Multithreaded Matrix - Lab2

Mariam Ahmed Fathy 19016627

March 19, 2023

1 Introduction

This program offers various methods for handling matrices. First, read the dimensions of the matrices from a text file, read the actual matrices from a text file, solve the matrices using a single thread, solve the matrices by rows (with one thread per row), and solve the matrices by element (with one thread per element). Additionally, there are two different methods for creating threads: one for solving the matrices by rows, and another for solving the matrices by elements.

2 Code Main functions

1. readrowcol(FILE *file ,int *row,int *col)

   is used to extract the row and column dimensions from a given file.

2. void readmat(int **arr,FILE *file,int r,int col)
   is utilized to copy the contents of a file into a two-dimensional array. The parameters of this function include the array to be copied into, the file to be read, and the row and column dimensions of the matrix.

3. void SolveMat(int **mata,int **matb,int **matc)
   is responsible for solving a matrix using a single thread.

4. void *SolveRow(void *obj)
   uses a thread for each row of the matrix to solve it.

5. void *SolveElement(void *obj)
   utilizes a thread for each element of the matrix to solve it.

6. void RowThread(int **mata,int **matb,int **matc)
   the function creates threads to solve the matrix by dividing it into rows.

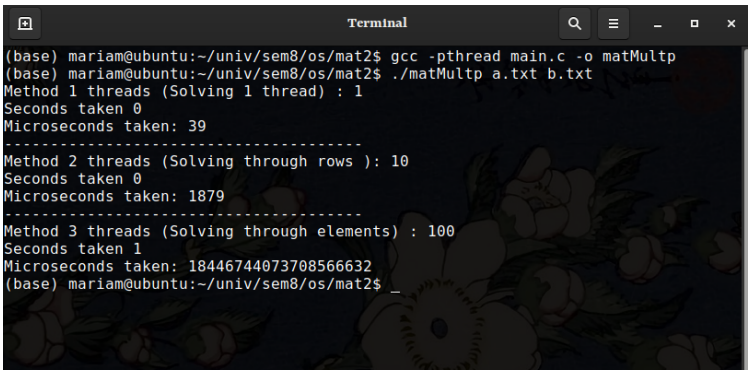
7. void ThreadsElements(int **mata,int **matb,int **matc)
   the function creates threads to solve the matrix by dividing it into individual elements.

8. void outputres(int **matc, char *outfilepath)
   creates the output matrix

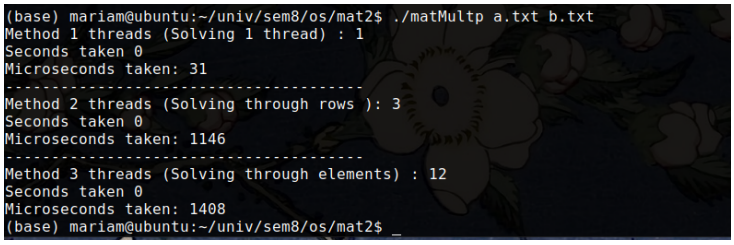
9. int main(int argc, char *argv[])

3 Running Examples

3.1 Test 1



3.2 Test 2



4 A comparison between the three methods

The amount of time it takes to execute a task remains constant, but when a method generates multiple threads, it takes longer due to the need for synchronization. Each thread waits for the others to finish, causing a delay. The first method of using a thread per matrix is likely performing better than the second and third methods because it reduces the overhead associated with creating and managing multiple threads. In the first method, a single thread is responsible for processing an entire matrix, so there are fewer context switches between threads, which reduces overhead.

In contrast, the second method of using a thread per row and the third method of using a thread per element both require significantly more threads to be created and managed, which increases the overhead. In the second method, there are multiple threads that each process a row of the matrix, which can cause contention and synchronization issues between threads. Similarly, in the third method, each element of the matrix is processed by a separate thread, leading to a high number of threads that need to be created and managed.

Moreover, when using multiple threads, there is always a tradeoff between increased parallelism and increased overhead. While multi-threading can improve performance by utilizing multiple processing cores, the overhead of creating and managing threads can actually reduce performance if the number of threads created is too high.

Therefore, when deciding whether to use multi-threading, it's important to carefully consider the task at hand and the amount of parallelism that is truly necessary. It's not always the best solution to use multiple threads, as the overhead associated with creating and managing them can actually reduce performance in some cases.