15-640 Distributed System Lab4 Report

Name: Zhengxiong Zhang
AndrewID: zhengxiz

i. compile and run.

Makefile:

1. **change the parameters** at the top of the Makefile.
2. **make compile**: compile the code.
3. **make generate**: generate the raw point data and dna data.
4. **make runpointp**: run point data kmeans with parallelizing algorithm.
5. **make runpoints**: run point data kmeans with sequential algorithm.
6. **make rundnap**: run dna data kmeans with parallelizing algorithm.
7. **make rundnas**: run dna data kmeans with sequential algorithm.

ii. describes your approach to parallelizing the algorithm, including pseudocode.

1. Point Data

a. First, user decide the number of the clusters (k), and input it into the program through arguments.
b. Initialize. Read the data file into the memory. Randomly select k centroid points.

Repeat:

c. Master broadcast the centroid points. Slaves receive the list of the centroid points.

d. All processes compute a part of the data, and get the group number.
e. All processes compute the sum and the count of points in the group determined by each centroid points. Slaves send the centroid points to master. Master receive the information.
f. Master merges the centroid points, e.g. sums the semi-sum and count up. Then, we can compute the new centroid points, using the sum divided by the count.
g. Repeat c to f until converge.

Master:

```
-Initialize $m_i$ to k random $x^d$, for i = 1, …, k and $x^d \in X$ that contains each of our d-dimension data point.
-Repeat
-Broadcast the mi (initial centroid points) to all the slaves;
-For $x^d$ in $X_j$, for $j = 1, ..., NumOfProcesses$ and
$X_j = X[Size / NumOfProcesses * myrank : Size / NumOfProcesses * (myrank + 1)]$
        -bid □ 1 if $\|x^d - m_i\| = min_j \|x^d - m_j\|$
        -bid □ 0 otherwise
-For all $m_i$, $i = 1, ..., k$
        -$S_j$ □ sum over $b_i^d x^d$
        -$C_j$ □ $sum\ over\ b_i^d$
-Receive $S_j$ and $C_j$ from slaves, $for j = 1, ..., NumOfProcesses$.
-For all $m_i$, $i = 1, ..., k$
        -$M_j$ □ sum over $S_j / sum\ over\ C_j$
-Until converge
```

Slaves:

```
-Repeat
-Receive the mi (initial centroid points) from the master;
-For $x^d$ in $X_j$, for $j = 1, ..., NumOfProcesses$ and
$X_j = X[Size / NumOfProcesses * myrank : Size / NumOfProcesses * (myrank + 1)]$
        -bid □ 1 if $\|x^d - m_i\| = min_j \|x^d - m_j\|$
        -bid □ 0 otherwise
```

```
-For all $m_i$, $i = 1, ..., k$
        -$S_j$ □ sum over $b_i^d x^d$
        -$C_j$ □ $sum\ over\ b_i^d$
-Send $S_j$ and $C_j$ to the master
-Until converge
```

2. DNA Data

There are two differences between DNA data and Point data algorithms. First, to compute the Euclidean distance. I defined the Euclidean distance as the number of different characters between two DNAs. Second, merge the centroid points. I compute the counts of each characters and then sum them up in master. Then, I choose the character with greatest count in DNA data. That is it.

iii. the findings from your "Experimentation and Analysis", as described above.

	process:2 cluster:4 points:25	process:4 cluster:4 points:25	process:8 cluster:4 points:25	process:12 cluster:4 points:25
Point Data Parallel	369ms	418ms	664ms	1055ms
Point Data Sequential	11ms	11ms	11ms	11ms
DNA Data Parallel	225ms	363ms	612ms	918ms
DNA Data Sequential	28ms	28ms	28ms	28ms

	process:2 cluster:100 points:1000	process:4 cluster:100 points:1000	process:8 cluster:100 points:1000	process:12 cluster:100 points:1000
Point Data Parallel	107276ms	55881ms	46666ms	45406ms
Point Data Sequential	124051ms	124051ms	124051ms	124051ms
DNA Data Parallel	6412ms	6034ms	5967ms	5896ms
DNA Data Sequential	8951ms	8951ms	8951ms	8951ms

What I see from the above two tables:

1. When the data is small, sequential algorithm is much faster than parallel. The more mpi processes, the more slower the program runs. The reason is that when the data is small, the communication cost is big. So, parallel algorithm becomes slow and more processes more slower.

2. When the data is big, parallel algorithm is much better than sequential. Also, the more processes, the more better. The reason is that when the data is big, the computation costs most of the time, and the network cost becomes small. So, parallel is good and the more processes, it turns out the better performance.