CS F372 Operating Systems

Course Project: Multithreaded Matrix Multiplication

By Group 13

Nikhil MS- 2020A7PS1303H
Yashovardhan Gunjal-2020A7PS2092H
Madhur Saraf Jain- 2020A7PS1106H
Siddhesh Bahety- 2020A7PS1686H
Kulkarni Sreyas-2020A7PS0076H
MITTAPALLI SREETHEERDHA-2020A7PS1889H

DATA PRE_PROCESSING

We use a python script to preprocess the data. We add a new row at the top of the matrix which gives column numbers to the matrix. For input2, we also transpose the matrix before doing the above step. Both the scripts have been included in the given code folder(trans1.py and trans2.py respectively). Our file names for P1, P2 and S process are p1.c, p2.c and group13_assignment2.c respectively. Inputs are taken as inp1.txt and inp2.txt and output is generated as out.txt. Our programs to benchmark P1 and P2 are p1_bench1.c and p2_bench2.c.

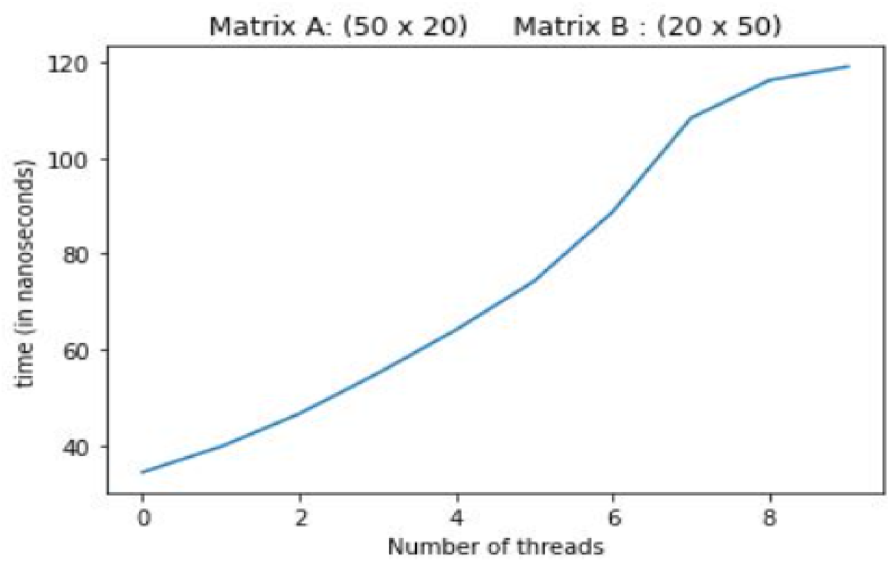
PROCESS P1

AIM:

We have to write C program P1 which takes three filenames (in1.txt, in2.txt and out.txt) as command line arguments. The first two files (in1.txt and in2.txt) contain two matrices of arbitrary size but satisfying the criteria for matrix multiplication. The sizes will be passed in the command line. P1 spawns n threads which each read row(s) and column(s) each from in1.txt and in2.txt.

APPROACH:

We have a file pointer that stores the offset size of the array elements ,i.e., each line till the pointer that it reaches is “\n” and stores it.Now using the offset size array, we allot the different threads their respective OFF_SET passed in the fseek() function which helps them to read the data according to the size of the line in the matrix. We divide the number of rows/columns equally into all the threads until we reach a pointer where number of threads = number of rows/columns. Beyond that, every new thread adds to the overhead of the process. Once threads have been alloted, we have a parallel array isRead() which acts like a mutex lock and informs the threads if the row/column has been read.



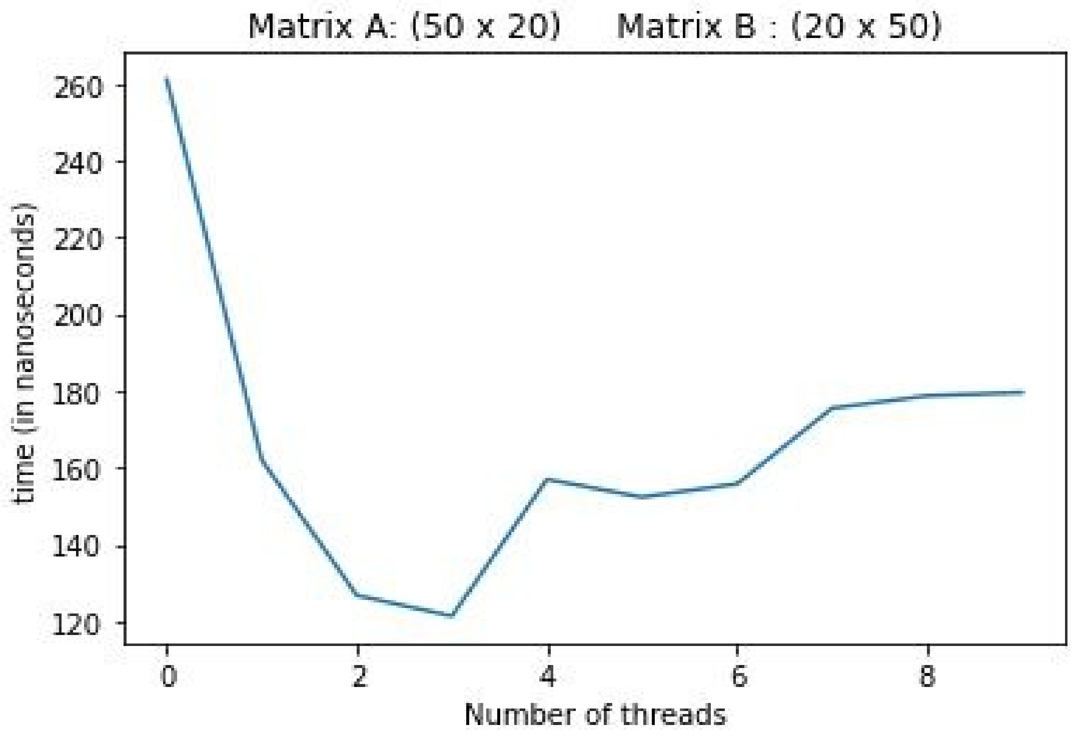
Our analysis of all the graphs in the project shows the code is too optimized for each thread and hence one single thread is enough for reading the matrix. Hence we see an increasing fashion in the graph attached below. Hence we find out the optimum number of threads in the multiplication process P2 since it is much more demanding in the compute time and we see the ideal graph shapes there. We use bash scripts to run each thread epoch 1000 times(i.e., 1000 times for noOfThreads = 1, 1000 times for noOfThreads = 2 and so on) and average them to get stable values and remove outliers.

PROCESS P2

AIM:

To write a C program P2 which uses IPC mechanisms to receive the rows and columns read by P1. P2 spawns multiple threads to compute the cells in the product matrix. The program stores the product matrix in the file out.txt

APPROACH:



A total of four shared memories have been created post P1 process and a struct has been created to allocate rows and columns for each thread and each thread first checks if the row has been read by checking the value in the “isRead” shared memory. If the value is true then the thread proceeds to multiply the corresponding rows and columns and finally it saves this value in the output matrix. And it finally detaches the shared memory and frees the allocated memory. Our analysis of the graph below shows that the optimal value of threads for the multiplication process is N = 3, and hence since reading matrix doesn’t take much time our global optimal value for number of threads is also N = 3.
We use bash scripts to run each thread epoch 1000 times(i.e., 1000 times for noOfThreads = 1, 1000 times for noOfThreads = 2 and so on) and average them to get stable values and remove outliers.

SCHEDULER

AIM:

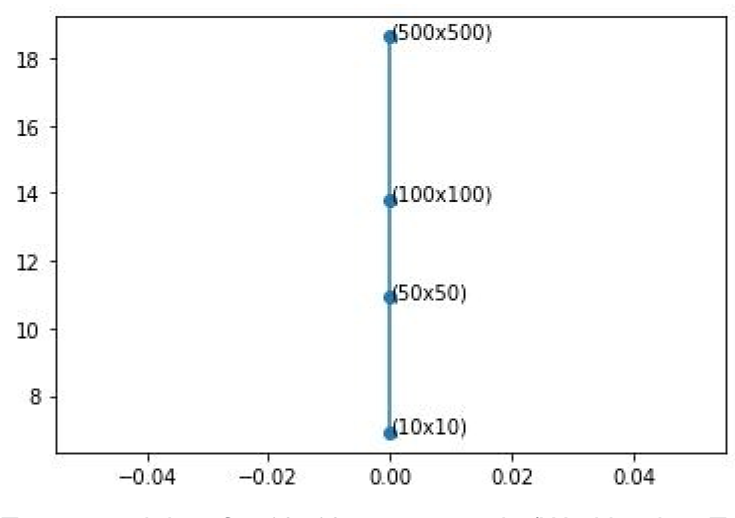
We have to write a scheduler program S. S spawns 2 children processes which exec to become the processes P1 and P2 in part (b) and part (c) above. S uses some mechanism to simulate a uniprocessor scheduler. That is, it suspends P1 and lets P2 execute, and vice versa.
To Simulate the following scheduling algorithms in S:

i)Round Robin with time quantum 2 ms
ii)Round Robin with time quantum 1 ms

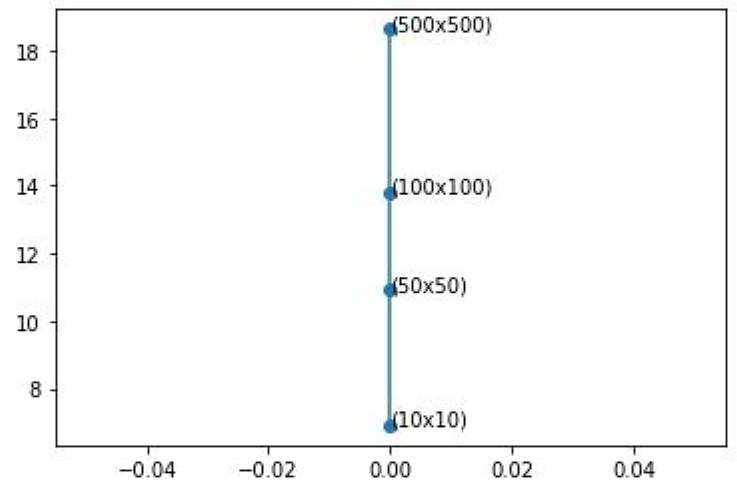
APPROACH:

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way. It is basically the preemptive version of First come First Serve CPU Scheduling algorithm.
We use round robin scheduling to switch between the process P1 and P2. We start by forking the main S process twice and exec them into P1 and P2 respectively. We then use sigtstp signal to stop both the processes and then use a while loop to switch between both of them using sigtstp and sigcont signals of kill function alternatively after every time quantum. We also check the signal sent by kill 0 to check if the process has ended successfully. We then calculate the turnaround time, waiting time and switching overhead for each process using their individual time quantums.

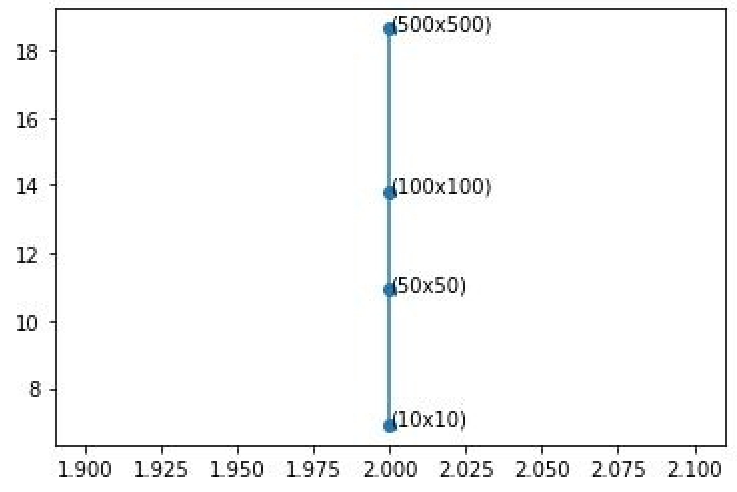
We analyzed the graphs for TurnAround time vs workload size and wait time vs workload size for both processes and our results are as follows: we use system V shared memory segment and memory mapping; as a result the time required to perform the operations was limited to multiplication and simple data fetching of file from fast memory. That explains why the turnaround time was 0 time quantums for the first two workloads and 2 for the other two workloads.



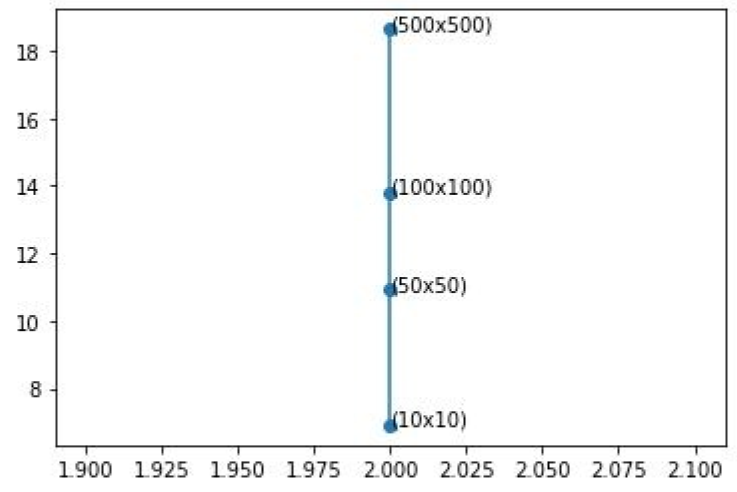
Turnaround time for 10x10 output matrix (Workload vs Turnaround time(in time quantums))



Waiting time for 50x50 output matrix (Workload vs Waiting time(in time quantums))



Turnaround time for 100x100 output matrix (Workload vs Turnaround time(in time quantums))



Waiting time for 500x500 output matrix (Workload vs Waiting time(in time quantums))