Competitive and Multithreaded Programming

Lab3. Integration and LU factorization

TYTGAT Karel

1. Goal

The goal of this laboratory is to implement the problem of numerical integration and LU matrix decomposition with OpenMP in C++

2. Numerical integration

a. Definition

Numerical integration is an approximation of the computation of an integral. The idea of numerical integration is to combine multiple evaluations of the integrand to obtain the approximation of the integral. This method is used in computer science instead of the mathematical integration, which is much harder to compute.

b. Theory and implementation

In this implementation, we use the rectangle rule. The integral is calculated by dividing the area into rectangle shapes. The area of each rectangle is then computed separately, then the sum of all these areas determine the integral.
To compute an integral we need a function to integrate, an upper bound, a lower bound and the number of segments. This number of segments determine the number of rectangles to be computed to approximate the integral. The higher this value is, the more precise the approximation is:

$$\int_a^b f(x)dx \approx \frac{b-a}{h} \sum_{n=a}^{b} f(a + n\frac{b-a}{h}) \qquad \text{step} = \frac{b-a}{h}$$

This function is used to compute the integral.

- a is the lower bound
- b is the upper bound
- $f(x)$ is the function
- h is the number of segments

The complexity of this algorithm is $O(h)$. The parallelizing of this algorithm is placed on the only *for* loop to parallelize the computation of the function on each steps
The import part of this parallelizing is to get the sum of the results of each threads. The execution with and without multithreading must show the same final result.

Implementation of the integration :

```
double ComputeIntegralThread(double lowLimit, double upLimit, int numberOfSegments) {
 if (lowLimit > upLimit) {
  return ComputeIntegralThread(upLimit, lowLimit, numberOfSegments);
 }
```

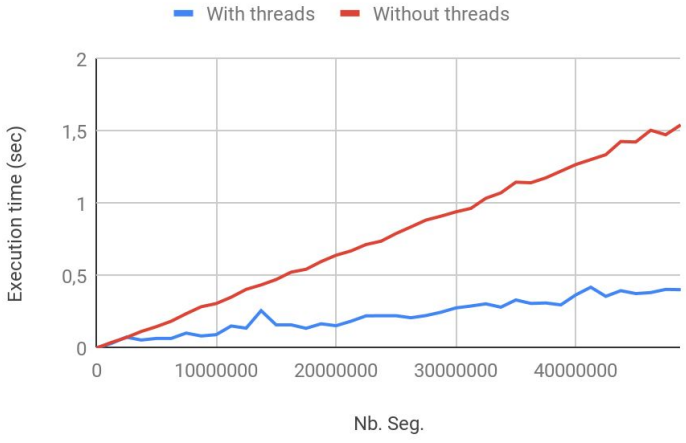
```
 double pas = (upLimit - lowLimit) / numberOfSegments;
 double sum = 0.0;
 double r = myFunction(lowLimit);

 #pragma omp parallel for reduction(+: r)
 for (int limit = 1; limit < numberOfSegments; limit++) {
  double c = lowLimit + (double) limit * pas;
  r += myFunction(c);
 }
 r = pas * r;
 return r;
}
```

c. Results

The graphic below shows the execution time in seconds of the integral of two different functions with and without using parallelizing. The two functions tested are $f(x) = x^2$ and $f(x) = x + tan(sin(x^2))$

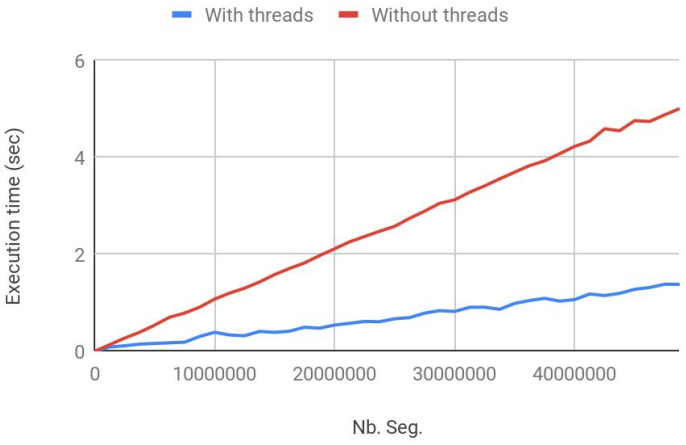
Evolution of the execution time of numerical integration

f(x) = x²



Evolution of the execution time of numerical integration

f(x) = x + tan(sin(x²))



We can see that for the two functions, the behavior is the same. The use of multithreading decrease the execution time and the bigger the problem is, the bigger the ratio between them is. Also, we can see that the execution time for small problems is almost the same, which mean that the parallelizing is useful for complex problem and does not bring additional performances on smaller problems.
We can conclude that the evolution of the execution time does not depend on the computed function, but in the parallelizing of the tasks and the size of the problem.

3. LU matrix decomposition

a. Definition

The matrix decomposition aims to factorize a matrix into a product of matrices. The LU decomposition can only be applied on square matrices. This method decomposes a matrix into two others matrices. The L (lower) matrix and the U (upper) matrix. LU is the factorization of the original matrix M where :

$$M = LU$$

b. Theory and implementation

In this implementation, the diagonal of L is always equal to 1 to be easy to compute with U. U is built from the top row to the bottom row and L is built from the left column to the right column. The building of the L and U matrices are dependant. The first row of the U matrix is built then the first column of the L matrix etc …

LU matrix decomposition :

```
int main()
{
 /*Init Mat*/
 float ** L = new float*[SIZE];
 float ** U = new float*[SIZE];
 float ** M = new float*[SIZE];
 for (int i = 0; i < SIZE; i++) {
  L[i] = new float[SIZE];
  U[i] = new float[SIZE];
  M[i] = new float[SIZE];
 }

 /*Fill Mat*/
 for (int i = 0; i < SIZE; i++) {
  for (int j = 0; j < SIZE; j++) {
   M[i][j] = float(rand() % 10);
   L[i][j] = 0;
   U[i][j] = 0;
  }
 }

 for (int i = 0; i < SIZE; i++) {
  {
   //Para U computing --> U[i][k] is modified but never used (i <> j)
#pragma omp parallel for shared(U)
   /* U */
   for (int k = i; k < SIZE; k++) {
    float sum = 0;
    //Para sum of values
#pragma omp parallel for  reduction(+:sum)
    for (int j = 0; j < i; j++) {
     sum += (L[i][j] * U[j][k]);
    }
    U[i][k] = M[i][k] - sum;
```

```
   }

   //Para L computing --> L[i][k] is modified but never used (i <> j)
#pragma omp parallel for shared(L)
   /* L */
   for (int k = i; k < SIZE; k++) {
    if (i == k)
     L[i][i] = 1; // Diag
    else {
     float sum = 0;
     //Para sum of values
#pragma omp parallel for reduction(+:sum)
     for (int j = 0; j < i; j++) {
      sum += (L[k][j] * U[j][i]);
     }
     L[k][i] = (M[k][i] - sum) / U[i][i];
    }
   }
  }
 }

 if (checkResult(L,U,M)) {
  cout << "Result OK" << endl;
 }
 else {
  cout << "Wrong result" << endl;
 }

 /*Free Mat*/
 for (int i = 0; i < SIZE; i++) {
  free(L[i]);
  free(U[i]);
  free(M[i]);
 }
 free(L);
 free(U);
 free(M);
}
```

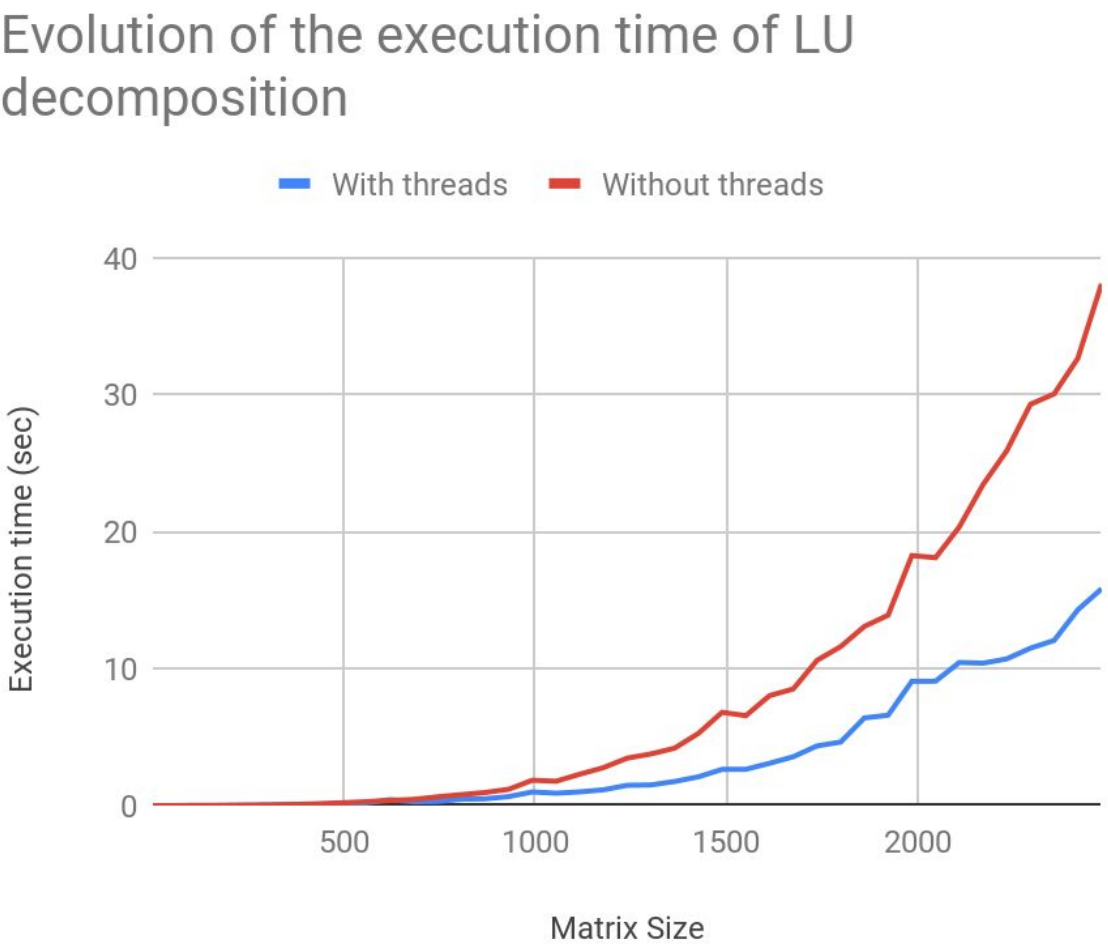
The parallelizing is not made on the main loop (i loop) because L and U are interdependent, and the U matrix needs the L matrix to finish the it processes to compute the next step. However, the matrix does not need the update of itself to be properly computed. That is the same for the sum needed to compute the value, as long as the sum is summed up at the end of the process, the threads are not interdependent.
Therefore, the L matrix waits for the U matrix to compute to compute the two matrices step by step.

The last step of the algorithm checks the result to verify of the sum up of the different threads computed properly.
In this check the L and U matrices are multiplied and the result matrix is compared to the M matrix. In this part of the program, the numbers need to be rounded because it happened that the values had 0.01 of difference between them. The matrix product is paralyzed to increase the speed of the test, however the comparison between the 2 results is not paralyzed because the function can be stopped in the middle of the process, by a return function. In this situation, it is not possible to use multiple threads.

Implementation of the matrix comparaison :

```
int checkResult(float** L, float ** U, float ** M, int size) {
 float ** C = new float*[size];
 for (int i = 0; i < size; i++) {
  C[i] = new float[size];
 }
#pragma omp parallel for shared(C)
 for (int i = 0; i < size; i++) {
#pragma omp parallel for shared(C,i)
  for (int j = 0; j < size; j++) {
   float sum = 0;
   for (int k = 0; k < size; k++) {
    sum += L[i][k] * U[k][j];
   }
   C[i][j] = sum;
  }
 }
 for (int i = 0; i < size; i++) {
  for (int j = 0; j < size; j++) {
   if (roundFloat(C[i][j]) != roundFloat(M[i][j])) {
    for (int i = 0; i < size; i++) {
     free(C[i]);
    }
    free(C);
    return false;
   }
  }
 }

 for (int i = 0; i < size; i++) {
  free(C[i]);
 }
 free(C);
 return true;
}
```

c. Results



This graph shows the computation time of the LU decomposition with and without using threads. We can clearly see that the computation time of the threaded process increases much slower that the computation time of the non threaded process. However, we can see that the computation time of the two processes is similar for small matrices. This show that for small problems, using multithreading is not necessary, but the bigger the problem is better the computation time of the multithreading will be.

4. Conclusion

In these two implementations, we can observe the same pattern. In both cases, if the problem is small, there are no real differences in the execution time. However, if compute on bigger problems, the differences between using multithreading or not will be significant. The choice of implementing multithreading in a project in interesting only on a larger scale. On smaller projects, there is no interest in implementing multithreading solution and in some situations it could even lose in performances.

5. Annexes

Table of the execution time in seconds of the integral computation

	$f(x) = x^2$		$f(x) = x + \tan(\sin(x^2))$	
Nb. Seg.	With threads	Without threads	With threads	Without threads
0	0,002	0,000	0,003	0,000
1250000	0,032	0,039	0,090	0,136
2500000	0,076	0,073	0,115	0,272
3750000	0,056	0,115	0,149	0,392
5000000	0,067	0,148	0,164	0,539
6250000	0,067	0,186	0,177	0,703
7500000	0,104	0,238	0,189	0,788
8750000	0,084	0,286	0,306	0,909
10000000	0,093	0,308	0,393	1,076
11250000	0,153	0,352	0,338	1,200
12500000	0,138	0,406	0,323	1,302
13750000	0,259	0,437	0,411	1,430
15000000	0,161	0,474	0,392	1,582
16250000	0,161	0,525	0,416	1,709
17500000	0,137	0,545	0,498	1,823
18750000	0,168	0,598	0,478	1,974
20000000	0,155	0,642	0,543	2,113
21250000	0,186	0,672	0,578	2,257
22500000	0,223	0,716	0,617	2,367
23750000	0,224	0,739	0,611	2,475
25000000	0,224	0,792	0,672	2,576
26250000	0,210	0,838	0,696	2,741
27500000	0,225	0,885	0,788	2,889
28750000	0,248	0,912	0,840	3,051
30000000	0,278	0,942	0,823	3,123
31250000	0,291	0,966	0,908	3,279
32500000	0,306	1,036	0,911	3,410
33750000	0,283	1,073	0,867	3,555
35000000	0,333	1,147	0,989	3,692
36250000	0,309	1,143	1,048	3,828

37500000	0,312	1,177	1,093	3,926
38750000	0,299	1,223	1,036	4,073
40000000	0,368	1,269	1,067	4,224
41250000	0,421	1,303	1,183	4,331
42500000	0,358	1,337	1,151	4,586
43750000	0,397	1,428	1,196	4,548
45000000	0,377	1,425	1,279	4,755
46250000	0,384	1,506	1,317	4,738
47500000	0,406	1,475	1,383	4,875
48750000	0,404	1,543	1,382	5,003

Table of the execution time in seconds of the LU matrix decomposition

Matrix Size	With threads	Without threads
1	0,001	0,000
63	0,002	0,001
125	0,006	0,004
187	0,017	0,012
249	0,033	0,027
311	0,049	0,050
373	0,096	0,075
435	0,104	0,126
497	0,159	0,204
559	0,193	0,282
621	0,431	0,373
683	0,284	0,451
745	0,301	0,636
807	0,465	0,795
869	0,491	0,951
931	0,652	1,188
993	0,992	1,849
1055	0,902	1,774
1117	0,998	2,289
1179	1,150	2,770
1241	1,476	3,463
1303	1,506	3,767
1365	1,756	4,191

1427	2,097	5,267
1489	2,640	6,807
1551	2,647	6,561
1613	3,085	8,024
1675	3,562	8,510
1737	4,352	10,597
1799	4,637	11,604
1861	6,398	13,078
1923	6,595	13,897
1985	9,072	18,247
2047	9,077	18,095
2109	10,443	20,334
2171	10,392	23,422
2233	10,705	25,890
2295	11,488	29,289
2357	12,054	30,049
2419	14,291	32,648
2481	15,824	38,087