Andrew ID - sjan

Characterizing MapReduce Task Parallelism using K-Means

Introduction

This project centers around using Hadoop MapReduce to implement the K-Means clustering algorithm on 2D data and on DNA strands. As mentioned in the previous project, 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 and K-Means does exactly this. What I have tried to achieve in this project is to apply MapReduce on a real - life problem, then characterize the map phase parallelism by using the K Means clustering algorithm and then to compare my results from MPI and the Hadoop MapReduce implementations of K – Means and see how each perform. In order to do this, I conducted various experimental analysis to measure the efficacy of the Hadoop 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, MPI, Hadoop – Development

For all the implementation, the algorithm of computing the means was exactly the same as I explained in the Project 3 report. The only difference is in how the data was received in each implementation. For the sequential implementation, I had to explicitly read all the points into an array and find the centroids normally using the clustering algorithm. With MPI I had to again read all the points into an array and then split the points across all the slaves. And each machine in MPI would calculate its sums for the x and y coordinates and the number of points belonging to one centroids and sends it to the master who will then compute the new centroids. With Hadoop, things are different. Hadoop classifies two functions, Map and Reduce. Each of these functions take in (key, value) pairs as their input and emit (key, value) pairs as well. In the map function we deal with only with one point at the time and the defined, calculates the closest centroid to that point and emits a (key, value) pair of (centroid, point). I defined a class called Coord that implements the WritableComparable Class in Java to represent the 2D points. The reduce function then takes in the input from the map class and computes the new centroids. Each reduce function is given all the values for one key – meaning all points allocated to one centroid are fed to one reduce task. The function then calculates the sum of x and y coordinates along with the number of points allocated to that centroid and emits the following (key, value) pair – (old centroid + new centroid, count). In the main we then check for convergence by comparing the old and the new centroids and if it has converged, iterations stop, otherwise we do the mapreduce job once more.

Andrew ID - sjan

In terms of development, the Hadoop was the easiest to develop as we only needed to be concerned with working with one point at a time and we primarily only needed to write 2 functions, map and reduce. This made the implementation much easier. One of the things I struggled with however was trying to understand the Hadoop library as it was very confusing at first.

2D Points Sequential, MPI and Hadoop – Performance

What I found in terms of performance was that the MPI version of the implementation was faster than both the sequential and Hadoop version of the implementation. I was not surprised that the Hadoop version took a lot longer to work since each map reduce job requires Hadoop to fetch the points from disk which causes the execution to run for a much longer time. Another thing in Hadoop is that the reduce tasks cannot start until a percentage of the map tasks have been completed. But in the case of MPI, the machines can do their local computations parallely without waiting for any other machine and it is only at the end of their computation that they send their data to the master – so there is greater parallelism in terms of computation of the centroids in MPI.

DNA Sequential, MPI and Hadoop – Development

The same idea that was used to cluster the 2D data points was then extended to cluster the DNA strands. For the Hadoop version, I used the Text data type to represent the DNA strands and used the exact same method of calculating the maximum occurring base at each index of the DNA strand to calculate the new centroid. In terms of development, again, the Hadoop version was much easier to develop as I only needed to write 2 functions – map and reduce. The map function emits (centroid, point) where the closest centroid to the point is defined as the key. The reduce function then calculates the new centroids by the above-mentioned method.

DNA Strands Sequential, MPI and Hadoop – Performance

In terms of performance, the results were similar to what was noticed in the 2D points implementation. The MPI version, again, performed much better compared to the Hadoop and sequential implementations.

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

In order to test the performance of my Hadoop implementation, I performed 3 scalability studies on my implementation. They are as follows:

1) Using different number of VMS on fixed data set size with HDFS block size of 64MB.

For this text I used a data set size of 20 million points with 5 clusters and ran the program under 3 iterations with 2 map slots on each VM. Here we can see that as the number of VMs the job is run on increases the faster the job takes to complete. These results are valid as

the more machines you have the greater parallelism there is. It can be seen that the sweet spot for the job is when 3VMs were used and this is because of the fact that the VMs were fully utilized and there were no idle map slots or machines.

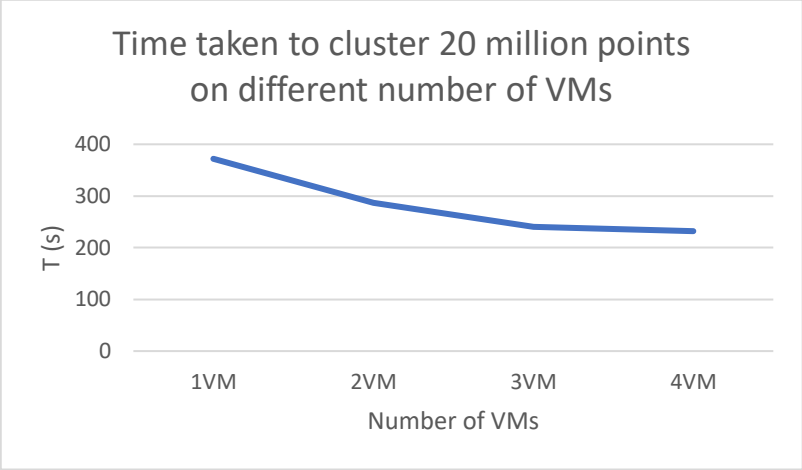

Figure 1 - Clustering 20 million points using different number of Map Slots

2) Using different number of HDFS block sizes, particularly, 32MB, 64MB and 128MB on a data set size of 20 million

Like in the previous study, I used a dataset size of 20 million with 5 clusters and ran the program for 3 iterations. Each map task had the default 2 map slots running on the VM. So, we can see here that when the program was run with an HDFS block size of 32MB, the execution time was the largest when the block size was 64MB as seen from the graph. After doing some calculations I found that 12 map task are launched given the HDFS block size. This means there will be a total of 2 waves with full utilization of the VMs which doesn't happen in any of the other cases. In the case an HDFS block size of 32MB there are 3 waves and in the case of 128MB there is only one wave but there is underutilization of the VMs.

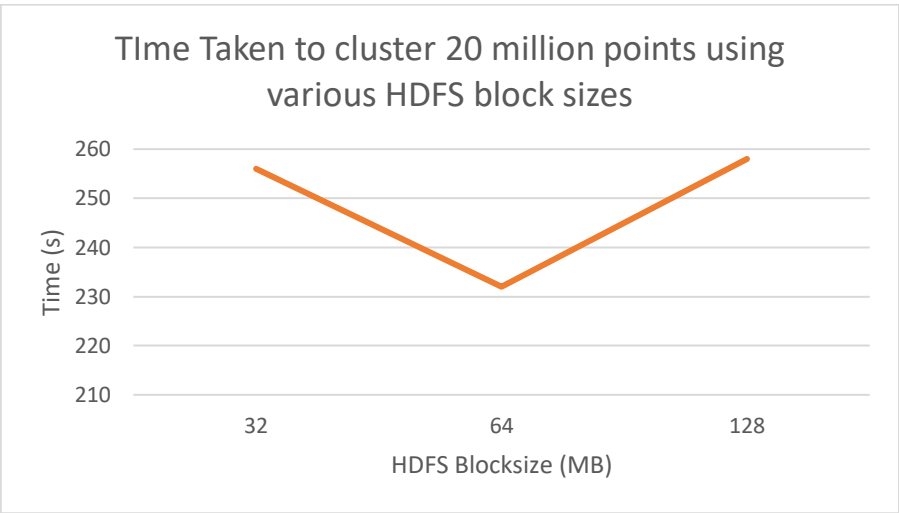

Figure 2 - Clustering 20 million points using different HDFS block sizes

3) Using different number of map slots one 4VMs with block size of 64MB on a data set size of 20 million

In this study, I used a data set size of 20 million points with 5 clusters and the program for 3 iterations. However, on each run I changed the number of map slots on the VMs. I ran the job under 1, 2 and 4 map slots. Here we can see a clear trend that as the number of map slots increase, the execution time decreases, which is expected as there is greater parallelism. We can see that there is a sharp change in time between the execution for when 1 map slot was used for when 2 map slots are used. This could show that when 2 map slots are used there is much greater parallelism as 2 map tasks are run simultaneously on each VM. However, there is not much of a decrease in execution time when 4 map slots were used. This is probably because the VMs only have 2 cores, means that 2 tasks are run on each core and the keep context switching between each other causing the execution time to be not be significantly less.

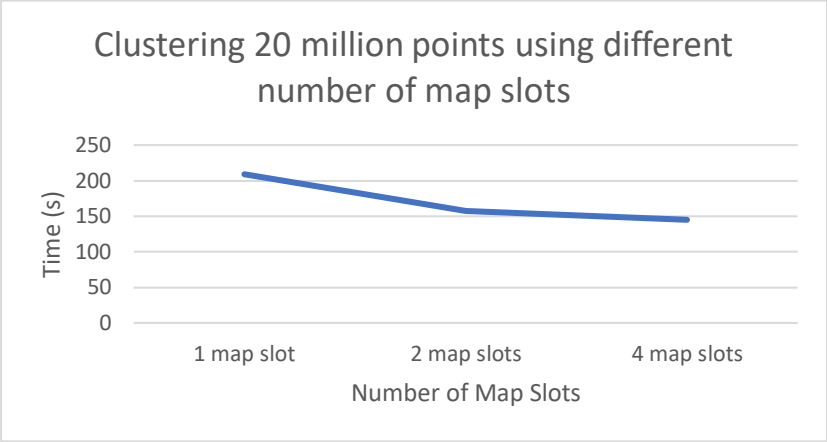

Figure 3 - Clustering 20 million using different number of map slots

Discussion

The experiments that I conducted display various trends of the execution time of the Hadoop version of the K Means clustering algorithm. I will now further discuss these results in this section.

Experience on applying MapReduce to the K Means clustering algorithm

Personally, MapReduce was far more easier to code than both the sequential and MPI version since in MapReduce we don't explicitly need to split our data between machines – it all done by Hadoop. What even more convenient was that all the underlying communication between machines was also done by Hadoop, making things straightforward. Also as programming since we don't have to worry about synchronization, we can code our program as we are writing a sequential program. MapReduce solved all the problems I was facing in the sequential and MPI implementation in terms distribution of work load to the machines.

Andrew ID - sjan

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

Comparing both sequential and MPI to MapReduce, I found that MapReduce was definitely much slower than more both of the former implementations. Out of the 3, MPI was the fastest of all the 3 implementations. The reason that MapReduce is so slow is that whenever it fetches its data, it fetches it from the machine's disk and disk access is the slowest out all the data accesses. If I were try the experiment again, I would try it with a larger data set size and see whether it offsets the time for fetching from, but I assume that that it wouldn't make a difference. Also, I saw that the optimum parameters to run the job for 20 million points was using 3 VMs with 2 map slots on each VM with a 64MB HDFS block size.

Applicability of K Means to MapReduce

In my opinion, K Means is really with map reduce in terms of the development effort. I was able to program the MapReduce implementation in have the time I took to program the sequential and MPI versions. Hadoop does a lot ground work for the program and allows then to have a lot of flexibility in terms of the job configurations. However, if we're talking in terms of performance, MapReduce didn't work as well. In MapReduce, you generally don't perform iterative jobs, which is what we're doing here since the overhead of fetching from disk causes the execution time to increase. So, MapReduce works much better when you only need to run the job once.

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

As mentioned previously, I would use MapReduce if I'm looking for ease of programming as you practically only need to write 2 functions. But if execution time is something that needs to be considered, I would not recommend using Hadoop MapReduce due to the fetching of data from the disk and since an algorithm like K Means is iterative, there will be many disk access which cause the execution time to be longer.