K-means with OpenMPI Analysis

Michael Hansen (mhansen1) & Michael Rosen (mrrosen)
May 5, 2013

K-means Implementation with OpenMPI Overview:

In order to optimize the k-means algorithm given in the lab handout, the summation of the data points for each centroid occurred within the primary loop over all data points and the secondary loop was merely to recalculate the centroids from this data. So, the psuedo code for the algorithm is as follows:

```
Pick $m_i$ to k random $x^d$ (i in [1, k] and $x^d$ in the data set X)

Repeat
for $x^d$ in X: // primary loop
    if $||x^d - m_i|| = \min_j ||x^d - m_j||$
        $m_i$.sum += $x^d$
        $m_i$.members++

for all $m_i$: i in [1, k] // secondary loop
    $m_i$ = $m_i$.sum / $m_i$.members
    $m_i$.members = 0
    $m_i$.sum = 0

Until all $m_i$ converge
```

Thus, by localizing the process of summing the members of each cluster, the second loops does not need to loop over all members in the cluster and the additionally storage required for cluster membership is eliminated (as we only care about the centroids, cluster membership is not something that needs to be remembered, though if we wanted to remember membership, it would not be difficult to all this feature to the current implementation).

In order to utilize the parallelism of OpenMPI, the work of running through all the data points (x^d) is distributed to the system by partitioning the data points between nodes. Thus, each node only reads in the data set on which it will operate from the file. The process of picking random node for the initial centroids is performed on the "primary" node (node with rank 0) and distributed to the others using MPI's Bcast functionality. The work of the primary loop is done by all nodes on their respective partition of the data set and then is gathered (via MPI's Gather). The secondary loop is then executed on the primary node (as distributing this work seems excessive; the number of centroids is relatively very small and the overhead of transferring data is high). Then, the resulting new centroids are distributed to the rest of the system, again using Bcast along with the total number of stable nodes used to decide when the centroids have converged. Once the centroids all converge, the centroids are returned to the user and the program closes the MPI connection. Below is the algorithm with MPI calls added:

```
Pick $m_i$ to k random $x^d$ (i in [1, k] and $x^d$ in the data set X)
Bcast(all  $m_i$)

Repeat
for $x^d$ in X: // primary loop
     if $||x^d - m_i|| = \min_j ||x^d - m_j||$
          $m_i$.sum += $x^d$
          $m_i$.members++
Gather(all  $m_i$)

for all $m_i$: i in [1, k] // secondary loop
     $m_i$ =  $m_i$.sum /  $m_i$.members
     $m_i$.members = 0
     $m_i$.sum = 0
Bcast(all  $m_i$)
Bcast(number of converged $m_i$)

Until all $m_i$ converge
```

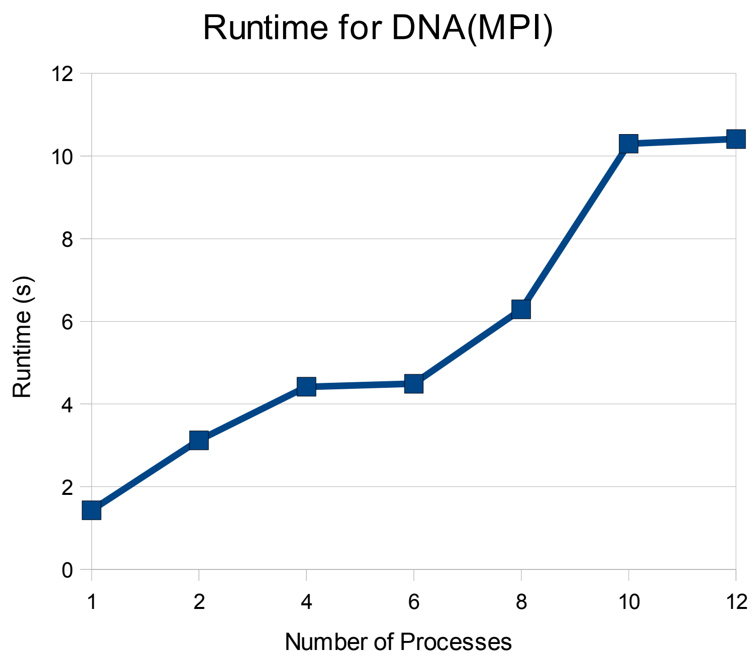
Performance Evaluation:

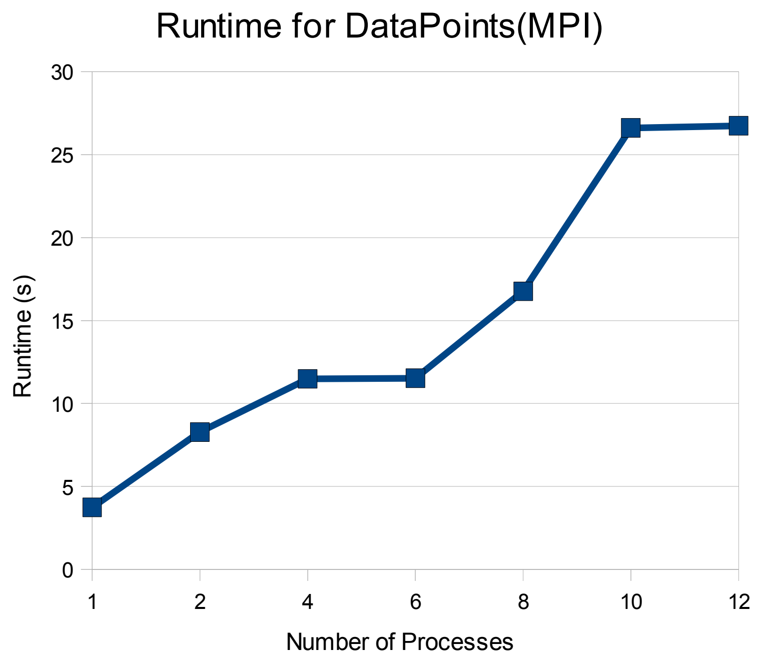
Number of Nodes	**Execution Time (DataPoints)**	**Execution Time (DNA)**
Non-MPI (Serial)	3.74	1.43
2	8.28	3.12
4	11.49	4.42
6	11.52	4.49
8	16.76	6.29
10	26.61	10.29
12	26.74	10.41

Note that the execution time was averaged over 6 runs of each command (using UNIX time command) and that the following parameters were used:

DataPoints:
Input File: clustersbig.csv
Centroids: 4
Seed: 15440

DNA:
Input File: DNAStrands.txt
Centroids: 20
Seed: 15440





Note that the first point (1 process) is the serial code's runtime (not -np 1 for mpirun). While runtime increases monotonically, the number of nodes (GHC machines) that functioned properly for OpenMPI with Java was limited to 3. Thus, the runtimes for 4+ processes had more than one process running on each node, slowing down execution time significantly. When the machinefile contained 3 nodes and 3 processes were run, the execution time was less than running 2 processes on 2 nodes (about 8.2s versus 8.35s at that time). When the number of nodes was decreased, 3 processes on 2 nodes took much longer (11.5s at that time). Thus, as these execution times are based on only 3 nodes , they might not accurately reflect the trend that would occur with x nodes (x being the number of processes). Furthermore, as these tests were run with three nodes and the execution times jumped with each multiple of 3, it seems the execution time is related to the number of processes per node. While the trend suggests increased parallelism does not aid execution time, with the proper number of nodes, increased parallelism might improve execution time significantly.