Lloyd's K-means Clustering algorithm

1st Rita Gomes
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG50723

2nd Pedro Araújo
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG50684

***Abstract*—This report consists of presenting an optimized version of a k-means algorithm based on Lloyd's algorithm using OpenMp primitives, developed by the group in C programming language.**

***Index Terms*—scalability, perfomance, sequential, parallel, OpenMP, optimization, metrics**

I. Introduction

The aim of this project is to evaluate the learning of code optimization techniques, including code analysis techniques/tools through the use of C programming language and the Search application as programming interface. For this, the Lloyd's K-means Clustering algorithm will be studied and the main objective is to implement it in the most efficient and appropriate way possible. Next, we will make a small explanation of the algorithm used and an overview of all phases of the project and at the end, through the analysis of some metrics such as such as running average runtime, scalability and others, the final results of the experiment allow evaluating the performance of the final algorithm and how the various optimization techniques influenced the performance improvement.

II. Brief description of Lloyd's K-means Clustering algorithm

To process the learning data, the K-means algorithm starts with a given a number k and separates all data in a given partition into k separate clusters, each with a center that acts as a representative, and there are also N points generated randomly that will be assign to the clusters. There are iterations that reset these centers then reassign each point to the closest center calculating the euclidean distances from the point to all centroid clusters. Then the next iteration repeats until the centers do not move. The algorithm is as follows:

- Assign each data point to the cluster Ci corresponding to the closest cluster representative xi (1 i k).
- After the assignments of all n data points, compute new cluster representatives according to the center of gravity of each cluster.

We used C as the programming language to implement this algorithm using one structure for the points (which was used both for Points and Clusters), where the variable idCsize was used for differentiation, so it can be either the id of the associated cluster for the Points array or the size of each cluster for the Clusters array.

```
typedef struct points
{
    int idC_size;
    float x;
    float y;
}Points;
```

Fig. 1. Struct Points

The code implemented by the group starts by initializing a matrix of points and a matrix of clusters. The **inicializa** function initializes the points and clusters by randomly assigning values of x and y within the range of 0 to 1. The **calculaDist** function calculates the Euclidean distance between two points. The function **associaPontosInit** associates each point to the closest cluster, initializing the association of each point to the closest cluster and updating the cluster size. The **associaPontos** function re-associates each point to the closest cluster and returns a flag if the cluster association of any point has changed. The **recalculaCentroid** function updates the cluster centroids based on the average position of the points within the cluster. The main loop keeps calling the **associaPontos** and **recalculaCentroid** functions until the association of the points to the clusters doesn't change anymore.

After a detailed analysis of the implemented code, the group realized that it could be optimized in the most time-consuming functions, potentially accelerating the overall execution time. Therefore, the **associaPontosInit** and **associaPontos** functions both iterate through all of the points and all of the clusters, calculating the Euclidean distance between each point and each cluster. This can be computationally expensive, especially for large datasets. The same is valid for the **atualizaCentroids** function, when it updates the cluster centroids based on the average position of the points within the cluster. In this way, these three functions will be the ones where we must parallelize.

III. First phase of the project

In the first phase of the project, the group implemented a version of the algorithm that resorted to strategies such as spatial and temporal locality, loop unrolling and vectorization. After testing several flags, the group concluded that the flag that demonstrated the best performance was the -O3 flag. Although the strategies used progressively improved performance as they were implemented, it was found that the number of cycles was a little high, which may have affected the processor speed to execute the code.

Metrics	-O0	-O2	Loop unrolling && -O3 && vetorization	Loop unrolling && -O2	Loop unrolling && vetorization
Average Runtime	1min5.145s	29.1s	9.5s	28.07s	12.5s
Clock cycles	233.745.156.724	88.295.827.208	40.725.178.770	97.241.958.118	39.658.343.494
Instructions	285.940.139.906	66.865.344.392	39.553.696.185	65.939.115.471	63.859.409.989
Average CPI	0.8	1.3	1.0	1.5	0.6
L1-D Misses	153.392.938	143.051.315	140.245.357	143.735.845	139.841.602

Fig. 2. Influence of different flags in the sequential version

	Sequencial	
	4 clusters	32 clusters
Cycles	40.725.178.770	130.103.224.278
Instructions	39.553.696.185	178.124.710.425
Execution time (seconds)	9,50	43,10
L1 misses	140.245.357	90.935.712

Fig. 3. Sequential version results

IV. Second phase of the project

In a second phase, the group implemented a parallel version of the algorithm using openMP. For this, the group had to analyze the code and understand which functions were executed more often and adopt clauses such as **pragma omp for reduction, pragma omp for, pragma omp atomic** and **pragma omp set in a thread**. Due to the fact that there are mutual exclusion sentences, it was not possible to reach the acceleration ideal (which is proportional to the number of cores) since according to the number of threads grows, more time to wait for each thread to access memory, consequently increases the number of L1 misses and cycles. Despite this, it should be noted that the code had no data races or parallel execution problems.

	Parallel					
	4 clusters			32 clusters		
	2 threads (cpus)	8 threads (cpus)	16 threads (cpus)	2 threads (cpus)	8 threads (cpus)	16 threads (cpus)
Cycles	36.383.615.608	60.040.217.813	91.667.115.931	213.739.647.746	233.167.825.282	253.535.954.045
Instructions	35.375.496.957	35.692.727.177	36.221.835.336	178.287.775.316	178.746.123.737	179.265.426.393
Execution time (seconds)	6.0	3,17	2.69	34.57	10.37	6.18
L1 misses	82.976.497	174.560.767	217.428.581	93.702.134	234.047.289	371.960.909
#I / thread	17.687.748.479	4.461.590.897	2.263.864.709	89.143.887.658	22.343.265.467	11.204.089.150
#I / L1 misses	426.33	204.47	166.60	1902.71	763.72	481.95

Fig. 4. Parallel version results

V. Third phase of the project

In this last phase of the project, the group decided to try to implement CUDA but without success. In this way, the group opted to make an improvement to the code of the second phase.

A. CUDA

In a final phase of the project, the group tried to implement a Cuda architecture developed by NVIDIA, which allows the execution of code in parallel on graphics processing devices (GPUs). To accomplish this, we had to make some changes to the code, such as:

- Move the data to the GPU: Before starting to process the data, it is necessary to transfer it to the GPU. This can be done with the cudaMemcpy function.
- Run the centroid calculator in parallel: The new centroid calculator for each cluster can also be done in parallel using a CUDA kernel.
- Perform assignment of points to centroids in parallel: Assignment of points to centroids can also be parallelized using a CUDA kernel.

However, the group was unable to finish this implementation, mainly because the program had some data dependencies that compromised a lot the program's performance. The group attempted to resolve the data dependencies but it involved too many changes to the code structure, so it was considered better to improve the last OpenMp implementation.

B. OpenMP optimization

After a detailed analysis of the last version OpenMP parallel code, and with studies on how to improve scalability, the group made some optimizations:

- Fine-grained parallelism: In the code, there were some **for** sections that could be causing excessive parallelism overhead. Therefore, in order to reduce this, we have grouped the **for** sections into one **pragma omp parallel section**.
- Eliminate task synchronisation: In the main function of the program (associaPontos), which has more calls, there were data dependencies that were being resolved by

atomic synchronisation. However, when many threads are created, the performance of program decreases a lot, because there is more waiting time for threads. Then, the data dependency of the critical function was separated, and placed in an auxiliary function.

- Improve load distribution: The work of each thread was poorly distributed, causing an active wait at the end of the function. To improve this, the OpenMp scheduling tool (pragma omp scheduling) was used, the dynamic option, this means dividing tasks into blocks and distributing them to threads dynamically, based on the availability of threads and the size of task blocks.

	Optimized Parallel					
	4 clusters			32 clusters		
	2 threads (cpus)	8 threads (cpus)	16 threads (cpus)	2 threads (cpus)	8 threads (cpus)	16 threads (cpus)
Cycles	37.692.668.112	46.017.616.503	62.444.867.264	221.293.884.798	226.583.830.633	252.132.431.065
Instructions	36.564.989.610	37.682.311.188	37.713.617.364	179.894.666.748	180.792.510.198	181.230.438.817
Execution time (seconds)	6,2	2.8	2	34	9	6.2
L1 misses	130.238.801	151.457.570	153.804.894	140.784.989	183.379.304	233.396.350
#I / thread	1.828.249.481	4.710.288.899	2.357.101.085	8.994.733.337	2.259.906.377	1.132.690.243
#I / L1 misses	280,75	248,80	245,20	1277,80	985,89	776,49

Fig. 5. Optimized parallel version results

VI. Cache Comparison

When talking about a machine's memory hierarchy, it usually refers to the data storage structure at different access levels, from the fastest to the slowest. In general, there are three levels of memory in a machine: cache memory, main memory (RAM) and secondary storage (hard disk). There are several levels of cache memory, each with different sizes and speeds:

- L1 Cache: It is the fastest and most expensive cache memory in the machine. It is divided into two parts, one for storing data and the other for storing instructions. It is usually built into the processor and is a few kilobytes in size.
- L2 Cache: It is a cache memory that is a little slower and cheaper than L1. It is usually built into the processor and is a few megabytes in size.
- L3 cache: it is a cache memory that is even slower and cheaper than L2. It is usually located outside the processor and has a size of several megabytes or even a few gigabytes.

Main memory (RAM) is slower-access memory located outside the processor that is used to store data and instructions that are accessed less frequently.

A. Memory usage Analysis

In order to verify the memory usage at different levels of cache, the 'valgrind' tool was used to collect the related data. The following tables are about the total results of the program and the functions that have the most memory usage.

Program Totals								
#I load	#L1 Inst load	#L2 Inst load	# Data load	# L1 Data load	# L2 Data Load	# Data written	# L1 data written	# L2 data written
31.940.637.041	1.560	1.532	8.131.818.668	7.595.467	7.580.603	2.010.002.347	4.164.918	4.164.754

Fig. 6. Total results

/build/glibc-SzIz7B/glibc-2.31/../../../../../e_pow.c:__ieee754_pow_fma								
#I load	#L1 Inst load	#L2 Inst load	# Data load	# L1 Data load	# L2 Data Load	# Data written	# L1 data written	# L2 data written
17,546,232,372	15	15	3,618,721,000	212	207	0	0	0

Fig. 7. Usage of pow library

calculaDist								
#I load	#L1 Inst load	#L2 Inst load	# Data load	# L1 Data load	# L2 Data Load	# Data written	# L1 data written	# L2 data written
2,294,621,796	15	15	807,303,886	1,875,117	1,875,052	203,653,952	625	625

Fig. 8. calculaDist function

associaPontos								
#I load	#L1 Inst load	#L2 Inst load	# Data load	# L1 Data load	# L2 Data Load	# Data written	# L1 data written	# L2 data written
1,830,741,138	15	15	659,442,762	1,327,184	1,327,160	159,293,385	413.682	413.672

Fig. 9. associaPontos function

By analyzing the results shown in the tables, it is observed that the 'associaPontos' function, which has the two other functions called within it, is the one that uses the program's memory the most. Furthermore, it is possible to take from the tables that more than 50% of instructions read are from 'pow' function of the ¡math.h¿ library, and almost all data is read from RAM, which is much slower, as mentioned before. A possible optimization, in order to make the same operation using cache memory, is to manually create a function within the program that makes pow work.

To consolidate how the the program and related functions are performed, a profiling tool 'gprof' was used, which profiles data. From the following table it is possible to verify that the 'calculaDist' function, which has the pow operation, is also the most usage and time-consuming.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 57.90      5.98     5.98                             main
 35.25      9.62     3.64 116944675     0.00     0.00  calculaDist
  5.82     10.22     0.60       20    30.05    30.05  changePoints
  1.21     10.35     0.13        1   125.22   125.22  inicializa
  0.00     10.35     0.00       20     0.00     0.00  associaPontos
  0.00     10.35     0.00       20     0.00     0.00  inicializaVectors
  0.00     10.35     0.00       20     0.00     0.00  initPontosAux
  0.00     10.35     0.00       20     0.00     0.00  recalculaCentroid
  0.00     10.35     0.00        2     0.00     0.00  freePoints
  0.00     10.35     0.00        2     0.00     0.00  initPoints
  0.00     10.35     0.00        1     0.00     0.00  associaPontosInit
```

Fig. 10. gprof results

VII. Evaluation of the solution

In order to analyze the performance of the various techniques used, the group chose to present the graph below. This graph compares the scalability of the parallel and optimized parallel implementation of OpenMp with the ideal speed-up.

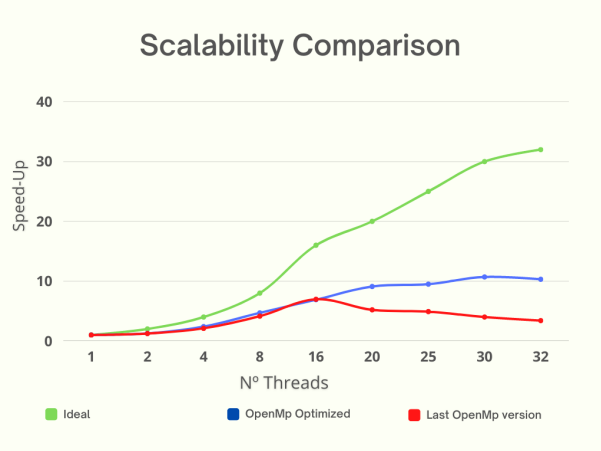

Fig. 11. Speed-up analysis

From the graph above, we conclude that the optimized version of OpenMp has a better scalability than the old one, having the maximum speed-up peak in 30 threads against 16 of the last OpenMp version. So, it was proved that the optimizations made were effective.

VIII. Conclusion and future work

With the accomplishment of this practical work, the group concludes that it was possible to consolidate knowledge taught during the classes of Parallel Computing. It was also a great way for us to explore OpenMP and practise more about optimizing code, whether parallel or sequential. It was also good to be familiar with profiling tools such as 'perf' and 'gprof' to get metrics of program performance, and see memory usage tool with 'valgrind'.

Some difficulties were encountered during the develop stage, like trying to implement CUDA. However, for a future work the group intends to do the CUDA implementation, and do more optimizations, such as create a 'pow' function to explore better the memory usage.

Given that all the requirements demanded, the team considers that this practical assignment was successfully completed.

References

[1] OpenMP, https://www.openmp.org/
[2] Gprof, https://www.ibm.com/docs/en/aix/7.1?topic=g-gprof-command
[3] Valgrind, https://www.man7.org/linux/man-pages/man1/valgrind.1.html