Parallel version of the K-means algorithm

Diogo Ramos
Physical Engineering
University of Minho, Portugal
Email: a95109@uminho.pt

Gabriel Costa
Physical Engineering
University of Minho, Portugal
Email: a94893@uminho.pt

***Abstract*—This report analyzes the performance of the parallel version of the k-means clustering algorithm, based on Lloyd's algorithm.**

1. Introduction

In this new version of the k-means clustering algorithm, we decided to introduce parallelization of the code using OpenMP, while also making changes to allow the user to introduce the number of points, clusters and threads to use when running the program.

2. Code

2.1. Optimizations

Our first order of work was to analyze the code in order to detect hotspots which would be prime candidates to run in parallel has they would most likely yield the greatest reduction in execution time. Such an analysis led us to profile our code and produce , using a sample size of 10'000'000 points and 4 clusters, the presented dot graph.

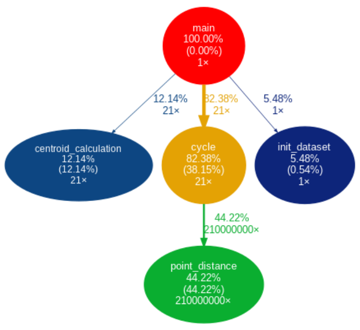

Figure 1: Dot graph created using 10'000'000 points and 4 clusters

From the dot graph it becomes clear that distributing the points is the most time consuming part of the algorithm (function *cycle* does precisely this), with calculating the distance being the most expensive task of the whole program. We can easily parallelize those calculations since they only depend on constant values (in reality only the points are constant but we can consider the cluster to also be since they only change values afterwards).

We can now consider the other part of the function which is responsible of choosing which point belongs to each cluster, this cycle poses a challenge since it is clear that a data race will emerge, reason being that the size of each cluster is determined with a compound assignment, meaning there could be a scenario where multiple threads write to the same memory address. To combat this we could resort to a reduction, but in OpenMP 4.5 there's no implementation for heap allocated arrays, so we must do them by hand. There's also another block whose cost isn't nearly as high as the *cycle* function which can be computed in parallel if we use a similar approach to the reduction problem. This block (function *centroid_calculation*) can be viewed as a compound assignment over a single list.

Doing a reduction by hand implies that each thread should allocate it's own list which will be exclusive to itself and thus not raise any data races and then merge it to the original list through a critical section (this adds a *worse than sequential* section but it's over K elements and not N elements which should be at least a order of magnitude less sequential instructions)

Using this implementation we've achieved, for 4 and 32 clusters, the following execution times.

Thread Counter	4 Clusters (s)	32 Clusters (s)
1	2.868	12.699
2	1.543	5.877
4	0.921	3.239
6	0.701	2.270
8	0.600	1.768
10	0.531	1.464
12	0.645	1.883
14	0.600	1.740
16	0.556	1.573
18	0.521	1.431
20	0.500	1.314

Table 1: Times obtained while running the code to 4 and 32 clusters, varying the number of threads.

2.2. Scalability analysis

Given the times achieved, we decided to do a scalability analysis of the code. For that we built the two following graphics of the speed up in function of the number of threads:

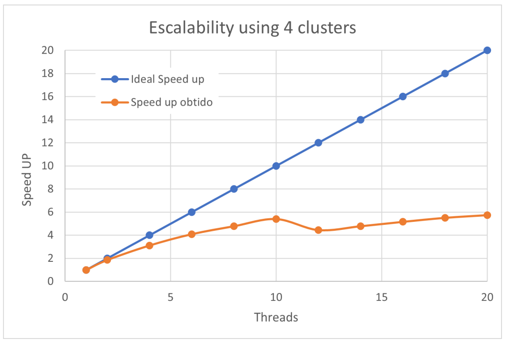

Figure 2: Scalability analysis using 4 clusters

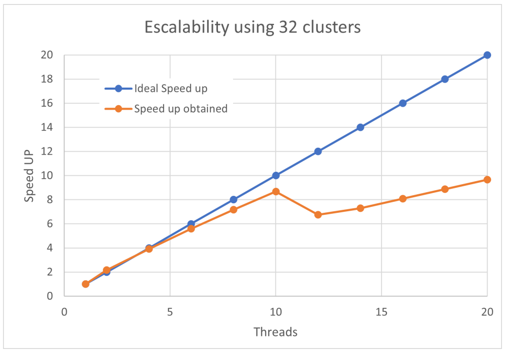

Figure 3: Scalability analysis using 32 clusters

Analysing the graphs reveals a somewhat unexpected behaviour when surpassing a certain amount of threads this might be caused by the increase in thread overhead, profiling the code in two points where there's a performance drop off we can see that there's a big difference in OpenMP related function calls, one of them being the barrier function which is used to synchronize threads at the end of a parallel region, we can also see that when entering a new region the performance hit when creating threads increases drastically.

We should also note that after the big drop off performance starts to increase again with the number of threads.

It also stands out that with a bigger cluster number the program has a much better speed up factor, to analyze this we should also profile the code as we did for 4 clusters - Fig. 1, producing the following dot graph.

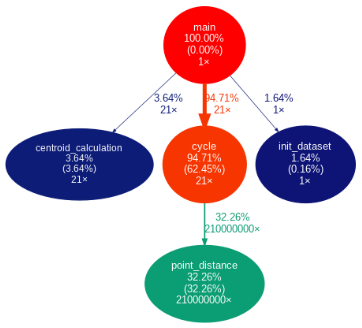

Figure 4: Dot graph created using 10'000'000 points and 32 clusters

From this we see that as the number of clusters increases calculating distances starts to diminish in impact compared to the computation of choosing which points belongs to which cluster, which should be expected since it can't be vectorized as the calculations (for a sufficiently big amount of clusters the calculations should run in a vector unit increasing efficiency), as such increasing the number of threads will lead to a bigger gain in performance since we can do a costlier part of the code in parallel.

It's also pretty clear by this point that to parallelize the code we can't expect the performance to linearly increase with the number of threads since the process itself adds complexity and overhead (creating threads and managing them) and not all parts of code can be executed in parallel since some dependencies require that code must run sequentially.

3. Conclusion

Concluding, we can say that parallelizing the code introduces a lot more work to the programmer than just doing it sequentially. The programmer needs to be very careful in order to identify how to parellelize the code in order to avoid data races. However, this extra works allows the code to run a lot faster than it's sequential version.

In order to achieve this performance we also need to give up on some code readability, making the maintenance of the code harder to someone new, and to the programmer himself.