SC 290: High-Performance Computing  
Spring 2015  
Homework #6  
Due Date: Friday, March 13 (before class)

**Total Points Possible:** 20

1. Research software packages used in your field that you are interested in running in parallel, or algorithms in your field that you want to parallelize (with PThreads, MPI, or CUDA). Describe the package or algorithm. If you're focusing on parallelizing an algorithm, write some pseudo-code that demonstrates how you hope to achieve parallel processing. If you're focusing on a software package, describe (or speculate if you can't find sufficient information) how the package achieves parallel processing. (2 points)

2. Starting with your matrix multiplication code from homework #5 (save a copy of the original), develop a matrix multiplication program that runs in parallel using Posix Threads. You will once again be computing the product of a square \((n \times n)\) matrix with itself \((A = B^T B)\). Once you have computed the product matrix, your code should compute the sum across all elements in the product matrix in parallel. More details are provided below.

**Command line arguments:** Your program should be able read in a text file (the name of which should be passed from the command line) containing a \(n \times n\) matrix \((n\) integers per line, \(n\) lines) and copy the data into a dynamically allocated 1d array (like hwk #5). Optionally, if you do not supply a matrix in a text file, your code should fill out the matrix array with random integers between 0 and 20. You should also give your program the ability to control whether it outputs the matrix to stdout. Finally, pass the number of threads you would like to run your program with from the command line. The idea is to be able to quickly test and benchmark your code without recompiling. You can do this however you’d like, but I’d recommend using flags:

**Example 1:** read in matrix from file, \(n=50\), spawn 4 worker threads, write product matrix to stdout  
./pthread_matrix_multiply -f 50_50_matrix.txt -n 50 -t 4 -o

**Example 2:** generate random 50x50 matrix, 2 worker threads, do not write results to the screen  
./pthread_matrix_multiply -n 50 -t 2

See `fillOutArray()` for an example of creating a matrix with random ints:

Since you are passing the number of threads from the command line, your pthread_t objects will need to be allocated dynamically. This can be done like a dynamically allocated array (use malloc() and free() functions).

The worker thread function should do two things:

- Compute the product matrix in parallel. Each thread should be responsible for computing the same number of elements in the product matrix (or close to the same number, if the matrix size is not evenly divisible by the number of threads). This is most easily accomplished by assigning each thread a unique range of indices that it is responsible for computing.

- Compute the sum of all elements in the product matrix in parallel. Specifically, the worker threads should compute the sum across each of the rows in the product matrix first, and then add those results together into a global sum (yes, I realize this is not the most efficient way to accomplish this!). Each thread should be responsible for the same number of rows (or close to the same number, if the number of rows in the matrix is not evenly divisible by the number of worker threads).

Run extensive tests on your code to ensure that it is producing consistent results with the serial version.

Run benchmarks of your code, and generate plots for speedup and efficiency as a function of matrix size and number of worker threads. For the benchmarks, you can use the Linux “time” command (e.g. time ./pthread_matrix_multiply -n 100 -t 2) to measure the execution time, generate a random matrix (do not read from file), and do not output the product matrix to stdout. Test the following conditions:

threads = 1; n=1000,5000,10000,20000,50000,100000,200000
threads = 2; n=1000,5000,10000,20000,50000,100000,200000
threads = 4; n=1000,5000,10000,20000,50000,100000,200000
threads = 8; n=1000,5000,10000,20000,50000,100000,200000

n=5000; threads=1,2,4,8,16,32
n=20000; threads=1,2,4,8,16,32
n=50000; threads=1,2,4,8,16,32
n=200000; threads=1,2,4,8,16,32

Comment on the performance of your code.

Point me to the file location. (18 points)