**Approach to Parallelizing the K-means Algorithm**

1. Select K data points from the dataset to act as the initial centroids.

\*Note that we can’t parallelize centroid selection because our algorithm is as follows:

* + - Randomly select a point to be a centroid
    - Remove from the dataset the (# of points / k) closest points to that centroid
    - Repeat K times

If we were to run centroid selection in parallel by having each machine select a subset of the centroids, one machine could potentially select a centroid that has already been removed from the dataset by another machine. The only way to keep the datasets consistent is by waiting for one machine to finish, and then have it broadcast to the other machines the updated dataset, which is simply the sequential algorithm with more overhead.

1. The master broadcasts the clusters (initially only the centroids selected in the previous step) to each of the machines and scatters the data points evenly among the machines.
2. Each machine then calculates for its data points which cluster centroid they are nearest to and stores the results in a map{cluster, list of datapoints}.
3. On the master machine, gather all the separate maps into one central map and broadcast it to all the machines.
4. Partition the clusters, and scatter them across all machines, so that each machine can use the map to populate the clusters it’s responsible for with the proper points
5. The master then gathers the clusters, updates the centroids for each cluster and determines whether, for all clusters, the variance of each cluster’s centroid is below a given threshold, and if so, broadcasts to the other machines that the algorithm has completed and returns the centroids

**Pseudo-code**

(start at master)

centroids = Choose k initial centroids (list of clusters of points, one point per cluster)

while True

clusters = centroids (list of clusters of points)

Broadcast(clusters)

partitionedpoints = Partition(points)

scatteredpoints = scatter(partitionedpoints)

(done across all machines)

for each point in scatteredpoints

Add point to the cluster with the nearest centroid in the chunkedPtClstrMap{cluster id,list of points}

(done in master)

chunkedPtClstrMap = gather(chunkedPtClstrMap)

for each map m in chunkedPtClstrMap

Iterate through the keys/clusters of m

add values for the key/cluster to the unified map: ptClstrMap

Broadcast(ptClstrMap)

clusterpartitions = Partition(clusters)

scatteredClusters = scatter(clusterpartitions)

(done across all machines)

for each cluster c in scatteredClusters

c.add(ptClstrMap.get(c))

(done in master)

gatheredClusters = gather(scatteredClusters)

maxvar = -infinity

for each centroid c in centroids

Update(c,gatheredClusters[c])

var = recalculateCentroid(c) **recalculateCentroid returns change in centroid**

maxvar = max(maxvar, var)

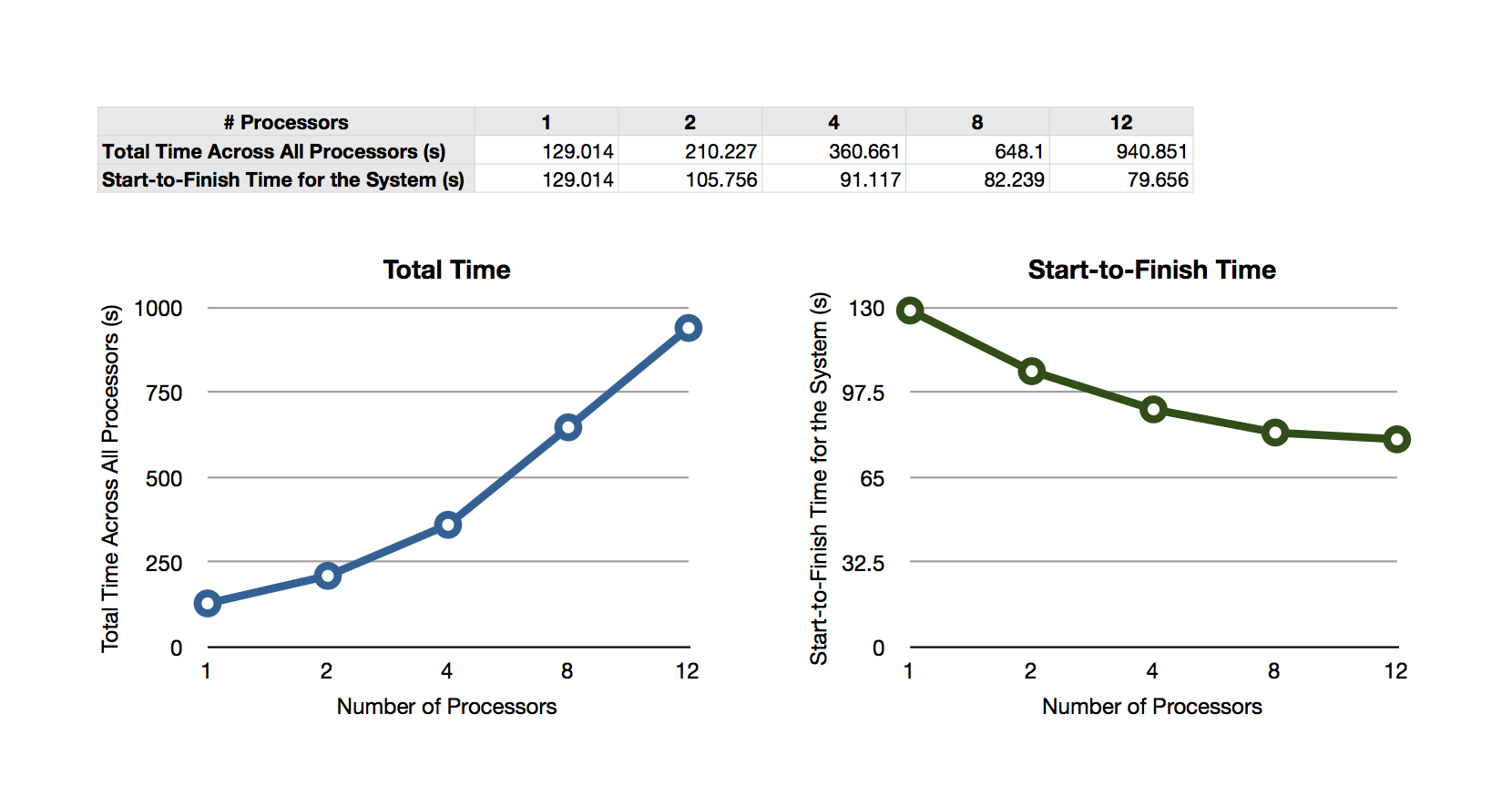
if maxvar < var\_cutoff

done = true

broadcast(done)

return centroids

**Experimentation and Analysis**

In order to evaluate the performance of our sequential and parallel k-means implementations, we ran both implementations with a cutoff variance of .0001 on a fixed data set of 2D data points with 24 clusters, 10,000 points per cluster, and a maximum coordinate value of 1000. We chose to do this with 24 clusters, which is a common multiple of 1,2,4,8, and 12, such that for all numbers of processors we’d be testing, the data could be scattered evenly between processors. In order to measure the runtime we used the cProfile module in Python to run our main method. The results we obtained are depicted below. (Graphs overflowed to next page).

As can be seen, the total time spent across all processors had a linear positive correlation with the number of processors. This is likely due to the increase in calculation and MPI calls required by the parallel implementation as the number of processors used increases. Meanwhile, the start-to-finish time for the system decreased by a large percentage with the addition of the second processor, and slightly less with the addition of the next two processors, even less with the next four, and least with the last for. Clearly there are diminishing returns to our parallelization of the k-means algorithm as we add more and more processors. It would seem that we got the most significant increase in performance for the least number of additional processors with the addition of the second processor and so I would say that two processors would be the “sweet spot” in that sense. We get diminishing marginal returns starting after the second processor is added, marginal return seems to quickly approach zero after the eighth processor is added, where the cost of all the additional calculations and mpi operation costs begin to near the additional benefit of splitting up the data further.