Lloyd's K-means Clustering algorithm - OpenMP

1st Rita Gomes
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG50723

2nd Pedro Araújo
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG50684

***Abstract*—This report consists of presenting an optimized version of a k-means algorithm based on Lloyd's algorithm using OpenMp primitives, developed by the group in C programming language.**

***Index Terms*—scalability, perfomance, OpenMP, optimization, metrics**

I. Introduction

The aim of this phase of the project is to evaluate the ability of the students to develop programs that explore parallelism with the main objective of reducing program execution time through the use of C programming language and the Search application as programming interface. For this, the Lloyd's K-means Clustering algorithm will be studied and the main objective is to implement it in the most efficient and appropriate way possible. Note that the work done in the first phase of the project will be reused and modified according to the new requirements requested for this phase. To carry out this phase of the project, students are suggested to identify which code blocks have the highest computational load, present and analyze different alternatives for exploring parallelism for each block identified and select the most viable alternative. At the end, through the analysis of scalability and load balancing, the final results of the experiment allow evaluating the performance of the algorithm.

The input will be generated again through the function developed for the first phase of the work, but at this stage the number of points and samples must be passed as parameters. Therefore, the program must receive 3 input parameters: 1st number of points, 2nd number of clusters and 3rd number of running threads.

II. Exploration and evaluation of the solution

In order to implement parallelism in our algorithm, the team had to use some clauses from the OpenMP API such as:

- **pragma omp set num threads**: affects the number of threads to be used for subsequent parallel regions that do not specify a num_threads clause.
- **pragma omp atomic**: ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location.
- **pragma omp for**: which instructs the compiler to distribute loop iterations within the team of threads that encounters this work-sharing construct.
- **pragma omp for reduction**: the reduction operation is applied to private variables and result is aggregated to the shared variable

The function presented below is the one that consumes the most in terms of the number of instructions: 20*N*K*2. Therefore, the group decided to put the local variables inside the *pragma omp for* loop to make them private. In this way, each thread can run regardless of *pragma omp atomic*. This is necessary because there is a data racing in the shared variable (cluster[]) and by applying this parallelism each thread updates the state of the variable each time. The group preferred to apply the *pragma omp atomic* over the *pragma omp critical*, because the first one does not make the entire statement atomic, it is just atomic in memory. With this strategy, compilers perform special instructions that perform better than *pragma omp critical*.

```
void associaPontos(Points *cluster, Points *pontos, int N, int K){
  //int flag = 0;
  int i;
  #pragma omp parallel for
  for(i = 0; i < N; i++){
          int clusterAntigo,clusterIdMin,j;
          float distMin;
          Points ponto,centroidMin;
          float distancias[K]; // vetor que irá conter as distanci
          ponto = pontos[i];
          for(j = 0; j < K; j++){
              distancias[j] = calculaDist(ponto,cluster[j]);
              distancias[j+1] = calculaDist(ponto,cluster[j+1]);
              j++;
          }
          clusterIdMin = ponto.idC_size;
          clusterAntigo = clusterIdMin;
          centroidMin = cluster[clusterAntigo];
          distMin = calculaDist(ponto,centroidMin);
          for (j = 0; j < K; j++)
          {
             if (distancias[j]<distMin){
                 distMin = distancias[j];
                 clusterIdMin = j;
             }
          }
          //realizar troca de cluster
          if(clusterIdMin!=clusterAntigo){
              //flag=1;
              pontos[i].idC_size = clusterIdMin;
              #pragma omp atomic
              cluster[clusterAntigo].idC_size--;
              #pragma omp atomic
              cluster[clusterIdMin].idC_size++;
```

Fig. 1. Application of *pragma omp for* and *pragma omp atomic*

The following function executes 20*N. Here, the group decided to put the local variables inside *pragma omp for reduction* to be private and run independently. We decided to apply this type of *pragma omp*, because there is data racing in the variables **acumulaX** (vector of the summed x positions of each cluster) and **acumulaY** (vector of the summed y

positions of each cluster) and, therefore, it was necessary to create a copy of these variables in each thread, and after executing them all, sum all the results in the same variable. This was considered the best option comparing to another mutual exclusions pragma omp clauses, because the wait for each threads are performed at the end of for statement.

```
void recalculaCentroid(Points *cluster, Points *pontos, int N, int K, float acumulaX[], float acumulaY[]){

    /* realiza soma dos pontos */
    // local de paralelismo com reduce com cuidado
    #pragma omp for reduction(+:acumulaX[:K]) reduction(+:acumulaY[:K])
    for(int y = 0; y < N ;y++){
        int clustId;
        Points ponto;
        ponto = pontos[y];
        clustId = ponto.idC_size;
        acumulaX[clustId] += ponto.x;
        acumulaY[clustId] += ponto.y;
    }
```

Fig. 2. Application of *pragma omp reduction*

A. Scalability analysis and Load balancing

Was used the perf stat with different combinations to obtain the results expressed in the tables below. In addition, the group also decided to present load balancing and computational intensity in the table.

	Parallel					
	4 clusters			32 clusters		
	2 threads (cpus)	8 threads (cpus)	16 threads (cpus)	2 threads (cpus)	8 threads (cpus)	16 threads (cpus)
Cycles	36.383.615.608	60.040.217.813	91.667.115.931	213.739.647.746	233.167.825.282	253.535.954.045
Instructions	35.375.496.957	35.692.727.177	36.221.835.336	178.287.775.316	178.746.123.737	179.265.426.393
Execution time (seconds)	6.0	3,17	2.69	34.57	10.37	6.18
L1 misses	82.976.497	174.560.767	217.428.581	93.702.134	234.047.289	371.960.909
#I / thread	17.687.748.479	4.461.590.897	2.263.864.709	89.143.887.658	22.343.265.467	11.204.089.150
#I / L1 misses	426.33	204.47	166.60	1902.71	763.72	481.95

Fig. 3. Scalability analysis for parallel code

	Sequencial	
	4 clusters	32 clusters
Cycles	40.725.178.770	130.103.224.278
Instructions	39.553.696.185	178.124.710.425
Execution time (seconds)	9,50	43,10
L1 misses	140.245.357	90.935.712

Fig. 4. Scalability analysis for sequencial code

From the analysis of these two tables, we can see that as the number of threads increases, the number of cycles ad L1 misses increase too. Calculating the theoretical load balancing (since it is not certain that the instructions were equally distributed in the threads) we can see how the core work is reduced, leading to a reduction in the execution time.

The group decided to use one of the most common metrics in parallel applications, the speed-up, and present the results in a graph.

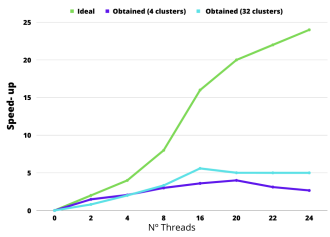

Fig. 5. Speed-up analysis

The ideal speed-up would be proportional to the number of cores, which is not the case. This behavior was already expected analysing the table of parameters showed before. The difference between ideal and obtained speed-up is explained most because of mutual exclusion sentences, as the number of threads grows, more wait time for each thread to access memory, consequently increases the number of L1 misses and cycles.

B. Execution profile measurement

In order to measure the application time profile (profiling), the group decided to use the **gprof** command and got the following:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 85.31      1.51     1.51                             frame_dummy
 10.73      1.70     0.19       20     9.50     9.50  recalculaCentroid
  3.95      1.77     0.07        1    70.00    70.00  inicializa

            Call graph


granularity: each sample hit covers 4 byte(s) for 0.56% of 1.77 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     85.3    1.51    0.00                 frame_dummy [1]
-----------------------------------------------
                                                 <spontaneous>
[2]     14.7    0.00    0.26                 main [2]
                0.19    0.00      20/20          recalculaCentroid [3]
                0.07    0.00       1/1           inicializa [4]
-----------------------------------------------
                0.19    0.00      20/20          main [2]
[3]     10.7    0.19    0.00      20           recalculaCentroid [3]
-----------------------------------------------
                0.07    0.00       1/1           main [2]
[4]      4.0    0.07    0.00       1           inicializa [4]
-----------------------------------------------
```

Fig. 6. Execution profile measurement

The results were not conclusive with this report, but the group considers that the **frame dummy** function is the **associaPontos** function and, as we had previously mentioned, it is the one that consumes the most number of instructions (85%)

III. Conclusion and Future Work

With the accomplishment of this practical work, the group concludes that it was possible to consolidate knowledge taught during the classes. It was also a great way for us to explore OpenMP and practise more about optimizing code, whether parallel or sequential.

The group believes that there is room for improvement in future work, especially the challenge of exploring further and solving scalability issues, in order to further improve the performance of the program.

References

[1] OpenMP, https://www.openmp.org/