Exploring LU Factorization with Partial Pivoting

Work Assignment 2

Carlos Sá - A59905
carlos.sa01@gmail.com

Bruno Barbosa - A67646
a67646@alunos.uminho.pt

May 3, 2016

Abstract

This report is a result of a study about LU decomposition exploring partial pivoting with Matlab. In this work we'll gonna use two provided Matlab codes based on BlAS2 and BLAS3 and implement partial pivoting in both. The first one is called BLAS2LU.m wich applies a row permutation to matrix wich has m rows and n columns where $m \geq n$. The second code provided is BLAS3LU.m wich applies a block LU factorization and calls BLAS2LU to perform multiple block factorization. Both codes initially without pivoting. The main goal of this work is modifying original codes and implement partial pivoting on BLAS2LU.m and BLAS3LU.m. Our partial pivoting implementation will call BLAS2LUPP and BLAS3LUPP respectively.

On experimental component of this work we will test both codes with matrices generated randomly with different dimensions. For both solutions produced, we'll compute the numerical error using the permutation matrix and a speedup analysis to draw some conclusions.

I. Introduction to LU decomposition

Many high performance computations are based on solving a linear system such as:

$$Ax = b$$

where A is a matrix with real values and x and b are two vectors with dimension n where n is the number of columns of matrix A.

One numerical solution for solving these type of systems implies a permutation P, L and U matrices:

$$PA = LU$$

where L is the sub-matrix of A wich contains the values below the principal diagonal of A (lower triangular). The matrix L is the sub-matrix of A wich has 1's along the diagonal and the lower values of the diagonal. U is a sub-matrix of A, wich contains the values above principal diagonal of A (upper triangular). These type of decomposition is known as LU decomposition and can be ilustrated as the image bellow[1]:

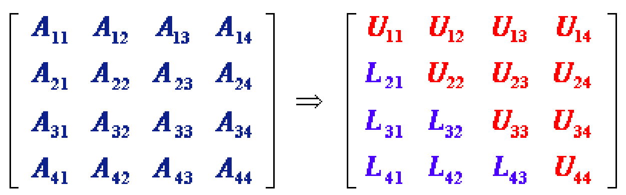

Figure 1: *$A = LU$ Decomposition of a 4×4 matrix*

[1]Image from: `http://csep10.phys.utk.edu/guidry/phys594/lectures/linear_algebra/lanotes/node3.html`

So, why using these decomposition? In high performance computing many algorithms envolves solving a linear system where A has a huge dimension also as b. The factorization effect becomes a computational expensive operation. Solving a linear system using LU decomposition allow us to decouple the factorization phase from the solving phase. Using Gaussian elimination with LU decomposition when we have b_1, b_2, b_n vectors (with A matrix constant!) is much more efficient since we just need to perform Gaussian elimination once. That's what happen when we're solving a linear system of equations for example.

However, as we learn in the Parallel algorithm classes, Gauss elimination uses partial pivoting wich is necessary for numerical stability reasons, but computational expensive. In this work we are going to explore the impact of using partial pivoting with LU decomposition in terms of numerical error and speedup.

II. Matlab codes provided

In this work we'll implement the partial pivoting with LU decomposition, and perform a numerical error and performance analisys. For doing that, 2 different Matlab codes was provided without partial pivoting.

The first one is called **BLAS2LU.m** for rectangular matrices (with m row and n columns with $m \geq n$). The second one is **BLAS3LU.m** wich applies b LU factorizations where b is the number of sub-matrices wich we divide the original matrix .BLAS3LU.m uses BLAS2LU.m to apply row permutations in blocks. The main goal is implement a partial pivoting algorithm for BLAS2LU.m and BLAS3LU.m wich we'll call they

BLAS2LUPP and BLAS3LUPP respectively.

III. BLAS Level 2 and Blas Level 3 Analisys

Both codes provided are based on BLAS Level 2 and BLAS Level 3 block LU factorizations. The next figure[1] ilustrates BLAS2 and BLAS3 LU factorizations implementation on a step **i**:

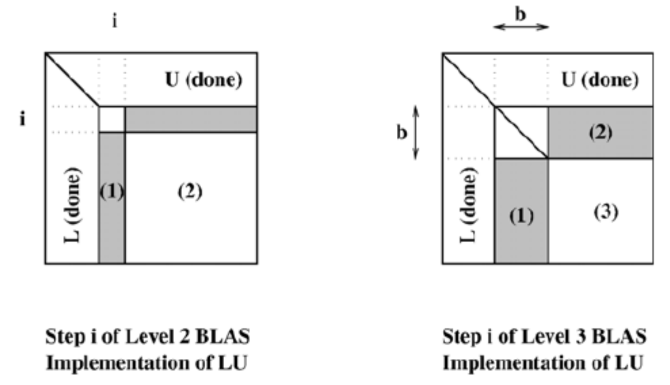
i
i
U (done)
L (done)
(1)
(2)
Step i of Level 2 BLAS Implementation of LU
b
b
U (done)
L (done)
(1)
(2)
(3)
Step i of Level 3 BLAS Implementation of LU

Figure 2: *LU implementation on BLAS2 and BLAS3*

On section VI we'll see **i** is actually a step index of BLAS2 and BLAS3 algorithm. BLAS2 and BLAS3 has different approaches but both uses LU factorization in order to solve matrix-matrix and matrix-vector more efficiently.

As we can see by the *Demmel's [1]* image above, on the left side, column 1 to i-1 of L and rows 1 to i-1 of U matrix are already done. The column i of L and row i of U are going to be computed. The A's submatrix are going to be updated by a rank-1. The submatrices on the right side are labeled by lines.

On the right side we have BLAS level 3 LU implementation. BLAS3 is focus on reorganizing the computation by delaying the submatrix **(2)** update. This procedure is performed by **b** steps. The value of **b** is actually the number of blocks that BLAS3 uses to divide the matrix into blocks of **b** columns. Later the **b** rank-1 updates are performed all at once in a single-matrix multiplication.

We've started to understand that the number of blocks chosen can influence the algorithm's speed.

By our study of *Demmels[1]* book, we understood that the correct size for **b** is always a committed relationship: b large represents speed increases when we're multiplying large matrices. But picking large b also represents performing much more floating point operations wich can be painfull in terms of performance.

So the most common criteria for choosing the correct number of blocks is analysing the cache line size's in the processor of the machine we're using. Our team laptop hardware information are ilustrated on the table 2.

We are going to use 32 and 64 values for *blocksize* as we'll see on section IV.

IV. Work Methodology

To organize the work we have done, we'll see that we can divide it into 5 different phases:

1. Understand what is LU Decomposition and the importance of partial pivoting;

2. Explore and implement LU Factorization with Partial Pivoting on Matlab (with validation with lu MATLAB's function and relative error);

3. Relative numerical error analysis according to:

$$\varepsilon_{withoutPP} = \frac{\|A-LU\|}{\|A\|} \qquad \varepsilon_{withPP} = \frac{\|PA-LU\|}{\|A\|}$$

4. SpeedUp analysis with and without partial pivoting;

5. Execution times comparison between BLAS2LUPP and BLAS3LUPP

6. Draw some conclusions about the achieved results.

In the experimental component of this work we're going to use Matrices with different sizes for testing both codes:

Matrix Dimensions	Element Type
256x256	Double (8 bytes)
512x512	Double (8 bytes)
1024x1024	Double (8 bytes)
2048x2048	Double (8 bytes)

Table 1: *Dense matrix dimensions used with BLAS2LUPP and BLAS3LUPP*

For BLAS3LUPP the number of blocks *b* used are 32 and 64. These values are related to cache line size of team laptop (see table 2) and by *Demmel's*[1] recomendation.

All of matrices were generated randomly just **once**. For that we use **rand** MATLAB function, and all matrices variables were saved[2]. Thus we can assure that all results in this work can be reproducibly. All tests was performed with same matrices. This is important so we can compare the results (with/without partial pivoting and BLAS3LUPP vs BLAS2LUPP performance). The result's validation was performed comparing our BLAS2LUPP and BLAS3LUPP with **lu** MATLAB function output's also with relative numerical error as we'll explore on section VIII.

In this work a speedup experimentation was performed to study the block partition and partial pivoting advantages/disadvantages. Therefore documenting the hardware used is important. For all tests we'll use a **Macbook Air Early 2014** wich has the following specs:

[2]The MATLAB variables file for all matrices can be downloaded here: `https://www.dropbox.com/s/v5eueyxcf54a3i1/matrices_Alldimension.mat?dl=0`

Manufacturer	Intel® Corporation
Processor	Core™ i5-4260U
Microarchitecture	Haswell
Processor's Frequency	1.40 GHz
Intel® Turbo-Boost	2.7 GHz
#Cores	2
#Threads	4
Lithography	22nm
Peak FP Performance	44.8 GFlops/s
ISA extensions	AVX 2.0 and SSE4.2
CPU technologies	Hyper-Threading Turbo Boost 2.0
Cache L1 **Cache L2** **Cache L3**	2x 32KB (I) + 2x 32KB (D) 2x 256KB 3MB
Associativity (L1/L2/L3)	8-way / 8-way / 12-way
Memory access bandwidth	13.2 GB/s
RAM Memory	2x 4GB 1600Mhz DDR3
Memory Channels	2 (0 free)

Table 2: *Team Laptop Hardware*

The version of MATLAB used was **R2015b**. Each test was **repeated 5 times** and the **median** of all was used to attenuate large swing values. All matrix elements are double wich in MATLAB uses 8bytes each.

V. Utilization of Partial Pivoting

As we'll see the LU decomposition is performed due to performance reasons. LU decomposition allow us to decouple the factorization phase (wich is computational expensive) from the solving phase. So the first step in our work was study this factorization and explore the importance of partial pivoting since Guass elimination can takes more than 90% of the computational load. That is a great motivation for using LU decomposition.

For this type of factorization we can use two types of pivoting: partial pivoting and full pivoting. The first one is based on permutation rows while the second one is based on permutation of rows and column respectively. In this study we're going to use **partial pivoting** with so we need to study how row permutation is performed.

In Parallel algorithm classes we could realise that pivoting is used just for numerical error safety. Without partial pivoting using Gauss elimination, the method is numerically unstable so using partial pivoting avoids the introduction of computation errors but also offers performance issues.

We will also gonna prove with our experimental analysis that making a partition of the original matrix into *b* blocks and perform a row permutation in each sub-matrix, allow us to achieve more than 3x speedup with our implementation. For the matrix dimensions used in this study, the impact of pivotation in terms of execution time was noticed, and bigger as we increase the matrix size. When comparing BLAS2LUPP and BLAS3LUPP implementation we noticed a more efficiently factorization on BLAS3LUPP due to block partition strategy. This will be studied in more detail on section IX.

VI. Implementation of Partial Pivoting for BLAS2LU and BLAS3LU

I. BLAS2LUPP Implementation

```
function [A,L,U,P] = BLAS2LUPP(A)
% Author Carlos Sá e Bruno Barbosa
% VERSION WITH PARTIAL PIVOTING

A_ORIGINAL = A;
start_time = tic;
[m n]=size(A);
P = eye(m);

for i=1:min(m-1,n)

    [~,p] = max(abs(A(i:n,i)));
    p = p+i-1;
    % swap rows only if pivot is not i
    if p~=i
        A([i p], :) = A([p i], :);
        P([i p], :) = P([p i], :);
    end
    A(i+1:m,i)=A(i+1:m,i)/A(i,i);
    if i<n
        A(i+1:m,i+1:n)=A(i+1:m,i+1:n)-A(i+1:m,i)*↩
            A(i,i+1:n);
    end
end

%Measuring Time
total_time = toc(start_time);

% Decomposition L U and P
L = tril(A,-1)+eye(size(A));
U = triu(A);

% Computing lu for validation and returning ↩
    permutation vector
% LU_RESULT = lu(A_ORIGINAL)

% Compute Error
 Relative_Error = norm(P*A_ORIGINAL - L*U)/norm(A↩
     )
```

II. BLAS3LUPP Implementation

```
function [A,L,U,P] = BLAS3LUPP(A,b)

A_ORIGINAL = A;
n=length(A); P = eye(n);
[m n] = size(A);

start_time = tic;
for i=1:b:n-1

 for ln=i:i+b-1
    [~,p] = max(abs(A(ln:n,ln)));

    p = p+ln-1;
    if p~=ln
        A([ln p], :) = A([p ln], :);
        P([ln p], :) = P([p ln], :);
    end
 end

 last=min(i+b-1,n);
 [A_res,~,~,P2]=BLAS2LUPP(A(i:m,i:last)); % step ↩
     1 (L22, L32)
 A(i:m,1:n) = P2 * A(i:m,1:n);
 P(i:m,1:n) = P2 * P(i:m,1:n);
 A(i:m,i:last) = A_res;

 % SIZE OF REMAINING BLOCK LARGER THAN b
 if n-i+1 > b
     L22=tril(A(i:last,i:last),-1)+eye(b);
     A(i:last,i+b:n)=inv(L22)*A(i:last,i+b:n); % ↩
         step 2 (U22)
     A(i+b:n,i+b:n)=A(i+b:n,i+b:n)-A(i+b:n,i:last↩
         )*A(i:last,i+b:n); % step 3 (U33)
    end
end
total_time = toc(start_time)

% Decomposition L U and P
L = tril(A,-1)+eye(size(A));
U = triu(A);

%Computing lu for validation
%LU_RESULT = lu(A_ORIGINAL)

% Compute Error
Relative_Error = norm(P*A_ORIGINAL - L*U)/norm(A)
```

VII. Output validation for BLAS2LUPP and BLAS3LUPP using lu MATLAB function

Since **lu** MATLAB function[3] uses partial pivoting, we use it for results validation with our implementation of LU factorization (BLAS2LUPP and BLAS3LUPP) wich also implements partial pivoting. In order to do that, we generate a small 4×4 matrix and perform the LU factorization with **lu** MATLAB function also as with our BLAS2LUPP, BLAS3LUPP:

$$A = \begin{bmatrix} 0.8687 & 0.8001 & 0.2638 & 0.5797 \\ 0.8173 & 0.2599 & 0.1818 & 0.8693 \\ 0.0844 & 0.4314 & 0.1455 & 0.5499 \\ 0.3998 & 0.9106 & 0.1361 & 0.1450 \end{bmatrix}$$

```
>> lu(A)

ans =

    0.8687    0.8001    0.2638     0.5797
    0.4602    0.5424    0.0147    -0.1218
    0.0972    0.6519    0.1103     0.5729
    0.9408   -0.9086   -0.4806     0.4885

>> BLAS2LUPP(A)

ans =

    0.8687     0.8001     0.2638     0.5797
    0.4602     0.5424     0.0147    -0.1218
    0.0972     0.6519     0.1103     0.5729
    0.9408    -0.9086    -0.4806     0.4885

>> BLAS3LUPP(A,2)

ans =

    0.8687     0.8001     0.2638     0.5797
    0.4602     0.5424     0.0147    -0.1218
    0.0972     0.6519     0.1103     0.5729
    0.9408    -0.9086    -0.4806     0.4885
```

Through this result we could validate our implementation of rows permutation. The standard MATLAB output results, only give us 4 decimals. So comparing both BLAS2LUPP and BLAS3LUPP output's with **lu** with just 4 decimals is not enough for validation. Compute the relative error is also needed.

VIII. Validation BLAS2LUPP and BLAS3LUPP with RELATIVE ERROR analysis

After validating the output results, we also compute the relative error. If the computed relative error is not in order of magnitude 10^{-15}, the partial pivoting implemented is cannot be relied and it means that something is wrong in the algorithm.

In order to analyse that, our group performed an error analysis for all algorithms (with and without partial pivoting) in execution for all all matrices. The relative error tests was performed according to:

$$\varepsilon_{withoutPP} = \frac{\|A - LU\|}{\|A\|} \tag{1}$$

[3]More about lu function: `http://www.mathworks.com/help/matlab/ref/lu.html`

and

$$\varepsilon_{withPP} = \frac{\|PA - LU\|}{\|A\|} \qquad (2)$$

On the following table 3 we can see the results achieved for error tests:

File	Partial Pivoting (PP)	Block Size	Relative_Error			
			256x256	512x512	1024x1024	2048x2048
BLAS2LU.m	without PP	-	5.0199e-16	5.8858e-16	1.9963e-16	3.1113e-16
BLAS2LUPP.m	with PP	-	2.8725e-16	4.1138e-16	4.9922e-16	6.8129e-16
BLAS3LU.m	without PP	32	5.5652e-14	4.0152e-14	4.6326e-14	3.2063e-14
		64	3.6586e-13	5.1319e-13	1.8056e-13	5.4729e-13
BLAS3LUPP.m	with PP	32	1.7302e-15	1.0989e-15	1.4381e-15	1.8987e-15
		64	2.2303e-15	4.6600e-15	5.0550e-15	3.4517e-15

Figure 3: *Relative error for all matrices with and without partial pivoting*

For BLAS2LU and BLAS3LU we use $\varepsilon_{withoutPP}$ since these codes are codes provided for this work without partial pivoting. For our BLAS2LUPP and BLAS3LUPP we use ε_{withPP}, since these two functions are our implementation of partial pivoting based on the provided code by teacher and not on any other codes we found during these study.

Our implementation reveals that all relative error values has an acceptable magnitude: 10^{-16} and 10^{-15}.

By the output validation we made on VII and this error analysis, we can conclude now that both algorithms are correct.

The table 3 shows us some interesting conclusions about the relative error among all versions of BLAS LU factorization. We can cleary see from it that BLAS2 implementations tend to produce less error than BLAS3 implementations. Nevertheless, between BLAS2 results we get a smaller error with partial pivoting since that matrices fits on cache memory (L2 and L3).

For bigger matrixes the errors is, at least, 2 times greater than the version without partial pivoting. On BLAS3 implementations we have one more variant to analyse which is the block size. This allow us to achieve a improvement due to the spatial locality. With or without partial pivoting, the error gets larger when using blocks of size 64. And this time, the use of the partial pivoting means a smaller error, even with the matrix's growth.

IX. SpeedUP Analysis

Since the partial pivoting implementation introduced on BLAS2LU and BLAS3LU changes the algorithm pattern we thought maybe this could affect the execution time. In order to understand the pivoting impact on the BLAS2LU and BLAS3LU (our BLAS2LUPP and BLAS3LUPP) we measured the execution time on BLAS2LU, BLAS3LU, and BLAS2LUPP, BLAS3LUPP for all matrix dimensions. The table 4 below show the results we achieved:

File	Partial Pivoting (PP)	Block Size	Time(sec)			
			256x256	512x512	1024x1024	2048x2048
BLAS2LU.m	without PP	-	0,0547	0,4054	4,1318	33,6542
BLAS2LUPP.m	with PP	-	0,0663	0,4855	4,4228	35,1016
BLAS3LU.m	without PP	32	0,0281	0,0553	0,2394	1,491
		64	0,0234	0,0596	0,2352	1,1457
BLAS3LUPP.m	with PP	32	0,0449	0,2423	2,422	19,1568
		64	0,042	0,1881	1,339	10,8342

Figure 4: *Execution Times for all matrices*

Our execution time results for all BLAS codes allow us to conclude that partial pivoting does not affect too much the execution times. However, lower execution times was registered on BLAS2LU and BLAS3LU where partial pivoting does not exist. The chart 5 ilustrates the speedup we can get by not using partial pivoting:

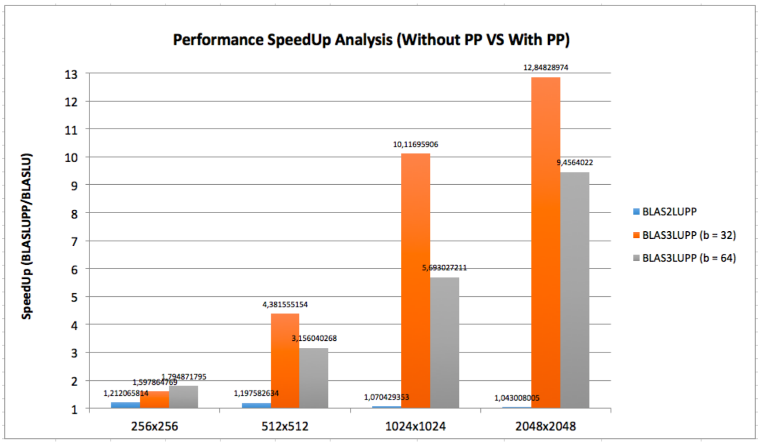

Figure 5: *SpeedUp: With and Without Partial Pivoting analysis*

The speedups of Figure 5 were obtained by the following formula:

$$SpeedUp_{withoutPP\_VS\_with\_PP} = \frac{T_{exec_{BLASXLUPP}}}{T_{exec_{BLASXLU}}} \qquad (3)$$

We were able to prove that partial pivoting affects the execution time.

Excepting on small matrices, $b = 32$ reveals to be a better block size when we compare partial pivoting with no partial pivoting. The speedups on BLAS3LUPP increases as we increase matrix dimensions.

As we explore on section V, on BLAS3 we reorganize the computation by delaying sub-matrix updates for b blocks. The update occurs later in a single matrix multiplication.

Given this strategy, we could achieve the speedup we saw on the chart below:

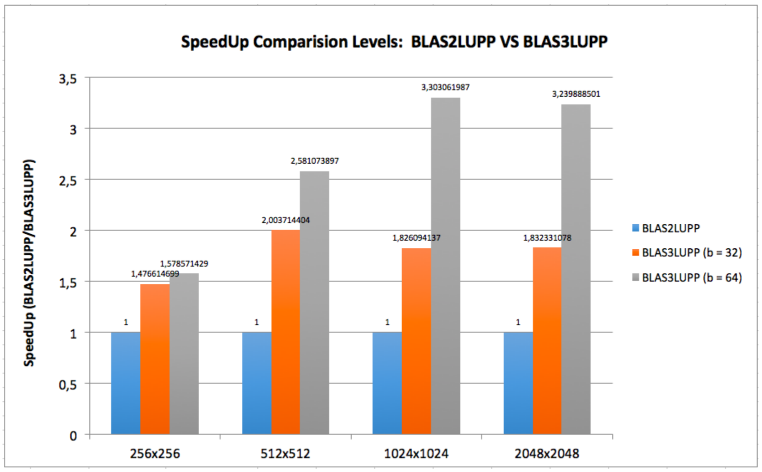

Figure 6: *SpeedUp: BLAS2 VS BLAS3 with Partial Pivoting*

The speedups from Figure 6 were calculated by the formula underneath.

$$SpeedUp_{BLAS2LUPP\_VS\_BLAS3LUPP} = \frac{T_{exec_{BLAS2LUPP}}}{T_{exec_{BLAS3LUPP}}} \tag{4}$$

In this case, we took the execution time of BLAS2LUPP as reference so blue bars has speedup = 1. From three implementations, BLAS3LUPP using b = 64 reveals to have a better performance for all matrices dimensions. This effect is more evident when we used matrices with bigger dimensions.

X. Conclusions

In this work we made an analysis to the importance of LU decomposition in linear algebra based algorithms that are computacionally intensive. We could understand that solving a linear system using LU decomposition allow us to decouple the factorization phase from the solving phase.

We can solve a linear equation system using no pivoting, using partial pivoting, or using full pivoting. In this work we made an analysis focused on partial pivoting (only with rows permutation) and no pivoting. We could verify that without using pivoting we achieved lower execution times but the algorithm becomes numerically unstable. The partial pivoting affects negatively the performance of the algorithm. Therefore using pivoting it's very important if we need numerical stability of the solution.

We could also prove that BLAS3 is generally more efficient than BLAS2 (with and without pivoting) since it uses a block partition strategy wich involves multiple factorizations and allow us to achieve a bigger speedup.

[3] [1] [2]

References

[1] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[2] Jack J. Dongarra, Lain S. Duff, Danny C. Sorensen, and Henk A. Vander Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[3] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.