Recitation 9

Laila Elbeheiry

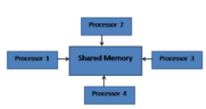
March 17, 2022

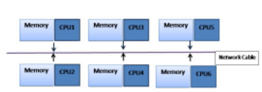
Logistics

- P2 done (almost)
- P3 out (discussion next week)
- No office hours next week

Parallel Programming Models

- Shared Memory Model
- Message Passing Model





Parallel Programming Models

Shared Memory	Message Passing

Parallel Programming Models

Shared Memory	Message Passing
Communicating processes usually reside on the same machine	Typically used in a distributed environment where communicating processes reside on remote machines connected through a network.
Faster communication strategy.	Relatively slower communication strategy
More difficult to synchronize	Easier to synchronize
Example: OpenMP	Example: MPI

What is MPI?

- Message Passing Interface
- Defines a set of API declarations on message passing (such as send, receive, broadcast, etc.), and what behavior should be expected from the implementations.
- The *de-facto* method of writing message-passing applications
- Applications can be written in C, C++ and calls to MPI can be added where required

MPI Program Skeleton

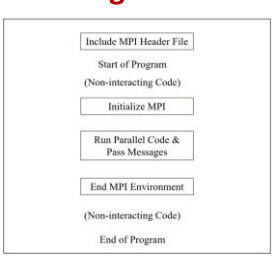


Photo credits:
https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf

MPI Program Skeleton

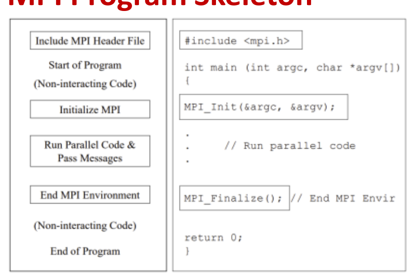
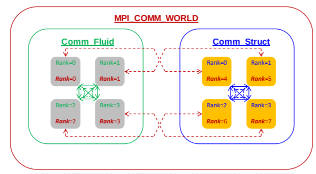


Photo credits:
https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf

MPI Concepts

- *Communicator*
  - Defines which collection of processes may communicate with each other to solve a certain problem
  - In this collection, each process is assigned a unique *rank*, and they explicitly communicate with one another by their ranks.
  - When an MPI application starts, it automatically creates a communicator comprising all processes and names it MPI_COMM_WORLD
- *Rank*
  - Within a communicator, every process has its own unique ID referred to as *rank*
  - Ranks are used by the programmer to specify the source and destination of messages

MPI Concepts



✓Ranks within MPI_COMM_WORLD are printed in red
✓Ranks within Comm_Fluid are printed in green
✓Ranks within Comm_Struct are printed in blue

MPI Concepts

`MPI_Init(int *argc, char ***argv)`	• Initialize the MPI library (must be the first routine called)
`MPI_Comm_rank(comm, &rank);`	• Returns the rank of the calling MPI process within the communicator, comm • `MPI_COMM_WORLD` is set during `Init(…)` • Other communicators can be created if needed
`MPI_Comm_size(comm, &size)`	• Returns the total number of processes within the communicator, comm

Let’s write our first MPI program…

MPI Send and Recv

```
MPI_Send(void *buf , int count , MPI_Datatype datatype , int dest ,
         int tag , MPI_Comm comm )
```

- The first argument is the data buffer
- The second and third arguments describe the count and type of elements that reside in the buffer
- MPI Datatype is very similar to a C datatype: MPI_INT, MPI_CHAR
- The fourth and fifth arguments specify the rank of the sending/receiving process and the tag of the message

Why do we need a tag?

- The sixth argument specifies the communicator

```
MPI_Recv( void *buf, int count, MPI_Datatype datatype, int src, int tag,
MPI_Comm comm, MPI_Status *status )
```

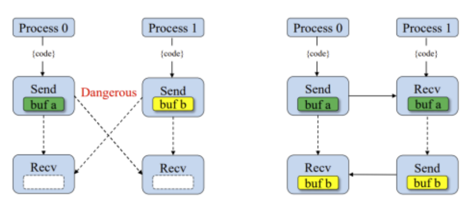
Let's look at some parallel programs

Point-to-Point Communication

- Blocking
  - Only returns after completed
  - Receive: data has arrived and ready to use
  - Send: safe to reuse sent buffer
  - Be aware of deadlocks
  - Tip: Use when possible
- Non-Blocking
  - Returns immediately
  - Unsafe to modify buffers until operation is known to be complete
  - Allows computation and communication to overlap
  - Tip: Use only when needed

Credits:
https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf

Deadlock Scenario



Credits:
https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf

Collective Communication

- Collective communication allows you to exchange data among a group of processes
- It must involve all processes in the scope of a communicator
- Hence, it is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operation

Patterns of Collective Communication

1. Broadcast

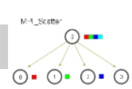
Patterns of Collective Communication

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm)
```

- Broadcasts a message from the process with rank `root` to all other processes of the group

Patterns of Collective Communication

```
MPI_Scatter(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



- Distributes elements of `sendbuf` to all processes in `comm`
- Although the root process (sender) contains the entire data array, `MPI_Scatter` will copy the appropriate element into the `recvbuf` of the process.
- `sendcount` and `recvcount` are counts per process

Patterns of Collective Communication

```
MPI_Gather(const void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Inverse of `MPI_Scatter`
- Only the root process needs to have a valid receive buffer. All other calling processes can pass NULL for recv_data

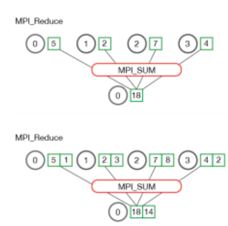
Computing average of numbers with MPI_Scatter and MPI_Gather

Patterns of Collective Communication

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm)
```

- Reduces values on all processes within a group.
- The `sendbuf` parameter is an array of elements of type datatype that each process wants to reduce.
- The `recvbuf` is only relevant on the process with a rank of root.
- The `recvbuf` array contains the reduced result and has a size of `sizeof(datatype) * count.` Why not just sizeof(datatype)?
- The op parameter is the operation that you wish to apply to your data.
- MPI contains a set of common reduction operations that can be used

Patterns of Collective Communication



Patterns of Collective Communication

1. *Broadcast*
2. *Scatter*
3. *Gather*
4. *Allgather*
5. *Alltoall*
6. *Reduce*
7. *Allreduce*
8. *Scan*
9. *Reducescatter*

Let’s implement a more efficient parallel_sum