Competitive and Multithreaded Programming

Lab2. Matrix multiplication with openMP

TYTGAT Karel

1. Goal

The objective of this laboratory was to write an algorithm to compute matrix multiplication using multithreading method

2. Definition

A matrix multiplication is a binary operation which need 2 matrix to create a third one. The result of the operation will be a product matrix of the 2 starting matrix. Two matrices can multiply only if the number of columns of the first matrix is equalled of the number of rows of the second matrix. The result matrix will have the number of rows of the first matrix and the number of columns of the second matrix.

3. Theory and implementation

Note : In this implementation, only square matrices will be used.

The main element in implementing a matrix multiplication with multithreading is how to divide the program. To have an optimal algorithm, the goal is to use the parallel tasks to compute multiple operations at the same time. But the main issue with that is to be able to build the final matrix which was computed in different threads.

To parallelize process, we use OpenMP. It allows to create threads and share memory between the created threads really easily.

There are three different ways to parallelize this program. Because the complexity of the matrix multiplication algorithm is $O(n^3)$, where n is the size of the matrix. Three loops are necessary to implements this algorithm. That's mean that it is possible to parallelize all of them. We will run some experimentations on the execution time to define which combination of parallelization is optimal.

Matrix multiplication implementation :

```
#include "pch.h"
#include <iostream>
#include <stdlib.h>
#include <ctime>

#define SIZE 60

using namespace std;

void printMat(int** M) {
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      cout << M[i][j] << ",";
    }
    cout << endl;
  }
  cout << endl;
}

int main() {
  /*Init Mat*/
  int** A = new int*[SIZE];
  int** B = new int*[SIZE];
  int** C = new int*[SIZE];
  for (int i = 0; i < SIZE; i++) {
    A[i] = new int[SIZE];
    B[i] = new int[SIZE];
    C[i] = new int[SIZE];
  }

  /*Fill Mat*/
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      A[i][j] = rand() % 100;
      B[i][j] = rand() % 100;
    }
  }
  clock_t begin = clock();
#pragma omp parallel for shared(C)
  {
    for (int i = 0; i < size; i++) {
#pragma omp parallel for shared(C)
      for (int j = 0; j < size; j++) {
        int sum = 0;
#pragma omp parallel for shared(C, sum)
        for (int k = 0; k < size; k++) {
          sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
      }
    }
  }
  clock_t end = clock();
  float elapsed_secs = float(end - begin) / CLOCKS_PER_SEC;
  cout << elapsed_secs << "s" << endl;
  cout << endl;

  /*Free Mat*/
  for (int i = 0; i < SIZE; i++) {
    free(A[i]);
    free(B[i]);
    free(C[i]);
  }
  free(A);
  free(B);
  free(C);
}
```

4. Results

We have tested the execution of the algorithm in diverse configurations. In the algorithm implementation, there are 3 loops that are executed each time by the size of the matrix. The test consist of executing the algorithm on different matrix sizes to see the execution time of the algorithm. Each line represents a different configuration. The different loops are named I, J and K.

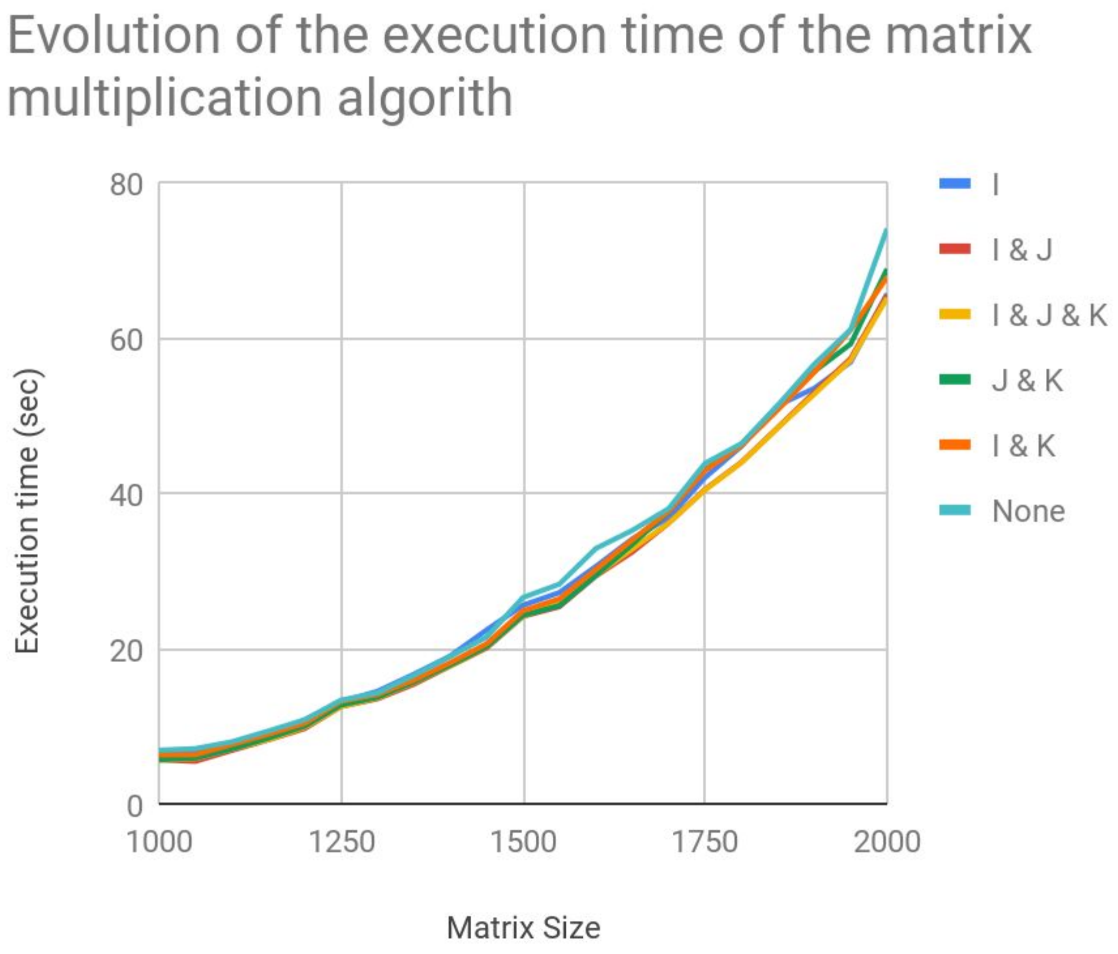
Main algorithm execution :

```
#pragma omp parallel for shared(C)
  {
    for (int i = 0; i < size; i++) { // I loop
#pragma omp parallel for shared(C)
      for (int j = 0; j < size; j++) { // J loop
        int sum = 0;
#pragma omp parallel for shared(C, sum)
        for (int k = 0; k < size; k++) { // K loop
          sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
      }
    }
  }
```

These results are not perfect, there were launch from a single machine. To have reliable test, a better infrastructure only dedicated to the program would be better.

In the graphic below, we can see that the execution time between the different paralyzing are really similar. This can be explained by the fact that when the program, create a thread, it divides the data between the different threads. Moreover, at the end of the execution, a thread need to cross the computed data with the other threads. This is the first reason why the algorithm is not faster.

Also, the difference in execution time can be explained because of the conception of the program. for example, even if there are 100 threads execution on a single computer. Not all the threads will be executed at the same instant, it depends on the number of cores of the processor. Only one task can be executed at a given instant on a core, which mean that is the computer have 4 cores, only 4 threads will be executed at the same time.



5. C#

This algorithm was also implemented in C# (not using OpenMP).

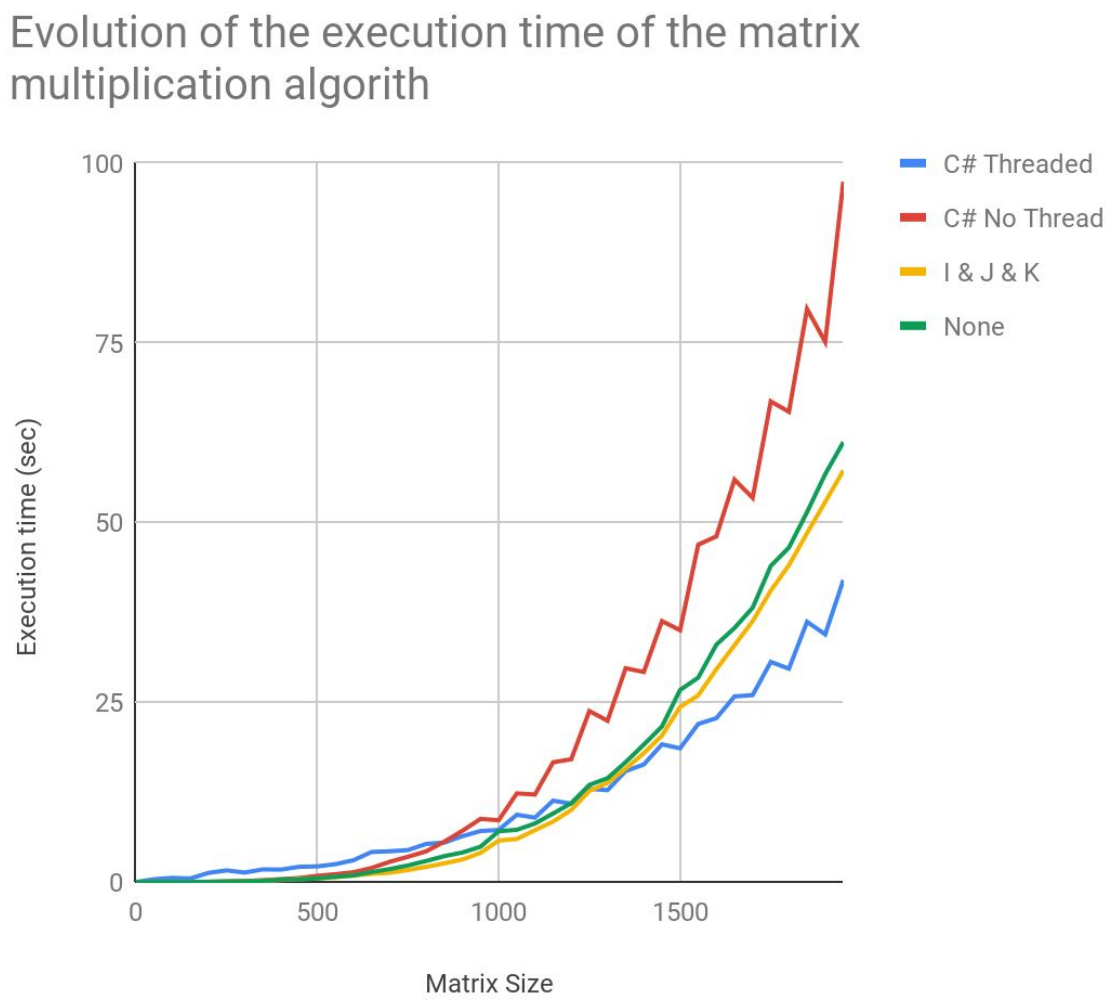
C# implementation of matrix multiplication :

```
public static double multiplyMatrix(bool useThread) {
 double execTime = 0;
 var watch = System.Diagnostics.Stopwatch.StartNew();

 int maxSize = Math.Max(MATWIDTH, MATHEIGHT);
 int minSize = Math.Min(MATWIDTH, MATHEIGHT);

 Thread[] threadList = new Thread[maxSize];

 double[][] m1 = generateMatrix(minSize, maxSize);
 double[][] m2 = generateMatrix(maxSize, minSize);

 double[][] matFinal = new double[maxSize][];

 for (int i = 0; i < maxSize; i++) {
  matFinal[i] = new double[maxSize];

  double[] row = m2[i];
  double[] column = getColumnFromMat(m2, i);

  object[] o = new object[4];
  o[0] = matFinal;
  o[1] = row;
  o[2] = m1;
  o[3] = i;

  if (useThread) {
   Thread t = new Thread(computeOneColumn);
   t.Start(o);
   threadList[i] = t;
  } else {
   computeOneColumn(o);
  }
 }

 if (useThread) {
  foreach(Thread t in threadList) {
   t.Join();
  }
 }

 watch.Stop();
 execTime = watch.ElapsedMilliseconds;
 return execTime;
}
```

The same tests were run on both programs are, here are the results. This graph shows a comparison of execution time between the C# program with thread, the C# program without using threads and the I & J & K and the execution without threads of the OpenMP algorithm.

From this graph below, we can see that the results are much more interesting. The execution time of the multiplication with big matrices is much more efficient than the execution without threads. How even, we can also see that using threads on small matrices does not improve the performances at all.

Comparing with the OpenMP implementation, we can see that it represents an in-between of the C# program. The C++ execution (with or without OpenMP) is in any situations faster than a not threaded C# program, however, it is slower than a threaded C# program.



6. Conclusion

The multithreading solution is not always the answer to a performance problem. Implementing a multithreading architecture in a software can be really time and resource consuming. That is why it is important to define the specificities of an algorithm, like the architecture of the computer or the performances require, before implementing it.