Fast and effective reordering of columns within supernodes using partition refinement*

Mathias Jacquelin † Esmond G. Ng † Barry W. Peyton ‡

Abstract

In this paper, we consider the problem of computing a triangular factorization of a sparse symmetric matrix using Gaussian elimination. We assume that the sparse matrix has been permuted using a fill-reducing ordering algorithm. When the matrix is symmetric positive definite, the sparsity structure of the triangular factor can be determined once the fill-reducing ordering has been computed. Thus, an efficient numerical factorization scheme can be designed so that only the nonzero entries are stored and operated on. On modern architectures, the positions of the nonzero entries in the triangular factor play a big role in determining the efficiency. It is desirable to have dense blocks in the factor so that the computation can be cast in terms of level-3 BLAS as much as possible. On architectures with GPUs, for example, it is also desirable for these dense blocks to be as large as possible in order to reduce the times to transfer data between the main CPU and the GPUs. We address the problem of locally refining the ordering so that the number of dense blocks is reduced and the sizes of these dense blocks are increased in the triangular factor.

1 Introduction

In this paper, we address a combinatorial problem that arises in the solution of a large, sparse, symmetric, positive definite linear system of equations using Cholesky factorization. We assume that an n by n sparse, symmetric, positive definite matrix A has been reordered to reduce fill by some ordering heuristic, such as minimum degree [7] or nested dissection [6]. It is well known that in most practical problem settings, the columns of the Cholesky factor L are partitioned into sets of columns sharing the same sparsity structure. Each such set of columns corresponds to a so-called *supernode* [4, 15].

Now, columns within the same supernode are numbered consecutively. Furthermore, columns within the same supernode can be numbered in any order without changing the number of nonzeros in the factor matrix. On certain machines—particularly those using GPUs—it is desirable to exploit this freedom to reorder columns within supernodes in order to improve factorization efficiency. Loosely speaking, what is desired here is to reorder the columns within supernodes so that the nonzeros of the factor matrix are organized within fewer and larger dense blocks. The number of nonzeros in the factor matrix remains the same.

At this point, we wish to be more precise about what we mean here by dense blocks and our objective of enlarging the dense blocks. Let J be the set of all column indices of a supernode; i.e., $J = \{j_{\min}, j_{\min} + 1, \ldots, j_{\max}\}$. We will also refer to the supernode itself as supernode J. Let $I = \{i_{\min}, i_{\min} + 1, \ldots, i_{\max}\}$ be a supernode with columns numbered higher than those of J. We use the notation $J < I$ to mean that $j_{\max} < i_{\min}$. The submatrix of the Cholesky factor L with rows from I and columns from J will be written

$$L_{I,J} := \{L_{i,j} : i \in I \text{ and } j \in J\}.$$

When we say that columns within a supernode J share the same structure, we mean that for each $i \in I$, we have either $L_{i,j} \neq 0$ for all $j \in J$, or we have $L_{i,j} = 0$ for all $j \in J$. A *block* in $L_{I,J}$ will be defined as a submatrix $L_{I',J}$ of $L_{I,J}$ with $I' = \{i', i' + 1, \ldots, i' + s\} \subseteq I$ such that

1. $L_{i,j} \neq 0$ for $i \in I'$ and $j \in J$,
2. $i' = i_{\min}$, or $L_{i'-1,j} = 0$ for $j \in J$, and
3. $i' + s = i_{\max}$, or $L_{i'+s+1,j} = 0$ for $j \in J$.

We will call such a block an (I, J)-*block*. Note that an (I, J)-block is defined in such a way that it is

*This work was supported in part by the Scientific Discovery through Advanced Computing (SciDAC) Program of the U.S. Department of Energy (DOE), Office of Science, Advanced Scientific Computing Research at Lawrence Berkeley National Laboratory (LBNL) through Contract No. DE-AC02-05CH11231. Computational resources were provided by the DOE National Energy Research Scientific Computing Center (NERSC), located at LBNL.

†Applied Mathematics Department, Lawrence Berkeley National Laboratory, (mjacquelin|egng@lbl.gov)

‡Dalton State College, (bpeyton@daltonstate.edu)

maximal within the encompassing submatrix $L_{I,J}$; that is, the block cannot be enlarged by including the row immediately above the first row of the block or the row immediately below the last row of the block. Let $b(I,J)$ be the number of blocks within $L_{I,J}$; i.e., the number of (I,J)-blocks. Our central problem can be stated as follows: we would like to reorder the columns of supernode I to minimize the total number of (I,J)-blocks over all supernodes $J < I$. More precisely, we would like to reorder the columns of supernode I to minimize

$$\sum_{J<I} b(I,J). \tag{1.1}$$

Reordering each supernode is a separate and independent reordering problem.

Two groups of researchers have worked on the problem of reordering columns within supernodes. Pichon, Faverge, Ramet, and Roman [17] worked directly on the optimization problem stated above. They were able to formulate the problem as a traveling salesman problem (TSP), which is an NP-hard problem [11]. Because there are heuristics to solve their instance of TSP to within a factor of two [19], Pichon et al. [17] were able to reduce the total number of blocks to within a factor of two of optimal. The problem with their approach is its cost in time. The worst-case time complexity can be equal to that of the numerical factorization. The run-times reported by Pichon et al. [17] did, with some frequency, exceed the time required to compute the initial fill-reducing ordering. The strength of their method, however, is that it is very effective at reducing the total number of blocks.

Hogg, Reid, and Scott [9] proposed a simple algorithm for reordering columns within supernodes that requires very little run-time. In their algorithm, each submatrix $L_{I,J}$ is guaranteed to have a single block for a limited and carefully selected subset of supernodes $J < I$ that can be handled very easily. Its strength is that it costs very little time to compute; its weakness is that it is not very effective at reducing the total number of blocks [17].

In this paper, we introduce an algorithm for reordering columns within supernodes based on partition refinement. Partition refinement has been an extremely useful tool in many graph algorithms [8, 16, 18]. Our approach using partition refinement is much more comprehensive than the approach used in Hogg, Reid and Scott [9], but it is much less involved than the TSP modeling used in Pichon et al. [17]. We will show that the quality of the reorderings obtained using our method is nearly as good as the quality of the more expensive reorderings obtained using the TSP-based method of Pichon et al. [17]. We will also show that the time required by our method is linear in the number of subscripts used to represent the sparsity structure of the Cholesky factor (one list of row indices for each supernode).

It is well known that the symbolic factorization routines used between the fill-reducing reordering step and the numerical factorization step typically take significantly less time than either of these two steps. In this paper, we have provided a reordering algorithm for supernodes whose time requirement is much the same as that of a symbolic factorization algorithm, but whose ability to reduce the number of blocks approaches that of the powerful TSP-based approach used by Pichon et al. [17].

Our paper is organized as follows. We give some more background material and detailed notation in Section 2. Our algorithm based on partition refinement is given in Section 3. We present test results in Section 4 to support our contention that our algorithm requires very little time to run, and yet it produces high-quality reorderings of columns within supernodes. Finally, Section 5 gives our concluding remarks.

2 Background and notation

In our discussion, we will be solving a linear system $Ax = b$, where A is a large n by n sparse, symmetric, positive definite matrix. In order to solve the linear system, we will compute the Cholesky factorization of A ($A = LL^T$), where L is a lower triangular matrix. It is well known that the matrix A suffers fill during the factorization; that is, some zero entries in A become nonzero entries in the Cholesky factor L. It is also well known that the amount of fill incurred by the Cholesky factor depends on how the rows and columns of A are permuted. Throughout our discussion we assume that the rows and columns of A already have been symmetrically permuted by some fill-reducing ordering heuristic, such as minimum degree [2, 7, 13, 21] or nested dissection [6, 12].

To facilitate our discussion, undirected graphs are useful in modeling the sparsity structure of the relevant matrices. The sparsity structure of A is given by the undirected graph $G = (V,E)$, with vertex set $V = \{1,2,3,\ldots,n\}$ and edge set $E = \{\{i,j\} : A_{i,j} \neq 0 \text{ and } i \neq j\}$. The sparsity structure of $L + L^T$ is given by the undirected graph $G^+ = (V,E^+)$, with vertex set $V = \{1,2,3,\ldots,n\}$ and edge set $E^+ = \{\{i,j\} : L_{i,j} + L^T_{i,j} \neq 0 \text{ and } i \neq j\}$. The graph G^+ is known as the *fill graph*. When accidental numerical cancellation is ignored, it is known that $E \subseteq E^+$, and the edges of $E^+ \setminus E$ are precisely the fill edges needed to represent the nonzeros in L that

were originally zeros in A.

In a given graph $G' = (V', E')$, the set of neighbors of a vertex $i \in V'$ (the *adjacency set* of i) is given by

$$\mathrm{adj}_{G'}(i) := \{j : \{i, j\} \in E'\}.$$

The *higher adjacency set* of i is the set of neighbors greater than i; that is,

$$\mathrm{hadj}_{G'}(i) := \{j : \{i, j\} \in E' \text{ and } j > i\}.$$

Let $K \subseteq V'$. Let $k_{\max} \in K$ be the maximum vertex in K; i.e., $k_{\max}$ has the largest index. We define the higher adjacency set of K by

$$\mathrm{hadj}_{G'}(K) := \{j : j \in V' \setminus \{1, 2, \ldots, k_{\max}\} \text{ and } j \in \mathrm{adj}_{G'}(i) \text{ for some } i \in K\}.$$

The set $K \subseteq V'$ is a *clique* in G' if the vertices of K are pairwise adjacent in G'.

The algorithms discussed in this paper assume that a supernode partition has been computed during the symbolic factorization (analyze) phase of the solution process. An algorithm for computing the fundamental supernode partition can be found in Liu, Ng, and Peyton [15]. In our discussion, we have no restriction on the supernode partition, except the following. Let the vertices of a supernode be numbered consecutively. For $K = \{k_{\min}, k_{\min} + 1, \ldots, k_{\max}\}$ to qualify as a supernode, the vertices of K must satisfy two further conditions:

1. The vertices of K form a clique in G^+, and
2. $\mathrm{hadj}_{G^+}(k) \setminus K = \mathrm{hadj}_{G^+}(k') \setminus K$ for every pair of vertices $k, k' \in K$.

Note that $\mathrm{hadj}_{G^+}(K) = \mathrm{hadj}_{G^+}(k) \setminus K$ for every vertex $k \in K$, whenever K is a supernode. We also wish to point out that there is nothing to prevent the accomodation of relaxed supernodes, as long as the stored logical zeros are within the sparsity structure defined by the higher adjacency sets of the supernodes.

Let us return to our definition of blocks, but do so now using the graph notation introduced above. Let $J = \{j_{\min}, j_{\min} + 1, \ldots, j_{\max}\}$ and $I = \{i_{\min}, i_{\min} + 1, \ldots, i_{\max}\}$ be supernodes with $J < I$. Let $I' = \{i', i' + 1, \ldots, i' + s\} \subseteq I$. It is the case then that I' contains the row indices of an (I, J)-block if

1. $I' \subseteq \mathrm{hadj}_{G^+}(J) \cap I$,
2. $i' = i_{\min}$ or $i' - 1 \notin \mathrm{hadj}_{G^+}(J)$, and
3. $i' + s = i_{\max}$ or $i' + s + 1 \notin \mathrm{hadj}_{G^+}(J)$.

Elimination trees [14] have proved very useful in sparse matrix computations. For any vertex i, the parent of i in the elimination tree is given by

$$\mathrm{parent}(i) := \min\{j : j \in \mathrm{hadj}_{G^+}(i)\},$$

and if $\mathrm{hadj}_{G^+}(i) = \emptyset$, then $\mathrm{parent}(i) :=$ **null**. For any vertex i, we will define $\mathrm{snode}(i) := I$ if i belongs to supernode I. We then can define the *supernodal elimination tree* as follows. In the supernodal elimination tree, the parent of supernode I is defined by $\mathrm{parent}(I) := \mathrm{snode}(j)$, where

$$j = \min\{k : k \in \mathrm{hadj}_{G^+}(I)\}.$$

When $\mathrm{hadj}_{G^+}(I) = \emptyset$, we have $\mathrm{parent}(I) :=$ **null**.

3 Our partition refinement algorithm

Partition refinement has proven to be a very useful tool in a number of graph algorithms. The reader will find this tool used in a variety of ways in the following references [8, 16, 18]. In our paper, we wish to use partition refinement to produce high-quality reorderings of the columns within each supernode.

We begin by giving a brief high-level description of our algorithm. The algorithm will maintain an ordered partition $\mathcal{P}$ of the vertex set V, and this partition will be refined during successive steps of the algorithm.

Let $K_1, K_2, \ldots, K_r$ be the supernodes passed into the algorithm, listed in elimination order. During the initialization phase, the algorithm assigns $S_i := K_i$ for $i := 1, 2, \ldots, r$. The ordered partition is initialized as follows:

$$\mathcal{P} := (S_1, S_2, \ldots, S_r),$$

which is essentially a list of the supernode sets in elimination order. The algorithm processes the higher adjacency sets $\mathrm{hadj}_{G^+}(K_s)$ in reverse order, that is, in the order $s := r, r - 1, \ldots, 1$. When $\mathrm{hadj}_{G^+}(K_s)$ is processed, each set S in the current ordered partition $\mathcal{P}$ for which $S \cap \mathrm{hadj}_{G^+}(K_s) \neq \emptyset$ is partitioned (if possible) into two non-empty sets

$$S' := S \cap \mathrm{hadj}_{G^+}(K_s)$$

and

$$S'' := S \setminus \mathrm{hadj}_{G^+}(K_s).$$

If both S' and S'' are non-empty, then the set S is removed from its position in $\mathcal{P}$, and the pair of sets S' and S'' are inserted side by side into the position in $\mathcal{P}$ formerly occupied by S. Upon completing this refinement process for all of the higher adjacency sets, the algorithm reorders the vertices *consistent with* the final ordered partition $\mathcal{P}$. This simply means that any

vertex i is ordered before vertex j if the set to which i belongs lies in front of the set to which j belongs in the final ordered partition $\mathcal{P}$.

It is easy to give some motivation for using partition refinement in this way on our problem of reordering columns within supernodes. Assume that the higher adjacency set $\text{hadj}_{G^+}(K_s)$ is being processed, and assume, for some set S currently in the ordered partition $\mathcal{P}$, that we have

$$S' := S \cap \text{hadj}_{G^+}(K_s) \neq \emptyset$$

and

$$S'' := S \setminus \text{hadj}_{G^+}(K_s) \neq \emptyset.$$

Assume also that $S \subseteq K_k > K_s$. Since sets are only refined in the algorithm, the set S will be numbered consecutively in the final ordering produced at the end of the algorithm. Since both S' and S'' are non-empty, *at least one* (K_k, K_s)-block will begin or end between two consecutive vertices in $S \subseteq K_k$. By ordering both the vertices of S' consecutively and the vertices of S'' consecutively, our algorithm ensures that there will be *only one* (K_k, K_s)-block that begins or ends beween two consecutive vertices in $S \subseteq K_k$. The discussion here should illustrate at least one way that our algorithm can be viewed as a greedy heuristic.

3.1 The detailed algorithm. Now, we wish to give the reader a more detailed description of our algorithm, primarily for two reasons. First, more detail is probably required in order to enable the reader to implement the algorithm so that it runs efficiently. Second, more detail is needed in order to support our future claim that the algorithm can be implemented so that it runs in time proportional to the number of subscripts needed to represent the sparsity structure of L; that is, it requires $O(n + \sum_{t=1}^{r} |\text{hadj}_{G^+}(K_t)|)$ time. The detailed description of our algorithm is given in Algorithm 1; our discussion of the algorithm follows.

Input into the algorithm are the vertices $V = \{1, 2, \ldots, n\}$, the supernodes $K_1, K_2, \ldots, K_r$, and the higher adjacency sets $\text{hadj}_{G^+}(K_s)$ for $s \in \{1, 2, \ldots, r\}$. The algorithm maintains a collection of sets S_t, where $1 \leq t \leq n$. The sets $S_1, S_2, \ldots, S_r$ are initialized to the sets $K_1, K_2, \ldots, K_r$, respectively, and the initial ordered partition is

$$\mathcal{P} := (S_1, S_2, \ldots, S_r).$$

We maintain a collection of free (empty) sets in FREE to be used to split sets during the partitioning process. Initially, FREE comprises the empty sets

Input: The vertices $V = \{1, 2, \ldots, n\}$,
the supernodes $K_1, K_2, \ldots, K_r$,
the higher adjacency sets
$\text{hadj}_{G^+}(K_s)$ for $s = 1, 2, \ldots, r$
Output: A reordering of the vertices
consistent with the final ordered partition

```
/* Initializations */;
for t := 1 to r do
    S_t := K_t;
    for i ∈ S_t do set(i) := S_t;
    if |S_t| = 1 then Comp(S_t) := −1;
    else Comp(S_t) := 0;
for t := r + 1 to n do S_t := ∅; Comp(S_t) := 0;
FREE := {S_{r+1}, S_{r+2}, ..., S_n};
P := (S_1, S_2, ..., S_r);
/* For each supernode K_s in reverse order ... */;
for s := r to 1 step −1 do
    S := ∅;
    /* Use hadj_{G+}(K_s) to partition the sets in P. */;
    for i ∈ hadj_{G+}(K_s) do
        S_t := set(i);
        if S_t ∉ S then
            Add S_t to set S;
            if Comp(S_t) = 0 then
                /* Begin the partitioning of S_t. */;
                Choose a free set S_ℓ from FREE and remove
                it from FREE;
                Comp(S_t) := S_ℓ;
                Remove i from set S_t; S_ℓ := {i};
                set(i) := S_ℓ;
        else
            /* Continue the partitioning of S_t. */;
            S_ℓ := Comp(S_t);
            Remove i from set S_t; Add i to set S_ℓ;
            set(i) := S_ℓ;
    /* Update the ordered partition P. */;
    for S_t ∈ S such that Comp(S_t) ≠ −1 do
        S_ℓ := Comp(S_t);
        if S_t = ∅ then
            /* S_t was copied into S_ℓ. */;
            Replace S_t in P with S_ℓ;
            Put S_t back in the set of free sets FREE;
            Comp(S_t) := 0; Comp(S_ℓ) := 0;
        else
            /* Both S_t and S_ℓ are non-empty. */;
            Replace S_t in P with S_t, S_ℓ in that order;
            if |S_t| = 1 then Comp(S_t) := −1;
            else Comp(S_t) := 0;
            if |S_ℓ| = 1 then Comp(S_ℓ) := −1;
            else Comp(S_ℓ) := 0;
Compute a reordering of V consistent with P;
```

Algorithm 1: Refine_supernode_reordering – Description of our partition refinement algorithm for reordering the columns within supernodes.

$S_{r+1}, S_{r+2}, \ldots, S_n$. We also maintain a "membership" n-vector named set throughout the algorithm so that $\text{set}(i) = S_t$ if and only if $i \in S_t$.

We maintain a multi-purpose n-vector named Comp that is used as follows. When a set has only a single member, clearly it cannot be partitioned. We mark such a set S_t by setting $\text{Comp}(S_t) := -1$. Once a set has only a single member, it will have only that single member from then on in the algorithm. For any set S_t that is empty and free, we will maintain $\text{Comp}(S_t)$ so that $\text{Comp}(S_t) = 0$. For any set S_t that has two or more members and *is not* yet currently being partitioned, $\text{Comp}(S_t)$ is maintained so that $\text{Comp}(S_t) = 0$. Finally, for any set S_t that has two or more members and *is* already currently being partitioned, $\text{Comp}(S_t)$ is maintained so that $\text{Comp}(S_t) = S_\ell$, where S_ℓ is the *companion set* being used to partition S_t.

After the initialization steps are completed, the higher adjacency sets $\text{hadj}_{G^+}(K_s)$ are processed in "reverse" order. Assume that the algorithm has just begun processing $\text{hadj}_{G^+}(K_s)$. Immediately before that, the set $\mathcal{S}$ is initialized to be the empty set. The set $\mathcal{S}$ will be used to collect every set in the current partition $\mathcal{P}$ that will be partitioned by the higher adjacency set $\text{hadj}_{G^+}(K_s)$.

When $i \in \text{hadj}_{G^+}(K_s)$ is processed, its set S_t is determined. If i is the first vertex from $S_t \cap \text{hadj}_{G^+}(K_s)$ to be processed, then S_t is added to $\mathcal{S}$. If, moreover, S_t has at least two members, then a free set S_ℓ is chosen from (and removed from) FREE in order to serve as the companion set for S_t. We remove i from S_t, and initialize its companion set $S_\ell := \{i\}$.

If, on the other hand, i's set S_t already belongs to $\mathcal{S}$, then the partitioning of S_t was begun for some previous vertex of $S_t \cap \text{hadj}_{G^+}(K_s)$. Here, we get the previously-assigned companion set S_ℓ of S_t, remove i from S_t, and add i to S_ℓ.

After $\text{hadj}_{G^+}(K_s)$ has been used to partition the appropriate sets in $\mathcal{P}$, we must update the ordered partition $\mathcal{P}$ with the newly-partitioned sets. The sets $S_t \in \mathcal{S}$ that originally had only one member are ignored, as they are not partitioned. The other sets in $\mathcal{S}$ are processed as follows.

If $S_t = \emptyset$, then the splitting process merely copied the original set S_t into its companion set S_ℓ, and hence the original set was not partitioned. So we remove S_t from its position in $\mathcal{P}$ and insert S_ℓ into the position in $\mathcal{P}$ formerly occupied by S_t. We also put the now-available empty free set S_t in FREE.

If, on the other hand, $S_t \neq \emptyset$, then both S_t and S_ℓ are non-empty, and the original set was partitioned. In this case, the set S_t is replaced in $\mathcal{P}$ with the pair of sets S_t and S_ℓ in that order. Also, in this part of the algorithm, the values for the Comp vector are updated for use in the next iteration of the outermost **for** loop on s.

Finally, after all of the higher adjacency sets have been used to perform this partition refinement process, the vertices are ordered consistent with the final ordered partition $\mathcal{P}$. (The details of this are not shown in the algorithm.) This concludes our description of the detailed algorithm.

3.2 Time complexity. We do not wish to discuss the implementation issues associated with the algorithm in great detail, but we need to mention the following. If all the sets and the ordered partition are implemented as simple doubly-linked lists, then all of the individual operations on these sets (and the partition) can be performed in constant time. As a consequence, each iteration on $i \in \text{hadj}_{G^+}(K_s)$ can be performed in constant time.

Each set $S_t \in \mathcal{S}$ originally contained at least one member of $\text{hadj}_{G^+}(K_s)$. Recall also that the sets in $\mathcal{S}$ are taken from a partition $\mathcal{P}$. As a result, we have

$$|\mathcal{S}| \leq |\text{hadj}_{G^+}(K_s)|.$$

Note that, based on the same observations used above, each iteration on $S_t \in \mathcal{S}$ can also be performed in constant time. To conclude, the time required by the algorithm is $O(n + \sum_{t=1}^{r} |\text{hadj}_{G^+}(K_t)|)$.

3.3 Alternating the beginning and ending of blocks. Assume that $S \in \mathcal{P}$ and $S \subseteq K_k$, and assume that our algorithm is currently processing $\text{hadj}_{G^+}(K_s)$. Earlier, we pointed out that whenever we have

$$S' := S \cap \text{hadj}_{G^+}(K_s) \neq \emptyset$$

and

$$S'' := S \setminus \text{hadj}_{G^+}(K_s) \neq \emptyset$$

our partition refinement reordering algorithm will order the vertices of S' consecutively and the vertices of S'' consecutively. Consequently, there will be *only one* (K_k, K_s)-block that begins or ends between two consecutive vertices of $S \subseteq K_k$.

In this subsection, we would like to take advantage of the fact that we can control whether this (K_k, K_s)-block indeed begins or indeed ends (left-to-right) in order to further increase the block sizes obtained using our algorithm. To do so, we will need to partition the set $\mathcal{S}$ of "partitioned" sets into what we will call *maximal intervals*.

An *ordered* subset $\mathcal{I} = \{S_{t_1}, S_{t_2}, \ldots, S_{t_v}\} \subseteq \mathcal{S}$ is a *maximal interval* in $\mathcal{S}$ if

1. $\mathcal{I} = (S_{t_1}, S_{t_2}, \ldots, S_{t_v})$ is an interval of consecutive sets in $\mathcal{P}$,

2. $S_{t_1} \subseteq K_k$, $S_{t_2} \subseteq K_k, \ldots, S_{t_v} \subseteq K_k$ for some supernode $K_k > K_s$, and

3. no set $S \in \mathcal{S}$ can be added to $\mathcal{I}$ so that both properties 1 and 2 above still hold true.

We will let $\mathrm{Int}(\mathcal{S})$ be the collection of all such maximal intervals into which $\mathcal{S}$ is partitioned.

The idea behind our enhancement to the algorithm is to have the algorithm alternate appropriately between beginning blocks and ending blocks as it processes each maximal interval $\mathcal{I} \in \mathrm{Int}(\mathcal{S})$. A section of pseudo-code that implements the enhancement is given in Algorithm 2. The pseudo-code replaces the segment of pseudo-code in Algorithm 1 that updates the ordered partition.

```
/* Update the ordered partition P. */;
for I ∈ Int(S) do
    Right := true;
    for S_t ∈ I in order do
        if Comp(S_t) = −1 then
            Right := false;
        else
            S_ℓ := Comp(S_t);
            if S_t = ∅ then
                /* S_t was copied into S_ℓ. */;
                Replace S_t in P with S_ℓ;
                Put S_t back in the set of free sets FREE;
                Comp(S_t) := 0; Comp(S_ℓ) := 0;
                Right := false;
            else
                /* Both S_t and S_ℓ are non-empty. */;
                if Right = true then
                    /* Begin a block and toggle direction. */;
                    Replace S_t in P with S_t, S_ℓ in that order;
                    Right := false;
                else
                    /* End a block and toggle direction. */;
                    Replace S_t in P with S_ℓ, S_t in that order;
                    Right := true;
                if |S_t| = 1 then Comp(S_t) := −1;
                else Comp(S_t) := 0;
                if |S_ℓ| = 1 then Comp(S_ℓ) := −1;
                else Comp(S_ℓ) := 0;
```

Algorithm 2: Our implementation of alternating the start and end of blocks in order to enhance our partition refinement reordering algorithm.

The code alternates between placing the neighbors of $\mathrm{hadj}_{G^+}(K_s)$ (namely, S_ℓ) to the right or to the left. When the placement is to the right, a new (K_k, K_s)-block is started between the last vertex of S_t and the first vertex of S_ℓ (proceeding from left-to-right). When the placement is to the left, the *previous* (K_k, K_s)-block is ended between the last vertex of S_ℓ and the first vertex of S_t (again, proceeding from left-to-right). Thus, the total number of (K_k, K_s)-blocks is potentially cut to half the number that may occur if the neighbors of $\mathrm{hadj}_{G^+}(K_s)$ were always placed to the right. Indeed, in our experiments, using this enhancement consistently results in higher-quality reorderings (i.e., fewer total blocks).

3.4 Improving upon the order in which the supernodes are processed. Our algorithm presented in Algorithm 1 processes the supernodes in the outermost **for** loop in reverse order by their labels. We will call this order the reverse of the *natural order*. In our sparse Cholesky software, the elimination tree is postordered, as a practical matter, in order to number the vertices within supernodes consecutively. This means that the supernodal elimination tree is postordered also—again, as a practical matter. We have observed in our experiments that reversing a postordering is not the best option available.

A *topological ordering* of a rooted tree numbers each vertex (or node) of the tree with a number greater than that of any of its descendants in the tree. Equivalently, it numbers a vertex (or node) of the tree with a number less than that of any of its ancestors in the tree. Our algorithm can process the supernodes in the reverse order of any topological ordering of the supernodal elimination tree.

We have found two reverse topological orderings that improve the quality of our reorderings (i.e., reduce the number of blocks). The first of these will be called *maximum cardinality search*. (It is at least analogous to the maximum cardinality search algorithm introduced by Tarjan and Yannakakis [20].) Initially, any root supernode is selected to be processed first. At any subsequent step, any supernode K_s eligible to be selected next for processing must satisfy the following:

1. All ancestors of K_s have already been selected and processed, and

2. $|\mathrm{hadj}_{G^+}(K_s)|$ is maximum among the supernodes satisfying condition 1 above.

The idea behind the heurisic is to process the largest adjacency sets early, so that the largest sets in the partition also appear early, and are available for partitioning as soon as possible. It is hoped that larger sets (early) will lead to larger blocks in the end.

We will call the second of these reverse topological orderings *maximum descendants search*. As above, initially, any root supernode is selected to be processed first. At any subsequent step, any supernode K_s eligible to be selected next for processing must satisfy the following:

1. All ancestors of K_s have already been selected and processed, and

2. the number of descendants in the supernodal elimination subtree rooted at K_s is maximum among the supernodes satisfying condition 1 above.

The idea behind the heurisic is to process the supernodes with the largest number of descendants early, because the resulting sets will be partitioned the greatest number of times due to the large number of descendants.

In our experiments, both the maximum cardinality search and the maximum descendants search reverse topological orderings consistently resulted in higher-quality reorderings of the supernodes (i.e., fewer total blocks).

4 Experimental evaluation

We evaluate our supernode reordering algorithm using a direct linear solver on the Cori supercomputer from the National Energy Research Scientific Computing Center (NERSC).

Cori has two partitions with different architectures. The first one is the Haswell partition, in which computing nodes are each equipped with two 2.3 GHz 16-core Intel Haswell processors and 128 GB of memory [1]. Each core has its own 64 KB L1 and 256 KB L2 caches, and there is also a 40 MB shared L3 cache per socket.

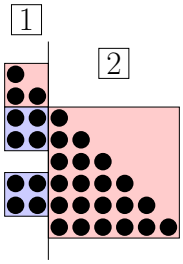


Figure 1: Two regular blocks (in blue) and two diagonal blocks (in red) belonging to two supernodes. In supernode 1, the topmost set of contiguous rows is split into two distinct blocks as the topmost block of a supernode is always a square block.

The second partition is the Intel Knights Landing (KNL) partition. Each computing node is equipped with a single manycore Intel KNL processor, containing 68 cores, with 4 hardware threads per core and 96 GB of memory. The 68 KNL cores are interconnected in a high-speed 2D mesh network with 2 cores per tile, with a 1 MB cache-coherent L2 cache shared between 2 cores in a tile, and with two vector processing units per core.

We evaluate the impact of the supernode reordering strategies on 37 positive definite matrices coming from the SuiteSparse matrix collection [5]. Each matrix corresponds to a specific value on the x-axis in Figures 2 to 5.

In the following, we define a *block* to be delimited by a set of contiguous rows within the same supernode. The topmost block of a supernode, which comprises the diagonal entries, is always considered to be a *square* block. It is referred to as a *diagonal* block while the other blocks are referred to as *regular* blocks. An example of two diagonal blocks and two regular blocks is given in Figure 1.

We use Liu's minimum degree with multiple elimination (`MMD`) [13] and Karypis and Kumar's `METIS` nested dissection (`METIS`) [12] as our fill-reducing orderings. We use the resulting block sizes to evaluate the heuristics for ordering columns within supernodes. We also use factorization times and triangular solve times using `symPACK` [10], a solver for sparse symmetric matrices relying upon the supernodal Fan-Both approach [3].

In the discussions below, we use the following abbreviations for all the variants of Algorithm 1, which is denoted as `SET` in the following, and for the TSP-based heuristic in [17]:

- `SET10`: natural, no alternating.
- `SET11`: natural, alternating.
- `SET20`: maximum cardinality, no alternating.
- `SET21`: maximum cardinality, alternating.
- `SET30`: maximum descendants, no alternating.
- `SET31`: maximum descendants, alternating.
- `TSP`: traveling-salesman-problem-based heuristic of [17].

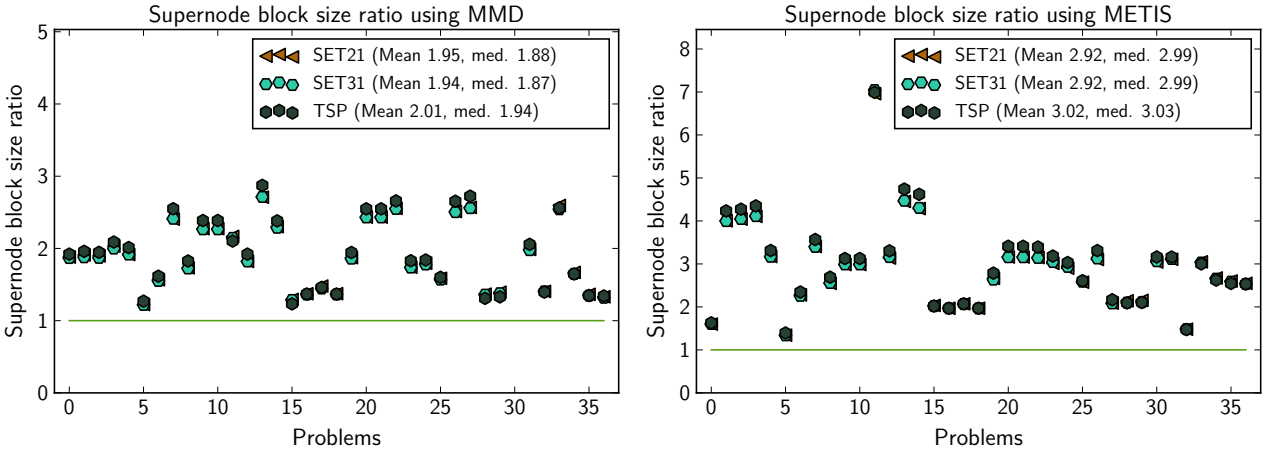
Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	1.333	1.217	0.323	0.967	2.604
SET11	1.553	1.411	0.392	1.118	3.215
SET20	1.553	1.453	0.395	1.007	2.979
SET21	1.951	1.881	0.556	1.222	3.950
SET30	1.563	1.472	0.403	1.009	3.036
SET31	1.938	1.868	0.555	1.219	3.925
TSP	2.005	1.944	0.613	1.233	4.193

(a) `MMD`

Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	2.191	2.159	0.599	1.176	4.386
SET11	2.471	2.460	0.777	1.264	5.729
SET20	2.388	2.312	0.703	1.265	4.856
SET21	2.918	2.986	1.021	1.341	6.961
SET30	2.390	2.327	0.707	1.258	4.894
SET31	2.919	2.989	1.036	1.340	7.044
TSP	3.019	3.031	1.081	1.393	6.991

(b) `METIS`

Table 1: Impact of reordering on block sizes using `MMD` and `METIS` with `symPACK`.



(a) `MMD` (b) `METIS`

Figure 2: Impact of reordering on block sizes using `MMD` and `METIS` with `symPACK`.

The best three average results are highlighted using a gray background in all of the following tables (Tables 1 to 4).

In the first experiment, the results of which are summarized in Table 1, we compute the ratio of the average block sizes obtained using `TSP` and each variant of Algorithm 1 to the average block sizes using the fill-reducing ordering with no reordering within supernodes. A ratio of one indicates no improvement. By looking at the results, we see that `TSP`, `SET21`, and `SET31` produce the best results. For the sake of readability, we will include only these three variants in all figures in the rest of the paper. Block size ratios for these three variants are depicted in Figure 2, in which "block size" thus refers to the average block size in a matrix. As can be seen in the figures and tables, the best variants of our approach are consistently `SET21` and `SET31`, which lead to 1.95x and 1.94x larger blocks respectively on average when `MMD` is used and 2.92x for both variants when `METIS` is used. As expected, `TSP` achieves the best ratio, leading to improvements of 2.01x and 3.02x in the same configurations. Our experiments thus show that the lower-complexity heuristic presented in Algorithm 1 is close to the higher-complexity `TSP` algorithm in terms of quality.

The next experiment aims at characterizing the cost of reordering columns within supernodes. We therefore compute the ratio of the time required to compute the initial fill-reducing ordering, using either `MMD` or `METIS`, to the time required to reorder the columns within supernodes. A ratio greater than one indicates that the reordering heuristic is faster than the fill-reducing algorithm. Results show that all the `SET` variants are far less expensive to compute than an `MMD` ordering (7.79x faster on average); moreover, they are even less expensive to compute relative to the nested dissection orderings computed by `METIS` (38.13x faster on average). The data for `METIS` orderings is depicted in Figure 3; we have not included a figure for `MMD` orderings. The `TSP` approach, by contrast, is more expensive to compute than `MMD` in some cases (0.29x) and is only slightly less expensive to compute on average (1.29x). `TSP` is faster than `METIS` by a ratio of 6.79x on average. This demonstrates the low overhead of the method presented in this work compared to the higher-complexity `TSP` heuristic.

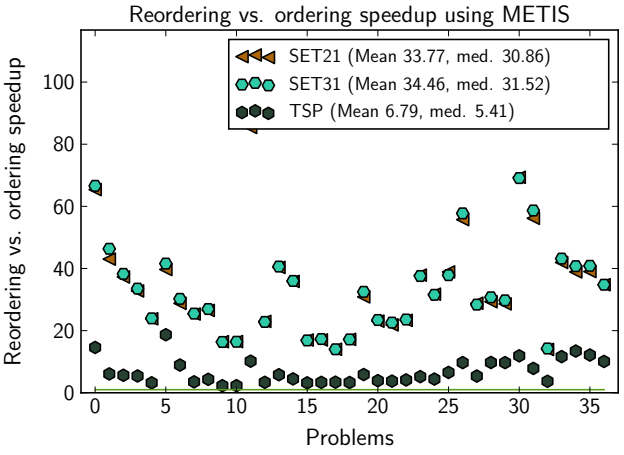


Figure 3: Cost of reordering on NERSC Cori (Haswell) compared to `METIS`.

In the next experiment, we evaluate the impact

Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	1.094	1.015	0.357	0.921	3.064
SET11	1.103	1.010	0.415	0.929	3.411
SET20	1.103	1.023	0.345	0.993	3.013
SET21	1.107	1.014	0.415	0.931	3.408
SET30	1.092	1.012	0.362	0.855	3.089
SET31	1.113	1.020	0.414	0.992	3.393
TSP	1.099	1.019	0.367	0.892	3.126

(a) MMD

Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	1.191	1.055	0.286	0.727	1.929
SET11	1.207	1.080	0.306	0.610	2.038
SET20	1.202	1.076	0.286	0.848	1.967
SET21	1.223	1.091	0.313	0.779	2.156
SET30	1.200	1.070	0.293	0.687	1.955
SET31	1.224	1.093	0.311	0.883	2.141
TSP	1.214	1.085	0.300	0.903	1.992

(b) METIS

Table 2: Speedup on factorization after reordering on NERSC Cori Haswell with symPACK.

Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	1.044	1.006	0.199	0.917	2.116
SET11	1.055	1.012	0.236	0.909	2.328
SET20	1.052	1.011	0.196	0.947	2.102
SET21	1.064	1.018	0.237	0.984	2.346
SET30	1.052	1.015	0.180	0.972	2.021
SET31	1.060	1.017	0.235	0.983	2.317
TSP	1.056	1.013	0.205	0.967	2.161

(a) MMD

Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	1.144	1.045	0.238	0.947	2.097
SET11	1.157	1.058	0.287	0.860	2.490
SET20	1.148	1.038	0.249	0.956	2.135
SET21	1.166	1.056	0.319	0.925	2.670
SET30	1.151	1.046	0.248	0.957	2.151
SET31	1.174	1.048	0.316	0.944	2.575
TSP	1.164	1.055	0.277	0.896	2.339

(b) METIS

Table 3: Speedup on factorization after reordering on NERSC Cori KNL with symPACK.

that reordering columns within supernodes has on the numerical factorization time. The first set of experiments is conducted on the NERSC Cori platform on both the Haswell and the KNL partitions using the symPACK solver. We use two Haswell nodes (a total of 64 cores), and one node on the KNL partition (a total of 68 cores). Our results are summarized in Tables 2 and 3. On both architectures, the improvement in factorization time when using SET over the timing achieved when using solely MMD is 1.10x on Haswell and 1.05x on KNL, while the TSP approach achieves an improvement of, respectively, 1.10x and 1.06x. It is interesting to note that both SET21 (1.107x and 1.064x) and SET31 (1.113x and 1.060x) display a higher impact on factorization than TSP on both platforms (resp. 1.099x and 1.056x). When using nested dissection computed by METIS, the improvement is much higher: 1.21x on average for SET on Haswell while TSP achieves a 1.21x improvement; on KNL, the SET heuristics lead to a 1.16x improvement, which is the same as that of TSP. This shows that reordering columns within supernodes does improve factorization timings by forming larger contiguous blocks.

The impact of reordering columns within supernodes on the triangular solve of symPACK is much higher. As depicted in Figures 4 and 5 and summarized in Tables 4a and 4b, all variants of our reordering techniques achieve an average improvement of 37% in triangular solve time (symPACK) over no reordering when using MMD on the KNL partition, with the SET21 variant achieving 1.46x on average, which is a slightly higher performance improvement than that of the TSP reordering technique. Using METIS, improvements are 1.83x, 1.84x, and 1.86x respectively

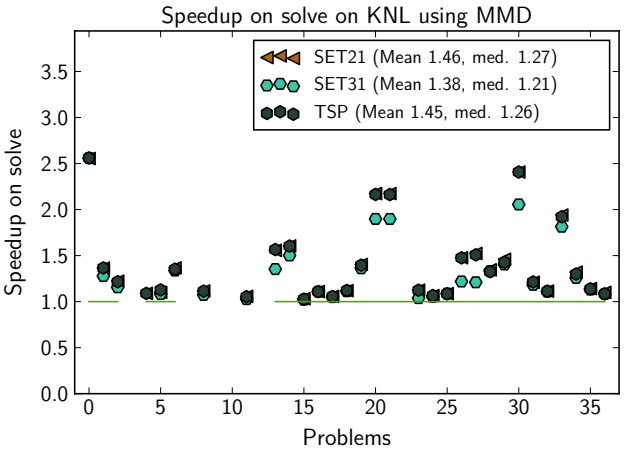


Figure 4: Speedup on solve after reordering on NERSC Cori (KNL) using MMD with symPACK.

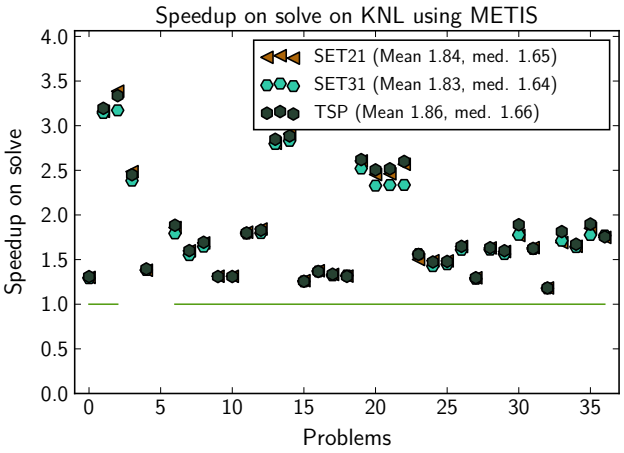


Figure 5: Speedup on solve after reordering on NERSC Cori (KNL) using METIS with symPACK.

Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	1.299	1.124	0.455	0.997	2.949
SET11	1.333	1.163	0.463	1.014	2.983
SET20	1.421	1.242	0.514	1.025	3.208
SET21	1.460	1.271	0.532	1.032	3.284
SET30	1.321	1.122	0.451	0.998	2.944
SET31	1.384	1.210	0.481	1.022	3.102
TSP	1.454	1.258	0.528	1.027	3.248

(a) MMD

Reordering	Mean	Median	Std. dev.	Min.	Max.
SET10	1.670	1.525	0.454	1.162	2.867
SET11	1.697	1.589	0.459	1.164	2.923
SET20	1.766	1.583	0.555	1.126	3.306
SET21	1.844	1.648	0.587	1.182	3.386
SET30	1.729	1.552	0.517	1.140	3.111
SET31	1.834	1.643	0.551	1.178	3.172
TSP	1.858	1.660	0.589	1.182	3.336

(b) METIS

Table 4: Speedup on solve after reordering on NERSC Cori KNL with symPACK.

for SET31, SET21, and TSP. This is of particular importance for applications requiring multiple consecutive solves with different right-hand sides (RHS) coming one-by-one (each RHS is a vector) or group-by-group (each RHS is a matrix).

This led us to simulate such a scenario. In this experiment, we compare the processing time required to perform one matrix factorization and a sequence of solves using METIS on the KNL partition. We let the number of solves vary and depict the speedup achieved when reordering columns within supernodes relative to the time required when not reordering. As can be seen in Figure 6, the best heuristic overall, when performing a large number of solves, is SET21, closely followed by SET31 and TSP. These three heuristics all achieve a speedup of more than 1.6x when doing a hundred solves. Note that at least two solves are required to overcome the overhead associated with the TSP reordering (represented by a vertical line in Figure 6). This demonstrates the much lower overhead of the SET heuristic compared to that of TSP, and it demonstrates that reordering allows significantly higher performance when doing up to a hundred solves.

5 Conclusion

In this paper, we have provided a heuristic for enhancing the structure of the Cholesky factor of a sparse symmetric positive definite matrix in order to improve the efficiency of the numerical factorization. The approach is based on partition refinements within supernodes so that there are fewer dense blocks and the dense blocks are larger. We have demonstrated that the orderings produced by our heuristcs are very competitive with those produced by the TSP approach described in [17], but at much lower costs.

A surprising outcome of the work is the huge improvement in the performance of triangular solution observed for some of the test matrices when the locally refined orderings are used. Further investigations are needed in order to understand this behavior.

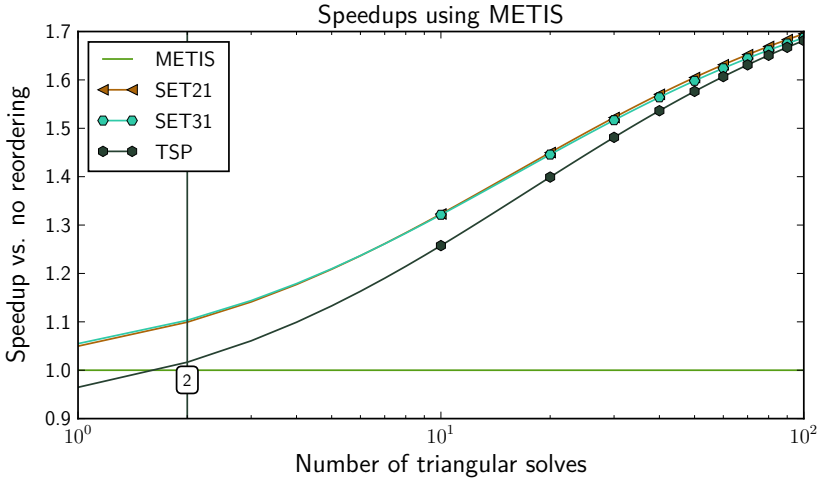


Figure 6: Speedup achieved when reordering columns within supernodes with METIS using symPACK for a single factorization and a varying number of solves on KNL.

References

[1] National Energy Research Scientific Computing Center (NERSC). `http://www.nersc.gov/users/computational-systems/cori/cori-phase-i`, Mar 2016.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.

[3] C. C. Ashcraft, *A taxonomy of column-based Cholesky factorizations*, PhD thesis, Yale University, 1996.

[4] C. C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, H. D. Simon, and P. E. Bjørstad, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, The International Journal of Supercomputing Applications, 1 (1987), pp. 10–30.

[5] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–28.

[6] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[7] A. George and J. W. H. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Rev., 31 (1989), pp. 1–19.

[8] M. Habib, R. McConnell, C. Paul, and L. Viennot, *Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing*, Theor. Comput. Sci., 234 (2000), pp. 59 – 84.

[9] J. D. Hogg, J. K. Reid, and J. A. Scott, *Design of a multicore sparse Cholesky factorization using DAGs*, SIAM J. Sci. Comput., 32 (2010), pp. 3627–3649.

[10] M. Jacquelin, Y. Zheng, E. Ng, and K. Yelick, *An asynchronous task-based fan-both sparse Cholesky solver*, arXiv:1608.00044, (2016).

[11] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, eds., Springer US, Boston, MA, 1972, pp. 85–103.

[12] G. Karypis and V. Kumar, *METIS – unstructured graph partitioning and sparse matrix ordering system, version 2.0*, tech. rep., Department of Computer Science, University of Minnesota, 1995.

[13] J. W. H. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Trans. Math. Software, 11 (1985), pp. 141–153.

[14] ———, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.

[15] J. W. H. Liu, E. G. Ng, and B. W. Peyton, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 242–252.

[16] R. Paige and R. E. Tarjan, *Three partition refinement algorithms*, SIAM J. Comput., 16 (1987), pp. 973–989.

[17] G. Pichon, M. Faverge, P. Ramet, and J. Roman, *Reordering strategy for blocking optimization in sparse linear solvers*, SIAM J. Matrix Anal. Appl., 38 (2017), pp. 226–248.

[18] D. J. Rose, R. E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[19] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, *An analysis of several heuristics for the traveling salesman problem*, SIAM J. Comput., 6 (1977), pp. 563–581.

[20] R. E. Tarjan and M. Yannakakis, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM J. Comput., 13 (1984), pp. 566–579.

[21] W. F. Tinney and J. W. Walker, *Direct solution of sparse network equations by optimally ordered triangular factorization*, Proc. IEEE, 55 (1967), pp. 1801–1809.