



**HAL**  
open science

## Parallel QR factorization of block-tridiagonal matrices

Alfredo Buttari, Søren Hauberg, Costy Kodsı

► **To cite this version:**

Alfredo Buttari, Søren Hauberg, Costy Kodsı. Parallel QR factorization of block-tridiagonal matrices. SIAM Journal on Scientific Computing, 2020, 42 (6), pp.C313-C334. 10.1137/19M1306166 . hal-02370953v2

**HAL Id: hal-02370953**

**<https://hal.science/hal-02370953v2>**

Submitted on 5 Nov 2020

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# PARALLEL $QR$ FACTORIZATION OF BLOCK-TRIDIAGONAL MATRICES

A. BUTTARI <sup>\*</sup>, SØREN HAUBERG <sup>†</sup>, AND COSTY KODSI <sup>†</sup>

**Abstract.** In this work, we deal with the  $QR$  factorization of block-tridiagonal matrices, where the blocks are dense and rectangular. This work is motivated by a novel method for computing geodesics over Riemannian manifolds. If blocks are reduced sequentially along the diagonal, only limited parallelism is available. We propose a matrix permutation approach based on the Nested Dissection method which improves parallelism at the cost of additional computations and storage. We show how operations can be arranged to keep this extra cost as low as possible. We provide a detailed analysis of the approach showing that this extra cost is bounded. Finally, we present an implementation for shared memory systems relying on task parallelism and the use a runtime system. Experimental results support the conclusions of our analysis and show that the proposed approach leads to good performance and scalability.

**Key words.**  $QR$  factorization, nested dissection, task-based parallelism

**AMS subject classifications.** 68W10, 68W40, 65F05, 65F20

**1. Introduction.** In this work, we deal with the solution of linear least squares problems where the system matrix is block-tridiagonal. By this we mean that our matrices have a block structure with  $c$  block-rows and  $c$  block-columns and along block-row  $k$  we only have nonzeros in block-columns  $k - 1$  to  $k + 1$  as depicted in Figure 1. The blocks are assumed to be dense and rectangular of size  $m \times n$  with  $m > n$ .

In a LAPACK-like  $QR$  factorization where sub-diagonal coefficients are eliminated by means of Householder reflections column-by-column in the natural order, only limited parallelism is available because of the lack of update operations (i.e., application of the Householder reflections to the trailing submatrix) stemming from the potentially narrow bandwidth and because each elimination step depends on the previous. Even the use of *tiled* [11] or *communication-avoiding* [14] approaches can barely improve the situation especially when  $c$  is high and  $m$  as well as  $n$  is small. By choosing a different order for reducing the matrix columns, which essentially amounts to permuting the matrix columns, it is possible to achieve better parallelism; this, however, generates additional *fill-in* — coefficients that are zero in the original matrix and are turned into nonzeros by the factorization — which is responsible for an increase in the operation count and the storage. We propose an approach that computes this permutation by applying the Nested Dissection method to the compressed matrix graph. This allows us to identify groups of blocks that can be reduced independently; this first phase is followed by a reduction step that deals with the blocks that are at the interface of the groups. Although the structure and value of the  $R$  factor only depends on column permutations, the structure of the  $Q$  factor and, consequently, the overall cost of the factorization greatly depend on the order in which coefficients are annihilated within each column or, more generally, on row permutations. Taking this into account, we propose an approach based on variable pivoting that reduces the operational complexity by avoiding unnecessary computations, especially in the reduction phase which involves communications. We provide a detailed analysis of the operational and memory cost showing that the overhead

---

<sup>\*</sup>Université de Toulouse, CNRS-IRIT

<sup>†</sup>DTU, Department of Applied Mathematics and Computer Science

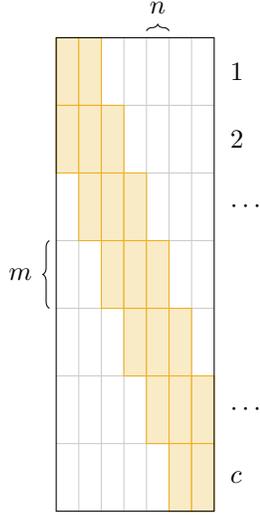


FIG. 1. A block-tridiagonal matrix.

45 depends on how many nested dissection levels are used but is bounded by a modest  
 46 constant. Finally, we present a parallel implementation for shared memory systems.  
 47 This is based on the use of task parallelism where the whole workload is represented on  
 48 the form of a *Directed Acyclic Graph* (DAG) of tasks which is automatically created  
 49 and scheduled by a runtime system. This implementation is achieved using features  
 50 of the `qr_mumps` software [1] which relies on the StarPU runtime system [5]. Exper-  
 51 imental results obtained with this implementation show that the proposed method  
 52 achieves good performance and scalability on a 36 cores system and has the potential  
 53 to perform even better on large-scale distributed memory systems.

54 **1.1. Motivation.** One of the key challenges in *machine learning* (ML) is to  
 55 learn (i.e., estimate) a representation of data that is suitable for a given task [6].  
 56 Learned representations for *supervised ML* tasks, such as classification and regression,  
 57 are optimal only for specific tasks. In contrast, representations for *unsupervised ML*  
 58 tasks, such as data exploration, are concerned with the underlying structure governing  
 59 the observed data.

60 It has been shown in [19, 4] that an effective approach for unsupervised ML in-  
 61 volves treating the learned representation as a *Riemannian manifold*. An unfortunate  
 62 implication, however, is that vector space operations are no longer applicable. Instead,  
 63 Riemannian counterparts, such as *logarithm* and *exponential maps* take the place of  
 64 subtraction and addition operations. Also, *geodesics* in their role as generalizations  
 65 of line segments in Euclidean space become prominent. Just as a segment of a line  
 66 is the shortest distance connecting two distinct points, a geodesic on a Riemannian  
 67 manifold is a smooth parametric curve that is a local length minimizer connecting any  
 68 two distinct points. Additionally, a point moving along a geodesic does not experience  
 69 acceleration in the tangent direction, i.e., the velocity has a constant magnitude.

70 Consider now an  $n$ -dimensional manifold embedded in  $m$ -dimensional Euclidean  
 71 space (with  $m > n$ ) described by the parametric map

$$y : \mathbb{R}^n \rightarrow \mathbb{R}^m.$$

73 A rather simple and intuitive strategy for determining a geodesic given any two dis-  
74 tinct points on a Riemannian manifold requires working with the discretized version  
75 of a smooth parametric curve. It is then possible to model the curve as a *series of*  
76 *connected linear elastic springs*. A spring force  $f \in \mathbb{R}^m$  is related to the end points  
77  $y(x_i), y(x_j) \in \mathbb{R}^m$  of a curve segment by  $f = K(y(x_j) - y(x_i))$ , in which  $x \in \mathbb{R}^n$  and  
78  $K$  is the identity matrix representing the stiffness. Even though the springs are linear  
79 in nature, the parameterization introduces potential nonlinearity. Imposing boundary  
80 conditions (constituting the given points) on the assembled block tridiagonal (total)  
81 stiffness matrix as well as force vector, a system of equations can be solved for the  
82 unknown curve points yielding the stable equilibrium of the spring assemblage and  
83 geodesic. If a system of nonlinear equations has to be tackled, which is the working  
84 assumption, Newton’s method or a variation of Newton’s method is a popular choice.  
85 In each iteration of Newton’s method, the solution of an overdetermined system of  
86 linear equations featuring a full-rank Jacobian is sought as a trial step. It is possible  
87 to multiply the transpose of the Jacobian to both sides of the overdetermined system  
88 for symmetry purposes in preparation for solution. Since the Jacobian is likely to  
89 be poorly conditioned, this is not desirable as it has the effect of squaring the con-  
90 dition number of the problem. Alternatively, a solution methodology involving  $QR$   
91 factorization of the Jacobian does not unnecessarily degrade the conditioning of the  
92 problem.

93 **1.2. Related work.** Our work can be related to the factorization of banded,  
94 (block) bi or tridiagonal matrices. Most of the approaches presented in the literature  
95 draw parallelism from *cyclic reduction* [22], *wrap-around* [20], partitioning methods  
96 or minor variants thereof; these techniques amount to identifying independent sets  
97 of computations that allow for reducing multiple (block) columns at once at the cost  
98 of additional computations due to fill-in. The method we propose in this work relies  
99 on the use of nested dissection (see Sections 2.1 and 2.2) and, essentially, leads to  
100 permutations that are equivalent to those computed with cyclic reduction or wrap-  
101 around; because nested dissection relies on graph theory, we believe it provides a  
102 clearer framework that allows for a better understanding and analysis of parallelism  
103 and fill-in.

104 Early work on the  $QR$  factorization of square tridiagonal systems by Givens ro-  
105 tations is proposed by Sameh and Kuck [28]. Computations are organized in two  
106 stages. In the first diagonal blocks are reduced independently; this introduces fill-in  
107 at the interface of contiguous diagonal blocks which is reduced in the second stage.  
108 These ideas eventually lead to the definition of *spike* [27, 25] algorithm for the  $LU$   
109 factorization of banded matrices. This approach was also used by Berry and Sameh  
110 [7] to compute the parallel  $LU$  factorization of block-tridiagonal, diagonally dominant  
111 (i.e., they do not require pivoting) matrices.

112 Arbenz and Hegland [3] propose a method for the stable solution of (periodic)  
113 square banded linear systems. The factorization is achieved in two phases assuming  
114 that the original matrix is permuted into *standard form* (i.e., lower-banded periodic)  
115 through a circular shift; this permutation introduces some unnecessary fill-in. In the  
116 first phase, a partitioning of the matrix (which amounts to a dissection step) and  
117 block-columns associated with the resulting parts are reduced independently. This  
118 leads to a periodic block-bidiagonal system which is reduced in parallel using cyclic  
119 reduction. They discuss both the  $LU$  and  $QR$  factorization although mostly focus on  
120 the first. Note that the  $QR$  factorization of square, periodic block-bidiagonal systems  
121 through cyclic reduction is also addressed by Hegland and Osborne [21]

122 None of the aforementioned approaches consider the case of overdetermined sys-  
 123 tems; this has non-trivial implications because when a block-column is reduced, some  
 124 coefficients along the corresponding pivotal block-row remain which have to be re-  
 125 duced. Also, all of these methods only draw parallelism from the matrix permutation  
 126 or partitioning whereas our approach exploits multiple levels of parallelism as de-  
 127 scribed in Section 3.

128 All of the methods referenced, including ours, are, in essence, specialized mul-  
 129 tifrontal methods [15]. Algebraic multifrontal methods for the  $QR$  factorization of  
 130 sparse matrices have been proposed in the literature by Amestoy, Duff, and Puglisi  
 131 [2], Davis [13] and Buttari [10], for instance. These methods and the correspond-  
 132 ing tools are designed for generic sparse matrices; although they can obviously be  
 133 used for solving block-tridiagonal systems, they may not be the best suited tools.  
 134 Indeed, unlike common sparse matrices, our block-tridiagonal systems may be rela-  
 135 tively dense in the beginning (multiple thousands of coefficients per row) and fill up  
 136 moderately during the factorization (see Section 2.4). As a result, algebraic sparse  
 137 multifrontal solvers may suffer from the excessive overhead due to the handling of  
 138 the coefficients of the original matrix based on indirect addressing. Clearly, these  
 139 methods could be extended to efficiently handle block matrices; this is the object of  
 140 future work. Additionally, multifrontal methods involve explicit assembly of frontal  
 141 matrices which can be avoided in our case due to the simple, regular structure of the  
 142 data (see Section 2.3). Finally, in sparse multifrontal solvers, frontal matrices are  
 143 typically reduced using a LAPACK-like factorization which does not fully take into  
 144 account their possibly sparse nature as explained in Section 2.3.

145 **2. Algorithm.** In this section we describe our approach to parallelize the  $QR$   
 146 factorization of a block-tridiagonal matrix. Our description and analysis will rely on  
 147 the theory of sparse matrix factorizations. The necessary theoretical background is  
 148 briefly described in the next section; we refer the reader to the cited documents for a  
 149 detailed discussion of this topic.

150 **2.1. Preliminaries.** As it is commonly done in the literature [12, 10], our sym-  
 151 bolic analysis of the  $QR$  factorization of a sparse matrix relies on the equivalence  
 152 between the  $R$  factor of a real matrix  $A$  and the Cholesky factor of the normal equa-  
 153 tion matrix  $B = A^T A$ , whenever the Strong Hall property holds. As a result of  
 154 this equivalence, the  $QR$  factorization of a sparse matrix  $A$  follows the same com-  
 155 putational pattern as the Cholesky factorization of  $B$ . This can be modeled us-  
 156 ing the *adjacency graph* of  $B$  defined as a graph  $\mathcal{G}(B) = (\mathcal{V}, \mathcal{E})$  whose vertex set  
 157  $\mathcal{V} = \{1, 2, \dots, cn\}$  includes the unknowns associated with rows and columns of  $B$  and  
 158 edge set  $\mathcal{E} = \{(i, j) \mid \forall b_{i,j} \neq 0\}$  includes an edge for each nonzero coefficient of  $B$ . In  
 159 our case, the symmetry of  $B$  implies that if  $(i, j)$  is in  $\mathcal{E}$ , then so is  $(j, i)$ , and therefore  
 160 we will only represent one of these edges; such a graph is called *undirected*.

161 It is possible to use the adjacency graph to model the Cholesky factorization of  
 162  $B$ . Specifically we can build a sequence of graphs  $\mathcal{G}(B) = \mathcal{G}_0(B), \mathcal{G}_1(B), \dots, \mathcal{G}_{cn-1}(B)$ ,  
 163 called *elimination graphs*, such that  $\mathcal{G}_k(B)$  is the adjacency graph of the trailing  
 164 submatrix after elimination of variables  $1, 2, \dots, k$ . Elimination graph  $\mathcal{G}_k(B)$  is built  
 165 from  $\mathcal{G}_{k-1}(B)$  by removing node  $k$  as well as all its incident edges and by updating  
 166 the connectivity of the remaining nodes: for any two nodes  $i$  and  $j$  that are neighbors  
 167 of  $k$ , we have to add an edge connecting them if it does not exist already. One such  
 168 edge, called a *fill edge*, models the occurrence of a new *fill-in* coefficient.

169 A sparse factorization can achieve better parallelism than a dense one because  
 170 sparsity implies that the elimination of one unknown does not affect all the unlimi-

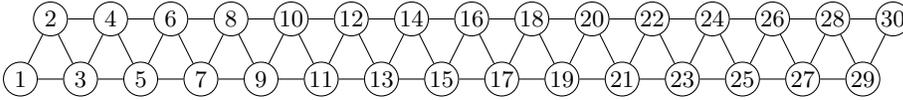


FIG. 2. *Compressed graph for the normal equation matrix  $B = A^T A$ . The label inside nodes show the unknown number which, in this case, corresponds to the order in which unknowns are eliminated.*

171 nated ones but only part of them. The dependencies between variables are represented  
 172 by the so-called *directed filled graph* which corresponds to the original graph  $\mathcal{G}(B)$  plus  
 173 all the fill edges; in this graph an edge  $(i, j)$  is directed from  $i$  to  $j$  if  $j$  is eliminated  
 174 after  $i$  and expresses the fact that  $j$  depends on  $i$ . Because many of the dependencies  
 175 expressed by the filled graph are redundant, its transitive reduction is computed to  
 176 obtain the smallest complete set of dependencies; this results in a tree graph called  
 177 an *elimination tree*. The elimination tree plays a central role in sparse matrix factor-  
 178 izations [24] because it dictates the order in which the unknowns can be eliminated:  
 179 any order that follows a topological (i.e., bottom-up) traversal of the tree leads to the  
 180 same result. A consequence of this fact is that nodes belonging to different branches  
 181 are independent and can, thus, be eliminated in parallel. Nodes of the elimination  
 182 tree are commonly amalgamated into supernodes containing nodes that share the  
 183 same neighbors in the filled graph and can thus be eliminated at once; the resulting  
 184 amalgamated tree is commonly referred to as an *assembly tree*.

185 *Nested Dissection* is a method introduced by George [17] to reduce the complexity  
 186 of sparse factorizations. The basic step of this method is based on the idea of com-  
 187 puting a separator  $\mathcal{S}$  of a graph  $\mathcal{G}$ ; this is defined as a subset of nodes which splits the  
 188 graph into two non-connected subgraphs  $\mathcal{G}_1$  and  $\mathcal{G}_2$ . If the nodes in  $\mathcal{S}$  are eliminated  
 189 last, no fill-in is possible between the nodes of  $\mathcal{G}_1$  and those of  $\mathcal{G}_2$  because of the Rose,  
 190 Tarjan and Lueker theorem [26]. This generally reduces the overall amount of fill-in  
 191 and improves parallelism because the nodes of  $\mathcal{G}_1$  can be eliminated independently,  
 192 and, thus in parallel, than those of  $\mathcal{G}_2$ . Nested Dissection applies this procedure re-  
 193 cursively until subgraphs of a given minimum size are achieved. The resulting tree of  
 194 separators matches the assembly tree.

195 **2.2. A Nested Dissection based approach.** Because our matrices are made  
 196 of dense blocks, our analysis can focus on the *compressed graph* where a node repre-  
 197 sents all the unknowns in a block-column and an edge represents a set of edges that  
 198 connect all the unknowns in one block-column to all those in another block-column.  
 199 Figure 2 shows the compressed graph for the normal equation matrix  $B = A^T A$ ; this  
 200 can be easily derived knowing that  $b_{i,j} \neq 0$  if there exist a row  $k$  in  $A$  such that  $a_{k,i} \neq 0$   
 201 and  $a_{k,j} \neq 0$ . Because all the columns (rows) within a block-column (block-row) are  
 202 indistinguishable, we will use the term “column” (“row”) to refer to a block-column  
 203 (block-row).

204 A few observations can be made about this graph. First, regardless of the order in  
 205 which unknowns are eliminated, some fill-in will appear in  $R$  compared to  $A$ . In fact,  
 206 although block-row  $k$  of  $A$  only has nonzero blocks in block-columns  $k - 1$  to  $k + 1$ ,  $R$   
 207 will also have a nonzero block in block-column  $k + 2$ ; these fill-in blocks are represented  
 208 by the horizontal edges in Figure 2. We will refer to these fill blocks as *unavoidable*  
 209 because, as said above, they will exist in  $R$  regardless of the nodes elimination order.  
 210 Second, besides these unavoidable fill-in blocks, no additional fill-in is introduced in  $R$   
 211 by the factorization if unknowns are eliminated in the natural order. This is because,

212 upon elimination of node  $k$ , its neighbors are already connected to each other. Third,  
 213 the transitive reduction of the graph in Figure 2 leads to a linear graph, i.e., a tree  
 214 with a single branch which does not provide any opportunities for parallelism.

215 Nested dissection can be used to improve parallelism as briefly explained in the  
 216 previous section. One level of dissection can be applied by choosing, for example, the  
 217 two middle nodes as a separator as shown in the top graph of Figure 3, where the  
 218 nodes have been re-labeled to match the elimination order. This leads to the tree  
 219 shown in the bottom left part of the same figure. This tree has two separate branches  
 220 which can be traversed in parallel. It must be noted that the order in which the nodes  
 221 on the right of the separator are visited is reversed with respect to the original one; as  
 222 a result, in this case, no additional fill-in is added because the nodes are eliminated  
 223 starting at both ends of the graph.

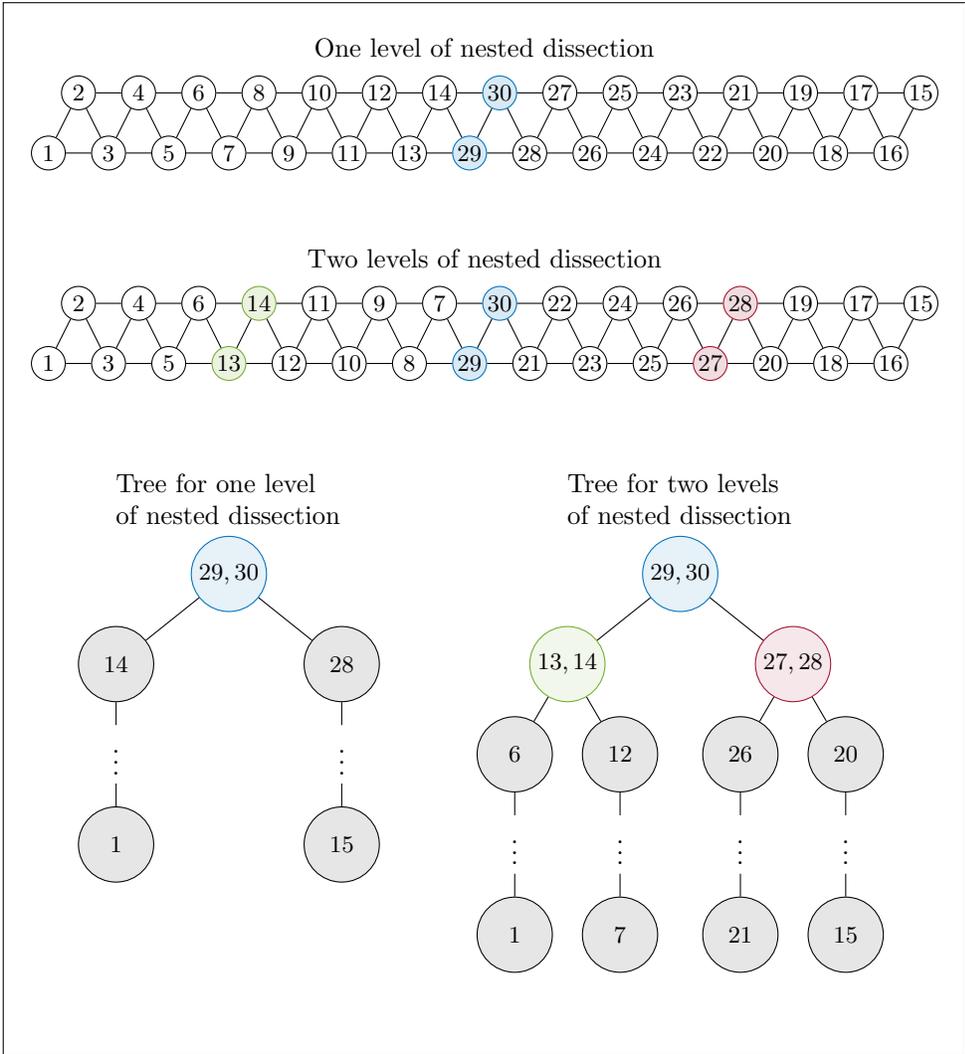


FIG. 3. The effect of one and two levels of nested dissection on the compressed graph of the normal equation matrix.

224 Additional parallelism can be achieved by recursively applying this bisection step.  
 225 Figure 3 shows two levels of nested dissection lead to a tree with four parallel branches  
 226 and Figure 4 (left) shows the matrix permuted accordingly; note that in Figure 4 we  
 227 have applied the permutation also to block-rows and, although this is not necessary  
 228 in practice (as explained in the next section), we will assume it is always the case for  
 229 the sake of simplicity. It is important to note that when two levels of nested dissection  
 230 are used some fill-in is generated in  $R$  when processing the nodes in the two middle  
 231 branches; for example, upon elimination of node 7, three fill edges appear to connect  
 232 nodes 8, 9, 29 and 30 to each other, as described in the elimination graph procedure  
 233 presented in the previous section. The final structure of the  $R$  factor is depicted in  
 234 Figure 4 (right). With additional levels of nested dissection, a moderate increase  
 235 in the fill-in happens as a result of more nodes being processed in internal branches  
 236 rather than the two outer ones; this will be quantified shortly.

237 The nested dissection method is commonly used for reducing the complexity of a  
 238 sparse factorization. In our case, although it is very effective in improving parallelism,  
 239 it increases the overall cost of the factorization both in terms of flop count and memory  
 240 consumption. Consequently, operations have to be scheduled carefully to keep this  
 241 overhead as low as possible and a model of the factorization complexity as a function  
 242 of the number of nested dissection levels  $l$  must be computed to understand whether a  
 243 suitable compromise between these two parameters can be found. This is the objective  
 244 of the next section.

245 A block  $QR$  factorization algorithm can be roughly described as a method which  
 246 runs in  $c$  steps,  $c$  being the number of block-columns, where at step  $k$  all the nonzero  
 247 blocks in column  $k$  are annihilated except one which is, instead, reduced to an upper  
 248 triangle; as a result of this column reduction we obtain one block-row of the global  
 249  $R$  factor. In our method, this is achieved through a bottom-up traversal of the tree  
 250 where, at each node, one column (or two for nodes associated with separators) is  
 251 reduced. Reducing column  $k$  implies the update of a subset of the blocks in column  
 252  $k + 1, \dots, c$ ; we refer to the ensemble of the blocks affected by the reduction of one  
 253 column as a *frontal matrix*. Depending on the size and shape of this frontal matrix,  
 254 the nodes of the tree can be classified into four different types: leaf, chain, separator  
 255 and root nodes. The first corresponds to the nodes that are at the lowest level of the  
 256 tree (e.g., nodes 7 or 15 in the bottom-right tree of Figure 3); the second corresponds  
 257 to nodes in the sequential branches associated with the subdomains resulting from  
 258 nested dissection (e.g., node 12 or 20 in Figure 3); the third corresponds to nodes  
 259 belonging to some separator (e.g., node (13, 14) in Figure 3); the fourth is the root  
 260 node of the tree. For each type, a different sequence of transformations is used to  
 261 operate the corresponding column reduction; we refer to this as the *pivotal sequence*.  
 262 Moreover, for each type, nodes belonging to the two outer branches need some special  
 263 treatment because, as explained above, reducing the associated columns does not  
 264 introduce any additional fill-in in the  $R$  factor; this will be discussed in deeper details  
 265 in the next section.

266 **2.3. Node elimination.** As explained in the previous section, at each node of  
 267 the tree, one block-column (two for separator nodes) is reduced by annihilating all  
 268 the non-zero blocks in the column except one which is reduced into a triangle. The  
 269 annihilation of a block not only involves its block-row but also a second one which  
 270 we refer to as *pivotal row*; if, for example, the blocks are of size  $1 \times 1$ , this can be  
 271 achieved by a Givens rotation. In our case, because we are dealing with dense blocks of  
 272 potentially large size, Householder reflections will be used instead because they heavily

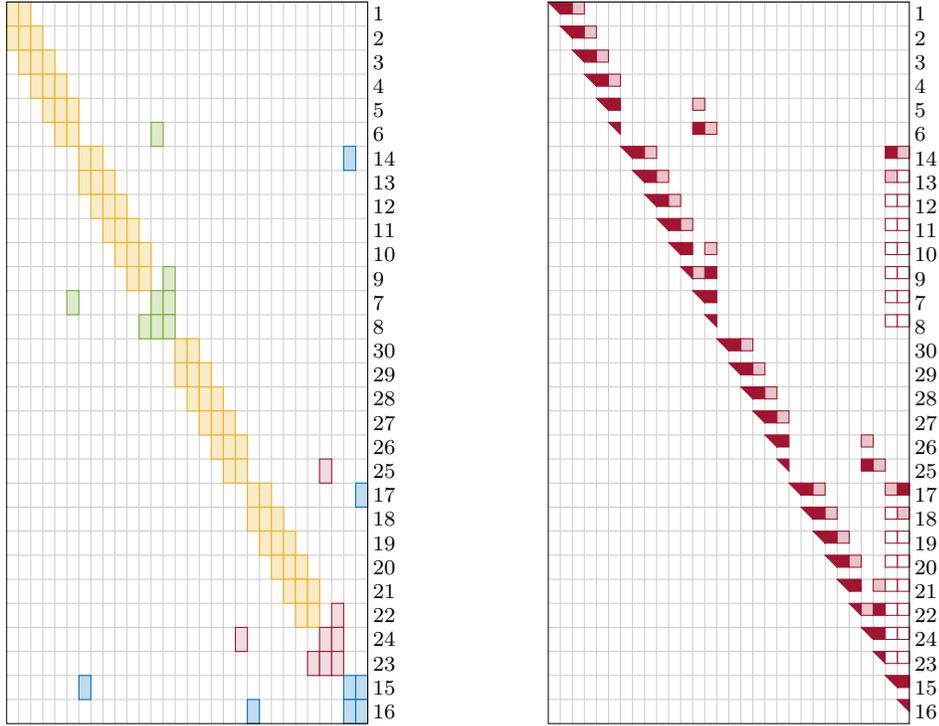


FIG. 4. On the left, a block-tridiagonal matrix with  $c = 30$  permuted using two levels of nested dissection as in Figure 3. Note that, for the sake of illustration, block-columns are permuted in the same way as block-rows although this need not be the case in practice. The numbers on the right of the matrix describe the index of each block-row in the original, unpermuted matrix; the same list of indices applies to block-columns. On the right, the structure of  $R$  after the factorization; blocks with a dark fill color correspond to nonzero blocks in  $A$ ; bordered blocks with a lighter fill color correspond to unavoidable fill blocks (i.e., those associated with horizontal edges in Figure 3); bordered blocks with no fill color correspond to those that are due to the chosen pivotal order.

273 rely on Level-3 BLAS operations and, therefore, can achieve much higher performance  
 274 on modern processors; the LAPACK library provides the basic operations to achieve  
 275 these transformations which are presented below. As a result of a block annihilation in  
 276 column  $k$ , the structure of both the involved rows in columns  $k + 1, \dots, c$  will be equal  
 277 to the union of their prior structures. This implies that some fill-in may be introduced  
 278 by the transformation; obviously this fill-in depends on which pivotal row is chosen  
 279 to annihilate a block. It must be noted that different pivotal sequences always lead  
 280 to the same (numerically)  $R$  factor which only depends on column permutations; as a  
 281 consequence, different pivotal sequences only have an effect on the trailing submatrix  
 282 of  $A$  and, ultimately, the  $Q$  factor. Several strategies were proposed in the past for  
 283 choosing pivotal sequences that reduce the fill-in. The approach that we adopted in  
 284 our work relates to the *variable pivoting* technique proposed by Gentleman [16] for  
 285 the  $QR$  factorization of sparse matrices by means of Givens rotations; this method  
 286 is strongly related to the *row-merge* scheme, later proposed by George and Heath  
 287 [18] and Liu [23] (see the work by Liu for a discussion on how the two approaches  
 288 compare). Variable pivoting can be summed up as a method where, when an entry in  
 289 position  $(i, k)$  has to be annihilated, a pivotal row  $j$  is chosen such that its structure  
 290 is as close as possible to that of row  $i$ .

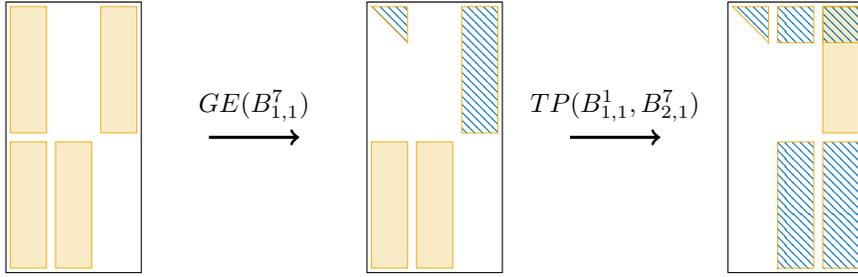


FIG. 5. Example showing the use of the notation used to describe pivotal sequences. The data modified at each step are shown with a fill pattern.

291 The annihilation of blocks and the corresponding updates can be achieved through  
 292 the `geqrt`, `gemqrt`, `tpqrt` and `tpmqrt` elementary operations which are available in  
 293 the LAPACK library. The first of these, `geqrt`, computes the  $QR$  factorization of  
 294 a rectangular block; the second, `gemqrt`, updates a block by applying the  $Q$  matrix  
 295 (or its transpose) resulting from a `geqrt` operation; the third, `tpqrt`, computes the  
 296  $QR$  factorization of a matrix formed by a triangular block on top of a pentagonal  
 297 one (in our case it will only be either rectangular or triangular); the fourth, `tpmqrt`,  
 298 updates a couple of blocks by applying the  $Q$  matrix (or its transpose) resulting from  
 299 a `tpqrt` operation. All these routines are based on Householder transformations and,  
 300 especially for the update routines, heavily rely on Level-3 BLAS operations. Without  
 301 loss of generality, we assume that the number of rows in a block is at least twice  
 302 the number of columns, i.e.,  $m \geq 2n$ ; therefore, each block can be logically split into  
 303 three parts with the third being, possibly, empty: the first containing rows  $1, \dots, n$ ,  
 304 the second containing rows  $n+1, \dots, 2n$  and the third containing rows  $2n+1, \dots, m$ .  
 305 We will refer to these parts using a bitmap: for example  $A^3$  denotes the first and  
 306 second parts of a block  $A$ , 3 being equal to 011 in binary format. The algorithm will  
 307 be described using the following notation:

- 308 •  $GE(B_{i,j}^u)$ : corresponds to reducing part  $u$  of the block in block-row  $i$  and  
 309 block-column  $j$  through a `geqrt` operation and updating all the corresponding  
 310 blocks in row  $i$  using `gemqrt` operations;
- 311 •  $TP(B_{i,j}^u, B_{k,j}^v)$ : corresponds to annihilating part  $v$  of the block in position  
 312  $(k, j)$  using part  $u$  of the block in position  $(i, j)$  through a `tpqrt` operation and  
 313 updating the corresponding blocks along rows  $i$  and  $k$  with `tpmqrt` operations.

314 An example of the use of this notation is provided in Figure 5.

315 For the sake of readability, here we only provide the pivotal sequence for a chain  
 316 node because this is where most of the computations are done assuming  $c \gg 2^l$ ; the  
 317 other node types are discussed in the appendix. Reducing a column  $k$  associated with  
 318 one of these nodes affects blocks in columns  $k, k+1, k+2, a_1$  and  $a_2$  and rows  $k,$   
 319  $k+1$  and  $a_1$ , where  $a_1$  and  $a_2$  are the indices of the nodes in the separator that the  
 320 chain is moving away from. For example, when node 8 in Figure 3 (*bottom-right*) is  
 321 visited, it is connected to nodes 9, 10, 29 and 30 (because node 7 has been previously  
 322 eliminated) which explains why these columns are concerned by its elimination. Note  
 323 also that when node  $k$  is visited, block-rows  $k$  and  $a_1$  and block-columns  $k, k+1,$   
 324  $a_1$  and  $a_2$  have been already updated upon processing of nodes that are lower in the  
 325 same branch; this leads to the particular frontal matrix structure which is illustrated  
 326 in the left part of Figure 6. The pivotal sequence that we use on chain node  $k$  is:

$$\begin{aligned}
& GE(B_{k+1,k}^7), \quad TP(B_{k,k}^1, B_{k+1,k}^1), \quad GE(B_{k+1,k+1}^7), \\
& TP(B_{k+1,k+1}^1, B_{k,k+1}^2), \quad GE(B_{k+1,k+2}^6), \quad TP(B_{k+1,k+2}^2, B_{k,k+2}^2), \\
& \underline{TP(B_{a_1,a_1}^1, B_{k,a_1}^2)}, \quad \underline{TP(B_{a_1,a_1}^1, B_{k+1,a_1}^4)}, \\
& \underline{TP(B_{a_1,a_2}^2, B_{k,a_2}^2)}, \quad \underline{TP(B_{a_1,a_2}^2, B_{k+1,a_2}^4)}.
\end{aligned}$$

328 These operations transform the blocks shown in the left part of Figure 6 into those  
329 shown in the right part of the same figure; the middle part of the figure shows the  
330 state of the factorization after the first column is reduced by means of the first two  
331 operations  $GE(B_{k+1,k}^7)$ ,  $TP(B_{k,k}^1, B_{k+1,k}^1)$ . After these transformations, the blocks  
332 in row  $k$ , with a dark fill color in the figure, are in their final state and will not be  
333 updated anymore; the bordered blocks with a lighter fill color in rows  $k+1$  and  $a_1$   
334 will, instead, be updated in the parent node. Finally, the bordered blocks with no  
335 fill color blocks end up in the  $H$  matrix that implicitly represents  $Q$  by means of  
336 the Householder vectors; these are also in their final state and will not be modified  
337 anymore. Note that the two outer chains advance from one end of the graph towards  
338 the center and, therefore, they do not move away from any separator. As a result only  
339 columns  $k$ ,  $k+1$  and  $k+2$  and rows  $k$  and  $k+1$  are concerned by the elimination of  
340 the nodes therein; this is illustrated with a dashed line in Figure 6. Consequently, the  
341 last four (underlined) operations in the pivotal sequence above need not be executed  
342 on these nodes.

343 This process closely resembles the  $QR$  multifrontal method [2, 13, 10]. It must be  
344 noted, however, that in the multifrontal method, frontal matrices are explicitly assem-  
345 bled by copying coefficients into dense matrix data structures which are allocated in  
346 memory and initialized to zero beforehand; these copies can be extremely costly due  
347 to the large size of blocks and the heavy use of indirect addressing. In our approach,  
348 instead, the frontal matrices need not be formed explicitly but the constituent blocks  
349 can be easily accessed through pointers (see Section 3 for further details). Addition-  
350 ally, the multifrontal method can only take advantage of the zeroes in the bottom-left  
351 part of frontal matrices (this is referred to as “Strategy 3” in the work of Amestoy,  
352 Duff, and Puglisi [2]) whereas, through the use of variable pivoting, our approach can  
353 avoid more unnecessary computations.

354 **2.4. Complexity.** The cost of the elementary operations that are used in our  
355 method are reported in Table 1. For the `tpqrt` and `tpmqrt` operations, two cases are  
356 reported: the first (in the middle column) where the bottom block is a rectangle of  
357 size  $p \times n$  and the second (in the right column) where the bottom block is a triangle of  
358 size  $n \times n$ . Note that,  $p$  is a generic block-row size and, in the algorithm of the previous  
359 section, it does not necessarily correspond to the row size  $m$  of an entire block but the  
360 part that is actually concerned by the operation. The number of columns, instead,  
361 will always be equal to  $n$  (the block column size) which is assumed to be the same for  
362 all blocks.

363 Summing up the values in Table 1 for the operations of the pivotal sequence from  
364 the previous section yields the value in row two, column three of Table 2. For a node  
365 in an outer chain, the cost, reported in row and column two, is lower because fewer  
366 operations are needed as explained in the previous section. In the top rows of the  
367 table we also report the operational complexity of all the other node types and we  
368 refer the reader to the appendix for details of the related pivotal sequences. With a

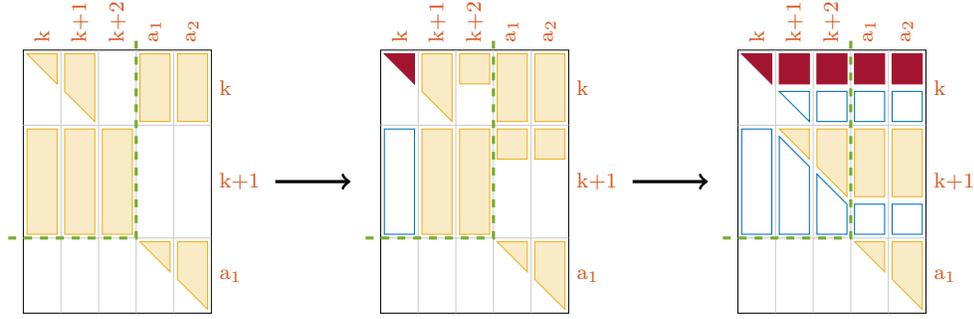


FIG. 6. The structure of the frontal matrix associated with chain node  $k$  at the beginning of the node reduction (left), after the first two steps  $GE(B_{k+1,k}^7)$ ,  $TP(B_{k,k}^1, B_{k+1,k}^1)$  (middle) and at the end of the node reduction (right). The dashed line shows the difference between inner and outer chain nodes.

geqrt	$2n^2(p-n/3)$	tpqrt	$2pn^2$	tpqrt*	$2n^3/3$
gemrt	$2n^2(2p-n)$	tpmqrt	$4pn^2$	tpmqrt*	$2n^3$

TABLE 1

Operational complexity of the elementary operations.  $p$  corresponds to the number of rows in a block that are concerned by the operation. The \* suffix denotes the case where the lower block is a triangle.

369 slight abuse of notation, we denote “outer root” the root of the tree for the case  $l = 0$   
 370 and “middle root” the tree root for the case  $l > 0$ .

371 By the same token, it is possible to compute the amount of memory (in number  
 372 of coefficients) consumed at each node type, reported in the bottom part of Table 2;  
 373 this is made up of the blocks coming from the original matrix (underlined in the  
 374 table) plus the fill-in generated during the processing of the node (for example, for  
 375 chain nodes this can be computed as the difference between the right and left parts  
 376 of Figure 6).

377 Summing up the values in Table 2 for all the nodes of the tree leads to the overall  
 378 cost of the factorization, which is

$$\begin{aligned}
 (2.1) \quad \mathcal{C}(c, l > 0) &= (2^l - 2)\mathcal{C}_{lm} + 2\mathcal{C}_{lo} && \% \text{ leaf nodes} \\
 &+ (2^l - 2) \left( \frac{c+2}{2^l} - 3 \right) \mathcal{C}_{cm} + 2 \left( \frac{c+2}{2^l} - 3 \right) \mathcal{C}_{co} && \% \text{ chain nodes} \\
 &+ (2^l - 2 - 2(l-1)) \mathcal{C}_{sm} + 2(l-1) \mathcal{C}_{so} && \% \text{ separator nodes} \\
 &+ \mathcal{C}_{rm} && \% \text{ root node} \\
 \mathcal{C}(c, l = 0) &= \mathcal{C}_{lo} && \% \text{ leaf nodes} \\
 &+ (c-3)\mathcal{C}_{co} && \% \text{ chain nodes} \\
 &+ \mathcal{C}_{ro}. && \% \text{ root node}
 \end{aligned}$$

380 In the above formula,  $\mathcal{C}$  can be replaced with either  $\mathcal{F}$  or  $\mathcal{M}$  leading to, respec-  
 381 tively, the overall flop count or the overall memory consumption.

382 In this formula, for the case where no nested dissection is used (i.e.,  $l = 0$ ), the  
 383 root node corresponds to the penultimate chain node where the last two block-columns  
 384 of the matrix are reduced.

385 Figure 7 (top) and (middle) shows how the relative cost of the factorization,

	Outer	Middle
Leaf	$\mathcal{F}_{lo} = 32mn^2 - 16n^3$	$\mathcal{F}_{lm} = 86mn^2 - 100n^3/3$
Chain	$\mathcal{F}_{co} = 18mn^2 - 2n^3/3$	$\mathcal{F}_{cm} = 42mn^2 - 2n^3/3$
Separator	$\mathcal{F}_{so} = 34mn^2 - 10n^3/3$	$\mathcal{F}_{sm} = 98mn^3 - 10n^3/3$
Root	$\mathcal{F}_{ro} = 8mn^2$	$\mathcal{F}_{rm} = 16n^3/3$
Leaf	$\mathcal{M}_{lo} = mn + \underline{5mn}$	$\mathcal{M}_{lm} = 4mn + 2n^2 + \underline{9mn}$
Chain	$\mathcal{M}_{co} = 2n^2 + \underline{3mn}$	$\mathcal{M}_{cm} = 2mn + 2n^2 + \underline{3mn}$
Separator	$\mathcal{M}_{so} = 3n^2$	$\mathcal{M}_{sm} = 12n^2$
Root	$\mathcal{M}_{ro} = \underline{2mn}$	$\mathcal{M}_{rm} = 0$

TABLE 2

Flops (top) and storage (bottom) count at each node of the tree. Each value in the top part is computed as a sum of the complexity of the elementary operations reported in Table 1 over the corresponding pivotal sequence. Each value in the bottom part is computed as the difference between the structure of blocks before and after processing the corresponding node.

386 respectively  $\mathcal{F}(c, l)/\mathcal{F}(c, 0)$  and  $\mathcal{M}(c, l)/\mathcal{M}(c, 0)$ , varies for different problem sizes  
387 and different levels of nested dissection for the case where  $m = 2n$ ; it is possible to see  
388 that, for growing values of  $l$  and  $c$ , these curves converge to, respectively,  $\mathcal{F}_{cm}/\mathcal{F}_{co}$   
389 and  $\mathcal{M}_{cm}/\mathcal{M}_{co}$  because for  $l = 0$  the tree is essentially made of a single outer chain,  
390 whereas for large  $l$  and  $c$  values the vast majority of computations are done in nodes  
391 belonging to middle chains. For growing values of  $m$ , these two ratios converge,  
392 respectively, to  $2.3\bar{3}$  and  $1.6\bar{6}$ .

393 Other pivotal sequences can be used on each node type. Finding an optimal  
394 pivotal sequence is an extremely challenging task due to its combinatorial nature. The  
395 sequence(s) proposed above, however aim at achieving as many operations as possible  
396 within the chains which can be processed in an embarrassingly parallel fashion; indeed  
397 all the operations in separator nodes only involve parts of blocks that are of size  $n \times n$   
398 (either square or triangular; see the appendix for further details). This is likely to  
399 reduce the relative weight of the reduction phase associated with the top part of the  
400 tree where communications happen. Figure 7 (bottom) shows which fraction of the  
401 total floating point operations is performed within chain or leaf nodes demonstrating  
402 that even with matrices of relatively small size it is possible to achieve high levels of  
403 parallelism (by increasing  $l$ ) without incurring an excessive volume of communications.

404 **3. A solver with multiple levels of parallelism.** In this section, we describe  
405 how an actual parallel implementation of the described algorithm for shared memory  
406 systems (e.g., single node, multicore machines) was achieved.

407 As explained, our method relies on four basic kernels which are used to achieve the  
408 pivotal sequences described in Section 2.3 and in the appendix. Although these kernels  
409 are available in the LAPACK library, we have chosen to rely on their implementation  
410 in the `qr_mumps` library. This software implements a sparse multifrontal solver based  
411 on the  $QR$  factorization; as such it relies on dense linear algebra operations for pro-  
412 cessing frontal matrices. Among others, `qr_mumps` provides an implementation of the  
413 `geqrt`, `gemqrt` operations; to these we have added an implementation of the `tpqrt`  
414 and `tpmqrt` (see below for the details).

415 One immediate advantage of this choice is that, in `qr_mumps` these operations are  
416 efficiently parallelized through the use of scalable *tiled* algorithms [11] (also referred  
417 to as *communication avoiding*): blocks are divided into square tiles of size  $b \times b$  which

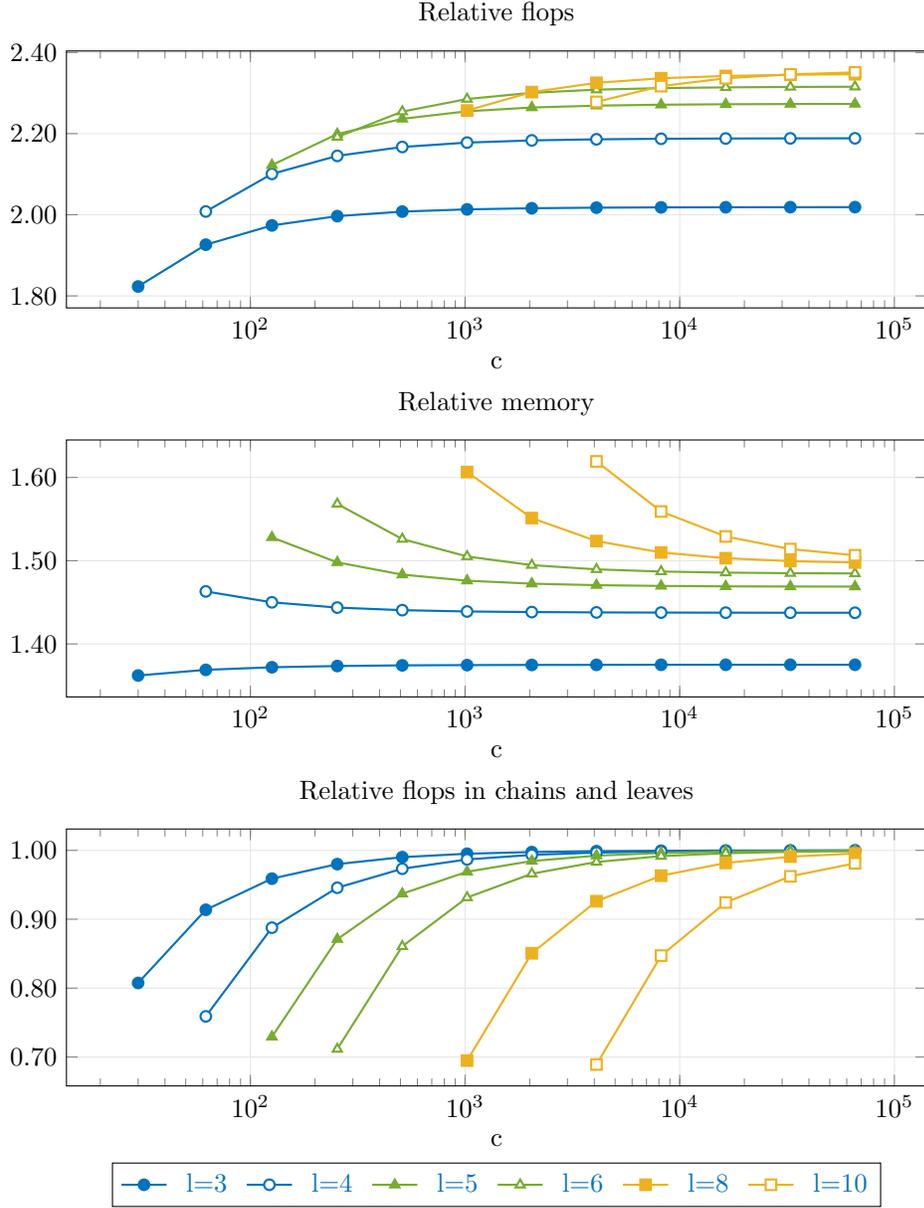


FIG. 7. Relative number of floating point operations (top) and memory consumption (middle) with respect to no nested dissection based permutation. Relative number of floating point operations performed in chain of leaf nodes with respect to the total (bottom).  $m = 2n$  is assumed for all.

418 are eliminated by means of block Householder transformations according to different  
 419 schemes [9] which aim at improving concurrency by reducing unnecessary synchroniza-  
 420 tions and communications. Our implementation of the `tpqrt` and `tpmqrt` operations  
 421 relies on the same approach and uses a tile elimination scheme which is referred to as  
 422 Sameh-Kuck by Bouwmeester et al. [9]: the tiles in the trapezoidal block (remember  
 423 that we only consider the two special cases where the trapezoidal degenerates to a

424 triangle or a rectangle) along a column are eliminated one after the other starting  
425 from the top using the corresponding diagonal tile in the triangular block. Through  
426 the use of these parallel kernels some parallelism can be achieved within each node of  
427 the tree and even in the case where no nested dissection permutation is used.

428 `qr_mumps` relies on task based parallelism for the implementation of these meth-  
429 ods. The workload is represented in the form of a Directed Acyclic Graph (DAG)  
430 where nodes are elementary operations (e.g., elimination of a tile) and edges are  
431 the dependencies among them; this representation allows for very quick access to all  
432 the available concurrency at any moment during the execution and for using very  
433 dynamic and asynchronous execution approaches. `qr_mumps` achieves task based par-  
434 allelism through the use of a *runtime system*, namely StarPU [5], which provides a  
435 programming interface for defining the DAG and a scheduling engine for tracking  
436 the status of the DAG and triggering the execution of ready tasks on the available  
437 processing units [1]. StarPU provides a programming API based on the Sequential  
438 Task Flow (STF) model (sometimes also referred to as *superscalar*). In this model,  
439 a master thread submits tasks to the runtime specifying how each of them accesses  
440 the data (whether in read or write mode); based on this information and the tasks  
441 submission order, the runtime can automatically define the dependencies among tasks  
442 and generate the DAG. This model is also used in the OpenMP [8] standard (through  
443 the `task` directive and the `depend` clause) and other runtime systems.

444 A further advantage of using the kernels available in `qr_mumps` is that its asyn-  
445 chronous API provides additional parallelism besides that available among different  
446 branches of the tree and within each block operation. With the standard synchronous  
447 API, each call to an operation is blocking, which means that control is returned to  
448 the caller only after the DAG for that operation is created and all of its tasks are  
449 executed. With the asynchronous API, instead, only the DAG is created and control  
450 is immediately returned to the caller; methods are available in `qr_mumps` to check  
451 whether all the tasks associated with one called operation have been executed. When  
452 multiple operations have to be executed, asynchronous calls can be made for all of  
453 them, which will generate a single DAG for all at once where dependencies are inferred  
454 between tasks belonging to different operations that access the same data. Thanks  
455 to the availability of this asynchronous interface, our implementation of the block  
456 tridiagonal  $QR$  factorization algorithm described in the previous section consists of a  
457 sequential code that traverses the tree in a topological order and, for each node, calls  
458 the `qr_mumps` operations corresponding to the pivotal sequences described and in the  
459 appendix; parallelism is automatically taken care of by `qr_mumps`.

460 The whole matrix data structure is a simple bi-dimensional array of pointers to  
461 blocks where only those that point to nonzero blocks (either in the original matrix or  
462 in the factors) are non-null. Consequently, as already mentioned above, in our code  
463 frontal matrices need not be explicitly formed but, rather, the constituent blocks  
464 can be easily accessed with a simple indirection without any data movement. This  
465 allows for significant time savings with respect to a general purpose multifrontal solver  
466 whereas frontal matrices are explicitly assembled through time consuming and poorly  
467 parallelizable memory-bound (that is, limited by the speed of the main memory)  
468 operations.

469 **4. Experimental results.** In this section we report experimental results that  
470 illustrate the behavior of the algorithm presented in Section 2 and its implementation  
471 described in Section 3. These were obtained on one node of the Olympe supercomputer  
472 of the CALMIP regional supercomputing center in Toulouse, France. One such node

		$l$					
		0	1	2	3	4	5
#threads	1	153.8	-	-	-	-	-
	2	78.9	77.7	-	-	-	-
	4	40.4	39.0	57.9	-	-	-
	8	26.3	20.2	29.7	34.8	-	-
	16	25.8	15.2	17.0	19.7	21.3	-
	32	25.9	15.6	12.5	14.5	15.3	16.1
	36	26.5	17.3	14.5	15.1	16.0	16.4
#threads	1	149.1	-	-	-	-	-
	2	81.9	74.2	-	-	-	-
	4	59.4	40.7	55.9	-	-	-
	8	52.5	29.8	27.5	32.6	-	-
	16	53.2	28.7	16.7	18.0	19.3	-
	32	57.3	28.7	15.3	9.7	10.1	10.7
	36	67.1	34.7	15.9	9.6	9.4	9.7

TABLE 3

Execution time, in seconds, for the  $QR$  factorization of a problem with  $m = 640$ ,  $n = 320$  and  $c = 4094$  with tile size  $b = 160$  (top) and  $b = 320$  (bottom).

473 is equipped with two Intel(R) Xeon(R) Gold 6140 CPUs for a total of 36 cores. We  
474 have used the Intel compilers and the Intel MKL library v2018.2, StarPU revision  
475 581387c from the public GIT repository<sup>1</sup> and `qr_mumps` revision 8b160df<sup>2</sup>.

476 For these experiments, we have set the block size to  $m = 640$ ,  $n = 320$  and  
477 number of block-columns and block-rows  $c = 4094$ ; note that choosing  $c = 2^t - 2$  for  
478 some  $t$  ensures that when nested dissection is applied all the chains have the same  
479 length  $2^{t-l} - 2 \forall l = 1, \dots, t - 1$ . We will comment on how the behavior will change  
480 for different values of these parameters but we will not report further experiments  
481 because they do not provide additional insight to the experimental analysis. Two  
482 different tile sizes have been chosen for the partitioning of blocks within the `qr_mumps`  
483 operations, namely,  $b = 160$  and  $b = 320$ . Finally, we have executed experiments  
484 for varying values of the number of nested dissection levels  $l$  from 0 (i.e., no matrix  
485 permutation) to 5; going beyond this value does not make sense in our experimental  
486 setting because of the limited number of cores.

487 Table 3 shows the execution time of the  $QR$  factorization for different numbers of  
488 threads and values of the  $l$  parameter with  $b = 160$  in the top part and  $b = 320$  in the  
489 bottom part. The “-” entries correspond to the cases where the number of threads  
490 is lower than the number of chains  $2^l$ ; these are uninteresting because of the excessive  
491 flops overhead spent for an unnecessary amount of parallelism. When nested dissection  
492 is not used, scalability is obviously limited by the lack of parallelism; obviously, better  
493 results are obtained by using small tiles because higher parallelism is available within  
494 each branch of the tree. When one level of nested dissection is applied, parallelism  
495 is roughly doubled with no additional cost, which explains why the execution times  
496 in the second column of the table are always below the corresponding ones in the  
497 first. The improvement is, as expected, higher in the case of large tiles and when the

<sup>1</sup><https://gforge.inria.fr/projects/starpu/>

<sup>2</sup>[http://buttari.perso.enseeiht.fr/qr\\_mumps/releases/qr\\_mumps\\_8b160df.tgz](http://buttari.perso.enseeiht.fr/qr_mumps/releases/qr_mumps_8b160df.tgz)

		$l$					
		0	1	2	3	4	5
#threads	1	30.8	-	-	-	-	-
	2	60.0	60.9	-	-	-	-
	4	117.0	121.2	128.0	-	-	-
	8	180.0	234.2	249.5	251.5	-	-
	16	183.3	311.5	435.2	444.0	441.2	-
	32	182.5	303.1	590.8	600.7	615.8	603.1
	36	178.5	272.6	509.7	579.2	588.1	591.6
#threads	1	31.7	-	-	-	-	-
	2	57.8	63.8	-	-	-	-
	4	79.6	116.3	132.6	-	-	-
	8	90.1	158.5	268.8	268.5	-	-
	16	88.9	164.8	442.3	486.0	488.0	-
	32	82.6	165.0	482.5	901.9	932.0	907.4
	36	70.5	136.5	464.1	905.2	997.8	997.8

TABLE 4

Speed in Gflop/s for the QR factorization of a problem with  $m = 640$ ,  $n = 320$  and  $c = 4094$  with tile size  $b = 160$  (top) and  $b = 320$  (bottom).

498 number of threads grows. For values of  $l$  greater than or equal to two, the use of nested  
499 dissection implies a computational overhead. This explains why, when the number  
500 of threads is relatively low, increasing  $l$  results in a slowdown; this is especially true  
501 when small tiles are used because sufficient parallelism is available even with small  
502 values of  $l$ . When the number of threads is relatively high, the benefit of the additional  
503 parallelism overcomes the extra computational cost and the execution time is reduced.  
504 Additionally, we can observe that, when the number of threads is relatively high, the  
505 best results are achieved with large tiles: this is because of the larger granularity of  
506 tasks which allows for a better efficiency of the BLAS operations executed on tiles.  
507 Ultimately, we can conclude that, when the number of threads is relatively low, using  
508 small tiles and few levels of nested dissection provides sufficient parallelism with no  
509 or little computational overhead; inversely, when more threads are used, using large  
510 tiles is beneficial for improving the efficiency of elementary operations and the extra  
511 parallelism is worth the computational overhead. This is illustrated in Figure 8 (left)  
512 where the best execution times are taken along each row of Table 3 and compared  
513 to the best sequential result, i.e., 149.1s. The number next to each data point shows  
514 the nested dissection levels used to obtain the corresponding result. The best time  
515 improvement with respect to the sequential case is 15.8 for the case  $l = 4$  which results  
516 in roughly twice as many floating-point operations.

517 In order to assess the efficiency and scalability of our implementation, in Table 4  
518 we provide the speed of computations in Gigaflops per second which is insensitive to  
519 the computational overhead imposed by the use of nested dissection. As expected,  
520 performance grows for higher values of  $l$  because more embarrassing parallelism is  
521 available; for a given value of  $l$ , performance at first grows with the number of threads  
522 and eventually decreases when the available parallelism is not enough to feed all the  
523 working threads. The peak performance of a single core is 73.6 Gflop/s, which means  
524 that the sequential execution has an efficiency of 43%. This relatively poor efficiency

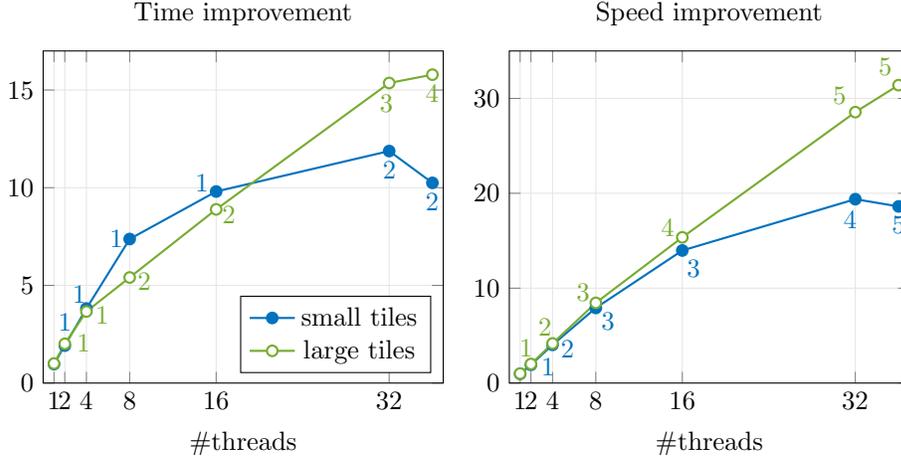


FIG. 8. Best time and speed improvement over the sequential case. The values beside each data point shows the corresponding value of  $l$ .

	$l$					
	0	1	2	3	4	5
160	76.3	56.5	79.9	96.8	104.2	106.6
320	138.2	75.1	49.8	58.2	61.9	64.7
160	496.4	670.4	741.9	723.2	723.0	732.2
320	274.2	504.5	1191.3	1202.9	1217.1	1205.2

TABLE 5

Execution time (top) and speed in Gflop/s (bottom) for the QR factorization of a problem with  $m = 1280$ ,  $n = 640$  and  $c = 4094$  with tile size  $b = 160$  (top) and  $b = 320$  (bottom) on 36 cores.

525 can be explained by the relatively small values of  $m$  and  $n$  which makes the relative  
526 cost of BLAS-2 rich operations (i.e., `geqrt` and `tpqrt`) greater; higher speeds can be  
527 observed when larger blocks are used. The best speed improvement is a remarkable  
528 31.4 out of 36; it must be noted that for the case of  $l = 5$ , roughly 99% of the  
529 computations are done in chain or leaf nodes.

530 For the sake of completeness, in Table 5 we report the execution time and speed  
531 for the factorization of a matrix with large blocks ( $m = 1280$  and  $n = 640$ ), same  
532 value of  $c = 4094$ , obtained with 36 threads. In this case, more parallelism is available  
533 within each branch and, consequently, the best shortest execution time is achieved  
534 with fewer levels of nested dissection with respect to the previous case. It can also  
535 be noted that the overall performance is higher thanks to a better ratio between  
536 Level-3 and Level-2 BLAS operations; this also holds for the sequential case where  
537 the execution time is 960.7 seconds for a speed of 39.46 Gflop/s.

538 The above results suggest that, in order to obtain the lowest execution time, a  
539 careful combination of the `qr_mumps` block size  $b$  and nested dissection depth  $l$  must be  
540 chosen. The optimal values mostly depend on the characteristics of the used platform  
541 such as number of cores, relative speed between cores and memory, and efficiency  
542 of low-level BLAS operations. The proposed approach proved to be scalable in our  
543 experimental setting because the shortest execution time is obtained using all the

	$l$				
	0	1	2	3	4
flops ratio	1.00	1.00	1.11	1.14	1.16
seq time ratio	1.66	1.66	1.62	1.57	1.57
par time ratio	4.08	6.30	8.98	11.34	11.14

TABLE 6

*Ratios of floating point and execution times between the developed implementation of the proposed approach and `qr_mumps` for different values of  $l$  with  $m = 640$ ,  $n = 320$  and  $c = 2046$ .*

544 available threads; for smaller problems (i.e., smaller values of  $m$ ,  $n$  or  $c$ ), using all the  
545 available cores may not be beneficial though. Because the cost of the factorization as a  
546 function of  $l$  is upper bounded by a modest constant and because most computations  
547 are performed in chain or leaf nodes (see Figure 7 and the related discussion), for  
548 relatively large values of  $c$  we can expect to achieve good scalability even on large  
549 numbers of cores.

550 Although, as discussed above, some concurrency is available within each branch  
551 or node of the tree, this parallelism involves communications due to the fact that  
552 multiple processes share the same data. On shared memory systems, these communi-  
553 cations take the form of memory traffic and synchronizations. In distributed memory  
554 systems, these communications amount to transferring data through the slow net-  
555 work interconnection and, therefore, are much more penalizing. The use of nested  
556 dissection introduces embarrassing parallelism because each process may potentially  
557 work on a different branch without communicating with others. For this reason, we  
558 speculate that the benefit of increasing  $l$  over the operational overhead is better on  
559 distributed memory systems than shared memory ones. We reserve the development  
560 and analysis of a distributed memory parallel implementation for the future.

561 **4.1. Comparison with `qr_mumps`.** In this section we document an experimen-  
562 tal comparison between the proposed approach and `qr_mumps`. This is intended to  
563 demonstrate the advantage over a general-purpose (structure unaware) tool. Care  
564 must be taken, however, in interpreting the results as it is not a direct comparison of  
565 theory. After all, design choices were made in `qr_mumps` to cater for general structures.

566 For this comparison we have chosen to use the reference block size  $m = 640$ ,  
567  $n = 320$ . With such a block size, the largest possible value of  $c$  is 2046; larger values  
568 lead to a total number of nonzeros which exceeds the capacity of 32bit integers used  
569 in `qr_mumps` (note that our code never needs to compute nor use the total number of  
570 nonzeros in the matrix). For a fair comparison, the same column permutation (as  
571 discussed in Section 2.2) was used in `qr_mumps` and the developed implementation of  
572 the proposed approach. The operation count as well as the sequential and parallel  
573 (with 36 threads) execution times for values of  $l$  from 0 to 4 were measured. Table 6  
574 contains the ratios of the values obtained with `qr_mumps` and the proposed approach.

575 The difference in terms of floating point operations is due to the use, in our  
576 approach, of pivotal sequences that can take advantage of empty blocks within the  
577 structure of frontal matrices. This difference is negligible (less than 1%) for  $l = 0$  or 1  
578 because in these cases most fronts (all fronts except leaves and root) are of type outer  
579 chain (the part above and on the left of the dashed line in Figure 6) and almost full.  
580 As  $l$  grows, the tree includes more and more fronts with a more complex structure  
581 where better gains can be achieved. This ratio was also observed to increase when

582 the value  $m/n$  grows because the empty blocks within fronts are larger.

583 The ratio between the sequential execution times is around 1.6 and is partly  
584 due to the extra floating point operations and, in most part, due to the symbolic  
585 operations done in `qr_mumps` to assemble frontal matrices. In a parallel execution,  
586 this ratio becomes much larger and increases with  $l$ . This is due to the fact that  
587 symbolic operations are memory-bound and, thus, scale very poorly; moreover some  
588 of these operations are not parallelized at all in `qr_mumps` because in its intended  
589 use they only account for a small, often negligible, part of the execution time. As a  
590 result, these symbolic operations become dominant in a parallel execution and, unlike  
591 our approach, `qr_mumps` cannot take advantage of the parallelism provided by higher  
592 values of  $l$ .

593 **5. Conclusion.** In this work, we have presented a method for computing the  
594  $QR$  factorization of block-tridiagonal matrices. Our approach draws parallelism from  
595 the nested dissection method which allows for structuring the elimination process  
596 as a parallel topological order traversal of a tree graph. At each node of this tree,  
597 one block-column is reduced through Householder reflections. These operations are  
598 computed in such a way that the sparsity of the data handled at each node of the tree  
599 is taken advantage of.

600 A detailed analysis of the operational and memory complexity of the resulting  
601 method was provided. Although we demonstrated that this complexity grows with  
602 the number of used nested dissection levels, the overhead with respect to the sequential  
603 (i.e., no nested dissection) case is bounded by a moderate constant.

604 An actual implementation of the factorization method that can take advantage  
605 of multiple levels of parallelism was also presented. Not only does our solver achieve  
606 parallelism through the concurrent traversal of separate branches of the elimination  
607 tree but also from the intrinsic concurrency available at each tree node and from  
608 pipelining the processing of nodes at different levels of the tree. This is achieved  
609 by using the parallel, asynchronous dense linear algebra methods available in the  
610 `qr_mumps` software and developing the missing operations.

611 Finally, experimental results on a shared memory architecture that show the  
612 effectiveness of our approach as well as of its actual implementation were included.

613 **6. Acknowledgments.** This work was granted access to the HPC resources  
614 of CALMIP supercomputing center under the allocation 2018-p0989. S. Hauberg  
615 was supported by a research grant (15334) from VILLUM FONDEN. C. Kodsı and  
616 S. Hauberg have received funding from the European Research Council (ERC) un-  
617 der the European Union’s Horizon 2020 research and innovation programme (grant  
618 agreement n° 757360). A. Buttari is partially supported by 3IA Artificial and Natu-  
619 ral Intelligence Toulouse Institute, French “Investing for the Future - PIA3” program  
620 under the Grant agreement ANR-19-PI3A-0004.

## 621 **References.**

- 622 [1] Emmanuel Agullo et al. “Implementing Multifrontal Sparse Solvers for Mul-  
623 ticore Architectures with Sequential Task Flow Runtime Systems”. In: *ACM*  
624 *Trans. Math. Softw.* 43.2 (Aug. 2016), 13:1–13:22. ISSN: 0098-3500. DOI: [10.1145/2898348](https://doi.org/10.1145/2898348).  
625  
626 [2] Patrick R. Amestoy, Iain S. Duff, and Chiara Puglisi. “Multifrontal QR factor-  
627 ization in a multiprocessor environment”. In: *Int. Journal of Num. Linear Alg.*  
628 *and Appl.* 3(4) (1996), pp. 275–300. DOI: [10.1002/\(SICI\)1099-1506\(199607/08\)3:4<275::AID-NLA83>3.0.CO;2-7](https://doi.org/10.1002/(SICI)1099-1506(199607/08)3:4<275::AID-NLA83>3.0.CO;2-7).  
629

- 630 [3] Peter Arbenz and Markus Hegland. “On the stable parallel solution of general  
631 narrow banded linear systems”. In: *High performance algorithms for structured*  
632 *matrix problems*. Vol. 2. Adv. Theory Comput. Math. Commack, NY: Nova Sci.  
633 Publ., 1998, pp. 47–73.
- 634 [4] Georgios Arvanitidis, Lars Kai Hansen, and Søren Hauberg. “Latent Space Odd-  
635 ity: on the Curvature of Deep Generative Models”. In: *International Conference*  
636 *on Learning Representations (ICLR)*. 2018.
- 637 [5] Cedric Augonnet et al. “StarPU: A Unified Platform for Task Scheduling on Het-  
638 erogeneous Multicore Architectures”. In: *Concurrency and Computation: Prac-*  
639 *tice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–  
640 198. DOI: [10.1002/cpe.1631](https://doi.org/10.1002/cpe.1631).
- 641 [6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. “Representation learning:  
642 A review and new perspectives”. In: *IEEE transactions on pattern analysis and*  
643 *machine intelligence* 35.8 (2013), pp. 1798–1828.
- 644 [7] Michael W. Berry and Ahmed Sameh. “Multiprocessor Schemes for Solving  
645 Block Tridiagonal Linear Systems”. In: *The International Journal of Supercom-*  
646 *puting Applications* 2.3 (1988), pp. 37–57. DOI: [10.1177/109434208800200304](https://doi.org/10.1177/109434208800200304).
- 647 [8] The OpenMP architecture review board. *OpenMP 5.0 Complete specifications*.  
648 2018.
- 649 [9] Henricus Bouwmeester et al. “Tiled QR Factorization Algorithms”. In: *Pro-*  
650 *ceedings of 2011 International Conference for High Performance Computing,*  
651 *Networking, Storage and Analysis*. SC ’11. Seattle, Washington: ACM, 2011,  
652 7:1–7:11. ISBN: 978-1-4503-0771-0. DOI: [10.1145/2063384.2063393](https://doi.org/10.1145/2063384.2063393).
- 653 [10] Alfredo Buttari. “Fine-Grained Multithreading for the Multifrontal QR Factor-  
654 ization of Sparse Matrices”. In: *SIAM Journal on Scientific Computing* 35.4  
655 (2013), pp. C323–C345. DOI: [10.1137/110846427](https://doi.org/10.1137/110846427).
- 656 [11] Alfredo Buttari et al. “A class of parallel tiled linear algebra algorithms for  
657 multicore architectures”. In: *Parallel Comput.* 35 (1 Jan. 2009), pp. 38–53. ISSN:  
658 0167-8191. DOI: [10.1016/j.parco.2008.10.002](https://doi.org/10.1016/j.parco.2008.10.002).
- 659 [12] Tim Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial  
660 and Applied Mathematics, 2006. DOI: [10.1137/1.9780898718881](https://doi.org/10.1137/1.9780898718881).
- 661 [13] Timothy A. Davis. “Algorithm 915, SuiteSparseQR: Multifrontal multithreaded  
662 rank-revealing sparse QR factorization”. In: *ACM Trans. Math. Softw.* 38.1  
663 (Dec. 2011), 8:1–8:22. ISSN: 0098-3500. DOI: [10.1145/2049662.2049670](https://doi.org/10.1145/2049662.2049670).
- 664 [14] James Demmel et al. “Communication-optimal Parallel and Sequential QR and  
665 LU Factorizations”. In: *SIAM J. Sci. Comput.* 34.1 (Feb. 2012), pp. 206–239.  
666 ISSN: 1064-8275. URL: <http://dx.doi.org/10.1137/080731992>.
- 667 [15] Iain S Duff, Albert M Erisman, and John K Reid. *Direct Methods for Sparse*  
668 *Matrices*. New York, NY, USA: Oxford University Press, Inc., 1986. ISBN: 0-  
669 198-53408-6.
- 670 [16] W. Morven Gentleman. “Row elimination for solving sparse linear systems and  
671 least squares problems”. In: *Numerical Analysis*. Ed. by G. Alistair Watson.  
672 Berlin, Heidelberg: Springer Berlin Heidelberg, 1976, pp. 122–133. ISBN: 978-3-  
673 540-38129-7. DOI: [10.1007/BFb0080119](https://doi.org/10.1007/BFb0080119).
- 674 [17] Alan J. George. “Nested dissection of a regular finite-element mesh”. In: *SIAM*  
675 *J. Numer. Anal.* 10.2 (1973), pp. 345–363. DOI: [10.1137/0710032](https://doi.org/10.1137/0710032).
- 676 [18] Alan J. George and Michael T. Heath. “Solution of Sparse Linear Least Squares  
677 Problems Using Givens Rotations”. In: 34 (1980), pp. 69–83.
- 678 [19] Søren Hauberg. “Only Bayes should learn a manifold”. In: (2018).

- 679 [20] Markus Hegland. “On the parallel solution of tridiagonal systems by wrap-  
680 around partitioning and incomplete LU factorization”. In: *Numerische Mathe-*  
681 *matik* 59.1 (Dec. 1991), pp. 453–472. ISSN: 0945-3245. DOI: [10.1007/BF01385791](https://doi.org/10.1007/BF01385791).
- 682 [21] Markus Hegland and Mike Osborne. “Algorithms for Block Bidiagonal Sys-  
683 tems on Vector and Parallel Computers”. In: *Proceedings of the 12th Inter-*  
684 *national Conference on Supercomputing*. ICS ’98. Melbourne, Australia: ACM,  
685 1998, pp. 1–6. ISBN: 0-89791-998-X. DOI: [10.1145/277830.277835](https://doi.org/10.1145/277830.277835).
- 686 [22] R. W. Hockney. “A Fast Direct Solution of Poisson’s Equation Using Fourier  
687 Analysis”. In: *J. ACM* 12.1 (Jan. 1965), pp. 95–113. ISSN: 0004-5411. DOI: [10.1145/321250.321259](https://doi.org/10.1145/321250.321259).
- 688 [23] Joseph W. H. Liu. “On General Row Merging Schemes for Sparse Givens Trans-  
689 formations”. In: *SIAM J. Sci. Stat. Comput.* 7 (1986), pp. 1190–1211.
- 690 [24] Joseph W. H. Liu. “The Role of Elimination Trees in Sparse Factorization”.  
691 In: *SIAM Journal on Matrix Analysis and Applications* 11 (1990), pp. 134–172.  
692 DOI: [10.1137/0611010](https://doi.org/10.1137/0611010).
- 693 [25] Eric Polizzi and Ahmed H. Sameh. “A parallel hybrid banded system solver:  
694 the SPIKE algorithm”. In: *Parallel Computing* 32.2 (2006). Parallel Matrix  
695 Algorithms and Applications (PMAA’04), pp. 177–194. ISSN: 0167-8191. DOI:  
696 <https://doi.org/10.1016/j.parco.2005.07.005>.
- 697 [26] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. “Algorithmic Aspects  
698 of Vertex Elimination on Graphs”. In: *SIAM Journal on Computing* 5.2 (1976),  
699 pp. 266–283. DOI: [10.1137/0205021](https://doi.org/10.1137/0205021).
- 700 [27] A. Sameh. “On Two Numerical Algorithms for Multiprocessors”. In: *Proc. of*  
701 *NATO Adv. Res. Workshop on High-Speed Comp., Series F: Computer and*  
702 *Systems Sciences*. Vol. 7. 1984, pp. 311–328.
- 703 [28] A. H. Sameh and D. J. Kuck. “On Stable Parallel Linear System Solvers”. In: *J.*  
704 *ACM* 25.1 (Jan. 1978), pp. 81–91. ISSN: 0004-5411. DOI: [10.1145/322047.322054](https://doi.org/10.1145/322047.322054).

## 706 Appendix A. Pivotal sequences.

707 *Leaf nodes.* The matrix blocks concerned by the operations on a leaf node are  
708 depicted in Figure 9 (left). At leaf node  $k$ , these are in columns  $k, k + 1, k + 2,$   
709  $a_1, a_2$  and rows  $k, k + 1, a_1$  where  $a_1$  and  $a_2$  are block-columns associated with an  
710 ancestor separator, the one that the chain is moving away from. Because the two  
711 outer branches are not moving away from any separator, only the first three rows and  
712 columns are concerned by the processing of a leaf node attached to one such chain, as  
713 illustrated by the dashed line in the figure. The operations executed on a leaf node  
714 result in the structure shown in the right part of the figure and are described by the  
715 following pivotal sequence:

$$\begin{aligned}
& GE(B_{k,k}^7), GE(B_{k+1,k}^7), TP(B_{k,k}^1, B_{k+1,k}^1), GE(B_{a_1,k}^7), \\
& TP(B_{k,k}^1, B_{a_1,k}^1), GE(B_{k+1,k+1}^7), TP(B_{k+1,k+1}^1, B_{k,k+1}^6), \\
& TP(B_{k+1,k+1}^1, B_{a_1,k+1}^1), GE(B_{k+1,k+2}^6), TP(B_{k+1,k+2}^2, B_{k,k+2}^6), \\
& TP(B_{k+1,k+2}^2, B_{a_1,k+2}^1), GE(B_{a_1,a_1}^7), GE(B_{a_1,a_2}^6), \\
& TP(B_{a_1,a_1}^1, B_{k,a_1}^6), TP(B_{a_1,a_1}^1, B_{k+1,a_1}^4), \\
& TP(B_{a_1,a_2}^2, B_{k,a_2}^6), TP(B_{a_1,a_2}^2, B_{k+1,a_2}^4).
\end{aligned}$$

716  
717 Operations that involve blocks in row or column  $a_1$  or  $a_2$  need not be executed for  
718 leaf nodes attached to outer branches. As a reminder of what was said in Section 2,  
719 at the end of these operations the  $k$ -th block-row of the global  $R$  factor is computed,

720 with a dark fill color in Figure 9 (right); the bordered blocks with a light fill color  
721 will be operated upon in nodes along the branch connecting the leaf to the root of the  
722 tree while the bordered blocks with no fill color correspond to the Householder  
723 vectors that implicitly represent the  $Q$  factor.

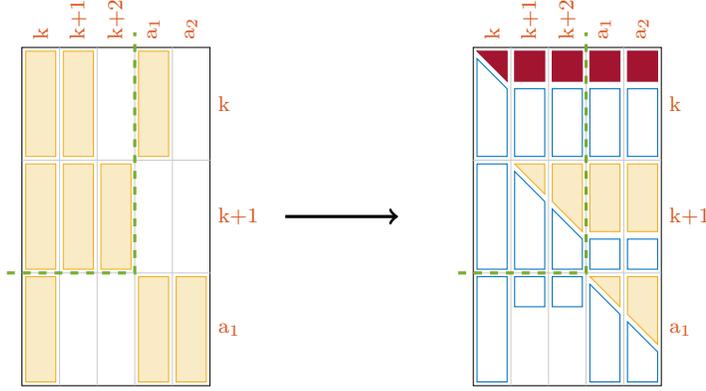


FIG. 9. The blocks concerned by the operations at leaf node  $k$ . Their structure before and after the processing of the node are shown in the left and right parts of the figure, respectively. The dashed line shows the difference between inner and outer leaf nodes.

724 *Separator nodes.* The blocks concerned by the operations on a separator node are  
725 depicted in Figure 10 (left). When all the chain nodes have been eliminated, each  
726 separator (except the two outer ones) is connected to two other separators: one is  
727 the parent node and the other an ancestor node in the tree. Therefore, these blocks  
728 are in columns  $s_1, s_2, p_1, p_2, a_1, a_2$  and rows  $s_1, s_2, p_1, a_1$ , where  $s_1$  and  $s_2$  are  
729 the block-columns associated with the separator,  $p_1$  and  $p_2$  those associated with the  
730 parent separator and  $a_1, a_2$  those associated with the ancestor separator. Separators  
731 belonging to one of the two outer branches are only connected to their parent and,  
732 therefore, columns  $a_1, a_2$  and row  $a_1$  do not exist; this is illustrated by the dashed  
733 line in Figure 10. Finally, the root node is a separator which does not have a parent  
734 nor an ancestor and, therefore, only the first two rows and columns are involved in  
735 its processing, as illustrated by the dashed purple line in Figure 10. The operations  
736 executed on a separator node result in the structure shown in the right part of the  
737 figure and are described by the following pivotal sequence:

$$\begin{aligned}
& TP(B_{s_1, s_1}^1, B_{s_2, s_1}^1), \quad GE(B_{s_2, s_2}^1), \quad TP(B_{s_2, s_2}^1, B_{s_1, s_2}^2), \quad TP(B_{s_2, s_2}^1, B_{s_2, s_2}^2), \\
& TP(B_{p_1, p_1}^1, B_{s_1, p_1}^2), \quad TP(B_{p_1, p_1}^1, B_{s_2, p_1}^2), \\
738 \quad & TP(B_{p_1, p_2}^2, B_{s_1, p_2}^2), \quad TP(B_{p_1, p_2}^2, B_{s_2, p_2}^2), \\
& TP(B_{a_1, a_1}^1, B_{s_1, a_1}^2), \quad TP(B_{a_1, a_1}^1, B_{s_2, a_1}^2), \\
& TP(B_{a_1, a_2}^2, B_{s_1, a_2}^2), \quad TP(B_{a_1, a_2}^2, B_{s_2, a_2}^2).
\end{aligned}$$

739 All the operations involving rows and columns  $a_1$  and  $a_2$  need not be done on  
740 outer separators and the root; all those involving rows and columns  $p_1$  and  $p_2$  need  
741 not be done on the root.

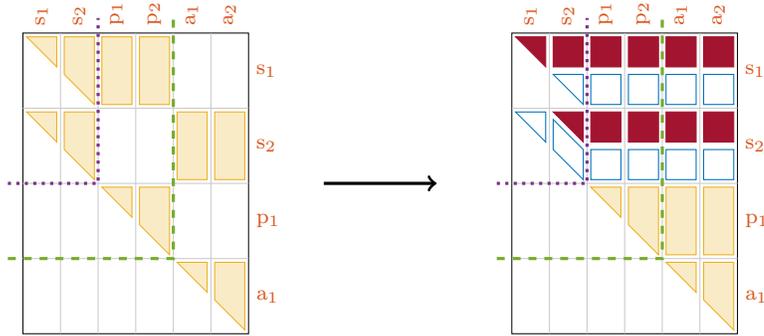


FIG. 10. The blocks concerned by the operations at separator node  $(s_1, s_2)$ . Their structure before and after the processing of the node are shown in the left and right parts of the figure, respectively. The dashed line shows the difference between inner and outer separator nodes. The dotted line delimits the rows and columns of the tree root node.