Chris Keefe
Assignment 5 - K Means Clustering

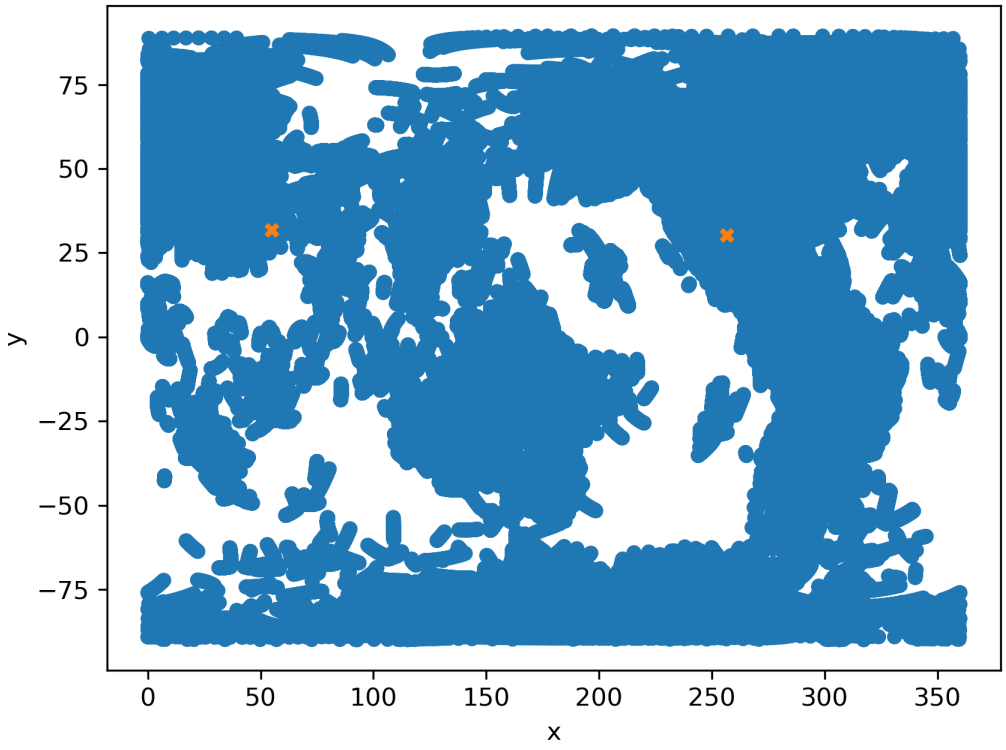
Activity 1: Code description

This implementation of K Means clustering begins importing data and sharing it (complete) to all ranks. Because all ranks have access to the full data, we must next assign data ranges describing the points for which each rank is responsible. Start index calculations are done locally on R0, and then MPI_Bcast to all ranks. Ranks then independently calculate their upper-bound indices. We select the first KMEANS points in our data set for use as initial centroids, and store them in a 1d array for simpler communication. Because “convergence” is being defined here as “completion of 10 iterations”, we loop while n_iters < KMEANSITERS. We report the centroid coordinates (conditionally) at the top of the loop for validation and plotting purposes.

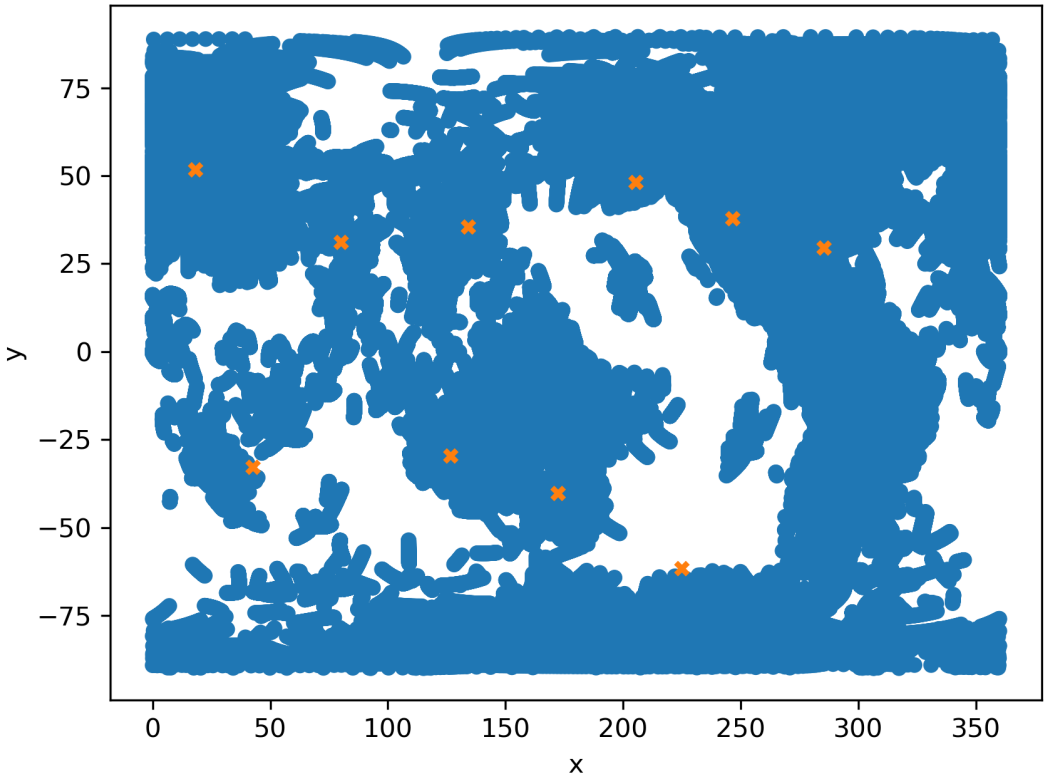
During the cluster assignment phase of the algorithm, we iterate over points, comparing each dimension of each point to the corresponding dimension of each centroid, capturing the “clustering” label of the nearest cluster center for each point in an array. For more utility, we would allow the user to capture total loss per iteration. This can be used to quantify quality of fit in model validation, and may also be used in checking for convergence. Doing so would require us to take the true euclidean distance, by taking the square root of the distance I’ve calculated here. This implementation uses the squared euclidean distance instead, to remove the runtime costs of the square root operation.

In the centroid adjustment phase of the algorithm, we find local cardinalities and per-dimension sums, and then use MPI_Allreduce to generate global cardinality data at all ranks. We might be able to hide some communication overhead by performing a nonblocking MPI_Iallreduce immediately after generating local cardinalities, and proceeding to compute our sums. This would reduce readability, and the overhead of looping over all values again seemed likely to outweigh the savings, so I kept those two operations packaged in one loop. Once all ranks have global cardinalities, I calculate weighted means, and use MPI_Allreduce again to sum them, producing new cluster centers at all ranks.

Timestamps are captured along the way, and timing calculations are performed outside of timed sections wherever it’s straightforward to do so. I use MPI_Reduce to find the max timings for each category across ranks, report timing data, clean up allocated memory, and exit. The code passes all validation checks, and produces the following clusterings:

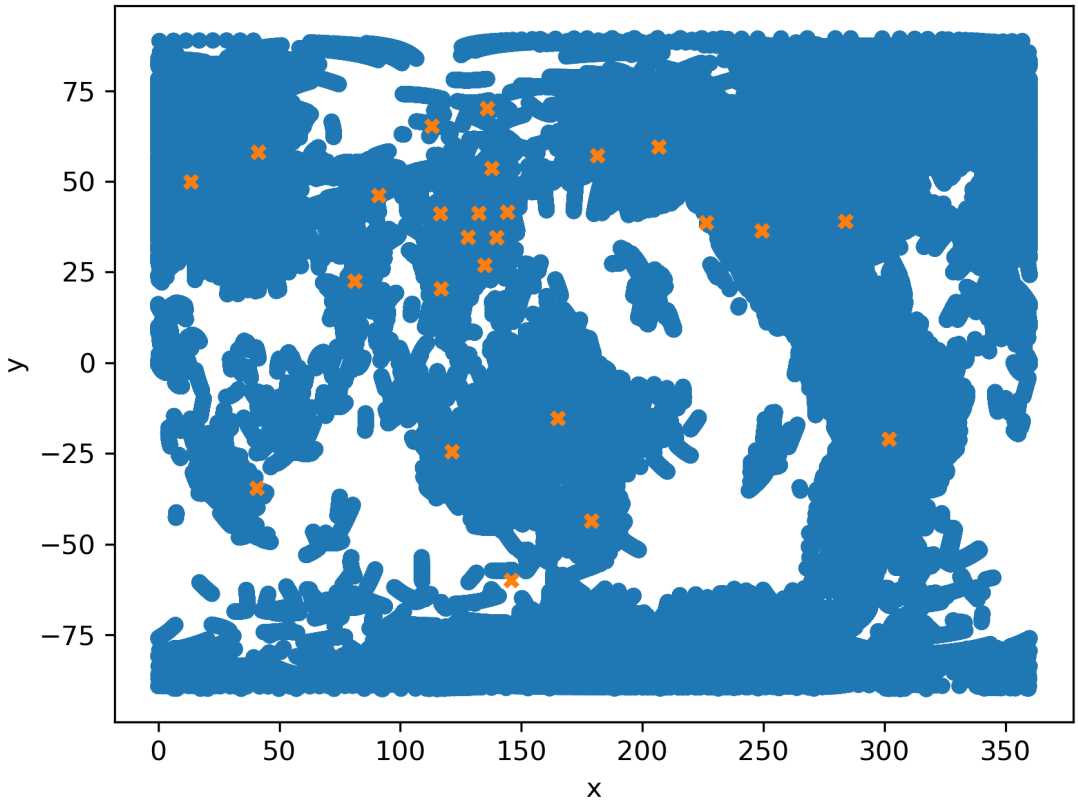

75
50
25
0
−25
−50
−75
y
0
50
100
150
200
250
300
350
x

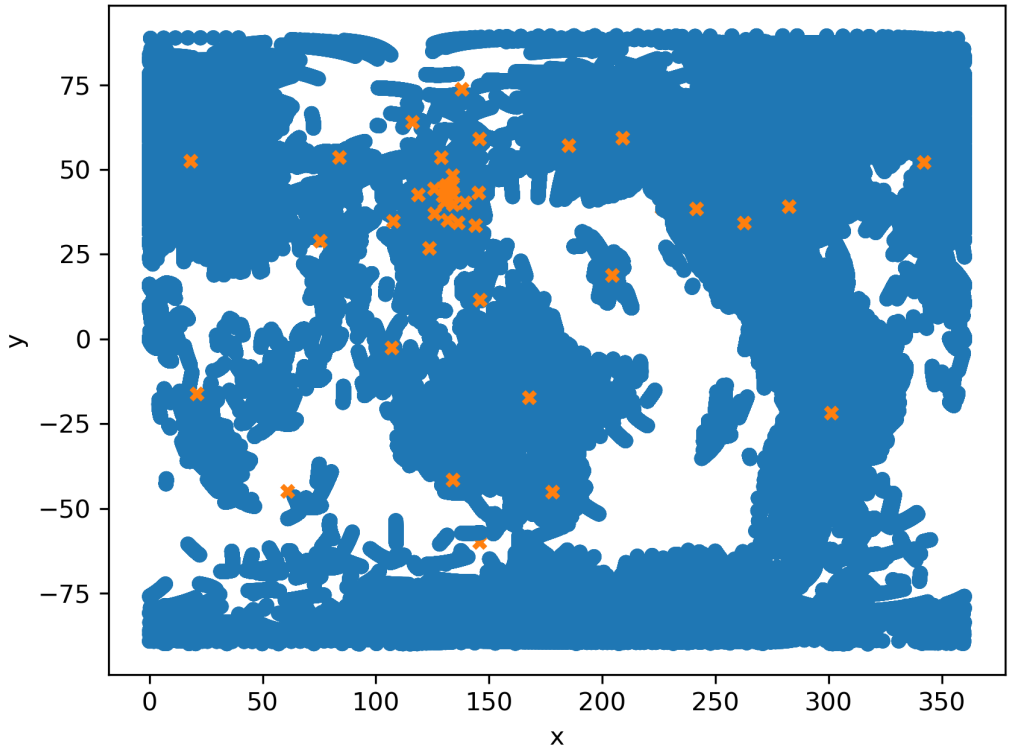
k=2


75
50
25
0
−25
−50
−75
y
0
50
100
150
200
250
300
350
x

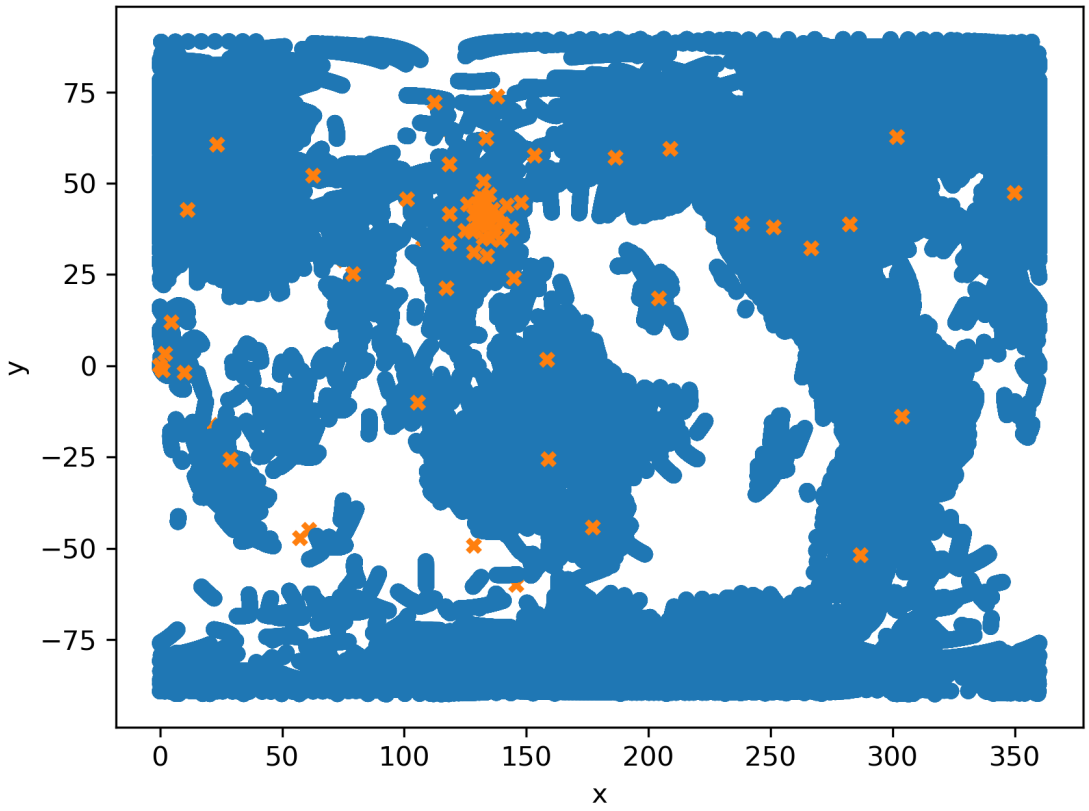
k=10

k=25


75
50
25
0
−25
−50
−75
y
0
50
100
150
200
250
300
350
x


75
50
25
0
−25
−50
−75
y
0
50
100
150
200
250
300
350
x

k=50



k=100

Jobscript/run description

As you've seen in previous assignments, I used bash scripts (`sbatch.command`) to submit custom jobscripts, exporting environment variables to set filenames and other job parameters. I also scripted data aggregation and mean timing calculations using bash and python respectively. This may have taken more time for this assignment than manual data input would have, but it was much more interesting, and provided a level of consistency, repeatability, and idiot-proofing I appreciate.

Q1: What component of the algorithm is straightforward to parallelize?

Calculating distances to generate clusterings. Every rank has access to all of the data, and so can assign centroids to its points without any communication.

Q2: When parallelizing the algorithm, what component of the algorithm requires communication between process ranks?

Updating cluster centers. Because clustering data is calculated in a distributed manner, ranks must share their clustering data (or a reduction of it) in order to calculate globally-shared cluster means.

One-node Tables

T1: Total Response Time

	K=2	K=25	K=50	K=100	Jobscript
1	0.5155	3.1919	6.3833	12.3833	jobscript.sh + sbatch.command
4	0.1477	0.8146	1.6079	3.0527	jobscript.sh + sbatch.command
8	0.0817	0.4099	0.8007	1.5382	jobscript.sh + sbatch.command
12	0.0529	0.2733	0.5315	1.0197	jobscript.sh + sbatch.command
16	0.0468	0.2046	0.4064	0.7703	jobscript.sh + sbatch.command
20	0.0372	0.1709	0.3289	0.6193	jobscript.sh + sbatch.command

T2: Max cumulative time calculating distances

	K=2	K=25	K=50	K=100	Jobscript
1	0.3534	3.0144	6.211	12.0027	jobscript.sh + sbatch.command
4	0.1043	0.7733	1.5669	3.002	jobscript.sh + sbatch.command
8	0.0582	0.3881	0.7791	1.5066	jobscript.sh + sbatch.command
12	0.034	0.2578	0.516	1.004	jobscript.sh + sbatch.command
16	0.0329	0.1926	0.3942	0.7507	jobscript.sh + sbatch.command
20	0.0239	0.1562	0.3151	0.6026	jobscript.sh + sbatch.command

T3: Max cumulative time updating means, including synchronization between ranks

	K=2	K=25	K=50	K=100	Jobscript
1	0.1622	0.1774	0.1722	0.2084	jobscript.sh + sbatch.command
4	0.0627	0.0696	0.0732	0.0722	jobscript.sh + sbatch.command
8	0.0413	0.0395	0.0332	0.0508	jobscript.sh + sbatch.command
12	0.026	0.0265	0.0206	0.027	jobscript.sh + sbatch.command
16	0.0271	0.0193	0.0232	0.0254	jobscript.sh + sbatch.command
20	0.0211	0.0227	0.0223	0.0231	jobscript.sh + sbatch.command

T4: Speedup

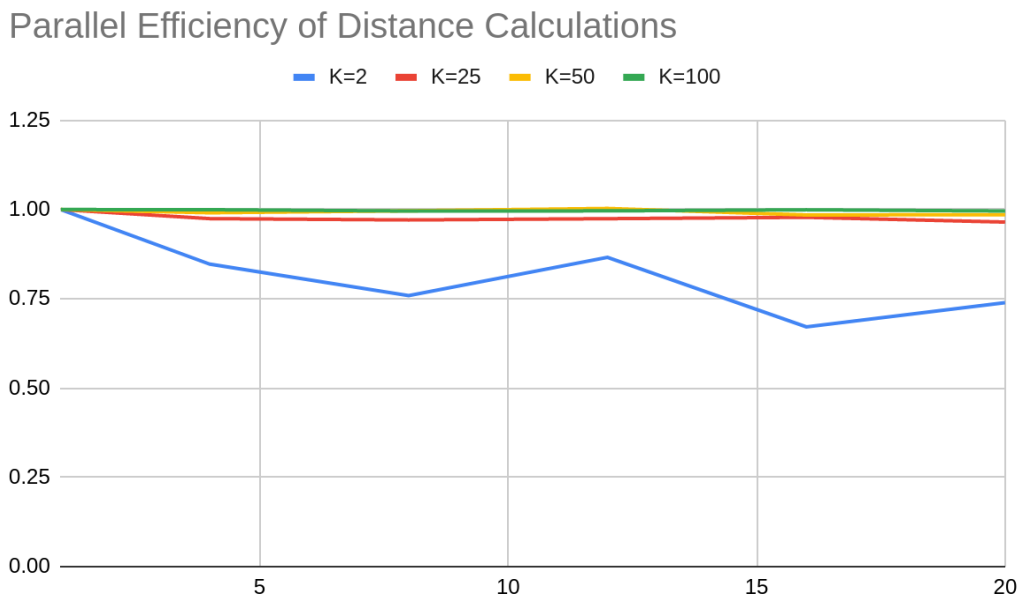
	K=2	K=25	K=50	K=100
1	1	1	1	1
4	3.490182803	3.918364842	3.969960818	4.056507354
8	6.309669523	7.787021225	7.972149369	8.050513587
12	9.744801512	11.67910721	12.00997178	12.14406198
16	11.01495726	15.60068426	15.70693898	16.07594444
20	13.85752688	18.6770041	19.40802676	19.99564024

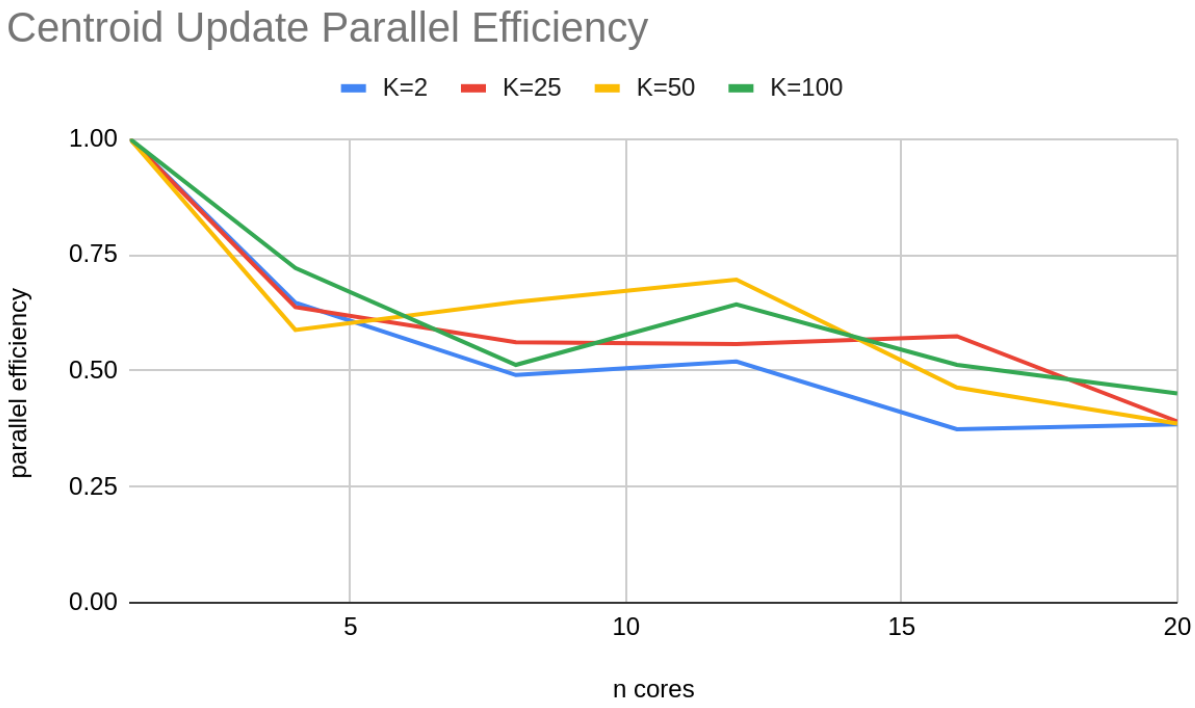
Q3: Describe how you implemented your algorithm.

A full description of my implementation is in the “Code description” section above, if that’s what you’re looking for in this question. If you’re looking for comments on the development process, I began by breaking up the code into subprocedures in pseudo code, and implemented each in turn. I then spent the obligatory hours tracking down sneaky c bugs and squashing them.

Q4: Explain the (potential) bottlenecks in your algorithm.

The distance calculations appear to be compute-bound, with near-perfect parallel efficiency for K=25 and higher, and pretty good parallel efficiency even at k=2. The cluster mean updates, on the other hand, appear to be communication-bound, with parallel efficiency rarely exceeding 0.6, and degrading as we scale up the number of cores.

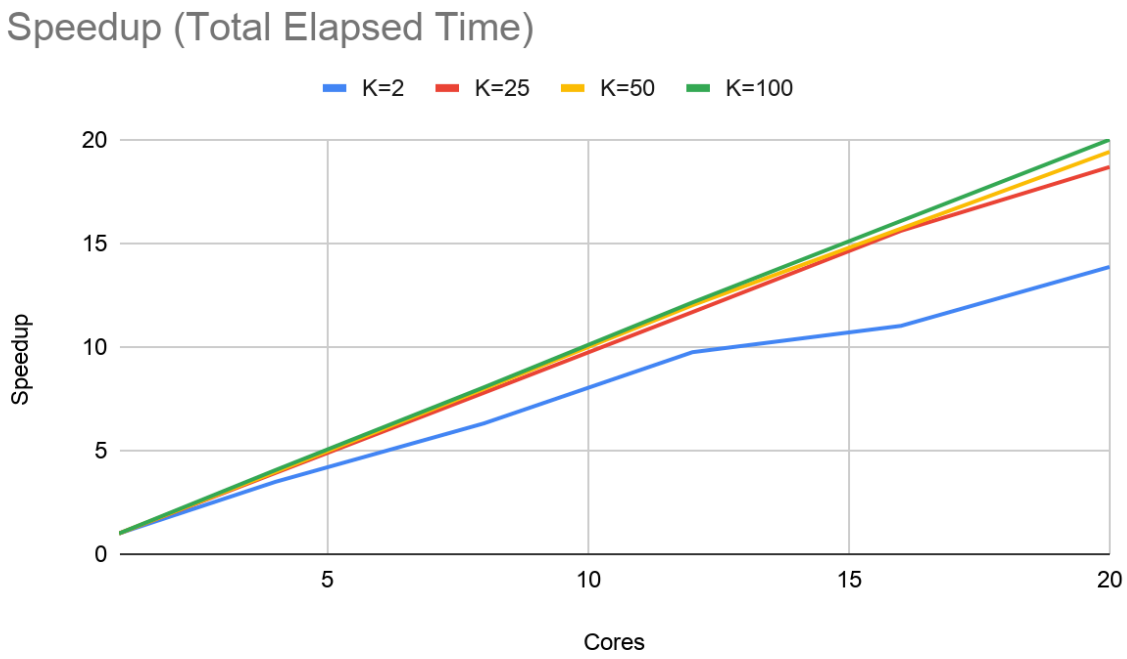


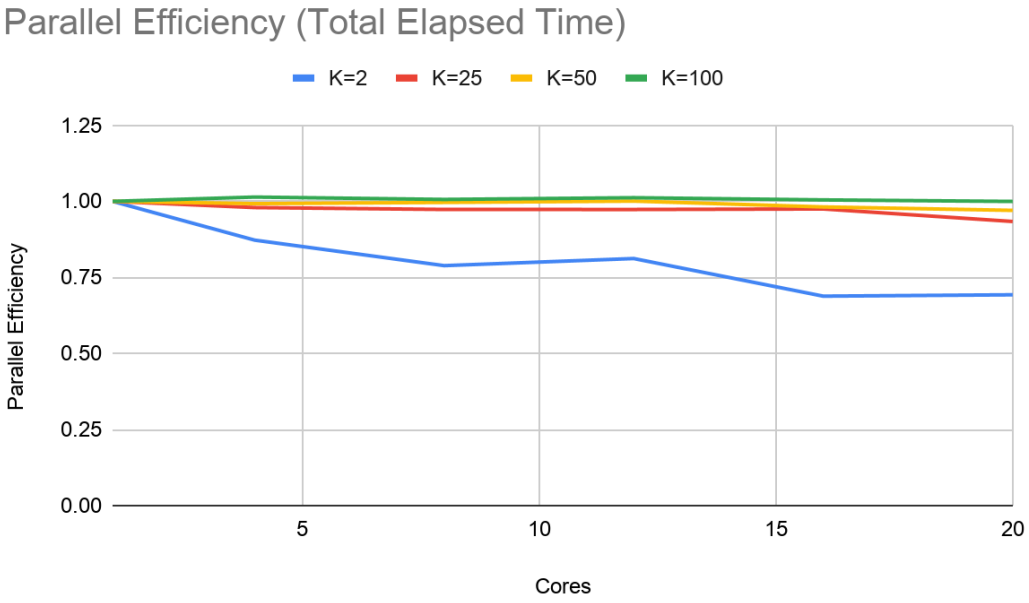


Because the time spent on this second part of the algorithm is regularly an order of magnitude shorter than the time spent on the distance calculations, the overall effect of this efficiency degradation is minimized.

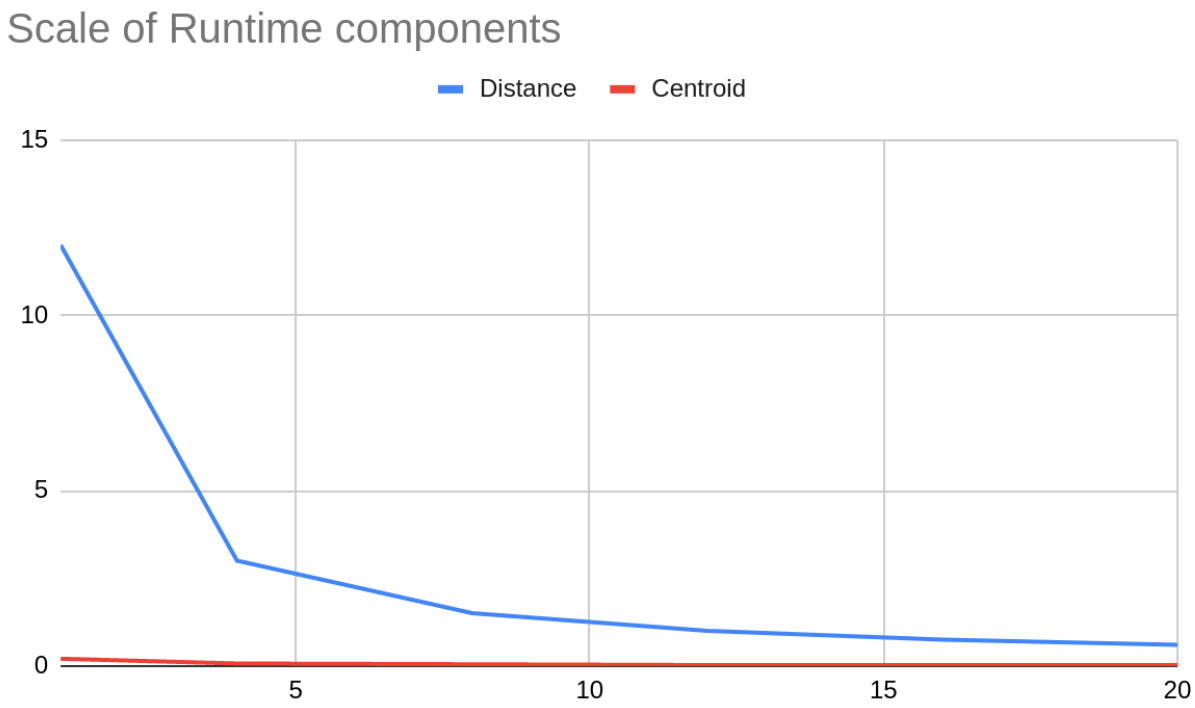
In general, cache efficiency is likely poor - a locality-aware approach to memory access would probably improve performance, and might change these dynamics.

Q5: How does the algorithm scale when k=2? Explain.
When k=2, the algorithm scales only moderately well, showing significantly poorer scalability than the algorithm when k > 25 or greater. Smaller values of k reduce the number of distance comparisons proportionally, and so decrease the discrepancy between the run times of the two parts of the algorithm. This, in turn, increases the impact of the performance degradation caused by communications during the centroid update phase.





Q6: How does the algorithm scale when k=100? Explain.
Scaling at k=100 is basically perfect, with parallel efficiencies marginally higher than 1 for most core counts. Here, the distance calculation compute time dwarfs the centroid adjustment time, reducing the impact of that component's poor scaling. This figure gives a clearer sense of the scale difference in those runtimes.



The fact that many of my runs show parallel efficiencies greater than 1 at k=100 was somewhat unsettling at first glance. This trend does not bear out for either of the algorithm's component parts, and appears to be an artifact of my decision to begin the overall timer before some roughly fixed-time-cost tasks (calculating and broadcasting per-rank data ranges). These tasks will not go any faster at scale, and result in this surprising effect on the overall times.

Activity 2: Multiple Nodes

T5: Total Response Time on 2 Cores

	K=2	K=25	K=50	K=100	Jobscript
24	0.0533	0.1564	0.2948	0.5385	jobscript.sh + sbatch.command
28	0.0488	0.1405	0.2489	0.4549	jobscript.sh + sbatch.command
32	0.029	0.1156	0.2112	0.3923	jobscript.sh + sbatch.command
36	0.0394	0.1106	0.1952	0.358	jobscript.sh + sbatch.command
40	0.0497	0.1022	0.1831	0.3237	jobscript.sh + sbatch.command

T6: Max cumulative time calculating distances on 2 cores

	K=2	K=25	K=50	K=100	Jobscript
24	0.0207	0.1329	0.2645	0.5053	jobscript.sh + sbatch.command
28	0.0231	0.1137	0.2267	0.4307	jobscript.sh + sbatch.command
32	0.0181	0.1004	0.1994	0.377	jobscript.sh + sbatch.command
36	0.0149	0.0874	0.1762	0.3347	jobscript.sh + sbatch.command
40	0.0127	0.0782	0.1569	0.3024	jobscript.sh + sbatch.command

T7: Max cumulative time updating means 2 cores

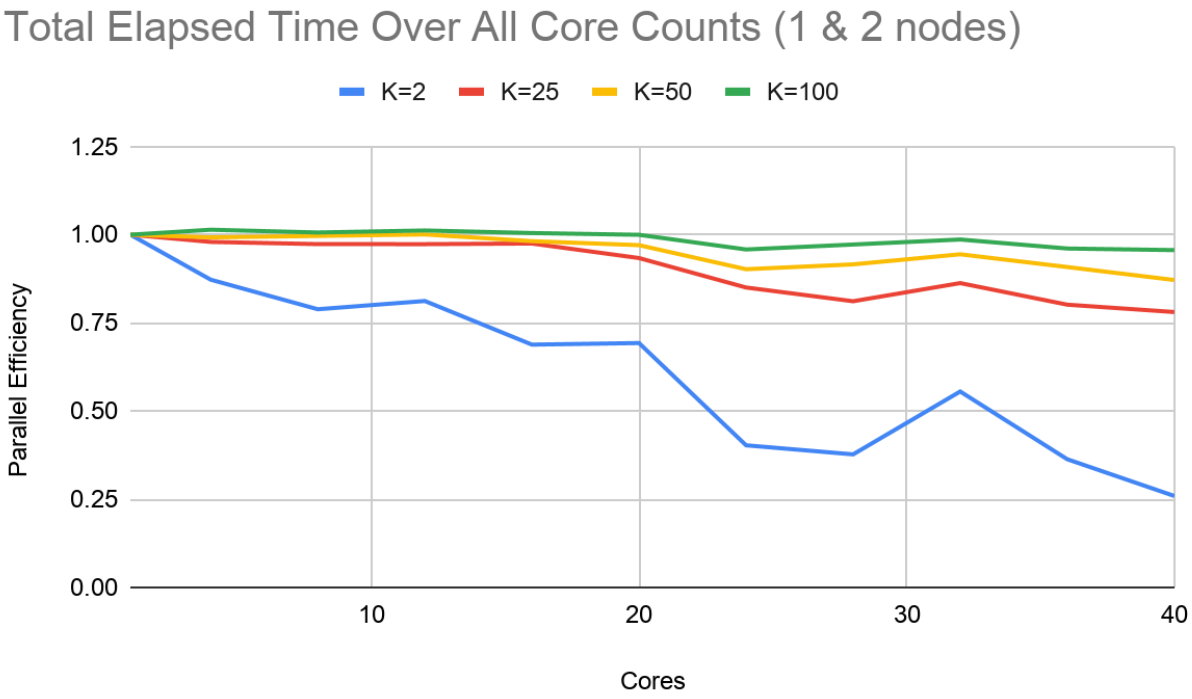
	K=2	K=25	K=50	K=100	Jobscript
24	0.0335	0.022	0.026	0.0335	jobscript.sh + sbatch.command
28	0.028	0.0265	0.019	0.0269	jobscript.sh + sbatch.command
32	0.0172	0.0215	0.0179	0.019	jobscript.sh + sbatch.command
36	0.022	0.0216	0.0207	0.0208	jobscript.sh + sbatch.command
40	0.034	0.0219	0.0219	0.0214	jobscript.sh + sbatch.command

T8: Speedup on 2 Cores

	K=2	K=25	K=50	K=100
24	1	1	1	1
28	1.092213115	1.11316726	1.18441141	1.183776654
32	1.837931034	1.352941176	1.395833333	1.372673974
36	1.352791878	1.414104882	1.510245902	1.504189944
40	1.072434608	1.530332681	1.610049153	1.663577386

Q7: On two nodes, how does the algorithm scale when K=2? Compare with the single node results.

Scalability over two nodes is poor when K=2. Total time on two nodes does not show a consistent trend, and more data would be useful, but only K=32 outperforms the time at K=20, and we can say that performance does not generally improve with the addition of a second node. Runtimes for K=2 are quite fast, so the cost of internode communication is important here.



Q8: On two nodes, how does the algorithm scale when K=100? Compare with the single node results.
The algorithm continues to scale very well on two nodes at K=100, likely because the relative cost of communication is outweighed by the computational costs. There is still some performance loss, but speedup is near perfect, and parallel efficiencies remain in the high .9s.

Q9: Under what conditions do you expect the k-means algorithm to perform well on multiple nodes (e.g., two or more)? When preparing your response, consider the following factors: the number of centroids, the size of the dataset, the data dimensionality, and the number of iterations.

K Means scales well when computational costs outweigh communication and memory access costs. In this report, I've focused on communication, as that seems the likeliest cause of performance degradation for this algorithm. High values of K, larger and higher-dimensional datasets are all likely to increase compute costs disproportionately. Increases in the number of iterations, while increasing overall runtime, are unlikely to impact scaling significantly. I think we can safely say that larger values of K and larger data sizes are likely to perform better on multiple nodes.