Michael Moon

Garret Reeve

CPSC 474

Bein

10 December 2021

Project 2 – MPI KNN

Table of Contents

Introduction

This report aims to create a parallel kNN implementation via MPI commands, thus theoretically reducing the time complexity of generating classification information from a dataset. The project was finished in C++, and based off this github repository: https://github.com/ignatij/knn-mpi. This report will detail how to compile and run the project, detail the pseudocode used for the classification function, and provide output screenshots, and screenshots of 10 tests of our implementation, and the repository we based this project off of.

The dataset provides training data for RGB values to determine if the color provided is a skin-tone or not. The training data has 245,057 entries, of which 50859 instances are sampled from a variety of age groups, race groups, and genders providing a wide array of potential skin-tones. These are marked with the classification value 1; the other 194,198 entries are classified as class 2, or colors that were not derived from skin-tones.

Upon further inspection, this implementation was unoptimized, and there were many points of contention to rework and reduce the time complexity experienced by the many processors iterating through their tasks. So, along with understanding and working with MPI, this project aims to build the project from the ground up and improve the overall engine used.

The training and test sets are provided here: https://archive.ics.uci.edu/ml/datasets/Skin+Segmentation#.

Execution Instructions

Within a bash terminal, navigate the working directory to the file containing the project. Then, to compile the code:

```
mpic++ -o a.out main.cpp -std=c++17
```

To execute the code:

```
mpirun -n 100 -oversubscribe ./a.out
```

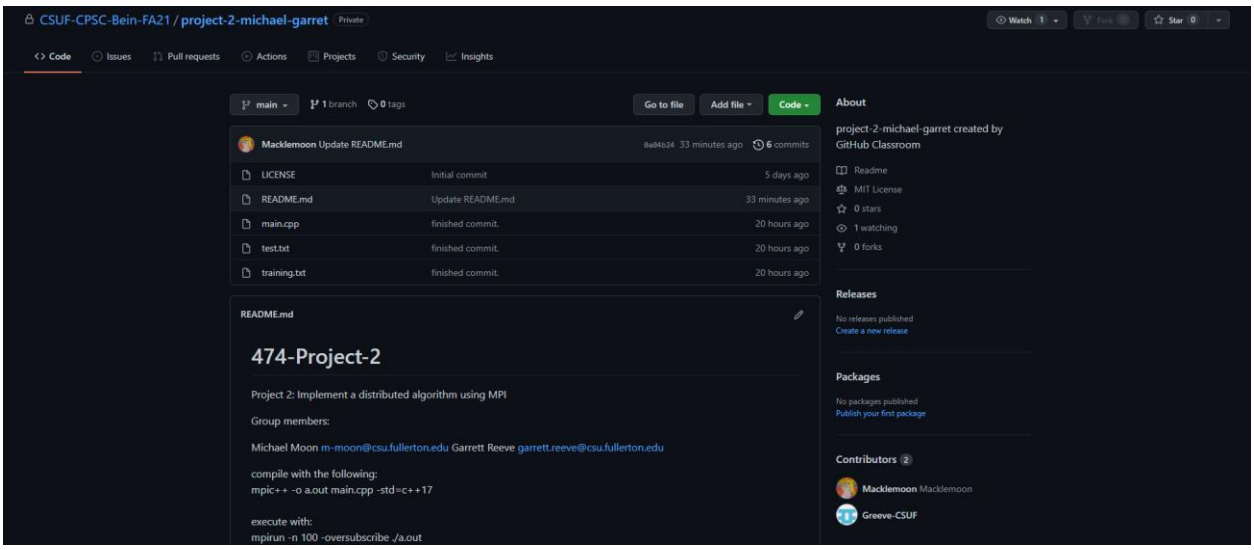
This project was completed with an *Intel® Core™ i5-10600K CPU @ 4.10GHz* with 16GB of RAM running *Windows 10 Home* ver. 20H2. The code was ran on *Tuffix 2020 Edition [Ubuntu 64-bit]* on *Oracle VirtualBox* ver. 6.1.28 r147628 (Qt 5.6.2). The system was allocated 2048 MB from the motherboard, 1 processor with a 100% execution cap. with this in mind, the task stalled for around 35-45 seconds before generating output.

Pseudocode

```
int knnClassifier(instance, k, trainingVector) {

	// the std::map is ordered, so we automatically find the
	// closest distance in index 0, and it ascends.
	std::map(distance, class) distances;

	int class1Counter = 0;
	int class2Counter = 0;

	// generate all distances from the given instance and the training vector.
	// O(n)
	for(trainingVector size) {
		double distance = trainingVector[i].calcDistance(instance);
		distances.insert(distance, trainingVector[i].skin());
	}

	// O(sqrt(n)); for loop breaks when we reach k iterations, like k neighbors.
	for(iterate through the new std::map distances) {
		if(val at current iteration == 1) { increment class1Counter; }
		else { increment class2Counter; }

		if(class1Counter && class2Counter == k) { break; }
	}

	// return based on which counter is bigger.

	if(class1Counter >= class2Counter) { return 1; }
	else { return 2; }
}
```

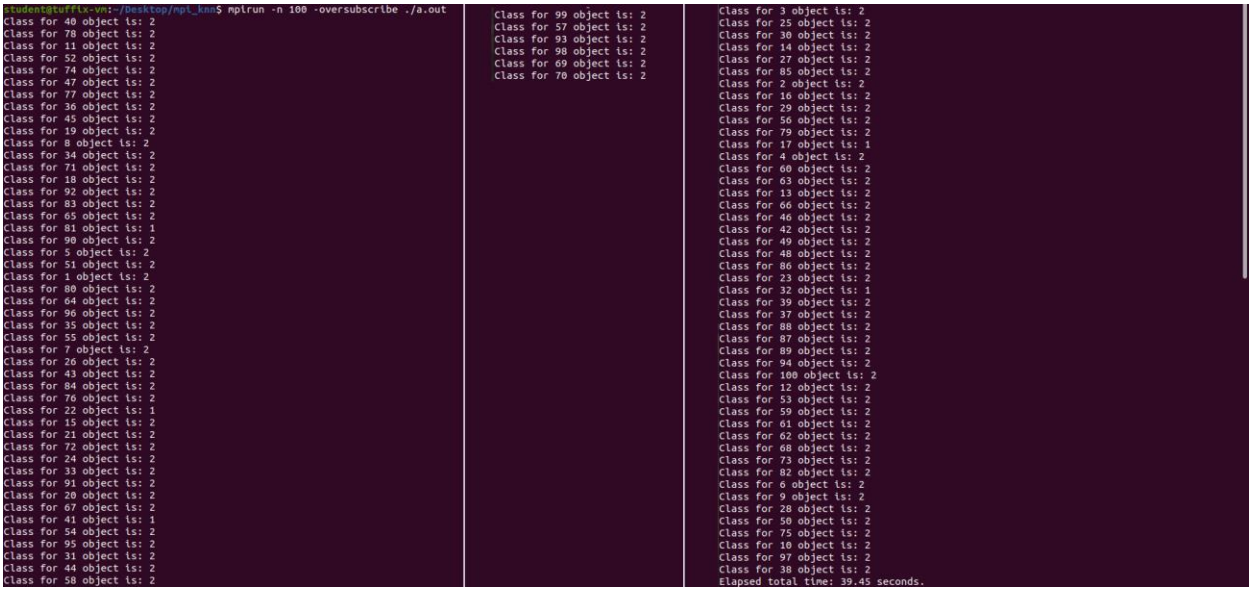
Screenshots

Team Github



https://github.com/CSUF-CPSC-Bein-FA21/project-2-michael-garret

Output

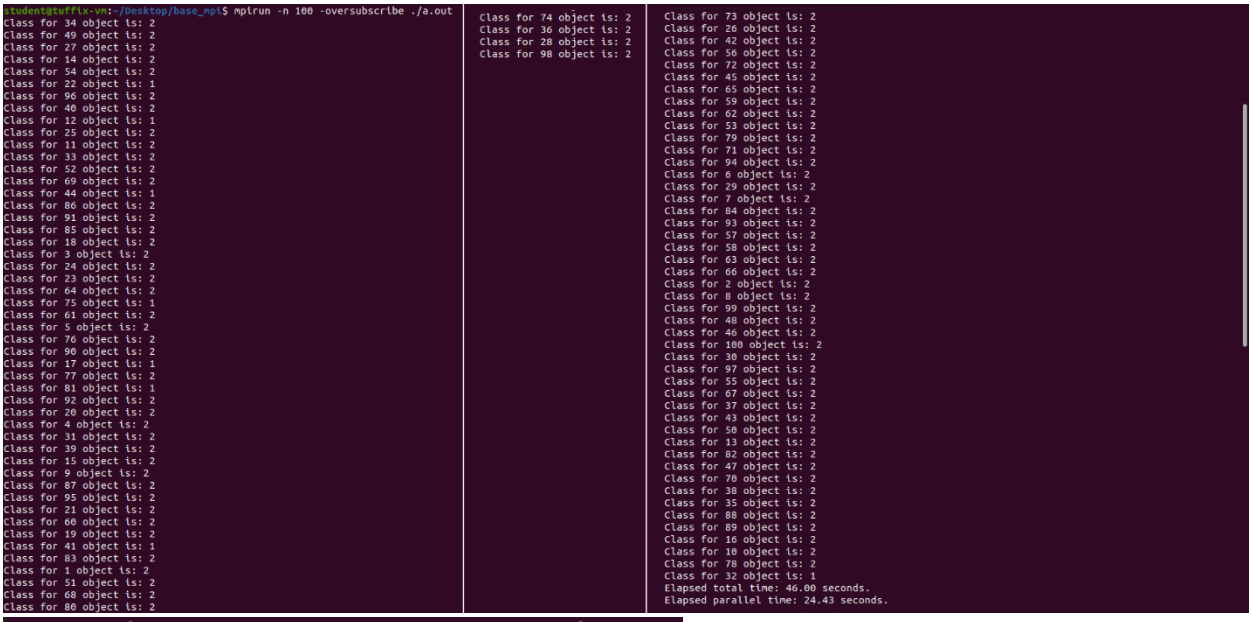


As stated in the execution instructions, executing the program took the testing computer around 40 seconds to finish in a VM. The processors are unordered.

Speed Tests

Here are the compilation times over the span of 10 tests for the old method, and the project's method:

Old method:




```
Elapsed total time: 44.38 seconds.
Elapsed parallel time: 22.95 seconds.
```

```
Elapsed total time: 47.09 seconds.
Elapsed parallel time: 25.37 seconds.
```

```
Elapsed total time: 43.87 seconds.
Elapsed parallel time: 22.14 seconds.
```

```
Elapsed total time: 46.99 seconds.
Elapsed parallel time: 25.41 seconds.
```

```
Elapsed total time: 44.65 seconds.
Elapsed parallel time: 22.94 seconds.
```

```
Elapsed total time: 46.31 seconds.
Elapsed parallel time: 22.08 seconds.
```

```
Elapsed total time: 43.24 seconds.
Elapsed parallel time: 21.82 seconds.
```

```
Elapsed total time: 47.60 seconds.
Elapsed parallel time: 26.17 seconds.
```

```
Elapsed total time: 44.63 seconds.
Elapsed parallel time: 22.31 seconds.
```

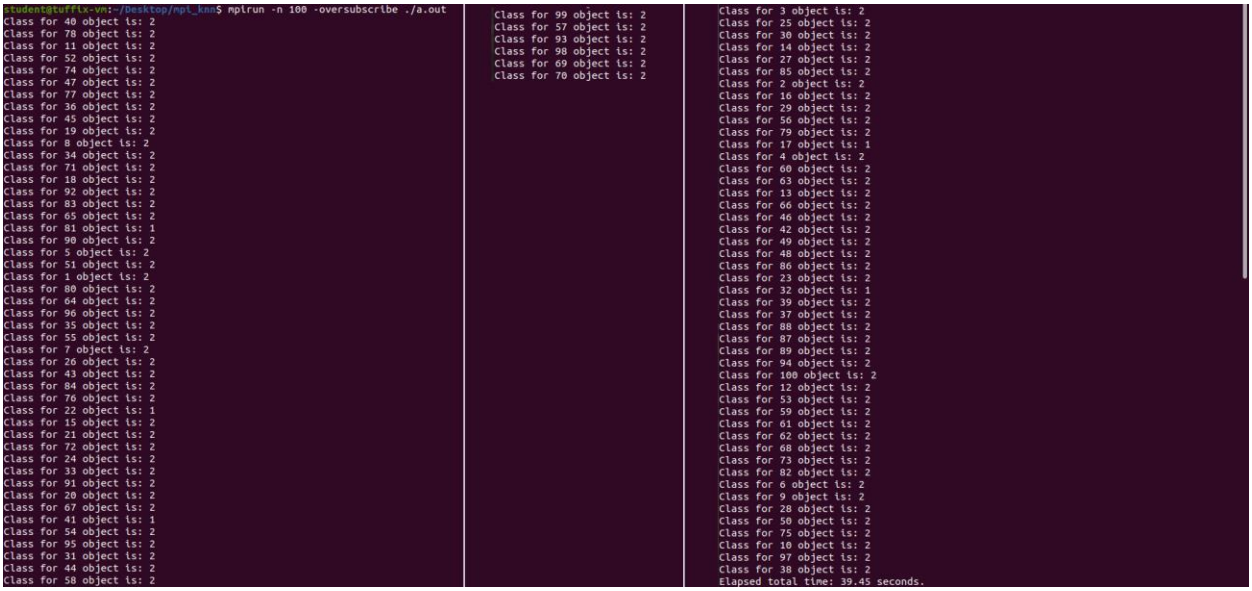
The average total time, that is, the time recorded from the beginning of the MPI connection in seconds is:

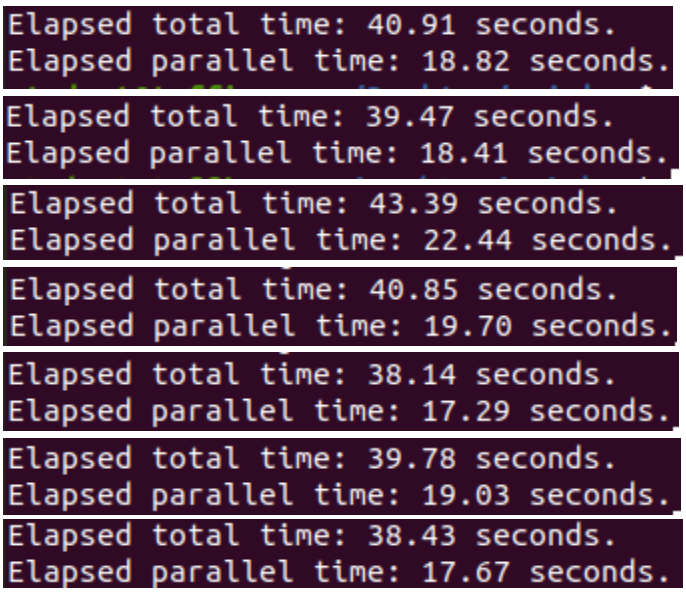
(46.00 + 44.38 + 47.09 + 43.87 + 46.99 + 44.65 + 46.31 + 43.24 + 47.60 + 44.63) / 10 = 45.476

And the total parallel time, measured when processor with rank = 0 beings its process, and ends after the barrier:

(24.43 + 22.95 + 25.37 + 22.14 + 25.41 + 22.94 + 22.08 + 21.82 + 26.17 + 22.31) / 10 = 23.562

Our method:





```
Elapsed total time: 39.70 seconds.
Elapsed parallel time: 18.65 seconds.
Elapsed total time: 41.07 seconds.
Elapsed parallel time: 20.03 seconds.
```

The same average calculations yield the following values:

(39.45 + 40.91 + 39.47 + 43.39 + 40.85 + 38.14 + 39.78 + 38.43 + 39.70 + 41.07) / 10 = 40.119

(18.00 + 18.82 + 18.41 + 22.44 + 19.70 + 17.29 + 19.03 + 17.67 + 18.65 + 20.03) / 10 = 19.004

Analysis

Overall time difference: 45.476 - 40.119 = 5.357seconds

Parallel time difference: 23.562 - 19.004 = 4.558 seconds

Serial time difference: 5.357 – 4.558 = 0.799 seconds

Essentially, the majority of the time saved occurred within the parallelized section; a rather significant change of around 4.6 seconds came from this portion, which can be attributed to the change in the classifier function. The original implementation populated an extra *std::set* used to store the values of the distances themselves, to use in tandem with the *std::map* that housed the distances as keys, and the classification values as vals.

Quick aside, these variables were passed in as arguments, but were empty and populated within the function, meaning there was an extra pointer and space allotted for the instantiation that was never used and consumed space and processing power, as functions make light copies of the pointers when passed in as arguments.

The original implementation used the set, which was ordered in the exact same manner as the map, as the data structure the iterator would pass through, and made a complex reference to the map to access the same index.

```
distanceToClass.find(*it) -> second == 1
```

Where it was a *std::set* iterator passing through the std::set, rather than the *std::map* directly.

The rest of the time saved came from cleaning up the code and doubling up on loops to avoid redundant loops.

For example, in the main function, in order to populate the training vector with standardized entries, the code would populate the entire 245,057 entries into a *std::vector* via *std::ifstream*, then iterate through all values of the std::vector again to find the min and max values for each feature. Once that was finished, it would iterate a third time to standardize each value with the newly processed information.

Finding this redundant, we coupled the calculations to find the min and max values for each feature with the original std::vector populating step; we checked each instance's features with the current min and max values as it was being read from the file, eliminating one entire loop of 245,057 iterations.

With other optimizations, we reduced the time taken by the program rather effectively, creating a much more time-efficient method of kNN classification for the problem posed by the dataset.