Logistics

• Approved code modules up on the web page.
  ➢ Thread Pool Package (tpool)
  ➢ mymalloc
  ➢ copy_patch
  ➢ accumulate_patch

• Any questions?
Logistics [2]

• MPI in Quinn e.g., reading assignments!
  - Chapter 4
    ▪ Basic MPI Commands
  - Chapter 5
    ▪ Introduces MPI_Bcast with the Sieve of Eratosthenes
  - Chapter 6
    ▪ Introduces MPI_Send and MPI_Recv with Floyd’s Algorithm
    ▪ Good description of deadlock! 6.5.3
  - Chapter 8
    ▪ Introduces gather, scatter, AlltoAll, and sub-communicators for grids
      ♦ Something we talk about!
Matrix Algorithms on MIMD

• Matrices stored in BLOCKS (e.g., patches)
• Blocks are manipulated to get all the elements in the right place at the right time.
• There are many different algorithms.
• A simple one is the Fox algorithm or the BMR (Broadcast Multiply and Roll)
  ➢ We will implement this one in homework #5
  ➢ MPI “helps”
    ▪ Multiple communicators for rows and columns of processes.
BMR or Fox Algorithm

• Assumptions level 1
  ➢ NOT REALITY but the theoretical!
  ➢ Square matrices of order $N$
  ➢ $N_{proc}$ processes where $N_{proc} = N^2$
  ➢ Elements $a_{ij}$, $b_{ij}$, $c_{ij}$ are mapped to processor $i*n + j$
    ▪ This is defined as process $(i,j)$
  ➢ There are $N$ stages of the computation:
    ▪ Each stage computes one dot product term on EACH processor
      ✷ $a_{ik} * b_{kj}$ for all $k$
Stages are “limited”

• At stage 0 on process \((i,j)\)
  \[ c_{ij} = a_{ii} \times b_{ij} \]

• The next stage each process multiplies:
  \(\uparrow\) The element immediately to the right of the diagonal of \(A\) (in it’s process row) by
  \(\uparrow\) The element of \(B\) directly beneath its own element of \(B\) (in it’s process column)

• At stage 1 on process \((i,j)\)
  \[ c_{ij} += a_{i,i+1} \times b_{i+1,j} \]

• At stage \(k\) on process \((i,j)\)
  \[ c_{ij} += a_{i,i+k} \times b_{i+k,j} \]
Limits on k

• You have to worry about the wrap around of k
  ➢ If k = N-1 then k++ is not a valid index in the array

• Therefore at stage k
  ➢ k’ = (i+k) % N (a range of 0 to N-1)
  ➢ c_\text{ij} += a_{i,k’} * b_{k’}^{\text{,j}}
# 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>
## Stage 0

<table>
<thead>
<tr>
<th></th>
<th>PC 0</th>
<th>PC 1</th>
<th>PC 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR 0</td>
<td>Bcast $A_{00}$</td>
<td>Bcast $A_{00}$</td>
<td>Bcast $A_{00}$</td>
</tr>
<tr>
<td></td>
<td>$C_{00} += A_{00}B_{00}$</td>
<td>$C_{01} += A_{00}B_{01}$</td>
<td>$C_{02} += A_{00}B_{02}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{00}$ up</td>
<td>Send $B_{01}$ up</td>
<td>Send $B_{02}$ up</td>
</tr>
<tr>
<td>PR 1</td>
<td>Bcast $A_{11}$</td>
<td>Bcast $A_{11}$</td>
<td>Bcast $A_{11}$</td>
</tr>
<tr>
<td></td>
<td>$C_{10} += A_{11}B_{10}$</td>
<td>$C_{11} += A_{11}B_{11}$</td>
<td>$C_{12} += A_{11}B_{12}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{10}$ up</td>
<td>Send $B_{11}$ up</td>
<td>Send $B_{12}$ up</td>
</tr>
<tr>
<td>PR 2</td>
<td>Bcast $A_{22}$</td>
<td>Bcast $A_{22}$</td>
<td>Bcast $A_{22}$</td>
</tr>
<tr>
<td></td>
<td>$C_{20} += A_{22}B_{20}$</td>
<td>$C_{21} += A_{22}B_{21}$</td>
<td>$C_{22} += A_{22}B_{22}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{20}$ up</td>
<td>Send $B_{21}$ up</td>
<td>Send $B_{22}$ up</td>
</tr>
<tr>
<td></td>
<td>PC 0</td>
<td>PC 1</td>
<td>PC 2</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------</td>
<td>-----------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td><strong>PR 0</strong></td>
<td>$\text{Bcast } A_{01}$</td>
<td>$\text{Bcast } A_{01}$</td>
<td>$\text{Bcast } A_{01}$</td>
</tr>
<tr>
<td></td>
<td>$C_{00} + = A_{01}B_{10}$</td>
<td>$C_{01} + = A_{01}B_{11}$</td>
<td>$C_{02} + = A_{01}B_{12}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{10}$ up</td>
<td>Send $B_{11}$ up</td>
<td>Send $B_{12}$ up</td>
</tr>
<tr>
<td><strong>PR 1</strong></td>
<td>$\text{Bcast } A_{12}$</td>
<td>$\text{Bcast } A_{12}$</td>
<td>$\text{Bcast } A_{12}$</td>
</tr>
<tr>
<td></td>
<td>$C_{10} + = A_{12}B_{20}$</td>
<td>$C_{11} + = A_{12}B_{21}$</td>
<td>$C_{12} + = A_{12}B_{22}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{20}$ up</td>
<td>Send $B_{21}$ up</td>
<td>Send $B_{22}$ up</td>
</tr>
<tr>
<td><strong>PR 2</strong></td>
<td>$\text{Bcast } A_{20}$</td>
<td>$\text{Bcast } A_{20}$</td>
<td>$\text{Bcast } A_{20}$</td>
</tr>
<tr>
<td></td>
<td>$C_{20} + = A_{20}B_{00}$</td>
<td>$C_{21} + = A_{20}B_{01}$</td>
<td>$C_{22} + = A_{20}B_{02}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{00}$ up</td>
<td>Send $B_{01}$ up</td>
<td>Send $B_{02}$ up</td>
</tr>
</tbody>
</table>
Stage2 (the final stage 😊)

<table>
<thead>
<tr>
<th>PR 0</th>
<th>PC 0</th>
<th>PC 1</th>
<th>PC 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bcast $A_{02}$</td>
<td>Bcast $A_{02}$</td>
<td>Bcast $A_{02}$</td>
</tr>
<tr>
<td></td>
<td>$C_{00} += A_{02}B_{20}$</td>
<td>$C_{01} += A_{02}B_{21}$</td>
<td>$C_{02} += A_{02}B_{22}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{20}$ up</td>
<td>Send $B_{21}$ up</td>
<td>Send $B_{22}$ up</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR 1</td>
<td>Bcast $A_{10}$</td>
<td>Bcast $A_{10}$</td>
<td>Bcast $A_{10}$</td>
</tr>
<tr>
<td></td>
<td>$C_{10} += A_{10}B_{00}$</td>
<td>$C_{11} += A_{10}B_{01}$</td>
<td>$C_{12} += A_{10}B_{02}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{00}$ up</td>
<td>Send $B_{01}$ up</td>
<td>Send $B_{02}$ up</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR 2</td>
<td>Bcast $A_{21}$</td>
<td>Bcast $A_{21}$</td>
<td>Bcast $A_{21}$</td>
</tr>
<tr>
<td></td>
<td>$C_{20} += A_{21}B_{10}$</td>
<td>$C_{21} += A_{21}B_{11}$</td>
<td>$C_{22} += A_{21}B_{12}$</td>
</tr>
<tr>
<td></td>
<td>Send $B_{10}$ up</td>
<td>Send $B_{11}$ up</td>
<td>Send $B_{12}$ up</td>
</tr>
</tbody>
</table>

The Fox Element problem

- At an element by element level (e.g., theory)
  - Too much communication relative to computation
    - Sending 2 Doubles
      - Bcast of A element and send of B element
    - Computing 2 floating point operations.
      - one multiply one add
  - $N^2$ processors required
    - Not likely to happen.
    - Too much $\$\$\$\$\$
  - $N^2$ threads is not likely available either 😊
Solution

- **Assumptions**
  - Sub-matrices that are square
    - $N/\sqrt{N_{\text{proc}}}$ by $N/\sqrt{N_{\text{proc}}}$ size
  - Instead of one element an equal number of elements of each matrix!

- **Communication**
  - Still two communications per stage but with larger buffers

- **Computations**
  - $N' = N/\sqrt{N_{\text{proc}}}$ (dimensions of the patch)
  - Operations now $(N')^3$ (in each process, simultaneously)

- **The exact same algorithm holds.**

- **Problems**
  - $N$ must be evenly divisible by $\sqrt{N_{\text{proc}}}$
  - Limits process counts to a perfect square!
    - 4, 16, 25, 36, 49, etc.
  - How do you compute the speedup?
Sub-Matrices (Nproc=4, N=4, N’=4/\sqrt{4} = 2)
Values in Sub-Matrices
Sub-Matrix Values to store

- Each sub-matrix (i,j)
  - $N'$ is $N/\sqrt{N_{proc}}$
  - Has values of $i$ that range from
    - For $i$: $i*N'$ to $i*N'+N'-1$
    - For $j$: $j*N'$ to $j*N'+N'-1$
  - This is a reason that the gen_X routines need to operate on “static” or linear storage not the full matrix.
“Buffering” For Fox’s method

- **On Each Node**
  - Sub-matrix C is fixed doesn’t move
    - The resultant matrix is computed in place!
  - Sub-Matrix A is broadcast to all same row processes.
    - A buffer to “receive” the A of the moment
  - Sub-Matrix B is sent to the “Up” process neighbor in the column set of processes
    - Should use a buffer to do this
    - If you look at the algorithm closely B is moved back to its original place but then send/receive in the same buffer is a race condition.
      - MPI does not allow buffer aliasing

- **Total**
  - A patch, B patch, C patch, A temporary, B temporary
    - All have the same size!
Pseudo-Code for Fox’s Method

- \( i = \) my process row; \( j = \) my process column;
- \( N_{sq} = \sqrt{N_{proc}}; \)
- \( \text{dest} = ((i-1) \mod N_{sq}, j); \)
- \( \text{source} = ((i+1) \mod N_{sq}, j); \)
- \( \text{for}(\text{stage} = 0; \text{stage} < N_{sq} \text{ ; stage}++; \}\) \\
  - \( k' = (i + \text{stage}) \mod N_{sq} \)
  - Broadcast sub-matrix \( A_{i,k'} \) to process row members
  - \( C_{i,j} += A_{i,k'} \ast B_{k',j} \)
  - Send \( B_{k',j} \) to dest
  - Receive \( B_{(k'+1),j} \) from source

- \}
Communicators and Groups

• So far we have talked only about MPI_COMM_WORLD the communicator for the group of all processes.

  ➢ No messages can be sent/received unless the communicator of the sending process is IDENTICAL to the communicator of the receiving process!

  ▪ True for both collective and point-to-point communication operations.
Group and Context

• **A group is an ordered collection of processes.**
  - Consists of Nproc processes,
  - Each process in the group is assigned a unique rank
    - A non-negative integer
      - Range 0 to Nproc-1

• **A context is a system-defined, run-time moiety that uniquely identifies a communicator.**
  - Two different communicator instances will have different contexts, even if they have identical underlying groups.
  - Like a “system-defined” tag that also has to match for message exchanges.
In Fox’s algorithm

- We will use a processor “grid” for the algorithm.
  - $N_{\text{proc}} = q^2$ where $q$ is the “rank” of the processor grid

- We need to define groups and communicators for
  - Rows of the processor “grid”
  - Columns of the processor “grid”

- To create a new communicator we need to use the context of a current communicator to “steal” processes.
Communicators and Groups.

- **Procedure for creating new communicators**
  - Identify current communicator
  - Get group from current communicator
  - Define processes to be in the new group
  - Create the new group
  - Create the new communicator!
Get the current group

• Assuming MPI_COMM_WORLD as the current communicator we get the associated group of processes.

```c
MPI_Group group_world;
MPI_Comm_group(MPI_COMM_WORLD, &group_world);
```

• The MPI_Group group_world identifies all the processes in the world group.
Identify processes in the new group

• Assume grid layout is the same as the layout of C-two-indexed arrays.
• The first row of processes is 0 to \( q \) (sqrt(Nproc))
• So we can simply:

```c
for (j=0; j<q; j++)
    New_ranks[j] = j;
```
Create the new group and Communicator

/* new group */
MPI_Group_incl(group_world, q, New_ranks, &first_row_group);

/* new communicator */
MPI_Comm_Create(MPI_COMM_WORLD, first_row_group, &first_row_comm);

• Now first_row_comm will have a size of the number of elements in the row and each process will have a rank in the new communicator context!
MPI_Comm_group

• **Accesses the group associated with given communicator**

• **Synopsis**
  
  ```
  #include "mpi.h"
  int MPI_Comm_group ( MPI_Comm comm, MPI_Group *group )
  ```

• **Input Parameter**
  
  • `comm` : Communicator

• **Output Parameter**
  
  • `group` : Group in communicator
MPI_Group_incl

- Produces a group by reordering an existing group and taking only listed members

- Synopsis
  - `#include "mpi.h"
  - `int MPI_Group_incl ( MPI_Group group, int n, int *ranks, MPI_Group *group_out )`

- Input Parameters
  - `group` group (handle)
  - `n` number of elements in array ranks [and size of newgroup] (integer)
  - `ranks` ranks of processes in group to appear in newgroup (array of integers)

- Output Parameter
  - `newgroup` new group derived from above, in the order defined by ranks (handle)
MPI_Group_excl

• Produces a group by reordering an existing group and taking only unlisted members

• Synopsis
  ➢ #include "mpi.h"
  ➢ int MPI_Group_excl ( MPI_Group group, int n, int *ranks, MPI_Group *newgroup )

• Input Parameters
  ➢ group group (handle)
  ➢ n number of elements in array ranks (integer)
  ➢ ranks array of integer ranks in group not to appear in newgroup

• Output Parameter
  ➢ newgroup new group derived from above, preserving the order defined by group (handle)
MPI_Comm_create

• Creates a new communicator

• Synopsis
  ➢ #include "mpi.h"
  ➢ int MPI_Comm_create ( MPI_Comm comm, MPI_Group group, MPI_Comm *comm_out )

• Input Parameters
  ➢ comm    communicator (handle)
  ➢ group   group, which is a subset of the group of comm

• Output Parameter
  ➢ comm_out new communicator (handle)
Row and column Communicators

- Using these functions would require a lot of work.
  - Multiple calls to get all groups and communicators.
- MPI offers two other ways to do this.
  - MPI_Comm_split
  - Topologies graphs and grids
MPI_Comm_split

• Creates new communicators based on colors and keys

• Synopsis
  ➢ include "mpi.h"
  ➢ int MPI_Comm_split ( MPI_Comm comm, int color, int key, MPI_Comm *comm_out )

• Input Parameters
  ➢ comm communicator (handle)
  ➢ color control of subset assignment (nonnegative integer)
  ➢ key control of rank assignment (integer)

• Output Parameter
  ➢ newcomm new communicator (handle)
MPI_Comm_split [2]

- Every process that has the same “color” is a part of the group that is associated with the new communicator.
  - Only the processes with the same color thus have the same communicator returned!!
  - Creates more than one communicator at a time.
    - A collective call.

- Simple example
  - color = myrank % 2;
  - if (color)
    - MPI_Comm_split(MPI_COMM_WORLD, color, myrank,&odd_comm);
  - else
    - MPI_Comm_split(MPI_COMM_WORLD, color, myrank,&even_comm);
MPI_Comm_split for Fox Algorithm

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