CIS 631
Parallel Processing

Lecture 7: MPI Programming and Applications

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Acknowledgements

- Portions of the lectures slides were adopted from:
  - Lawrence Livermore National Laboratory, MPI tutorials.
Outline

- Matrix multiplication example
- Global operations
- Non-blocking communication
- Groups, communicators, and contexts
- Matrix multiplication: Cannon’s algorithm
MPI Matrix Multiplication (no index translation)

- $A \times B = C$
- $A[i,:] \cdot B[:,j] = C[i,j]$

```
main(int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    from = (myrank * n)/p;
    to = ((myrank+1) * n)/p;
    /* Data distribution */ ...
    /* Computation */ ...
    /* Result gathering */ ...
    MPI_Finalize();
}
```
MPI Matrix Multiplication: Data Distribution

/* Data distribution - Block by rows */
if( myrank != 0 ) {
    MPI_Recv( &a[from], n*n/p, MPI_INT, 0, tag, 
               MPI_COMM_WORLD, &status );
    MPI_Recv( &b, n*n, MPI_INT, 0, tag, 
               MPI_COMM_WORLD, &status );
} else {
    for( i=1; i<p; i++ ) {
        MPI_Send( &a[from], n*n/p, MPI_INT, i, tag, 
                   MPI_COMM_WORLD );
        MPI_Send( &b, n*n, MPI_INT, i, tag, 
                   MPI_COMM_WORLD );
    }
}
MPI Matrix Multiplication: Computation

/* Computation */

for (i=from; i<to; i++)
    for (j=0; j<n; j++) {
        c[i][j]=0;
        for (k=0; k<n; k++)
            c[i][j] += a[i][k]*b[k][j];
    }
/* Result gathering */
if (mynum!=0)
    MPI_Send( &c[from], n*n/p, MPI_INT, 0,
              tag, MPI_COMM_WORLD);
else
    for (i=1; i<p; i++)
        MPI_Recv( &c[from], n*n/p, MPI_INT,
                   i, tag, MPI_COMM_WORLD,
                   &status);
MPI Matrix Multiplication (index translation)

- Why do you need “rank” indexing on local arrays?
- Suppose want to use “standard” indexing

```c
main(int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    from = (myrank * n)/p;
    to = ((myrank+1) * n)/p;
    /* Data distribution */ ...
    /* Computation */ ...
    /* Result gathering */ ...
    MPI_Finalize();
}
```
/* Data distribution - Block by rows */
if( myrank != 0 ) {
    MPI_Recv( &a, n*n/p, MPI_INT, 0, tag, 
               MPI_COMM_WORLD,&status );
    MPI_Recv( &b, n*n, MPI_INT, 0, tag, 
               MPI_COMM_WORLD,&status );
} else {
    for( i=1; i<p; i++ ) {
        MPI_Send( &a[from], n*n/p, MPI_INT, i, tag, 
                   MPI_COMM_WORLD );
        MPI_Send( &b, n*n, MPI_INT, I, tag, 
                   MPI_COMM_WORLD );
    }
}
MPI Matrix Multiplication: Computation

/* Computation */

for ( i=0; i<n/p; i++ )
    for ( j=0; j<n; j++ ) {
        c[i][j]=0;
        for ( k=0; k<n; k++)
            c[i][j] += a[i][k]*b[k][j];
    }
MPI Matrix Multiplication: Results Gathering

/* Result gathering */
if (myrank!=0)
    MPI_Send( &c, n*n/p, MPI_INT, 0,
              tag, MPI_COMM_WORLD);
else
    for ( i=1; i<p; i++ )
        MPI_Recv( &c[from], n*n/p, MPI_INT,
                   i, tag, MPI_COMM_WORLD,
                   &status);
Owner Computes Rule

- The process that owns the data is the process that will compute new data values
- Ownership is determined by data distribution
  - Domain decomposition
- Ownership may be implicitly determined
- Why is the owner computes rule useful in distributed memory parallel programming?
Buffering Issues

- Where does data go when you send it?
- One possibility is:

- This is not very efficient
  - Three copies in addition to the exchange of data
  - Copies are “bad”
Better Buffering

- Prefer:

  A: [Diagram showing data transfer between processes]

- Requires either
  - MPI_SEND does not return until data delivered
  - Or allow a send to return before completing transfer

- In the latter case, we need to test for completion later
**Blocking vs. Non-Blocking Send and Receive**

- The semantics of blocking/non-blocking has nothing to do with when messages are sent or received
- The difference is when the buffer is free to re-use

- **Blocking**
  - A blocking send routine will only "return" if it is safe to modify the application buffer (your send data)
  - A blocking send is *synchronous* when the receiver confirms a safe send
  - A blocking send is *asynchronous* if a system buffer is used to hold the data for eventual delivery to the receiver
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program
Non-Blocking

- Non-blocking
  - Send and receive routines do not wait for any communication events to complete
    - message copying from user memory to system buffer space
    - actual arrival of message
  - Simply "request" the MPI library to perform the operation when it is able, but cannot predict when
  - It is unsafe to modify the application buffer until you know the non-blocking operation was actually performed
    - use "wait" routines to do this
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.
Send-Receive

- Send-receive operation combine in one function call the sending of a message to one process and receiving of a message from another
  - `MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)`

- Use same buffer for sending and receiving
  - `MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)`

- Exchange data with neighbors
  - `MPI_Sendrecv(&a,n,MPI_DOUBLE,right,TAG,&b,n,MPI_DOUBLE,left,TAG,MPI_COMM_WORLD,&Status);`
Non-Blocking MPI Send and Receive Operations

- Saw how non-blocking operations help to avoid deadlock
- Important for overlapping communication / computation
- Non-blocking send and receive routines
  - MPI_Isend(buf, count, datatype, dest, tag, comm, request)
  - MPI_Irecv(buf, count, datatype, source, tag, comm, request)
- request used as an argument in test and wait routines
MPI Test and Wait

- Routines to test status of non-blocking send and receive
  - MPI_Test(request, flag, status)
  - MPI_Wait(request, status)

- MPI_Test tests whether or not the non-blocking operation identified by request has finished
  - Returns true or false in flag
  - If true, request object deallocated

- MPI_Wait blocks until the non-blocking operation identified by request completes
  - request object deallocated on return
Waiting on Several Completions

- It is often desirable to wait on multiple requests
  - MPI_Waitall(count, request_array, status_array)
  - MPI_Waitany(count, requests_array, index, status)
  - MPI_Waitsome(incount, requests_array, outcount, indices_array, status_array)

- There are corresponding versions of test for each of these
# Exchanging Data with Neighbors

<table>
<thead>
<tr>
<th></th>
<th>BlockingRecv</th>
<th>Non-blockingRecv</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blocking Send</strong></td>
<td><code>MPI_Send(A,...,left,...);</code></td>
<td><code>MPI_Send(A,...,left,...);</code></td>
</tr>
<tr>
<td></td>
<td><code>MPI_Recv(B,...,right,...);</code></td>
<td><code>MPI_Irecv(B,...,right,..., &amp;req);</code></td>
</tr>
<tr>
<td></td>
<td><code>MPI_Wait(&amp;req,&amp;status);</code></td>
<td><code>MPI_Wait(&amp;req,&amp;status);</code></td>
</tr>
<tr>
<td><strong>Non-blocking Send</strong></td>
<td><code>MPI_Isend(A,...,left,..., &amp;req);</code></td>
<td><code>MPI_Isend(A,...,left,..., &amp;req1);</code></td>
</tr>
<tr>
<td></td>
<td><code>MPI_Recv(B,...,right...);</code></td>
<td><code>MPI_Irecv(B,...,right..., &amp;req2);</code></td>
</tr>
<tr>
<td></td>
<td><code>MPI_Wait(&amp;req,&amp;status);</code></td>
<td><code>MPI_Wait(&amp;req1,&amp;status);</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>MPI_Wait(&amp;req2,&amp;status);</code></td>
</tr>
</tbody>
</table>
**Probe and Cancel**

- **MPI_Probe()** and **MPI_Iprobe()** allow incoming messages to be checked for, without receiving them
  - **MPI_Probe(source, tag, comm, status)**
  - **MPI_Iprobe(source, tag, comm, flag, status)**

- **MPI_Cancel()** cancels pending communication
  - **MPI_Cancel(request)**
Persistent Communication Requests

- MPI communication functions called repeatedly inside a loop
  
  ```c
  for(i=0; i<nsteps; i++){
    MPI_Send(&a, n, MPI_DOUBLE, right, TAG, MPI_COMM_WORLD);
    MPI_Recv(&b, n, MPI_DOUBLE, left, TAG, MPI_COMM_WORLD, &status);
    ...
  }
  ```

- Persistent requests reduce overhead of creating messages

  ```c
  MPI_Send_init(&a, n, MPI_DOUBLE, right, TAG,
                MPI_COMM_WORLD, &req1);
  MPI_Recv_init(&b, n, MPI_DOUBLE, left, TAG,
                MPI_COMM_WORLD, &req2);
  
  for(i=0; i<nsteps; i++){
    MPI_Start(&req1); MPI_Start(&req2);
    MPI_Wait(&req1, &status);
    MPI_Wait(&req2, &status);
    ...
  }
  ```
Communication Modes

- Safety: A program is safe if it will produce correct result even if the system provides no buffering
- MPI offers the choice of four communication modes
  - Standard:
    - MPI_Send, MPI_Isend, MPI_Send_init
  - Buffered:
    - MPI_Bsend, MPI_Ibsend, MPI_Bsend_init
  - Synchronous:
    - MPI_Ssend, MPI_Issend, MPI_Ssend_init
  - Ready:
    - MPI_Rsend, MPI_Irsend, MPI_Rsend_init
- There is only one receive mode
Buffered Mode

- A send can complete without matching receive
- The system must provide buffering
  - `MPI_Bsend(buf, count, datatype, dest, tag, comm)`
  - `MPI_Ibsend(buf, count, datatype, dest, tag, comm, request)`
  - `MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request)`
Synchronous Mode

- Send can start before a matching receive has been posted
- Won’t complete until matching receive has been posted and the matching receive has begun reception of the data
  - `MPI_Ssend(buf, count, datatype, dest, tag, comm)`
  - `MPI_Issend(buf, count, datatype, dest, tag, comm, request)`
  - `MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request)`
Ready Mode

- A send may start only if the matching receive is posted
- Otherwise, the operation is erroneous and outcome is undefined
  - MPI_Rsend(buf, count, datatype, dest, tag, comm)
  - MPI_Isend(buf, count, datatype, dest, tag, comm, request)
  - MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request)
**MPI Global Operations**

- Often, it is useful to have one-to-many or many-to-one message communication.
- This is what MPI’s global operations do
  - MPI_Barrier
  - MPI_Bcast
  - MPI_Gather
  - MPI_Scatter
  - MPI_Reduce
  - MPI_Allreduce
Barrier

- **MPI_Barrier(comm)**
  - Global barrier synchronization
  - All processes in communicator wait at barrier
  - Release when all have arrived
Broadcast

- MPI_Bcast(inbuf, incnt, intype, root, comm)
  - inbuf: address of input buffer on root
  - inbuf: address of output buffer elsewhere
  - incnt: number of elements
  - intype: type of elements
  - root: process id of root process
Before Broadcast

Before the broadcast, the inbuf is accessible to all processes.

- **proc0**
- **proc1**
- **proc2**
- **proc3**

All processes should have access to the root node to ensure data propagation.

**Diagram Notes:**
- Proc0 has a unique identifier and color.
- Proc1, Proc2, and Proc3 are connected to the root node, indicating their accessibility to the root node.

This setup ensures that the broadcast can be effectively managed among all processes.
After Broadcast

proc0  proc1  proc2  proc3

root

inbuf
MPI Scatter

- `MPI_Scatter(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)`
  - `inbuf`: address of input buffer
  - `incnt`: number of input elements
  - `intype`: type of input elements
  - `outbuf`: address of output buffer
  - `outcnt`: number of output elements
  - `outtype`: type of output elements
  - `root`: process id of root process
Before Scatter

proc0

proc1

proc2

proc3

root

inbuf

outbuf
After Scatter

proc0  proc1  proc2  proc3

inbuf  outbuf

root
### MPI Gather

- **MPI_Gather**(inbuf, incnt, intype,
  }
  outbuf, outcnt, outtype, root, comm)
- **inbuf**: address of input buffer
- **incnt**: number of input elements
- **intype**: type of input elements
- **outbuf**: address of output buffer
- **outcnt**: number of output elements
- **outtype**: type of output elements
- **root**: process id of root process
Before Gather

- proc0
- proc1
- proc2
- proc3

root

inbuf
outbuf
After Gather

root

proc0

proc1

proc2

proc3

inbuf

outbuf
Broadcast / Gather / Scatter

- These three primitives combine sends and receives
  - May be confusing
  - Collective operations

- Perhaps un-intended consequence
  - Requires global agreement on layout of array
Groups, Communicators, and Contexts

- A group is an ordered set of processes
  - Each process in a group has a unique integer rank
  - Rank values are from 0 to N-1, where N is the group size
  - A group is accessible to the programmer by a "handle"

- A communicator is a group of processes that are allowed to communicate between themselves
  - All MPI messages must specify a communicator
  - Communicators accessible to programmer by "handles"

- A context is a system-defined object that uniquely identifies a communicator
More Groups and Communicators

- Groups/communicators are dynamic
- Processes may be in more than one group/communicator
  - They will have a unique rank within group/communicator
- Typical usage:
  1. Extract handle using `MPI_Comm_group`
  2. Form new group using `MPI_Group_incl`
  3. Create new communicator using `MPI_Comm_create`
  4. Determine new rank in communicator using `MPI_Comm_rank`
  5. Conduct communications using any MPI routine
  6. Free up communicator and group
Logical View

MPI_COMM_WORLD

- Group1
  - Comm1
    - Communications
  - Comm2

- Group2
  - Comm1
  - Comm2
Group Constructors

- **MPI_Comm_group(comm, group)**
  - Returns in `group` a handle to the group of `comm`

- **MPI_Group_union(group1, group2, newgroup)**
  - Returns a group including all elements of the first group (`group1`), followed by all elements of second group (`group2`) not in first.

- **MPI_Group_intersection(group1, group2, newgroup)**
  - Returns a group including elements of the first group that are also in the second group, ordered as in first group.

- **MPI_Group_difference(group1, group2, newgroup)**
  - Returns a group including all elements of the first group that are not in the second group, ordered as in the first group.
Group Constructors

- **MPI_Group_incl(oldgroup, n, ranks, newgroup)**
  - Creates a group newgroup that consists of the n processes in oldgroup with ranks rank[0], ..., rank[n-1]; the process with rank[i] in newgroup is the process with rank ranks[i] in the oldgroup.

- **MPI_Group_excl(oldgroup, n, ranks, newgroup)**
  - Creates a group newgroup that is obtained by deleting from oldgroup those processes with ranks ranks[0], ..., ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in oldgroup.
Communicator Constructors

- **MPI_Comm_create(oldcomm, group, newcomm)**
  - Creates a new communicator `newcomm` from a group `group` in communicator defined by `oldcomm`.

- **MPI_Comm_split(oldcomm, color, key, newcomm)**
  - Partitions the group associated with `oldcomm` into disjoint subgroups, one for each value of `color`.
  - Each subgroup contains all processes of the same color.
  - Within each subgroup, the processes are ranked in the order defined by the value of the argument `key`, with ties broken according to their rank in the old group.
  - A new communicator is created for each subgroup and returned in `newcomm`. 
Predefined Communicator

- **MPI_COMM_WORLD**
  - Contains all processes available at start of the program
- **MPI_COMM_NULL**
  - An invalid communicator
- **MPI_COMM_SELF**
  - Contains only the local process
- **MPI_COMM_EMPTY**
  - There is no such thing as **MPI_COMM_EMPTY**
  - (Why not?)
Split Example

MPI_Comm my_row_comm;
int my_row;
my_row = my_rank / 3;

MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank, &my_row_comm);
Virtual Topologies

☐ What Are They?
  ☐ A mapping/ordering of MPI processes into a geometric "shape"
  ☐ Two main MPI supported topologies are Cartesian (grid) and Graph

☐ MPI topologies are virtual
  ☐ There may be no relation between the physical structure of the parallel machine and the process topology.

☐ Virtual topologies are built upon MPI communicators and groups
  ☐ Must be "programmed" by the application developer
Why use Virtual Topologies?

- Useful for specific communication patterns
- Simplified programming
- Communication efficiency
- Particular implementation may optimize process mapping based upon physical characteristics of parallel machine
- Mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored
Cartesian Virtual Topology

- `MPI_Cart_create(comm_old, ndims, dims, wrap, reorder, comm_cart)`
  - Returns a handle to a new communicator to which the Cartesian topology information is attached
  - If `reorder=false`, then the rank of each process in the new group is identical to its rank in the old group
  - Otherwise, the function may reorder the processes
  - `ndims`: number of dimension
  - `dims`: dimension sizes
  - `wrap_around`: whether each dimension is circular
Graph Topology

- **MPI_Graph_create**(comm_old, nnodes, index, edges, reorder, comm_graph)
  - Returns a new communicator to which the graph topology information is attached
  - If reorder=false, then the rank of each process in the new group is identical to its rank in the old group
  - Otherwise, the function may reorder the processes
  - **nnodes**: number of nodes
  - **index**: the ith entry indicates the position of the node list in edges containing all nodes that are connected to the ith node
  - **edges**: a flatten list of edges represented by the connected nodes
Example of Creating a Graph

- Process neighbors
  - 0: 1, 2, 3
  - 1: 0
  - 2: 1, 3
  - 3: 0, 2

- Input arguments:
  - nnodes: 4
  - index: 3, 4, 6, 8
  - edges: 1, 2, 3, 0, 1, 3, 0, 2
Matrix Multiplication: Cannon’s Algorithm

- Step 1: Initially processor P(i,j) has submatrices A(i,j) and B(i,j)
- Step 2: Submatrices are moved from their initial position to an “aligned” position. The complete i-th row of A is shifted i places left, and the complete i-th column of B is shifted i places upwards. All shifts are wraparound.
Matrix Multiplication: Cannon’s Algorithm

- Step 3: Each P(i,j) processor multiplies its elements and adds sum to C(i,j)
- Step 4: Every row of A is shifted one place left, and every column of B is shifted one place upwards
- Repeat step 3 and 4 until the final result is obtained

```
A(0,1)  A(0,2)  A(0,3)  A(0,0)
B(1,0)  B(2,1)  B(3,2)  B(0,3)
A(1,2)  A(1,3)  A(1,0)  A(1,1)
B(2,0)  B(3,1)  B(0,2)  B(1,3)
A(2,3)  A(2,0)  A(2,1)  A(2,2)
B(3,0)  B(0,1)  B(1,2)  B(2,3)
A(3,0)  A(3,1)  A(3,2)  A(3,3)
B(0,0)  B(1,1)  B(2,2)  B(3,3)
```
## Implementing Cannon’s Algorithm

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

int main(int argc, char* argv[]){
    int n,size,my_rank,remote_rank;
    int i,j;
    int a,b,c;
    int left,right,up,down;
    int dimensions[2],wrap_around[2],coordinates[2];
    MPI_Status status;
    MPI_Comm cart_comm;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    for(n=1;n*n<size;n++);
    if(n*n!=size) {
        printf("size %d is not a square.\n",size);
        MPI_Finalize();
        return 1;
    }

    // More code here...

    return 0;
}
```
Implementing Cannon’s Algorithm (continued)

dimensions[0]=dimensions[1]=n;
wrap_around[0]=wrap_around[1]=1;
MPI_Cart_create(MPI_COMM_WORLD,2,dimensions,wrap_around,1,&cart_comm);
MPI_Comm_rank(cart_comm,&my_rank);
MPI_Cart_coords(cart_comm,my_rank,2,coordinates);
MPI_Cart_shift(cart_comm,1,1,&left,&right);
MPI_Cart_shift(cart_comm,0,1,&up,&down);
a=b=coordinates[0]+coordinates[1];

for(i=0;i<coordinates[0];i++)
  MPI_Sendrecv_replace(&a,1,MPI_INT,left,0,right,0,cart_comm,&status);
for(i=0;i<coordinates[1];i++)
  MPI_Sendrecv_replace(&b,1,MPI_INT,up,0,down,0,cart_comm,&status);

for(c=0,i=0;i<n;i++)
  c+=a*b;
  MPI_Sendrecv_replace(&a,1,MPI_INT,left,0,right,0,cart_comm,&status);
  MPI_Sendrecv_replace(&b,1,MPI_INT,up,0,down,0,cart_comm,&status);
}
Implementing Cannon’s Algorithm (continued)

```c
if (my_rank==0)
{
    for (remote_rank=0, i=0; i<n; i++)
    {
        for (j=0; j<n; j++, remote_rank++)
        {
            if (remote_rank!=0)
                MPI_Recv(&c,1,MPI_INT,remote_rank,0,cart_comm,&status);
            printf("%d ",c);
        }
        printf("\n");
    }
}
else
    MPI_Send(&c,1,MPI_INT,0,0,cart_comm);

MPI_Finalize();

return 0;
}```
MPI Parallelization Process

- Divide program in parallel parts
- Create and destroy processes to do above
- Partition and distribute the data
- Communicate data at the right time
- (Sometimes) perform index translation
- Still need to do synchronization?
  - Sometimes, but many times goes hand in hand with data communication
Next Class

- Shared memory parallel programming
- Threads