ALTERNATIVE PARALLEL K-MEANS ALGORITHM

Ana Filipa Ribeiro Murta
Informatics Department
Master in Informatics Engineering
University of Minho
Guimarães, Braga, Portugal
pg50184

Ana Paula Oliveira Henriques
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Braga, Portugal
pg50196

***Abstract*—This report aims to present and analyze the performance of an optimized parallel version of a simple k-means algorithm, inspired on the Lloyd algorithm.**

***Index Terms*—algorithm, parallelism, OpenMP, k-means, cluster, optimization, performance, thread, analysis, solution, data races, MPI, task, CUDA, scalability.**

I. Introduction

The purpose of this last phase is to implement a version of the k-means algorithm previously developed that efficiently explores parallelism, resorting to alternative parallel programming environments. Like in the other phases, the team always targeted the reduction of the program's runtime.

The first step taken was correcting some serious problems the solution we formerly submitted carried, which were data races. Secondly, and since the program's runtime was still not satisfying, the team deepened their knowledge about OpenMP primitives to optimize another block of code with high computational loading, processing it by multiple threads.

For matters of testing, the program, that supports a range of 4 to 48 (and more) clusters and assigns 10 000 000 points to their closest cluster, has now another variant that must be taken under consideration while exploring its parallelism with MPI: the number of tasks which refers to the number of processes that are created and run concurrently.

This report will begin by presenting the corrections done to the second phase's solution and the optimizations added to the algorithm using more OpenMP primitives, alongside the results that guided us to take conclusions. At a later stage, the use of other programming environment to explore parallelism, in our case MPI (*Message Passing Interface*), will be not only analysed by introducing its differences from other parallel computing plataforms and the reason that encouraged us to choose it, but also the effects it had on the final solution's performance. The solution's scalability will also be posteriorly explored, resorting to different machines and testing with more data to validate its quality. Towards the end, matters such as performance and efficiency will be discussed after analyzing the results obtained. At the last section, there's a brief conclusion to summarize the work done.

II. Optimizations with OpenMP

In the second phase's report, we had concluded that the computational overhead was concentrated in the function `addToClosestCluster`, turning it into our source to explore the parallelism techniques learned. Moreover, we also presented the two arrays passed as arguments that had to be privatized to each thread so that the new parallel version of this algorithm could be successfully implemented:

- `num_elems` - the number of elements in each cluster;
- `sum` - the sum of the points in each cluster

Our biggest problem in the former phase was the presence of data races in the algorithm's implementation because of the variables `minDistance`, `minCluster` and `newDistance` that were used in the second parallelized loop, but defined outside of it. A data race occurs when two or more threads access the same shared memory location concurrently and at least one of the accesses is a write. This can lead to a range of issues, such as incorrect results and program crashes. Therefore, the first measure taken was to correct this problem and, for that, we had two options: defining those variables in each thread according to its specific value; or else using the OpenMP's `private` clause.

- **private(var)** – a clause in OpenMP used to specify variables that should have private storage within a parallel region. This means that each thread can access and modify its own private copy of the variable without affecting the values of the variable for other threads.

One way or another, each thread ends up having its own copy of those variables and can modify them independently without affecting the other threads. Although the `reduction` clause also avoids the need for explicit synchronization to protect the shared variables, that was insufficient to ensure the non-existence of data races.

- **reduction (operator:var)** – used to specify private variables that should be reduced across multiple threads in a parallel region using an operator into a single value.

The reduction clause in OpenMP is used to specify variables that should be reduced across multiple threads within a parallel region. A reduction is a operation that combines the values of a variable from multiple threads into a single value.

After fixing these serious issues, the team spent some time thinking of new ways to improve the previous version and came up with the idea of parallelizing the first loop of `addToClosestCluster`, which calculates the new centroids for each cluster, through the OpenMP's directive `#pragma omp parallel for schedule(static)`.

III. Software Platforms for Parallelism

In the theoretical classes of this course, we learned that there are many hardware and software platforms available for parallel computing. Focusing on the software platforms, in addition to OpenMP (*Open Multi-Processing*), there are many others, such as CUDA (*Compute Unified Device Architecture*), MPI (*Message Passing Interface*) and OpenCL (*Open Computing Language*). Choosing one of these platforms for a program depends on its specific requirements and constraints, as well as the hardware and software resources available.

As four pratical classes were dedicated to study CUDA and MPI, the group decided to adopt between one of these platforms, starting by analyzing the differences between both:

MPI	**CUDA**
A parallel computing plataform implemented for multiple processes running on different machines to work together by exchanging messages through a network.	A parallel computing platform built for GPU (*Graphics Processing Unit*) acceleration. It allows developers to use the power of GPUs to accelerate tasks with higher computing intensity in their applications.
Used for programming distributed memory systems, where each processor has its own memory and communicates with other processors through message passing;	Used on shared memory systems, where all processors have access to a common memory spaces;
It relies on a message passing interface to communicate data between processors;	It includes a number of features and mechanisms specifically designed for shared memory programming, such as shared memory, atomics, and parallel execution of GPU threads;
Supported on a wide range of platforms and hardware architectures;	Only supported on NVIDIA GPUs due to being proprietary to NVIDIA.
It has a large and active user community, with a wide range of tools and libraries available for tasks such as debugging, profiling, and optimization;	Its ecosystem is more focused on specific use cases, such as high-performance computing and machine learning.

TABLE I
Differences between MPI and CUDA

When it comes to improving the performance of parallel applications, both CUDA and MPI are functional. However, the decisive point lays in the type of parallelism desired. While CUDA is designed for data parallelism where the same operation is performed on multiple pieces of data concurrently, MPI is designed for task parallelism where different tasks are performed concurrently by different processes. Considering this to the parallel k-means algorithm we developed, there are two efficient methods that can optimize it:

A) Resorting to data parallelism, the algorithm can be parallelized by dividing the data into chunks and each chunk is processed by a GPU. This can improve its performance by allowing it to take advantage of the high parallelism and high-bandwidth memory of the GPU.

B) Resorting to task parallelism, the algorithm can be parallelized by dividing the data into chunks and assigning each chunk to a separate process for processing. This can improve the performance of the algorithm by allowing it to run on multiple machines in a cluster.

After analysing every difference between these two platforms, the group decided to utilize both MPI and OpenMP to explore the parallelism in our k-means algorithm. In fact, there are many reasons why this decision might be beneficial:

1) The combination of MPI, appropriate for tasks that require data to be distributed across multiple nodes or systems, with OpenMP, suitable for shared memory programming on a single node, offers greater flexibility in exploiting the program's parallelism.
2) It can potentially lead to a better performance as it allows for more rigid control over the parallelism in the application. For example, MPI can be used to distribute data across multiple nodes while OpenMP can be used to parallelize computations within each node.
3) Both software platforms have large and active user communities, with several tools and libraries available for tasks such as debugging, profiling, and optimization. The two of them are also supported on a wide range of platforms and hardware architectures.

IV. Exploring Parallelism with MPI and OpenMP

In the `main` function, the algorithm starts by initializing MPI and sets the number of threads through the OpenMP's `omp_set_num_threads` function. The number of clusters and the number of threads are given as input arguments from the command line. Preserved from the second phase's version, the `sum` matrix holds the sum of each cluster's centroids' x and y coordinates, the `num_elems` array keeps the number of points in each cluster and the `centroids` matrix stores the coordinates of each cluster's centroids.

The `init` function is called to initialize the N points and the K centroids with the first K points, as well as setting the initial values for the `sum` and `num_elems` arrays based on the first K centroids. Posteriorly, the total size of points (N) is divided among different processes. The `scattered_points` array is allocated with enough space to store the data that will be sent to the current process. The `MPI_Scatter` function is called to asynchronously send the points chunks from the root process to every other process.

After concluding this distribuition, the algorithm then calls the addToClosestCluster function to assign each point to the closest cluster, calculating the distance between each point and the clusters' centroids. It also updates the sum and num_elems arrays to reflect the new assignment.

```
int points_chunks = N / num_processes;
float *scattered_points =
        (float*)malloc(points_chunks * 3
                        * sizeof(float));
MPI_Scatter(points, points_chunks * 3,
            MPI_FLOAT, scattered_points,
            points_chunks * 3, MPI_FLOAT,
            0, MPI_COMM_WORLD);
```

After this initial stage of the program, the addToClosestCluster function is executed repeatedly until the algorithm has not converged or the maximum number of iterations (defined as MAX_ITERATIONS) is reached and, at the end of each iteration, the new centroids are discovered. The code block where the clusters' centroids are recalculated is executed by multiple threads in parallel through the OpenMP's directive #pragma omp parallel for. The clause schedule(static) specifies that the loop should be statically scheduled, meaning that the iterations are divided into equal-sized blocks and assigned to different threads.

Before the new assignment of the points to their closest cluster, that occurs in a loop, also executed by multiple threads in parallel, where each process works only on its chunk of points, the root process broadcasts the updated centroids to every other process in the MPI_COMM_WORLD communicator so that they can have the same copy of the centers. For that, it is used the MPI_Bcast function, that resorts to efficient algorithms that can minimize the time and resources required.

As mentioned before, in the code block where the points are reassigned to their closest cluster, it is specified the #pragma omp parallel for directive and the reduction clause, where the operation done is +, meaning that, at the end, each element in the sum and num_elems arrays will contain the sum of the corresponding elements across every thread.

```
MPI_Bcast(centroids, K * 2, MPI_FLOAT,
          0, MPI_COMM_WORLD);

#pragma omp parallel for schedule(static)
        reduction(+:sum[:(K * 2)])
        reduction(+:num_elems[:K])
        reduction(+:allEquals)
for (int i=0; i<points_chunks*3+=; i  3){
    // Reassignment of points
}
```

Once the reassignment is finished, the chunks of points processed by each process are gathered into the root process, using the MPI_Gather. In other words, it asynchronously gathers the data from every process and stores it in a specified array in the root process.

Another MPI function that takes place in our code is MPI_Reduce, a collective communication function used to perform a reduction operation on the data provided by every process in a communicator and return the final result to the root process. In our case, it combines the *sum* and *num_elems* arrays from all processes running and return the global_sum and the global_num_elems arrays to the root process.

If the rank of the current process is 0 (meaning it's the root process), there's another loop that updates the centroids based on the values gathered in global_sum and global_num_elems. To calculate the new centers, the results in global_sum are divided by the total number of elements stored in global_num_elems.

```
float global_sum[K * 2];
int global_num_elems[K];

MPI_Gather(scattered_points, points_chunks*3,
    MPI_FLOAT, points, points_chunks*3,
    MPI_FLOAT, 0, MPI_COMM_WORLD);

MPI_Reduce(sum, global_sum, K*2, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(num_elems, global_num_elems,
    K, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0){
    for (int j = 0; j < K * 2; j += 2){
        // Recalculation of centroids
    }
}
```

Back to the main function, the process with rank 0 (the root process) prints the output results and then the MPI is finalized using MPI_Finalize. To identify the current process executing, in the beginning of the main function, after initializing MPI, there's MPI_Comm_rank for that purpose.

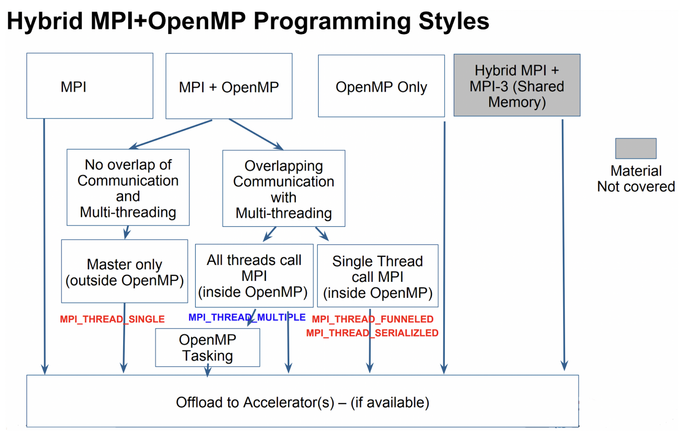
Hybrid MPI+OpenMP Programming Styles
MPI
MPI + OpenMP
OpenMP Only
Hybrid MPI + MPI-3 (Shared Memory)
Material Not covered
No overlap of Communication and Multi-threading
Overlapping Communication with Multi-threading
Master only (outside OpenMP)
All threads call MPI (inside OpenMP)
Single Thread call MPI (inside OpenMP)
MPI_THREAD_SINGLE
MPI_THREAD_MULTIPLE
MPI_THREAD_FUNNELED
MPI_THREAD_SERIALIZLED
OpenMP Tasking
Offload to Accelerator(s) – (if available)

Fig. 1. Hybrid MPI-OpenMP Programming

A. Commands for testing

```
$ make clean
$ make k_means
$ make sbatch NTASKS=2 CP_CLUSTERS=4 THREADS=32
```

V. Solution's Scalability

To study the solution's scalability, it's good practice to run the program giving different numbers of points, numbers of clusters and numbers of threads, and measuring the time it takes to complete the computation as the input size increases. Another method to analyse an algorithm's scalability is running it on different hardwares, like a CPU with more cores, and compare the performances to see how it scales in different architectures. The CPU's specifications of the hardwares provided by this group are represented in Table II.

CPU Specifications			
Model Name	Number of CPUs	Num of cores per CPU	Num of threads per CPU
Intel® Core™ i7-8550U CPU @ 1.80GHz 1.99GHz	-	4	8
Macbook Air (M1, 2020), Apple M1 chip	8	8	8

TABLE II
CPU specifications of the team's hardwares

In terms of L1, L2, and L3 cache, the solution is likely to scale well as long as the size of the data fits within the cache of each processor. The main factors that may affect the cache performance are the number of threads and the amount of data that each thread accesses. In this implementation, that amount is limited to the subset of points assigned to it, which should not significantly spoil the cache performance. The same applies to RAM as long as the data's size fits within the available RAM of each processor. However, worst case scenario, if L1, L2 and L3 cache sizes are not large enough or the points array's size exceeds the available RAM, then this may result in cache misses and lower performance. Presented down below, there's Table III that illustrates the specifications of our machines's memory hierarchy levels.

Memory Hierarchy Specifications				
Model Name	L1 Cache Size	L2 Cache Size	L3 Cache Size	RAM Size
Intel® Core™ i7-8550U CPU @ 1.80GHz 1.99GHz	64KB	-	8MB	8GB
Macbook Air (M1, 2020), Apple M1 chip	32KB	32MB	-	16GB

TABLE III
Memory hierarchy specifications of the team's hardwares

Considering that the workload is evenly distributed among the processes and there is sufficient available memory to store the data, the program should have no problem scaling as the number of processes and cores increases. However, the sizes of the `sum`, `num_elems`, and `centroids` arrays are proportional to the number of clusters (K), which means that if this number increases, the size of these arrays will also increase, potentially leading to more memory usage and caches misses. One last factor that may reflect on an negative impact is the function `calculateDistance`, which involves floating-point arithmetic because of conditioning elements such as the speed of the processor's floating-point units and its ability to perform multiple floating-point operations in parallel.

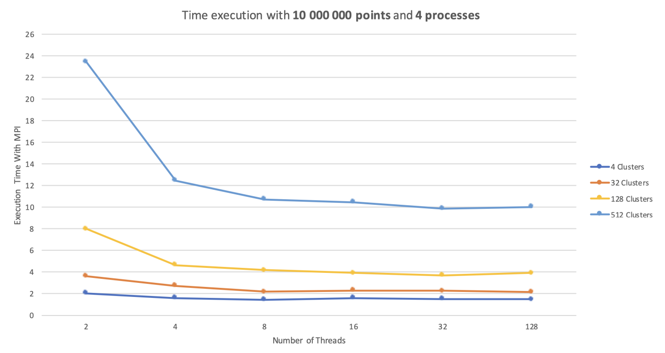

Fig. 2. Different number of clusters

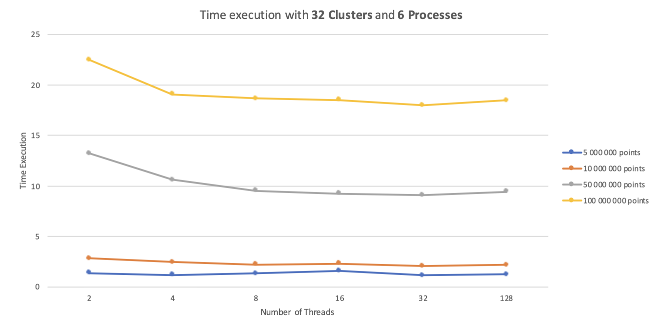

Fig. 3. Different number of points

The solution's evaluations structured above can be verified in Figs. 2 and 3, where the execution time improves as more threads are added for processing. Nevertheless, this doesn't mean that the runtime will always increase proportionally; there's a point where the algorithm can't scale anymore. Another analysis that we can take from these Figures is that the algoritm doesn't scale well after a certain number of clusters and a certain number of points.

VI. Explaining Results

The performance and quality of the results will depend on various variables such as the number of points, the number of clusters, the number of threads, the number of iterations and specially the initial centroids atribuited to the clusters.

This k-means algorithm has a time complexity of `O(N*K*I)`, where N is the number of points, K is the number of clusters, and I is the number of iterations of `main`'s loop. This means that the time required to run it increases with the number of points, clusters, and iterations. There are also some additional operations that are performed in parallel, like calculating the new centers and summing the results returned by every process, but these operations are typically much faster than the loop and do not significantly affect the overall time complexity.

In this code, the `addToClosestCluster` function is executed in parallel on multiple CPU cores. This can potentially improve the performance of the algorithm by reducing the time required to complete the iterations. However, the actual performance improvement will depend on the number of CPU cores available and the workload distribution among the cores.

This implementation also uses the platform MPI to distribute the workload among multiple processes running on different machines. This can further improve the performance of the algorithm by allowing it to be run on a distributed computing environment with multiple machines. However, the actual performance improvement will depend on the number and capabilities of the machines in the environment, as well as the workload distribution among the processes.

In addition, calling `MPI_Bcast` instead of `MPI_Send` may also contribute for a better performance. Even though the time taken by each one of these functions increases linearly with the number of processes, `MPI_Send` is a point-to-point communication function that allows a process to send a message to a specific destination process, unlike `MPI_Bcast`. So, to broadcast a message to multiple processes with `MPI_Send`, the root process would need to send the message to all of them individually, which can be inefficient, particularly for large communicators, as it requires multiple send operations and potentially involves more communication overhead.

In general, the time execution results will depend on the balance between the complexity of the algorithm and the parallelism and distribution techniques used to accelerate it.

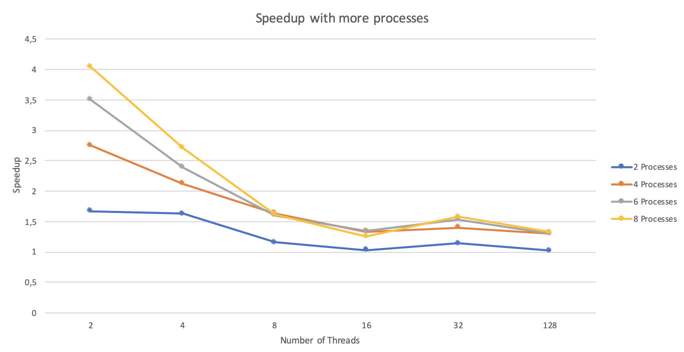

Fig. 4. Speedup obtained with the increase of processes

Represented in Fig. 3, there's the time execution speedup obtained comparing an implementation with no MPI and another with MPI. It's important to keep in mind that the speedup will increase as the number of processes increases. However, at some point it will start to decrease due to the overhead of communication between processes. The ideal number of processes will depend on the specific characteristics of the program and the hardware it is running on.

For matters of optimization, this k-means algorithm could consider the following ideas:

- Rather than recalculating the clusters' centroids at each iteration, which has a time complexity of `O(K*K)`, they could only be updated for the clusters that have points added or removed in the current iteration, which would reduce the time complexity to `O(K)`.
- Use a more efficient data structure for storing the points and centroids, such as a *k-d tree* or a *quadtree* to store the points, which would allow for faster search and insertion of points. Currently, the points and centroids are stored in flat arrays, which can be inefficient for large datasets.
- Perhaps, pre-allocate memory for the `sum` and `num_elems` arrays to reduce the overhead of dynamically allocating memory at each iteration.
- Use compiler optimization flags when compiling the program to improve the performance of the program. After all, flags like `-O2` and `-O3` are options that control the level of optimization applied to the compiled code.

VII. Conclusion

By accomplishing this final project, the group was able to consolidate what they learned in the scope of this course, regarding not only about the various types of parallel computing platforms and their differences but also about MPI and the impact of its implementation when combining with OpenMP.

Despite some details we wished we had improved, from a different stopping criterion, like when the clusters' centers no longer change, to the application of the optimization ideas previously indicated, the team believes to have successfully addressed every requirement demanded in this report.

For future work, we would like to experiment other parallel computing platforms, specially CUDA since it is highly recommend for data machine algorithms, and compare its results with the results obtained with this hybrid MPI-OpenMP implementation. Furthermore, executing more tests with different metrics and elaborate more comparison analysis to explain results would also be something to consider in the future.