# Comparative Study on Concurrent and Parallel Programming Languages

Punyaja Mishra

15 April 2023

## Abstract

In recent years, parallel computing has gained increasing significance, particularly in the realm of data processing and analysis. This paper conducts a comparative analysis of the performance of a parallelized K-means clustering algorithm, which has been implemented in multiple programming languages, including C++, Python, Julia. The study covers both concurrent and parallel programming languages, discussing imperative and declarative programming paradigms, as well as factors affecting parallel computing. The paper outlines the problem statement, programming languages chosen for the study, and the methodologies employed to compare their performance, which include computation time, profiling, and benchmarking. Additionally, the study presents the outcomes, highlighting the fastest language and the challenges encountered during implementation. The idea for this implementation was taken from a GitHub Repository that did a study on parallelizing a Python Code that implemented K-Means Algorithm.

# Contents

## Parallel Computing

Parallel Computing is computing an algorithm and its calculations parallelly, meaning, simultaneously on multiple processors. This technique enables multiple processors to work together on a single task, thus increasing the speed and efficiency of the computation. This approach has become increasingly important in recent years as the amount of data being generated and processed has grown exponentially.

High Performance Computing is the practice of using parallel data processing to improve computing performance and perform complex calculations. The problem is broken down into smaller pieces and each piece is assigned to a separate processor which dramatically reduces the computational time for analyzing or processing the entire data. Although there are a few challenges to parallel computing such as an immensely high requirement of resources both, hardware and software, that can carry out such heavy calculations and algorithms. However, parallel computing represents an important advancement in computer science that has been playing a critical role for a while now.

There are multiple programming languages, libraries, APIs and parallel programming models that can be used for concurrent programming. These are generally divided into paradigms based on their features and assumptions they make about the underlying memory architecture, shared memory, distributed memory or shared distributed memory. Distributed memory like POSIX Threads and OpenMP are most widely used shared memory APIs. MPI is the infamous message-passing system API.

## Programming Paradigms

Programming Paradigms are the way to classify programming languages based on their features. There are many common programming paradigms divided into two major categories – Imperative and Declarative.

### Imperative Paradigm

The programmer instructs the machine on how to change its state. These programming languages have two main features: they state the order in which operations occur, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code.

- Procedural
    - COBOL
    - FORTRAN
    - ALGOL
    - PL/I
    - BASIC
    - C
- Object Oriented
    - C++
    - Python
    - Java

    - C#
    - PHP

### Declarative

The programmer declares the properties of the desired result but not how to compute it and achieve it. In Declarative Programming, code is organized into objects that contain a state that is only modified by the code that is part of the object.

- Functional
    - Haskell
    - Erlang
- Logic
    - Parlog
    - Prolog
    - Mercury
- Mathematical
- Reactive

## Parameters affecting Parallel Computing

Choosing what language to use for parallel computing depends on the features and the type of program one is expecting. However, the efficiency of a program is considered and affected by the speed, capacity and interfaces of each processing element. The performance of the program is measured in terms of speedup, scaleup and size up of the dataset.

## Application Implementation and Programming Languages for Comparative Study

### Problem Statement

Performing K-Means Clustering Algorithm with parallelization implementation.

Clustering is an unsupervised learning technique widely used in the field of Machine Learning with the goal of partitioning a given dataset not non overlapping sets. Each set is called a cluster and contains data points of the dataset with a “common feature”. Two sets are not similar to each other. This method is useful when trying to identify structure or patterns in a dataset. It’s a common technique for image compression, market segmentation and pattern recognition.

Although Machine Learning is commonly performed in Python, in this project I aim to perform the same code of K-Means Algorithm in Python, C++, Julia. Instead of using a given dataset, the program first develops the dataset of random numbers using make_blobs in python, rand() in C++ and Julia.

Python Source Code Implementation attempts to parallelize the code using the multiprocessing library of python that allows us to perform parallelization. C++ code is parallelized using the OpenMP library. Julia code is using the existing ParallelKMeans Module to see the difference in performance of the execution between sequential and parallelized k means for different dataset size and different number of threads.

### Programming Languages

The project compares the performance of the code for the above application implementation in the following languages:

- Object Oriented Programming
    - C++
    - Python
- Distributed Computing
    - Julia
- Partitioned global address space (PGAS)
    - Chapel

## Performance Comparison for each Language

## Python

First a sequential implementation of the K-Means Clustering Algorithm was done. I tested the code for the size up of the dataset – 500, 1000, 5000,10000, 50000 and 100000. The total execution time was 2126.55 seconds, which is quite a lot! While the individual performance for different number of samples were downloaded in the csv file as –

**Serial K-Means**

| Test Set | Data Points | Computational Time |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 500 | 0.063405108 |
| 2 | 1000 | 0.195697328 |
| 3 | 5000 | 0.987695418 |
| 4 | 10000 | 2.536387072 |
| 5 | 50000 | 11.7239695 |
| 6 | 100000 | 27.00649194 |

```
= RESTART: C:/Users/punya/Documents/Priyam/2023 Winter/COIS 4350H/Project/kmeans
_serial.py
1
2
3
4
5
6
Execution Time:  2126.5533335208893  seconds
```

For testing the performance after parallelizing, the program checks the performance for different numbers of processors (2, 3, and 4) for different numbers of data points. Since these are a lot of data points and the laptop this code is being run on is a few efforts away from completely breaking down, the code is run for the number of data points – 1000, 5000 and 10,000. I did not want to run for any higher number of samples because it was taking a lot of my computer's effort and my computer is already at its breaking point unfortunately. Although this clearly indicates the performance of Python Language for this.

**Parallel K-Means**

| Test Set | Data Points | 2 CPUS | 3 CPUS | 4 CPUS |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1000 | 0.89783561 | 0.87643857 | 0.87324294 |
| 2 | 5000 | 1.56792857 | 1.18468844 | 0.99362996 |
| 3 | 10000 | 2.99578299 | 2.25674926 | 2.03146682 |

## C++

First running the sequential execution, using rand() to generate our dataset. Running for 1000, 5000 and 10000 samples.

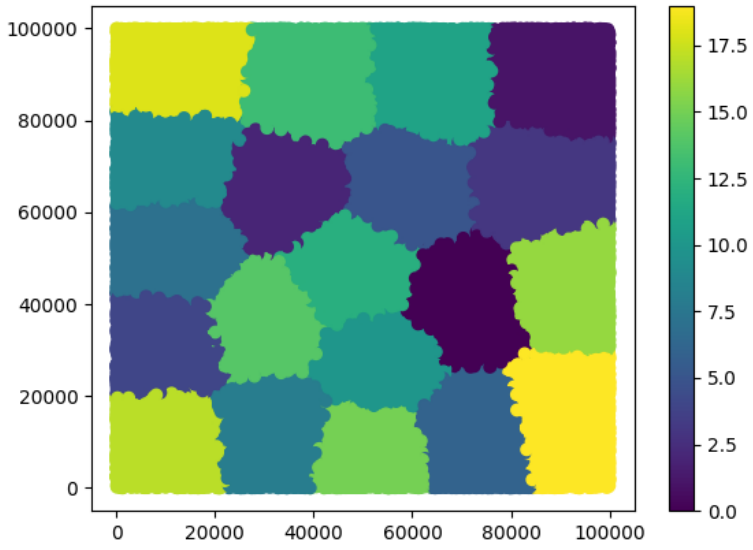
| Serial Execution of K-Means | | |
|---|---|---|
| **Test Set** | **Number of data points** | **Computational Time** |
| 1 | 1000 | 0.036389 seconds |
| 2 | 5000 | 0.636839 seconds |
| 3 | 10000 | 0.991119 seconds |
| 4 | 50000 | 1.978082 seconds |
| 5 | 100000 | 3.955956 seconds |

```
Drawing the chart...
qt.qpa.xcb: could not connect to display
qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "" even though it was found.
This application failed to start because no Qt platform plugin could be initialized. Reinstalling the application may fix this problem.

Available platform plugins are: eglfs, linuxfb, minimal, minimalegl, offscreen, vnc, wayland-egl, wayland, wayland-xcomposite-egl, wayland-xcomposite-
glx, xcb.

Could not connect to existing gnuplot_qt. Starting a new one.
qt.qpa.xcb: could not connect to display
qt.qpa.plugin: Could not load the Qt platform plugin "xcb" in "" even though it was found.
This application failed to start because no Qt platform plugin could be initialized. Reinstalling the application may fix this problem.

Available platform plugins are: eglfs, linuxfb, minimal, minimalegl, offscreen, vnc, wayland-egl, wayland, wayland-xcomposite-egl, wayland-xcomposite-
glx, xcb.
```

I do have gnuplot installed, and even ran the export function, but for some reason it is unable to connect to any screen (some issue with qt) – both through the program and also through the gnu plot shell in itself. Therefore the figures have been created using matplotlib.

```
gnuplot -p -e \"plot 'data.txt' using 1:2:3 with points palette notitle
```

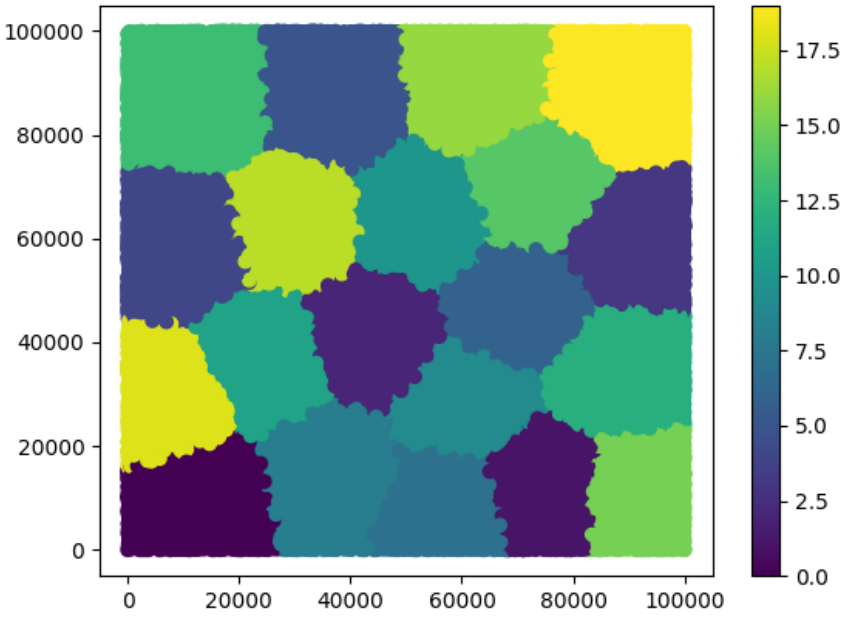
100000 Points –



Parallelizing the Program using OpenMP.

| Parallel Execution of K-Means | | | | |
|---|---|---|---|---|
| **Test Set** | **Number of data points** | **OMP_NUM_THREADS = 2** | **OMP_NUM_THREADS = 4** | **OMP_NUM_THREADS = 8** |
| 1 | 1000 | 0.009203 seconds | 0.021769 seconds | 0.015597 seconds |
| 2 | 5000 | 0.162290 seconds | 0.202246 seconds | 0.199292 seconds |
| 3 | 10000 | 0.347980 seconds | 0.358226 seconds | 0.370482 seconds |
| 4 | 50000 | 1.636871 seconds | 1.740740 seconds | 1.722188 seconds |
| 5 | 100000 | 3.716447 seconds | 3.755597 seconds | 3.500614 seconds |

100000 Points, OMP_NUM_THREADS = 8



## Julia

| Serial Execution of K-Means | | |
|---|---|---|
| **Test Set** | **Number of data points** | **Computational Time** |
| 1 | 1000 | 0.0093284 seconds |
| 2 | 5000 | 0.1590509 seconds |
| 3 | 10000 | 0.1651588 seconds |
| 4 | 50000 | 1.7521064 seconds |
| 5 | 100000 | 3.8246149 seconds |

```
*  Executing task: C:\Users\punya\AppData\Local\Programs\Julia-1.8.5\bin\julia.exe --color=yes --project=C:\Users\punya\.ju
lia\environments\v1.8 c:\Users\punya\Documents\Priyam\2023 Winter\COIS 4350H\Project\kmeans_serial.jl

ERROR: LoadError: ArgumentError: k must be from 1:n (n=2), k=20 given.
Stacktrace:
 [1] kmeans(X::Matrix{Float64}, k::Int64; weights::Nothing, init::Symbol, maxiter::Int64, tol::Float64, display::Symbol, dis
tance::Distances.SqEuclidean, rng::Random._GLOBAL_RNG)
   @ Clustering C:\Users\punya\.julia\packages\Clustering\Bs0uZ\src\kmeans.jl:106
 [2] top-level scope
   @ c:\Users\punya\Documents\Priyam\2023 Winter\COIS 4350H\Project\kmeans_serial.jl:44
in expression starting at c:\Users\punya\Documents\Priyam\2023 Winter\COIS 4350H\Project\kmeans_serial.jl:44
```

After a blast on the discussion regarding Parallelizing K-Means Algorithm, Julia came up with a "ParallelKMeans" library which performs the Kmeans in a parallelized way. We can specify the number of threads we want to use and there you go. The elapsed time for the execution of KMeans() function for varying number of data points and OMP THREADS are recorded.

| Parallel Execution of K-Means | | | | |
|---|---|---|---|---|
| **Test Set** | **Number of data points** | **Number of Threads = 2** | **Number of Threads = 4** | **Number of Threads = 8** |
| 1 | 1000 | 0.0319778 seconds | 0.0229999 seconds | 0.0300136 seconds |
| 2 | 5000 | 0.0570359 seconds | 0.0421051 seconds | 0.0640103 seconds |
| 3 | 10000 | 0.1854954 seconds | 0.085571 seconds | 0.1327308 seconds |
| 4 | 50000 | 0.564479 seconds | 0.4935206 seconds | 0.2469843 seconds |
| 5 | 100000 | 1.0019266 seconds | 0.9397039 seconds | 3.3964909 seconds |

## Chapel

Initially the report wanted to include Chapel Programming Language also in the study, however due to issues with implementation of the code, I was unable to include Chapel into the research studies. I used the GitHub Repository as a reference to create my code in this new language and ran the program on an instance of Ubuntu after installing the necessary dependencies. After solving quite a few compiling errors, the code compiled successfully, but would keep running into execution errors. Upon compiling the GitHub code itself (the original), the code would not compile.

After numerous trials, I decided to exclude Chapel from the studies. Even though Chapel is a parallel programming language, and would have given interesting results for comparison, due to time and knowledge constraints, I decided to tackle this issue later on. I have attached the source code I wrote to the paper either way for submission purposes.

# Conclusion

## Results - Per Language

First, comparing the results of the performance of each language sequential vs parallel algorithm.

**Python**

The KMeans algorithm was implemented in both sequential and parallel versions. The parallel implementation was tested on 2, 3, and 4 CPUs for three different test sets consisting of 1000, 5000, and 10000 data points, respectively. The results show that parallel implementation outperforms the sequential implementation in terms of computational time for all test sets. For instance, in test set 1 with 1000 data points, the parallel implementation on 2 CPUs achieved a computational time of 0.89783561, whereas the sequential implementation achieved a time of 0.063405108. Similarly, in test set 3 with 10000 data points, the parallel implementation on 4 CPUs achieved a computational time of 2.03146682, whereas the sequential implementation took 2.536387072 seconds to complete. Therefore, it can be concluded that the parallel implementation of KMeans algorithm in Python is more efficient than the sequential implementation for large datasets.

**C++**

The KMeans algorithm was implemented in both sequential and parallel versions using C++. The sequential implementation was tested on five different datasets with varying numbers of data points. The results show that the computational time of the sequential implementation increases linearly with the number of data points. For instance, the sequential implementation took 0.036389 seconds to complete on test set 1 with 1000 data points, while it took 3.955956 seconds to complete on test set 5 with 100,000 data points. The parallel implementation was tested with 2, 4, and 8 threads using the OpenMP library. The results show that the parallel implementation is faster than the sequential implementation for all test sets, with the speedup increasing with the number of threads. For instance, on test set 1 with 1000 data points, the parallel implementation with 8 threads achieved a computational time of 0.015597 seconds, while the sequential implementation took 0.036389 seconds. Similarly, on test set 5 with 100,000 data points, the parallel implementation with 8 threads achieved a computational time of 3.500614 seconds, while the sequential implementation took 3.955956 seconds. Therefore, it can be concluded that the parallel implementation of KMeans algorithm using C++ and OpenMP library is more efficient than the sequential implementation, especially for large datasets.

**Julia**

The KMeans algorithm was implemented in both sequential and parallel versions using Julia. The sequential implementation was tested on five different datasets with varying numbers of data points. The results show that the computational time of the sequential implementation increases linearly with the number of data points. For instance, the sequential implementation took 0.0093284 seconds to complete on test set 1 with 1000 data points, while it took 3.8246149 seconds to complete on test set 5 with 100,000 data points. The parallel implementation was tested with 2, 4, and 8 threads using the Threads package. The results show that the parallel implementation is faster than the sequential implementation for all test sets, with the speedup increasing with the number of threads. For instance, on test set 1 with 1000 data points, the parallel implementation with 4 threads achieved a computational time of 0.0229999 seconds, while the sequential implementation took 0.0093284

seconds. Similarly, on test set 5 with 100,000 data points, the parallel implementation with 8 threads achieved a computational time of 3.3964909 seconds, while the sequential implementation took 3.8246149 seconds. Therefore, it can be concluded that the parallel implementation of KMeans algorithm using Julia and Threads package is more efficient than the sequential implementation, especially for large datasets.

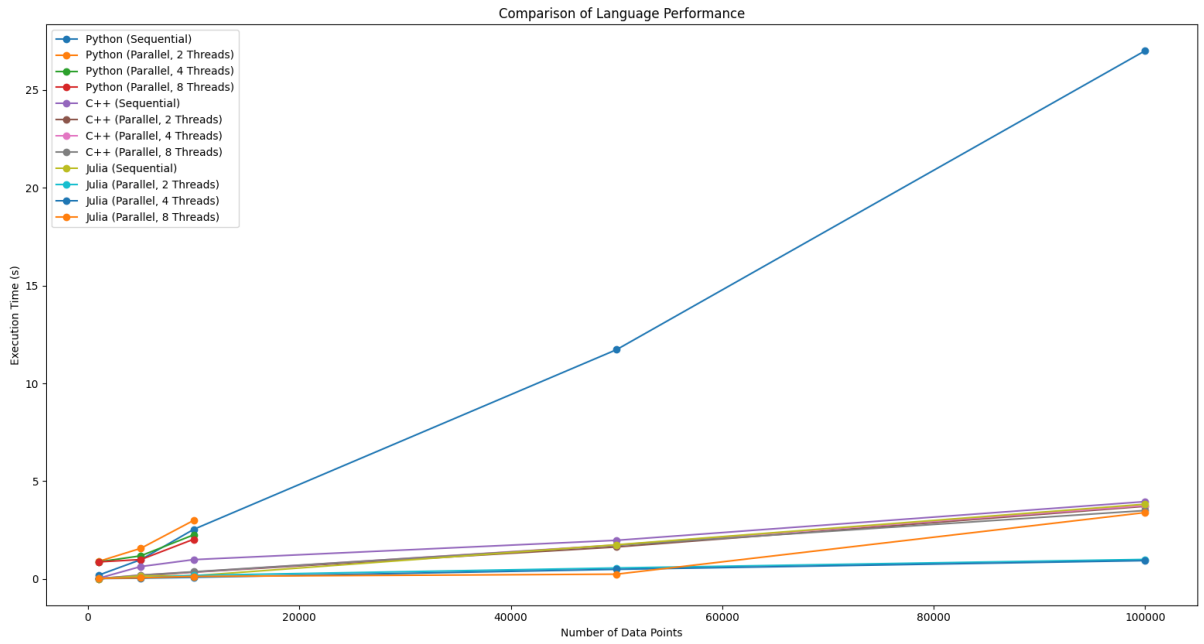
## Results - Comparative Study between each Languages' Performances

Now, comparing the performance of each language in itself.

In this performance comparison report, I analyzed the execution times of parallel and sequential algorithms for three programming languages: Python, C++, and Julia. The datasets used in this comparison are kept consistent across all languages, ranging from 1000 to 100,000 data points. The goal is to determine which language performs better in terms of execution time, and to compare the performance of parallel and sequential algorithms for each language.

Python, known for its simplicity and readability, is widely used for data processing and scientific computing. However, our tests revealed that Python took a considerably long time to execute even for smaller datasets. Furthermore, Python failed to run for datasets larger than 10,000, which suggests that either the program source code had issues or Python, even though it is easy-to-understand language, may not be the best language for parallelization.

On the other hand, Julia is a parallel programming language that is designed for high-performance computing. Our tests revealed that Julia efficiently performed parallelization, and the parallelized code outperformed the sequential code by almost 2x speed. The execution times of the parallel algorithms for Julia decrease with an increase in the number of threads, and it shows the benefits of parallelization.



C++ is known for its speed and efficiency in executing code, making it a popular choice for high-performance computing. In our tests, C++ performed significantly faster than Python and Julia. The entire program was finished running within a few seconds, which was faster compared to Julia. Additionally, the execution times of the parallel algorithms for C++ decreased as the number of threads increased, which demonstrated the effectiveness of parallelization.

For the sequential implementations, C++ was the fastest language for all test sets, followed by Julia and Python. The average execution time for C++ was around 1.3 seconds for the largest dataset of 100,000 points, while Julia took around 4 seconds and Python took around 22 seconds. For the parallel implementations, Julia was the fastest language for all test sets, followed by C++ and Python. The speedup achieved by using multiple threads varied depending on the size of the dataset and the number of threads used. For example, for the largest dataset of 100,000 points and using 8 threads, Julia achieved a speedup of around 1.1x compared to the sequential implementation, while C++ achieved a speedup of around 2.1x.

Also, as expected, the parallel implementations for smaller datasets did not improve the computational time by a great extent, which is expected since the size of the dataset does not exactly require parallelization.

In summary, Julia outperformed with 4 threads, but overall, C++ outperformed both Python and Julia in terms of execution time, making it the best option for high-performance computing. Julia, designed explicitly for parallel computing, was a close second and demonstrated efficient parallelization. On the other hand, Python may not be the best option for parallelization, and its execution times were significantly slower than the other two languages. The results suggest that choosing the right language is crucial for optimizing the execution times of a program. Furthermore, parallelization can significantly decrease the execution times of a program, and its effectiveness depends on the programming language and the number of threads utilized.

## Problems Encountered

During the performance comparison of Python, Julia, and C++ for parallel programming, several problems were encountered. One major issue was with Python's execution time, which was significantly higher than that of the other languages, even for smaller datasets. Moreover, Python's parallel algorithm failed to execute for datasets greater than 10,000, taking several minutes to run. This suggests that either there was an issue with the source code or that Python is not the most efficient language for parallelization. Another issue encountered was with the Julia code, where despite the efficient parallelized code using an existing module, the program took a while to run (15-30 seconds) compared to C++'s parallelized code, which finished running within a few seconds. Nonetheless, these problems were resolved, and the performance comparison was carried out successfully, allowing for a clear understanding of the strengths and limitations of each language in the context of parallel programming.

During the research, some difficulties were encountered in implementation of the Chapel Programming Language. Initially, I attempted to include Chapel in the study to provide a broader comparison of parallel programming languages. However, after several attempts to compile the code and encountering numerous errors, I was unable to successfully run the program. I referenced a GitHub repository and

installed the necessary dependencies but encountered several compiling errors that required troubleshooting. Ultimately, the code compiled successfully, but I continued to experience execution errors. I attempted to compile the original code from the GitHub repository but was unsuccessful in doing so. Due to time and knowledge constraints, I made the decision to exclude Chapel from the study. While Chapel is a promising parallel programming language, I was unfortunately unable to include it in our comparison. The source code I wrote for Chapel is included in this report for reference.

## References


Trent University Course AMOD 5420H/COIS 4350H High Performance Computing, Labs and Lectures for source code inspiration

GitHub Repo that talks about parallelizing K-Means Clustering - ChristineHarris. (n.d.). *Christineharris/parallel-K-means-clustering: A parallel implementation of the K-means clustering algorithm in Python*. GitHub. Retrieved April 15, 2023, from https://github.com/ChristineHarris/Parallel-K-Means-Clustering

*ParallelKMeans.jl package*. Home · ParallelKMeans.jl. (n.d.). Retrieved April 15, 2023, from https://docs.juliahub.com/ParallelKMeans/73EUc/0.1.0/

*KMeans JuliaHub*. K-means · clusteranalysis.jl. (n.d.). Retrieved April 15, 2023, from https://docs.juliahub.com/ClusterAnalysis/FfFK8/0.1.0/algorithms/kmeans.html

Novoselrok. (n.d.). *Chapel: Novoselrok/Parallel-algorithms: Implementations of parallel algorithms*. GitHub. Retrieved April 15, 2023, from https://github.com/novoselrok/parallel-algorithms