ME759
High Performance Computing for Engineering Applications
Assignment 11
Date Assigned: December 2, 2015
Date Due: December 9, 2015 – 11:59 PM

Pick at least two out of the five problems below and provide their solution by the due date (use Learn@UW to that end).

The goals of this assignment are as follows:

- Understand why a solution based on collective communication, specifically on the `MPI_Bcast` service, is superior to ad-hoc solutions drawing on point-to-point communication (Problem 1)
- Exercise the use of `MPI_Send` and `MPI_Isend` in MPI parallel programming and understand the benefits/drawbacks associated with each one of them (Problem 2)
- Getting more versed with using MPI by implementing code to evaluate an integral (Problem 3)
- Understanding the potential of MPI relative to that of CUDA. To this end, you'll perform a vector reduction using the native MPI reduction support in OpenMPI and then compare its efficiency with that of handcrafted MPI and then CUDA implementations (Problem 4)
- Gauge the overhead associated with passing messages between compute nodes when using OpenMPI over a 40 Gbs Infiniband interconnect (Problem 5)

Problem 1. Write an MPI program that utilizes 16 processes to do one thing: process 0 will send to the other 15 processes an array of data. Specifically, transfer from process 0 to the other processes 2^0 bytes; 2^1 bytes; 2^2 bytes; …; 2^{30} bytes. Generate a **png** log-log plot that shows the amount of time required by each of these transfers. Do not register the amount of time necessary to allocate memory. You might want to allocate memory once, for the most demanding case (2^{30} bytes), and then use it for all the other data transfer cases. Make sure you run your production code (compiled with –O2 or –O3) several times to get a good idea about the average amount of time you can expect in a real-life application.
You will have to compare (on the same plot) two scenarios:

a) You use a `MPI_Bcast` operation to transfer the data
b) You use a `for`-loop to carry out the data transfer using point-to-point Send/Receive operations

Your report should include

- A "results table" that summarizes your findings
- A discussion of the timing results for the two scenarios above
- The **png** plot (upload to Forum as well)

Problem 2. Imagine you launch an MPI job with P processes. Each process is supposed to generate a set of N random integers between -5 and 5 using a uniform distribution. To this end, use the function `rand()` seeded by each process in a way that is unique to it. For instance, you can seed based on the rank of the process.
In an effort to understand if `rand()` actually generates random numbers based on a uniform distribution, each process J, $0 \le J < P$, is supposed to compute the average and standard deviation for the array of random numbers it generated. Once process J finishes this operation, it should store the two values (average and standard deviation) in a local array of dimension $2P$, call it R, from results. As process J does not "trust" process $J-1$, it will ask for the N random numbers stored by $J-1$, and once it gets this data it computes a new average and standard deviation, which it also stores in R. This is a "grab-from-the-left" approach, where each process, once it computes an average and standard deviation, grabs new data from the left and passes its data to the right process. Note that process 0 grabs from process $P-1$ and passes to process 1.

After $P-1$ rounds of grab-from-left-and-pass-to-write steps, each process should have its array R that stores the average and standard deviation of each set of data generated by each of the other $P-1$ processes. At this point, the process with rank 0, which plays the role of root, does a collective operation to get all these local arrays into one large array in order to verify that indeed all these local arrays store identical average/stdev data. Once this confidence/sanity-check test is cleared, the root prints out the P average/stdev values stored in *its* local array.

You are supposed to play this game for $N = 2, 4, 8, \ldots, 2^{26}$ integers.

a) Implement a solution to this problem that relies on plain vanilla `MPI_Send` operations
b) In an attempt to improve the performance of the solution above, implement a solution to this problem that relies on `MPI_Isend` operations

Your report should include:

- A **png** log-log plot with the timing results for a) and b) above for $P = 4$
- A **png** log-log plot with the timing results for a) and b) above for $P = 13$
- Answers to the following two questions:
  i) Based on your observations, is `rand()` producing numbers based on a uniform distribution?
  ii) Can you briefly describe a more efficient approach for the problem described? Plain words suffice, provide pseudo-code only if you want to.

Problem 3. Drawing on the integral calculation example presented in class, write a program that uses the MPI parallel programing paradigm to evaluate the integral

$$I = \int_0^{100} e^{\sin x} \cos(\frac{x}{40}) dx$$

Note that the value provided by MATLAB for this integral is I = 32.121040688226245. To approximate the value of I use the following extended Simpson's rule:

$$\int_0^{100} f(x)dx \approx \frac{h}{48}\left[17f(x_0)+59f(x_1)+43f(x_2)+49f(x_3)+48\sum_{i=4}^{n-4} f(x_i)+49f(x_{n-3})+43f(x_{n-2})+59f(x_{n-1})+17f(x_n)\right]$$

In the approximation above, $x_0 = 0$, $x_n = 100$, $h = 10^{-4}$, and $n = \frac{100-0}{h} = 10^6$. This value of n goes to say that you divide the interval $[0, 100]$ in 10^6 subintervals when evaluating I.

After implementing the code, you will have to run the code on Euler using several scenarios of your choice. Here's a set of possible cases:

- one node and one core
- one node and four cores
- one node and eight cores (Euler has on each compute node two quad-core Intel Xeon 5520)
- two nodes and four cores on each node
- four nodes and two cores on each node
- etc.

Your report should include

- A "results table" that summarizes your findings based on the scenarios that you ran
- The execution configuration; i.e., number of compute nodes and number of cores per node that produces the value of I in the shortest amount of time. Report this combination along with the corresponding timing result on the forum.

Problem 4. Go back to your favorite implementation of the reduction operation on the GPU. The array that you handle for this problem is of random integers between -5 and 5. First, figure out the largest array size that the CUDA implementation that you can handle. With this array:

- Perform a reduction operation using the $+$ operator on the GPU
- Write your own MPI implementation of the reduction task by combining a straight sequential summation by each process, collecting results to a root process and producing the final result therein. Use as many processes NP as you find advantageous.
- Use the MPI_Reduce function. Use as many processes NP as you find advantageous.

Report on the forum your results in milliseconds using the following format:

```
Max array size handled: blah
CUDA Implementation: blahblah
Handcrafted implementation (NP=…): blahblahblah
MPI_Reduce (NP=…): blahblah
```

Problem 5. This problem is the sister of Assignment's 7 Problem 1.

i) Run an analysis to gauge how much time it takes to move data from a process A to a process B using Euler's OpenMPI implementation of the MPI standard. To this end, transfer from a process A to a process B 2^0 bytes; 2^1 bytes; 2^2 bytes; …; 2^{30} bytes. Generate a **png** plot that shows the amount of time required by each of these transfers. Do not register the amount of time necessary to allocate

memory. You might want to allocate memory once, for the most demanding case (2^{30} bytes), and then use it for all the other data transfer cases. Make sure you run your code several times to get a good idea about the average amount of time you can expect in a real-life application.

ii) Do the same thing as before but instead time a data transfer from A to B followed immediately by a data transfer from B to A. This is called Ping-Pong. To this end, allocate 2^{30} bytes on A. Then allocate the same amount on B. Time the following sequence of two operations: copy N bytes from A into B, then from B back into A. Run this analysis for $N = 2^0, \ldots, 2^{30}$ bytes and generate a **png** plot with timing results for each value of N.

For this problem, upload the FOUR **png** plots to the. The plots you provide should be identified as follows and capture four different scenarios:

- "PLOT 1" – i) above & processes A and B live on the same node
- "PLOT 2" – i) above & processes A and B live on different nodes
- "PLOT 3" – ii) above & processes A and B live on the same node
- "PLOT 4" – ii) above & processes A and B live on different nodes.

In your report, answer these two questions:

- How are your results correlating when A and B are on the same node as opposed to two nodes?
- How are your results correlating when one way vs. two way (Ping-Pong) data movement operations are considered?

NOTE: You should use the `MPI_Ssend` flavor of the send operation to endure synchronization of the send/receive operation. This is just to make sure things are fair for small data transactions, when the data might be buffered by OpenMPI for you. In other words, we are enforcing a "rendezvous" always policy and not get tricked by "eager" mode transfers.