Safiya Jan Andrew ID - sjan

Clustering Data Points and DNA Strands using MPI

Introduction

This project centers around using a message passing model to implement a clustering algorithm on 2D data points and on DNA strands. During this project, I will be using MPI (Message Passing Interface, a library standard that is used for writing message passing program. The clustering algorithm that I will be implementing is k Means. Clustering allows us to assign a set of data points to a group so that the degree of similarity between the member of a group is high but the similarity between groups is small. Therefore, the objective of clustering is to maximize intra-cluster similarity and minimize inter-cluster similarity. K Means is an iterative algorithm that tries to find k similar groups in a data set via minimizing a mean squared distance function. Initial guesses of k means are made and these estimates are then used to classify the data points into k clusters. After each iteration, each mean is re-calculated to reflect the actual mean of its constituent objects. The algorithm keeps iterating until the calculated means stop changing.

In this project, I have implemented the K Means algorithm using sequential and parallel programming using MPI for a DNA dataset and a 2D data set. I then conducted various experimental analysis to measure the efficacy of the parallel implementation. This report outlines and details the analysis and results, along with providing a discussion of the implementation and the results.

Comparison of Implementations – Parallel and Sequential

2D Points Sequential and Parallel – Development

To find the which points belongs to which group (or cluster), we find the Euclidean distance of each point to all the initial means (centroids). The minimum distance between each point and each centroid is calculated and the point then belongs cluster with the centroid to which its closest to. This way, the algorithm finds the closest cluster to each point and assigns that cluster to that point. The centroids are re-calculated by averaging all points within a cluster and the algorithm iterates again. The iterations continue until the centroids stop changing. The only difference between the sequential and parallel implementation is that the points and initial centroids are distributed across different machines and each machine calculates which cluster each point belongs to and sends its local sums of x and y coordinates to the master so the master can calculate the new centroids. The master then sends the re-calculated centroids to the slaves and the process begins again. The communication between the master and slave machines is done through **MPICH2** (an implementation of the Message Passing).

In terms of development, I started off with the sequential implementation and then moved on to the parallel computation. The sequential implementation was not as hard because you do not have to deal with any synchronization concurrency issues like with MPI. In the parallel implementation, one thing I struggled with was the distribution of to each machine. I was reading all the data points into an array of 2D arrays and sending a point at a time to each slave and because the MPI Send and Receive function are extremely expensive in time this caused the distribution of the points to became extremely slow. This was solved by

reading all points into a 1D array and sending the points by indexing into that array. This allowed for far less communication between the machines.

2D Points Sequential and Parallel – Performance

What I've found in terms of performance was that the parallel version performed far better when there was a large dataset and a large number of iterations to be done. Even though there is an overhead of time in distribution of points to the slaves in the MPI version, this is offset by the fast computation of the final centroid as the work is split across multiple machines and done parallely

DNA Strands Sequential and Parallel – Development

The same idea that was used for the 2D data point clustering was then extended for clustering DNA strands. Rather than using the Euclidean distance for finding the closest clusters, a similarity function was used instead. This similiarity function measures how similar 2 DNA strands are. For example, ACTG and TCGA are 2 DNA strands that have a similarity value of 0, whereas ACTG and ACTT are strands that have a similarity value of 3. To recalculated the new centroids, the max occurring base at each index of each strand was used to create the new centroids.

In terms of development I did the same as what I did for the 2D points. I started with creating the sequential implementation and then the parallel one. One thing I struggled with was reading the strands from the file, since each strand was a line and I was reading characters not numbers, that made the reading a little more complicated.

DNA Strands Sequential and Parallel – Performance

In terms of performance, it was similar as to what was noticed in the 2D data points implementation. The MPI version performed much better when there was a large data set and a large number of iterations

Scalability Study for Parallel Implementation (on 2D data set)

In order to test my MPI version, I performed 3 scalability studies on my implementation. They are as follows.

1) ### The number of processes with a fixed data set size using 2, 4, 8 and 12 processes on 4 virtual machines

   For this study, I used a dataset size of 1 million with 5 clusters and ran the program on 2, 4, 8 and 12 processes. I ran the program 3 times each on 2, 4, 8 and 12 process and got the average run time for each. Figure 1 shows the result for this experiment. Here we can see that as the number of processes on the VMs increase, the time taken for clustering decreases. There is a significant decrease between the 2 and 8 processes. However, there is not much of a difference between time taken for clustering using 8 and 12 processes. This is probably due to the fact that the data set is not large enough to be split across 12 processes and this causes an increase in communication and so the benefits of the fastness of MPI is not necessarily seen

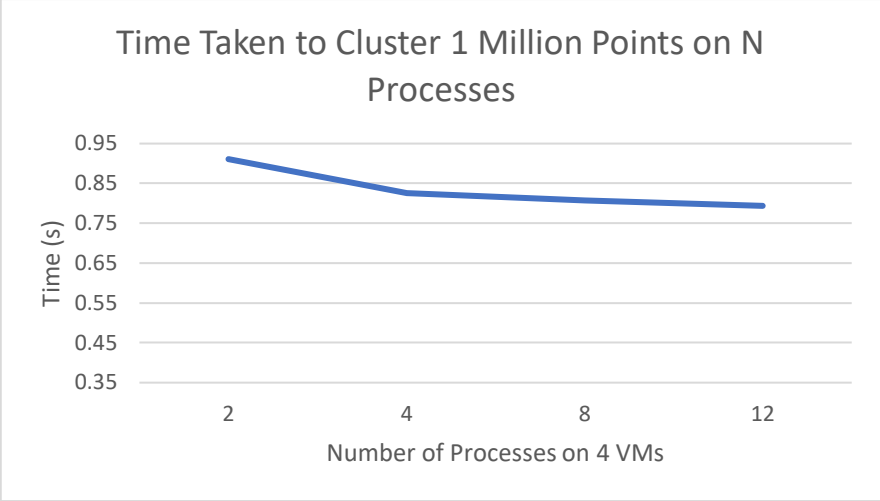

Figure 1 – Clustering 1 million points using multiple processes

2) The number of VMs with a fixed data set size using 1, 2, 3 and 4 VMs running 8 processes in total

Like in the previous study, I used a dataset size of 1 million with 5 clusters. I ran the program using different number of VMs but running a total of 8 processes across all the machines. Figure 2 shows the results for this study. Evidently 1 VM running K Means took the longest as there were 8 processes running on just one machine. There was then a significant decline in execution time and the lowest time taken was when 3 VMs with 8 processes were used. This could be because the data size was an optimal size to be split across the 8 process on the 3 machines, allowed better parallelism and execution time. Contrastingly, when 4 VMs were used, the time taken increased slightly, due to there being more communication between the master and slave machines.

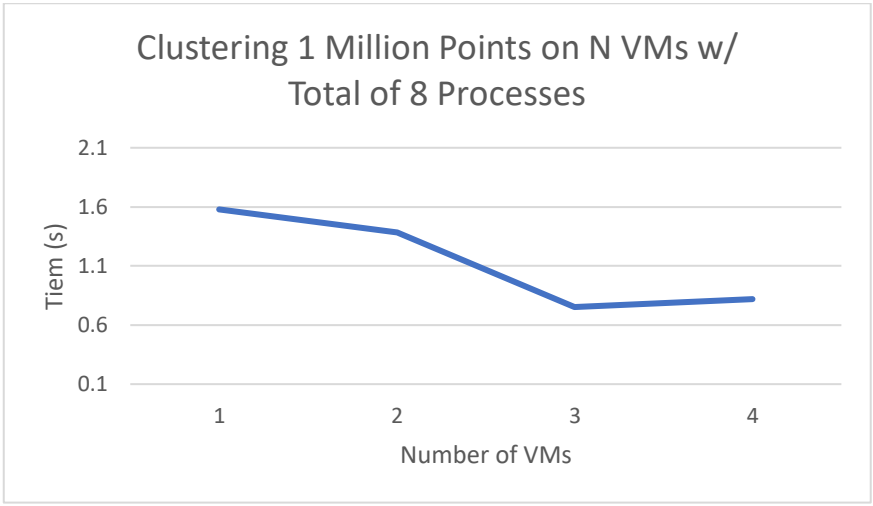

Figure 2 – Clustering 1 million points using multiple VMs

3) The number of data points in the data set of 2D points with a specific number of processes and a fixed number of VMs using 20 million, 30 million and 40 million data points

In this study, I ran my program to cluster 20 million, 30 million and 40 million data points with 5 clusters each and ran the program on 4 VM's running a total of 8 processes across

the machines. Figure 3 shows the result of this study. This result that were calculated show linear relationship between the number of points and the time it took to cluster them. This is sort of intuitive, in the sense that the more number of points there are to cluster, the greater time it takes

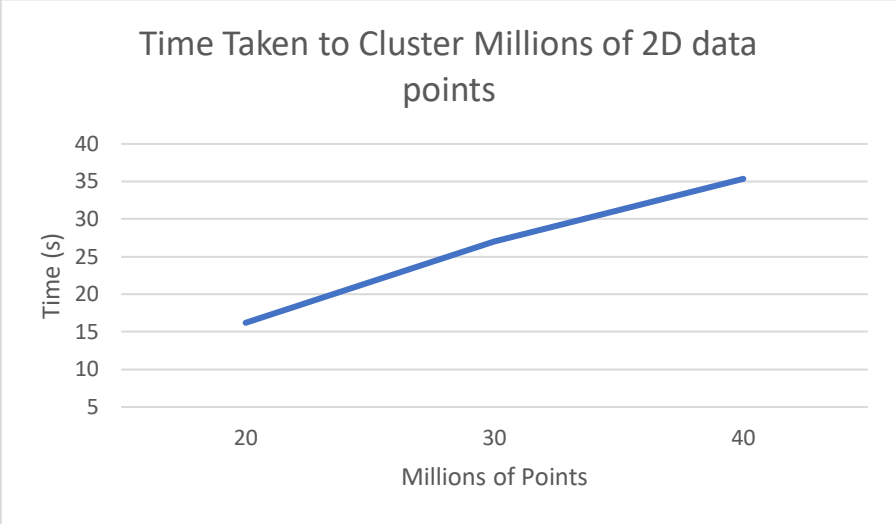

Figure 3 – Clustering millions of points using 8 processes on 4 VMs

Discussion

The experiments that I conducted display various term in terms of the execution time of the MPI version of the implementation. I will now further discuss these results in this section

Experience on applying MPI to the K Means clustering

In my opinion, applying MPI, was obviously much more complicated than doing a straightforward sequential implementation of the clustering as we have to split the tasks in terms of what the master does and what the slave does, then communicate the data and compute the centroids at only the master – the development effort is much greater. We have to ensure that there are no race-conditions, that the points are evenly distributed among all processes and that the master and slaves are receiving the correct data and so on. All of these issues are not present in the sequential implementation.

Performance trade-offs of MPI and sequential programming with K Means

One thing that I clearly noticed was when I compared the clustering times for MPI and for sequential, the MPI version was much faster since its parallelized with multiple processes running the clustering algorithm. I noticed that for clustering 40 million points with 10 iterations, it took 39 seconds for the MPI version but took 57 seconds for the sequential version. So, the parallelism really helped speed up the clustering process. But there is a trade – off in terms of the number of processes that is used in the MPI version. It was seen when 1 million points were clustered, with 8 processes on 3 VMs took around 0.7 seconds whereas with 8 processes on 4 the same clustering took around the 0.9 seconds. It is not a big difference, but with a larger data set size, a significant different could be seen. I also noticed that as the number of processes increased across the 4VMs, the time taken to cluster increased after around 12 processes were used and I believe that this is because

there is much more communication that the master is doing with all the other processes, which incurs more time.

Applicability of K Means to MPI

In my opinion implementing K Means with MPI is good when you have a large data (in the order of millions) and the program needs to run for multiple iterations. These factors really exploit the parallelism of MPI and make the program much faster than a sequential implementation Although the MPI version takes an extra bit of time at the start to distribute the points, this time is offset by the fast performance of the clustering itself. However, if your dataset size is smaller, then a sequential implementation would work better, since the small number of points, wouldn't really justify having multiple machines to do the clustering.

Recommendations regarding the usage of MPI for algorithms similar to K Means

As mentions previously, it would be better to use MPI when there is a large dataset and when you need to process that data a large number of times because with the iterative manner of such clustering algorithms can we truly take advantage of the parallelism provided