ECE 4730/6730 Programming Assignment
Due Date Specified on Canvas
Matrix Multiply Parallel

Task:

This is the serial and **parallel Cannon's** versions of a standard O(n^3) Matrix Multiply.

Write the following programs:

1) **make-matrix -r 100 -c 100 -l 50 -u 500000 -o output_file**

This program writes to output_file a binary file with an n by n (square) matrix representing one of the matrices to be multiplied, where the value of each element of the matrix is randomly generated based on a uniform random variable

-r <rows> number of rows in the matrix

-c <cols> number of columns in the matrix

-l <low> lower bound on the values in the matrix

-u <up> upper bound of the values in the matrix

-o <fname> output file name

Data is 64-bit double precision floating point. For each value generate a random integer between 'l' and 'u', convert to double, and then divide by 1000.0

```
double v = ((double) l + ((double)random() / (u – l))) / 1000.0;
```

The first two 32-bit integer words of the file should be 'r' and 'c' the number of rows and columns of matrix, and the data should be stored in row major order.

2) **print-matrix input_file**

Reads a matrix file created with 1) and prints in ascii.

3) **mm-serial input_file1 input_file2 output_file**

Reads 2 input matrix files and computes the product of the matrix in input_file1 with the matrix in input_file2 (remember in matrix multiply, order matters), and then writes resulting product into the output file. The columns of input_file1 and the rows of input_file2 must be equal, otherwise print an error and stop. This may be implemented using the traditional algorithm or a blockwise algorithm.

3) mpirun -np __ **mm-parallel input_file1 input_file2 output_file**

Reads 2 input matrix files and computes the product of the matrix in input_file1 with the matrix in input_file2 (remember in matrix multiply, order matters), and then writes resulting product into the output file. This method will implement the Cannon's matrix multiply discussed in the book and in class. You may require the number of rows and columns of the matrices to be evenly divided by the square of the number of processors (which should be square). You must use an MPI 2D Cartesian topology to manage communication between tasks. Your 2D matrix should be represented in memory as presented in class and in the book with an array of doubles and an array of pointers to double.

Turn in Instructions:

All your source files should be in a single directory. The name of that directory should be

first_last_HW06. Inside that directory, you can have the files called what you want, along with all of the source and a Makefile to compile down to the executables mentioned above. What I'm saying is that there should be a Makefile, where I can simply type make, and everything will be compiled and linked down to what is specified. Also, all of your source files (mains, source, and headers) should have a comment block at the top that has your name, and other information. Also, all programs must be documented well. You should have a report in a text of pdf that describes the project and your results. The zip (or .tar.gz) should have the same name at the underlying directory. Upload on Canvas by the specified time.

Project Report:

Use your sequential program to generate results that you can compare with your parallel results for correctness and to compute speedup. You'll need to submit (in your zip file) a report that addresses various performance metrics of the code. I expect that you run your program on 1 (the sequential program,4, 9, 16, 25, and 36 CPUs at least and try to get timings for 49, and 64 if possible. You should test your code with 100, 200, and 300 elements per row/column per processor. In other words, test 200x200, 400x400, 600x600 on 4 tasks; 300x300, 600x600, and 900x900 on 9 tasks, 400x400, 800x800, and 1200x1200 on 16 tasks, and 500x500, 1000x1000, and 1500,1500 on 25 tasks etc.. After you collect these two sets of data, you should process it in a spreadsheet and create some graphs of, execution times, speedups, and efficiencies across the ranges of processors you used.

Plot all of these points on a graph. For each matrix size, connect the points into a curve with the speedup verses P, one line for each matrix size NxN. Comment on the apparent scalability and practical speedup value of your code within the confines given.

Do not include IO in your timings. You should choose n (number of nodes in the directed graph) to be fairly large, so that when you run the program on the largest number of CPUs, it still take at least 10 or 15 seconds, because otherwise it will run too quickly, and the timing measurements will not be trustworthy. Also, on the sequential version, you should run this using PBS as well, using a PBS script, so that you can compare apples to apples (if you run it on the head node, other people are using that, and the timing measurement will not be accurate).

As with the previous project, most likely when P gets to 36 speedup will go down, maybe even negative. Try an alternative allocation of processors and report on your results:

#PBS -l select=4:ncpus=9 …
#PBS -l select=6:ncpus=6 …
#PBS -l select=9:ncpus=4 …

Do these 3 different ways of allocating 36 processors give different performance results? If so, why?

DONT WAIT UNTIL THE LAST MINUTE TO RUN THIS ON PALMETTO AND EXPECT TO GET IT ALL DONE IN A FEW HOURS!!!!!