NLAFET Working Note 20

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

Sébastien Cayrols, Iain Duff and Florent Lopez

October 2018

Document information

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

Sébastien Cayrols[†], Iain Duff[†] and Florent Lopez[†]

ABSTRACT

We describe the parallelization of the solve phase in the sparse Cholesky solver SpLLT [Duff, Hogg, and Lopez. Numerical Algebra, Control and Optimization. Volume 8, 235-237, 2018] when using a sequential task flow (STF) 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 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 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

AMS(MOS) subject classifications: 65F30, 65F50

[†]Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Campus, Oxfordshire, OX11 0QX, UK.

Correspondence to: sebastien.cayrols@stfc.ac.uk

October 10, 2018

Contents

1   Introduction

In this study, we are solving the linear system

$$Ax = b, \tag{1.1}$$

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

$$P^T A P = L L^T, \tag{1.2}$$

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

$$Ly = Pb, \tag{1.3}$$

followed by a *backward substitution* step

$$L^T z = y, \tag{1.4}$$

so that $x = P^T z$ is the solution to the system in equation (1.1). 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 paper we concentrate on the steps in equations (1.3) and (1.4). Most work on sparse solvers concentrates on the factorization in equation (1.2) 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 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 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 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.

The goal of a modern task-based runtime system is to provide an abstraction level such that the low-level hardware details are hidden. There exists different task-based programming models such as *parametrized Task Graph* (PTG), or *sequential Task Flow* (STF). The first explicitly uses the dependencies between tasks whereas the second relies on an explicit data access mode. The sparse Cholesky solver SpLLT [8] has been developed with the STF programming model using the runtime system StarPU [2]. Although there are more overheads to using StarPU than OpenMP [5], SpLLT is very competitive with the OpenMP based state-of-the-art HSL_MA87 code while offering more flexibility and more maintainability because of the runtime system used. In this paper, we consider the solve phase of SpLLT. 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 the number of right-hand sides. 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 lower triangular dense factor $L \in \mathbb{R}^{n \times n}$ into blocks as:

$$
L = \begin{pmatrix} L_{11} & & & \\ L_{21} & L_{22} & & \\ \vdots & & \ddots & \\ L_{k1} & \ldots & & L_{kk} \end{pmatrix},
\tag{2.1}
$$

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 (1.3) that we present in Algorithm 1.

Algorithm 1 forwardSolve(L, b, nb)

Input: $L \in \mathbb{R}^{n \times n}$ partitioned as in equation (2.1),
    b the right-hand side,
    nb the block size
Output: y the solution of the system $Ly = b$
 1: $k = \lceil (n/nb) \rceil$
 2: **for** $i = 1$ **to** k **do**
 3:     Solve $L_{ii} y_i = b_i$
 4:     **for** $j = i + 1$ **to** k **do**
 5:         Update $b_j = b_j - L_{ji} y_i$
 6:     **end for**
 7: **end for**

Algorithm 1 shows that the forward substitution requires two computational kernels. The first kernel at line 3 corresponds to the classical triangular solve, that is referred to

hereafter as the *solve kernel*, denoted by `dtrsv` in the BLAS [3]. We note that the *solve kernel* is different from the abovementioned solve phase and is indeed just part of this phase. The second kernel at line 5 is the update of the right-hand side by performing a matrix-vector product, that is referred to hereafter as the *update kernel*, `dgemv` in BLAS parlance.

From Algorithm 1, we note that the update follows the solution of part of the right-hand side, and triggers the solution of another part of b. That is there is a dependency between the update and the solve. We can generate a directed acyclic graph (DAG) to show the dependencies in Algorithm 1. To illustrate this, we consider the case $k = 4$ and present the associated DAG in Figure 2.1(b).

(a) Partitioned matrix
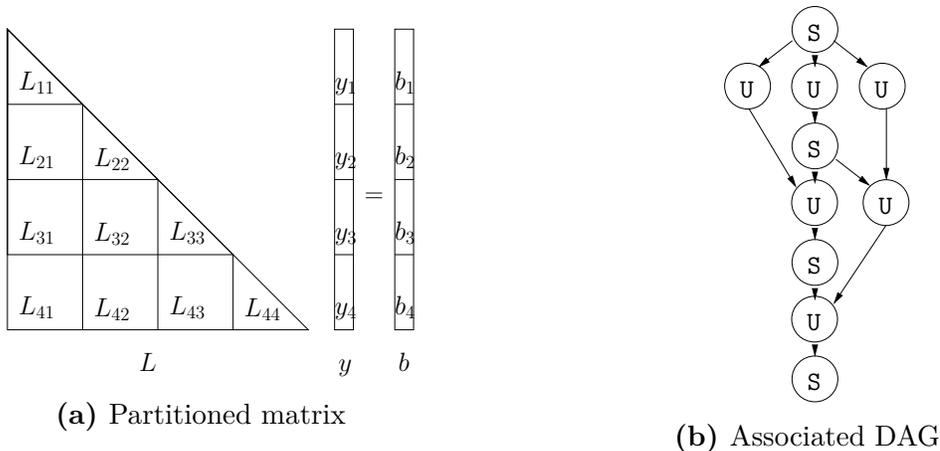
(b) Associated DAG

Figure 2.1: DAG associated with the forward solve on a partitioned L, as presented in Algorithm 1, with $k = 4$. The nodes labelled S correspond to a call to the solve kernel (line 3), and the nodes labelled U correspond to a call to the update kernel (line 5).

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 2.1(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 (1.4). 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. Similarly to the forward substitution, this algorithm requires two computational kernels, the solve kernel and the update kernel, which give the same DAG as in Figure 2.1(b).

3

Algorithm 2 backwardSolve(L, y, nb)

Input: $L \in \mathbb{R}^{n \times n}$ partitioned as in equation (2.1),
    y the vector returned by Algorithm 1,
    nb the block size

Output: x the solution of the system $L^T x = y$, and so of $Ax = b$

1: $k = \lceil (n/nb) \rceil$
2: **for** $i = k$ **to** 1 **do**
3:     Solve $L_{ii}^T x_i = y_i$
4:     **for** $j = i - 1$ **to** 1 **do**
5:         Update $y_j = y_j - L_{ij}^T x_i$
6:     **end for**
7: **end for**

---

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 (1.1) 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 [7] for the details. This step can use an ordering algorithm such as AMD [1] or Metis [12]. 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 (2.1). 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 $\tilde{L}_2$ viz.

$$\tilde{L} = \begin{pmatrix} \tilde{L}_1 \\ \tilde{L}_2 \end{pmatrix}. \tag{3.1}$$

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 3.1 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 2.1. We label the blocks from the leaves to the root. If there are rows in common between blocks blk_7 and blk_{15} 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 inter-node 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.

(a) Partitioned L matrix
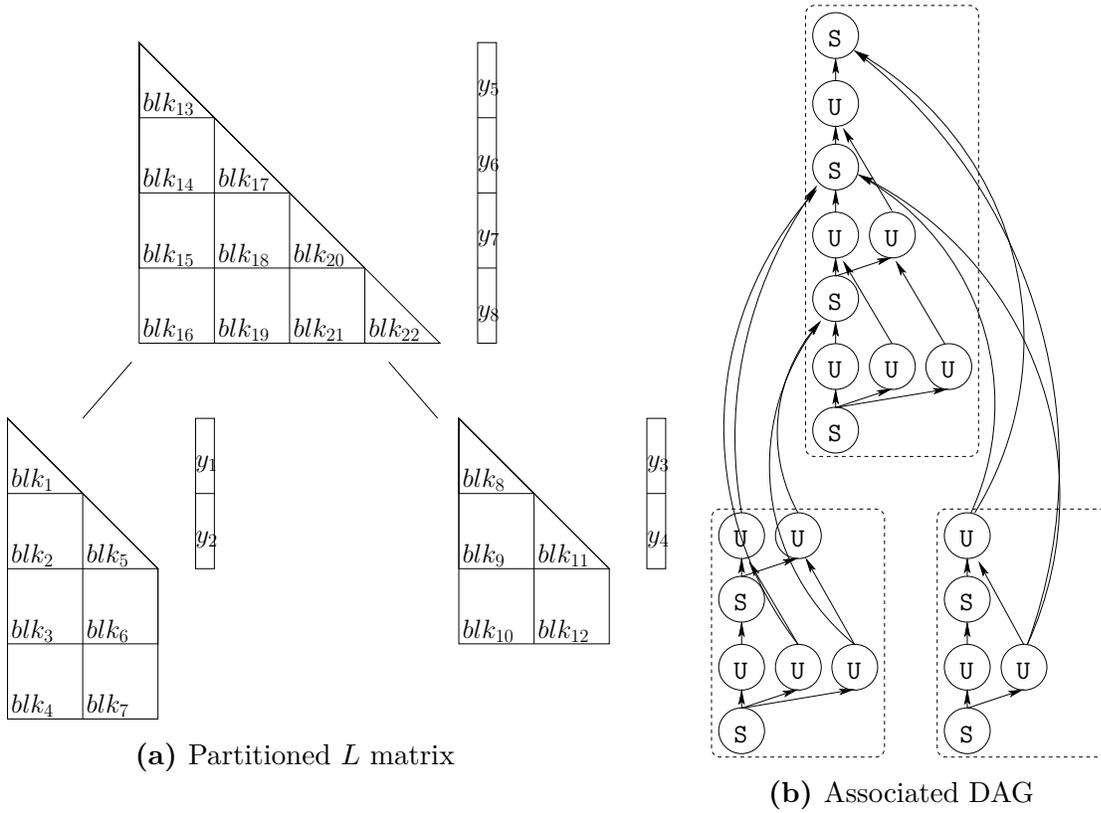
(b) Associated DAG

Figure 3.1: Tree representation of the sparse triangular factor L and its associated DAG for the forward solve. The matrix corresponding to each node is partitioned into blocks of size nb, as in equation (2.1). The nodes labelled S correspond to a call to the solve kernel (line 3), and the nodes labelled U correspond to a call to the update kernel (line 5).

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 $\tilde{L}$ associated with each node of the tree is trapezoidal, except

for the root. Thirdly, $\tilde{L}_2$ 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 $\tilde{L}_1$ 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 (1.3). Thus the part of the vector corresponding to the triangular block $\tilde{L}_1$ 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 $\tilde{L}$ and the movement or equivalence of global and local vectors in Figure 3.2. As mentioned earlier, the part of $\tilde{y}$ corresponding to $\tilde{L}_1$ 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 $\tilde{L}_2$. 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 $\tilde{L}$ using some entries from the child workspace. In Figure 3.2 we show in the dark shaded areas the solution for the fully summed variables that involve the block $\tilde{L}_1$ 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 $\tilde{L}_{3j}\tilde{y}_j$, 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.

Figure 3.2: Data movement required by the computation of $\tilde{L}_{31}\tilde{y}_1$ and $\tilde{L}_{32}\tilde{y}_2$, during the forward solve of a node $\tilde{L}$.

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 3.2 with its associated dense lower trapezoidal matrix $\tilde{L}$. In order to update the solution vector $z = Px$ from equation (1.4) 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 3.3. Then, the update of z is performed by the computation of $(\tilde{L}_{31})^T\tilde{z}_3$ and $(\tilde{L}_{32})^T\tilde{z}_3$. Each triangular solve computes part of Px. That is, the result of the triangular solve is then permuted to obtain the solution vector x.
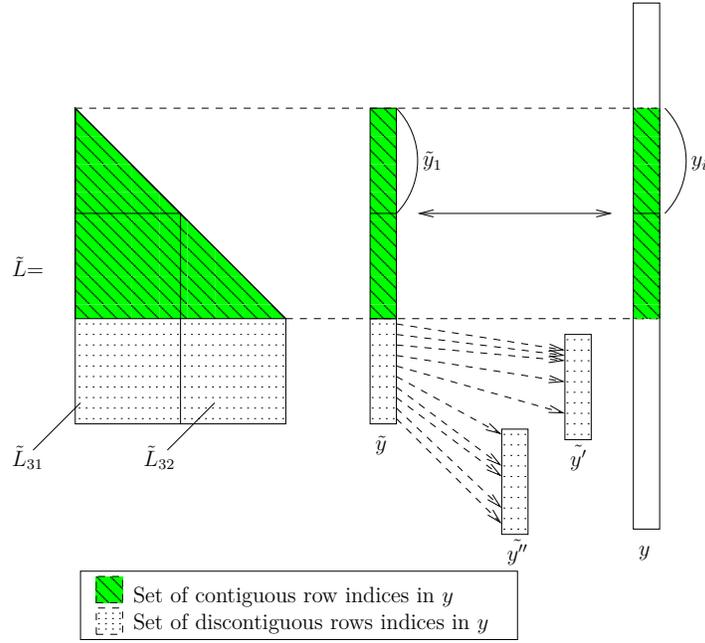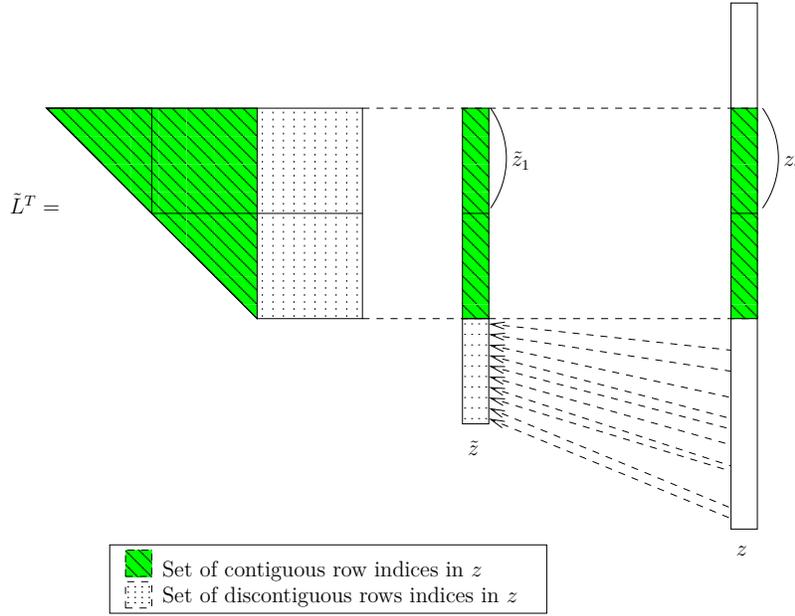
Figure 3.3: Data movement required by the computation of $(\tilde{L}_{31})^T \tilde{z}_3$ and $(\tilde{L}_{32})^T \tilde{z}_3$, during backward solve of a node $\tilde{L}$.

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 3.1, the L factor is now a tree, where the nodes are associated with trapezoidal matrices, and has inter-node dependencies. Secondly, as shown in Figures 3.2 and 3.3, 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 $\tilde{L}$, as written in equation (3.1), and a local vector $\tilde{y}$. We start by forming the part of $\tilde{y}$ associated with $\tilde{L}_1$. That is, we gather the corresponding components of Pb 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.

Algorithm 3 forwardSolve(L, b, nb)

Input: $L \in \mathbb{R}^{n \times n}$ stored as a tree,
$\quad$ b the right-hand side,
$\quad$ nb the block size

Output: y the solution of the system $Ly = b$

1: Let $\mathcal{N}$ be the ordered list of nodes of the tree from the bottom to the top
2: **for** $node \in \mathcal{N}$ **do**
3: $\quad$ Let $\tilde{L}$ be the dense trapezoidal matrix associated with the current node, and let it be partitioned as in equation (3.1)
4: $\quad$ Let $\tilde{L}$ have dimensions $m \times n$, $m \geq n$
5: $\quad$ Let $\tilde{y}$ be the local solution vector associated with the current node
6: $\quad$ $k = \lceil n/nb \rceil$
7: $\quad$ $l = \lceil m/nb \rceil$
8: $\quad$ /* Form the local vector associated with $\tilde{L}_1$ */
9: $\quad$ **for** $i = 1$ **to** k **do**
10: $\quad\quad$ Initialize $\tilde{y}_i$ using the permuted right-hand side and the appropriate components computed by the child nodes
11: $\quad$ **end for**
12: $\quad$ /* Solve using $\tilde{L}_1$ as in Algorithm 1 */
13: $\quad$ **for** $i = 1$ **to** k **do**
14: $\quad\quad$ Solve $\tilde{L}_{ii}\tilde{y}_i = \tilde{y}_i$ so that the result is stored in-place
15: $\quad\quad$ **for** $j = i + 1$ **to** k **do**
16: $\quad\quad\quad$ Update $\tilde{y}_j = \tilde{y}_j - \tilde{L}_{ji}\tilde{y}_i$
17: $\quad\quad$ **end for**
18: $\quad$ **end for**
19: $\quad$ /* Additional loops to process $\tilde{L}_2$ */
20: $\quad$ **for** $j = k + 1$ **to** l **do**
21: $\quad\quad$ Compute $\tilde{y}_j = \tilde{L}_{j1}\tilde{y}_1$
22: $\quad$ **end for**
23: $\quad$ **for** $i = 2$ **to** k **do**
24: $\quad\quad$ **for** $j = i + 1$ **to** l **do**
25: $\quad\quad\quad$ Update $\tilde{y}_j = \tilde{y}_j - \tilde{L}_{ji}\tilde{y}_i$
26: $\quad\quad$ **end for**
27: $\quad$ **end for**
28: $\quad$ /* Gather entries into the local vector associated with $\tilde{L}_2$ */
29: $\quad$ **for** $i = k + 1$ **to** l **do**
30: $\quad\quad$ Gather the appropriate components computed by the child nodes into $\tilde{y}_i$
31: $\quad$ **end for**
32: **end for**

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

In this section, we give details of the parallel implementation of the solve phase of SpLLT. Our task-based algorithms described in Section 3 can use a runtime system to provide an abstraction layer for their implementation. The runtime system also provides maintainability, flexibility and robustness. We have therefore modified both the forward and backward algorithms from Section 3 using a *Sequential Task Flow* (STF) programming model as in the SpLLT factorization. The STF programming model is based on sequential consistency so that it is trivial to implement. It also allows an implicit detection of the dependencies between tasks using the access mode for the data of the task. In the context of runtime systems, a worker is defined as an entity that executes one task at a time, and a task is ready to be executed when its dependencies are satisfied. We use the runtime system of OpenMP that implements the STF model. Based on the representation of the factor L, we first show how to determine the data dependencies of the tasks in the solve phase and then show how the tasks are managed by the OpenMP runtime system.

4.1 Data dependencies of a task

From the DAG as presented in Figure 3.1(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. The submission of the tasks related to the forward substitution is presented in Listing 1. The computational operations within a node in Algorithm 3 are now considered as tasks that are submitted to the runtime system. We provide in addition the access mode to the data in each task to enable the runtime system to detect the dependencies between tasks.

```
1    !Process of node node
2    !Form the local vector associated with L̃₁
3    do i = 1,  k
4       dep_i =  getForwardDependencies(L, node, indices(ỹᵢ))
5       Submit(Initialize, ỹᵢ:W, dep_i:R, Pb:R)
6    end do
7    !Process L̃₁ as in Algorithm 1
8    do i = 1, k
9       Submit(Solve, ỹᵢ:RW,  L̃ᵢᵢ:R)
10      do   j =  i + 1, k
11         Submit(Update, ỹⱼ:RW,  L̃ⱼᵢ:R,  ỹᵢ:R)
12      end do
13   end do
14   !Additional loops to process L̃₂
15   do j = k+ 1, l
16      Submit(Compute,  ỹⱼ:W,  L̃ⱼ₁:R,  ỹ₁:R)
```

```
17        end do
18        do i = 2, k
19          do j = i + 1, l
20            Submit(Update, ỹⱼ:RW, L̃ⱼᵢ:R, ỹᵢ:R)
21          end do
22        end do
23        !Gather the local vector associated with L̃₂
24        do i = k+ 1, l
25          dep_i = getForwardDependencies(L, node, indices(ỹᵢ))
26          Submit(Gather, ỹᵢ:RW, dep_i:R)
27        end do
```

Listing 1: Task-based implementation of Algorithm 3.

The data accessed local to a node, in lines 9, 11, 16 and 20, is similar to the dense case in Algorithm 1 and is independent of the pattern of the matrix. However, the sparse aspect requires gathering entries from a set of local vectors that share rows with the local vector, $\tilde{y}_i$, of the node. We refer, hereafter, to this set as the data dependencies for $\tilde{y}_i$, denoted by dep_i. Once the data dependencies are computed in line 4, a task is submitted in line 5 that overwrites $\tilde{y}_i$ with the appropriate part of Pb added to the contributions coming from the vectors in dep_i. On the other hand, in line 26, the submitted task modifies $\tilde{y}_i$ by gathering the contributions coming from the vectors in dep_i obtained in line 25.

We show how dep_i is computed in Algorithm 4. As in lines 4 and 25 of Listing 1, the initial parameters are the row indices of $\tilde{y}_i$ and the node that owns it. This algorithm creates dep by visiting the children of the node in line 3. For each child node of $node$, the part of the local vector that shares at least one row index with $\tilde{y}_i$ 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.

Algorithm 4 getForwardDependencies(L, $node$, $\mathcal{I}_{blk}$)

Input: $L \in \mathbb{R}^{n \times n}$ stored as a tree,
    $node$ is a node in L,
    $\mathcal{I}$ is a set of row indices in L.
Output: dep is a set of vectors that share indices with $\mathcal{I}$ in L
  1: $dep \leftarrow \emptyset$
  2: Let $\mathcal{C}$ be the list of the children of $node$
  3: **for** $child \in \mathcal{C}$ **do**
  4:     Let $\mathcal{I}_{child}$ be the row indices of the node $child$ in L
  5:     $\mathcal{I}_\cap \leftarrow \mathcal{I}_{child} \cap \mathcal{I}$
  6:     Let dep_child be the set of vectors in node $child$ that contains the indices in $\mathcal{I}_\cap$
  7:     $dep \leftarrow dep \cup dep\_child$
  8: **end for**

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 *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 *synchronisation tasks* to control the scheduling of the task. A synchronisation 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 $N = |dep|$ 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, *i.e.* 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 *dep* are considered to be *unsatisfied* and are split into $\lceil N/c \rceil$ chunks. For each chunk, a synchronisation task is submitted to the runtime system. This synchronisation task is scheduled when all dependencies in the chunk are *satisfied*. It then releases the first dependency of the chunk for the next level by flagging it as *unsatisfied*. Thus, the k-ary reduction leads to at most c unsatisfied dependencies, as presented in Figure 4.1. 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 *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 *dep*.

Along with this k-ary tree, we use a case statement over the number of unsatisfied dependencies in *dep*, where each case corresponds to the submission of a (synchronisation) task having i dependencies ($0 \le i \le c$), where c is 10 by default.
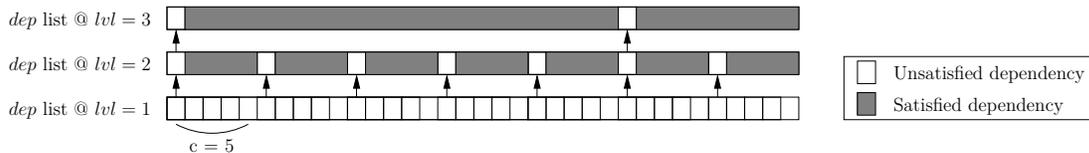

Figure 4.1: Content of the set *dep* at each level of the k-ary tree, with a chunk size $c = 5$. Each white square represents a dependency in *dep*, and each grey rectangle is composed of dependencies already satisfied.

4.3 The pruning strategy

The nodes of the tree of the L factor may be so small that the time spent in submitting and scheduling the task is larger than the runtime of the kernel. We use the same pruning strategy as described in [4] to address this problem. This pruning strategy consists of going through the tree from the root to the leaves to select nodes that have a similar arithmetic intensity. By arithmetic intensity we mean the number of operations performed by the kernels in the node and in all its descendants. The tree is therefore split into subtrees, where each has a selected node as its root. Each subtree then becomes a super task submitted to the runtime system and is processed by a single worker. The number of tasks submitted to the runtime system is therefore drastically reduced, as well as the number of the dependencies. Although the pruning decreases the tree-level parallelism, it leads to better data locality. Note that the number of subtrees is generally not equal to the number of workers even though the number of subtrees depends on the number of workers.

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 SPD matrices, presented in Table A.1. These matrices come from the Suitesparse matrix collection [6] and are those used in [8, 11]. They correspond to a wide range of different applications including CFD, circuit simulation, FEM, and gas reservoir modelling. The dimensions of the matrices range from 36×10^3 to 1.5×10^6. We focus our experiments on parallel efficiency both when the number of right-hand sides increases, and when the number of workers increases. We also compare the performance of the SpLLT solve phase with the solve phase of other sparse direct solvers.

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 128GB of memory. Each core has a theoretical peak of 36.8G flop/s for a frequency of 2.3GHz, 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. We have done preliminary tests to study the reproducibility of the execution times of the machine by solving equation (1.1) ten times and increasing the number of workers. We show, in Figure 5.1, 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 the number of workers increases.
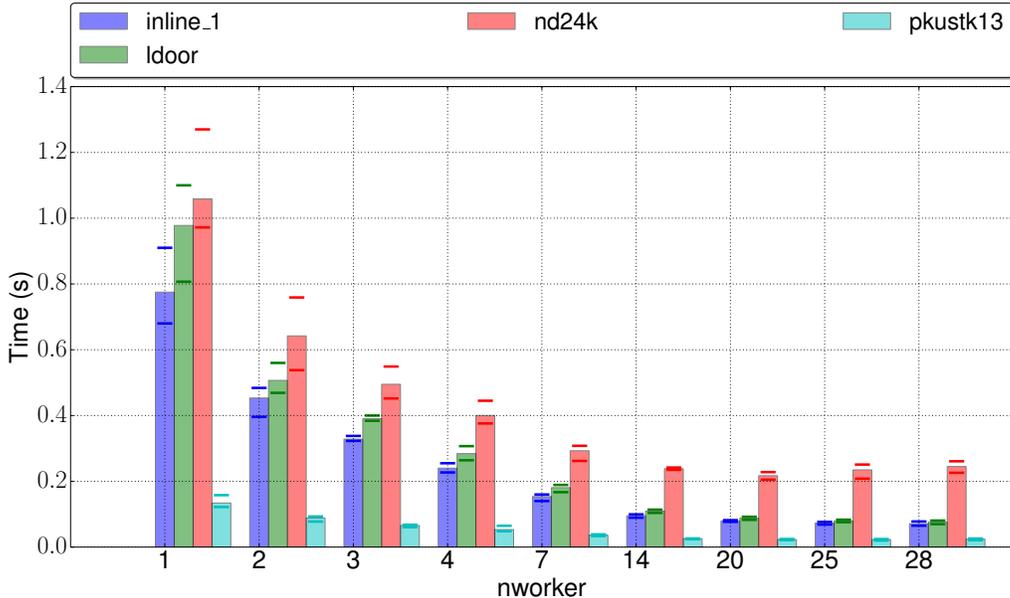
Figure 5.1: Reproducibility of execution times.

We first study the impact of pruning in Section 5.1, of synchronisation 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 [13] and PaStiX [10] by increasing the number of right-hand sides (*nrhs*) in 5.4 and the number of workers (*nworker*) in 5.5.

5.1   Impact of pruning on the performance

As discussed in Section 4.3, we use a pruning mechanism to reduce the impact of small nodes on the solve time. Table 5.1 presents the times to solve the system with one right-hand side. The number of workers ranges from 1 to 28, and the number of subtrees generated by the pruning increases when the number of workers increases. The set of matrices presented in Table 5.1 is a subset of the matrices in Table A.1 which perform a high number of flops during the factorization. We see that pruning improves the performance in all cases. The effect is less marked when the number of workers is low but we observe that pruning reduces the time by up to a factor of 6 for boneS10 with 28 workers. In the following we always activate this feature, unless we explicitly state otherwise.

Matrices		#worker				
		1	2	4	14	28
boneS10	# Subtree	1	11	25	60	178
boneS10	no pruning	1.26	0.80	0.55	0.32	0.60
boneS10	with pruning	1.14	0.79	0.38	0.15	**0.10**
StocF-1465	# Subtree	29105	29108	29115	29177	29189
StocF-1465	no pruning	3.03	2.54	1.46	0.82	1.51
StocF-1465	with pruning	2.83	2.41	1.27	**0.66**	0.75
Fault_639	# Subtree	1	5	18	43	88
Fault_639	no pruning	4.02	2.23	1.07	0.43	0.55
Fault_639	with pruning	2.48	2.05	0.99	0.40	**0.33**
Hook_1498	# Subtree	1	11	19	82	111
Hook_1498	no pruning	5.01	3.62	1.87	0.96	1.39
Hook_1498	with pruning	4.92	3.43	1.71	0.58	**0.43**
Emilia_923	# Subtree	1	6	21	67	99
Emilia_923	no pruning	5.10	2.85	1.59	0.65	0.83
Emilia_923	with pruning	4.66	2.77	1.49	0.61	**0.47**
Serena	# Subtree	1	5	11	37	110
Serena	no pruning	6.42	4.88	2.72	1.12	1.35
Serena	with pruning	6.13	4.56	2.41	0.96	**0.76**

Table 5.1: The solve time (secs) and number of subtrees when pruning is either enabled or disabled.

5.2 Impact of the number of synchronisation 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 synchronisation tasks on the solve time. We consider the case of one right-hand side and 64 right-hand sides. Table 5.2 presents the results on the same subset of matrices as in Table 5.1. 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 synchronisation tasks is divided by at least three when increasing the chunk size from two to three. After some point, the number of synchronisation 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 synchronisation tasks is divided by at least a factor of four for a chunk size of two. 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 synchronisation 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 synchronisation tasks. In the following, we

use a chunk size of 10 that we consider large enough to avoid noticeable impact on the results due to the synchronisation tasks.

Matrix	pruning	nrhs	Chunk size								
			2	3	4	5	6	7	8	9	10
boneS10	# synchronisations		38048	11750	1768	80	4	0	0	0	0
	no	1	0.79	0.60	0.65	0.58	0.53	**0.51**	0.53	0.58	0.55
		64	0.59	0.59	0.59	0.59	0.58	0.58	**0.57**	0.59	0.58
	# synchronisations		3822	1330	454	128	36	4	4	0	0
	yes	1	0.12	0.11	0.11	0.11	0.11	0.11	**0.10**	0.11	0.11
		64	0.45	**0.44**	0.49	0.46	0.49	0.47	0.50	0.46	0.48
StocF-1465	# synchronisations		111357	37026	8436	210	36	6	4	0	0
	no	1	2.21	1.70	1.89	1.62	1.68	1.59	**1.51**	1.62	1.57
		64	1.97	2.01	**1.87**	2.03	2.01	2.00	2.04	2.08	2.07
	# synchronisations		11544	3558	1257	336	267	220	200	177	169
	yes	1	0.72	0.74	0.73	0.76	0.79	0.73	0.74	0.84	**0.72**
		64	1.78	1.71	1.77	1.74	1.73	1.75	1.82	1.82	**1.68**
Fault_639	# synchronisations		75675	24751	5804	278	54	6	0	0	0
	no	1	0.94	0.70	0.59	0.55	**0.49**	0.60	**0.49**	0.59	0.55
		64	1.15	1.16	1.19	1.14	**1.13**	1.16	1.19	1.14	1.18
	# synchronisations		17021	4296	1531	416	290	217	190	172	166
	yes	1	0.34	0.33	0.34	0.33	0.33	0.34	0.33	0.35	**0.32**
		64	1.07	1.12	**1.03**	1.06	1.06	1.06	1.05	1.12	1.04
Hook_1498	# synchronisations		157924	46416	10999	699	102	26	6	2	2
	no	1	2.21	1.54	1.56	1.40	1.36	**1.26**	1.29	1.49	1.28
		64	1.97	1.90	1.89	1.92	**1.85**	1.91	1.87	1.88	1.89
	# synchronisations		20128	4306	1611	404	265	226	196	177	165
	yes	1	0.45	0.46	0.44	0.44	0.45	0.44	0.44	0.46	**0.43**
		64	**1.60**	1.69	1.66	1.62	1.62	1.65	1.66	1.63	1.70
Emilia_923	# synchronisations		109177	35937	8369	341	77	22	8	4	2
	no	1	1.49	1.00	0.84	0.73	**0.70**	0.80	0.82	0.80	0.78
		64	1.80	1.65	**1.58**	1.72	1.61	1.72	1.63	1.77	1.71
	# synchronisations		21420	5252	1871	550	382	307	258	231	210
	yes	1	0.46	0.46	**0.44**	0.46	0.45	0.45	0.47	0.46	0.45
		64	1.66	1.57	1.43	1.67	1.56	1.59	1.44	1.45	**1.42**
Serena	# synchronisations		186918	52156	12392	640	162	55	28	22	18
	no	1	2.63	1.65	1.31	1.32	**1.27**	1.28	1.31	1.36	1.41
		64	2.70	2.69	2.72	2.67	2.67	2.71	2.68	**2.57**	2.72
	# synchronisations		47629	8668	3083	639	456	356	300	272	246
	yes	1	0.78	0.76	0.74	**0.72**	0.76	0.76	0.75	0.75	0.75
		64	2.34	2.36	2.47	2.41	2.35	2.35	2.39	**2.32**	2.40

Table 5.2: Time and number of synchronisation tasks for our SpLLT solve with respect to the chunk size. Number of workers is 28.

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 5.3. 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 the number of workers 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.

| Matrices | | 128 | block size | | | |
			256	512	1024	2048
boneS10	Factor time	1.17	1.06	**1.03**	1.54	2.33
	Solve time	**0.84**	0.85	0.96	1.10	1.18
nd24k		46.70	13.60	**6.04**	6.73	8.73
		1.73	2.23	3.21	3.36	4.49
StocF-1465		32.70	8.97	8.15	**8.10**	10.70
		2.87	3.05	3.15	3.48	3.79
Fault_639		49.90	17.80	**13.60**	13.90	16.50
		1.96	**1.94**	2.76	3.82	4.84
Hook_1498		68.20	20.30	**16.60**	17.10	22.10
		3.02	3.08	3.52	4.03	5.00
Emilia_923		112.00	40.70	**21.30**	22.80	23.90
		3.08	**2.76**	3.39	3.81	4.94
Serena		222.00	83.00	52.10	**49.70**	51.30
		4.51	**4.43**	5.27	6.51	8.88

Table 5.3: The effect of different block sizes on both the factorize and solve phases. *nrhs* is equal to 128. The number of workers is 28.

5.4 Strong scaling on the number of right-hand sides

In this section, we focus on the effect of the number of right-hand sides on the time to solve the system. We fix the number of workers to 28, the block size to 256, and we vary the number of right-hand sides from 1 to 128 in powers of 2.

We first compare, in Table 5.5, SpLLT with PARDISO, 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 A.1. SpLLT is the fastest to solve the system, up to 8 times faster than PARDISO for one right-hand side and up to over a factor of 10 on two right-hand sides. There are only five problems for which SpLLT is not the fastest of all four codes for both one 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 1×1 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, where PaStiX is faster than SpLLT for one right-hand side (43.70 v 44.70 secs) SpLLT is faster for two right-hand sides (47.70 v 58.6 secs). 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 Table 5.6 we give the time for SpLLT, PARDISO, PaStiX and HSL_MA87, for all matrices of Table A.1 when solving for 16 and 128 right-hand sides. When *nrhs* increases, PaStiX and HSL_MA87 become less competitive compared to 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 5.4. These 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.

Matrix	Time 1rhs (10^{-2}s)	T_2/T_1	T_{16}/T_1	T_{128}/T_1
boneS10	10.10	1.23	1.82	8.52
nd12k	12.90	1.08	1.76	8.60
nd24k	22.40	1.41	2.04	9.87
Flan_1565	38.90	1.12	1.61	7.02
bone010	28.70	1.11	1.59	6.52
StocF-1465	76.60	1.14	1.33	3.72
audikw_1	35.50	1.09	1.59	6.54
Fault_639	33.10	1.15	1.61	5.74
Hook_1498	43.60	1.23	1.59	7.32
Emilia_923	44.70	1.07	1.51	6.51
Geo_1438	64.70	1.11	1.62	6.24
Serena	75.80	1.06	1.54	5.84

Table 5.4: Ratio of time$_{nrhs}$/time$_{1rhs}$.

nrhs	1				2			
	SpLLT	PARDISO	MA87	PaStiX	SpLLT	PARDISO	MA87	PaStiX
Matrices								
thermal2	**5.83**	21.70	16.20	13.20	**7.63**	107.00	26.90	15.70
gearbox	**1.84**	10.30	2.79	3.38	**2.20**	17.10	4.24	4.36
m_t1	**1.64**	7.93	2.17	2.20	**1.89**	10.10	3.63	2.78
pwtk	**2.48**	12.00	4.01	3.37	**2.77**	20.30	6.86	4.14
pkustk13	**1.90**	6.56	2.13	2.43	**1.98**	10.50	3.41	2.75
crankseg_1	**1.61**	6.92	2.02	1.90	**1.97**	9.52	2.85	2.33
cfd2	**2.01**	8.66	2.85	3.12	**2.55**	15.30	3.91	3.66
thread	2.30	6.27	**1.63**	1.81	2.23	6.82	2.38	**2.16**
shipsec8	**2.37**	8.31	2.69	2.68	**2.80**	11.10	3.97	3.86
shipsec1	**2.43**	8.93	2.69	3.07	**2.68**	16.10	4.26	3.82
crankseg_2	**2.08**	8.67	2.36	2.51	**2.60**	10.70	3.18	2.78
fcondp2	**2.86**	12.80	3.93	3.87	**3.22**	21.30	6.45	4.69
af_shell3	**4.25**	27.30	8.32	8.15	**5.09**	39.20	13.20	9.33
troll	**3.10**	18.60	4.03	3.75	**3.85**	24.40	7.60	4.48
G3_circuit	**8.89**	34.90	25.70	17.40	**9.98**	163.00	35.40	19.50
bmwcra_1	**2.56**	15.10	3.27	3.78	**2.95**	19.90	4.87	4.61
halfb	**3.29**	15.90	5.26	4.31	**3.70**	26.70	7.15	5.87
2cubes_sphere	**2.63**	8.53	3.03	2.83	**3.11**	14.80	4.31	3.69
ldoor	**6.92**	42.10	14.60	13.30	**8.26**	69.10	24.30	14.00
ship_003	**3.18**	10.60	3.52	3.29	**3.82**	16.10	4.86	4.31
fullb	**3.91**	16.40	4.53	4.40	**4.40**	20.80	6.74	5.68
inline_1	**6.35**	40.90	9.73	11.40	**7.55**	57.00	16.90	12.00
pkustk14	**4.83**	23.30	5.67	4.94	7.56	29.50	**6.12**	6.23
apache2	**7.99**	28.70	13.40	12.90	**10.20**	96.10	20.20	15.50
F1	**5.92**	33.40	9.72	8.62	**6.92**	48.70	11.40	11.10
boneS10	**10.10**	61.30	16.80	17.10	**12.40**	105.00	28.40	20.60
nd12k	12.90	19.40	**4.55**	5.12	13.90	28.90	5.90	**5.11**
nd24k	22.40	54.60	10.90	**10.70**	31.50	84.80	**13.30**	17.50
Flan_1565	**38.90**	290.00	54.70	46.90	**43.40**	369.00	72.60	65.30
bone010	**28.70**	209.00	33.80	50.10	**31.80**	264.00	47.90	61.20
StocF-1465	76.60	223.00	**45.00**	51.80	87.00	354.00	66.00	**61.20**
audikw_1	**35.50**	215.00	36.70	47.80	**38.60**	268.00	55.50	53.70
Fault_639	**33.10**	226.00	33.30	45.30	**38.10**	253.00	45.40	42.20
Hook_1498	**43.60**	298.00	59.00	91.70	**53.70**	393.00	80.10	96.80
Emilia_923	44.70	358.00	50.80	**43.70**	**47.70**	345.00	63.60	58.60
Geo_1438	**64.70**	481.00	76.60	93.90	**71.50**	504.00	103.00	107.00
Serena	**75.80**	485.00	86.30	130.00	**80.70**	630.00	114.00	113.00

Table 5.5: Comparison of SpLLT, PARDISO and PaStiX, for one and two right-hand sides. Times in 10^{-2} seconds. Number of workers is 28.

nrhs		16				128		
	SpLLT	PARDISO	MA87	PaStiX	SpLLT	PARDISO	MA87	PaStiX
Matrices								
thermal2	**11.60**	179.00	215.00	68.80	**55.50**	289.00	1420.00	484.00
gearbox	**3.63**	14.40	22.80	7.17	**14.70**	28.50	178.00	69.00
m_t1	**2.93**	12.30	18.90	5.36	**12.90**	31.70	118.00	39.50
pwtk	**4.32**	20.00	32.40	8.39	**18.50**	55.20	307.00	92.60
pkustk13	**3.55**	18.00	15.20	4.80	**15.40**	40.40	135.00	40.10
crankseg_1	**3.16**	11.60	10.30	4.60	**13.20**	31.00	77.50	23.50
cfd2	**3.95**	19.20	22.80	6.91	**15.80**	54.20	147.00	56.60
thread	**3.36**	12.30	7.39	4.10	**11.60**	28.40	52.00	19.10
shipsec8	**4.27**	15.60	19.00	7.16	**18.40**	35.30	168.00	52.90
shipsec1	**4.09**	15.30	27.10	7.58	**17.40**	37.20	173.00	56.80
crankseg_2	**4.01**	15.00	12.00	5.15	**15.50**	46.40	102.00	25.50
fcondp2	**5.40**	21.70	30.60	10.60	**23.00**	51.60	237.00	73.60
af_shell3	**9.06**	44.30	73.00	22.80	**36.20**	102.00	584.00	215.00
troll	**5.92**	21.10	39.10	10.40	**24.30**	72.90	279.00	85.30
G3_circuit	**17.40**	249.00	228.00	102.00	**74.70**	371.00	1840.00	808.00
bmwcra_1	**4.67**	21.00	23.40	8.20	**20.40**	40.60	180.00	61.70
halfb	**6.17**	23.60	34.90	10.60	**24.80**	66.40	272.00	101.00
2cubes_sphere	**4.98**	22.80	18.10	7.23	**21.80**	51.30	133.00	49.30
ldoor	**18.20**	69.40	138.00	50.70	**63.80**	149.00	1340.00	491.00
ship_003	**5.55**	21.60	21.00	7.20	**22.70**	62.50	156.00	60.60
fullb	**6.99**	26.90	31.80	10.70	**30.30**	62.40	244.00	93.00
inline_1	**13.60**	65.90	75.40	31.80	**54.40**	192.00	603.00	208.00
pkustk14	**9.95**	32.30	32.20	10.70	**39.20**	85.90	188.00	79.90
apache2	**14.50**	122.00	132.00	49.10	**66.20**	250.00	855.00	406.00
F1	**12.40**	60.30	66.00	21.40	**50.00**	98.80	508.00	159.00
boneS10	**18.40**	115.00	165.00	72.80	**86.10**	197.00	1310.00	415.00
nd12k	22.70	42.00	13.90	**8.90**	111.00	112.00	75.50	**34.10**
nd24k	45.70	76.10	29.90	**21.50**	221.00	203.00	148.00	**75.70**
Flan_1565	**62.60**	383.00	284.00	138.00	**273.00**	663.00	2030.00	764.00
bone010	**45.60**	263.00	171.00	99.40	**187.00**	468.00	1480.00	545.00
StocF-1465	**102.00**	345.00	308.00	152.00	**285.00**	822.00	1910.00	710.00
audikw_1	**56.40**	322.00	204.00	117.00	**232.00**	835.00	1440.00	551.00
Fault_639	**53.20**	290.00	144.00	87.20	**190.00**	520.00	1020.00	428.00
Hook_1498	**69.30**	353.00	266.00	203.00	**319.00**	1030.00	1920.00	1040.00
Emilia_923	**67.70**	391.00	222.00	147.00	**291.00**	717.00	1520.00	878.00
Geo_1438	**105.00**	512.00	334.00	190.00	**404.00**	1130.00	2130.00	981.00
Serena	**117.00**	654.00	295.00	211.00	**443.00**	1380.00	2130.00	1080.00

Table 5.6: Comparison of SpLLT, PARDISO and PaStiX, when the number of right-hand sides is 16 and 128. Times in 10^{-2} seconds. Number of workers is 28.

Since HSL_MA87 does not scale with the number of right-hand sides, we compare the time to solve for SpLLT with PARDISO and PaStiX on four problems that show different behaviour in Figure 5.2.

In all cases and for all *nrhs*, SpLLT is the fastest solver sometimes by a considerable amount. On two of the cases, PARDISO does better than PaStiX but the opposite is true in the other two cases. PaStiX scales much worse than SpLLT as *nrhs* increases. The scalability of PARDISO can vary considerably (see Emilia_923 and Serena). Since the source code for PARDISO is not distributed, we do not know why this is the case. We have similarly erratic scalability of several other matrices in our test set. In Figure 5.2(a), SpLLT is six times faster than PARDISO for two right-hand sides, and these two curves increase similarly when the number of right-hand sides increases. On the other hand, the time to solve using PaStiX is close to SpLLT for two right-hand sides, but is eight times slower on 128 right-hand sides. In all cases, SpLLT scales better than the other codes.
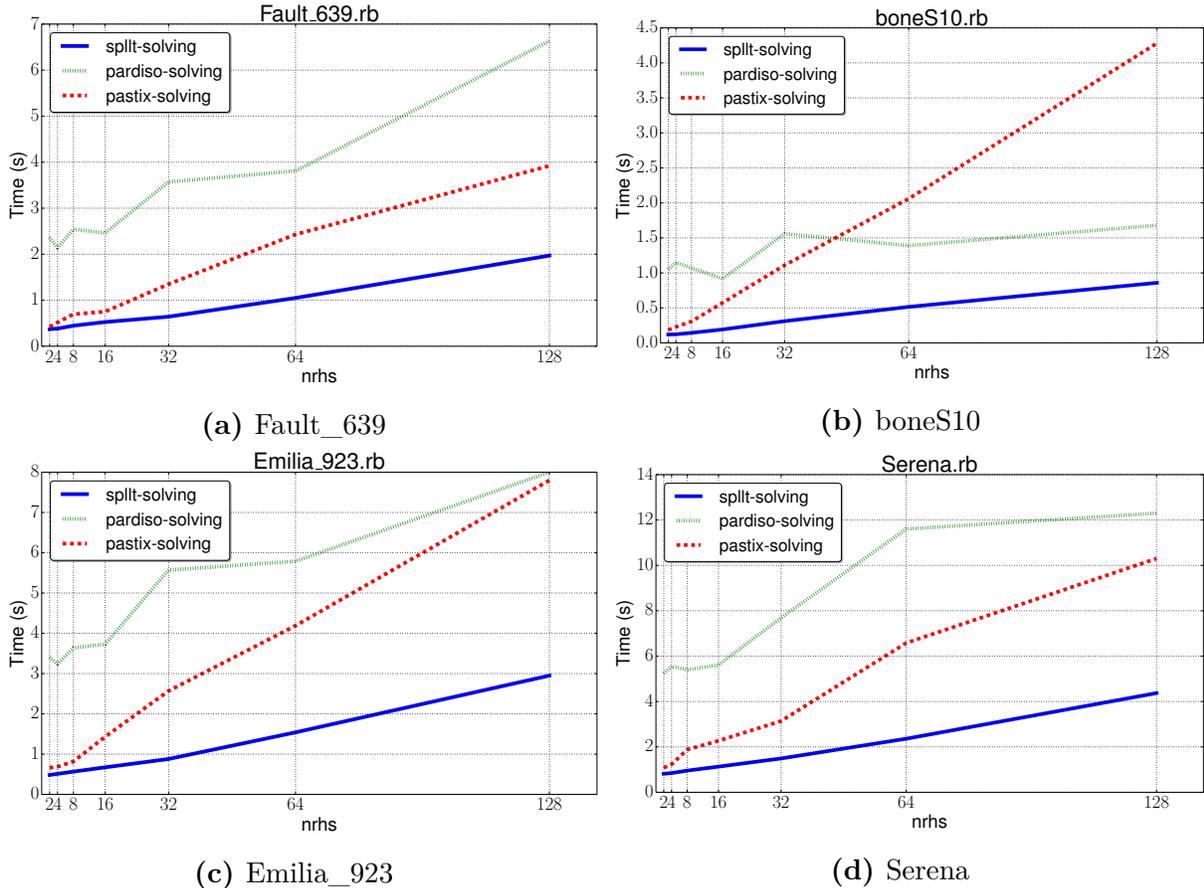

(a) Fault_639

(b) boneS10

(c) Emilia_923

(d) Serena

Figure 5.2: Comparison of the time to solve the system with multiple right-hand sides using SpLLT, PARDISO and PaStiX. The number of workers is set to 28, and the number of right-hand sides increases from 2 to 128 in powers of 2.

5.5 Strong scaling on the number of workers

In this section we study the impact of the number of workers on the time to solve a system with 1 and 64 right-hand sides. The number of workers increases from 1 to 28, and the block size is 256. We first consider one right-hand side and present some results in Figure 5.3. We use the same four matrices as in Section 5.4. The experiments show that for a small number of workers, the time to solve using SpLLT is greater than both other solvers. When the number of workers increases, the solid curve that represents SpLLT becomes closer to the other curves. For 28 workers, SpLLT is the fastest solver.

Figure 5.4 shows the impact of increasing the number of workers when there are 64 right-hand sides. SpLLT is more competitive than for a small number of right-hand sides. In fact, SpLLT outperforms PARDISO for all numbers of workers and is faster than PaStiX when the number of workers increases. For more than nine workers, SpLLT is clearly the fastest solver.
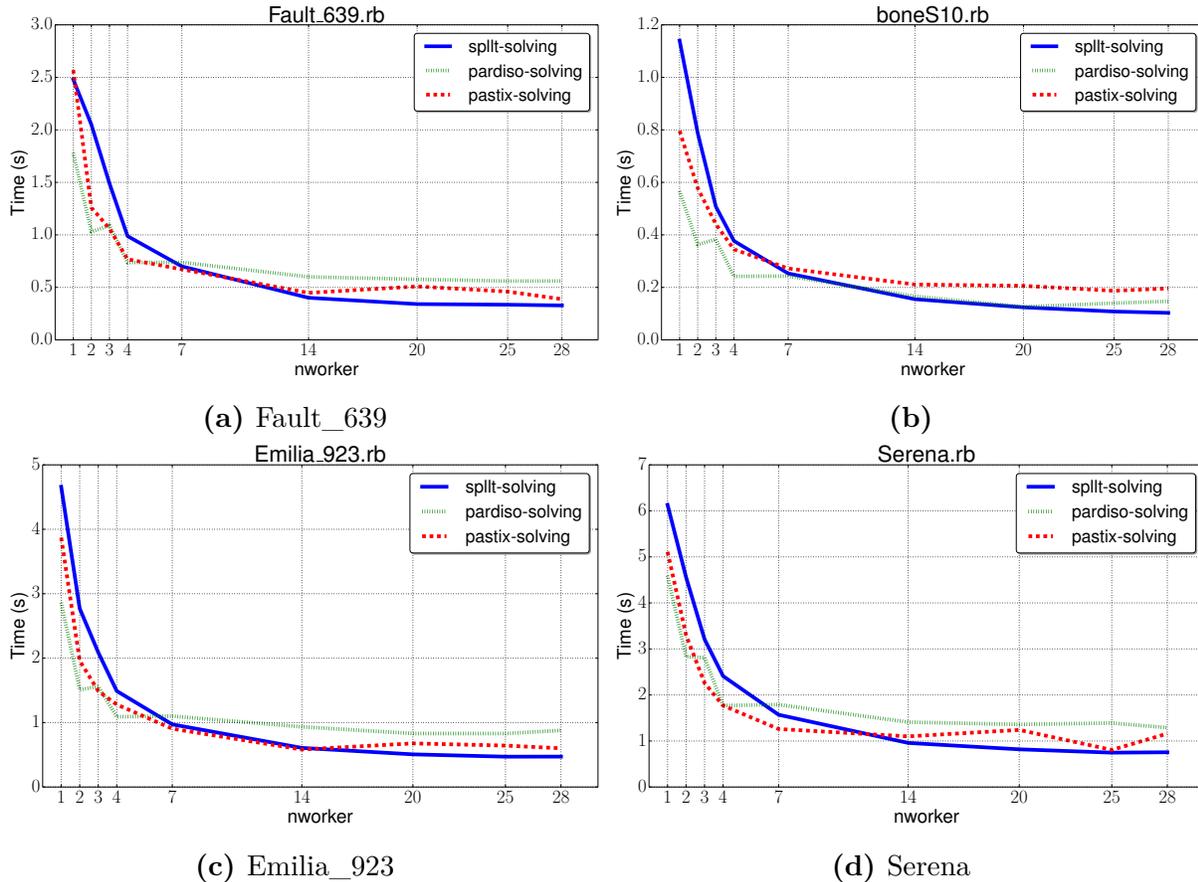

(a) Fault_639

(b)

(c) Emilia_923

(d) Serena

Figure 5.3: Comparison of the time to solve the system with one right-hand side for SpLLT, PARDISO and PaStiX.

(a) Fault__639

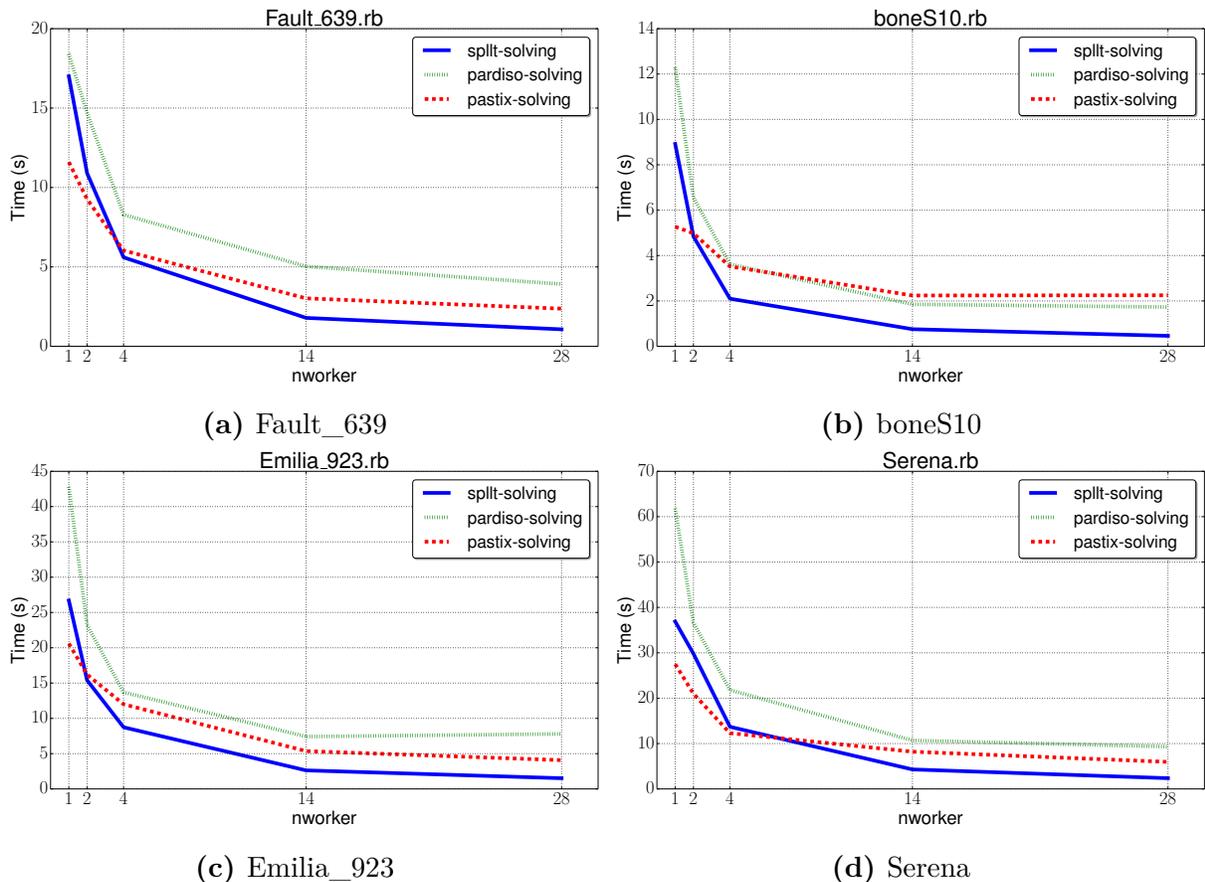**(b)** boneS10

(c) Emilia__923

(d) Serena

Figure 5.4: Comparison of the time to solve the system with 64 right-hand sides for SpLLT, PARDISO and PaStiX.

5.6 Application case : Enlarged Conjugate Gradient solver

In this section we present some results obtained with the Enlarged CG solver (ECG) [9]. Our runs in this section are on a different machine to our earlier results. We use the Kebnekaise system at Umeå in Sweden[1]. 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 a 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 conjugate gradient 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 5.7 shows the time to solve the global system with ECG using PARDISO or SpLLT on a subset of matrices from Table A.1. We show results using from 16 to 256 MPI processes. We see little difference in solution times for the shipsec1 problem but, on the two other problems that are ten times larger, we

---

observe that replacing PARDISO solver with the SpLLT solver leads to a speedup over a factor of two. In all cases, solving the system with 16 MPI processes using SpLLT is faster than using PARDISO with 32 MPI processes.

Problem	# MPI	PARDISO t (s)	PARDISO # iter	SpLLT t (s)	SpLLT # 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

Table 5.7: Time to solution using PARDISO and SpLLT. 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 the number of right-hand sides. 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 ten. 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 paper. For example, on a matrix from a diffusion problem of order nearly five 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 two on 256 processes.

7 Acknowledgements

References

[1] P. Amestoy, T. Davis, and I. Duff, *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, 17 (1996), pp. 886–905.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: A unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23 (2011), pp. 187–198.

[3] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, *An updated set of Basic Linear Algebra Subprograms (BLAS)*, ACM Trans. Math. Softw., 28 (2002), pp. 135–151.

[4] A. Buttari, *Fine granularity sparse QR factorization for multicore based systems*, in Applied Parallel and Scientific Computing, K. Jónasson, ed., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 226–236.

[5] L. Dagum and R. Menon, *OpenMP: An industry-standard API for shared-memory programming*, IEEE Comput. Sci. Eng., 5 (1998), pp. 46–55.

[6] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.

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

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

[9] L. Grigori and O. Tissot, *Reducing the communication and computational costs of enlarged Krylov subspaces conjugate gradient*, Research Report RR-9023, Inria, Feb 2017.

[10] P. Hénon, P. Ramet, and J. Roman, *PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems*, Parallel Computing, 28 (2002), pp. 301–321.

[11] J. Hogg, J. Reid, and J. Scott, *Design of a multicore sparse Cholesky factorization using DAGs*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3627–3649.

[12] G. KARYPIS AND V. KUMAR, *Multilevel k-way partitioning scheme for irregular graphs*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 96–129.

[13] O. SCHENK AND K. GÄRTNER, *Two-level dynamic scheduling in PARDISO: improved scalability on shared memory multiprocessing systems*, Parallel Computing, 28 (2002), pp. 187–197.

Appendix A   Test problems

#	Name	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	Pressurised wind tunnel
5	Chen/pkustk13	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	Schenk_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/bmwcra_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/ldoor	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/pkustk14	152	7.5	106.8	146.4	Tall building
24	GHS_psdef/apache2	715	2.8	134.7	174.3	3D structural problem
25	Koutsovasilis/F1	344	13.6	173.7	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	30048.9	Gas reservoir

Table A.1: Test matrices and their characteristics without node amalgamation. 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.