

# Individual Project

Georgios Davakos

High-Performance Programming

Last edited: March 23, 2020

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is the LU-Factorisation . . . . .	1
1.2	Motivation and goal . . . . .	1
<b>2</b>	<b>The Implementation</b>	<b>1</b>
2.1	Structs . . . . .	1
2.2	Gaussian Elimination . . . . .	2
2.3	Parallel programming . . . . .	2
<b>3</b>	<b>Performance</b>	<b>3</b>
3.1	Runtime . . . . .	4
3.2	Cache Performance . . . . .	5
3.3	Memory Management . . . . .	5
<b>4</b>	<b>Discussion</b>	<b>5</b>
<b>A</b>	<b>Appendix</b>	<b>6</b>
	<b>References</b>	<b>6</b>

# 1 Introduction

The following report covers the *LU-Factorisation* of a matrix using Gaussian Elimination. The goal is to implement an algorithm with high time complexity and optimise it. Finally it should be noted that the LU-Factorisation is implemented in C.

## 1.1 What is the LU-Factorisation

LU-Factorisation, or lower-upper factorisation is the process of factoring a square-matrix  $A$  into a lower triangular matrix and an upper triangular matrix.

$$\begin{aligned} A &= LU \\ A &= \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \\ L &= \begin{vmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{vmatrix} \\ U &= \begin{vmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{vmatrix} \end{aligned}$$

For a matrix  $A$  to have an LU factorisation, matrix  $A$  must have an inverse and the determinant of all the leading principal minors of matrix  $A$  must be non-zero.

With LU-Factorisation it is possible for computers to solve systems of equations, find the inverse of a matrix, as well as the determinant of a matrix. Explaining how this is possible is outside the scope of the report.

## 1.2 Motivation and goal

Most algorithms for LU-Factorisation have a time complexity of  $O(n^3)$ . This is problematic when having to either compute big matrices or when the LU-factorisation has to be performed many times. There for the goal of this report is to not only implement an algorithm of a high time complexity, the goal is also to try and optimise the implementation.

# 2 The Implementation

There are many ways to solve the LU-factorisation of a matrix  $A$ , Doolittle algorithm, Crout's matrix decomposition as well as Gaussian elimination. To perform the LU-factorisation we used the Gaussian elimination due to its easy to use algorithm.

## 2.1 Structs

Before we dive into the algorithm it's important to quickly cover the struct we will be working with and explain why such a struct was chosen.

```

struct __attribute__((__packed__)) matrix
{
    int size; //size of the matrix
    double **matrix;
};

```

The matrix is of type `double**` because the program was designed to work more like a library where the user creates the matrix they want to perform the LU-Factorisation on. The matrix is a double pointer which means that we are using malloc to create the matrix in order to prevent the user from creating a matrix so massive it might not fit into either the stack or in a single array inside the heap. Finally when the LU-factorisation is performed accuracy is important so a value of type double is less likely to have round-off errors compared to a value of type float.

## 2.2 Gaussian Elimination

The Gaussian Elimination is primarily an algorithm used to find the inverse matrix of a given matrix (if the inverse exists). It is also used for solving a system of linear equations.

When applying the algorithm to perform an LU-factorisation it results in the following code[Cho+10]:

```

//size is the size of the matrix
for(int i = 0; i < size; i++)
{
    diag = m[i][i];
    for(int j = i + 1; j < size; j++)
    {
        m[j][i] = m[j][i] / diag;
        for(int k = i + 1; k < size; k++)
        {
            m[j][k] = m[j][k] - m[j][i] * m[i][k];
        }
    }
}

```

This algorithm results in altering the original matrix and turning it into a matrix that contains both the upper and the lower triangle from the LU-factorisation. So in the program the user has to use functions *create\_lower\_matrix* and *create\_upper\_matrix* to get the separate upper and lower triangular matrix.

## 2.3 Parallel programming

From observing the Gaussian Elimination algorithm it is clear that theoretically the two inner loops can be ran in parallel to improve performance but first let us explain what parallel programming is.

Parallel programming is the use of multiple cores inside a processor to run your program. This works by dividing a program into smaller chunks, then distributing each chunk to a core. To be more precise you distribute your chunks into threads, these threads are in turn connected to a core. Typically each

core has a single thread but it is possible to generate more threads to ensure that the core is always working. For example let's say the core needs to fetch data from the memory, this process will take multiple cycles, to avoid having the core stay in idle we can provide it with data to compute from an other thread until the data located in the memory has been fetched.

This is very powerful if the program has processes that can be ran concurrently. As mentioned earlier in the inner loops we can apply parallel programming to improve performance. In this report we are going to use the OpenMP API due to its' simplicity.

### 3 Performance

To conduct the performance test, a matrix was generated equal to the desired size. The value assigned to each index was done using the following function.

```
void random_matrix(matrix_t *matrix)
{
    const int size = matrix->size -1;
    double **m = matrix->matrix;
    int j = 0;
    for(int i = 0; i < size; ++i)
    {
        m[i][i] = 4;
        m[i][j+1] = -2;
        m[i+1][j] = -2;
        j++;
    }
    m[size][size] = 2;
}
```

All tests were performed on a "AMD Ryzen 3 3200U CPU 1.5Ghz" machine with a max clock speed being 2.6Ghz, 4Mb L3 cache and a gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0.

### 3.1 Runtime

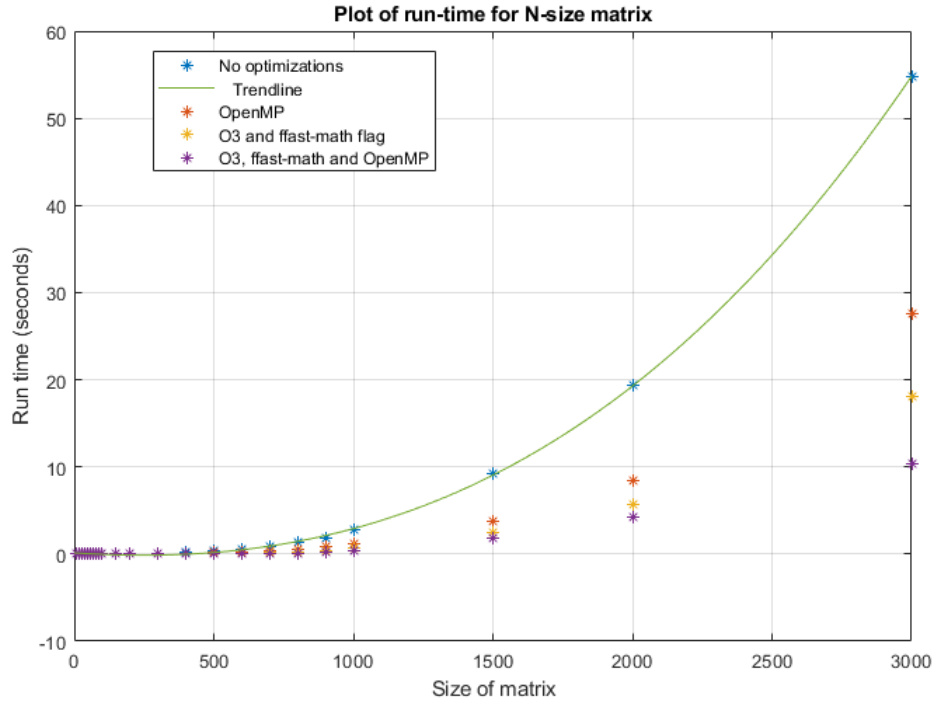


Figure 1: Graph over the execution time for `lu` for different sizes of matrix with parameters and the number of threads was 4.

No optimization	with openmp	O0	O1	O2	O3	Os	march	ffast-math
9.242	3.619	✓						
3.869	1.811		✓					
2.248	1.647			✓				
2.241	1.667				✓			
2.473	1.669					✓		
2.413	1.576				✓			✓
8.951	3.792	✓					native	
4.313	1.894		✓				native	
2.366	1.846			✓			native	
2.331	1.805				✓		native	
2.378	1.750					✓	native	
2.355	1.773				✓		native	✓

The program appears to run slower with `march=native` flag when using OpenMP.

### 3.2 Cache Performance

Matrix size:	Miss prediction
3	11,7%
50	7,1%
100	2,9%
200	1,0%
500	0,3%

### 3.3 Memory Management

The program has memory leaks when it is using OpenMP. This is due to the fact that it is compiled using GCC [L]. If it is ran without OpenMP there are no memory leak or any form of unsolved errors.

## 4 Discussion

Overall, the program is capable of performing LU-Factorisation for matrices of all sizes using the Gaussian Elimination algorithm. Also all most optimisations that could have been thought of based on the algorithm (for example the parallelism optimisation) were immediately applicable and yielded desired improvements. It should be noted that the LU-Factorisation function will work with matrices who do not have an inverse. So it is up to the user to ensure the matrix they are performing an LU-Factorisation has an inverse. This could of course be prevented but due to the restrictive time schedule we chose to make the program as simple as possible.

When optimising the code, it was first of all important to figure out which Oflag provided the most optimisation. After that gprof was used to figure were the program was spending most of its time. Since it ended up being the function which performed the LU-factorisation, it was only natural to perform the parallelism that was mentioned earlier.

```
static void inner_loop(int size, int i, double diag, double **m)
{
#pragma omp parallel for num_threads(nthreads)
    for(int j = i + 1; j < size; j++)
    {
        m[j][i] = m[j][i] / diag;
        for(int k = i + 1; k < size; k++)
        {
            m[j][k] = m[j][k] - m[j][i] * m[i][k];
        }
    }
}
```

```
for(int i = 0; i < size; i++)
{
    diag = m[i][i];
    inner_loop(size, i, diag, m);
}
```

An other optimisation that could have been performed is to utilise "Single instruction, multiple data (SIMD) to perform the divisions quicker, unfortunately it was low in the priority of optimisation and never got around to do it.

Finally to touch on the cache performance. Looking from the result it seems that the percentage of miss predictions are lowered as the size of the matrix increases. This can be explained by the fact that the inner most loop in the Gaussian Elimination algorithm spends longer and longer time going through every single element in the same row. This would lead to fewer branch miss predictions since the function doesn't need to just to a new row or to a new diagonal as frequently as it would need to do with smaller matrices.

Closing thoughts, the LU-Factorisation works like it suppose to which also is in line with the scope of this project. This library could very easily be expanded to perform the determinant and the inverse of a matrix. Sadly we don't have the time to do that but it would be a educational direction to continue on with this project.

## A Appendix

### References

- [Cho+10] Ji-Woong Choi et al. "Energy efficient hardware architecture of LU triangularization for MIMO receiver". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 57.8 (2010), pp. 632–636.
- [L] Sergey L. *Does openmp allocate memory and free all after*. URL: <https://stackoverflow.com/questions/15659832/does-openmp-allocate-memory-and-free-all-after>. (edited: 03-27-2013 at 14:29).