Distributed Systems Lab 4

Evan Palmer - esp at andrew
Titouan Rigoudy - trigoudy at andrew

December 5, 2014

1 Parallelization of K-Means

Our parallelization method had a single coordinator, and many worker nodes.

The worker nodes were each assigned a partition of the data. These workers received the current means from the coordinator nodes. They then looked at their portion of the data, decided which mean was closest to each point. For each mean, the worker node calculated the sum of the X and Y values of points which were closest to that mean. Once this calculation completed, they reported back to the coordinator node with the sum of coordinates of points belonging to each mean, and the number of points in these sums.

The coordinator repeatedly sent out the current k mean candidates to all of the worker nodes, and then waited for the sums and counts. Once all sums and counts were received, the coordinator node computed the new mean candidates by combining the sums and dividing by the total number of points which belonged to each mean. Since the number of computations performed by the coordinator depended only on the number of means, its workload was relatively low.

This method of parallelization has the advantage that as long as the nodes can communicate sufficiently quickly, adding more nodes to the cluster will decrease the number of points assigned to each worker node, and therefore, the runtime of the algorithm.

1.1 Pseudo Code for Coordinator Node

```
Points, NumPoints, NumClusters, Iterations, Machines <- GetInput()

# Divide the points into partitions based on the number of available machines
Partitions <- Partition(Points, Count(Machines))

# Send one partition to each worker. This worker will be responsible for
# Computing the closest mean for each of the points in the partition.
AssignPartitonsToWorkers(Partitions, Machines)

# Randomly select NumClusters mean values to start with
Means <- RandomlySelectK(Points, NumClusters)

Repeat Iterations Times
    # Send the current means to each worker
    For Worker in Workers
        Send(Worker, Means)
    # Receive the sums and counts vectors from each worker
    For Worker in Worker
        MeanSums, CountSums += Receive(Worker)

    # After combining all the MeanSums and CountSums,
    # Element wise divide the vectors to get the averages
    For Mean, MeanSum, CountSum in Means, MeanSums, CountSums
        Mean = MeanSum / CountSum

# Finally output the means
Output(Means)
```

1.2 Pseudo Code for Worker Node

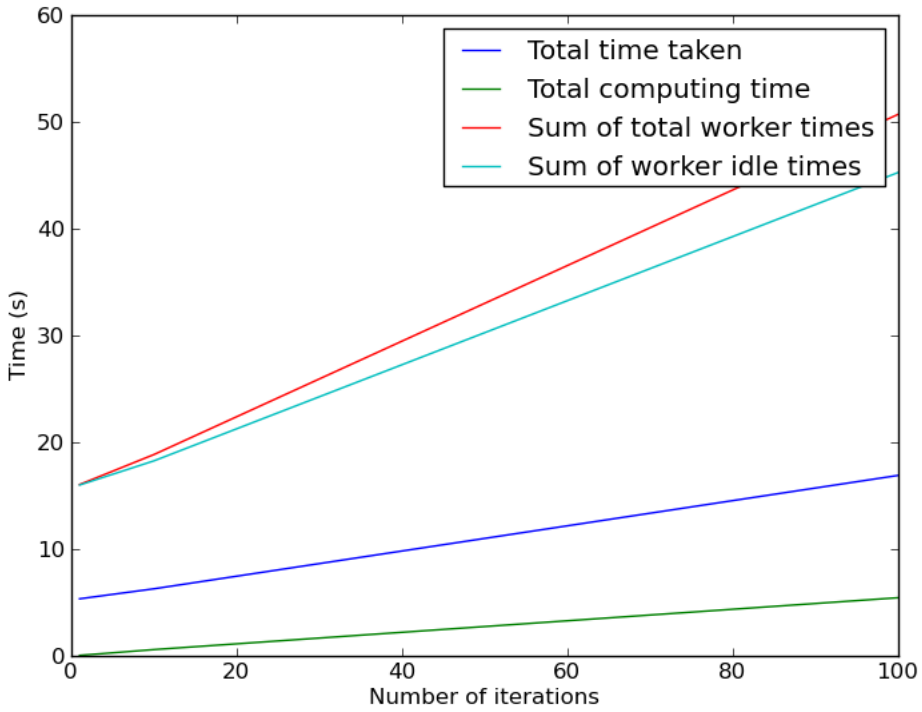
```
PointPartition <- Receive(Coordinator)

While Not Done

    # The coordinator will tell us when we are done
    If Coordinator Says Exit
        Exit

    # Get the current means from the coordinator
    CurrentMeans <- Receive(Coordinator)

    # Iterate over each point which belongs to our partition
    For Point in PointPartition
        # Find the mean which this point is closest to
        ClosestMean = FindClosestMean(CurrentMeans, Point)
        # Add that point to the mean sum for the mean it is closes to
        MeanSums[ClosestMean] += Point
        # Increment the number of points for the mean the point was closes to
        MeanCount[ClosestMean] += 1

    # Send the coordinator the mean sums and mean counts for our partition
    Send(Coordinator, MeanSums, MeanCounts)
```

1.3 Comparison of DNA Sequences

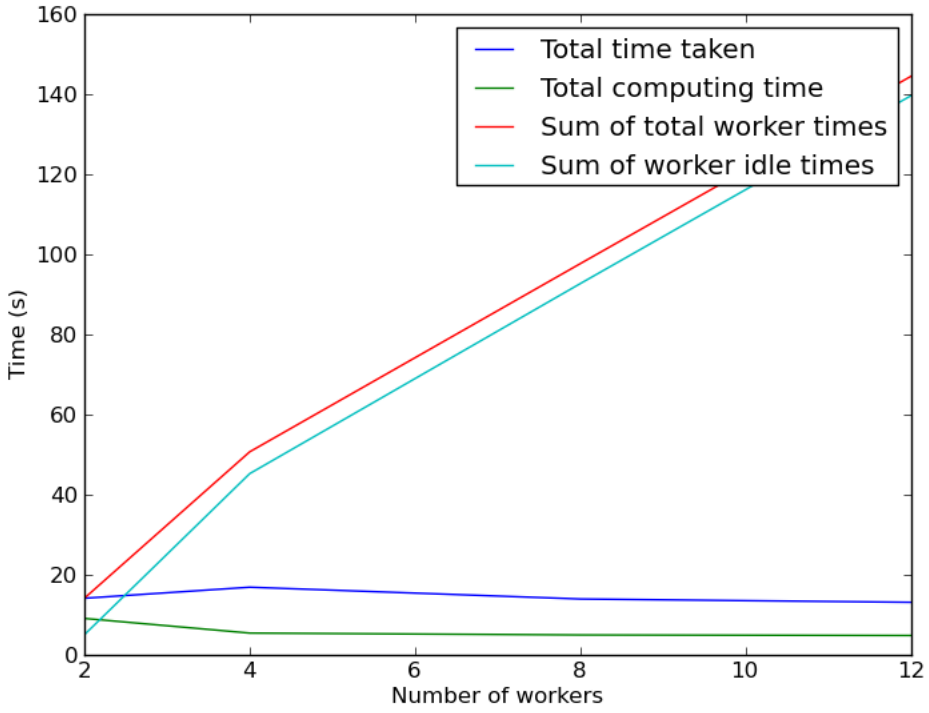
Our k-means algorithm for DNA Sequences was very similar to our algorithm for points. However, instead of using squared error to measure similarity, we used a boolean dot product. Also, since there is not a meaningful way to report the average of several letters, we instead reported the counts of all possible values [T,C,A,G] at each position. On the coordinator node, we summed these counts, and then for each position, chose the most common value for the mean at that position.

2 Performance

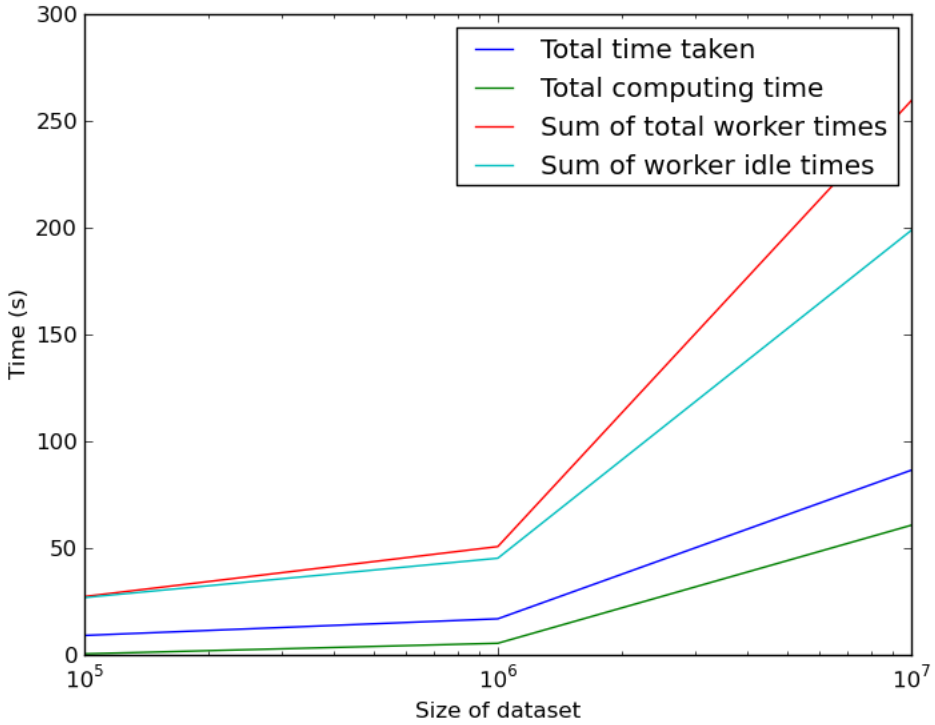
2.1 Networked Performance



Unfortunately our performance on the network was heavily influenced by communication time. As is shown in the graph below. For small numbers of iterations, the processes spend almost all of their time communicating. This reflects the initialization cost of our parallel system. The data must be distributed to the workers who will then do computations on it. For small amounts of data, it would be more efficient to do all the computation on one machine and avoid the network entirely. However as number of iterations grows larger, the workers spend less of their time getting the data, and more time doing computations on it.



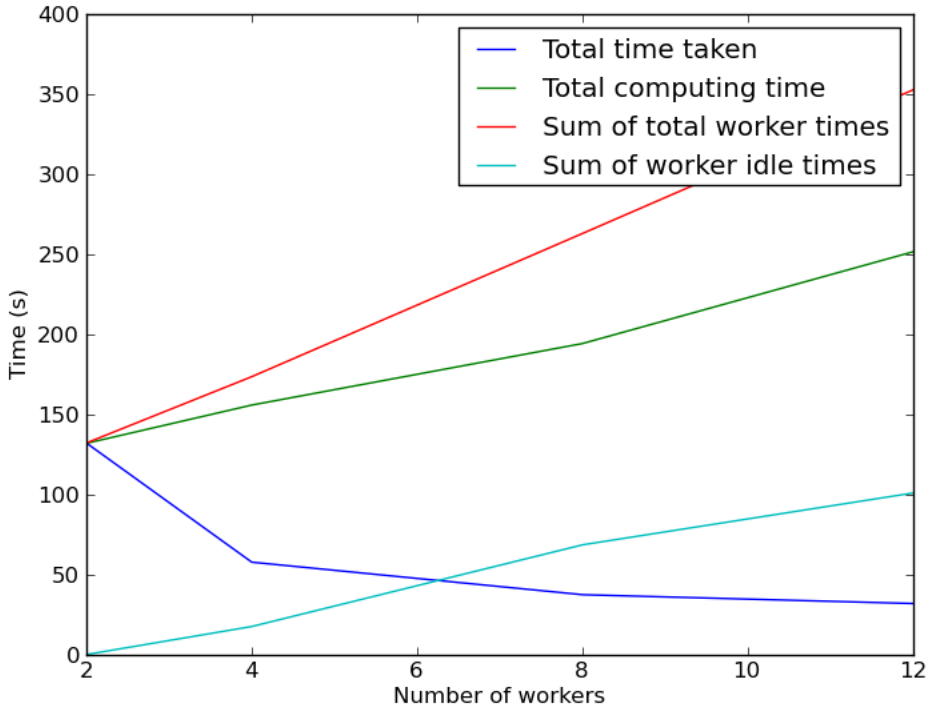
Distributing the data across many networked nodes did not help significantly. Too much time was spent communicating, and there was essentially no performance speedup. We were not able to generate a dataset large enough to really see the benefit of parallelization over the network.



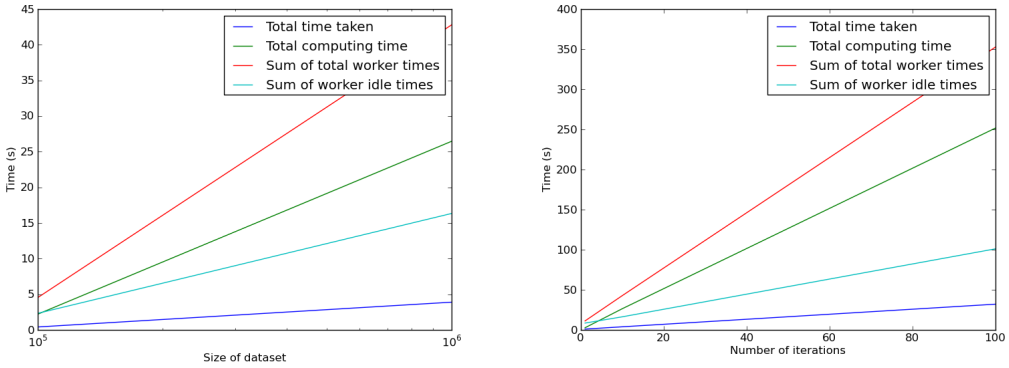
As can be seen in the figure below, as the dataset gets larger, parallelization becomes more effective. With the two smaller datasets, almost none of the running time is spent computing things, and the workers are nearly always idle. This suggests that they are spending all of their time waiting for information to be sent across the network. When the dataset gets larger, we see that more of the time is spent doing actual computation, and the workers spend much less of their time idle. We hypothesize that on large datasets the efficiency introduced by parallelism, even over the network, would overcome the time wasted in data transmission.

2.2 Local Performance

Since the network performance was not as strong as we hoped, we also ran the processes locally.



Here, with a large dataset (100,000 DNA strands), and local parallelism, we can see that parallelizing the computation significantly speeds up the process. There is still an overhead incurred, but here it is clearly worth running on at least four processors (the number on the machine) and it seems to still improve even after this mark. We do see that as the number of processes increases, even locally, we pay a noticeable overhead.



As expected, on the local machine, performance was linear in both the number of iterations, and the size of the dataset.

2.3 Baseline

Given below is the baseline performance in seconds of sequential DNA k-means on a cluster machine, without using MPI at all.

Number of iterations \ Dataset size	100000	1000000
1	0.2	1.9
10	1.6	16
100	17	174

3 Data Set Generators

3.1 Points Generator

We used a very slightly modified version of the point generator provided.

3.2 DNA Sequence Generator

Our DNA sequence generator generated clusters through the following method.

First we generated k random DNA sequences - One for each desired cluster. For each one of these parent sequences, we proceeded to generate children. Children are generated by iterating over all bases of the parent, and, for each base, either keeping the base with probability p, or replacing it with a random base with probability $1 - p$. The parameter p controls the tightness of the cluster, with a p close to one producing tight distributions.

4 Building and Running

4.1 Building

First build our project!

1. Untar handin.tar into an empty directory.
2. Build the executables with `$ make`
3. Build the test datasets with `$ make test`

There should now be four executables in your current directory:

`points_seq` - The sequential implementation of k-means on points
`points_mpi` - The parallelized implementation of k-means on points
`dna_seq` - The sequential implementation of k-means on dna
`dna_mpi` - The parallelized implementation of k-means on dna

There will also be many datasets: `points_100000.dat`, `points_1000000,dat`, `points_10000000.dat`, `dna_100000.dat`, `dna_1000000.dat`.

If you would like to experiment with other datasets, you can use our dataset generators.

In `src/gen`, you will find two files:

`dna_gen.py` - A generator for dna datasets
`points_gen.py` - A generator for points datasets

These can be run as follows:

```
$ python3 dna_gen.py
$ python2 points_gen.py
```

Please consult the help messages for usage.

4.2 Running

Running k-means:

All of the executables take the same arguments, as shown below. Note that the DNA executables should only be run with the DNA datasets, and the points executables with the points datasets.

```
./executable -d DATASET -o OUTFILE -i ITERATIONS -c CLUSTERS
   -d DATASET      The dataset to run k-means on
   -o OUTFILE      The file to write the output of k-means to
   -i ITERATIONS   The number of iterations of k-means to run
   -c CLUSTERS     The number of clusters to attempt to find
```

You can also use the `run.sh` script to run our programs on the datasets, remotely or on the local machine, like so:

```
$ ./run.sh (remote|local) (dna|points)
```