Learning Over Massive Data

Assignement 1 – KMeans

Davide Vitagliano 904120

Academic Year 2024/2025

1. Compilation

In addition to the code files, in the zip, there is the *CMakeList.txt* used for compilation. The data is the provided one, thus is not included.

Follows the steps to compile and execute the code, starting inside the project directory:

```
mkdir build && cd build && cmake .. && make
```

```
[slurm command] ./parallel_kmeans <n_threads>
```

Or `perf stat -B -e cache-references,cache-misses ./parallel_kmeans <n_threads>`

2. Algorithm

The *k_means* class is built with the methods used to solve the clustering task.

The class represents the clusters and the respective centroids, along with the true label clusters.

The `constructor` takes a dataset, true cluster labels, the number of threads, the number of clusters (k), batch size, and the maximum number of iterations as parameters. It first allocates memory for *centroids* and *clusters*, then organizes data points into *labelClusters* based on their true labels. It ensures efficient memory use by shrinking the cluster vectors. Finally, it initializes each cluster centroid using the first data point from its corresponding true cluster in the *dataset*.

The `findCentroidIdx` helper method determines the index of the closest centroid for a given data point using the squared Euclidean distance. It initializes the minimum distance using the first centroid and iterates through the remaining centroids, updating the closest centroid index whenever a smaller distance is found.

Another relevant helper method is `scanAssign`, which assigns each data point in a batch to the nearest centroid and organizes them into clusters. It first clears previous cluster assignments and then uses OpenMP for parallel computation. Each thread maintains its own local cluster assignments to avoid contention. After computing assignments in parallel, the local clusters are merged into the main cluster structure. Finally, memory is optimized using `shrink_to_fit`, and the cluster indices are sorted to maintain order.

Follows the pseudocode of the scanAssign method:

```
function scanAssign(batch)
    localClusters <- array of size n_threads and value <- array of size k
    parallel section
        tid <- get thread number
        parallel for i from 0 to size of batch:
            localclusters[tid][findCentroidIdx(batch[i])].push_back(i)
    for t from 0 to n_threads:
        for i from 0 to k:
            clusters[i].insert(clusters[i].end, localClusters[t][i].begin, localClusters[t][i].end)
```

The main method of the class is the fit method: it trains the KMeans model using an iterative approach with batch updates. It initializes variables for tracking centroid changes and sets up OpenMP locks for thread-safe updates. The algorithm iterates until the centroid shift (delta) falls below a tolerance threshold or reaches the maximum number of iterations. Each iteration samples a batch of data points, assigns them to the nearest centroids in parallel, and updates the centroids using an adaptive learning rate that decreases as more points are assigned. Once convergence is reached, the function finalizes assignments, prints cluster sizes, and returns metrics like inertia and normalized mutual information (NMI) to evaluate clustering performance.

Follows the pseudocode of the fit method:

```
function fit(dataset, tol)
    counts <- array of size k
    delta <- tol + 1.0
    prevChange <- 0.0
    prevCentroids <- centroids
    locks <- initialize k OpenMP locks
    for i from 0 to maxIter && delta > tol:
        batch <- sampleData(dataset)
        indices <- array of size batchSize
        parallel for j from 0 to batchSize:
            indices[j] <- findCentroidIdx(batch[j])
        parallel for j from 0 to batchSize:
            idx <- indices[j]
            atomic capture of curCount <- counts[idx] + 1
            lr <- 1.0 / curCount
            lock locks[idx]
            for l from 0 to 784:
                centroids[idx][l] <- (1-lr) * centroids[idx][l] + lr * batch[j][l]
            unlock locks[idx]
        delta <- deltaChange(prevChange, prevCentroids)
    destroy locks
    scanAssign(dataset)
    return inertiaError(dataset), nmiError(size of dataset)
```

The sampleData method randomly selects *batchSize* data points from the dataset, and the two error methods compute the measures with the standard formula.

3. Testing

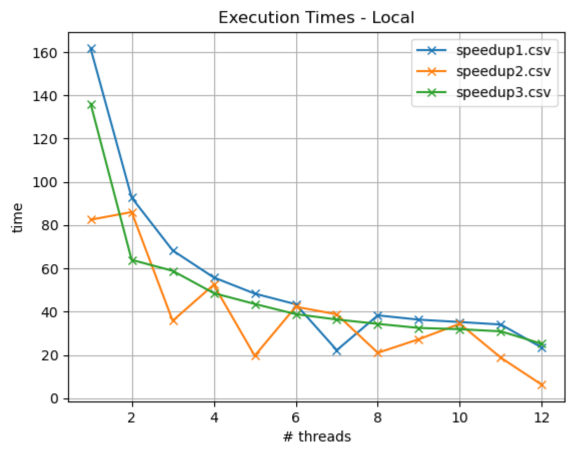
All tests are performed on a local machine with a Ryzen 5 5600x (6 cores and 12 threads) and 3733 MHz 32 GB of RAM, and on the university cluster with 20 threads.

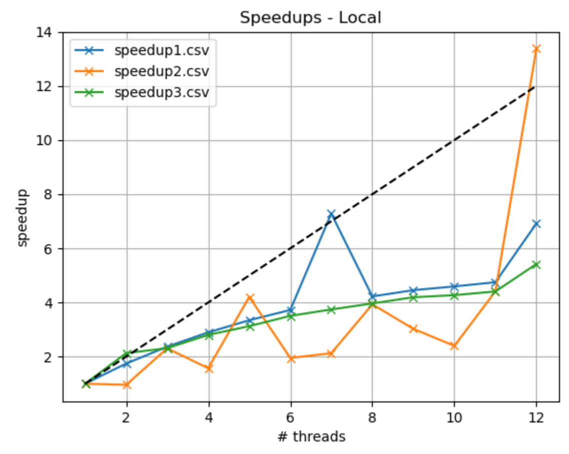
The correctness of the algorithm is verified by comparing the results with the sequential version of the algorithm, commenting out a parallel section that ignore the seeding of the random sampling.

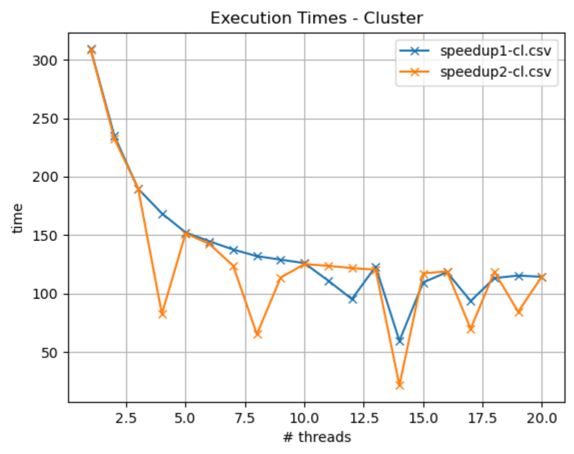
The performance and scalability of the algorithm is evaluated by measuring the speedup of the parallel version with respect to the sequential one. The execution time is measured with the omp_get_wtime() method, and the cache miss rate is measured with the perf tool on the local machine.

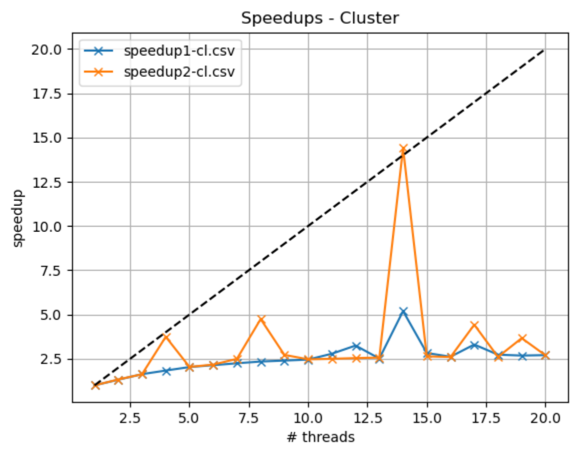
Cache miss rate: ~13.11%

Run	k	Batch size
1	9	67500
2	8	67500
3	8	62500
1 – cluster	8	62500
2 - cluster	8	62500









Each line represents a run of the algorithm, as expected the execution times are similar and the variation is due to the randomness of the batch selection. An interesting observation is that when the batch size is smaller, the stability of the speedup is smaller.

Overall, the speedup is not impressive since the sequential algorithm is already quite efficient, and the loops don't operate on many iterations, in fact the parallelization is done on loops iterating the batch size, instead of k or the dimension of the images.

The biggest improvement is in the finding of the nearest centroid for each image in the batch: it halves the computation time, while we have an improvement of about 10 seconds in the update of the centroids. Finally, the least improvement is on the assignment of each data point to the corresponding cluster, in which we have a gain of just few seconds.