Parallelization of the solve phase in a task-based Cholesky solver using a sequential task flow model

Sébastien Cayrols, Iain S Duff and Florent Lopez

Abstract
We describe the parallelization of the solve phase in the sparse Cholesky solver SpLLT when using a sequential task flow model. In the context of direct methods, the solution of a sparse linear system is achieved through three main phases: the analyse, the factorization and the solve phases. In the last two phases, which involve numerical computation, the factorization corresponds to the most computationally costly phase, and it is therefore crucial to parallelize this phase in order to reduce the time-to-solution on modern architectures. As a consequence, the solve phase is often not as optimized as the factorization in state-of-the-art solvers, and opportunities for parallelism are often not exploited in this phase. However, in some applications, the time spent in the solve phase is comparable to or even greater than the time for the factorization, and the user could dramatically benefit from a faster solve routine. This is the case, for example, for a conjugate gradient (CG) solver using a block Jacobi preconditioner. The diagonal blocks are factorized once only, but their factors are used to solve subsystems at each CG iteration. In this study, we design and implement a parallel version of a task-based solve routine for an OpenMP version of the SpLLT solver. We show that we can obtain good scalability on a multicore architecture enabling a dramatic reduction of the overall time-to-solution in some applications.

Keywords
Sparse Cholesky, backward substitution, forward substitution, SPD systems, runtime systems, OpenMP

1. Introduction
In this study, we are solving the linear system

$$Ax = b$$ (1)

where $A$ is a large sparse symmetric positive-definite matrix. In order to do this, we use a direct method where the solution process consists of three main steps: the analyse, the factorization and the solve phases. We use a Cholesky factorization given by

$$P^TAP = LL^T$$ (2)

where $L$ is a sparse lower triangular matrix, and $P$ is a permutation matrix needed to preserve the sparsity. Since $L$ is triangular, the solution can be obtained using a forward substitution step

$$Ly = Pb$$ (3)

followed by a backward substitution step

$$L^Tz = y$$ (4)

so that $x = P^Tz$ is the solution to the system in equation (1). Note that, although the factor $L$ has a sparse structure, it is usually denser than $A$ because of fill-in. The purpose of the analyse phase is to determine precisely the structure of $L$ and to find a pivot order for limiting the fill-in. The returned permutation matrix $P$ reduces the storage for the factor as well as the number of floating-point operations for factorization and solution.

In this article, we concentrate on the steps in equations (3) and (4). Most work on sparse solvers concentrates on the factorization in equation (2) because that is the most costly step of the solution process. As a consequence, the solve phase is often not as optimized as the factorization in the state-of-the-art solvers, and opportunities for parallelism are often not exploited in this phase. However, in some applications, the time spent in the solve phase is comparable to or even greater than the time for the factorization, and the user could dramatically benefit from a faster solve routine. This is the case, for example, for a conjugate
In this section, we discuss in detail the forward and backward substitutions. We also compare our code with other state-of-the-art solvers such as PaStiX and HSL_MA87. Although the PTG model can offer some benefits in terms of performance over the STF model, it is generally much more difficult to use in practice especially in the context of sparse algorithms where the task dependency pattern is very irregular.

When using runtime systems, the runtime system is responsible for handling the DAG on the target architecture by managing the data dependencies, data coherency and task scheduling. One main advantage of this compared to using low-level synchronization mechanisms lies in the fact that it offers better portability and maintainability of the code. Furthermore, we conducted extensive experiments on using runtime systems in previous work for a sparse Cholesky factorization. We used both the STF model (Duff et al., 2018) and the PTG model (Duff and Lopez, 2018), in the context of the SpLLT solver, and we showed it was efficient in terms of performance when running on multicore architectures compared to the state-of-the-art solver HSL_MA87 from the Harwell Subroutine Library (HSL) (http://www.hsl.rl.ac.uk/).

In this article, we extend our approach to the solve phase in SpLLT and design DAG-based algorithms for implementing the parallel version of the forward and backward substitutions. Our implementation relies on an STF model, and the parallelization is achieved using the tasking features available in the OpenMP standard. We experimentally show that, using our DAG-based approach, we are able to obtain much better performance for computing the forward and backward substitutions on multicore architectures compared to the state-of-the-art solvers PARDISO, PaStiX and HSL_MA87.

In Section 2, we discuss the solve phase for dense systems, since we will use similar kernels in our sparse solve phase. We then describe the sparse solve algorithm in Section 3 before considering how to implement this in parallel in Section 4. We present our experimental results in Section 5 where we illustrate the scalability both with respect to the number of cores and to the number of right-hand sides (nrhs). We also compare our code with other state-of-the-art codes. Finally, we present some conclusions in Section 6.

2. DAG-based dense forward and backward substitutions

In this section, we discuss in detail the forward and backward substitutions for the case of dense matrices. For the sake of clarity and without loss of generality, we do not include the permutation matrix in this section. When performing these operations, we first partition the triangular dense factor $L \in \mathbb{R}^{n \times n}$ into blocks as:

$$L = \begin{bmatrix} L_{11} & L_{12} \\ L_{21} & L_{22} \\ & \ddots \\ L_{k1} & \cdots & L_{kk} \end{bmatrix}$$

(5)

where $L_{ii} \in \mathbb{R}^{nb \times nb}$ are lower triangular matrices for $1 \leq i < k$, and $L_{kk}$ is also a square lower triangular matrix but with dimension $n - (k - 1)nb \leq nb$.

We first discuss the forward substitution in Section 2.1 before considering the backward substitution in Section 2.2.

2.1. Forward substitution

Consider first the forward substitution in equation (3) that we present in Algorithm 1.

Algorithm 1. forwardSolve ($L$, $b$, $nb$)

Require: $L \in \mathbb{R}^{n \times n}$ partitioned as in equation (5), $b$ the right-hand side, $nb$ the block size

Ensure: $y$ the solution of the system $L y = b$

1: $k = \left\lceil \frac{n}{nb} \right\rceil$
2: for $i = 1$ to $k$
3: Solve $L_{ii} y_i = b_i$
4: for $j = i + 1$ to $k$
5: Update $b_j = b_j - L_{ji} y_i$
6: end for
7: end for

The DAG starts by solving the equation $L_{11} y_1 = b_1$, represented by the top node $S$. Once, $y_1$ is computed, an update of $b$ can be performed. The updates to $b_2$, $b_3$ and $b_4$ correspond to different parts of $b$, and they can all be performed in any order. The update of $b_3$ using $y_1$ and the
update of \( b_3 \) using \( y_2 \) have to be completed before solving \( L_{33}y_3 = b_3 \). However, as the destination space is the same for both updates, these two operations cannot be performed at the same time. We see from the DAG in Figure 1(b) that some parallelism can be exploited in the updates to the right-hand side.

### 2.2. Backward substitution

Once the forward substitution has returned the vector \( y \), the solve phase performs a backward substitution to solve equation (4). In this equation, we use the transpose of \( L \) to compute the solution vector \( x \). However, this substitution does not really differ from the previous one, only the loop order is changed, as shown in Algorithm 2. Similar to the forward substitution, this algorithm requires two computational kernels, the solve kernel and the update kernel, which give the same DAG as shown in Figure 1(b).

### 3. DAG-based solve in sparse case

In this section, we extend the algorithms presented in Section 2 to the sparse case. We now consider \( A \) as a sparse matrix of dimension \( n \times n \). The first step for solving equation (1) in SpLLT is the analysis of the pattern of the matrix \( A \) in order to reduce the fill-in in \( L \) during the factorization. This approach to solving sparse systems is well-documented, and we refer the reader to Duff et al. (2017) for the details. This step can use an ordering algorithm such as AMD (Amestoy et al., 1996) or Metis (Karypis and Kumar, 1998). The analysis then generates a tree representation for the factorization process. We first present the internal representation of the \( L \) factor in SpLLT and then discuss the design and implementation of the forward and backward substitutions.

#### 3.1. Internal representation of the L factor

At the root node of the tree, that part of the \( L \) factor is held as a dense lower triangular matrix, and we will partition it as in equation (5). For the other tree nodes, the \( L \) factor is stored as a dense trapezoidal matrix, \( \tilde{L} \), that can be written as a square lower triangular matrix \( \tilde{L}_1 \) stacked over a rectangular matrix \( \tilde{L}_2 \), namely

\[
\tilde{L} = \begin{pmatrix} \tilde{L}_1 \\ \tilde{L}_2 \end{pmatrix}
\]  

In the parlance of tree-based factorizations, the block \( \tilde{L}_1 \) corresponds to variables that are fully summed, that is to say variables that are appearing for the last time and do not appear at any ancestor nodes in the tree, so the corresponding unknowns can be solved as in Algorithm 1. The entries in \( \tilde{L}_2 \) are then used to perform updates to the variables that will later be fully summed at the parent or ancestor nodes of the tree. We can use these dependencies to generate a DAG for the whole solve phase that allows us to use inter-level parallelism. That is, a task at a node can be processed before some of the tasks for its children as long as the task has all its dependencies satisfied. We emphasize in passing two properties of the tree. Firstly, the rows within a block in \( \tilde{L}_2 \) may match rows of more than one block in the parent. Secondly, one row in the fully summed part of a node can be shared with more than one child node.

We illustrate this in Figure 2, where the factor \( L \) consists of three nodes. The parts of the DAG associated with the leaves of the tree are very similar to the DAG shown in Figure 1. We label the blocks from the leaves to the root. If
there are rows in common between blocks blk7 and blk15, then there will be a dependency between \( y_2 \) and \( y_7 \). Moreover, we observe that there is at least one row present in all three nodes, so that the solve using the corresponding block in the parent node requires the updates from both the children. Note that the first solve task in the root node has no dependency with its children. The resulting internode parallelism means that the solve of the corresponding part of the solution vector, \( y_5 \), can be executed before the tasks at the child nodes complete.

For efficient implementation of the forward substitution, we must modify Algorithm 1 to take account of three elements. Firstly, the \( L \) factor is spread over the nodes of the tree. Secondly, the shape of the \( L \) associated with each node of the tree is trapezoidal, except for the root. Thirdly, \( L_2 \) is composed of rows that have discontinuous indices in \( L \) so that indirect addressing is required.

### 3.2. Dense kernels and data movement

Our construction of the tree representation of the factor \( L \) is such that the rows of a node may not correspond to contiguous entries in \( b \). However, the computational kernels do not handle indirect addressing. This leads to data movement between the global vector and a local vector so that the data are contiguous in the local vector. Thus, a local vector is associated with each node of the tree and is split into two parts: the part associated with \( L_1 \) consists of a contiguous portion of the global solution vector, whereas the second part is a workspace. One way in which we avoid indirect addressing is to use the right-hand side vector permuted according to the pivot ordering. That is to say that, for the forward substitution, we work from the vector \( Pb \) in equation (3). Thus, the part of the vector corresponding to the triangular block \( L_1 \) will be contiguous in the global vector and so we can work directly on this when solving for the fully summed variables. We believe this to be a novel feature for the sparse solve phase. We do not actually permute the right-hand side prior to the forward substitution but only when the entries of the right-hand side are needed by the solve kernel.

#### 3.2.1. Forward solve and forward update kernels

We illustrate the two parts of \( L \) and the movement or equivalence of global and local vectors in Figure 3. As mentioned earlier, the part of \( \tilde{y} \) corresponding to \( L_1 \) is identical to the appropriate part of the global vector so no indirect addressing is required. The remaining part of \( \tilde{y} \) is a workspace local to the node. This workspace is related to the global vector through indirect addressing corresponding to row indices in the blocks of \( L_2 \). The first part of the local vector is set with the appropriate entries of \( Pb \). In addition, both parts of the
local vector may require a gather operation with workspaces of the children of the node. That is, when a row of \( \tilde{L} \) is present in a child of the node and the associated block in the child is processed, there is an update of the local vector of \( L \) using some entries from the child workspace. In Figure 3, we show in the dark shaded areas the solution for the fully summed variables that involve the block \( L_1 \) with the blocks \( y_i \) and \( y_{i+1} \) computed in-place. Since \( \tilde{L}_2 \) is composed of rows whose indices in \( y \) are discontinuous, indirect addressing is required in order to map these components into the contiguous local vector. These are then used for computing the matrix vector product \( \tilde{L}_3 y_j \), with \( j = 1, 2 \), and the resulting vector is subsequently scattered directly into the parent local vector. This is also a novel feature of the code.

3.2.2. Backward solve and backward update kernels. The backward substitution shares the same data locality issues as for the forward substitution. Consider the same node as in Figure 3 with its associated dense lower trapezoidal matrix \( \tilde{L} \). In order to update the solution vector \( z = \tilde{P}x \) from equation (4) with \( \tilde{L}_2^T \), the backward update kernel has first to gather the corresponding components of \( z \) into a local vector, denoted by \( \tilde{z} \) in Figure 4. Then, the update of \( \tilde{z} \) is performed by the computation of \( (L_{31})^T \tilde{z}_3 \) and \( (L_{32})^T \tilde{z}_3 \). Each triangular solve computes part of \( \tilde{P}x \). That is, the result of the triangular solve is then permuted to obtain the solution vector \( x \).

3.3. Sequential sparse forward solve algorithm

Moving from the dense forward solve in Algorithm 1 to the sparse case requires several changes. Firstly, as shown in Figure 2, the \( L \) factor is now a tree, where the nodes are associated with trapezoidal matrices, and has internode dependencies. Secondly, as shown in Figures 3 and 4, processing \( \tilde{L}_2 \) leads to indirect addressing. We present the sequential sparse solve in Algorithm 3. This algorithm takes as input the same parameters as Algorithm 1, except that the representation of the factor \( L \) is a list of nodes of the tree. The forward substitution operates over the nodes from the leaves to the root of the tree. For each node, we have the associated dense matrix \( L_i \), as written in equation (6), and a local vector \( \tilde{y} \). We start by forming the part of \( \tilde{y} \) associated with \( L_1 \). That is, we gather the corresponding components of \( \tilde{P} \) with the contributions of the children of the node. This operation overwrites the content so that we avoid resetting the local vector. Then, \( \tilde{L}_1 \) is processed as in Algorithm 1. An additional loop is introduced in Algorithm 3 to process \( \tilde{L}_2 \). This step is done so that the result of the first update of each \( \tilde{y}_j \), with \( j \in \{i + 1, \ldots, l\} \), overwrites its content. Finally, we sum the contribution from the children to the local vector.

The backward substitution algorithm is very similar but, in this case, we use the \( L \) factor starting with the root node and working down the tree to the leaf nodes.

4. Parallel implementation of the solve phase

As mentioned in Section 1, the parallel implementation of our task-based algorithms, introduced Section 3, is achieved using a runtime system which is responsible for executing the DAG in parallel on the target architecture. This involves enforcing data dependencies to ensure the correctness of the execution and managing the task scheduling for efficiently exploiting the resources. This design strategy has the advantage of producing a code that is portable and easy to maintain.

For our implementation, we decide to rely on an STF model for its greater usability than the PTG model in the context of complicated algorithms such as sparse forward
and backward substitutions. The difficulties for exploiting a PTG model compared to the STF model were exposed in previous work for the parallelization of the factorization phase in SpLLT (Duff and Lopez, 2018). The simplicity of the STF model lies in the fact that the parallel code is very similar to the sequential one. In this model, the tasks are submitted to the runtime system following the sequential algorithm and for each task, the manipulated data must be associated with a data access such as Read (R), Write (W) and Read–Write (RW) depending on the operations performed on the data. Using the task submission order and data access information, the runtime system is capable of building the DAG such that the sequential consistency is guaranteed during the parallel execution. Finally, we choose to use the OpenMP standard for our implementation, because we observed in previous work that it comes with a slightly lower overhead for the task management with OpenMP.

4.1. Data dependencies of a task

From the DAG as presented in Figure 2(b), we associate a task with a computational kernel that takes as input a block within a node of the tree and the related part of the input and output vector. The sparse aspect requires an additional task that handles the indirect addressing. This task forms the local vector by adding the contributions of the children.

In the following, we first show how the data dependency is determined using the representation of the factor \( L \), and then, we describe the task management with OpenMP.
Algorithm 4. getForwardDependencies \((L, \text{node}, \beta_\text{dep})\)

\[
\begin{align*}
\text{Require: } L & \in \mathbb{R}^{m \times n} \text{ stored as a tree,} \\
& \text{node is a node in } L, \\
& \mathcal{I} \text{ is a set of row indices in } L. \\
\text{Ensure: } & \text{dep is a set of vectors that share indices with } \mathcal{I} \text{ in } L. \\
1: & \text{dep} \leftarrow \emptyset \\
2: & \text{Let } C \text{ be the list of the children of node} \\
3: & \text{for } \text{child} \in C \text{ do} \\
4: & \text{Let } L_{\text{child}} \text{ be the row indices of the node } \text{child} \text{ in } L \\
5: & \mathcal{I}_\text{child} \leftarrow L_{\text{child}} \cap \mathcal{I} \\
6: & \text{Let } \text{dep}_\text{child} \text{ be the set of vectors in node } \text{child} \text{ that contains the indices in } \mathcal{I}_\text{child} \\
7: & \text{dep} \leftarrow \text{dep} \cup \text{dep}_\text{child} \\
8: & \text{end for}
\end{align*}
\]

refer, hereafter, to this set as the data dependencies for \(\hat{y}_i\), denoted by \(\text{dep}_i\). Once the data dependencies are computed in line 4, a task is submitted in line 5 that overwrites \(\hat{y}_i\) with the appropriate part of \(Pb\) added to the contributions coming from the vectors in \(\text{dep}_i\). On the other hand, in line 26, the submitted task modifies \(\hat{y}_i\) by gathering the contributions coming from the vectors in \(\text{dep}_i\) obtained in line 25.

We show how \(\text{dep}_i\) is computed in Algorithm 4. As in lines 4 and 25 of Listing 1, the initial parameters are the row indices of \(\hat{y}_i\) and the node that owns it. This algorithm creates \(\text{dep}_i\) by visiting the children of the node in line 3. For each child node of \(\text{node}\), the part of the local vector that shares at least one row index with \(\hat{y}_i\) is considered as a data dependency. Note that, in practice, we compute this list once before the solve, and we use the list during the forward and backward substitutions. This saves computations when the solve phase is done multiple times.

The submission of the tasks for the backward substitution is similar to the forward substitution, except for the computations of the data dependencies, where the role of child nodes is replaced by the ancestor nodes.

4.2. Management of the tasks

Our implementation uses the runtime system OpenMP, Release 4.5. This release offers a runtime system that schedules the execution of tasks with dependencies. There are several ways to describe the dependencies. One simple way is to provide the address of the data that are being accessed through a read, write or read/write mode as in Listing 1. Thus, the dependencies have to be known at compilation time by OpenMP. However, the set \(\text{dep}\) returned by Algorithm 4 has a variable length and its length cannot be predicted prior to the execution of the code.

To handle a variable length of dependencies in OpenMP, we use a k-ary tree combined with synchronization tasks to control the scheduling of the task. A synchronization task is a task submitted to the runtime system that does nothing, like a no-op instruction, except to release a dependency for further use. The purpose of using a k-ary tree is to reduce the number of tasks from \(N = |\text{dep}|\) to \(c\), an acceptable number of dependencies, given that we have to statically define each possibility in our code. To do this, we define a chunk, that is, a subset of unsatisfied dependencies, equal to \(c\). The k-ary tree is processed as follows. At the first level of the k-ary tree, the dependencies in \(\text{dep}\) are considered to be \textit{unsatisfied} and are split into \([N/c]\) chunks. For each chunk, a synchronization task is submitted to the runtime system. This synchronization task is scheduled when all dependencies in the chunk are \textit{satisfied}. It then releases the first dependency of the chunk for the next level by flagging it as \textit{unsatisfied}. Thus, the k-ary reduction leads to at most \(c\) unsatisfied dependencies, as presented in Figure 5. In this example, the size of the chunk is 5, and \(N = 35\). A white square represents an unsatisfied dependency, and a grey rectangle represents a set of satisfied dependencies. The first dependency of each chunk is still \textit{unsatisfied} between two levels. A task is finally submitted to the runtime system with a list of dependencies composed of the remaining unsatisfied dependencies of \(\text{dep}\).

Along with this k-ary tree, we use a case statement over the number of unsatisfied dependencies in \(\text{dep}\), where each case corresponds to the submission of a (synchronization) task having \(i\) dependencies (\(0 \leq i \leq c\)), where \(c = 10\) by default.

4.3. The pruning strategy

It is worth noting that the matrix sizes in the tree can vary significantly between the nodes of the tree. More specifically, the matrices are generally small near the leaf nodes and get bigger as we traverse the tree toward the root node. As a consequence, there is plenty of inter-level parallelism at the bottom of the tree with limited opportunity for intra-node parallelism. On the other hand, there is much less inter-level parallelism near the top of the tree, where there is much more intra-node parallelism. In our task-based algorithm, this would result in the creation of many small granularity tasks at the bottom of the tree which can be costly for the runtime system to manage compared to the actual workload being performed in these tasks. In addition, inter-level parallelism at the bottom of the tree can greatly exceed the amount of available resources. Therefore, a large number of tasks is generated at this level, increasing the cost for the task scheduling without bringing much benefit in terms of performance. In order to alleviate...
these problems, we employ a tree pruning strategy, largely inspired by the one used in the qr_mumps solver and described in the work of Buttari (2012). This allows us to dramatically reduce the number of small granularity tasks in the DAG. This pruning strategy aims at identifying a set of subtrees which are processed as a single task meaning that each subtree is processed in serial mode.

In more detail, the pruning algorithm performs a top-down traversal of the tree and selects a sufficiently large set of independent nodes in the tree so that the computational weight of the subtrees rooted at each node in this set is balanced. In practice, we force the number of subtrees to be equal to at least twice the number of resources, because we want to generate enough parallelism to compensate for potential load imbalance during the execution of the subtree within the whole DAG. One potential source of imbalance is that the metric used for estimating the computational weight for each subtree, the number of floating-point operations, might not accurately reflect the execution time.

This pruning strategy allows us to dramatically reduce the number of tasks submitted to the runtime system as well as the number of dependencies in the DAG. Moreover, processing the subtrees as a single task improves the exploitation of data-locality which increases the kernel efficiency.

5. Experimental results

In order to assess the performance of the solve phase of our SpLLT code, we ran tests on a set of 37 symmetric definite positive matrices, presented in Table 1. These matrices come from the Suitesparse matrix collection Davis and Hu (2011) and are those used in the work of Duff et al. (2018) and Hogg et al. (2010). They correspond to a wide range of different applications including computational

Table 1. Test matrices and their characteristics without node amalgamation.a

| # | n (10^3) | nz (A) (10^6) | nz (L) (10^6) | Flops (10^9) | Application/description |
|---|----------|---------------|---------------|-------------|------------------------|
| 1 | Schmid/thermal2 | 1228 | 4.9 | 51.6 | 14.6 | Unstructured thermal FEM |
| 2 | Rothberg/gearbox | 154 | 4.6 | 37.1 | 20.6 | Aircraft flap actuator |
| 3 | DNVS/m_t1 | 97.6 | 4.9 | 34.2 | 21.9 | Tubular joint |
| 4 | Boeing/pwtk | 218 | 5.9 | 48.6 | 22.4 | Pressurized wind tunnel |
| 5 | Chen/plustk13 | 94.9 | 3.4 | 30.4 | 25.9 | Machine element |
| 6 | GHS_psdef/crankseg_1 | 52.8 | 5.3 | 33.4 | 32.3 | Linear static analysis |
| 7 | Rothberg/cfd2 | 123 | 1.6 | 38.3 | 32.7 | CFD pressure matrix |
| 8 | DNVS/thread | 29.7 | 2.2 | 24.1 | 34.9 | Threaded connector |
| 9 | DNVS/shipsec8 | 115 | 3.4 | 35.9 | 38.1 | Ship section |
| 10 | DNVS/shipsec1 | 141 | 4.0 | 39.4 | 38.1 | Ship section |
| 11 | GHS_psdef/crankseg_2 | 63.8 | 7.1 | 43.8 | 46.7 | Linear static analysis |
| 12 | DNVS/fcondp2 | 202 | 5.7 | 52.0 | 48.2 | Oil production platform |
| 13 | Shenk_AFE/af_shell3 | 505 | 9.0 | 93.6 | 52.2 | Sheet metal forming |
| 14 | DNVS/troll | 214 | 6.1 | 64.2 | 55.9 | Structural analysis |
| 15 | AMD/G3_circuit | 1586 | 4.6 | 97.8 | 57.0 | Circuit simulation |
| 16 | GHS_psdef/bmwcrca_1 | 149 | 5.4 | 69.8 | 60.8 | Automotive crankshaft |
| 17 | DNVS/halfb | 225 | 6.3 | 65.9 | 70.4 | Half-breadth barge |
| 18 | Um/2cubes_sphere | 102 | 0.9 | 45.0 | 74.9 | Electromagnetics |
| 19 | GHS_psdef/fdoor | 952 | 23.7 | 144.6 | 78.3 | Large door |
| 20 | DNVS/ship_003 | 122 | 4.1 | 60.2 | 81.0 | Ship structure |
| 21 | DNVS/fullb | 199 | 6.0 | 74.5 | 100.2 | Full-breadth barge |
| 22 | GHS_psdef/inline_1 | 504 | 18.7 | 172.9 | 144.4 | Inline skater |
| 23 | Chen/plustk14 | 152 | 7.5 | 106.8 | 146.4 | Tall building |
| 24 | GHS_psdef/fapache2 | 715 | 2.8 | 134.7 | 174.3 | 3D structural problem |
| 25 | Koutsovasilis/F1 | 344 | 13.6 | 176.5 | 218.8 | AUDI engine crankshaft |
| 26 | Oberwolfach/boneS10 | 915 | 28.2 | 278.0 | 281.6 | Bone micro-FEM |
| 27 | ND/nd12k | 36.0 | 7.1 | 116.5 | 505.0 | 3D mesh problem |
| 28 | ND/nd24k | 72.0 | 14.4 | 321.6 | 2054.4 | 3D mesh problem |
| 29 | Janna/Flan_1565 | 1565 | 59.5 | 1477.9 | 3859.8 | 3D mechanical problem |
| 30 | Oberwolfach/bone010 | 987 | 36.3 | 1076.4 | 3876.2 | Bone micro-FEM |
| 31 | Janna/Stocf_1465 | 1465 | 11.2 | 1126.1 | 4386.6 | Underground aquifer |
| 32 | GHS_psdef/audikw_1 | 944 | 39.3 | 1242.3 | 5804.1 | Automotive crankshaft |
| 33 | Janna/Fault_639 | 639 | 14.6 | 1144.7 | 8283.9 | Gas reservoir |
| 34 | Janna/Hook_1498 | 1498 | 31.2 | 1532.9 | 8891.3 | Steel hook |
| 35 | Janna/Emilia_923 | 923 | 21.0 | 1729.9 | 13661.1 | Gas reservoir |
| 36 | Janna/Geo_1438 | 1438 | 32.3 | 2467.4 | 18058.1 | Underground deformation |
| 37 | Janna/Serena | 1391 | 33.0 | 2761.7 | 30489.9 | Gas reservoir |

a n is the matrix order, nz (A) represents the number of entries in the matrix A, nz (L) represents the number of entries in the factor L and Flops corresponds to the operation count for the matrix factorization.
fluid dynamics (CFD), circuit simulation, finite element modelling (FEM) and gas reservoir modelling. The dimensions of the matrices range from $36 \times 10^3$ to $1.5 \times 10^6$.

We focus our experiments on parallel efficiency both when $\text{nrhs}$ increases and when the number of workers ($\text{nworker}$) increases. We also compare the performance of the SpLLT solve phase with the solve phase of other sparse direct solvers. Note that in the following tables, fastest times are in bold.

We perform the tests on a multicore machine which has two Intel Xeon E5-2695, v3 CPUs, that is 2 NUMA nodes composed of 14 cores each, with a total 128 GB of memory. Each core has a theoretical peak of 36.8 GFlop/s for a frequency of 2.3 GHz, so the peak of the multicore node is 1.03 TFlop/s in double precision. The code is compiled with GNU 6.2 and MKL 17.0.2 and linked to Metis 5.1.0 and SPRAL (https://github.com/ralna/spral/). In most cases, we display results on a representative subset of matrices. We have done preliminary tests to study the reproducibility of the execution times of the machine by solving equation (1) 10 times and increasing $\text{nworker}$. Figure 6 shows the range in the times for the solve phase on four of our test problems by horizontal bars. We note that this range can be noticeable when there are few workers, but this variability decreases markedly when $\text{nworker}$ increases.

We study the impact of pruning in Section 5.1, of synchronization tasks in Section 5.2 and of block size in Section 5.3. In Sections 5.4 and 5.5, we then compare the solve phase of SpLLT with Intel MKL PARDISO 17.0.2 (Schenk and Gärtner, 2002) and PaStiX (Hénon et al., 2002) by increasing $\text{nrhs}$ in Section 5.4 and $\text{nworker}$ in Section 5.5.

### 5.1. Impact of pruning on the performance

In this section, we illustrate the impact of using the tree pruning algorithm discussed in Section 4.3 on the solve times. For this, we performed the solve on a subset of the matrices given in Table 1 using one right-hand side and for a number of workers ranging between 1 and 28. Note that for this experiment, we selected the matrices which perform a high number of floating-point operations during the factorization. Our results, given in Table 2, include the solve times, with and without pruning, as well as the number of subtrees selected by the pruning algorithm when using 1, 2, 4, 14 and 28 workers.

![Figure 6. Reproducibility of execution times.](image)

#### Table 2. The solve time (s) and number of subtrees when pruning is either enabled or disabled.

| Matrices | #Subtree | #Worker | 1  | 2  | 4  | 14 | 28 |
|----------|----------|---------|----|----|----|----|----|
| boneS10  | #Subtree | 1       | 1.26 | 0.80 | 0.55 | 0.32 | 0.60 |
|          | No pruning | 1.14 | 0.79 | 0.38 | 0.15 | 0.10 |
| StocF-1465 | #Subtree | 29,105 | 3.03 | 2.54 | 1.46 | 0.82 | 1.51 |
|          | No pruning | 2.83 | 2.41 | 1.27 | 0.66 | 0.75 |
| Fault_639 | #Subtree | 1       | 4.02 | 2.23 | 1.07 | 0.43 | 0.55 |
|          | No pruning | 2.48 | 2.05 | 0.99 | 0.40 | 0.33 |
| Hook_1498 | #Subtree | 1       | 5.01 | 3.62 | 1.87 | 0.96 | 1.39 |
|          | No pruning | 4.92 | 3.43 | 1.71 | 0.58 | 0.43 |
| Emilia_923 | #Subtree | 1       | 6.42 | 4.88 | 2.72 | 1.12 | 1.35 |
|          | No pruning | 6.13 | 4.56 | 2.41 | 0.96 | 0.76 |
It should be noted that for all tested matrices, the best solve times are obtained when the tree pruning algorithm is enabled. Moreover, the impact of pruning is particularly high when nworker is large. For instance, enabling the tree pruning for boneS10 when using 2 workers is insignificant whereas the solve time is divided by a factor of 6 for this matrix when enabling tree pruning on 28 workers. Another notable effect of using tree pruning is that, although the solve times between 14 workers and 28 workers tend to increase without tree pruning, they keep decreasing if tree pruning is activated. This shows the benefits of using the pruning strategy to improve the scalability of our algorithm on a large number of resources.

As mentioned in Section 4.3, the tree pruning strategy improves the performance mainly for the following two reasons: firstly, it reduces the number of tasks in the DAG and more particularly small granularity tasks, and secondly, it increases kernel efficiency by improving the exploitation of data locality. This can be seen when looking at the data in the first column of Table 2. When using only one worker and disabling the pruning, all the tasks in the DAG are generated, submitted to the runtime system and processed by a single thread. On the other hand, when the pruning is enabled, the number of subtrees is equal to one which means that only one task, responsible for processing the whole tree, is generated. In this case, the cost for the task

As mentioned in Section 4.3, the tree pruning strategy improves the performance mainly for the following two reasons: firstly, it reduces the number of tasks in the DAG and more particularly small granularity tasks, and secondly, it increases kernel efficiency by improving the exploitation of data locality. This can be seen when looking at the data in the first column of Table 2. When using only one worker and disabling the pruning, all the tasks in the DAG are generated, submitted to the runtime system and processed by a single thread. On the other hand, when the pruning is enabled, the number of subtrees is equal to one which means that only one task, responsible for processing the whole tree, is generated. In this case, the cost for the task

### Table 3. Time and number of synchronization tasks for our SpLLT solve with respect to the chunk size.\(^a\)

| Matrix   | Pruning | nrhs | 2     | 3     | 4     | 5     | 10    |
|----------|---------|------|-------|-------|-------|-------|-------|
| boneS10  | #Synchro| 1    | 0.79  | 0.60  | 0.65  | 0.58  | 0.55  |
|          |         | 64   | 0.59  | 0.59  | 0.59  | 0.59  | 0.58  |
|          | Yes     | 1    | 0.12  | 0.11  | 0.11  | 0.11  | 0.11  |
|          |         | 64   | 0.45  | 0.44  | 0.49  | 0.46  | 0.48  |
| StocF-1465 | #Synchro | 1    | 2.21  | 1.70  | 1.89  | 1.62  | 1.57  |
|          |         | 64   | 1.97  | 2.01  | 1.87  | 2.03  | 2.07  |
|          | #Synchro | 1    | 0.72  | 0.74  | 0.73  | 0.76  | 0.72  |
|          |         | 64   | 1.78  | 1.71  | 1.77  | 1.74  | 1.68  |
| Fault_639 | #Synchro | 1    | 0.94  | 0.70  | 0.59  | 0.55  | 0.55  |
|          |         | 64   | 0.15  | 0.16  | 0.19  | 0.14  | 0.18  |
|          | #Synchro | 1    | 0.34  | 0.33  | 0.34  | 0.33  | 0.32  |
|          |         | 64   | 1.07  | 1.12  | 1.03  | 1.06  | 1.04  |
| Hook_1498 | #Synchro | 1    | 2.21  | 1.54  | 1.56  | 1.40  | 1.28  |
|          |         | 64   | 1.97  | 1.90  | 1.89  | 1.92  | 1.89  |
|          | #Synchro | 1    | 0.45  | 0.46  | 0.44  | 0.44  | 0.43  |
|          |         | 64   | 1.60  | 1.69  | 1.66  | 1.62  | 1.70  |
| Emilia_923 | #Synchro | 1    | 1.49  | 1.00  | 0.84  | 0.73  | 0.78  |
|          |         | 64   | 1.80  | 1.65  | 1.58  | 1.72  | 1.71  |
|          | #Synchro | 1    | 0.46  | 0.46  | 0.44  | 0.46  | 0.45  |
|          |         | 64   | 1.66  | 1.57  | 1.43  | 1.67  | 1.42  |
| Serena   | #Synchro | 1    | 2.63  | 1.65  | 1.31  | 1.32  | 1.41  |
|          |         | 64   | 2.70  | 2.69  | 2.72  | 2.67  | 2.72  |
|          | #Synchro | 1    | 0.78  | 0.76  | 0.74  | 0.72  | 0.75  |
|          |         | 64   | 2.34  | 2.36  | 2.47  | 2.41  | 2.40  |

*nrhs: the number of right-hand sides; nworker: the number of workers.

*\(nworker = 28\).*
management is negligible and, as we can see in Table 2, the solve times are always reduced.

5.2. Impact of the number of synchronization tasks on the performance

As discussed in Section 4.2, tasks have a variable number of dependencies that we handle through a k-ary tree. In this section, we give details of the effect of the synchronization tasks on the solve time. We consider the case of 1 right-hand side and 64 right-hand sides. Table 3 presents the results on the same subset of matrices as in Table 2. We first focus on the case of one right-hand side where the pruning is disabled. We observe that the time to solve is the highest when $c$ is equal to two. Moreover, the number of synchronization tasks is divided by at least a factor of 3 when increasing the chunk size from 2 to 3. After some point, the number of synchronization tasks is low enough that reducing it further does not noticeably improve the execution time. When pruning is used, we observe that the number of synchronization tasks is divided by at least a factor of 2 for a chunk size of 2. This means that the sensitivity of the times on the chunk size is much reduced when pruning is used. Likewise, when we solve for 64 right-hand sides, the extra work required means that the number of synchronization tasks has little impact on the time, with or without pruning. The arithmetic intensity in the kernels is much larger than the overhead of scheduling and executing synchronization tasks.

However, it is possible that for larger workloads than our test matrices, a small chunk size would penalize the performance because of a high number of synchronization tasks. For this reason, we use a chunk size of 10 in our solver that we consider to be large enough to avoid noticeable impact on the results on the tested matrices and potentially on bigger problems as well.

5.3. Impact of the block size on the performance

In this section, we study the impact of the block size on the time to solve the system and show our results in Table 4. We consider matrices that require a large number of flops during the factorization so that the block size should be greater than 256. We set nworker to 28 and nrhs to 128. For all these matrices, the optimal block size for the solve phase is lower than the optimal block size for the factorization. This leads us to conclude that we should use a different block size, depending on nrhs and the application. For example, in a code like preconditioned CG, the overall performance may depend on the time spent in applying the preconditioner. In that case, an efficient factorization is not as important as an optimal solve. We use a default block size of 256 in the following experiments to obtain a fair comparison with other solvers.

5.4. Strong scaling on nrhs

In this section, we focus on the effect of nrhs on the time to solve the system. We fix nworker to 28 and the block size to 256, and we vary nrhs from 1 to 128 in powers of 2.

We first compare, in Tables 5 and 6, SpLLT with PAR-DISO, PaStiX and HSL_MA87 when solving for one or two right-hand sides. These results show that PARDISO is the slowest for all the matrices in Table 1. SpLLT is the
fastest to solve the system, up to 8 times faster than PAR- DISO for one right-hand side and up to over a factor of 10 on two right-hand sides. There are only five problems in Table 1 for which SpLLT is not the fastest of all four codes for both one right-hand side and two right-hand sides. In two cases, the difference from the fastest code is marginal. The only matrices for which SpLLT is significantly slower are the two matrices nd12k and nd24k and the matrix StocF-1465. In the case of the first two, the assembly tree has long chains that are not combined by pruning so the runtime overhead for many small tasks causes the poor performance. If we force more node amalgamation in the analyse phase, the number of long chains decreases significantly, pruning is more effective, and we are much more competitive. The StocF-1465 matrix is very reducible with many $1 \times 1$ blocks. Again our pruning has little effect and we are penalized by the number of small tasks. We recommend that reducibility is respected in the solution process, and elimination techniques are only used on irreducible blocks. We see that on one of the matrices, Emilia_923, PaStiX is faster than SpLLT for one right-hand side ($43.70 \times 10^{-2}$ s vs. $44.70 \times 10^{-2}$ s) and SpLLT is faster than PaStiX for two right-hand sides ($47.70 \times 10^{-2}$ s vs. $58.6 \times 10^{-2}$ s). The solve time for SpLLT on two systems is always lower than solving one system twice.

We investigate this for more right-hand sides. In Tables 7 and 8, we give the time for SpLLT, PARDISO, PaStix and HSL_MA87, for all matrices of Table 1 when solving for 16 and 128 right-hand sides. When nworkers increases, PaStix and HSL_MA87 become less competitive than SpLLT. We show the ratio of the time for solving several right-hand sides to the time to solve one right-hand side in Table 9. These

| Table 5. Comparison of SpLLT, PARDISO, HSL_MA87 and PaStiX for one right-hand side. | Table 6. Comparison of SpLLT, PARDISO, HSL_MA87 and PaStiX for two right-hand sides. |
|----------------------------------|----------------------------------|
| Matrices                        | Matrices                        |
| SpLLT                           | PARDISO                         | MA87                            | PaStix |
| thermal2                        | 5.83                            | 21.70                           | 16.20   | 13.20 |
| gearbox                         | 1.84                            | 10.30                           | 2.79    | 3.38  |
| m_t1                            | 1.64                            | 7.93                            | 2.17    | 2.20  |
| pwtk                            | 2.48                            | 12.00                           | 4.01    | 3.37  |
| pkustk13                        | 1.90                            | 6.56                            | 2.13    | 2.43  |
| crankseg_1                      | 1.61                            | 6.92                            | 2.02    | 1.90  |
| cfd2                            | 2.01                            | 8.66                            | 2.85    | 3.12  |
| thread                          | 2.30                            | 6.27                            | 1.63    | 2.68  |
| shipsc8                         | 2.37                            | 8.31                            | 2.69    | 3.07  |
| shipsc1                         | 2.43                            | 8.93                            | 2.69    | 3.07  |
| crankseg_2                      | 2.08                            | 8.67                            | 2.36    | 2.51  |
| fcondp2                         | 2.86                            | 12.80                           | 3.93    | 3.87  |
| af_shell3                       | 4.25                            | 27.30                           | 8.32    | 8.15  |
| troll                           | 3.10                            | 18.60                           | 4.03    | 3.75  |
| G3_circuit                      | 8.89                            | 34.90                           | 25.70   | 17.40 |
| bmwcrca_l                       | 2.56                            | 15.10                           | 3.27    | 3.78  |
| halfb                           | 3.29                            | 15.90                           | 5.26    | 4.31  |
| cubes_sphere                    | 2.63                            | 8.53                            | 3.03    | 2.83  |
| ldoor                           | 6.92                            | 42.10                           | 14.60   | 13.30 |
| ship_003                        | 3.18                            | 10.60                           | 3.52    | 3.29  |
| fullb                           | 3.91                            | 16.40                           | 4.53    | 4.40  |
| inline_1                        | 6.35                            | 40.90                           | 9.72    | 8.62  |
| pkustk14                        | 4.83                            | 23.30                           | 5.67    | 4.94  |
| apache2                         | 7.99                            | 28.70                           | 13.40   | 12.90 |
| F1                              | 5.92                            | 33.40                           | 9.72    | 8.62  |
| boneS10                         | 10.10                           | 61.30                           | 16.80   | 17.10 |
| nd12k                           | 12.90                           | 19.40                           | 4.55    | 5.12  |
| nd24k                           | 22.40                           | 54.60                           | 10.90   | 10.70 |
| Flan_1565                       | 38.90                           | 290.00                          | 54.70   | 46.90 |
| bone010                         | 28.70                           | 209.00                          | 33.80   | 50.10 |
| StocF-1465                      | 76.60                           | 223.00                          | 45.00   | 51.80 |
| audiwk_l                        | 35.50                           | 215.00                          | 36.70   | 47.80 |
| Fault_639                       | 33.10                           | 226.00                          | 33.30   | 45.80 |
| Hook_1498                       | 43.60                           | 298.00                          | 59.00   | 91.70 |
| Emilia_923                      | 44.70                           | 358.00                          | 50.80   | 43.70 |
| Geo_1438                        | 64.70                           | 481.00                          | 76.60   | 93.90 |
| Serena                          | 75.80                           | 485.00                          | 86.30   | 130.00|

nworker: the number of workers. *Times in 10^{-2} s. nworker is 28.
results show that the power of level 3 BLAS means that there is often very little overhead for solving two right-hand sides over one and the extra cost when solving for 16 or 128 right-hand sides is remarkably low.

Since HSL_MA87 does not scale with nrhs, in the following, we compare the time to solve for SpLLT with PARDISO and PaStiX, and we display the ratio of times in Figure 7.

Figure 8 presents two representative problems that illustrate the behaviour of each solver. For all nrhs, SpLLT is the fastest solver sometimes by a considerable amount. PaStiX scales much worse than SpLLT as nrhs increases. The scalability of PARDISO can vary considerably (see Serena). Since the source code for PARDISO is not distributed, we do not know why this is the case. We have similarly erratic scalability for several other matrices in our test set. In Figure 8(a), SpLLT is eight times faster...
than PARDISO for two right-hand sides, and these two curves increase almost similarly when nrhs increases. On the other hand, the time to solve using PaStiX is close to SpLLT for 2 right-hand sides but is 5 times slower on 128 right-hand sides. In general, SpLLT scales better than the other codes.

5.5. Strong scaling on nworker

In this section, we study the impact of nworker on the time to solve a system with 1 right-hand side and 64 right-hand sides. nworker increases from 1 to 28, and the block size is 256. We first consider one right-hand side and present some results in Figure 9. We use the same two matrices as in Figure 8. The experiments show that for a small number of workers, the time to solve using SpLLT is greater than both other solvers. When nworker increases, the solid curve that represents SpLLT becomes closer to the other curves. For 28 workers, SpLLT is the fastest solver.

Figure 10 shows the impact of increasing nworker when there are 64 right-hand sides. SpLLT is more competitive than for a small nrhs. In fact, SpLLT outperforms PARDISO for all numbers of workers and is faster than PaStiX when nworker increases. For more than nine workers, these two representative problems show that SpLLT is clearly the fastest solver.

Figure 7. Comparison of the times to solve the system with multiple right-hand sides using PARDISO and PaStiX, relative to SpLLT. nworker is set to 28, and nrhs is equal to (a) 1, (b) 2, (c) 16 and (d) 128. nrhs: the number of right-hand sides; nworker: the number of workers.

Figure 8. Comparison of the time to solve the system with multiple right-hand sides using SpLLT, PARDISO and PaStiX. nworker is set to 28, and nrhs increases from 2 to 128 in powers of 2. (a) boneS10 and (b) Serena. nrhs: the number of right-hand sides; nworker: the number of workers.
5.6. Application case: Enlarged CG solver

In this section, we present some results obtained with the enlarged CG solver (ECG) (Grigori and Tissot, 2017). Our runs in this section are on a different machine to our earlier results. We use the Kebnekaise system at Umeå in Sweden (https://www.hpc2n.umu.se/resources/hardware/kebnekaise). Each compute node contains 28 Intel Xeon E5-2690v4 cores organized into 2 NUMA islands with 14 cores in each. The nodes are connected with an FDR Infiniband Network. The total amount of RAM per compute node is 128 GB. ECG and SpLLT are compiled with Intel 18.0.1 and linked to Metis 5.1.0.

The ECG solver is a preconditioned CG solver that augments the number of working vectors to reduce the number of iterations as well as to obtain better parallelism and to reduce the amount of communication. ECG uses a block Jacobi preconditioner. Although each diagonal block is factorized only once, the solve of each associated local system is performed at each iteration of ECG. Table 10 presents the time to solve the global system with ECG using PARDISO or SpLLT on a subset of matrices from Table 1. We show results from 16 MPI processes to 256 MPI processes. We see little difference in solution times for the shipsec1 problem but, on the two other problems that are 10 times larger, we observe that replacing the PARDISO Solver with the SpLLT solver leads to a speed-up over a factor of 2. In all cases, solving the system with 16 MPI processes using SpLLT is faster than using PARDISO with 32 MPI processes.

Table 10. Time to solution using PARDISO and SpLLT.

| Problem     | #MPI | PARDISO |      | SpLLT |      |
|-------------|------|---------|------|-------|------|
|             |      | t (s)   | #Iter| t (s) | #Iter|
| shipsec1    | 16   | 3.58    | 204  | 2.66  | 204  |
|             | 32   | 2.90    | 290  | 2.28  | 290  |
|             | 64   | 1.96    | 361  | 1.59  | 362  |
|             | 128  | 1.21    | 447  | 1.16  | 447  |
|             | 256  | 1.19    | 544  | 1.01  | 544  |
| Flan_1565   | 16   | 57.82   | 141  | 23.18 | 141  |
|             | 32   | 32.92   | 177  | 14.94 | 177  |
|             | 64   | 20.15   | 216  | 9.77  | 216  |
|             | 128  | 11.35   | 270  | 6.53  | 270  |
|             | 256  | 6.70    | 325  | 5.50  | 325  |
| Hook_1498   | 16   | 32.10   | 87   | 12.68 | 87   |
|             | 32   | 16.04   | 101  | 7.84  | 101  |
|             | 64   | 9.85    | 128  | 5.53  | 128  |
|             | 128  | 6.05    | 154  | 4.00  | 154  |
|             | 256  | 3.46    | 183  | 2.55  | 183  |

ECG: enlarged conjugate gradient solver.

*ECG is set with a tolerance of 10^{-5}, an enlarge factor of 12, and SpLLT has a block size of 256 with 14 workers.
6. Conclusions

We have designed, implemented and tested new routines for the forward and backward substitution steps in the parallel solution of sparse positive definite systems using a Cholesky factorization. We have given details of how our design exploits parallelism while keeping data movement low. We have developed a code that we have tested on a multicore node and have targeted and achieved good scalability both with respect to the number of cores and to nrhs. We have compared our code to other state-of-the-art codes and shown that it is usually far superior sometimes outperforming the other codes by a factor of over 10. We have shown that our code is strongly scalable both for cores and right-hand sides.

Finally, we have tested our code within a real application and shown big gains over the previous version that used another solver. We are continuing our work with the authors of ECG at Inria and have seen even bigger gains on markedly larger problems than those used in this article. For example, on a matrix from a diffusion problem of order nearly 5 million with over 34 million entries, we reduce the solution time by a factor of nearly 3 on 16 MPI processes and by a factor of nearly 2 on 256 processes.

Acknowledgements

The authors would like to thank Tyrone Rees of RAL for his comments on an early draft of this article and the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support for the work in Section 5.6.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the NLAfET Project funded by the European Union’s Horizon 2020 Research and Innovation Programme under Grant Agreement 671633.

ORCID iD

Sébastien Cayrols https://orcid.org/0000-0003-3740-8985

References

Amestoy P, Davis T and Duff I (1996) An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications* 17(4): 886–905. DOI: 10.1137/S0895479894278952.

Augonnet C, Thibault S, Namyst R, et al. (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23: 187–198. DOI:10.1002/cpe.1631. Available at: http://hal.inria.fr/inria-00550877 (accessed 12 November 2019).

Blackford LS, Demmel J, Dongarra J, et al. (2002) An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software* 28(2): 135–151.

Bosilca G, Bouteiller A, Danalis A, et al. (2013) PaRSEC: exploiting heterogeneity to enhance scalability. *Computing in Science and Engineering* 15(6): 36–45. DOI: 10.1109/MCSE.2013.98.

Buttari A (2012) Fine granularity sparse QR factorization for multicore based systems. In: Jonasson K (ed.) *Applied Parallel and Scientific Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 226–236. ISBN 978-3-642-28145-7.

Davis TA and Hu Y (2011) The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38(1): 1:1–1:25. DOI: 10.1145/2049662.2049663.

Duff I and Lopez F (2018) Experiments with sparse Cholesky using a parametrized task graph implementation. In: Wyrzykowski R, Dongarra J, Deelman E and Karczewski K (eds) *Parallel Processing and Applied Mathematics*. Cham: Springer International Publishing, pp. 197–206. ISBN 978-3-319-78024-5.

Duff IS, Erisman AM and Reid JK (2017) *Direct Methods for Sparse Matrices*, 2nd ed. Oxford, UK: Oxford University Press. ISBN 978-0-19-850838-0 (hardcover).

Duff IS, Hogg J and Lopez F (2018) Experiments with sparse Cholesky using a sequential task-flow implementation. *Numerical Algebra, Control and Optimization* 8: 235–258.

Grigori L and Tissot O (2017) Reducing the Communication and Computational Costs of Enlarged Krylov Subspaces Conjugate Gradient: Research Report RR-9023. Paris: Inria. Available at: https://hal.inria.fr/hal-01451199v2/document (accessed 12 November 2019).

Hénon P, Ramet P and Roman J (2002) PaStiX: a high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing* 28(2): 301–321. Available at: https://hal.inria.fr/inria-00346017 (accessed 12 November 2019).

Hogg J, Reid J and Scott J (2010) Design of a multicore sparse Cholesky factorization using DAGs. *SIAM Journal on Scientific Computing* 32(6): 3627–3649. DOI: 10.1137/090757216.

Karypis G and Kumar V (1998) Multilevel k-way partitioning scheme for irregular graphs. *SIAM Journal on Scientific Computing* 48: 96–129.

Schenk O and Gärtner K (2002) Two-level dynamic scheduling in PAR-DISO: improved scalability on shared memory multiprocessor systems. *Parallel Computing* 28(2): 187–197.

Author biographies

Sébastien Cayrols earned his PhD degree in Applied Mathematics from Sorbonne University in Paris, France. Under the supervision of his advisor, Laura Grigori, he worked on communication avoidance for incomplete factorizations and low-rank approximations. He was a visiting scholar of
Aspire Lab at the University of California, Berkeley, for 5 months. He then completed a postdoctoral fellowship at Rutherford Appleton Laboratory in Oxfordshire, UK, under the supervision of Prof. Iain S Duff, and was part of the Parallel Numerical Linear Algebra for Extreme Scale Systems (NLAFET) effort – a Horizon 2020 project. He worked on sparse direct solvers as well as least square problems. He is now a member of the Innovative Computing Laboratory’s Linear Algebra group at the University of Tennessee, Knoxville. His personal research interests include sparse numerical linear algebra, high-performance computing and communication-avoiding algorithms.

Iain S Duff is an STFC Honorary Scientist in the Scientific Computing Department at the STFC Rutherford Appleton Laboratory in Oxfordshire, UK. He is also the Scientific Advisor for the Parallel Algorithms Group at CERFACS in Toulouse and is a Visiting Professor of Mathematics at the University of Strathclyde. After completing his D Phil at Oxford, he was a Harkness Fellow in the United States visiting Stony Brook and Stanford. He then spent 2 years lecturing at the University of Newcastle upon Tyne before joining the Harwell Laboratory where he became Group Leader of Numerical Analysis in 1986. In 1990, the Group moved to the Atlas Centre at the Rutherford Appleton Laboratory. He has had several extended visits to Argonne National Laboratory, the Australian National University, the University of Colorado at Boulder, DTU in Lyngby Denmark, the University of Newcastle in NSW, Stanford University and the University of Umeå. He is a life Fellow of the Institute of Mathematics and its Applications and a member of LMS (UK), SIAM (USA) and SMAI (France). He is an IMA representative on the ICIAM Board and is chairman of the IMA Journals Board of Management. He was chairman of the SIAM Board of Trustees for 7 years until December 2011 and was on the Applied Mathematics panel for RAE 2008. He was elected as a Fellow of the Royal Society of Edinburgh in 2006 and was selected as an SIAM Fellow in 2010. His current research interests include numerical linear algebra, sparse matrices, parallel computing, scientific computation and mathematical software. He has written several books and over 200 papers in these areas, more than half of which have appeared in refereed journals. He is recognized by ISI Thomson as a Highly Cited Researcher.

Florent Lopez is a postdoctoral researcher in the Innovative Computing Laboratory at the University of Tennessee. He received a PhD in computer science from Université Paul Sabatier, Toulouse, France, in 2015. After his PhD, he joined the Computational Mathematics Group in the Rutherford Appleton Laboratory, UK, to work in the context of the NLAFET EU project until the beginning of 2019 before joining ICL. His research interests are linear algebra and high-performance computing and more particularly, his work focuses on solving large sparse linear systems of equations on modern architectures such as GPU-accelerated multicore systems.