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*, *parallel*, *OpenMP*, *k-means*, *cluster*, *optimization*, *performance*, *thread*, *analysis*, *solution*, *metrics*.**

I. Introduction

The purpose of this second phase is to implement, through OpenMP primitives, a parallel version of the k-means algorithm previously developed, where the code blocks with higher computational loading are processed by multiple threads.

Whereas, in the first phase, the number of points and clusters were static, the program must now reach a range of 4 to 32 clusters and assign 10 000 000 points to their closest cluster. For reasons of algorithm validation, it was required to set the number of the loop's iterations – where the clusters' centers are recalculated and the points are reassigned to their closest cluster – up to 20.

This report will begin by presenting the optimizations made throughout the implementation, alongside the metrics considered and their results. 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. Parallel Implementation

Firstly, and as mentioned before, we started by identifying which code blocks had the greater computational loading. This analysis led us to conclude that the computational overhead was concentrated in the function `addToClosestCluster`, turning it into our source to explore the parallelism techniques learned. In resume, this function is responsible for assigning the N points to their closest cluster and, for that matter, there's a loop that iterates N points and reassigns them to their closest cluster as the K centroids are recalculated. Also in this function, there are two arrays, passed as arguments, that had to be taken under analysis so that the new parallel version of this algorithm could be successfully implemented. The first array is named `num_elems` and it keeps the number of elements in each cluster; the second one is known as `sum` and it keeps, as the name suggests, the sum of the points in each cluster. To build the parallel region here, we had to come up with a solution that would allow us to private these variables to each thread because, since they were defined outside this region, they would be shared as soon as the parallel region is encountered, causing wrong results as output.

The necessity of privatizing these variables comes from the case where two threads are adding their respective point to the same position of the array sum or increasing some value in the same position of the array num_elems at the same time. When added at the same time to the same destination, there will be a conflict writing both results.

After analysing different alternatives to explore the parallelism, we considered using `#pragma omp critical` when accessing these two arrays, but quickly realized that this was not the best solution.

- **#pragma omp critical** – used to identify a section of code that must be executed by a single thread at a time.

We later decided on a better practice by opting to use `#pragma omp parallel for reduction (operator:var)` instead because the OpenMP reduction clause has a much lower overhead than a critical section. This way, we can specify the arrays created as a private copy to each thread and, in the end of the parallel region, the final value is the result of a private variables' reduction according to the operator +.

- **#pragma omp parallel for reduction (operator:var)** – used to specify one or more private variables to each thread. The final value is the result of reduction of private values using the operator defined.

Another significant detail is that we called the OpenMP function `omp_set_num_threads` in the main fuction to indicate the number of threads that are going to be used in the parallel regions. The number of threads passed as argument is read from input as required.

- **omp_set_num_threads(int number_of_threads)** – used to affect the number of threads used for subsequent parallel regions that do not specify a num_threads clause, by setting the value of the first element of the nthreads-var ICV of the current task.

III. Solution Evaluation

As we can conclude from Table 1, the execution time starts to get worse from a certain number of threads. However, increasing the number of threads doesn't mean that the execution time will always increase proportionally; there's a point where the algorithm can't scale anymore.

The reason behind this can lay in the task management, required when applying parallelism, i.e. additional work that needs to be done. If it happens to take on excessive task granularity, the algorithm may face scalability problems.

Any time a task spends waiting for another task, also known as synchronization overhead, may cause scalability problems as well due to dependencies. On the other hand, running tasks without synchronization can provide wrong results.

Another reason for scalability problems can also come from load imbalance, which results from over-decomposition. However, we don't consider that load imbalance is the case in our algorithm because, since the schedule clause is not defined, by default it is static. Consequently, due to running only one parallel region, the number of instructions, #I, doesn't change from thread to thread.

Evaluating our solution, we guess that the threads actually stop running parallel after a certain point, explaining why the algorithm stops scaling after that point.

Clusters	Cores	Threads	Execution Time (sec)
4	2	2	3.482
		4	2.626
		6	2.602
		8	2.423
		10	3.322
		20	3.375
		30	3.556
		50	3.585
		128	3.508
	3	2	2.395
		4	2.389
		6	2.379
		8	2.383
		10	2.387
		30	2.387
	4	4	1.801
	6	4	1.570
	8	8	1.252
	10	10	1.363
32	2	2	9.935
	4	4	9.313
	8	8	4.303
	10	10	3.516

Fig. 1. Parallel Version Results

In Table 2, it's presented the time results obtained, in the first phase, after executing the sequential version of this algorithm. Comparing both Tables, we can verify a huge improvement and optimization in the solution.

N Points	Flags		Time (sec)
1000	-O2		0.003
	-O3		0.003
	-O2	-funroll-loops	0.002
		-ftree-vectorize -mss4	0.003
	-O3	-funroll-loops	0.002
		-ftree-vectorize -mss4	0.003
1000000	-O2		0.875
	-O3		0.658
	-O2	-funroll-loops	0.793
		-ftree-vectorize -mss4	0.556
	-O3	-funroll-loops	0.725
		-ftree-vectorize -mss4	0.753
10000000	-O2		8.565
	-O3		7.306
	-O2	-funroll-loops	7.802
		-ftree-vectorize -mss4	5.358
	-O3	-funroll-loops	7.193
		-ftree-vectorize -mss4	7.374

Fig. 2. Sequential Version Results

IV. Conclusion

After putting in practice the content learned in the practical classes, the group believes to have presented a sufficient and optimized parallel solution for the k-means algorithm previously implemented. Therefore, and given every requirement demanded for this phase, we considered to have successfully completed this project. As future work, the group would like to explore more techniques for parallelism as well as learn more about the OpenMP API.

```
int addToClosestCluster(
        int count, int K, int num_elems[K], float centroids[K * 2], float sum[K * 2]
){
    // ...

    int sizeSum = K * 2;
    int sizeNumElems = K;

#pragma omp parallel for reduction(+:sum[:sizeSum]) reduction(+:num_elems[:sizeNumElems])
    for (int i = value * 3; i + 2 < N * 3; i += 3)
    {
        beforeCluster = points[i + 2];
        minDistance = calculateDistance(centroids[0], centroids[1],
                                        points[i], points[i + 1]);
        minCluster = 0;
        for (int j = 2; j + 1 < K * 2; j += 2)
        {
            newDistance = calculateDistance(centroids[j], centroids[j + 1],
                                            points[i], points[i + 1]);
            if (newDistance < minDistance)
            {
                minDistance = newDistance;
                minCluster = j / 2;
            }
        }
        sum[2 * minCluster] += points[i];
        sum[(2 * minCluster) + 1] += points[i + 1];

        if (beforeCluster == minCluster)
            allEquals++;
        points[i + 2] = minCluster;
        num_elems[minCluster]++;
    }
```