CS3210 – 2016/17 Semester I

Lab 5
Distributed-Memory Programming using MPI

Objectives:

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

This lab aims to provide a more detailed coverage of the MPI libraries calls. You will have most of the required tools to tackle assignment 2 after this lab.

Note: All files in this lab can be found on IVLE workbin, or directly downloaded via wget:

```
$ wget www.comp.nus.edu.sg/~sooyj/cs3210/Lab5/<filename>

e.g.

$ wget www.comp.nus.edu.sg/~sooyj/cs3210/Lab5/cart.c
```

1. Process-to-process Communication

MPI provides two types of process-to-process communication: blocking and non-blocking. Each of these communication types is accomplished with a send and a receive function.

1.1 Blocking communication

Blocking send stops the caller until the message has been copied over to the underlying communication buffer (i.e. network buffer). Similarly, **blocking receive** blocks the calling process until the message has been received in the MPI process.

The syntax for these two functions is:

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

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

The arguments of these functions are:

buf	Pointer to the memory buffer that holds the contents of the message to be sent or received
count	The number of items that will be send.

datatype	Specifies the primitive data type of the individual item sent in the message, and can be one of the following:
	
	`MPI_CHAR`
	`MPI_SHORT`
	`MPI_INT`
	`MPI_LONG`
	`MPI_UNSIGNED_CHAR`
	`MPI_UNSIGNED_SHORT`
	`MPI_UNSIGNED_LONG`
	`MPI_UNSIGNED`
	`MPI_FLOAT`
	`MPI_DOUBLE`
	`MPI_LONG_DOUBLE`
	`MPI_BYTE`
	`MPI_PACKED`

dest/source	Specifies the rank of the source / destination process
tag	An integer that allows the receiving process to distinguish a message from a sequence of messages originating from the same sender
comm	The MPI communicator
status	Pointer to an MPI_STATUS structure that allows us to check if the receive has been successful.

Exercise 1: Compile the program *block_comm.c* and run it with two processes.

Exercise 2: Modify the file *block_comm.c* such that process 1 sends back to process 0 ten floating point values using only one message. Compile the program and run it.

Exercise 3: What happens if we flip the order of `MPI_Send` and `MPI_Recv` in the master process? Discuss the implication.

1.2 Non-blocking communication

Non-blocking communication, in contrast to blocking communication, does not block either the sender or the receiver.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request)
```

The only new parameter in the call is:

request	Pointer of type `MPI_Request` which provides a handle to this operation. Using this handle, the programmer can inquire later whether the communication has completed.

Both functions return immediately, and the communication will be performed asynchronously w.r.t. the rest of the computation. **Using these functions, the MPI program can overlap communication with computation.**

To check whether the functions have finished communicating, we can use:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

`MPI_Test` takes a handle of an `MPI_Isend` or `MPI_Irecv` and store a true / false in the `flag` variable which indicate whether the operation has been completed or not.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

`MPI_Wait` blocks the current process until the request has finished.

MPI provides quite a few variants of these basic two functions. You are encouraged to read the MPI manual for these functions:

```
MPI_Test, MPI_Testall, MPI_Testany, MPI_Testsome
MPI_Wait, MPI_Waitany, MPI_Waitsome
```

Exercise 4: Compile the program *nblock_comm.c* and run it with 3 processes. Modify the source code to flip the order of the `MPI_Isend` and `MPI_Irecv`. Compile and run it again. What do you observe?

2. Collective Communication

MPI provides **collective communication** functions which **must involve all processes in the scope of a communicator**. By default, all processes are members in the communicator **`MPI_COMM_WORLD`**.

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

There are three types of collective communication:

2.1. Synchronization communication;
2.2. Data movement operations;
2.3. Collective computations (data movement with reduction operations).

2.1 Synchronization communication

There is only one collective synchronization operation in MPI:

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

All processes from 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.

2.2 Data Movement Operations

The data-movement operations provided by MPI 1.0 are:

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

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

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

`MPI_Scatter` - sends data from one process to all processes in a communicator.

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

`MPI_Allgather` - gathers data from a group of processes into each of the process from that group.

```
int MPI_Alltoall(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 index.

2.3 Collective computations

The MPI functions that achieve collective computations are:

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

`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

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
           MPI_Datatype datatype, MPI_Op op, 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_Reduce_scatter(void *sendbuf, void *recvbuf, int *recvcounts,
                MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

`MPI_Reduce_scatter` – first, it does an element-wise reduction on a vector across all processes in the group. Next, the result vector is split into disjoint segments and distributed across the processes. This is equivalent to an `MPI_Reduce` followed by an `MPI_Scatter` operation.

Exercise 5: Compile the program *col_comm.c* and run it. What can we do if we want to scatter the data only in a subset of the processes of the MPI communicator?

3. Managing MPI Communicators

One of the major disadvantages of using collective communications is that we must use all the processes in the MPI communicator. To overcome this, MPI allows us to create custom communicators, add / remove processes to / from communicator and destroy communicators.

The MPI functions for communicator management are:

`int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)` `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_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)` `MPI_Comm_create` - creates a new communicator with a group of processes.
`int MPI_Group_rank(MPI_Group group, int *rank)` `MPI_Group_rank` - returns the rank of the calling process in the given group.

Exercise 6: Compile the program *new_comm.c* and run it. What does the program do?

4. Cartesian Virtual Topologies

Often the MPI processes access data in a regular structured pattern in Cartesian space. In those case, it is useful to arrange the logical MPI processes into a Cartesian virtual topology to facilitate coding and communication.

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

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                int *periods, int reorder, MPI_Comm *comm_cart)
```

`MPI_Cart_create` – makes a new communicator to which Cartesian topology information has been attached.

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

`MPI_Cart_coords` - determines process coordinates in the Cartesian topology, given its rank in the group.

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

`MPI_Cart_shift` - returns the shifted source and destination ranks, given a shift direction and amount.

Exercise 7: Compile the program *cart.c* and run it. What does the program do?