Lab 5
Distributed-Memory Programming using MPI

CS3210 – 2024/25 Semester 1

Learning Outcomes

1. Learn to use synchronization and collective communication.
2. Learn how to create, destroy and manage new MPI communicators
3. Learn how to arrange MPI processes into a Cartesian virtual topology

Lab 4 provided you with basic knowledge on MPI programming. This lab aims to provide more detail on MPI communication calls. Log into a lab machine as in the previous labs. Download the code from Canvas or https://www.comp.nus.edu.sg/~srirams/cs3210/L5_code.zip

Part 1: Collective Communication

MPI provides collective communication functions which must involve (be invoked by) all processes in the scope of a communicator. By default, all processes are members of the global communicator `MPI_COMM_WORLD`.

Important:
It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations! Failure to do so may result in a deadlock.

There are three types of collective communication:

1. Synchronization communication
2. Data movement operations
3. Collective computation (data movement with reduction operations)

Part 1.1: Synchronization Communication

There is only one collective synchronization operation in MPI - a barrier:

```
-----------------------------------------------------------------
int MPI_Barrier(MPI_Comm comm)
-----------------------------------------------------------------
```

All processes in the MPI communicator `comm` will block until all of them reach the barrier. Failure to call this function from all the processes will result in deadlock.

Exercise 1

Compile and run `barrier.cpp` with 24 processes across 4 nodes. Currently, the master process (rank 23) receives numbers in an arbitrary order. Modify `barrier.cpp` by *only* adding one or more MPI_Barrier calls so that the master process receives numbers in this order: $0, 1, 2, \cdots, 21, 22, 0, 1, 2, \cdots$.

If you are interested, there is a non-blocking variant `MPI_Ibarrier` which returns immediately, independent of whether other processes have called `MPI_Ibarrier`. The barrier semantics are then only enforced at the corresponding completion operation (`MPI_Test` or `MPI_Wait`) with the `MPI_Request` provided to `MPI_Ibarrier`.

Part 1.2: Data Movement Operations

The data movement (distribution) operations provided by MPI are:

```
-------------------------------------------------------------------
/* MPI_Bcast - broadcasts (sends) a message from the process with rank
   root to all other processes in the group */

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
-------------------------------------------------------------------
/* MPI_Scatter - sends data from one process to all processes in
   a communicator */

int MPI_Scatter(const void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf,
                int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
-------------------------------------------------------------------
/* MPI_Gather - gathers data from a group of processes into one root
   process */

int MPI_Gather(const void *sendbuf, int sendcount,
               MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
-------------------------------------------------------------------
/* MPI_Allgather - gathers data from a group of processes into every
   process of that group */

int MPI_Allgather(const void *sendbuf, int sendcount,
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm)
-------------------------------------------------------------------
/* MPI_Alltoall - each process in a group performs a
   scatter operation,sending a distinct message to all the
   processes in the group in order by their rank */

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

The MPI-3 standard introduced Non-blocking Collective Data Movement operations such as `MPI_Ibroadcast` or `MPI_Iscatter`. Additionally, some of the operations above possess a variant that allows you to specify a varying number of data items to sent to each process, such as `MPI_Scatterv` or `MPI_Allgatherv`. To learn more, check the OpenMPI documentation for details of the non-blocking and variable variants of the above operations.

Exercise 2

Open the `bcast.cpp` program, which has a rudimentary (and non-optimal) attempt at implementing MPI_Bcast, which we call `my_bad_bcast`.

`bcast.cpp` intends to compare the runtime of `my_bad_bcast` with the true MPI_Bcast. However, it does not yet contain the MPI_Bcast call. Insert a call to MPI_Bcast (that has the same behavior as the current call to `my_bad_bcast`) below the `TODO` comment.

Now, run your modified `bcast.cpp` with the arguments `5000000 10` (broadcasting 5000000 numbers for 10 trials) on either the `xs-4114` or `i7-7700` partition, for 3 configurations:

- 2 nodes, 2 MPI processes x1.7 speedup
- 2 nodes, 3 MPI processes x1.8
- 2 nodes, 4 MPI processes x1.05

MPI_Bcast often (though not specified by the standard) uses a *tree-style* algorithm to send data between nodes. You can see the comments from line 157 – 177 in this source as an example of what MPI_Bcast might do. How does this information explain the performance of 4 nodes in our example vs 2 or 3?

Exercise 3

Compile the program `col_comm.cpp` and run it with 4 processes across 4 nodes. Explore the code and output to understand how scatter works in this example.

We now want to compute the **sum** of all numbers and gather the result back at the root process (rank 0). Each process should compute the sum of the numbers it receives from MPI_Scatter, and then call MPI_Gather to send all of these individual sums to the root process. The root process should then do one last sum of the gathered values to get the final sum of the array, and print it out. You should get an answer of 136. Insert your code below the TODO comment (you can add other variables anywhere as necessary).

Part 1.3: Collective Computation

The MPI functions that enable collective computations are:

```
-------------------------------------------------------------------
/* MPI_Reduce - reduces values on all processes within a group; the
   reduction operation must be one of the following:
   MPI_MAX maximum |  MPI_MIN minimum | MPI_SUM sum | MPI_PROD product |
   MPI_LAND logical AND | MPI_BAND bit-wise AND | MPI_LOR logical OR |
   MPI_BOR bit-wise OR  | MPI_LXOR logical XOR  | MPI_BXOR bit-wise XOR |
   MPI_MAXLOC max value and location | MPI_MINLOC min value and location
*/

int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
-------------------------------------------------------------------
/* MPI_Allreduce - applies a reduction operation and places the
   result in all processes in the communicator
   (this is equivalent to an MPI_Reduce followed by an MPI_Bcast) */

int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
-------------------------------------------------------------------
/* MPI_Reduce_scatter - first performs an element-wise reduction on a
   vector across all processes in the group, then splits the result
   vector into disjoint segments to distribute across the processes
   (this is equivalent to an MPI_Reduce followed by an MPI_scatter) */

int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,
                       const int recvcounts[], MPI_Datatype datatype,
                       MPI_Op op, MPI_Comm comm)
-------------------------------------------------------------------
```

Exercise 4

Open the `reduce.cpp` program. Each process in the program is currently generating one random integer. Modify `reduce.cpp` by using MPI_Reduce below the TODO point to *sum* all these random values, and then print the final sum in the master process (rank 0).

With one more simple change, try to have *every* process print out this sum, and not just the master process.

Part 2: Managing Communicators

One of the major disadvantages of using collective communication with MPI_COMM_WORLD is that all the processes must be involved. To overcome this, MPI allows us to create custom communicators, add / remove processes to / from communicators and destroy communicators as needed. **Note that a communicator comprises a set of processes (a `MPI_Group`) with an associated context.**

The MPI functions for communicator management are:

```
-------------------------------------------------------------------
/* MPI_Comm_group - returns the group associated with a communicator */

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
-------------------------------------------------------------------
/* MPI_Group_incl - produces a group by reordering an existing group and
   taking only listed members */

int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
                   MPI_Group *newgroup)
-------------------------------------------------------------------
/* MPI_Comm_create - creates a new communicator with a group of
   processes */

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
-------------------------------------------------------------------
/* MPI_Group_rank - returns the rank of the calling process in the given
   group */

int MPI_Group_rank(MPI_Group group, int *rank)
-------------------------------------------------------------------
/* MPI_Comm_rank - returns the rank of the calling process in the given
   communicator */

int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Exercise 5

Compile the program `new_comm.cpp` and run it. What does the program do?

Intra-group and Inter-group Communication

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called *intra-group communication* and the communicator used is called an *intra-communicator*. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It would be natural to specify the target process by rank within the target group in such applications. This type of communication is called *inter-group communication* and the communicator used is called an *inter-communicator*.

The group containing a process that initiates an inter-communication operation is called the *local group* that is, the sender in a send and the receiver in a receive. The group containing the target process is called the *remote group* that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint in order to avoid deadlock.

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
    MPI_Comm peer_comm, int remote_leader, int tag,
    MPI_Comm *newintercomm)
```

The function requires some explanation. First, `MPI_Intercomm_create` is collective over the union of the two intra-communicators that it is joining. Each intra-communicator will have a leader process within the inter-communicator; these can be thought of like network gateways; in MPI, point-to-point communications are enabled between the leader processes. The local_leader is the rank of the leader in the local communicator, where as the remote_leader is the rank of the leader in the peer communicator.

Both point-to-point and collective communications can be applied to inter-communicators. There are several process relationships in collective communications: all-to-one, one-to-all, all-to-all, and other (MPI_Scan would be the notable member of other). Point-to-point qualifies as one-to-one. In each case, when the "one" process belongs to one of the two member intra-communicators in the inter-communicator, the *all* corresponds to all the processes in the other member intra-communicator. In one-to-one communication, the two processes belong to the two separate inter-communicators (of course, otherwise, it would be intra-communication). Perhaps unintuitively, MPI_Barrier is included as a one-to-all operation, where the one calling process in a sub-group waits for all other processes to enter the barrier call in the other sub-group.

Part 3: Cartesian Virtual Topologies

In the context of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric space. Generally, there are two main types of topologies supported by MPI, namely, (i) Cartesian and (ii) graph. In this lab, we only introduce the Cartesian virtual topology.

Depending on the problem, the MPI processes may access data in a regular structured pattern in Cartesian space. In these cases, it is useful to arrange the logical MPI processes into a Cartesian virtual topology to facilitate programming and communication. **Remember that there may be no relation between the**

physical organisation (layout) of the parallel machines and the MPI virtual topology. Additionally, the MPI topology must be configured and managed by the programmer.

MPI provides three functions to help us manage a Cartesian topology:

```
------------------------------------------------------------------
/* MPI_Cart_create - makes a new communicator to which Cartesian
   topology information has been attached */

int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                    const int periods[], int reorder,
                    MPI_Comm *comm_cart)
------------------------------------------------------------------
/* MPI_Cart_coords - determines process coordinates in the Cartesian
   topology, given its rank in the group */

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
------------------------------------------------------------------
/* MPI_Cart_shift - returns the shifted source and destination ranks,
   given a shift direction and amount */

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int *rank_source, int *rank_dest)
------------------------------------------------------------------
```

Exercise 6

cart.cpp is an example implementation of a Cartesian topology. Study the program and understand how it works. Compile cart.cpp and run it. What does the program do?

More information and further reading:

- LLNL MPI Tutorial: https://computing.llnl.gov/tutorials/mpi/
- OpenMPI FAQ: https://www.open-mpi.org/faq/?category=running
- Another MPI Tutorial: http://mpitutorial.com/
- Short Video on Cartesian Topology: https://youtu.be/dyVOdlC0y7w
- Advanced Parallel Programming with MPI: https://htor.inf.ethz.ch/teaching/mpi_tutorials/ppopp13/2013-02-24-ppopp-mpi-advanced.pdf