Lab #6 – Running Programs in Parallel

In today's lab, we will practice writing very simple parallel programs. Also, we will take a look at some MPI functions that support collective communication (section 3.4 in the text). You may find it helpful to refer to some online documentation on MPI functions. Here is one place to find it:

http://www.mcs.anl.gov/research/projects/mpi/www/www3/

At last, we now know enough programming and system nuts and bolts to begin writing programs that actually run in parallel. Here is the general procedure. The first few steps are already familiar to you.

1. Serial version. Create a new directory (for convenience), and write an ordinary serial version of the program you need to write. Compile with gcc and run. The purpose of this is so that you can verify that your general problem-solving algorithm is correct. If there are bugs later, you would at least be assured that your basic algorithm is right.

2. Add MPI stuff. Create a second version of your program, this time incorporating MPI. The code needs to be structured in such a way that you delegate work among several processes. Assume that you don't know the number of processes when you write the program. It is not known at compile time. In other words, if your parallel algorithm works elegantly with 10 processes and you know that you have only 7 machines, then just go ahead with the 10 processes anyway. Make your programming life easy. Later on, when you are all done, you can focus on optimizing your code if you want.

   The number of processes and machines are given at run time when the user invokes the program. Also, note that the number of processes may exceed the number of machines that you have available. This is no problem: the operating system can multitask. But to make the program run efficiently, the number of processes should match the number of machines. As the book points out, in the future your program may run on a system with more machines!

3. Verify that the MPI version works. Compile with mpicc. Let's run your program using processes on one machine only. For example: mpiexec –n 8 ./integral.

4. Get ready for multiple machines. Let's add code to your program so that we can demonstrate that different machines are actually in use.

   a. At the beginning of your program, declare an int variable called name_length.
   b. Also declare a string like this:
      char processor_name[MPI_MAX_PROCESSOR_NAME];

c. Just after the point in your program where you obtain your process rank and the number of processes, invoke this function:
   MPI_Get_processor_name(processor_name, &name_length);
d. Print out the processor's name, e.g. like this
   fprintf(stderr, "Process %d on %s\n", my_rank, processor_name);

The reason why we are using fprintf and writing to stderr instead of stdout is that this is just diagnostic output and we don't want it to be confused with the actual computed output of your program. Note that by default, stdout and stderr both go to the screen simultaneously. If you redirect your stdout to a file, the stderr output can still be seen on the screen. This is a very helpful feature! (And it's also possible to redirect the stderr to a file if there is a lot of it.) If your program won't have that much output anyway, you could use just the regular printf() call here.

5. Copy files. Copy your machinefile file that you created in the original mpi_testing folder into the current folder. Note that this is the file that contains the IP numbers of every machine in your cluster.

   Next, we need to copy your executable file (the one that you would invoke with mpiexec) to all the other machines in your cluster. You can use scp for this purpose.

   Please note that your executable must be in the same place in the file system on all machines. So, you might find it convenient to have all of your MPI executables stored in the same directory. If you use a different directory for each program, then you will have to create the same directory on each machine.

   Good news: scp to the rescue! This command allows you to copy entire directories. Suppose you have a directory called integral that contains your C source file, along with the executable file and machinefile (listing the IP numbers). Then, it's possible to copy everything you need to another machine with one call to scp. Go up a directory, and run the scp command with the –r option in order to copy the integral directory and its contents. As the command runs, you should see diagnostic output showing you each file that is being transferred.

6. Run your parallel program, e.g. like this:
   mpiexec –f machinefile –n 8 ./integral

7. That's it! Please note, however, that each time you make a change to your source program and recompile it, you will have to copy your new executable file over to the other machines in your cluster. This is why it's a good idea in general to make sure you are happy with your MPI implementation before rushing to do step 5.

Practice this new procedure: take a look at an MPI program you wrote recently, and apply steps 4-6 above. Running the updated version, you should notice that all of your machines are talking. You might even notice the program running faster! This will become especially apparent later when we experiment with longer running programs.

For the rest of today's lab we begin to experiment with some "collective communication" as described in section 3.4 in your book. This is a long section, and we won't finish it today! We'll start with the "reduce" operation in MPI.

1. Create a new folder called reduce. Inside this folder create a new C source file called reduce.c. You may want to refer to the integral program you worked on earlier to understand the basic layout of a program that uses MPI.

2. This program will be simple, and will consist of just a main() function and a compute() function. The compute() function will basically take the place of the Trap() function we used in integral.

   Your main function will have this basic structure:

   a. Declare variables. We will especially need an int variable my_rank which will hold the logical process number obtained from MPI_Comm_rank(). We also need two double variables: local_result and total_result, similar to the integral program.
   b. Set the value of local_result equal to compute(my_rank).
   c. There should be an if-else statement in the middle of your main() function. It should basically do the following:

      ```
      If my process rank is not zero:
         Use MPI_Send to send local_result, which is just 1 value of type double,
      to the process at rank zero.
      Else (my rank is zero):
         total_result = local_result
         for all other ranks, use MPI_Recv to grab their local_result values, and
      accumulate these values in my total_result variable.
      ```

   d. Near the end of your main() function, just before MPI_Finalize(), use an if-statement to see if our process rank is 0. If it is, then we should print the value of total_result.

   e. Now, we are ready to write the compute() function. It will take an int parameter (my process rank), and return a double. The value to be returned should be:

      ```
      3.0 + (rank + 2) % 8
      ```

3. Carefully build this program. Follow steps 3-7 in the problem solving procedure given above. Use 8 processes. Do not continue unless the output is correct and makes sense. What should the output be? Experiment with a different number of processes to verify that the program still gives the correct output.

4. Good news! It turns out that there is a useful MPI function that can save us about 10 lines of code. It's called MPI_Reduce. Comment out the if-else statement you wrote in step 2c earlier. Immediately before it, enter this statement:
   MPI_Reduce(&local_result, &total_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   Compile, distribute and run your program again to verify that your results have not changed.

5. Modify your program so that it will calculate the product (instead of the sum) of all the numbers being computed by each process. Verify that your program produces the correct results. The diagnostic output should show you each individual value being produced (local_result) as well as the final answer (total_result).

Once you understand how to use the reduce operation, let's use it to solve another parallel problem. Let's write a program to count the number of prime numbers between, say, 1 and 10,000. In general, we can count the number of primes between 1 and n thousand, where n is the number of processes we want. This is convenient because we can specify the number of processes (and thus the range of values to search for primes) at the command line. Each process will be in charge of counting the number of primes in its thousand. For example, process 0 will count how many primes exist from 1 to 1000, process 6 will look at the range 5001 to 6000, etc.

Follow the parallel problem solving procedure. Feel free to structure your program along the lines of the integral and reduce programs we just finished. You should use the MPI_Reduce() function in solving this problem. You should print the result from each process as well as the final total.

When you are done, please review the material we have worked on up to this point. Tomorrow we will start class with a quiz. Then, we will have a lab continuing the MPI collective communication functions in section 3.4.