THE UNIVERSITY OF ALABAMA IN HUNTSVILLE
Department of Electrical and Computer Engineering

CPE 412/512
Fall Semester 2015

Homework Assignment Number 3

1. Students are to compile and execute on the Alabama Supercomputer's dmc system (dmc.asc.edu) the matrix/matrix multiplication program (*mm_mult_serial.cpp)* that is provided to them as part of this assignment. This reference code first generates a pseudorandom data set using input dimension parameters of *l*, *m*, and *n*, where the first matrix to be multiplied is of size *l* x *m* and the second matrix is of size *m* x *n*. These dimensions are entered on the linux command line. This code can be obtained by coping it from the shared directory on the dmc system using the command

   ```
   cp /home/shared/wells/hw3/mm_mult_serial.cpp .
   ```

   or by downloading it directly from the UAH CPE 512/412 Canvas^{TM} site. Each student is to illustrate the correct operation of this program by compiling and running it in the interactive mode for at least one small case where the dimensions *l*, *m*, and *n* each have distinct/different values. ***In this listing they should include the resulting input/output as part of this homework write-up.***

2. Next student should create a Single-Program-Multiple-Data (SPMD) matrix/matrix MPI multiplication program (*mm_mult_mpi.cpp)* by expanding the serial matrix/matrix multiplication program (*mm_mult_serial.cpp)* that was provided in this assignment in a manner that allows for message-passing based general purpose data parallelism. This program should divide the data set as evenly as possible using the row decomposition method that was outlined in class. The parallel code should generate the same pseudorandom data set on the Rank 0 MPI process as the original serial program where the first matrix to be multiplied is of size *l* x *m* and the second matrix is of size *m* x *n*. Both matrices should be generated and stored in the memory space of the Rank 0 process and the time needed to generate and output these matrices should not be included as part of the timing data. The parallel code should be written in a manner that allows the resultant matrix to be communicated to a contiguous region within the memory space of the Rank 0 process. The time to output this matrix to the screen should also not be included in this timing. The data type used in the parallel version should also be the same as the serial version which is of type ***float***. The program should be written in a general manner that allows the number of MPI processes to be specified at run time using the *mpirun* command. Students should illustrate the correctness of this program for values of processes from 2,4,6, and 8 using a small but representative data set that demonstrates that the parallel version of the program functions correctly for at least one small case where the dimensions *l*, *m*, and *n* each have distinct/different values and the number of MPI process is takes on the values of 2, 4, and 8, respectively. ***Students should include the resulting input/output as part of this homework write-up.***

3. After the small scale verification runs for both the serial and parallel version of this program have been verified as discussed in parts 1 and 2 of this assignment students are to perform detailed time measurements on both versions of the code for various data sizes using both the gnu and intel compilers. These timing measurements should record the total execution time but should exclude the time it takes to generate and echo the input matrices as well as the time it takes to output the resultant matrix to the screen. In the parallel case it should include the time it takes to perform inter-process communication operations (i.e. broadcast, scatter, gather, etc.) and parallel computation time. It would be prudent for students to comment out the screen I/O statements during these runs. To allow for extended execution times and non-shared access to the CPU cores the dmc's batch queuing system must be employed. A separate handout contains information on how to access the batch queuing system. It is strongly suggested that students place all the run time commands into a single script and to execute this script using the *run_script* command in a manner where a total of 8 processing cores are reserved for the duration of the entire set of time measurements (even though most of the routines will use less than the 8 processing cores this will ensure that the same set of processing cores are used throughout the timing process). A separate script file is included for each compiler (*mm_gnu.scr*, and *mm_intel.scr,* respectively) which is present on the Canvas^{TM} site or can be obtained by coping it from the shared directory on the dmc system using the commands

```
cp /home/shared/wells/hw3/mm_gnu.scr .
cp /home/shared/wells/hw3/mm_intel.scr .
```

These commands contained in these scripts assume that only the data size and run time measurement information is supplied from the program and all other I/O is commented out.

Before the scripts can be invoked using the *run_script* command, students are to create separate executables for both the serial and parallel versions for each compiler. If the serial source file is named m*m_mult_serial.cpp* and the parallel source file is named *mm_mult_MPI.cpp* then the following commands can be used from the linux command line to generate the four resulting executables.

First make sure that the environment for MPI is set appropriately.

```
module load openmpi
```

Then to generate the gnu compiler executables for both the serial and parallel source code files enter the following commands:

```
g++ mm_mult_serial.cpp -o mm_mult_serial_gnu -lm -O3
export OMPI_CXX=g++
mpic++ mm_mult_MPI.cpp -o mm_mult_MPI_gnu -lm  -O3
```

Then to generate the Intel compiler executables for both the serial and parallel source code files enter the following commands:

```
module load intel
icc mm_mult_serial.cpp -o mm_mult_serial_intel -lm -O3
export OMPI_CXX=icc
mpic++ mm_mult_MPI.cpp -o mm_mult_MPI_intel -lm -O3
```

Note that the same *mpic++* command is used to generate both MPI based executables but the environment variable **OMP_CXX** is changed before the *mpic++* command is executed to allow the appropriate compiler to be called.

The batch queuing system should be used to execute both of these scripts separately, using the **class** queue, and reserving **8** processing cores (all other parameters should be given their defaults with the system to be executed on being specified as the **dmc**). These scripts will systematically execute each version of the program produced by the targeted compiler multiple times by sweeping through data sizes of 200 to 5000 in increments of 200, where the data size is equal to the square dimension of the matrices are (i.e. $l = m = n =$ data size). The size information is passed to the program by the script using a single command line argument that each version of the program is to interpret as the size of all three dimensions of the input/output matrices. Both scripts are designed to execute the serial case and the parallel MPI representation for 2, 4, 6, and 8 MPI process implementations. They produce separate outputs for each compiler that is employed and for each implementation. After this data has been successfully generated students are to analyze the performance data that is generated as part of their homework report.

a. For the gnu compiler generated single process (serial) program, graph the runtime characteristics versus data size. How does the base algorithm behave? What is the algorithmic **order** of this algorithm in terms of its common dimension, l? Does the runtime data support this?

b. For the intel compiler generated single process (serial) program, graph the runtime characteristics versus data size. How does the base algorithm behave? What is the algorithmic **order** of this algorithm in terms of its common dimension, l? Does the runtime data support this?

c. For the gnu compiler generated output, graph the runtime characteristics versus data size for the 2, 4, 6, and 8 MPI process implementations. Create four separate graphs. How do these implementations behave as the data size is increased? What is the algorithmic order of these implementations?

d. Using the gnu compiler generated single process serial program as the base, on a single graph, show the Relative Speedup versus the data size for each of the 2, 4, 6, and 8 process MPI implementation. Also create a graph that shows the Relative Efficiency versus the data size for each of the 2, 4, 6 and 8 process MPI implementations.

e. For the intel compiler generated output, graph the runtime characteristics versus data size for the 2, 4, 6, and 8 MPI process implementations. Create four separate graphs. How do these implementations behave as the data size is increased? What is the algorithmic order of these implementations?

f. Using the intel compiler generated single process serial program as the base, on a single graph, show the Relative Speedup versus the data size for each of the 2, 4, 6, and 8 process MPI implementation. Also create a graph that shows the Relative Efficiency versus the data size for each of the 2, 4, 6 and 8 process MPI implementations.

g. What might be some of the reasons the intel compiler produced significantly different performance results than the gnu compiler? List at least three valid possibilities in your answer.

Students are to upload a single document file on the HW3 Assignment on the UAH Course Canvas^{TM} site on or before its due date. Embedded in the final document submission should be the source code of the MPI multiple process SPMD program that was created by the student. *Due date is Wednesday September 30, 2015 by 11:59 PM.*