Alex Cappiello (acappiel)
Michael Ryan (mer1)

15-440 Project 4: K-Means Clustering with MPI

Parallel Approach

We took a straightforward approach to parallelizing the algorithm. The master node (with rank 0) reads the input data, then randomly selects initial centroids and buckets the data with the closest centroid. It broadcasts the assignments and datapoints to each other node to begin the K-Means algorithm. For the given number of iterations, each node first groups its datapoints into buckets corresponding to the closest centroid. It then updates each other node with the points that are now closest to that node's centroid. After this round of communications, we recalculate centroids and update each other node about our new means. The process then repeats for the specified number of iterations.

Pseudocode:

```
rank = MPI.getRank()

if (rank==0):
      datapoints = readDataFromFile();
      centroids = sample(dataPoints, numClusters);
      data = {centroids : closestPoints}
      bcast(centroids, points)

myCentroids = [centroids[i] for i in rank:comm.size:numClusters]
myDataPoints = [points[i] for i in rank:comm.size:numClusters]
allCentroids = {x[0]: [] for x in data}

for iter in numIters:

      for point in myDataPoints:
            allCentroids[closestMean(point, allCentroids)].append(point);

      for cent in allCentroids:
            if cent not in myCentroids:
                  mpi.send(allCentroids[cent]);
            else:
                  myDataPoints[cent] = allCentroids[cent]

      for cent in myCentroids:
            mpi.recv(updates)
            myDataPoints[cent].append(updates)

      newCentroids = [mean(myDataPoints[cent]) for cent in myCentroids)]
      mpi.send(newCentroids)

      for node in nodes:
            mpi.recv(updates)
            newCentroids.append(updates)
```

Usage Instructions

Data generation:

Generate data points with one of the following Python scripts:

./generatePoints.py <required args> [optional args]

-c <#>	Number of clusters to generate
-p <#>	Number of points per cluster
-o <file>	Data output location
-v [#]	Ceiling on coordinate values

./generateDNA.py <required args>

-n <#>	Number of strands to generate
-l <#>	Length per strand
-o <file>	Data output location

Each script writes to an output CSV file with one point of data per line. These are interpreted by the K-Means scripts.

Sequential K-Means:

./kmeansPoints.py <required args>

-c <#>	Number of clusters to generate
-p <#>	Number of iterations
-o <file>	Data output location
-i <file>	Data input location

./kmeansDNA.py <required args>

-c <#>	Number of clusters to generate
-p <#>	Number of iterations
-o <file>	Data output location
-i <file>	Data input location

Distributed K-Means:

./pointsDist.py <required args>

-c <#>	Number of clusters to generate
-p <#>	Number of iterations
-o <file>	Data output location
-i <file>	Data input location

./dnaDist.py <required args>

-c <#>	Number of clusters to generate
-p <#>	Number of iterations
-o <file>	Data output location
-i <file>	Data input location

To run with mpi, run the following:

mpirun -np <numNodes> -hostfile <hostFile> ./pointsDist.py <required args>

mpirun -np <numNodes> -hostfile <hostFile> ./dnaDist.py <required args>

Do not run the program with more nodes than clusters.

We were not able to successfully connect to remote GHC hosts, but the script executes correctly on a local machine.

Experimentation and Analysis

Parallelizing the K-Means clustering produced noticeable speedup on large enough datasets. Below a certain threshold, the network overhead meant that the distributed program was not faster, but above the threshold there are noticeable gains in speed.

Our inability to run on multiple GHC machines limited our ability to test effectively at large numbers of nodes. However, a performance difference is apparent even at only 4 processes. Here is a chart of script performance for various load levels and number of processors.

Points

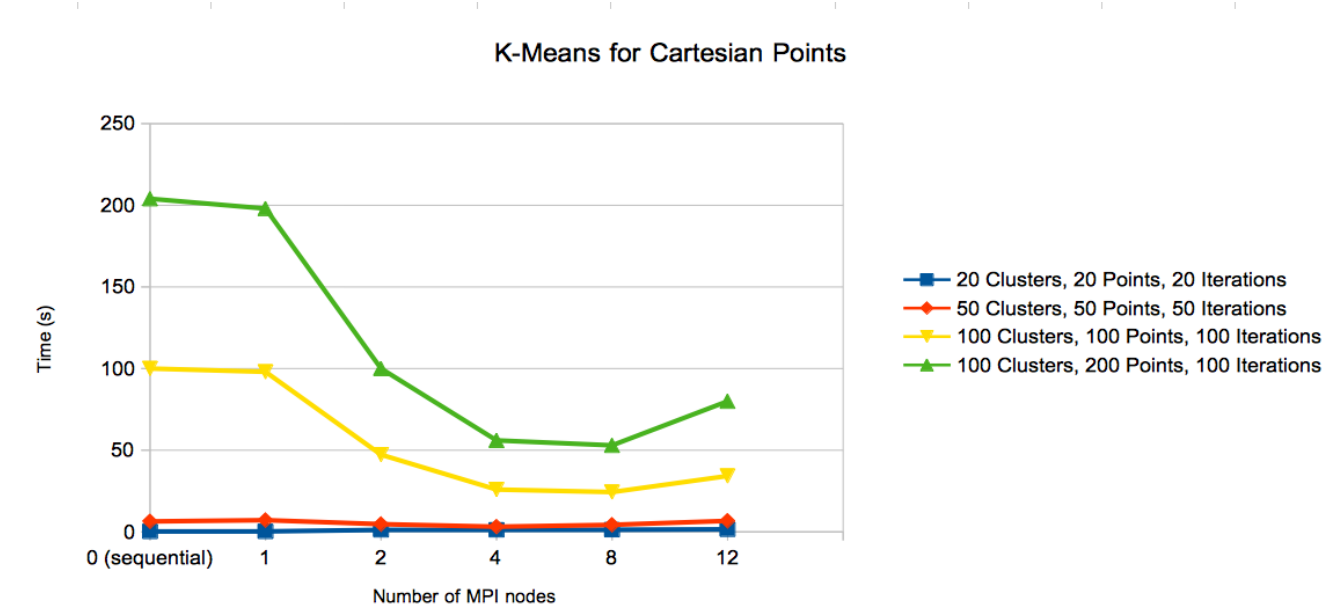
Clusters	Points Per	nIter	Sequential	MPI1	MPI2	MPI4	MPI8	MPI12
20	20	20	0.240	0.270	1.193	1.221	1.292	1.534
50	50	50	6.377	7.129	4.697	3.154	4.299	6.742
100	100	100	1:39	1:38	47.234	25.92	24.34	34.234
100	200	100	3:24	3:18	1:40	56.00	53.00	1:20

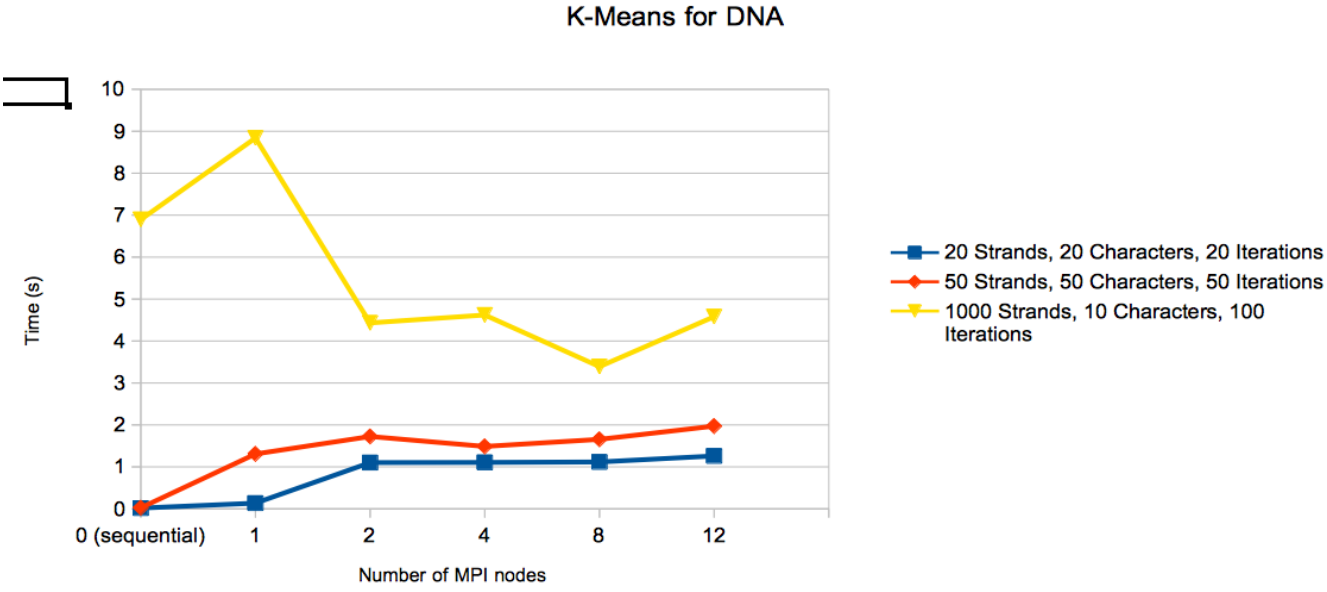
DNA

Strands	Length	nIter	Sequential	MPI1	MPI2	MPI4	MPI8	MPI12
20	20	20	0.018	0.136	1.102	1.106	1.117	1.260
50	50	50	.024	1.308	1.725	1.487	1.656	1.972
1000	10	100	6.896	8.840	4.432	4.62	3.39	4.58

It is apparent in the numbers for two nodes on small datasets that MPI's network traffic imposes a significant performance penalty initially, but this is mitigated by the parallelization that takes place for larger datasets. Also note that the slowdown for 12 nodes is a result of running on a single machine, not in a distributed manner. The load of 12 processes on a 2 or 4 core system in addition to the extra communications is substantial.

It is worth noting that though DNA processing was very fast overall, it required a larger sample size to actually make gains from parallelization.


K-Means for Cartesian Points
250
200
150
100
50
0
Time (s)
0 (sequential)
1
2
4
8
12
Number of MPI nodes
20 Clusters, 20 Points, 20 Iterations
50 Clusters, 50 Points, 50 Iterations
100 Clusters, 100 Points, 100 Iterations
100 Clusters, 200 Points, 100 Iterations


K-Means for DNA
10
9
8
7
6
5
4
3
2
1
0
Time (s)
0 (sequential)
1
2
4
8
12
Number of MPI nodes
20 Strands, 20 Characters, 20 Iterations
50 Strands, 50 Characters, 50 Iterations
1000 Strands, 10 Characters, 100 Iterations