H2020-FETHPC-2014: GA 671633

NLAFET Working Note 22

Design and implementation of a parallel Markowitz threshold algorithm

Timothy Davis, Iain S. Duff, and Stojce Nakov

Febryary 2019

Document information

This preprint report is also published as Technical Report RAL-TR-2019-003., Science & Technology Facilities Council, UK.

Acknowledgements This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Design and implementation of a parallel Markowitz threshold algorithm

Timothy Davis[1], Iain S. Duff[2], and Stojce Nakov[2]

ABSTRACT

We develop a novel algorithm for the parallel factorization of an unsymmetric sparse matrix using a Markowitz threshold algorithm. We implement this algorithm using OpenMP and show its performance on a set of standard sparse matrix test problems.

Keywords: Sparse unsymmetric matrices, Luby's algorithm, OpenMP, multicore, parallel pivot selection, sparse factorization

AMS(MOS) subject classifications: 65F30, 65F50

---

[1]Texas A & M University, College Station , Texas, USA. davis@tamu.edu
[2]Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Campus, Oxfordshire, OX11 0QX, UK.

NLAFET Working Note 22. Also published as RAL Technical Report RAL-TR-2019-003.

This work is supported by the NLAFET Project funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement 671633. Davis was supported by the National Science Foundation (CNS-1514406).

February 20, 2019

Contents

1 Introduction

We develop a novel algorithm for the parallel factorization of an unsymmetric sparse matrix using a Markowitz threshold algorithm.

We wish to solve the system of linear equations

$$Ax = b, \tag{1.1}$$

where A is a sparse matrix of dimensions $n \times n$. An entry in row i and column j is designated by a_{ij}. The right-hand side vector b and the solution vector x are of length n. In this paper, we consider the vectors x and b as dense. The method that we use for solving equation (1.1) is a direct method. That is we form an LU factorization of a permutation of the matrix A, where L is a sparse lower triangular matrix and U is a sparse unit upper triangular matrix. The permutation is chosen to maintain sparsity in the matrices L and U while also producing a numerically stable factorization.

We are particularly targeting equations with a highly unsymmetric matrix that we define as a matrix whose structure is not well approximated by the structure of $|A| + |A|^T$. Various authors have defined a measure of the asymmetry of a matrix, and here we use that defined in [27] which is the proportion of off-diagonal entries for which there is a corresponding entry in the transpose, viz.

$$si(A) = \frac{number_{i \neq j}\{a_{ij} * a_{ji} \neq 0\}}{nz\{A\}},$$

where si is called the symmetry index and $nz\{A\}$ is the number of off-diagonal entries in the matrix A. A matrix with a symmetric structure will thus have a symmetry index of 1.0. We define $0/0$ to have the value 1.0 so that a diagonal matrix will be symmetric. A triangular matrix will have symmetry index zero. Our experiments suggest that matrices with symmetry indices of less than 0.9 can be considered highly unsymmetric and these are the main target of our current work.

There are many applications that give rise to such matrices, for example, econometric modelling, chemical engineering, power systems, and linear programming although the latter has developed a large cohort of software where special techniques are used to update the factors for sequences of very related matrices.

In contrast to the case of nearly symmetric matrices, there is very little software for this class of matrices and almost no work on parallel algorithms. Available codes for this case include MA48 and `HSL_MA48` [25], UMFPACK [16], `LUSOL` [49], and `KLU` [18].

Our new algorithms and code are designed for execution on shared memory multicore nodes. To obtain an efficient implementation on such architectures is already a challenge because of the low arithmetic intensity and the complexity of the underlying algorithms and data management necessitating the design of our own memory allocation routines.

Section 2 gives an overview of related work. We discuss the Markowitz threshold algorithm in Section 3 and its parallel implementation in Section 4. We then discuss our pivot search based on Luby's algorithm in Section 5 and the parallel updating of the

matrix using these pivots in Section 6. We have written a code in C using OpenMP that is available from GitHub[1]. We use the acronym ParSHUM for our code that stands for **Par**allel **S**olver for **H**ighly **U**nsymmetric **M**atrices. We discuss the performance of this code in Section 7 before including a few concluding remarks in Section 8.

2 Related Work

In contrast to the case of nearly symmetric matrices, there is very little software for this class of matrices and no viable parallel algorithms. Currently available codes for this case include MA48 and `HSL_MA48` [25], UMFPACK [16], `LUSOL` [49], and `KLU` [18].

Prior work on parallel sparse LU factorization methods primarily focuses on matrices with mostly symmetric nonzero pattern, or they restrict the pivoting in some way. There has been far less work on more general factorization schemes and, to our knowledge, no available software. We give a brief summary of what has been done, first on left-looking algorithms then on right-looking.

The left-looking method selects the column ordering prior to factorization [34]. This allows the column elimination tree to be determined a priori. In the column elimination tree [32,33], nodes on independent branches form independent sets of pivots; George and Ng exploited this property to create a parallel factorization method [30]. Parallel left-looking variants either do not use pivoting during numerical factorization at all [48] or they only allow for partial pivoting, which does not enable further parallelism to be found via the pivot strategy. Examples of the latter approach include `SuperLU_MT` [20] and `NICSLU` [13].

Many parallel right-looking methods are based on the idea of finding independent sets of pivots prior to factorization, with no pivoting during factorization [12, 35, 36, 40, 41, 43, 46, 47, 50, 51, 53]. These methods are suitable for matrices where numerical pivoting is not required, such as sparse Cholesky factorization or the LU factorization of matrices that are diagonally dominant or nearly so.

In contrast, the `PSolve` method [15] finds pairs of rows, in parallel during numerical factorization, where the leftmost nonzero in both rows falls in the same column. One row eliminates a single entry in the other, using pairwise pivoting.

Several prior methods find independent set of pivots during numerical factorization [1, 3, 19, 38, 52]; these methods are the most closely related to the method presented in this paper. No software developed from these methods is available. In these methods, a set of pivots is found such that they form a diagonal matrix in the upper left corner when permuted to the diagonal. Alaghband and Jordan [1, 3] use a dense binary matrix to find compatible pivots, which are constrained to the diagonal; the method was then extended to allow for sequential unsymmetric pivoting [2]. Davis and Yew [19] used a non-deterministic parallel Markowitz search, where each thread searches independently. Candidate pivots are added to the pivot set in a critical section, which limits parallelism.

[1] https://github.com/NLAFET/ParSHUM

Van der Stappen and Bisseling [52] and Koster and Bisseling [38] found independent sets of pivots in a distributed memory setting (a mesh of processors).

An entirely different approach for a parallel right-looking method is to partition the matrix into independent blocks and to factorize the blocks in parallel. Duff [23] permutes the matrix into bordered block triangular form, and then factorizes each independent block with `MA28`. Geschiere and Wijshoff [31] and Gallivan et al. [28, 29] do this in `MCSPARSE`. The diagonal blocks are factorized in parallel, followed by the factorization of the border, which is a set of rows that connect the blocks. Duff and Scott [26] use a similar strategy in `MP48`, a parallel extension of `MA48`. They partition the matrix into a singly bordered block diagonal form and then use `MA48` simultaneously on each block.

The many variations of the parallel multifrontal sparse LU method (including for example `MA41` and `MUMPS`) all assume a symmetric nonzero pattern of the matrix [4–10, 14, 21, 22, 37, 39, 42]. These methods do not attempt to discover parallelism via independent sets of pivots found during numerical factorization. If numerical pivoting does occur during factorization, parallelism is reduced rather than enhanced.

3 Markowitz threshold pivoting

We first note how important it is to order a sparse matrix before or during numerical factorization. To demonstrate this, we show a table from [24] where both the benefits of using sparsity and of ordering the matrix for factorization are clearly seen. There is a substantial reduction in both operations and storage for the factorization by using sparse data structures and another significant gain if a good ordering strategy is used.

Treating matrix as	dense	sparse	
		unordered	ordered
Operations to factorize matrix ($\times 10^9$)	31 251	17.6	3.6
Storage for LU factors ($\times 10^6$)	1 300	33.4	5.3

Table 3.1: Operations for Gaussian elimination on `onetone1`, which has order 36 057 and 341 088 entries.

Right-looking sequential algorithms for Gaussian elimination choose one pivot at a time and update the trailing matrix (called the active matrix) before choosing the next pivot. In our discussions and notation in this section and the next one, we describe the situation at the beginning when the active matrix is the original matrix but the pivot selection is performed similarly for the successive active matrices.

We define by fill-in an entry that is zero in A but is nonzero in the corresponding entry of the factors. We note that we want our algorithm to reduce the fill-in since more fill-in has a direct adverse impact on storage and consequent extra cost in system solution. Clearly, for each pivot in Gaussian elimination the maximum fill-in that can be caused by using this pivot is the product of the number of other entries in the pivot row with the

number of other entries in the pivot column. Thus if there are c_j entries in column j and r_i entries in row i, then we define the Markowitz count [45] for a potential pivot in row i, column j as

$$Mark_{ij} \;=\; (r_i - 1) \times (c_j - 1). \tag{3.1}$$

We choose candidate entries with low or minimum Markowitz count to reduce the amount of fill-in. Of course such a candidate would be unacceptable if its numerical value was zero or very small relative to other entries. We therefore introduce a threshold of acceptability for a pivot and only consider entries a_{ij} that satisfy

$$|a_{ij}| \geq u \cdot \max_k |a_{kj}|, \quad k = 1, \ldots, n \tag{3.2}$$

where u is a threshold parameter $0 < u \leq 1.0$. That is to say we only consider entries that are at least u times as large as the largest entry in modulus of all entries in the column. We call such entries eligible entries. If u were equal to 1.0 then we would be using partial pivoting that is the most common algorithm for dense matrices.

To continue with the factorization we must first update the trailing matrix using the outer product of the pivot row and column, updating the numerical entries and normally introducing fill-in. This is clearly a right-looking algorithm. For selecting the next pivot we then perform the Markowitz threshold algorithm on this trailing or active matrix of order one less than the previous one, and we continue in this way until all n pivots have been chosen. The algorithm is simple but the data structures to implement it efficiently, even in serial mode, are not. We consider the details of the data structures that we use in the next section.

4 Parallel implementation of Markowitz/threshold pivoting

For our parallel implementation, we use essentially the same pivoting strategy, that is a Markowitz threshold algorithm using the same terminology as the previous section. As is common in the design of a parallel code, we will obtain much of our parallelism using blocking. We find a block of pivots at each step rather than a single pivot as described in the previous section.

In our implementation, we find a set of independent pivots that can be used in parallel. We illustrate this in Figure 4.1 where the independent pivots have been permuted to the top left-hand corner of the matrix. We then use these as a block pivot to update in parallel the active matrix. We repeat these two steps on the updated Schur complement (that is the updated active matrix) and continue doing this until either the Schur complement becomes denser than a preset value or the number of pivots found is less than a preset value. In fact, when we conducted the experiments that we describe in Section 7, we found that the number of pivots chosen at each stage was not, as might be expected, monotonically decreasing but the selection of a low number of pivots might be followed by a much larger

number of independent pivots. We thus monitor the number at each stage and do not switch unless the last few steps (the actual number is a parameter) have yielded only a very few pivots (another parameter). We then switch to using a dense factorization routine on the remaining Schur complement. In our present implementation on multicore machines we use `GETRF` from `PLASMA` [11].

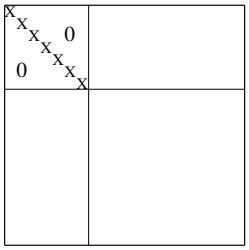

Figure 4.1: Block of independent pivots.

In sequential codes like MA48 [25], we select the eligible pivot which has the minimum Markowitz count, as defined in equation (3.1). Because we want to get large blocks of independent pivots, we relax this by accepting eligible pivots within a factor of the minimum, that is an entry (i, j) can be chosen as a pivot if its Markowitz count satisfies the condition

$$Mark_{ij} \leq \alpha_{Mark} \times Best_{Mark} \tag{4.1}$$

where the Markowitz factor α_{Mark} is greater than or equal to one and $Best_{Mark}$ is the lowest Markowitz count among all eligible entries.

We have developed a completely new pivot selection algorithm based on Luby's algorithm [44] for obtaining a maximal independent set of nodes in undirected graphs. One of our main contributions is to extend this algorithm to deal with directed graphs corresponding to unsymmetric matrices.

It is very important to first remove singletons. A *singleton* is an entry a_{ij} having no other entries in either its row i or column j. An entry a_{ij} is a row singleton if it is the only entry in its row and is a column singleton if it is the only entry in its column. Clearly, choosing a singleton as pivot will incur no fill-in.

There are two reasons for identifying and choosing singleton pivots before continuing with the main pivot selection phase. If these pivots are not handled properly, they can cause the unsymmetric-Luby's search phase to find a very small pivot set. That is to say an entry a_{ij} can be a column singleton, meaning that entries a_{kj} are all zero for $k \neq i$ but there can be many entries a_{ik} that are nonzero. This is fine as a pivot choice as there would be no fill-in but the presence of the dense row will greatly reduce the number of pivots that are independent and could be chosen at this stage. For example, if the row of the column singleton were completely dense then it would not be possible to choose any independent pivots after the choice of the column singleton. The other issue is that a singleton would

have zero Markowitz count. Thus the condition (4.1) would mean that we can only choose singletons in this pass of the algorithm.

Singletons can occur both in the original input matrix and also in the active submatrix as the factorization progresses. We thus precede the unsymmetric-Luby's pivot search algorithm (described in the next section) with a phase for selecting singletons. This phase finds all the singleton pivots and eliminates them. No update of the Schur complement is required for singleton pivots, except for the removal of entries. In this case, the set of independent pivots may not form a diagonal submatrix, but they would still be independent since singletons have no effect on the Schur complement. We improve the performance by allowing singletons and Luby's-selected pivots to be used in the same pass in the update of the active matrix.

We discuss the implementation of our algorithm in the following two sections. We give details for the unsymmetric Luby's-style pivot search in Section 5 and a detailed description of how we update the active submatrix via a Schur complement computation on both the column-form (pattern and values) and row-form (just the pattern) in Section 6.2. The two steps in these sections repeat until the matrix is factorized, or until the pivot sets become too small, or until the density of the active submatrix becomes too high. If the matrix is not factorized we complete the factorization using the dense factorization code `GETRF` from `PLASMA` [11].

5 Design of an unsymmetric "Luby's" algorithm

Our new algorithm for finding a set of independent pivots uses an extension of Luby's algorithm [44] for finding a Maximal Independent Set (MIS) of nodes in an undirected graph. In our case there are two major differences from the original Luby's algorithm: instead of applying the algorithm on an undirected graph, we adapt it for directed graphs and rather than searching for a set of independent nodes we are searching for a set of edges representing independent pivots.

A pseudo-code for Luby's algorithm for undirected graphs is given in Algorithm 1. The main idea behind the algorithm is to assign random scores to each node (line 6) and then select the nodes that have the highest score among all of their neighbours (line 7). This will guarantee that the nodes are independent. These nodes are added to the independent set I (line 8). Together with their neighbours they are removed from the original graph (lines 9-10). Performing this will ensure that the nodes that are left for the next iteration will not be dependent on any of the nodes that are already in I. This process is repeated on the reduced graph G' until it becomes empty.

We now extend this algorithm to directed graphs. The reason for this is that an unsymmetric matrix is represented as a directed graph. For a $n \times n$ matrix A, the associated graph has n nodes and for each nonzero entry in the matrix, a_{ij}, an edge exists from node i to node j. We first introduce the following terms:

- $SRC(\varepsilon_1, \varepsilon_2, \ldots)$, the source nodes for the edges $\varepsilon_1, \varepsilon_2, \ldots$

Algorithm 1 Luby's algorithm.

Input G = (V,E) an undirected graph
Output $I \subseteq G$, a MIS
$I \leftarrow \varnothing$
$G' = (V', E') \leftarrow G = (V, E)$
while $G' \neq \varnothing$ **do**
    assign random score to each node in V'
    $I' \leftarrow$ *nodes having highest score among their neighbours*
    $I \leftarrow I \cup I'$
    $Y \leftarrow I' \cup N(I')$
    $G' = (V', E')$ *is the induced subgraph on* $V' - Y$
end while

- $DST(\varepsilon_1, \varepsilon_2, \ldots)$, the destination nodes for the edges $\varepsilon_1, \varepsilon_2, \ldots$
- $IN(\nu_1, \nu_2 \ldots)$, the incoming edges to nodes $\nu_1, \nu_2 \ldots$
- $OUT(\nu_1, \nu_2 \ldots)$, the outgoing edges from nodes $\nu_1, \nu_2 \ldots$

From a matrix point of view, the $SRC()$ and $DST()$ terms, for a given list of nonzero entries (edges) $\varepsilon_1, \varepsilon_1, \ldots$ represent its row and column indices respectively, while the $OUT()$ and $IN()$ terms for a given list of indices (nodes) $\nu_1, \nu_2 \ldots$ return their rows and columns respectively. With these notations, a set of edges representing independent pivots I, as presented in Figure 4.1, is defined as:

$$\begin{aligned} &\{\ \forall \varepsilon \in I | \nexists \varepsilon' \in I, \varepsilon' \neq \varepsilon \textbf{ and} \\ &(\ \varepsilon' \in IN(DST(OUT(SRC(\varepsilon)))) \textbf{ or } \varepsilon' \in OUT(SRC(IN(DST(\varepsilon))))\ )\ \} \end{aligned} \tag{5.1}$$

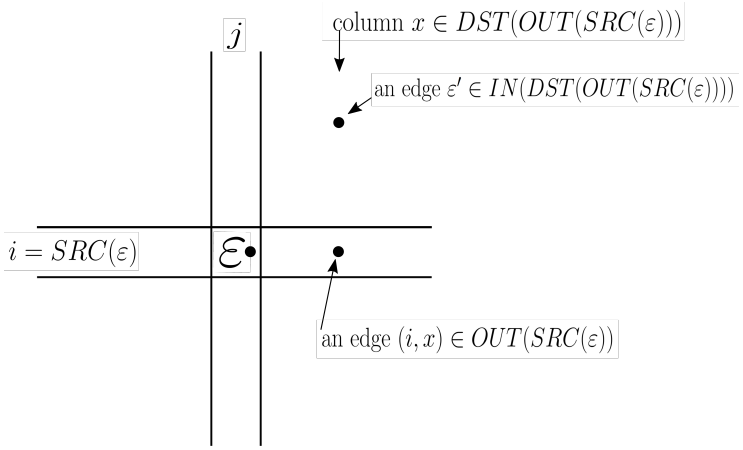

Figure 5.1: Independent set condition

The first part of equation (5.1) is illustrated in Figure 5.1, for a single edge in the independent set, $\varepsilon = (i, j) \in I$. The central point in the figure is the edge ε. Consider

the first part of (5.1), for this edge. The $SRC(\varepsilon)$ for the edge ε is the node (or row) i, and $OUT(i)$ is then the set of edges $(i, x) \in A$; that is, the nonzeros in row i. The set $DST(OUT(SRC(\varepsilon)))$ is the set of all column indices x for these nonzeros in row i. Finally, $IN(DST(OUT(SRC(\varepsilon))))$ is the set of all edges in any of these columns x. No edge ε' in this final set can appear in the independent set I (unless it is the same edge as ε itself). The second part of Equation (5.1) is similar; it states the transpose of the condition illustrated in Figure 5.1.

Our extension of Luby's algorithm is presented in Algorithm 2, where for a given directed graph G, we calculate a set of edges I satisfying the condition in equation (5.1). For each node, we pick one of the incoming edges and assign a score to it (line 6). These

Algorithm 2 Unsymmetric Luby's algorithm for pivot search algorithm.

```
1: Input G = (V,E) a directed graph
2: Output I ⊆ E
3: I ← ∅
4: G' = (V', E') ← G = (V, E)
5: while E' ≠ ∅ do
6:    I'' = {∀ν ∈ V', IN(ν) ≠ ∅, choose an edge ε |
             ε‘ ∈ IN(ν) and assign score to ε}
7:    I' ← I''
8:    for all ε ∈ I'' do
9:       for all ε' ∈ IN(DST(OUT(SRC(ε)))) do
10:         if (ε' ∈ I'') and (ε ≠ ε') then
11:            if SCORE(ε) > SCORE(ε') then
12:               remove ε' from I'
13:            else
14:               remove ε from I'
15:            end if
16:         end if
17:      end for
18:   end for
19:   I ← I ∪ I'
20:   Y_dst ← {∀ε ∈ I', OUT(SRC(IN(DST(ε))))}
21:   Y_src ← {∀ε ∈ I', IN(DST(OUT(SRC(ε))))}
22:   Y ← Y_src ∪ Y_dst
23:   E' ← E' − Y
24: end while
```

edges will be called the chosen edges. Instead of relying on a random score as in Luby's algorithm, we assign a score that will be presented in the next section. By doing this operation per node we ensure that no two edges will have the same destination node. Next, for each chosen node, among all the incoming edges to nodes that can be reached from the

source node of the chosen edge, we check if there is another chosen edge among them. If this is the case, the edge with the lower score is removed from the set of independent pivots I' calculated at this iteration. By doing this, the first condition part of equation (5.1) is satisfied for all the nodes in I'. Next, we need to make sure that there are no chosen edges among all the outgoing edges from nodes for which there is an edge that have the same destination node as our chosen edge. But if an edge like this exists, by the check that is done on the first condition of equation (5.1), one of the two will be removed. Thus, once the loop over all chosen edges (line 8) is performed, all the edges that are still in I' are independent pivots satisfying the two conditions from equation (5.1). Next, the edges that are in I' are added to the final set of independent pivots I and all the edges that satisfy the two conditions are removed from E'. The process is repeated until E' becomes empty.

In the next section we present implementation details for our extension of Luby's algorithm the Schur update of the trailing matrix.

6 Implementation details

Our algorithm can be divided in two main phases: finding a set of independent pivots and the Schur complement update on the trailing matrix. During the first phase, we rely on our unsymmetric Luby's pivot search algorithm (see Algorithm 2) which finds a set of pivots that are structurally independent, forming a diagonal submatrix when they are permuted to the diagonal. A parallel Schur complement update to the active matrix is then performed. The entire process repeats until the matrix is factorized, or until too few pivots are selected or the matrix reaches a prescribed density, at which point a dense matrix factorization algorithm is used.

In our implementation, the active matrix is held in two forms: as a set of sparse column vectors and as set of sparse row vectors. The column-form holds the pattern and the numerical values, while only the pattern is stored by rows. Since the active matrix grows because of the fill-in, every row and column is given extra space at the beginning of the factorization to accommodate some fill-in before needing to be reallocated. Additionally, instead of calling the memory manipulation functions provided by the system, we rely on a dynamic memory allocator that we wrote specifically for this algorithm and data structure. Both of the factors, L and U, are stored as a set of sparse column vectors. Additionally we use nine integer arrays and one real array. During the unsymmetric Luby's algorithm, a real array is used for storing the randomised score and one additional array is used for flagging, two integer arrays are used for storing the row and column permutation, two for the inverse permutations, two arrays that store the remaining rows and columns in the active matrix and two arrays for the computing the logical sum of the rows and columns of the set of pivots. Furthermore, we have three arrays of size n per thread, two integer and one real array. Their use will be explained in the following. We now present implementation details for the unsymmetric Luby's algorithm.

6.1 Implementation of the unsymmetric Luby's algorithm

As we explained in Section 4, we search for singletons in the active matrix. This is done by checking if a row or a column exists with only just one entry. If this is the case, these entries are flagged as pivots and are treated together with the set of independent pivots found by the unsymmetric Luby's algorithm. During this phase, the two arrays that hold the rows and columns in the active matrix are updated, removing the pivots found in the previous iteration. Algorithm 2 is then performed on the active matrix without considering the singleton rows and columns.

Following the pivoting strategy presented in Section 3, we want to consider only entries that meet some requirements. We restrict our choice, as indicated in Section 3, to eligible entries that are numerically acceptable in the sense of inequality (3.2). Additionally, the entries that we are considering as pivots should have a Markowitz count that is no higher than a given multiple of the minimum Markowitz count (see condition (4.1)). Line 6 from Algorithm 2 will be performed only on eligible entries, while the rest is performed on the entire matrix. As mentioned above, two arrays of size n are needed for the implementation of this algorithm: one used for flagging columns and another for storing the randomised score. The pivot search algorithm can be divided into two phases: the initialisation phase and the search phase. The initialisation phase is done just once for each successive active matrix, while the search phase can be repeated until all the entries are discarded.

Initialisation During this phase, a first pass by columns on the active matrix is done in parallel and the numerically ineligible entries are discarded. Discarded entries are kept in the Schur complement but simply excluded as pivot candidates. This is done by partitioning each column into two sets: eligible entries that would be numerically acceptable as pivots, and ineligible entries that are too small. At the same time, the minimum Markowitz count over all currently eligible entries is calculated. For working on subsequent Schur complements, we only have to perform the partitioning of columns that have been changed by the previous update. We identify these at the end of the search phase and thus perform the partitioning. The partitioning on all the columns is done only during the first pivot search. For each column, a *potential* pivot, with the lowest Markowitz cost, will be chosen from its eligible entries, if it satisfies the Markowitz condition (4.1). These columns are flagged using the flagging array and only these columns are considered for the remainder of the algorithm.

Main loop The search phase is divided into the following six steps. Each step is performed fully in parallel and a synchronisation is needed between each of the six steps. No atomic operations and no other critical sections are used.

- Step 1: The columns that have been flagged during the initialisation phase are split into subsets and each thread handles the columns within a subset. To each *potential*

pivot a score is assigned equal to:

$$tmp = rand(0,1)$$
$$score_{ij} = tmp \times (1 - \frac{Mark_{ij}}{\alpha_{Mark} \times Best_{Mark}}) + (1 - tmp) \times \frac{a_{ij}}{max_i} \tag{6.1}$$

where max_i corresponds to the largest absolute value in column i and $rand(0,1)$ returns a random value between zero and one. The *potential* pivots are a superset of the final *chosen* pivots, but they may form an incompatible set. By incompatible we mean that they do not satisfy the conditions in equation (5.1). In fact they could even have the same row index. Steps 2 and 3 of this search phase prune this set of *potential* pivots to find a set of valid chosen pivots.

- Step 2: Each thread examines each of its *potential* pivots and discards those that are incompatible with other potential pivots. In Algorithm 2, this operation corresponds to the loop in line 8. A thread may discard both its own potential pivots and those of another thread. Let a_{i_1,j_1} be a potential pivot. The thread that owns this pivot examines the column indices j of all nonzero entries in row i_1. If $a_{i_1,j}$ is nonzero and column j contains a potential pivot then we have two pivots that are incompatible (line 10): one in column j and one in column j_1. The one with the lower score is then flagged using the flagging array (line 11).

- Step 3: All threads now re-examine their own *potential* pivots. If the column j_1 of a potential pivot a_{i_1,j_1} has not been flagged in Step 2, then it becomes a *chosen* pivot. Each thread makes a local list of its chosen pivots. We assume that we have p threads and that thread t $(1 \leq t \leq p)$ has found k_t chosen pivots.

- Step 4: We construct a global list of the rows and columns of the *chosen* pivots, by first computing the cumulative sum of $k_1, k_2, \ldots, k_p$. If p is large, this can be done in parallel in $O(\log p)$ time. If p is small then a single thread can compute the cumulative sum in $O(p)$ time. This provides each thread with its positions in the global list of pivots for Step 5.

- Step 5: Each thread copies its set of chosen pivots into the global list of pivots and updates the array with the inverse permutations.

- Step 6: At the synchronisation barrier between Steps 5 and 6, we can redistribute work among the threads to give an equal amount of work to each thread. Next we need to discard all the entries that are compatible with the chosen pivots (lines 20 to 23). For that we compute the logical sum of all column indices in the pivotal rows and the logical sum of all row indices in the pivotal columns. As described above, we use two auxiliary array of length n in order to do this. We discard all remaining *potential* pivots for which their column index is in the logical sum that we have calculated for the columns or if their row indices are in the logical sum that

we have calculated for the rows. Finally, the arrays holding the permutations and inverse permutations are updated including the *chosen* pivots.

This process is repeated until there are no more eligible entries. However, in our experiments we have found that a single pass provides the best overall performance. Therefore, we perform only one iteration.

The logical sum for the pivotal rows and columns of the final set will determine what rows and columns are active during the Schur update. Therefore, we extract all the rows and columns that are not pivots but that have been marked. This information is then used during the Schur complement update.

6.2 Schur complement update

Updating the column-form All pivotal columns are removed from the column-form of the active submatrix and placed in L. Once this is done, the update of the column-form and the row-form of the active matrix is performed concurrently.

The update of each column in the list constructed in Step 6 of the Luby's pivot search is independent of any other column, and no locks are needed unless memory reallocation is required. We discuss the operations for column j in this list. First, column j is scanned and any pivotal rows (identified by an O(1) test on the inverse row permutation array) are removed and placed in the jth column of U. Next, the updates from each of the pivots corresponding to these rows in U are found and are applied to column j. We make this update more efficient and avoid potential multiple memory reallocations by using three temporary vectors of length n per thread (one with values, one with row indices, and the other with pointers into these arrays) to accumulate the updated column vector so that the resultant vector is only updated once, with possible memory reallocation, if the updates cause column j to exceed its current allocated space. Additionally this allows the allocator to be independent of the previous state of the column, reducing this operation to only the manipulation of pointers. Since the memory allocation is done inside a critical section, this decreases the time spent in the critical section. Finally, the partitioning of each column into eligible and ineligible entries is performed.

Updating the row-form A list with all active rows exists once Step 6 is performed and each thread treats independently a subset of that list. Entries in pivotal columns in any given row i are removed from the row and the updates from these pivots are applied one at a time, but with their pattern only, so no numerical values are computed. Similarly as in the column-form update, we use two extra temporary vectors of size n (one with column indices, and the other with pointers into this array). If row i exceeds its space, new memory is allocated to hold the row.

7 Performance and comparison with other codes

In this section we present results obtained from the ParSHUM package. All the tests in this section were performed on a system called Kebnekaise, which is located in the High Performance Computing Center North (HPC2N) at Umeå University[2]. Each compute node contains 28 Intel Xeon E5-2690v4 cores organised into 2 NUMA islands with 14 cores in each. The nodes are connected with a FDR Infiniband Network. Each CPU core has 32 KB L1 data cache, 32 KB L1 instruction cache and 256 KB L2 cache. Moreover, for every NUMA island there is 35 MB of shared L3 cache. The total amount of RAM per compute node is 128 GB. In our experiments, we use only one NUMA node, so all the tests presented below are executed on fourteen cores.

All the libraries used in these runs were compiled with GCC V7.3.0. PLASMA V3.0.0 is used for the dense factorization. PLASMA uses the Intel MKL 2019.0.117 Library for the BLAS operations. The matrices lung2, twotone, hvdc2, mac_econ_fwd500, rajat21mc2depi and pre2 are from the SuiteSparse Matrix Collection [17][3], the three matrices InnerLoop1, Jacobian_unbalancedLdf and Newton_Iteration1 are from the Power Systems application supplied by Bernd Klöss of DigSILENT GmbH and the last two matrices nug30 and esc32a are basis matrices for a quadratic assignment problems from QAPLIB[4]. The main attributes of the matrices used in this study are given in Table 7.1.

Matrix	n (10^3)	nz (10^6)	si
nug30	53	0.24	0.00
esc32a	64	0.31	0.00
lung2	109	0.49	0.57
twotone	120	1.22	0.26
hvdc2	190	1.35	0.99
InnerLoop1	197	0.75	0.44
Jacobian_unbalancedLdf	203	2.41	0.80
mac_econ_fwd500	206	1.27	0.07
rajat21	411	1.89	0.76
Newton_Iteration1	427	2.38	0.14
mc2depi	525	2.10	0.00
pre2	659	5.96	0.36

Table 7.1: Statistics for the matrices that are used in this study.

For a given matrix A, ParSHUM factorizes the matrix as:

$$PAQ = LU,$$

[2]See https://www.hpc2n.umu.se/resources/hardware/kebnekaise.
[3]Formerly called the University of Florida Sparse Matrix Collection.
[4]http://anjos.mgi.polymtl.ca/qaplib/inst.html

where P and Q are row and column permutation matrices respectively, and L and U are lower and upper triangular matrices respectively. We define the fill-in factor as the number of entries (nz) in the L and U factors divided by the number of entries in A viz:

$$\frac{nz\{L\} + nz\{U\}}{nz\{A\}}.$$

First we investigate the numerical stability of our algorithm by calculating the backward error as:

$$\frac{\|b - Ax\|_2}{\|b\|_2 + \|A\|_\infty \|x\|_2}.$$

In Figure 7.1 we present the impact on the backward error of the threshold parameter u in equation (3.2) for selecting pivots. When a low threshold is used, pivots with smaller values are accepted resulting in an increase of the backward error. This is more noticeable for the mc2depi and hvdc2 matrices (see Figure 7.1). However, when the threshold is increased, we obtain a backward error of at most 10^{-13} for all the matrices.

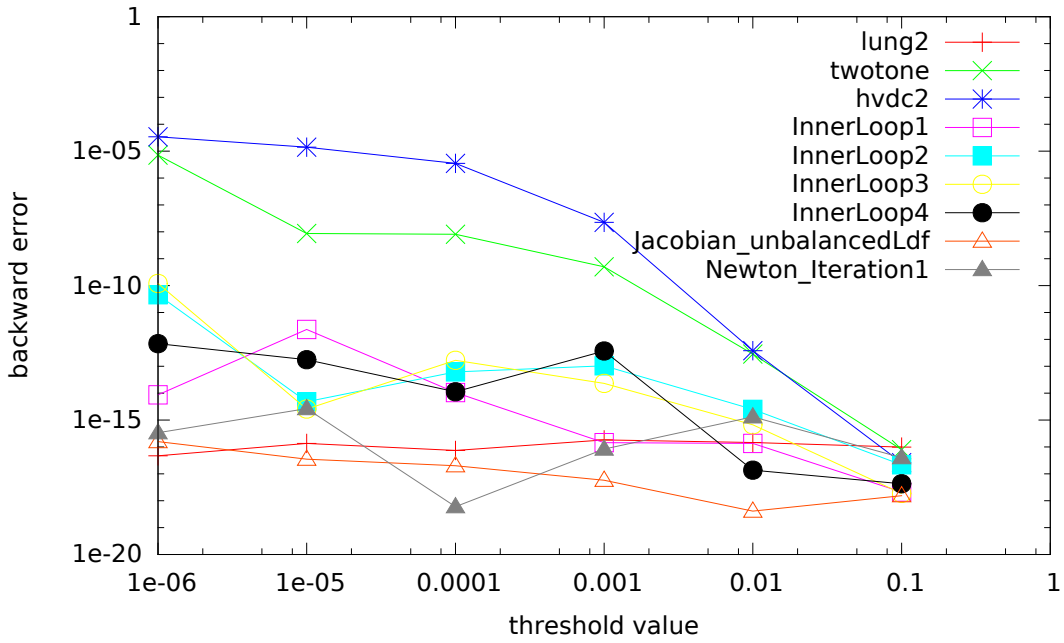

Figure 7.1: The impact of the threshold parameter, u, on the backward error for ParSHUM.

Another parameter in our algorithm is the Markowitz threshold. By relaxing this parameter, we allow pivots with higher Markowitz count to be chosen as pivot. These pivots could potentially increase the fill-in in the factors. We observe the effect of this parameter on the fill-in for each matrix in Figure 7.2. When the Markowitz threshold is increased, each matrix tends to have increased fill-in.

In order to get good performance for our algorithm, we need to find the best combination of three main parameters: a good threshold parameter so that we get a numerically correct solution; a small enough Markowitz tolerance so that the fill-in is

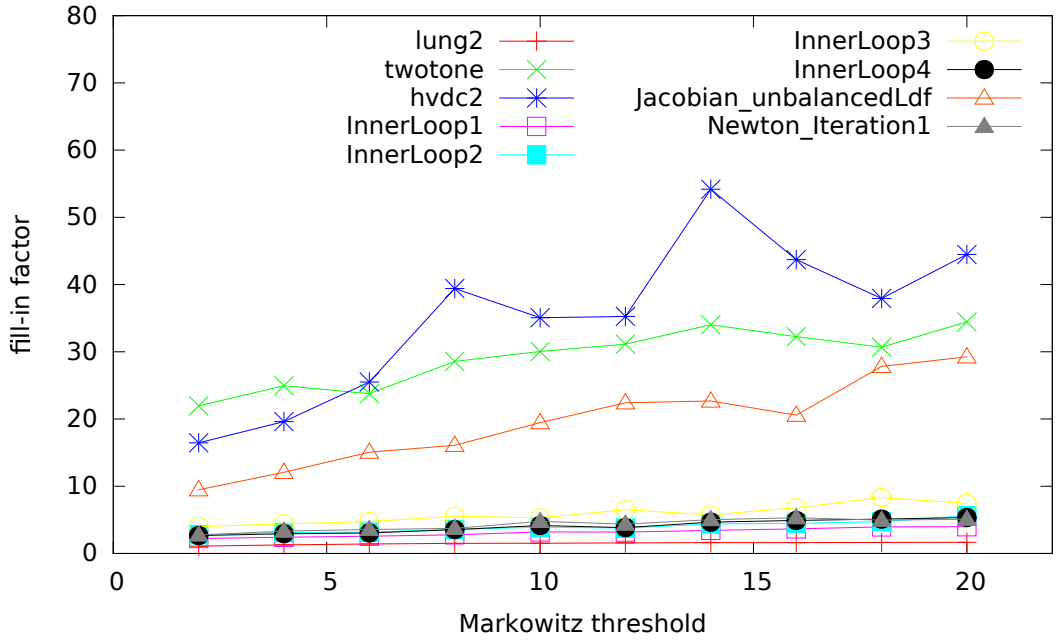

Figure 7.2: The impact of the Markowitz threshold on the fill-in factor for ParSHUM. The results for nug30, esc32a, mac_econ_fwd500, mc2depi and pre2 are not shown here because their high values dominate the graph. However, they follow a similar behaviour to the other matrices.

reasonable but a relatively large number of pivots are obtained at each stage; and the Schur density for determining when to switch to the dense code. We have done extensive testing and for each matrix we have calculated the best combination of parameters in terms of execution time for all three solvers (the Markowitz threshold does not make sense for the MA48 and UMFPACK solvers, and the density switch does not make sense for the UMFPACK solver since it already does its numerical factorization via a sequence of dense rectangular frontal matrices). The parameters are presented in Table 7.2. All the tests from now on use these optimal values, and we note that all our results have a backward error of at most 10^{-12} (10^{-16} with the exception of mc2depi).

Next, we investigate the parallel behaviour of our algorithm. In Figure 7.3, we present the execution time for two matrices, mc2depi and twotone when the number of threads is increased. The fill-in factor for the mc2depi and twotone matrices is 302 and 7.37 respectively. The parallel behaviour of our algorithm is mainly limited by the granularity of the data on which it operates and the amount of available parallelism. For matrices with a large fill-in factor, these two limitations are less important because the algorithm will operate on row and columns of larger size which will lead to more efficient execution. Additionally more parallelism is available. For instance, for the mc2depi matrix a speed-up of 11.1 is achieved when 14 threads are used. On the other hand, for the twotone matrix a speed-up of 4.7 is obtained mainly due to lower granularity and limited parallelism.

In Table 7.3 the best execution times and the fill-in factor for MA48, UMFPACK and

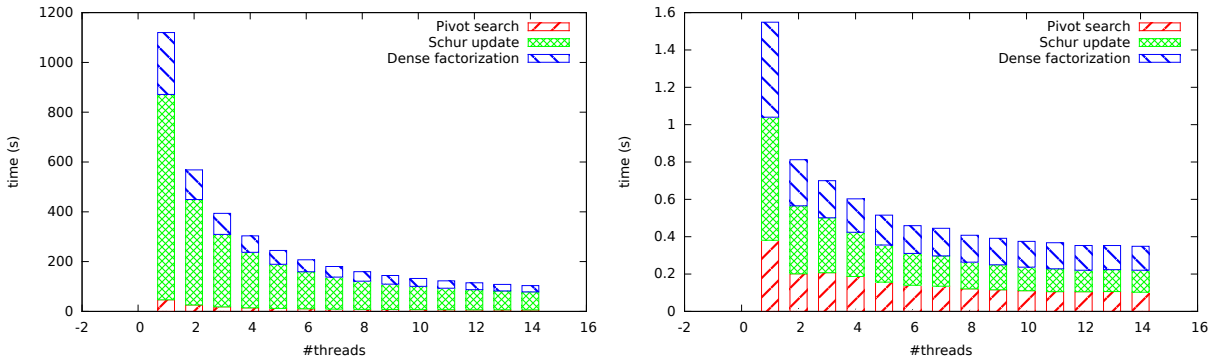

Figure 7.3: Multi-threaded results for our algorithm. Results for the mc2depi matrix are presented on the left and for the twotone matrix on the right.

ParSHUM are presented. We obtain the lowest execution times with ParSHUM for all the matrices except the Jacobian_unbalancedLdf, mac_econ_fwd500 and mc2depi matrices for which the UMFPACK solver yields the lowest execution time. The main reason for this is the fill-in factor. Indeed, the ParSHUM solver tends to obtain a larger fill-in factor than the other two solvers, but manages to obtain better performance due to its parallel execution in comparison with the sequential execution of UMFPACK and MA48. For instance, the same execution time is obtained for the hvdc2 matrix with ParSHUM and UMFPACK with a fill-in factor that is more than three times larger for ParSHUM. On the other hand, for the mac_econ_fwd500 matrix, the UMFPACK solver has a fill-in of 57.5 while a fill-in of 450 is obtained with the ParSHUM solver, resulting in a much higher execution time.

The exceptions are the nug30 and esc32a matrices, for which a considerably lower fill-in is obtained with the ParSHUM solver, resulting in much lower execution time. For these matrices, the switch to dense code for MA48 is done very early, when density of only one percent is achieved for the nug30 matrix for instance. If the switch is done later, the sparse part of the algorithm becomes too expensive for the MA48 solver. But since MA48 relies on its own sequential dense factorization, the execution time is high. This is the main reason for the overall higher execution times of MA48.

	ParSHUM			MA48		UMFPACK
Matrix	u	α_{Mark}	Φ	u	Φ	u
nug30	10^{-6}	2	0.1	10^{-2}	0.01	10^{-1}
esc32a	10^{-6}	2	0.15	10^{-2}	0.05	10^{-1}
lung2	10^{-4}	8	0.1	1.0	0.4	10^{-4}
twotone	10^{-2}	2	0.2	10^{-2}	0.8	10^{-5}
hvdc2	10^{-2}	4	0.1	10^{-1}	0.4	10^{-5}
InnerLoop1	10^{-5}	16	0.1	10^{-3}	0.8	10^{-7}
Jacobian_unbalancedLdf	10^{-8}	4	0.1	10^{-3}	0.8	10^{-1}
mac_econ_fwd500	10^{-4}	3	0.2	10^{-1}	0.1	10^{-4}
rajat21	10^{-9}	16	0.1	10^{-5}	0.2	10^{-2}
Newton_Iteration1	10^{-6}	16	0.1	10^{-2}	0.8	10^{-4}
mc2depi	10^{-1}	2	0.1	10^{-1}	0.1	10^{-1}
pre2	10^{-4}	2	0.1	10^{-3}	0.2	10^{-7}

Table 7.2: The parameters and their optimal values for the results in Figure 7.3 and Table 7.3 are shown in columns 2-4 (ParSHUM), 5-6 (MA48) and 7 (UMFPACK). The parameters u and α_{Mark} are defined in Sections 3 and Φ is the density at which the switch to full code is made.

	ParSHUM		MA48		UMFPACK	
Matrix	time	fill-in	time	fill-in	time	fill-in
nug30	**0.71**	142.	31.0	450.	48.15	463.
esc32a	**0.46**	70.9	7.54	154.	71.0	341.
lung2	**0.04**	1.48	0.37	2.55	0.10	1.44
twotone	**0.35**	7.37	4.00	18.8	0.49	5.45
hvdc2	**0.38**	6.91	0.42	1.82	**0.38**	2.06
InnerLoop1	**0.13**	2.71	0.22	1.99	0.30	2.48
Jacobian_unbalancedLdf	0.99	10.9	4.21	5.40	**0.74**	3.88
mac_econ_fwd500	33.5	450.	4137.	483.	**4.62**	57.5
rajat21	**0.11**	1.64	9.74	1.26	44.5	1.89
Newton_Iteration1	**0.46**	3.46	1.23	2.55	1.00	2.55
mc2depi	104.	302.	1816.	144.	**4.36**	38.6
pre2	**13.0**	57.0	124.1	44.8	25.8	32.3

Table 7.3: The execution time and the fill-in factor for ParSHUM, MA48 and UMFPACK solvers. Best results (or results within 5% of the best) are highlighted in bold.

8 Conclusions

We have presented a completely new approach for finding sets of independent pivots in an unsymmetric matrix (that is, a directed graph), based on a novel extension of Luby's method for undirected graphs [44]. We have also developed a robust library code called ParSHUM that implements a sparse factorization method based on this idea, as well as a parallel Schur complement update. The package has been published as easily-accessible, well-documented open source software (https://github.com/NLAFET/ParSHUM). The performance results of our method show that it obtains excellent performance for highly unsymmetric matrices. It is typically much faster than two widely-used packages on a multicore system (MA48 and UMFPACK), even though it sometimes produces factorizations with more fill-in.

Acknowledgements

This project is funded from the European Union's Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633. Davis was supported by the National Science Foundation (CNS-1514406). We thank the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support. We would like to thank Alan Ayala (Inria), Carl Christian Kjelgaard Mikkelsen (Umeå University), and Sébastien Cayrols (RAL) for their comments on an early draft of this paper that was presented as Deliverable 3.5 in the NLAFET Project.

References

[1] G. Alaghband, *Parallel pivoting combined with parallel reduction and fill-in control*, Parallel Computing, 11 (1989), pp. 201–221.

[2] ——, *Parallel sparse matrix solution and performance*, Parallel Computing, 21 (1995), pp. 1407–1430.

[3] G. Alaghband and H. F. Jordan, *Sparse Gaussian elimination with controlled fill-in on a shared memory multiprocessor*, IEEE Trans. Comput., 38 (1989), pp. 1539–1557.

[4] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Computer Methods Appl. Mech. Eng., 184 (2000), pp. 501–520.

[5] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41.

[6] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li, *Impact of the implementation of MPI point-to-point communications on the performance of two general sparse solvers*, Parallel Computing, 29 (2003), pp. 833–947.

[7] P. R. Amestoy, I. S. Duff, S. Pralet, and C. Vömel, *Adapting a parallel sparse direct solver to architectures with clusters of SMPs*, Parallel Computing, 29 (2003), pp. 1645 – 1668.

[8] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel Computing, 32 (2006), pp. 136 – 156.

[9] P. R. Amestoy, J.-Y. L'Excellent, F.-H. Rouet, and W. M. Sid-Lakhdar, *Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver*, in Proc. High-Performance Computing for Computational Science, VECPAR 2014, Eugene, Oregon, USA, 2014.

[10] P. R. Amestoy, J.-Y. L'Excellent, and W. M. Sid-Lakhdar, *Characterizing asynchronous broadcast trees for multifrontal factorizations*, in Proc. SIAM Workshop on Combinatorial Scientific Computing (CSC14), Lyon, France, July 2014, pp. 51–53.

[11] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Computing, 35 (2009), pp. 38–53.

[12] D. A. Calahan, *Parallel solution of sparse simultaneous linear equations*, in Proceedings of the 11th Annual Allerton Conference on Circuits and System Theory, 1973, pp. 729–735.

[13] X. Chen, Y. Wang, and H. Yang, *NICSLU: an adaptive sparse matrix solver for parallel circuit simulation*, IEEE Trans. Computer-Aided Design Integ. Circ. Sys., 32 (2013), pp. 261–274.

[14] J. M. Conroy, S. G. Kratzer, R. F. Lucas, and A. E. Naiman, *Data-parallel sparse LU factorization*, SIAM J. Sci. Comput., 19 (1998), pp. 584–604.

[15] T. A. Davis and E. S. Davidson, *Pairwise reduction for the direct, parallel solution of sparse unsymmetric sets of linear equations*, IEEE Trans. Comput., 37 (1988), pp. 1648–1654.

[16] Timothy A. Davis and Iain S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 140–158.

[17] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.

[18] Timothy A. Davis and Ekanathan Palamadai Natarajan, *Algorithm 907: KLU, A direct sparse solver for circuit simulation problems*, ACM Trans. Math. Softw., 37 (2010), p. 17 pages.

[19] T. A. Davis and P. C. Yew, *A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 383–402.

[20] J. W. Demmel, J. R. Gilbert, and X. S. Li, *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 915–952.

[21] I. S. Duff, *The parallel solution of sparse linear equations*, in CONPAR 86, Proc. Conf. on Algorithms and Hardware for Parallel Processing, Lecture Notes in Computer Science 237, W. Handler, D. Haupt, R. Jeltsch, W. Juling, and O. Lange, eds., Berlin: Springer-Verlag, 1986, pp. 18–24.

[22] ——, *Multiprocessing a sparse matrix code on the Alliant FX/8*, J. Comput. Appl. Math., 27 (1989), pp. 229–239.

[23] ——, *Parallel algorithms for sparse matrix solution*, in Parallel computing. Methods, algorithms, and applications, D. J. Evans and C. Sutti, eds., Adam Hilger Ltd., Bristol, 1989, pp. 73–82.

[24] Iain S. Duff, Albert M. Erisman, and John K. Reid, *Direct Methods for Sparse Matrices. Second Edition.*, Oxford University Press, Oxford, England, 2016.

[25] Iain S. Duff and John K. Reid, *The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations*, ACM Trans. Math. Softw., 22 (1996), pp. 187–226.

[26] I. S. Duff and J. A. Scott, *A parallel direct solver for large sparse highly unsymmetric linear systems*, ACM Trans. Math. Softw., 30 (2004), pp. 95–117.

[27] A. M. Erisman, R. G. Grimes, J. G. Lewis, W. G. Poole Jr., and H. D. Simon, *Evaluation of orderings for unsymmetric sparse matrices*, SIAM Journal on Scientific and Statistical Computing, 7 (1987), pp. 600–624.

[28] K. A. Gallivan, P. C. Hansen, Tz. Ostromsky, and Z. Zlatev, *A locally optimized reordering algorithm and its application to a parallel sparse linear system solver*, Computing, 54 (1995), pp. 39–67.

[29] K. A. Gallivan, B. A. Marsolf, and H. A. G. Wijshoff, *Solving large nonsymmetric sparse linear systems using MCSPARSE*, Parallel Computing, 22 (1996), pp. 1291–1333.

[30] A. George and E. G. Ng, *Parallel sparse Gaussian elimination with partial pivoting*, Annals of Oper. Res., 22 (1990), pp. 219–240.

[31] J. P. Geschiere and H. A. G. Wijshoff, *Exploiting large grain parallelism in a sparse direct linear system solver*, Parallel Computing, 21 (1995), pp. 1339–1364.

[32] J. R. Gilbert and J. W. H. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 334–354.

[33] J. R. Gilbert and E. G. Ng, *Predicting structure in nonsymmetric sparse matrix factorizations*, in Graph Theory and Sparse Matrix Computation, A. George, J. R. Gilbert, and J. W. H. Liu, eds., vol. 56 of IMA Volumes in Applied Mathematics, Springer-Verlag, New York, 1993, pp. 107–139.

[34] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Comput., 9 (1988), pp. 862–874.

[35] J. W. Huang and O. Wing, *Optimal parallel triangulation of a sparse matrix*, IEEE Trans. Circuits and Systems, CAS-26 (1979), pp. 726–732.

[36] J. A. G. Jess and H. G. M. Kees, *A data structure for parallel LU decomposition*, IEEE Trans. Comput., C-31 (1982), pp. 231–239.

[37] Kyungjoo Kim and Victor Eijkhout, *A parallel sparse direct solver via hierarchical DAG scheduling*, ACM Trans. Math. Softw., 41 (2014), pp. 3:1–3:27.

[38] J. Koster and R. H. Bisseling, *An improved algorithm for parallel sparse LU decomposition on a distributed-memory multiprocessor*, in Proc. Fifth SIAM Conference on Applied Linear Algebra, Snowbird, Utah, June 1994, SIAM, pp. 397–401.

[39] S. G. Kratzer and A. J. Cleary, *Sparse matrix factorization on SIMD parallel computers*, in Graph Theory and Sparse Matrix Computation, A. George, J. R. Gilbert, and J. W. H. Liu, eds., vol. 56 of IMA Volumes in Applied Mathematics, Springer-Verlag, New York, 1993, pp. 211–228.

[40] M. Leuze, *Independent set orderings for parallel matrix factorization by Gaussian elimination*, Parallel Computing, 10 (1989), pp. 177–191.

[41] J. G. Lewis, B. W. Peyton, and A. Pothen, *A fast algorithm for reordering sparse matrices for parallel factorization*, SIAM J. Sci. Comput., 10 (1989), pp. 1146–1173.

[42] J.-Y. L'Excellent and W. M. Sid-Lakhdar, *Introduction of shared-memory parallelism in a distributed-memory multifrontal solver*, Parallel Computing, 40 (2014), pp. 34–46.

[43] J. W. H. Liu and A. Mirzaian, *A linear reordering algorithm for parallel pivoting of chordal graphs*, SIAM J. Disc. Math., 2 (1989), pp. 100–107.

[44] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1053.

[45] H. M. Markowitz, *The elimination form of the inverse and its application to linear programming*, Management Science, 3 (1957), pp. 255–269.

[46] F. J. Peters, *Parallel pivoting algorithms for sparse symmetric matrices*, Parallel Computing, 1 (1984), pp. 99–110.

[47] ——, *Parallelism and sparse linear equations*, in Sparsity and Its Applications, D. J. Evans, ed., Cambridge, United Kingdom: Cambridge University Press, 1985, pp. 285–301.

[48] P. Sadayappan and V. Visvanathan, *Efficient sparse matrix factorization for circuit simulation on vector supercomputers*, IEEE Trans. Computer-Aided Design Integ. Circ. Sys., 8 (1989), pp. 1276–1285.

[49] Michael A. Saunders, *LUSOL: Sparse LU for $Ax = b$*, 2009. http://stanford.edu/group/SOL/lusol/.

[50] D. Smart and J. White, *Reducing the parallel solution time of sparse circuit matrices using reordered Gaussian elimination and relaxation*, in Proceedings of the IEEE International Symposium Circuits and Systems, 1988.

[51] M.A. Srinivas, *Optimal parallel scheduling of gaussian elimination DAG's*, IEEE Trans. Comput., C-32 (1983), pp. 1109–1117.

[52] A. F. Van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst, *Parallel sparse LU decomposition on a mesh network of transputers*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 853–879.

[53] O. Wing and J. W. Huang, *A computation model of parallel solution of linear equations*, IEEE Trans. Comput., C-29 (1980), pp. 632–638.