EELE565 Parallel Processing
Parallel K-Means Implementation
Final Project Report

Nat Sweeney

December 2023

1 Overview

My final project involved implementing a parallel version of K-Means clustering applied to a hyperspectral image dataset. The parallel implementation was accomplished using multithreading in Julia, but the data preprocessing and figure generation was done in Matlab due to my familiarity with it. This allowed me to spend a majority of my time learning Julia with parallel processing as opposed to creating good figures and processing data. Computational efforts were performed on the Tempest High Performance Computing System, operated and supported by University Information Technology Research Cyberinfrastructure at Montana State University.

2 Math

The math behind the K-Means algorithm is least squared Euclidean distance and a mean calculation. The Euclidean distance is used to measure the distance between the vectors and the mean calculation is for calculating the value for the centers that are the intended output. The Euclidean distance is calculated with

$$S_i^{(t)} = \{x_p : \|x_p - m_i^{(t)}\|^2 \leq \|x_p - m_j^{(t)}\|^2 \forall j, 1 \leq j \leq k\} \tag{1}$$

The mean calculation is calculated with

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum x_j \in S_i^{(t)} x_j \tag{2}$$

All of these calculations were handled using the built in K-Means function from the Clustering.jl[1] library, as the focus of the project was to parallelize the algorithm as opposed to writing the algorithm from scratch. However, writing the algorithm from scratch would allow a level of algorithm parellelization as opposed to just data parallelization that was used for the sake of the project.

3 Data

The data used is the Salinas hyperspectral dataset[2]. This dataset is a hyperspectral image consisting of 54129 wavelength vectors representing a 204-band wavelength with 16 classes in the full image. Figure **1** shows the ground truth labels for the dataset. While this dataset does contain the ground truth for the samples, they aren't used outside of data preprocessing due to K-Means clustering being an unsupervised machine learning method. Because the data isn't particularly large in size it can be stored in DRAM, which is the main reason why multithreading was used as opposed to a multi-node implementation like MPI.

[1] https://juliapackages.com/p/clustering
[2] https://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes

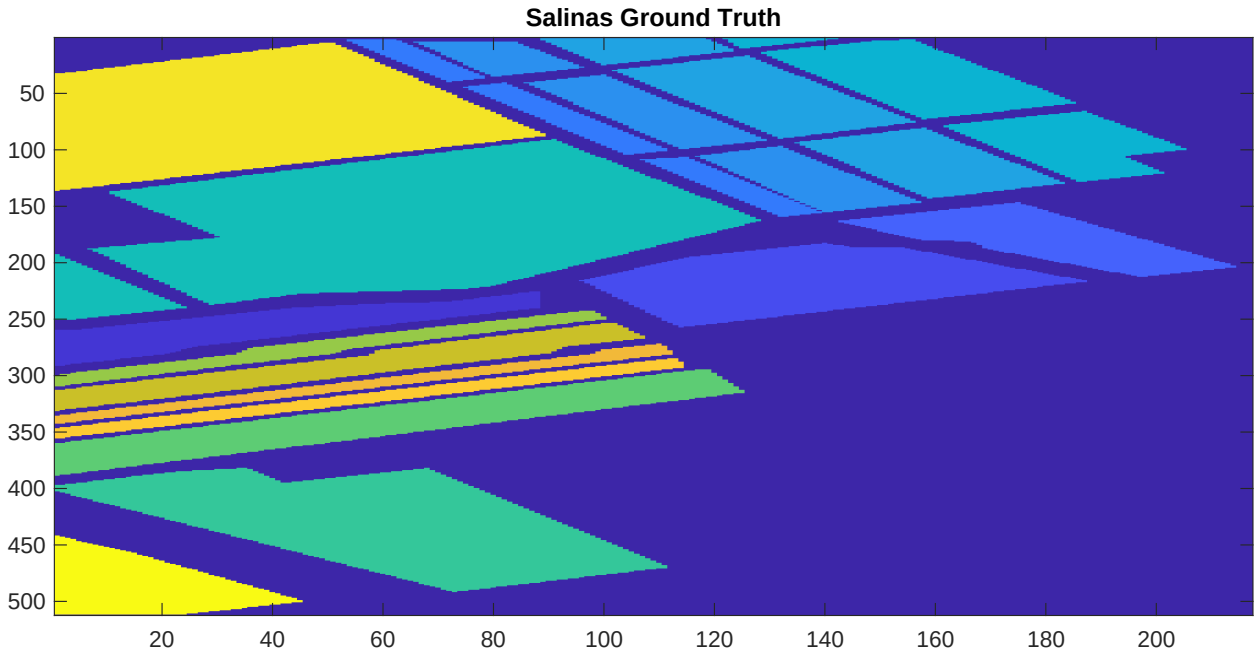

Figure 1: Ground truth for the Salinas dataset.

Table 1: Data Samples and Sizes

Dataset Size	Samples	MB
Small	1000	1000
Medium	2000	2000
Large	3000	3000
Full	4000	4000

3.1 Data Preprocessing

The original data comes in the form of a datacube with dimensions of 512x217x204 and a double datatype. In order to feed these into the K-Means function in Julia, the vectors were flattened into a N by 204 matrix, where N is the number of samples. For this project, varying subscenes of the dataset were used in order to gather performance info based on the size of the data. These samples were then shuffled to remove any possible correlation between neighbors and to allow each partition to get a distributed split of the data. Once shuffled, the data is saved to a .mat file, which can then be opened in Julia using the MAT.jl package[3]. When opened in Julia, the datatype stays the same, but for syntax purposes it is a Float64 in Julia: the equivalent of a double in Matlab.

4 Partitioning

The shuffled matrix of wavelength vectors was partitioned equally between the number of threads used for that iteration. If the data wasn't able to be equally partitioned, it would equally distribute between T-1 threads, and give the remaining vectors to the first thread in the segmentation. The partitioning process for this returns the indices that each thread is given, so the original data doesn't have to be written to another location in memory but can just be sliced into instead. Each thread is given the same set of tasks to calculate the K-Means of its subset of data and return the K centers that it calculated. All of the multithreading was handled using Julia's built-in multithreading functionality.

[3]https://github.com/JuliaIO/MAT.jl

5 Orchestration

The orchestration of the code involved sending each thread a subset of the data and calculating the K-Means for the subset. This resulted in K clusters for each thread, which are then communicated back to the main thread. Once all of these centers are together, a thread runs K-Means on the centers to calculate the center of the subsets' centers. The vectors are then assigned based on the newly found K centers. K increased depending on the size of the dataset used. In order to synchronize the K-Means of the individual subsets, a parallel for-loop was used, which handled the assignment of the data to each thread along with gating calculation of the final K-Means behind the other calculations being completed.

6 Software

The software for the project is split into two languages: Matlab and Julia. The Matlab scripts in the repository handle converting the datacube into a matrix, along with splitting it up into smaller subsets of the data so that the increase in data size could be used as a parameter for the benchmarking. Matlab also handles the figure generation from the results. The Julia scripts handle the actual calculations for the data, with the scripts being split into serial, parallel, and benchmarking. All three of these have different structures, with the serial simply loading data and calculating K-Means and the parallel using multithreading. The benchmarking scripts required a restructure of the code in order to use Julia's BenchmarkingTools.jl[4] package.

For the parallel processing, Julia was launched by specifying the number of threads using `julia --threads <thread #> <script name>`. You can also export an environmental variable prior to launching Julia using `export JULIA_NUM_THREADS = <thread #>`, but the `-t` flag overrides the environmental variable. In order to utilize the multiple threads for the for-loop used, `Threads.@threads for ...` was specified prior to the for loop, which allows Julia to handle the distribution between the for-loop iterations to the threads. One important note for using parallel for-loops is to have an independent variable handle indexing for anything as opposed to using the thread ID with `Threads.threadid()`. This is due to the fact that the threads could theoretically change tasks, or one thread could not do anything while another handles double duty. In the for-loop, the data writes to slices of a pre-allocated array, thus avoiding a datarace as the threads don't have to to try and write to the same memory locations while another reads or writes to it. As far as parallel processing goes, that's all there is to utilizing multithreading in Julia. Instead of using a parallel for-loop, the `@spawn` and `fetch` commands can be used to send tasks and retrieve data to and from threads.

7 Results

The end results of this implementation show that the increase in threads degrades performance until the total number of threads reaches 64, in which it is almost the exact same as the serial K-Means implementation. While this issue could be that the algorithm is simply a poor fit for parallelization, it could also be an inefficient implementation based on user error. The parallel for-loop could be adding to a big increase in latency and the use of spawn and fetch could increase the performance of the implementation. It also took some troubleshooting in order for the code to actually run on multiple threads on Tempest, requiring an environmental variable for Open BLAS to be initialized to `OPENBLAS_NUM_THREADS = 1`. For the parameter sweep of the implementation, varying data sizes and thread counts were used, with the data roughly doubling with each increase and the threads increasing by a power of 2 with each iteration. Figure **2** shows the runtimes relative to the thread count for varying sizes of datasets. The benchmarking for each of the iterations used 100 samples in order to gather the median, mean, and standard deviation. The figure presents the mean of the runtimes with the standard deviation being represented by the vertical bar at each point. The timing for the data loading is also presented, and in general was constant for each size of the dataset. For the true results, Figure **3** shows the spatial pixel assignment between the serial and parallel results for 64 threads, although all of the different values for the threads give similar results.

[4] https://juliaci.github.io/BenchmarkTools.jl/stable/

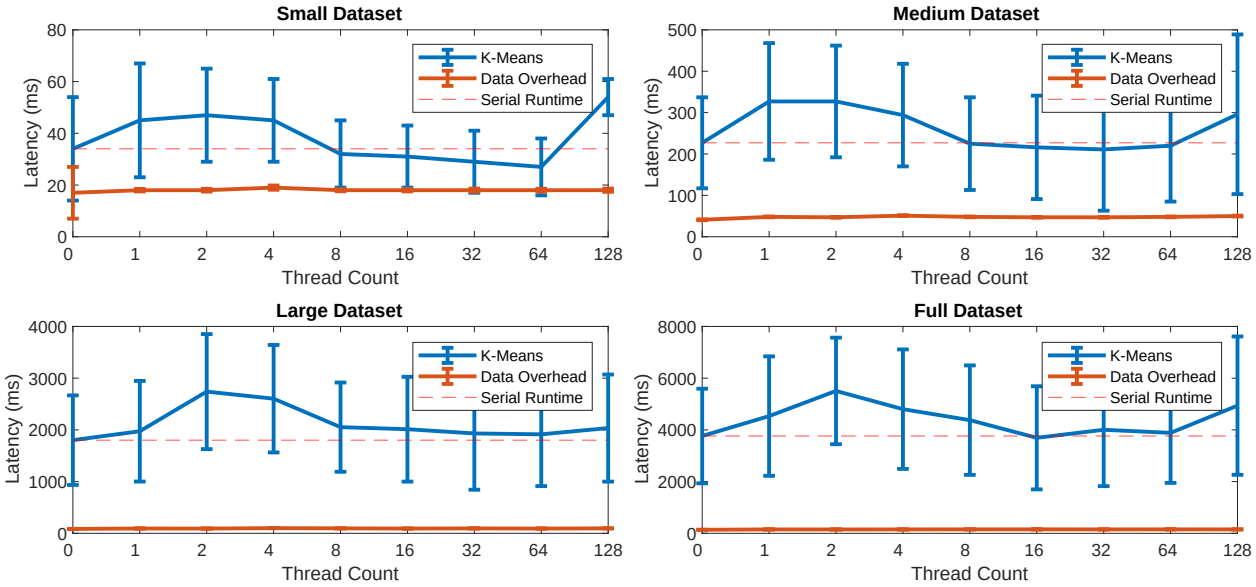

Figure 2: Runtime results over varying sized datasets as the number of threads increase. 0 threads is true serial while 1 thread is using the multithreading with only 1 thread

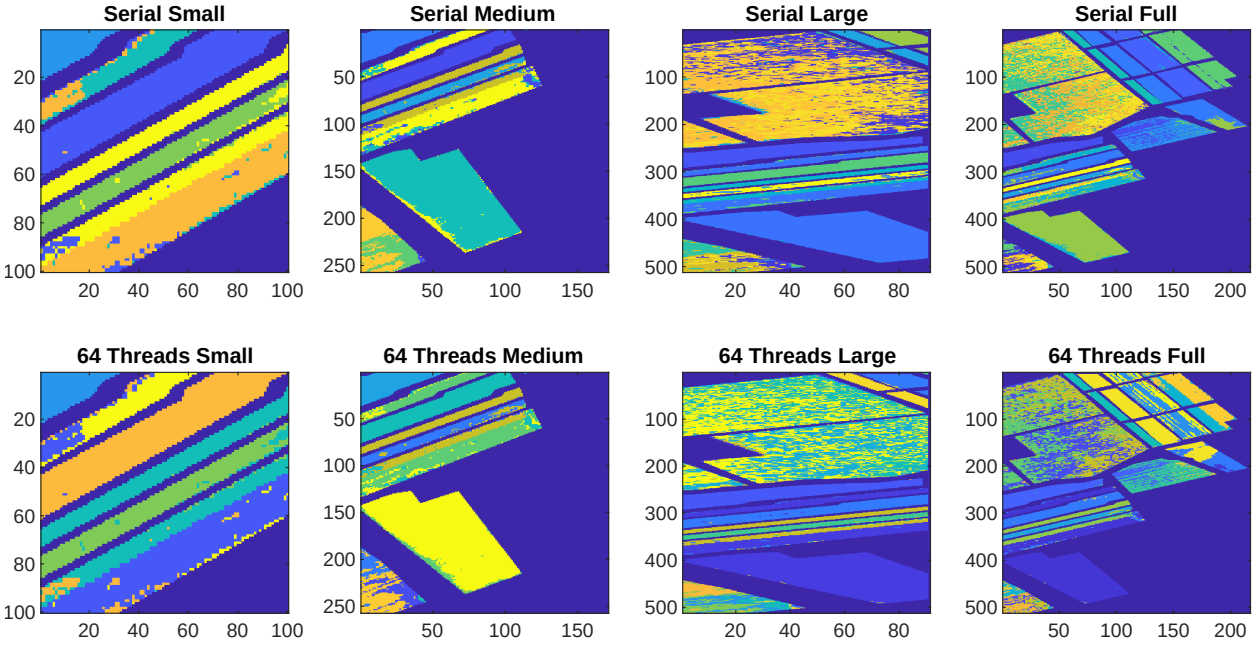

Figure 3: Assignments for serial and 64 thread implementations.

Overall I enjoyed the project. I think the poor results could either be due to the algorithm not being a good algorithm to parallelize, or the implementation has too much latency in how the parallel processing was handled. If I had more time I would have liked to try and benchmark the parallel implementation on a low level to determine the bottleneck, but it would have required a slight restructure of the code and taken a significant chunk of time that I do not have.