Parallelization of K-Means Clustering

Erik Amézquita

December 6, 2018

1 Introduction

One of the most popular unsupervised clustering algorithms is the classic K-Means Clustering (KMC), dating back to the late 50s [Ste57]. KMC provides an easy to compute and implement approximate way to cluster a set of given data into K different clusters. KMC starts with K centroids picked at random and later proceeds to update the centroid and cluster position based on the actual data set. Each update depends intrinsically on the previous stage, which makes the tasks in KMC serial. However, we can parallelize each step so the updates can be computed faster. We tried to implement KMC both with MPI and OpenMP

Section 2 presents the mathematical formulation behind clusterization in general and states the serial algorithm for KMC. Section 3.1 presents the pseudocodes used to run KMC using purely MPI and purely OpenMP. We present scaling results in Section 4.

2 Background

Given a collection of data points $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$, and a number of clusters $K \in \mathbb{N}$, we want to find a collection of centroids $\mathcal{C} = \{c_1, \ldots, c_k\}$ such that the cost function

$$\varphi_X(\mathcal{C}) := \frac{1}{n} \sum_{i=1}^{n} \min_{1\leq k\leq K} \|x_i - c_k\|^2. \tag{1}$$

is minimized. With a given collection $\mathcal{C}$ of centroids, we can arrange our initial data into K clusters defined by such centroids as,

$$C_k := \{x_i \in X : \|x_i - c_k\| < \|x_i - c_j\|, k \neq j\}, \quad 1 \leq k \leq K. \tag{2}$$

Unfortunately, the computation of the global minimum for φ_X is a NP-complete problem [GJW82]. The idea of K-Means Clustering (KMC) provides an easy-to-implement, easy-to-compute approximate solution to (1) [DM00]. The classic algorithm for KMC can essentially be described as 5 steps as in Algorithm 1.

1. **Initialization:** Select K data points $\{c_1, \ldots, c_K\} \subset X$ at random and regard them as the initial clusters' centroids. (Lines 2–4).
2. **Distance calculation:** Compute the distance matrix $D \in \mathbb{R}^{n \times K}$ where the ik-th entry is the distance $\|x_i - c_k\|^2$. (Line 6).
3. **Assign new clusters:** For $1 \leq \kappa \leq K$, the κ-th new cluster C'_κ is formed by all those data points $\{x_i\}$ such that c_κ is their closest centroid. (Lines 7–8).
4. **Update centroids:** Compute a new set of centroids $\mathcal{C}'$ defined by the centroids of $\{C'_1, \ldots, C'_K\}$. (Line 9).
5. **Convergence condition:** Observe how different is the set of the new centroids compared to the current set. If the difference is minimal, return the last update. (Line 10).

Algorithm 1 Classic serial KMC

```
 1: Input: $X \subset \mathbb{R}^d, K \in \mathbb{N}, \varepsilon > 0$
 2: for all $k = 0 \to k$ do                          ▷ Pick $k$ initial centroids
 3:     $i \leftarrow$ uniform random $(1, n)$        ▷ get a random number between $1$ and $n$
 4:     $\mathcal{C} \leftarrow \mathcal{C} \cup \{x_i\}$
 5: while $RSS > \varepsilon$ do
 6:     $D_{ik} \leftarrow \|x_i - c_k\|^2$           ▷ Compute a $D \in \mathbb{R}^{n \times K}$ matrix
 7:     $\kappa_i \leftarrow \arg\min_{1 \le k \le K} D_{ik}$   ▷ Compute new indexes $\kappa_i \in \{1, \ldots, K\}$
 8:     $C'_{\kappa_i} \leftarrow C'_{\kappa_i} \cup \{x_i\}$   ▷ Compute new clusters
 9:     $c'_k \leftarrow$ centroid of $C'_k$          ▷ Centroids of the new clusters
10:     $RSS \leftarrow \sum_{i=1}^{k} \|c_k - c'_k\|^2$.
11: return $\varphi_X(\mathcal{C})$                   ▷ Cost as in (1)
```

It is worth to notice that Algorithm 1 is sensible to the initialization step. A good clustering result depends on a good set of initial centroids. Similarly, a good set of initial centroids reduces the number of iterations needed for convergence. The original KMC algorithm assigns centroids randomly among the data set.

Nonetheless, there is a rich area of research behind optimal ways to initialize the centroids, so that the final $\mathcal{C}$ has small cost $\varphi_X(\mathcal{C})$. A popular initialization algorithm is `k-means ++` [AV07], which guarantees with high probability that the initial centroids will be as far away as possible. This in turn guarantees low final costs φ_X and rapid convergence to the final solution.

3 Parallel implementation of KMC

As discussed in [DM00], the second step of the algorithm, the computation of the distance matrix D, is the most expensive step computation-wise. We proceed to implement a data-based parallel implementation, where the data set X is split among different threads or ranks. Each thread or rank computes then a subset of the new clusters C'_k and their respective partial centroid c'_k. Later, the threads or ranks share their partial information regarding clusters and centroids to compute the updated centroid set $\mathcal{C}$. This new centroid set is broadcasted to all threads or ranks to start the next iteration.

On the other hand, the update pipeline as a whole (lines 5–10) is intrinsically serial, which makes difficult a task-based parallel scheme. Any updated centroid collection $\mathcal{C}'$ depends on the previous centroid collection $\mathcal{C}$.

Recall that KMC is sensitive to the choice of initial centroids. A possible task-based parallel scheme could be a MISD setting: every rank or thread solves the same problem with a different initialization step. This would return different solutions, where we could pick the solution corresponding to the least cost 1.

3.1 One particular implementation

For given $n, d, K \in \mathbb{N}$, we first constructed a $n \times d$ array where the data is split in K clusters. To this end, the k-th cluster is constructed as Gaussian distributed vectors with mean $(5k, \ldots, 5k) \in \mathbb{R}^d$ and covariance matrix $\Sigma = \sigma I_d$, with $0 < \sigma_i < 2$ a random number chosen uniformly and I_d the d dimensional identity matrix.

From this construction, we actually know beforehand the collection $\mathcal{C}_{true}$ of true centroids. We later compared the costs $\varphi_X(\mathcal{C}_{true})$ versus $\varphi_X(\mathcal{C})$ to gauge how our parallel implementations of KMC fared.

Since the test data was generated within the same code, the only inputs were the number n of data points to generate, the dimension d on which these points live and the number K of clusters to find. The output was simply the cost $\varphi_X(\mathcal{C})$ as (1) to gauge the correctness of the final algorithm.

We tested our implementations with $2^10 < n < 2^{23}$ points in 4 and 8 dimensions organized in 4 and 8 different clusters. The arrays and subsequent operations were done with single-precision floats. We measured the time the algorithm took for each of the main stages: initialization, distance computation, clustering update. We repeated each clustering 64 times to account for statistical noise.

Also, each clustering was carried out twice for two different initialization methods. First we initialized the centroids randomly as suggested by the classic KMC approach. Second we initialized via `kmc++`[AV07], where we chose a collection of K points as far away as possible.

3.2 MPI

A quick MPI implementation is outlined below in Algorithm 2. Usually, n is much larger than K or d, so that the additional communication cost from lines 10–11 is negligible when compared to the gains of computing the distance matrix D. Since the lines 5–9 are embarrassingly parallel, we expect a linear gain efficiency as we increase the number of ranks. A full working code is found in the repo as `create_kmc.c`.

Algorithm 2 Parallel KMC with MPI (`create_kmc.c`)

```
 1: **Input:** $X \subset \mathbb{R}^d, K \in \mathbb{N}, \varepsilon > 0$
 2: Initialize $\mathcal{C}$ in rank 0.
 3: Broadcast $\mathcal{C}$ to the rest of ranks.
 4: **while** $RSS > \varepsilon$ **do**
 5:     **for** $\text{rank} \cdot \frac{n}{\text{size}} \leq i < (\text{rank}+1) \cdot \frac{n}{\text{size}}$ **do**     ▷ Split data among ranks
 6:         $\kappa_i \leftarrow \arg\min_{1 \leq k \leq K} D_{ik}$     ▷ Compute new partial indexes
 7:         $|C_{\kappa_i}^{\text{rank}}| \leftarrow |C_{\kappa_i}^{\text{rank}}| + 1$     ▷ Compute cardinality of new clusters
 8:         $c_{\kappa_i}^{\text{rank}} \leftarrow c_{\kappa_i}^{\text{rank}} + x_i$     ▷ Compute new partial centroids
 9:     $c_k \leftarrow$ Allreduce($c_k^{\text{rank}}$, MPI_SUM)     ▷ Add all the partial results
10:     $|C_k| \leftarrow$ Allreduce($|C_k^{\text{rank}}|$, MPI_SUM)
11:     $c'_k \leftarrow c_k / |C_k|$     ▷ Centroids of the new clusters
12:     $RSS \leftarrow \sum_{i=1}^{k} \|c_k - c'_k\|^2$.
13: **return** $\varphi_X(\mathcal{C})$     ▷ Cost as in (1)
```

3.3 OpenMP

A quick implementation in OpenMP is outlined below in Algorithm 3. It follows the same idea as in the MPI case above. The important observation is to make the update operations atomic to avoid a data race condition. A full working code is found in the repo as `openmp_kmc.c`.

Algorithm 3 Parallel KMC with OpenMP (`openmp_kmc.c`)

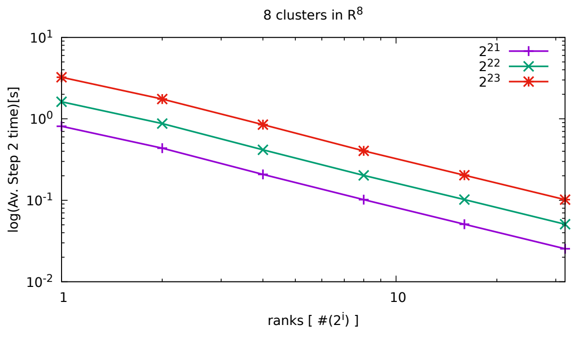
```
1: Input: $X \subset \mathbb{R}^d, K \in \mathbb{N}, \varepsilon > 0$
2: Initialize $\mathcal{C}$
3: while $RSS > \varepsilon$ do
4:     #pragma omp parallel for private ($i$) shared ($X, \mathcal{C}$)
5:     for $0 \leq i < n$ do
6:         $\kappa_i \leftarrow \arg\min_{1 \leq k \leq K} D_{ik}$        ▷ Compute new partial indexes
7:         #pragma omp atomic                                ▷ Avoid data races
8:         $|C_{\kappa_i}| \leftarrow |C_{\kappa_i}| + 1$        ▷ Compute cardinality of new clusters
9:         #pragma omp atomic                                ▷ Avoid data races
10:        $c_{\kappa_i} \leftarrow c_{\kappa_i} + x_i$        ▷ Compute new partial centroids
11:    $c'_k \leftarrow c_k / |C_k|$        ▷ Centroids of the new clusters
12:    $RSS \leftarrow \sum_{i=1}^{k} \|c_k - c'_k\|^2$.
13: return $\varphi_X(\mathcal{C})$        ▷ Cost as in (1)
```

4 Results

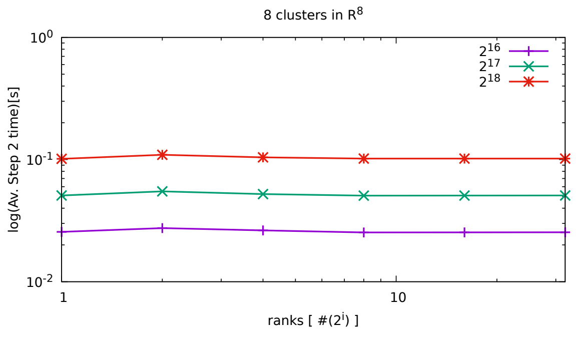
MPI yielded promising results. OpenMP not so much. The code was reviewed but we couldn't figure out a compromising mistake.

4.1 MPI

As reflected in Figure 1, the second and third steps of the classic MPI algorithm are embarrassingly parallel. We see that the average running time for these stages follows closely the theoretical expectation. We see both strong and weak scaling. As discussed above, the cost communication stage is relatively low compared to the computational cost of the distance matrix D. We see that even when we measure the average running time for the five main stages (we excluded the time needed to generate the clusters), the scalability behavior remains. Refer to Figure 2. Similar results were observed when we worked with 4 different clusters in $\mathbb{R}^4$.

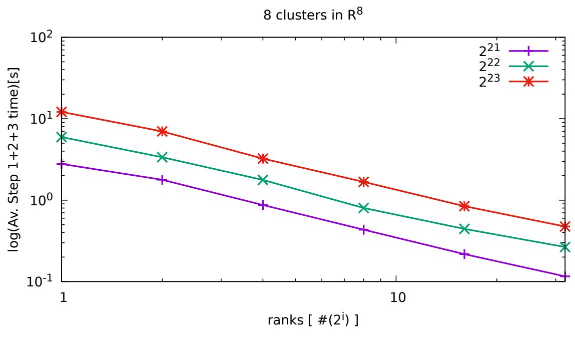


(a) Strong scaling: Fixed the size of X.



(b) Weak scaling: Fixed the chunk of X per rank

Figure 1: Average running time of lines 5–8 in Algorithm 2



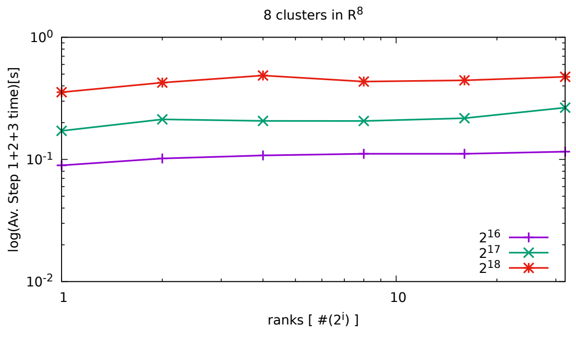


Figure 2: Average running time for whole Algorithm 2

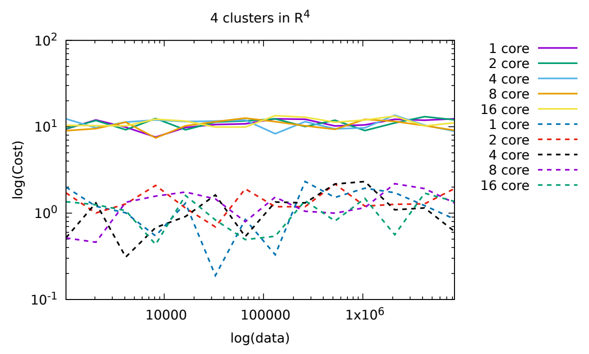
We also observe that whenever we initialize the centroids randomly, we always obtain the same final cost (1), regardless of the number of cores. This shows that Algorithm 2 behaves well under scalability.

It is also worth noticing that if we initialize the centroids via `k-means++`, the final clustering has practically the same cost as the true clustering $\mathcal{C}_{true}$. This accuracy is also true for different number of ranks. Refer to Figure 3.

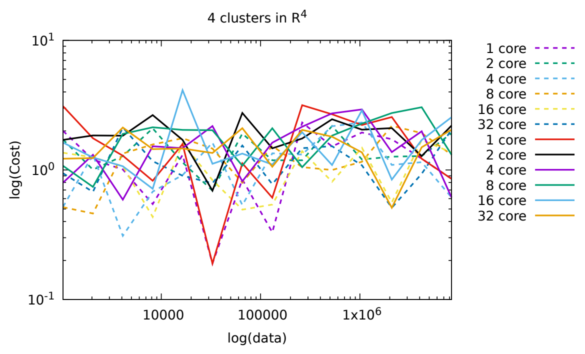
4.2 OpenMP

Unfortunately when running Algorithm 3 we didn't obtain results similar to the ones from MPI. We observe that the results tend to show scalable behaviour as the number of threads increases.

One possible hypothesis can be the fact that the while cycle in Algorithm 3 forces the

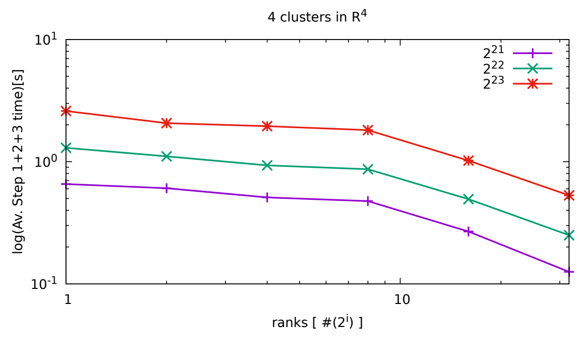


(a) Random initialization

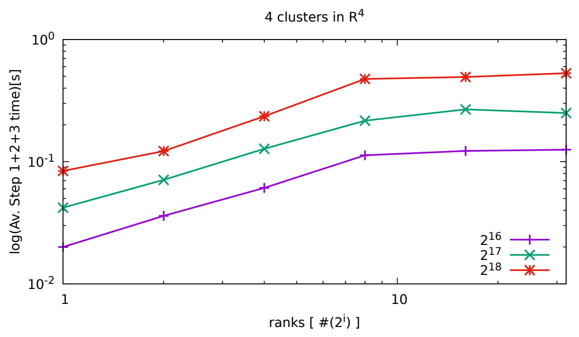


(b) Random initialization

Figure 3: Dotted lines refer to the cost from $\mathcal{C}_{true}$ while the solid lines refer to the obtained clustering.



(a) Fixed size of data set X



(b) Fixed chunk size for each thread

Figure 4: Average time for running Algorithm 3.

closure and creation of several parallel regions throughout the algorithm's run. This can mean that the overhead associated to the creation of these regions is only justified when the computation of stages 2 and 3 is done quickly enough.

The lack of positive results in OpenMP discouraged us to pursue further a hybrid strategy MPI+OpenMP.

5 Conclusions

KMC layout favors a data-based parallel scheme to compute the distance D matrix. However, the final answer depends on a series of updates, which must be carried out sequentially. This makes difficult a task-based scheme. On a more machine-learning setting, we can take advantage of the task-based scheme from a SIMD architecture. Since the initialization of the algorithm is done randomly, different ranks or threads could carry out different initial values. This would yield different clusterings for the same data, from which we can choose the one that minimizes the cost (1) as the correct clustering. Speaking of random initializations, the stochastic nature of the initialization step makes difficult to draw precise conclusions about the overall performance of the KMC without a large number of experiment repetitions.

MPI shows a considerable improvement over the sequential algorithm. Comparing figures 1 vs 2 we observe that the cheap cost of step 2 will offset the communication costs incurred during step 3. On the other hand, the intrinsic `while loop` in Algorithm 3 (Line 3) forces the creation and destruction of several parallel regions. This in turn makes the solution of KMC with OpenMP a bad option. This also discouraged the pursue of any hybrid strategy between MPI and OpenMP.

References

[AV07] David Arthur and Sergei Vassilvitskii. "K-means++: the advantages of careful seeding". In: *In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms.* 2007.

[DM00] Inderjit S. Dhillon and Dharmendra S. Modha. "A Data-Clustering Algorithm on Distributed Memory Multiprocessors". In: *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD.* Berlin, Heidelberg: Springer-Verlag, 2000, pp. 245–260. ISBN: 3-540-67194-3. URL: `http://dl.acm.org/citation.cfm?id=648035.744385`.

[GJW82] M. Garey, D. Johnson, and H. Witsenhausen. "The complexity of the generalized Lloyd - Max problem (Corresp.)" In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982), pp. 255–256. ISSN: 0018-9448. DOI: `10.1109/TIT.1982.1056488`.

[Ste57] Hugo Steinhaus. "Sur la division des corps matériels en parties." French. In: *Bull. Acad. Pol. Sci., Cl. III* 4 (1957), pp. 801–804. ISSN: 0001-4095.