Report

sdai, kpayson

**Design Decisions/Clarifications**:

First, we separated the K-Means algorithms for the points and the DNA data sets into separate classes, because they take different arguments.

**General K-Means**:

For both data sets, we first parse all of the input data (this portion is not counted in the runtime). Then, we simply run K-Means as defined. We choose the starting centroids by randomly selecting k points/strands, and using those as initial centroids. Then, we form the clusters, and recompute the centroids, repeating until the centroid does not change by more than some constant factor (explained in more detail below). Because we randomly select the starting centroids, our algorithm is not deterministic, and so can also have variable runtime (will take far less time if we manage to select one centroid from each cluster luckily, but could take a long time if all of our centroids are very close, like in the same cluster).

**Points K-Means Specifications**:

The stopping condition we use for the points is when the centroids move by less than some value epsilon (~.000001), so that we run until we make essentially no improvement in the centroids.

**DNA K-Means Specifications**:

To generate the median (centroid) strand, at each index we simply take the most common character at that index over all strands in that centroid's cluster.

The stopping condition for this algorithm is when every centroid remains in the same place for consecutive iterations (no improvement).

**Parallel Implementation Design**:

The overall parallelism of our algorithm comes from the splitting of the cluster finding (matching each point/strand to its closest centroid) and the centroid realignment (after readjusting clusters, compute the new centroid location). Both of these tasks can be done on arbitrary cores/processes, so long as we use MP to combine the results. So, for cluster matching, we give each node some subset of points/strands, and have it compute the best centroid for each point in the subset it handles. For centroid realignment, we give each node some subset of clusters, and have it compute the new centroid for each cluster. Then, we give the results of these parallel tasks back to the master and have it combine them.

//Ken expand this IE talk about specific stuff like master, message passing etc, also check ^

**Data Generation:**

To generate data sets for the point clusters, we used the provide python code to generate our data sets. We wrote another class for data generation of dna strands. The class takes a bunch of arguments like number of intended clusters, number of points per cluster, and length of strands. Then, it randomly generates centroids, and randomly varies the centroids so that they form clusters of strands.

Also, since our data is generated randomly, the runtimes and results are variable (we could have a bad data set where some clusters are interleaved).

**Experimental Data:**

We recorded runtime as when we start the algorithm (after parsing data), to when the algorithm terminates. We do not include any file i/o as part of the runtime.

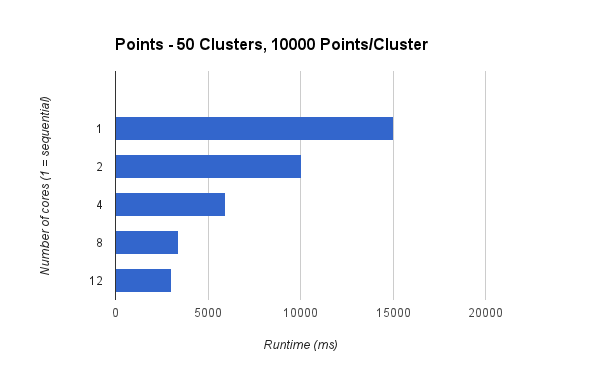
The input data for points was all generated using 2 dimensions, and the data for DNA was generated at a fixed length of 100 (to reduce variability). We fixed these values to have fewer variables, because we did not inherently parallelize the Euclidean distance or dna difference functions. So, the length of the string / dimensions of the point would not cause any difference between sequential and parallel implementations (no parallelism is used regarding the comparison of points/strands).

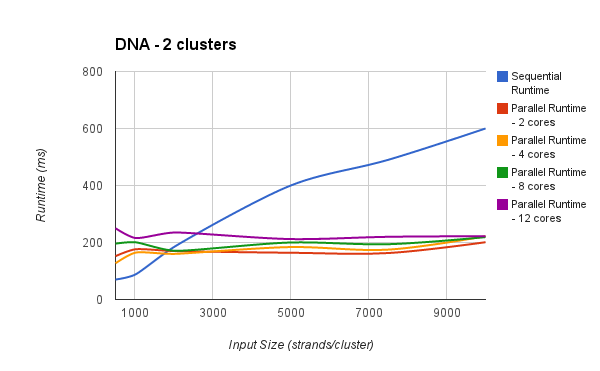
For DNA: Our testing input was generated using intended clusters of 2, 10, and 50 (we check each of these separately). Then, we generate inputs of size (number of points/cluster or strands/cluster) 500, 1000, 2000, 5000, 7500, 10000 for each cluster size. We ran each of these inputs through the sequential implementation, and parallel implementation using 2, 4, 8, and 12 cores.

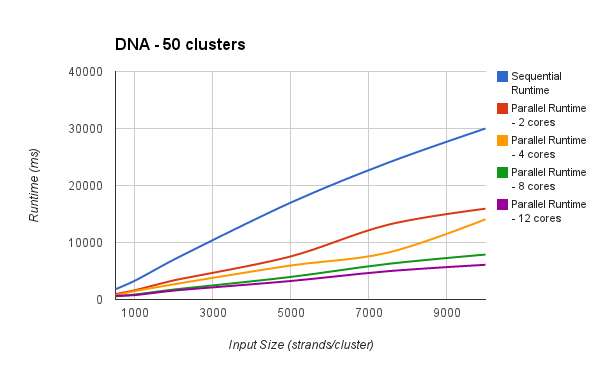
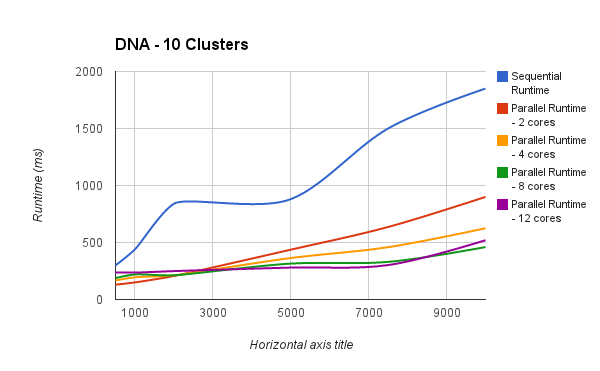
For Points: we ran our points tests on fixed input data of 50 clusters and 10000 points per cluster (500,000 points total). We compared the runtimes for running sequentially, 2, 4, 8, and 12 cores.

The graphs associated with the data collected are below.

**Graphs**:







**Analysis:**

**DNA Results:**

Our algorithms performed about what we expected. In the average case (10 clusters), we can see that the sequential is clearly the worst. When we introduced a second core, the performance improved dramatically, but you can still see that the runtime grows with larger input size. The parallel implementations scaled MUCH better with larger inputs though (the slope is very flat), which is due to the fact that the work is being split up well (only performing a fraction of work on each node).

At least for our inputs, it is notable that the parallel runtimes for the 4, 8, and 12 cores did improve with more cores, but only on larger inputs (you can see the lines intersect at around 10 clusters, 3000 points). This is the point where the processing time becomes longer than the overhead and latency associated with parallelizing the task. As data grows even larger, as we would expect the more cores we have, the faster our algorithm will complete.

An interesting case happens under the scenario of small samples (2 clusters). The sequential version actually performs the best under small amounts of data, which is probably due to the lack of overhead associated with message passing (in this case, the processing time is trivial enough that it does not justify the latency involved with MP). However, as we increase the size of the sample, the sequential implementation’s runtime still increases linearly, so the parallel implementations using 2, 4, 8 cores are better with larger data. Notably, for these small samples, a smaller amount of cores actually has better runtime than running with more cores, which is again probably because the overhead for message passing grows with more nodes/cores.

**Points Results:**

Notice that we gain massive improvement from sequential to parallel implementations, but between parallel implementations using 8 and 12 cores, adding more cores gives us diminishing returns. This is because once we hit a “sweet spot” where the processing task (computation) becomes spread out enough that each processor is finishing reasonably quickly, a lot of the time is spent on overhead.

Overall, however, the parallel implementations were much faster than the sequential version, reducing the computation time by a large factor. In general, we see that the number of cores is inversely proportional to the runtime (more cores = less runtime). In fact, for lower number of cores, we have that doubling the number of cores will approximately halve the runtime.

**Running** **Code**:

To run our data set generator (dna), simply complile the DNAGenerator.java code using javac, then run with:

java DNAGenerator [strandLength] [numClusters] [strandsPerCluster] [outputFile]

To run the sequential versions of K-Means, compile the code with javac, then run with:

Points: java SequentialKMeans [numClusters] [dimensions (2)] [inputFile] [outputFile]

DNA: java SequentialKMeansDNA [numClusters] [inputFile] [outputFile]

To run the parallel versions of K-Means, compile the code with mpijavac, then run with:

Points: mpirun -np [numProcesses] MPIKmeans [numClusters] [dimensions] [numPoints] [numProcesses] [inputFile] [outputFile]

DNA: mpirun -np [numProcesses] ???? [numClusters] [dimensions] [???] [numProcesses] [inputFile] [outputFile]