

Matrix Multiplication (Multi-Threading)

Operating Systems - Lab 2

Code Organization

Matrices are represented using **matrix** structs. It contains the dimensions of the matrix and a 2D array to store the data. The space for the 2D array is allocated dynamically and always deallocated before exit even if an error occurs.

The definition of this struct along with code used to manipulate it is in the **matrix.h** and **matrix.c** files.

matMult.c contains the initialization and calculation code.

Main Functions:

init_matrix:

Matrices are initialized using the `init_matrix` which reads the dimensions, allocates the appropriate space, and then reads the actual data. If any part of this process fails, the function returns false.

alloc_matrix:

Matrix "**data**" type is (`**int`). `Alloc_matrix` first allocates an array of pointers of size `n`. Where `n` is the number of rows. Then, it allocates an array of size `n*m`. The i^{th} pointer in the array of pointers points at the i^{th} `m` sized chunk of the array representing one row of the matrix.

dealloc_matrix:

First it checks if the `int** data` points at anything to avoid deallocating a null pointer. Then it deallocates the `n*m` sized array of ints, then the `n` sized array of pointers.

one_thread, thread_per_row, thread_per_element”

These functions calculate the matrix multiplication, store the result in the global matrix **c**, and return the total execution time in microseconds.

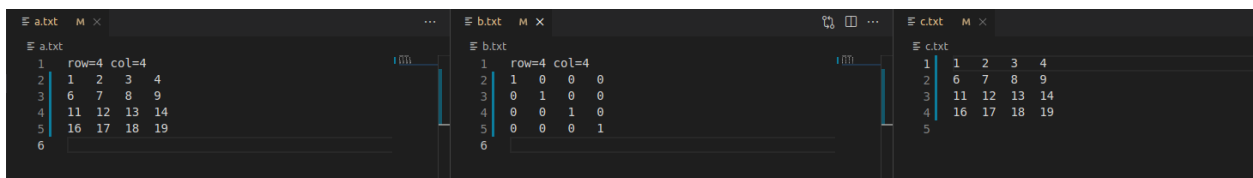
How to Compile and Run

1. make
2. ./matMultp

Or ./matMultp [first filename.txt] [second filename.txt] [output file name]

e.g. : ./matMultp a.txt b.txt c.txt

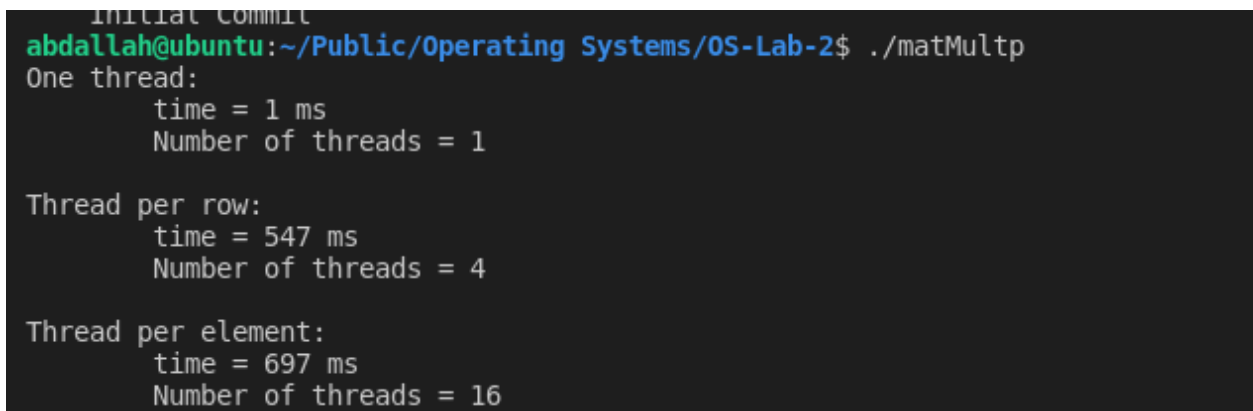
Sample Runs



```
a.txt  M x
1 row=4 col=4
2 1 2 3 4
3 6 7 8 9
4 11 12 13 14
5 16 17 18 19
6

b.txt  M x
1 row=4 col=4
2 1 0 0 0
3 0 1 0 0
4 0 0 1 0
5 0 0 0 1
6

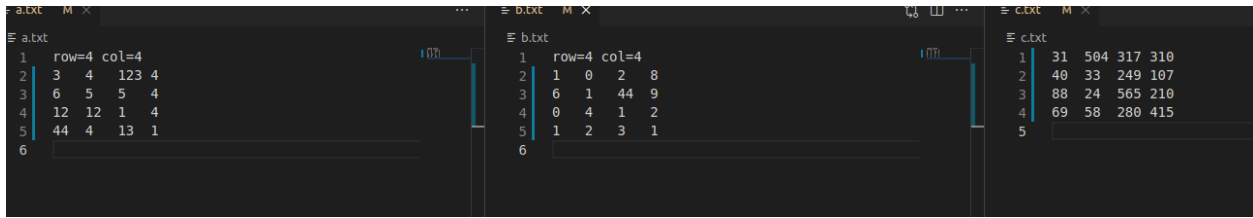
c.txt  M x
1 1 2 3 4
2 6 7 8 9
3 11 12 13 14
4 16 17 18 19
5
```



```
Initial Commit
abdallah@ubuntu:~/Public/Operating Systems/OS-Lab-2$ ./matMultp
One thread:
    time = 1 ms
    Number of threads = 1

Thread per row:
    time = 547 ms
    Number of threads = 4

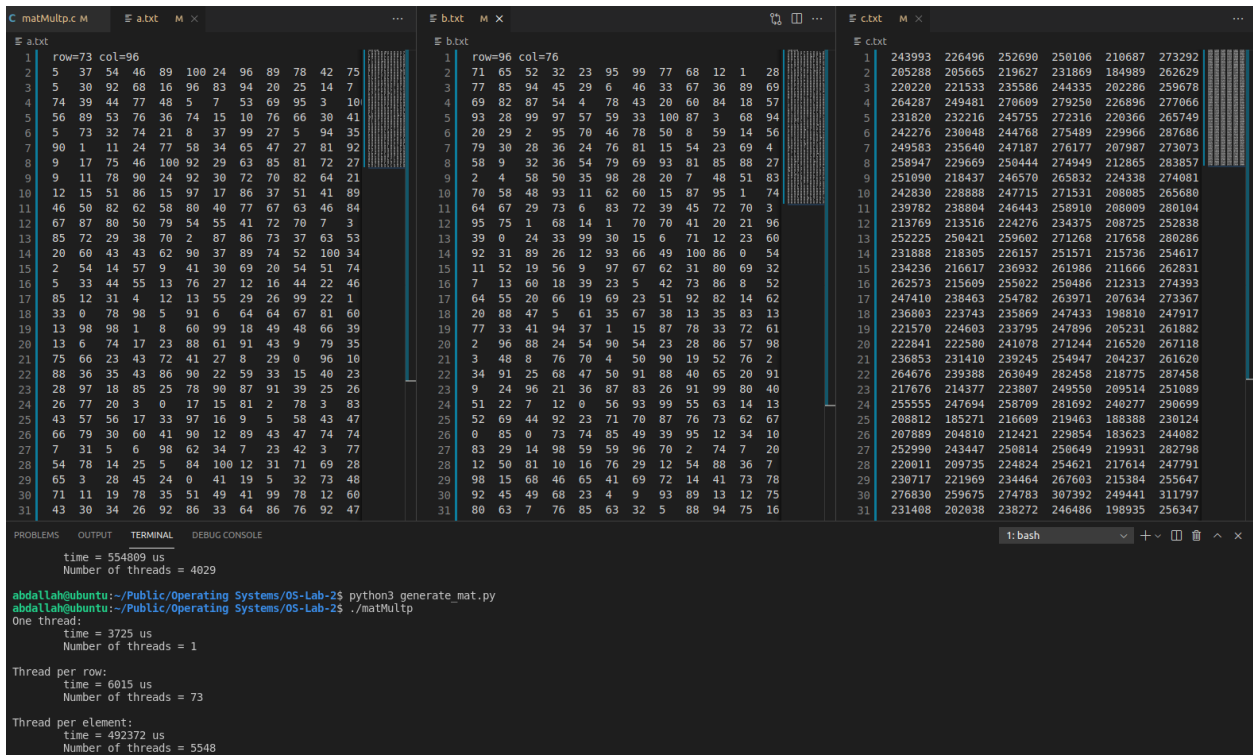
Thread per element:
    time = 697 ms
    Number of threads = 16
```



```
abdallah@ubuntu:~/Public/Operating Systems/OS-Lab-2$ ./matMultp
One thread:
    time = 1 ms
    Number of threads = 1

Thread per row:
    time = 307 ms
    Number of threads = 4

Thread per element:
    time = 3373 ms
    Number of threads = 16
```



Comparing the Different Strategies:

The `one_thread` approach was always the fastest, the `thread_per_row` came in second, `thread_per_element` was much slower.

The reason is because the overhead of switching threads is more expensive than the speed gained by multithreading. Especially since I have only two cores.