PARALLEL MINI-BATCH K-MEANS

Ca' Foscari University of Venice
Master's Degree in Computer Science & Information Technology [LM-18]

Palmisano Tommaso, 886825

I. Abstract

K-Means is perhaps the simplest clustering algorithm to implement, making it suitable for experiments and performance evaluation. In this paper, we analyze a C++ implementation of **Mini-Batch K-Means** along with a parallelized version to demonstrate how algorithms can benefit from **multi-threading**. Additionally, we employ **vectorization** to further enhance performance. Note that this code is optimized for **ARM** CPUs, and most of the choices made were driven by that.

II. Implementation and Optimization

2.1. Evaluation Metrics

As we already stated, we will examine the *Mini-Batch* variation of K-Means, which, of course, makes use of mini-batches at every iteration of the algorithm to improve performance when working with huge datasets. The metrics we will use to evaluate the quality of our algorithm are the **Error** (a.k.a. inertia) and the **Normalized Mutual Information**, defined as:

$$E = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - c_i||^2 \qquad NMI = \frac{MI(C, L)}{\frac{1}{2}(H(C) + H(L))}$$

where C are the cluster assignments and L are the true labels. The first is an unsupervised measure, used to track improvements in clustering between successive steps of the algorithm and to decide when to stop. The second is a supervised measure, used to evaluate the final clustering quality against the ground-truth labels.

2.2. Data Structures

Regarding the data structures used to implement mini-batch K-means, we stored the images and the centroids as `std::vectors<>` of `std::vectors<float>`, those in practice are $n \times m$ matrices where n is the numbers of images or centroids, and m the dimension of the vector space, while the true labels are represented as a `std::vectors<int>`. Our first optimization leans here: we make the choice of representing the cluster as a simple `std::vector<size_t>` where each index corresponds to the index of an image, and the value to the index of the assigned centroid.

2.3. Optimizations

Before explaining how we employed parallelization, let's first discuss other optimization techniques we introduced. K-Means, with its variants, is notoriously initialization dependent: a completely random choice of the initial centroids may lead to suboptimal results and unexpected behavior. To avoid this, we implemented K-Means++, a well-known heuristic which chooses every centroid with probability proportional to the squared distance from the centroids already chosen. A first kind of parallelization was made using vectorization. Beside the initialization, K-Means mainly consists of two steps: the *Assignment Step* and the *Update Step*, which makes use of two functions: one to calculate the *Euclidean Distance*, and another to calculate the mean point of a cluster, thus the new centroid. Both functions were optimized using SIMD CPU instruction, which allows to execute instructions in parallel on multiple data, and achieve respectively a $\sim$10x speedup for the euclidean distance and a $\sim$3x speedup for the centroid calculation. Finally, a minor tweak: if we implement the formula for the Error as it is, we end up accessing the data points in centroid order. That's a mistake, because doing so breaks spatial locality and introduces a huge unnecessary overhead. Accessing the memory in points order instead solves the issue.

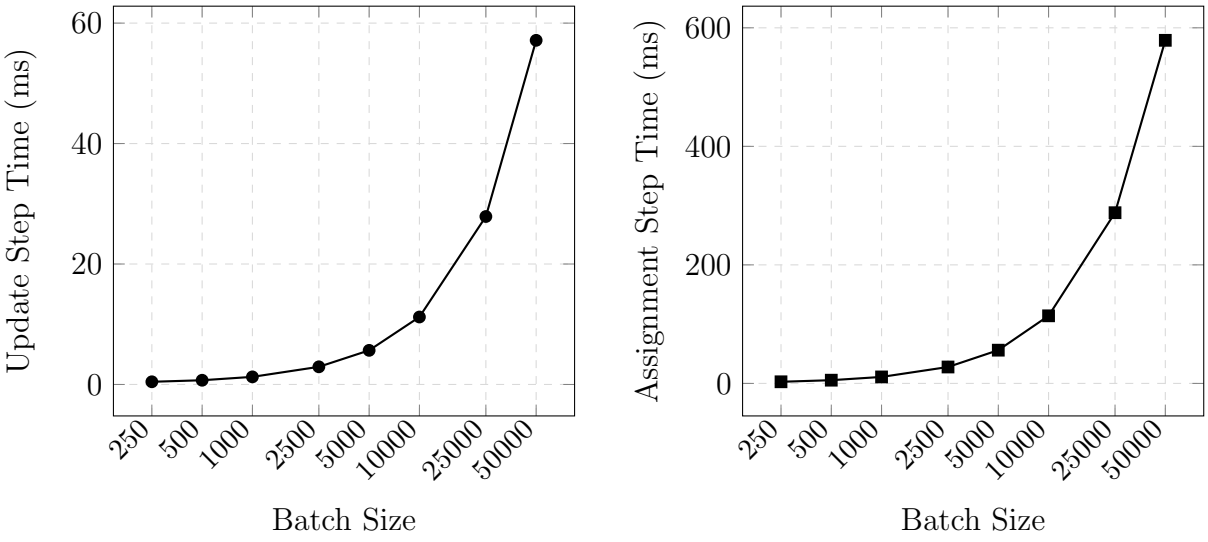
2.4. Parallelization

We now discuss how we employed parallelization to further enhance performances. The choice was to use **OpenMP**, a high-level API which makes use of compiler directives to provide a high-level abstraction layer over the low-level multithreading primitives. The first function making use of multiple threads was `kmeans_pp()`: since every centroid we assign depends from the ones we already assigned, the other loop couldn't be parallelized, but only the inner one. `#pragma omp parallel for schedule(dynamic)` is the directive used to achieve the desired result. When it comes to the main functions of the algorithm, it makes sense to parallelize the only loop of `updateStep()` with `schedule(dynamic)` because clusters might be unbalanced, with some having more points than others, and `dynamic` helps balance the load. Less intuitive is the use of `schedule(dynamic)` for the functions `assignmentStep()` and `assignWholeDataset()`. At first glance it seems that the load of every iterations of the loops is the same, and that's in fact the case. Interestingly, experimental results indicate that the use of `dynamic` yields a noticeable improvement in execution time. That's likely due to other memory accesses overhead getting mitigated by the dynamic scheduling. Lastly, we parallelized the functions used to calculate our evaluation metrics: `clusteringError()` was easily parallelized since it has only one loop, while in `normalizedMutualInformation()` OpenMP was used on the outer loop. Note that in both cases `reduction(+:...)` was necessary because threads updates a shared variable.

III. Analysis

We now move on to the analysis of our algorithm: we evaluate the algorithm on the well-known MNIST dataset, which contains 70,000 grayscale images of handwritten digits. All tests were performed on an M4 Mac mini with 16 GB of RAM and over 10000 iterations unless otherwise stated.

Firstly, we break down how the batch size influences the different steps of the sequential algorithm, for simplicity we assumed the number of clusters k to be equal to 10. As expected, the initialization and the calculation of the evaluation metrics phases are not effected, while the *assignment* and *update* steps are influenced by the batch size. The graphs below shows this behavior:



(a) Update step time vs batch size

(b) Assignment step time vs batch size

Figure 1: Effect of batch size on Update and Assignment step runtimes.

Next, we are interested in how the value of k influences the clustering quality, thus the NMI. We assume a batch size of 10,000 and we stop the algorithm either after 10,000 iterations or when the Error falls below a threshold. Since as we said the initialization can have an huge effect n the final results we reported the best result of a couple of runs: the heat map below portrays them:

	3	5	10	15	20	30	40	50
NMI	0.269	0.402	0.507	0.521	0.543	0.537	0.545	0.547

The results are clear: increasing the number of clusters generally makes it easier to achieve higher clustering quality. However, obtaining a high NMI with a large number of clusters is not always desirable, since the final result may not accurately represent the underlying structure of the data. That being said, the actual best value of k was 50, so we pick that for the next experiment: we want to discover how the batch size influences the clustering quality. We run the experiments using the previusly used batch sizes:

	250	500	1000	2500	5000	10000	25000	50000
NMI	0.534	0.537	0.536	0.546	0.539	0.543	0.556	0.545

It is easy to see that the batch size have little to no impact on the clustering quality. The tradeoff is clear: a larger batch leads on average to a faster convergence time, while being more computationally intensive. So, if we take into account the fluctuations caused by the initialization, the slightly better results obtained with larger batch sizes, do not justify the considerably longer time they require. The sweet spot with this particular dataset seems to 1,000 samples.

Moving on, we take into account the number of threads. We choose the values of k and batch size found in the previous experiments, thus 50 and 1,000 respectively. We test out algorithm over 1,000 iterations, varying the number of threads used and measuring the total amount of time taken for the algorithm to complete. The graph below portraits the results:

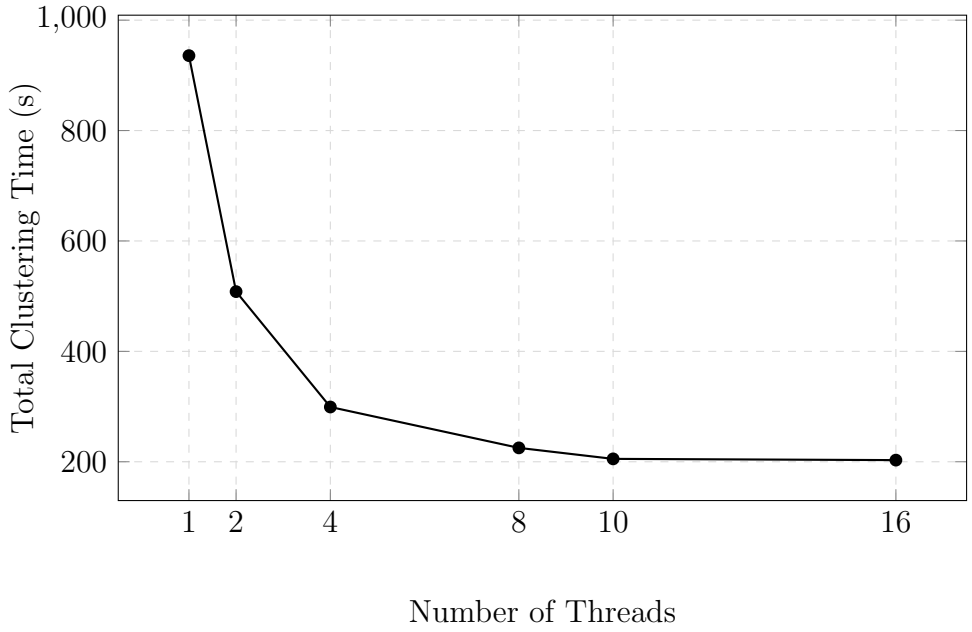


Figure 2: Effect of thread count on total clustering runtime.

We point out that initially, as we double the number of threads, the clustering time halves, but as we approach the number of physical cores, the effects of multi threading get mitigated by other factors, like overheads caused by threads and memory management. However, the biggest limitation, which prevents us of reaching a linear speedup, is perhaps the fact that the machine's CPU uses the big.LITTLE architecture, with 4 core faster than the other 6. In fact, we can clearly see how past 4 threads, the speedup achieved drops substantially, and using more than 10 thread have no effect.

As a last experiment, we are interested in discovering whats the largest dataset we can cluster in a fixed amount of time, let's say 5 minutes (300,000 ms). In order to being able to collect the data we need, we are required to augment our dataset. Our approach is to add some noise to the existing images by flipping the values of any of the pixels which compose on e of them with 10% probability. Finally, because the number of sample is considerable, we fix the batch size to 5,000 and we will use k equal to 10, since it is more representative and faster to compute. It turn out that with a bit of luck the largest dataset we can cluster with our machine in 5 minutes consists of 300,000 samples.