Exploring parallelism on Lloyd's K-means algorithm using C

João Silva
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50495@alunos.uminho.pt

João Torres
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50499@alunos.uminho.pt

***Abstract*—The present assignment experimentally applied different OpenMP directives in order to extract more performance from the Lloyd's K-means algorithm using the C programming language. Some changes were made to the original code in order to increase performance, as well as optimizing it further to work better with parallelism.**

***Index Terms*—C, K-means, Lloyd, Parallelism, OpenMP**

I. Introduction

In this assignment we were directed to explore the effects of parallelism in the previous K-means algorithm. Several OpenMP directives were tested, as well as in-code techniques, in order to get lower execution times.

The stopping parameter for the algorithm has been modified to when it reaches 20 iterations, instead of it finishing whenever an optimal solution has been achieved.

After analysing the scalability of the algorithm, as well as running performance tests, we were able to draw some conclusions regarding the difference between using vs not using parallelism.

II. Performance analysis

An important step when considering how to implement parallelism is to analyse the code and establish where there's more computational load.

In this specific case, there's only one block of code where this load is significant, which is when the points are being assigned to each cluster. In this loop, there's a need to iterate through every point. In addition, in every single iteration, the program also runs another loop to compare the distances between clusters, meaning that for each point, there are K additional iterations (K being the number of clusters). This block of code corresponds to step 2 of the $k-means$ section, described on TP1's report.

III. Changes made to the original algorithm

The way that the data is stored was changed in comparison to the first assignment. Instead of storing points and clusters inside of structures, these are simply global arrays now. This implementation provides better overall performance.

Since the algorithm must stop at 20 iterations, we removed the step where it checked if it reached an optimal solution, since this check is no longer necessary.

In the loop mentioned in the previous section, we just store the value of $clusterIndex$ in an array, instead of summing the value of the corresponding point to $newCentroid$ and updating the $size$.

After this loop is done executing, a new loop is created to iterate through the aforementioned array, in order to update the values of $newCentroid$ and $size$. This makes sure that all values are consistent and it allows us to use the $pragma\ omp\ parallel\ for$ directive.

The if statement was replaced by a ternary conditional operator ($a\ ?\ b\ :\ c$), this resulted in slightly better performance.

Lastly, the compilation flags were changed to "$-std\ =\ c99\ -Ofast\ -funroll-loops\ -fopenmp$". After testing several other flags, these were the ones that ended up providing the best performance.

IV. Applying OpenMP directives

The step of the algorithm that takes up more computational load has been modified with parallelism in mind, so it was only necessary to use the "$pragma\ omp\ parallel\ for$" directive. This directive provided significant improvements in performance. Since the critical areas have been moved to be executed after the loop is done, we managed to avoid the overhead caused by needing to use the $critical$ / $atomic$ OpenMP directives.

The same directive was applied in the loop that resets the cluster's $size$ and $newCentroid$ values. The difference in performance isn't too noticeable but it provides scalability in case the number of clusters increases significantly.

V. Results

The tests were made using the "*srun –partition=cpar –exclusive perf -e instructions,cycles -M cpi -r 3*" command.

TABLE I
Comparing execution times between parallel vs sequential execution

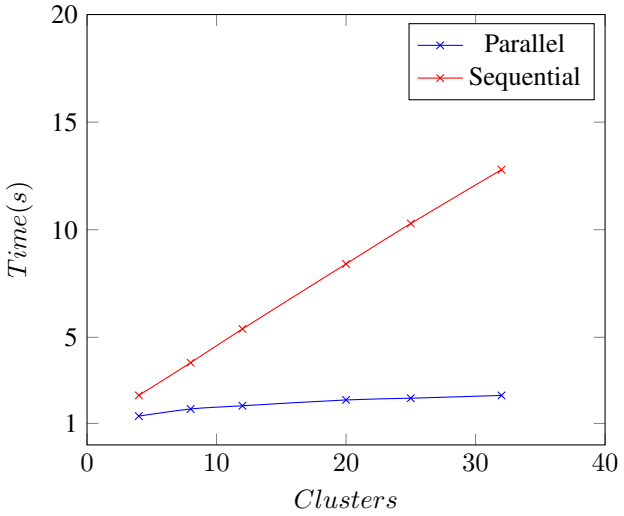
	4 clusters		32 clusters	
Type of execution	Sequential	Parallel	Sequential	Parallel
Time (s)	2.20	0.80	13.11	1.49

Looking at the results, we can see that the more clusters, the bigger the difference between the execution times. It's

worth noting the difference in execution times when going from 4 clusters to 32 clusters. On the parallel execution, it's not a very significant difference ($\Delta t = 0.69s$), while on the sequential execution, there's a massive difference in execution time ($\Delta t = 10.91s$).

This suggests that the parallel version should scale a lot better than the sequential version. To evaluate this, more tests were made with varying number of clusters, in order the make the following chart:

Note: The execution time tests were performed without using the $--exclusive$ flag



After looking at the results provided in the chart, we can conclude that the parallel version scales much better than the sequential version, providing near constant performance (up to 32 clusters).

VI. Conclusion

At this stage of the project we got to the conclusion that we were able to fulfill everything we were asked.

The main goal of this project was to explore parallelism, having in mind the reduction of the program's execution time. With this being said, we made some changes to the code that we already had, as we previously explained on this report. The most challenging part was deciding were to use pragma, since in many test cases it provided worse performance than not using it at all.

All in all, we consider that this work was very interesting and challenging. It made us practise a lot about the things we have been taught during the first two months of this semester.

As a group, we managed to work together and mutually support each other. In general, we had a positive performance.

In conclusion, this project helped us develop new skills and consolidate what we've been learning about optimization and parallelism.