

## Lecture 1: February 13

*Lecturer: Guest Speaker: Johnathan Carter**Scribes: Jia Zou*

**Disclaimer:** *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

Lecture focus: Programming Distributed Memory Machines with Message Passing

## 1.1 MPI Generals

10 years ago a lot of message passing libraries were around. Today only MPI is really being used, and nearly all vendors support it.

All communications are done through making library calls to MPI.

Novel Features of MPI:

- Communicators: Make sure all messages within one library is encapsulated in itself.
- Asynchronous, synchronous communicates modes are available.
- Ability to define topologies, such as 2D grid, 3D grid. MPI allow you to map these topologies to the machines.

Side note: Many books and references are available for MPI

## 1.2 MPI Details

General notes about MPI:

- Each process executes independently from each other.
- MPI doesn't give you a way to test if the message passing was successful or no.
- The datatypes of the sending and the receiving calls need to be the same.

How many processes are participating in the computation?

- `MPI_Comm_size` reports the number of processes, `MPI_Comm_rank` identifies your process.
- All MPI programs start with `MPI_Init()` and end with `MPI_Finalize()`.
- `rank()` and `size()` tells you about which one are you among the processes.

### 1.2.1 MPI Datatypes

- MPI datatypes can be recursively defined.
- A lot of support is available for datatypes. However MPI libraries have optimizations for simple datatypes, but not complex ones.
- MPI tags allow the user to order the messages.
- `status` is a data structure containing value of the tag and the value of the source. This is useful if `wild-card recv` is used, and the `status` structure would tell you where the message came from. `wild-card recv` will be defined later in the notes...

### 1.2.2 MPI function calls

- Different processes can be collected into groups. `MPI_COMM_WORLD` is all the processes within the system.
- MPI has more than 200 functions, 6 are most useful: `MPI_INIT`, `MPI_FINALIZE`, `MPI_COMM_SIZE`, `MPI_COMM_RANK`, `MPI_SEND`, `MPI_RECV`
- MPI Basic Send call is `MPI_SEND`, in which `start`, `count`, `datatype`, `destination`, `tag`, and `communicator` are the arguments in this call.
- `MPI_RECV` is almost the same as the `SEND`. `RECV` provides a buffer with length of "count". the count needs to be larger than the size of the message sent, otherwise an error occurs. User could also define who the sender is, or receive from any user.
- Function calls such as `wild-card recv` are also available, which means that the receiver can receive from all other processes.

There's also a widely used set, that are collective functions, which means everyone calls collective calls at the same time. Data is distributed from one-to-many, or many-to-many at the same time.

Examples of Collective Functions:

- `BCAST`(BroadCast) distributes data from one process to all others in the communicator.
- `REDUCE`: gets data from lots of processes. `REDUCE` allow all other processes to send to source at the same time. Also `REDUCE` allow the data to be manipulated.

### 1.2.3 MPI Collectives

There are no non-blocking collectives. When the sending/receive returns, the message passing is for sure to be completed.

These classes of operations are available:

- Synchronization:
  - `MPI_Barrier`: blocks until all processes in the group of the communication `comm` call it.

However most times these barriers are not useful at all. Usually synchronization is enough with send/receives.

- data movement:
  - Broadcast: to all
  - Scatter: distribute to all different processes.
  - Gather: Get from all processes and gather to one processes.
  - Allgather: gather messages in all processes to all processes.
  - Alltoall: Take a distinctive piece of data in each process, and take these pieces to one process. The complexity is increasing in these MPI calls.
- Collective Computation:
  - Reduce: Data is combined while receiving from all processes
  - Scan: Data is collected and then distributed by a hierarchy.
  - Operations available: max, min, prod, sum, logical and, logical or, logical xor, binary and, binary or, binary xor, etc.

Collective calls start with All deliver results to all processes.

### 1.2.4 Communication Modes

Different communication modes can be used to test if the system deadlocks, also allows fast access to protocols, these might be useful or not depending on the application.

Note: MPI\_Recv receives all messages sent in different modes.

### 1.2.5 MPI Communicators

Motivation: The MPI implementation should have the smarts to put the processes onto the physical hardware.

Topologies map a communicator onto different processor grid, such as 3D Cartesian processor grid, etc.

### 1.2.6 Using MPI for I/O Purposes

Rich set of I/O functions are available.

One-sided communication is for loser communication model, where a one-sided call allow the data to be transmitted unilaterally.

Note: This standard is not very widely used because the synchronization has to be taken care of by the programmer. It is a bit difficult to understand.

### 1.3 Problem in MPI Usages

- Has to do with buffering and deadlocks.
- If the message is very long, then both sending and receiving processes need to buffer the data, and that's very expensive. This can be solved if the network is directly connected to the user data of process 0 and 1. But this implies process 0 needs to wait for process 1 to receive.
- If both processes are sending at the same time to each process, then both process ends up waiting for the other process's buffer to empty. This results in deadlock.
- Another possibility of deadlock is for both process to wait to receive from each other.
- One option is to order these processes more carefully. But for complex algorithm this might be difficult to do.
- Another solution is sendrecv, provided by MPI, which means a receive buffer is supplied at the same time as send.
- Another solution is to use non-blocking send/receives. It's like asynchronous I/O.
- non-blocking operations are triggered by a request.

### 1.4 Final Example

The standard does not specify how message passings are implemented under the covers.

This is how a regular send is implemented:

1. Initiate send
2. Address translation on destination port.
3. send-ready request
4. Remote check for posted receive
5. Reply transaction
6. Bulk data transfer