CSE x61 : Operating System

Multi-Threading Matrix Multiplication

Name	Abdelrahman Ibrahim Gaber
ID	19015881
Department	Electronics and communication

1 Code Organization

The code is mainly using Multi-Threading technique using pthread.h library this library works in user level so it's not actually using a core for each thread but the library handle that and the threads works in parallel in such away that the library handle.
Here, Matrix Multiplication is a good example to show the concept of Multi-Threading since this program calculate the multiplication of 2 matrices that read it from 2 files in 3 ways each way is done in parallel with the others.
Method 1 : one Thread for whole matrix.
Method 2 : Thread per row.
Method 3 : Thread per each element.
Each matrix is setted as global pointer to pointer and be allocated in heap using dynamic memory allocation concepts by malloc function,
Main function gets files names that the user want to multiply the matrices including this files and start to read the input from the files and set it in its global pointer after allocated then main call the function that execute that 3 methods that we talked about every method has its routine function that passed to the thread function and start to execute the program in parallel according to number of threads used in every method. finally, every memory location we allocate it in the heap we should free it to prevent memory leakage.

2 Main functions

2.1 Read From File

```
   void readFromFile(char *path, int fileNumber)
{
    FILE *f = fopen(path, READ_FROM_FILE);
    if(fileNumber == FIRST_FILE){
      fscanf(f , "row=%d col=%d",&rowsA , &colsA);
      allocateMatrix(f, fileNumber);
    }
    else{
      fscanf(f , "row=%d col=%d",&rowsB , &colsB);
      allocateMatrix(f, fileNumber);
```

```
    }
    fclose(f);
}
```

2.1.2 Allocate Matrix

```
    void allocateMatrix(FILE* fileName , int fileNumber){
    if(fileNumber == FIRST_FILE){
        matA = (long **)malloc(rowsA * sizeof(long **));
        for(int i = 0 ; i < rowsA ; i++){
            matA[i] = (long *)malloc(colsA * sizeof(long *));
            for(int j = 0 ; j < colsA ; j++){
                fscanf(fileName , "%ld" , &matA[i][j]);
            }
            fgetc(fileName);
        }
    }
    else{
        matB = (long **)malloc(rowsB * sizeof(long **));
        for(int i = 0 ; i < rowsB ; i++){
            matB[i] = (long *)malloc(colsB * sizeof(long *));
            for(int j = 0 ; j < colsB ; j++){
                fscanf(fileName , "%ld" , &matB[i][j]);
            }
            fgetc(fileName);
        }
    }
}
```

2.2 Heap Allocator

```
void heapAllocator(void){
    matC_perEle = (long **)malloc(sizeof(long *) * rowsA);
    matC_perRow = (long **)malloc(sizeof(long*) * rowsA);
    matC_whole = (long **)malloc(sizeof(long *) * rowsA);
    for(int i = 0 ; i < rowsA ; i++){
```

```
        matC_perEle[i] = (long *)malloc(sizeof(long) * colsB);
        matC_perRow[i] = (long *)malloc(sizeof(long) * colsB);
        matC_whole[i] = (long *)malloc(sizeof(long) * colsB);
    }

}

}
```

2.3.1 Run case 1

```
void runCase1(void){
    struct timeval stop , start;
    gettimeofday(&start , NULL);
    pthread_t threadPerWholeMatrix ;
    /*one thread for the whole matrix no arguments*/
    if(pthread_create(&threadPerWholeMatrix , NULL , &mutrixMul , NULL)!= 0)
        perror("Error creating thread\n");
        exit(EXIT_FAILURE);
    }
    pthread_join(threadPerWholeMatrix , NULL);
    gettimeofday(&stop , NULL);
    printf("Thread per matrix taken in Micro Second %lu\n",stop.tv_usec - st
    printf("Threads Created = 1\n");
}
}
```

2.3.2 Run case 2

```
void runCase2(void){
    struct timeval stop , start ;
    gettimeofday(&start , NULL);
    pthread_t threadPerRow[rowsA];
    for(int i = 0 ; i < rowsA ; i++){
        int* arg = malloc(sizeof(int)*20);
        *arg = i;
        if(pthread_create(&threadPerRow[i] , NULL , &mutrixMulPerRow , arg)
```

```
            perror("Error creating thread\n");
            exit(EXIT_FAILURE);
        }
    }
    /*wait for all threads being created*/
    for(int i = 0 ; i < rowsA ; i++){
        pthread_join(threadPerRow[i] , NULL);
    }
    gettimeofday(&stop , NULL);
    printf("Thread per Row taken in Micro Second %lu\n",stop.tv_usec - start
    printf("Threads Created = %d\n",rowsA);
}

}
```

2.3.3 Run case 3

```
void runCase3(void){
    struct timeval stop , start;
    gettimeofday(&start , NULL);
    pthread_t threadPerElement[rowsA * colsB];
    int threadIndex = 0 ;
    for(int i = 0 ; i < rowsA ; i++){
        for(int j = 0 ; j < colsB ; j++){
            matData *args = malloc(sizeof(matData));
            args->cuurentRow = i;
            args->currentColoumn = j;
            if(pthread_create(&threadPerElement[threadIndex++] , NULL ,
             &mutrixMulPerElement , args)){
                perror("Error creating thread\n");
                exit(EXIT_FAILURE);
             }
        }
    }
    for(int i = 0 ; i < rowsA *colsB; i++){
        pthread_join(threadPerElement[i], NULL);
```

```
    }
    gettimeofday(&stop , NULL);
    printf("Thread per Element taken in Micro Second %lu\n",stop.tv_usec - s
    printf("Threads Created = %d\n",(rowsA * colsB));
}
}
```

2.4 Output Handler

```
void outputHandler(int argc , char*argv[]){
    if(argc == 1){
        writeInFile("c_per_matrix.txt",METHOD_1);
        writeInFile("c_per_row.txt",METHOD_2);
        writeInFile("c_per_element.txt",METHOD_3);
    }
    else{
        char*matrix_1 = malloc(sizeof(char) * 20);
        char*matrix_2 = malloc(sizeof(char) * 20);
        char*matrix_3 = malloc(sizeof(char) * 20);
        strcpy(matrix_1 , argv[3]);
        strcat(matrix_1 , "_per_matrix.txt");
        writeInFile(matrix_1 , METHOD_1);
        free(matrix_1);
        strcpy(matrix_2 , argv[3]);
        strcat(matrix_2 , "_per_row.txt");
        writeInFile(matrix_2 , METHOD_2);
        free(matrix_2);
        strcpy(matrix_3 , argv[3]);
        strcat(matrix_3 , "_per_element.txt");
        writeInFile(matrix_3 , METHOD_3);
        free(matrix_3);
    }
}
}
```

2.5 Lite Garbage Collector

```
void liteGarbageCollector(void){
    /*free pointers that take place in heap*/
    for(int i = 0 ; i < rowsA ; i++){
        free(matA[i]);
        free(matC_whole[i]);
        free(matC_perRow[i]);
        free(matC_perEle[i]);
    }
    for(int i = 0 ; i < rowsB ; i++) free(matB[i]);
    free(matA);
    free(matB);
    free(matC_whole);
    free(matC_perRow);
    free(matC_perEle);
}
}
```

3 How to compile this code

3.1 This program simply can be run from the terminal using this command gcc fileName.c -o fileName -lpthread

3.2 -lpthread is required to enable the operating system to run in Multi-Threading way.

3.3 then there is 3 cases you can run

3.3.1 case 1 : ./fileName : run using the default files a.txt , b.txt and write the ouput in c.txt for each method.

3.3.2 case 2 : ./fileName a b c : a and b are such arguments that are also considered the default and the program handle that case and write the ouput in c.txt.

3.3.3 case 3 : ./fileName x y z : custom inputs and the program consider x and y the 2 inputs files that multiply the 2 matrices on it and write the result in the last argument z.txt

3.4 Now the user can read the output from the certain file that he choose

4 Sample Run

4.1 Test case 1 (No Arguments)

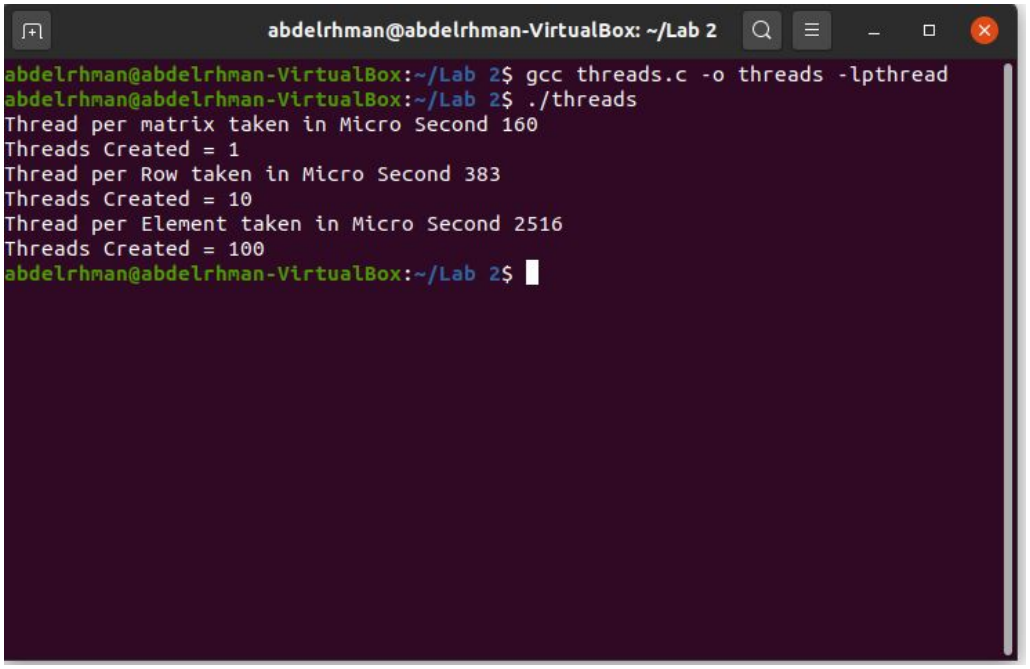

Figure 1: compile test 1 with no argument

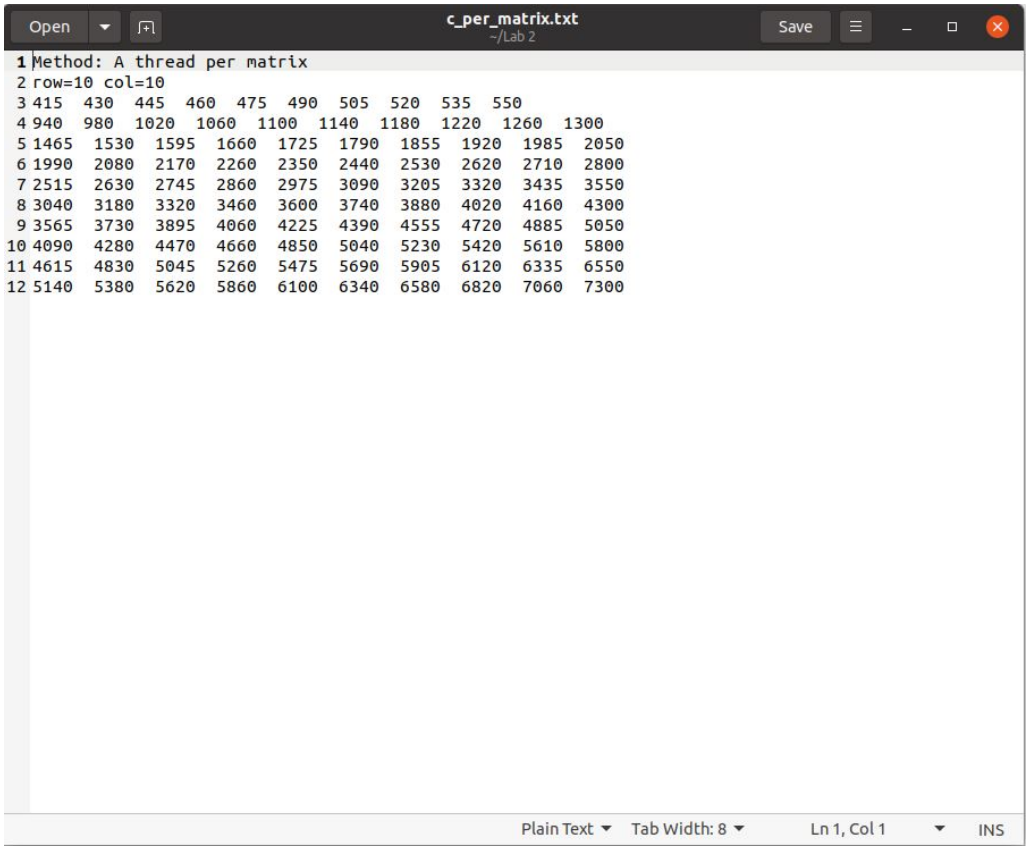

Figure 2: one thread per matrix

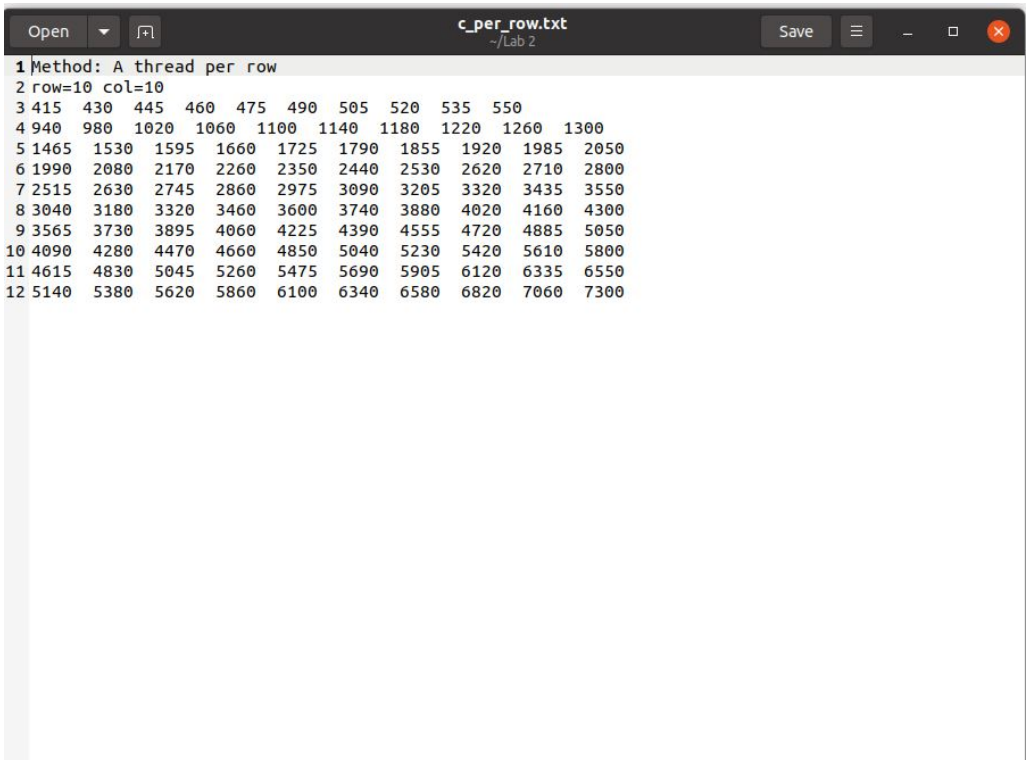

Figure 3: one thread per row

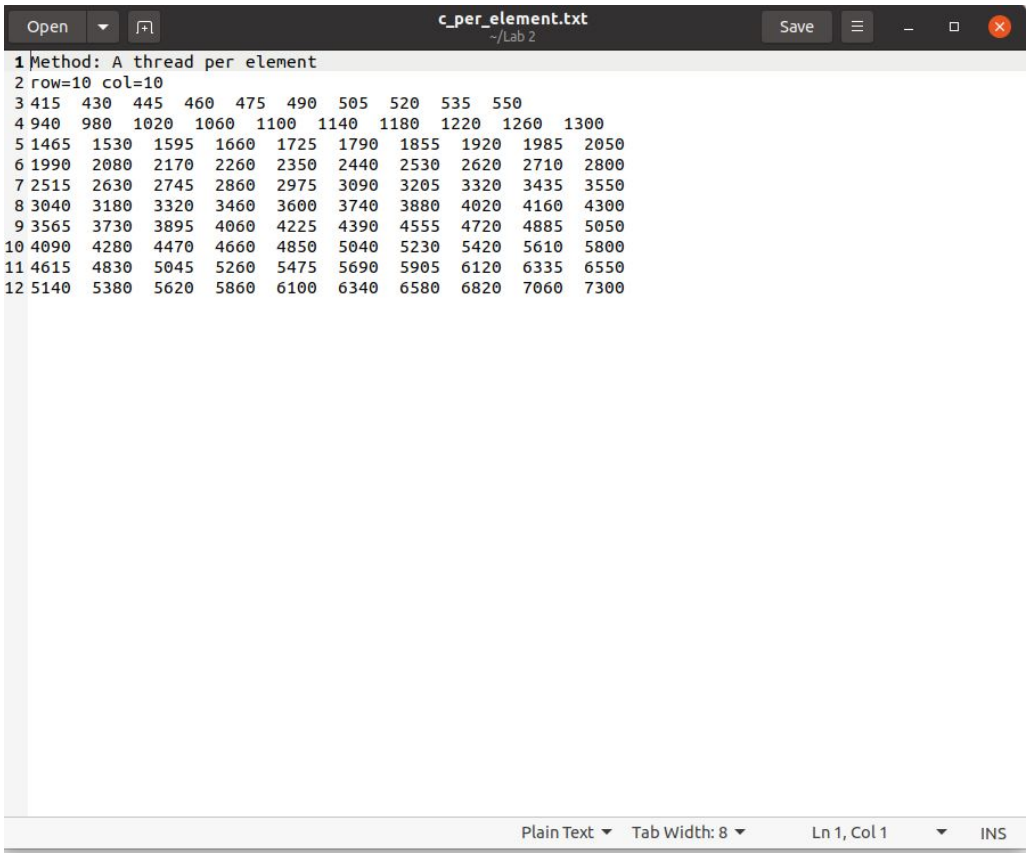

Figure 4: one thread per element

4.2 test case 2 (arguments)

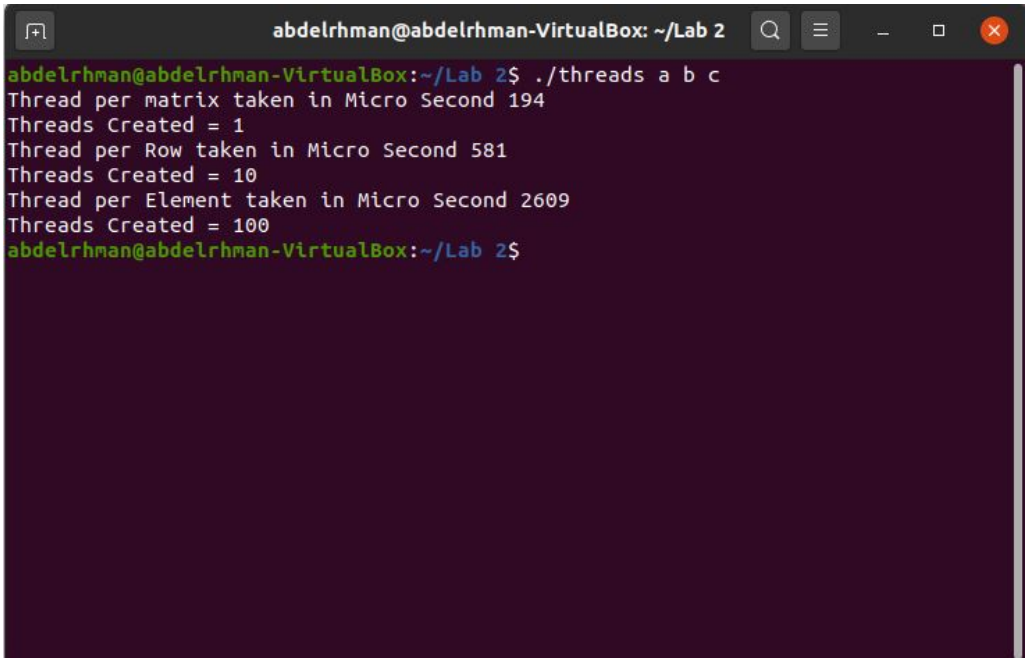

Figure 5: compile test 2 with no argument

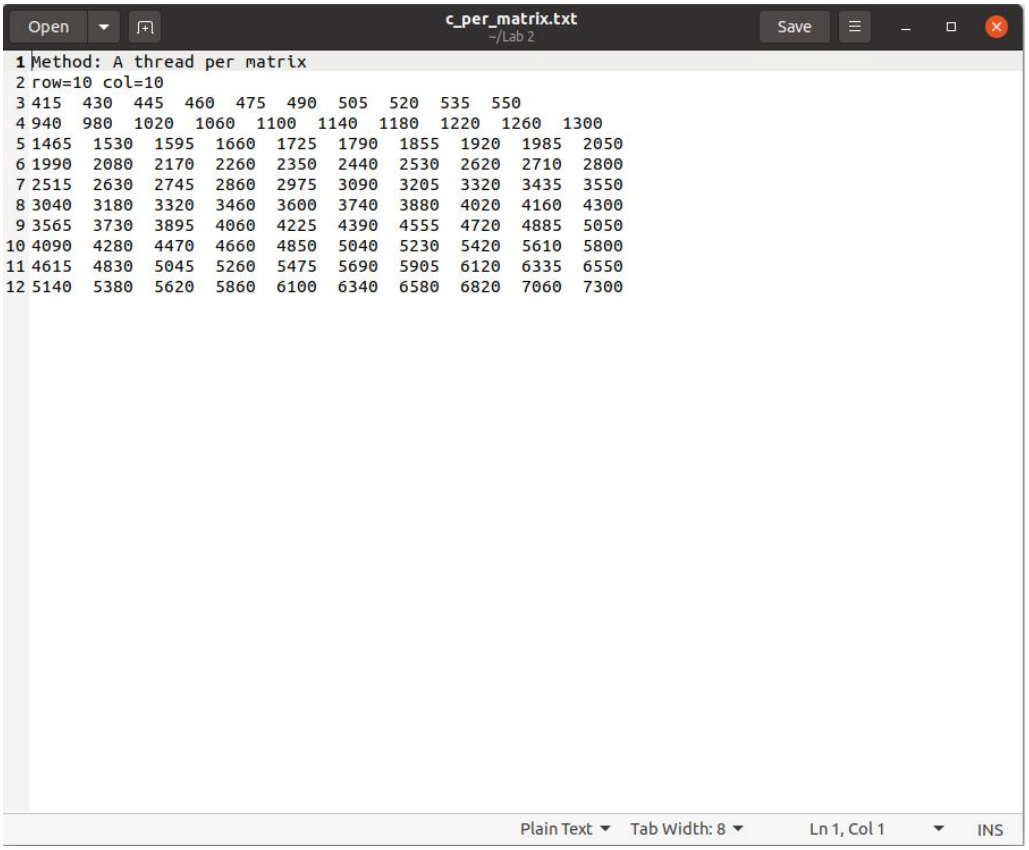

Figure 6: one thread per matrix

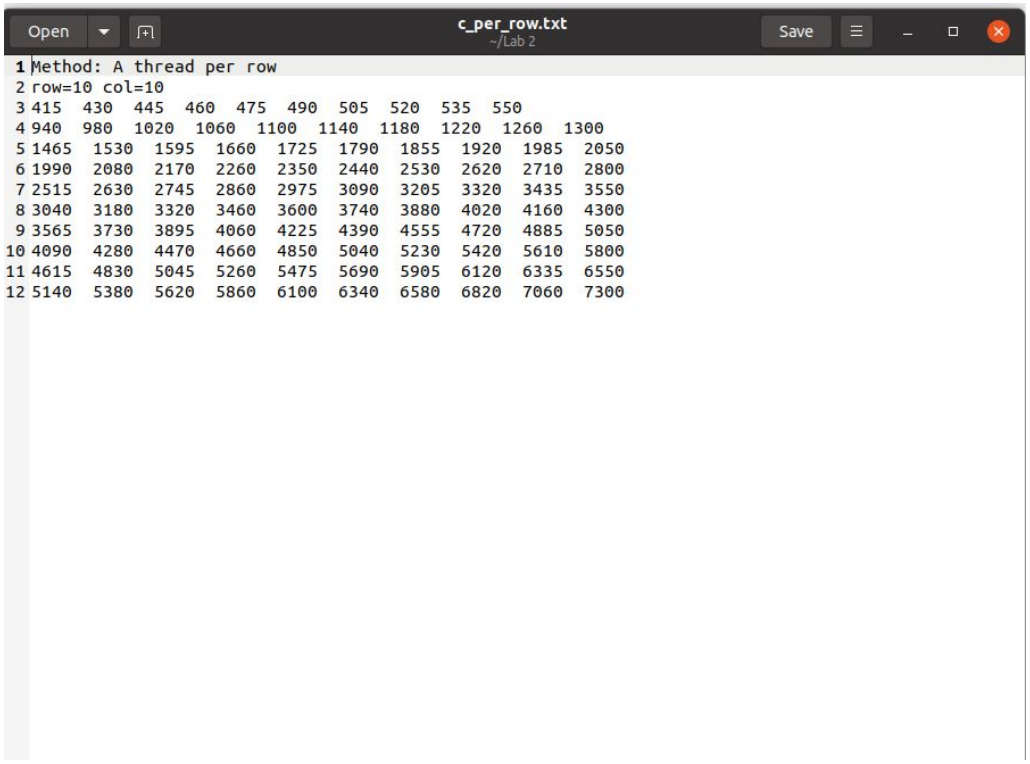

Figure 7: one thread per row

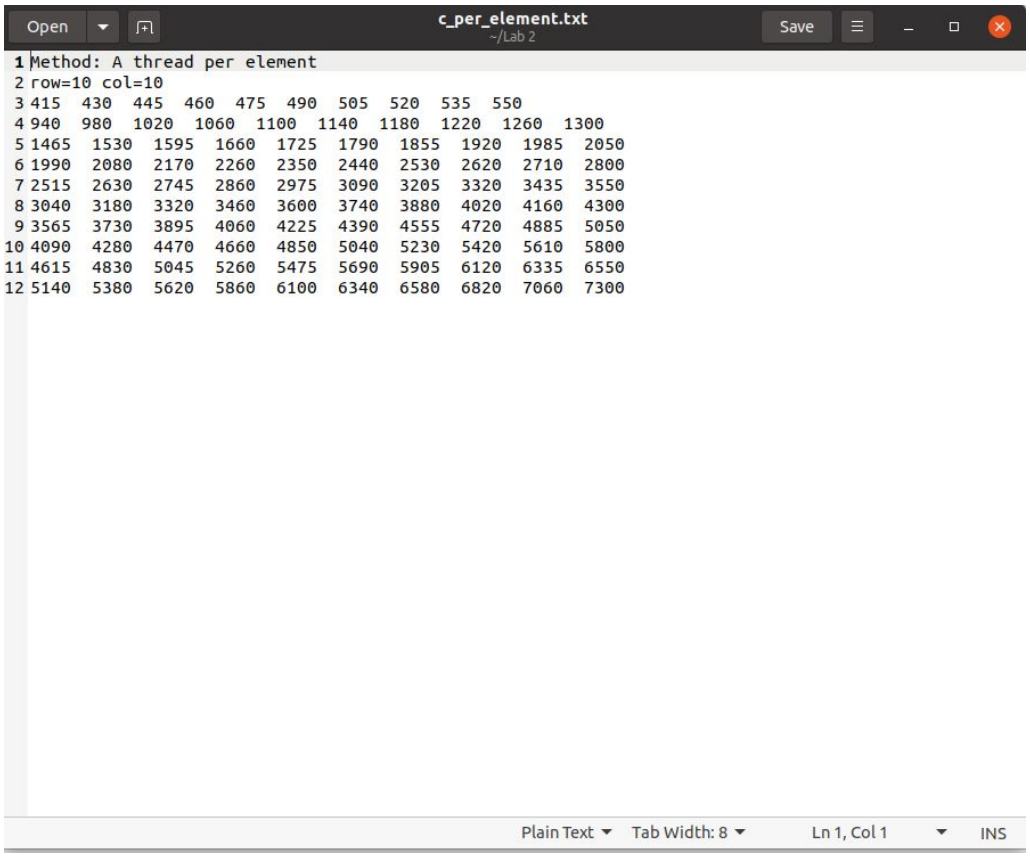

Figure 8: one thread per element

4.3.1 test case 3 (Custom inputs)

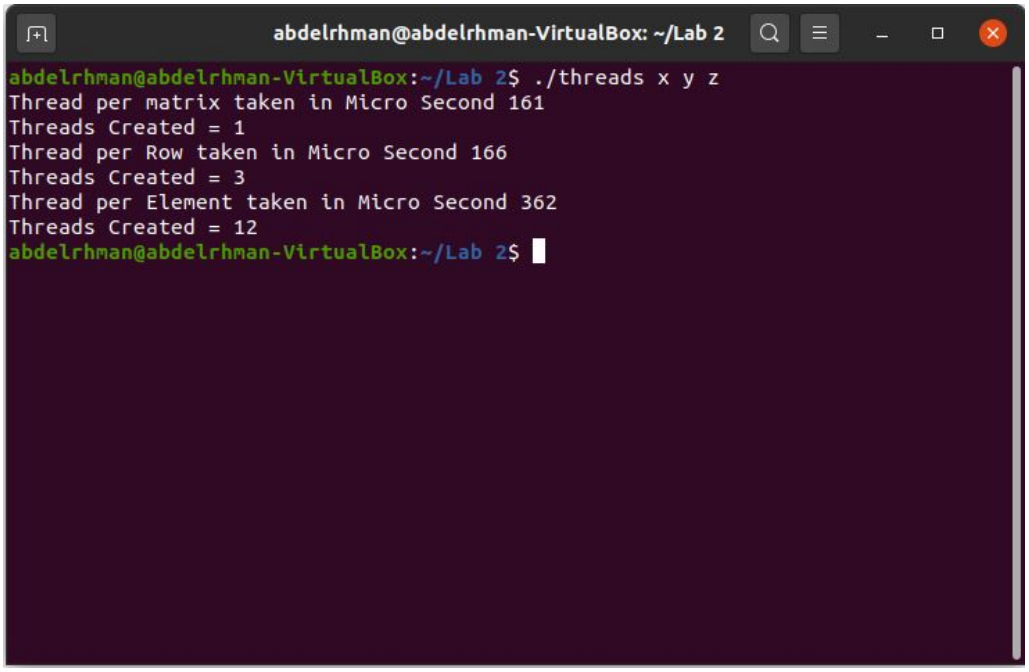

Figure 9: compile test 3 with no argument

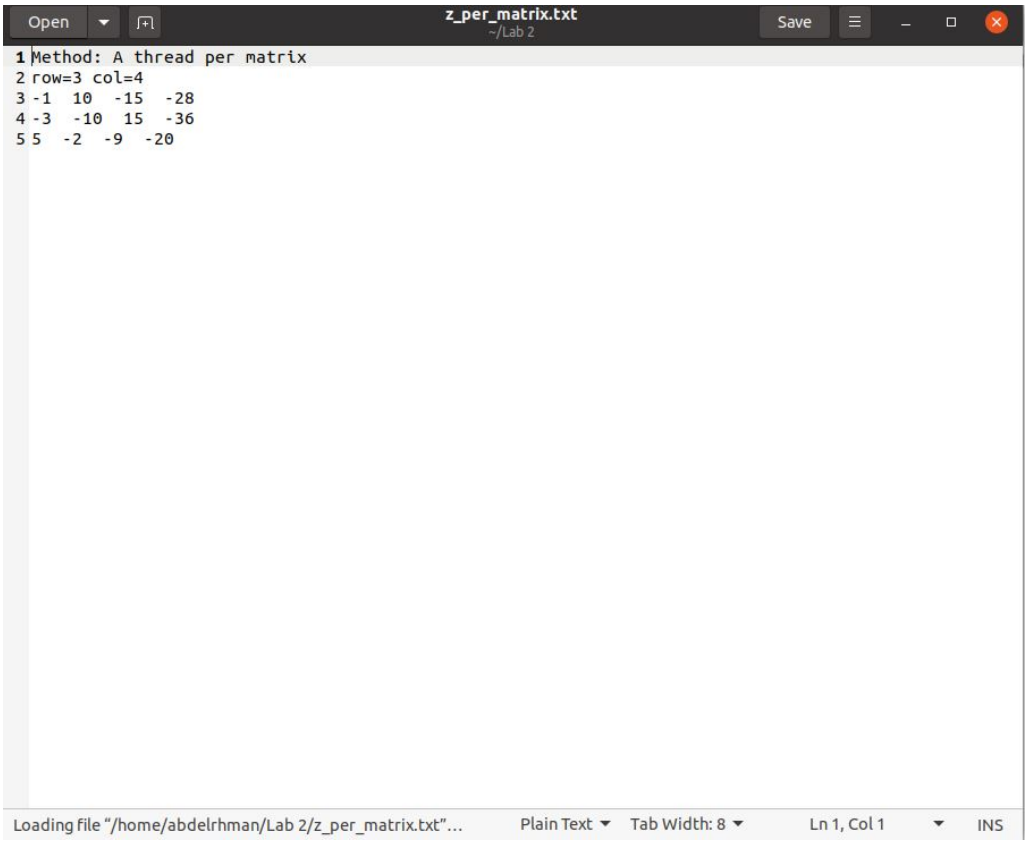

Figure 10: one thread per matrix

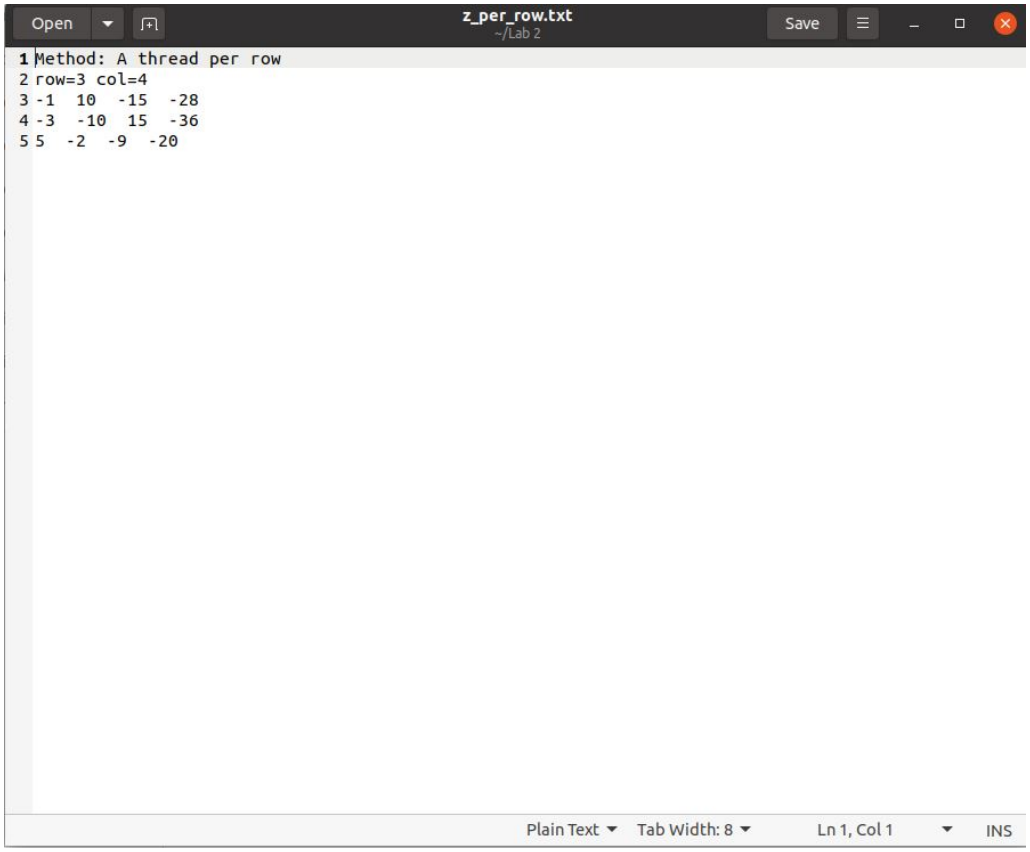

Figure 11: one thread per row

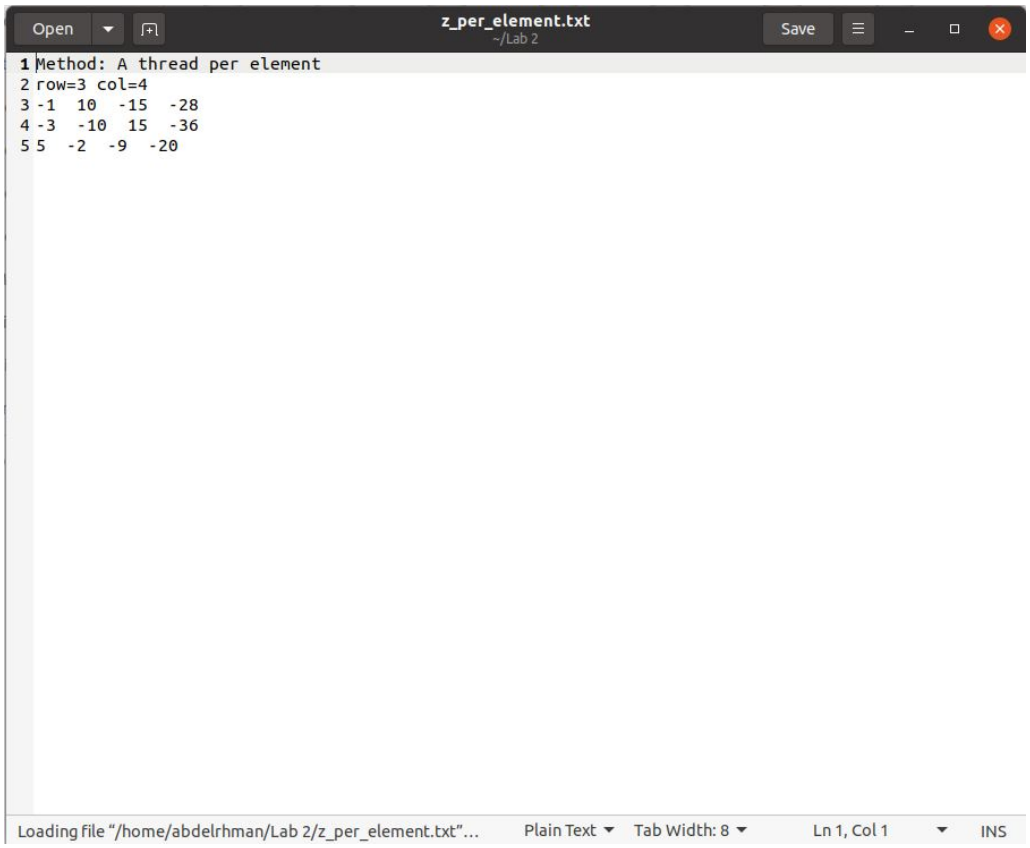

Figure 12: one thread per element

4.3.2 another custom inputs

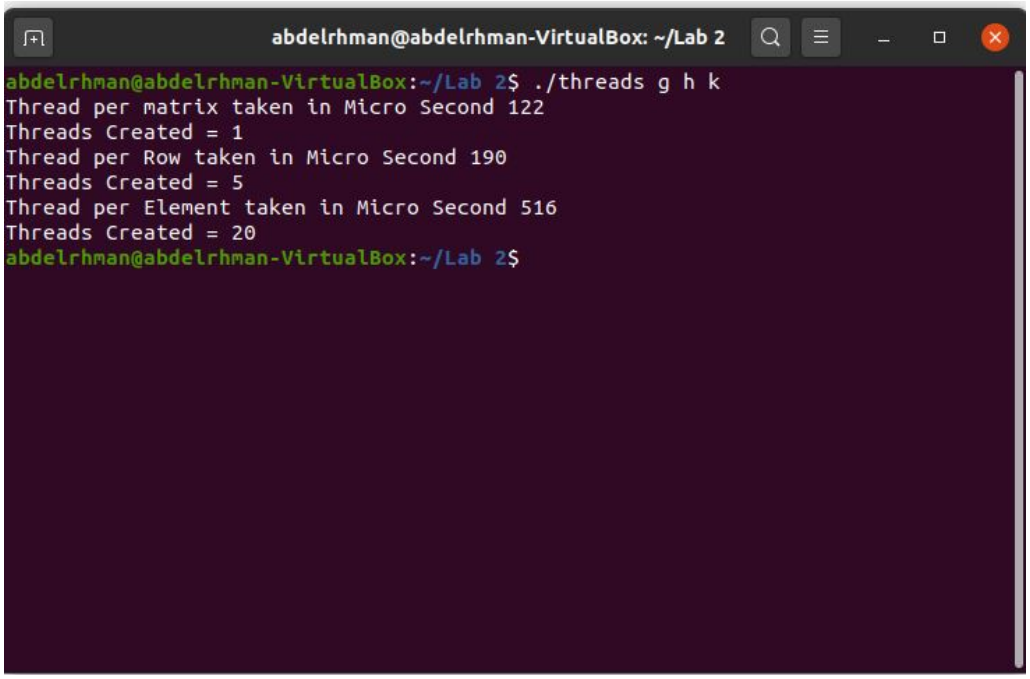

Figure 13: compile test 3 with no argument

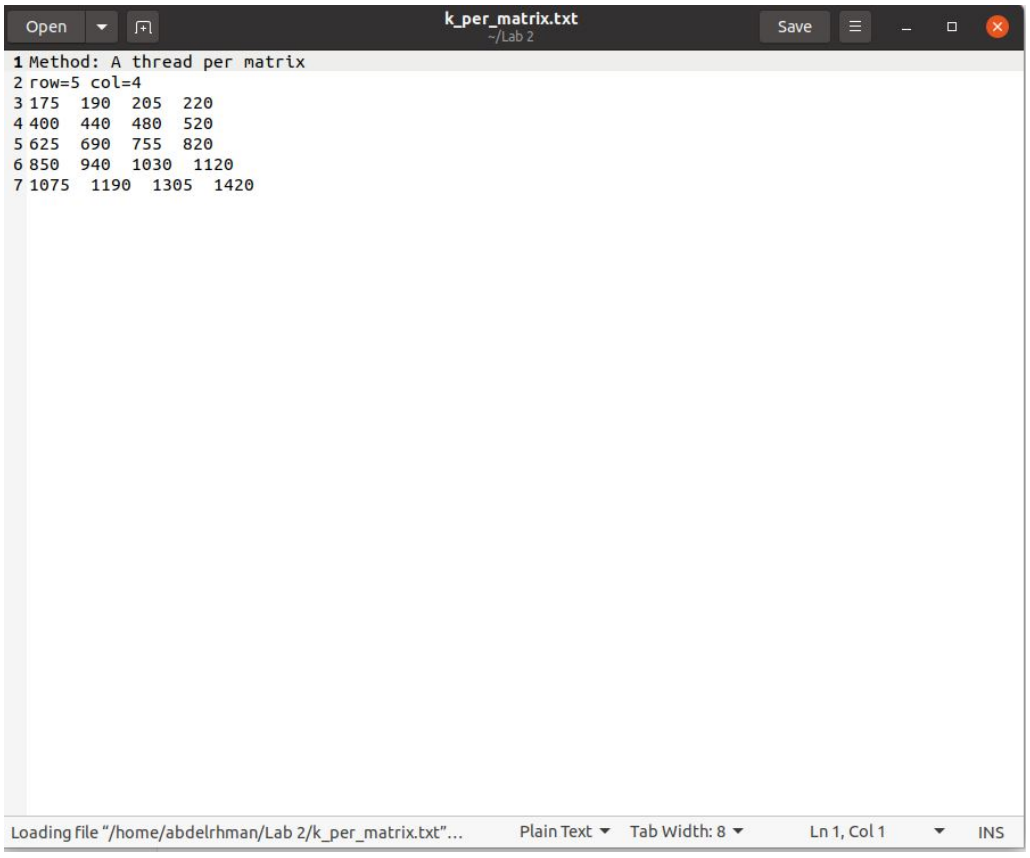

Figure 14: one thread per matrix

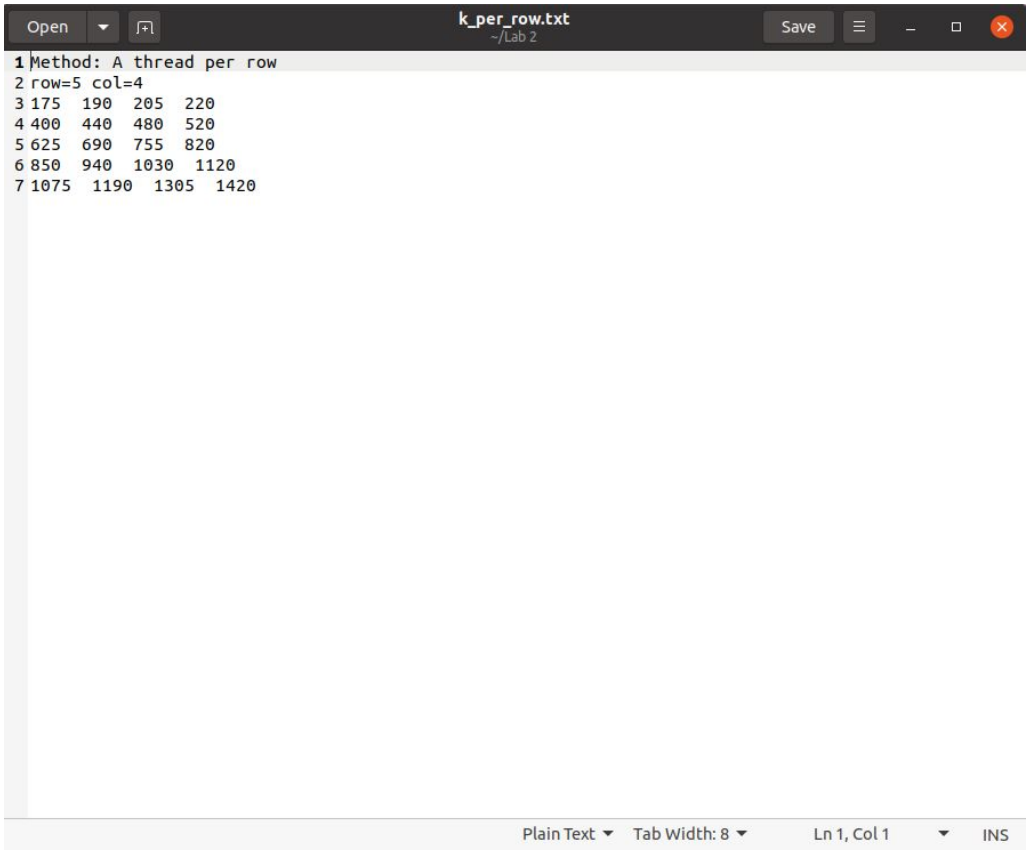

Figure 15: one thread per row

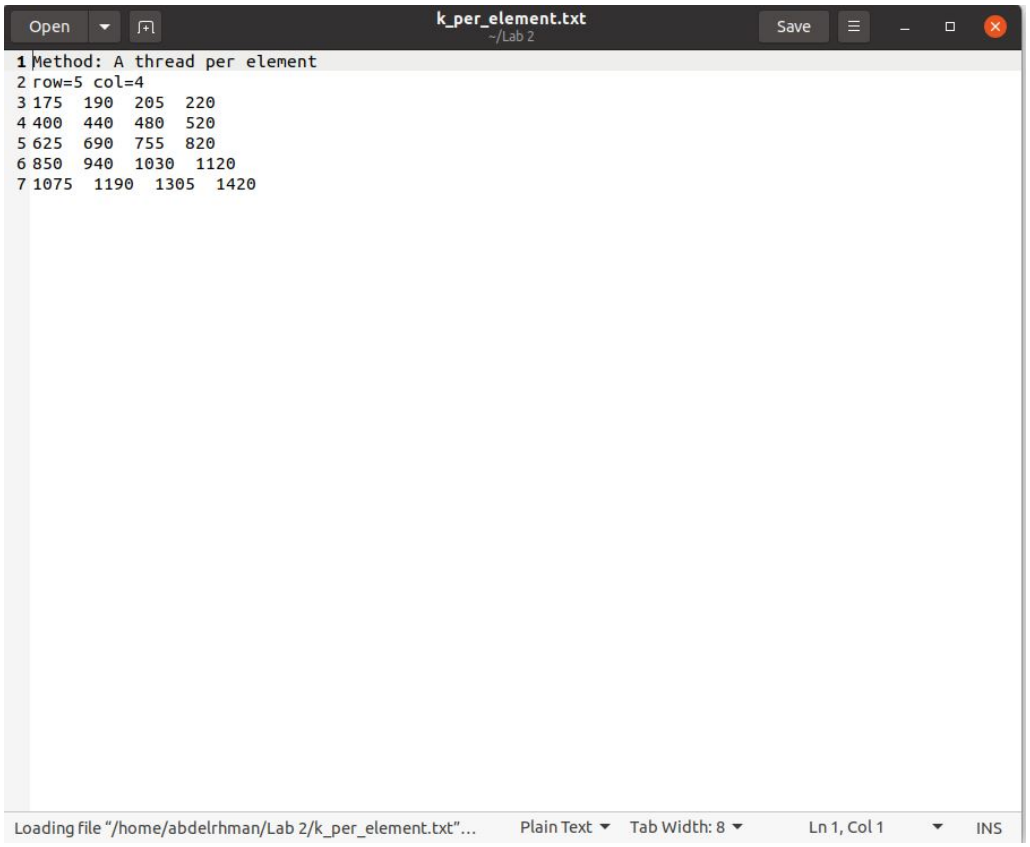

Figure 16: one thread per element

It's noticeable that the result in each method to multiply is same because each of them do the same functionality but in different number of threads

5 Comparison between three methods

5.1 time of execution in test 1

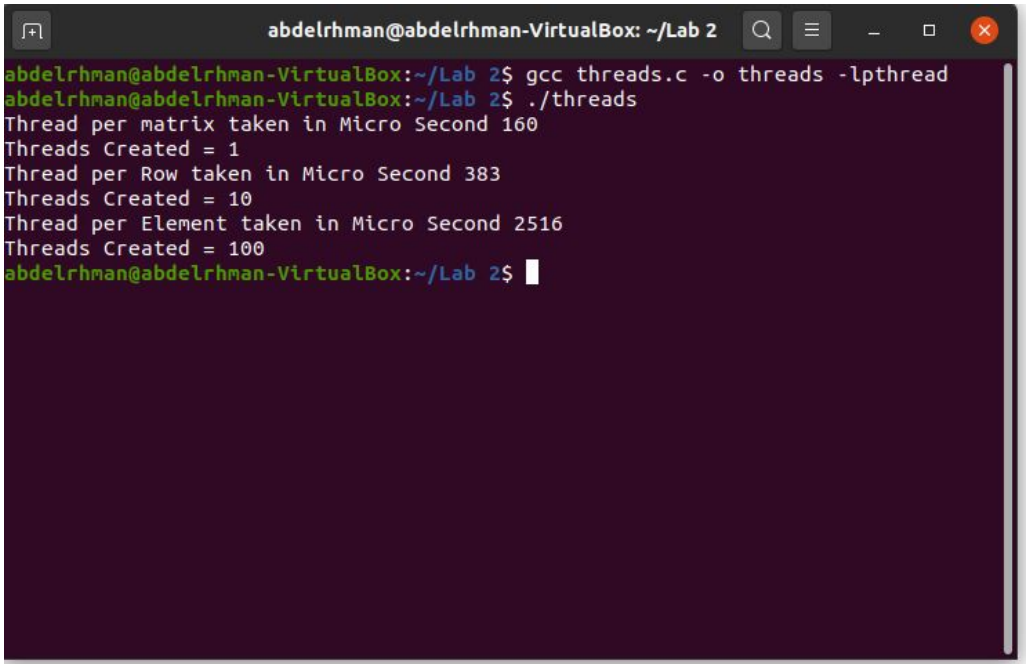

Figure 17: Time of execution

5.2 time of execution in test 2

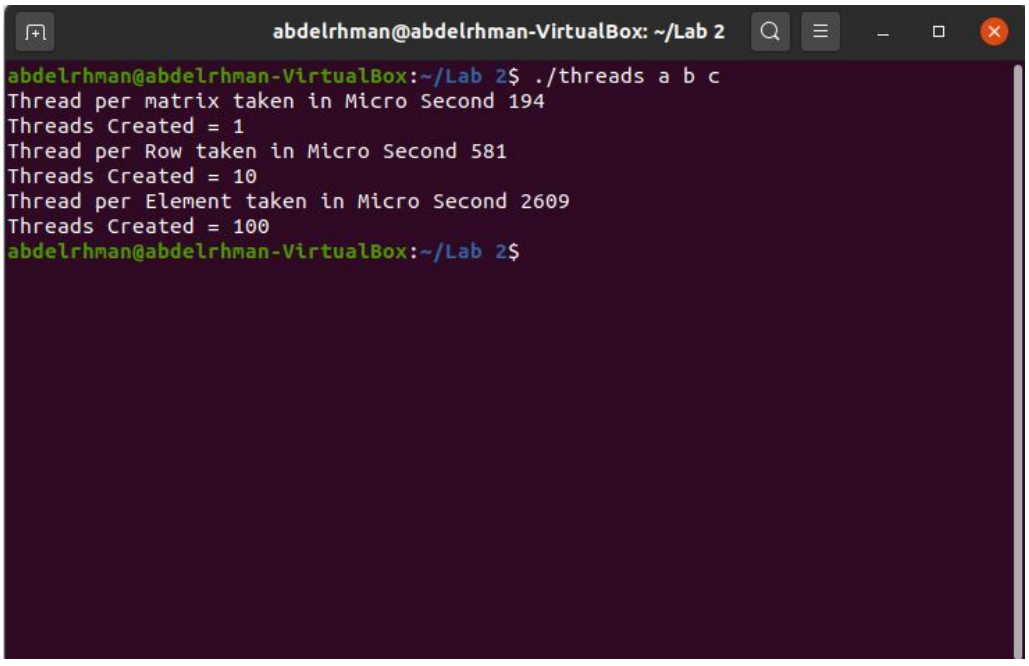

Figure 18: Time of execution

5.3 time of execution in test 3

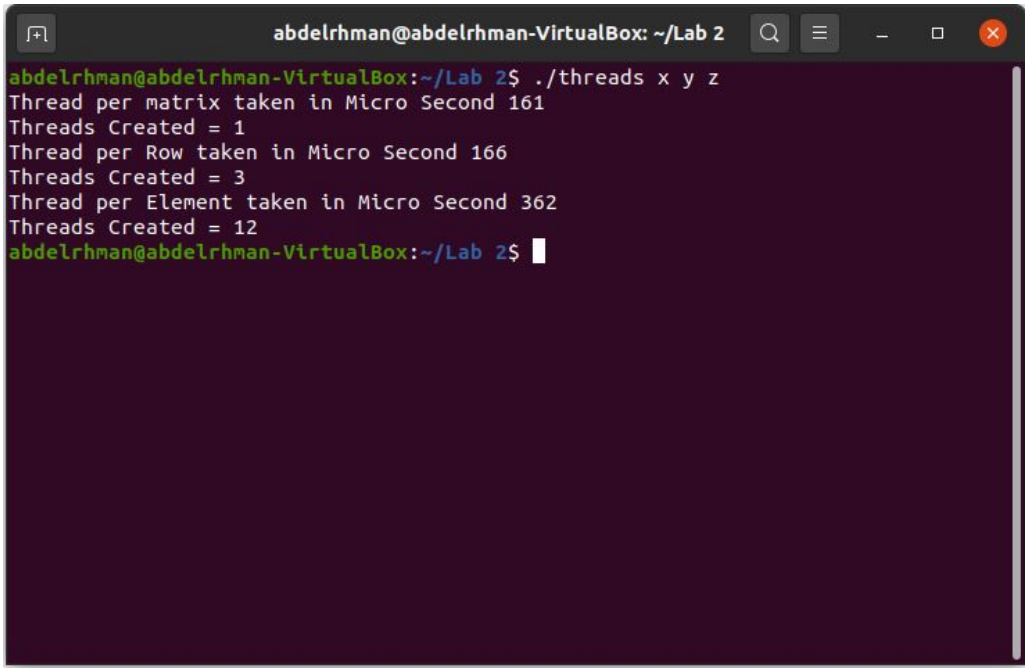

Figure 19: Time of execution

In this figures we can notice that the methods are different in time execution.

- Method 1 : the fastest and uses only 1 thread
- Method 2 : faster from method 3 and slower than method 1 since uses thread per row
- Method 3 : the slowest method since uses thread per element

The number of threads mainly have the upper hand in calculate the execution time since the switching between threads takes time to switch between thread context (pc,register,etc) so according to that :-

- Method 1 : only have one thread so it is the fastest
- Method 2 : have threads bigger than method 1 and bigger than method 2 so its speed in between
- Method 3 : have the biggest number of threads , a thread per element so it's clearly that it's the slowest method