Concurrent Systems Lab #1

Group Members: Aaron Joyce & Neil Barry-Murphy

Objective:

The objective of this lab was to create a method that would concurrently calculate the resulting matrix of a matrix multiplication operation between two matrices, where the number of rows of the first matrix are equal to the number of columns of the second matrix. Our goal was to significantly improve the performance of this method when compared with another similar method that did not utilize aspects of concurrent programming, such as threading and vectorization.

Procedure:

There were many ways in which we could have approached this task, but we ultimately decided to simply enable concurrent threading of our new method, and the transposition of the second matrix so that its dimensions would be stored in cache before any actual matrix multiplication operations were completed.

Our newly created method; “team_matmul()” uses the same algorithm as the matmul() method, but enables multiple threads to access the loops of the method concurrently. This is achieved simply by using the line:

```
#pragma omp parallel for collapse(2)
```

Parallel concurrent threading has now been enabled for this method. The “for” refers to the fact that the upcoming for() loop can be

accessed concurrently, while the collapse(2) statement allows for the following two for() loops to also be accessed concurrently.

It is important to note at this point that the machine used to calculate the resulting matrix and potential speedup factor was the cs stoker machine.

The next stage of this process was to calculate the transposition of the second matrix involved in the multiplication operation. This was achieved using the “transpose()” method. (Please refer to attached c file). Following the conversion of the matrix, the accesses to each row and column to the second matrix had to be reversed. This means that at any stage where a **A[k][j].real** or **A[k][j].imag** was encountered, it was replaced with a **temp[j][k].real** or a **temp[j][k].imag**, where **temp = transpose(B, temp, a_cols, b_cols);**

We then tested many matrix multiplication combinations, but we found that the performance of our method improved greatly with the increase of each matrices' dimensions.

For example:

```
Matmul time: 186060 microseconds
control time : 11630369 microseconds
speedup: 62.51x
barrymun@stoker:~/college-repo/conc-sys$ vi alt.sh
barrymun@stoker:~/college-repo/conc-sys$ cat alt.sh
clear
gcc -O3 -msse4 -fopenmp matrices.c
./a.out 1000 1000 1000 1000
barrymun@stoker:~/college-repo/conc-sys$
```

It is evident from this screen dump that the speedup factor is quite significant for two square 1000 x 1000 matrices. However, when we attempted to multiply two square matrices of size 100 x 100, we found that the speedup factor dramatically decreased.

Conclusion:

To finish, it is worth mentioning that we attempted to vectorize the function, but we noticed some unusual performance dis-improvements. Our speedup factor actually decreased to less than 55 when calculating the multiplication of two 1000 x 1000 matrices, so after much deliberation, we decided to leave this attribute of concurrent programming out of our equation.

Please refer to the attached c file if any of the above explanation is unclear.