CS301: High Performance Computing

Course Instructor: Prof. Bhaskar Chaudhury

Course Project Autumn 2015

Efficient Optimization of High Dimensional Data Clustering Algorithm

Authors:
Shaleen Kumar Gupta [201301429]
Visharad Bansal [201301438]

Table of Contents

1. Introduction
    1.1. Problem Introduction
        1.1.1. Problem Statement:
        1.1.2. Mathematical Details and Equations:
        1.1.3. Functionality Correctness
        1.1.4. Hardware Specifications
    1.2. Sample Input and Output
    1.3. Real World Applications
2. Approach to solve the problem
    2.1 Serial Algorithm
    2.2. Serial Pseudo-Code
    2.3. Scope of Parallelism
    2.4. Link to download Serial Code
    2.5. Effect of increasing problem size on serial code
3. Towards Parallelizing the Algorithm
    3.1. Description of the Parallel Code (Pseudo Code)
    3.2. Strategy of Parallelization
        3.2.1. Reading from the csv file: takeDataInput()
        3.2.2. Choose Initial Centroids: chooseCenterPointsRandom()
        3.2.3. The Clustering Algorithm
    3.3. Link to download Parallel Code
4. Results and Discussion
    4.1. Effect of Parallelization: No. of Threads v/s Time Taken
    4.2. Speedup Curve as a Function of No. of Threads
    4.3. Speedup Curve as a Function of Problem Size
    4.4. Variation of serial part with increase in number of threads
5. Observations, Conclusions and future scope
    5.1 Observations and Conclusions
    5.2 Future Scope
6. Bibliography: References and Credits

1. Introduction

1.1. Problem Introduction

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.

1.1.1. Problem Statement:

Cluster a high dimensional unlabelled data set. Assign membership to each of the data points.

1.1.2. Mathematical Details and Equations:

The Euclidian metric is used to calculate the distance between any two data points.

Euclidian distance between any two points **x** and **y,** having dimensions [x0, x1, … , xk] and [y0, y1, … , yk] is given by:

$$E(x,y) = \sqrt{\sum_{i=1}^{k} (xi - yi)^2}$$

A point is included in the cluster to which it is closest, i.e the cluster whose center is at the least Euclidean distance from the given point.

1.1.3. Functionality Correctness

The Dunn Index is the ratio of the smallest distance between observations not in the same cluster to the largest intra-cluster distance. The Dunn Index has a value between zero and one, and should be maximized.

Apart from the Dunn Index, another accuracy metric we have used to study the efficiency was the Mean Squared Error. We used an octave script to calculate the same.

The script may be downloaded from this GitHub link (goo.gl/0aNwRT).

1.1.4. Hardware Specifications

A High Performance Cluster developed by Intel as part of the High Performance Computing Conference, Bangalore, 2015, was used in evaluating the compute times and speedups in the course of the project.

Architecture:	GenuineIntel x86_64
Byte Order:	Little Endian
Cores per Socket:	8
Sockets:	2
CPU MHz:	1200.000
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	20480K

Compilers and other specifications:

Operating System: CentOS release 6.4 over Red Hat 4.4.7-3
C Compiler: gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3)
Python Interpreter: Python 2.6.6
Octave Version: GNU Octave, version 3.8.2
Graph Plotter: Plotly 1.8.11

1.2. Sample Input and Output

We will be given a High dimensional unlabelled data set with all real values. The data file is in csv format. Each line is a record, or data point. The attributes within the records or data points separated by commas.

Example:
|<------- attributes ------>|
_
1.321, 2.345, ... , 3.35 \n |
2.324, 1.115, ... , 7.95 \n |
.... |
.... | No. of data points.
.... |
|
1.821, 7.916, ... , 2.79 EOF |
-

The output would be an ASCII file containing comma separated integers (essentially a csv file). The clusters would be labelled with integers starting from 1. If M clusters are found, the labels should be 1, 2, ... M. Each data point should be assigned to a cluster. The results should be a comma separated list of integer labels showing the membership of the data point to the cluster.

Example: A sample output file may contain the following result:

[10, 15, 2, 7, 31, 10, 1, 1....]

Which shows that the 1st and 6th data points are member of the cluster labelled 10, the second data point is member of the cluster labelled 15 and so on...

1.3. Real World Applications

Data Clustering algorithms are predominantly used in Machine Learning Techniques where a huge of amount of sampled data is studied to predict the outcome of an event for a particular entity in or outside the data set. A quantitative value of a set of attributes is assigned as various dimensions of a data point and the situation is modelled in an n-dimensional vector space. Some specific examples of such algorithms are mentioned as follows (references mentioned below each application)

1) Clustering Algorithm in Identifying Cancerous Data

Clustering algorithm can be used in identifying the cancerous data set. Initially we take known samples of cancerous and noncancerous data set. Label both the samples data set. We then randomly mix both samples and apply different clustering algorithms into the mixed samples data set (this is known as learning phase of clustering algorithm) and accordingly check the result for how many data set we are getting the correct results (since this is known samples we already know the results beforehand) and hence we can calculate the percentage of correct results obtained.

References

1) A Comparison of Fuzzy and Non-Fuzzy clustering Techniques in Cancer Diagnosis by X.Y. Wang and J.M. Garibaldi.
2) Probability Density Estimation from Optimally Condensed Data Samples by Mark Girolami and Chao He.

2) Clustering Algorithm in Search Engines

Clustering algorithm is the backbone behind the search engines. Search engines try to group similar objects in one cluster and the dissimilar objects far from each other. It provides result for the searched data according to the nearest similar object which are clustered around the data to be searched. Better the clustering algorithm used, better are the chances of getting the required result on the front page. Hence, the definition of **similar object** play a crucial role in getting the search results, better the definition of similar object better the result is.

Most of the brainstorming activities needs to be done for defining the criteria to be used for similar object.

References

1) Clustering Billions of Images with Large Scale Nearest Neighbor Search by Ting Liu, Charles Rosenberg and H.A. Rowley.
2) Probability Density Estimation from Optimally Condensed Data Samples by Mark Girolami and Chao He.

3) Clustering Algorithm in Academics

The ability to monitor the progress of students' academic performance has been the critical issue for the academic community of higher learning. Clustering algorithm can be used to monitor the students' academic performance. Based on the students' score they are grouped into different-different clusters (using k-means, fuzzy c-means etc), where each clusters denoting the different level of performance. By knowing the number of students' in each cluster we can know the average performance of a class as a whole.

References

1) Application of k-means clustering algorithm for prediction of students' academic performance by O.J. Oyelade, O.O. Oladipupo and I.C. Obagbuwa.

4) Clustering Algorithm in Wireless Sensor Networks based Application

Clustering Algorithm can be used effectively in Wireless Sensor Networks based application. One application where it can be used is in Landmine detection. Clustering algorithm plays the role of finding the Cluster heads(or cluster center) which collects all the data in its respective cluster.

References

1) Clustering of wireless sensor and actor networks based on sensor distribution and connectivity by Kemal Akkaya, Fatih Senel and Brian McLaughlan.
2) Wireless Sensor Network based Adaptive Landmine Detection Algorithm by Abhishek Saurabh and Azad Naik.

2. Approach to solve the problem

2.1 Serial Algorithm

Let us say that we are provided with a set of n data points X1 , X2, ⋯ , Xn such that each data point is in R^d. The problem of finding the minimum variance clustering of this data set into k clusters is that of finding k points $\{m_j\}_{j=1}^k$ in R^d such that the Mean Squared Error, as will be described in the following paragraph, is minimized. The points $\{m_j\}_{j=1}^k$ =1 are known as cluster centroids or as cluster means.

In a nutshell, the problem in (1) is that of finding k cluster centroids such that the average squared Euclidean distance (which is basically the mean squared error or MSE) between a data point and its nearest cluster centroid is minimized. Unfortunately, this problem is known to be NP-complete[1]. The classical k -means algorithm provides an easy-to-implement approximate solution to (1). Reasons for popularity of k -means are ease of interpretation, simplicity of implementation, scalability, speed of convergence and its ability to adapt to sparse data. We describe our algorithm[1] as follows:

1. (Initialization) Select a set of k starting points $\{m_j\}_{j=1}^k$ =1 in Rd. The selected may be done in any random manner.
2. (Distance Calculation) For each data point Xi, 1 ≤ i ≤ n, compute its Euclidean distance to each cluster centroid mj, 1≤j≤k, and then find the closest cluster centroid.

3. (Centroid Recalculation) For each 1≤j≤k, recompute cluster centroid mj as the average of data points assigned to it.
4. (Convergence Condition) Repeat steps 2 and 3, until convergence.

This algorithm is essentially a typical Machine-Learning type gradient descent algorithm. This algorithm starts with taking random centroids and then going on updating them cycles by cycle until the mean squared error comes within a definite range.

Our code is an adaptation of the standard k-means algorithm, with modifications done in order to improve the scope of parallelization while keeping in mind the functionality correctness (Dunn Index and MSE) of the output. The basic deviation from the standard algorithm is the process of center updation, wherein after adding a data point to a cluster, the centroid of the corresponding cluster is updated to the average of the current centroid and the data point.

The complexity of the serial code is **O(n*x*k)**, where n is the number of data points, x is the number of clusters and k is the number of dimensions in each data point.

2.2. Serial Pseudo-Code

As adapted from the Euclidean Distance Based K-Means Algorithm[1].

1: Take data input from file
2: Select initial cluster centroids $\{m_j\}_{j=1}^{k}$;
3: do{
4: for j= 1 to k

```
5:          m′j= 0;nj= 0;
6:      endfor;
7:      for i=1 to n
8:          for j= 1 to no_of_clusters
9:              compute squared Euclidean distance d2(Xi, mj);
10:         endfor;
11:         find the closest centroid mℓ to Xi;
12:         update centroid  mℓ =  (mℓ + Xi) / 2
13:     endfor;
14: }while(number_of_itrations>0);
```

2.3. Scope of Parallelism

Reading from a .csv file and filling up the data array with the appropriate values. This requires massive time, as reading from a text file is inherently serial. However, it does have certain scope of parallelization, in terms of thread safe dynamic memory allocation and some inherent libraries provided for parallel data input.

The clustering algorithm requires massive computations, with distance between each data point and each centroid being calculated. Since calculation of the appropriate centroid for each data point is independent of the others, the algorithm provides a good scope for parallelism.

However, there is a bottleneck. The threads need to communicate among themselves to keep the centroid values updated, as more than one thread might try to access the same centroid point. In that case, it is imperative to ensure that both threads do not try to modify the centroid at the same time, as it might result in corrupted values.

2.4. Link to download Serial Code

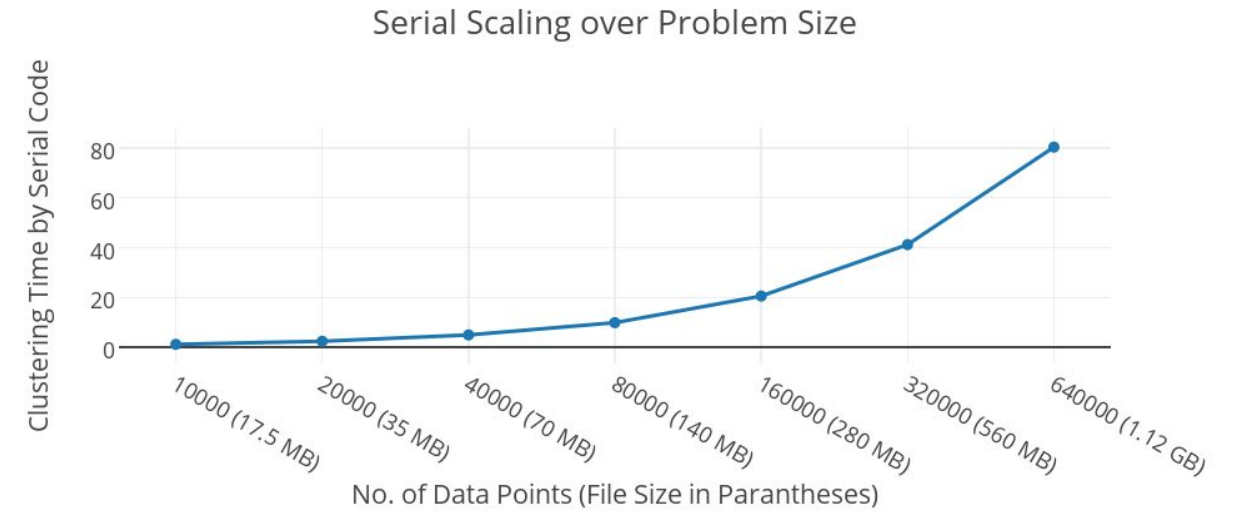
The source code of the Serial Implementation of the algorithm may be downloaded from this GitHub link (https://goo.gl/HLLClL).

2.5. Effect of increasing problem size on serial code

An increase in problem size essentially means an increase in the number of data points to be clustered. In such a scenario, the limitation comes in two forms: time taken in data input and the time taken for actual clustering.

Given that reading from the file is done sequentially, there is a linear rise in the amount of time taken as the data size increases.

With an increase in the number of data points processed, the problem becomes more computationally intensive, leading to an increase in the time taken.



3. Towards Parallelizing the Algorithm

3.1. Description of the Parallel Code (Pseudo Code)

```
1: Take data input
2: Select initial cluster centroids {m_j}^k_{j=1};
3: do{
4:      Spawn p threads{
5:              for i=myid∗(n/size) to (myid + 1)∗(n/size)
6:                      for j= 1 to k
7:                              compute squared Euclidean distance d^2(X_i, m_j);
8:                      endfor;
9:                      find the closest centroid m_ℓ to X_i;
10:                     update centroid value critically: m_ℓ = (m_ℓ + X_i) / 2
11:             }
12:  }while(iterations>0);
```

3.2. Strategy of Parallelization

The algorithm is a coarse grain, 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:

3.2.1. Reading from the csv file: takeDataInput()

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

fread() function in C. The only parallelizable portion in the code is the heap space allocation call to malloc. Threads are used to allocate space to the arrays data [ ][ ] and center_array [ ][ ]. 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.

3.2.2. Choose Initial Centroids: chooseCenterPointsRandom()

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, although this gives negligible improvement in the overall running time.

3.2.3. 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 Euclidian 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 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.

3.3. Link to download Parallel Code

The source code of the Serial Implementation of the algorithm may be downloaded from this GitHub link (https://goo.gl/4OOSXI).

4. Results and Discussion

4.1. Effect of Parallelization: No. of Threads v/s Time Taken

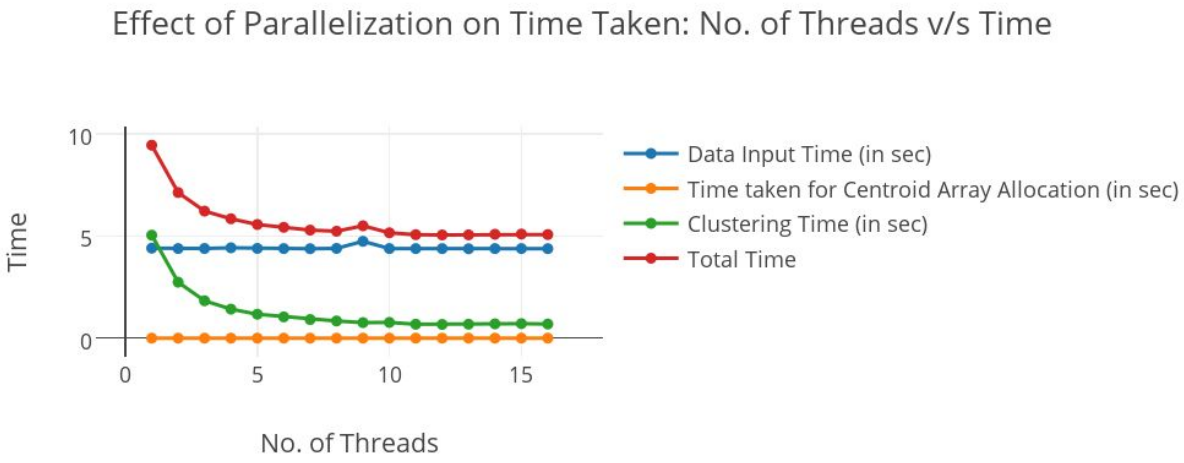


Fig. 4.1

As can be seen from the graph [Fig. 4.1], data input, although taking somewhat constant time, becomes the driving factor of the total time taken by the code. With an increase in the number of threads, a clear decrease is seen in the time taken for clustering, whereas the time for choosing the initial centroids is almost negligible. There are a few kinks to be observed, owing to the fact that a random function is used to allocate the initial centers, which at times can give some erratic results. However, the overall analysis shows a positive effect on the speed of the clustering algorithm with an increase in the number of threads.

4.2. Speedup Curve as a Function of No. of Threads

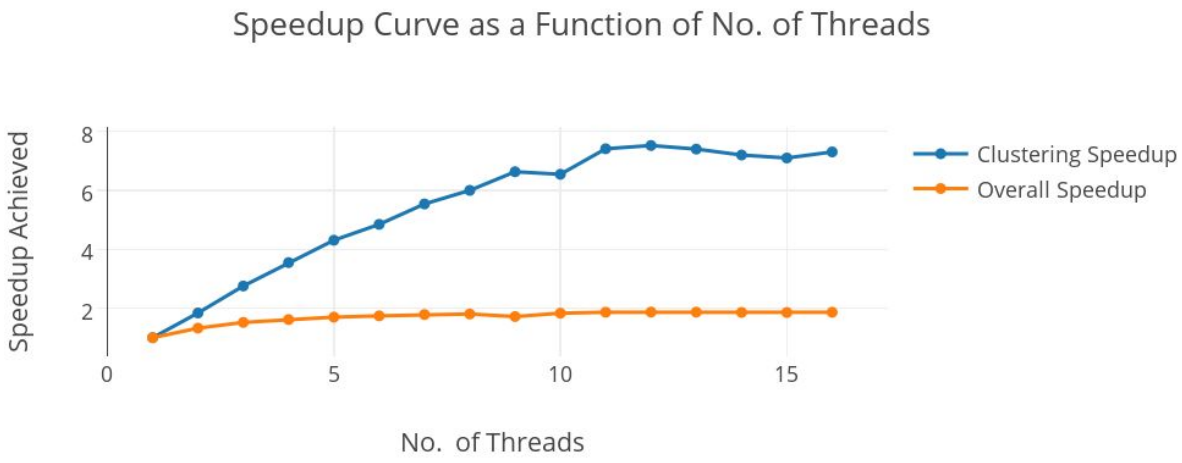


Fig. 4.2

Evidently, we can observe from Fig. 4.2 that the speedup achieved for the Data clustering increases with the number of threads.

However, the increase in the overall speedup is not much over the span of 1 to 15 threads. This is because the performance bottleneck lies in the taking the data input. Since taking the input could not be parallelized, it remains inherently serial, and hence it majorly dominates the clustering time. However, there are certain third party libraries which support parallel data input but that appeared a bit out of scope of this project. See Section 5.2 for more information on this.

The speedup for the clustering increased almost linearly in the relatively lesser number of threads range, while it was towards flattening out in the farther range. The maximum speedup achieved was 7.52238805970149.

We also observe that there are a few irregularities in the speedup curve, thus obtained. This is primarily because every output relies on a set of random points chosen at the start of the algorithm. Different set of points, or different inputs may vary the time taken by the program to undertake the

clustering. So, in spite of taking averages, the times written thus are approximate.

The above calculations were made for a Data Input Size of 97-dimensional 40k points (File Size: 70 MB), and the clustering was done over 22 clusters and in a span of 10 iterations. The choice of these values were made after taking into consideration the Mean Square Error as calculated by the octave script [Refer Section 1.1.3].

4.3. Speedup Curve as a Function of Problem Size

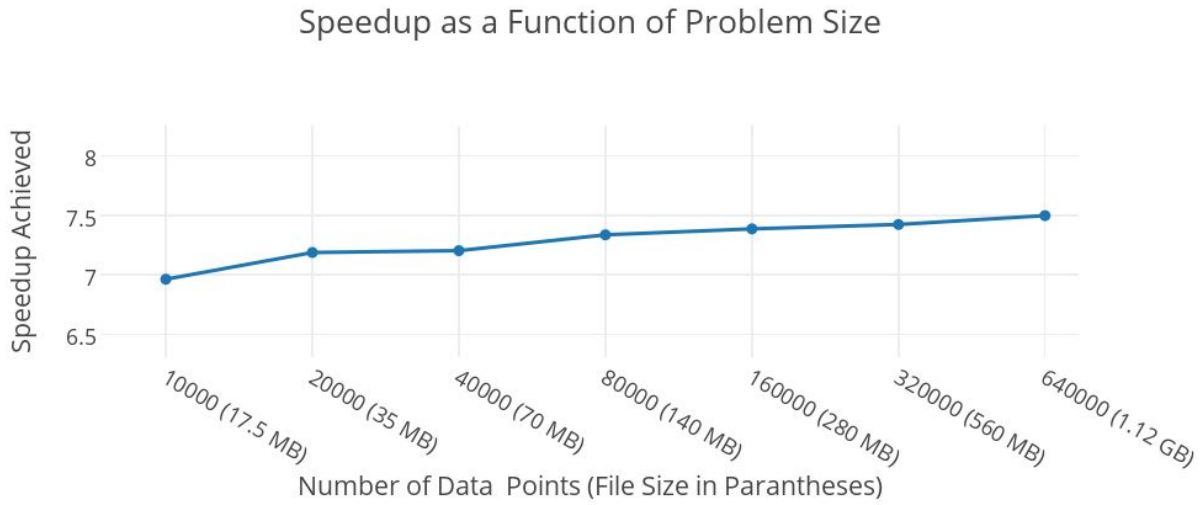


Fig. 4.3

We observe from Fig. 4.3 that the speedup achieved remains almost constant for varying the size of inputs. Over a range of input sizes, the speedup achieved mildly increases from about 6.93 to 7.51. We discuss the reasons for this in the discussion that follows [Section 5.1].
These calculations were made by fixating the number of threads at 12, no. of clusters at 22, and no. of iterations at 10.

4.4. Variation of serial part with increase in number of threads

The fundamental formula for experimentally determining the Serial fraction of the code is defined as follows:

$$e = \{(p-1)\sigma(n) + pK(n, p)\}/\{(p-1)T(n, 1)\} \quad \text{- (A)}$$

where e is the experimentally determined serial fraction of the program, sigma is the time taken by the serial code, p is the number of threads, n is the problem size, and T is the time taken by the parallel code.

Using the simplified Karp-Flatt Metric, as defined below, we attempt to figure out possibly explanations for our observations.

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}} \quad \text{- (B)}$$

We use (B) in order to theoretically determine the experimentally serial fraction of our code using the practically obtained values of speedup. Analysis is later done for both cases: the speedup for Data Input+Clustering and only for Clustering. Following are the observed values:

No. Of Threads	4	8	12	16
Experimentally Determined Overall Serial Fraction	0.491749175	0.490655686	0.492776886	0.505978776

Experimentally Determined Serial Fraction in the Clustering part of the program	0.042328654	0.047619048	0.054114252	0.079366036

Using (B) to determine the experimental serial fraction(e) of our code (Data Input+Clustering), we observe that the value of e does not show much variation with increase in number of threads.
According to (A), it shows the inefficiency of our algorithm, which agrees with our analysis, given that too much time is used up in data input.

Next, we do the analysis only for the clustering part of our code, and observe a linearly increasing values of e. Using (A), we come to the conclusion that there is a loss due to overhead 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 its cousin to finish updating the centroid value.

5. Observations, Conclusions and future scope

5.1 Observations and Conclusions

We observe the speedup curve for a variable number of threads flattening out at around 8 threads, and remaining almost constant thereafter.

To explain our observations from Fig 4.2, one of the possible reasons we could figure out for the observed flattening is that the program employs a ‘critical’ section in every thread when the center array is being updated by any of the threads. With increasing number of threads, more number of threads would be left idle with any thread executing the critical section. To put it simply, as the number of threads increase, the computation per thread goes down as each thread now works on fewer number of data points (n/p). At the same time, it causes an increase in the synchronization among the threads as more and more threads now have to synchronize the centroid values.

Also, our hardware specifications indicate that the node consists of 2 sockets, each having 8 cores. This could be another possible reason for a performance decrease beyond 8 threads, as it now involves inter socket communication as well.

To address our observations from Fig. 4.3, we observe that the speedup remained almost constant on increasing the problem size. We believe that there are two reasons for this. One of the reasons we believe it is so, is because we figure that our test cases are heavily memory bound. 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. This will probably change if we reduce

the problem size so it will fit into cache or for sure if we increase the number of calculations per data. But, to address the real world problems that data scientist across the world encounter, doing so would be very unrealistic. Only if we have access to a second memory bus with more threads, as with some multi-socket architectures, we could expect to get an increase in the speedup achieved with increasing problem size. Another reason for this, is that the parallel fraction of time taken by the program is very meagre when compared to the serial fraction of the code. So, for greater problem sizes, the serial part of the code majorly dominates the parallel part, which results in a downfall in the rate of increase of speedup. After a point of time, the speedup almost flattens out, because the serial fraction dominates the parallelization.

5.2 Future Scope

A major bottleneck in the efficiency turned out to be the data input function, with the reading of huge data files consuming a lot of time. As a solution, in multithreading one can use tcmalloc(), or thread safe malloc, in order to make dynamic memory allocation more efficient as well as thread safe.
The given algorithm is coarse grain, specially for large data inputs. In such cases, it might be a good idea to implement the given code on a distributed memory system. Also, MPI provides special libraries for parallel data input, which can be a huge advantage in terms of speedup.

6. Bibliography: References and Credits

[1] HiPC 2-15 Parallel Programming Challenge Mentoring Forum. [External Link]

[2] Inderjit S. Dhillon, Dharmendra S. Modha, in ‘A Data Clustering Algorithm On Distributed Memory Multiprocessors’. [External Link]

[3] Maria Halkidi, Yannis Batistakis, Michalis Vazirgiannis, Department of Informatics, Athens University of Economics & Business, in ‘Clustering Validity Checking Methods’, ACM Sigmod Record, 2002 [External Link]