Parallel K-means Algorithm Using MPI

Fan Xie fx349@nyu.edu

1. Clustering Problem Description

1.1 Description

Given a set of data points X = {x1, x2, … xn}, where each data point is a d-dimensional vector, clustering algorithm aims to partition those data points into k sets C = {C1, C2, … Cn} so as to minimize the within-cluster sum of squares. Each point is closer to points inside than outside of its subset[1].

1.2 Mathematical Definition:

$$J_{k\text{-means}}(C_1, \ldots, C_k) = \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu(C_i))^2$$

$$= \min_{\mu_1, \ldots, \mu_k \in \mathcal{X}} \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i)^2$$

2. K-means Algorithm Description

2.1 Description

Finding the best Partitions in a clustering problem is actually an NP-hard problem[1], so we need some kind of approximation algorithm to solve it in practice. K-means is one of the heuristic algorithms to solve the clustering problem. It converges quickly to a local optimum and its performance heavily depends on the initialization step. Usually we run K-means algorithm several times to get a better result.

2.2 K-means Algorithm Pseudocode

Input: *a set of data points X = {x1, x2, .. xn} and a parameter k*

***Initialize**: Randomly choose initial k centroids*

Repeat until convergence:

> ***Assignment step:** Compute the distance from each point xi to each cluster centroid Cj and assign each point to the centroid it is closest to*
>
> ***Update step:** Recompute each centroid as the mean of all points assigned to it*

3. Implement K-means Algorithm Using MPI

3.1 Description

As the number of data points grows, the k-means algorithm can be very slow. This is mainly because in every iteration we need to traverse all the data points and calculate the distances between each data point and k centers. Assume that ***m*** is the number of iterations, ***n*** is the number of data points, ***k*** is the number of centers and ***d*** is the dimension of each data point. Then the time complexity of the K-means algorithm would be ***O(m*n*k*d)***. As the number of data points increases, the algorithm can become very slow. So we need to find some way to parallel it.

After careful analysis and profiling, I found that the assignment step is the most time-cost step in this algorithm. In the assignment step, each data point needs to calculate its distance between k centers and assign itself to a new parent, which is extremely slow in a single thread implementation. And this step is relatively easy to parallelize because each data point does their own works individually. No information except the k centers needs to be shared between different process[2]. So we can easily adopt a data parallelism method and use MPI to speed up the process. And below is the pseudocode of the parallel algorithm.

3.2 Pseudocode

***Initialize**:*

1. *Master process read all the data points and randomly initialize k centroids.*
2. *Master process divide those data points evenly and send them to the corresponding processes (MPI_Bcast, MPI_Scatter)*

Repeated until converge:

Assignment step

1. *Master process broadcast k centroids to all the processes(MPI_Bcast)*
2. *Each process calculate new centroids for all the data points that belong to it*

Update Step

3. *Each process tells the master process what's the new centroid of each data point (MPI_Gather)*
4. *Master process recomputes k new centroids base on the new assignment*
5. Master process checks whether the algorithm is converged or not

4. Results

I tested the performance and scalability of my implementation in NYU Prince HPC Cluster. Below are the details of the testing environment and the testing results.

4.1 Testing Environment[3]

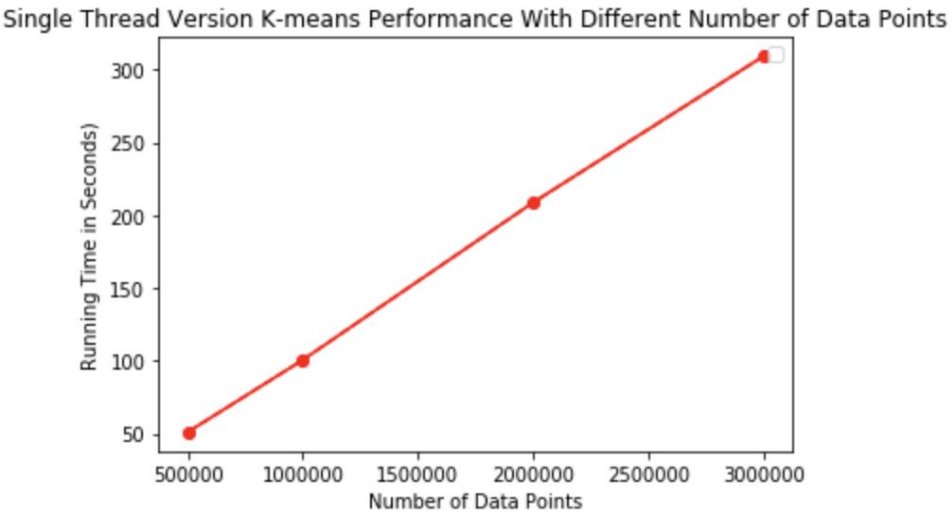
Cluster Name	NYU HPC Cluster Prince
Network	Infiniband
Total Nodes	430
Total CPU Cores	10,084
Total Memory	58TB
File System	ZFS
Operating System	CentOS 7.4
Compiler	G++ (GCC) 6.3.0
MPI Version	Openmpi
Compiler Options	-std=c++11 -O3 -march=native

4.2 Single Thread Version Testing

4.2.1 Single Thread Version With Different Number of Data Points

Fixed Parameters: K = 4, Number of MPI tasks = 1

Result:

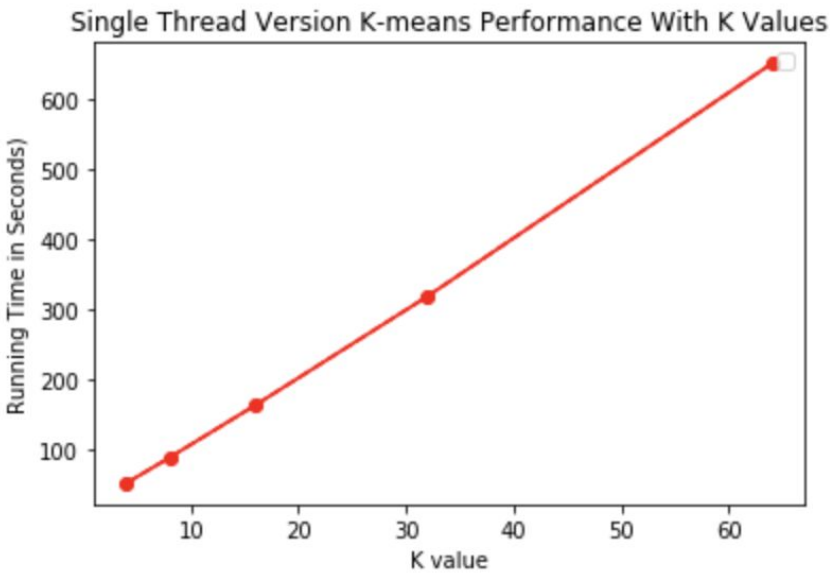


We can see that the running time of the single thread version k-means algorithm increase linearly as the number of data points increase. This is consistent with our analysis of time complexity.

4.2.2 Single Thread Version With Different K Value

Fixed Parameters: Number of Data Points = 500000, Number of Nodes = 1, Number of MPI tasks = 1

Result:



We can see that the running time also increases linearly as k-value increases.

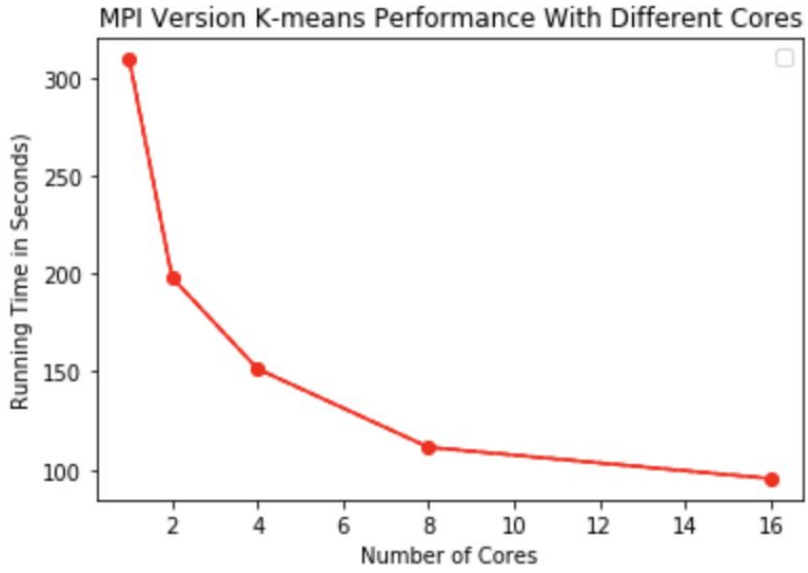
4.3 MPI Version Testing

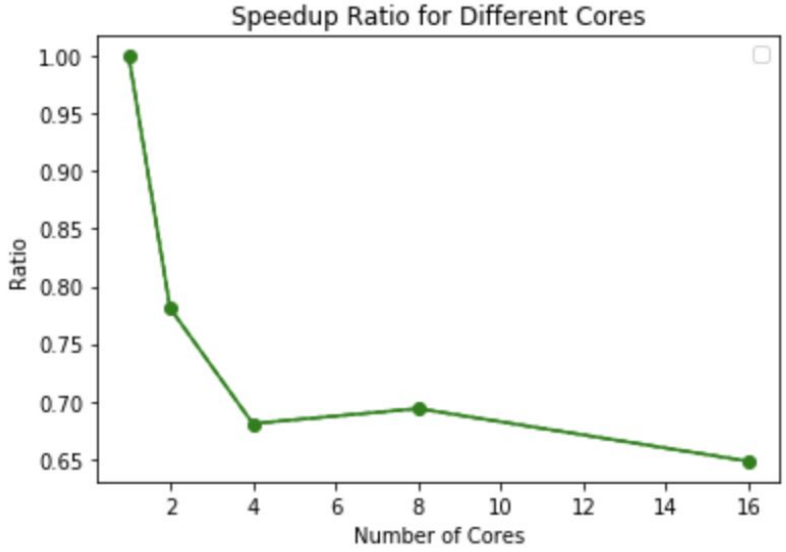
4.3.1 Scalability Within a Single Node

We test our algorithm within a single node with different numbers of cores to see the scalability of our implementation without the effect of communication cost.

Fixed Parameters: Number of Data Points = 3000000, Number of Nodes = 1, K = 4

Results:


MPI Version K-means Performance With Different Cores
Running Time in Seconds)
300
250
200
150
100
2
4
6
8
10
12
14
16
Number of Cores

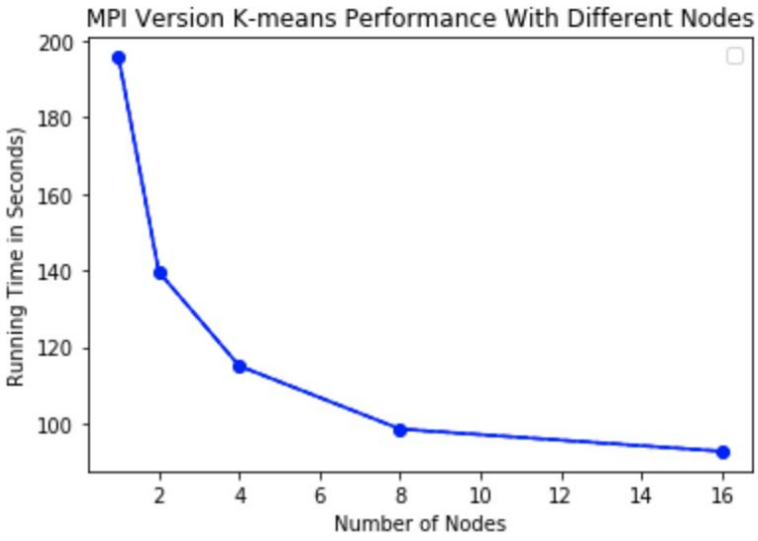

Speedup Ratio for Different Cores
Ratio
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
2
4
6
8
10
12
14
16
Number of Cores

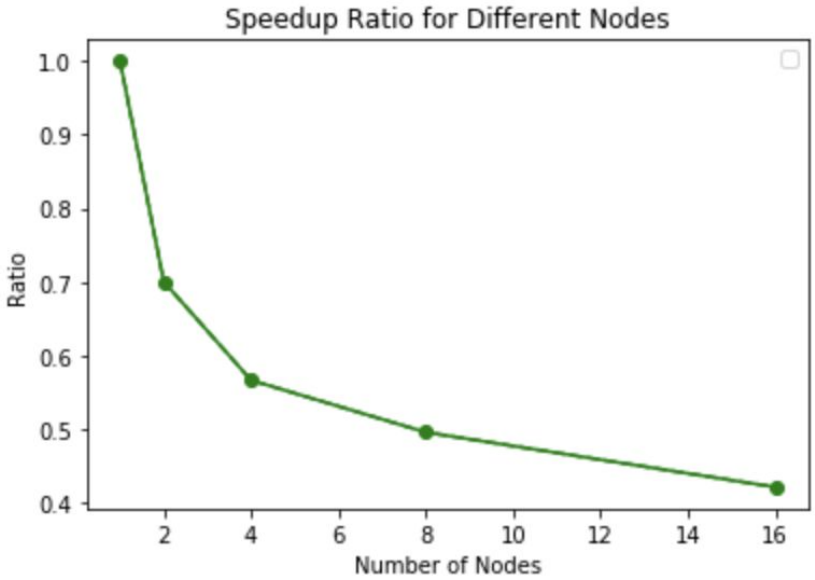
4.3.1 Scalability Across Multiple Nodes

We test our algorithm across multiple nodes to see how network communication cost affect our performance.

Fixed Parameters: Number of Data Points = 3000000, K = 4, Tasks Per Node = 2

Results:





5. Conclusion

From the performance and scalability tests, we found that our MPI implementation speeds up the K-means algorithm considerably. But we also found that as the number of MPI tasks increases, the speed up ratio is becoming lower and lower. This is because of two main reasons. The first reason is that the update step is a single thread operation and it becomes a new bottleneck as we parallelize the assignment step better and better. The second reason is that as the number of processes increases, the communication cost is also becoming more and more expensive. The communication cost also restricts our performance heavily.

6. Code Repository

https://github.com/xiefan46/homework2/tree/master/project/k-means

7. Reference

[1] https://en.wikipedia.org/wiki/K-means_clustering

[2] http://www.goldsborough.me/c++/python/cuda/2017/09/10/20-32-46-exploring_k-means_in_python,_c++_and_cuda/

[3] https://wikis.nyu.edu/display/NYUHPC/Clusters+-+Prince