ARCHITECTURE AND PLATFORMS FOR ARTIFICIAL INTELLIGENCE

Project Exam - Module 2
OpenMP parallel programming: implementation of K-means clustering algorithm

Andrea Proia
Id: 0001006958

Abstract. The objective of the project is to experiment parallel programming by means of the implementation of the popular cluster algorithm “k-means”. The project is implemented using C language, exploiting OpenMP and allows a dataset to be partitioned correctly in a specified number of clusters with good results in terms of parallel programming performances. Some limitations and scalability issues are discussed in the following chapters.

Keywords: OpenMP, k-means, multiprocessing

1 Introduction

OpenMP (Open Multi-Processing) is an application programming interface that supports multi-platform shared-memory multiprocessing programming in different languages.
As mentioned above, this technology is used to implement one of many variants of the k-means algorithm. In particular, the version presented in this project is the standard *"naïve k-means"* which bases the computation of the distance on the Squared Euclidean distance. Moreover, the initial set of cluster centres is chosen simply selecting the first K points of the given dataset (where K is defined by the user at run-time) and a δ threshold is defined in order to set a termination condition for the clustering iteration process.

First of all, let’s briefly recall how the algorithm is structured: a set of values (d-dimensional real vectors) is partitioned into k groups by computing k representative points (*centroids*). As an output, input values that share the same nearest centroid belong to the same computed partition. The following diagram shows the implemented workflow:

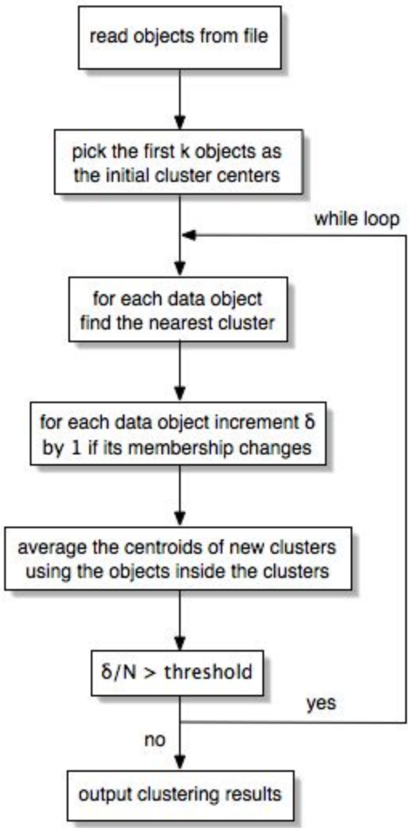

read objects from file
pick the first k objects as the initial cluster centers
while loop
for each data object find the nearest cluster
for each data object increment δ by 1 if its membership changes
average the centroids of new clusters using the objects inside the clusters
δ/N > threshold
yes
no
output clustering results

Figure 1: k-means workflow.

2 Project Structure

This section presents the organization of the project in terms of: input data, structure of the code, design decisions and output.

2.1 Input data

As said before, this implementation of the algorithm is able to deal with datasets which contain d-dimensional input vectors, where d can vary a lot. Some tests have been done with different, artificially synthesized, input data with $d = 3, 6, 9$ (other variants are possible), and with various dataset dimensions (in terms of quantity of rows/data points). In any case, such data sets are structured as follows: the first column is a simple, unique identifier and each subsequent column corresponds to a dimension, i.e. a coordinate. An example is provided below:

```
1 -542 -686 386
2 42 -734 295
3 -368 -584 -486
4 -16 -705 180
5 691 -78 47
6 -522 -602 -364
7 -519 -644 347
8 -470 -620 -360
9 -53 -601 167
10 79 -675 310
```

Figure 2: Input data example with d=3

2.2 Structure

The code organization is very linear, only 3 files are necessary:

- `main.c`: prepares structures, sets time counters, invokes the core algorithm, prints results and statistics
- `IO_lib.c`: contains function definitions to deal with input dataset reading and clustering results writing
- `kmeans.c`: contains all the instructions needed to perform clustering (included distance computation)

For what concerns distance, in this case the Squared Euclidean distance is implemented, in order to minimize within-cluster variance. Many variants are possible by simply changing the typology of distance inside the already existing function (located in *kmeans.c*)

2.3 Design decisions for parallel implementation

This topic is one of the key points of the entire project: how can we parallelize the code? How can we obtain an improvement in performances by means of the choice of an architecture instead of another? Of course, many different design decisions can be implemented and tested, since OpenMP provides a large set of instruments that allow to structure different solutions.
In this context, two variants are presented and both include the use of a `#pragma omp for` to distribute the workload among all the threads available, while also performing a `reduction(+:delta)` to ensure the computation of the total δ (i.e. the total amount of points that have changed cluster in a certain iteration) at the end of the loop. The main difference is how they manage the computation of the sum of the points and the size of the cluster, for the subsequent calculation of the mean:

A) (*ATOMIC*) The first variant computes the sum and the size of each new cluster directly on the shared structures of the function. To do so, a `#pragma omp atomic` directive is provided for both operations to ensure no race conditions on such structures.

B) (*LOCAL*) The second variant computes the sum and the size using thread local structures and, in the end, only the main thread performs the array reduction to obtain the "total" sum and the size of each new cluster.

These two distinct design decisions emphasize the importance of a well-thought-out architecture since, in fact, it turns out that the latter is quite faster than the former in terms of computation time.

In addition to that, `schedule(static)` directive has been added in both versions to guarantee a pre-determined and predictable work for each iteration (i.e. scheduling is done at compile-time, so least work has to be done at run-time).

2.4 Output

About the output, once the clustering algorithm is terminated, two files are created to write the results:

- `cluster_centres.csv`: contains a list of the final cluster centres obtained for the K clusters (written using the same format of the input data set)
- `membership.csv`: contains a list of number (one for each row) from 0 to K, each of which represent the number of the cluster the point belongs to.

```
0 -653.114014 -445.596191 -418.541351
1 248.831985 583.317200 -692.673401
2 248.322525 723.905884 -695.691528
3 199.108643 -424.837402 255.961258
```

(a) `cluster_centres.csv`

```
0
1
2
0
2
3
3
0
2
3
```

(b) `membership.csv`

Figure 3: Example of output files

An interesting evaluation of the quality of the results can be done also by using the `print.py` python script provided in the project folder. This script is suitable for 3-dimensional input dataset and presents a 3-D graph representation of the data points divided in clusters using colors. The script can be invoked from the command line with `python print.py n_cluster dataset_name`, this is an example:

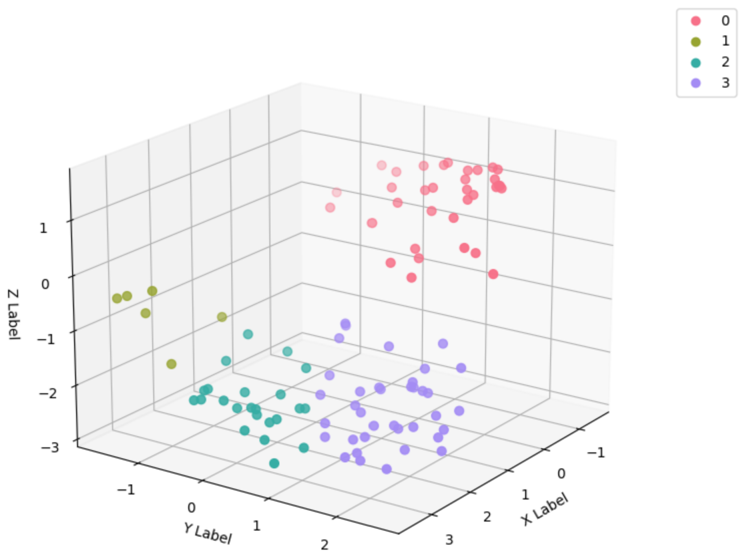
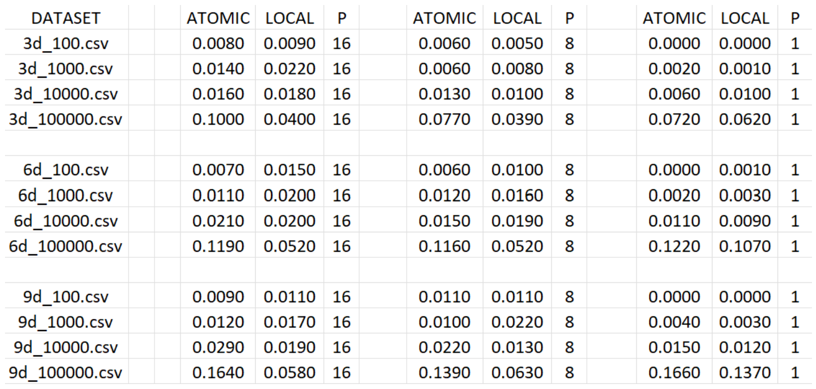


Figure 4: Example of 3-D result

3 Results

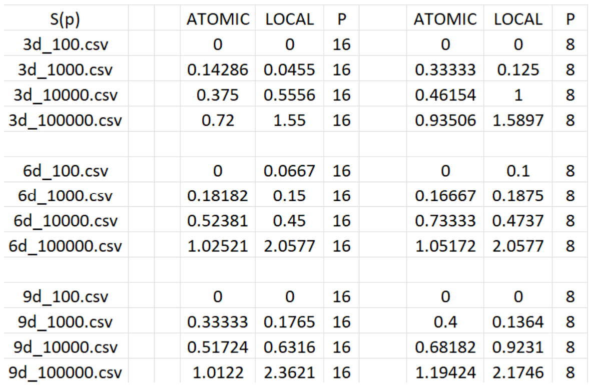
For what concern performances, some time indicators are printed at the end of the computation: *I/O time* and *Computation Time*. Obviously, in this context, we are interested in evaluating the latter, since design decisions on parallel programming greatly affect the quality of the performance. In order to measure the *Computation time* spent, the `omp_get_wtime()` function has been used immediately before and after the invocation of the clustering function. These are the results obtained for each dataset tested, both for the *ATOMIC* version and for the *LOCAL* version, and with different number of threads (P = 1, 8, 16):



DATASET	ATOMIC	LOCAL	P	ATOMIC	LOCAL	P	ATOMIC	LOCAL	P
3d_100.csv	0.0080	0.0090	16	0.0060	0.0050	8	0.0000	0.0000	1
3d_1000.csv	0.0140	0.0220	16	0.0060	0.0080	8	0.0020	0.0010	1
3d_10000.csv	0.0160	0.0180	16	0.0130	0.0100	8	0.0060	0.0100	1
3d_100000.csv	0.1000	0.0400	16	0.0770	0.0390	8	0.0720	0.0620	1
6d_100.csv	0.0070	0.0150	16	0.0060	0.0100	8	0.0000	0.0010	1
6d_1000.csv	0.0110	0.0200	16	0.0120	0.0160	8	0.0020	0.0030	1
6d_10000.csv	0.0210	0.0200	16	0.0150	0.0190	8	0.0110	0.0090	1
6d_100000.csv	0.1190	0.0520	16	0.1160	0.0520	8	0.1220	0.1070	1
9d_100.csv	0.0090	0.0110	16	0.0110	0.0110	8	0.0000	0.0000	1
9d_1000.csv	0.0120	0.0170	16	0.0100	0.0220	8	0.0040	0.0030	1
9d_10000.csv	0.0290	0.0190	16	0.0220	0.0130	8	0.0150	0.0120	1
9d_100000.csv	0.1640	0.0580	16	0.1390	0.0630	8	0.1660	0.1370	1

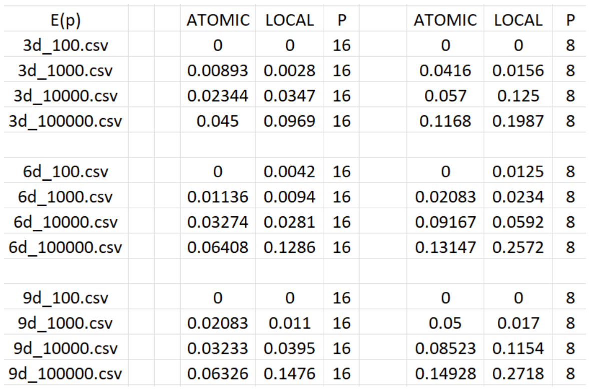
Figure 5: Table of time results

It is clear that the results start to become significant when we deal with datasets whose data point number is ≥`100000` since, when we deal with small datasets, the parallel efficiency improvement could be not detectable with precision. This is due to the fact that the computation time takes into account also different factors, such as the startup overhead for the creation of the parallel environment. Moreover, when the dataset is too small, the timer is not able to measure the computation cost even with 4 decimal digits.
Knowing that, *Speedup S(p)* and *Strong Scaling Efficiency E(p)* are computed for P = 8, 16 and shown below:



S(p)	ATOMIC	LOCAL	P	ATOMIC	LOCAL	P
3d_100.csv	0	0	16	0	0	8
3d_1000.csv	0.14286	0.0455	16	0.33333	0.125	8
3d_10000.csv	0.375	0.5556	16	0.46154	1	8
3d_100000.csv	0.72	1.55	16	0.93506	1.5897	8
6d_100.csv	0	0.0667	16	0	0.1	8
6d_1000.csv	0.18182	0.15	16	0.16667	0.1875	8
6d_10000.csv	0.52381	0.45	16	0.73333	0.4737	8
6d_100000.csv	1.02521	2.0577	16	1.05172	2.0577	8
9d_100.csv	0	0	16	0	0	8
9d_1000.csv	0.33333	0.1765	16	0.4	0.1364	8
9d_10000.csv	0.51724	0.6316	16	0.68182	0.9231	8
9d_100000.csv	1.0122	2.3621	16	1.19424	2.1746	8

Figure 6: Speedup



E(p)		ATOMIC	LOCAL	P		ATOMIC	LOCAL	P
3d_100.csv		0	0	16		0	0	8
3d_1000.csv		0.00893	0.0028	16		0.0416	0.0156	8
3d_10000.csv		0.02344	0.0347	16		0.057	0.125	8
3d_100000.csv		0.045	0.0969	16		0.1168	0.1987	8
6d_100.csv		0	0.0042	16		0	0.0125	8
6d_1000.csv		0.01136	0.0094	16		0.02083	0.0234	8
6d_10000.csv		0.03274	0.0281	16		0.09167	0.0592	8
6d_100000.csv		0.06408	0.1286	16		0.13147	0.2572	8
9d_100.csv		0	0	16		0	0	8
9d_1000.csv		0.02083	0.011	16		0.05	0.017	8
9d_10000.csv		0.03233	0.0395	16		0.08523	0.1154	8
9d_100000.csv		0.06326	0.1476	16		0.14928	0.2718	8

Figure 7: Strong Scaling Efficiency

4 Conclusions

In conclusion, what we can notice is that, increasing the number (for each version) of threads leads to an improvement in the overall performance of the algorithm in terms of computation time spent. This emphasizes the importance of a good multiprocessing programming architecture, if the goal is to increase the time efficiency. It is obvious that performing tests with a high performance machine and with even larger input datasets, could better highlight the parallel scalability and also the real scalability boundaries.
Another point to discuss is the fact that, in general, the *LOCAL* version is faster than the *ATOMIC* one, due to the absence of a great number of concurrent access to shared structures. Those have to be managed with a `#pragma omp atomic` directive to ensure no race conditions, and that inevitably leads to an overhead. In any case, the implementation presented in this project has some limitations:

- The number of data points in the input dataset cannot be unbounded. The architecture should be revised to adapt it to different contexts.
- This implementation uses C `float` data type for coordinates, so each data point coordinate must stay inside the range of representability of the `float` data type.

One last thought: this project makes it clear that parallel programming is a powerful tool, which allows you to achieve a great improvement in terms of performance if you can take advantage of the tools provided by OpenMP or similar APIs. The key point is the design of a correct and efficient architecture that can adapt to the needs highlighted by the problem.