Tanay Gavankar – tgavanka
Sid Soundararajan – ssoundar

Project 3 - Report

The parallelizing algorithm that we used is as follows:

1. Implicitly label the processor with rank 0 as the root processor. This is where data will be collected from other processors and distributed out to other processors.
2. Select *k* (where *k* is the number of desired clusters) elements from the input data set. To ensure an even distribution of starting centroids, use the QuickSelect algorithm and distribute the *k* different required selections across the processors (i.e. via scatter) if the data can be absolutely ordered (i.e. strings). In this specific implementation, the *k* positions are evenly spaced throughout the data point list. For 2d points, use the K-Means++ algorithm, on the root processor, as it is inherently sequential and cannot be parallelized.
3. A gather() call is then used to regroup all the initial centroids. The root processor combines and flattens this list, and adds them to the list of Cluster objects.
4. The root processor saves the centroids of the original clusters so it can calculate deltas at a later time.
5. The root processor broadcasts the list of clusters to all the other processors.
6. The data points are split (relatively) evenly into *np* pieces (where *np* is the number of processors).
7. These parts are then scattered to all other processors.
8. Each processor runs an iteration of k-means for its respective chunk of data.
9. The root processor then gathers the minimum clusters that the k-mean algorithm produced along with the data points associated with that cluster from each processor.
10. The root processor then go though the clusters we received from all of the processors, and combine all of the data points that are associated with each cluster. This is done since each processor was working on a chunk of the data.
11. The new clusters (and associated data points) are now split again by the number of processors and scattered to the processors.
12. Each processor takes their assigned clusters and calculates the centroid for each cluster.
13. The root processor gathers the newly calculated centroid clusters and recombines them.
14. The root processor checks if these new clusters pass the variance test by comparing the new centroids to the old saved (in step 4) centroids.
15. If it does, the root processor broadcasts a flag to all the processors to signal them to exit the loop. If it does not, the algorithm goes back to step 4.
16. The root processor outputs all the calculated data and clusters and terminates.

Here is the pseudo code for this algorithm. All references to MPI refer to MPI.COMM_WORLD.

```
main()
	Initialize_MPI()
	data_points = MPI.scatter(data_points, root=0)
	k_points_to_pick = MPI.scatter(k_points_to_pick, root=0)
	partial_clusters = select_points(data_points, k_points_to_pick)
	clusters = MPI.gather(partial_clusters, root=0)

	myrank = MPI.Get_rank()
	num_proccesors = MPI.Get_size()

	while true
		if myrank == 0
			save_old_centroids()
			clusters = MPI.bcast(clusters, root = 0)

		data = MPI_scatter(chunk_data(data_points, num_proccesors),root = 0)

		new_clusters = one_iteration_of_kmeans(data)

		reply = MPI_gather(new_clusters, root=0)

		if myrank == 0
			then combine each cluster's associated points from each proccesor

		split_clusters = chunk(clusters)

		clusData = MPI.scatter(split_clusters, root=0)

		for c in clusData:
			c.add(p)
			c.calcCentroid()

		gatheredClus = MPI.gather(clusData, root=0)

		if myrank == 0 and new_clusters passes variance test
			end_reached = True
		else
			end_reached = False

		break_out =MPI.bcast(end_reached, root=0)

		If break_out
			break

	if myrank == 0
		output all calculated data
	MPI.Finalize()
```

Generalized Algorithm

Since the two problems we were given (2D points and DNA strands) differed slightly, we felt that it would be best for each to have its own specific algorithm in our implementation. This section describes the chosen and implemented algorithm for each in sequential terms, as the algorithm for parallelization is given elsewhere.

First, we will cover the input data set of 2D points. The points are read into memory from the file into a list, and the first step is to select the starting centroids. To do this in a deterministic fashion (so as to avoid the worst case selection possibly caused by randomized selection), we use the K-Means++ algorithm to select the centroids. This works by randomly selecting a point as the first centroid, and then after that selecting the succeeding centroids with probability proportional to $D(x)^2$, where D(x) is the distance between that point and the nearest already chosen centroid. This makes centroids that are farther away be more likely to be chosen, and thus avoids the issue of selecting centroids that are too close together. Then, the current centroids are saved for future comparison. For every point, we then compute the distance to each centroid and choose then add the point to the cluster with the closest centroid. After all the points have been iterated over, the new centroids for each cluster are calculated and respectively updated. Then the largest change in centroids (calculated by distance between new centroid and old centroid for that cluster) is compared to the given number of k-means iterations, and if the centroid has changed less than it, then the algorithm is done and it simply outputs the results. If not, then it loops back to the part with saving the old centroids and performs the steps again.

The algorithm for an input data set of DNA strings is very similar to the algorithm outline above for 2D points, but differs in two major aspects. The first distinction is in how the initial centroids are chosen. Since strings can be absolutely ordered, it is better to simply use the QuickSelect algorithm to pick points that are far apart in the data set. This is done by partitioning the data set (as if it were sorted) into k pieces and selecting the median of each of those k pieces. However, since sorting is an O(nlogn) operation, the data set itself is never fully sorted - we simply pull out the k elements. In particular, we use QuickSelect at positions i*(numPoints/k) + numPoints / (2k) for i in range(0, k). This reduces the runtime to O(kn) (since QuickSelect is O(n) and we call it k times). The other difference in this algorithm and the one outlined above is in how the centroid of a cluster is calculated. For 2D points, it is simply a matter of averaging each coordinate. However, this does not work when working with strings. Our solution consisted of determining what the most common character in each position was, and setting that as the character in the centroid for that position. However, this does not work well for cases where there are ties for the most common character. For example, consider the case with strings ACG, ATG, GGG. Clearly, the first character and the third character of the centroid of these strings must be A and G respectively. However, there is a 3-way tie for the middle character. If we pick ACG as the centroid, then that gives us Hamming distances of 0, 1, 2, respectively. If we pick ATG, we get Hamming distances of 1, 0, 2. However, if we pick AGT, we get Hamming distances of 1, 1, 1. We felt that while the average distance might be the same, the individual distances were "better" when distances of 1, 1, 1 were obtained. To ensure this, we modified the centroid calculation algorithm for the specific case of when there was a tie in most common character. When this occurs, it goes through all the other already selected characters and strings and keeps track of how many times a string contained a "winning" character. Then, the string with the lowest amounts of "wins" was given the "win" for this character by selecting its character. Going back to the example, we notice that our centroid has the form A_G, meaning that ACG has won twice, ATG has won twice, while GGG has only won once. So, since it has the minimum number of wins, we give it the win by using T in the open slot. This tremendously evens out cluster sizes and gives better cluster centroid calculations.

Output Files

The format of the output files is a CSV. For the 2D points, the output file consists of 2k columns, with the odd columns being the x coordinate and the even columns being the y coordinate. The first line is always the calculated centroid.

Below is a sample of a 2 cluster, 2 pt per cluster table.

CENTROID1_X	CENTROID1_Y	CENTROID2_X	CENTROID2_Y
p1_x	p1_y	p3_x	p3_y
p2_x	p2_y	p4_x	p4_y

The same applies for DNA strand CSV output, but the x coordinate is replaced with the DNA strand string, and the y coordinate is replaced with the Hamming distance between that string and its corresponding centroid.

Analysis

Graph reference sheets are in the following pages, along with raw data.

For this analysis we ran our code on several test data files. We varied the number of processors , the number of k-clusters, and the number of points per cluster (ppc). For the number of processors, we used 1, 2 4, 8 or 12 processors. For clusters, we used 2, 4, 8, 12, or 25 clusters. Finally, for ppc, we used 100, 1000, 5000, and 10000.

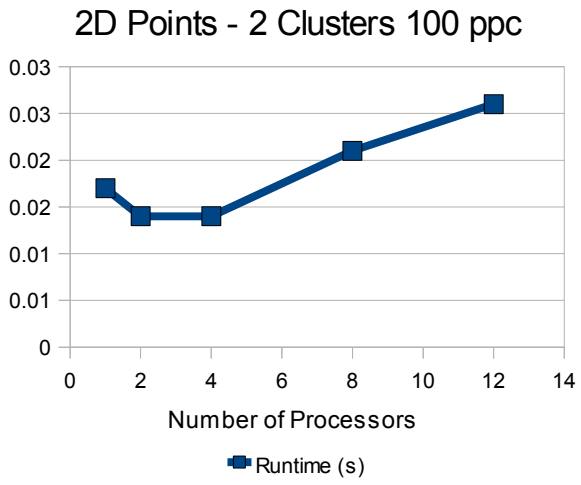
From the generated graphs we can observe a few points:

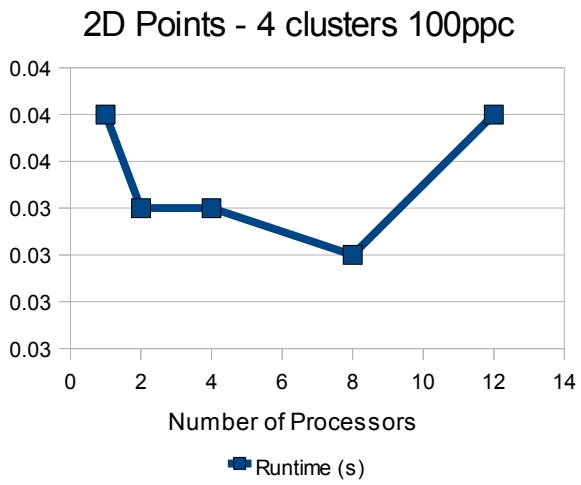
1) In general for the 100 ppc we can see that we want to pick between 4 and 8 processors. However, the optimal choice depends on the number of clusters. As the number of clusters increase, we see the higher number of processors begin to give us better results. When we look at the graphs for 1000 ppc, 5000 ppc, and 10000 ppc, we see that this seems to be true for them as well. Thus, the optimal choice for processors is dependent on the number of clusters. If we have a low number of clusters, we want a lower number of processors, however, as we use more clusters, the higher number of processors look more appealing.
2) We can also see that increasing the number of points, but not the clusters does not change the shape of the graphs very much. This indicates that the optimal processor number is not dependent on the ppc. This along with point 1 means that the optimal number of processors is dependent on only the number of clusters.

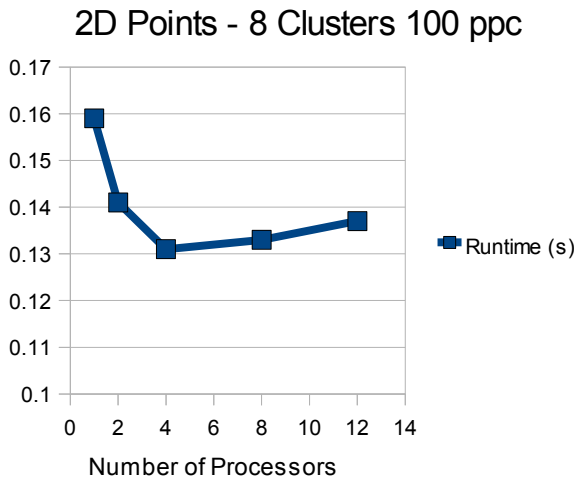
Now we can see if these two points make sense according to our code. The parallelizing algorithm works by splitting up the clusters among the processors. Thus having more processors should not help very much for a low number of clusters. Instead it would probably hurt us, as we try to send and receive data from those clusters. However, when we get to a higher number of clusters, we will start using more of the processors, thus having a higher number of processors will help us complete the task faster.

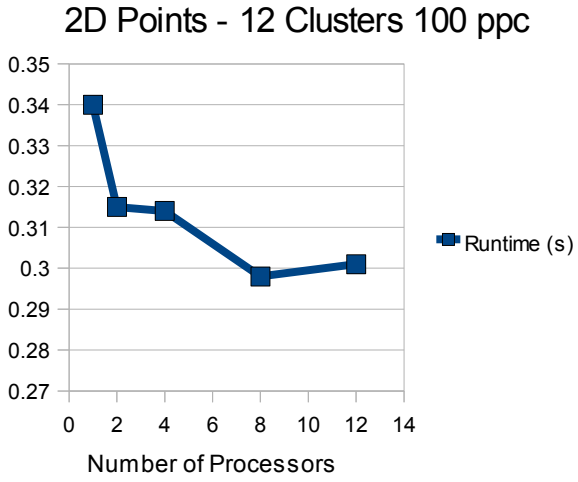
If we look at the graphs, we notice some processor number is very good for each data set. In almost all of the cases, it is either 4 processors or 8. The difference in time between those is also quite small. After that, the cost usually goes up. Thus the optimal number of processors is between 4 and 8 for most cases. The increase in cost of adding more processors is probably due to an increase in the amount of communication between processors. The communication is has a high cost, and so after a certain number of processors, we reach a point where adding another processors will increase the cost more than it decreases, thus leading to diminishing returns. In this case, that point seems to be around 8 processors.

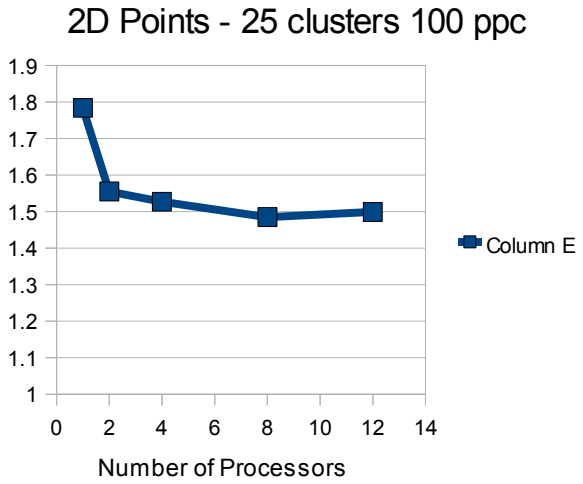
Graphs – 100 ppc



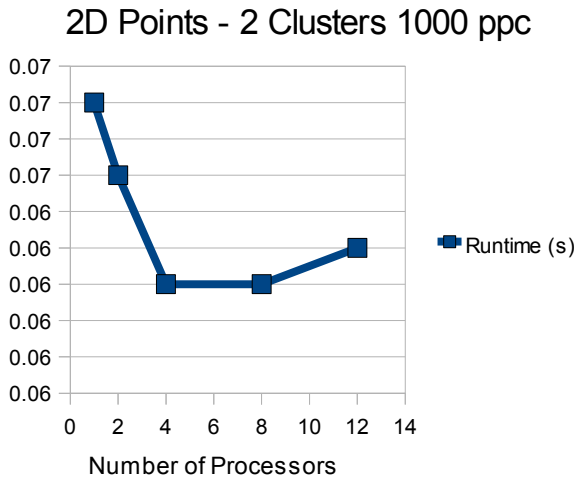


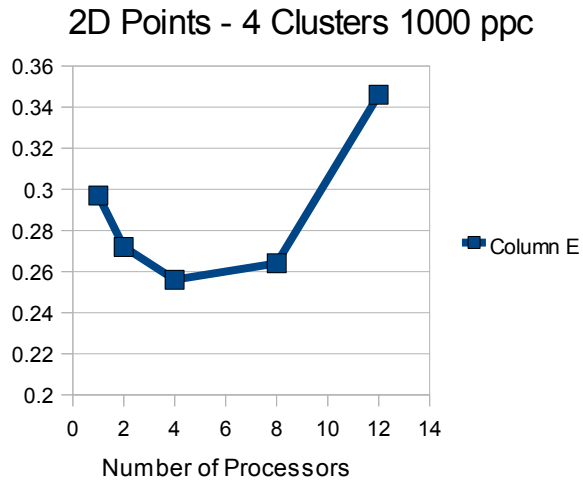


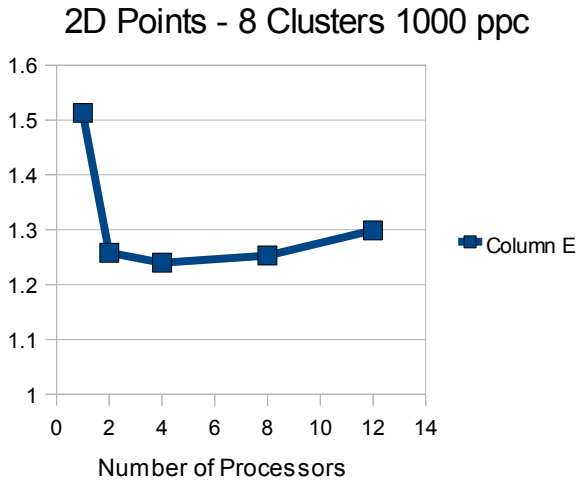


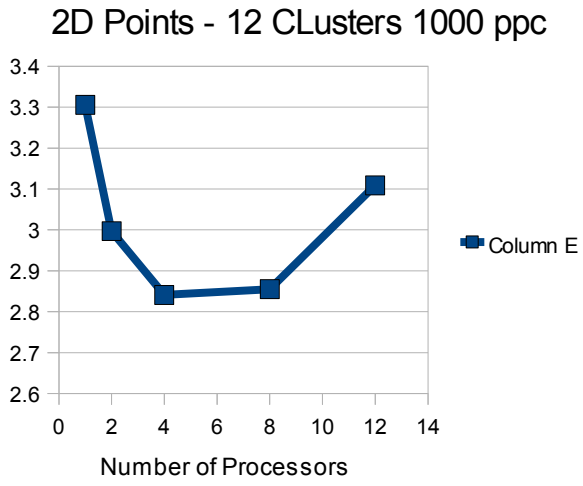


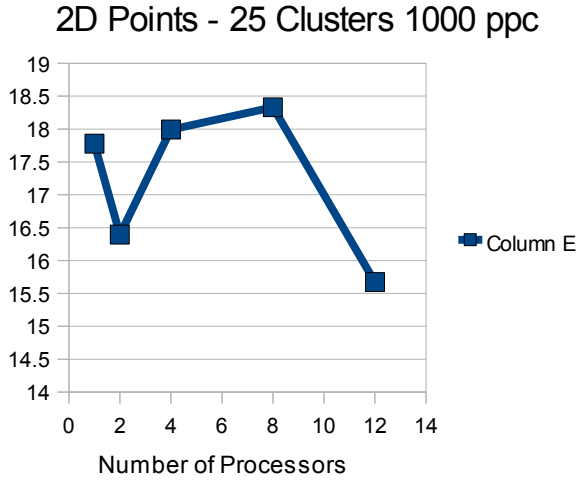
Graphs – 1000 ppc



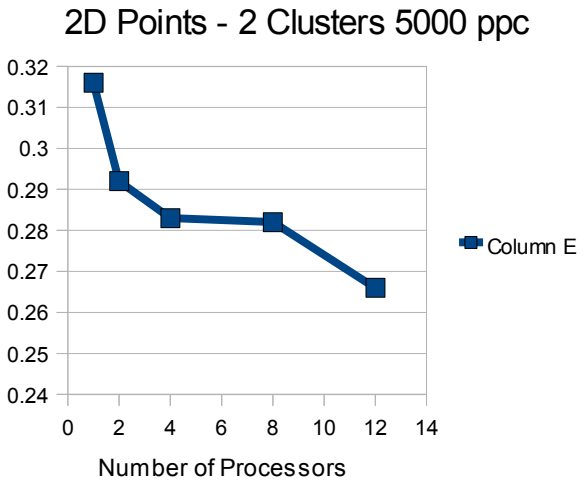


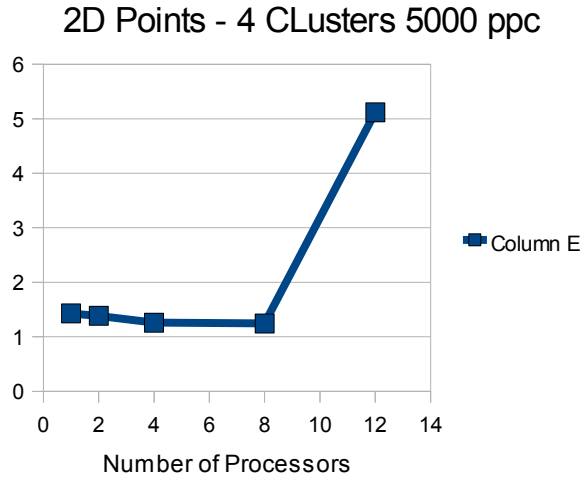


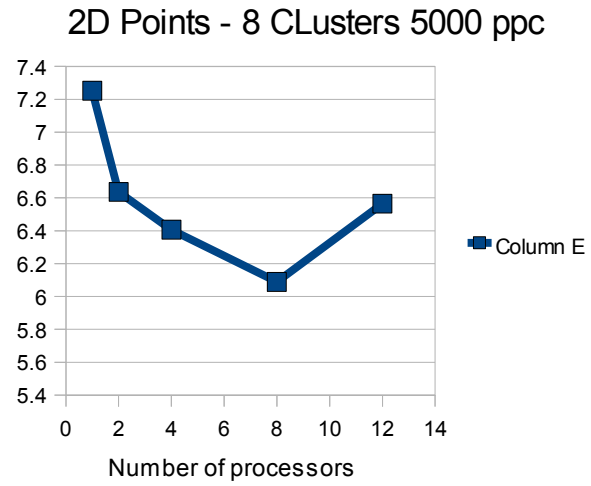


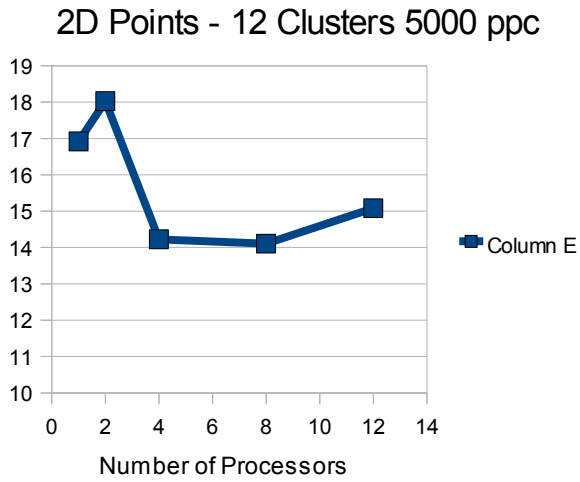


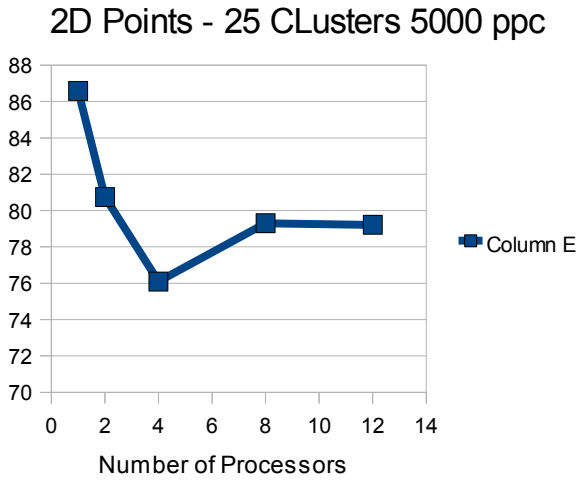
Graphs – 5000 ppc



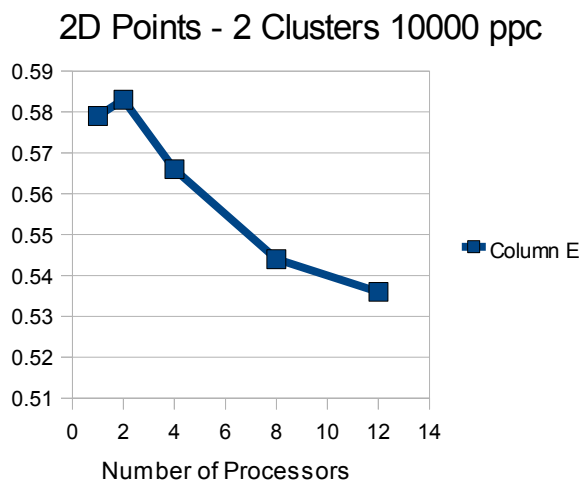


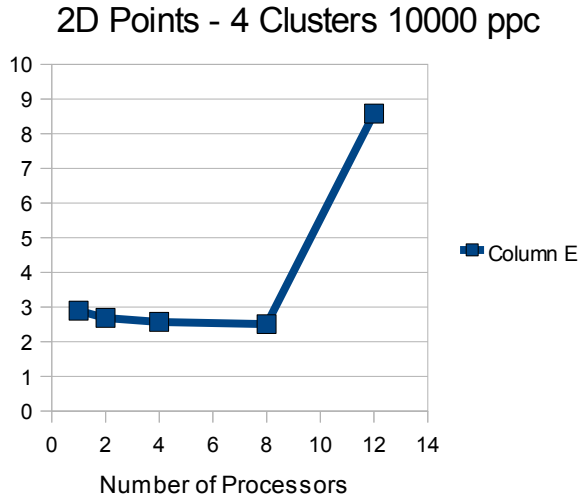


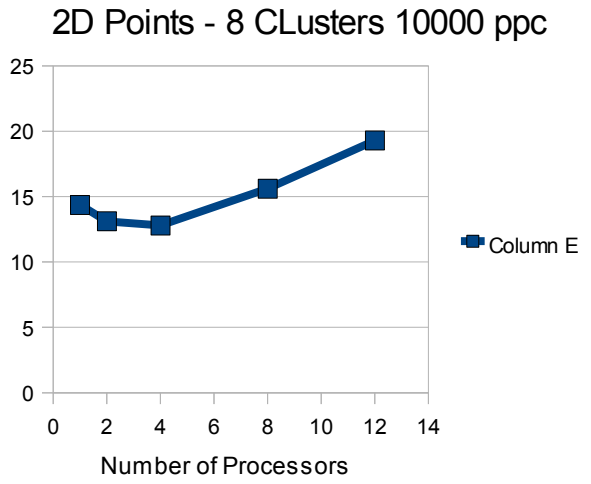


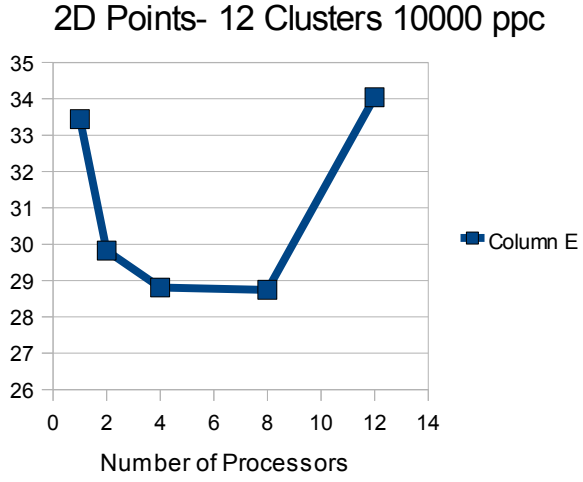


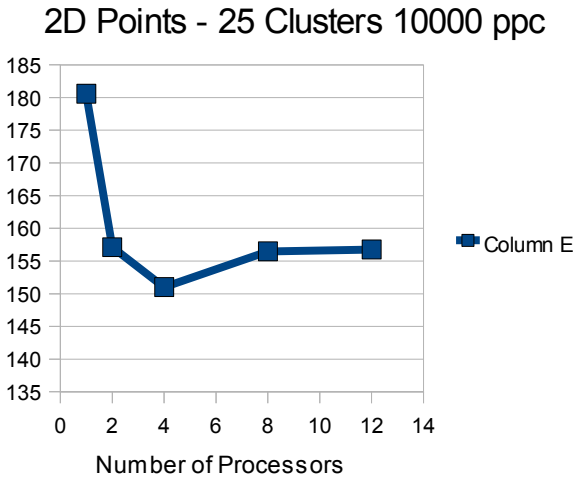
Graphs – 10000 ppc











Raw Data

Data Type	Number of Clusters (k)	Points per Cluster	Number of Processors	Implementation Type	Runtime (s)
2D Point	2	100	1	seq	0.017
2D Point	2	1000	1	seq	0.067
2D Point	2	5000	1	seq	0.316
2D Point	2	10000	1	seq	0.579
2D Point	2	100	2	mpi	0.014
2D Point	2	1000	2	mpi	0.065
2D Point	2	5000	2	mpi	0.292
2D Point	2	10000	2	mpi	0.583
2D Point	2	100	4	mpi	0.014
2D Point	2	1000	4	mpi	0.062
2D Point	2	5000	4	mpi	0.283
2D Point	2	10000	4	mpi	0.566
2D Point	2	100	8	mpi	0.021
2D Point	2	1000	8	mpi	0.062
2D Point	2	5000	8	mpi	0.282
2D Point	2	10000	8	mpi	0.544
2D Point	2	100	12	mpi	0.026
2D Point	2	1000	12	mpi	0.063
2D Point	2	5000	12	mpi	0.266
2D Point	2	10000	12	mpi	0.536
2D Point	4	100	1	seq	0.036
2D Point	4	1000	1	seq	0.297
2D Point	4	5000	1	seq	1.427
2D Point	4	10000	1	seq	2.894
2D Point	4	100	2	mpi	0.034
2D Point	4	1000	2	mpi	0.272
2D Point	4	5000	2	mpi	1.384
2D Point	4	10000	2	mpi	2.687
2D Point	4	100	4	mpi	0.034
2D Point	4	1000	4	mpi	0.256
2D Point	4	5000	4	mpi	1.259
2D Point	4	10000	4	mpi	2.573
2D Point	4	100	8	mpi	0.033
2D Point	4	1000	8	mpi	0.264
2D Point	4	5000	8	mpi	1.243
2D Point	4	10000	8	mpi	2.509
2D Point	4	100	12	mpi	0.036
2D Point	4	1000	12	mpi	0.346
2D Point	4	5000	12	mpi	5.117
2D Point	4	10000	12	mpi	8.583
2D Point	8	100	1	seq	0.159
2D Point	8	1000	1	seq	1.513

2D Point	8	5000	1	seq	7.252
2D Point	8	10000	1	seq	14.357
2D Point	8	100	2	mpi	0.141
2D Point	8	1000	2	mpi	1.258
2D Point	8	5000	2	mpi	6.636
2D Point	8	10000	2	mpi	13.105
2D Point	8	100	4	mpi	0.131
2D Point	8	1000	4	mpi	1.24
2D Point	8	5000	4	mpi	6.406
2D Point	8	10000	4	mpi	12.79
2D Point	8	100	8	mpi	0.133
2D Point	8	1000	8	mpi	4.233
2D Point	8	5000	8	mpi	6.089
2D Point	8	10000	8	mpi	15.606
2D Point	8	100	12	mpi	0.137
2D Point	8	1000	12	mpi	1.299
2D Point	8	5000	12	mpi	6.565
2D Point	8	10000	12	mpi	19.282
2D Point	12	100	1	seq	0.34
2D Point	12	1000	1	seq	3.306
2D Point	12	5000	1	seq	16.921
2D Point	12	10000	1	seq	33.442
2D Point	12	100	2	mpi	0.315
2D Point	12	1000	2	mpi	2.997
2D Point	12	5000	2	mpi	18.027
2D Point	12	10000	2	mpi	29.821
2D Point	12	100	4	mpi	0.314
2D Point	12	1000	4	mpi	2.841
2D Point	12	5000	4	mpi	14.227
2D Point	12	10000	4	mpi	28.808
2D Point	12	100	8	mpi	0.298
2D Point	12	1000	8	mpi	2.855
2D Point	12	5000	8	mpi	14.106
2D Point	12	10000	8	mpi	28.744
2D Point	12	100	12	mpi	0.301
2D Point	12	1000	12	mpi	3.109
2D Point	12	5000	12	mpi	15.082
2D Point	12	10000	12	mpi	34.046
2D Point	25	100	1	seq	1.784
2D Point	25	1000	1	seq	17.779
2D Point	25	5000	1	seq	86.583
2D Point	25	10000	1	seq	180.581
2D Point	25	100	2	mpi	1.555
2D Point	25	1000	2	mpi	16.394
2D Point	25	5000	2	mpi	80.746

2D Point	25	10000	2	mpi	157.113
2D Point	25	100	4	mpi	1.527
2D Point	25	1000	4	mpi	17.991
2D Point	25	5000	4	mpi	76.092
2D Point	25	10000	4	mpi	151.017
2D Point	25	100	8	mpi	1.485
2D Point	25	1000	8	mpi	18.331
2D Point	25	5000	8	mpi	79.301
2D Point	25	10000	8	mpi	156.48
2D Point	25	100	12	mpi	1.499
2D Point	25	1000	12	mpi	15.674
2D Point	25	5000	12	mpi	79.212
2D Point	25	10000	12	mpi	156.765
2D Point	50	100	1	seq	6.844
2D Point	50	1000	1	seq	72.475
2D Point	50	5000	1	seq	346.772
2D Point	50	10000	1	seq	698.82
2D Point	50	100	2	mpi	9.256
2D Point	50	1000	2	mpi	63.39
2D Point	50	5000	2	mpi	317.367
2D Point	50	10000	2	mpi	632.181
2D Point	50	100	4	mpi	6.069
2D Point	50	1000	4	mpi	60.906
2D Point	50	5000	4	mpi	315.562
2D Point	50	10000	4	mpi	610.272
2D Point	50	100	8	mpi	6.164
2D Point	50	1000	8	mpi	60.672
2D Point	50	5000	8	mpi	301.176
2D Point	50	10000	8	mpi	618.8
DNA Strand	2	100	1	seq	0.012
DNA Strand	2	1000	1	seq	0.083
DNA Strand	2	5000	1	seq	0.22
DNA Strand	2	10000	1	seq	0.687
DNA Strand	4	100	1	seq	0.02
DNA Strand	4	1000	1	seq	0.205
DNA Strand	4	5000	1	seq	0.6
DNA Strand	4	10000	1	seq	1.881
DNA	8	100	1	seq	0.116

Strand					
DNA Strand	8	1000	1	seq	0.107
DNA Strand	8	5000	1	seq	1.224
DNA Strand	8	10000	1	seq	0.272
DNA Strand	12	100	1	seq	0.125
DNA Strand	12	1000	1	seq	0.064
DNA Strand	12	5000	1	seq	0.246
DNA Strand	12	10000	1	seq	0.221
DNA Strand	25	100	1	seq	0.215
DNA Strand	25	1000	1	seq	0.233
DNA Strand	25	5000	1	seq	0.229
DNA Strand	25	10000	1	seq	0.183
DNA Strand	50	100	1	seq	0.242
DNA Strand	50	1000	1	seq	0.219
DNA Strand	50	5000	1	seq	0.233
DNA Strand	50	10000	1	seq	0.266