Parallel Statistic Analyzer (PaStA)

Dominic J. Catalano

March 12, 2016

1 Project Description

In order to clearly understand data in a dataset, it is necessary to see the distribution of the data. By creating a histogram of the data allows users to see the distribution of the dataset while also highlighting any anomalies that may exist. The Parallel Statistic Analyzer (PaStA) scans over a given dataset and produces such a histogram based on the frequency of entries that exist within a certain standard deviation range while also displaying the mean and standard deviation of the data. Such a histogram can help a data administrator note trends while also identifying possible outliers agnostic of the dataset provided.

The program is designed to work with any set of numbers. However for the purposes of testing, a random number generator library (rngs and rvgs) with a set seed will be used to generate a geometric distribution of high variance. This generator was chosen because of its high variance, resulting in larger, more meaningful histograms. It was designed by Steve Park and Dave Geyer in conjunction with a paper on random number generation. It will also provide similar histogram results regardless of set size.

2 Code Design

Code for the calculation of the standard deviation along with the construction of the histogram was designed specifically for this project using a two pass algorithm. The two pass algorithm should further the performance gains via parallelism while also simplifying some mathematics involved. The first pass is required to find the mean of the data as $\bar{x} = \frac{\sum x_i}{n}$. The second pass is then used to calculate the deviation as $\sigma = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$. Once these values are known, a histogram can be constructed. Another pass over the data is then necessary to count the numbers in each standard deviation bin. The histogram is then iterated over, normalized, and printed for the benefit of the user along with the set statistics.

This project will demonstrate that parallelization of these processes will increase the performance by markedly decreasing the runtime of each test when compared to the serial version. Different forms of parallelization will be tested

to determine the best implementation and coding paradigm to achieve optimal performance.

3 Execution Instruction

Because the dataset is generated by the program, very little work is needed on the client side. Each iteration of the program will be designed such that five runs of each set size (10^6, 10^7, 10^8, and 10^9) are run for certain environmental conditions that are configured in the bash script. Thus, each version of the program will be accompanied by its bash script that is meant to produce a plethora of results (i.e. serial.bash) bashed on configurable conditions. For the serial version, no environment needs to be created. However for methods like MPI and OpenMP, parameters like thread number or workers can be configured. The output for each test will be generated in an output file prefixed by the execution time (i.e. serial_output_file).

Each execution method will be encapsulated inside a function of each run executable. In the main method of the c code, there exists a flag to suppress output of PaStA, allowing only the runtime to be printed. This flag is only activated after ensuring the algorithmic correctness of the particular method in question. This allows the client to collect large amounts of data referring to the runtime of each environment.

The main function is tasked with the creation of a dataset of size 10^9 which is then passed into the runTest function. This function takes a subset of the data along with all necessary information, performs the algorithm, and returns the time needed to execute.

4 Serial

The initial results come from serial.bash whereby SerialPaStA was run with tests sets of 10^6, 10^7, 10^8, and 10^9 as an input. The timing was done by using the time.h library to time the execution from after the generation of the dataset to the completion of the program. The histograms each provided results that were comparable but not exact. This is to be expected because, while the seed is set, the sets are all different sizes. For this concision the result sets are not reproduced here. A table of the run times compared to set size and execution method can be seen in Table 1.

As can be expected, larger set sizes result in longer execution times. This can best be seen in Figure 4

5 OpenMP

For the OpenMP implementation of PaStA (OpenMPPaStA), the runTest function takes advantage of built in OpenMP calls for parallelism. The sum of the

Set Size	Serial(s)
10^6	0.058
10^7	0.356
10^8	2.833
10^9	28.034

Table 1: Serial Runtimes

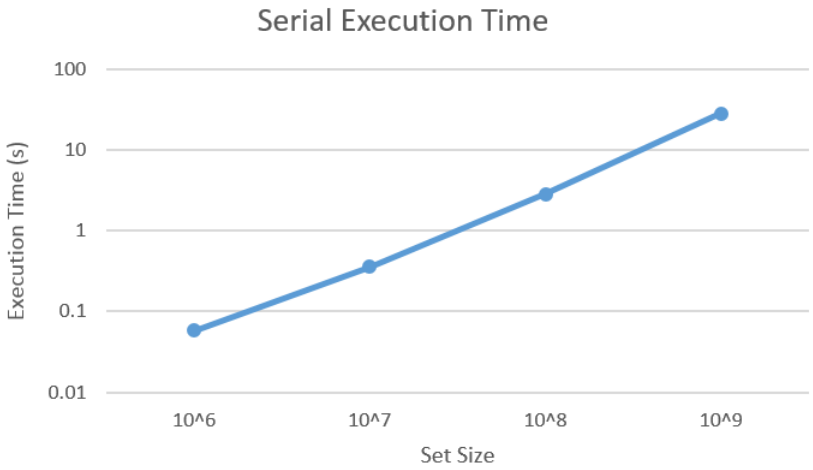

Figure 1: A plot of execution time against set size.

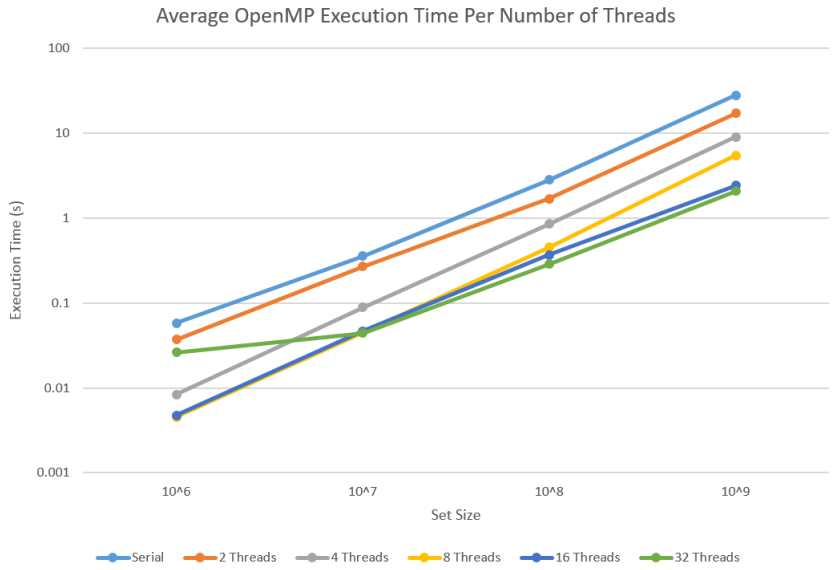

Figure 2: A plot of execution time against set size for a given number of threads.

set is found using a reduction sum across the set. This sum is then used to calculate the set's mean on each of the worker threads available. From there, the variance is calculated by reducing the squared difference between each element and the mean. Once each thread calculates the standard deviation from the variance, each thread creates a partial histogram on a subset of data assigned to it via a parallel for loop. Finally, an atomic statement is used to aggregate the results of each partial histogram into the final result set. This result set is then printed by thread 0 (if output is not suppressed). The run time is then calculated and returned to the main function.

The bash script for this environment, openmp.bash, runs the program using thread numbers of 2, 4, 8, 16, and 32. This test bench clearly shows the relationship between performance gain and number of threads. Because the main function runs each test five times for each set size, the average runtime of each set for each environmental set up is shown in Figure 5.

Performance shows a positive relationship between set size and the number of threads with the exception of the 32 thread test on the smallest set size. This can be contributed to overhead required for communication between the threads. However this still shows a gain over the serial execution in every situation.

6 MPI

The MPI implementation of PaStA (MPIPaStA) leverages SIMD by partitioning the dataset created in the main function among the available workers. The main

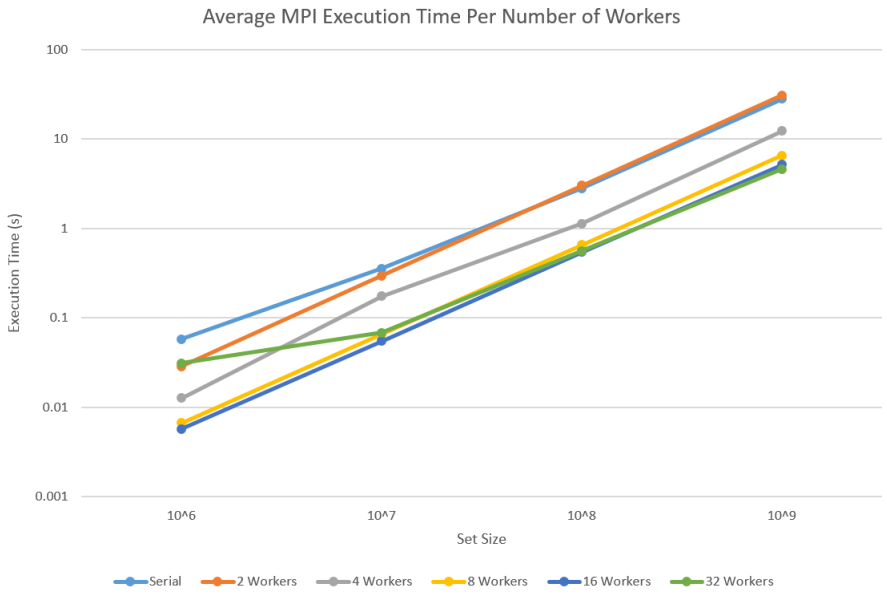

Figure 3: A plot of execution time against set size for a given number of workers.

function is also responsible for the creation of the MPI Communicators necessary for MPI calls. Firstly, the dataset is partitioned and sent to each available worker who sums the subset of data. This partial sum is reduced and distributed to all workers via an all reduce call. From there, the mean is calculated and the partial squared distance between the data and the mean is found. This partial delta is then reduced via another all reduce, allowing each worker to calculate the standard deviation and build a partial histogram based on its data. The partial histogram is then reduced to the master's final histogram, which is then printed unless output is suppressed. The time is then calculated and returned to the main function.

Like the OpenMP version, the bash script (mpi.bash) runs the executable with 2, 4, 8, 16, and 32 workers. The five runs for each set size and number of workers is averaged and plotted in Figure 6. Similarly, a positive relationship is shown between set size and execution time where the number of threads decreases the time needed. Once again, a larger number of workers reduces the time needed with the exception of the smallest set size. This can once again be attributed to the communication overhead required between the larger number of workers. Usually, the MPI version of PaStA performed better than the serial version. The exception to this would be the test using only two workers. This is intuitive however because the master does not perform any intensive calculations. Meaning, the one worker is tasked with the entire vector without benefiting from parallelism.

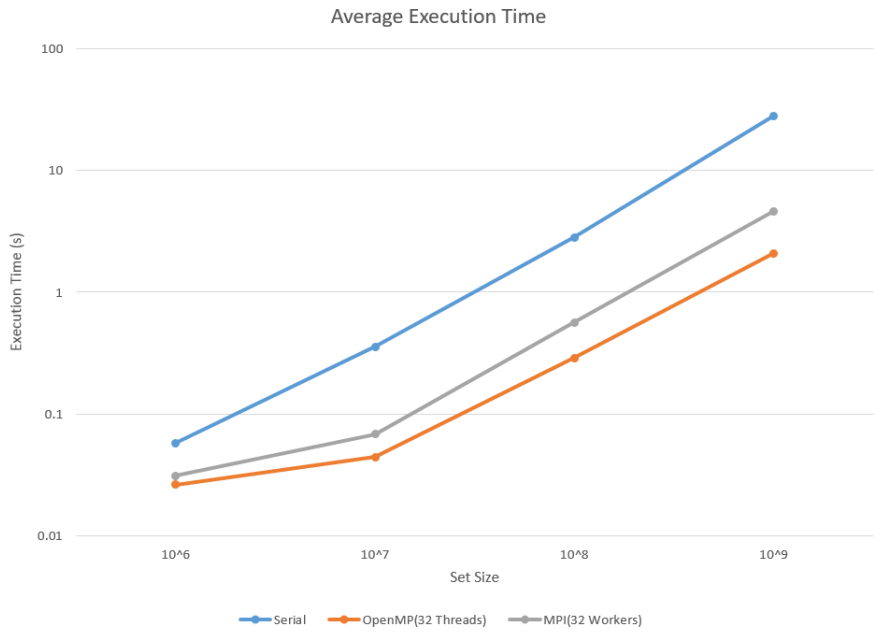

Figure 4: A plot of execution time against set size for each execution method.

7 CUDA

8 Analysis

Overall, the parallel versions of PaStA utilizing OpenMP and MPI prove to be more efficient than the serial version. Figure 8 plots the run time of the serial version against the parallel varietals. The chart clearly shows how the execution time is lower, especially given a larger dataset. For concision, only the tests of 32 threads for OpenMP and 32 workers for MPI are reproduced. However, the comparison between the serial and other tests can be found in Figure 5 and Figure 6 respectively. Figure 8 computes the gain with the equation $F(p) = t_s/t_p$. While speedup is less given small set sizes, the gain achieved with larger set sizes show that PaStA benefits from a parallel implementation.

In this situation, MPI performs worse than other parallel implements due to the communication costs required to span a distributed memory system. By that logic, OpenMP performed better because it did not have to spend time performing a partitioning of data or communicating between multiple nodes.

9 Git Repository

The git repository can be found at *https://github.com/dom-catalano12/PaStA*.

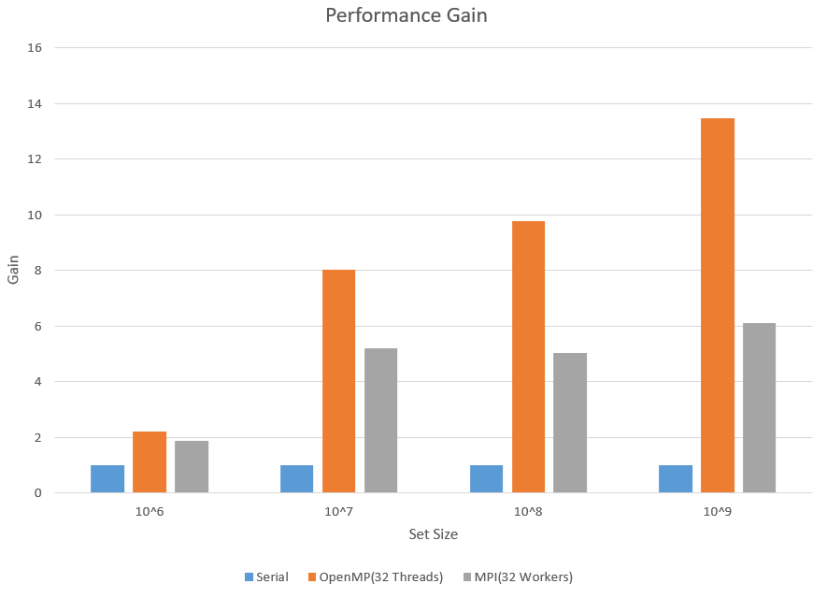

Figure 5: Gain comparison for each execution method.

References

[1] S.K. Park & K.W. Miller *Random number generators: good ones are hard to find* Communications of the ACM, Volume 31 Issue 10, Oct. 1988