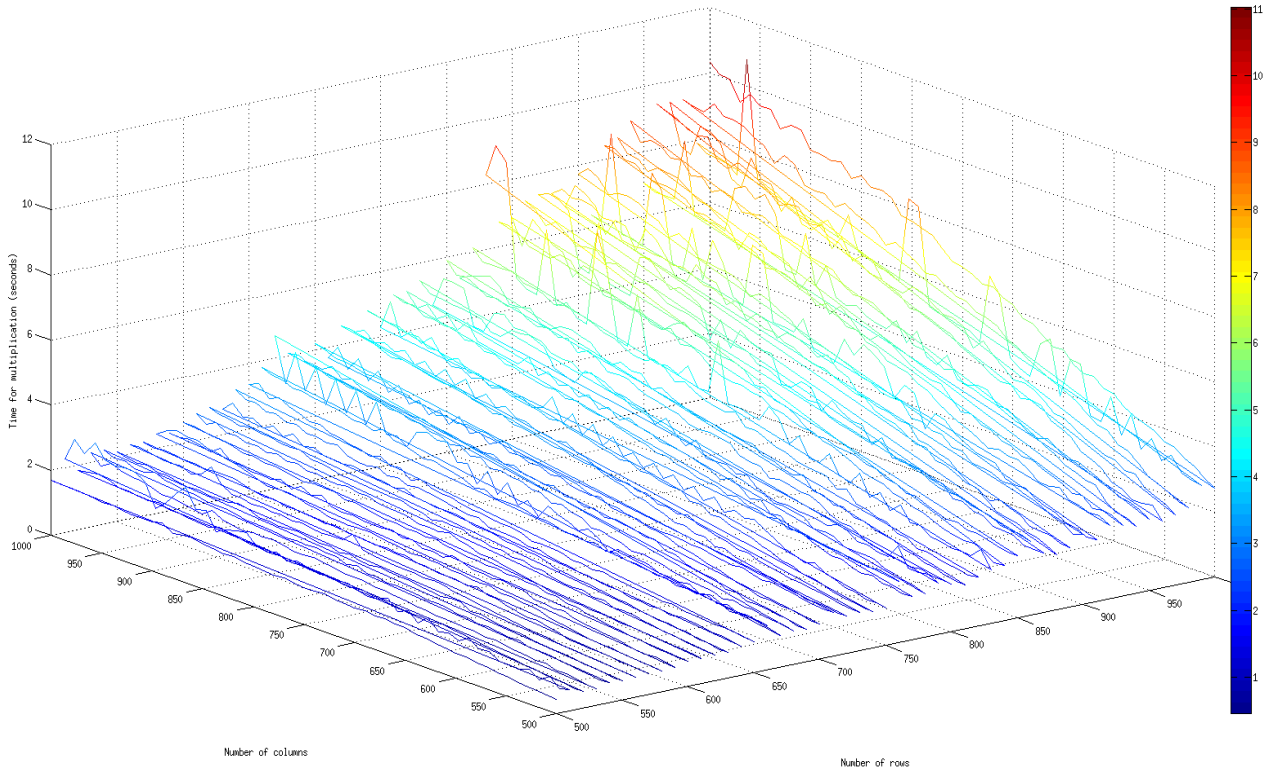
Stephen Ranshous
COP4331 Bonus Project
DGEMM

For this project I used Open MP for parallelism. There was no installation required and the GNU C compiler handled it perfectly fine. My parallelism strategy was very simple, as this was just a proof of concept for myself and a shallow look into how to use the OMP directives. The problem we are looking as is multiplying two matrices, A and B, to produce a third matrix, C. C = A * B.

**Straight-Forward Implementation**

In the first implementation of the DGEMM I use a single threaded, naïve approach to multiply the two matrices together. It uses the traditional approach of three nested loops going row major order across A, and column major across B. As a quick aside another way to I could have drastically reduced the run time of this program would be to switch around the for loops and do partial summations all the way, taking advantage of C's row major ordering. Using row major order for both A and B would have greatly increased cache hits. I chose not to do this, however. These are the results of the naïve implementation.



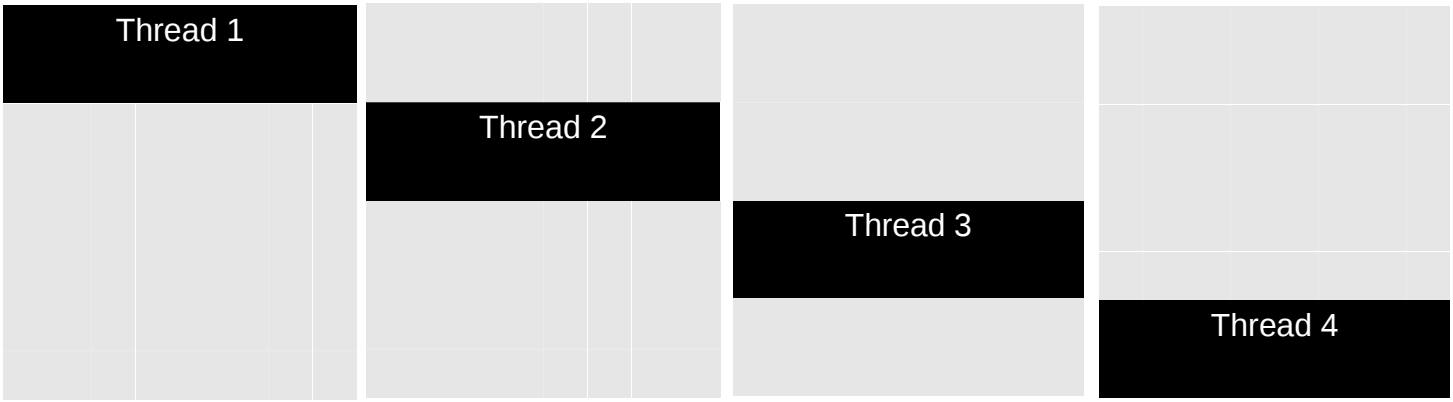

This picture is hard to see in this report but is included in the submission package so you can view it there. Instead for the report I'll describe it. The two axis on the bottom are the row and column sizes for the matrix A in the multiplication, B is therefore the opposite. It ranges from 500 up to 1000 for each, incrementing by 10 each time. For every number of rows it runs the multiplication using 500 up to 1000 columns, then increments the rows.

```
for i in {500..1000..10}
do
     for j in {500..1000..10}
     do
          matrix_multiply with i rows and j columns
     done
done
```

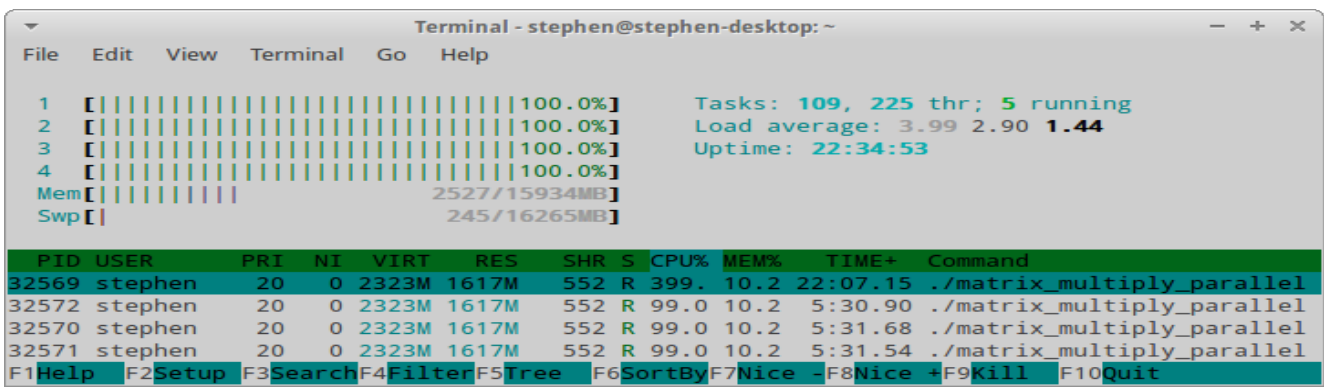
The run time increases from a fraction of a second at 500 rows by 500 columns up to almost 12 seconds to perform a 1000x1000 matrix multiplication.

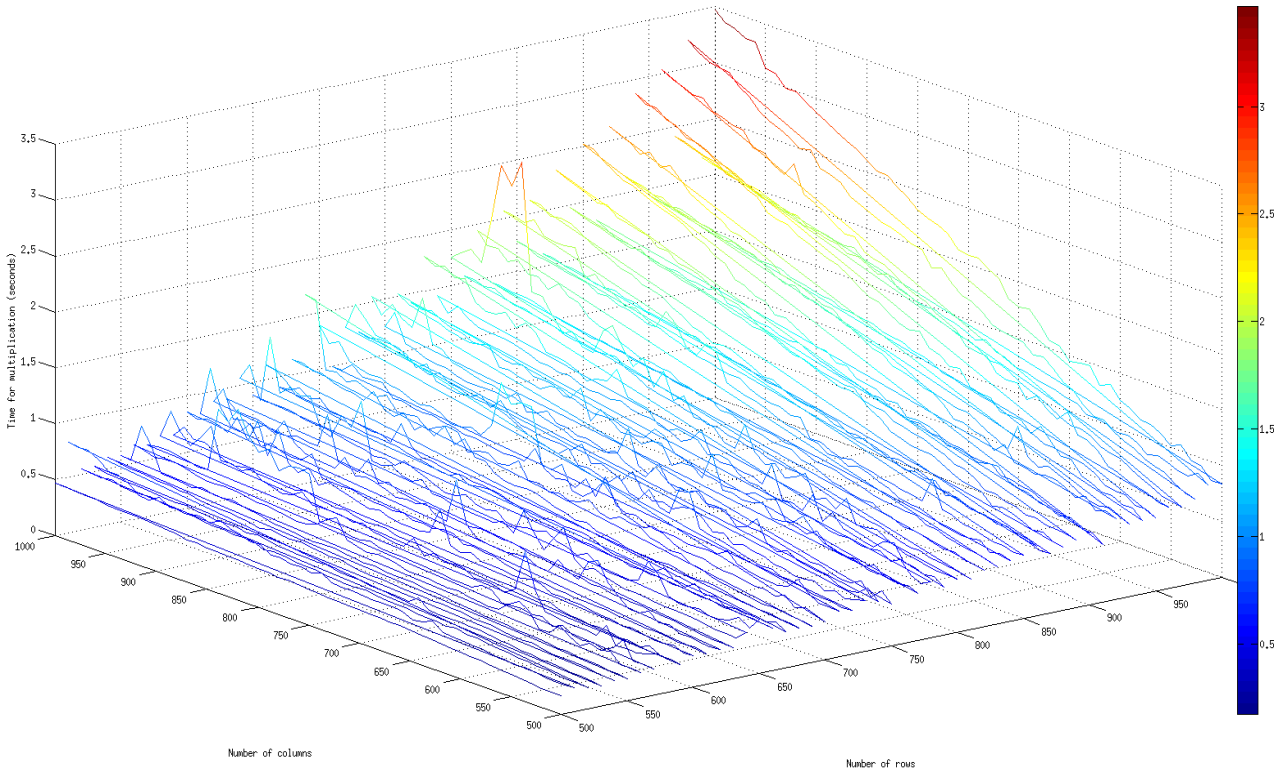
**Parallelism at work**

For the second implementation I used a very basic parallelism strategy. I stuck to using mostly the default values for the directives for OMP because they seemed to be what I would choose anyway. By default it will create as many threads as there are cores or processors. On my desktop at home this is 4. I left it at this because if we created more threads than there were processors, then the threads would just be fighting each other for run time and that may end up being even worse. The way I chose to parallelize it was to break the number of rows in the resulting matrix into however many threads we have (4 in my case), and each thread would handle those rows. One thread would be responsible for something like this:



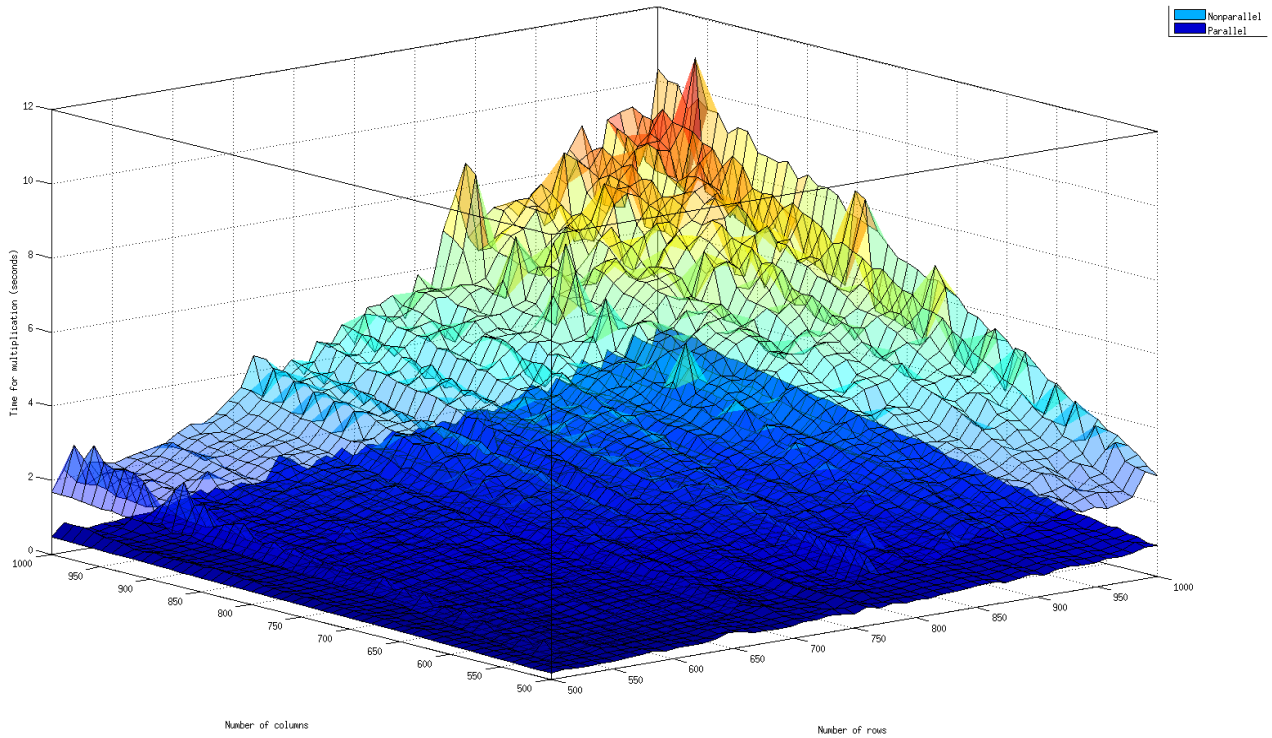

As can be expected this pounded on my CPU hard each time. Heres a view from htop.

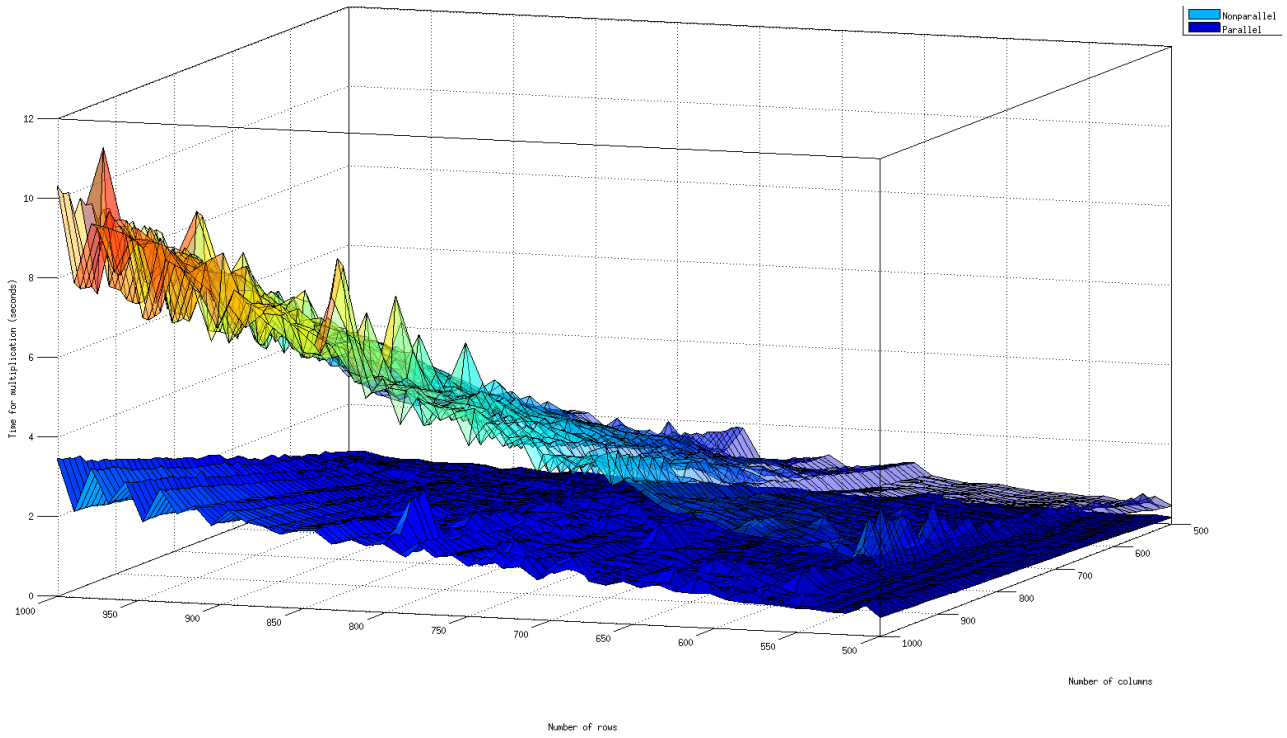





The results from running it in this manner are as I expected. The total run time dropped quite substantially for all sizes. Where the maximum on the nonparallel version was around 12s the maximum for the parallel was around 3.5s. This makes sense since the division of work was each thread does about 25% of the work, all working at the same time, so the total run time is now cut into ¼ the run time from the serial version, plus a little overhead to manage the threads. Here are the official results.

The next page shows a comparison of the run time outputs.




All images were generated using Matlab. All the source and images for this project can be found at https://github.com/sranshous/Parallel-vs-Nonparallel-DGEMM