Malik CMSC 502 Fall 2020 Research – KNN Parallelization Using MPI

Mohammad Maik
Computer Science Graduate Student
Virginia Commonwealth University
907 Floyd Ave. Richmond, Virginia, United States, 20136
`malikmui@mymail.vcu.edu`

***Abstract*— This paper was written with the intent to explore the performance speedup of a K Nearest Neighbor Implementation designed sequentially versus a distributed model with the distributed performance done via MPI, Message-Passing Interface. With a K equal to 5, It was determined that the speedup increase is nearly exponential for all number of processes. The KNN implementation in this case is that which features an array as the core data structure of the KNN algorithm as opposed to the more efficient KD-trees algorithm. The biggest difference being between going from 1 process to 4 processes. Beyond 4 processes the increase begins to plateau, however it is still quite a significant increase. The theoretical speedup based on Amdahl's law determined the parallelism of the code to be 20x at a 95% percentage of code that is parallelizable. The biggest performance gain was on the small dataset, with 16 processes there was a 18.333x speedup. For the medium dataset, with 16 processes there was a 15.04x speedup, and for the large dataset with 16 processes there was a 14.60x speedup. Furthermore, The execution fails when the number of processes is specified to be 1024, even though the calculated speedup at 1024 processes with a parallelism of 95% is 19.63x that of the sequential KNN, which is believed to be because the server in this case has a total number of 56 processors.**

I. INTRODUCTION

Within the realm of data mining, it is quite common to have datasets consisting of millions of instances. Many machine learning algorithms have become more and more computationally complex by nature to increase performance, however, this practice is very limited in regards to datasets of this nature. The K-nearest-neighbors, KNN, finds the k number of instances you specify to any given instance. As shown in Figure 1, it heavily relies on the assumption that items within a dataset with similar characteristics tend to be near each other and grouped when plotted in N dimensions of space. It is considered one of the simplest data classifying algorithms in that it's computational complexity is $O(n^2)$, where n is the number of instances for classifying n instances in a dataset with respect to n-1 distances when compared to the other distances in the dataset. The distances in most cases is the Euclidean Distance.

The Euclidean Distance is the straight line segment distance between two points in N dimensions. It then predicts the class of the data instance from the k neighbors it determined to be the closest and finds their respective classes.

Message-Passing Interface, MPI, is a message-passing library in C/C++ designed to distribute logic and computations over multiple processes or a distributed system and allowing them to communicate. Amdahl's law, which we used to calculate speedup, is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.

Iris Dataset is a very well known dataset hosted by UCL's Machine Learning repository and has about 150 instances with 4 classes and 9 attributes descrubing the datasets. However, this of which does not actively describe the performance capabilities of the MPI library and parallelization and therefore we tested on a multitude of datasets, many stemming from the repository library, such as, segment-test and segment-challenge, having 810 and 1500 instances respectively with 19 attributes and 4 classes. To test the scalability even further, two more datasets were curated to test the boundaries, both datasets had several attributes with an equal number of classes. The total instances for both of these datasets, further to be referred to as medium and large, respectively had, 4898 instances and 19592 instances.

In computer architecture, Amdahl's law is a formula which gives the theoretical speedup in latency of the execution of a task at a fixed workload that can be expected of a system whose resources are improved. This law yields a formula, shown in Figure 3, that can derive the theoretical speedup of a program based on the parallelism of the code base. This was further used to also calculate the asymptotic speedup as well as the speedup at each respective processor count versus the total theoretical speedup.

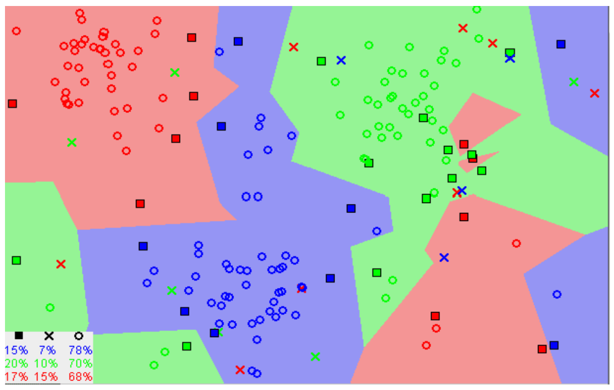
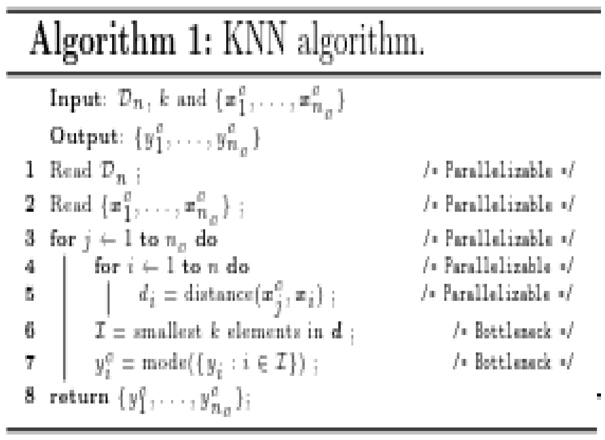


Fig. 1. KNN

I'd like to thank Dr. Preetam Ghosh at VCU for his guidance in this.



Algorithm 1: KNN algorithm.

```
Input: $\mathcal{D}_n$, $k$ and $\{x_1^o, \ldots, x_{n_o}^o\}$
Output: $\{y_1^o, \ldots, y_{n_o}^o\}$
1 Read $\mathcal{D}_n$ ;                                   /* Parallelizable */
2 Read $\{x_1^o, \ldots, x_{n_o}^o\}$ ;                     /* Parallelizable */
3 for $j \leftarrow 1$ to $n_o$ do                          /* Parallelizable */
4     for $i \leftarrow 1$ to $n$ do                        /* Parallelizable */
5         $d_i = \mathrm{distance}(x_j^o, x_i)$ ;           /* Parallelizable */
6     $\mathcal{I}$ = smallest $k$ elements in $\mathbf{d}$ ;   /* Bottleneck */
7     $y_i^o = \mathrm{mode}(\{y_i : i \in \mathcal{I}\})$ ;   /* Bottleneck */
8 return $\{y_1^o, \ldots, y_{n_o}^o\}$;
```

Fig. 2. KNN Algorithm's Inherent Parallelism

$$Speedup = \frac{1}{\left((1-P)+\frac{P}{N}\right)}$$

Fig. 3. Amdahl's law - Theoretical Speedup in Latency Calculation

II. Literature Review

Similar papers and research have been done in regards to parallelizing the KNN algorithm in order to demonstrate the power of big data in the cloud as well as in a three level structured hierarchy.

Big Data Analytics in the Cloud [1] uses the KNN algorithm in order to achieve and demostrate the power and capabilities of processing vast amounts of information into a reasonable finite time via various implementation of differing distributed computing frameworks. They implemented simply the KNN, KNN using OpenMP and MPI, OpenMP is a another programming model designed to divide loops across processors based on the number of processors available, similar to using threads however it is designed to handle everything for you. They also implemented the KNN algorithm on Spark on a Hadoop Cluster in Java, java being mentioned here as it is an inherently slower language than C and C++ therefore that alone can demonstrate a bit of slowdown, all onto different Virtual Machine clusters on the Google Cloud Platform. Their results yielded that which align with mine and that the KNN algorithm is nearly 10x faster than that of just sequential and it continues to scale to be quicker and quicker at larger scales. Their MPI implementation also was magnitudes faster than that of Spark as well however, they concluded that Spark may be a better implementation, not because of speed, but because of Spark's inherent architecture protecting from node failures and data replication which is better for mission critical systems and production environments.

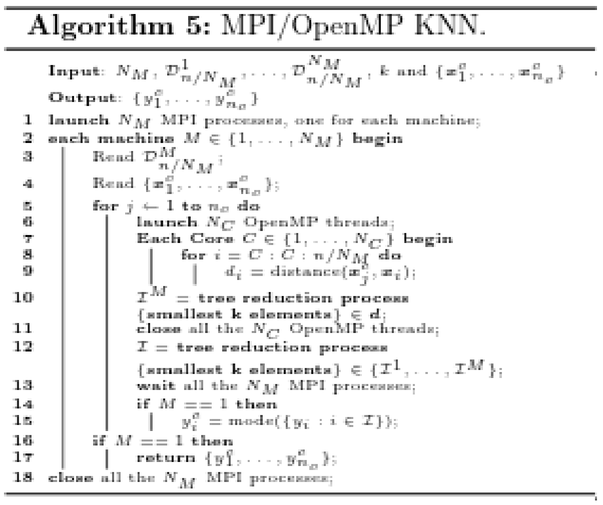
In a Parallel Implementation of KNN in 3 Levels: Threads, MPI, and the Grid [2], their goal was to implement the KNN algorithm in order to tackle the problem of data mining and classification of large amounts of data however, in their implementation this ws done by utilizing a three level architecture: Grid environments of multiprocessor computer farms being in tandem with, the MPI library, and POSIX threads. Grid Computing is the concurrent usage of different computing resources in different domains as a large scale batch queue, by combining these levels, each level's respective speedup yield combines to result in a large scale speedup. This architecture is well-designed as well as each level inherently supports the other levels flaws. Their implementation showed that MPI alone over sequential resulted in a 12x speedup and that the three level architecture resulted in a much larger speedup of 18x that of sequential, however, these were found with a k value equal to 1 which furthers the speedup as a majority vote is not necessary and finding the 1NN can be done in $O(n)$.

III. Methodology

Based on a given skeleton code of a 1NN, which just finds the single closest neighbor and guesses the data class of the instance based on that one instance's class, ie. a Majority voting algorithm was implemented in order to decide the class. The task was to design and implement a KNN, in this paper K=5. This KNN must first work in a single-threaded manner prior to a distributed implementation of the algorithm.

The difference between the single-threaded and the distributed method is that it requires passing the dataset to be divided up amongst the specified number of processes according with little overlap. This was handled by getting the number of processes and dividing the dataset's total number of instances by it, $count = ceil((float)(dataset- > num_instances()/(float)(size))$. This alone was not nearly enough as the start and stop points for each thread also needed to be found. This was found by multiplying rank with the number we had just found to get the start, $start = rank * count$. To get stop was a little more challenging because based on your implementation this may be slightly different. I chose to also utilize rank 0, or the master thread, to not only receive the answers from the other threads, but to also calculate some of the dataset. Therefore, this needed to be included in calculating stop, $stop = ((rank+1)*count)-1 > dataset- > num_instances() - 1?dataset- > num_instances()-1 : ((rank+1)*count)-1$. The euclidean distance was used to calculate the distance between 2 points in n-dimensions, in our case between each instance in the dataset. This was then stored in a tuple where the key are the indices, which was then sorted based on the distances and passed into a neighbors array that consists of the keys that are now sorted.

For the core data structure of the KNN, an array implementation was chosen here, as shown in Figure 2, as opposed to a k-d tree is based on a few key factors. First, simplicity in terms of sticking with the notion that KNN being the one of the least complex and most simple algorithms how easy using the MPI library can be to show speedup. This is to show that the impact is because of MPI and parallelism, not fancy tricks beyond the capabilities of the standard traditional programmer. Second, arrays being computationally efficient as well as how easy arrays are to code and parallelize based on the indices and portions of the code, allows it to be very easy to understand what block is being worked on by what processor or what computer. In terms of simplicity this allows a deeper understanding of what is going on as opposed to the numerous papers on making the KNN more efficient by using a k-d tree allows this paper to have a more substantial use case in demonstrating the usability of the MPI library at any level.



Algorithm 5: MPI/OpenMP KNN.

```
Input: $N_M$, $\mathcal{D}^1_{n/N_M}, \dots, \mathcal{D}^{N_M}_{n/N_M}$, $k$ and $\{x^o_1, \dots, x^o_{n_o}\}$
Output: $\{y^o_1, \dots, y^o_{n_o}\}$
launch $N_M$ MPI processes, one for each machine;
each machine $M \in \{1, \dots, N_M\}$ begin
    Read $\mathcal{D}^M_{n/N_M}$;
    Read $\{x^o_1, \dots, x^o_{n_o}\}$;
    for $j \leftarrow 1$ to $n_o$ do
        launch $N_C$ OpenMP threads;
        Each Core $C \in \{1, \dots, N_C\}$ begin
            for $i = C : C : n/N_M$ do
                $d_i = \text{distance}(x^o_j, x_i)$;
        $\mathcal{I}^M$ = tree reduction process {smallest k elements} $\in$ d;
        close all the $N_C$ OpenMP threads;
        $\mathcal{I}$ = tree reduction process {smallest k elements} $\in \{\mathcal{I}^1, \dots, \mathcal{I}^M\}$;
        wait all the $N_M$ MPI processes;
        if $M == 1$ then
            $y^o_i = \text{mode}(\{y_i : i \in \mathcal{I}\})$;
    if $M == 1$ then
        return $\{y^o_1, \dots, y^o_{n_o}\}$;
close all the $N_M$ MPI processes;
```

Fig. 4. KNN in MPI

Based on this neighbors array, I get the classes of each respective neighbor and store those instances into a map alongside a count of how many times each class appears. This count was used to calculate the mode and whichever class had the mode was returned as the data class for the instance. The only other difference other than chunking the data is here, where instead of returning the array, the MPI library is used to send the mode to the array which is currently waiting in the first process for the other processes to finish and send their respective class value calculations.

IV. RESULTS

As shown in figures 5, 6 and 7, the sequential, single-threaded KNN, actually slows down when the number of processes is increased, most notably around 8 processes, whereas the distributed version rapidly increases in speed. The biggest gains for the distributed version on all datasets is going from 1 process to 2 it basically does in fact cut the length of time taken, or latency, by half. The next biggest gain is from 2 processes to 4 processes, which also cuts the time taken in half. Beyond 4 processes the distributed slowly starts to decrease in speedup and begins to gradually plateau. With respect to the small dataset, with a total of 336 instances, the increase at 16 processes was 18.33x the sequential version. With respect to the medium dataset, with a total of 4,912 instances, the increase at 16 processes was 15.04x that of the sequential version. With respect to the large dataset, with a total of 19,606 instances, the increase at 16 processes was 14.59x that of the sequential version. This is in accordance with Amdahl's law, which calculates the theoretical speedup in latency of the execution of a task at a fixed workload that can be expected of a system whose resources are improved. As shown in *Figure 3* the speedup was calculated. To calculate the theoretical speedup you assume P/N to be approaching ∞, thus determining max speedup at 95% code parallelization to be 20x that of the single-threaded, sequential version. In figure 8, the sequential vs MPI chart shows the speedup difference visually and shows the average speedup is 10x that over sequential and continues to grow logarithmically over sequential. Figure 9 follows that up with the exact timings of each run. Each timing is an average of 5 runs to ensure no outliers were found.

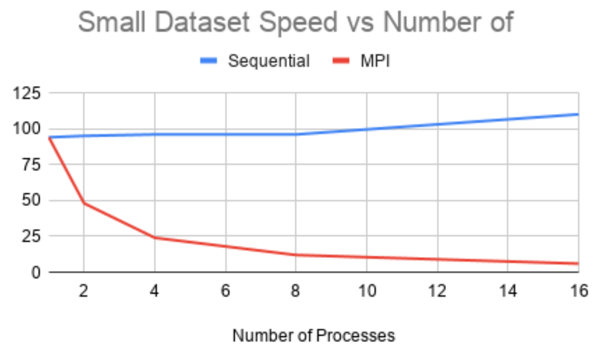


Fig. 5. Time vs. Number of Processes - Small Dataset

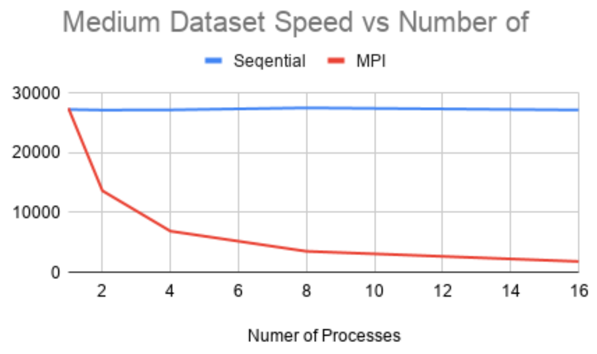


Fig. 6. Time vs. Number of Processes - Medium Dataset

V. CONCLUSIONS

The importance of the work is to demonstrate the capabilities of not just the MPI library, but that of demonstrating how much of a speedup one can attain with very little effort/additional lines of code. The KNN in its single-threaded state takes around 8 minutes to complete whereas the distributed version takes at most 4 minutes

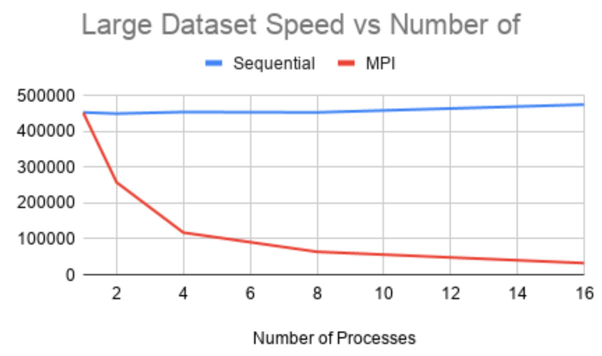
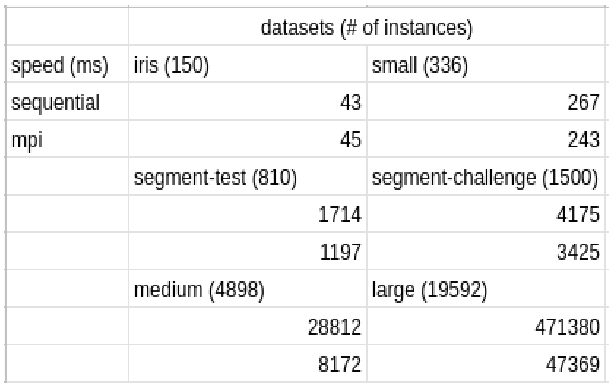


Fig. 7. Time vs. Number of Processes - Large Dataset



	datasets (# of instances)	
speed (ms)	iris (150)	small (336)
sequential	43	267
mpi	45	243
	segment-test (810)	segment-challenge (1500)
	1714	4175
	1197	3425
	medium (4898)	large (19592)
	28812	471380
	8172	47369

Fig. 9. Avg Timings of all Datasets

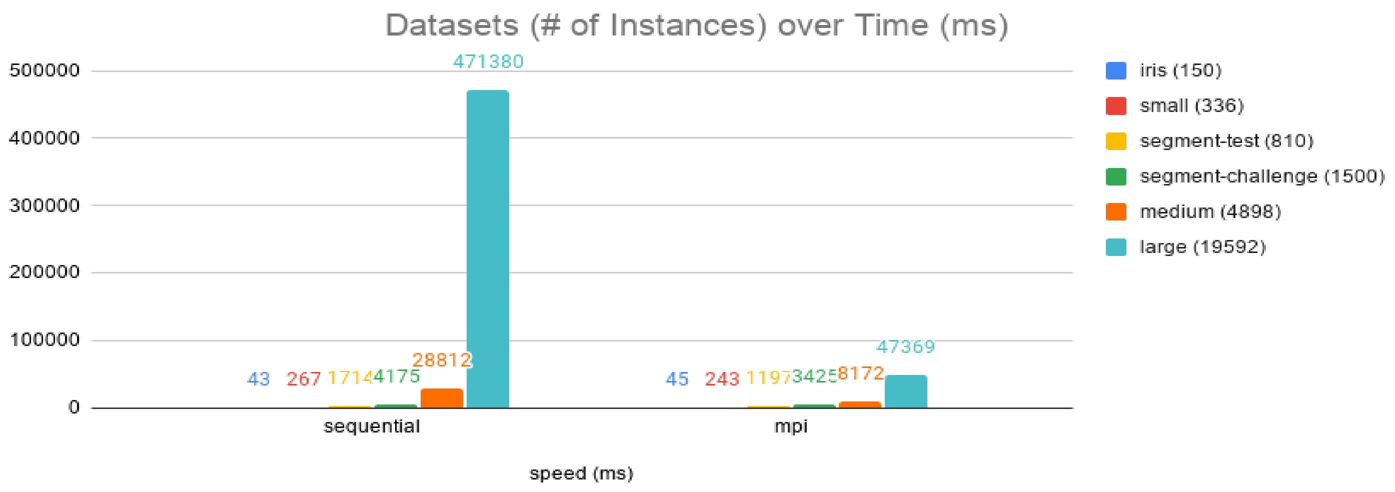


Fig. 8. All Datasets over Time

simply by increasing the number of processese from 1 to 2 and spreading the calculations over multiple processes or a distributed number of systems as opposed to just the one. The performance gains on KNN alone being this drastic with a computational complexity of $O(n^2)$ is amazing and should be much more widely utilized. Especially since in modern times, most if not all current CPUs come with multiple cores and higher capabilities allowing computations to be significantly increased if it were used in many other fields such as gaming, which is currently shifting from high output from 1 core to distributing the workload over more cores. The Speedup is calculated by the widely known Amdahl's law, and is theoretically in this instance of a 95% parallelizable code percentage, capable of a 20x speedup. However, being limited in number of processes available, the best speedup attained was 18.33x on the small dataset of 336 instances. For the large dataset of 19,606 instances the speedup was 14.59x the sequential version. MPI's efficiency over Sequential is not to be argued with, especially when it comes to the KNN, because of it's inherent architecture and data structure implementation, it therefore allows for such a high theoretical speedup.

In a future implementation I'd like to fix part of the implementation as during the Send and Recieve calls of the MPI library for the data transfer back to the master rank, some of the data gets lost or is not being sent and is drastically reducing the accuracy of the KNN whereas it should be able to maintain the exact same accuracy of the sequential implementation.

Acknowledgment

I would like to thank my Parallel Algorithms professor, Dr. Ghosh, who aided me in designing and implementing this project. Without him, I would have never had the push needed to research this fascinating project.

References

[1] Reyes-Ortiz, Jorge & Oneto, Luca & Anguita, Davide. (2015). Big data analytics in the cloud: Spark on Hadoop vs MPIOpenMP on Beowulf. Procedia Computer Science. 53. 121-130. 10.1016/j.procs.2015.07.286.

[2] Aparicio, Gabriel & Blanquer, Ignacio & Hernandez, Valerie. (2007). A Parallel Implementation of the K Nearest Neighbours Classifier in Three Levels: Threads, MPI Processes and the Grid. 4395. 225-235. 10.1007/978-3-540-71351-7_18.