ME964
High Performance Computing for Engineering Applications
Assignment 11
Date Assigned: April 14, 2012
Date Due: April 22, 2012 – 11:59 PM

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)
- Understand how to use `MPI_Send` and `MPI_Isend` in MPI parallel programming, and understand the benefits/drawbacks associated with each one of them (Problem 2)

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 http://sbel.wisc.edu/Forum/viewtopic.php?f=13&t=335)

Problem 2. Imagine you have 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, $0 \leq J < P$, 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. As process J does not "trust" process $J-1$, it will ask for the data generated by $J-1$, and once it gets it, it computes a new average and standard deviation, which it stores in the local array mentioned earlier. 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 a local array 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.