Individual Project

Georgios Davakos

High-Performance Programming

Last edited: March 23, 2020

Contents

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.

$$\mathbf{A} = LU$$

$$A = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$L = \begin{vmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{vmatrix}$$

$$U = \begin{vmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{vmatrix}$$

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

Their are many ways to solve the LU-factorisation of a matrix A, Doolittle algorithm, Crouts 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 the algorithm its 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 were 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 $create_{l}ower_{m}atrix$ and $create_{u}pper_{m}atrix$ 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 precis 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.

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`.

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

3.1 Runtime

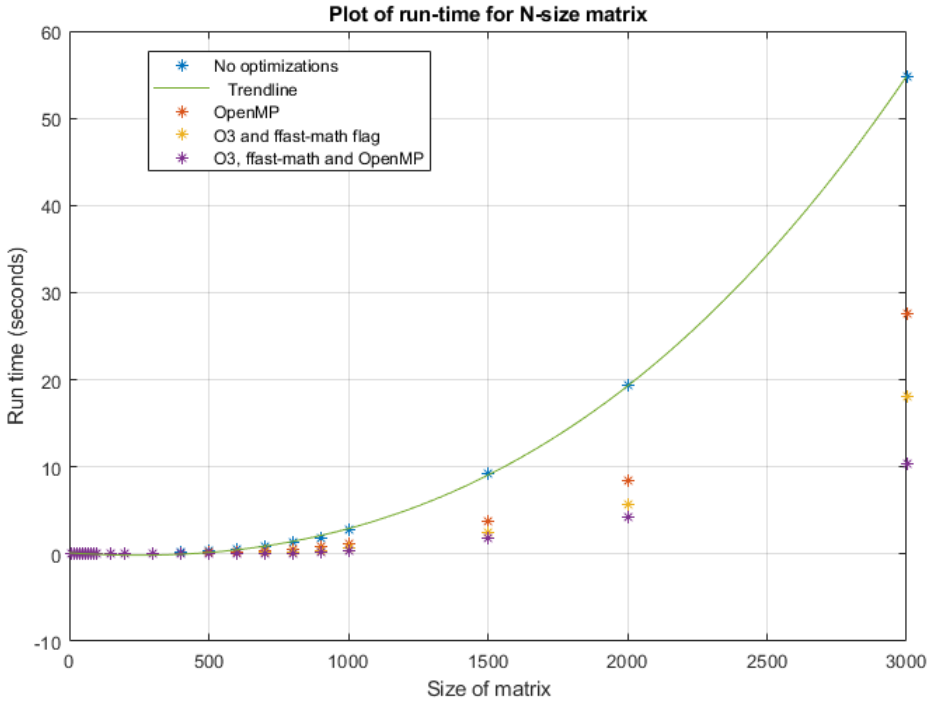

Figure 1: *Graph over the execution time for* `lu` *for different sizes of matrix with parameters and the number of threads was 4.*

`Compiler flags:`	`Time`
No flags	8,9s
O3 with ffast-math	2,4s
No flags with OpenMP	3,8
03 with ffast-math and OpenMP	1,8

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 leaks when it is using OpenMP. This is due to the fact that it is compiled used GCC. The program was tested on the Clang compiler and it leaked less than on the GCC. If it is ran without OpenMP their 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. Despite the program works as it's intended it can't be called a successful project since the program wasn't tested thoroughly. This is primarily due to time constrains which we will dive into.

To start things of, the program requires a matrix that is guaranteed to have an inverse and with all the leading principal minors of matrix A must be non-zero. Sadly the matrix that was used to test the program didn't have an inverse. The program was used on a matrix that did have an inverse which can be found in page 633 in the article "Energy efficient hardware architecture of LU triangularization forMIMO receiver" [Cho+10]. The program was also tested on the following matrix:

$$A = \begin{vmatrix} 4 & -2 & 0 \\ -2 & 4 & -2 \\ 0 & -2 & 2 \end{vmatrix}$$

What unfortunately happened was that to performance test the program, function `random_matrix` needed to always generate a matrix that had an inverse which wasn't easy to figure out. Also when the program prints the lower and upper matrices some indexes have value -nan which also indicates the given matrix can't be LU-factorised. Perhaps generating a sparse matrix like the one above could have worked but due to how close to the deadline this was discovered their isn't time to test this.

It should be noted that since the function that performs the LU-Factorisation completes its job even if the given matrix doesn't have an inverse we can still get valuable data out of the program. For instance if we assume all the subtractions and divisions occur in constant time no matter how big the values are we can use values generated from the tests to better understand the time complexity of the Gaussian Elimination as well as the performance impact of the optimisations.

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 prediction is 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 trough every single element in the same row. This would lead to fewer branch miss predictions since the function doesn't make to just to a new row or to a new diagonal as frequently as it would need to do on smaller matrices.

For closing thoughts, the LU-Factorisation works like it suppose to which also is in line with the scope of this project. The lack of a robust function to generate matrices to test the LU-Factorisation function definitely isn't ideal but with it not being the main focus of this project it is a small setback that could have been fixed if the issue had surfaced earlier.

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.