sum = sum + A[i][k] * B[k][j] ;
        }
        C[i][j] = sum ;
    }
}
```

• The complexity for an n x n matrix is \( O(n^3) \)
# How to Decompose the Matrix

- **Some choices**

\[
\begin{pmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33}
\end{pmatrix}
\]

\[
\begin{pmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33}
\end{pmatrix}
\]

Which choice do you think will be most productive?
### Multiplication of Submatrices

\[
\begin{pmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & a_{12} & a_{13} \\
  a_{20} & a_{21} & a_{22} & a_{23} \\
  a_{30} & a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
  b_{00} & b_{01} & b_{02} & b_{03} \\
  b_{10} & b_{11} & b_{12} & b_{13} \\
  b_{20} & b_{21} & b_{22} & b_{23} \\
  b_{30} & b_{31} & b_{32} & b_{33}
\end{pmatrix}
\]

\[
\begin{pmatrix}
  \stackrel{\wedge}{A_{00}} & \stackrel{\wedge}{A_{01}} \\
  \stackrel{\wedge}{A_{10}} & \stackrel{\wedge}{A_{11}}
\end{pmatrix}
\begin{pmatrix}
  \stackrel{\wedge}{B_{00}} & \stackrel{\wedge}{B_{01}} \\
  \stackrel{\wedge}{B_{10}} & \stackrel{\wedge}{B_{11}}
\end{pmatrix}
= 
\begin{pmatrix}
  \stackrel{\wedge}{C_{00}} & \stackrel{\wedge}{C_{01}} \\
  \stackrel{\wedge}{C_{10}} & \stackrel{\wedge}{C_{11}}
\end{pmatrix}
\]

\[
\begin{align*}
\stackrel{\wedge}{C_{00}} &= \stackrel{\wedge}{A_{00}} \stackrel{\wedge}{B_{00}} + \stackrel{\wedge}{A_{01}} \stackrel{\wedge}{B_{10}} \\
\stackrel{\wedge}{C_{10}} &= \stackrel{\wedge}{A_{10}} \stackrel{\wedge}{B_{00}} + \stackrel{\wedge}{A_{11}} \stackrel{\wedge}{B_{10}} \\
\stackrel{\wedge}{C_{01}} &= \stackrel{\wedge}{A_{00}} \stackrel{\wedge}{B_{01}} + \stackrel{\wedge}{A_{01}} \stackrel{\wedge}{B_{11}} \\
\stackrel{\wedge}{C_{11}} &= \stackrel{\wedge}{A_{10}} \stackrel{\wedge}{B_{01}} + \stackrel{\wedge}{A_{11}} \stackrel{\wedge}{B_{11}}
\end{align*}
\]
Sub-block definition of Matrix Multiply

- Let $\hat{C}^{lk}$ be the sub-block at position $(l,k)$, then the problem can be stated in block matrix form:

$$\hat{C}^{i\kappa} = \sum_{n=0}^{\sqrt{N}-1} \hat{A}^i_n \hat{B}^{n\kappa}$$

- Example of sub-block structure of matrix $A$ where $N = 16$ ($\sqrt{N} = 4$):

<table>
<thead>
<tr>
<th>$\hat{A}^0_0$</th>
<th>$\hat{A}^0_1$</th>
<th>$\hat{A}^0_2$</th>
<th>$\hat{A}^0_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\hat{A}^1_0$</td>
<td>$\hat{A}^1_1$</td>
<td>$\hat{A}^1_2$</td>
<td>$\hat{A}^1_3$</td>
</tr>
<tr>
<td>$\hat{A}^2_0$</td>
<td>$\hat{A}^2_1$</td>
<td>$\hat{A}^2_2$</td>
<td>$\hat{A}^2_3$</td>
</tr>
<tr>
<td>$\hat{A}^3_0$</td>
<td>$\hat{A}^3_1$</td>
<td>$\hat{A}^3_2$</td>
<td>$\hat{A}^3_3$</td>
</tr>
</tbody>
</table>

The entire matrix is 16 x 16; each sub-block is 4 x 4.

Note indices start at 0 for rows and columns of matrices. They start at 1 for rows and columns of processors.
Fox’s Algorithm
(Broadcast, Multiply, and Roll)

Set $C = 0$;
for ($p = 0: \sqrt{N} - 1$)

In each row $i$ of processors broadcast the sub-block $\hat{A}^{ij}$ to the other processors in the row, where $j = (i+p) \mod \sqrt{N}$. Each processor stores the broadcast sub-block in an array $T$.

Multiply temporary submatrix $T$ in each processor by the current $B$ sub-block and add the result to $C$.

Each processor sends its current $B$ sub-block to the processor above and receives a sub-block from the processor below and makes it the new current $B$ sub-block. This wraps around from top to bottom.

end for
The first stage -- index \( n=0 \) in sub-block sum -- of the algorithm on \( N=16 \) example

- **Broadcast sub-blocks of \( \hat{A} \) along rows to form matrix \( T \)**

- **Multiply sub-blocks and add into \( C \)**

\[
\begin{array}{ccc}
\hat{A}^{00} & \hat{A}^{01} & \hat{A}^{02} & \hat{A}^{03} \\
\hat{A}^{10} & \hat{A}^{11} & \hat{A}^{12} & \hat{A}^{13} \\
\hat{A}^{20} & \hat{A}^{21} & \hat{A}^{22} & \hat{A}^{23} \\
\hat{A}^{30} & \hat{A}^{31} & \hat{A}^{32} & \hat{A}^{33} \\
\end{array}
\begin{array}{ccc}
\hat{A}^{00} & \hat{A}^{00} & \hat{A}^{00} & \hat{A}^{00} \\
\hat{A}^{11} & \hat{A}^{11} & \hat{A}^{11} & \hat{A}^{11} \\
\hat{A}^{22} & \hat{A}^{22} & \hat{A}^{22} & \hat{A}^{22} \\
\hat{A}^{33} & \hat{A}^{33} & \hat{A}^{33} & \hat{A}^{33} \\
\end{array}
\begin{array}{ccc}
\hat{B}^{00} & \hat{B}^{01} & \hat{B}^{02} & \hat{B}^{03} \\
\hat{B}^{10} & \hat{B}^{11} & \hat{B}^{12} & \hat{B}^{13} \\
\hat{B}^{20} & \hat{B}^{21} & \hat{B}^{22} & \hat{B}^{23} \\
\hat{B}^{30} & \hat{B}^{31} & \hat{B}^{32} & \hat{B}^{33} \\
\end{array}
\end{array}
\]
The second stage -- n=1 in sum over subblock indices -- of the algorithm on N=16 example

- Roll B up one row (picture shows result after roll with correct elements of B in place)

- Broadcast the next sub-blocks of A along the rows
Second stage, continued

- Multiply sub-blocks and add into $C$

\[
\begin{array}{cccc}
\hat{A}^{00} & \hat{B}^{00} & + & \hat{A}^{01} \hat{B}^{10} \\
\hat{A}^{11} & \hat{B}^{10} & + & \hat{A}^{12} \hat{B}^{20} \\
\hat{A}^{22} & \hat{B}^{20} & + & \hat{A}^{23} \hat{B}^{30} \\
\hat{A}^{33} & \hat{B}^{30} & + & \hat{A}^{00} \hat{B}^{00} \\
\end{array}
\begin{array}{cccc}
\hat{A}^{00} & \hat{B}^{01} & + & \hat{A}^{01} \hat{B}^{11} \\
\hat{A}^{11} & \hat{B}^{11} & + & \hat{A}^{12} \hat{B}^{21} \\
\hat{A}^{22} & \hat{B}^{21} & + & \hat{A}^{23} \hat{B}^{31} \\
\hat{A}^{33} & \hat{B}^{31} & + & \hat{A}^{00} \hat{B}^{01} \\
\end{array}
\begin{array}{cccc}
\hat{A}^{02} & \hat{B}^{02} & + & \hat{A}^{01} \hat{B}^{12} \\
\hat{A}^{12} & \hat{B}^{12} & + & \hat{A}^{13} \hat{B}^{22} \\
\hat{A}^{22} & \hat{B}^{22} & + & \hat{A}^{23} \hat{B}^{32} \\
\hat{A}^{33} & \hat{B}^{32} & + & \hat{A}^{00} \hat{B}^{02} \\
\end{array}
\begin{array}{cccc}
\hat{A}^{03} & \hat{B}^{03} & + & \hat{A}^{01} \hat{B}^{13} \\
\hat{A}^{13} & \hat{B}^{13} & + & \hat{A}^{12} \hat{B}^{23} \\
\hat{A}^{23} & \hat{B}^{23} & + & \hat{A}^{22} \hat{B}^{33} \\
\hat{A}^{33} & \hat{B}^{33} & + & \hat{A}^{00} \hat{B}^{03} \\
\end{array}
\]

\[
\begin{array}{ccc}
\hat{A}^{01} & \hat{A}^{01} & \hat{A}^{01} \\
\hat{A}^{12} & \hat{A}^{12} & \hat{A}^{12} \\
\hat{A}^{23} & \hat{A}^{23} & \hat{A}^{23} \\
\hat{A}^{30} & \hat{A}^{30} & \hat{A}^{30} \\
\end{array}
\begin{array}{ccc}
\hat{B}^{10} & \hat{B}^{11} & \hat{B}^{12} \\
\hat{B}^{21} & \hat{B}^{22} & \hat{B}^{23} \\
\hat{B}^{31} & \hat{B}^{32} & \hat{B}^{33} \\
\hat{B}^{00} & \hat{B}^{01} & \hat{B}^{02} \\
\end{array}
\begin{array}{ccc}
\hat{A}^{01} & \hat{A}^{01} & \hat{A}^{01} \\
\hat{A}^{12} & \hat{A}^{12} & \hat{A}^{12} \\
\hat{A}^{23} & \hat{A}^{23} & \hat{A}^{23} \\
\hat{A}^{30} & \hat{A}^{30} & \hat{A}^{30} \\
\end{array}
\begin{array}{ccc}
\hat{B}^{10} & \hat{B}^{11} & \hat{B}^{12} \\
\hat{B}^{21} & \hat{B}^{22} & \hat{B}^{23} \\
\hat{B}^{31} & \hat{B}^{32} & \hat{B}^{33} \\
\hat{B}^{00} & \hat{B}^{01} & \hat{B}^{02} \\
\end{array}
\]

\[
C = T + B
\]
Look at the whole algorithm on one element

\[
p = 0: \quad T = A^{2^2}, \quad B = B^{2^1} \\
C = A^{2^2} B^{2^1}
\]

\[
p = 1: \quad T = A^{2^3}, \quad B = B^{3^1} \\
C = A^{2^2} B^{2^1} + A^{2^3} B^{3^1}
\]

\[
p = 2: \quad T = A^{2^0}, \quad B = B^{0^1} \\
C = A^{2^2} B^{2^1} + A^{2^3} B^{3^1} + A^{2^0} B^{0^1}
\]

\[
p = 3: \quad T = A^{2^1}, \quad B = B^{1^1} \\
C = A^{2^2} B^{2^1} + A^{2^3} B^{3^1} + A^{2^0} B^{0^1} + A^{2^1} B^{1^1}
\]
More Details in Pseudocode

- Calculating sub-block $C[i,j]$
  
  $q = \sqrt{p}$;
  
  $dest = ((i-1) \mod q, j);$
  
  $source = ((i+1) \mod q, j);$
  
  for(stage = 0; stage < q; stage++) {
    
    $k_{bar} = (i + stage) \mod q;$
    
    Broadcast $A[i, k_{bar}]$ across row i;
    
    $C[i,j] = C[i,j] + A[i, k_{bar}] \times B[k_{bar}, j]$;
  
    Send $B[k_{bar}, j]$ to dest;
    
    Receive $B[(k_{bar}+1) \mod q, j]$ from source;
  
  }

- We will use communicators for the Broadcast, Send and Receive
Holding information in a struct

- We will use a struct to facilitate passing a heterogeneous set of information easily

```c
typedef struct {
    int p;    /* number of processes */
    MPI_Comm comm;    /* Communicator for entire grid */
    MPI_Comm row_comm;    /* Communicator for my row */
    MPI_Comm col_comm;    /* Communicator for my col */
    int q;    /* order of grid */
    int my_row;    /* my row number */
    int my_col;    /* my col number */
    int my_rank;    /* my rank in the grid comm */
} GRID_INFO_T;
```
Setting up the grid - 1

• Setting up grid information
  
  MPI_Comm_size(MPI_COMM_WORLD, &grid->p);
  MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

  • Assuming that p is a perfect square we assign
    
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;

  • Setting up for a vertical circular shift
    
    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_rank);
    MPI_Cart_coords(grid->comm, grid->my_rank2,
                     2, coordinates);
    grid->my_row = coordinates[0];
    grid->my_col = coordinates[1];
Setting up the grid - 2

• Setting up the row communicators

```c
free_coords[0] = 0;
free_coords[1] = 1;
MPI_Comm_sub(grid->comm, free_coords,
    &(grid->row_comm));
```

• Setting up the column communicators

```c
free_coords[0] = 1;
free_coords[1] = 0;
MPI_Comm_sub(grid->comm, free_coords,
    &(grid->col_comm));
```
A New MPI Command

• When the row of B are rolled upward (with wrap around), you will need to use the MPI command

```c
int MPI_Sendrecv_replace (
    void* buffer /* in/out */,
    int count /* in */ ,
    MPI_Datatype datatype /* in */ ,
    int dest /* in */ ,
    int send_tag /* in */ ,
    int source /* in */ ,
    int recv_tag /* in */ ,
    MPI_Comm comm /* in */ ,
    MPI_Status* status /* out */
)
```
Set up for Fox’s Algorithm - 1

• Parameters

```c
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 Declarations

```c
LOCAL_MATRIX_T* temp_A;
int stage;
int bcast_root;
int n_bar;
int source;
int dest;
MPI_Status status;
```
Set up for Fox’s Algorithm - 2

• Initial Assignments

\[
\begin{align*}
n_{\text{bar}} &= n / \text{grid->q;} \\
\text{Set\_to\_zero(local\_C);} \\
\text{source} &= (\text{grid->my\_row + 1}) \mod \text{grid->q;} \\
\text{dest} &= (\text{grid->my\_row + grid->q - 1}) \mod \text{grid->q;} \\
\text{temp\_A} &= \text{local\_matrix\_allocate(n\_bar);} \\
\end{align*}
\]

• Other tasks

you need to write a method Local\_matrix\_multiply that multiplies sub-block A times sub-block B and returns sub-block C as an out parameter
Implementing the Main Loop

• For the grid->p stages do the following

```plaintext
set bcast_root to (grid->my_row + stage) mod grid->q
if the root equals my_col then
    broadcast local_A across the row
    calculate local_C as local_A * local_B
else
    broadcast temp_A across the row
    calculate local_C as temp_A * local_B
end if
call MPI_Sendrecv_replace to roll the rows of B upward
```
Timing Routines

• **MPI_Wtick**
  Returns the resolution of **MPI_Wtime**

  **Synopsis**
  
  ```c
  #include "mpi.h"
  double MPI_Wtick()
  ```

  **Return value**
  
  Time in seconds of resolution of **MPI_Wtime**

• **MPI_Wtime**
  Returns an elapsed time on the calling processor

  **Synopsis**
  
  ```c
  #include "mpi.h"
  double MPI_Wtime()
  ```

  **Return value**
  
  Time in seconds since an arbitrary time in the past
Example of Running Fox’s Algorithm

• Number of processes is a perfect square and divides the size of the matrix

• Two matrices of all 1’s are generated

[blk@virtual fox]$ mpirun -n 4 ./fox

What's the order of the matrices?
8

Enter A

We read A =

1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
Enter B
We read B =

1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

The product is

8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
8.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0

time of computation = 0.002921
[blk@virtual fox]$
Program: Complete Fox’s Algorithm

• Your task is to get Fox’s algorithm fully functional using MPI
  – Define the method Fox
  – Define the local matrix multiply

• Add timing routines to Fox’s algorithm so that you can collect performance data
  – Collect data for Fox’s algorithm compared with the traditional sequential algorithm
  – Increase the matrix size until you see some differences in performance
  – Include your results in a readme file