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);
}
```

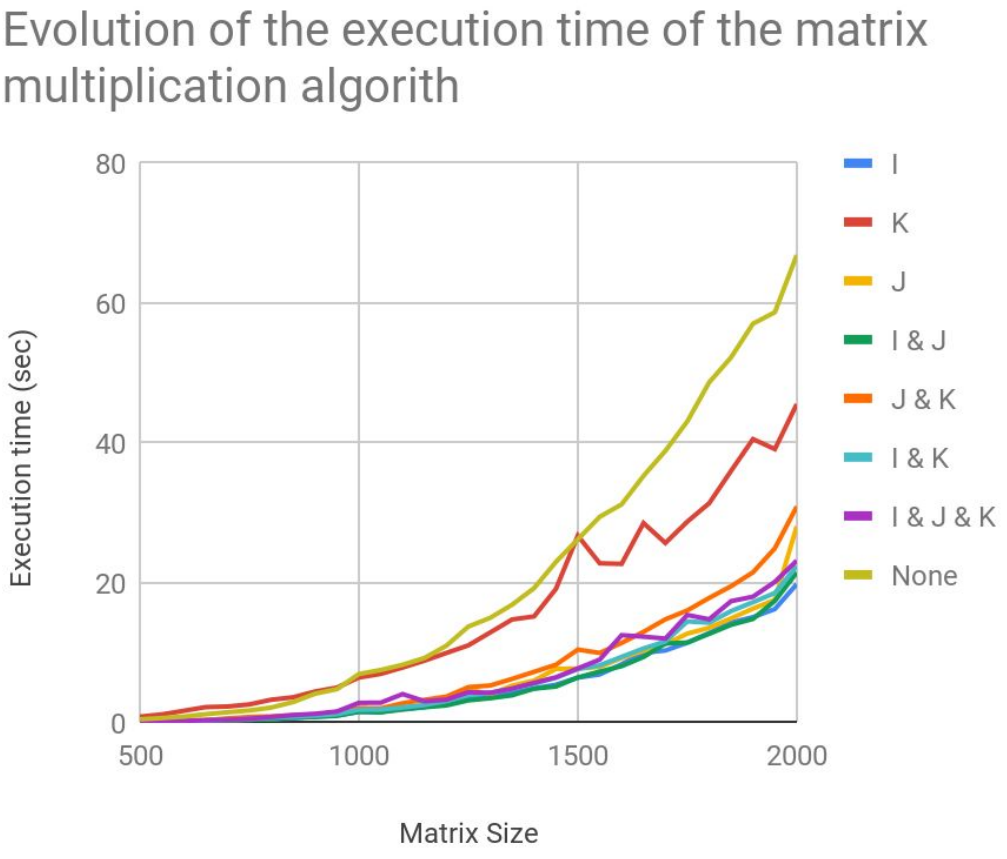
4. Results

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 :

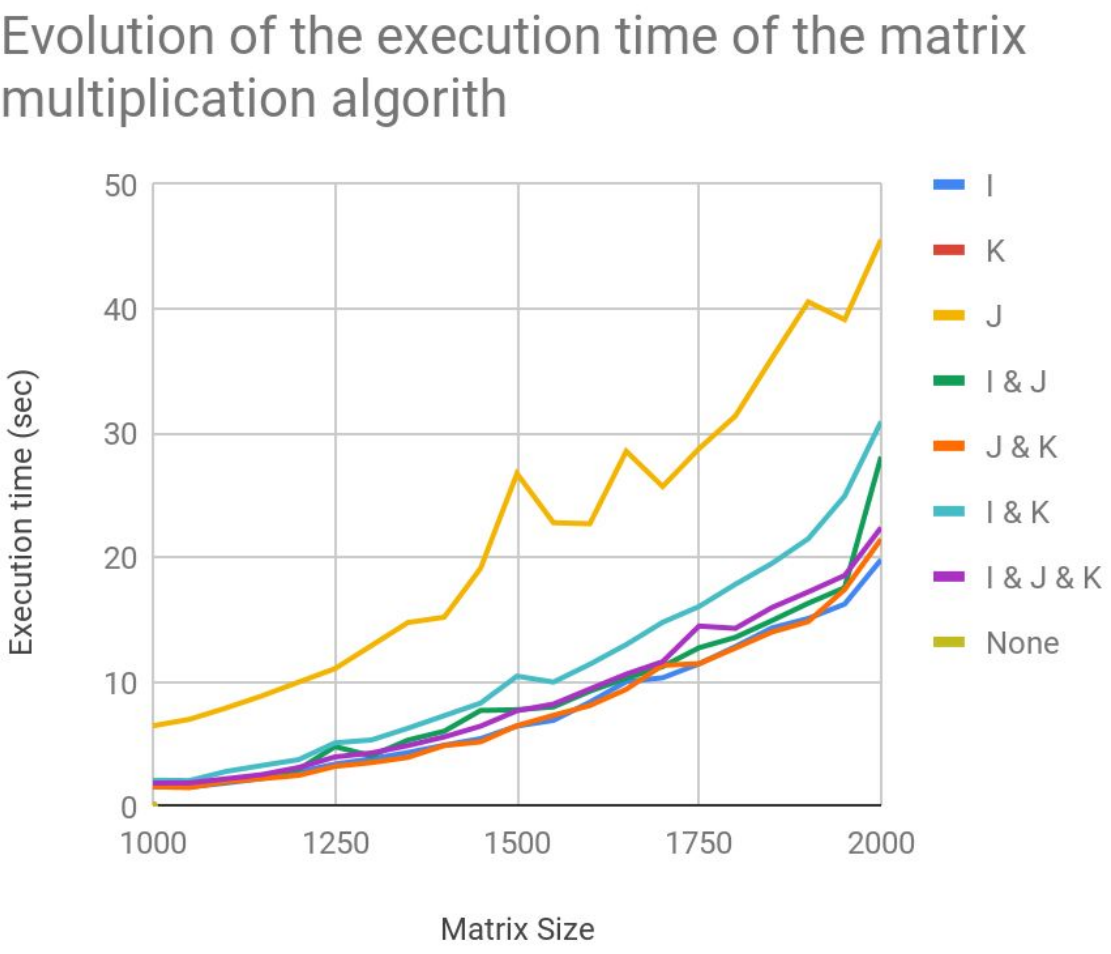
```
#pragma omp parallel for shared(C)
    for (int i = 0; i < size; i++) { // I loop
#pragma omp parallel for shared(C)
      for (int j = 0; j < size; j++) { // J loop
        int sum = 0;
#pragma omp parallel for shared(C, sum)
        for (int k = 0; k < size; k++) { // K loop
          sum += A[i][k]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



We can easily see that OpenMP drastically reduce the execution time of the program. The bigger the problem is and the bigger is the gain in execution time.

Now let's take the graphic without the line showing the program that is not using OpenMP.



In the graphic, we can see with more details the execution time of the different combination of the algorithm OpenMP. As we can see not all combinations are interesting. For example, only parallelize k is not interesting at all because we are just reducing the execution time of an O (n) complexity (only one loop is affected).

However, we can see that the parallelizing of all the loops is not the best solution in this test. We can see that the I & J & K combination is slower than J & K or just I. This can be explained by the number of operations that are needed to divide then rebuild the matrix before and after the dividing in threads. 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.

The difference in execution time can also 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).

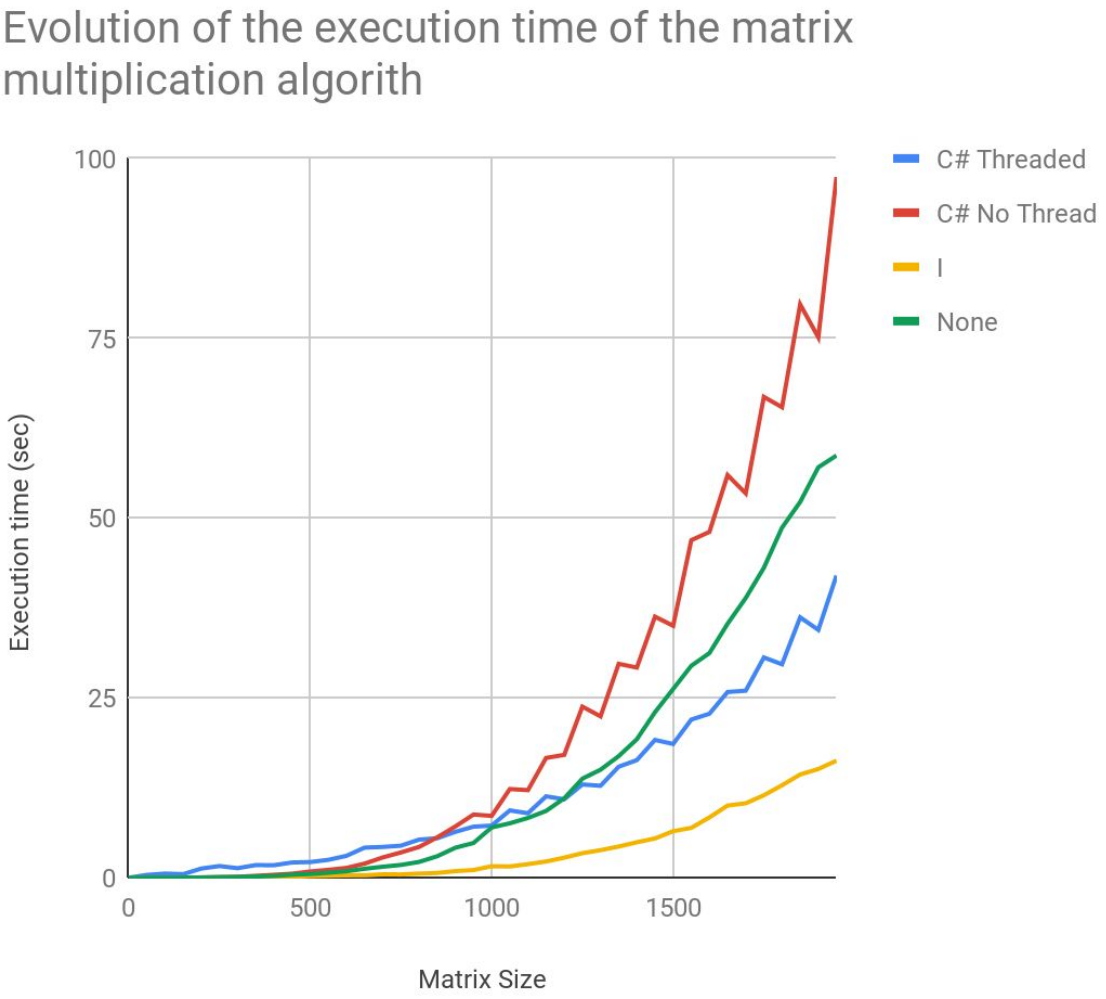
C# implementation of matrix multiplication :

```
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 just by changing the language and not using multithreading, C++ is faster. If we look at the OpenMP results, we can see that OpenMP is much more efficient than the C# execution time and seems to continue that way for bigger the problem.



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 to determine what is the best implementation.

7. Annexes

Table of the execution time in seconds for the OpenMP algorithm

Matrix Size	I	K	J	I & J	J & K	I & K	I & J & K	None
0	0,012	0,000	0,000	0,003	0,000	0,001	0,002	0,000
50	0,001	0,017	0,001	0,000	0,008	0,001	0,001	0,000
100	0,002	0,065	0,002	0,002	0,014	0,007	0,005	0,003
150	0,005	0,194	0,007	0,007	0,063	0,017	0,011	0,011
200	0,036	0,371	0,017	0,015	0,024	0,039	0,019	0,029
250	0,025	0,229	0,065	0,026	0,051	0,051	0,040	0,075
300	0,040	0,242	0,068	0,044	0,073	0,041	0,040	0,097
350	0,059	0,407	0,091	0,061	0,257	0,089	0,077	0,161
400	0,081	0,522	0,191	0,082	0,309	0,113	0,114	0,251
450	0,128	0,715	0,268	0,119	0,193	0,153	0,148	0,419
500	0,227	0,900	0,160	0,197	0,196	0,191	0,285	0,515
550	0,282	1,215	0,165	0,189	0,289	0,383	0,289	0,692
600	0,335	1,734	0,245	0,264	0,398	0,289	0,279	0,904
650	0,338	2,258	0,347	0,317	0,401	0,361	0,385	1,248
700	0,480	2,346	0,371	0,335	0,652	0,495	0,454	1,529
750	0,464	2,645	0,525	0,484	0,847	0,570	0,591	1,782
800	0,573	3,320	0,578	0,510	0,932	0,748	0,852	2,197
850	0,663	3,664	0,804	0,772	1,102	0,890	1,127	2,989
900	0,918	4,490	0,965	0,837	1,211	1,043	1,304	4,174
950	1,072	5,054	1,118	1,023	1,450	1,255	1,640	4,832
1000	1,597	6,489	1,996	1,595	2,116	1,888	2,873	6,988
1050	1,569	7,009	1,635	1,523	2,085	1,902	2,896	7,569
1100	1,890	7,915	1,955	1,991	2,817	2,226	4,126	8,312
1150	2,262	8,901	2,513	2,248	3,303	2,568	3,098	9,288
1200	2,792	10,002	2,981	2,504	3,769	3,137	3,322	11,044
1250	3,413	11,085	4,817	3,229	5,119	3,985	4,393	13,774
1300	3,833	12,933	4,075	3,533	5,351	4,305	4,305	15,021

1350	4,338	14,787	5,337	3,949	6,290	4,898	4,911	16,904
1400	4,930	15,219	6,053	4,900	7,302	5,596	5,730	19,251
1450	5,459	19,152	7,721	5,201	8,317	6,462	6,516	23,013
1500	6,473	26,762	7,781	6,503	10,481	7,700	7,788	26,235
1550	6,928	22,814	8,008	7,333	10,003	8,234	9,053	29,462
1600	8,388	22,729	9,267	8,115	11,437	9,447	12,542	31,234
1650	10,037	28,561	10,316	9,422	13,018	10,641	12,328	35,309
1700	10,351	25,721	11,249	11,356	14,809	11,644	12,056	38,889
1750	11,467	28,764	12,762	11,484	16,075	14,506	15,426	43,092
1800	12,856	31,398	13,601	12,737	17,868	14,324	14,811	48,673
1850	14,354	36,028	14,943	14,022	19,539	15,991	17,402	52,245
1900	15,117	40,563	16,341	14,866	21,524	17,251	18,042	57,058
1950	16,273	39,134	17,594	17,447	24,960	18,572	20,160	58,700
2000	19,841	45,565	28,109	21,493	30,931	22,421	23,198	66,879