HIGH PERFORMANCE COMPUTING FOR ENGINEERING APPLICATIONS

PROJECT REPORT

Parallel Implementation of K-means Clustering using Open MPI

Roll Number: BS17B024 **Name: Ranjith Kumar R M**

Abstract

The main idea of this project was to get acquainted with the basic concepts of parallel programming and apply parallelism in k-means clustering algorithms, and thereby estimate the degree of boost of performance under different constraints (ie., various combinations of number of clusters and number of processes in the parallel computing architecture). The parallelization was realized in the Python (Version 3.7.9) program language and implemented using the open MPI Message Passing Interface library project, “mpi4py” (a standardized and portable message-passing system designed to function on a wide variety of parallel computers). The algorithms for parallel and serial versions of k-means clustering algorithms were composed in the Python programming language and the validation of outputs from the codes were assessed using Adjusted rand index (ARI, a measure of the similarity between two data clusterings).

k-means clustering algorithm

k-means clustering is a method of vector quantization, originally from signal processing, that aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean (cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. *k*-means clustering minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances, which would be the more difficult Weber problem: the mean optimizes squared errors, whereas only the geometric median minimizes Euclidean distances.
The problem is computationally difficult (NP-hard); however, efficient heuristic algorithms converge quickly to a local optimum. These are usually similar to the expectation-maximization algorithm for mixtures of Gaussian distributions via an iterative refinement approach employed by both *k-means* and *Gaussian mixture modeling*.

The algorithm

The most common algorithm uses an iterative refinement technique. Due to its ubiquity, it is often called "the *k*-means algorithm"; it is also referred to as Lloyd's algorithm, particularly in the computer science community. It is sometimes also referred to as "naïve *k*-means", because there exist much faster alternatives. Given a set of observations (x_1, x_2, ..., x_n), where each observation is a *d*-dimensional real vector, *k*-means clustering aims to partition the *n* observations into *k* (≤ *n*) sets S = {S_1, S_2, ..., S_k} so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance).

Formally, the objective is to find:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i$$

where μ_i is the mean of points in S_i.
This is equivalent to minimizing the pairwise squared deviations of points in the same cluster:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{\mathbf{x},\mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2$$

The equivalence can be deduced from identity $\sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \sum_{\mathbf{x} \neq \mathbf{y} \in S_i} (\mathbf{x} - \boldsymbol{\mu}_i)^T (\boldsymbol{\mu}_i - \mathbf{y})$. Because the total variance is constant, this is equivalent to maximizing the sum of squared deviations between points in *different* clusters (between-cluster sum of squares, BCSS),which follows from the law of total variance.
The "assignment" step is referred to as the "expectation step", while the "update step" is a maximization step, making this algorithm a variant of the *generalized* expectation-maximization algorithm.

Given an initial set of *k* means $m_1^{(1)}$,...,$m_k^{(1)}$ (see below), the algorithm proceeds by alternating between two steps:

Assignment step: Assign each observation to the cluster with the nearest mean: that with the least squared Euclidean distance. (Mathematically, this means partitioning the observations according to the Voronoi diagram generated by the means.)

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \le \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \le j \le k \right\},$$

where each x_p is assigned to exactly one $S^{(t)}$, even if it could be assigned to two or more of them.

Update step: Recalculate means (centroids) for observations assigned to each cluster.

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

The algorithm has converged when the assignments no longer change. The algorithm is not guaranteed to find the optimum.

Parallelism with OPENMPI

As was mentioned above, in order to achieve parallelism, "mpi4py" package which provides MPI bindings for the Python language, allowing programmers to exploit multiple processor computing systems was deployed.

Classifications of parallel programming models can be divided broadly into two areas:

Process interaction

Process interaction relates to the mechanisms by which parallel processes are able to communicate with each other. Message-passing model, one of the most common forms of process interaction was exploited in this project wherein parallel processes exchange data through passing messages to one another. These communications can be asynchronous, where a message can be sent before the receiver is ready, or synchronous, where the receiver must be ready.

Message-passing communication: Master and workers

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/-computer instances). Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec. MPI library functions include, but are not limited to, point-to-point rendezvous-type send/receive operations, choosing between a Cartesian or graph-like logical process topology, exchanging data between process pairs (send/receive operations), combining partial results of computations (gather and reduce operations), synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session, current processor identity that a process is mapped to, neighboring processes accessible in a logical topology, and so on. Collective communication which was used majorly in our implementation involves communication of data using all processes inside of a given communicator, the default communicator that contains all available processes is called **MPI_COMM_WORLD**. When a collective call is made it must be called by all processes inside of the communicator. Collective communications will not interfere with point-to-point communications nor will point-to-point communications interfere with collective communication. Collective communications also do not need the use of tags. Send and receive buffers when using collective communication calls must match in order for the call to work and there is no guarantee that a function will be synchronizing (except for "barrier"). Also all collective communication operations are blocking.

Problem decomposition

A parallel program is composed of simultaneously executing processes. Problem decomposition relates to the way in which the constituent processes are formulated. In this work, the essence of both the following problem decomposition models are present.

Task parallelism

A task-parallel model focuses on processes, or threads of execution. These processes will often be behaviourally distinct, which emphasises the need for communication. Task parallelism is a natural way to express message-passing communication.

Data parallelism

A data-parallel model focuses on performing operations on a data set, typically a regularly structured array. A set of tasks will operate on this data, but independently on disjoint partitions. Computations are synchronously performed. Amount of parallelization is proportional to the input size. As there is only one execution thread operating on all sets of data, the speedup is more. It is designed for optimum load balance on a multiprocessor system.

Adjusted Rand index

The adjusted Rand index is the corrected-for-chance version of the Rand index. Such a correction for chance establishes a baseline by using the expected similarity of all pairwise comparisons between clusterings specified by a random model. Given a set S of n elements, and two partitions of S to compare, $X = \{X_1, X_2,..., X_r\}$, a partition of *S* into *r* subsets, and $Y = \{Y_1, Y_2,..., Y_s\}$, a partition of *S* into *s* subsets, the overlap between X and Y can be summarized in a contingency table [n_{ij}] where each entry n_{ij} denotes the number of objects in common between X_i and Y_j : $n_{ij} = |X_i \cap Y_j|$.

$X \backslash Y$	Y_1	Y_2	$\cdots$	Y_s	sums
X_1	n_{11}	n_{12}	$\cdots$	n_{1s}	a_1
X_2	n_{21}	n_{22}	$\cdots$	n_{2s}	a_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
X_r	n_{r1}	n_{r2}	$\cdots$	n_{rs}	a_r
sums	b_1	b_2	$\cdots$	b_s	

$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2} - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \Big/ \binom{n}{2}}{\frac{1}{2}\left[\sum_i \binom{a_i}{2} + \sum_j \binom{b_j}{2}\right] - \left[\sum_i \binom{a_i}{2} \sum_j \binom{b_j}{2}\right] \Big/ \binom{n}{2}}$$

where n_{ij}, a_i, b_j are values from the contingency table.

Results and Data Analysis:

In this section, I have presented result of validation (to justify that written program code works correctly), result of clustering (to justify that obtained results of clustering correspond to the problem posed in the theory of the algorithm) and performance evaluation of parallel and serial k-means on datasets of different sizes.

3D Road Network (North Jutland, Denmark) Data Set was used for experimental purposes. This dataset was constructed by adding elevation information to a 2D road network in North Jutland, Denmark (covering a region of 185 x 135 km2). It contains 434874 samples and 4 features.

Feature information:

1. OSM ID: OpenStreetMap ID for each road segment or edge in the graph.
2. LONGITUDE: Web Mercator (Google format) longitude
3. LATITUDE: Web Mercator (Google format) latitude
4. ALTITUDE: Height in meters.

Validation

To confirm that written parallel and serial program codes work correctly (in the sense of clustering) it was decided to calculate Adjusted Rand index (ARI) that allows us to understand the level of similarity between the clustering vectors yielded from written codes and k-means function from "sklearn.cluster" python library. ARI calculated between clustering vectors yielded from written k-means (for both parallel and serial) and sklearn k-means for k = 2, 3, 4 were very close to 1 indicating a profound amount of similarity.

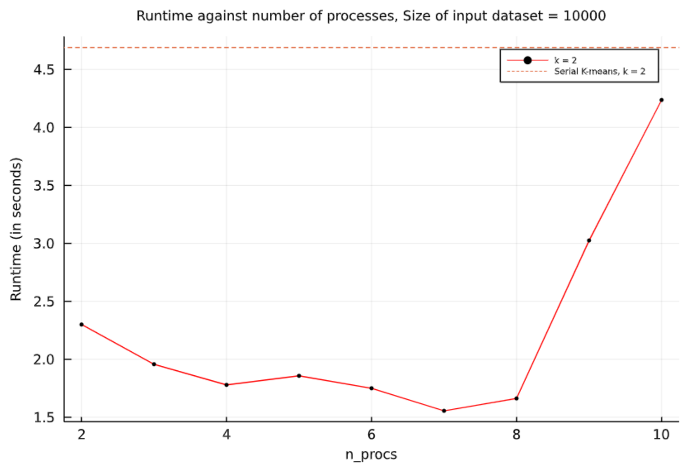
Parallel implementation

Algorithm 1 Parallel k-means clustering algorithm

```
1:  MASTER (rank = 0)
2:  Inputs:
        np = input(" number of processes ")
        k = input(" number of clusters ")
        path = input(" path to file with data set")
3:  Initialize:
        df ← read_csv(path, sep(","))
    4:  for $i = 1$ to k do
    5:      Initial[i] ← append random sample from df (data set)
    6:  end for
    7:  df = list[$df_1 + df_2 + ... + df_{np}$] #divide data set into np parts, where np is number of processes.
8:  ALL RANKS
9:  df=comm.scatter(list(df),root=0) #scattering parts of dataset on processes
10: k=comm.bcast(k, root=0) #broadcast number of clusters from rank=0 to other ranks
11: Initial=comm.bcast( initial, root = 0) # broadcast centroid matrix from rank=0 to other ranks
12: flag = True
13: while (flag == true) do
14:     for $j = 0$ to k do
15:         for $i = 0$ to len(df) do
16:             dist[i][j]= euclidean distance (initial[j],data[i])
17:         end for
18:     end for
19:     for $i = 0$ to len(dist) do
20:         clusters.append(np.argmin(dist[i]) # identify the cluster label of each sample for each process
21:     end for
22:     Q_clusts= collections.Counter (clusters) #identify what is the labels was faced in each process and their frequency
23:     totcounter = comm.allreduce(Q_clusts) #gather Q_clusts from each process and broadcast the result back
24:     cluster=comm.gather(clusters,root=0)#gather clusters from each processes and send it in MASTER
25:     for $m = 0$ to k do:
26:         indices=[i for i, j in enumerate(clusters) if j == m] #identify indexes of samples that belong to 1st,2nd,...k cluster
27:         cetroids[m]=$\frac{\sum df[i]\ for\ i\ in\ indices}{totcounter[m]}$ #recalculate means of centroids in each process
28:     end for
29:     centorid=comm.allreduce(centroid,MPI.SUM) #summing all means of each centroid from each process and broadcast new centroid matrix to all processes
30:     if centroid == initial then
31:         flag = False
32:     else
33:         initial = centroid
34:     end if
35: end while
```

Results and Data Analysis

From the following plots, it is evident enough that the performance is boosted significantly by parallelizing the algorithm using Open MPI. With increase in number of processes, one ideally expects a decrease in runtime but the output clearly violates it which may be explained by the element of hyperthreading that naturally comes into play with PCs of limited processors. Another potential reason for the latter observation is that the composed code for parallel version involves collective communication calls like *scatter*, *gather* and *reduce* under a *while* loop that terminates as the algorithm converges to a locally optimum centroid of clusters. For the dataset of size 4x10^4, at k = 3, the runtime for both serial and parallel forms of the algorithm is much higher than at k = 2 as (at least for the given dataset) the rate of convergence decreases with increase in the number of clusters which is not presented though.



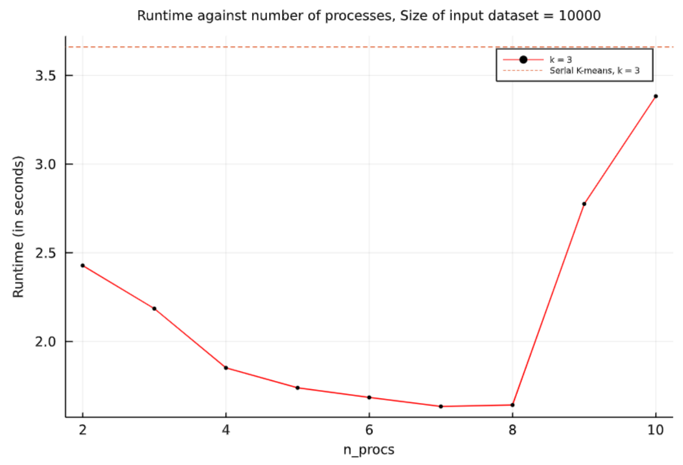
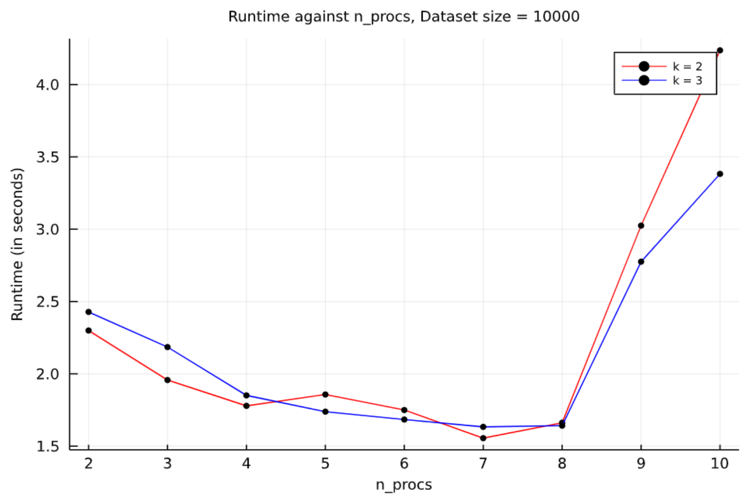


Fig 1: Plots of run-time of serial and parallel algorithms against different number of processes for dataset size = 10^4 and k = 2 & 3



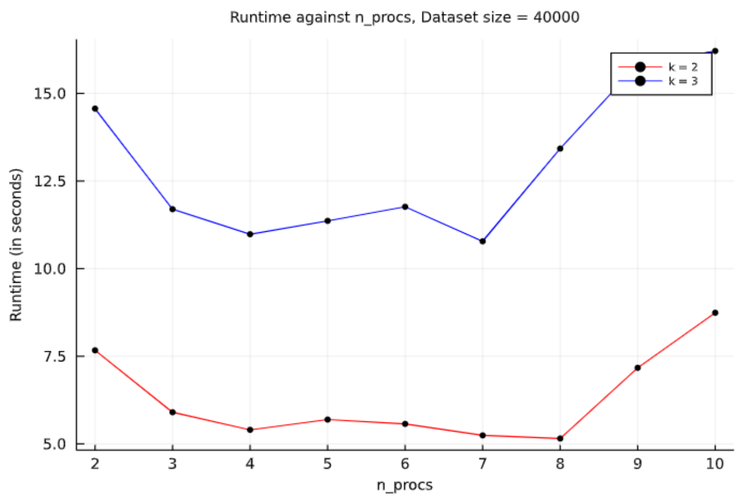


Fig 2: Plots of run-time of parallel algorithm against different number of processes for dataset size = 10^4, 4x10^4 and k = 2,3

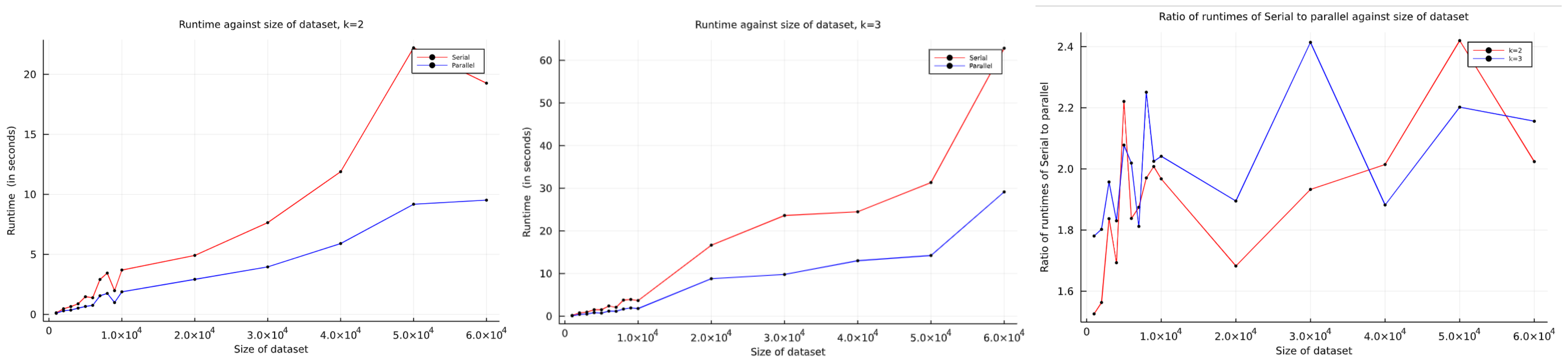


Fig 3: First two plots correspond to plots of runtimes of serial and parallel algorithms against different sizes of datasets for k = 2,3. Last plot corresponds to the plot of ratio of serial against size of dataset that reveals the amount of boost in performance.

References:

- Andrea Vattani. k-means Requires Exponentially Many Iterations Even in the Plane. Discrete Comput Geom (2011).
- David Arthur,Sergei Vassilvitskii†. k-means++: The Advantages of Careful Seeding. http://theory.stanford.edu/ sergei/papers/kMeansPP-soda.pdf
- Parallel programming model. https://en.wikipedia.org/wiki/Parallel programming model
- Message Passing Interface. https://en.wikipedia.org/wiki/Message Passing Interface
- Michael Jay Quinn. Parallel Programming in C with Mpi and Openmp . London : McGraw-Hill Higher Education, c2004.
- K-means clustering. https://en.wikipedia.org/wiki/K-means_clustering
- Christophe Picard. Chapter 3: Parallel Patterns, MPI and some examples. http://chamilo.grenoble-inp.fr/courses/ENSIMAGWMM9MO16/document/Resources/Lectures/3-chapter.pdf

Thanks!