Dhirubhai Ambani
Institute of Information and Communication Technology

Parallel Implementation of High Dimensional Data Clustering K-means Algorithm

Saumya Shah [201701442] , Harsh Tank [201701460]

Course Project Report : CS301 High Performance Computing
Assigned By : Prof. Bhaskar Chaudhury

Abstract
Despite its simplicity, serial k-means algorithm time complexity is expensive when it is applied to the high dimensional data. In data mining where lots of data is available for analysis in almost all the domains, clustering of high dimensional data has become one of the profound problem. Parallel approach of standard K-means algorithm, with shared memory model, is proposed in this paper. Algorithm is implemented on Intel Cluster using OpenMP Multi-threading Library. Our dataset includes .csv file having 32,000 of datapoints with 50 dimensions. Functionality correctness was done by metric - Dunn Index of the cluster.

[1] *201701442, 201701442@daiict.ac.in*
[2] *201701460, 201701460@daiict.ac.in*

Contents

Introduction

On everyday basis large amount of data is being collected in many domains. And to get the information from this data various methods are there. One method is to first group the data which are having similarity and this method is called Clustering and the groups formed are called Clusters. K-Means clustering algorithm is broadly used as a partitioning technique. The paper describes the parallel algorithm and results obtained where the similarity between the data is calculated by the Euclidean distance formula. One important assumption to be made is the data points are independent of each other. In other words there exists no dependency between any data points.

1. Serial Algorithm

Consider we have dataset of **n** points. $X_1, X_2, ..., X_n$ and every point is a vector in R_d (d-dimensions).The problem aims to partition the n data points into k clusters such that the average Euclidean distance between data points and cluster centroid is minimum and Dunn Index is maximum.
Euclidean Distance between point **p** and **q** each having d dimensions:

$$Distance(\mathbf{p},\mathbf{q}) = \sqrt{\sum_{i=1}^{d}(p_i - q_i)^2}$$

Algorithm is as follows:
Input: $N = X_1, X_2, ..X_n$
Output: K cluster
Procedure:

1. Randomly choose k data points c_j with $j = 1, 2..k$ from input set N as preliminary centroids.
2. Calculate the Euclidean distance between each data point X_i and cluster centroid c_j to find the closest cluster centroid.
3. Update dynamically cluster centroid as average of data point assigned to it.

$$c_j = \frac{c_j + X_i}{2}$$

4. Continue the process for certain number of iterations.

Considering the functionality correctness (Dunn Index) of the output, how better the clusters are formed can be known. See Functionality Correctness point in Further Detailed Analysis section for more information.

1.1 Complexity Analysis

- Time Complexity of Serial Code : $\mathcal{O}$(n*d*k*i)
  where, n = number of data points
  d = number of dimensions
  k = number of clusters
  i = number of iterations

- Space complexity of serial Code : $\mathcal{O}$([n+k]*d)
  where, storing datapoints takes $\mathcal{O}$(n*d) and centroids $\mathcal{O}$(k*d)

2. Scope, Strategy of Parallelization

2.1 Scope Of Parallelism

The profiling tool **gprof** indicates the more time is spent in the clustering part where data points is assigned to a cluster. Since it computes distance between each data points and cluster centroid. As each data point assignment to cluster is independent to other, it can be parallelized.

1. **In Clustering Algorithm**
   For computing minimum distance of data points from cluster centroids, data points are divided among the threads, each thread getting $\left\lceil \frac{n}{p} \right\rceil$ data points to work.
   Each thread take datapoint, find the distance from all centroids, and add suitably to that particular cluster having minimum distance i.e. maximum similarity.

   But the bottleneck involved. Communication between threads is needed to have the updated cluster centroid values and more than one thread may try to access the same centroid. And the modification should not be allowed at the same time otherwise it will cause data race. And for avoiding the possible loss of clustering efficiency because of contention of cluster centroid values, the algorithm is run for multiple times.
2. **Selecting Initial cluster centroids** Initial centroids are selected randomly from the datapoints given in input. To avoid centroid values contention it requires communication between threads if we parallelize this random selection.Instead, only way is to copy the selected centroid values in another initialized array. This can be done parallely but it provides negligible changes in execution time.
3. **Getting Input Data from csv file**
   The function is defined for reading the file to get 32,000 datapoints each having 50 dimensions. The function defined gets the data using file I/O functions. The reading of file is inherently done serially. The only scope is to parallelize call to **malloc** for heap space allocation. But being thread safe, it does not provide significant optimization.

2.2 Parallelization Strategy

The strategy used is loop parallelization. Granularity of algorithm which is coarse gain since high amounts of computation between small communication among the parallel threads is there. The program was parallelized using OpenMP multi threading library for C. To parallelize the code in OpenMP, the compiler directives are used.

- (Work Sharing) **#pragma omp for** is used to parallelize for loops, including malloc calls to allocate space. Also providing initial values to count based arrays and fixed increments to the output array. This directive divides iterations among threads dynamically to have maximum efficiency.
- (Parallel Regions) **#pragma omp parallel** is used in the clustering algorithm by creating parallel region and spawning threads. Also data scoping concept was used. Loop variables were made private and centroid values array was shared to give access to every required thread. False sharing can occurred which can be solved as given in next point.
- (Mutual Exclusion) **#pragma omp critical** is used to ensure that blocking access is given to a particular thread when it is updating the shared centroids. This is to ensure the synchronization of shared centroid values and false sharing/mutiple write operations do not cause contention of centroid values.

Proposed algorithm flow is as followed:
Reference to Figure 1.

3. Parallel Algorithm

Input: $N = X_1, X_2, ..X_n$
Output: K cluster
Procedure:

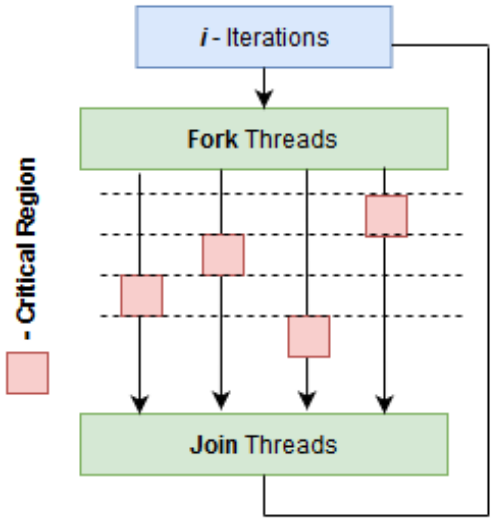

Figure 1. Algorithm Flow

1. Randomly choose k data points c_j with $j = 1, 2..k$ from input set N as preliminary centroids.
2. Initialize the p threads and partition the data into p parts. Giving each thread n/p data points.
3. In each thread compute the Euclidean distance between X_i and c_j. Find the closest one.
4. Update the centroid value critically and synchronize these values.
5. Continue the process for certain number of iterations.

- **Complexity analysis :**
  Time Complexity: $\mathcal{O}$(n*d*k*i/p) Space complexity: $\mathcal{O}$([n+k]*d)
  where, n = number of data points
  d = number of dimensions
  k = number of clusters
  i = number of iterations
- **Cost of parallel algorithm :**
  We have to pay cost as in parallel alogrithm there is communication overhead. To find it we run parallel code on 1 thread and serial code. The difference of time between them is the parallel overhead.
- **Theoretical speedup :**
  Theoretical Speedup $= \frac{Time-taken-for-serial-code}{Time-taken-for-parallel-code}$
  Theoretical Speedup $= p$
  Hence, Speedup directly depends on number of processors/threads.

4. Results and Discussion

4.1 Hardware Architecture Specifications

Reference to Table 1.

Table 1. Specifications

Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	24
On-line CPU(s) list	0-23
Thread(s) per core	2
Cores(s) per socket	6
Socket(s)	2
NUMA node(s)	2
CPU MHz	2085.562
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	15360K

4.2 Curve Based Analysis

- **Execution Time curve :**
  Figure2 is graph of Execution Time vs. Problem size for number of threads vary from 1 to 9. One can see in Figure 2 that as Number of data points(Problem Size n) increase execution time increase linearly, because Time complexity is linearly depend on Number of data points(n). Also figure 2 shows that as number of threads increase execution time decrease as paralleling increase. Also when number of thread is very high(more than 4) effect of parallelization decrease because of bottleneck discussed in section 2.1 as result we can see that execution time decrease very low.
- **Speedup curve :**
  Figure 3 is graph of speedup vs. Problem size for number of threads vary from 1 to 9. From figure 3 we can see that speedup increase as number of thread increase. The speedup remained almost constant on increasing thread (speedup for thread 8 and thread 9 is almost same). reason for this is because our test cases are heavily memory bound and we have to change in centers serially to maintain efficiency of algorithm. On the hardware we are using, where typically all the cores share one memory bus, so using more threads does not give us more bandwidth and, hence this is the bottleneck for speedup.If we have more threads(more than 15) than speedup decrease because of communication overhead. We can say that if we want to minimize the cost and maximize efficiency than we should take 4 or 5 thread.
- **Efficiency curve :** Figure 3 is graph of efficiency vs. Problem size for number of threads vary from 1 to 9. From figure 3 we can see that efficiency decrease as number of thread increase. For small problem size efficiency is less for given number of thread as we can from graph and as problem size efficiency increase but after some problem size efficiency remain constant and if problem size increase further than after some point efficiency will decrease.The efficiency at thread 1 and 2 goes above 1.0 maybe due to more speedup or due to out-of-control system latency during particular executions.

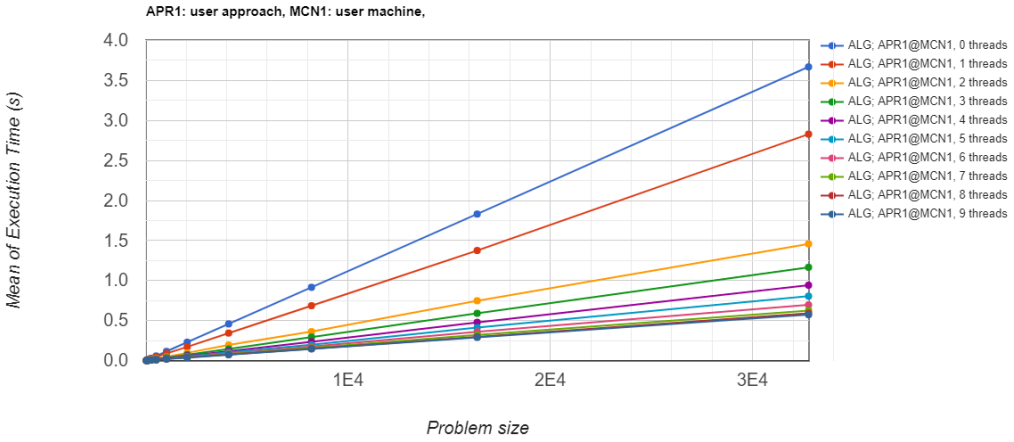

Figure 2. Execution Time

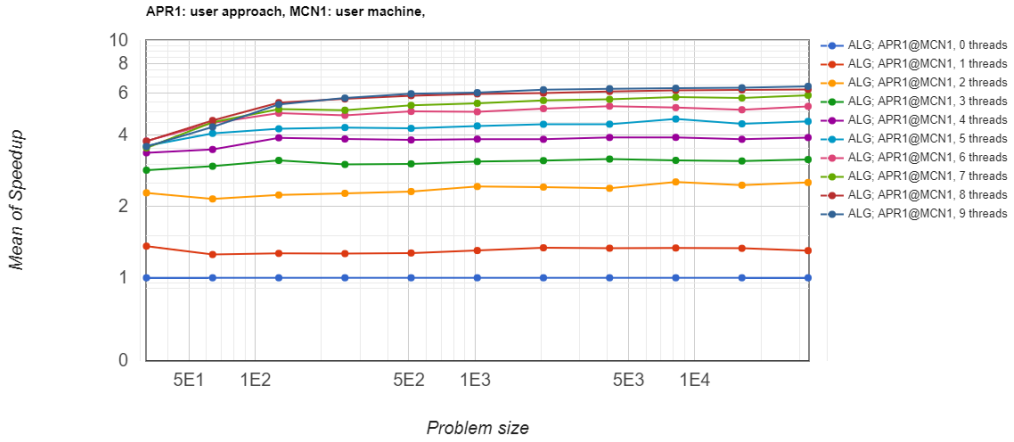

Figure 3. Speedup

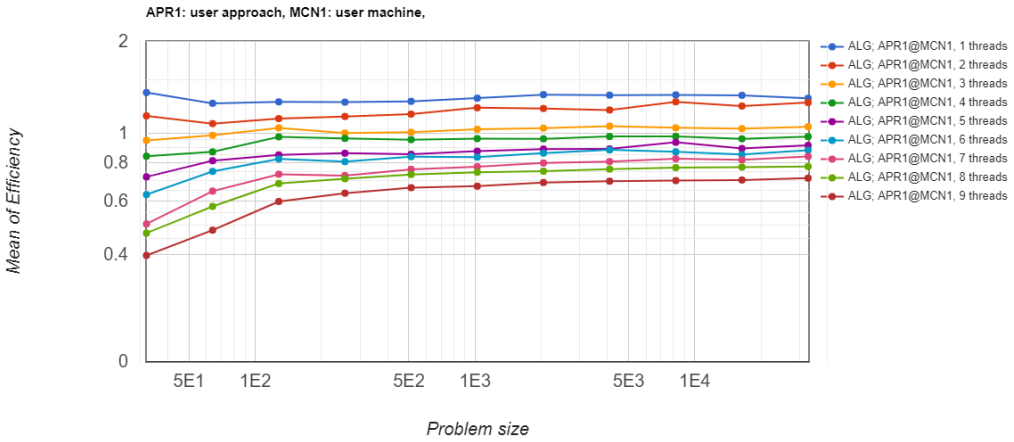

Figure 4. Efficiency

- **Karp Flatt metric :**
  Experimentally serial fraction can be determined using fundamental formula as defined below:

$$e = \frac{(p-1)\sigma(n) + pK(n,p)}{(p-1)T(n,1)}$$

where,
p = number of threads
n = problem size
σ = time taken by serial code which is function of n
K(n,p) = parallelization overhead
T = time taken by parallel code

But Karp FlatT Metric provides simplified computation giving the theoretical serial fraction using experimentally obtained speedup.

$$e = \frac{\frac{1}{\psi(n,p)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

where, $\psi(n,p)$ is practically obtained values of speedup.
e = theoretical serial fraction of code.

Analysis is done for different value of the speedup and theoretical serial part for Clustering. Following are the observed values :

NO. of Threads	Speedup	Serial Fraction
2	2	0
4	3.5	0.047
8	6	0.05

Using experimental speedup to determine the experimental serial fraction(e) of our code, we observe that the value of e does not show much variation with increase in number of threads. it shows the inefficiency of our algorithm.we come to the conclusion that there is a loss due to over head in the parallel fraction, and a possible explanation for that is the loss due the critical section of the code,where each thread has to wait for finish updating the centroid value.

4.2.1 Further Detailed Analysis

- **Functionality correctness**
  For this we have used Dunn Index which is the metric for evaluating the clustering algorithm. It is the ratio of the minimum inter-cluster distance to the maximum of average size of cluster. Inter-cluster distance $\delta(C_i, C_j)$ is the distance between the two cluster centroids . And average size of a cluster Δ_p is the distance between the centroid of cluster **p** and the data points in that cluster divided by the total points in that cluster.

$$\Delta_i = \frac{\sum_{x \in C_i} d(x, \mu)}{|C_i|}$$

where,

$$\mu = \frac{\sum_{x \in C_i} x}{|C_i|}$$

Dunn Index

$$DI = \frac{\min_{1 \leqslant i < j \leqslant k} \delta(C_i, C_j)}{\max_{1 \leqslant p \leqslant k} \Delta_p}$$

The DI lies between 0 and 1. A higher Dunn index indicates better clustering. The drawback of Dunn Index is, as the number of clusters and dimensionality of the data increase, there will be more computation. We get Dunn Index between 0.5-0.7. From this, it can be said that best clustering is not done everytime. As initial centroids chosen randomly, sometime convergence is high or less and it would affect the clusters formed.

- **Parallel Overhead**
  It is because there is critical section in code where at a time only one thread can be in that section and updating the cluster centroid value. As the number of threads increases, more threads will have to wait for their turn to go into that critical section to update the centroid value. For an instance, as the number of threads(p) increases, each thread have to do lesser computations since the load balancing would cause thread to work with lesser number of data points. But in turn it would require more time for the synchronization of cluster centroid value among the threads, leading to communication overhead.
  Time spend in communicating the data over threads and thus synchronizing the threads is also accounted in overhead due to parallelization.

- **Load balancing**
  It was done through **#pragma omp for** which shares iterations of loop between the given number of threads leading to $\left\lceil \frac{n}{p} \right\rceil$ data points for every threads to work.

- **Synchronization** Implemented using **#pragma omp critical** directive. Race conditions can be avoided by controlling access to shared variables by allowing threads to have exclusive access to the variables. But it inherently serialize the code as only one thread can have access at a time.

- **Scalability**
  The bus contention in shared memory leads to poor scalability.

5. Further Improvements

Different research shows that the time for computation of cluster can further be reduced using the hybrid implementation of this code in OpenMP and MPI. MPI may provides libraries

for taking input data parallely. It will become more efficient as reading of input data cosumes more time which is the bottleneck of efficiency.
Since it is an NP Hard problem we can use approximation techniques to reduce the number of iterations and increase speedup.

Acknowledgments

We would like to show our gratitude towards our course instructor Prof. Bhaskar Chaudhury for giving this opportunity to project our understandings of this course. Also thanks to our TAs for their help whenever required.

References

[1] V Ramakrishna Sajja Ds Bhupal Naik, S. Deva Kumar. Ieee international conference on computational intelligence and computing research (iccic). *Parallel processing of enhanced K-means using OpenMP*, DOI: 10.1109/ICCIC.2013.6724291:5, 2013.

[2] Antonis Maronikolakis. geeksforgeeks. *K means Clustering – Introduction*.