Akshat Patni

Application of OpenMP to parallelize K-means Clustering Algorithm in C++

Abstract

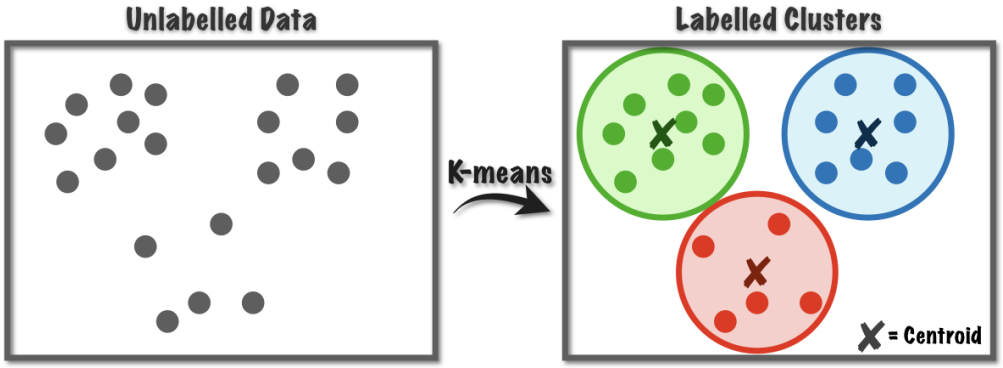
In this study, I combined OpenMP parallel programming methods with data science concepts in order to make an algorithm that can perform an analysis on exceptionally large datasets without compromising speed and performance. The analysis performed in this algorithm is k-means clustering, a methodology in data science that takes unlabeled/uncategorized data points and groups them by similarity factors and features into 'clusters'. These clusters are made based on magnitudes of 'distance' between data points in cartesian space so the ones that are closer together are more similar. K-means clustering is an example of unsupervised learning. The algorithm was written with links to run under different versions and conditions. The program can run as a serial or parallel version, and it can either read in an input.txt file with a formatted list of data coordinates or the algorithm can generate N number of random points itself to use. The main parallel region in the program is a region containing for loops that iterates through the data points and centroids, updating clusters locally with new points and moving reclassified points into the correct cluster, and finally merging everything to the main clusters object. I performed strong scaling and weak scaling experiments in order to determine the speedup of the parallel algorithm in comparison to the serial/sequential version, and how that scales with variables such as problem size, number of clusters, and the number of threads/cores that the computational work was divided into. The results demonstrated a substantial amount of speedup in the parallel version as compared to the serial version. The parallel version performed better than the serial version and passed tests under both strong scaling and weak scaling conditions. Using the valgrind tool, I analyzed the program for instances of memory leakage and found it to be fairly negligible. Overall, the parallel algorithm was successfully optimized and delivered desired outcomes for parallel performance of k-means clustering.

Introduction and Background

K-Means Clustering

Machine learning is a set of methodologies that allows us to analyze datasets with computer programs that automate certain tasks, manipulating or classifying data with only implicit instruction. Supervised learning requires labeled and categorized data, while unsupervised learning uses completely unlabeled data to identify patterns and perform tasks.

K-Means clustering is a low level application of unsupervised learning. First, it requires the initialization of a predetermined number of clusters (k), and arbitrary centroids in each cluster. The algorithm would then iterate through all of the points, calculating the Euclidean distance between each point and the centroids of the different clusters to identify which cluster it is "closest" to in Cartesian space. With each successive assignment for each point, the centroids and clusters are updated as the means change. As it runs through more points, the clusters become more defined. The final output is a set of k pre-defined clusters in which all of the points are labeled and grouped by similarity in position on the Cartesian plane. Clustering can be performed on higher dimensions to group datapoints based on multiple factors, however for the purposes of this study I have written a naïve 2-dimensional k-means clustering algorithm.



Objective

The aim of this study was to first develop a serial algorithm that performs k-means clustering. The serial program iterates through all of the points sequentially and calculates the closest centroids to assign them to clusters. I then parallelized this program using OpenMP so that the loop iterations can be divided across a number of threads that each do their computational work and then collapse all threads together to the final output. The desired outcome is that the parallel implementation of the program results in a substantial speedup of k-means clustering compared to the serial/sequential implementation. I expect that if the parallel implementation is successful, the magnitude of the speedup would be linearly correlated to the number of threads that are used since each thread is performing the exact same work.

Algorithm Design and Methodology

I implemented both the serial and parallel versions of the program in the same kmeans.cpp file with different links and command line usage depending on which one is being used. I designed the program to be able to generate N number of random points to be clustered (for testing/performance purposes), or to take in a standard input stream (stdin) from any input.txt file with a list of 2D datapoints (for application/implementation purposes). The input file method allows us to apply this program to essentially any 2-D dataset, as long as the points are numeric.

One of the primary functions applied to the points is sqr_dist() to calculate the Euclidean distances. Since we only care about comparing the distances and not the absolute values, I saved time by omitting the sqrt() call as well as using dx*dx instead of pow(dx, 2) for speedup purposes. This was among a number of smaller serial optimizations applied to save computational time.

The most time-consuming kernels of the program are the for-loops that read in points/centroids and performs the assignments of each points to the clusters, using the find_closest() function. It also required for some of the previously clustered points to be re-assigned as the centroids and effective cluster boundaries were updated. I parallelized this region using #pragma omp for, dividing the work statically amongst each thread (each thread executed the same amount of work). In my first version of the program, I failed to achieve a speedup because all of the threads were writing to the same cluster array. Each thread had to update the main clusters array for each point, causing too much overlap and overwriting between threads which prevented a speedup. It was important that each thread had its own copy of data in the cache, not interfering with cache data with other threads.

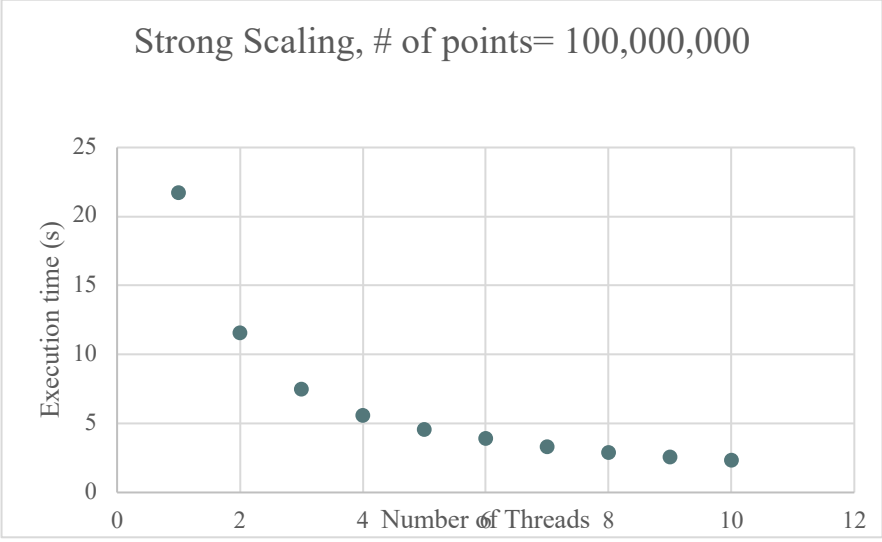
Subsequently, I structured the code of the parallel version differently from the serial version. I included a new vector of local clusters for each thread, using slightly more memory which is ultimately negligible. In the first for-loop, I only updated the local clusters. This region was marked with #pragma omp for. When a thread has iterated through all of its points, it copies its data from the local clusters array into the main clusters array in a subsequent for-loop. This for loop is marked with #pragma omp critical, to get a lock on the main clusters object so that no other thread can update it while a single thread is working. This section can only be executed by a single thread at a time. The cluster merging for-loop is much faster than the previous clustering

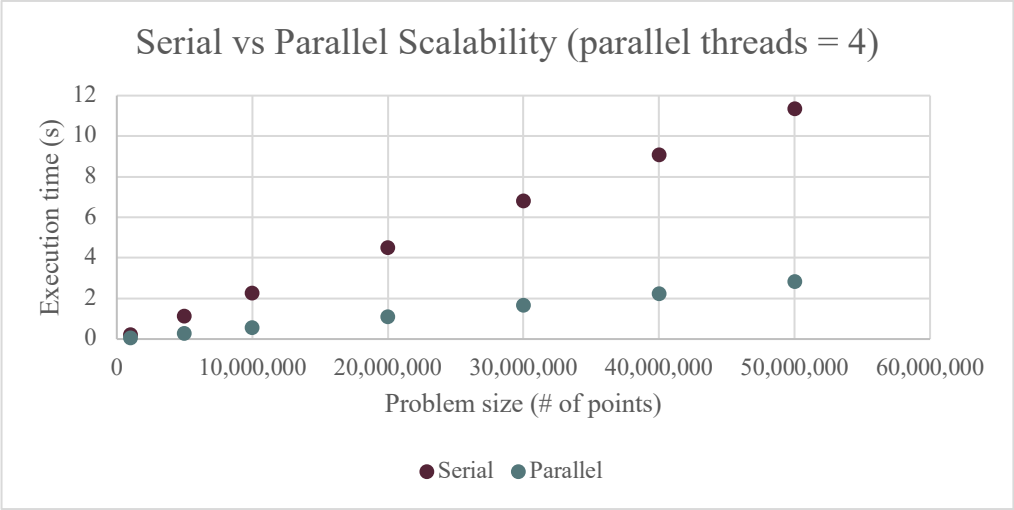
assignment for-loop because it only iterates through a small number of clusters, compared to the other loop iterating through all the (millions) of points. Therefore, the cluster assignment for-loop was the most time-intensive region of code in the program and had to be parallelized including the local clusters vectors.

Experiments and Performance Testing

In order to test the performance of my parallel k-means clustering algorithm, I performed strong scaling and weak scaling experiments to determine if there was any speedup compared to the serial version, and how that would scale with increases in problem size (increasing N number of points) and increases in number of threads. In the strong scaling experiment, I increased the number of threads while keeping the problem size constant. In another scaling experiment, I increased the problem size while maintaining a constant number of threads (and compared it to a scaled serial version). I then performed a weak scaling experiment to test performance as I increased both problem size and number of threads. I also used the valgrind tool to analyze memory leakage.

Results





Weak Scaling

Num_thread	Problem_Size	time
2	20,000,000	2.31
3	30,000,000	2.225
4	40,000,000	2.227
5	50,000,000	2.24
6	60,000,000	2.23

Discussion and Summary

When testing the performance of the parallel k-means algorithm, I found the speedup of the execution to work almost exactly as desired. In the strong scaling experiment, increasing number of threads reduced execution time significantly although it seemed to slow down approaching a limit of about 2 seconds, which is probably the time cost of all of the non-parallel overhead code. Comparing the scalability of the serial program compared to the parallel version, the execution time was almost perfectly linearly related. With 4 threads compared to 1, the execution time for the parallel version was always around $1/4^{th}$ of the time for the serial version. With the weak scaling experiment where I increased both the number of threads and the problem size, I got an expected result where the execution time stayed the same. This is because the workload remains constant for each processor, whereas in the strong scaling experiment the workload becomes reduced for each successive additional thread. Overall the performance of the k-means parallel algorithm satisfied my initial hypothesis, and successfully fulfilled the objective of linearly reducing execution time via OpenMP parallel threading.

References

1. Dabbura, Imad. “K-Means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks.” *Medium*, Towards Data Science, 10 Aug. 2020, towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a.
2. Jeffares, Alan. “K-Means: A Complete Introduction.” *Medium*, Towards Data Science, 19 Nov. 2019, towardsdatascience.com/k-means-a-complete-introduction-1702af9cd8c.