Mark Klick

CS 315 Final

**Matrix Multiplication**

**Introduction:**

I learned a great deal and had a lot of fun working on this final project for several reasons. Firstly, I spent a fair amount of time trying to parallelize my bioinformatics mutual information algorithm but then realized that the F# language already utilizes lazy evaluation and is already inherently parallel. F# has a Pseq (parallel sequence) module, but using that on top of the piping operator and recursive folds seemed unnecessary. When I did use Pseq there was no apparent speed-up anyway. Then my original backup plan was to expand the image processing program we made as a class by adding several more image filters and implement them using CUDA and the GPU. However, I realized that in another bioinformatics algorithm I was working on, written in R, there were a lot of matrix multiplications. This gave me the great idea of implementing a simple serial algorithm for matrix multiplication using c++11 and see how many different ways I could parallelize it and make it faster. Multiplying matrices are used most commonly in linear discrete systems, e.g. recurrence relations, and have many practical applications in graphical processing e.g. filters (sobel filter!), scaling, adjacency matrices of directed and undirected graphs. As I was developing the first variation of the matrix multiplication function I realized that there was another, more efficient way to multiply two matrices which is to first transpose one of the matrices and essentially vectorize the multiplication using the dot product. This improved the initial serial function I had written and allowed me to parallelize two different functions and analyze the speed-ups associated with the parallelization of these functions.

**Implementation:**

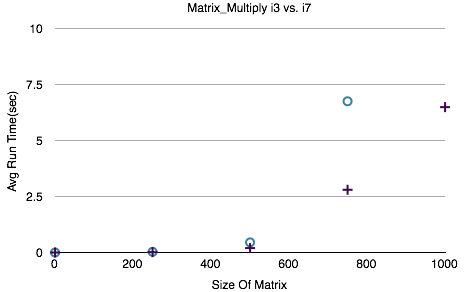
I was able to successfully parallelize both versions of matrix multiplication using C++11’s parallel\_for method and the OpenMP API. The strategy was simple enough in that the outer-most for loop in each multiplication method was the one I would want to run in parallel. After the coding was completed, it was time to test the run-time’s for each serial and parallel versions and compute the speed-ups! For my project, I also compared the run times and speed-ups between two processors, a 3.6GHz Intel i7-3820 and a 1.5GHz i3-2377M, when executing my program.

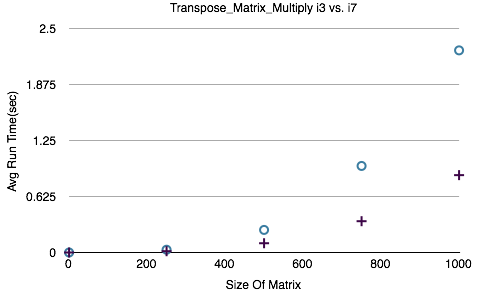
**Data & Analysis:**

After I executed the program on the i7 processor and i3 processor, there was no surprise that the i7’s runtime was faster for both serial versions of the multiplication, the question is how much faster. For smaller matrices the serial run-time between both processors was more similar for both versions. But as the matrices size increased, the run time for the i3 increased at a much higher rate than i7 as can be seen in the graph presented below.

The next question I was trying to answer, was what the speedup of my parallel versions of the functions were, and was this speedup going to vary between the i3 and i7 processors and size of matrices being multiplied. The run time of each parallel function was tested a 1000 times to ensure accurate values for computing the speedup. For the PMatrix\_Multiply function, the avg speedup was about 3 for the i7 and about 2 for the i3. For the PTranspose\_Matrix\_Multiply function, the avg speedup was about 7 for the i7 and about 2.2 for the i3. However, both processors had the lowest speedup for 500x500 multiplication which was weird and I’d like to find out why in my future work. Also, for my future work I’d like to implement my functions using CUDA. Overall, I really enjoyed this project and had a lot of fun testing my program on my own machines at home.

**Average Runtime Graphs:**





**Speedup Graphs of Parallelized methods:**

