A Parallel Approach to High Dimensional Data Clustering (October 2015)

S. Chaitanya Prasad, Shaleen Kumar Gupta, Visharad Bansal;
Enrolled in B.Tech. (Hons. In ICT with minors in Computational Science),
Dhirubhai Ambani Institute of Information and Communication Technology, Gandhinagar

***Abstract*—High Dimensional Data Clustering has become one of the most important problems in data mining today, with gigabytes of data available for analysis in all domains including industries and social media. This paper presents a parallel approach to a modified version of the standard k-means data clustering algorithm, with a shared memory model used to increase the performance of the algorithm. Our data set consists of a .csv file containing 40,000 data points, each having 97 dimensions. We implemented our algorithm on an Intel Cluster having 16 cores using the OpenMP Multi threading Library. Performance was measured on two metrics, the Dunn index of the cluster and the Root Mean Squared error.**

***Index Terms*—Data Clustering, k-means, OpenMp, Dunn Index**

I. Introduction

Data sets measuring gigabytes and even terabytes have become increasingly common today, with at least a few billion points of data in any standard raw data set. Given that, a lot of work is being done to come up with efficient algorithms for data clustering based on various metrics, both in terms of time efficiency of the algorithm and its output efficiency. Given that, parallel computing becomes a natural solution to the problems. In this paper, we present an algorithm which is a slight modification of the standard k-means algorithm for high dimensional data clustering, with the Euclidian metric used as a standard for calculating the distance between two data points.

Section II gives a summary about the algorithm applied, Section III mentions the performance optimization techniques, Section IV delves on the details of the parallelization techniques applied in the algorithm,. Finally, Section V analyses the functional correctness and and efficiency measurements of the algorithm.

II. Clustering algorithm

Suppose that we are given a set of N data points $X_1, X_2, \ldots, X_n$ such that each data point is in dimensions R_d. The algorithm followed is a modification of the popular k-means clustering algorithm which seeks to minimize:

$$\left(\frac{1}{n}\right)\sum_{i=1}^{n} \min_j d^2(X_i, m_j) \qquad \text{- (1)}$$

The clustering algorithm is as follows:-

1. (Initialization) Select a set of k starting points $m_j \; with \; j=1,\ldots k$. The selection may be done in a random manner or according to some heuristic.
2. (Distance Calculation) For each data point X_i, $1 \le i \le n$, compute its Euclidean distance to each cluster centroid m_j, $1 \le j \le k$, and then find the closest cluster centroid.
3. (Centroid Recalculation) As each data point X_i is assigned a cluster j, the centroid of the cluster is updated dynamically according to the following rule:-

$$m_j = \frac{(m_j + X_i)}{2} \qquad \text{-(2)}$$

4. (Repeat Step) Repeat the above step for a set number of iterations.

In the classical k-means algorithm the centroid recalculation is performed by calculating the median of all the points belonging to a cluster j at the end of one run of classification.
In the algorithm followed by us, as soon as a data point is clustered to a particular cluster, the cluster point is recalculated according to point 3 given above. This recalculation is performed before moving onto the classification of the next data point.

III. Performance optimizations

The “gprof” profiling tool revealed that the program spent a majority of its time in the actual clustering algorithm where clusters were assigned to each data point. The program was parallelized using OpenMP multi threading library for C.

The parallel algorithm is a coarse grain algorithm, with high amounts of computation between small communication among the parallel threads. Following is an analysis of the various parts of the algorithm and the extent of parallelism explored in them:

1. Reading from the .csv file

This function reads the contents of the input data file containing the data points. The given input file consists of 40,000 data points and 97 dimensions in each data point. The contents of the file are read using the File I/O functions in C. The only parallelizable portion in the code is the heap space allocation call to the memory allocator “malloc”. Since the “malloc” call is thread safe, it gives a marginal improvement over the serial version, with the serial function taking an average time of 3.9 seconds and the parallel version taking an average of 3.8 seconds.

2. Choose Initial Centroids

The initial centroids are chosen by randomly selecting data points from the given data set. Parallelization of the choosing process does not add to the efficiency, as it involves extensive communication between the threads in order to avoid contention for the same data points. Thus, the only scope for parallelization is in the array initialization loop and the copy loop which copies the selected data points into the center array containing the initial centroids. This gives negligible improvement in the overall running time.

3. The Clustering Algorithm

This function gives the maximum scope for parallelization, given its inherent nature. The task of computing the minimum distance of the data points from the cluster centroids is divided among p threads, with each thread getting n/p data points to work upon. Thread number i gets data points i, 2i, 3i….upto n. Each thread takes a data point, computes its distance from all the centroids using the Euclidean metric, and records the minimum distance as well as the cluster number which gives this distance. The data point is suitably added to that particular cluster.

The second part of the algorithm involves communicating the updated the centers among the various threads, to ensure proper clustering. When a thread assigns a data point to a particular cluster, it updates the shared value of the center array (containing the centroid coordinates) for all the threads, by taking the mean of the distance between the data point and the current centroid. In order to overcome the possible loss of clustering efficiency because of the contention for centroid points, the clustering algorithm is run multiple times. This ensures a good clustering of the data points. Given that there are 40,000 data points in the given input data set, there is enough computation for each thread to overcome the communication overheads between the threads.

Apart from this, simple “for” loops are parallelized, including the loop working on the output array to make it a one based index.

IV. Parallelization techniques applied

The OpenMP multi threading library was used to parallelize the algorithm. The following constructs were involved in the same:

1. The **#pragma omp for** construct is used to efficiently parallelize for loops, including malloc calls the allocate space to 2 dimensional arrays, providing initial values to count based arrays and fixed increments to the output array. The construct dynamically divides the iterations among the threads to ensure maximum efficiency.

2. The **#pragma omp parallel** construct is used in the clustering algorithm, with the threads spawned diving the data points among themselves. Also, the concept of private and shared variables is used, making loop variables private and data, centroid points shared, in order to give proper access to the required variables to all the threads without concurrency problems.

3. The **#pragma omp critical** construct is used to ensure that blocking access is given to a particular thread when it is updating the shared centroids. This is to make sure that multiple write operations do not corrupt the centroid values.

V. Functionality correctness and results

We observed the clustering time by varying the number of iterations we were looping till and the number of clusters being formed. By hit and trial, we concluded that giving the number of clusters = 22 and the number of iterations 10 gave us the desired output in a very less time with an appreciable Dunn Index varying approximately between 0.44 and 0.51 and giving a Mean Squared Error of 8.04 which was within our limits of error, which was fixated at 10.

So, following were our results with 10 iterations in the program.

- Number of clusters = 22
- MSE = 8.0448
- Best end-to-end time (in secs) = 5.20274
- Average end-to-end time (n secs) = 5.58055664
- Best Clustering Time (in usecs) = 815904
- Average Clustering Time (in usecs) = 1199376.7

REFERENCES

[1] Inderjit S. Dhillon, Dharmendra S. Modha, in A Data-Clustering Algorithm On Distributed Memory Multiprocessors