Computer & Systems Engineering Department

Operating Systems

Lab 2 : Matrix Multiplication (Multi-Threading)

Name: Mahmoud Attia Mohamed ID: 20011810

I. Code organization:

- First I take the input and output file names to read the matrices while execution by passing (int argc, char** argv) parameters to the main function and if the user didn't enter them they are by default a, b, c.
- I used global variables for the matrices and their dimensions a(x,y), b(y,z), c(x,z), then I read the input files and assign the values of a, b , x, y, and z.
- Then for the first method I created a single thread which use the **"multiply_per_matrix"** function which multiply matrix a by matrix b using single thread and put the result in matrix c1.
- After thread creation I used **thread_join** to wait for this thread to terminate.
- For the second method I created row(x) threads to calculated each row of c2 matrix using a thread and I used the function **"multiply_per_row"** which multiply single row of matrix a by the whole matrix b to get each row of c2 and I passed the row number as a parameter to that function.
- After creation of all x threads (not after creation of each single thread as it will be sequential execution), I used a **thread_join** to wait for these threads to terminate.
- For the third method I used element(x*z) threads to calculate each element of c using a thread and I used the function **"multiply_per_element"** which multiply a single row of matrix a by a single column of matrix b to get each element in matrix c3 and I passed the row and col number to the function

using a struct which contains 2 attributes row, col and it's allocated in the dynamic heap and then freed after the thread is terminated.

- After creation of all (x*z) threads (not after creation of each single thread as it will be sequential execution) I used **thread_join** to wait for these threads to terminate.
- Before the creation of the threads of each method I calculated the start execution time and after the join of the threads method I calculated the end time and printed the execution time (end-start) of each method and also printed the number of used threads.
- After calculating the multiplication of matrix a, b using each method I printed the result to the output file of each method for example: **c_per_matrix.txt, c_per_row.txt, c_per_element.txt**.

II. Main functions:

1) Multiply_per_matrix function:

These function multiply two matrices using a single thread and put the result in c1 (so all elements of c1 are calculated using a single thread).

```
21  //function to multiply 2 matrices using 1 thread
22  void* multiply_per_matrix(){
23      for(int i = 0; i < x; i++){
24          for(int j = 0; j < z; j++){
25              c1[i][j] = 0;
26              for(int k = 0; k < y; k++){
27                  c1[i][j] += a[i][k] * b[k][j];
28              }
29          }
30      }
31      pthread_exit(NULL);
32  }
```

2) Multiply_per_row function:

This function calculates each row of c2 using a thread by multiplying each single row of matrix a by the whole matrix b. So to calculate c2 we need row(x) threads.

```
36
37  //function to multiply 2 matrices using row threads
38  void* multiply_per_row(void* arg){
39      int row = (int) arg;
40      for(int j = 0; j < z; j++){
41          c2[row][j] = 0;
42          for(int k = 0; k < y; k++){
43              c2[row][j] += a[row][k] * b[k][j];
44          }
45      }
46      pthread_exit(NULL);
47  }
48
```

3) Multiply_per_element function:

This function calculates each element of c3 using a thread by multiplying each a single row of matrix a by each column of matrix b. So to calculate c3 we need elements(x*z) threads.

```
49  //function to multiply 2 matrices using row*col threads
50  void* multiply_per_element(void* args){
51      struct thread_data* data = (struct thread_data *)args;
52      int row = data->row;
53      int col = data->col;
54      c3[row][col] = 0;
55      for(int k = 0; k < y; k++){
56          c3[row][col] += a[row][k] * b[k][col];
57      }
58      pthread_exit(NULL);
59  }
```

4) readFile function:

This function read the input files and assign the values of matrix a, matrix b and their dimensions x, y, z.

```c
//function to read file
void readFile(char* fileName, int check){
    FILE *file = fopen(fileName, "r");  // Open the file for reading
    int row, col;
    int arr[MAX_SIZE][MAX_SIZE];
    fscanf(file, "row=%d col=%d\n", &row, &col);
    // Read the integers from the file and store them in the a
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            fscanf(file, "%d", &arr[i][j]);
        }
    }
    if(check == 1){
        //first array
        x = row; y = col; memcpy(a, arr, sizeof(arr));
    }else{
        //second array
        y = row; z = col; memcpy(b, arr, sizeof(arr));
    }
    fclose(file);  // Close the file
}
```

5) Thread_per_matrix_method function:

- This function takes as a parameter the prefix name of the output file to put results in and I created a single thread which use the **“multiply_per_matrix”** function which multiply matrix a by matrix b using single thread and put the result in matrix c1.
- Then I used **pthread_join** to wait for that thread to terminate.
- Before the creation of the thread I calculated the start execution time and after the join of the thread method I calculated the end time and printed the execution time (end-start) of each method and also printed the number of used threads.
- After calculating the multiplication of matrix a, b, I printed the result to the output file for example: **c_per_matrix.txt.**

```c
void thread_per_matrix_method(char *outputFile){
    //use stop and start to calculate time for each method
    struct timeval start, end;
    // multiply_per_matrix
    pthread_t thread_per_matrix;
    gettimeofday(&start, NULL); //start checking time
    //create the single thread
    if (pthread_create(&thread_per_matrix, NULL, multiply_per_matrix, NULL)){
        printf("ERROR in create thread of multiply_per_matrix\n");
        exit(-1);
    }
    //join for the single thread
    if(pthread_join(thread_per_matrix, NULL)){
        printf("ERROR in joining thread of multiply_per_matrix\n");
        exit(-1);
    }
    gettimeofday(&end, NULL); //end checking time
    printf("\033[0;36mMethod: A thread per matrix\033[0m\n");
    printf("*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*\n");
    printf("\033[0;32mNumber of created threads:\033[0m \033[0;36m%d thread\033[0m\n", 1);
    printf("\033[0;32mTime taken in microseconds:\033[0m \033[0;36m%lu us\033[0m\n", end.tv_usec - start.tv_usec);
    printf("------------------------------------------------------------\n");

    //write to per_matrix text file
    char temp[20];
    strcpy(temp, outputFile);
    strcat(temp, "_per_matrix.txt\0");

    FILE *file1 =  fopen(temp,"w");
    fprintf(file1, "Method: A thread per matrix\nrow=%d col=%d\n", x, z);
    for(int i = 0; i < x; i++){
        for(int j = 0; j < z; j++){
            fprintf(file1,"%d ", c1[i][j]);
        }
        fprintf(file1, "\n");
    }
    fclose(file1);
}
```

6) Thread_per_row_method function:

- This function takes as a parameter the prefix name of the output file to put results in and I created row(x) threads to calculated each row of c2 matrix using a thread and I used the function **“multiply_per_row”** which multiply single row of matrix a by the whole matrix b to get each row of c2 and I passed the row number as a parameter to that function.
- After creation of all x threads (not after creation of each single thread as it will be sequential execution), I used a **thread_join** to wait for these threads to terminate.

- Before the creation of the threads I calculated the start execution time and after the join of the threads method I calculated the end time and printed the execution time (end-start) of each method and also printed the number of used threads.
- After calculating the multiplication of matrix a, b, I printed the result to the output file for example: **c_per_row.txt.**

```
119 void thread_per_row_method(char* outputFile){
120     //multiply_per_row
121     pthread_t rowThreads[x];
122     struct timeval start, end;
123     gettimeofday(&start,NULL);
124     //create the row threads
125     for(int i = 0; i < x; i++){
126         if (pthread_create(&rowThreads[i], NULL, multiply_per_row, (void*) i)){
127             printf("ERROR in create thread %d of multiply_per_row\n", i);
128             exit(-1);
129         }
130     }
131     //join for the row threads
132     for(int i = 0; i < x; i++){
133         if(pthread_join(rowThreads[i], NULL)){
134             printf("ERROR in joining thread %d of multiply_per_row\n", i);
135             exit(-1);
136         }
137     }
138     gettimeofday(&end,NULL);
139 
140     printf("\033[0;36mMethod: A thread per row\033[0m\n");
141     printf("*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*\n");
142     printf("\033[0;32mNumber of created threads:\033[0m \033[0;36m%d threads\033[0m\n", x);
143     printf("\033[0;32mTime taken in microseconds:\033[0m \033[0;36m%lu us\033[0m\n", end.tv_usec - start.tv_usec);
144     printf("------------------------------------------------------------------\n");
145 
146     //write to per_row text file
147     char temp[20];
148     strcpy(temp, outputFile);
149     strcat(temp, "_per_row.txt\0");
150 
151     FILE *file1 =  fopen(temp,"w");
152     fprintf(file1, "Method: A thread per row\nrow=%d col=%d\n", x, z);
153     for(int i = 0; i < x; i++){
154         for(int j = 0; j < z; j++){
155             fprintf(file1,"%d ", c2[i][j]);
156         }
157         fprintf(file1, "\n");
158     }
159     fclose(file1);
160 }
```

7) Thread_per_element_method function:

- This function takes as a parameter the prefix name of the output file to put results in and I used element(x*z) threads to calculate each

element of c using a thread and I used the function **“multiply_per_element”** which multiply a single row of matrix a by a single column of matrix b to get each element in matrix c3 and I passed the row and col number to the function using a struct which contains 2 attributes row, col and it’s allocated in the dynamic heap and then freed after the thread is terminated.

- After creation of all (x*z) threads (not after creation of each single thread as it will be sequential execution) I used **thread_join** to wait for these threads to terminate.
- Before the creation of the threads I calculated the start execution time and after the join of the threads method I calculated the end time and printed the execution time (end-start) of each method and also printed the number of used threads.
- After calculating the multiplication of matrix a, b, I printed the result to the output file for example: **c_per_element.txt.**

```
15
16  //struct to carry thread data to be passed as thread create function argument
17  struct thread_data
18  {
19      int row;
20      int col;
21  };
22
23
```

```c
void thread_per_element_method(char* outputFile){
    //multiply_per_element
    pthread_t elementThreads[x][z];
    struct thread_data *tdata[x][z];
    struct timeval start, end;
    gettimeofday(&start, NULL);
    //create the row*col threads
    for(int i = 0; i < x; i++){
        for(int j = 0; j < z; j++){
            tdata[i][j] = malloc(sizeof(struct thread_data));
            tdata[i][j]->row = i;
            tdata[i][j]->col = j;
            if (pthread_create(&elementThreads[i][j], NULL, multiply_per_element, (void*) tdata[i][j])){
                printf("ERROR in create thread of row: %d, col: %d of multiply_per_element\n", i, j); exit(-1);
            }
        }
    }
    //join for the row*col threads
    for(int i = 0; i < x; i++){
        for(int j = 0; j < z; j++){
            if(pthread_join(elementThreads[i][j], NULL)){
                printf("ERROR in joining thread of row: %d, col: %d of multiply_per_element\n", i, j); exit(-1);
            }else{
                //free the the memory allocated in the dynamic heap for the struct
                free(tdata[i][j]);
            }
        }
    }
    gettimeofday(&end, NULL);
    printf("\033[0;36mMethod: A thread per element\033[0m\n");
    printf("*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*\n");
    printf("\033[0;32mNumber of created threads:\033[0m \033[0;36m%d threads\033[0m\n", x*z);
    printf("\033[0;32mTime taken in microseconds:\033[0m \033[0;36m%lu us\033[0m\n", end.tv_usec - start.tv_usec);
    //write to per_element text file
    char temp[20];
    strcpy(temp, outputFile);
    strcat(temp, "_per_element.txt\0");

    FILE *file1 =  fopen(temp,"w");
    fprintf(file1, "Method: A thread per element\nrow=%d col=%d\n", x, z);
    for(int i = 0; i < x; i++){
        for(int j = 0; j < z; j++){
            fprintf(file1,"%d ", c3[i][j]);
        }
        fprintf(file1, "\n");
    }
    fclose(file1);
}
```

8) main function:

- First I take the input and output file names to read the matrices while execution by passing (int argc, char** argv) parameters to the main function and if the user didn't enter them they are by default a, b, c.

Then I read the input files and assign the values of a, b matrices, x, y, and z dimensions.

- Then I called thread_per_matrix_method, thread_per_row_method, thread_per_element_method.

```
211 int main(int argc, char **argv){
212     //get files names
213     char *f1, *f2, *f3;
214     if(argc == 1){
215         f1 = "a"; f2 = "b"; f3 = "c";
216     }else if(argc == 2){
217         f1 = argv[1]; f2 = "b"; f3 = "c";
218     }else if(argc == 3){
219         f1 = argv[1]; f2 = argv[2]; f3 = "c";
220     }else if(argc == 4){
221         f1 = argv[1]; f2 = argv[2]; f3 = argv[3];
222     }else{
223         printf("wrong input format\n");
224         return 1;
225     }
226     //read the first array
227     char temp[20];
228     strcpy(temp, f1);
229     strcat(temp, ".txt\0");
230     readFile(temp, 1);
231 
232     //read the second array
233     strcpy(temp, f2);
234     strcat(temp, ".txt\0");
235     readFile(temp, 2);
236 
237     //multiply per matrix
238     thread_per_matrix_method(f3);
239     //multiply per row
240     thread_per_row_method(f3);
241     //multiply per element
242     thread_per_element_method(f3);
243     return 0;
244 }
```

III. How to compile and run:

- Open the terminal in the folder of the lab then write make to compile the file (as I used makeFile to compile)
- Then write ./matMultp to run and a.txt, b.txt will be the default input files and c.txt will be the default prefix output file.

```
mahmoud@mahmoud-IdeaPad-5-15ITL05: /media/mahmoud/...
mahmoud@mahmoud-IdeaPad-5-15ITL05:/media/mahmoud/New Volume/CSED/level2/2nd seme
ster/operating systems/labs/lab2/OS-Lab2$ make
make: 'matMultp' is up to date.
mahmoud@mahmoud-IdeaPad-5-15ITL05:/media/mahmoud/New Volume/CSED/level2/2nd seme
ster/operating systems/labs/lab2/OS-Lab2$ ./matMultp
```

- Or specify input and output files for example: ./matMultp x y z. So x.txt, y.txt will be our input files and z.txt will be the prefix of our output file.

```
mahmoud@mahmoud-IdeaPad-5-15ITL05:/media/mahmoud/New Volume/CSED/level2/2nd seme
ster/operating systems/labs/lab2/OS-Lab2$ ./matMultp x y z
```

IV. Sample runs:

- **Sample run 1:**

➢**Input files:**

Open | a.txt — New Volume /media/mahmoud/New Volume/CSED/level...nd s

```
1 row=10 col=5
2 1       2       3       4       5
3 6       7       8       9       10
4 11      12      13      14      15
5 16      17      18      19      20
6 21      22      23      24      25
7 26      27      28      29      30
8 31      32      33      34      35
9 36      37      38      39      40
10 41     42      43      44      45
11 46     47      48      49      50
```

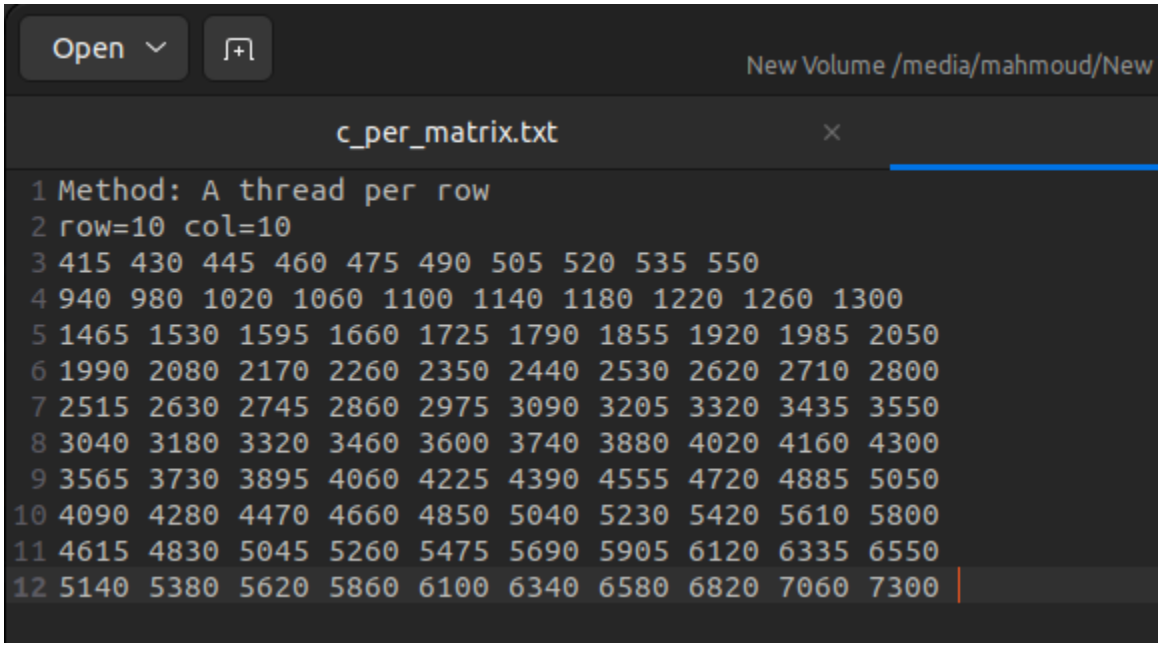
Open | b.txt — New Volume /media/mahmoud/New Volume/CSED/level...nd semeste

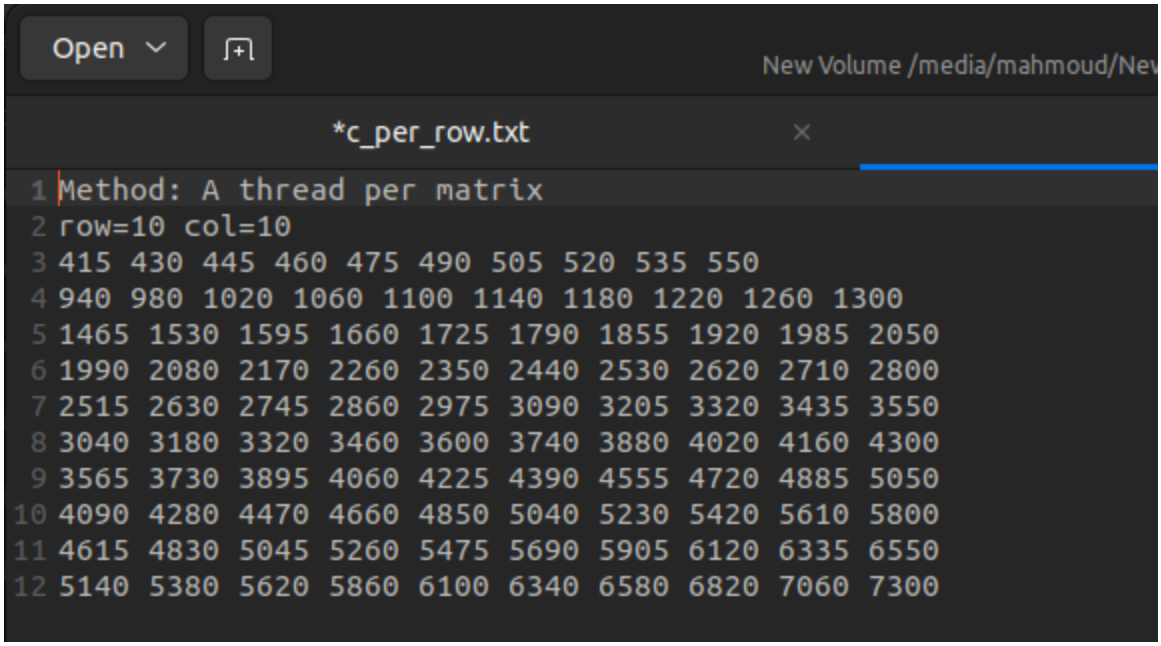
```
1 row=5 col=10
2 1       2       3       4       5       6       7       8       9       10
3 11      12      13      14      15      16      17      18      19      20
4 21      22      23      24      25      26      27      28      29      30
5 31      32      33      34      35      36      37      38      39      40
6 41      42      43      44      45      46      47      48      49      50
7
8
```

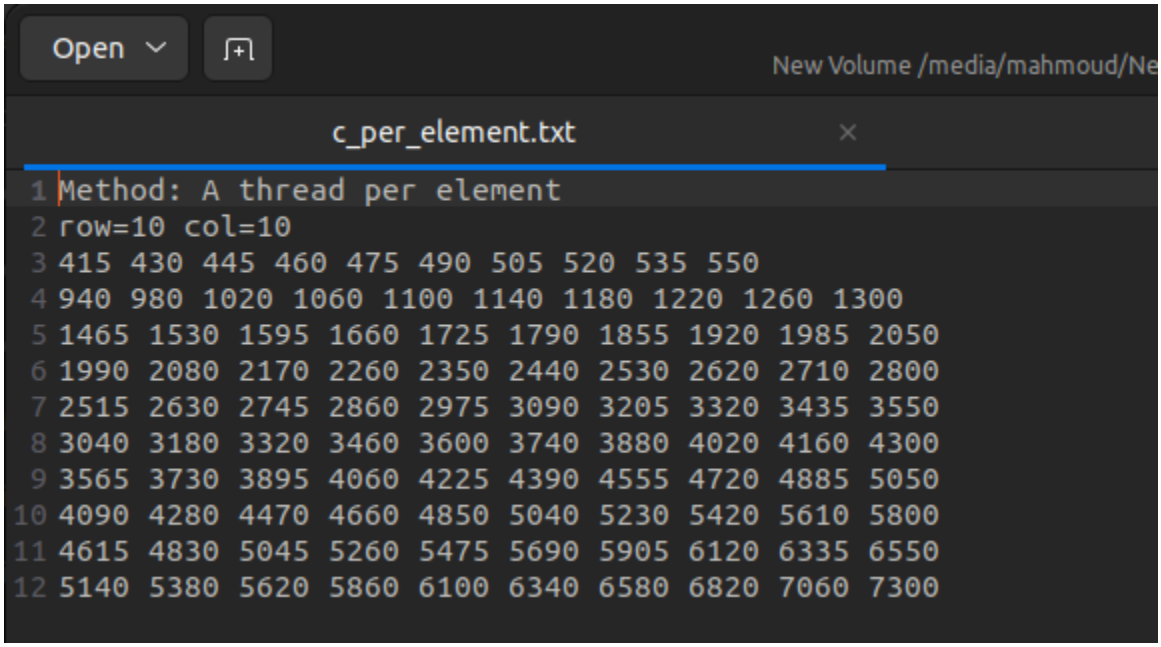
➢**Run code:**

```
mahmoud@mahmoud-IdeaPad-5-15ITL05:/media/mahmoud/New Volume/CSED/level2/2nd seme
ster/operating systems/labs/lab2/OS-Lab2$ ./matMultp
Method: A thread per matrix
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 1 thread
Time taken in microseconds: 358 us
---------------------------------------------------------------
Method: A thread per row
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 10 threads
Time taken in microseconds: 514 us
---------------------------------------------------------------
Method: A thread per element
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 100 threads
Time taken in microseconds: 3939 us
```

➢Output files







- **Sample run 2:**

➢**Input files**

Open

x.txt

New Volume /media/mahmoud/New Volume/CSED/level...nd semeste

x.txt

```
1 row=3 col=5
2 1       -2      3       4       5
3 1       2       -3      4       5
4 -1      2       3       4       5
```

Open

y.txt

New Volume /media/mahmoud/New Volume/CSED/level...nd semeste

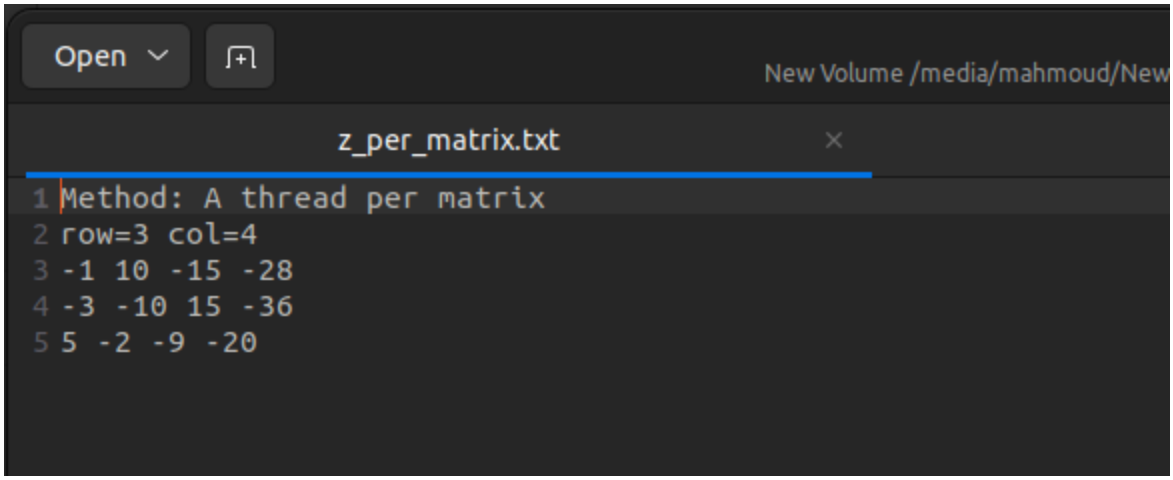
y.txt

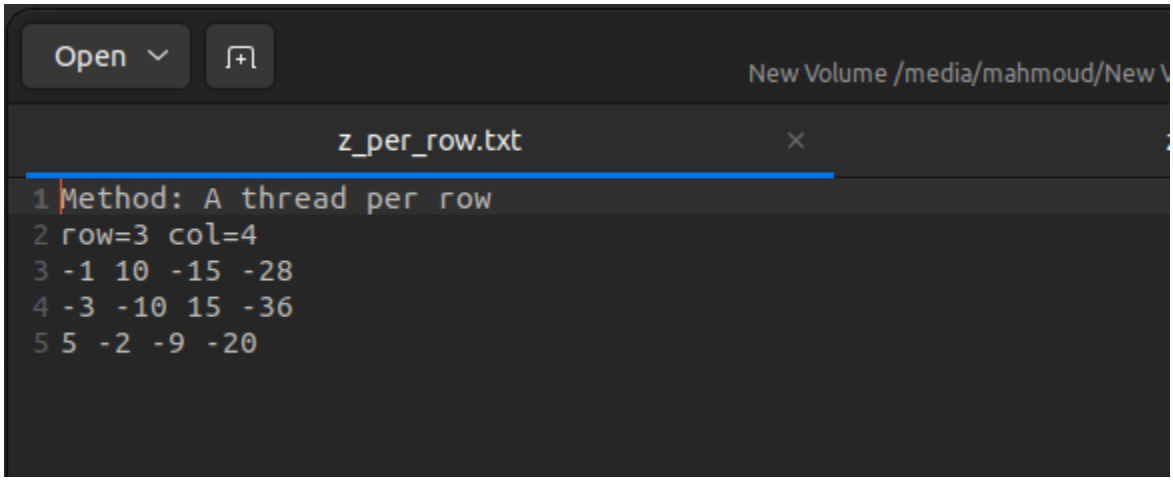
```
1 row=5 col=4
2 -1      2       3       4
3 1       -2      3       4
4 1       2       -3      4
5 1       2       3       -4
6 -1      -2      -3      -4
```

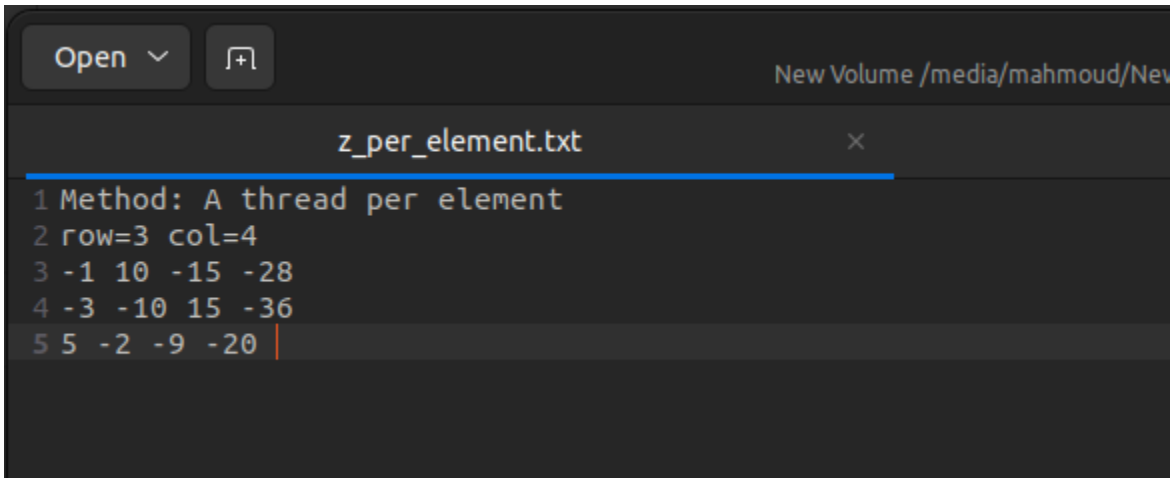
➢Run code:

```
mahmoud@mahmoud-IdeaPad-5-15ITL05:/media/mahmoud/New Volume/CSED/level2/2nd seme
ster/operating systems/labs/lab2/OS-Lab2$ ./matMultp x y z
Method: A thread per matrix
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 1 thread
Time taken in microseconds: 339 us
------------------------------------------------------------
Method: A thread per row
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 3 threads
Time taken in microseconds: 208 us
------------------------------------------------------------
Method: A thread per element
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 12 threads
Time taken in microseconds: 485 us
```

➢Output files:







- **Sample run 3:**

➢**<u>input files:</u>**

Open

a.txt

New Volume /media/mahmoud/New Volume/CSED/level...nd semeste

a.txt

```
1 row=5 col=5
2 1       2       3       4       5
3 6       7       8       9       10
4 11      12      13      14      15
5 16      17      18      19      20
6 21      22      23      24      25
```

Open

b.txt

New Volume /media/mahmoud/New Volume/CSED/level...nd semeste

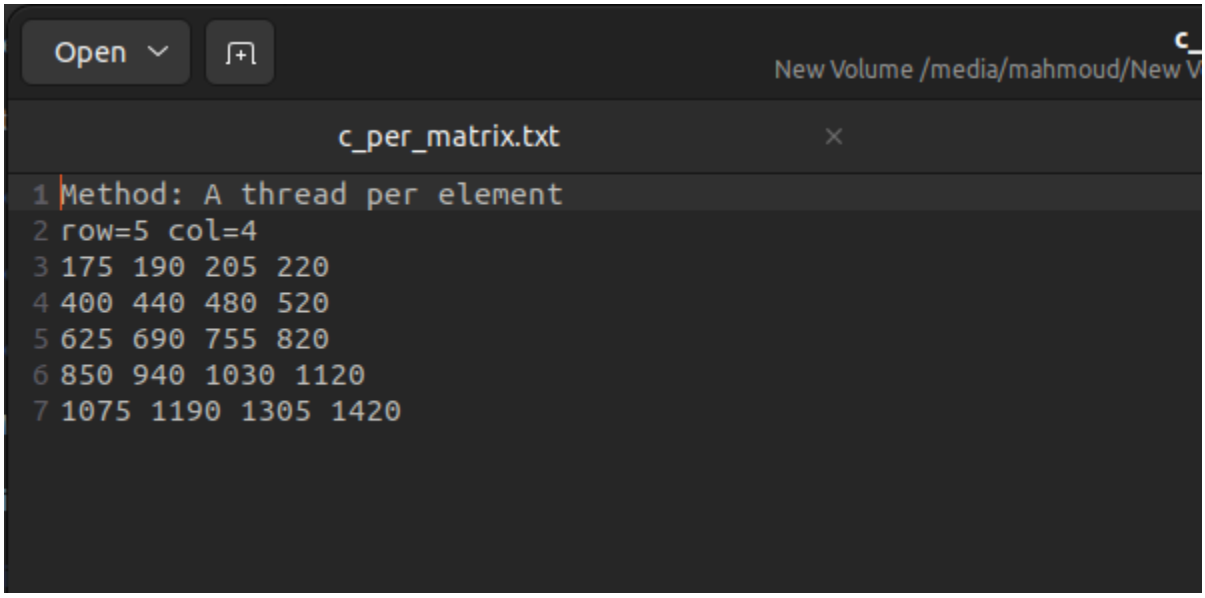
b.txt

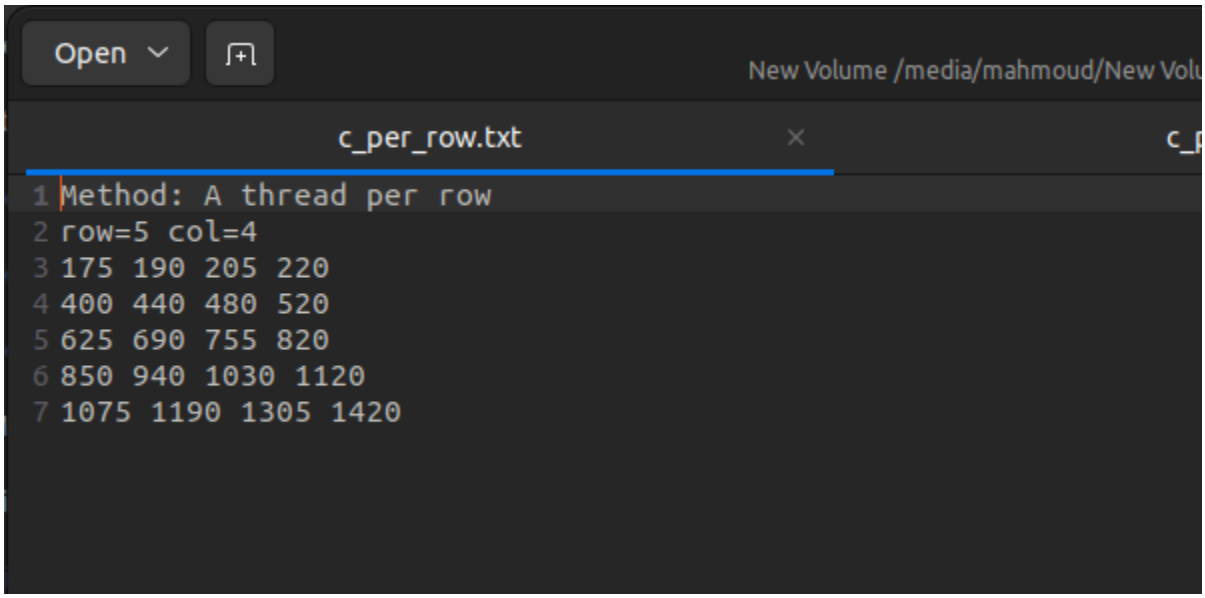
```
1 row=5 col=4
2 1       2       3       4
3 5       6       7       8
4 9       10      11      12
5 13      14      15      16
6 17      18      19      20
```

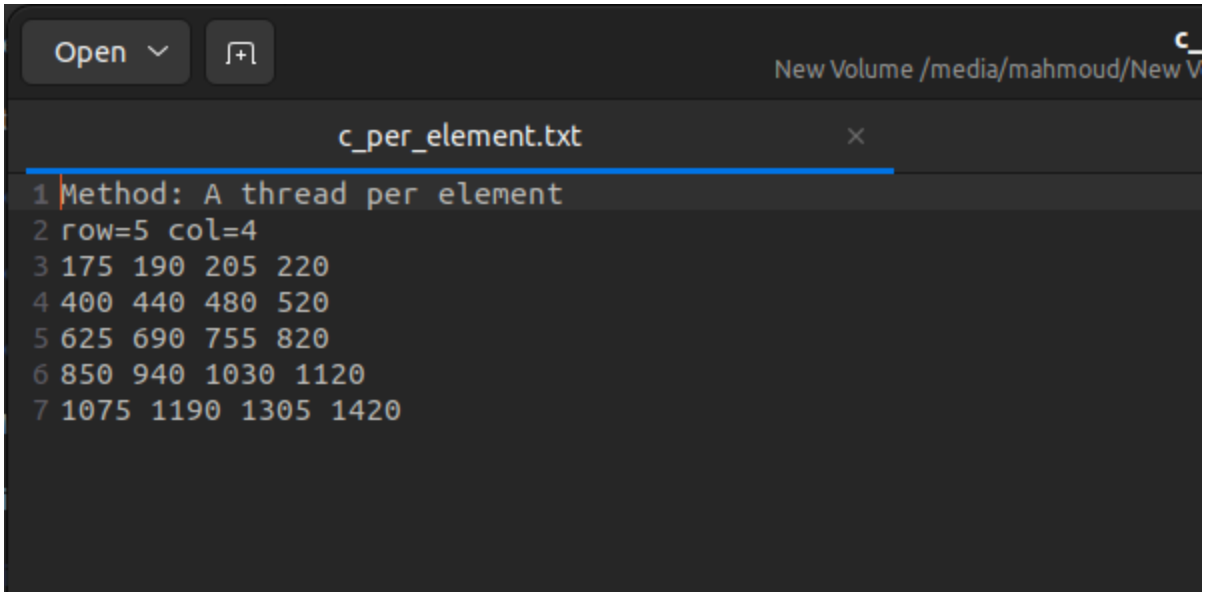
➢Run code:

```
mahmoud@mahmoud-IdeaPad-5-15ITL05:/media/mahmoud/New Volume/CSED/level2/2nd seme
ster/operating systems/labs/lab2/OS-Lab2$ ./matMultp
Method: A thread per matrix
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 1 thread
Time taken in microseconds: 342 us
------------------------------------------------------------
Method: A thread per row
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 5 threads
Time taken in microseconds: 296 us
------------------------------------------------------------
Method: A thread per element
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
Number of created threads: 20 threads
Time taken in microseconds: 760 us
mahmoud@mahmoud-IdeaPad-5-15ITL05:/media/mahmoud/New Volume/CSED/level2/2nd seme
```

➢Output files:







V. Comparison between the 3 methods:

- **General comparison:**

In general, the first method (thread per matrix) has less overhead than the second (thread per row) and third method(thread per element) as creating and managing threads requires more overhead and computations so in this problem (matrix multiplication) we will find that the first method takes less time than the second and third method.

- **Comparison considering test cases:**

➢ In the first test case I have shown in the sample runs section we will find the first method is faster than the second(10 threads) and the second is faster than the third (100 threads) and that is because of the additional overhead of creating and handling threads.

➢ In the second test case I have shown in the sample runs section we will find that the second method (3 threads) is faster than the first and the first is faster than the third (12 threads) and I think the second is faster as it includes a low number of threads so the overhead is low.

➢ In the third test case I have shown in the sample runs section we will find that the second method (5 threads) is faster than the first

and the first is faster than the third (20 threads) and I think the second is faster as it includes a low number of threads so the overhead is low.

VI. Video link:

video-link-click here
