Parallel Computing
K-Means Algorithm

Diogo Rebelo
Informatics Department
University of Minho
Braga, Portugal
pg50327@alunos.uminho.pt

Henrique Alvelos
Informatics Department
University of Minho
Braga, Portugal
pg50414@alunos.uminho.pt

***Abstract*—Parallelization of the k-means algorithm based on Lloyd's algorithm using *OpenMP* and *MPI* directives, and exploring the different clauses and alternatives for our code. This project is focused on a design methodology development of parallel programs. A discussion is made on the scalability of the created algorithm, through the selection and analysis of metrics that may be relevant in this context.**

***Index Terms*—K-Means, ML, OpenMP, Threads, Performance, Parallel, Optimization, Threads, Lloyd's, Clustering, MPI.**

I. Introduction

The main purpose of this assignment is to understand the advantages of algorithms' parallelization in C, through the development of a third version of K-Means, an optimized and parallelized algorithm. The underlying idea is to make corrections to the version of the previous assignment, improving it through the use of alternative parallelization environments, as MPI. The objective of reducing the execution time is maintained. In addition, there is an analysis of the scalability of the solution, with the appropriate justification for the metrics that were used.

II. Implementation

A. Improvements and Corrections

We made several changes to the previous version. These changes follow below.

1) Reduce the number of functions: First, instead of having the different phases of the algorithm divided by different functions, we no longer have these several functions. So we have just one random number generation function to call in the main function, where most of the rest of the algorithm takes place.

2) Reduce the number of data structures: Second, we reduced the number of data structures used: instead of having an array for each coordinate of the points, we started to use the same array for both coordinates, where the abscissa appears first and the ordinate appears at the next index. This allowed to use less 2 arrays: we just have one array of points and one array of centroids now.

3) Limit the number of iterations criteria: Third, since, for execution purposes, the use of 20 iterations as a stopping criterion was required, we made changes to improve performance in this regard. So, instead of checking the convergence for each iteration, we stopped doing it and started to consider only the 20 iterations. This change arose because, regardless of the number of iterations performed until convergence, this number is limited by 20 iterations, and some time can be significantly reduced by removing the array duplication procedure (from the previous and current state) and its comparison, to check the convergency of the algorithm.

B. MPI analysis

Regarding the MPI directives used, we appealed to their understanding in theoretical terms before selecting which ones would be relevant. In this phase, we begin by starting the execution environment and assigning a unique identifier to each process. Then, we perform the algorithm using communication routines, and OpenMP directives. Finally, we end the MPI environment.

The directives used, accompanied by their description appear next.

1) Differentiating Tasks through Rank:

- `MPI_Init (&argc,&argv)`: Initializes the MPI runtime;
- `MPI_Comm_rank(MPI_COMM_WORLD, &world_rank)`: Used to associate a rank of the requesting process to the communicator;
- `MPI_Comm_size(MPI_COMM_WORLD, &world_size)`: Determines the number of processes in a group associated with a communicator. It is used with the MPI_COMM_WORLD communicator to determine the number of processes being used by an application;
- `MPI_Finalize ()`: Terminates the execution of the MPI environment.

2) Collective communication routines:

- `MPI_Scatter(...)`: Sends data from one process to all other processes in a communicator.
- `MPI_Bcast(...)`: Broadcasts a message from the process with rank "root" to all other processes of the communicator.

Developed in University of Minho

- `MPI_Reduce(...)`: Combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank "root".
- `MPI_Barrier(MPI_COMM_WORLD)`: Blocks until all processes in the communicator have reached this routine.

C. Analysis of MPI flow

This section serves to briefly analyze the flow of MPI messages and how communication between the various processes takes place. It is important to explain how data is distributed according to each policy.

We start by initializing the environment, assigning different ranks to each process. We allocate memory for the centroids array (`centroides`), which stores the coordinates of the points that will be the centroids. Then, The root process randomly creates the points, initializes the centroids, allocates the necessary memory for the array to store the coordinates of the points that belong to each cluster and the arrays that have the sum of each coordinate of the points of each centroid (`NPontosClusters`, `SomatorioCentroidesX`, `SomatorioCentroidesY`). The number of elements of each process (`elements_per_proc`) is defined as the number of points passed, divided by the number of processes multiplied by 2 (`n/world_size * 2`). In this way, the points will be distributed in the same way for each process, we multiply by 2 to consider the 2 coordinates. Next, we initialize a pointer `*sub_points` that will store the points of each process. Its size is the previously determined number of processes. Now, we make the first distribution of the data:

- **`MPI_Scatter:`**

```
MPI_Scatter(pontos, // address of send buffer

elements_per_proc, // number of elements sent
to each process

MPI_FLOAT, // data type of send buffer elements

sub_pontos, // address of receive buffer
 (for each proccess)

elements_per_proc, // number of elements received
for each process

MPI_FLOAT, // data type of receive buffer elements

0, // rank of sending process

MPI_COMM_WORLD); // communicator
```

So what happens is that the root process distributes the coordinates of the points to each child process, so each process will have a different set of data to use and process. Each process will have `n/world_size * 2` elements.

Then we enter the main loop that controls the number of iterations. Starting with a new communication:

- **`Bcast:`**

```
MPI_Bcast(centroides, // starting address
of buffer

k * 2 , // number of entries in buffer

MPI_FLOAT, // data type of buffer

0, // rank of broadcast root

MPI_COMM_WORLD); // communicator
```

So, the root process broadcasts the coordinates of points of each cluster from the process with rank "root" to all other processes of the communicator: the process that will handle the calculation of each point distance.

It is now time to calculate the euclidean distance from each point to each cluster, obtaining this distance and storing it, so that we can find the minimum one, assigning each point to its nearest cluster. Then, we make:

- **`MPI_Gather`**:

```
MPI_Gather(sub_centroideNearPonto, // starting address
of send buffer

elements_per_proc, // number of elements
in send buffer

MPI_FLOAT, // data type of send
buffer elements

centroideNearPonto, // address of receive buffer

elements_per_proc, // number of elements for any
single receive

MPI_FLOAT, // data type of receive buffer elements

0, // rank of receiving process

MPI_COMM_WORLD); // communicator
```

We have already calculated the closest points of each centroid, of each data block that was distributed, so, the result must be united/gathered in the same data block, of the same process, and this is exactly what this directive is for.

It is now time for us to perform the reduction of values on all processes to a single value:

- **`MPI_Reduce:`**

```
// reduction 1
    MPI_Reduce(NPontosClusters, // address of send buffer

    NPontosClusters, // address of receive buffer

    k, // number of elements in send buffer

    MPI_FLOAT, // data type of elements of send buffer

    MPI_SUM, // reduce operation

    0, // rank of root process

    MPI_COMM_WORLD); // communicator
```

This is maybe the most complex operation to understand, but let us clarify. In the first reduction: each process has a block of data points, and has calculated the nearest centroíd for each point. We know that we have points with the same nearest centroid, so we can count the number of points that have the same centroid and use that number as the content of each index in `NPontosCluster`. The idea is the same for the other arrays. In order to clarify what happens, take a look at the next sheme:

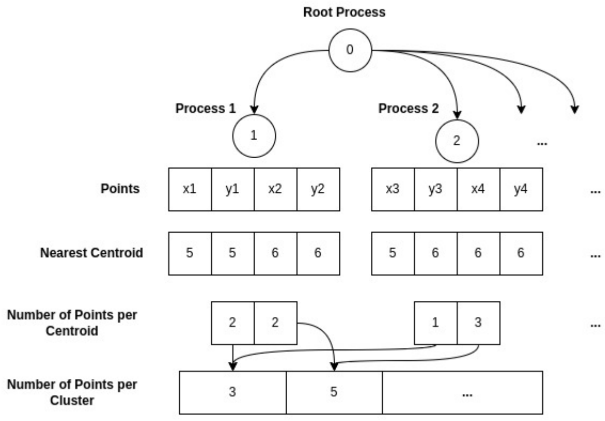


Fig. 1. Process of first reduction in K-means.

It is importante to clarify that there are performed other MPI operations beside the ones showed above. As we can see in the previous diagram, the coordiantes are distributed in blocks for each process to process them (`Scatter`). As the centroids initial configuration is used by all process, the `Bcast` allows the root process to send this information across all the other process. After the calculation of the euclidean distance, each block of information is gathered together, with `Gather`, so all the coordinate have its nearest centroid associated: coordinates of the same point have the same nearest centroid, naturally. Finaly, are performed 3 reductions, but the most important one the the one presented before, because it is responsible for calculating the final number of points that each cluster has.

III. Results

A. Environment

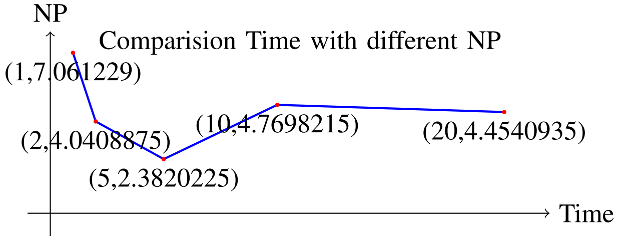
Here are the machine' specs we've used to get our results:

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	8
On-line CPU(s) list	0-7
Thread(s) per core	1
Core(s) per socket	1
Socket(s)	8
NUMA node(s)	1
Vendor ID	AuthenticAMD
CPU family	16
Model	9
Model name	AMD Opteron(tm) Processor 6174
Stepping	1
CPU MHz	2200.000
BogoMIPS	4400.00
Hypervisor vendor	VMware
Virtualization type	full
L1d cache	64K
L1i cache	64K
L2 cache	512K
L3 cache	10236K
NUMA node0 CPU(s)	0-7

TABLE I
Specs of Search Cluster

B. Discussion

In this section, we present the results of the parallel implementation of the K-Means algorithm comparing them to the sequencial implementation.



As we can see, as the number of MPI processes increases, the time it takes to complete the task decreases, reaching the lowest value when 5 MPI processes are used. After that, the time increases again when 10 and 20 MPI processes are used. In general, it's expected that the time to complete the task will decrease as the number of MPI processes increases, this is known as speedup. However, it is also expected that there will be a point of diminishing returns, where adding more processes will not result in a significant decrease in time and, in this case, 5 MPI processes are the best option.

Number of Points	Time Sequential	Time with NP=5	Speed-Up
100	0,0167855	0,328135	0,051
1000	0,0174475	0,304108	0,057
100000	0,088946	0,292766	0,303
1000000	0,72316	0,4357835	1,66
10000000	7,0910315	2,812646	2,51

TABLE II
Variation of number of points with K=4

From the results of table 1, we can see that as the number of points increases (N) , the speed-up provided by using NP=5 decreases. At the lower end, with 100 points, the speed-up is around 19 times faster, but as the number of points increases to 10 million, the speed-up is only 2.51 times faster. This suggests that the method used for parallelizing the computation becomes a little less effective as the number of points increases. Additionally, the sequential time also increases with number of points, which is expected as the complexity of the problem increases.

Number of Centroids	Time Sequential	Time with NP=5	Speed-Up
1	2,3595675	1,4538675	1,62
2	5,0442895	2,0794275	2,42
4	7,114679	2,5354425	2,8
8	15,7547445	3,4234700	4,6
16	28,2366805	7,2599640	3,89

TABLE III
Variation of number of centroids with N=10000000

From the results presented in the table 2, it is apparent that the parallel implementation using NP=5 does not exhibit a consistent increase in performance as the number of centroids

(k) increases. The speed-up value, defined as the ratio of the sequential execution time to the parallel execution time, decreases from 1.62 for k=1 to 3.89 for k=16. This suggests that the parallelization method employed may not be perfectly scalable with respect to the number of centroids.

As we have seen, both tables highlight that as the size of the problem increases, the parallel implementation becomes a little less effective, which is expected. However, the parallel implementation still provides a speedup, even for larger datasets and number of centroids, which can be beneficial in scenarios where computation time is a concern.

Overall, the tables provide valuable information about the performance of the parallel implementation of the K-Means algorithm and can be used to guide decisions about the use of parallelization for similar problems.

IV. Conclusion

This phase of the work allowed to deepen aspects that had not been addressed previously, namely parallelism using MPI message passing. It also allowed to improve implementation aspects that reduced the execution time and to understand how this time varied with the execution of this different type of parallelism.

V. References

[1] https://mpitutorial.com/tutorials/
[2] https://www.mpich.org/