Parallel Computing
K-means Algorithm

Laura Rodrigues
Masters in Software Engineering
University of Minho
Braga, Portugal
pg50542@alunos.uminho.pt

Mariana Rodrigues
Masters in Software Engineering
University of Minho
Braga, Portugal
pg50628@alunos.uminho.pt

***Abstract*—The K-means Algorithm takes a given number of points and groups them by clusters. As a result we get a set of cluster centroids that allow us to organize the points. Each point is closer to it's centroid than to any other centroid. In this paper, we implemented and analysed a parallelized version of this algorithm, using OpenMP.**

***Index Terms*—k-means, OpenMP, parallel**

I. Introduction

The goal of this project was to test the use of directives and clauses of OpenMP while trying to minimize the execution time.

II. Implementations

We started this assignment by looking at the program we previously developed. Using the command *perf*, we concluded that the hotspots of our implementation were the following, with the associated overhead:

Function	**Overhead**
update_cluster_points	76%
determine_new_centroid	16%
init	2%

Table 1: Results of the command perf

We started the parallelization of our algorithm with the function *update_cluster_points*. Previously, we saw that this function was the one with the most significant overhead, so it was a must to parallelize it.

This function was composed by an outer *for loop* and an inner *for loop*. In order to take advantage of the OpenMP directives, we needed to parallelize significant chunks of work. As the inner loop only calculates the distance between each point and all of the centroids, using the threads available to parallelize this chunk of work wouldn't make much of a significant difference.

Next, we thought of using the directive *#pragma omp for* on the outer loop. It seemed like a better option, as this way, each thread would execute more work. In other words, we believe the speed-up gain from this, would be greater than the overhead of creating the threads, therefore justifying it.

This parallelization would create data races, because of that we needed to explore *pragma clauses*.

On *update_cluster_points*, we used the reduction clause with *points_changed* accumulator in order to try and bypass the data racing that this variable brings. Otherwise, the loop would not be able to perform this operation correctly as multiple threads would try to write in the same variable at the same time.

Additionally, we noticed that we had a few variables, *diff*, *newCluster* and *diff_temp* that should be private to each thread. As they would be used by the multiple threads and would store important data, they should be protected from data races. We defined then after *#pragma omp parallel*, so this way, they could only be accessed by the thread itself.

Lastly, we though about what schedule we should use. Analysing our code, we could see that each iteration executes almost the same amount of *"work"*, as all have to calculate the distance of a certain point to the K centroids, comparing it to find a smaller value, and lastly, changing the cluster. Even though, the cluster may not be changed on all iterations, we don't consider it that significant. Therefore, we chose to use static scheduling. Doing so, we would also spare some overhead of managing the dynamic and guided schedule.

Afterwards, we looked into the function *determine_new_centroid*. This function is the one with the second most significant overhead. Therefore, parallelizing it will be the second most important change.

We chose to parallelize all three *loop for* of this function. In the second loop, as the number of iterations was very high, when using *#pragma omp for* we made the program more efficient. In the last loop, as the operations executed are significantly expensive, we could also benefit from the parallelization.

In this function, we would calculate the mean of all the points of each clusters. As the data structure we chose to work with keeps all the points in a array, and each point only has an identifier to the cluster, we could not predict where we would add the coordinates of a point. Therefore, we need to add some protection to data races that would rise up.

The arrays *SumX*, *SumY* and *sizeA* would be changed by each thread, as they would store the results of the addictions. So, we used the reduction clause to make each array private to each thread and at the end of each we add all the values.

In order to obtain the correct mean, we would have to initialize all accumulators with zero. Therefore, before the second loop, we used the *#pragma omp barrier* directive so that all threads would wait at the barrier. Before the last loop and before the division, we needed to make sure that all the iterations of the previous loop had been concluded. Therefore, we would once again use the directive *barrier*.

When considering the load balance, we determined that the amount of work done by each thread would be similar, therefore, we decide to keep the static schedule for all *#pragma omp for*.

The function *init* was kept sequential, because of the random generation of values. The seed used would have to be shared by all threads and this value is not so easily privatized.

III. Performance tests

A. Sequential code

In our implementation, we had some sections that we weren't able to parallelize, such as the init and code from the main function. This would not allow us to get the ideal speed-up, as according to the Amdahl's Law a program speed-up is limited by the sequential code. Therefore, we tried to calculate the theoretical speed-up we could get, by analysing the complexity of the functions with the greater overhead, of the sequential and parallelized versions.

Knowing that N was 10000000 and number of iterations is 20, because of an additional iteration to initialize the clusters, we present in table 2 the results of the calculated speed-up for different number of clusters, according to the calculations presented in the Appendix.

#T	**1**	**4**	**8**	**12**	**16**	**20**
K = 4	1	3,89	7,5	10,41	14	16,94
K = 32	1	3,98	7,91	11,8	15,64	19,44

Table 2: Expected Speed-up for different number of threads with 4 and 32 clusters

B. Workload balance

As mentioned previously, in our solution we used the pragma clause *schedule(static)*. We wanted to be sure that each thread did approximately the same workload, as to avoid having threads waiting for others to finish, expect when extremely necessary. We used *perf report -s pid* and in section B of the Appendix, we can see that apart from one of the threads, all of them had approximately the same percentage of overhead. As the master thread has to do the sequential part of the program, we can justify the difference mentioned. This further helped to validate our decision of keeping the schedule static.

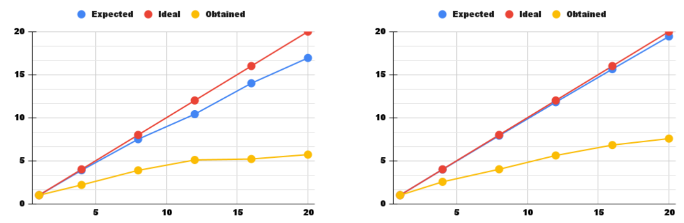
C. Parallelisation scalability

After using adequate directives to tackle potential data races, we tested our code's scalability by measuring the performance of the code for 4, 8, 12, 16 and 20 threads and calculated the speedup relative to its sequential execution, to do so, we used the command *perf*.

#T	**TExec**	**SpeedUp**	**#Ix**10^9	**#CCx**10^9	**#Mx**10^7	**CPI**
1	3	1	20,17	8,35	8,53	0,4
4	1,53	2,19	20,23	15,7	8,56	0,8
8	0,86	3,88	20,4	14,37	8,56	0,7
12	0,66	5,09	20,26	15,93	8,57	0,8
16	0,64	5,20	20,26	15,9	8,58	0,8
20	0,59	5,71	20,28	16,4	8,58	0,8

#T	**TExec**	**SpeedUp**	**#Ix**10^9	**#CCx**10^9	**#Mx**10^7	**CPI**
1	17	1	113	49,82	9	0,43
4	6,69	2,55	112,87	63,9	8,9	0,6
8	4,26	4,01	113,08	82,19	9,06	0,7
12	3,04	5,61	113,24	80,87	9,18	0,7
16	2,50	6,82	114,06	98,53	9,22	0,87
20	2,26	7,57	115,26	101,57	9,27	0,9

Tables 3 & 4: Results obtained using different numbers of threads with 4 and 32 clusters respectively



Images 1 & 2: Speed-up graphic representation with 4 and 32 clusters

As we can see, we didn't get the ideal speed-up nor the expected one, presented in Table 2. We believe that one cause may have been the overhead of launching and synchronizing the threads, cause by the reduction clause, the barrier and parallel directive. We also believe that there may be an additional sequential work that may be not taken into consideration. As the complexity determine doesn't take into account the actual work each function implies.

We don't believe that this significant different was caused by load imbalance nor by loads of cache. We previously saw that the threads perform a similar amount of work and we can see that the L1-Misses don't have much of a difference from the parallel to the sequential.

We can also notice that the parallelism granularity we chose is reasonable, as the number of instructions doesn't change a lot when using 4 clusters but changed slightly when K = 32.

IV. Solution's performance

With the version we implemented, we obtained a faster version than with the sequential code. However, the speed-up we obtained from the parallelization is not near the ideal speed-up of a completely parallelizable algorithm. Overall, we believe we implemented a decent algorithm, however, the speed-up could have been better if the algorithm chosen was designed since the beginning to be parallelizable. We decided that our program performed better when 20 threads where being used as we can see from the previously done analysis.

V. Appendix

A. Sequential code

The sequential implementation has the following complexity: $O(N + K + N * K + It * (N * K + N + K + K))$. The parallel implementation would have the complexity: $O(N + K + \frac{(N*K)}{\#Threads} + It * (\frac{(N*K)+N+K+K}{\#Threads}))$.

Our code's complexity can be expressed in pseudo-code with:

```
kmeas(){
    init();
    update_cluster_points();
    cycle:
        determine_new_centroid();
        update_cluster_points();
}
```

Each function's complexity is:

- init: $N + K$
- update_cluster_points: $N * K$
- determine_new_centroid: $N + K + K$

Therefore we obtained these results while trying to get the expected speed-up:

#T	**Init**	$\frac{\textbf{update}}{\#Threads}$	It*($\frac{\textbf{determine+update}}{\#Threads}$)	**Final**
1	10000004	40000000	1000000160	1050000164
4	10000004	10000000	250000040	270000044
8	10000004	5000000	125000040	140000044
12	10000004	3333333,333	87500042	100833379,3
16	10000004	2500000	62500040	75000044
20	10000004	2000000	50000040	62000044

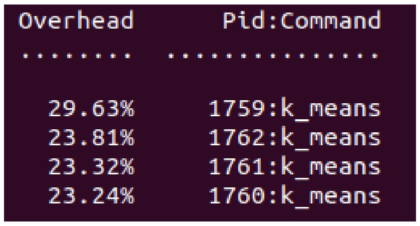
Table 5: Complexity for 4 clusters

#T	**Init**	$\frac{\textbf{update}}{\#Threads}$	It*($\frac{\textbf{determine+update}}{\#Threads}$)	**Final**
1	10000032	320000000	6600000160	6610000192
4	10000032	80000000	1650000040	1660000072
8	10000032	40000000	825000040	835000072
12	10000032	26666666,67	550000040	560000072
16	10000032	20000000	412500040	422500072
20	10000032	16000000	330000040	340000072

Table 6: Complexity for 32 clusters

B. Workload balance

As we can see all the threads have approximatelly the same workload.



```
Overhead       Pid:Command
........  ...............

  29.63%     1759:k_means
  23.81%     1762:k_means
  23.32%     1761:k_means
  23.24%     1760:k_means
```

Fig. 1: perf report -s pid results

C. Parallelisation scalability

In tables 3 and 4:

- #T - Number of Threads
- TExec - Execution time
- #I - Number of instructions x 10^9
- #CC - Number of clock cycles x 10^9
- #M - Number of misses x 10^7
- CPI - Cycles per Instruction