Parallel Computing Project 2

1st Lucas Carvalho
Minho's University
pg50555
Braga, portugal
pg50555@alunos.uminho.pt

2nd Duarte Moreira
Minho's University
pg50349
Braga, Portugal
pg50349@alunos.uminho.pt

***Abstract*—This phase of the practical work aims to assess students ability to develop programs that explore parallelism with the main objective of reducing the execution time of the program.**

***Index Terms*—optimization, parallelism, performance, execution time, k-means**

I. Introduction

As a second assignment for the PC - Parallel Computing - subject, we were now proposed to analyse the first version of the program (sequential k-means algorithm) and improve it to a parallel version via **OMP** primitives.

II. Code Analysis

As seen in project 1 and presented in **Fig.1**, the functions with the highest computational load are *cluster_points* (**Fig.2**) followed by *reevaluate_centers* (**Fig.3**).

Since *reevaluate_centers* does not seem to present any problems in terms of parallelization, we will focus the analysis on *cluster_points*, the most problematic function. As we can see, this is very likely to generate race conditions in the last two lines, since if several threads calculate the same cluster_id, they will access and write at the same time in shared memory.

To deal with this problem, two solutions were considered. The first would be to treat the zone as critical and use the **omp critical** directive, which would make this zone accessible by only one thread at a time. As expected, this solution would greatly harm the execution time the greater the number of threads.

Therefore, we opted for the second option. This consisted of slightly changing the data structure according to the number of threads. The idea here is that each thread contains its own memory space so that there are no reading and writing conflicts between threads, thus allowing greater flexibility in terms of scalability.

III. Code Alterations

A. Algorithm

In terms of changes made to the algorithm, since the stopping criterion became a maximum number of 20 iterations, the functions that checked whether the clusters had already converged and also the one that maintained the previous positions of the clusters were removed, since its uses were no longer justified.

B. Data Structure

Regarding the data structure (**Fig.4**), as previously mentioned, the field related to the previous position of the cluster was removed from struct **Cluster**. Three global variables corresponding to the number of samples, clusters and threads were added. Finally, the **clusters** variable was changed, and now it represents a pointer to a set of clusters, simulating a TxK matrix (T threads and K clusters).

Changing the **clusters** variable will now allow each thread to maintain the state of all clusters individually. Take as an example a case where T=2 and K=4, so we will have a list of 8 clusters where thread 0 is responsible for the first block of 4 clusters and thread 1 for the remaining 4.

C. Parallel Implementation

Fig.5 shows the implementation of the parallel version of the *cluster_points* function. For this, the **omp parallel for** directive was used, which will allow the code inside the loop to be distributed and executed by the threads. As these iterations are performed on the points, it was necessary to pay attention to their distribution among the threads. Thus, to maintain the order in which the points are worked, we chose to use the clause **schedule(static,N/T)** (N samples and T threads) since the sum of **floats** is not associative.

To take greater advantage of parallelization, the **omp parallel for** directive was used again for the functions *reevaluate_centers* - **Fig.6** - and *update_clusters_size* - **Fig.7**. Unlike the previous one, these loops iterate over the number of clusters and the order is not relevant, thus opting for a default/static distribution.

IV. Performance Analysis

As a way of analyzing the performance of the developed program, several tests were carried out where the number of clusters (4 and 32), the number of threads (1,2,4,8,16 and 32) and the number of **CPU's** (1,20 and 40) were varied. The results can be analyzed using the graphics in figures **Fig.8**, **Fig.9** and **Fig.10**.

Its analysis allows us to conclude that the use of threads or parallelism is more efficient when the number of clusters is higher. In addition, we also realize that a greater number of **CPU's** does not always imply better performance, as can be seen in the last two graphs.

Thus, the best results achieved within the tests carried out were obtained using 20 **CPU's** and 32 threads for a sample of 10000000 points and 4 and 32 clusters. These were carried out on the provided **Shearch** machine.

ATTACHMENTS

```
# Samples: 32K of event 'cycles:ppp'
# Event count (approx.): 23286877137
#
# Overhead  Command  Shared Object     Symbol
# ........  .......  ................  ......................
#
    87.55%  k_means  k_means           [.] cluster_points
     7.69%  k_means  k_means           [.] reevaluate_centers
     1.70%  k_means  libc-2.17.so      [.] __random_r
     1.23%  k_means  libc-2.17.so      [.] __random
     0.59%  k_means  k_means           [.] initialize
     0.51%  k_means  [unknown]         [k] 0xffffffffa1195377
     0.36%  k_means  libc-2.17.so      [.] rand
     0.08%  k_means  k_means           [.] rand@plt
     0.02%  k_means  [unknown]         [k] 0xffffffffa158c48a
     0.01%  k_means  [unknown]         [k] 0xffffffffa158ca5b
     0.01%  k_means  [unknown]         [k] 0xffffffffa158c4ef
     0.01%  k_means  [unknown]         [k] 0xffffffffa158cbc0
```

Fig. 1. Perf report

```
void cluster_points() {
    for (register int i = 0; i < N; i++) {

        register float lowest_dist = dist(points[i],clusters[0].pos);
        register int cluster_id = 0;

        for (register int j = 1; j < K; j++) {
            register float distance = dist(points[i],clusters[j].pos);

            if (distance < lowest_dist) {
                lowest_dist = distance;
                cluster_id = j;
            }
        }

        clusters[cluster_id].points[clusters[cluster_id].size] = points[i];
        clusters[cluster_id].size++;
    }
}
```

Fig. 2. Cluster_points code

```
void reevaluate_centers() {
    for (int i = 0; i < K; i++) {
        float x = 0;
        float y = 0;

        // vectorizable
        for (int j = 0; j < clusters[i].size; j++) {
            x += clusters[i].points[j].x;
            y += clusters[i].points[j].y;
        }

        clusters[i].pos.x = x / (float) clusters[i].size;
        clusters[i].pos.y = y / (float) clusters[i].size;
    }
}
```

Fig. 3. Reevaluate_centers code

```c
typedef struct Cluster {
  int size;       // cluster size
  Point pos;      // centroid position
  Point * points; // points that belong to the cluster
} Cluster;

int N;  // number of samples
int K;  // number of clusters
int T;  // number of threads

Point * points;       // N points coordinates
Cluster * clusters;   // Simulates an matrix[T][K] but contiguous in memory
                      // row major ordering -> int offset = i * cols + j;
```

Fig. 4. Data Structure

```c
void cluster_points() {
    #pragma omp parallel for // (same as) #pragma omp parallel for schedule(static, N/T)
    for (register int i = 0; i < N; i++) {
        register int cluster_id = 0;
        register int t = omp_get_thread_num();
        register float lowest_dist = dist(points[i],clusters[t * K].pos); // k = 0

        for (register int k = 1; k < K; k++) {
            register float distance = dist(points[i],clusters[t * K + k].pos);
            if (distance < lowest_dist) {
                lowest_dist = distance;
                cluster_id = k;
            }
        }

        clusters[t * K + cluster_id].points[clusters[t * K + cluster_id].size++] = points[i];
    }
}
```

Fig. 5. Parallel cluster_points

```c
void reevaluate_centers() {
    #pragma omp parallel for
    for (register int k = 0; k < K; k++) {
        register float x = 0, y = 0, s = 0;

        for (register int t = 0; t < T; t++) {
            for (register int j = 0; j < clusters[t * K + k].size; j++) {
                x += clusters[t * K + k].points[j].x;
                y += clusters[t * K + k].points[j].y;
            }
            s += clusters[t * K + k].size;
        }

        for (register int t = 0; t < T; t++) {
            clusters[t * K + k].pos.x = x / s;
            clusters[t * K + k].pos.y = y / s;
        }
    }
}
```

Fig. 6. Parallel reevaluate_centers

```c
void update_clusters_size() {
    for (register int t = 0; t < T; t++) {
        #pragma omp parallel for
        for (register int k = 0; k < K; k++)
            clusters[t * K + k].size = 0;
    }
}
```

Fig. 7. Parallel update_clusters_size

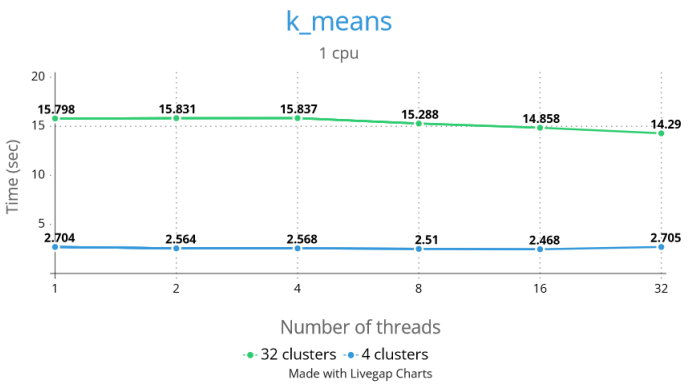

Fig. 8. Graph test with 1 CPU

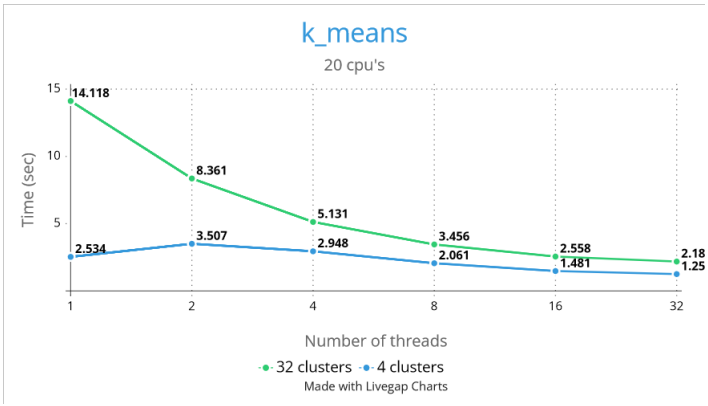

Fig. 9. Graph test with 20 CPU's

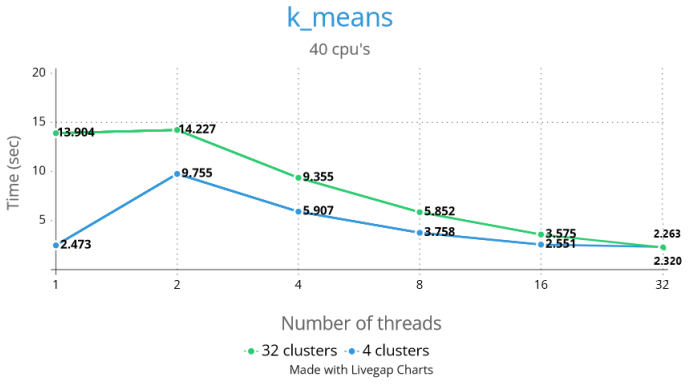

Fig. 10. Graph test with 40 CPU's