Operating System - Assignment 2 Report

SECOND SEMESTER 2022-2023

DATE: 02/12/22

SUBMITTED BY

Shubhankar Vivek Shastri	ID - 2020A7PS2054H
Akshat Khaitan	ID - 2020A7PS2055H
Aditya Dhanekula	ID - 2020A7PS0205H
Ishan Changani	ID - 2020A7PS0230H
Tanmay Agarwal	ID - 2020A7PS2057H
Abhiraj Khare	ID - 2020A7PS0161H

Analysis

P1

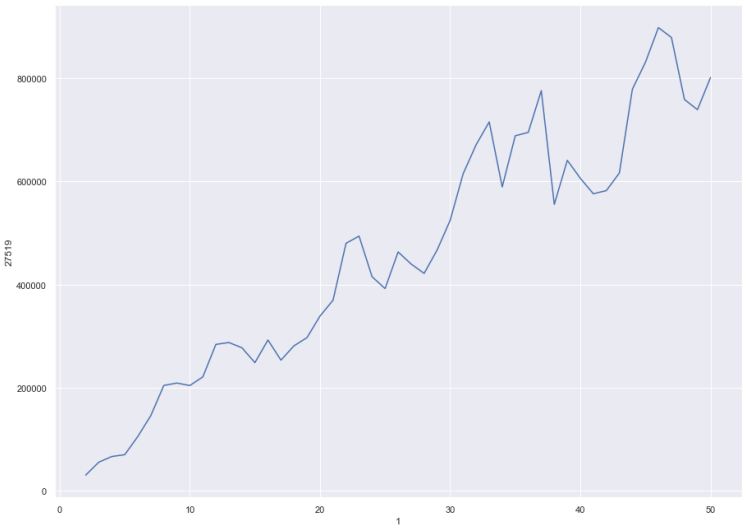
Arguments: i, j, k, in1.txt, in2.txt, p1.txt, thread_count, flag
Flag is a utility argument that helps generate the content in the CSV files: flag==0 indicates that shared memory should not be released. flag==1 indicates that shared memory needs to be released. The value of flag is sent to P1 by the scheduler.

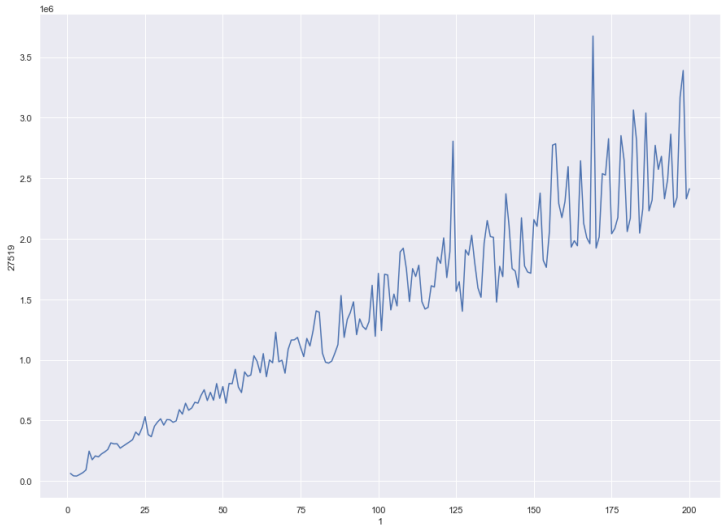
For the first part of the program, P1.c takes input via two text files, in1.txt and in2.txt. We have varied the input sizes of the matrices as follows (square matrices for simplicity in analysis) using 12 different input matrices:
5x5, 10x10, 15x15, 20x20, 25x25, 30x30, 35x35, 40x40, 45x45, 50x50

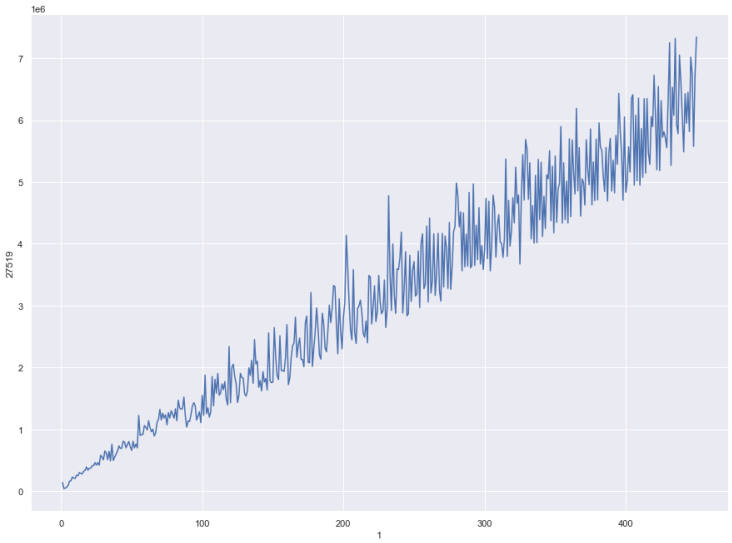
Now for taking inputs by varying the number of threads from 1 to n -
For one thread, the single thread takes input from both matrices spanning all elements.
For two threads, we are allotting a single matrix per thread to read values.
Beyond this, for an odd number of threads, we have made the split as (n/2 + 1) threads to the first matrix and (n/2) number of threads to the second matrix. For an even number of threads, (n/2) threads for the first matrix and (n/2) for the second matrix.
.
.
The maximum number of threads that can be used for taking input is i *j + j * k, where i * j is the size of the first matrix, i being the number of rows and j is the number of columns of the first matrix and j *k is the size of the second matrix. Beyond this, more threads are not required as, in this case, one thread is reading one element only. Moreover, there is no critical section created in Program1.c as we are taking simple input via multiple threads, and one set of elements are accessed only by a single thread. Each element is being read by a single thread.

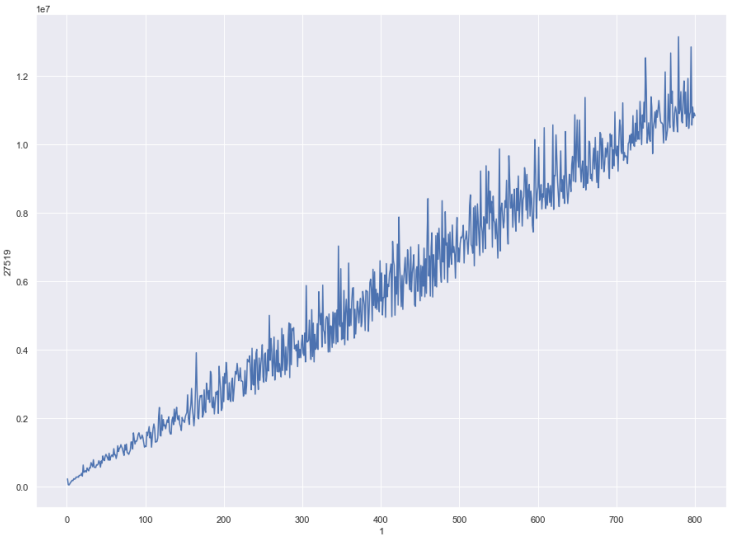
For preprocessing, we are storing the positions of elements in newly generated position matrices using ftell() (storing the transpose of the second position matrix). Now using fseek() and the obtained position matrices, we take the file pointer to the appropriate positions and read the inputs and store it in the shared memory.
For recording the time taken by the process to read the complete inputs for any given number of threads (this time doesn't include the time for preprocessing), we have used the inbuilt CPU timer function.
Once P1 is done, P1 is detaching itself from the shared memory as it does not need to access it anymore but not destroying it as P2 needs to access this shared memory. Moreover, we have destroyed this shared memory in Scheduler.c once P1.c and P2.c are done with their respective jobs.

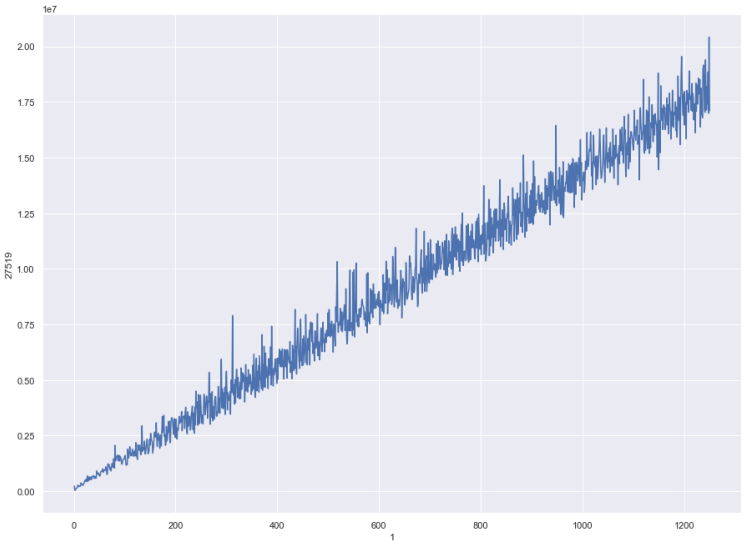
Graphs for P1

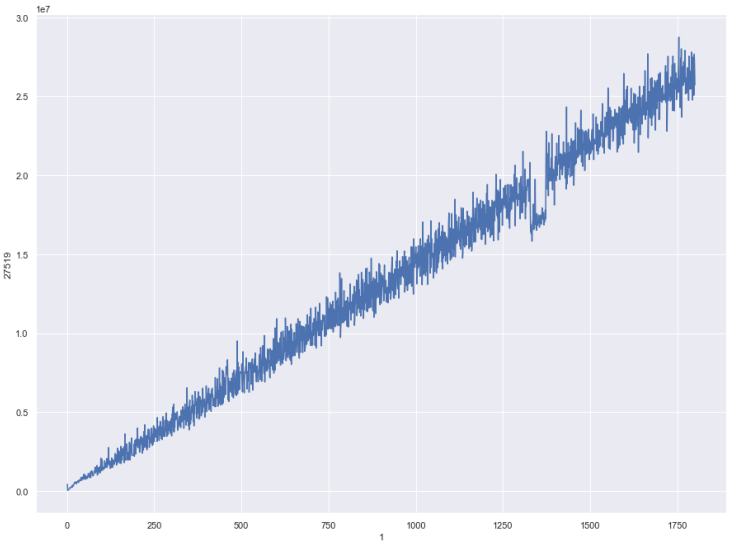


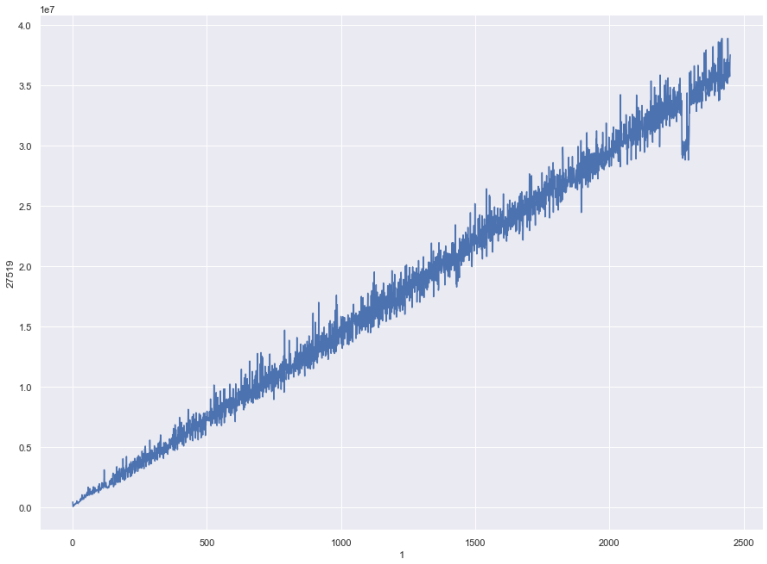


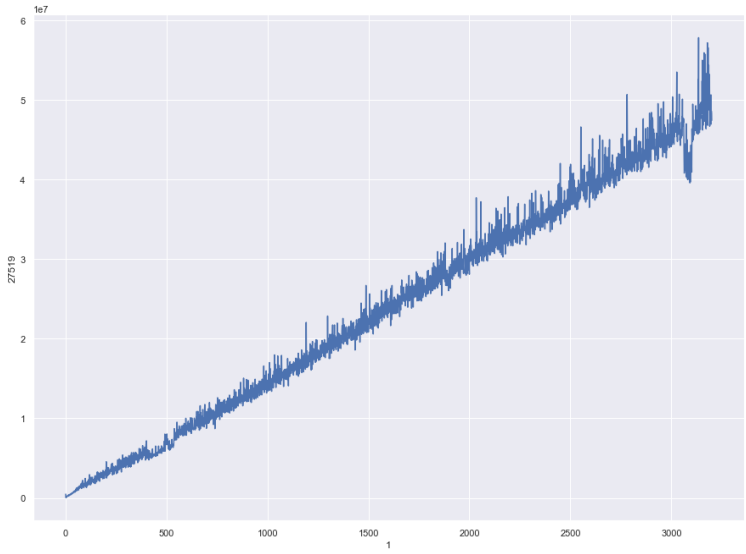


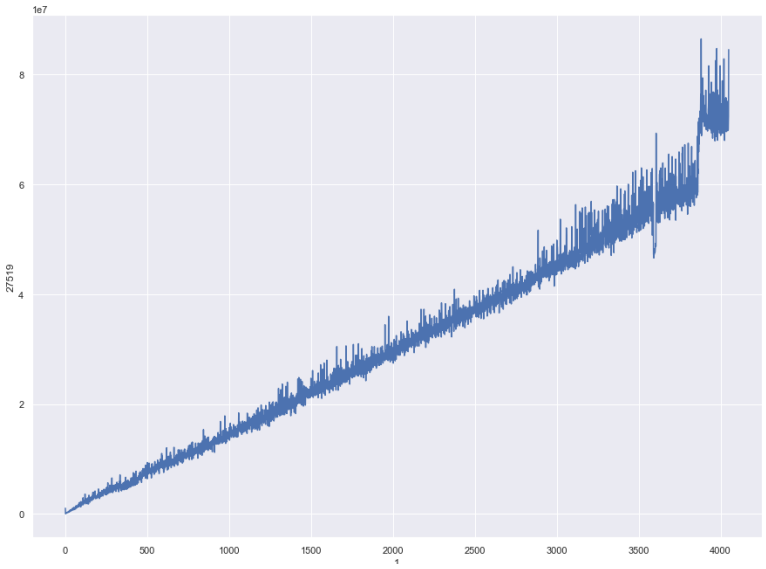


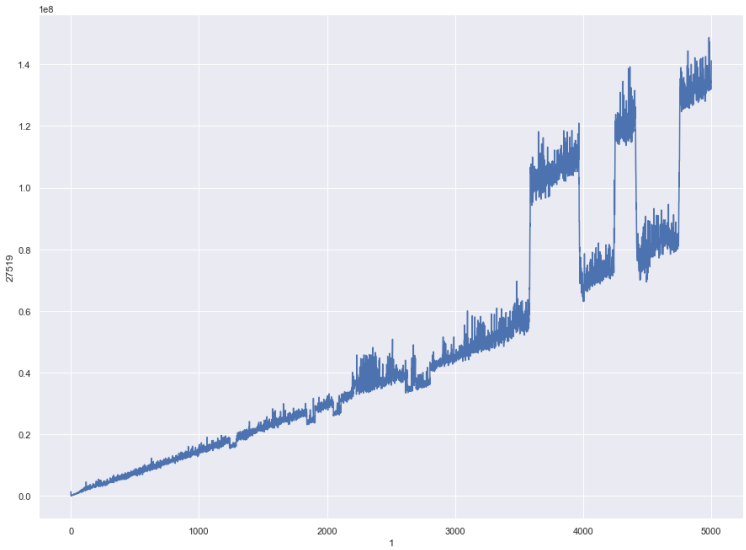





1e7
27519
1


1e7
27519
1


1e7
27519
1


1e8
27519
1


1e6
27519
1
5
10
15
20
25
30
35
40
45
50

Explanation

- We observe that when we increase the number of threads for any input size, the time taken to complete process one increases generally. We know that for a particular thread, the time taken to read x numbers is always greater than the time taken to read y numbers where x>y. Hence the graphs for all input sizes follow a general increasing trend.
- As there are two input files that are to be read, the observed results follow the intuitive solution that both files can be read parallelly by two different threads. Hence, this is always better than the sequential solution of reading both the input files using just a single thread. This is also supported by the results from the graphs as the time taken to read the input files dips sharply when we increase the number of threads to 2 from 1.
- But as we keep on increasing the number of threads beyond 2, the overhead of creating new threads becomes too much when compared to the optimization they provide.
- We can observe that in the close neighborhood of a particular thread when we increase the number of threads, the processing time decreases, but as we increase the number of threads further, the time taken by P1 increases.
- The above observations can be explained by the following reasoning:
    1. As we increase the number of threads, we distribute the work between the increased number of threads which leads to a decrease in the time because parallel threads work together to take the input. As we increase the number of threads, we create an overhead for the OS to create and manage multiple threads simultaneously.
    2. Thus, the time saved due to parallel execution is coming at the cost of extra load on the OS for managing the threads.
    3. Therefore, in the neighborhood of a particular number of threads, the time decreases, but as the latter overpowers the time saved by parallel execution hence the overall time increases again.
- Note: There are certain deviations from the general trend. This is due to the OS scheduling some other background processes while we were running our program.

P2

Arguments: i, j, k, out.txt, p2.txt, thread_count, flag
Flag is a utility argument that helps generate the content in the CSV files: flag==0 indicates that shared memory should not be released. flag==1 indicates that shared memory needs to be released. The value of flag is sent to P1 by the scheduler.

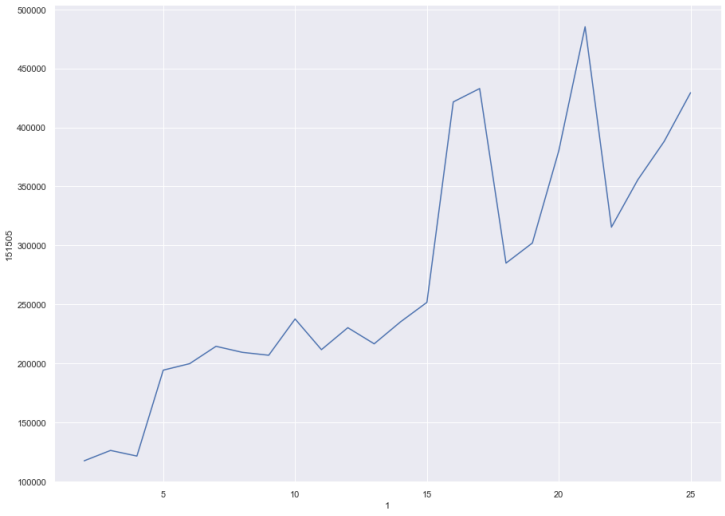
First of all, P2.c needs to attach itself to the shared memory, so it uses the same SHM key as P1 and Scheduler.

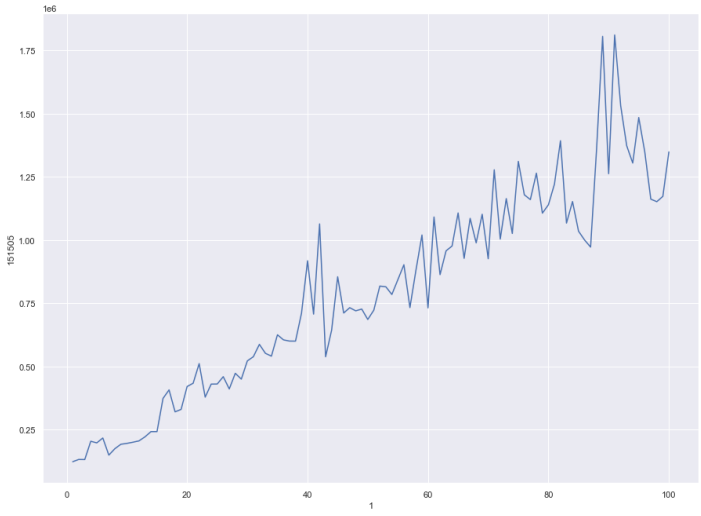
P2.c creates output.txt, which generates a matrix of size i * k, and hence the number of threads can be varied from 1 to n where n is i * k as an individual thread can, at minimum, be involved in the computation of a single element of the output matrix.

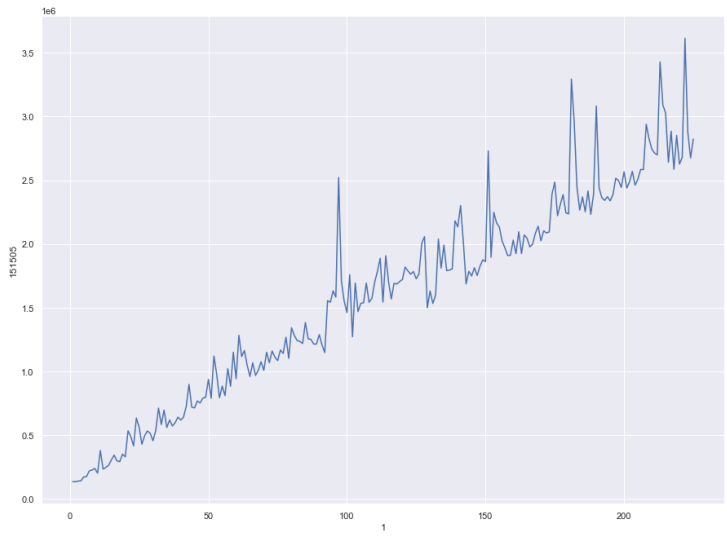
For the computation of a single element (let's say the first element of the output matrix), the first row of the first matrix needs to be multiplied with the first row of the second matrix (transpose), and in case of any flag element is detected in any of the rows used then P2 needs to wait for P1 to take the appropriate inputs by entering a busy wait

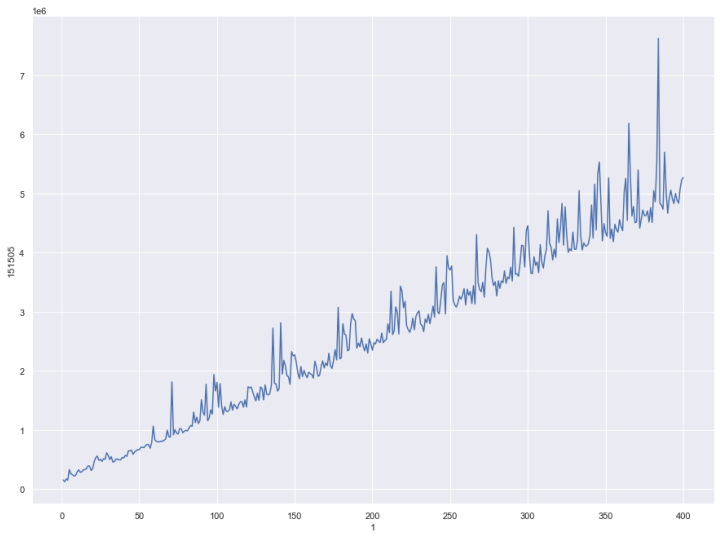
loop. Threads in P2 will begin execution once all elements that are needed by those particular threads have been updated by P1 into the shared memory.
P2 then computes the elements of the output matrix using a given number of threads. The timer begins as soon as P2 starts creating threads and stops when all elements of the output matrix have been computed. The difference between the end time and start time gives the running time of the process for that particular number of threads.

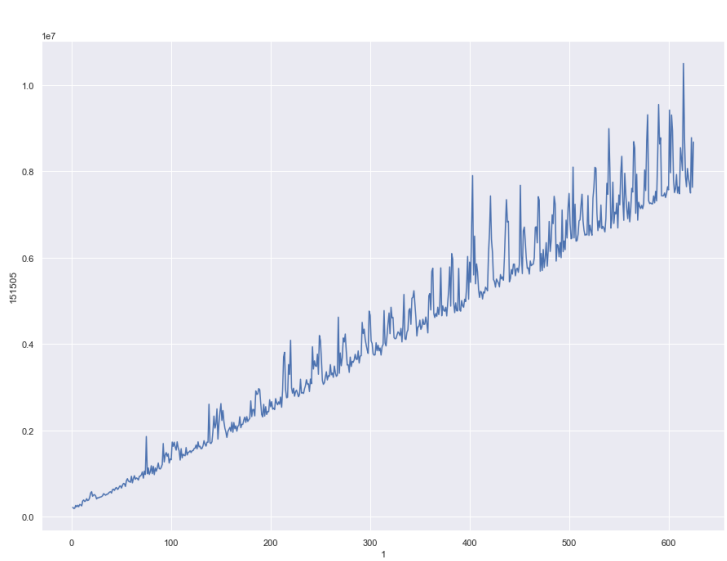
Graphs for P2

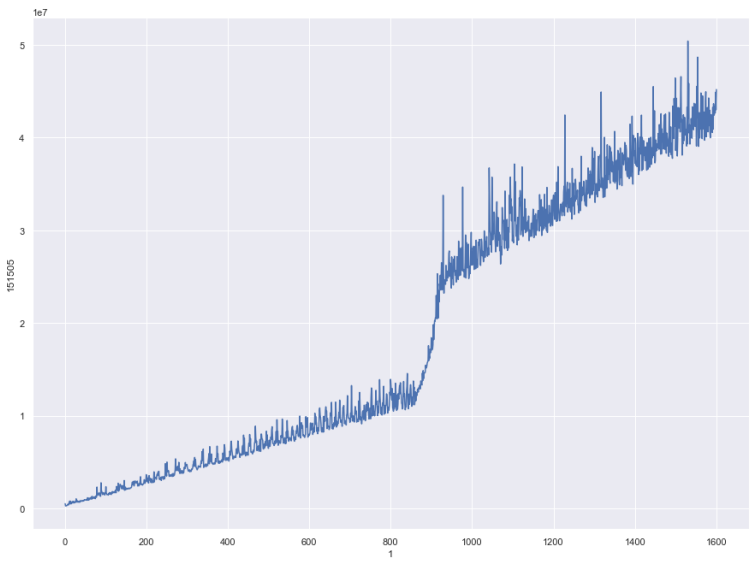


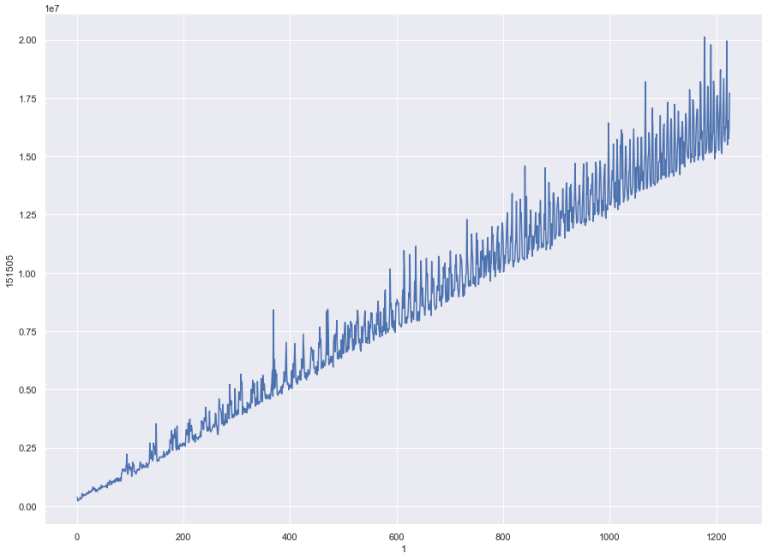


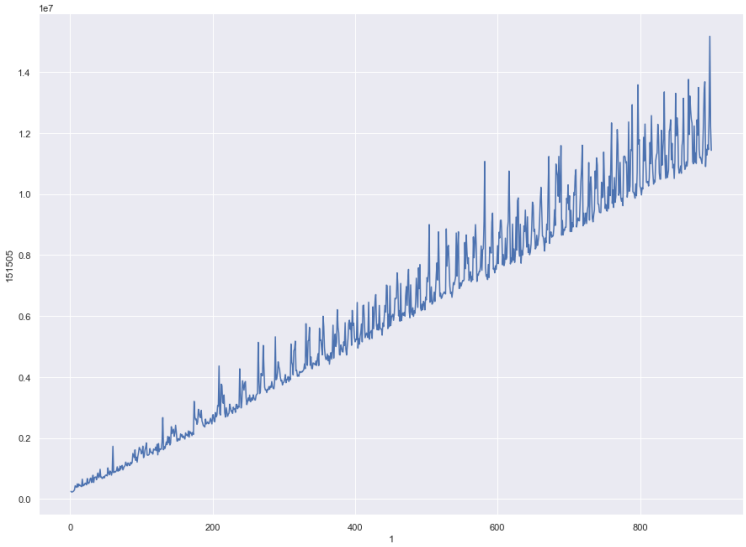


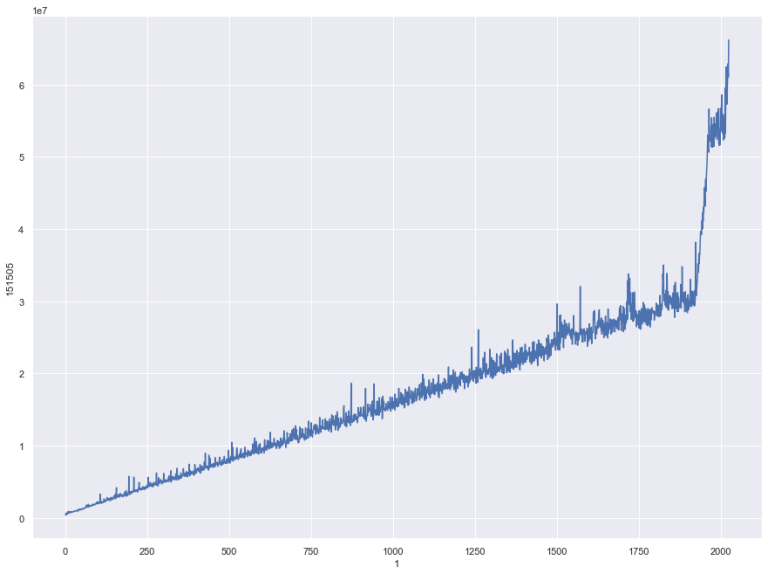


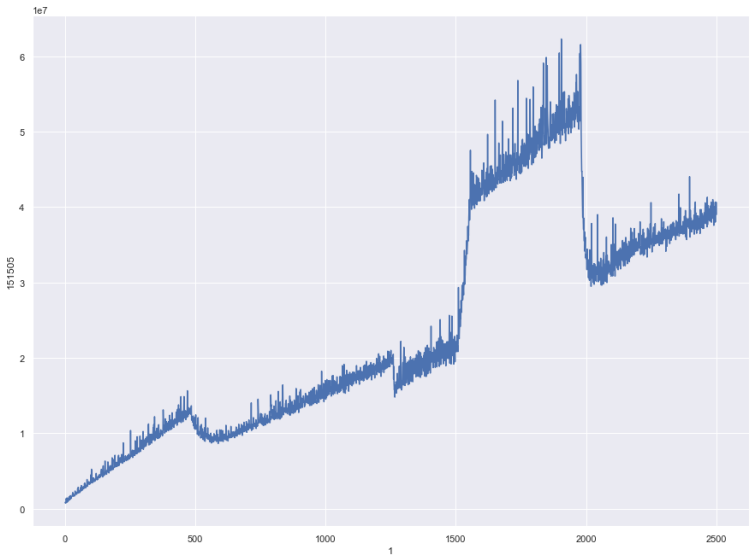


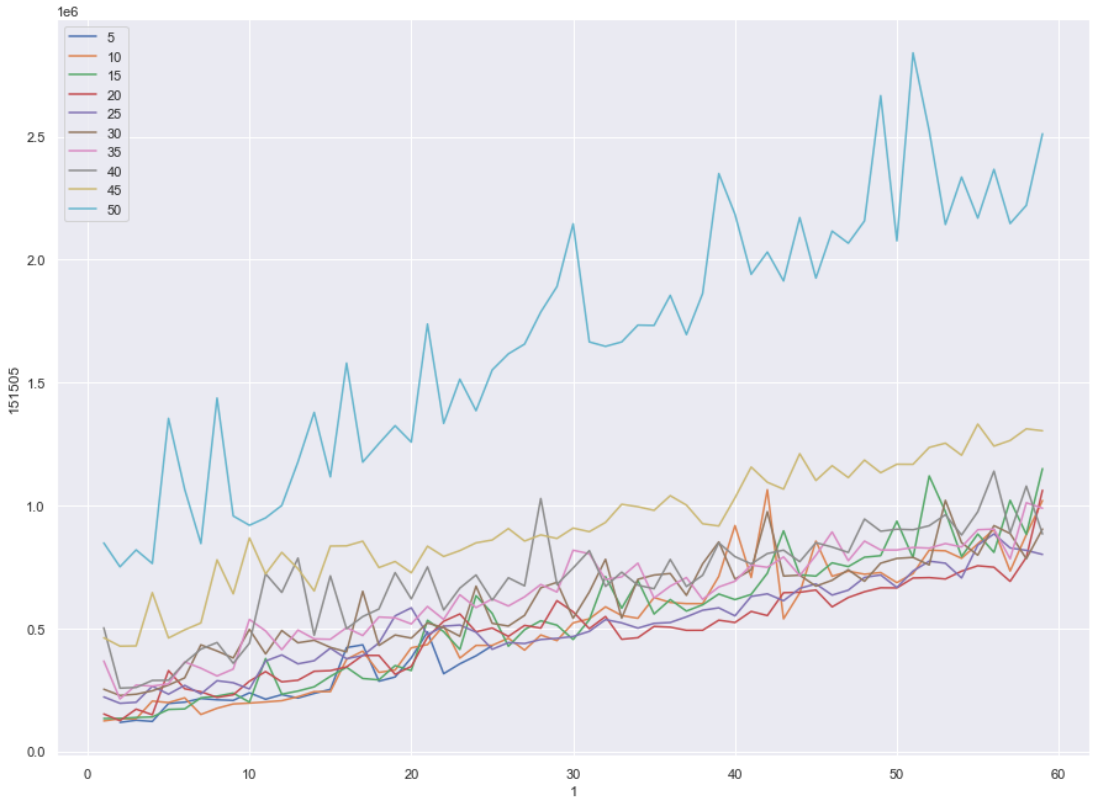



1e7
151505
1


1e7
151505
1


1e7
151505
1


1e7
151505
1


1e6
5
10
15
20
25
30
35
40
45
50
151505
1

Explanation

- We observe that when we increase the number of threads for any input size, the time taken to complete process two increases generally. We know that for a particular thread, the time taken to perform an arithmetic operation between x numbers is more than the time taken to perform the same arithmetic operation between y numbers where x>y. In our problem statement, the arithmetic operation that a thread needs to perform is multiplication followed by addition.
- We can observe that in the close neighborhood of a particular thread, when we increase the number of threads, the processing time decreases, but as we increase the number of threads further, the time taken by P2 increases.
- The above observations can be explained by the following reasoning:
    1. As we increase the number of threads, we distribute the work between the increased number of threads which leads to a decrease in the time because parallel threads work together to perform arithmetic operations to calculate the output matrix. As we increase the number of threads, we create an overhead for the OS to create and manage multiple threads simultaneously.
    2. Thus, the time saved due to parallel computation comes at the cost of extra load on the OS for managing the threads.
    3. Therefore, in the neighborhood of a particular number of threads, the time decreases, but as the latter overpowers the time saved by parallel computation hence the overall time increases again.
- Note: There are certain deviations from the general trend. This is due to the OS scheduling some other background processes that might be running while our program is being executed.

Scheduler

From the graphs of P1 and P2, we get the optimal number of threads for P1 and for P2 for a particular input size. Now we run the scheduler program by varying the input size and using the optimal number of threads, and calculating the waiting time and turnaround time. To run the scheduler program, we first compile the program and generate the .out file and then run this newly generated file using the following command:

```
./group55_assignment2.out i j k in1.txt in2.txt out.txt
```

i: number of rows for the first matrix
j: number of columns of the first matrix and also the number of rows of the second matrix
k: number of columns of the second matrix

To simulate a round-robin scheduler, we first fork the parent into one child(C1) and get access(p1) of that child in the parent. Then we again fork the parent to create one more child (C2) and control it from the parent using p2 where p1 and p2 are process ids of both processes.

The child C1 execs into process P1. For a particular input size, we pass the corresponding optimal thread count that we had obtained from the graph earlier as an argument. Similarly, the other child execs into P2, and we pass the corresponding optimal thread count that we had obtained from the graph earlier as an argument.
We used signals to communicate between two processes. The signals are

- SIGCONT
- SIGSTOP

We used the kill() function to pass a signal to a particular process using the target process id(pid).

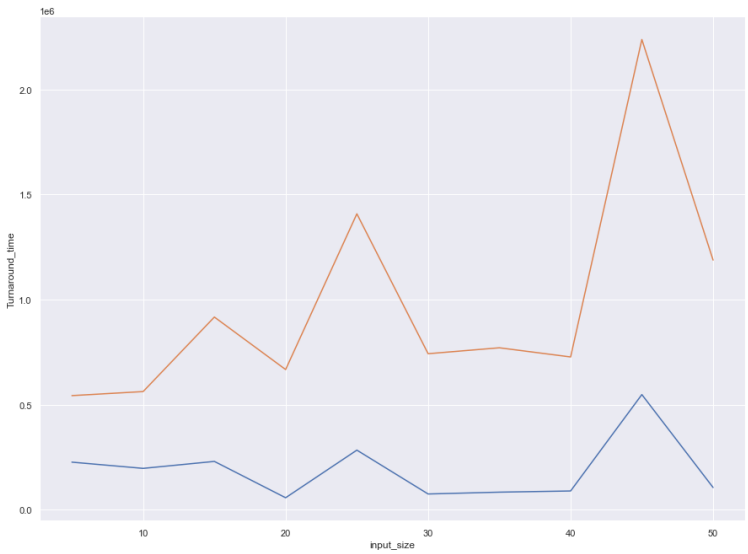
When a process receives a SIGCONT signal, a default signal handler is invoked. This signal handler ensures that if the process was paused by an earlier signal, then it will resume all the paused threads of the process and resume its working.
When a process receives a SIGSTOP signal, it invokes a default signal handler which pauses the execution of all the threads corresponding to that process until a further signal is received.
We utilize this property to simulate the round-robin scheduler.

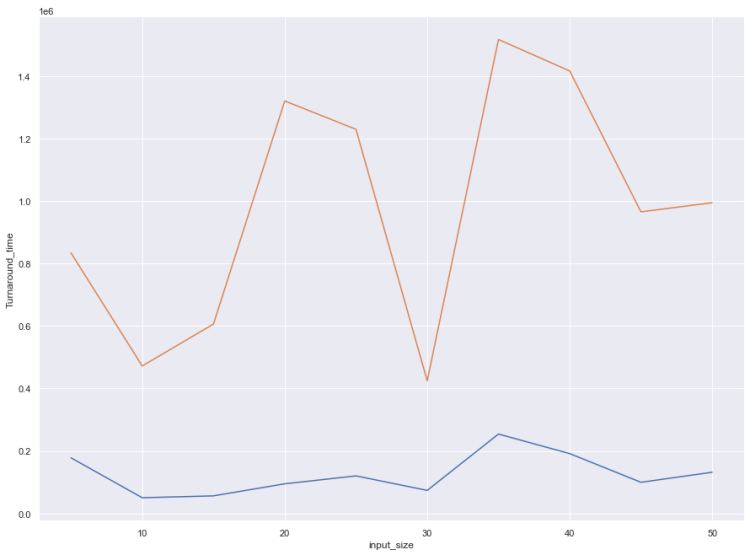
As soon as the parent starts running after forking the two children, we pause the execution of both the children C1 and C2 using P1 and P2, respectively.
Then we create a do while loop in which we alternatively continue C1, pausing C2, and vice versa. The exit condition for this loop will be when both C1 and C2 have finished their execution. If only C1 has finished execution, then the loop will take care that only C2 is scheduled in the subsequent time quantum.

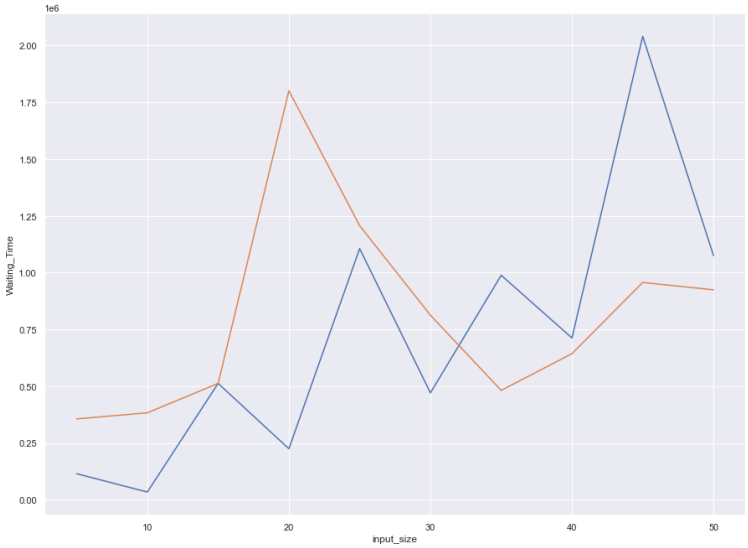
We record the starting and ending times of the schedular program, C1 and C2.
Let's assume six variables
a: Real Staring time of scheduler (ms)
b: Real Starting time of C1 (ms)
c: Real Starting time of C2 (ms)
d: Real Ending time of scheduler (ms)
e: Real Ending time of C1 (ms)
f: Real Ending time of C2 (ms)
g: Process Staring time of scheduler (ms)
h: Process Starting time of C1 (ms)
i: Process Starting time of C2 (ms)
j: Process Ending time of scheduler (ms)
k: Process Ending time of C1 (ms)
l: Process Ending time of C2 (ms)

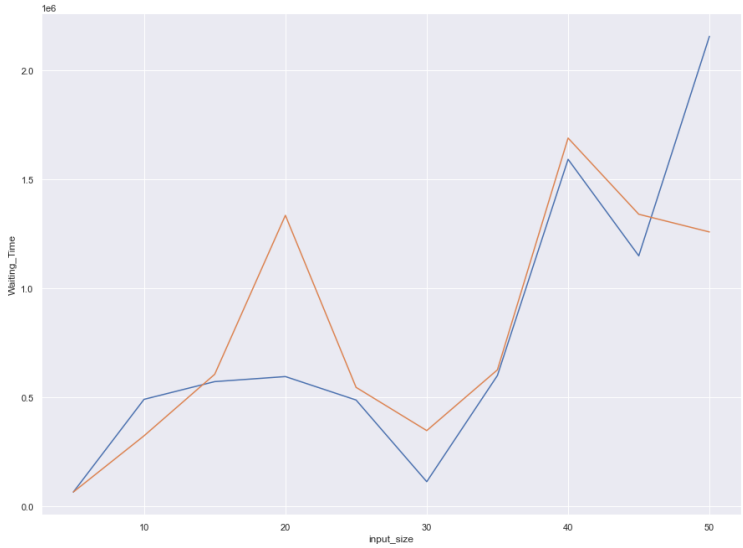
Therefore turnaround time for C1: e - b (ms)
Similarly for C2: f - c (ms)
Waiting time for C1: (e - b) - (k - h)
Waiting time for C2: (f - c) - (l - h)

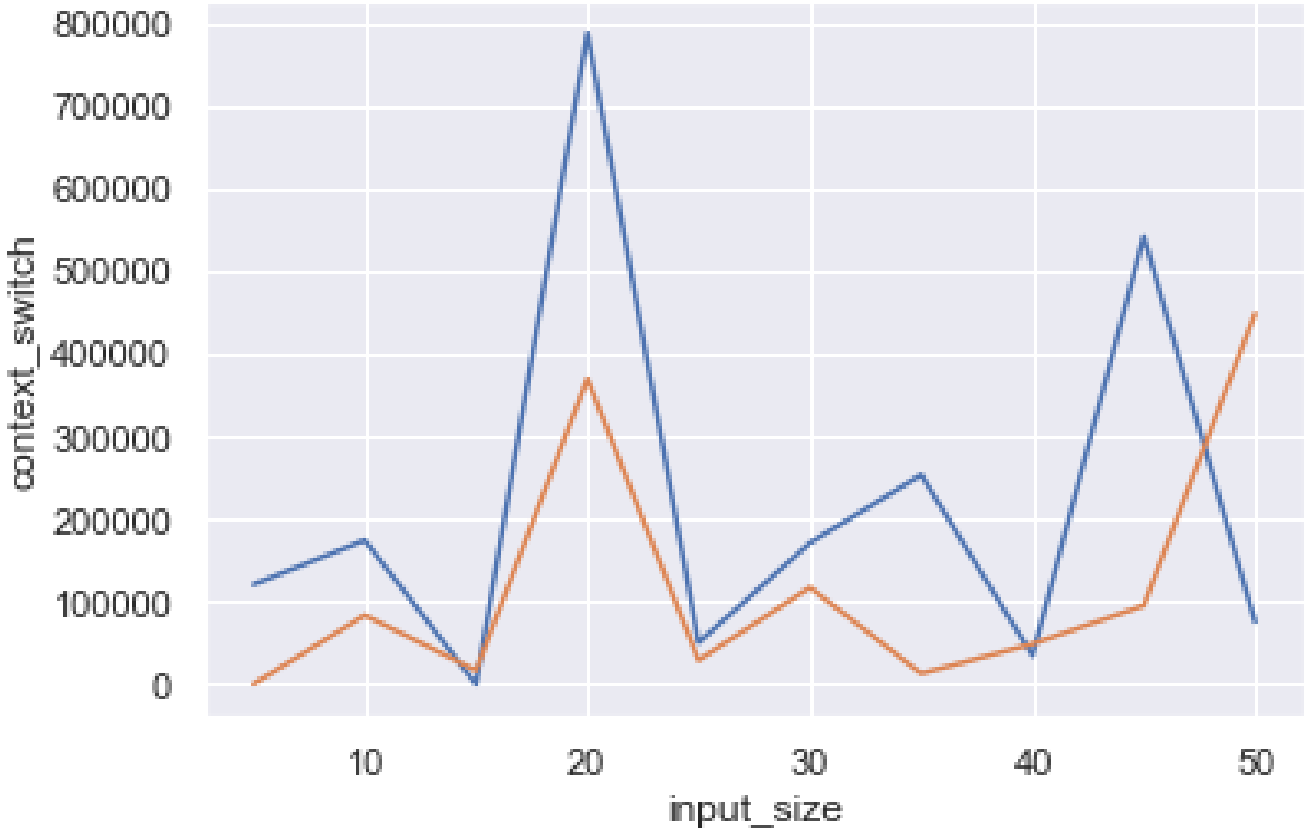
Graphs for Scheduler






1e6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Waiting_Time
10
20
30
40
50
input_size


1e6
2.0
1.5
1.0
0.5
0.0
Waiting_Time
10
20
30
40
50
input_size


800000
700000
600000
500000
400000
300000
200000
100000
0
context_switch
10
20
30
40
50
input_size

Explanation

- For the turnaround time vs input size graph, we see that for both time quanta tq = 1ms and tq = 2ms the turnaround time for process 2 (C2) is more than the turnaround time for process 1 (C1). This observation holds as we gradually increase the input size. For tq = 1ms the same graph is observed.
    1. Since P2's computation depends on how fast P1 is reading the input, it is guaranteed that P2 will always finish after P1 has completed taking the inputs. Since both P1 and P2 start at the same time and P2 ends after P1, the turnaround time for P2 is guaranteed to be more.
- For the waiting time vs input size graph for both tq = 1ms and tq = 2ms, the waiting time for P2 is more than the waiting time for P1
    - In terms of the programming logic, P1 only waits when P2 has been scheduled by the processor but for P2 in addition to waiting while P1 is running it also has to wait for P1 to take the input which it is going to compute. If P1 has not yet read the input from the file, then P2 must wait until the input is ready. This dependency of P2 on P1 creates additional waiting overhead for P2. Thus we observe for both graphs tq = 1 and tq = 2, the waiting time for P2 is much more than the waiting time for P1.
    - For the context switch graph for the line graph represented by the blue line where the time quantum is = 1 the number of context switches is more as for less time quantum our process will have to change its context more. Whereas, for the time quantum = 2 the number of context switches will always remain lesser than that for tq = 1 as we can observe in the graph. Thus, in terms of context overhead, we will choose time quanta = 2ms for all the input sizes.