SIAM J. Matrix Anal. Appl.
Vol. 41, No. 2, pp. 573–590

DESIGN AND IMPLEMENTATION OF A PARALLEL MARKOWITZ THRESHOLD ALGORITHM[*]

TIMOTHY A. DAVIS[†], IAIN S. DUFF[‡], and STOJCE NAKOV[§]

Abstract. We develop a novel algorithm for the parallel factorization of an unsymmetric sparse matrix using a Markowitz threshold algorithm. Our algorithm uses a significant extension of a maximum independent set algorithm of Luby to select a block of independent pivots in parallel. We then use this block to update the matrix again in parallel. This algorithm does not require any symmetry in the matrix and is particularly efficacious when the matrix is structurally very unsymmetric using metrics defined in the paper. We implement this algorithm using OpenMP and show its performance against other state-of-the-art codes on a standard set of sparse matrix test problems.

Key words. Luby's algorithm, multicore, OpenMP, sparse unsymmetric matrices, parallel pivot selection, sparse factorization

AMS subject classifications. 65F30, 65F50

DOI. 10.1137/19M1245815

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 (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 [26], which is the proportion of off-diagonal entries for which there is a corresponding entry in the transpose, viz.,

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

[*]Received by the editors February 19, 2019; accepted for publication (in revised form) by M. Tůma January 16, 2020; published electronically April 29, 2020.
https://doi.org/10.1137/19M1245815

Funding: The work of the authors was supported by the European Union's Horizon 2020 research and innovation program under NLAFET grant 671633. The work of the first author was supported by National Science Foundation grant CNS-1514406.

[†]Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843-3112 (davis@tamu.edu).
[‡]Computational Mathematics Group, STFC Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, UK (iain.duff@stfc.ac.uk), and CERFACS, Toulouse, France.
[§]Computational Mathematics Group, STFC Rutherford Appleton Laboratory, Didcot, Oxon OX11 0QX, UK (stojce.nakov@stfc.ac.uk).

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 modeling, 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.

Our new algorithms and code are designed for execution on shared memory multicore nodes. Obtaining 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, which 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.

Prior work on parallel sparse LU factorization methods primarily focuses on matrices with a mostly symmetric nonzero pattern, or they restrict the pivoting in some way. There has been far less work on more general factorization schemes. Currently available codes for this case include MA48 and HSL_MA48 [24], UMFPACK [16], LUSOL [48], and KLU [18]. We give a brief summary of what has been done first on left-looking algorithms and then on right-looking.

The left-looking method selects the column ordering prior to factorization [33]. This allows the column elimination tree to be determined a priori. In the column elimination tree [31, 32], nodes on independent branches form independent sets of pivots; George and Ng exploited this property to create a parallel factorization method [29]. Parallel left-looking variants either do not use pivoting during numerical factorization at all [47] or 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, 34, 35, 39, 40, 42, 45, 46, 49, 50, 52]. 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.

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

Several prior methods find independent set of pivots during numerical factorization [1, 3, 19, 37, 51]; 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 [1] and Alaghband and Jordan [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 nondeterministic parallel Markowitz search where each thread searches independently. Candidate pivots are added to the pivot set in a critical section which limits parallelism. Van der Stappen, Bisseling, and van de Vorst [51] and Koster and Bisseling [37] 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 [30], Gallivan et al. [27], and Gallivan, Marsolf, and Wijshoff [28] 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 [25] 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, 5, 6, 7, 8, 9, 10, 14, 21, 22, 36, 38, 41]. 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.

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 time. 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 [44] 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 were 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

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

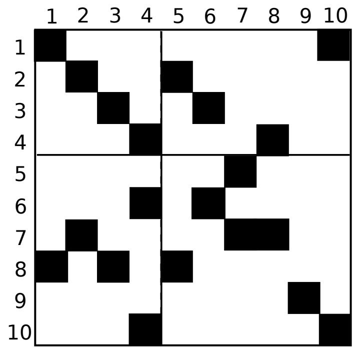

FIG. 1. *Block of independent pivots.*

where u is a threshold parameter $0 < u \leq 1.0$. That is, 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, which 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 1, where the independent pivots have been permuted to the top left-hand corner of the matrix. We could choose something else as a block of pivots as long it is easy to factorize. We have opted for the independent set because it does not introduce any fill-in in the factors, so we have more efficient parallelism. 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].

In sequential codes like MA48 [24], we select the eligible pivot which has the minimum Markowitz count, as defined in (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 [43] 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 select 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. First, if these pivots are not handled properly, they can cause the unsymmetric Luby search phase to find a very small pivot set. That is, 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. Second, 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 in the active submatrix as the factorization progresses. We thus precede the unsymmetric Luby 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-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-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, 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 [43] 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 pseudocode 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 neighbors (line 7). This will guarantee that the nodes are independent. These nodes are added to the independent set I (line 8). Together with their neighbors, they are removed from the original graph (lines 9–11). 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.

Algorithm 1 Luby's algorithm.

```
1: **Input** G = (V,E) an undirected graph
2: **Output** $I \subseteq V$, a MIS
3: $I \leftarrow \varnothing$
4: $G' = (V', E') \leftarrow G = (V, E)$
5: **while** $V' \neq \varnothing$ **do**
6:     *assign random score to each node in* $V'$
7:     $I' \leftarrow$ *nodes having highest score among their neighbors in* $G'$
8:     $I \leftarrow I \cup I'$
9:     $Y \leftarrow I' \cup N(I')$
10:    $V' \leftarrow V' - Y$
11:    $G' = G'(V')$
12: **end while**
```

We now extend this algorithm to directed graphs. The reason for this is that an unsymmetric matrix is represented as a directed graph. For an $n \times n$ 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. Note that the graph may have self-loops. Specifically, each diagonal nonzero entry a_{ii} corresponds to a self-loop in the directed graph. We first introduce the following terms:

- $SRC(\varepsilon_1, \varepsilon_2, \ldots)$, the source nodes for the edges $\varepsilon_1, \varepsilon_2, \ldots$;
- $DST(\varepsilon_1, \varepsilon_2, \ldots)$, 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 the nonzeros of their rows and columns, respectively. With these notations, a set of edges representing independent pivots I, as presented in Figure 1, is defined as

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

The first part of (5.1) is illustrated in Figure 2 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.

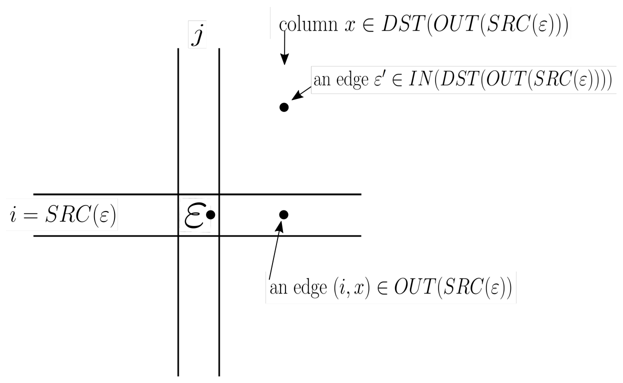
j
column $x \in DST(OUT(SRC(\varepsilon)))$
an edge $\varepsilon' \in IN(DST(OUT(SRC(\varepsilon))))$
$i = SRC(\varepsilon)$
ε
an edge $(i, x) \in OUT(SRC(\varepsilon))$

FIG. 2. *Independent set condition.*

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 (5.1) is similar; it states the transpose of the condition illustrated in Figure 2.

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 (5.1).

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

```
1:  **Input** G = (V,E) a directed graph
2:  **Output** $I \subseteq E$, *an independent set of pivot edges in G*
3:  $I \leftarrow \varnothing$
4:  $E' \leftarrow E$
5:  **while** $E' \neq \varnothing$ **do**
6:    $I'' = \{\forall \nu \in V,\ IN(\nu) \neq \emptyset,$ *choose an edge* $\varepsilon \mid$
              $\varepsilon' \in IN(\nu)$ *and assign score to* $\varepsilon\}$
7:    $I' \leftarrow I''$
8:    **for all** $\varepsilon \in I''$ **do**
9:      **for all** $\varepsilon' \in IN(DST(OUT(SRC(\varepsilon))))$ **do**
10:       **if** $(\varepsilon' \in I'')$ **and** $(\varepsilon \neq \varepsilon')$ **then**
11:         **if** $SCORE(\varepsilon) > SCORE(\varepsilon')$ **then**
12:           *remove* $\varepsilon'$ *from* $I'$
13:         **else**
14:           *remove* $\varepsilon$ *from* $I'$
15:         **end if**
16:       **end if**
17:     **end for**
18:   **end for**
19:   $I \leftarrow I \cup I'$
20:   $Y_{dst} \leftarrow \{\forall \varepsilon \in I', OUT(SRC(IN(DST(\varepsilon))))\}$
21:   $Y_{src} \leftarrow \{\forall \varepsilon \in I', IN(DST(OUT(SRC(\varepsilon))))\}$
22:   $Y \leftarrow Y_{src} \cup Y_{dst} \cup I'$
23:   $E' \leftarrow E' - Y$
24: **end while**
```

For each node, we pick one of the incoming edges and assign a score to it (line 6). These edges will be called the chosen edges. Instead of relying on a random score as in Luby's algorithm, we assign a semirandomized 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 (5.1) is satisfied for all the edges 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 has 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 (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 (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 and the Schur update of the trailing matrix.

6. Implementation details.

Our algorithm could be considered as an iterative process repeated until the matrix is factorized or we switch to dense factorization. Each iteration can be divided into two 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 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 (see Figure 1). The second step consists of a parallel Schur complement update to the active matrix. These two steps are repeated 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 once a row or a column consumes its free space, 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 algorithm, a real array is used for storing the randomized 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, and two arrays for the remaining rows and columns in the active matrix. Two arrays for computing the combined structure of the rows and columns of the set of pivots are used as well. More details about these two arrays are given below. Furthermore, we have three arrays of size n per thread: two integer arrays and one real array. Their use will be explained in the following. We now present implementation details for the unsymmetric Luby algorithm.

6.1. Implementation of the unsymmetric Luby algorithm.

As we explained in section 4, we first search for singletons in the active matrix. This is done by checking if a row or a column exists with only 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 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 randomized score. The pivot search algorithm can be divided into two phases: the initialization phase and the search phase. The initialization phase is done just once for each successive active matrix, while the search phase can be repeated until all the columns are discarded.

Initialization. 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 active matrix but are 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 iterations, 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 (explained below) and thus perform the partitioning of these columns only during the update of the active matrix. 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. This phase is divided into the following six steps. Each step is performed fully in parallel, and a synchronization 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 initialization phase are split into subsets, and each thread handles the columns within a subset. To each *potential* pivot, a score is assigned equal to

$$tmp = rand(0,1)$$

$$score_{ij} = tmp \times \left(1 - \frac{Mark_{ij}}{\alpha_{Mark} \times Best_{Mark}}\right) + (1 - tmp) \times \frac{|a_{ij}|}{max_j}, \tag{6.1}$$

  where max_i corresponds to the largest absolute value in column j 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

(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 the row of 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). In the case where the two scores are the same, the one with lower column index is chosen.
- Step 3: All threads now reexamine 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 and counts the number of them. We assume that we have p threads and that thread t $(1 \leq t \leq p)$ has found k_t chosen pivots.
- Step 4: Next, we construct a global list of the rows and columns of the *chosen* pivots. First, we compute the cumulative sum of $k_1, k_2, \ldots, k_p$.
- Step 5: The cumulative sum provides each thread with its positions in the global list of pivots. Each thread copies its set of *chosen* pivots into the global list of pivots and updates the array with the inverse permutations.
- Step 6: Next, we need to discard all the *potential* pivots that are compatible with the chosen pivots (lines 20–23) for the next iteration, i.e., *potential* pivots that do not satisfy (5.1). In terms of a matrix point of view, this means that a *potential* pivot is discarded if there is an entry in its row or column in the combined *chosen* pivots row or column structure. For example, in Figure 1, if the *potential* pivot in column 7 is the entry a_{77}, then it is discarded because of the entry a_{72}. On the other hand, if it was the entry a_{57}, then it is not discarded since in row 5 and column 7, there are no entries in the combined structure of the rows and columns of the *chosen* pivots. During this step, only the *chosen* pivots are involved, so we can redistribute work among the threads to give an equal amount of work to each thread. In order to perform this step, we first compute the combined structure of all column indices in the *chosen* pivotal rows and the combined structure of all row indices in the *chosen* pivotal columns. Then we discard all remaining *potential* pivots for which their column or row index is in the combined structure. 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 combined structure for the *chosen* pivotal rows and columns of the final set will determine what rows and columns are active during the Schur update. This information is then used during the Schur complement update.

6.2. Schur complement update.

Updating the column-form. During this phase, the factor L is first updated. All pivotal columns are removed from the column-form of the active submatrix and are placed in L concurrently by all threads. 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 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 organized into two NUMA islands with 14 cores in each. The nodes are connected with an FDR Infiniband Network. Each CPU core has a 32 KB L1 data cache, a 32 KB L1 instruction cache, and a 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 14 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 1. In the rest of this section, we first investigate the behavior of the ParSHUM solver and then present a comparison with four state-of-the-art solvers: MA48 [24], UMFPACK [16], SuperLU_MT [20], and NICSLU [13].

[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.

Table 1
Statistics for the matrices that are used in this study.

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

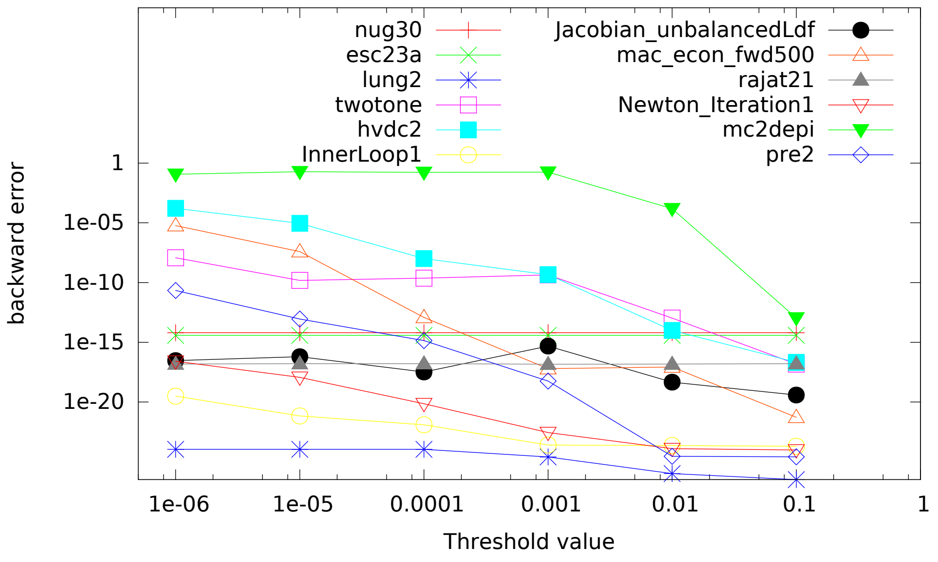

Fig. 3. *The impact of the threshold parameter, u, on the backward error for* ParSHUM.

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 3, we present the impact on the backward error of the threshold parameter u in (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 3). However, when the threshold is increased, we obtain a backward error of at most 10^{-13} for all the matrices.

Another parameter in our algorithm is the Markowitz threshold. By relaxing this parameter, we allow pivots with higher Markowitz count to be chosen as pivots.

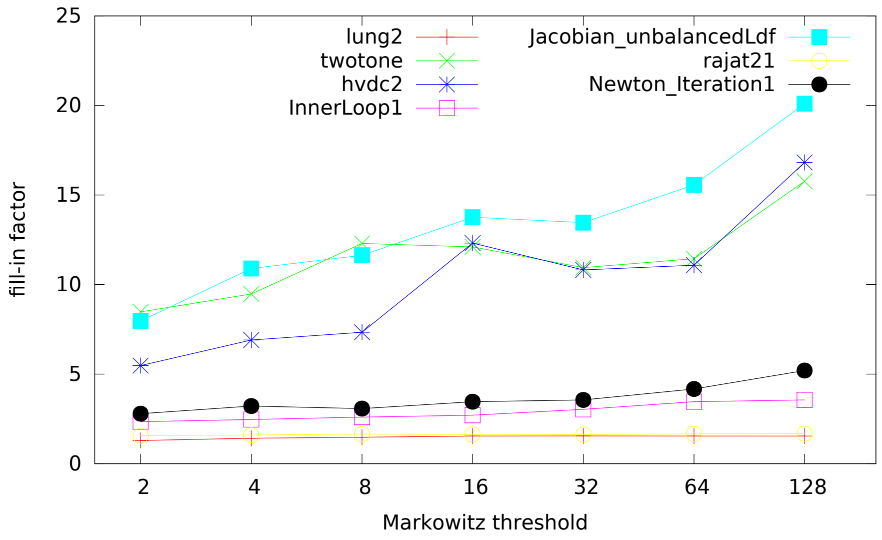

Fig. 4. *The impact of the Markowitz threshold on the fill-in factor for* ParSHUM*. The results for* nug30*,* esc32a*,* mac_econ_fwd500*,* mc2depi*, and* pre2 *are not shown here because their high values dominate the graph. However, they follow a similar behavior to the other matrices.*

These pivots could potentially increase the fill-in in the factors. For a given matrix A, ParSHUM factorizes the matrix as

$$PAQ = LU,$$

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.,

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

We observe the effect of this parameter on the fill-in for each matrix in Figure 4. 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 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 solvers. The Markowitz threshold makes sense only for the ParSHUM solver, while the density switch makes sense for the ParSHUM and the MA48 solvers. The parameters are presented in Table 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}.

Next, we investigate the parallel behavior of our algorithm. In Figure 5, we present the execution time for two matrices, mc2depi and twotone, when the number of threads is increased. The fill-in factors for the mc2depi and twotone matrices are 302

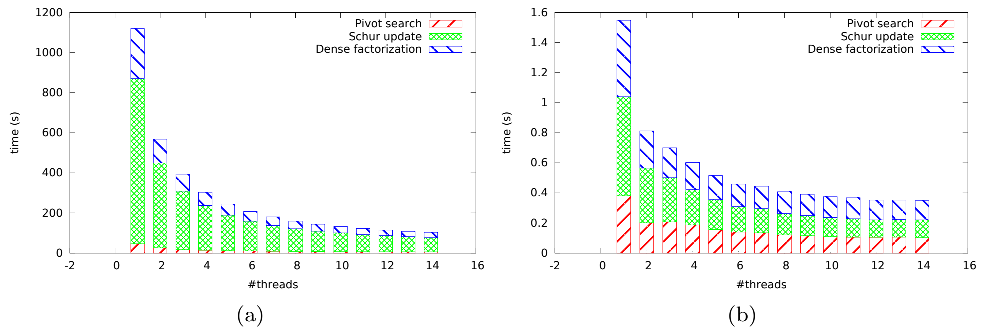

FIG. 5. *Multithreaded results for our algorithm. Results for the* mc2depi *matrix are presented on the left and for the* twotone *matrix on the right.*

TABLE 2
The parameters and their optimal values for the results in Figure 5 *and Table* 3 *are shown in columns* 2–4 *(*ParSHUM*),* 5–6 *(*MA48*), and* 7 *(*UMFPACK*). The parameters u and α_{Mark} are defined in section* 3*, and* Φ *is the density at which the switch to full code is made. "—" means that the tests have failed.*

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

and 7.37, respectively. The parallel behavior 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 rows and columns of larger size, which will lead to more efficient execution. Additionally, more parallelism is available. For instance, for the mc2depi matrix, a speedup of 11.1 is achieved when 14 threads are used. On the other hand, for the twotone matrix, a speedup of 4.7 is obtained mainly due to lower granularity and limited parallelism.

In Table 3, the best execution times and fill-in factors for ParSHUM, MA48, UMFPACK, SuperLU_MT, and NICSLU are presented. When the rajat21 matrix is used with the SuperLU_MT solvers, "—" means that the tests have failed (and have been reported to the developing team). We obtain the lowest execution times with ParSHUM for all the matrices except the hvdc2, Jacobian_unbalancedLdf, mac_econ_fwd500, and mc2depi matrices, for which the UMFPACK and the SuperLU_MT solvers yield 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 solvers but manages to obtain better performance due to its efficient parallel execution. For instance, for the hvdc2 matrix, NICSLU has the lowest execution time of 0.33 seconds, followed by the ParSHUM and the UMFPACK solvers with an execution time of 0.38 seconds,

TABLE 3
The execution times and the fill-in factors for ParSHUM, MA48, and UMFPACK solvers. Best results are highlighted in bold. "—" means that the tests have failed.

	ParSHUM		MA48		UMFPACK		SuperLU_MT		NICSLU	
Matrix	time	fill-in	time	fill-in	time	fill-in	time	fill-in	time	fill-in
nug30	**0.71**	142.	31.0	450.	48.15	463.	735	1994	119	513
esc32a	**0.46**	70.9	7.54	154.	71.0	341.	1808	2430	98.8	354
lung2	**0.04**	1.48	0.37	2.55	0.10	1.44	0.12	1.53	0.08	1.00
twotone	**0.35**	7.37	4.00	18.8	0.49	5.45	0.70	17.2	4.65	19.6
hvdc2	0.38	6.91	0.42	1.82	0.38	2.06	0.40	2.87	**0.33**	1.89
InnerLoop1	**0.13**	2.71	0.22	1.99	0.30	2.48	0.28	2.91	0.37	2.19
Jacobian_unbalancedLdf	0.99	10.9	4.21	5.40	0.74	3.88	**0.44**	6.23	1.59	3.1
mac_econ_fwd500	33.5	450.	4137.	483.	4.62	57.5	**3.99**	83.2	5.80	33.4
rajat21	**0.11**	1.64	9.74	1.26	44.5	1.89	—	—	0.91	1.34
Newton_Iteration1	**0.46**	3.46	1.23	2.55	1.00	2.55	0.73	4.18	1.70	2.68
mc2depi	104.	302.	1816.	144.	4.36	38.6	**2.21**	56.88	6.43	22.2
pre2	**13.0**	57.0	124.1	44.8	25.8	32.3	44.6	61.9	19.5	14.99

but for this matrix, the ParSHUM solver creates a fill-in factor that is more than three times larger in comparison with the other solvers. On the other hand, for the mac_econ_fwd500 matrix, for example, a fill-in of 450 is obtained with the ParSHUM solver, resulting in an execution time of 33.5 seconds, while the UMFPACK solver has a fill-in of 57.5, resulting in an execution time of 4.62. However, for the nug30 and esc32a matrices, a considerably lower fill-in is obtained with the ParSHUM solver, resulting in a much lower execution time.

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 [43]. We have also developed a robust library code called ParSHUM, which 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.[5] The performance results of our method show that it obtains excellent performance for highly unsymmetric matrices. It is typically much faster than two well-established state-of-the-art packages on a multicore system, even though it sometimes produces factorizations with more fill-in.

One possible direction for future work would be to exploit distributed systems. For this, we are planning to first partition the matrix into singly bordered block diagonal form and then factorize each diagonal block using ParSHUM. Also, the ParSHUM algorithm is clearly bandwidth dependent and offers a lot of parallelism, which makes it a great for candidate for GPUs, which have large bandwidth.

Acknowledgments. 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 Comput., 11 (1989), pp. 201–221.

[2] G. ALAGHBAND, *Parallel sparse matrix solution and performance*, Parallel Comput., 21 (1995), pp. 1407–1430, https://doi.org/10.1016/0167-8191(95)00029-N.

[5]urlhttps://github.com/NLAFET/ParSHUM.

[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*, Comput. Methods Appl. Mech. Engrg., 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 Comput., 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 Comput., 29 (2003), pp. 1645–1668, https://doi.org/10.1016/j.parco.2003.05.010.

[8] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel Comput., 32 (2006), pp. 136–156, https://doi.org/10.1016/j.parco.2005.07.004.

[9] P. R. Amestoy, J.-Y. L'Excellent, F.-H. Rouet, and W. M. Sid-Lakhdar, *Modeling* 1*D distributed-memory dense kernels for an asynchronous multifrontal sparse solver*, in 11th International Meeting High-Performance Computing for Computational Science, VECPAR 2014, Eugene, OR, 2014.

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

[11] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Comput., 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, Monticello, IL, 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. Syst., 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, https://doi.org/10.1137/S1064827594276412.

[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] T. A. Davis and I. 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. Software, 38 (2011), pp. 1:1–1:25, https://doi.org/10.1145/2049662.2049663.

[18] T. A. Davis and E. P. Natarajan, *Algorithm* 907: *KLU, A direct sparse solver for circuit simulation problems*, ACM Trans. Math. Software, 36 (2010), 36, https://doi.acm.org/10.1145/1824801.1824814.

[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, https://doi.org/10.1137/S0895479897317685.

[21] I. S. Duff, *The parallel solution of sparse linear equations*, in CONPAR 86, Conference 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., Springer-Verlag, Berlin, 1986, pp. 18–24, https://doi.org/10.1007/3-540-16811-7.

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

[23] I. S. Duff, *Parallel algorithms for sparse matrix solution*, in Parallel Computing. Methods, Algorithms, and Applications, D. J. Evans and C. Sutti, eds., Adam Hilger, Bristol, UK, 1989, pp. 73–82, https://books.google.com/books?id=2z5ipEZQOZEC.

[24] I. S. Duff and J. K. Reid, *The design of MA*48*, a code for the direct solution of sparse unsymmetric linear systems of equations*, ACM Trans. Math. Software, 22 (1996), pp. 187–226.

[25] I. S. Duff and J. A. Scott, *A parallel direct solver for large sparse highly unsymmetric linear systems*, ACM Trans. Math. Software, 30 (2004), pp. 95–117, https://doi.org/10.1145/992200.992201.

[26] 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 J. Sci. Statist. Comput., 7 (1987), pp. 600–624.

[27] K. A. Gallivan, P. C. Hansen, T. 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, https://doi.org/10.1007/BF02238079.

[28] K. A. Gallivan, B. A. Marsolf, and H. A. G. Wijshoff, *Solving large nonsymmetric sparse linear systems using MCSPARSE*, Parallel Comput., 22 (1996), pp. 1291–1333, https://doi.org/10.1016/S0167-8191(96)00047-6.

[29] A. George and E. G. Ng, *Parallel sparse Gaussian elimination with partial pivoting*, Ann. Oper. Res., 22 (1990), pp. 219–240, https://doi.org/10.1007/BF02023054.

[30] J. P. Geschiere and H. A. G. Wijshoff, *Exploiting large grain parallelism in a sparse direct linear system solver*, Parallel Comput., 21 (1995), pp. 1339–1364, https://doi.org/10.1016/0167-8191(95)00024-I.

[31] 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, https://doi.org/10.1137/0614024.

[32] 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., IMA Vol. Appl. Math. 56, Springer-Verlag, New York, 1993, pp. 107–139.

[33] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Comput., 9 (1988), pp. 862–874, https://doi.org/10.1137/0909058.

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

[35] 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.

[36] K. Kim and V. Eijkhout, *A parallel sparse direct solver via hierarchical DAG scheduling*, ACM Trans. Math. Software, 41 (2014), pp. 3:1–3:27.

[37] J. Koster and R. H. Bisseling, *An improved algorithm for parallel sparse LU decomposition on a distributed-memory multiprocessor*, in Proceedings of the Fifth SIAM Conference on Applied Linear Algebra, Snowbird, UT, 1994, SIAM, Philadelphia, pp. 397–401.

[38] 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., IMA Vol. Appl. Math. 56, Springer-Verlag, New York, 1993, pp. 211–228.

[39] M. Leuze, *Independent set orderings for parallel matrix factorization by Gaussian elimination*, Parallel Comput., 10 (1989), pp. 177–191, https://doi.org/10.1016/0167-8191(89)90016-1.

[40] 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, https://doi.org/10.1137/0910070.

[41] J.-Y. L'Excellent and W. M. Sid-Lakhdar, *Introduction of shared-memory parallelism in a distributed-memory multifrontal solver*, Parallel Comput., 40 (2014), pp. 34–46, https://doi.org/10.1016/j.parco.2014.02.003.

[42] J. W. H. Liu and A. Mirzaian, *A linear reordering algorithm for parallel pivoting of chordal graphs*, SIAM J. Discrete Math., 2 (1989), pp. 100–107, https://doi.org/10.1137/0402011.

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

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

[45] F. J. Peters, *Parallel pivoting algorithms for sparse symmetric matrices*, Parallel Comput., 1 (1984), pp. 99–110, https://doi.org/10.1016/S0167-8191(84)90446-0.

[46] F. J. Peters, *Parallelism and sparse linear equations*, in Sparsity and Its Applications, D. J. Evans, ed., Cambridge University Press, Cambridge, 1985, pp. 285–301.

[47] P. Sadayappan and V. Visvanathan, *Efficient sparse matrix factorization for circuit simulation on vector supercomputers*, IEEE Trans. Computer-Aided Design Integ. Circ. Syst., 8 (1989), pp. 1276–1285, https://doi.org/10.1109/43.44508.

[48] M. A. Saunders, *LUSOL: Sparse LU for $Ax = b$*, 2009, https://stanford.edu/group/SOL/lusol/.

[49] 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, Espoo, Finland, 1988, https://doi.org/10.1109/ISCAS.1988.15004.

[50] M. Srinivas, *Optimal parallel scheduling of Gaussian elimination DAG's*, IEEE Trans. Comput., C-32 (1983), pp. 1109–1117, https://doi.org/10.1109/TC.1983.1676171.

[51] 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, https://doi.org/10.1137/0614059.

[52] O. Wing and J. W. Huang, *A computation model of parallel solution of linear equations*, IEEE Trans. Comput., C-29 (1980), pp. 632–638, https://doi.org/10.1109/TC.1980.1675634.