SIAM J. SCI. COMPUT.
Vol. 42, No. 6, pp. C313–C334

PARALLEL QR FACTORIZATION OF BLOCK-TRIDIAGONAL MATRICES*

ALFREDO BUTTARI[†], SØREN HAUBERG[‡], AND COSTY KODSI[†]

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 which relies on task parallelism and makes use of 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, task based parallelism, nested dissection

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

DOI. 10.1137/19M1306166

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 nonzeroes 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 subdiagonal 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* [10] or *communication-avoiding* [13] 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 nonzeroes by the factorization—which is responsible for an increase in the operation count and the storage. We propose an approach that computes this per-

*Submitted to the journal's Software and High-Performance Computing section December 11, 2019; accepted for publication (in revised form) July 31, 2020; published electronically November 2, 2020.
https://doi.org/10.1137/19M1306166
Funding: The work of the first author was partially supported by 3IA Artificial and Natural Intelligence Toulouse Institute, French "Investing for the Future—PIA3" program under grant agreement ANR-19-PI3A-0004 and by the SOLHARIS project, ANR AAPG 2019 program under grant agreement ANR-19-CE46-0009. The work of the second author was supported by VILLUM FONDEN through grant 15334. The work of the second and third authors was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program through grant 757360.

[†]Université de Toulouse, CNRS, 31055, Toulouse, Cedex 4, France (alfredo.buttari@irit.fr).
[‡]Department of Applied Mathematics and Computer Science, DTU, Lyngby, Denmark (sohau@dtu.dk, coko@dtu.dk).

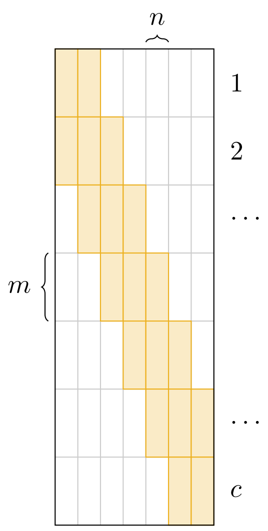

FIG. 1. *A block-tridiagonal matrix.*

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

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

It has been shown in [18, 4] that an effective approach for unsupervised ML involves treating the learned representation as a *Riemannian manifold.* An unfortunate implication, however, is that vector space operations are no longer applicable. Instead, Riemannian counterparts, such as *logarithm* and *exponential maps*, take the place of subtraction and addition operations. Also, *geodesics* in their role as generalizations of line segments in Euclidean space become prominent. Just as a segment of a line is the shortest distance connecting two distinct points, a geodesic on a Riemannian manifold is a smooth parametric curve that is a local length minimizer connecting any two distinct points. Additionally, a point moving along a geodesic does not experience acceleration in the tangent direction, i.e., the velocity has a constant magnitude.

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

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

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

1.2. Related work. Our work can be related to the factorization of banded, (block) bi- or tridiagonal matrices. Most of the approaches presented in the literature draw parallelism from *cyclic reduction* [21], *wrap-around* [19], or partitioning methods or minor variants thereof; these techniques amount to identifying independent sets of computations that allow for reducing multiple (block) columns at once at the cost of additional computations due to fill-in. The method we propose in this work relies on the use of nested dissection (see sections 2.1 and 2.2) and, essentially, leads to permutations that are equivalent to those computed with cyclic reduction or wrap-around; because nested dissection relies on graph theory, we believe it provides a clearer framework that allows for a better understanding and analysis of parallelism and fill-in.

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

Arbenz and Hegland [3] propose a method for the stable solution of (periodic) square banded linear systems. The factorization is achieved in two phases assuming that the original matrix is permuted into *standard form* (i.e., lower-banded periodic) through a circular shift; this permutation introduces some unnecessary fill-in. In the first phase, a partitioning of the matrix (which amounts to a dissection step) and block-columns associated with the resulting parts are reduced independently. This leads to a periodic block-bidiagonal system which is reduced in parallel using cyclic reduction. They discuss both the LU and QR factorization although mostly focus on the first. Note that the QR factorization of square, periodic block-bidiagonal systems through cyclic reduction is also addressed by Hegland and Osborne [20]

None of the aforementioned approaches consider the case of overdetermined systems; this has nontrivial implications because when a block-column is reduced, some coefficients along the corresponding pivotal block-row remain which have to be reduced. Also, all of these methods only draw parallelism from the matrix permutation or partitioning whereas our approach exploits multiple levels of parallelism as described in section 3.

All of the methods referenced, including ours, are, in essence, specialized multifrontal methods [14]. Algebraic multifrontal methods for the QR factorization of sparse matrices have been proposed in the literature by Amestoy, Duff, and Puglisi [2], Davis [12], and Buttari [9], for instance. These methods and the corresponding tools are designed for generic sparse matrices; although they can obviously be used for solving block-tridiagonal systems, they may not be the best suited tools. Indeed, unlike common sparse matrices, our block-tridiagonal systems may be relatively dense in the beginning (multiple thousands of coefficients per row) and fill up moderately during the factorization (see section 2.4). As a result, algebraic sparse multifrontal solvers may suffer from the excessive overhead due to the handling of the coefficients of the original matrix based on indirect addressing. Clearly, these methods could be extended to efficiently handle block matrices; this is the object of future work. Additionally, multifrontal methods involve explicit assembly of frontal matrices which can be avoided in our case due to the simple, regular structure of the data (see section 2.3). Finally, in sparse multifrontal solvers, frontal matrices are typically reduced using a LAPACK-like factorization which does not fully take into account their possibly sparse nature as explained in section 2.3.

2. Algorithm.

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

2.1. Preliminaries.

As commonly done in the literature [11, 9], our symbolic analysis of the QR factorization of a sparse matrix relies on the equivalence between the R factor of a real matrix A and the Cholesky factor of the normal equation matrix $B = A^T A$, whenever the strong Hall property holds. As a result of this equivalence, the QR factorization of a sparse matrix A follows the same computational pattern as the Cholesky factorization of B. This can be modeled using the *adjacency graph* of B defined as a graph $\mathcal{G}(B) = (\mathcal{V}, \mathcal{E})$ whose vertex set $\mathcal{V} = \{1, 2, \ldots, cn\}$ includes the unknowns associated with rows and columns of B and edge set $\mathcal{E} = \{(i, j) \; \forall \; b_{i,j} \neq 0\}$ includes an edge for each nonzero coefficient of B. In our case, the symmetry of B implies that if (i, j) is in $\mathcal{E}$, then so is (j, i), and therefore we will only represent one of these edges; such a graph is called *undirected*.

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

A sparse factorization can achieve better parallelism than a dense one because sparsity implies that the elimination of one unknown does not affect all the uneliminated ones but only part of them. The dependencies between variables are represented by the so-called directed filled graph which corresponds to the original graph $\mathcal{G}(B)$ plus all the fill edges; in this graph an edge (i, j) is directed from i to j if j is eliminated after i and expresses the fact that j depends on i. Because many of the dependencies expressed by the filled graph are redundant, its transitive reduction is computed to obtain the smallest complete set of dependencies; this results in a tree graph called an *elimination tree*. The elimination tree plays a central role in sparse matrix factorizations [23] because it dictates the order in which the unknowns can be eliminated: any order that follows a topological (i.e., bottom-up) traversal of the tree leads to the same result. A consequence of this fact is that nodes belonging to different branches are independent and can, thus, be eliminated in parallel. Nodes of the elimination tree are commonly amalgamated into supernodes containing nodes that share the same neighbors in the filled graph and can thus be eliminated at once; the resulting amalgamated tree is commonly referred to as an *assembly tree*.

Nested dissection is a method introduced by George [16] to reduce the complexity of sparse factorizations. The basic step of this method is based on the idea of computing a separator $\mathcal{S}$ of a graph $\mathcal{G}$; this is defined as a subset of nodes which splits the graph into two nonconnected subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$. If the nodes in $\mathcal{S}$ are eliminated last, no fill-in is possible between the nodes of $\mathcal{G}_1$ and those of $\mathcal{G}_2$ because of the Rose, Tarjan, and Lueker theorem [26]. This generally reduces the overall amount of fill-in and improves parallelism because the nodes of $\mathcal{G}_1$ can be eliminated independently from, and thus in parallel with, those of $\mathcal{G}_2$. Nested dissection applies this procedure recursively until subgraphs of a given minimum size are achieved. The resulting tree of separators matches the assembly tree.

2.2. A nested dissection based approach.

Because our matrices are made of dense blocks, our analysis can focus on the *compressed graph* where a node represents all the unknowns in a block-column and an edge represents a set of edges that connect all the unknowns in one block-column to all those in another block-column. Figure 2 shows the compressed graph for the normal equation matrix $B = A^T A$; this can be easily derived knowing that $b_{i,j} \neq 0$ if there exists a row k in A such that $a_{k,i} \neq 0$ and $a_{k,j} \neq 0$. Because all the columns (rows) within a block-column (block-row) are indistinguishable, we will use the term "column" ("row") to refer to a block-column (block-row).

A few observations can be made about this graph. First, regardless of the order in which unknowns are eliminated, some fill-in will appear in R compared to A. In fact, although block-row k of A only has nonzero blocks in block-columns $k-1$ to $k+1$, R will also have a nonzero block in block-column $k+2$; these fill-in blocks are represented by the horizontal edges in Figure 2. We will refer to these fill blocks as *unavoidable* because, as said above, they will exist in R regardless of the nodes elimination order.

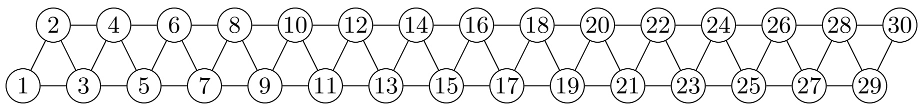

Fig. 2. *Compressed graph for the normal equation matrix $B = A^T A$. The labels inside nodes show the unknown number which, in this case, corresponds to the order in which unknowns are eliminated.*

Second, besides these unavoidable fill-in blocks, no additional fill-in is introduced in R by the factorization if unknowns are eliminated in the natural order. This is because, upon elimination of node k, its neighbors are already connected to each other. Third, the transitive reduction of the graph in Figure 2 leads to a linear graph, i.e., a tree with a single branch which does not provide any opportunities for parallelism.

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

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

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

A block QR factorization algorithm can be roughly described as a method which runs in c steps, c being the number of block-columns, where at step k all the nonzero blocks in column k are annihilated except one which is, instead, reduced to an upper triangle; as a result of this column reduction we obtain one block-row of the global R factor. In our method, this is achieved through a bottom-up traversal of the tree where, at each node, one column (or two for nodes associated with separators) is reduced. Reducing column k implies the update of a subset of the blocks in column

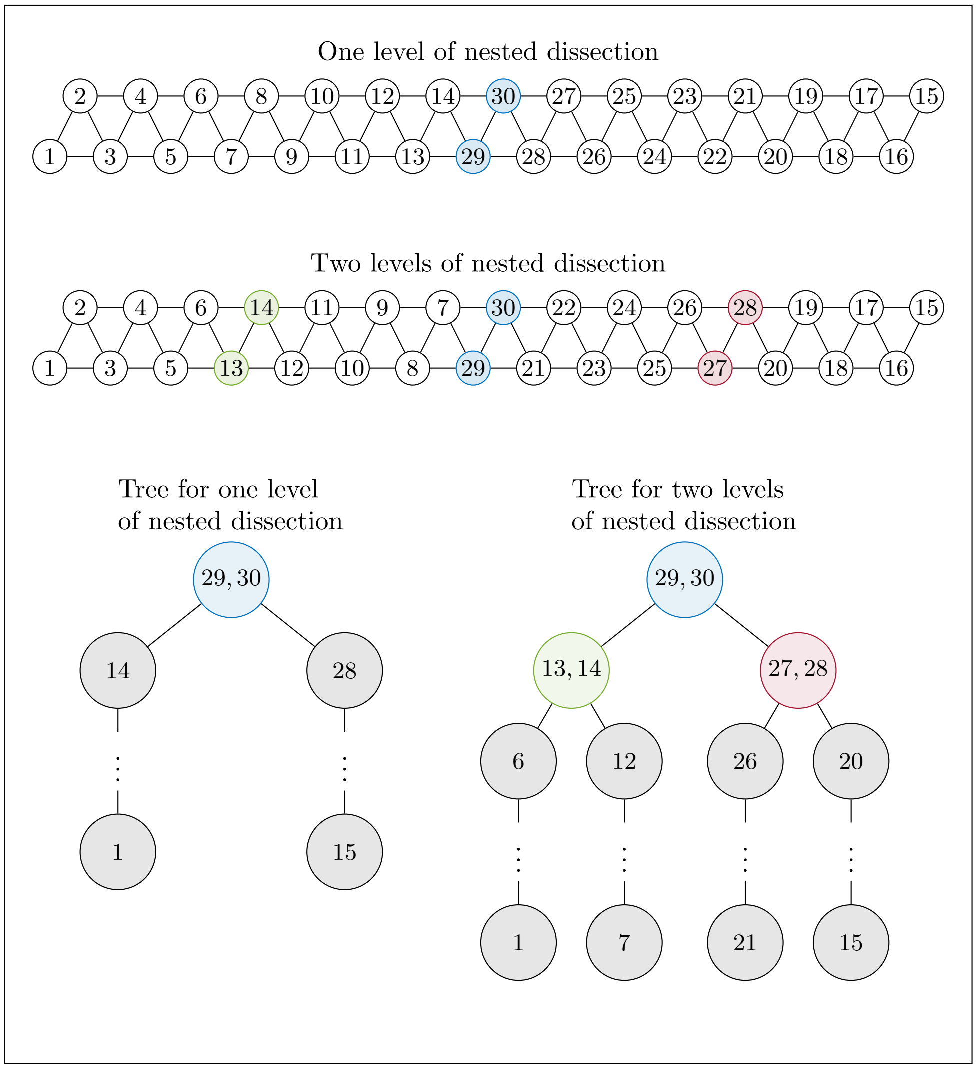
One level of nested dissection
Two levels of nested dissection
Tree for one level of nested dissection
Tree for two levels of nested dissection
29, 30
14
28
1
15
13, 14
27, 28
6
12
26
20
1
7
21
15

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

$k+1,\dots,c$; we refer to the ensemble of the blocks affected by the reduction of one column as a *frontal matrix*. Depending on the size and shape of this frontal matrix, the nodes of the tree can be classified into four different types: leaf, chain, separator, and root nodes. The first corresponds to the nodes that are at the lowest level of the tree (e.g., node 7 or 15 in the bottom-right tree of Figure 3); the second corresponds to nodes in the sequential branches associated with the subdomains resulting from nested dissection (e.g., node 12 or 20 in Figure 3); the third corresponds to nodes belonging to some separator (e.g., node $(13, 14)$ in Figure 3); the fourth is the root node of the tree. For each type, a different sequence of transformations is used to operate the corresponding column reduction; we refer to this as the *pivotal sequence*. Moreover, for each type, nodes belonging to the two outer branches need some special treatment because, as explained above, reducing the associated columns does not

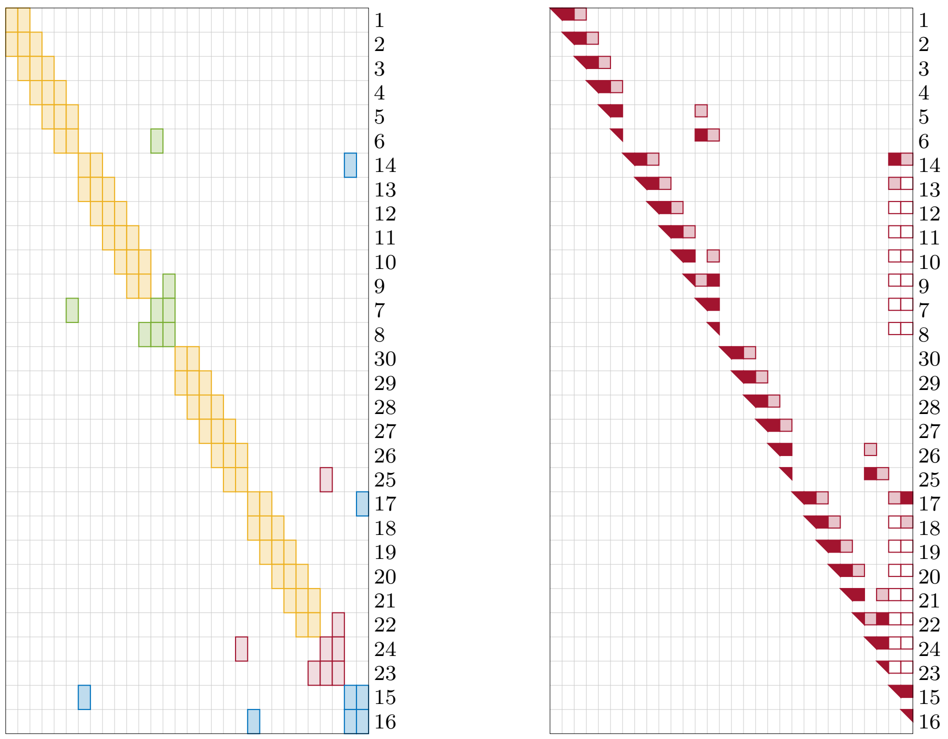

FIG. 4. *On the left, a block-tridiagonal matrix with* $c = 30$ *permuted using two levels of nested dissection as in Figure* 3*. Note that, for the sake of illustration, block-columns are permuted in the same way as block-rows, although this need not be the case in practice. The numbers on the right 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.*

introduce any additional fill-in in the R factor; this will be discussed in deeper detail in the next section.

2.3. Node elimination. As explained in the previous section, at each node of the tree, one block-column (two for separator nodes) is reduced by annihilating all the nonzero blocks in the column except one which is reduced into a triangle. The annihilation of a block involves not only its block-row but also a second one which we refer to as *pivotal row*; if, for example, the blocks are of size 1×1, this can be achieved by a Givens rotation. In our case, because we are dealing with dense blocks of potentially large size, Householder reflections will be used instead because they heavily rely on level-3 BLAS operations and, therefore, can achieve much higher performance on modern processors; the LAPACK library provides the basic operations to achieve these transformations which are presented below. As a result of a block annihilation in column k, the structure of both the involved rows in columns $k+1, \ldots, c$ will be equal to the union of their prior structures. This implies that some fill-in may be introduced by the transformation; obviously this fill-in depends on which pivotal row is chosen to annihilate a block. It must be noted that different pivotal sequences always lead to the same (numerically) R factor which only depends on column permutations; as a

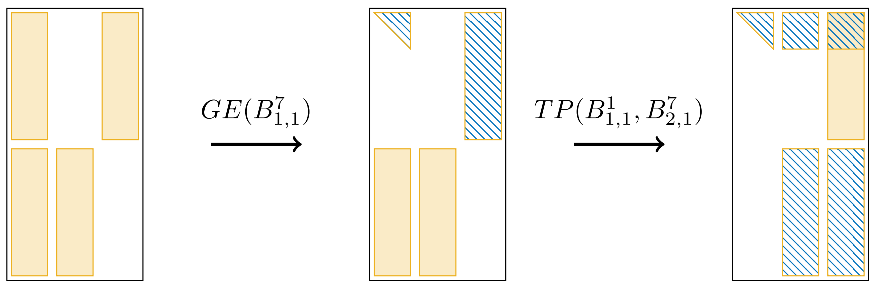

FIG. 5. *Example showing the use of the notation used to describe pivotal sequences. The data modified at each step are shown with a fill pattern.*

consequence, different pivotal sequences only have an effect on the trailing submatrix of A and, ultimately, the Q factor. Several strategies were proposed in the past for choosing pivotal sequences that reduce the fill-in. The approach that we adopted in our work relates to the *variable pivoting* technique proposed by Gentleman [15] for the QR factorization of sparse matrices by means of Givens rotations; this method is strongly related to the *row-merge* scheme, later proposed by George and Heath [17] and Liu [22] (see the work by Liu for a discussion on how the two approaches compare). Variable pivoting can be summed up as a method where, when an entry in position (i, k) has to be annihilated, a pivotal row j is chosen such that its structure is as close as possible to that of row i.

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

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

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

For the sake of readability, here we only provide the pivotal sequence for a chain node because this is where most of the computations are done assuming $c \gg 2^l$; the other node types are discussed in the appendix. Reducing a column k associated with

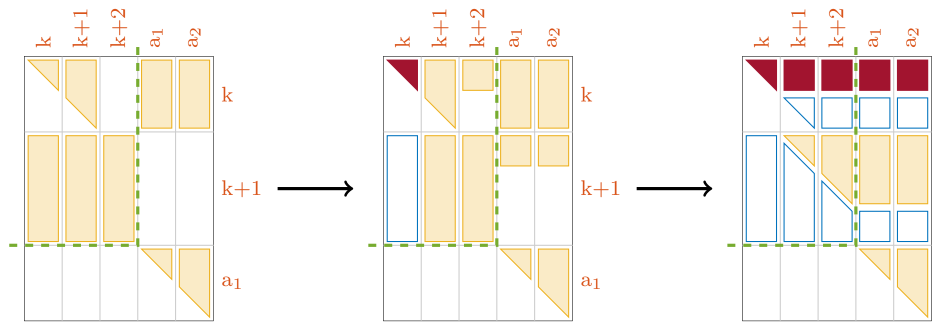

Fig. 6. *The structure of the frontal matrix associated with chain node k at the beginning of the node reduction (left), after the first two steps $GE(B^7_{k+1,k})$, $TP(B^1_{k,k}, B^1_{k+1,k})$ (middle), and at the end of the node reduction (right). The dashed line shows the difference between inner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

one of these nodes affects blocks in columns k, $k+1$, $k+2$, a_1, and a_2 and rows k, $k+1$, and a_1 where a_1 and a_2 are the indices of the nodes in the separator that the chain is moving away from. For example, when node 8 in Figure 3 (bottom-right) is visited, it is connected to nodes 9, 10, 29, and 30 (because node 7 has been previously eliminated), which explains why these columns are concerned by its elimination. Note also that when node k is visited, block-rows k and a_1 and block-columns k, $k+1$, a_1, and a_2 have already been updated upon processing of nodes that are lower in the same branch; this leads to the particular frontal matrix structure which is illustrated in the left part of Figure 6. The pivotal sequence that we use on chain node k is

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

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

This process closely resembles the QR multifrontal method [2, 12, 9]. It must be noted, however, that in the multifrontal method, frontal matrices are explicitly assembled by copying coefficients into dense matrix data structures which are allocated in memory and initialized to zero beforehand; these copies can be extremely costly due

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.

`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 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.

	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	$\mathcal{M}_{ro} = \underline{2mn}$	$\mathcal{M}_{rm} = 0$

to the large size of blocks and the heavy use of indirect addressing. In our approach, instead, the frontal matrices need not be formed explicitly but the constituent blocks can be easily accessed through pointers (see section 3 for further details). Additionally, the multifrontal method can only take advantage of the zeroes in the bottom-left part of frontal matrices (this is referred to as “Strategy 3” in the work of Amestoy, Duff, and Puglisi [2]), whereas, through the use of variable pivoting, our approach can avoid more unnecessary computations.

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

Summing up the values in Table 1 for the operations of the pivotal sequence from the previous section yields the value in row two, column three of Table 2. For a node in an outer chain, the cost, reported in row and column two, is lower because fewer operations are needed, as explained in the previous section. In the top rows of the table we also report the operational complexity of all the other node types and we refer the reader to the appendix for details of the related pivotal sequences. With a slight abuse of notation, we denote “outer root” the root of the tree for the case $l = 0$ and “middle root” the tree root for the case $l > 0$.

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

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

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

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

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

Figure 7 (top) and (middle) shows how the relative cost of the factorization, respectively, $\mathcal{F}(c, l)/\mathcal{F}(c, 0)$ and $\mathcal{M}(c, l)/\mathcal{M}(c, 0)$, varies for different problem sizes and different levels of nested dissection for the case where $m = 2n$; it is possible to see that, for growing values of l and c, these curves converge to, respectively, $\mathcal{F}_{cm}/\mathcal{F}_{co}$ and $\mathcal{M}_{cm}/\mathcal{M}_{co}$ because for $l = 0$ the tree is essentially made of a single outer chain, whereas for large l and c values the vast majority of computations are done in nodes belonging to middle chains. For growing values of m, these two ratios converge, respectively, to $2.3\overline{3}$ and $1.6\overline{6}$.

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

3. A solver with multiple levels of parallelism.

In this section, we describe how an actual parallel implementation of the described algorithm for shared memory systems (e.g., single node, multicore machines) was achieved.

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

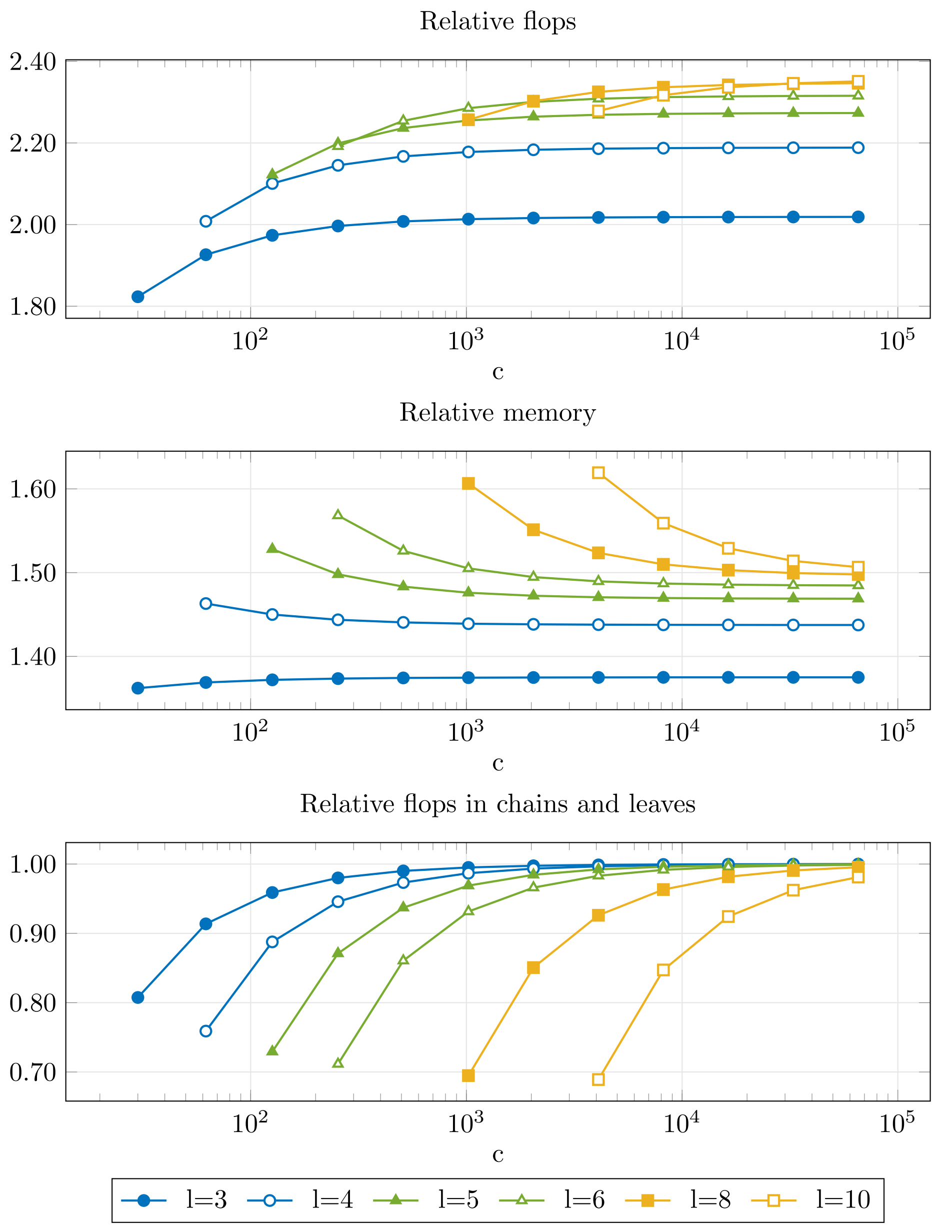

FIG. 7. *Relative number of floating point operations (top) and memory consumption (middle) with respect to no nested dissection based permutation. Relative number of floating point operations performed in chain of leaf nodes with respect to the total (bottom). $m = 2n$ is assumed for all.*

One immediate advantage of this choice is that in `qr_mumps` these operations are efficiently parallelized through the use of scalable *tiled* algorithms [10] (also referred to as *communication avoiding*): blocks are divided into square tiles of size $b \times b$ which are eliminated by means of block Householder transformations according to different schemes [8] which aim at improving concurrency by reducing unnecessary synchronizations and communications. Our implementation of the `tpqrt` and `tpmqrt` operations relies on the same approach and uses a tile elimination scheme which is referred to as Sameh–Kuck by Bouwmeester et al. [8]: the tiles in the trapezoidal block (remember that we only consider the two special cases where the trapezoidal degenerates to a triangle or a rectangle) along a column are eliminated one after the other starting from the top using the corresponding diagonal tile in the triangular block. Through the use of these parallel kernels some parallelism can be achieved within each node of the tree and even in the case where no nested dissection permutation is used.

`qr_mumps` relies on task based parallelism for the implementation of these methods. The workload is represented in the form of a DAG where nodes are elementary operations (e.g., elimination of a tile) and edges are the dependencies among them; this representation allows for very quick access to all the available concurrency at any moment during the execution and for using very dynamic and asynchronous execution approaches. `qr_mumps` achieves task based parallelism through the use of a *runtime system*, namely StarPU [5], which provides a programming interface for defining the DAG and a scheduling engine for tracking the status of the DAG and triggering the execution of ready tasks on the available processing units [1]. StarPU provides a programming API based on the sequential task flow model (sometimes also referred to as *superscalar*). In this model, a master thread submits tasks to the runtime specifying how each of them accesses the data (whether in read or in write mode); based on this information and the tasks submission order, the runtime can automatically define the dependencies among tasks and generate the DAG. This model is also used in the OpenMP [24] standard (through the `task` directive and the `depend` clause) and other runtime systems.

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

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

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).

		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

4. Experimental results.

In this section we report experimental results that illustrate the behavior of the algorithm presented in section 2 and its implementation described in section 3. These were obtained on one node of the Olympe supercomputer of the CALMIP regional supercomputing center in Toulouse, France. One such node is equipped with two Intel Xeon Gold 6140 CPUs for a total of 36 cores. We have used the Intel compilers and the Intel MKL library v2018.2, StarPU revision 581387c from the public GIT repository[1] and `qr_mumps` revision 8b160df.[2]

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

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

[1]https://gitlab.inria.fr/starpu/starpu.

[2]http://buttari.perso.enseeiht.fr/qr_mumps/releases/qr_mumps_8b160df.tgz.

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).

		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

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

In order to assess the efficiency and scalability of our implementation, in Table 4 we provide the speed of computations in Gigaflops per second, which is insensitive to the computational overhead imposed by the use of nested dissection. As expected, performance grows for higher values of l because more embarrassing parallelism is available; for a given value of l, performance at first grows with the number of threads and eventually decreases when the available parallelism is not enough to feed all the working threads. The peak performance of a single core is 73.6 Gflop/s, which means that the sequential execution has an efficiency of 43%. This relatively poor efficiency can be explained by the relatively small values of m and n which makes the relative cost of BLAS-2 rich operations (i.e., `geqrt` and `tpqrt`) greater; higher speeds can be observed when larger blocks are used. The best speed improvement is a remarkable

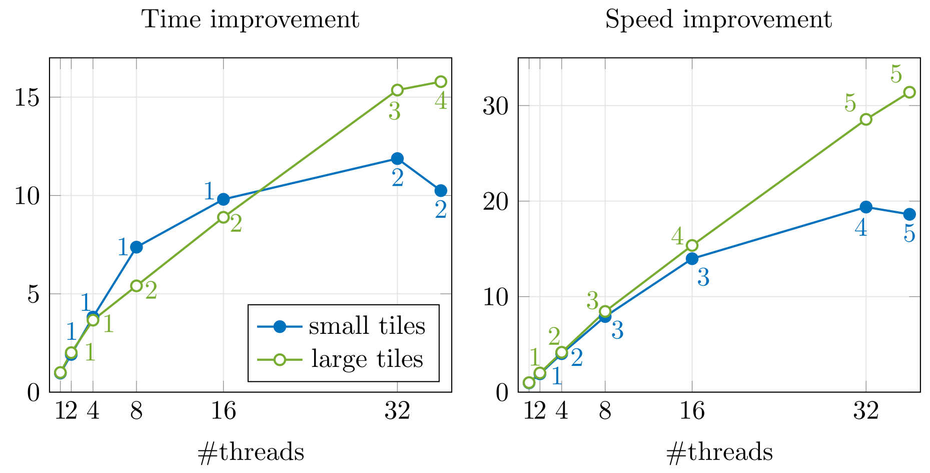

Fig. 8. *Best time and speed improvement over the sequential case. The value beside each data point shows the corresponding value of l.*

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.

	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

31.4 out of 36; it must be noted that for the case of $l = 5$, roughly 99% of the computations are done in chain or leaf nodes.

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

The above results suggest that in order to obtain the lowest execution time, a careful combination of the `qr_mumps` block size b and nested dissection depth l must be chosen. The optimal values mostly depend on the characteristics of the used platform such as number of cores, relative speed between cores and memory, and efficiency of low-level BLAS operations. The proposed approach proved to be scalable in our experimental setting because the shortest execution time is obtained using all the available threads; for smaller problems (i.e., smaller values of m, n, or c), using all the available cores may not be beneficial though. Because the cost of the factorization as a function of l is upper bounded by a modest constant and because most computations are performed in chain or leaf nodes (see Figure 7 and the related

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 $c = 2046$.*

	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

discussion), for relatively large values of c we can expect to achieve good scalability even on large numbers of cores.

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

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

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

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

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

5. Conclusion.

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

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

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

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

Appendix A. Pivotal sequences.

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

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

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

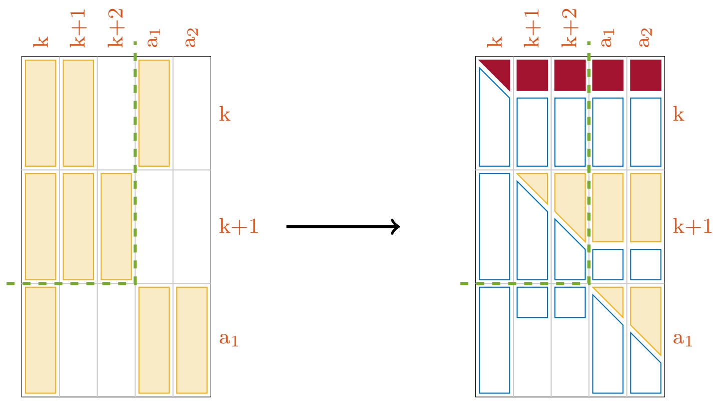

Fig. 9. *The blocks concerned by the operations at leaf node k. Their structure before and after the processing of the node is shown in the left and right parts of the figure, respectively. The dashed line shows the difference between inner and outer leaf nodes.*

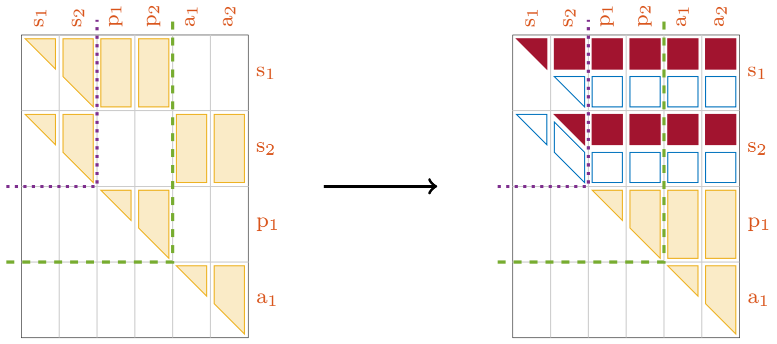

Fig. 10. *The blocks concerned by the operations at separator node (s_1, s_2). Their structure before and after the processing of the node is shown in the left and right parts of the figure, respectively. The dashed line shows the difference between inner and outer separator nodes. The dotted line delimits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

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

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

$$\begin{aligned}
&TP(B^1_{s_1,s_1}, B^1_{s_2,s_1}),\;\; GE(B^1_{s_2,s_2}),\;\; TP(B^1_{s_2,s_2}, B^2_{s_1,s_2}),\;\; TP(B^1_{s_2,s_2}, B^2_{s_2,s_2}),\\
&TP(B^1_{p_1,p_1}, B^2_{s_1,p_1}),\;\; TP(B^1_{p_1,p_1}, B^2_{s_2,p_1}),\\
&TP(B^2_{p_1,p_2}, B^2_{s_1,p_2}),\;\; TP(B^2_{p_1,p_2}, B^2_{s_2,p_2}),\\
&TP(B^1_{a_1,a_1}, B^2_{s_1,a_1}),\;\; TP(B^1_{a_1,a_1}, B^2_{s_2,a_1}),\\
&TP(B^2_{a_1,a_2}, B^2_{s_1,a_2}),\;\; TP(B^2_{a_1,a_2}, B^2_{s_2,a_2}).
\end{aligned}$$

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

REFERENCES

[1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, *Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems*, ACM Trans. Math. Software, 43 (2016), 13, https://doi.acm.org/10.1145/2898348.

[2] P R. Amestoy, I S. Duff, and C Puglisi, *Multifrontal QR factorization in a multiprocessor environment*, Int. J. Numer. Linear Alg. Appl., 3 (1996), pp. 275–300, https://doi.org/10.1002/(SICI)1099-1506(199607/08)3:4⟨275::AID-NLA83⟩3.0.CO;2-7.

[3] P. Arbenz and M. Hegland, *On the stable parallel solution of general narrow banded linear systems*, in High Performance Algorithms for Structured Matrix Problems, Adv. Theory Comput. Math. 2, Nova Science Publishers, Commack, NY, pp. 47–73.

[4] G. Arvanitidis, L. K. Hansen, and S. Hauberg, *Latent space oddity: On the curvature of deep generative models*, in Proceedings of the International Conference on Learning Representations (ICLR), 2018.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: A unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency Computation Practice Experience, 23 (2011), pp. 187–198, https://doi.org/10.1002/cpe.1631.

[6] Y. Bengio, A. Courville, and P. Vincent, *Representation learning: A review and new perspectives*, IEEE Trans. Pattern Anal. Mach. Intell., 35 (2013), pp. 1798–1828.

[7] M. W. Berry and A. Sameh, *Multiprocessor schemes for solving block tridiagonal linear systems*, Int. J. Supercomputing Appl., 2 (1988), pp. 37–57, https://doi.org/10.1177/109434208800200304.

[8] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert, *Tiled QR factorization algorithms*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, https://doi.org/10.1145/2063384.2063393.

[9] A. Buttari, *Fine-grained multithreading for the multifrontal QR factorization of sparse matrices*, SIAM J. Sci. Comput., 35, pp. C323–C345, https://doi.org/10.1137/110846427.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Comput., 35 (2009), pp. 38–53, https://doi.org/10.1016/j.parco.2008.10.002.

[11] T. Davis, *Direct Methods for Sparse Linear Systems*, Fundam. Algorithms 2, SIAM, Philadelphia, 2006, https://doi.org/10.1137/1.9780898718881.

[12] T. A. Davis, *Algorithm* 915*, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization*, ACM Trans. Math. Software, 38 (2011), 8, https://doi.org/10.1145/2049662.2049670.

[13] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput., 34 (2012), pp. 206–239, https://doi.org/10.1137/080731992.

[14] I. S. Duff, A. M. Erisman, and J. K Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, UK, 2017.

[15] W. M. Gentleman, *Row elimination for solving sparse linear systems and least squares problems*, in Numerical Analysis, G. Alistair Watson, ed., Springer, Berlin, pp. 122–133, https://doi.org/10.1007/BFb0080119.

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

[17] A. J. George and M. T. Heath, *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra Appl., 34 (1980), pp. 69–83.

[18] S. Hauberg, *Only Bayes Should Learn a Manifold*, arXiv:1806.04994, 2018.

[19] M. Hegland, *On the parallel solution of tridiagonal systems by wrap-around partitioning and incomplete LU factorization*, Numer. Math., 59 (1991), pp. 453–472, https://doi.org/10.1007/BF01385791.

[20] M. Hegland and M. Osborne, *Algorithms for block bidiagonal systems on vector and parallel computers*, in Proceedings of the 12th International Conference on Supercomputing, ACM, 1998, pp. 1–6, https://doi.org/10.1145/277830.277835.

[21] R. W. Hockney, *A fast direct solution of Poisson's equation using Fourier analysis*, J. ACM, 12 (1965), pp. 95–113, https://doi.org/10.1145/321250.321259.

[22] J. W. H. Liu, *On general row merging schemes for sparse Givens transformations*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 1190–1211.

[23] 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.

[24] OpenMP Architecture Review Board, OpenMP 5.0, Complete Specifications, 2018.

[25] E. Polizzi and A. H. Sameh, *A parallel hybrid banded system solver: The SPIKE algorithm*, Parallel Comput., 32 (2006), pp. 177–194, https://doi.org/10.1016/j.parco.2005.07.005,

[26] D. J. Rose, R. E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283, https://doi.org/10.1137/0205021.

[27] A. Sameh, *On two numerical algorithms for multiprocessors*, in Proceedings of the NATO Advanced Research Workshop on High-Speed Computing, Series F: Computer and Systems Sciences, Vol. 7, Springer, Berlin, 1984, pp. 311–328.

[28] A. H. Sameh and D. J. Kuck, *On stable parallel linear system solvers*, J. ACM, 25 (1976), pp. 81–91, https://doi.org/10.1145/322047.322054.