K-Means Clustering

Enhancing performance and scalability through parallelism using OpenMP

Luís Neto (a77763), Gonçalo Raposo (a77211)

November 21, 2018

1 Case of Study

K-means clustering is an algorithm commonly used to partition data into k groups based on their similarities. The algorithm consists in three phases:

1. Selection of k random centroids.
2. Assignment Step: each point is assigned to a cluster based on closest centroid determined with *Euclidean distance*.
3. Update Step: each centroid is updated to be the mean of his cluster.

The latter two phases are repeated until the centroids no longer change.

2 Tests and testing conditions

In order to test the developed algorithms, four different data sets were created by a Python script using a real uniform distribution, the first data set has 2048 points, the second has 16384, the third 1966080 points and the last one 62914560. The size of the data sets was chosen in order to fully occupy the different types of caches and the last one was calculated so that it wouldn't fit in any of the above.

Furthermore, to produce reliable results, all the measurements have been executed in the same computer with *Ivy Bridge micro-architecture*, *dual CPU*, *twenty-four physical cores* and *sixty-four GB of RAM*, before of any measurement the cache memory is always cleaned, finally, the time unit used for measurements was microseconds.

3 Algorithm Analysis

3.1 Data Representation

To implement the algorithm the cluster and cluster's size were represented using two arrays, the first array was used to store the cluster to which the i^{th} data set instance belongs and the latter to store the current number of points in each cluster. The centroids and data set instances are 2D points, initially, an array was used to represent both components.

With this representation, the algorithm couldn't take advantage of vectorization, since the elements were accessed with a stride of 2.

3.2 First Phase

This phase consists of finding the maximum value in the data set points. Then a uniform distribution between, zero and the maximum value, is used to generate the initial centroids.

3.3 Second Phase

3.3.1 Sequential Implementation

In this phase, for each instance i^{th} of the data set is calculated the closest centroid, based on the squared Euclidean distance, then the i^{th} element of the cluster's array is assigned the cluster that the centroid represents, finally, the cluster size is incremented.

This approach offers good spatial locality since all the arrays used in this phase are accessed sequentially, excepting the array that controls each cluster's size. Furthermore, using the squared Euclidean distance doesn't change the final result, that said it is possible to avoid a square root calculation.

3.3.2 Parallel Implementation

After measure the sequential algorithm, we concluded that this phase was the one that occupies most of the program time, being the most critical zone in the program. In this phase of the algorithm, we face two different problems, the false sharing and the possibly write conflicts in the array that controls the size of each cluster.

To solve this problems we applied a reduction primitive to the array. This primitive creates a private copy of the array for each thread resulting in an increase of memory overhead but avoiding false sharing. At the end of the reduction, the reduction variable is applied sequentially to all private copies of the shared array ensuring data coherency and avoiding write conflicts at the cost of an additional overhead.

3.4 Third Phase

3.4.1 Sequential Implementation

The first approach to implement this phase, two arrays were used, one to store the current centroids (centroid_old) and one to store the actualized centroids. After analyzing these phase's steps, we can point the following problems: the need to use an auxiliary array and the $2k$ divisions executed to update all the centroids.

To solve these problems the subsequent modifications were made: in the first loop, part of the error and the inverse of each size cluster were calculated, this optimization allowed to halve and to hide the latency of divisions. In the second loop, as the inverse of the size of each cluster has already been calculated, we can take advantage of FMA operations. Both versions are documented in the Appendix A.

3.4.2 Parallel Implementation

After reviewing the execution times, we concluded that this phase wasn't so critical as the latter, even so, it needs some attention. In the analysis of this phase, we found the same problems as in the second phase, false sharing and write conflicts thus the same solution was applied, a *reduction* to the X and Y components arrays.

4 Results analysis and demonstration

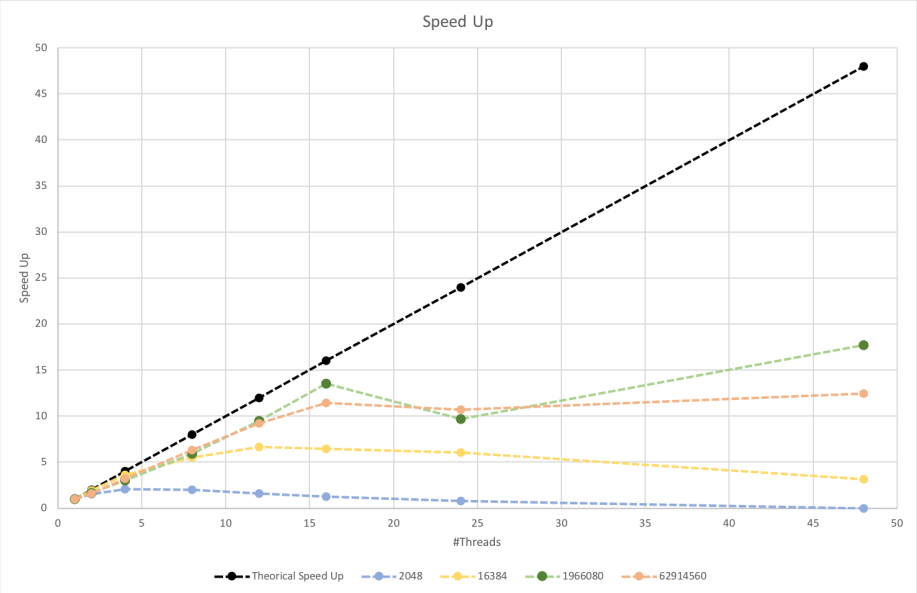

Figure 1: Speedup for different data sets

4.1 Data set with 2048 points

For this data set, we can conclude that the speedup doesn't increase by adding more cores. The use of reductions creates an overhead due to the creation of each thread private array and to the end operation that is applied to ensure data coherency. The performance is also affected by the overhead that comes from thread scheduling since the data set fits in level 1 cache this overhead can't be hidden by RAM reads.

4.2 Data set with 16384 points

With this data set, it's possible to obtain a speedup if the number of cores used is between two and twelve, whoever, if thirteen or more cores are used we can observe a speedup reduction. The reasons why this happens are the same as in the latter data set, furthermore, since we are using a machine with two chips, each one with twelve physical cores, the use of 13 or more cores is penalized due to the distance of the second chip to the memory that contains the necessary data to execution.

4.3 Data set with 1966080 and 62914560 points

In both data sets, it's possible to verify speedups close to theoretical ones, using a number of cores between two and twelve. Due to the size of the data set, the overhead created by the reductions and context switch are hid with memory reads latency, resulting in performance gains. Finally, with the measurements made, we can't explain the results observed in figure 1.

5 Conclusion

With this assignment, we learned that a careful analysis of a sequential algorithm could suggest optimizations that improve the overall performance and bottlenecks to parallelization. With the performance results obtained, we concluded that our parallel algorithm is scalable for a number of cores inferior to twelve. Finally, results obtained with a number of cores superior to twelve doesn't scale, however, if we use hyper-threading and thread mapping there could be space for performance improvements. This solutions could be easily implemented in a future assignment and be confirmed by a more detailed tests of speed up a a deeper study of the architecture used.

A Appendix

```
1: for i = 0 to #cluster do
2:    centroid_old[i * 2] = centroid[i * 2]
3:    centroid_old[i * 2 + 1] = centroid[i * 2 + 1]
4:    centroid[i * 2] = 0
5:    centroid[i * 2 + 1] = 0
6: end for
7: for i = 0 to #data_point do
8:    point_cluster = cluster[i] * 2
9:    centroid[point_cluster * 2]+ = xcomp[i]
10:   centroid[point_cluster * 2 + 1]+ = ycomp[i]
11: end for
12: for i = 0 to #cluster do
13:   size = cluster_size[i]
14:   centroid[i * 2] = centroid[i * 2]/size
15:   centroid[i * 2 + 1] = centroid[i * 2 + 1]/size
16:   error+ = entroid[i * 2] − centroid_old[i * 2]
17:   error+ = entroid[i * 2 + 1] − centroid_old[i * 2 + 1]
18: end for=0
```

Third Phase - Initial version.

```
1: for i = 0 to #cluster do
2:    error− = centroid_x[i] − centroid_y[i]
3:    centroid_x[i] = 0
4:    centroid_y[i] = 0
5:    cluster_size[i] = 1/cluster_size[i]
6: end for
7: for i = 0 to #data_point do
8:    point_cluster = cluster[i]
9:    size_inverse = cluster_size[i]
10:   centroid_x[point_cluster]+ = xcomp[i] * size_inverse
11:   centroid_y[point_cluster]+ = ycomp[i] * size_inverse
12: end for
13: for i = 0 to #cluster do
14:   cluster_size[i] = 0
15:   error+ = centroid_x[i] + centroid_y[i]
16: end for=0
```

Third Phase - Optimized version.

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
2048	44235	829	652	14338987
	44893	842	715	14339127
	44611	590	441	14334785
	44865	510	386	14334279
	44578	556	422	14334471
Median	44611	590	441	14334785

Table 1: PAPI measurements for data set with size 2048, Sequential version

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
16384	1237242	74705	970	380112304
	1236534	77034	1185	380109505
	1236372	87436	854	380108171
	1236201	80241	758	380107770
	1236280	77276	878	380102265
Median	1236372	77276	878	380108171

Table 2: PAPI measurements for data set with size 16384, Sequential version

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
1966080	194422872	37859641	11631894	56234045626
	194473239	37392953	11672227	56234045832
	194403955	37939698	11908456	56234042012
	194218604	39098256	12437797	56234041120
	194231246	38811046	12194447	56234041305
Median	194403955	37939698	11908456	56234042012

Table 3: PAPI measurements for data set with size 1966080

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
62914560	6286197150	1548008162	496122562	1823690747345
	6289167648	1529379740	488030453	1823690747580
	6287796966	1538553597	484043531	1823690742780
	6281790689	1553128958	491621233	1823690742640
	6281945573	1557574203	496046107	1823690742832
Median	6286197150	1548008162	491621233	1823690742832

Table 4: PAPI measurements for data set with size 62914560

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
2048	45514	744	636	12814717
	45366	884	702	12813256
	45962	537	425	12808914
	45574	468	352	12808428
	45436	527	393	12808556
Median	45514	537	425	12808914

Table 5: PAPI measurements for data set with size 2048, Sequential version with optimizations

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
16384	1237400	45521	965	340339789
	1237394	51962	1124	340335377
	1236512	45369	767	340334052
	1235996	48719	794	340333674
	1237230	48381	853	340328099
Median	1237230	48381	853	340334052

Table 6: PAPI measurements for data set with size 16384, Sequential version with optimizations

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
1966080	199023942	25017796	5890246	50355330272
	199097671	24937652	5863673	50355328864
	199168823	24904392	5834379	50355324536
	199076163	24720262	5760485	50355324662
	198739066	25055010	5874897	50355324294
Median	199076163	24937652	5863673	50355324662

Table 7: PAPI measurements for data set with size 1966080, Sequential version with optimizations

Size	L1 Misses	L2 Misses	L3 Misses	Total Instructions
62914560	6404252052	1228417412	271017691	1633136519086
	6371502373	1575804744	390225410	1633136456101
	6378118189	1533330740	390662862	1633136449263
	6372649648	1523493782	380763099	1633136445099
	6367576847	1551967212	397083530	1633136443770
Median	6372649648	1533330740	390225410	1633136449263

Table 8: PAPI measurements for data set with size 62914560, Sequential version with optimizations

Size	Phase 1	Phase 2	Phase 3	Total time
62914560,000	135,983	197712,000	20350,302	218198,285
	134,226	198112,000	20376,983	225733,904
	138,703	204622,000	20973,201	225733.904
	138,959	204594,000	21003,492	225736.451
	138,452	204580,000	20984,393	225702.845
Median	138,452	204580,000	20973,201	23157.453
	0,061%	90,646%	9,293%	225691,653

Table 9: Time of execution of each phase and respective percentage

Size	Measurement/#Threads	Seq	Seq+Opt	2	4	8	12	16	24	48
2048	1	4,56	2,28	1,59	1,31	1,47	2,33	1,94	2,89	496,69
	2	4,54	2,27	1,48	1,09	1,15	1,40	1,53	3,07	265,49
	3	4,53	2,26	1,48	1,13	1,13	1,60	1,82	3,63	264,69
	4	4,55	2,27	1,54	1,10	1,06	1,37	1,77	2,24	541,07
	5	4,54	2,26	1,48	1,09	1,13	1,41	1,80	2,99	615,28
	Median	4,54	2,27	1,48	1,10	1,13	1,41	1,80	2,99	496,69
	Speedup	1,00	2,00	1,53	2,07	2,00	1,61	1,26	0,76	0,00

Table 10: Speedup measurements for data set with 2048 points

Size	Measurement/#Threads	Seq+Opt	2	4	8	12	16	24	48	
16384	1	120,72	59,64	31,07	16,48	10,87	9,84	8,86	9,86	19,09
	2	120,73	59,68	30,81	16,40	10,01	8,99	9,23	9,50	18,83
	3	120,58	59,61	30,78	16,37	9,78	8,81	10,20	9,83	18,33
	4	120,72	60,93	30,76	16,38	11,24	8,80	9,76	19,32	20,14
	5	120,59	59,63	30,75	16,38	12,25	9,27	8,87	9,31	15,45
	Median	120,72	59,64	30,78	16,38	10,87	8,99	9,23	9,83	18,83
	SpeedUp	1,00	2,02	1,94	3,64	5,49	6,63	6,46	6,07	3,17

Table 11: Speedup measurements for data set with 16384 points

Size	Measurement/#Threads	Seq+Opt	2	4	8	12	16	24	48	
1966080	1	18009,60	6637,65	4219,77	2211,00	1313,49	751,41	655,23	688,05	361,49
	2	18005,70	6638,03	3485,25	2226,84	1122,65	867,15	531,56	672,45	433,15
	3	18271,10	6638,57	3628,73	2277,37	1124,85	700,72	489,91	651,70	375,12
	4	18370,20	6637,60	4436,29	2211,42	1020,50	650,52	485,53	736,72	464,51
	5	18378,90	6639,75	3996,06	2210,67	1005,66	650,80	484,74	741,21	373,81
	Median	18271,10	6638,03	3996,06	2211,42	1122,65	700,72	489,91	688,05	375,12
	Speedup	1,00	2,75	1,66	3,00	5,91	9,47	13,55	9,65	17,70

Table 12: Speedup measurements for data set with 1966080 points

Size	Measurement/#Threads	Seq+Opt	2	4	8	12	16	24	48	
62914560	1	1078460,00	415323,00	274603,00	135143,00	68560,70	47368,60	37957,60	33191,00	33311,30
	2	1076820,00	414637,00	274960,00	135317,00	68479,50	47159,90	38052,10	33609,80	50861,90
	3	1077870,00	433848,00	269968,00	134960,00	68445,30	47146,60	38407,10	40477,50	34098,00
	4	1079180,00	537387,00	271280,00	135039,00	68462,90	47153,20	37819,90	45000,80	34927,00
	5	1082830,00	531658,00	270735,00	135124,00	68447,90	47153,00	37944,70	48899,50	37874,30
	Median	1078460,00	433848,00	271280,00	135124,00	68462,90	47153,20	37957,60	40477,50	34927,00
	Speedup	1,00	2,49	1,60	3,21	6,34	9,20	11,43	10,72	12,42

Table 13: Speedup measurements for data set with 62914560 points

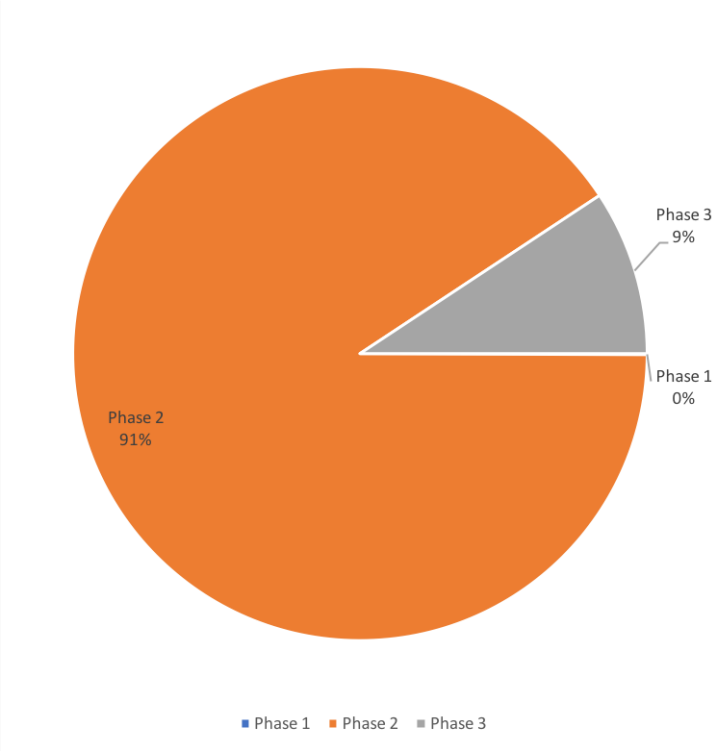

Figure 2: Time percentage of Algorithm Phases

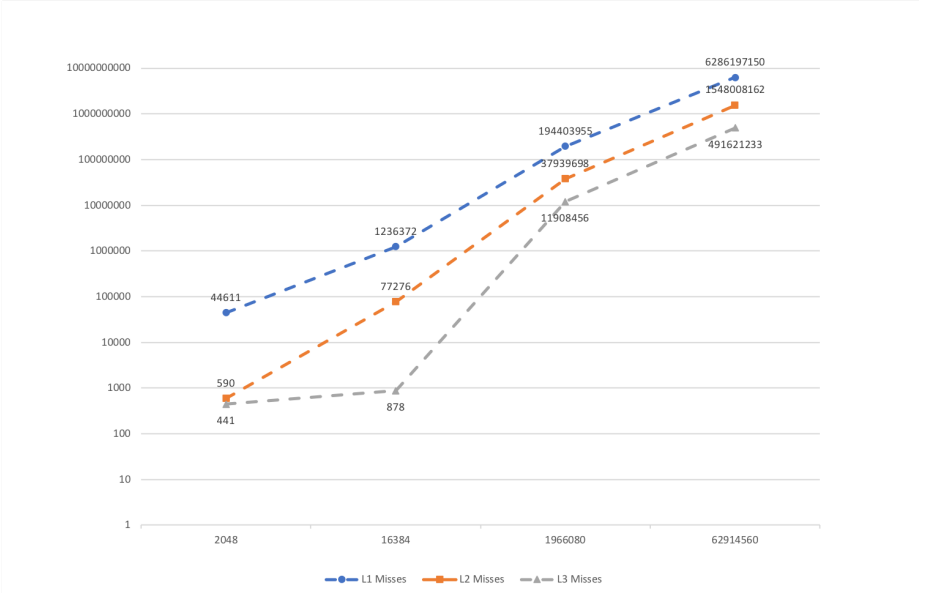

Figure 3: Sequential Version - Cache Misses

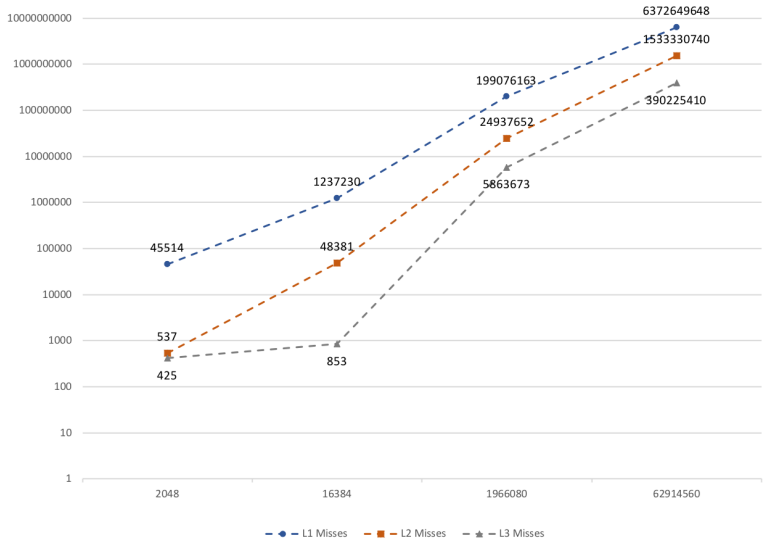

Figure 4: Sequential with optimizations Version - Cache Misses