HW3

Delibas, Sabirenur
sabirenurd@gmail.com

Ulkutasir, Batuhan
ulkutasir.batuhan@gmail.com

May 2024

1 Introduction

In this report, parallelization and increasing performance of the Lloyd's clustering algorithm which is used in many applications such as advanced data analytics and machine learning is presented. This clustering algorithm is an effective and fast way of clustering the data however, serial algorithms may result in long execution times while dealing with enormous data sets. In order to solve this problem, parallel computing techniques, especially libraries such as OpenMP, can be used to optimize processing processes. In this study, the development of Lloyd's clustering algorithm with a parallelized version with OpenMP is achieved. The aim is to make the algorithm run faster on large data sets by using the capabilities of OpenMP. The implementation details will be explained in detail in the further sections. Also, the results of the developed algorithm are provided for different data points, number of clusters, and dimensions, and lastly thread numbers and the obtained results will be compared with each other. At the end of the report, the effects of parallelization are discussed, and the capability of OpenMP will be understood.

2 Implementation Details

OpenMP is an API that facilitates parallel programming and can be used to parallelize a serial code. A serial k-means algorithm involves a series of loops in which data points and centroids must be processed collectively. The implementation of OpenMP on a serial code is described in this section.

Parallel Loops: There is parallelization in the assignPoints and updateCentroids functions' loops. The *#pragma omp parallel for* the directive is used to parallelize loops across threads. This shortens processing times and distributes the effort among threads. Since a single thread processes each loop iteration, many threads in a loop can operate concurrently.

Critical Regions: The *#pragma omp critical* directive was used to specify the critical region in order to address the issues with simultaneous access to the Ci and Ck arrays in the assignPoints function. This resolves the race situation problem by limiting the number of threads that may access these arrays at once. By limiting the number of threads that may operate concurrently, the crucial area preserves data integrity.

Atomic Operations: The updateCentroids function's whole set of operations was parallelized using the *#pragma omp atomic* directive. This reduces the likelihood of race situations and guarantees that a single thread completes the task at hand. Atomic operations should, however, be employed carefully because they are frequently expensive in terms of performance.

Time Calculation: In your code, you use two timestamps to measure the running time of the k-means algorithm: start and end times. These timestamps are taken at the beginning and end of the algorithm and

the difference between these times is calculated to obtain the total running time. This time is measured using the omp_get_wtime() function. This process is important to evaluate the performance of your code and measure the impact of parallelization.

Number of Threads: When parallelizeng with OpenMP, it is important to determine how many threads to use. The omp_get_max_threads() function returns the maximum number of threads that can run simultaneously on your system. However, this number can be set to a specific value with the omp_set_num_threads() function. Typically, an appropriate number of threads is selected based on the number of physical cores in the system or performance requirements.

3 Results & Discussion

The primary objective of this homework was to utilize multi-threading in our k-means algorithm that we built in the first homework. Naturally, as the number of parallel operations increases, a decrease in operation time becomes evident. We believe that by utilizing multi-threading, we can reduce the costs and time associated with computation. From the initial homework, we have selected two particularly challenging cases: First, the Birch_N100000 file, having the highest number of data points and the longest computation time in a single-threaded operation. Second, the Circle_N4000, which has a singularity at the center causing false cardinality calculations thus, increasing computation time. We believed that OpenMP could demonstrate distinct operational advantages while handling these cases. Based on our results, we have composed Figure 1 to demonstrate the decrease in operation time between the two cases. Evidently, the total number of data points is the main factor behind the computation time. Therefore, two cases having 100,000 points and the other having 4,000, are shown on different axes for increasing clarity.

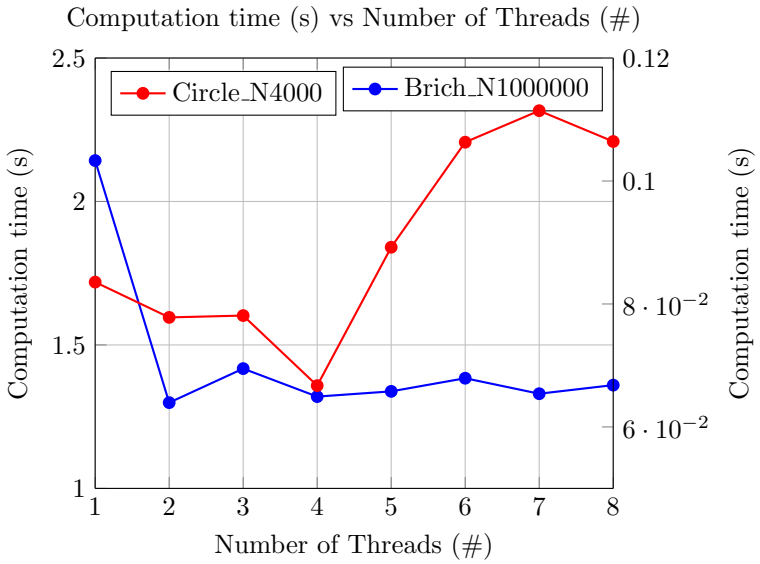

Figure 1: Effect of parallelization by multi-threading on computation time for two distinct cases Brich_N100000(left) and Circle_N4000(right)

Here, It is obvious that using multi-threading reduces computation time when compared to single-thread operation. However, the impact and extent of the decrease also depends on the computational data of the system. The blue line indicates that using a secondary thread decreases time by 50%. However, further usage of higher numbered threads becomes irrelevant since the reduction in time is no longer significant due to the total data amount being the limiting factor. On the other hand, orange graph shows that we can benefit

multi threading up until 4 threads cutting of time by only 10% but further increasing the thread number results in higher operation times. This condition is due to the "over-parallelization" of this smaller data cluster.

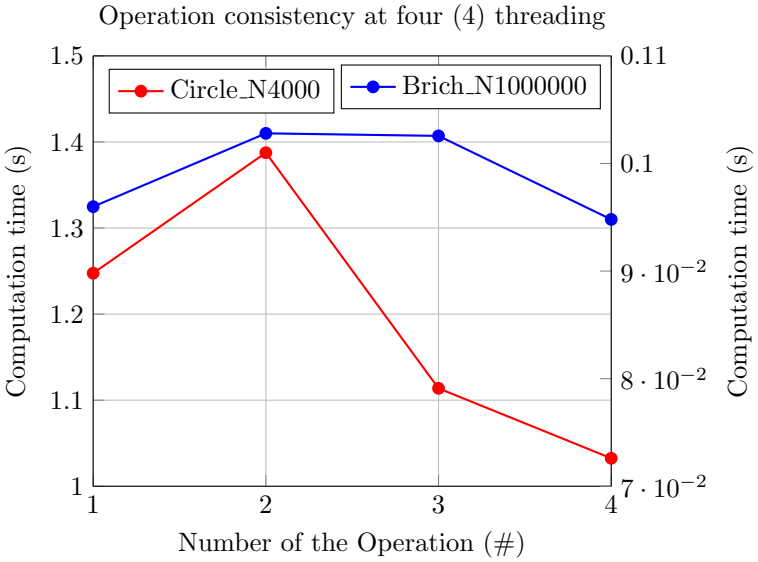

Figure 2: Consistency of the computation time in Open-MP for two distinct cases Brich_N100000(left) and Circle_N4000(right)

We also tested consistency of the computation time as shown in Figure 2 resulting low standard deviation values among themselves. Finally, multi-threading caused no effect on the accuracy of our results showing that our parallelization structured properly which resulting consistent results with HW2

4 Conclusion

In conclusion, this report has demonstrated that our code for implementation of Lloyd's clustering algorithm with Open-MP have been successfully implemented. Reducing computation times by ranging between 10% to 50%. We have also demonstrated over usage of multi threading could be non-beneficial for relatively small cluster of data. Therefore there is always a an optimization of the threading needs to be done considering amount of the data computed. Thus avoiding future problems. Moreover, the accuracy of the results remained same and standard deviation in the computation time resulted too low to mention confirming our operation and algorithm is well structured.