High Performance Computing 2022, Assignment 1: Sparse LU Factorization

Khan, Raashid
s2705745@vuw.leidenuniv.nl

October 21, 2022

1 Introduction

The aim of this report is to explain the implementation of Sparse LU factorization kernel in C++ that applies partial pivoting and solves a linear system with gaussian elimination using forward and backward substitution. Further, it provides in detail, the challenges encountered as well as benchmarking results of the implementation on ten Suite Sparse matrices.

2 Implementation

The sparse matrix is loaded in a compressed row storage format with a dense array that stores the non zero values and row pointer arrays to store the beginning and end of a row in the sparse matrix. To perform LU factorization of a linear system such as $AX = B$, where A is an m $\times$ n sparse matrix and b is a column vector with m entries and x is an unknown column vector with n entries, A is factorized into L and U matrices using the partial pivoting algorithm. L and U matrices are lower and upper triangular matrices and these are obtained with $PA = LU$ where P is the permutation matrix that swaps rows in A matrix. Upon multiple test runs, the chosen max_n_elements for is 1 million, which seems to work fine for all our test matrices while avoiding any segmentation faults/bad access exceptions.

The algorithm works by finding a pivot and eliminating values below the pivot row by subtracting pivot row with rows below the pivot row. For a pivot, we search for an absolute large number which is sufficient enough to retain numerical stability of the factorization. While doing elimination, we perform scattering of the row elements into a dense array and perform the pivot multiplication and subtraction of rows. We then gather the non zero values back into the sparse array. This is performed by the functions - `partial_pivoting(pivot_row_index)` and `elimination(pivot_row, add_row, pivot)`.

Below are some issues that were tackled while implementing Sparse LU factorization -

1. **Memory** -
   Initially the large arrays that store the matrices were part of the functions they are used in but this brought a lot of memory issues resulting in bad access exceptions and segmentation faults as they were using stack memory. To avoid this issue, all the arrays that require the size of the input matrix are declared static in global scope so that they are stored in the heap memory.
2. **Pivot Search** -
   Within the `partial_pivoting()` function, we iterate the values array columns to find the largest non-zero value in a row(since a zero pivot can cause incorrect gaussian elimination) and swap this value with current value if its greater. Upon finding the row with the largest value, we swap this row to the initial pivot row. We save this swap into a permutation_index array so that we can permute the column vector b later when solving for unknown vector x.
3. **Masking** -
   In order to eliminate column values efficiently below the pivot row by subtracting pivot row from below rows, we scatter the other row into a dense array and subtract the two rows using the dense array which is later gathered back into the original sparse array.

4. **Garbage collection** -
   To take care of the garbage collection when using temporary dense array of the same size, we clear the values of dense array to free memory. To handle fill-in, we tried allocating reserved space in memory, in case the rows gets too dense during masking but we got memory exceptions that were unclear. Hence this was not included in the implementation.

5. **Permutation** -
   In order to save the explicit permutation made to the sparse array, we create a separate index array, that tracks the swapping of rows within the partial pivoting step. This array is then used to permute vector b before performing substitution with LU.

Upon LU factorization, we perform gaussian elimination by solving for the unknown vector x. The gaussian elimination algorithm is done in two steps within the function `substitution(b, x)` by first calculating y from $Ly = Pb$ and then calculating the unknown vector x from $Ux = y$.

3 Benchmark

The benchmarking of the implementation was performed on the lab system in the LIACS room 303 which runs Linux OS. The code compiles and ran fine in our tests with the Suite Sparse matrices.

Table 1 and Figure 1 shows the benchmarking results of each test matrix in terms of factorization time, solution time and relative errors if any in a separate table.

Test Matrix	Size	Factorization Time	Solution time
HB/mcfe	765	0.015101	0.0003430
FIDAP/ex10	2410	0.10209409	0.001272
Schenk IBMNA/c-21	3509	0.15113474	0.0008477
Bai/mhd4800b	4800	0.11762766	0.000315420
Grund/meg4	5860	0.37221242	0.002133434
Lucifora/cell1	7055	0.34205769	0.00107010
Okunbor/aft01	8205	1.0924	0.00357850
Oberwolfach/flowmeter5	9669	0.812474156	0.002212344
Gaertner/nopoly	10774	3.57450953	0.00583278
Averous/epb1	14734	1.3211321	0.0022

Table 1: Benchmarking on Suite Sparse matrices

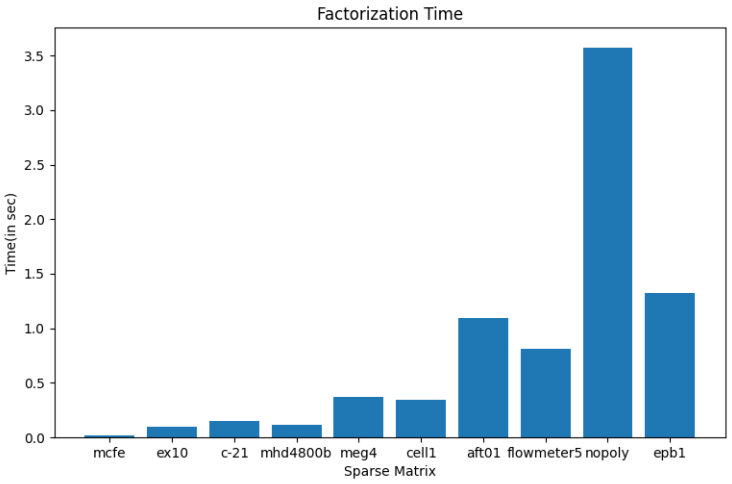

Figure 1: Benchmarking plot

Test Matrix	1	2	3	4	5
HB/mcfe	0	0	0	0	0
FIDAP/ex10	0	0	0	0	0
Schenk IBMNA/c-21	0	0	0	0	0
Bai/mhd4800b	1.001838	1.001838	0.991538	0.991538	0.991538
Grund/meg4	0	0	0	0	0
Lucifora/cell1	0	0	0	0	0
Okunbor/aft01	0	0	0	0	0
Oberwolfach/flowmeter5	0	0	0	0	0
Gaertner/nopoly	0	0	0	0	0
Averous/epb1	0	0	0	0	0

Table 2: Relative errors vs each solution vector

4 Results

As shown in the benchmarking plot Figure 1 from the data in Table 1, the factorization time for most sparse matrices is lesser than a second except for matrices Gaertner/nopoly and Averous/epb1. Upon checking the density of these two matrices, they appear to be more dense(high number of non zero values) than other matrices and hence require much more time to factorize. While in the case of Gaertner/nopoly, the matrix gets too dense during masking, causing more time for factorization. The time difference is also due to the fact that the compute time required for a sparse matrix operation is proportional to the number of arithmetic operations on nonzero quantities [1]. As for the solution time, it remains well under a second, which is expected since it only requires standard matrix vector multiplication.

Relative errors for most matrices were well below zero except for matrix Bai/mdf4800b, which was inconclusive as this only occurred for this matrix.

5 Conclusion

As part of assignment 1, we factorized an sparse matrix in compressed row storage format with partial pivoting method using only row permutation matrix and solved an linear system using gaussian elimination. In conclusion, we learned how to use partial pivoting for LU factorization but numerical stability of the solution can be better if both row and column permutations are made to eliminate values below the diagonal which was out of scope for this assignment/available time. Also since we were restricted to not use pointers and dynamic allocations, the complexity of the algorithm as well as factorization time can be reduced by utilizing them in C++.

References

[1] ADITYA SANJAY BELSARE. *SPARSE LU FACTORIZATION FOR LARGE CIRCUIT MATRICES ON HETEROGENOUS PARALLEL COMPUTING PLATFORMS*. 2014. URL: https://core.ac.uk/display/79649101?utm_source=pdf&utm_medium=banner&utm_campaign=pdf-decoration-v1.