Logistics

- More on MPI communicators today
  - Splitting them and “grids with them”
- I am supposed to be gone first week of April
  - Not all week but will likely miss one lecture
    - Details still unknown
- Any Questions?
### Layout of Processes

<table>
<thead>
<tr>
<th>Process Mappings</th>
<th>Process Column 0</th>
<th>Process Column 1</th>
<th>Process Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process Row 0</strong></td>
<td>0 (0,0)</td>
<td>1 (0,1)</td>
<td>2 (0,2)</td>
</tr>
<tr>
<td><strong>Process Row 1</strong></td>
<td>3 (1,0)</td>
<td>4 (1,1)</td>
<td>5 (1,2)</td>
</tr>
<tr>
<td><strong>Process Row 2</strong></td>
<td>6 (2,0)</td>
<td>7 (2,1)</td>
<td>8 (2,2)</td>
</tr>
</tbody>
</table>
MPI_Comm_split for Fox Algorithm

- Remember that \( q = \sqrt{\text{Nproc}} \)
- For the row communicators;
  - \( \text{Myrow} = \text{my\_rank}/q; \)
  - \( \text{MPI\_Comm\_split}(	ext{MPI\_COMM\_WORLD,Myrow, my\_rank,row\_comm}); \)
- For the column communicators
  - \( \text{Mycol} = \text{my\_rank} \mod q; \)
  - \( \text{MPI\_Comm\_split}(	ext{MPI\_COMM\_WORLD,Mycol, my\_rank,col\_comm}); \)
Code fragment for comm_split

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

/* confirm that the number of processes is a perfect square somehow */

/* generate row communicators by identifying my process row */
myrow = myrank / q;
MPI_Comm_split(MPI_COMM_WORLD,myrow,
               myrank,&MPI_row);

/* generate column communicators by identifying my process column */
mycol = myrank % q;
MPI_Comm_split(MPI_COMM_WORLD,mycol,
               myrank,&MPI_col);
/* Check ranks in row and column communicators */
MPI_Comm_rank(MPI_row,&rrow);
MPI_Comm_rank(MPI_col,&rcol);
sleep((1+2*myrank));
printf("world=%d,row=%d,col=%d,
 coords=(%d,%d)\n", myrank, rrow, rcol,
 myrow,mycol);
(void)fflush(stdout);
Output from Comm_split

world=0,row=0,col=0,coords=(0,0)
world=1,row=1,col=0,coords=(0,1)
world=2,row=2,col=0,coords=(0,2)
world=3,row=0,col=1,coords=(1,0)
world=4,row=1,col=1,coords=(1,1)
world=5,row=2,col=1,coords=(1,2)
world=6,row=0,col=2,coords=(2,0)
world=7,row=1,col=2,coords=(2,1)
world=8,row=2,col=2,coords=(2,2)
# Layout of Processes

<table>
<thead>
<tr>
<th>Process Mappings</th>
<th>Process Column 0</th>
<th>Process Column 1</th>
<th>Process Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Row 0</td>
<td>0 (0,0)</td>
<td>1 (0,1)</td>
<td>2 (0,2)</td>
</tr>
<tr>
<td>Process Row 1</td>
<td>3 (1,0)</td>
<td>4 (1,1)</td>
<td>5 (1,2)</td>
</tr>
<tr>
<td>Process Row 2</td>
<td>6 (2,0)</td>
<td>7 (2,1)</td>
<td>8 (2,2)</td>
</tr>
</tbody>
</table>
Virtual Topologies

• For Fox’s algorithm assume a square grid of processes that are in MPI_COMM_WORLD.

• We need to identify:
  ➢ The number of dimensions (2)
  ➢ The size of each dimension
    ▪ The number of rows and number of columns
      ♦ Square root of number of processes and both are “q”
  ➢ Periodicity in each dimension
    ▪ Is the first element in this dimension adjacent to the last element.
      ♦ For columns YES
      ♦ For Rows not important (broadcast).
  ➢ Do we allow MPI to reorder processes.
MPI_Cart_create

- **Makes a new communicator to which topology information has been attached**

- **Synopsis**
  - `#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
  - `dims` integer array 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)

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

- Determines process coords 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

Output Parameter

- coords integer array (of size ndims) containing the cartesian coordinates of specified process (integer)
Creating the Fox Grid

dims[0] = dims[1] = (int)sqrt((double)nproc);
wraps[0] = wraps[1] = 1;
reorder = 1
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, wraps, reorder, &grid_comm);
MPI_Comm_size(grid_comm, &grid_size);
MPI_Comm_rank(grid_comm, &mygrid_rank);
MPI_Cart_coords(grid_comm, mygrid_rank, 2, coords);
printf("oldrank=%d, grid_rank=%d, 
gird_size=%d coords(%d,%d)\n", oldrank, mygrid_rank, grid_size, coords[0], coords[1]);
When we run this fragment

- oldrank=0, grid_rank=0, grid_size=9 coords(0,0)
- oldrank=1, grid_rank=1, grid_size=9 coords(0,1)
- oldrank=2, grid_rank=2, grid_size=9 coords(0,2)
- oldrank=3, grid_rank=3, grid_size=9 coords(1,0)
- oldrank=4, grid_rank=4, grid_size=9 coords(1,1)
- oldrank=5, grid_rank=5, grid_size=9 coords(1,2)
- oldrank=6, grid_rank=6, grid_size=9 coords(2,0)
- oldrank=7, grid_rank=7, grid_size=9 coords(2,1)
- oldrank=8, grid_rank=8, grid_size=9 coords(2,2)
## Layout of Processes

<table>
<thead>
<tr>
<th>Process Mappings</th>
<th>Process Column 0</th>
<th>Process Column 1</th>
<th>Process Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Row 0</td>
<td>0 (0,0)</td>
<td>1 (0,1)</td>
<td>2 (0,2)</td>
</tr>
<tr>
<td>Process Row 1</td>
<td>3 (1,0)</td>
<td>4 (1,1)</td>
<td>5 (1,2)</td>
</tr>
<tr>
<td>Process Row 2</td>
<td>6 (2,0)</td>
<td>7 (2,1)</td>
<td>8 (2,2)</td>
</tr>
</tbody>
</table>
Now for column/row communicators

- Find a “sub-dimension” of the grid!

/* do rows */
freec[0] = 0;
freec[1] = 1;
MPI_Cart_sub(grid_comm, freec, &row_comm);
MPI_Comm_size(row_comm, &row_size);
MPI_Comm_rank(row_comm, &row_rank);

/* do column */
freec[0] = 1;
freec[1] = 0;
MPI_Cart_sub(grid_comm, freec, &col_comm);
MPI_Comm_size(col_comm, &col_size);
MPI_Comm_rank(col_comm, &col_rank);
printf(" grid_rank=%d, row_rank=%d, row_size=%d,
       col_rank=%d, col_size=%d\n",
       mygrid_rank,row_rank, row_size, col_rank,col_size);
MPI_Cart_sub

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

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

**Input Parameters**
- comm communicator with cartesian structure
- remain_dims 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)
Output from grid sub-division

grid_rank=0, row_rank=0, row_size=3, col_rank=0, col_size=3
grid_rank=1, row_rank=1, row_size=3, col_rank=0, col_size=3
grid_rank=2, row_rank=2, row_size=3, col_rank=0, col_size=3
grid_rank=3, row_rank=0, row_size=3, col_rank=1, col_size=3
grid_rank=4, row_rank=1, row_size=3, col_rank=1, col_size=3
grid_rank=5, row_rank=2, row_size=3, col_rank=1, col_size=3
grid_rank=6, row_rank=0, row_size=3, col_rank=2, col_size=3
grid_rank=7, row_rank=1, row_size=3, col_rank=2, col_size=3
grid_rank=8, row_rank=2, row_size=3, col_rank=2, col_size=3
## Layout of Data/Processes

<table>
<thead>
<tr>
<th>Process Mappings</th>
<th>Process Column 0</th>
<th>Process Column 1</th>
<th>Process Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Row 0</td>
<td>0 (0,0)</td>
<td>1 (0,1)</td>
<td>2 (0,2)</td>
</tr>
<tr>
<td>Process Row 1</td>
<td>3 (1,0)</td>
<td>4 (1,1)</td>
<td>5 (1,2)</td>
</tr>
<tr>
<td>Process Row 2</td>
<td>6 (2,0)</td>
<td>7 (2,1)</td>
<td>8 (2,2)</td>
</tr>
</tbody>
</table>
Pseudo-code for Homework 5

- Set up process grid, row, and column communicators.
- Allocate patch components
  - A, A’,
  - B, B’,
  - Ccomp, Canal
- In Gen_A, Gen_B, Gen_C
  - Identify process grid points (i,j)
  - Compute patch associated with grid points
- Broadcast, Multiply, Roll (do Fox algorithm)
- Compare locally Ccomp and Canal
  - Merge results with global operations
- Time wall time of every step.
  - Merge results, min, max, average
Equivalence Homework #5

• There is no analog to Fox’s algorithm for matrix vector operations which I have chosen as the class of application to do for the equivalence homework.

• The equivalence homework is:
  - $X*Y = Z$
  - Where $X$ is a matrix (rows$_X$, cols$_X$)
  - Where $Y$ is a vector (length$_Y$ = cols$_X$)
  - Where $Z$ is a vector (length$_Z$ = rows$_X$)

• The only similarity in data distribution is that the $X$ matrix is distributed as the $A$, $B$, $C$ matrices would be in the Fox Algorithm.
Distribution of Data for Matrix Vector Distributed Algorithm.
Important features of MatVec

• Note Y is replicated in each process of a column in the processor grid.
  ➢ No communication of Y is needed.
  ➢ Communication of B matrix in Fox’s Algorithm is needed.
    ▪ Uses the column communicator.

• Note a partial solution of Z is computed in each process of a row in the process grid.
  ➢ The process of stages is not needed.
  ➢ Only a global sum of the partial contributions is needed.
    ▪ This is accomplished via an MPI_Allreduce using the row communicator.
The Matvec Computational Kernel

```c
void matvec_distrib(
    double *X, double *Y, double *Ztmp,
    double *Z, int X_ROWS, int X_COLS,
    int prow, int pcol, int pdim,
    MPI_Comm grid_comm,
    MPI_Comm row_comm,
    MPI_Comm col_comm) {

    int LEN_Y, LEN_Z;
    LEN_Y = X_COLS; LEN_Z = X_ROWS;
    matvec_ddot(X,Y,Ztmp,X_ROWS,X_COLS);
    MPI_Allreduce(Ztmp,Z,LEN_Z,
    MPI_DOUBLE,MPI_SUM,row_comm);
}
```
Features of Code Segment

- Ztmp is needed because the MPI_Allreduce function does not allow users to “alias” pointers in the call.
  - Cannot use the same pointer for a buffer containing the data on the root process and the data where it will be stored.
  - All MPI routines prevent the aliasing of pointers.
- This is much simpler than what is needed for Fox’s algorithm for matrix matrix multiplication!!
Fox’s Algorithm

B_source = (prow + 1) % q;
B_destination = (prow + q - 1) % q;
A_size = A_ROWS*A_COLS;  B_size = B_ROWS*B_COLS;
for (stage = 0;stage < q; stage++) {
    a_root = (prow + stage) % q;
    if (a_root == pcol) copy_it(A,Atmp,A_size); /*start with diags*/
    MPI_Bcast(Atmp,A_size,MPI_DOUBLE,a_root,row_comm);
    mat_mul(Atmp,B,C,A_ROWS,A_COLS,B_COLS);
    /* B goes to the destination, B from source is now in Btmp*/
    MPI_Sendrecv(B,   B_size,MPI_DOUBLE,B_destination, 123,
                Btmp,B_size,MPI_DOUBLE,B_source, 123,
                col_comm, &status);
    copy_it(Btmp,B,B_size);  /* kernel multiply uses B so copy it*/
}
The copy_it routine is simple

/*--------------------------------------------------------*
 routine to copy a vector
 */
void copy_it(double *source, double *target, int size)
{
    int i;
    for(i=0; i<size; i++)
        target[i] = source[i];
}

Yes one could use memcpy 😊
Code features

• B matrix shifted in columns using column communicator
  ➢ MPI_Sendrecv sends and receives a message simultaneously.
  ➢ Why not use simple sends and receives?
    ▪ Deadlock; everyone is sending no one is receiving!

• A broadcast and copy
  ➢ Copy of the “A” used at each stage allows a single broadcast call.
  ➢ Note Diagonal elements in each row sent first.
  ➢ Atmp used in the kernel routine.
  ➢ Broadcast uses the row communicator!
MPI_Sendrecv

• Sends and receives a message
  
  ```c
  int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, 
                   int dest, int sendtag, void *recvbuf, int recvcount, 
                   MPI_Datatype recvtype, int source, int recvtag, 
                   MPI_Comm comm, MPI_Status *status )
  ```

• Input Parameters
  
  - `sendbuf` initial address of send buffer (choice)
  - `sendcount` number of elements in send buffer (integer)
  - `sendtype` type of elements in send buffer (handle)
  - `dest` rank of destination (integer)
  - `sendtag` send tag (integer)
  - `recvcount` number of elements in receive buffer (integer)
  - `recvtype` type of elements in receive buffer (handle)
  - `source` rank of source (integer)
  - `recvtag` receive tag (integer)
  - `comm` communicator (handle)

• Output Parameters
  
  - `recvbuf` initial address of receive buffer (choice)
  - `status` status object (Status). This refers to the receive operation
Can we avoid the B copy?

- MPI has a function that does the Send and Receive and does not require the second buffer.
  - MPI_Sendrecv_replace
  - Problems
    - Does not work properly on some systems.
    - As coded with the copy and MPI_Sendrecv works fine.
  - As always Implementations of the Standard vary 😊
    - E.g., your mileage may vary!
MPI_Sendrecv_replace

- Sends and receives using a single buffer
  - int MPI_Sendrecv_replace( void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status )

- **Input Parameters**
  - count number of elements in send and receive buffer (integer)
  - datatype type of elements in send and receive buffer (handle)
  - dest rank of destination (integer)
  - sendtag send message tag (integer)
  - source rank of source (integer)
  - recvtag receive message tag (integer)
  - comm communicator (handle)

- **Output Parameters**
  - buf initial address of send and receive buffer (choice)
  - status status object (Status)
Is there another way to do this?

- We can use non-blocking sends and receives to do this.
- The non-blocking interface for sending messages:
  - `MPI_Isend`
  - `MPI_Irecv`
  - `MPI_Wait`
The change in this line would be:

```c
MPI_Sendrecv(B, B_size, MPI_DOUBLE, B_destination, 123, Btmp, B_size, MPI_DOUBLE, B_source, 123, col_comm, &status);

MPI_Isend(B, B_size, MPI_DOUBLE, B_destination, 123, col_comm, &send_request);
MPI_Irecv(Btmp, B_size, MPI_DOUBLE, B_source, 123, col_comm, &recv_request);
MPI_Wait(&send_request, &status);
MPI_Wait(&recv_request, &status);
```
MPI_Isend

• **Begins a nonblocking send**
  - int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request )

• **Input Parameters**
  - buf initial address of send buffer (choice)
  - count number of elements in send buffer (integer)
  - datatype datatype of each send buffer element (handle)
  - dest rank of destination (integer)
  - tag message tag (integer)
  - comm communicator (handle)

• **Output Parameter**
  - request communication request (handle)
MPI_Irecv

- Begins a nonblocking receive
  - \texttt{int MPI_Irecv( void \*buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request \*request )}

- **Input Parameters**
  - \texttt{buf} initial address of receive buffer (choice)
  - \texttt{count} number of elements in receive buffer (integer)
  - \texttt{datatype} datatype of each receive buffer element (handle)
  - \texttt{source} rank of source (integer)
  - \texttt{tag} message tag (integer)
  - \texttt{comm} communicator (handle)

- **Output Parameter**
  - \texttt{request} communication request (handle)
MPI_Wait

• **Waits for an MPI send or receive to complete**

  ```c
  int MPI_Wait ( MPI_Request *request,
                MPI_Status *status)
  ```

• **Input Parameter**

  - `request`  request (handle)

• **Output Parameter**

  - `status`  status object (Status)