Samiur Rahman (20660512)
Nosheen Adil (20674286)
7 July 2018

ECE 254 - Lab 3 Report

Timing Data Results

Table 1

N	B	P	C	AVG Time Processes (ms)	AVG Time Threads (ms)	STD Time Processes (ms)	STD Time Threads (ms)
100	4	1	1	0.49975	0.402214	0.09772541	0.06731939
100	4	1	2	0.501478	0.39276	0.03314908	0.06507825
100	4	1	3	0.499848	0.384434	0.02798723	0.03911472
100	4	2	1	0.546396	0.394658	0.04982107	0.07510069
100	4	3	1	0.555136	0.40019	0.04698018	0.03258512
100	4	2	2	0.502852	0.37267	0.03312187	0.04537172
100	4	3	3	0.590996	0.418112	0.04698855	0.04830503
100	8	1	1	0.470106	0.38862	0.05167489	0.06458399
100	8	1	2	0.497224	0.384398	0.02766785	0.04787141
100	8	1	3	0.493706	0.37776	0.02635564	0.04741597
100	8	2	1	0.523474	0.368774	0.02819428	0.0734966
100	8	3	1	0.541918	0.365394	0.03877034	0.02927304
100	8	2	2	0.503416	0.339064	0.02812947	0.04585481
100	8	3	3	0.583298	0.373088	0.04172833	0.04047227
398	8	1	1	0.635972	0.53494	0.05209921	0.05156344
398	8	1	2	0.802412	0.59533	0.03171199	0.02838403
398	8	1	3	0.784496	0.64787	0.04102801	0.03442158
398	8	2	1	0.851692	0.60449	0.04893086	0.06757679
398	8	3	1	0.854796	0.734702	0.07406588	0.06297578
398	8	2	2	0.662586	0.541218	0.04473865	0.0526261
398	8	3	3	0.74055	0.583862	0.05339393	0.06214502

Graphs

Single Producer/Single Consumer

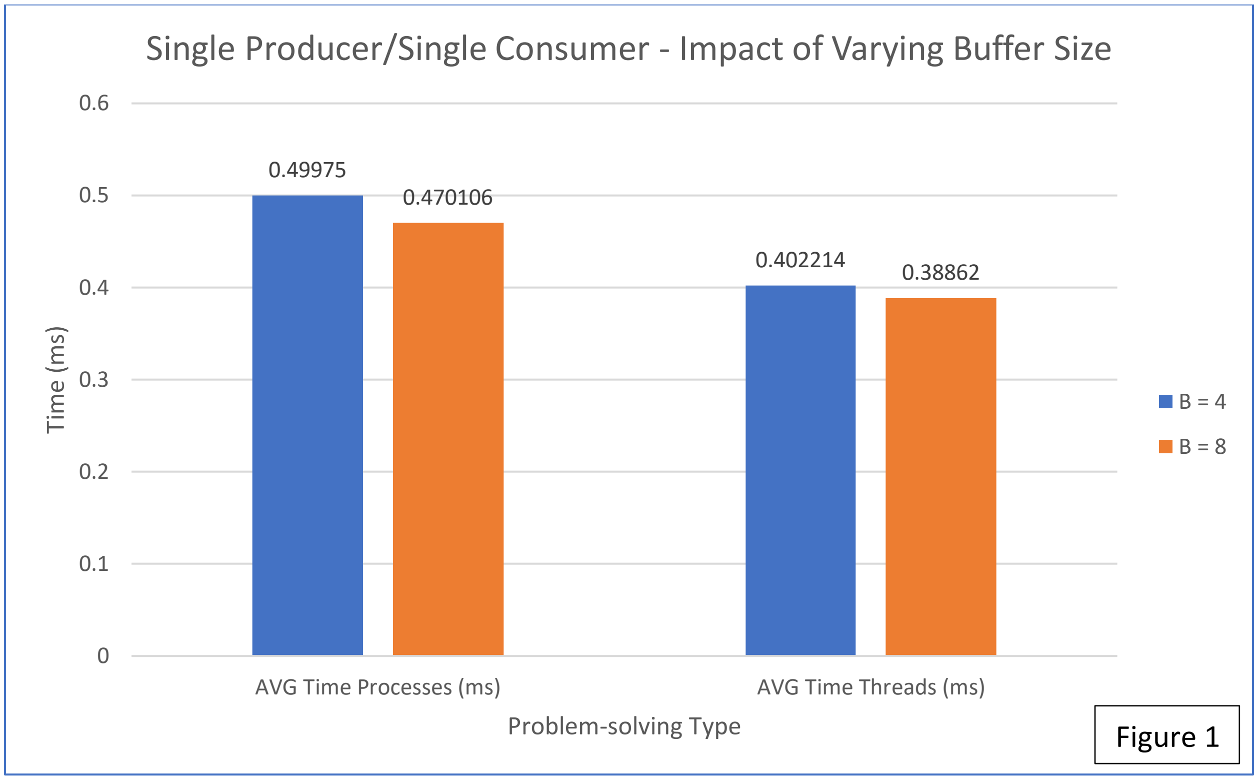

Figure 1

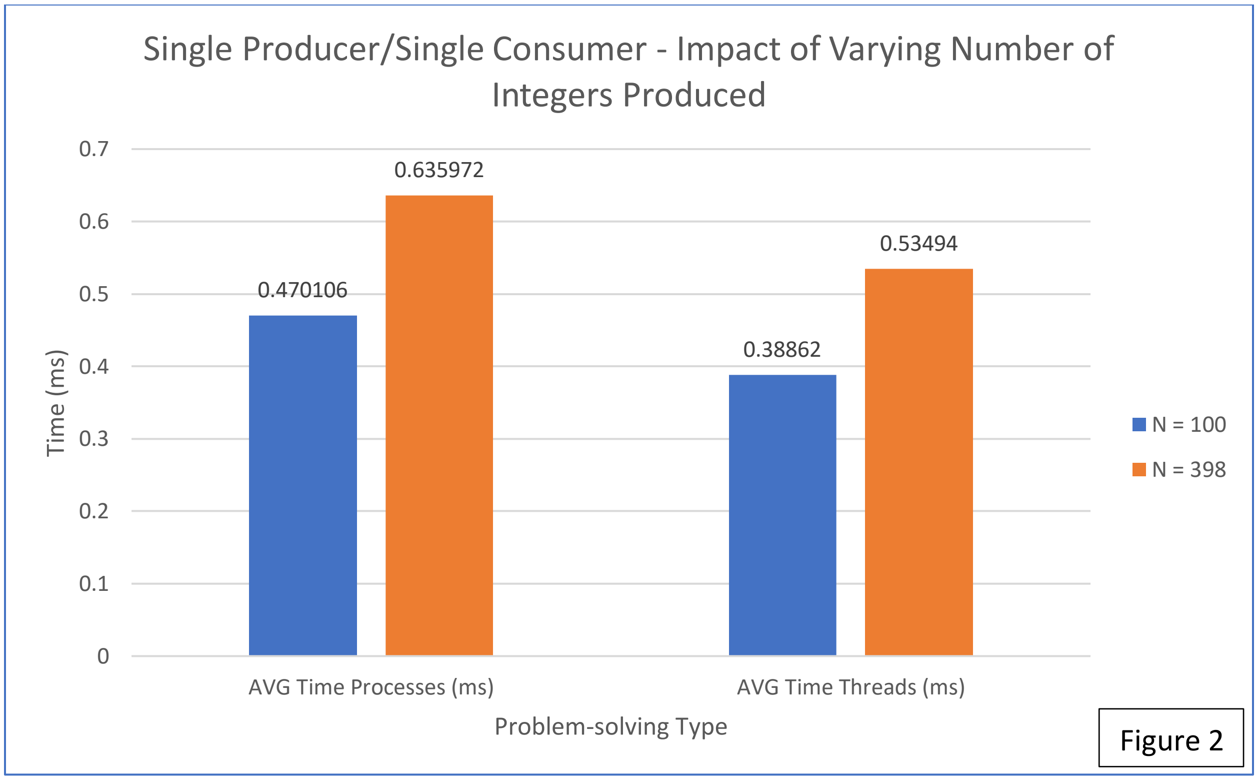

Figure 2

Single Producer/Multi Consumer (2)

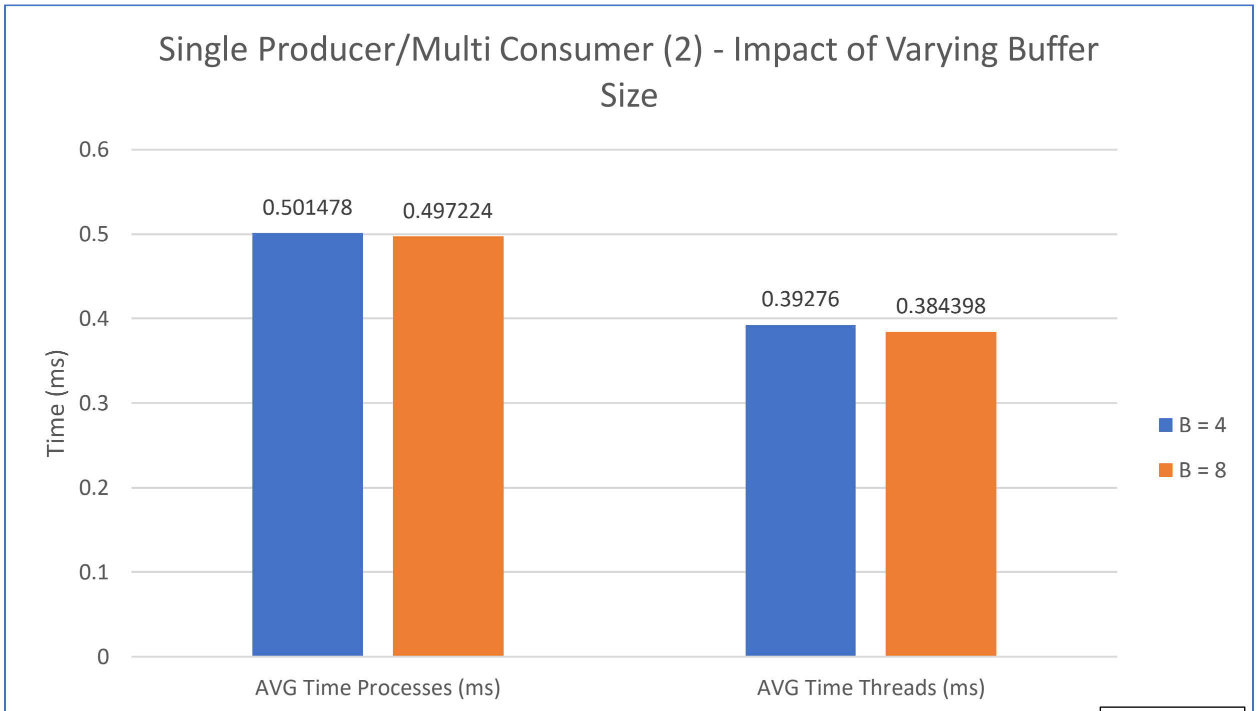

Figure 3

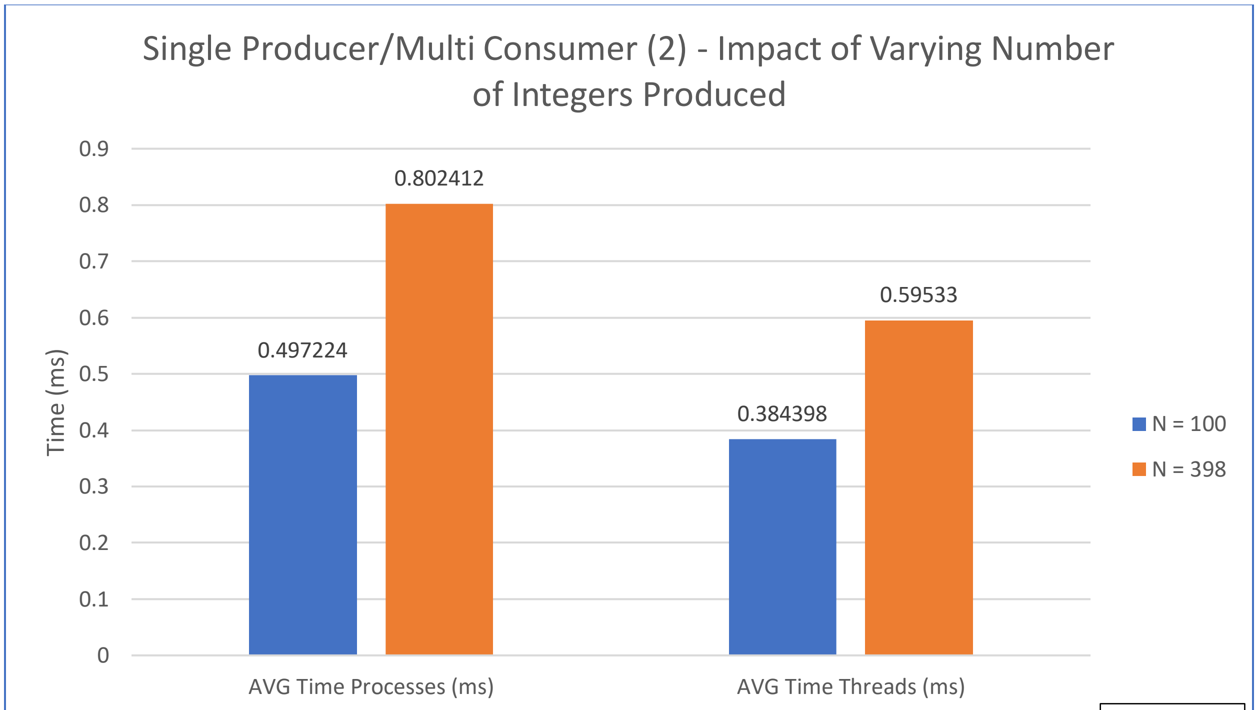

Figure 4

Single Producer/Multi Consumer (3)

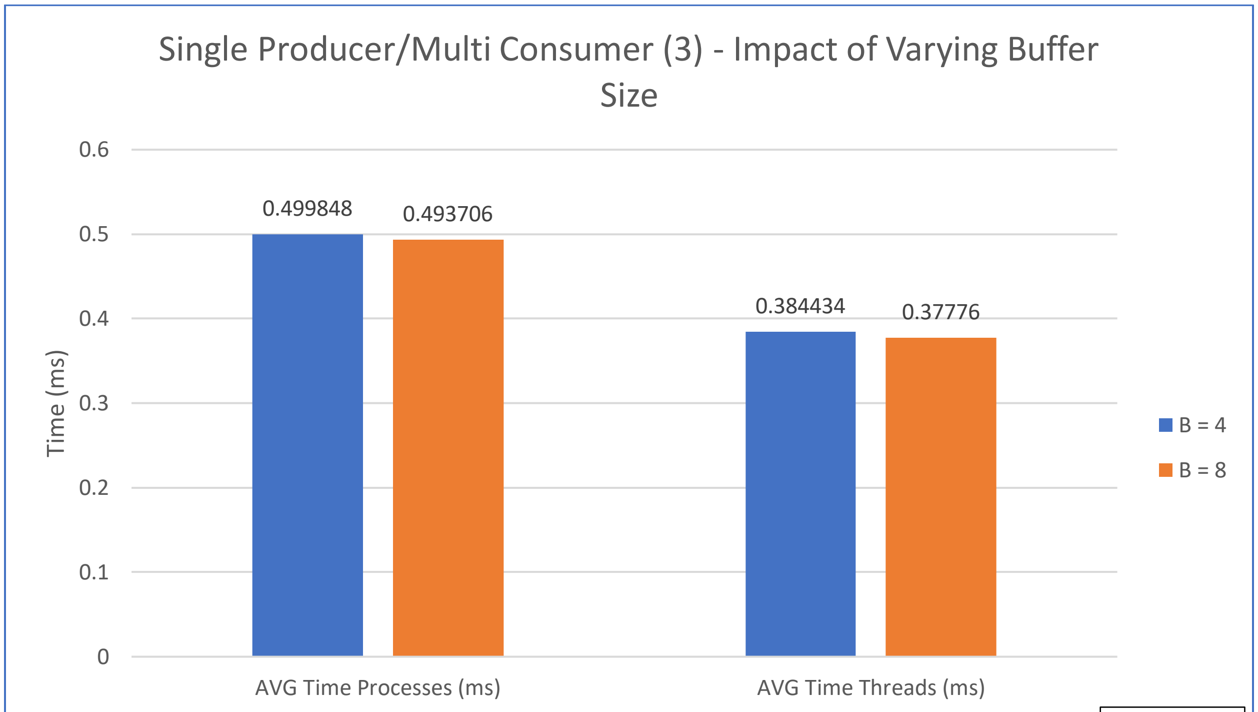

Figure 5

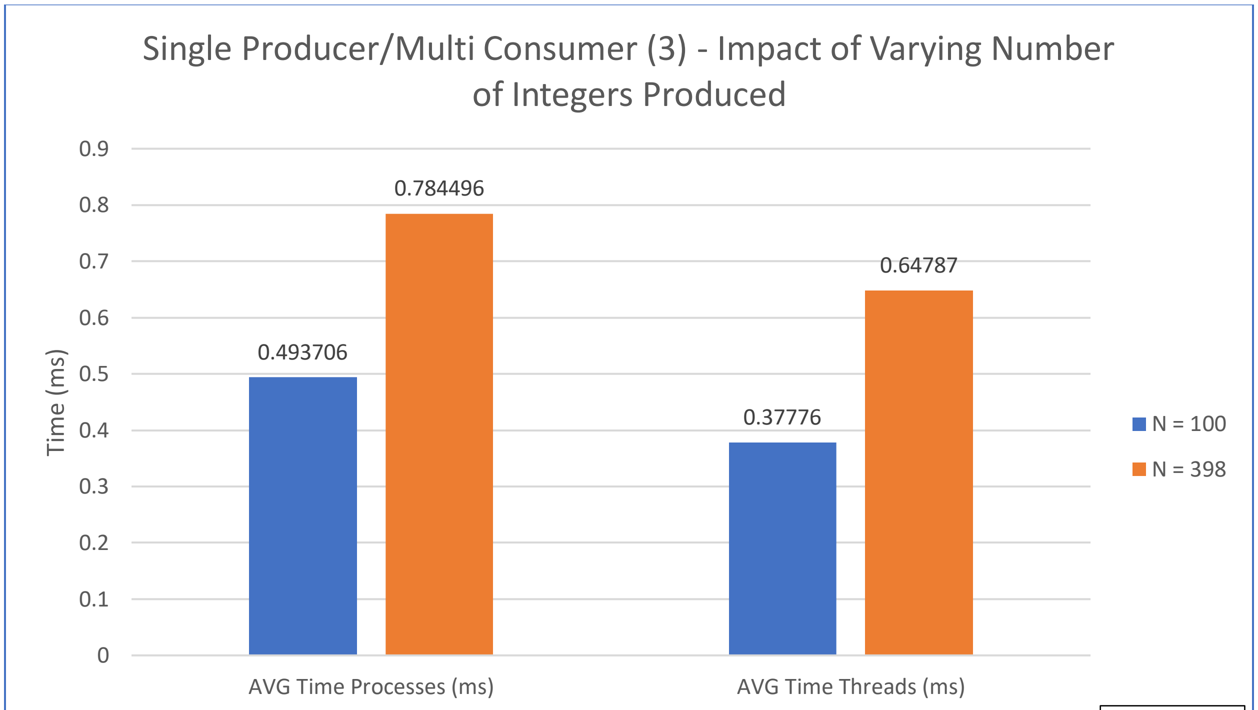

Figure 6

Multi Producer (2)/Single Consumer

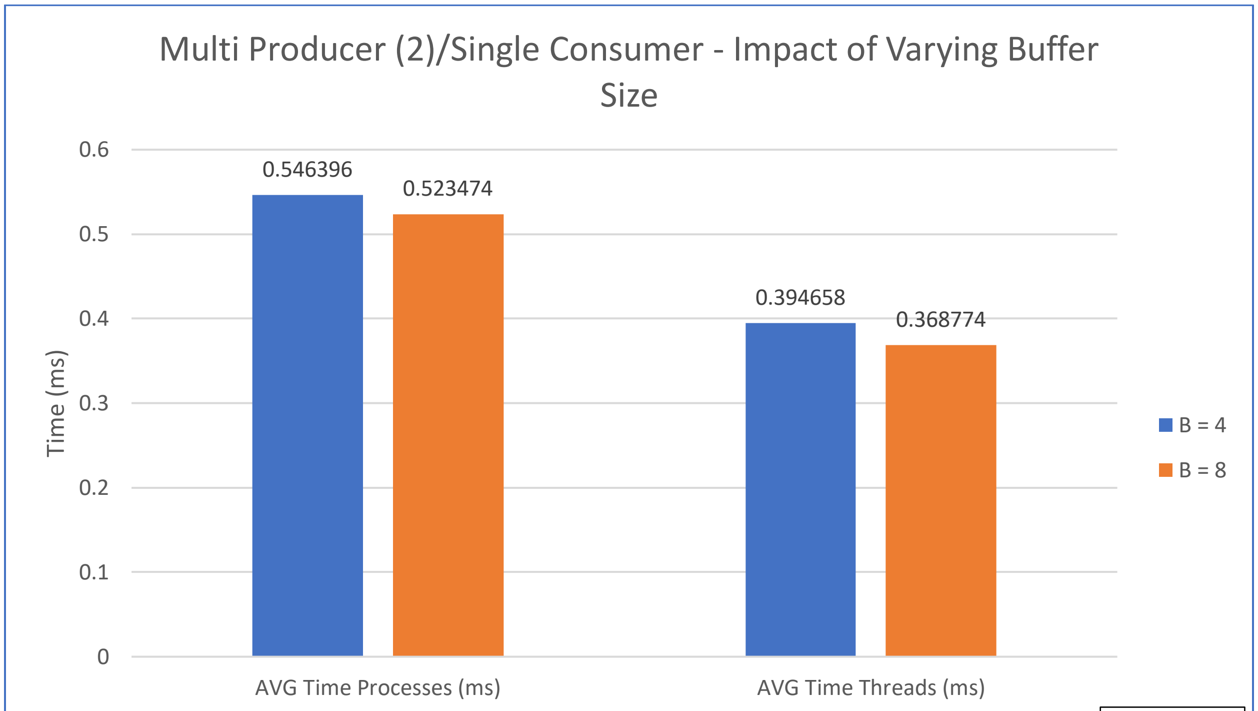

Figure 7

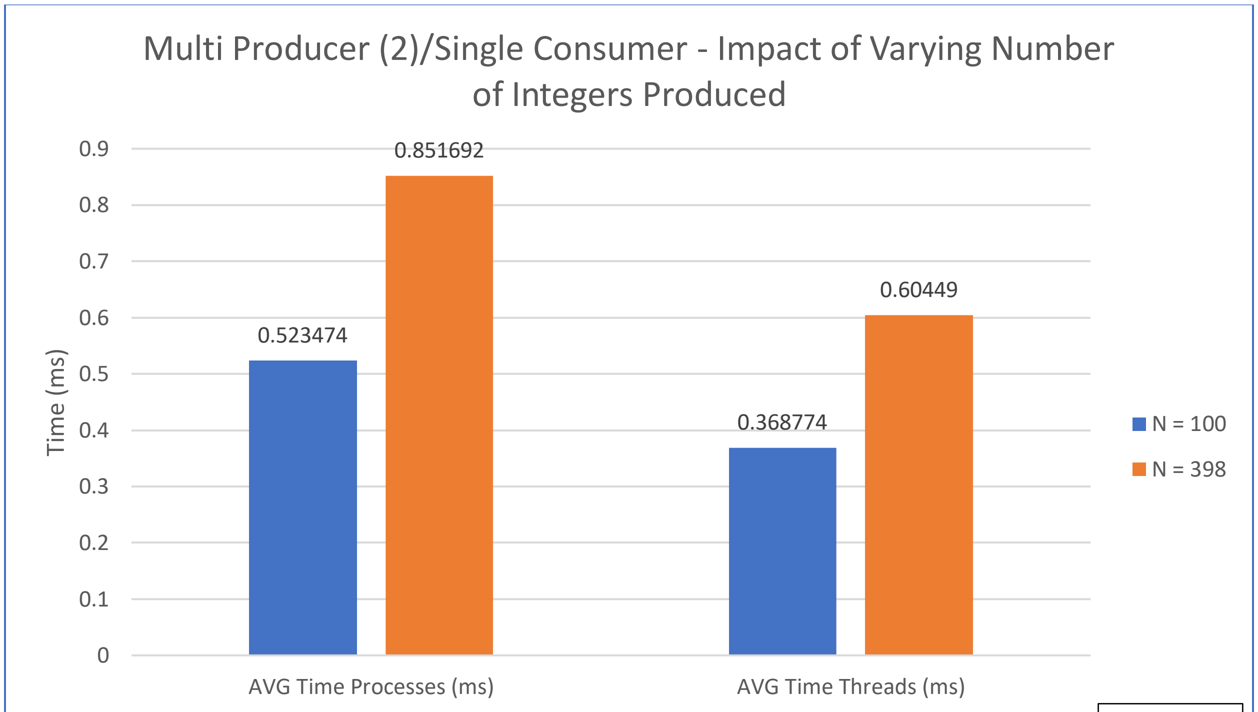

Figure 8

Multi Producer (3)/Single Consumer

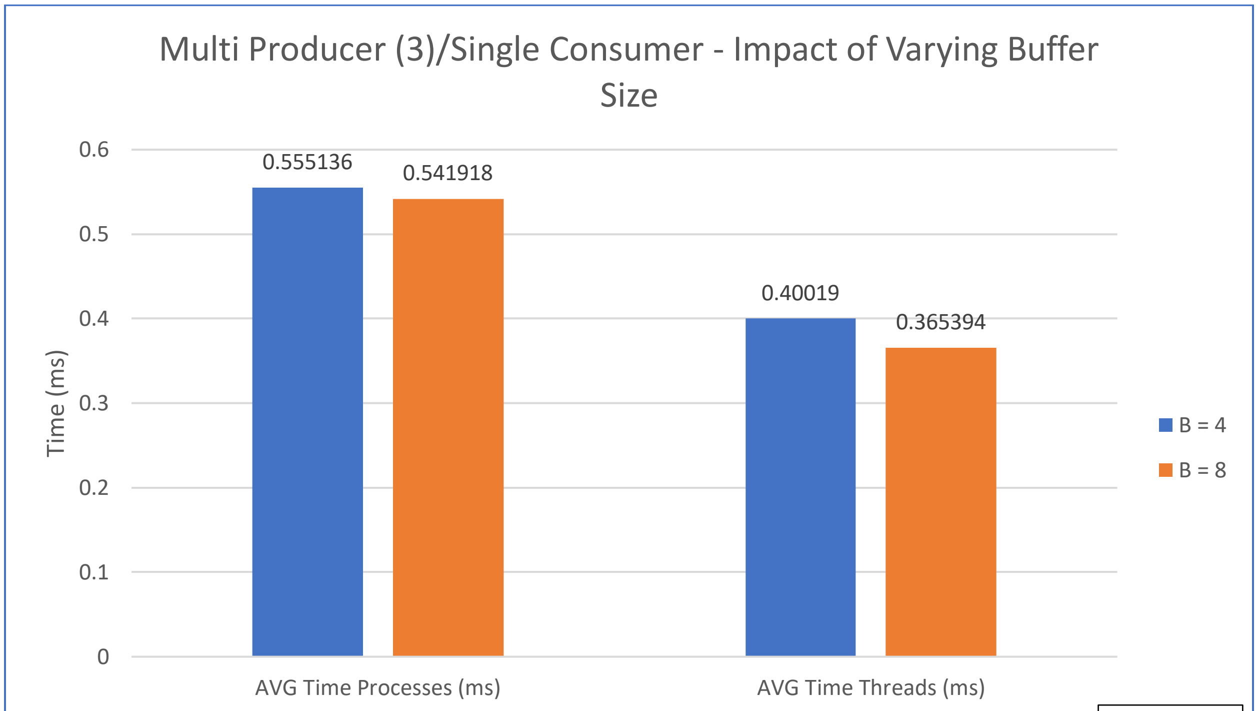

Figure 9

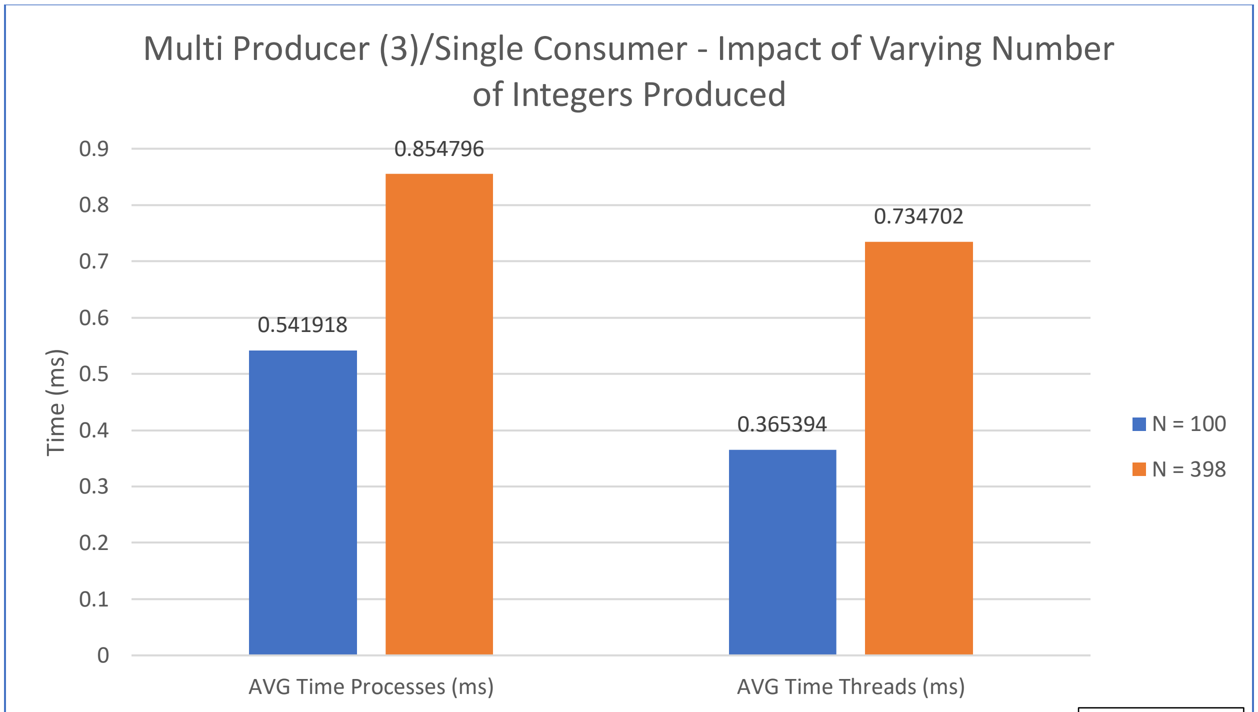

Figure 10

Multi Producer (2)/Multi Consumer (2)

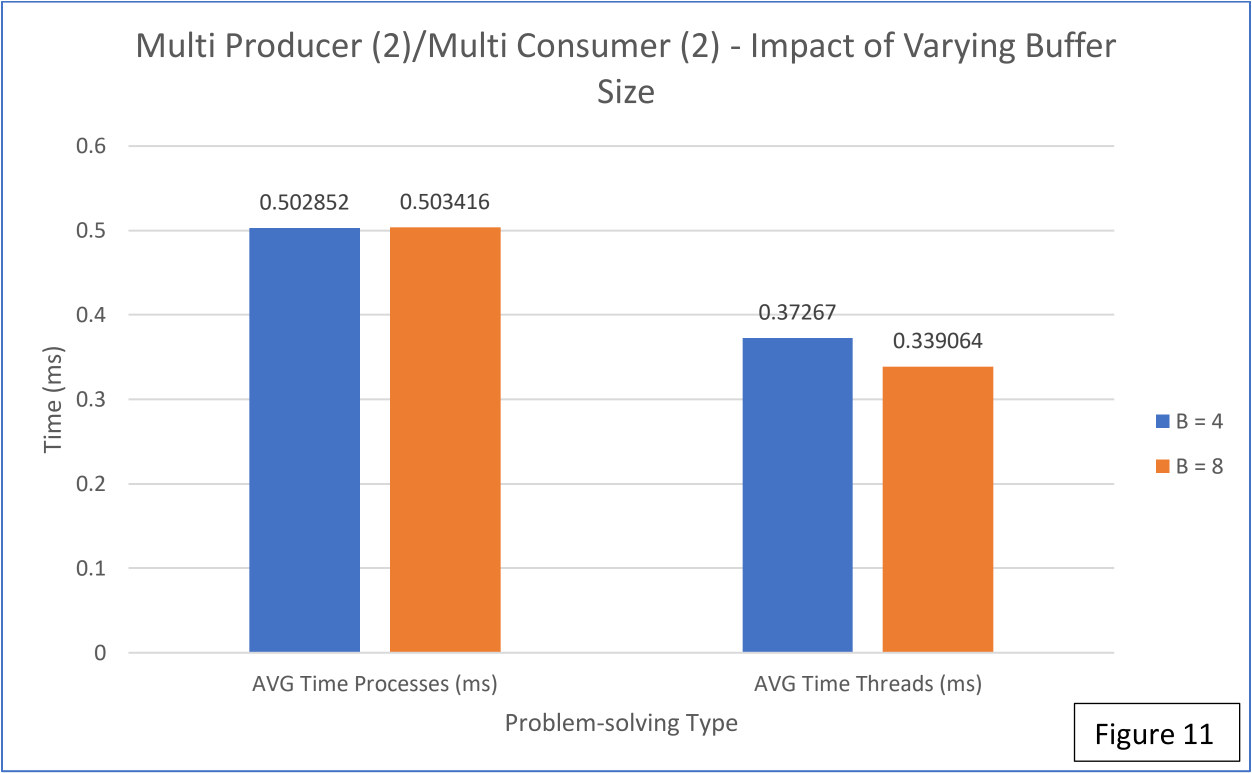

Figure 11

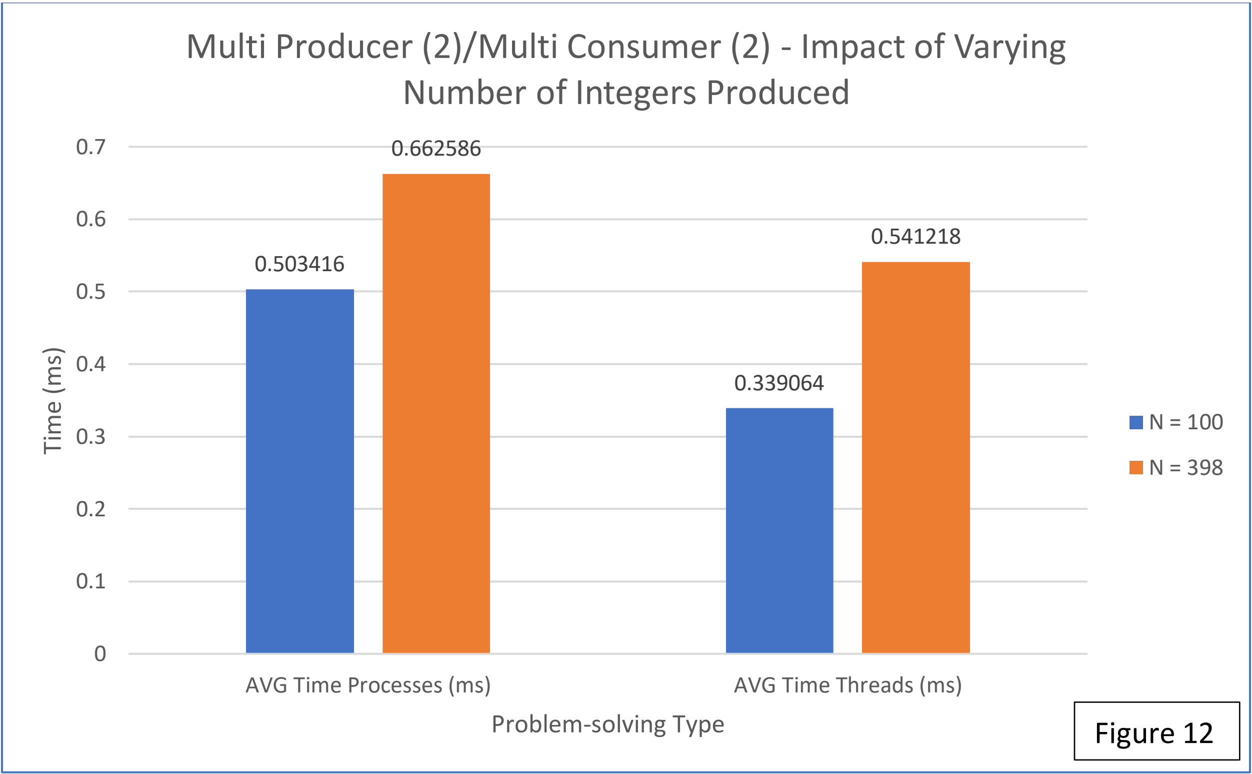

Figure 12

Multi Producer (3)/Multi Consumer (3)

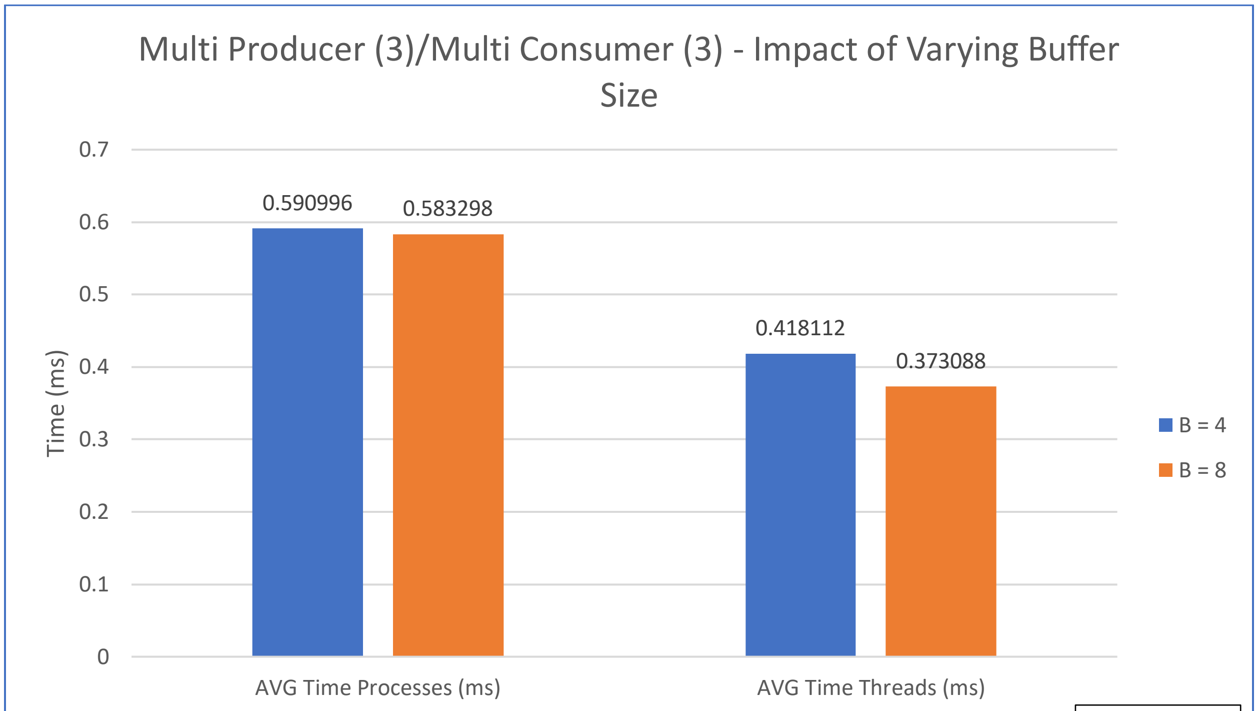

Figure 13

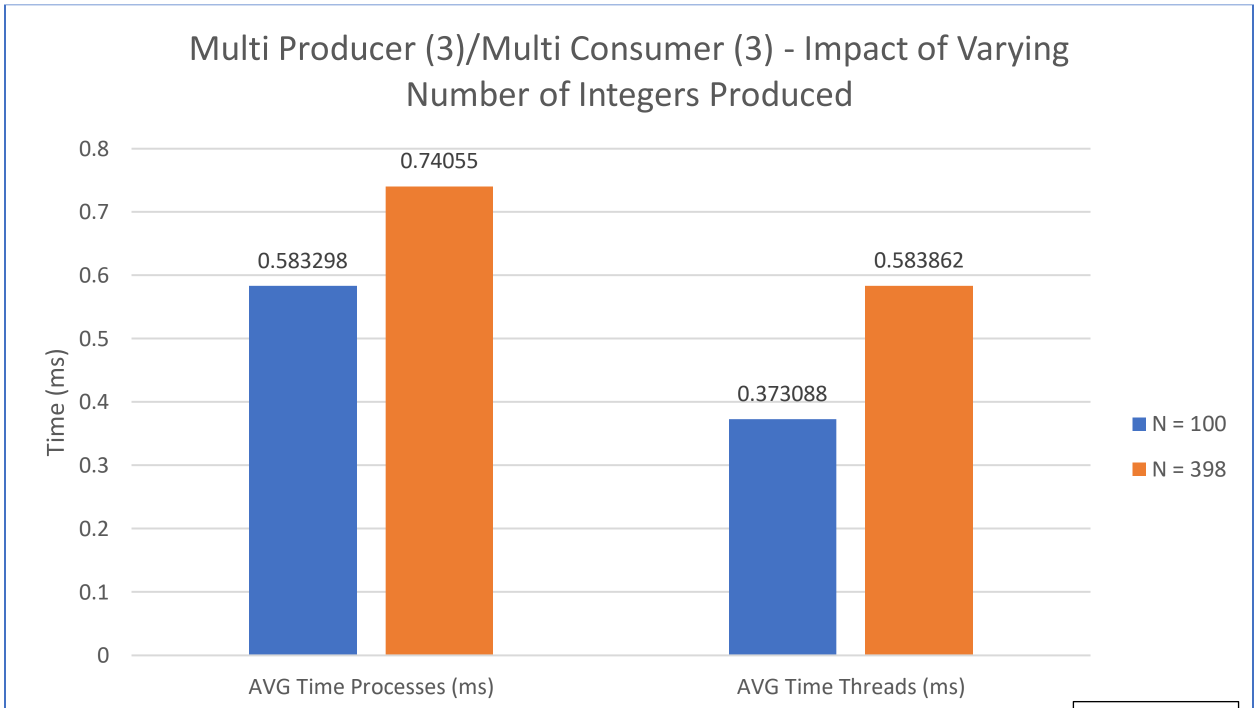

Figure 14

Discussion

Processes and the POSIX Message Queue

POSIX message queues are an efficient IPC mechanism for concurrent processing tasks. They are advantageous in the sense that they allow processes to read and write to an agreed upon location, which prevents processes from accessing each other's memory space – a safe practice. This partially negates the need for direct communication between processes, which would require a process to have access to information about other processes such as their process ID, which is not required with a POSIX message queue. However, POSIX message queues have limited memory space, and if the message that needs to be written to the queue does not fit in the queue, then processes will need to resort to direct communication in order to deliver a message, which is disadvantageous for IPC. Another disadvantage of POSIX message queues involves the copying of data from the process to the message queue, which is an additional step that must be performed in order for interacting processes to view the data.

Threads and Shared Memory

Threads are of the same process, and thus run in a shared memory space. The advantage that is presented in this case is immediate and direct access to data produced by other threads, making this approach much faster than IPC with POSIX message queues. Another advantage to note is that the shared memory space is not as limited in size as would be the case for POSIX message queues; a process' memory space can be up to a few gigabytes in size, so the threads sharing that memory space will certainly have a lot of memory to work with. On the other hand, due to having a shared memory space, safety is compromised for speed, as the data of each thread is transparent and can easily be accessed and manipulated. To make matters worse, if a thread experiences an issue such as a segmentation fault, the entire process will shut down, and consequently all of the other threads will shut down.

Analysis

In every single graph, you'll notice that the approach involving threads required less time than that of the processes, meaning this case applies under all varying conditions. This is due to the creation of processes resulting in greater overhead than threads (due to the child process having to be replicated from the parent process), as well as having to copy data to the message queue, which is not the case when using threads.

Varying Number of Producers/Consumers

In the case of a single producer (see Figure 1, Figure 3, and Figure 5), notice as we increase the number of consumers, the average time required for the approach involving processes actually ends up increasing, while the average time required for the approach involving threads steadily decreases.

- Processes: The average time increase can be attributed by the increase in the number of consumers while the number of producers remain constant – the production flow has not changed but the number of consumers has increased, which theoretically should result in a decrease in average time (more consumers = greater rate of consumption). However, process context switching involves switching the memory address space: this includes memory addresses, mappings, page tables, and kernel resources – a relatively expensive operation which ultimately presents itself as the more dominating factor (considering there's only one producer), which consequently led to an increase in average time.

- Threads: Thread switching is context switching from one thread to another in the same process. The only additional operation that needs to be performed is to update the processor state (such as the program counter and register contents), which is generally an efficient operation. As a result, the increased number of consumers and the lack of any tradeoff due to context switching results in a decreased average time.

In the case of two producers (see Figure 7 and Figure 11), notice as we increase the number of consumers, the average time required in both approaches demonstrates a decrease. As mentioned earlier, process switching is a relatively expensive operation that was a dominating factor in the case of a single producer. In this case, we initially had more processes producing data compared to consuming data. By adding an additional consumer process, even though process switching is costly, there was now a steady net flow (number of producing processes = number of consuming processes) which outweighed the cost of additional process switching, resulting in a decrease in average running time. The same logic can be applied to threads, but without considering the cost of context switching.

In the case of three producers (see Figure 9 and Figure 13), notice as we increase the number of consumers, the average time required in both approaches demonstrates an increase. As we increase the number of consumers, the amount of time that each consumer must wait to enter the critical section will also increase. This increased wait time accumulates across all processes/threads and increases the average running time as a result.

Varying Buffer Size

In nearly every case, it appears that a greater buffer size results in a shorter average execution time. There was a single anomaly, such as in Figure 13, for two producers and two consumers, where the average time is slightly greater for the larger buffer size than for the average time for the smaller buffer size. But generally, increasing the buffer size reduces the quantity of time a producer has to wait for available buffer space. Additionally, consumers will be blocked until producers have had the opportunity to write data to the buffer. Therefore, increasing the buffer size reduces the waiting time for a producer, which consequently benefits the consumer as the producer will be active more often.

Varying Number of Integers

In every case, increasing the number of integers to be produced results in a greater average execution time. This is pretty self-explanatory: greater quantity produced = greater amount of time to consume. From the perspective of the producer, it will have to perform a greater number of operations to write data to the buffer; the consumer will receive a greater load of data to process as a result, thus resulting in an increase in average running time.

Average Time and Standard Deviation

Looking at the results from the file output after running the python script, you’ll notice that the multiple test runs performed for each input combination yielded multiple varying runtimes, which is why an average time is calculated in order to get a better sense of generally how long it takes for that specific input combination to be processed. We use standard deviation as a form of measuring how far on average each data point was from the overall average run time. If the standard deviation is low, we can infer that the data set is more condensed around the average run time, which may indicate a strong correlation between the input and output variables. If the standard deviation is high, we can infer that the data set is more distributed and not as concentrated around the average run time, which may indicate a weak correlation between the input and output variables.