CS 475/575 Final Project
Blake Davis, Chris LaBerge, and Tyson O'Leary

Introduction:

We chose to improve a k-means clustering algorithm. This algorithm randomly chooses locations for a specified number of cluster centers and then clusters the provided data based on its distance to a cluster center. It does this for 500 iterations and the data is then properly clustered. This project matters because machine learning models often use large amounts of data that take a long time to process. If we can speed up that process, we could save a lot of time and money for a company or research group that is using the k-means clustering algorithm to analyze data. The code we used is from the Department of Electrical and Computer Engineering at Northwestern University by Wei-king Liao. The code included a Makefile, some timing and I/O files, with a Sequential, and OpenMP implementation of the program.

Analysis:

The sequential program has 137 lines of code. It ran 29400 data points with 5000 clusters with an execution time of 16.20 seconds with 13.22 GFLOPS. The intel tool performance is shown in image 1.1 below. The given OpenMP code has 251 lines of code with an execution time of 2.78 seconds and 76.97 GFLOPS as shown in the roofline performance image 1.2.

Image 1.1

Image 1.2

Improvements:

We initially started by analyzing the code to understand what it was trying to accomplish and how it worked. Our original plan was to parallelize the k-means code using CUDA, but we soon realized that this would be a monumental task that we would not be able to complete in a reasonable amount of time. This then decided to move to improving the time of the given OMP code. We started by switching the loop that goes through each cluster and the loop that goes through the dimensions of the current data point. To do this, we had to store the partially computed distance of each object from each center, which would be updated as we looped through the dimensions. This extra array is allocated for each thread, which adds memory overhead. This slowed our program down, even with compiler optimization. We then tried to transpose our matrix to make vectorization possible and increase cache locality. After implementing this seemingly correctly, we hit a roadblock in the form of wrong answers. With Sanjay's guidance, we removed a simple optimization that replaced redundant memory accesses with a variable, which was causing our problems. We theorize that it was an issue because we didn't make the variable private in the OpenMP pragma, so all threads were reading and writing to it at the same time. After this fix, our parallelization approach was successful and we

experienced a much larger amount of GFLOPS compared to the given OpenMP code. This showed that all of our hard work paid off and we all experienced much joy.

The roofline chart below (Image 1.3) shows the resulting performance of the improved OpenMP code, attaining an execution time of 1.26 seconds and 176.59 GFLOPS. The execution time and speedup charts in Images 1.4 and 1.5 show a clear improvement over the given Sequential and OpenMP code.

Image 1.3

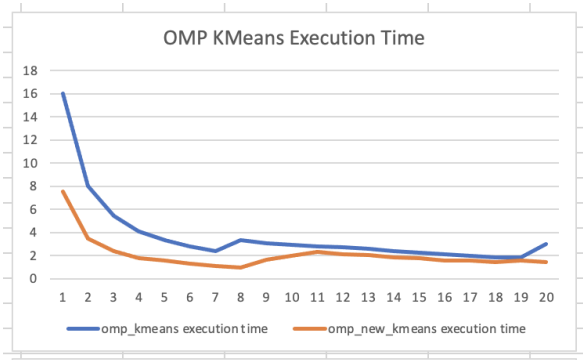

Image 1.4

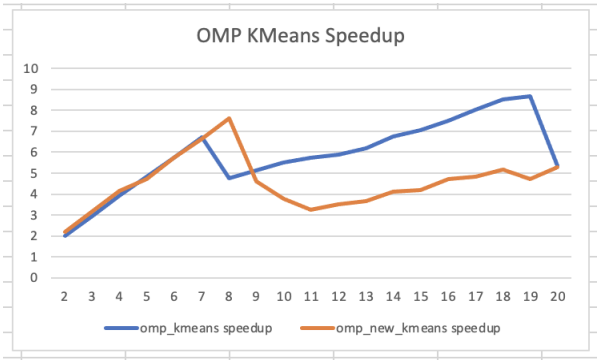

Image 1.5

Conclusion:

Similar to all parallel programming we have seen this semester, memory adjacency was key to optimizing this code to run in an efficient and accurate manner. As mentioned above, we found huge improvements in execution time and speedup by transposing the matrix in order to access data along each row, instead of along each column. Debugging this project proved to be challenging because tools like Valgrind do not work in parallel code. Our best-found practices for parallel debugging were including print statements to monitor thread id and variable behavior.