Parallel implementation of Lloyd's KMeans

Lorenzo Borelli (lorenzo.borelli@studio.unibo.it)

March 30, 2022

1 KMeans

KMeans is a clustering algorithm that groups samples in clusters of similar variance, trying to minimize the inertia of each group. In input, the algorithm requires the number of clusters, together with the samples: each cluster is represented by a centroid, the mean of every point in the cluster; a centroid doesn't have to necessarily be a part of the set of samples in its cluster.

KMeans suffers from a few drawbacks, namely:

- Can't easily group together elongated clusters
- Curse of dimensionality, points in high dimensional spaces do not respond well to euclidean distance used to compute the clusters.

One of the earliest implementations of KMeans is Lloyd's. It is made of 3 steps, mainly:

1. Initialize the centroids with a random point taken from the input samples.
2. Label samples with their cluster, according to the closest centroid in euclidean distance.
3. Update the centroids, computing the average of the points belonging to each cluster.

The second and third step are executed in loop, until either a maximum number of iterations is reached, or the difference between the previous centroids and the current ones is smaller than a fixed tolerance, meaning that further iterations would prove practically useless.

2 Parallel implementation

The main approach for selecting the parallelizable parts of the algorithm, entailed pinpointing the parts of the input regions that could be partitioned. Then, determine whether the tasks involving those partitions were independent, following an embarrassingly parallel paradigm. The most critical dimensions of the data space that were targeted for parallelization were the number of samples, and cluster; as for the number of features of the input, since KMeans should work with a low number of dimensions, no partitioning was applied.
At times, experimentally it was also discovered that increased parallelization would not be useful and too much time would be spent on scheduling jobs rather than actual computation.

The following are the tasks that were parallelized in the implementation:

- Storing the centroids at the beginning of the KMeans loop for the loop exit condition.
- Labeling the samples with their closest centroid, the threads were given partition of data samples.
- Updating the centroids by computing the average of points was parallelized over the clusters. A more fine-grained solution was examined, with a slightly different implementation, in order to partition jobs both in clusters and samples. However, such an implementation would need a critical section to avoid race conditions on the clusters, since then different threads could write to the same centroid; during tests, it was found that this synchronization deteriorated performances.
- Computation of Frobenius distance of the previous and current points, used to compare with the tolerance threshold, was solved with a sum reduction.

2.1 Complexity

A rough analysis on time complexity is needed to reason about weak scaling.
Given a maximum number of iterations *max_iter*, the number of clusters K and number of samples *MAX_POINTS*, as well as the number of features *DIM*, the overall complexity would be $O(K*max\_iter*DIM*MAX\_POINTS)$, which, in this case, converges to $O(K * max\_iter * MAX\_POINTS)$ since two-dimensional points have been used for stability of the algorithm.
Thus, when increasing the amount of work of parallel threads, number of clusters and samples were considered, since the main loop of KMeans is not parallelized.

3 Results

Tests for both strong and weak scaling were performed. Strong scaling entailed simply using from 1 up to the maximum number of available processors; as for weak scaling, for p processors, $\sqrt{p} * K, \sqrt{p} * MAX\_POINTS$ of data were considered, since, for $KMeans(K, MAX\_POINTS)$, complexity of the parallel work is $O(K * MAX\_POINTS)$.
Experiments were done running 5 times the algorithm for each configuration.

Cores	t_1	t_2	t_3	t_4	t_5	Avg
1	13.96	11.55	9.09	8.27	10.89	**10.752**
2	3.6	4.39	7.49	12.04	3.77	**6.258**
3	4.59	2.71	3.58	3.80	3.16	**3.568**
4	3.70	2.09	3.08	3.99	3.50	**3,272**

Table 1: Strong scaling execution times, 200K samples, 16 cluster

Cores	Samples	Clusters	t_1	t_2	t_3	t_4	t_5	Avg
1	100K	10	4.19	2.11	2.69	2.18	2.17	**2.668**
2	141421	17	5.63	4.86	7.18	6.53	5.70	**5.98**
3	173205	21	5.44	2.39	8.61	2.98	3.73	**3.86**
4	200K	24	3.88	4.63	4.92	4.60	6.63	**4.932**

Table 2: Weak scaling execution times

Observing Table 1, it seems the average time of execution decreases the more cores we add, however the speedup also slightly decreases, being limited by Amdahl's law. More formally, the speedup is computed as

$$S(p) = \frac{T_{parallel}(1)}{T_{parallel}(p)}$$

Thus
$S(2) = 1,72$
$S(3) = 3,01$
$S(4) = 3,28$
It should be noted that a real speedup of more than 3 for 3 cores is impossible, although it could be related to the stochastic nature of the algorithm itself, which might converge earlier or later than expected, depending on the random initialization of centroids, rather than a true gain due to parallelism.

As for weak scaling efficiency, it appears that the execution didn't yield great results. This might be due to the fact that some tasks should be further parallelized (e.g. not every piece of code that deals with the samples is parallelized) and the serial fraction α is holding back the efficiency, or the stochasticity of the algorithm.