High performance computing for mathematical modelling
A study on the Kmeans algorithm

Rebecca Bauer, Varsha Devi, Fanny Lehmann

February 15, 2020

Abstract

The purpose of the assignment is to parallelise some code with either OpenMP or MPI. We parallelise the Kmeans algorithm using OpenMP in two ways. We parallelised the main while loop of the algorithm including a critical section. A reference code is given. The main difference between our methodology and the reference is that we parallelised the main while loop, while they parallelised inner for loops. Runtime analysis shows that our second methodology is a lot faster compared to our first and close to the runtime of the reference. The former is due to the fact that parallelising inner for loops results in constantly creating and destroying threads, which is costly, while parallelising the outer while loop only creates and destroyes threads once. Overall the runtimes are not all too consistent, which makes results less reliable.

Contents

1 OpenMP Framework

OpenMP is an advanced library which provides high level of thread programming in which multiple threads run in parallel to execute the task. Threads are created and deleted dynamically. In OpenMP, parallelism is achieved by generating multiple threads that run in parallel. OpenMP consists of parallel regions, where all the threads work under one master thread (main function). OpenMP is the set of pragmas in C/C++, which are used to parallelize the sequential code for better performance and efficiency. This set consists of some directives for example *#pragma omp for, #pragma omp critical* which give high level front-end interface to the programmer which get translated as calls to threads (or other similar entities), in which the programmer can think parallel. In this project, our goal was to make clustering faster while increasing number of classes and data points.

In this project, we worked with *#pragma omp for* and *#pragma omp critical*, as we have to assign data points to their closest cluster which is the most time consuming task as the number of points increases. In order to iterate over data points and assigne them to the closest centeroid we used #pragma omp for, but for synchronization in convergence check of the algorithm only one cluster can update the iteration. So, we introduced a critical section, which is going to be executed sequentially by each thread.

2 Design Methodology: Parallelization of Kmeans algorithm

2.1 Kmeans algorithm

K-means algorithm is a popular technique aiming at clustering N observations into K groups. Each cluster $k \in \{1, \cdots, K\}$ is described by its mean μ_k and points x_n are iteratively assigned to their closest cluster. The algorithm can be decomposed in two steps after initialisation.

Firstly, each point x_n is assigned to the cluster k^* that satisfies

$$k^* = argmin_{1 \leqslant k \leqslant K} \|x_n - \mu_k\|$$

where $\| \cdot \|$ is the Euclidian distance.

Then, the centroid of each cluster is updated as the mean of the points it contains. If we denote S_k the set of points x assigned to cluster k at the current iteration, then

$$\mu_k = \frac{1}{|S_k|} \sum_{i \in S_k} x_i$$

From this sketch of algorithm, one clearly sees that a for loop is performed on all observations. Therefore, the execution time is proportional to the number of points and it seems clear that the algorithm should greatly benefit from parallelization.

2.2 Code

Since the aim of this project is not to write a complete parallel code from scratch but to study parallelisation, we start from a sequential Kmeans code found in https://github.com/vinayak1998/Parallel-K-Means-Clustering. This folder contains both a sequential and parallel code. We use the sequential one as the basis to parallelize and the parallel one to compare the performances of our code to theirs.

We learnt two ways of parallelising a code during lectures : using OpenMP or MPI. To choose between these two methods, we looked for a performance comparison. [1] shows that the computational time is smaller when using OpenMP than MPI and thus we choose to parallelise the code with OpenMP.

The code contains the main function `kmeans_sequential` (see figure 3 in Appendix) within which there is a while loop that iterates over several subfunctions: `mean_recompute`, `putback` and `assign_clusters`. The while loop is performed until a convergence condition (in this case, the cluster assignment stops varying) is reached or until the maximum number of iterations is attained. Inside the while loop, the function `assign_clusters` executes the first step of the algorithm described in the previous section which is assigning each point to its closest cluster. Then, the function `mean_recompute` does the second step by computing the mean position of all points contained in each cluster.

Thus, the while loop is the best place to introduce parallelism. The task now is to parallelise that section of the code.

2.3 Parallelisation

Methodology 1

In a first attempt we parallelized two subfunctions of the main function. As you can see in Analysis this was not time-efficient. The subfunctions are `mean_recompute` (see figure 4) and `assign_clusters` (see figure 5), both containing a for loop.

In this methodology we put a parallel environment `#pragma omp for` around for loop. We first attempted to put these around every for loop, but soon recognised that the programme takes incredibly long. The learning was that we constantly, with every environment, create and destroy threads - for several loops - in every iteration of the programme. So we took some out and left two parallelised for loops. However, still we were creating and destroying threads. A better version would be to only once create a set of threads and use them in every iteration.

Methodology 2

Since the first attempt took too much time creating and destroying threads we improved the code by parallelising the main while loop in the `kmeans_sequential` function, see figure 6. We first put the parallel environment within the while loop, because (1) the variable `count` is updated in each round and (2) there next round of clusters depends on the former one. So this loop should be sequential. In order to resolve the non-sequential access to the count variable we put a critical region around it, which is working. The results of the whole code were better than in Methodology 1.

As a next step we simply put the parallel environment around the whole code, see figure 7, still leaving the critical region around the update of the `count` variable. The results are a lot better than the previous approach.

Reference Methodology

In the reference code they have parallelised only one for loop, for the function `assign_clusters` (see figure 8) as we did in the beginning (Methodology 1). Their code is slower than our Methodology 2.

3 Analysis

3.1 Dataset

We expect some computational advantages of the parallel code over the sequential one. Especially, we would like the parallel version to be faster than the sequential one when the number of points increases. We can also expect the computational time to decrease when the number of threads increases because each thread then performed less tasks. Finally, we are curious to see the influence of the number of cluster on the computational time.

The reference parallel code is provided with 2 datasets : one that contains 5,000 points clustered in 3 groups, the second includes 50,000 points splitted in 4 clusters. Executing the Kmeans algorithm on the second dataset when looking for 3 clusters can lead to non-interpretable results since we cannot guarantee that the algorithm converges. Thus in the worst case, the while loop would stop after the maximum number of iterations at each time step.

To overcome this limitation, we created our own dataset with 500, 5,000 and 50,000 points clustered in 3, 4 and 6 clusters. In spite of the clusters being clearly separated from each other, we experienced some weird behaviour of our algorithm, both in the sequential and parallel cases. Weird behaviour means that the algorithm did not assign correctly some points to their clusters. We tried different spatial spreading of the clusters in the problematic cases but it had no influence. Thus, we remain careful when generalizing our results.

3.2 Time analysis

We firstly investigate the influence of the number of points on the computational time. Figure 1 shows that our parallel code is better than the sequential one for 50,000 points, excepted with 3 clusters and 6 threads. We notice also that our parallel code is faster than theirs with 3 clusters and 50,000 points but the contrary conclusion can be drawn for the plot with 6 clusters.

We expect the time to increase when the number of clusters increases because one costly operation that depends on the number of clusters is the assignment. The $\arg\min$ function requires all centroids to be examined. However since the number of clusters remains small, this influence may not be so clear.

Indeed, figure 2 shows that the number of clusters has no impact on the computational time with 5,000 points. However with 50,000 points, the time increases with the number of clusters.

One can also notice a strange behaviour of the reference algorithm with 50,000 points and 4 clusters. Indeed, this situation is an example where the algorithm performed badly whatever corresponding dataset we choose. Thus, one should probably not give too much importance to this outlier.

Finally, the dataset with 50,000 points in 6 clusters provides the results we are expecting :

- the sequential code is the slowest
- our parallel code is faster when more threads are used

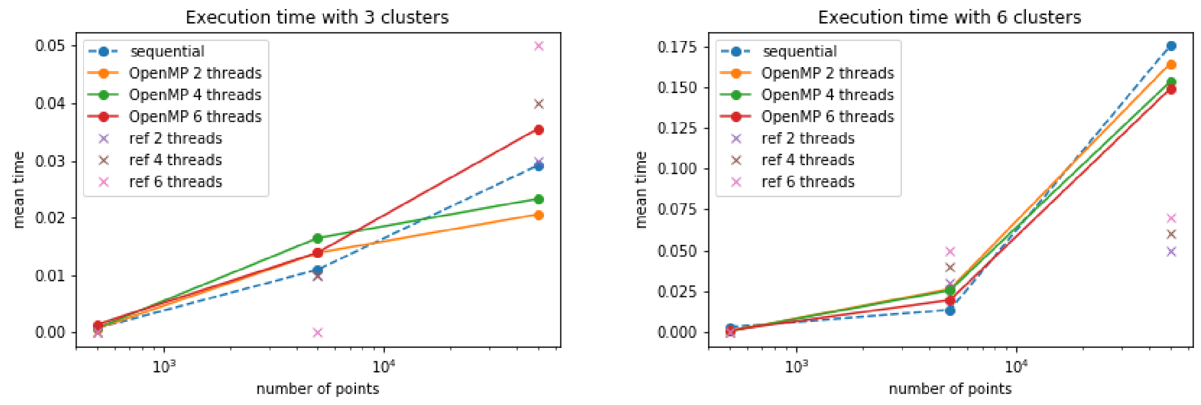

Figure 1: Influence of the number of points on the computational time on points are clustered in 3 groups (left) and 6 groups (right)

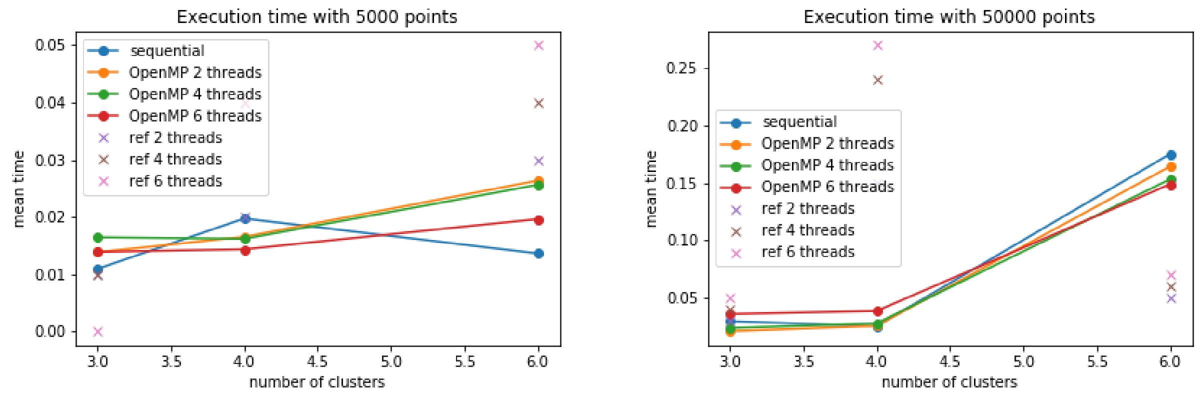

Figure 2: Influence of the number of clusters on the computational time with 5,000 (left) and 50,000 (right) points

We notice also in this situation that their parallel code is faster than ours and benefits too from an increasing number of threads.

We also tested our two parallel methodologies against each other and against the reference on a second machine. The results are given in table 1 for ($N = 5,000; K = 3$) and ($N = 50,000; K = 4$). We observe that our second methodology is a lot faster than our old, for both set of points - as expected, and is almost as fast as the reference.

We compared the number of threads and observe that 4 threads are always faster than both 6 and 8, although 6 is not always faster than 8 in the second methodology for $50,000$ points. This suggests that more threads are not always advantageous and less are more time efficient. The same trend can be observed in the reference analysis.

These observations are confirmed in figure 1 where we have 3 clusters. The trend cannot be observed for 6 clusters, which we haven't tested.

3.3 Comparison to the reference time data

The reference code we use comes with a time analysis on the datasets ($N = 5,000; K = 3$), ($N = 50,000; K = 4$), ($N = 5,000; K = 10$) and ($N = 50,000; K = 10$). Parallel codes is executed with 2, 4 and 8 clusters. Obviously we cannot compare these execution times with the one analysed previously because they were not obtained on the same machines. However, we can compare the trends.

Firstly, they state that their sequential code is always slower than their parallel one. This is not true with our parallel code.

Time Comparison					
Data Points	No. of clusters	No. of threads	Parallel†	Parallel§	Reference Parallel
5000	3	4	0.017	0.003	0.002
5000	3	6	0.039	0.005	0.002
5000	3	8	0.16	0.012	0.010
50000	4	4	0.055	0.035	0.034
50000	4	6	0.070	0.045	0.036
50000	4	8	0.10	0.041	0.039

Table 1: Time comparison between two versions of parallel algorithms(ours) and one reference algorithm. Where † and § represents methodology 1 and methodology 2 respectively.

Moreover, their code is faster with a small number of clusters. Overall, we see the same result with our code on figure 2.

Finally, their code is faster with a larger number of threads excepting in one situation where times are really close.

4 Conclusion

The purpose of the assignment was to parallelise code with OpenMP or MPI. We have parallelised the Kmeans clustering algorithm using OpenMP in two different ways and compared the runtimes against the given reference code. Analysis shows that we do have improvements with our parallel code against the reference sequential and parallel code, however, results are not stable across all experiments. As stated in the analysis we do get some irregularities in execution time and we had the impression that runtime heavily depended on our RAM, so results are not very reliable.

One result is that putting the parallel region outside the while-loop gave a large speed up compared to a parallel region within the while loop, which is mainly due to the fact that we only once create and destroy threads. Thus, a main takeaway is that constantly creating and destroying threads is costly and sometimes sequential code is more efficient.

References

[1] Ajay PadoorChandramohan University at Buffalo, editor. *Parallel Kmeans clustering.*

Appendix

```
    //---------------------------

    int iterations = 1;
    int count;
    do {
        mean_recompute(K, N, points,centr);
        vect.clear();
        putback(centr, K);

        //storing old values for convergence check
        int old[N];
        for (int i=0; i<N; i++){
            old[i] = points[i].cluster;
        }
        assignclusters(points, centr, K, N);
        iterations++;
        count = 0;
        for (int i=0; i<N; i++){
            if (old[i] == points[i].cluster)
                count++;
        }
    } while(count!=N);

    //---------------------------
```

Figure 3: Main while loop in `kmeans_sequential` function.

```
void mean_recompute(int K, int N, Point points[], Point centr[]){
  int count[K];
  Point sum[K];
  #pragma omp parallel num_threads(num_threads)
  {

    #pragma omp for
    for(int i=0; i< N ; i++){
      count[points[i].cluster]++;
      sum[points[i].cluster] = addtwo(points[i],sum[points[i].cluster] );
    }

  }
  for(int i=0; i< K ; i++){
      centr[i].x = sum[i].x/count[i];
      centr[i].y = sum[i].y/count[i];
      centr[i].z = sum[i].z/count[i];
    }
}
```

Figure 4: Parallelised for loop in `mean_recompute`.

```
void assignclusters(Point points[], Point centr[],int K, int N){
  double distances[K];
  //computing distance of a point and assigning all the data points, a centroid/cluster value
  #pragma omp parallel num_threads(num_threads)
  {
    #pragma omp parallel for
    for (int i=0; i<N; i++)
    {
      //assigning the minimum distance cluster, which is an index
      points[i].cluster = checkClosestCluster(points, centr,K, N, i);
      // points[i].cluster = index;
    }
  }
}
```

Figure 5: Parallelised for loop in `assign_clusters`.

```
  do {
    #pragma omp parallel num_threads(num_threads)
      {
      ...
      #pragma omp critical
      {
        for (int i=0; i<N; i++){
          if (old[i] == points[i].cluster)
          count++;
        }
      }
      }
  } while(count!=N);
```

Figure 6: Parallelised while loop (inner) in `kmeans_sequential` function.

```
  #pragma omp parallel num_threads(num_threads)
  {
  do {
      ...
      #pragma omp critical
      {
        for (int i=0; i<N; i++){
          if (old[i] == points[i].cluster)
          count++;
        }
      }
  } while(count!=N);
  }
```

Figure 7: Parallelised while loop (outer) in `kmeans_sequential` function.

```c
void assignclusters(Point points[], Point centr[],int K, int N){
    double distances[K];
    //computing distance of a point and assigning all the data points, a centroid/cluster value
    #pragma omp parallel num_threads(num_threads)
    {
    #pragma omp for
    for (int i=0; i<N; i++)
    {
        //assigning the minimum distance cluster, which is an index
        points[i].cluster = checkClosestCluster(points, centr,K, N, i);
        // points[i].cluster = index;
    }
  }
}
```

Figure 8: Parallelised function `assign_clusters` in reference code.