DUE DATE: **BY THE END OF YOUR CLASS**

CSE443/543: High Performance Computing

Exercise #19: Collective communication

Points: 40

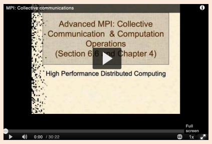
Objective: The objective of this exercise is to:

- Gain familiarity with collective communication.
- Understand the use of MPI collective communication operations.
- Learn to run MPI programs via SLURM on a compute cluster.

Submission: Upload the following at the end of the lab exercise via Canvas CODE plugin:

1. This MS-Word document saved as a PDF file with the convention `MUID_Exercise19.pdf`.
2. The program you completed as part of this exercise with the source file named with the convention `MUID_exercise19.cpp`.

Part #0: Review online lecture


MPI: Collective communications
Advanced MPI: Collective Communication & Computation Operations
(Section 6 ... Chapter 4)
High Performance Distributed Computing

Required video review:
Prior to working on this exercise, ensure you review the lecture video on collective communications on Canvas.

Part #1: Short answer questions

Provide a brief (2-to-3 sentences) response to each of the following questions.

1. What is a virtual synchronization point? Explain with a suitable MPI call. How is it different from a barrier?
   a. 2 Advantages

A virtual synchronization point is where all processes call the same function with the same parameters and the processes do not block each other.

```
int i = 10, dest;
brodcast(world, i, dest, 0);
```

A barrier blocks processes until all processes have called the barrier function at which it unblocks.

2. Briefly describe 2 significant differences between conceptual broadcast versus scatter operations

Broadcast	***Scatter***

Same value sent to all the processes	Different values are sent to different processes
Amount of data received by each process is the same	Amount of data received by each process can vary

3. Given the following MPI code fragment from process with rank 0, complete the complementary collective operation on other processes

```
void doManagerTasks(const std::string& data) {
    int strSize = data.size() + 1;
    MPI_Bcast(&strSize,     1, MPI_INT,  0, MPI_COMM_WORLD);
    MPI_Bcast(&data[0], size, MPI_CHAR, 0, MPI_COMM_WORLD);
}
```

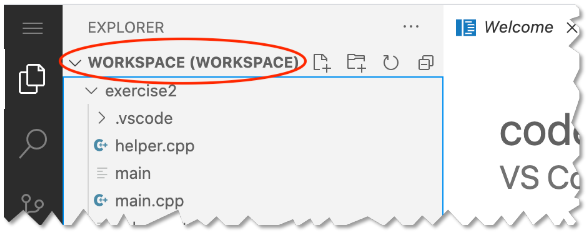
```
std::string recvData() {
    int strSize = 0;
    MPI_Bcast(&strSize,     1, MPI_INT,  0, MPI_COMM_WORLD);
    std::string data(strSize);
    MPI_Bcast(&data[0], size, MPI_CHAR, 0, MPI_COMM_WORLD);
    return data;
}
```

Part #2: Programming with collective communication

In this part of the exercise, you will be required to complete 3 interview/exam style questions in the supplied `exercise19.cpp` starter code.

Part #2.1: Setting up VS-Code project

Estimated time to complete: 5 minutes



1. Log into OSC's OnDemand portal via https://ondemand.osc.edu/. Login with your OSC id and password.
2. Startup a VS-Code server and connect to VS-Code. Ensure you switch to your workspace. Your VS-Code window should appear as shown in the adjacent screenshot.
3. Next, create a new VS-Code project in the following manner:
   a. Start a new terminal in VS-Code
   b. In the VS-Code terminal use the following commands:

```
$ # First change to your workspace directory
$ cd ~/cse443
$ # Use ls to check if workspace.code-workspace file is in pwd
$ # Next copy the basic template for a C++ project
$ cp -r /fs/ess/PMIU0184/cse443/templates/mpi exercise19
$ # Copy the starter code for this exercise
$ cp /fs/ess/PMIU0184/cse443/exercises/exercise19/* exercise19
```

Part #2.2: Mean and variance

In the supplied `exercise19.cpp` starter code, implement the `getMeanAndVar` method. The `getMeanAndVar` method is called on **n** processes of an MPI program. Each process will have a different `value` (as shown in the figure below). This method must be implemented to compute and return the mean and variance on every process (as shown in the figure below). **Note that your implementation must use only collective communication and computation operations**. Hint: `all_reduce`.

$$mean\ (average):\ \mu \qquad variance: \sigma^2$$

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n} \qquad \sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}$$

Rank:	0	1	2
value (the parameter to getMeanAndVar)	1	2	3
Return values	{2.5, 0.66}	{2.5, 0.66}	{2.5, 0.66}

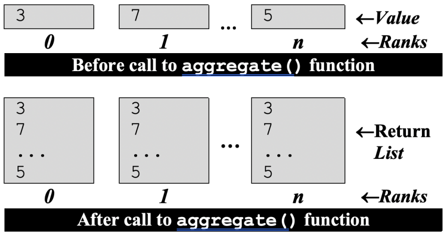
Expected output(s):

```
$ srun -A PMIU0184 -n 3 ./exercise19 q1 1 -2 3
srun: job 7294937 queued and waiting for resources
srun: job 7294937 has been allocated resources
mean: 0.666667, variance: 4.22222
```

```
$ srun -A PMIU0110 -n 4 ./exercise19 q1 1 2 3 4
srun: job 7301859 queued and waiting for resources
srun: job 7301859 has been allocated resources
mean: 2.5, variance: 1.25
```

Part #2.3: Aggregate

In the supplied `exercise19.cpp` starter code, implement the `aggregate` method. The `getMeanAndVar` method is called on **n** processes of an MPI program. Each process will have a different `value` (as shown in the adjacent). This method must be implemented to collect/aggregate the `values` (in order of rank) from all processes into a vector and return the vector in all processes.



Expected output(s):

```
$ srun -A PMIU0110 -n 4 ./exercise19 q2 1 2 3 4
srun: job 7302122 queued and waiting for resources
srun: job 7302122 has been allocated resources
List[0] = 1
List[1] = 2
List[2] = 3
List[3] = 4
```

```
$ srun -A PMIU0110 -n 4 ./exercise19 q2 1 5 9 3 7
srun: job 7302677 queued and waiting for resources
srun: job 7302677 has been allocated resources
```

```
List[0] = 1
List[1] = 5
List[2] = 9
List[3] = 3
List[4] = 7
```

Part #2.2: Print in-order

In the supplied `exercise19.cpp` starter code, implement the `printInOrder` method. This method is called on all processes of an MPI program. Each process will have a different `value` supplied as the parameter. Implement this method to print the `value` (on each process) in the order of the rank of the process. Hint: `barrier` in a loop and the ith process prints in the ith iteration of the loop.

Expected output(s):

```
$ srun -A PMIU0110 -n 5 ./exercise19 q3 1 5 9 3 7
srun: job 7302925 queued and waiting for resources
srun: job 7302925 has been allocated resources
Value at rank #0 = "data on 0 is 1_0"
Value at rank #1 = "data on 1 is 1_1"
Value at rank #2 = "data on 2 is 1_2"
Value at rank #3 = "data on 3 is 1_3"
Value at rank #4 = "data on 4 is 1_4"
```

```
$ srun -A PMIU0110 -n 3 ./exercise19 q3 test
srun: job 7302964 queued and waiting for resources
srun: job 7302964 has been allocated resources
Value at rank #0 = "data on 0 is test_0"
Value at rank #1 = "data on 1 is test_1"
Value at rank #2 = "data on 2 is test_2"
```

Part #3: Upload solution to Canvas

Once you have successfully completed and tested the program, submit the following via the Canvas CODE plugin

1. This MS-Word document saved as a PDF file with the convention `MUID_Exercise19.pdf`.
2. The program you completed as part of this exercise with the source file named with the convention `MUID_exercise19.cpp`.

Ensure you actually complete the submission process in Canvas (after you accept submission in CODE).