Design and Implementation

This section of the report describes design decisions and implementation details for each subtask of this project. We decided to use Python as the only programming language for this project.

Dataset generator

We didn't modify provided generator for 2D points. We used it as a reference for DNA strands generator. DNA strands generator creates a list of random centroid strands, verifies that they are not too close to each other, and then produces mutated DNA strands for each centroid.

Sequential version

We implemented the sequential version before moving on to the parallel algorithm. We created the sequential algorithm in such a way the the difference between points and DNA strands was abstracted away using a single base class. This was critical in efficient implementation of the algorithm because we did not have to debug the code separately for points and DNa strands. We also designed the sequential algorithm in a way that would allow us to reuse most of the functionality directly for the parallel algorithm. We simply copy-pasted most of the functions from the sequential algorithm to parallel.

Parallel version

This section of the report represents the team perspective about how we went about parallelizing K-Means algorithm.
Implementation using mpi4py uses almost the same algorithm as sequential, but it requires additional functionality to synchronize centroids among different worker processes.
There were 2 main concerns for us while parallelizing K-Means:

1. Communication overhead between machines on GHC clusters
2. Load balancing for effective parallelization

The issue of load balancing was effectively handled by MPI. The only thing we had to do was specify the amount of data that would go into each process. Communication is still a concern since we do not have shared memory. We identified the aspects of this algorithm which can be effectively parallelized and began to fit the mpi4py framework around these aspects. We thus identified that re-allocation of points to other clusters based on their distance from each centroid was a computation-intensive calculation which could be parallelized effectively. We wanted to implement a threshold above which the clustering will always take place, but there was shortage of time. Thus, we designed the other aspects of our algorithm with a view of parallelizing the computationally intensive parts.

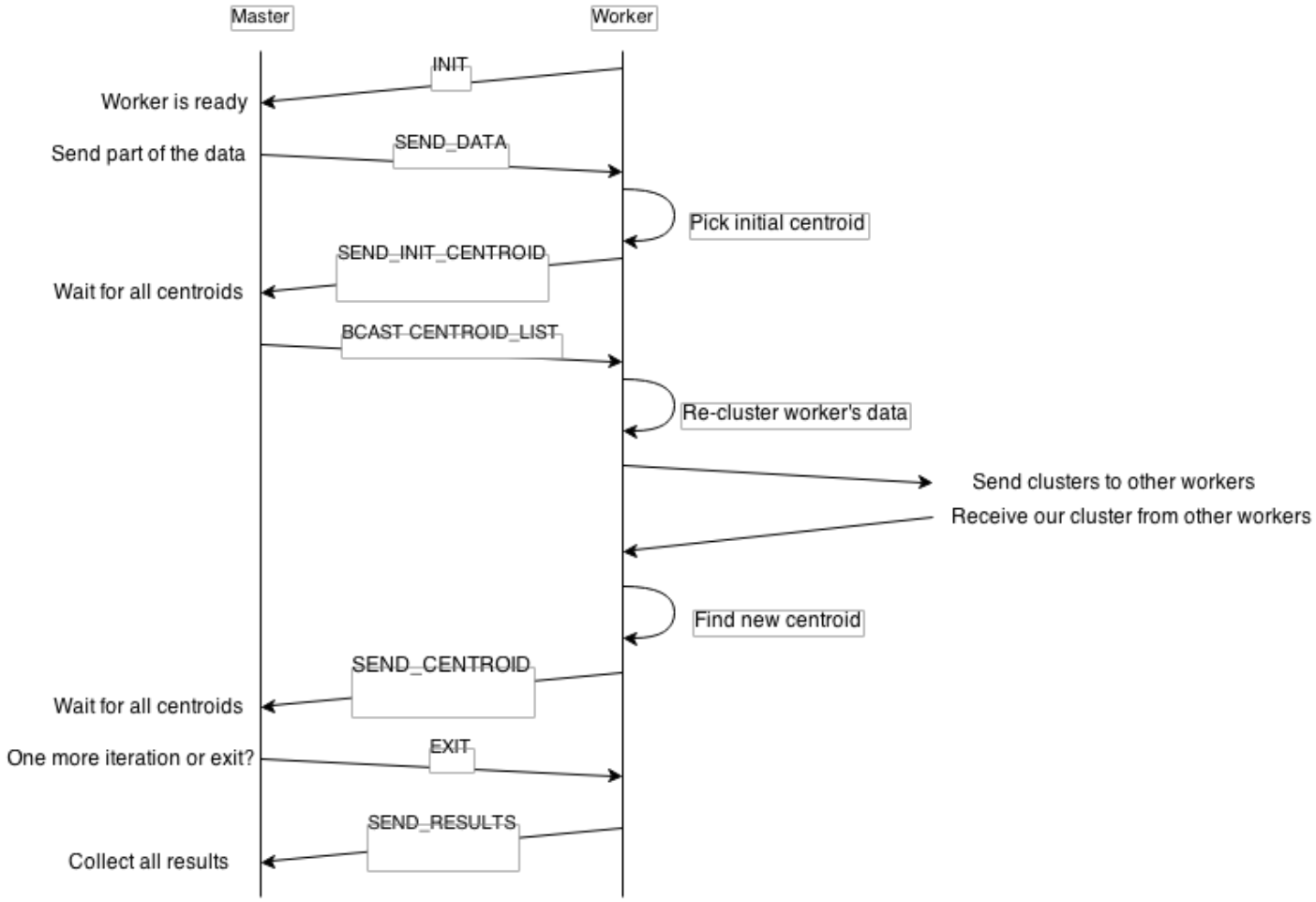

Figure 1. Sequence diagram of messages

The pseudocode for our parallel algorithm is:

1. Master initializes workers as shown in Figure 1.
2. Each worker sends an ‘INIT’ message to which the master replies back by giving a chunk of the data
3. After getting the data, the worker process will calculate the centroid of the chunk that it possesses and pass the centroid to the master
4. When the master gets replies from all workers and their respective centroids, it will broadcast the current centroids to all workers
5. After receiving the list of centroids, the worker will re-assign the cluster of all points it has.
6. The points which are closer to other centroids will be sent to their respective centroids
7. The workers will then calculate their new centroids and send the list to master
8. Steps 4-7 will repeat till a specified number of steps which is taken as input to the program

Thus, if you require more precision, then you should increase the amount of iterations that you specify as an input to the program. This provides some flexibility to the user to get the accuracy he desires in clustering. In order to increase reliability of the system, we introduced barriers using MPI. This was done so that all machines could synchronize before the beginning of each phase of the algorithm.

Testing

This section of the report provides guidance on testing our implementation of the dataset generator, sequential and parallel versions of k-mean clustering.

We didn't change the 2d points test data generation. So usage of the generator script remains the same:

```
$> python generaterawdata.py <required args> [optional args]
     -c <#>          Number of clusters to generate
     -p <#>          Number of points per cluster
     -o <file>       Filename for the output of the raw data
     -v [#]          Maximum coordinate value for points
```

Following example generates 5 clusters, each cluster contains 100 points, and all points will be written to 'input.txt' file:

```
python generaterawdata.py -c 5 -p 100 -o input.txt
```

Usage of the generator script for DNA strands:

```
$> python generaterawdata_dna.py <args>
     -c <#>          Number of clusters to generate
     -d <#>          Number of DNA strands per cluster
     -b <#>          Number of bases in each DNA strand
     -o <file>       Filename for the output of the raw data
```

Following example generates 8 clusters, each cluster contains 20 DNA strands, length of each strand is 15 bases, and results will be written to 'dna.txt' file:

```
python generaterawdata_dna.py -c 8 -d 20 -b 15 -o dna.txt
```

Usage of sequential clustering script:

```
$> python sequential_clustering.py <args>
     -d <point|dna>  Data type: 2D points or DNA strands
     -c <#>          Number of clusters
     -t <#>          Threshold
     -i <file>       Filename for the data input
     -o <file>       Filename for the results output
```

Following example applies sequential k-means clustering algorithm to DNA strands from 'dna.txt' file, which contains 8 clusters. And algorithm will stop, when amount of iterations is equal to or bigger than threshold.

```
python sequential_clustering.py -d dna -c 8 -t 5 -i dna.txt -o clusters.txt
```

Usage of parallel clustering script:

```
$> python parallel_clustering.py <args>
     -d <point|dna>  Data type: 2D points or DNA strands
     -t <#>          Threshold
     -i <file>       Filename for the data input
     -o <file>       Filename for the results output
```

Following example applies parallel k-means clustering algorithm to 2D points from 'point.txt' file, which contains 4 clusters (total number of processes minus master process)

```
mpirun -np 5 python parallel_clustering.py -d point -t 5 -i point.txt -o
clusters.txt
```

Also you can use provided ‘Makefile’ to quickly test generators, sequential and parallel clustering implementations:
`make gen_point` generates 10 clusters with 100 points in each cluster, and writes them to ‘point.txt’ file
`make gen_dna` generates 10 clusters with 100 strands in each cluster, and writes them to ‘dna.txt’ file
`make sequential` applies sequential algorithm to generated ‘dna.txt’ file
`make parallel` applies parallel algorithm to generated ‘dna.txt’ file

Experimentation and Analysis

This section of the report represents scalability analysis of the parallel version of clustering algorithm compared to the sequential implementation as a baseline.

Below is a graph of the performance benchmark for the various tests we ran. 1 processor represents that the algorithm was run in sequential on one machine.

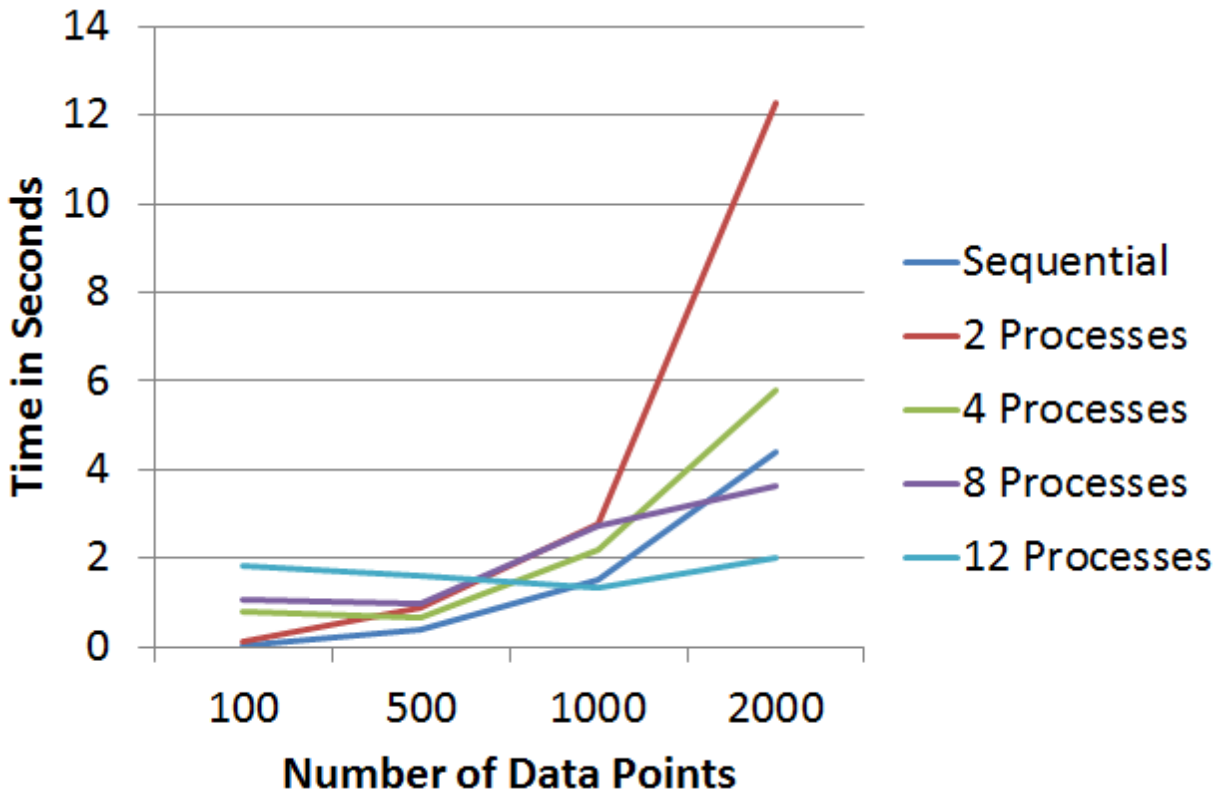

Figure 2: Performance Benchmark for Normal Loads

We noticed that parallelizing the algorithm does improve performance as the amount of processing to be done increases. However, for small loads, it is better to compute data sequentially. This might be because of the communication overhead associated with transferring data. Small data that fits into memory or which can cause a large amount of cache hits should be done in sequential on a single machine.

As we see a single sequential process is far better than all other alternatives in the beginning for low loads. As the amount of processing increases, it is better to use 8 or 12 processes. Using 2 or 4 processes leaves you worse off in any case because performance gain using small number of processes is not enough as compared to the cost of communication and context switching.

Below are the results we obtained from stress testing in Figure 3. We introduced a high computational load to see how better the parallel algorithm gets as compared to the sequential algorithm and the results we found were similar to what we had expected:

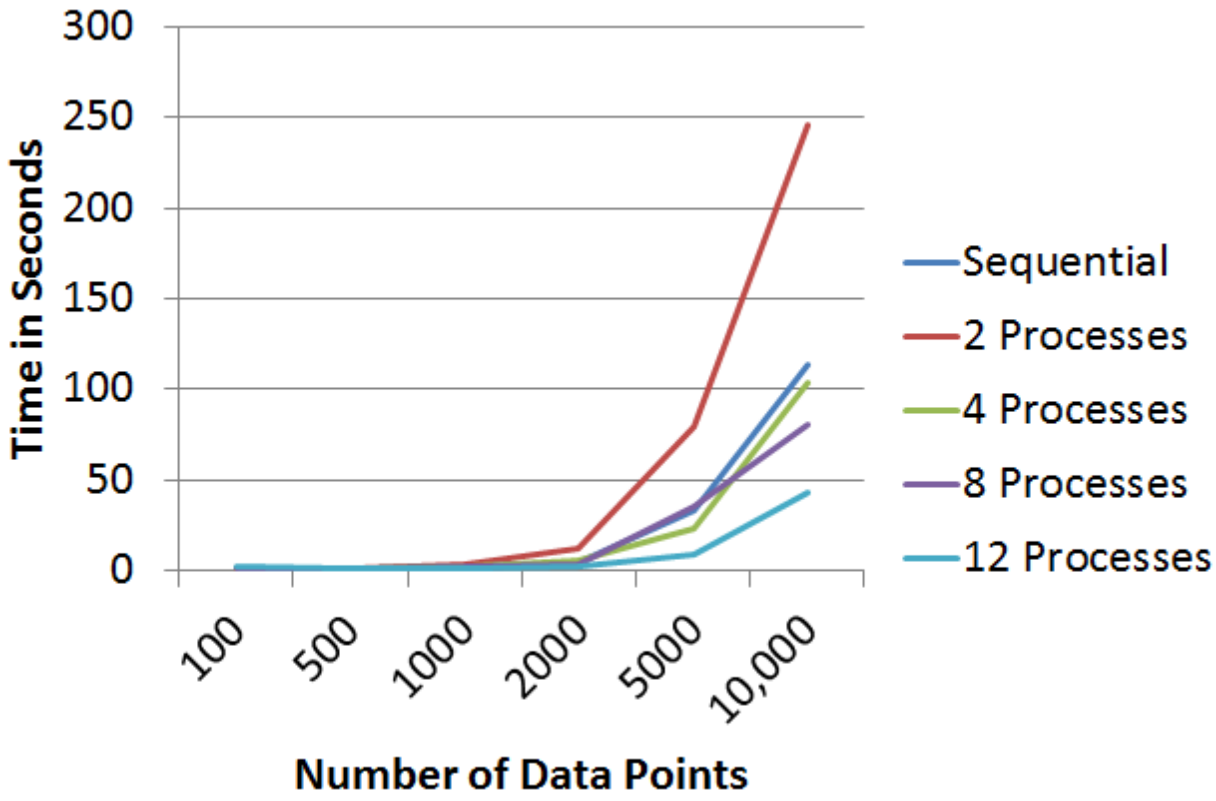

Figure 3: Stress Testing for High Loads

As we can see, the highly parallel algorithm is nearly 3 times as fast on a computational load of 10,000 points spread across 10 clusters. Thus, parallelization of such a high workload is a highly effective technique as compared to sequential processing. Below is the actual data from which Figure 2 and Figure 3 were derived:

	Sequential	**2 Processes**	**4 Processes**	**8 Processes**	**12 Processes**
100 points	*0.035*	0.14	0.81	1.07	1.83
500 points	*0.38*	0.88	0.67	0.98	1.6
1000 points	1.51	2.78	2.18	2.74	*1.35*
2000 points	4.39	12.26	5.79	3.62	*1.99*
5000 points	33.4	79.28	23.34	34.87	*8.42*
10,000 points	113.2	246.11	103.4	80.6	*42.79*

We noticed a special boost in performance when we used 4 processes. On looking further into the /proc/cpuinfo file, we found that the machine had 4 cores. This might be an important factor while

determining the “sweet spot” for deciding the number of machines and processes on each machine, as it gives improved performance over 8 processes.