SIAM J. SCI. COMPUT.
Vol. 41, No. 1, pp. A269–A291

ON EXPLOITING SPARSITY OF MULTIPLE RIGHT-HAND SIDES IN SPARSE DIRECT SOLVERS*

PATRICK R. AMESTOY†, JEAN-YVES L'EXCELLENT‡, AND GILLES MOREAU‡

Abstract. The cost of the solution phase in sparse direct methods is sometimes critical. It can be higher than that of the factorization in applications where systems of linear equations with thousands of right-hand sides (RHS) must be solved. In this paper, we focus on the case of multiple *sparse* RHS with different nonzero structures in each column. In this setting, *vertical* sparsity reduces the number of operations by avoiding computations on rows that are entirely zero, and *horizontal* sparsity goes further by performing each elementary solve operation only on a subset of the RHS columns. To maximize the exploitation of horizontal sparsity, we propose a new algorithm to build a permutation of the RHS columns. We then propose an original approach to split the RHS columns into a minimal number of blocks, while reducing the number of operations down to a given threshold. Both algorithms are motivated by geometric intuitions and designed using an algebraic approach so that they can be applied to general systems. We demonstrate the effectiveness of our algorithms on systems coming from real applications and compare them to other standard approaches. Finally, we give some perspectives and possible applications for this work.

Key words. sparse linear algebra, sparse matrices, direct method, multiple sparse right-hand sides

AMS subject classifications. 05C50, 65F05, 65F50

DOI. 10.1137/17M1151882

Introduction. We consider the direct solution of sparse systems of linear equations

$$AX = B, \tag{1}$$

where A is an $n \times n$ sparse matrix with a symmetric structure and B is an $n \times m$ matrix of right-hand sides (RHS). When A is decomposed in the form $A = LU$ with a sparse direct method [7], the solution can be obtained by triangular solves involving L and U. In this study, we are interested in the situation where not only A is sparse, but also B, with the columns of B possibly having different structures, and we focus on the efficient solution of the forward system

$$LY = B, \tag{2}$$

where the unknown Y and the RHS B are $n \times m$ matrices. We will see in this study that the ideas developed for (2) are indeed more general and can be applied in a broader context. In particular, they can be applied to the backward substitution phase in situations where the system $UX = Y$ must be solved for a subset of the entries in X [2, 15, 17, 18]. In direct methods, the dependencies of the computations for factorization and solve operations can be represented by a tree [12], which plays

*Submitted to the journal's Methods and Algorithms for Scientific Computing section December 7, 2017; accepted for publication (in revised form) October 19, 2018; published electronically January 15, 2019.

http://www.siam.org/journals/sisc/41-1/M115188.html

Funding: This work was partially supported by LABEX MILYON (ANR-10-LABX-0070).

†Univ. Toulouse, INPT, IRIT UMR5505, F-31071 Toulouse Cedex 7, 31660 France (patrick.amestoy@enseeiht.fr).

‡Univ. Lyon, CNRS, ENS Lyon, Inria, UCBL, LIP UMR5668, F-69342, Lyon Cedex 07, 69364 France (jean-yves.l.excellent@ens-lyon.fr, gilles.moreau@ens-lyon.fr).

a central role and expresses parallelism. The factorization phase is usually the most costly phase but, depending on the number of columns m in B or on the number of systems to solve with identical A and different B, the cost of the solve phase may also be significant. As an example, electromagnetism, geophysics, or imaging applications can lead to systems with sparse multiple RHS for which the solution phase is significantly more costly than the factorization [1, 14]. Such applications motivate the algorithms presented in this study.

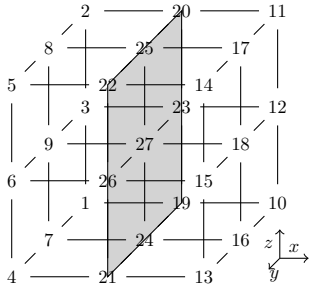
It is worth considering a RHS as sparse when doing so improves performance or storage compared to the dense case. The exploitation of RHS sparsity (later extended to reduce computations when only a subset of the solution is needed [15, 17]) was formalized by Gilbert [10] and Gilbert and Liu [11], who showed that the structure of the solution Y from (2) can be predicted from the structures of L and B. From this structure prediction, one can design mechanisms to reduce computation. In particular, *tree pruning* suppresses nodes in the elimination tree involving only computations on zeros. When solving a problem with multiple RHS, the preferred technique is usually to process all the RHS in one shot. The subset of the elimination tree to be traversed is then the union of the different pruned trees (see section 2). However, when the RHS have different structures, extra operations on zeros are performed. In order to limit these extra operations, several approaches may be applied. The one that minimizes the number of operations processes the RHS columns one by one, each time with a different pruned tree. However, such an approach is not practical and leads to a poor arithmetic intensity (e.g., it will not exploit level 3 BLAS [6]). In the context of blocks of RHS with a predetermined number of columns per block, heuristics based on a specific order of the columns or on hypergraph partitioning have been proposed to determine which columns to include within which block. The function to minimize might be the volume of accesses to the factor matrices [2], or the number of operations [16]. When possible, large sets of columns, possibly the whole set of m columns, may be processed in one shot. Thanks to the different sparsity structure of each column of B, it is then possible to work on less than m columns at most nodes in the tree. Such a mechanism has been introduced in the context of the parallel computation of entries of the inverse [3], where at each node, computations are performed on a contiguous interval of RHS columns.

In this work, we propose an approach to permute B that significantly reduces the amount of computation with respect to previous work. We then go further by dividing B into blocks of columns. However, instead of enforcing a constant block size, we aim at minimizing the number of blocks, which keeps flexibility for the implementation and may improve arithmetic intensity. Because of tree pruning, RHS sparsity limits the amount of available tree parallelism. Therefore, when possible, our heuristics aim at choosing an approach that maximizes tree parallelism.

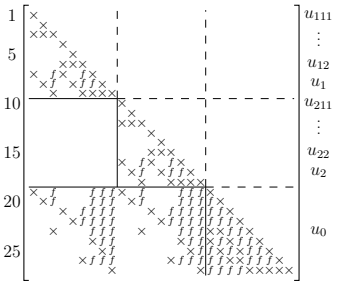
This paper is organized as follows. Section 1 presents the general context of our study, and section 2 describes the *tree pruning* technique arising from Gilbert results, as well as *node intervals*, where different intervals of columns may be processed at each node of the tree. In section 3, we introduce a new permutation to reduce the size of such intervals and thus limit the number of operations, first using geometric considerations for a regular nested dissection ordering, then with a purely algebraic approach that can be applied for the general case. We call it the *flat tree* algorithm because of the analogy with the ordering that one would obtain when "flattening" the tree. In section 4, an original blocking algorithm is then introduced to further improve the flat tree ordering. It aims at defining a limited number of blocks of RHS to minimize the number of operations while preserving parallelism. Section 5 gives

experimental results on a set of systems coming from two geophysics applications relying on Helmholtz or Maxwell equations. Section 6 discusses adaptations of the nested dissection algorithm to further decrease computation, and section 7 shows why this work has a broader scope than solving (2) and presents possible applications.

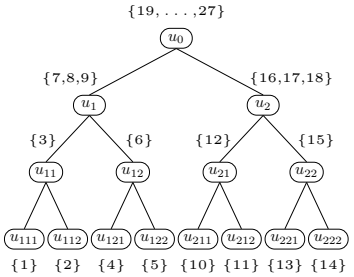
1. Nested dissection, sparse direct solvers, and triangular solve. Ordering the variables of A according to nested dissection [9] limits the operation count and storage of sparse direct methods.



(a) $3 \times 3 \times 3$ regular mesh.



(b) Structures of A and L.



(c) Separator tree T.

FIG. 1. (a) *A 3D mesh with a 7-point stencil. Mesh nodes are numbered according to a nested dissection ordering.* (b) *Corresponding matrix with initial nonzeros (×) in the lower triangular part of a symmetric matrix A and fill-in (f) in the L factor.* (c) *Separator tree, also showing the sets of variables to be eliminated at each node.*

We introduce a $3\times3\times3$ domain in Figure 1(a). It is first divided by a 3×3 constant-x plane separator u_0 and each subdomain is then divided recursively. By ordering the separators after the subdomains, large empty blocks appear in Figure 1(b). The elimination tree [12] represents the dependencies between computations. It can be compressed thanks to the use of supernodes, leading to the *separator tree* of Figure 1(c) when choosing supernodes identical to separators.[1] The order of the tree nodes (u_{111}, u_{112}, u_{11}, u_{121}, u_{122}, $u_{12}, \ldots,$ u_0), partially represented on the right of the matrix, is here a *postordering*; nodes in any subtree are processed consecutively. For $u \in T$ composed of α_u variables, the diagonal block associated with u in the factors matrix is formed of two lower and upper triangular matrices L_{11} and U_{11} of order α_u. The β_u nonzero off-diagonal rows (resp., columns) below (resp., to the right of) L_{11} (resp.,

[1] In this example, identifying supernodes with separators leads to relaxed supernodes. The sparsity in the interaction between u_1 and u_0 (and u_2 and u_0) is not exploited to benefit from larger blocks.

U_{11}) can be compacted into a contiguous matrix L_{21} of size $\beta_u \times \alpha_u$ (resp., U_{12} of size $\alpha_u \times \beta_u$). For the example of u_1 in Figure 1, $\alpha_{u_1} = 3$ (corresponding to variables $7, 8, 9$) and $\beta_{u_1} = 9$ (corresponding to the off-diagonal rows $19, \ldots, 27$). Starting from $y \leftarrow b$ and from the bottom of the tree, at each node u, the active components of y can be gathered into two dense vectors y_1 of size α_u and y_2 of size β_u to perform the operations

$$y_1 \leftarrow L_{11}^{-1} y_1 \quad \text{and} \quad y_2 \leftarrow y_2 - L_{21} y_1. \tag{3}$$

y_1 and y_2 can then be scattered back into y, so that y_2 will be used at higher levels of T. When the root is processed, y contains the solution of $Ly = b$. Denoting by $\delta_u = \alpha_u \times (\alpha_u - 1 + 2\beta_u)$ the number of arithmetic operations for (3), the number of operations to solve $Ly = b$ is

$$\Delta = \sum_{u \in T} \delta_u. \tag{4}$$

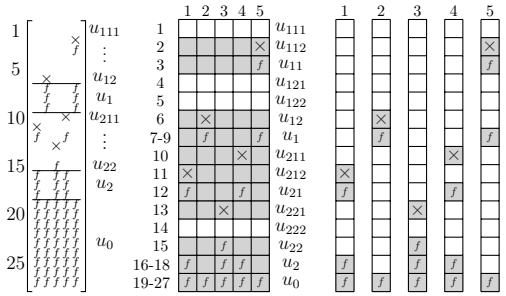
2. Exploitation of sparsity in RHS. Consider a nonsingular $n \times n$ matrix A with a nonzero diagonal, and its directed graph $G(A)$, with an edge from vertex i to vertex j if $a_{ij} \neq 0$. Given a vector b, let us define $\text{struct}(b) = \{i, b_i \neq 0\}$ as its nonzero structure, and $\text{closure}_A(b)$ as the smallest subset of vertices of $G(A)$ including $\text{struct}(b)$ without incoming edges. Gilbert [10, Theorem 5.1] characterizes the structure of the solution of $Ax = b$ by the relation $\text{struct}(A^{-1}b) \subseteq \text{closure}_A(b)$, with equality in case there is no numerical cancellation. In our context of triangular systems, ignoring such cancellation, $\text{struct}(L^{-1}b) = \text{closure}_L(b)$ is also the set of vertices reachable from $\text{struct}(b)$ in $G(L^T)$, where edges have been reversed [11, Theorem 2.1]. Finding these reachable vertices can be done using the transitive reduction of $G(L^T)$, which is a tree (the elimination tree) when L results from the factorization of a matrix with symmetric (or symmetrized) structure. Since we work with a tree T with possibly more than one variable to eliminate at each node, let us define V_b as the set of nodes in T including at least one element of $\text{struct}(b)$. The structure of $L^{-1}b$ is obtained by following paths from the nodes of V_b up to the root. The tree consisting of these paths is the *pruned tree* of b, and we denote it by $T_p(b)$. The number of operations Δ from (4) now depends on b:

$$\Delta(b) = \sum_{u \in T_p(b)} \delta_u. \tag{5}$$

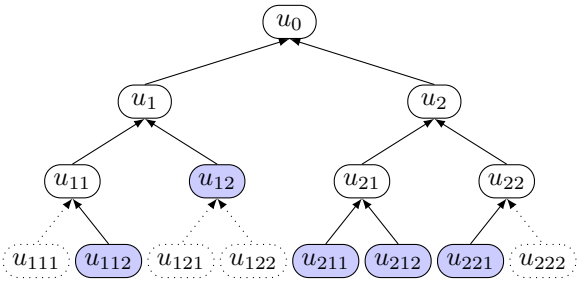
We now consider the multiple RHS case of (2), where RHS columns have different structures, and we denote by B_i the columns of B for $1 \leq i \leq m$. Rather than solving m systems each with a different pruned tree $T_p(B_i)$, we favor matrix-matrix computations by considering $V_B = \bigcup_{1 \leq i \leq m} V_{B_i}$, the union of all nodes in T with at least one nonzero from matrix B, and the pruned tree $T_p(B) = \bigcup_{1 \leq i \leq m} T_p(B_i)$ containing all nodes in T reachable from nodes in V_B. The triangular and update operations (3) become $Y_1 \leftarrow L_{11}^{-1} Y_1$ and $Y_2 \leftarrow Y_2 - L_{21} Y_1$, leading to

$$\Delta(B) = m \times \sum_{u \in T_p(B)} \delta_u. \tag{6}$$

An example is given in Figure 2, where $B = [\{B_{11,1}\}, \{B_{6,2}\}, \{B_{13,3}\}, \{B_{10,4}\}, \{B_{2,5}\}]$, $V_B = \{u_{212}, u_{12}, u_{221}, u_{211}, u_{112}\}$, $\delta(u_0) = 72$, $\delta(u_1) = \delta(u_2) = 60$, $\delta(u_{11}) = 12$, $\delta(u_{112}) = 6$, etc., $\Delta(B) = 5 \times 264 = 1320$, and $\Delta(B_1) + \Delta(B_2) + \cdots + \Delta(B_5) = 744$.



(a) Structures of $L^{-1}B$, $L^{-1}B_1$, ..., $L^{-1}B_5$.



(b) Pruned tree $T_p(B) = T_p(B_1) \cup \ldots \cup T_p(B_5)$.

FIG. 2. *Illustration of multiple RHS and tree pruning.* $\times$ *corresponds to an initial nonzero in* B *and* f *to "fill-in" that appears in* $L^{-1}B$*, represented in terms of original variables and tree nodes. Gray parts of* $L^{-1}B$ *(resp., of* $L^{-1}B_i$*) are those involving computations when RHS columns are processed in one shot (resp., one by one).*

At this point, we exploit tree pruning, or *vertical sparsity*, but perform extra operations by considering globally $T_p(B)$ instead of each individual pruned tree $T_p(B_i)$. Processing B by smaller blocks of columns would further reduce the number of operations at the cost of more traversals of the tree and a smaller arithmetic intensity, with a minimal number of operations $\Delta_{min}(B) = \sum_{i=1,m} \Delta(B_i)$ reached when B is processed column by column, as in Figure 2(a)(right). We note that performing this minimal number of operations while traversing the tree only once (and thus accessing the L factor only once) may require performing complex and costly data manipulations at each node u with copies and indirections to work only on the nonzero entries of $L^{-1}B$ at u. We now present a simpler approach which exploits the notion of intervals of columns at each node $u \in T_p(B)$. This approach to exploit what we call *horizontal sparsity* in B was introduced in another context [3].

Given a matrix B, we associate to a node $u \in T_p(B)$ its set of active columns

$$Z_u = \{j \in \{1, \ldots, m\} \mid u \in T_p(B_j)\}. \tag{7}$$

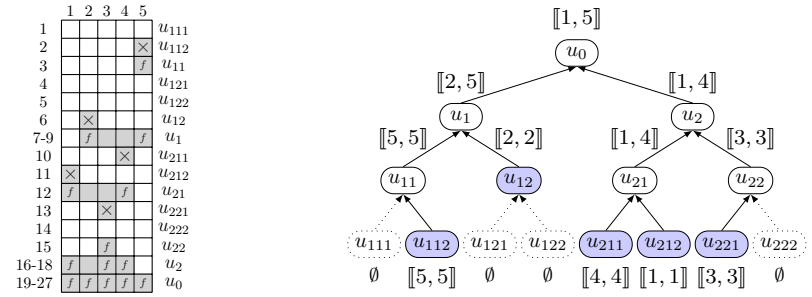
The interval $[\![\min(Z_u), \max(Z_u)]\!]$ includes all active columns, and its length is

$$\theta(Z_u) = \max(Z_u) - \min(Z_u) + 1.$$

Z_u is sometimes defined for an ordered or partially ordered subset R of the columns of B, in which case we use the notation $Z_u|_R$ and $\theta(Z_u|_R)$. For u in $T_p(B)$, Z_u is nonempty and $\theta(Z_u)$ is different from 0. As illustrated in Figure 3, the idea is then to perform the operations (3) on the $\theta(Z_u)$ contiguous columns $[\![\min(Z_u), \max(Z_u)]\!]$

instead of the m columns of B, leading to

$$\Delta(B) = \sum_{u \in T_p(B)} \delta_u \times \theta(Z_u). \tag{8}$$



(a) Structure of $L^{-1}B$.

(b) Pruned tree $T_p(B)$ with intervals.

FIG. 3. *Column intervals for the RHS of Figure* 2: *in gray* (a) *and above/below each node* (b). *Taking for instance* u_{21}, *there are nonzeros in columns* 1 *and* 4, *so* $Z_{u_{21}} = \{1, 4\}$. *Instead of performing the solve operations on* $m = 5$ *columns at* u_{21}, *computation is limited to the* $\theta(Z_{u_{21}}) = 4$ *columns of interval* $[\![1, 4]\!]$ *(and to a single column at, e.g., node* u_{221}*). Overall,* $\Delta(B)$ *is reduced from* 1320 *to* 948 *(while* $\Delta_{min}(B) = 744$*).*

3. Permuting RHS columns. It is clear from Figure 3 that when exploiting horizontal sparsity thanks to column intervals, the number of operations to solve (2) depends on the order of the columns in B. We express the corresponding minimization problem as

(9)

Find a permutation σ of $\{1, \ldots, m\}$ that minimizes

$\Delta(B, \sigma) = \sum_{u \in T_p(B)} \delta_u \times \theta(\sigma(Z_u))$,

where $\sigma(Z_u) = \{\sigma(i) \mid i \in Z_u\}$, and

$\theta(\sigma(Z_u))$ is the length of the permuted interval $[\![\min(\sigma(Z_u)), \max(\sigma(Z_u))]\!]$.

Rather than trying to solve the global problem with linear optimization techniques, we will propose a cheap heuristic based on the tree structure. We first define the notion of node optimality.

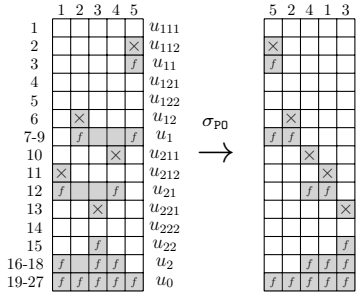
DEFINITION 1. *Given a node* u *in* $T_p(B)$, *and a permutation* σ *of* $\{1, \ldots, m\}$, *we say that we have node optimality at* u, *or that* σ *is* u*-optimal if and only if* $\theta(\sigma(Z_u)) = \#Z_u$, *where* $\#Z_u$ *is the cardinality of* Z_u. *Stated differently,* $\sigma(Z_u)$ *is a set of contiguous elements.*

$\theta(\sigma(Z_u)) - \#Z_u$, the number of columns (or padded zeros) on which extra computation is performed, is 0 if σ is u-optimal. Using the RHS of Figure 3(a), the identity permutation is u_0-optimal because $\#Z_{u_0} = \#\{1, 2, 3, 4, 5\} = 5 = \theta(Z_{u_0})$, but not u_1- and u_2-optimal because $\theta(Z_{u_1}) - \#Z_{u_1} = 2$ and $\theta(Z_{u_2}) - \#Z_{u_2} = 1$, respectively. Our aim is thus to find a permutation σ that reduces the difference $\theta(\sigma(Z_u)) - \#Z_u$. After describing a permutation based on a postordering of $T_p(B)$, we present our new heuristic, which targets node optimality in priority at the nodes near the top of the tree.

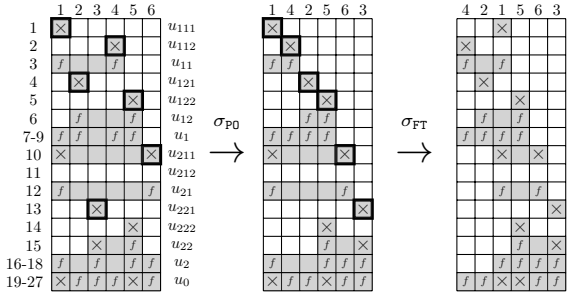
3.1. The postorder permutation. In Figure 1, the sequence $[u_{111}, u_{112}, u_{11}, u_{121}, u_{122}, u_{12}, u_1, u_{211}, u_{212}, u_{21}, u_{221}, u_{222}, u_{22}, u_2, u_0]$ used to order the matrix follows a postordering.

DEFINITION 2. *Consider a postordering of the tree nodes $u \in T$, and a RHS matrix $B = [B_j]_{j=1,\ldots,m}$ where each column B_j is represented by one of its associated nodes $u(B_j) \in V_{B_j}$ (see below). B is said to be postordered if and only if, $\forall j_1, j_2, 1 \leq j_1 < j_2 \leq m$, we have either $u(B_{j_1}) = u(B_{j_2})$, or $u(B_{j_1})$ appears before $u(B_{j_2})$ in the postordering. In other words, the order of the columns B_j is compatible with the order of their representative nodes $u(B_j)$.*

The postordering has been applied [2, 15, 16] to build regular chunks of RHS columns with "nearby" pruned trees, thereby limiting the accesses to the factors or the amount of computation. It was also experimented along with node intervals [3] to RHS with a single nonzero per column. In Figure 3(a), B has a single nonzero per column. The initial natural order of the columns (`INI`) induces computation on explicit zeros represented by gray empty cells, and we had $\Delta(B) = \Delta(B, \sigma_{\texttt{INI}}) = 948$ and $\Delta_{min}(B) = 744$ (see Figure 3). On the other hand, the postorder permutation, $\sigma_{\texttt{PO}}$, reorders the columns of B so that the order of their representative nodes u_{112}, u_{12}, u_{211}, u_{212}, u_{221} is compatible with the postordering. In this case, there are no gray empty cells (see Figure 4(a)) and $\Delta(B, \sigma_{\texttt{PO}}) = \Delta_{min}(B)$. More generally, it can be shown that the postordering induces no extra computations for RHS with a single nonzero per column [3]. For applications with multiple nonzeros per RHS, each column B_j may correspond to a set V_{B_j} with more than one node, in which case we choose the node that appears first in the sequence of postordered nodes of the tree [4].



(a) RHS with one nonzero per column.



(b) RHS with multiple nonzeros per column.

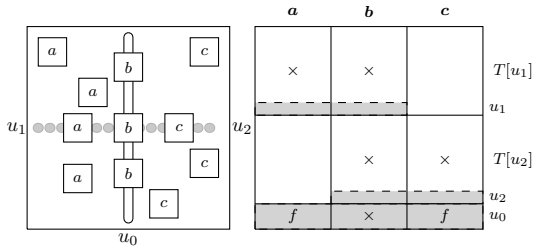
FIG. 4. *Illustration of the postordering of two RHS with one or more nonzeros per column.*

In Figure 4(b), some columns of B have several nonzeros. For example, $V_{B_1} = \{u_{111}, u_{211}, u_0\}$ is represented by u_{111}, and $V_{B_3} = \{u_{221}, u_{22}\}$ by u_{221} (cells with a bold contour). The postorder permutation yields $\sigma_{\texttt{PO}}(B) = [B_1, B_4, B_2, B_5, B_6, B_3]$, which reduces the number of gray cells and the number of operations from $\Delta(B) = 1368$ to $\Delta(B, \sigma_{\texttt{PO}}) = 1242$. Computations on padded zeros still occur, for example at nodes u_{211} where $\theta(\sigma_{\texttt{PO}}(Z_{u_{211}})) - \#Z_{u_{211}} = 5 - 2 = 3$.

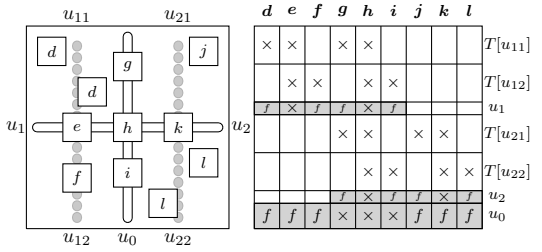
The quality of $\sigma_{\texttt{PO}}$ depends on the tree postordering. If u_{111} and u_{112} were exchanged in the original postordering, B_1 and B_4 would be swapped, further reducing Δ. One drawback of the postorder permutation is that, since the position of a column is based a single representative node, some information is unused. In Figure 4(b), we represented another permutation $\sigma_{\texttt{FT}}$ that yields $\Delta(B, \sigma_{\texttt{FT}}) = 1140$. This more general and powerful heuristic is presented in the next subsection.

3.2. The flat tree permutation. With the aim of satisfying node optimality (see Definition 1), we present another algorithm to compute the permutation σ by first illustrating its geometric properties and then extending it to rely only on algebraic properties.

3.2.1. Geometrical intuition. For $u \in T$, the domain associated with u is defined by the subtree rooted at u and is denoted by $T[u]$. The set of variables in $T[u]$ corresponds to a subdomain created during the nested dissection algorithm. As an example, the initial two-dimensional (2D) domain in Figure 5(a) (left) is $T[u_0]$, and its subdomains created by dividing it with u_0 are $T[u_1]$ and $T[u_2]$. In the following, $T[u]$ will equally refer to a subdomain or a subtree. We make the strong assumption here that the nonzeros in a RHS column correspond to geometrically contiguous nodes in a regular domain on which a perfect nested dissection has been performed.



(a) Flat tree step 1.



(b) Flat tree step 2.

FIG. 5. *Flat tree geometrical illustration. In* (a) *and* (b)*, the figure on the left represents different types of RHS, and the one on the right the permuted RHS matrix.* × *or* f *in a rectangle indicates the presence of nonzeros in the corresponding submatrix. Parts of the matrix filled in gray are fully dense and blank parts only contain zeros.*

The *flat tree* algorithm relies on the evaluation of the position of each RHS column compared to separators. The name flat tree comes from the fact that, given a parent node with two child subtrees, the algorithm orders first RHS columns included in the left subtree, then RHS columns associated to the parent (because they intersect both subtrees), and finally, as in an inorder, RHS columns included in the right subtree. The RHS columns in Figure 5(a) may be *identified* by three different types noted a, b, and c according to their positions set by the root separator u_0. A RHS column is of type a (resp., c) when its nonzero structure is included in $T[u_1]$ (resp., $T[u_2]$), and b when it is *divided* by u_0. First, we group the RHS according to their type (a, b, or c) with respect to u_0 which leads to the creation of submatrices/subsets of RHS columns noted $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$. Second, we make sure to place $\boldsymbol{b}$ between $\boldsymbol{a}$ and $\boldsymbol{c}$. Since all RHS in $\boldsymbol{a}$ and $\boldsymbol{b}$ have at least one nonzero in $T[u_1]$, u_1 belongs to the pruned tree of all of them, hence the dense area filled in gray in the RHS structure. The same is true for $\boldsymbol{b}$ and $\boldsymbol{c}$ and u_2. By permuting B as $[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]$ ($[\boldsymbol{c}, \boldsymbol{b}, \boldsymbol{a}]$ is also possible), $\boldsymbol{a}$ and $\boldsymbol{b}$, and $\boldsymbol{b}$ and $\boldsymbol{c}$, are contiguous. Thus, $\theta(Z_{u_1}) = \#Z_{u_1}$, $\theta(Z_{u_2}) = \#Z_{u_2}$, and we have minimized (8) for u_1 and u_2 thanks to u_1- and u_2-optimality. The algorithm proceeds recursively on each submatrix created to obtain *local* node optimality. First, $\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}$ (resp., $\boldsymbol{j}$, $\boldsymbol{k}$, $\boldsymbol{l}$) form subsets of the RHS of $\boldsymbol{a}$ (resp., $\boldsymbol{c}$) based on their position/type with respect to u_1 (resp., u_2); see Figure 5(b). Second, thanks to the perfect nested dissection assumption, u_1 and u_2 can be combined to form a single separator that subdivides the RHS of $\boldsymbol{b}$ into three subsets $\boldsymbol{g}$, $\boldsymbol{h}$, and $\boldsymbol{i}$. During this second step, B is permuted as $[\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}, \boldsymbol{l}]$. The algorithm stops when the tree is fully processed or the RHS sets contain a single RHS.

3.2.2. Algebraic approach. Let us consider the columns of B as an initially unordered *set* of RHS columns that we denote $R_B = \{B_1, B_2, \ldots, B_m\}$. $R \subset R_B$ is a subset of the columns of B and $r \in R$ is a generic element of R (one of the columns B_j). A permuted submatrix of B can be expressed as an ordered *sequence* of RHS columns. For two subsets of columns R and R', $[R, R']$ denotes a sequence of RHS columns in which the RHS from the subset R are ordered before those from R', without the order within R and R' necessarily defined. We found this framework of RHS sets and subsets better adapted to formalize our algebraic algorithm than matrix notation with complex index permutations. We now characterize the geometrical position of a RHS using the notion of *pruned layer*: for a given depth d in the tree, and for a given RHS $r \in R_B$, we define the *pruned layer* $L_d(r)$ as the set of nodes at depth d in the pruned tree $T_p(r)$. In the example of Figure 5(a), $L_1(r) = \{u_1\}$ for all $r \in \boldsymbol{a}$, $L_1(r) = \{u_2\}$ for all $r \in \boldsymbol{c}$, and $L_1(r) = \{u_1, u_2\}$ for all $r \in \boldsymbol{b}$. The notion of pruned layer formally identifies sets of RHS with common characteristics in the tree, without geometric information. This is formalized and generalized by Definition 3.

DEFINITION 3. *Let $R \subset R_B$ be a set of RHS, and let U be a set of nodes at depth d of the tree T. We define $R[U] = \{r \in R \mid L_d(r) = U\}$ as the subset of RHS with pruned layer U.*

We have for example (see Figure 5) $R[\{u_1\}] = \boldsymbol{a}, R[\{u_2\}] = \boldsymbol{c}$ and $R[\{u_1, u_2\}] = \boldsymbol{b}$ at depth $d = 1$.

The algebraic recursive algorithm is depicted in Algorithm 1. Its arguments are R, a set of RHS, and d, the current depth. Initially, $d = 0$ and $R = R_B = R[u_0]$. At each recursion step, the algorithm builds the distinct pruned layers $U_i = L_{d+1}(r)$ for the RHS r in R. Then, instead of looking for a permutation σ to minimize $\sum_{u \in T_p(R_B)} \delta_u \times \theta(\sigma(Z_u))$, it orders the $R[U_i]$ by considering the *restriction* of (9) to

R and to nodes at depth $d+1$ of $T_p(R)$. Furthermore, with the assumption that T is balanced, all nodes at a given level of $T_p(R)$ are of comparable size. δ_u may thus be assumed *constant* per level and need not be taken into account. The algorithm is a greedy top-down algorithm, where at each step a local optimization problem is solved. This way, priority is given to the top of the tree, which is, in general, more critical because factor matrices are larger.

Algorithm 1 Flat Tree.

procedure FLATTREE(R, d)
  (1) Build the set of children $C(R)$
  (1.1) Identify the distinct pruned layers (pruned layer = set of nodes)
  $\mathcal{U} \leftarrow \emptyset$
  for all $r \in R$ **do**
    $\mathcal{U} \leftarrow \mathcal{U} \cup \{L_{d+1}(r)\}$
  end for
  (1.2) $C(R) = \{R[U] \mid U \in \mathcal{U}\}$
  (2) Order children $C(R)$ as $[R[U_1], \ldots, R[U_{\#C(R)}]]$
  return [FLATTREE($R[U_1], d+1$),...,FLATTREE($R[U_{\#C(R)}], d+1$)]
end procedure

The recursive structure of the algorithm can be represented by a recursion tree T_{rec} defined as follows: each node R of T_{rec} represents a set of RHS, $C(R)$ denotes the set of children of R, and the root is R_B. By construction of Algorithm 1, $C(R)$ is a partition of R, i.e., $R = \dot{\bigcup}_{R' \in C(R)} R'$ (disjoint union). Note that all $r \in R$ such that $L_{d+1}(r) = \emptyset$ belong to $R[\emptyset]$, which is also included in $C(R)$. In this special case, $R[\emptyset]$ can be added at either extremity of the current sequence without introducing extra computation and the recursion stops for those RHS, as will be illustrated in Figures 6 and 7(a). With this construction, each leaf of T_{rec} contains RHS with indistinguishable nonzero structures, and keeping them contiguous in the final permutation avoids introducing extra computations. Assuming that for each $R \in T_{rec}$ the children $C(R)$ are ordered, this induces an ordering of all the leaves of the tree, which defines the final RHS sequence. We now explain how the set of children $C(R)$ is built and ordered at each step.

(1) *Building the set of children.* The set of children of $R \in T_{rec}$ is built by first identifying the pruned layers U of all RHS $r \in R$. The different pruned layers are stored in $\mathcal{U}$ and we have, for example, $\mathcal{U} = \{\{u_1\}, \{u_2\}, \{u_1, u_2\}\}$ (see Figure 5, first step of the algorithm). We define $C(R) = \{R[U] \mid U \in \mathcal{U}\}$ (Definition 3), which forms a partition of R. One important property is that all $r \in R[U]$ have the same nonzero structure at the corresponding layer so that numbering them contiguously prevents the introduction of extra computation.

(2) *Ordering the children.* The children sequence $[R[U_1], \ldots, R[U_{\#C(R)}]]$ at depth $d+1$ should minimize the size of the intervals for nodes u at depth $d+1$ of $T_p(R)$. The order inside each $R[U_i]$ does not impact the size of these intervals (it will only impact lower levels). For any node u at depth $d+1$ in $T_p(R)$, we have $\theta(Z_u|_R) = \max(Z_u|_R) - \min(Z_u|_R) + 1 = \sum_{i=i_{\min}(u)}^{i_{\max}(u)} \#R[U_i]$, where $Z_u|_R$ is the set of permuted indices representing the active columns restricted to R, and $i_{\min}(u) = \min\{i \in \{1, \ldots, \#C(R)\} \mid u \in U_i\}$ (resp., $i_{\max}(u) = \max\{i \in \{1, \ldots, \#C(R)\} \mid u \in U_i\}$) is the first (resp., last) index i such that $u \in U_i$. Finally, we minimize the local cost

function (sum of the interval sizes for each node at depth $d+1$):

$$\text{cost}([R[U_1],\dots,R[U_{\#C(R)}]]) = \sum_{\substack{u\in T_p(R)\\ depth(u)=d+1}} \sum_{i=i_{min}(u)}^{i_{max}(u)} \#R[U_i]. \tag{10}$$

To build the ordered sequence $[R[U_1],\dots,R[U_{\#C(R)}]]$, we use a greedy algorithm that starts with an empty sequence and at each step $k \in \{1,\dots,\#C(R)\}$ inserts a RHS set $R[U]$ picked randomly in $C(R)$ at the position that minimizes (10) on the current sequence. To do so, we simply start from one extremity of the sequence of size $k-1$ and compute (10) for the new sequence of size k for each of the k possible positions; if several positions lead to the same minimal cost, the first one encountered is chosen. In case u-optimality is obtained for each node u considered, then the permutation is said to be *perfect* and the cost function is minimal, locally inducing no extra operations on those nodes.

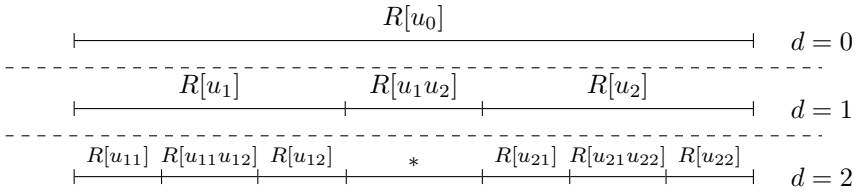
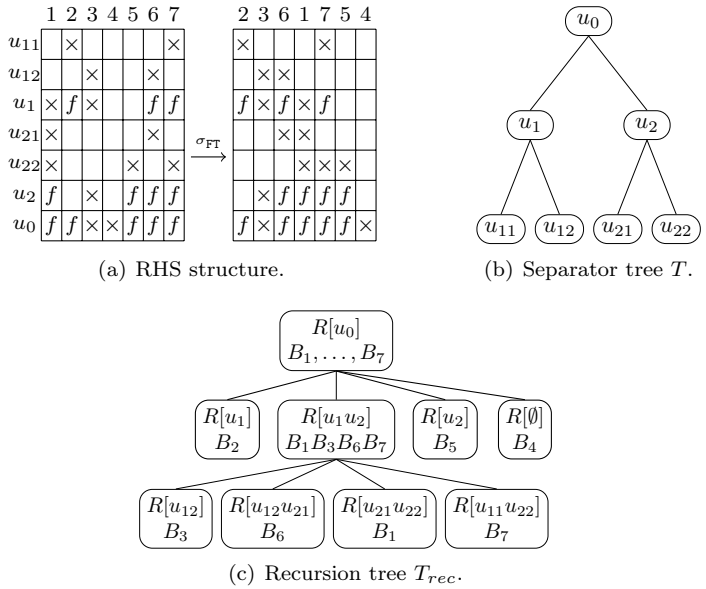


FIG. 6. *A layered sequence built by the flat tree algorithm on a binary tree. Sets $R[\emptyset]$ (not represented) could be added at either extremity of the concerned sequence (e.g., right after $R[u_2]$ for a RHS included in u_0). With the strong assumptions of Figure* 5, *$* = R[u_{11}u_{21}]$, $R[u_{11}u_{12}u_{21}u_{22}]$, $R[u_{12}u_{22}]$. *Otherwise, * is more complex.*

Figure 6 shows the recursive structure of the RHS sequence after applying the algorithm on a binary tree. We refer to this representation as the *layered sequence*. For simplicity, the notation for pruned layers has been reduced from, e.g., $\{u_1\}$ to u_1 and from $\{u_1,u_2\}$ to u_1u_2. From the recursion tree point of view, $R[u_1], R[u_1u_2], R[u_2]$ are the children of $R[u_0]$ in T_{rec}; $R[u_{11}], R[u_{11}u_{12}], R[u_{12}]$ those of $R[u_1]$, etc.

Although we still use a binary tree, we make no assumption on the RHS structure, the domain, or the ordering in the example of Figure 7. The set of pruned layers corresponding to $R[u_0]$ is $\mathcal{U} = \{u_1u_2, u_1, \emptyset, u_2\}$, so that $C(R[u_0]) = \{R[u_1u_2], R[u_1], R[\emptyset], R[u_1u_2]\}$. As can be seen in the nonpermuted RHS structure, $R[\emptyset] = B_4$ at depth 1 induces extra operations at nodes below u_0, which disappear when placing $R[\emptyset]$ at one extremity of the sequence. We choose to place it last and obtain the sequence $[R[u_1], R[u_1u_2], R[u_2], R[\emptyset]]$. A recursive call is done on the identified sets, as illustrated in Figure 7(c). Since $R[u_1]$, $R[u_2]$, and $R[\emptyset]$ contain a single RHS, we focus on $R = R[u_1u_2]$, whose set of pruned layers is $\mathcal{U} = \{u_{21}u_{22}, u_{12}, u_{12}u_{21}, u_{11}u_{22}\}$. The sequence $[R[U_1], R[U_2], R[U_3], R[U_4]]$, where $U_1 = u_{12}$, $U_2 = u_{12}u_{21}$, $U_3 = u_{21}u_{22}$, and $U_4 = u_{11}u_{22}$, is a perfect sequence which gives local optimality. However, taking the problem globally, we see that $\theta(Z_{u_{11}}) \neq \#Z_{u_{11}}$ in the final sequence $[B_2, B_3, B_6, B_1, B_7, B_5, B_4]$. Although not relying on geometric assumptions, particular RHS structures, or binary trees, computations on explicit zeros (for example, zero rows in column $\boldsymbol{f}$ and subdomain $T[u_{11}]$ in Figure 5(b)) may still occur with the flat tree algorithm. This will also be illustrated in section 5, where $\Delta(B, \sigma_{\texttt{FT}})$ is 39% larger than $\Delta_{min}(B)$, in the worst case. A blocking algorithm is now introduced to further reduce $\Delta(B, \sigma_{\texttt{FT}})$.



(a) RHS structure. (b) Separator tree T.

(c) Recursion tree T_{rec}.

FIG. 7. *Illustration of the algebraic flat tree algorithm on a set of* 7 *RHS.*

4. Toward a minimal number of operations using blocks.

4.1. Geometrical intuition. In terms of operation count, optimality ($\Delta_{min}(B)$) is obtained when processing the columns of B one by one, using m blocks. However, this requires processing the tree m times and will typically lead to a poor arithmetic intensity (and likely a poor performance). On the other hand, the algorithms of section 3 only use one block, which provides a higher arithmetic intensity but requires extra operations. In this section, our objective is to create a minimal number of (possibly large) blocks while reducing the number of extra operations by a given amount.

On the one hand, two RHS or sets of RHS included in two different domains exhibit interesting properties, as can be observed for sets $\boldsymbol{a} \in T[u_1]$ and $\boldsymbol{c} \in T[u_2]$ from Figure 5(a). No extra operations are introduced between them: $\Delta([\boldsymbol{a}, \boldsymbol{c}]) = \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{c})$. We say that $\boldsymbol{a}$ and $\boldsymbol{c}$ are *independent sets*, and they can be associated together. On the other hand, a set of RHS intersecting a separator (such as set $\boldsymbol{b}$) has zeros and nonzeros in rows common to their adjacent RHS sets ($\boldsymbol{a}$ and $\boldsymbol{c}$) which will likely introduce extra computation. In Figure 5(b), we have, for example, $\Delta([\boldsymbol{a}, \boldsymbol{b}]) = \Delta([\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]) > \Delta([\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}]) + \Delta([\boldsymbol{g}, \boldsymbol{h}, \boldsymbol{i}]) = \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{b})$ and $\Delta([\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]) > \Delta(\boldsymbol{a}) + \Delta(\boldsymbol{b}) + \Delta(\boldsymbol{c})$. We say that $\boldsymbol{b}$ is a set of *problematic* RHS. Figure 4(right) gives another example where extracting the problematic RHS B_1 and B_5 from $[B_4, B_2, B_1, B_5, B_6, B_3]$ suppresses all extra operations: $\Delta([B_4, B_2, B_6, B_3]) + \Delta([B_1, B_5]) = \Delta_{min} = 1056$. Situations where no assumption on the RHS structure is made are more complicated and require a general approach. For this, we formalize the notion of *independence*, which will be the basis for our blocking algorithm.

4.2. Algebraic formalization. In this section, we assume the matrix B to be flat tree ordered, and the recursion tree T_{rec} to be built and ordered. Using the notations of Definition 3, we give an algebraic definition of the independence property.

DEFINITION 4. *Let U_1, U_2 be two sets of nodes at a given depth of a tree T, and let*

$R[U_1], R[U_2]$ *be the corresponding sets of RHS.* $R[U_1], R[U_2]$ *are said to be* independent *if and only if* $U_1 \cap U_2 = \emptyset$.

With Definition 4, we are able to formally identify independent sets that can be associated together. Take, for example, $\boldsymbol{a} = R[u_1]$ and $\boldsymbol{c} = R[u_2]$ (Figure 5(a)); $R[u_1]$ and $R[u_2]$ are independent and $\Delta([R[u_1], R[u_2]]) = \Delta(R[u_1]) + \Delta(R[u_2])$. On the contrary, when $R[U_1], \ldots, R[U_n]$ are not pairwise independent, we group together independent sets of RHS, while forming as few groups as possible. This problem is equivalent to a graph coloring problem, where $R[U_1], \ldots, R[U_n]$ are the vertices, and an edge exists between $R[U_i]$ and $R[U_j]$ if and only if $U_i \cap U_j \neq \emptyset$. Several heuristics exist for this problem, and each color will correspond to one group.

Algorithm 2 Blocking algorithm.

for $d = 0$ **to** d_{max} **do**
  $j \leftarrow 0$ /* #groups at depth $d+1$ */
  for all groups g_i^d at detph d **do**
    $(g_{j+1}^{d+1} \ldots g_{j+k}^{d+1}) \leftarrow$ BUILDGROUPS$(g_i^d, d+1)$
    /* k new groups have been created */
    $j \leftarrow j + k$
  end for
end for

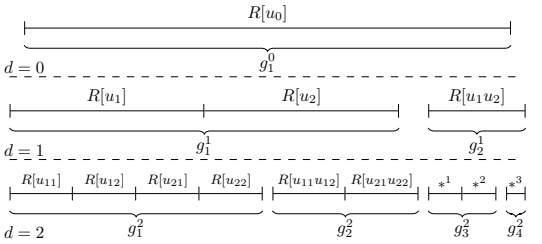


FIG. 8. *A first version of the blocking algorithm (top). It is illustrated (bottom) on the layered sequence of Figure* 6. *With the geometric assumptions of Figure* 5, $*^1 = R[u_{11}u_{21}], *^2 = R[u_{12}u_{22}]$, *and* $*^3 = R[u_{11}u_{12}u_{21}u_{22}]$.

The blocking algorithm as depicted in Figure 8 traverses T_{rec} from top to bottom. At each depth d, each intermediate group g_i^d verifies the following properties: (i) g_i^d can be represented by a sequence $[R[U_1], \ldots, R[U_n]]$, and (ii) the sequence respects the flat tree order of T_{rec}. Then, BUILDGROUPS$(g_i^d,\ d+1)$ first builds the sets of RHS at depth $d+1$, which are exactly the children of the $R[U_j] \in g_i^d$ in T_{rec}. Second, BUILDGROUPS$(g_i^d,\ d+1)$ solves the aforementioned coloring problem on these RHS sets and builds the k groups $(g_{j+1}^{d+1}, \ldots, g_{j+k}^{d+1})$.

In Figure 8(bottom), there is initially a single group $g_1^0 = [R[u_0]]$ with one set of RHS, which may be expressed as the ordered sequence $[R[u_1]R[u_1u_2]R[u_2]]$. g_1^0 does not satisfy the independence property at depth 1 because $u_1 \cap u_1u_2 \neq \emptyset$ or $u_2 \cap u_1u_2 \neq \emptyset$. BUILDGROUPS$(g_1^0,\ 1)$ yields $g_1^1 = [R[u_1], R[u_2]]$ and $g_2^1 = [R[u_1u_2]]$. The algorithm proceeds until a maximal depth d_{max}: $(g_1^2, g_2^2) =$ BUILDGROUPS$(g_1^1, 2)$, $(g_3^2, g_4^2) =$ BUILDGROUPS$(g_2^1, 2)$, etc. To illustrate the importance of property (ii), let us take sets $\boldsymbol{d} = R[u_{11}], \boldsymbol{f} = R[u_{12}], \boldsymbol{j} = R[u_{21}]$, and $\boldsymbol{l} = R[u_{22}]$ from Figure 5(b). One can see that $\Delta([\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]) = \Delta(\boldsymbol{d}) + \Delta(\boldsymbol{f}) + \Delta(\boldsymbol{j}) + \Delta(\boldsymbol{l}) < \Delta([\boldsymbol{d}, \boldsymbol{j}, \boldsymbol{f}, \boldsymbol{l}])$. Compared to $[\boldsymbol{d}, \boldsymbol{f}, \boldsymbol{j}, \boldsymbol{l}]$, which respects the global flat tree ordering and ensures u_1- and

u_2-optimality, $[\boldsymbol{d}, \boldsymbol{j}, \boldsymbol{f}, \boldsymbol{l}]$ does not and would thus increase $\theta(Z_{u_1})$ and $\theta(Z_{u_2})$. Furthermore, Algorithm 2 ensures the property (whose proof can be found in [4]) that the independent sets of RHS grouped together do not introduce extra operations.

PROPERTY 5. *For any group $g^d = [R[U_1], \ldots, R[U_n]]$ created through Algorithm 2 at depth d, we have $\Delta([R[U_1], \ldots, R[U_n]]) = \sum_{i=1}^{n} \Delta(R[U_i])$.*

Interestingly, Property 5 can be used to prove, in the case of a single nonzero per RHS, the optimality of the flat tree permutation.

COROLLARY 6. *Let R_B be a set of RHS such that $\forall r \in R_B, \#V_r = 1$. Then the flat tree permutation is optimal: $\Delta(R_B) = \Delta_{min}(R_B)$.*

Proof. Since $\forall r \in R_B, \#V_r = 1$, $T_p(r)$ is a branch of T. As a consequence, any set of RHS $R[U]$ built through the flat tree algorithm is represented by a pruned layer U containing a single node u. At each step of the flat tree algorithm (Algorithm 1), the RHS sets identified are thus all independent from each other. When applying Algorithm 2, a unique group R_B is then kept until the bottom of the tree. Blocking is thus not needed and Property 5 applies at each level of the recursion. $\Delta(R_B)$ is thus equal to the sum of the $\Delta(R[U])$ for all leaves $R[U]$ of the recursion tree T_{rec}. Since $\Delta(R[U]) = \Delta_{min}(R[U])$ on those leaves (all RHS in $R[U]$ involve the exact same nodes and operations), we conclude that $\Delta(R_B) = \Delta_{min}(R_B)$. ∎

This proof is independent of the ordering of the children at step 2 of Algorithm 1. Corollary 6 is thus more general: any top-down recursive ordering keeping together RHS with identical pruned layer at each step is optimal as long as the pruned layers identified at each step are independent.

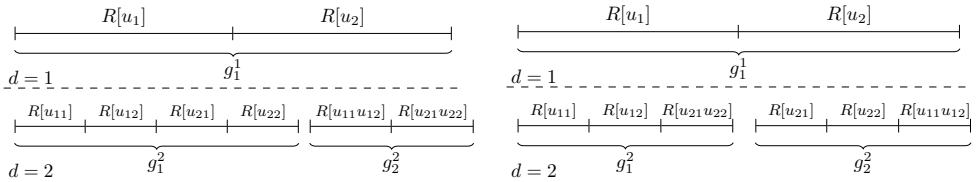


FIG. 9. *Two strategies to build groups:* CRITPATHBUILDGROUPS *(left) and* REGBUILDGROUPS *(right).*

Back to the BUILDGROUPS function, the solution of the coloring problem may not be unique. Even on the simple example of Figure 8, there are several ways to define groups, as shown in Figure 9 for g_1^1: both strategies satisfy the independence property and minimize the number of groups. The CRITPATHBUILDGROUPS strategy tends to create a large group g_1^2 and a smaller one g_2^2. In each group the computations on the tree nodes are expected to be well balanced because all branches of the tree rooted at u_0 might be covered by the RHS (assuming thus a reasonably balanced RHS distribution over the tree). The choice of CRITPATHBUILDGROUPS can be driven by tree parallelism considerations, namely, the limitation of the sum of the operation counts on the *critical paths* of all groups. The REGBUILDGROUPS strategy tends to balance the sizes of the groups but may create more unbalance regarding the distribution of work over the tree.

We note that for a given depth, applying BUILDGROUPS on *all* groups may not always be necessary, and that for a given group, enforcing the independence property may create more than two groups. In the next section we minimize the number of groups created using greedy heuristics.

4.3. A greedy approach to minimize the number of groups. Compared to Algorithm 2, Algorithm 3 adds the group selection, limits the number of groups created from a given group to two, and stops depending on a given tolerance on the amount of operations.

First, instead of stepping into each group as in Algorithm 2, we select among the current groups the one responsible for most extra computation, that is, the one maximizing $\Delta(g)-\Delta_{min}(g)$. This implies that groups that are candidates for splitting might have been created at different depths, and we use a superscript to indicate the depth d at which a group was split, as in the notation g_0^d.

Second, instead of a coloring problem which creates as many groups as colors obtained, we look (procedure BuildMaxIndepSet) inside the RHS sets of g_0^d for a maximal group of independent sets at depth $d+1$, denoted g_{imax}^{d+1}. The other sets are left in another group g_c^d, whose depth remains equal to d. g_c^d may thus consist of dependent sets that may be subdivided later if needed.[2] Rather than an exact algorithm to determine g_{imax}^{d+1}, we use a greedy heuristic.

Finally, we define μ_0 as the tolerance of extra operations authorized. With a typical value $\mu_0 = 1.01$, the algorithm stops when the number of extra operations is within 1% of the minimal number of operations Δ_{min}, returning G as the final set of groups.

Algorithm 3 Blocking algorithm.

$G \leftarrow \{R_B\}$, $\Delta_{min} \leftarrow \Delta_{min}(R_B)$, $\Delta \leftarrow \Delta(R_B)$
while $\Delta/\Delta_{min} > \mu_0$ **do**
  Select g_0^d s.t. $\Delta(g_0^d) - \Delta_{min}(g_0^d) = \max_{g\in G}(\Delta(g) - \Delta_{min}(g))$ ▷ Group selection
  $(g_{imax}^{d+1}, g_c^d) \leftarrow$ BuildMaxIndepSet(g_0^d, d+1)
  $G \leftarrow G \cup \{g_{imax}^{d+1}, g_c^d\} \setminus \{g_0^d\}$
  $\Delta \leftarrow \Delta - \Delta(g_0^d) + \Delta(g_{imax}^{d+1}) + \Delta(g_c^d)$
end while

5. Experimental results. In this section, we report on the impact of the proposed permutation and blocking algorithms on the forward substitution (see (2)), using a set of three-dimensional (3D) regular finite difference problems coming from seismic and electromagnetism modeling [1, 14], for which the solve phase is costly. The characteristics of the corresponding matrices and RHS are presented in Table 1. In both applications, the nonzeros of each RHS correspond to a small set of close points, near the top of the 3D grid corresponding to the physical domain, with some overlap between RHS. Except in section 5.3, a geometric nested dissection (ND) algorithm is used to reorder the matrix.

5.1. Impact of the flat tree algorithm. We first introduce the terminology used to denote the different strategies developed in this study and that impact the number of operations Δ. `DEN` represents the dense case where no optimization is used to reduce Δ, and `TP` means tree pruning. When column intervals are exploited at each tree node, we denote by `RAN`, `INI`, `PO`, and `FT` the random, initial ($\sigma = id$), postorder ($\sigma_{\texttt{PO}}$) and flat tree ($\sigma_{\texttt{FT}}$) permutations, respectively.

The improvements brought by the different strategies are presented in Table 2. Compared to the dense case, `TP` divides Δ by at least a factor 2. When column

[2] In case g_c^d consists of independent sets and is selected, $g_c^{d+1} = g_c^d$ will only be subdivided at depth $d+2$.

TABLE 1

Characteristics of the $n \times n$ matrix A and $n \times m$ matrix B for different test cases. $D(A) = nnz(A)/n$ and $D(B) = nnz(B)/m$ represent the average column densities for A and B, respectively.

Application	Matrix	$n(\times 10^6)$	$D(A)$	sym	m	$D(B)$
seismic modeling	5Hz	2.9	24	no	2302	567
	7Hz	7.2	25	no	2302	486
	10Hz	17.2	26	no	2302	486
electro-magnetism modeling	H0	.3	13	yes	8000	9.8
	H3	2.9	13	yes	8000	7.5
	H17	17.4	13	yes	8000	6
	H116	116.2	13	yes	8000	6
	S3	3.3	13	yes	12340	19.7
	S21	20.6	13	yes	12340	9.5
	S84	84.1	13	yes	12340	8.6
	D30	29.7	23	yes	3914	7.6

TABLE 2

Number of operations ($\times 10^{13}$) during the forward substitution ($LY = B$) according to the strategy used (ND ordering)

Δ	DEN	TP	RAN	INI	PO	FT	Δ_{min}
5Hz	1.73	.74	.74	.44	.36	**.28**	.22
7Hz	5.94	2.54	2.52	1.46	1.21	**.92**	.69
10Hz	20.62	9.01	8.92	4.78	3.85	**2.87**	2.26
H0	.39	.11	.11	.086	.070	**.057**	.050
H3	7.19	3.33	3.31	2.48	1.47	**1.26**	.95
H17	81.34	37.15	36.97	27.52	10.41	**10.21**	10.12
H116	990.02	448.31	445.91	327.89	123.79	**121.76**	120.68
S3	13.36	4.98	4.91	3.73	2.65	**2.17**	1.71
S21	156.20	49.04	48.07	35.42	25.73	**22.53**	19.43
S84	983.48	286.57	282.70	222.59	161.87	**138.56**	118.51
D30	71.60	39.78	39.38	19.49	10.93	**10.21**	7.31

intervals are exploited at each node, the large gap between RAN and INI shows that the original column order holds geometrical properties. FT behaves better than INI and PO and gets reasonably close to Δ_{min}. Overall, FT provides a 13% gain on average over PO. However, the gain on Δ decreases from 25% on the 10Hz problem to 1% on the H116 problem. This can be explained by the fact that B is denser for the seismic applications than for the electromagnetism applications (see Table 1). Indeed, the sparser the B, the closer we are to a single nonzero per RHS, in which case both FT and PO are optimal.

Second, we evaluate the impact of exploiting RHS sparsity on tree parallelism. Table 3 gives the maximal theoretical speed-up S that can be reached using tree parallelism only (node parallelism is also needed, for example, on the root). It is defined as $S = \Delta/\Delta_{cp}$, where Δ_{cp} is the number of operations on the critical path of the tree. We observe that, when sparsity is exploited, tree parallelism is significantly smaller than in the dense case. This is because the depth of the pruned tree $T_p(B)$ is similar to that of the original tree (some nonzeros of B generally appear in the leaves), while the tree effectively processed is pruned, and thus the overall amount of

TABLE 3
Theoretical tree parallelism according to the strategy used (ND ordering).

S	DEN	TP	RAN	INI	PO	FT
5Hz	8.60	3.91	3.88	3.11	2.39	**2.54**
7Hz	8.92	3.97	3.94	3.02	2.25	**2.48**
10Hz	9.10	4.04	4.02	2.96	2.30	2.30
H0	5.88	2.11	2.11	1.75	1.51	1.45
H3	5.99	3.22	3.21	2.47	2.02	2.11
H17	6.32	3.34	3.32	2.54	2.00	1.97
H116	7.92	3.63	3.61	2.75	2.05	2.02
S3	6.12	2.84	2.83	2.18	1.73	1.61
S21	6.30	2.56	2.46	1.85	1.49	1.47
S84	8.01	2.41	2.38	1.90	1.53	1.52
D30	8.50	4.73	4.70	2.86	2.05	2.56

TABLE 4
Impact of the number of groups NG on the normalized operation count until Δ_{NG}/Δ_{min} becomes smaller than the tolerance $\mu_0 = 1.01$ (ND ordering).

Δ_{NG}/Δ_{min}	FT	NG= 2	NG= 3	NG= 4	NG= 5
5Hz	1.283	1.111	1.001	x	x
7Hz	1.321	1.116	1.002	x	x
10Hz	1.269	1.029	1.002	x	x
H0	1.148	1.029	1.010	1.002	x
H3	1.329	1.068	1.027	1.005	x
H17	1.009	x	x	x	x
H116	1.009	x	x	x	x
S3	1.275	1.120	1.045	1.012	1.003
S21	1.160	1.037	1.015	1.003	x
S84	1.169	1.041	1.015	1.002	x
D30	1.397	1.082	1.058	1.024	1.004

operations is reduced. For the same reason, S is smaller for test cases where $D(B)$ is small. For the 5Hz, 7Hz, and 10Hz problems which have more nonzeros per column of B, besides decreasing the operation count more than the other strategies, FT exhibits equivalent or even better tree parallelism than PO. For such matrices where $D(B)$ is large, FT balances the work on the tree better than PO and reduces the work on the critical path more than the total work. Overall, FT reduces the operation count better than any other strategy and has good parallel properties.

5.2. Impact of the blocking algorithm. First, we show that the blocking algorithm decreases the operation count Δ while creating a small number of groups. Second, we discuss parallel properties of the clustering strategies illustrated in Figure 9. In Table 4, we report the value of Δ_{NG}/Δ_{min} as a function of the number of groups created. x means that the blocking algorithm stopped because the condition $\Delta_{NG}/\Delta_{min} \leq \mu_0$ was reached, with μ_0 for Algorithm 3 set to 1.01. Computing from Table 4 the ratio of extra operations reduction $1 - \frac{\Delta_{NG}-\Delta_{min}}{\Delta_1-\Delta_{min}}$ for NG groups created, we observe an average reduction of 74% of the extra operations when $NG = 2$, i.e., when only two groups are created. Table 4 also shows that Δ_{NG} reaches a value close to Δ_{min} very quickly.

TABLE 5

Sum of critical paths' operations ($\times 10^{13}$) for two grouping strategies when three groups are created.

$\sum_g \Delta_{cp}(g)$	5Hz	7Hz	10Hz	H0	H3
CritPathBuildGroup	**.092**	**.30**	**1.00**	**.037**	**.50**
RegBuildGroup	.12	.43	1.58	.044	.72

TABLE 6

Operation count $\Delta(\times 10^{13})$ for permutation strategies `PO` and `FT`, and number of groups NG required to reach $\frac{\Delta_{NG}}{\Delta_{min}} \leq 1.01$ for blocking strategies `REG` and `BLK`. Different orderings (AMD, AMF, SCOTCH, METIS) are used.

	AMD					AMF					SCOTCH					METIS				
σ	PO	REG	FT	BLK	Δ_{min}	PO	REG	FT	BLK	Δ_{min}	PO	REG	FT	BLK	Δ_{min}	PO	REG	FT	BLK	Δ_{min}
	Δ	NG	Δ	NG		Δ	NG	Δ	NG		Δ	NG	Δ	NG		Δ	NG	Δ	NG	
5Hz	1.44	53	**1.36**	4	1.25	**.75**	51	.87	7	.68	.47	328	**.32**	3	.25	.43	230	**.30**	3	.24
7Hz	5.03	38	**4.44**	4	4.35	**15.3**	18	17.8	12	14.9	1.60	287	**1.14**	3	.86	1.42	230	**1.08**	3	.82
10Hz	**19.3**	154	19.8	3	15.3	**96.9**	253	99.1	11	82.1	5.86	288	**4.21**	3	3.05	4.67	281	**3.44**	3	2.53
H0	.54	533	**.53**	4	.47	**.12**	333	.12	5	.091	.073	499	**.063**	3	.055	.077	615	**.067**	3	.057
H3	135	380	**106**	5	9.07	**101**	63	134	17	9.26	2.19	615	**1.76**	5	1.18	1.95	533	**1.53**	5	1.12
H17	183	266	**226**	7	136	**467**	173	558	50	396	22.8	380	**19.0**	5	12.6	21.49	242	**16.8**	4	12.3
H116	2244	1	2244	1	2244	**39383**	1	39384	1	39383	291	109	**224**	4	153	264	78	**215**	4	157
S3	20.9	725	**17.9**	6	15.1	**20.7**	184	24.8	10	17.8	4.5	771	**3.41**	5	2.72	3.24	771	**2.64**	5	2.09
S21	393	685	**349**	5	311	**1141**	493	1352	77	831	50.9	492	**39.6**	4	31.7	34.3	223	**28.5**	5	24.9
S84	3025	352	**2848**	5	2501	**38664**	725	45346	213	30977	289	286	**228**	4	193	207	171	**174**	4	151
D30	**115**	111	121	8	94.5	**1015**	139	1280	75	825	16.7	156	**12.8**	5	8.77	15.5	144	**13.0**	5	8.61

In Table 5, we report the sum of operation counts on the critical paths Δ_{cp} over all groups created using CritPathBuildGroup and RegBuildGroup strategies, when the number of groups created is three, leading to a value Δ close to Δ_{min}; see column "NG=3" of Table 4. In this case, the total number of operations Δ during the forward solution phase on all groups is equal regardless of whether we use CritPathBuildGroup or RegBuildGroup. Tree parallelism is thus a crucial discriminant between both strategies, and we indeed observe in Table 5 that CritPathBuildGroup effectively limits the length of critical paths over the three groups created, justifying its use.

5.3. Experiments with other orderings. As mentioned earlier, several orderings may be used to order the unknowns of the original matrix thanks to the algebraic nature of our flat tree and blocking algorithms. Although local ordering methods (AMD, AMF as provided by the MUMPS package)[3] are known not to be competitive with respect to algebraic nested dissection-based approaches such as SCOTCH[4] or METIS[5] on large 3D problems, we include them in Table 6 in order to study how the flat tree and blocking algorithms behave in general situations.

First, an important aspect of using other orderings is that they often produce much more irregular trees, leading to a large number of pruned layers to sequence. The `FT` permutation reduces the operation count significantly with SCOTCH and METIS, for which we observe an average 31% and 26% reduction compared to the `PO` permutation. Gains are also obtained with AMD for most test cases. However, `FT` does not perform well with AMF. This can be explained by the fact that AMF produces too irregular trees which do not fit well with our design of the `FT` strategy.

Second, we evaluate the blocking algorithm (`BLK`) and compare it with a regular blocking algorithm (`REG`) based on the `PO` permutation that divides the initial set of

[3] http://mumps.enseeiht.fr/

[4] http://www.labri.fr/perso/pelegrin/scotch/

[5] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

TABLE 7

Time (seconds) of the forward substitution according to the strategy used and number of groups NG created with `BLK` *and* `REG` *(ND ordering). The percentage of actual computation time (BLAS) is indicated in parentheses.*

	DEN	TP	INI	PO	REG	FT	BLK	NG_{BLK}	NG_{REG}
5Hz	3132 (34)	561 (80)	305 (88)	246 (89)	429 (95)	190 (89)	148 (91)	3	328
7Hz	9101 (40)	1727 (89)	951 (93)	784 (94)	1137 (97)	594 (94)	460 (95)	3	255
H0	862 (58)	161 (82)	125 (85)	97 (89)	88 (96)	79 (89)	67 (92)	4	328
H3	12419 (72)	4478 (92)	3274 (94)	1901 (96)	1709 (98)	1624 (96)	1234 (97)	4	306
S3	26328 (64)	6839 (89)	5005 (92)	3429 (95)	3450 (99)	2804 (96)	2195 (97)	5	536

TABLE 8

Operation count ($\times 10^{13}$) and time (seconds) of `REG` *for different block sizes.*

REG	64		128		256		512		1024	
	Δ	T_s	Δ	T_s	Δ	T_s	Δ	T_s	Δ	T_s
5Hz	.24	188	.25	179	.27	188	.27	188	.31	209
7Hz	.74	558	.78	538	.86	575	.93	611	.98	643
H0	.051	77	.051	74	.052	72	.054	74	.058	79
H3	.98	1447	1.02	1405	1.04	1369	1.09	1407	1.16	1499
S3	1.75	2533	1.78	2448	1.80	2358	1.86	2379	1.91	2424

columns into regular chunks of columns. Table 6 shows that the number of groups required to reach $\Delta/\Delta_{min} \leq 1.01$ is much smaller for `BLK` than for `REG` in all cases. Our blocking algorithm is very efficient with most orderings except AMF, where the number of groups created is high (but still lower than `REG`).

5.4. Performance. We analyze in this section the time for the forward substitution and for the flat tree and blocking algorithms on a single Intel Xeon core @2.2GHz. A performance analysis in multithreaded or distributed environment for the largest problems is outside the scope of this study. In Table 7, we report the time of the forward substitution of the MUMPS solver[6] and the percentage of time spent in BLAS operations, excluding the time for data manipulation and copies. Timings with the regular blocking `REG` to reach our target number of operations ($\Delta_{NG}/\Delta_{min} \leq 1.01$), at the cost of a larger number of groups, $NG \gg 1$, are also indicated.

Table 7 shows that the time reduction is in agreement with the operations reduction reported in Table 2. One can also notice that when exploiting sparsity, the proportion of time spent in BLAS operations increases. This is due to the fact that the relative weight of the top of the tree (with larger fronts for which time is dominated by BLAS operations) is larger when sparsity is exploited. When targeting a reduction of the number of operations with `REG`, the large number of blocks (last column of Table 7) makes the granularity of the BLAS operations too small to reach the performance of the `BLK` strategy. On the other hand, Table 8 shows that larger regular blocks improve the Gflop rate but are not competitive with respect to `BLK` because of the increase in the number of operations. We also observe that the best block size for `REG` is problem-dependent.

Table 9 relates the execution time of the `FT` and `BLK`. The execution times are reasonable compared to the corresponding execution time of the forward substitution. Moreover, we observe that the time of the flat tree algorithm only slowly increases

[6] http://mumps-solver.org

TABLE 9
Time (seconds) of the flat tree and blocking algorithms for different orderings.

	ND		AMD		AMF		SCOTCH		METIS	
	FT	BLK	FT	BLK	FT	BLK	FT	BLK	FT	BLK
5Hz	.77	.11	.64	.19	1.35	.34	.79	.20	1.02	.13
7Hz	1.30	.30	1.56	.62	5.68	3.9	1.85	.56	2.48	.34
H0	.09	.04	1.72	.04	.13	.04	.09	.02	.12	.04
H3	.52	.26	1.56	.29	1.15	3.0	.55	.24	1.15	.54
S3	.88	.33	2.20	.41	2.51	1.3	.84	.36	1.75	.49

with the problem size. The time of the blocking algorithm, due to its limited number of iterations, is not critical.

6. Guided nested dissection. Given an ordering and a tree, one may think of moving the unknowns corresponding to RHS nonzeros to supernodes higher in the tree with, on the one hand, a smaller pruned tree, but on the other hand, an increase in the factor size due to larger supernodes. Better, one may guide the ordering to include as many nonzeros of B as possible within separators during top-down ND and prune larger subtrees. This will, however, involve a significant extra cost for applications where each RHS contains several contiguous nodes in the grid, e.g., form a small parallelepiped. For such applications, the geometry of the RHS nonzeros could, however, be exploited. A first idea avoids problematic RHS by choosing separators that *do not* intersect RHS nonzeros. Although this idea could be tested by adding edges between RHS nonzeros before applying SCOTCH or METIS, this does not appear to be quite so useful in our applications, where we observed significant overlap between successive RHS. Another idea, when all RHS are localized in a specific area of the domain, is to shift the separators from the nested dissection to insulate the RHS in a small part of the domain. Such a modification of the ordering yields an unbalanced tree in which the RHS nonzeros appear at the smaller side of the tree, improving the efficiency of tree pruning and resulting in a reduction of Δ_{min}, and thus Δ. This so-called *guided* ND was implemented and tested on the set of test cases shown in Table 10, where we observe that the number of operations Δ_{min} is decreased, as expected. Since the factor size has also increased significantly, a trade-off may be needed to avoid increasing too much the cost of the factorization.

TABLE 10
Number of operations Δ_{min} and factor size for original (ND) and guided (GND) nested dissection orderings.

Matrices	5Hz		7Hz		10Hz		H0		H3	
Strategy	ND	GND	ND	GND	ND	GND	ND	GND	ND	GND
$\Delta_{min}(\times 10^{13})$	.22	**.19**	.69	**.62**	2.26	**1.99**	.050	**.025**	.95	**.81**
factor size ($\times 10^9$)	**3.72**	5.18	**12.8**	19.7	**44.8**	73.4	**.24**	.37	**4.50**	5.57

7. Applications and related problems. We describe applications where our contributions can be applied. When only part of the solution is needed, one can show that the approaches described in section 2 can be applied to the backward substitution ($UX = Y$), which involves mechanisms similar to the forward substitution [15, 17].

The backward substitution traverses the tree nodes from top to bottom so that the interval mechanism is reversed, i.e., the interval from a parent includes the intervals from its children and the properties of local optimality are preserved. If the structure of the partial solution requested differs from the RHS structure, another call to the flat tree algorithm must then be performed to optimize the number of operations. Exploiting sparsity also in the backward step can, for instance, be useful in some augmented approaches [18] to deal with small matrix updates without complete refactoring, and in some 3D EM geophysics applications [14]. Another application of this work is the computation of Schur complements, where instead of truncating a factorization of the whole system $\left(\begin{smallmatrix} A & C \\ B & D \end{smallmatrix}\right)$, one exploits the factorization of A to use triangular solves with sparse RHS. Taking the symmetric case where $C = B^T$, the Schur complement S can be written $S = D - BA^{-1}B^T = D - B(LL^T)^{-1}B^T = D - (L^{-1}B^T)^T(L^{-1}B^T)$, as in the PDSLin solver [16]. Since B is sparse, $B' = L^{-1}B^T$ can be computed thanks to the algorithms developed in this article before computing the sparse product $B'^T B'$.

Finally, we comment on related problems and algorithms. We indicated that the blocking algorithm is closely related to graph algorithms like coloring and maximum independent set. Concerning the minimization problem (9) which we addressed with the flat tree algorithm, it can also be regarded globally: using the structure of $L^{-1}B$, the objective is to find a permutation of the columns that minimizes the sum of the intervals weighted with δ_u. This interval minimization problem is similar to a sparse matrix profile reduction problem [5, 13] (and we thus suspect it to be NP-complete). As mentioned in the introduction, hypergraph models have been used in the context of blocking algorithms, with different constraints and objectives compared to ours [2, 16]. Modeling $L^{-1}B$ as a hypergraph might lead to heuristics other than the flat tree algorithm using some variants of hypergraph partitioning, although dense parts in $L^{-1}B$ might need a specific treatment. One advantage of our permutation and blocking algorithms is that, instead of tackling the problem globally, they decompose the problem into easier subproblems with low complexity by making use of the separator tree T, thereby exploiting the fact that $L^{-1}B$ has a very special structure closely related to the tree. In the context of general unsymmetric matrices, the structure of the solution of the forward step is given by the set of reachable vertices in the elimination dag of L^T [11]. To make the elimination dag a tree to be used in our context, one could consider adding a limited number of entries in L. Similarly to the case of matrices with a symmetric or quasi-symmetric pattern for which the elimination tree of the matrix $A + A^T$ is used, one could add entries in L having a symmetric counterpart in U. Another possibility is to use the work presented in [8] that extends the notion of elimination tree to unsymmetric matrices by considering paths in the factor matrices to characterize the elimination tree. How useful this generalization of the elimination tree can be in our context would deserve to be further studied.

Conclusion. We introduced permutation and blocking algorithms to improve the tree pruning [15] and the node interval [3] algorithms. A first contribution is the "flat tree" algorithm which permutes RHS to reduce the cost of the forward substitution. A second contribution is a blocking algorithm that further decreases this cost by adequately choosing groups of RHS that can be processed together. Although both algorithms are based on geometrical observations, they are designed with an algebraic approach, giving a general scope to this work. Notions of node optimality and RHS independence were introduced and formalized, together with theoretical properties to provide insight and to support the proposed algorithms. Experimental results on real test cases showed the effectiveness of both the flat tree and the blocking algorithms.

Compared to a postorder-based permutation, the flat tree permutation showed an average (resp., maximum) gain of 13% (resp., 25%) on the total operation count with a nested dissection ordering and interesting parallel properties. Moreover, results with the blocking algorithm validated our approach since only a handful of groups is created, compared to several hundreds when using a regular blocking technique. Finally, sequential performance results confirmed the good potential of the proposed approaches.

Acknowledgments. We thank the referees for their very constructive remarks, EMGS and Seiscope for providing the test cases, and F.-H. Rouet for his comments on an early version of this paper.

REFERENCES

[1] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto, *Fast 3D frequency-domain full waveform inversion with a parallel Block Low-Rank multifrontal direct solver: Application to OBC data from the North Sea*, Geophysics, 81 (2016), pp. R363–R383.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar, *On computing inverse entries of a sparse matrix in an out-of-core environment*, SIAM J. Sci. Comput., 34 (2012), pp. A1975–A1999, https://doi.org/10.1137/100799411.

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and F.-H. Rouet, *Parallel computation of entries of A^{-1}*, SIAM J. Sci. Comput., 37 (2015), pp. C268–C284, https://doi.org/10.1137/120902616.

[4] P. R. Amestoy, J.-Y. L'Excellent, and G. Moreau, *On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers*, Research report RR-9122, INRIA, 2017.

[5] M. W. Berry, B. Hendrickson, and P. Raghavan, *Sparse matrix reordering schemes for browsing hypertext*, Lecture Notes in Applied Mathematic, 32 (1996), pp. 99–124.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17, https://doi.org/10.1145/77626.79170.

[7] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, 2nd ed., Oxford University Press, London, 2017.

[8] S. C. Eisenstat and J. W. H. Liu, *The theory of elimination trees for sparse unsymmetric matrices*, SIAM J. Matrix Anal. Appl., 26 (2005), pp. 686–705.

[9] A. George, *Nested dissection of a regular finite-element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363, https://doi.org/10.1137/0710032.

[10] J. R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM Journal on Matrix Analysis and Applications, 15 (1994), pp. 62–79.

[11] J. R. Gilbert and J. W. H. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 334–352, https://doi.org/10.1137/0614024.

[12] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172, https://doi.org/10.1137/0611010.

[13] J. K. Reid and J. A. Scott, *Reducing the total bandwidth of a sparse unsymmetric matrix*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 805–821, https://doi.org/10.1137/050629938.

[14] D. Shantsev, P. Jaysaval, S. de la Kethulle de Ryhove, P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, *Large-scale 3D EM modeling with a Block Low-Rank multifrontal direct solver*, Geophys. J. Int., 209 (2017), pp. 1558–1571.

[15] Tz. Slavova, *Parallel Triangular Solution in the Out-Of-Core Multifrontal Approach for Solving Large Sparse Linear Systems*, Ph.D. dissertation, Institut National Polytechnique de Toulouse, Toulouse, France, 2009.

[16] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar, *On partitioning and reordering problems in a hierarchically parallel hybrid linear solver*, in Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Cambridge, MA, 2013, pp. 1391–1400.

[17] Y.-H. Yeung, J. Crouch, and A. Pothen, *Interactively cutting and constraining vertices in meshes using augmented matrices*, ACM Trans. Graph., 35 (2016), 18, https://doi.org/10.1145/2856317.

[18] Y. H. Yeung, A. Pothen, M. Halappanavar, and Z. Huang, *AMPS: An augmented matrix formulation for principal submatrix updates with application to power grids*, SIAM J. Sci. Comput., 39 (2017), pp. S809–S827, https://doi.org/10.1137/16M1082755.