Parallel Programming and MPI- Lecture 3

Abhik Roychoudhury
CS 3211
National University of Singapore

Sample material: Parallel Programming by Lin and Snyder, Chapter 7.

Summary of previous lectures

- MPI as a programming interface
- Message passing communication
  - Communicating sequential processes
- Entering and Exiting MPI
  - MPI_Init, MPI_Finalize
- Point-to-point communication
  - Blocking & Non-blocking
  - MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv
  - Wait and test operations to complete communication.
- Collective communication

In today's discussion

- Managing communicators in MPI
  - Defines a communication domain.
  - Used implicitly several times in our discussion in defining the communication primitives.

Why need communicators?

- Scenario: Weather forecasting application
  - 60% of the processes are predicting weather based on previously available weather data.
  - *In parallel*, 40% of the processes are doing the initial processing of the new data that is arriving.
  - Can put these groups into 2 separate communicators!
  - Any interesting management of communicators?
    - Example: when no new weather data is available, we might want to use 100% of the processes for weather prediction?

Size and rank

- int MPI_Comm_size(comm, &size);
  - # of processes in the communicator
- int MPI_Comm_rank(comm, &rank);
  - Rank of the process that calls it
  - In the range 0…size-1
- There is a pre-defined communicator
  - MPI_COMM_WORLD

So, what is a communicator?

- A group is an ordered set of processes.
- A communicator is a handle to a group of processes.
- A communicator thus defines a communication domain.
- Even for the same group of processes <p1,…,pN>, it might be convenient to describe disparate communication domains containing the same group of processes.
  - Why?
  - To separate library code execution from user code execution.
  - A send in library may be received by a receive in user code.
  - This can be prevented by making the library and user code operate in different communication domains!

Simple Mismatch scenario

- Suppose the user's code posts a non-blocking receive Irecv, before entering a library routine.
  - The first send in the library may be received by the user's posted receive.
  - This will cause the library to fail.
- Solution:
  - maintain separate communicators, as mentioned earlier.

Intra- and Inter-communicators

Intra-communicator

For communication within a group of processes.

Inter-communicator

For point-to-point communication between disjoint groups of processes.

Can we ignore communicators?

- There is a single global communicator
  - MPI_COMM_WORLD
  - Contains all processes.
  - We can only work with this one.
- However, it may be advantageous to separate out certain communications, to prevent executions with arbitrary send-receive matching!

Creating communicators

- int MPI_Comm_dup(comm, newcomm)
  - MPI_Comm comm
  - MPI_Comm *newcomm
  - Creates a new communicator with the same group of processes.
- int MPI_Comm_create(comm, group, newcomm)
  - MPI_Comm comm
  - MPI_Group group
  - MPI_Comm *newcomm
  - The argument group must be a subset of the group of comm
  - Always possible to use, with MPI_COMM_WORLD

Exercise

- We are trying to define a parallel library which does multi-cast (a variant of MPI_Bcast)
  - Differences between MPI_Bcast and our library
    - Instead of the root process in MPI_Bcast, the function takes a flag which is true if the calling process is root, and false otherwise.
    - All processes do not need to provide the id of the root process.
- Signature of MPI_Bcast
  - Int MPI_Bcast(buffer, count, datatype, root, comm)
    - Starting address of buffer
    - # of entries in buffer
    - Data type of buffer
    - Rank of the broadcasting process
    - The communicator capturing the group of processes.

Exercise

- Signature of mcast
  - Mcast(buf, count, type, isroot, comm)
    - Output buffer at root, input buffer at other processes
    - Number of items to be broadcast
    - Type of items to be broadcast
    - Flag saying whether the process is a root
    - Communicator.
- Algorithm
  - Uses a broadcast tree which is built dynamically.
  - Root divides the sequence of processes into 2 segments
    - Sends a message to 1st proc p in 2nd seg, p becomes root of 2nd seg
    - The procedure is repeated recursively within each sub-segment.

Example Multi-cast library

```
void mcast(void *buff, int count, MPI_Datatype type,
                int isroot, MPI_Comm comm)
{
   MPI_Comm_size(comm, &size);
   MPI_Comm_rank(comm, &rank);

    int numleaves,  /*number of leaves in broadcast tree */
        childleaves,   /*number of leaves in child's broadcast tree */
        child;   /* rank of current child in broadcast tree */
    if  (isroot){
          numleaves = size – 1;
    }
    else{   …
```


Example Multi-cast library

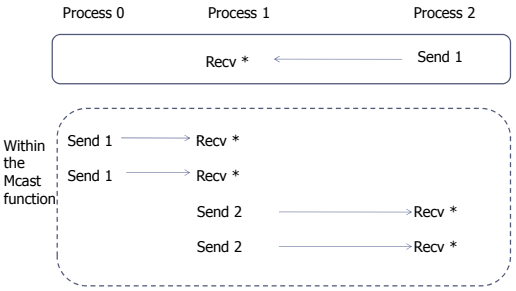
```
  else {  /* not a root process , receive leaf-  count and message from parent */
       MPI_Recv(&numleaves, 1,MPI_INT,MPI_ANY_SOURCE,0, comm, status);
       MPI_Recv(buff, count, type, MPI_ANY_SOURCE , 0, comm, status);
}
   while (numleaves > 0){
      /* pick child in the middle of current leaf processes */
      child = (rank + (numleaves + 1)/2 ) % size;
      childleaves = numleaves / 2;
     /* send leaf count and message to child */
      MPI_Send(&childleaves, 1, MPI_INT, child, 0, comm);
      MPI_Send(buff, count, type, child, 0);
      numleaves -= (childleaves + 1); /* remaining number of leaves */
    }
}
```


Now, consider the following code

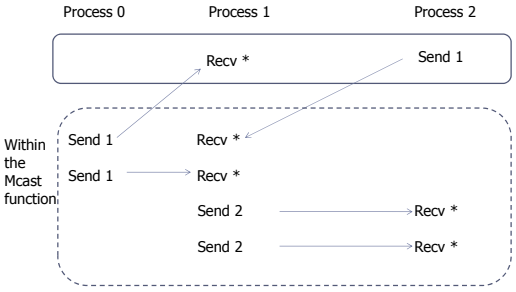
- Assume a group of 3 processes.

```
MPI_Comm_rank( comm,  &myrank );
if (myrank == 2){
    MPI_Send( …,1, MPI_INT,  1,  0, comm);
  } else if (myrank == 1){
        MPI_Recv(…,1, MPI_INT, MPI_ANY_SOURCE, 0, comm)
}
mcast(…,1, MPI_INT,  (myrank == 0), comm);
```


"Expected" Behavior



Possible & "Unexpected" Behavior



How can this happen?

- Invocation of mcast in the 3 processes is not simultaneous.
- Process 0 starts executing multi-cast earlier than other processes.
  - The processes are executing on different processors after all.
  - Different processors run at different speeds!
- Process 1 executes the MPI_Recv in the caller code
  - This matches with the first MPI_Send of process 0 executed inside the mcast library!
- This is why separate communicators are needed!

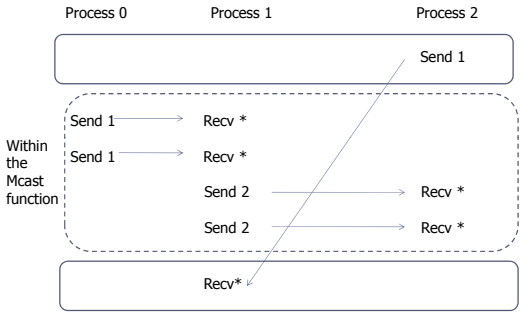
Solutions to the "problem"

- Call mcast as "synchronized" code
  - Is this a wise choice?
    - Unnecessary synchronization overhead.
    - Assumes certain ``well-formed" structure in the code- the code should obey the convention that --- *Messages sent before collective invocation (such as that of mcast) should also be received at the destination before the matching invocation.*
    - Is it reasonable to assume this?

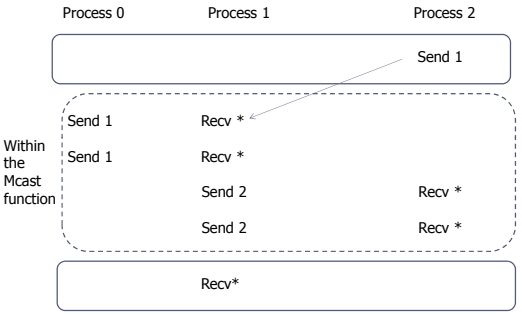
A more complex scenario

- MPI_Comm_rank(comm, &myrank);
- if (myrank == 2){
  - MPI_Send(…,1, MPI_INT, 1, 0, comm);
- }
- mcast(…,1, MPI_INT, (myrank == 0), comm);
- if (myrank == 1){
  - MPI_Recv(…,1, MPI_INT, MPI_ANY_SOURCE, 0, comm)
- }

The "expected" behavior

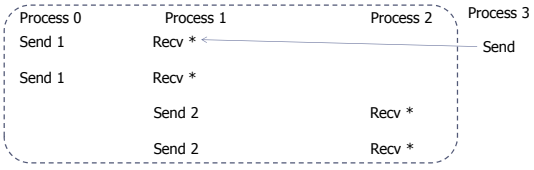


The "unexpected" behavior



Another scenario

- comm = {0,1,2}



what if process 3 sends msg to process 1,2?

- Gets matched with the receives within multi-cast?
- Solution:
  - Use a diff. communicator within the mcast function.

Using different communicators

```
void mcast(void *buf, int count, MPI_Datatype type,
           int isroot, MPI_Comm comm)
{
  int size, rank, numleaves,child, childleaves;
  MPI_Status status;
  MPI_Comm pcomm;  /* private communicator */

  MPI_Comm_dup(comm, &pcomm);
  MPI_Comm_size(pcomm, &size);
  MPI_Comm_rank(pcomm, &rank);

  /* dynamically build up a broadcast tree now */
```

Using different communicators

```
if (isroot){
    numleaves = size -1;
} else {  /* receive from parent */
      MPI_Recv(&numleaves, 1,MPI_INT,MPI_ANY_SOURCE,0,pcomm, &status);
      MPI_Recv(buf, count, type,MPI_ANY_SOURCE,0,pcomm, &status);
}
while (numleaves >0){
    child = (rank + (numleaves+1)/2)%size;
    childleaves = numleaves/2;    /* send to  child in the next 2 lines */
    MPI_Send(&childleaves,1, MPI_INT, child, 0, pcomm);
    MPI_Send(buf, count, type, child, 0, pcomm);
    numleaves -= (childleaves +1);  /* compute remaining number of leaves */
}
MPI_Comm_free(&pcomm);
}
```


Exercise

- Can there be other solutions which avoid the additional communicator allocation (pcomm)?
  - How about inserting a barrier at the beginning and at the end of the mcast function?
  - Can this solution be consistently employed for any parallel library?
  - What are the implications on
    - Performance?
    - Correctness?
      - Try out the communication scenarios we discussed earlier.

Wrapping up

- MPI programming
  - Explicit message passing, as opposed to shared memory.
- Important concepts
  - Point to point communication
    - Blocking send receives --- MPI_Send, MPI_Recv
    - Non-blocking send receives --- MPI_Isend, MPI_Irecv
  - Collective communication
    - Scatter, Gather
    - MPI_Reduce
  - Communicators
    - The default communicator is MPI_COMM_WORLD