Lab #7 – More collective communication

Up to this point, we have seen the following functions that communicate data between processes: MPI_Send, MPI_Recv and MPI_Reduce.

If you have not so already, please show me yesterday's program that counted primes.

There are a few other important communication functions we should take a look at.

MPI_Bcast – This function is used when 1 process wants to send data to all other processes. Using this function is faster than writing a loop and using MPI_Send to each process one by one.

MPI_Barrier – In a sense, calling this function introduces a breakpoint or pause in your program. This is used when you want all processes to stop until everybody reaches this point in the program. It's helpful when one process such as 0 is reading data that it needs to send to the other processes.

MPI_Scatter – This is used to send different portions of data among the processes. For example, if you have 10,000 values to sum, you might want each of 10 processes to work on just 1,000 values. Scattering the data means that each process only receives the data it needs to work on. It is not necessary to broadcast the entire array to all processes.

MPI_Gather – This is essentially the inverse of scatter.

Your book on pages 106-112 discusses these functions. However, the author does not provide a complete program to work on. I found a tutorial online that does provide a fairly complete example of using the MPI_Scatter() and MPI_Gather() functions. It is available here: http://www.mpitutorial.com/mpi-scatter-gather-and-allgather/

1. Download the avg.c program to your system. Compile and run the program.

2. Where does the program create the big array of random numbers? Where in the program is this array used? Write your answers here or mark your answers on the hard copy of the code.

3. Where in the program do we create a sub-array that will contain a portion of the entire set of random numbers? And how does this sub-array get used?

4. On lines 75-77 why do you think the author creates another array here?

5. I don't like floats. Copy avg.c to avg2.c and for this new version, make the necessary changes so that the program works with random integer values instead of random floats. Doing this will help us with our next exercise.

Now that we have a better understanding of how we can communicate arrays between processes, let's tackle this problem: Let's generate a list of random integers, and count how many times the number 42 (or some other number) appears. Of course, later we can modify the program so that this target number is specified as input.

1. Write a simple serial C program that prints out 100,000 random integers, each in the range from 0 to 9999. Print one number per line. Run this program, and redirect the output to a new text file called numbers.txt.

2. Next, we are going to write a parallel C program called count.c. It will have the same basic structure of the avg.c program we saw earlier. The differences are that we are now working with integers, and we are reading these values from a file instead of generating them randomly in this program. Begin this program by creating a global array for all the integers that will be read in. Make sure this array is large enough to hold MORE than 100,000 integers.

   And then write the function that will read numbers.txt and put these values into the array. Just to be on the safe side, this function should return the number of values that were encountered. In other words, our array could be declared to have more than enough space than was needed.

3. The original avg.c program had a compute_avg() function. Instead, our program will have an analogous count() function. It will be structured in much the same way, but note that our sub-array here consists of int, not float.

4. The main() function will contain the fun stuff.

   a. Begin by inserting the necessary MPI code to get us started. This includes finding out the number of processors, my process rank, and it's convenient also to have the name of our processor.

   b. If we are in process 0, then we should call the function that reads the input file.

   c. Next, we need to compute the number of values that each process will need to examine to find the number 42. For this calculation, you will need to have

variables for: the total number of values in the big array, the number of processes, and the number of elements per process. You need to make a special case for the last process in case the number of processes does not evenly divide into the number of values read into the array.

d. At this point it would be a good idea to compile and run your program so that you can verify that the values being computed so far are correct.

e. Something wrong? You might notice that process 0 is taking a lot longer than the other processes because it has to read a file! This means we must use the MPI_Barrier() function. Where should it go?

f. Compile and run the program again. Can you tell that something is wrong with the output? The issue here is that we need to broadcast the number of values in the array to all the processes. You can find out how to use MPI_Bcast either from the text on pages 106-108; or you can look at the online tutorial http://www.mpitutorial.com/mpi-broadcast-and-collective-communication/ for a very good explanation; or you can find a terse definition of the function on here: http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Bcast.html

g. Now, it's time to "scatter" the list of numbers to all the processes. Please follow the example in avg.c but of course we are dealing with integers.

h. Finish the implementation, keeping in mind that we are just finding a total count, not finding an average.

5. This program is a little complicated, and you are not expected to finish it today. Please take your time. We are here to learn, not just get programs completed. Study the examples and function specifications carefully, and take time to add diagnostic code to your program.