University of Puerto Rico
Campus of Mayagüez

Remove Duplicates Benchmark

Alfredo Pomales
David Riquelme
Victor Lugo
ICOM6025 – High Performance Computing
Professor Wilson Rivera Gallego

Outline

1. Executive Summary

The problem presented to the team was to choose a performance enhancing topic to be developed and presented. The team, having experienced working with various parallel implementations for previous assignments and understanding that it was one of the main topics of the course, decided to focus on downloading and running a benchmark for the Remove Duplicates algorithm in various machines such as CentOS, Ubuntu, and MacOS. If there was enough time to experiment with other forms of parallel programming extensions such as CUDA and MPI then the team would also implement these as external program runs since the benchmark did not provide the source code algorithm for removing duplicates.

The initial problem the team encountered was understanding how the benchmark ran after downloading and extracting the tar files. The files all seemed to be working correctly from the beginning, however, after various attempts to run the code, the team would get error messages that files were unable to be accessed or found. After much investigation, the solution was found that the benchmarks themselves had to first be configured using make files that were already present in each subdirectory which represented different benchmarks. The team also came to understand that the benchmark given to us by the professor was only a small fragment of the entire benchmark and thus constituted to errors experienced such as missing files and unfinished benchmark codes. However, after realizing this problem, the team adapted by running the benchmarks separately and obtaining each result from different problem size ranging from 10 to 10 million which was the limit. The limit was also tested for verification purposes only.

In this report we shall present the technical steps the team took to approach and resolve these problems and the decisions taken to further advance the results of our project. We decided upon running the benchmark on different operating systems to compare the performance of each operating system under the same hardware specifications. The team shall also present the metrics used to measure and compare their performance as well as the experimental settings used in the Chameleon Cloud website where the virtual machine had a certain number of nodes and processors to complete the tasks at hand.

After demonstrating the performance metrics and settings to be used, the team will discuss the infrastructure and programming model the team adhered to while running the code..

The will include the graphical analysis of the results for different data sizes obtained for the benchmark as well as how they compare to each operating system and brief discussion explaining these results.

Finally, the team will conclude our findings and discuss our future work to improve the benchmark testing and expand our parallel programming models to other more complex and time-consuming implementations.

2. Technical Approach

The technical approach to this project began with the decision to choose which kind of high performance project we would be interested in implementing. During this time, the professor had already made available the instruction files for assignments 1, 2, and 3. The team decided to view these documents as a way to help ourselves make a decision by observing the topics the assignments were related to. One assignment stood out immediately which was the third one. This was due to one of the problems being related to running a benchmark and observing its performance. It was obvious that the performance depended more on the system the benchmark would be running on rather than how the code was implemented. The team found that evaluating different operating system and hardware limitation performance would be an interesting topic. The team also observed that one of the possible topics for the project was the use of benchmarks and the document already provided a website link that guided us to various possible benchmarks.

After analyzing all the possible benchmarks for various algorithm, the team decided to use the benchmark that belonged to “removeDuplicates”. The decision was made based on previous knowledge the team members had of the algorithm. The team also decided that, if there was enough time to implement, it would be interesting to construct our own personal algorithm and test it with various parallel implementation such as those presented in assignments 1 and 2.

The team proceeded to download the files needed for the benchmark which came in the form of a tar file. All team members then created different reservations and instances on Chameleon Cloud to begin testing the benchmark on different operating systems. The work was divided as such: Alfredo Pomales ran the benchmark on his own machine of MacOS which had one processor with 4 cores; David Riquelme ran the benchmark on the virtual machine connected to an Ubuntu 16.04 operating system; and Victor Lugo ran the benchmark on the virtual machine connected to a CentOS7 operating system.

However, immediately problems were found while trying to run the benchmarks for all three operating system. After much investigation, the team was able to conclude that the errors were mainly due because the benchmark itself was not complete. This meant that, although the benchmark was exclusively for the remove duplicates algorithm, the benchmark itself was only a fragment of a much larger benchmark that analyzed various algorithms in a single run. The site, however, had made errors when partitioning the files for users to download them separately and left important folders and files out of the originally downloaded tar files. This caused confusion and required the team to have to resort to changing some of the original files to adapt the benchmark to the files the tar currently contained.

As these changes were being made, the team realized that attempting to change the original code from the benchmark was exceptionally difficult and time consuming due to various programming languages used in different files. As such, the team decided to use the performance metrics contained in the benchmark exclusively during our analysis, but to thoroughly run the benchmark for small arrays and appropriately increment the size until the limit was reach. The step size was chosen different for each increment and was chosen in such a way to accurately demonstrate the performance of the code across various sizes of array.

3. Performance Metrics and Experimental Settings

The only performance metric chosen to compare the operating systems through the benchmark was the time it took to complete. However, it is important to note that the time the benchmark outputs for each sequence of numbers does not seem to match the length of time it actually takes because it seems to take much more time from when it starts to run to when it outputs the first calculated time. The reason behind this is because the benchmark takes in an already computed file from a subdirectory called "sequenceData" where all the files needed to create input files are needed. These input files are treated as text files so the first step of the benchmark is to open the file and then read all the numbers, assort them into an array, and then apply the algorithm. The algorithm does not start to consider the time of execution until the third phase when the algorithm is called. It is also important to note that the reason the execution time is the only performance metric being observed is due to the nature of the benchmark itself and the team did not have the resources such as time and knowledge to be able to expand the performance metrics to other areas.

The experimental settings in which this code ran were all contained to the facility the professor provided for the course and one of team members personal laptop. The facility is known as Chameleon Cloud and it is a website that permits its users the ability to create reservations and instances of various operating system images to connect through a virtual machine. Besides this, Chameleon Cloud also contains various hardware specifications for their processors which will be explained with greater detail in the infrastructure section of this project. The operating system the team considered to use were the ones that the team either had familiarity with such as Ubuntu and MacOS or those whose instructions and understanding of the system were easily obtained with the user FAQ support provided by Chameleon Cloud.

The version of Ubuntu that the team will be using will be 16.04 which is the latest version that Chameleon Cloud can provide us; the version of CentOS is the 7^{th} which is also the latest version that Chameleon Cloud can provide us as well as the default appliance for beginning users of the website; and, finally, MacOS with version 10.13.2 High Sierra which is provided by the university to one of our team members for one of his investigations from another course. Each of these operating systems have the ability to use various nodes in the Chameleon Cloud cluster that are connected with an Infiniband. The details of this connection will be explained with greater details in the upcoming infrastructure section of this report.

4. Infrastructure and Programming Models

The infrastructure of the system mainly depends on the hardware specifications that Chameleon Cloud has in their cluster. It is important to note that a lot of the hardware provided to use by Chameleon Cloud will be unused since the only performance metric being evaluated in this report is the execution time of the benchmark. The hardware specification of the MacOS personal laptop is as follows: processor 2.2 GHz Intel Core i7 and 16 GB 1600 MHz DDR3 RAM. The hardware specifications for the machines located in the Chameleon Cloud cluster are as follow: 2 CPUs with 48 threads each, 134956859392 RAM, 3.10 GHz Intel Xeon with 3 caches of 32 KB, 256 KB, and 30MB respectively. The connection between them is an Infinibadnd. It is important to note that the type of nodes that were used were only for computing since this is the most effective way to obtain execution time. Also, the cluster contains 291 nodes of this type for parallel programming.

The programming model used for the benchmark is a mixture of various languages such as C, C++, and python. However, the most important aspect is that the parallel extension used for this benchmark was OpenMP. The amount of threads that were able to run is specified within the code and almost inaccessible to us, so we can not specify in the report the number of threads. However, to reimburse for this lack of information, we are thoroughly testing for almost 500 million data entries into the array of 6 various types of arrays. These are: random sequence, exponential sequence, tri-gram strings, tri-gram strings with auxiliary integers, added data sequence, and almost sorted sequence. As specified before, due to the complexity and missing files of the benchmark, OpenMP is the only programming model we are able demonstrate in this report.

5. Experimental Results

Before presenting the graphical results of all three operating systems, it is important to note that MacOS was being run on a single personal laptop provided by the university and, as such, had severe hardware limitations versus the performance Chameleon Cloud's cluster could offer. Even so, Chameleon Cloud's cluster was not able to handle an array size of 500 million data which will be shown as a blank space in the graphical results.

These are the graphical results:

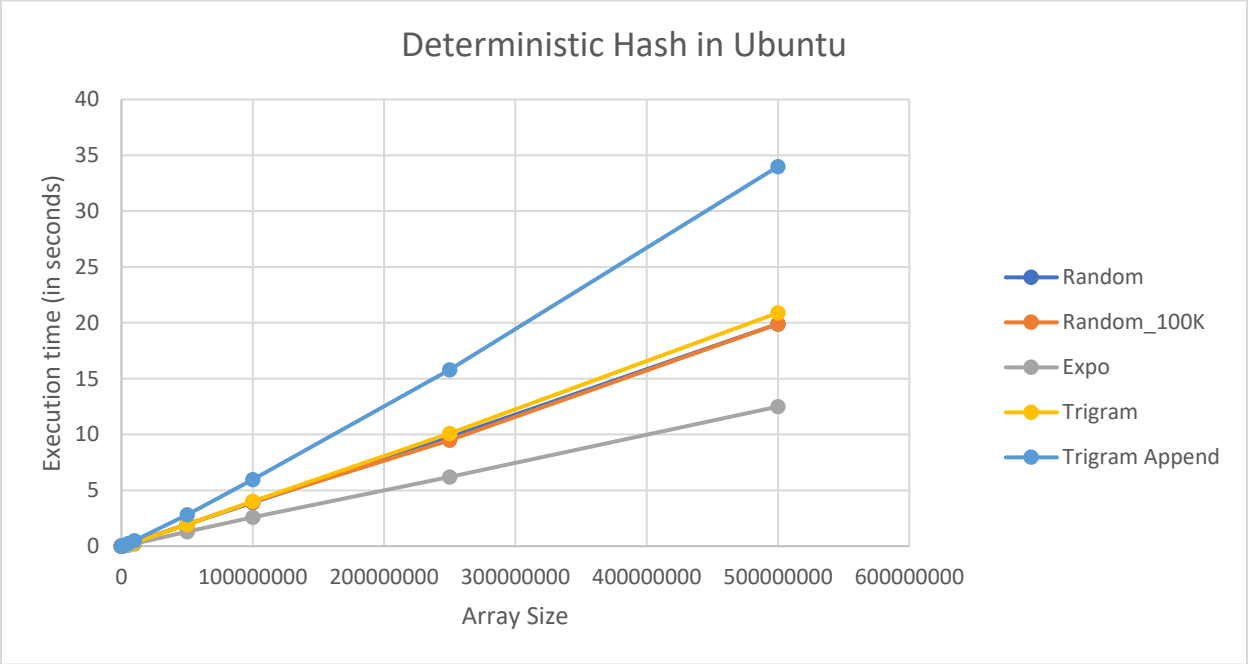

Figure 1: Deterministic Hash in Ubuntu

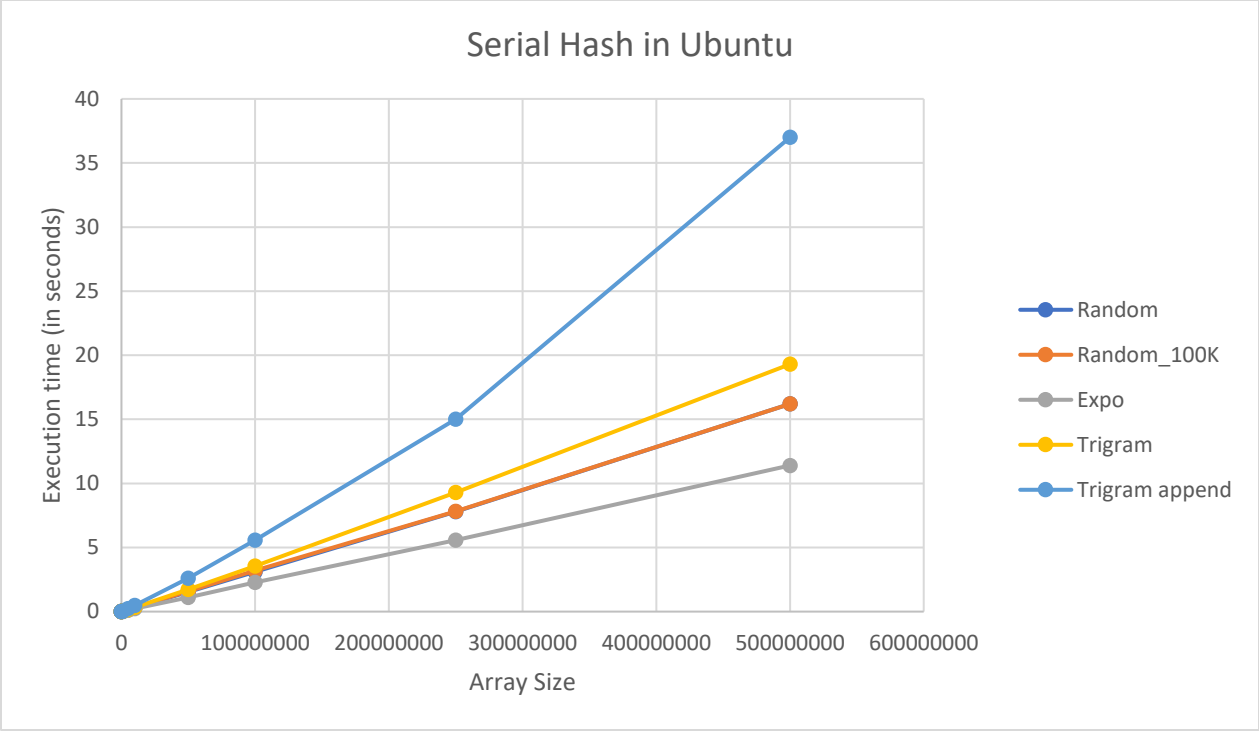

Figure 2: Serial Hash in Ubuntu

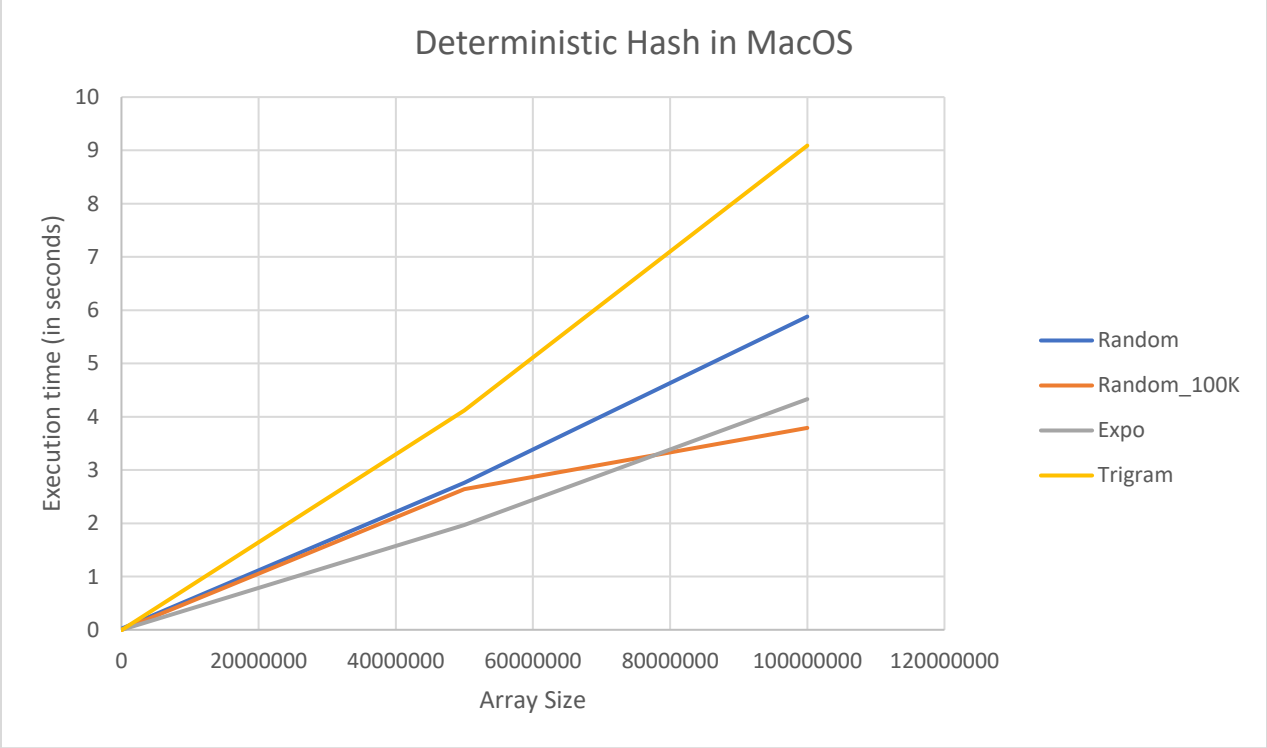

Figure 3: Deterministic Hash in MacOS

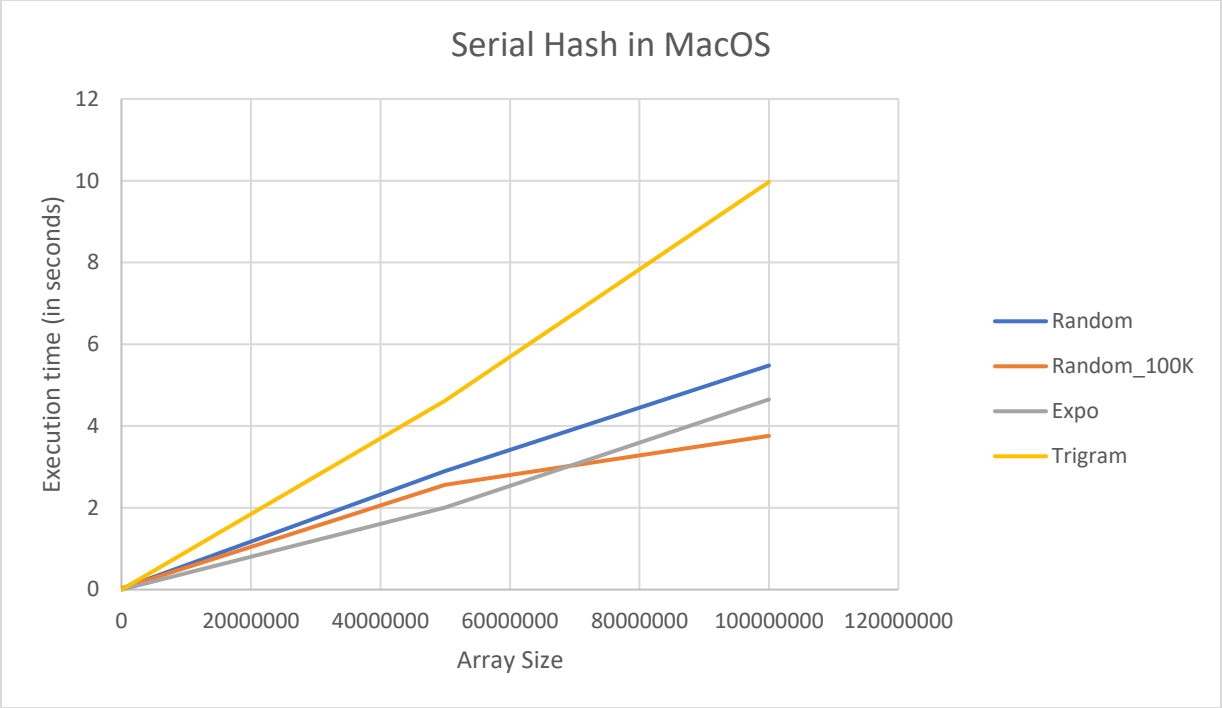

Figure 4: Serial Hash in MacOS

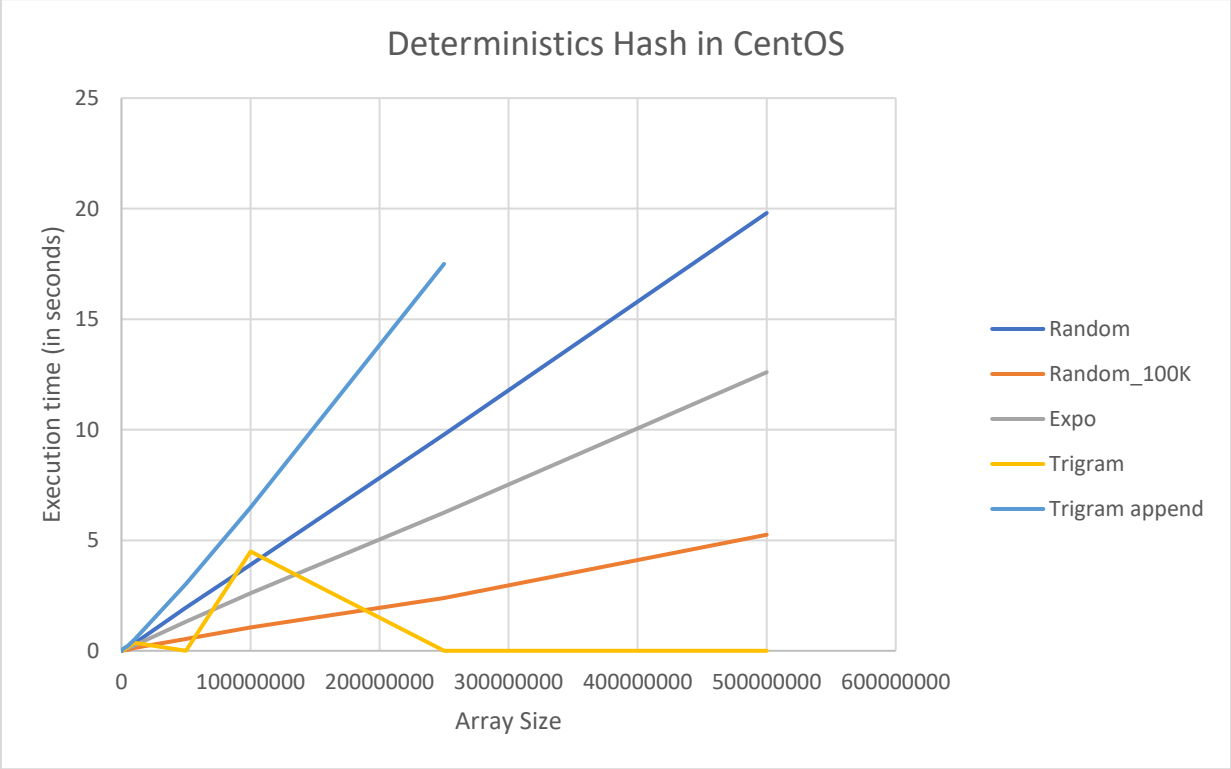

Figure 5: Deterministic Hash in CentOS

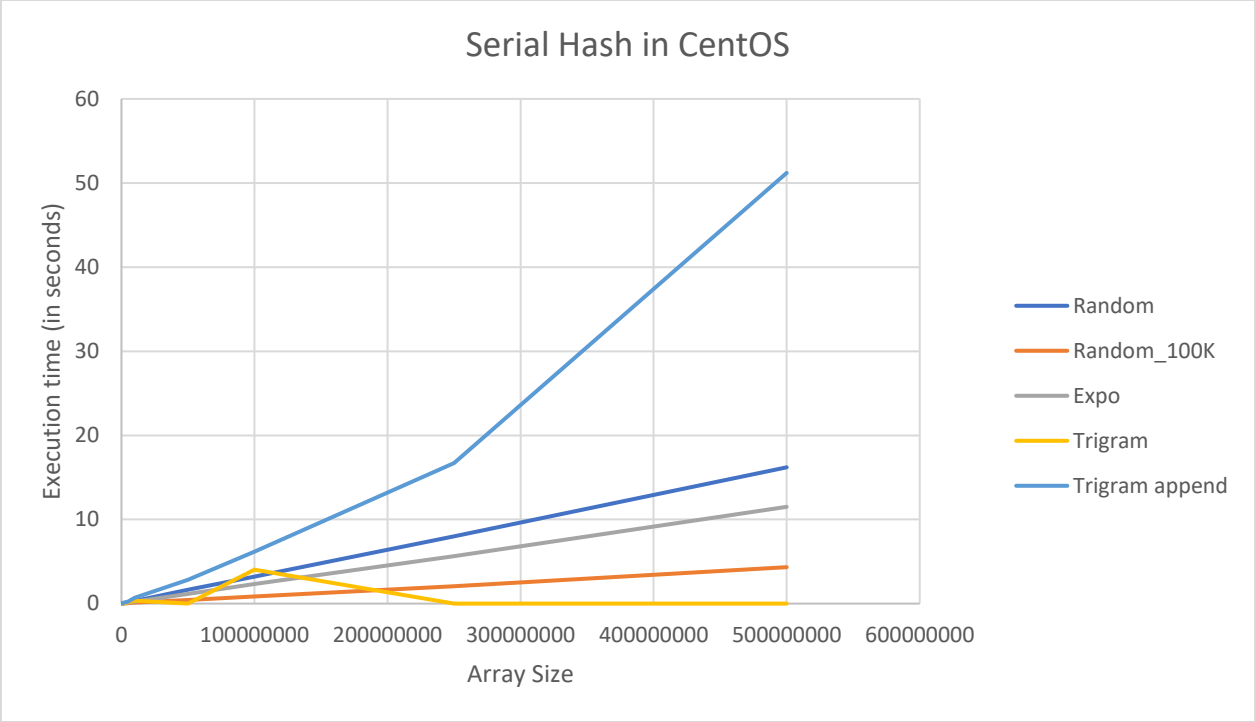

Figure 6: Serial Hash in CentOS

As we can observe from the various graphs presented above, each operating system has a different sequence that they perform better upon. For CentOS, Trigram has the best performance, this is in due to the fact that Trigram is no longer an integer, but a string. CentOS seems to have less computational needs when it comes to comparing and removing strings. Execution time only spikes at an array size of 100 million data points and goes back to less than milliseconds in computational performance. This applies for both deterministic and serial hash.

For the limited data that could be tested in the single computer of MacOS, a random sequence between 0 and 100 thousand had the better performance. While the other types of sequences kept increasing exponentially, the random sequence between 0 and 100 thousand was demonstrating a logarithmic bound.

For Ubuntu, the best sequence was the one that was obtained from an exponential distribution. Although all of the sequences increase exponentially, the exponential distribution sequence is the one that takes the least amount of time to execute.

6. Conclusion and Future Work

Parallel programming is not a simple thing to do especially when there is a need to test against supercomputers using benchmarks. One of the most important things to keep in mind when changing a sequential code to parallel is the data integrity. The steps are not as easy as such adding pragma directives for openMP or copying the sequential code to CUDA, these actions would just give you the wrong output. It is necessary to consider data integrity in each step when parallelizing the code.

In our project the team focused on observing how the remove duplicates benchmark performed in different operating systems for different input files. These input files were obtained by making random sequences of code with certain distributions. The first two used uniform distribution with different ranges of value, the third used exponential distribution, and the fourth and fifth used strings and strings with attached integers respectively. The overall result from these tests revealed that the remove duplicates algorithm is not a very difficult algorithm to process on a single machine when compared to the performance results obtained from the Chameleon Cloud cluster. However, it does require to have more security since the laptop itself might heat up to dangerous levels.

For all three operating system, the team also decided to test against certain protocols of hashing. We utilized serial hash and deterministic hash, the main different between the two being that deterministic hashing gave repeated has values for the same entry value while serial hashing could give different hash values for the same entry value. As a team, we personally learned a lot about the different benefits some operating systems can offer for certain algorithms and input files. However, a lot of work needs to be done to fully understand these principles.

For future work, the team hopes to be able to implement our own original algorithm of remove duplicates and then test the sequential timing of the code with the parallelized version. The team also plans to expand the programming model of parallelization to include CUDA, MPI, and other parallel architecture that exist for C++. Once obtained the parallelized code from different architecture, the team hopes to test these codes again in different operating system with different input files and different hashing protocols. We hope to expand our knowledge from this and include other types of algorithms for much later in the future.