NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D3.2

Algorithm design for symmetrically structured factorizations

October 2017

Document information

Scheduled delivery	2017-10-31
Actual delivery	2017-10-31
Version	1.0
Responsible partner	STFC

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2017-07-10	Iain Duff	Draft	0.1	Just starting up to get layout and contents
2017-10-09	Florent Lopez	Draft	0.2	Added descriptions of `SpLLT` and `SSIDS` solver
2017-10-25	Florent Lopez	Draft	0.3	Included modifications suggested by reviewers
2017-10-30	Florent Lopez	Version 1	1.0	Sent to UMEÅ for submission

Author(s)

Iain Duff, STFC
Florent Lopez, STFC

Internal reviewers

Jan Papez, INRIA
Mawussi Zounon, UNIMAN
Pierre Blanchard, UNIMAN
Sven Hammarling, UNIMAN

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Introduction

The *Description of Action* document states for Deliverable D3.2:

> "*D3.2 Algorithm design for symmetrically structured factorizations*
>
> *Report on algorithm design and to address issues around use of DAGs for sparse factorizations. Includes reporting on prototype code testing possible solutions.*"

This deliverable is in the context of Task–3.2 (Direct methods for (near-)symmetric systems).

This deliverable discusses the design of algorithms for the factorization of symmetrically structured matrices and is reporting on work in Task 3.2 of Workpackage 3.

In this work we are interested in solving the linear system of equations

$$Ax = b, \tag{1}$$

where A is a large, sparse, symmetrically structured matrix. Many applications require the solution of such linear systems of equations and in many cases direct methods are employed because of their robustness, accuracy and usability as black-box solvers. Using these methods, we aim to solve these problems as fast as possible by exploiting the computing capabilities of modern architectures that are highly parallel while ensuring the accuracy of the computed solution especially when tackling indefinite systems.

Modern HPC platforms typically consist of multiple compute nodes connected through a high speed network where each node is generally composed of multicore processors connected to a NUMA socket and accelerators such as Graphical Processing Units (GPU) or Xeon Phi devices. With the advent of multicore processors, task-based algorithms have been shown by Buttari's et al. [10] to yield good speedups for dense linear algebra factorization algorithms. The idea behind task-based algorithms is to decompose the workload into fine granularity tasks that can be executed in parallel as soon as their input data becomes available as a result of the execution of other tasks. These task inter-dependencies can be organised in a graph called a Directed Acyclic Graph (DAG). This approach has been exploited in the dense linear algebra software package PLASMA [3] which targets multicore architectures but has also been successfully adapted for targeting GPU-based heterogeneous systems in the Chameleon library [3]. This approach was used for sparse linear algebra with the design of new DAG-based algorithms for implementing sparse factorization. In the work by Hogg et al. [17] the authors used a DAG-based supernodal method for implementing the Sparse Cholesky solver `HSL_MA87`. Similarly Buttari implemented a DAG-based multifrontal QR method in the `qr_mumps` solver [9].

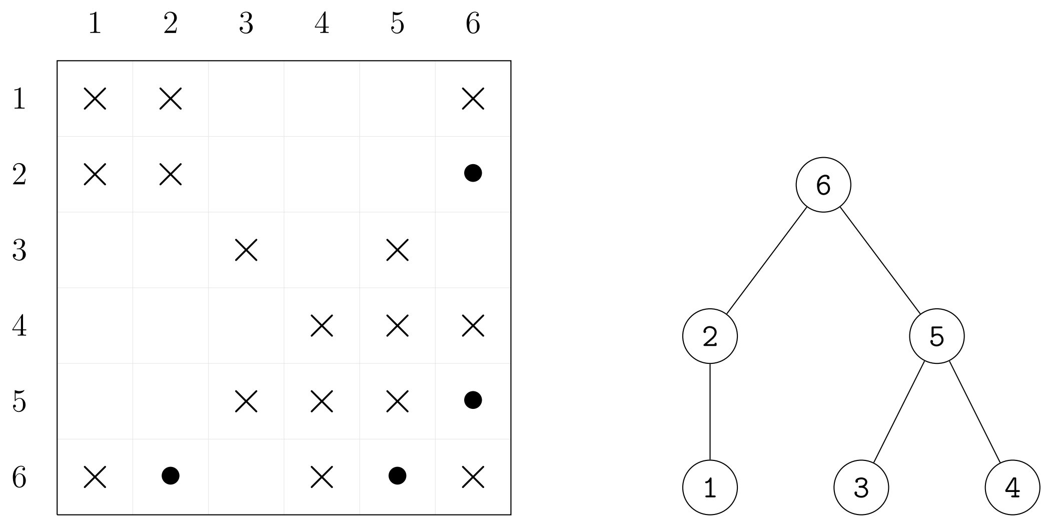
In order to face the complexity of modern architectures with an ever increasing number of cores per chip, a deep memory hierarchy and heterogeneous processors, a common approach consists in using a runtime system for executing DAGs in parallel. The runtime system plays the role of a software layer between the application, where the DAG is expressed using a high-level API, and the architecture. As it manages the task dependencies, data coherency and memory transfers, the runtime system removes the burden of handling low-level hardware details and increases portability and maintainability of the software. In this deliverable we are mainly interested in the two following runtime systems: StarPU [7] developed by the STORM team at INRIA

Bordeaux Sud-Ouest and Parsec [8] from ICL, University of Tennessee, Knoxville. We are also using the standard OpenMP with its tasking capabilities that were introduced in Version 3.5.

In this deliverable we describe the design of two sparse direct solvers for computing the solution of equation (1). First, we present in Section 6 the `SpLLT` package which is a Cholesky solver for solving symmetric positive-definite systems. Then, we introduce in Section 10 the `SSIDS` package which is a LDL^T solver for solving symmetric indefinite systems. In both cases we use a runtime system approach for implementing our parallel codes and we show competitive performance in comparison with the state-of-the-art solvers.

2 Sparse factorization for symmetrically structured systems

In this deliverable we focus on solving symmetrically structured sparse linear systems using direct methods. The solution is generally achieved in three main phases: the *analysis*, the *factorization* and the *solve* phases. The factorization phase consists in finding a decomposition of the original system into a product of matrices, called factors, with a simpler structure. The analysis phase is responsible for computing the dependencies between the coefficients in the factors and allocates the data structures needed during the factorization phase. The solve phase retrieves the solution using the decomposition produced by the factorization. The factorization phase is generally the most computationally intensive phase. For this reason most of our effort consists in speeding up the factorization by exploiting parallelism and improving kernel efficiency.



(a) Sparsity pattern. (b) Elimination tree.

Figure 1: Sparsity pattern of an illustrative 6-by-6 matrix (a) and its corresponding elimination tree (b). In (a) crosses represent nonzero entries in the original matrix whereas bullets represent fill-in entries.

2.1 Symmetric positive-definite systems

When the input system A is symmetric positive-definite, it can be decomposed using the Cholesky algorithm into

$$PAP^T = LL^T, \tag{2}$$

where the factor L is a lower triangular matrix. Then during the solve phase, the solution x in the original system can be computed through the solution of the systems $Ly = Pb$ and $L^T Px = y$ by means of forward and backward substitution. Note that the matrix L is normally denser than the matrix A because of nonzeros introduced in the elimination process. These are called *fill-in* and can be greatly reduced by a good choice for the permutation matrix P. Two main techniques for choosing a P to reduce fill-in are Minimum Degree [20, 19, 4, 5] or Nested Dissection [15] or variants of these methods. Note that in our software, the analysis is done by the `ssids_analyse` routine from the SSIDS package available in the SPRAL library[1].

In Figure 1(a) we show the sparsity pattern of a simple 6-by-6 symmetric matrix with 16 nonzero entries and 4 fill-in entries. In Figure 1(b) the so-called *elimination tree* associated with this matrix is shown. This tree, in which each node represents a column in the factor, expresses the dependencies between the coefficients during the factorization. The factorization is performed by traversing the assembly tree in a topological order i.e. a node can be processed after its children have been processed, and at each node performing two main operations: computing the column in the factor associated with the current node and then updating the ancestor node using the computed factor. Note that the columns that are located in independent branches can be processed in parallel. This is refereed to as *tree-level* parallelism. In our example, it is easy to see from the elimination tree in Figure 1(b) that columns 1, 3 and 4 can be treated in parallel at the beginning of the factorization.

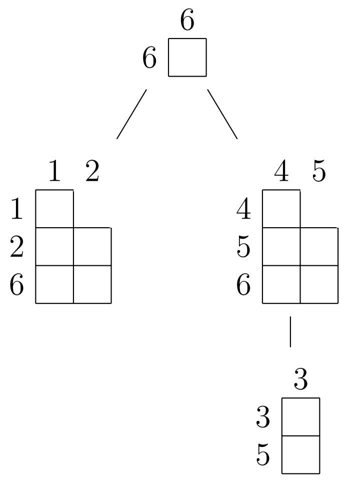


Figure 2: Assembly tree corresponding to the elimination tree in Figure 1(b) where the nodes pairs $\{1,2\}$ and $\{4,5\}$ have been amalgamated.

In order to increase the efficiency of operations by exploiting Level 3 BLAS routines we amalgamate columns that have a similar nonzero pattern and nodes in the elimination tree become dense matrices. This tree is called the *assembly tree* and the amalgamated columns

[1]https://github.com/ralna/spral

are referred to as a *supernode*. We illustrate such tree in Figure 2 with the assembly tree corresponding to our example matrix in Figure 1(a). Note that the supernodes have a trapezoidal shape because we exploit the symmetry and only manipulate coefficients in the lower triangular part of our matrix.

2.2 Symmetric indefinite systems

So far we have discussed the case of positive definite matrices, however when we are dealing with indefinite systems, we seek the decomposition

$$PAP^T = LDL^T, \tag{3}$$

where the permutation P is not only meant to preserve the sparsity of the system but is also used for pivoting to maintain stability during the factorization process and therefore allows us to compute the solution of equation (1) with good accuracy. Note that the matrix D is block diagonal with blocks of size 1×1 and 2×2. This is due to the fact that using only 1×1 pivots is not sufficient to guarantee the stability of the LDL^T factorization . In the indefinite case, the sparse factorization also uses the assembly tree except that at each supernode a dense LDL^T factorization is performed such as

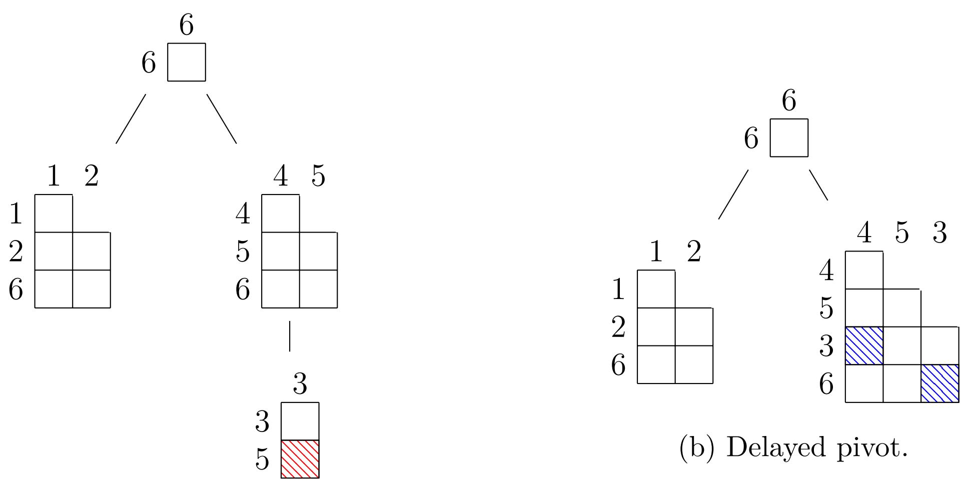
$$\begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} D_{11} & \\ & F_{21} - L_{21} D_{11} L_{21}^T \end{pmatrix} \begin{pmatrix} L_{11} & L_{21}^T \\ & I \end{pmatrix}, \tag{4}$$

where L_{11} is lower triangular, L_{21} rectangular and D_{11} is block diagonal with 1×1 and 2×2 blocks. Note that the pivots may only be chosen from F_{11}. Therefore, when computing the factor $L_{21} = F_{21}(D_{11}L_{11}^T)^{-1}$, it is necessary to verify the stability of the operation using some predetermined stability criterion. If a candidate pivot is found to be unsuitable it is moved to an ancestor supernode (the parent node in the multifrontal method) and is eliminated later during the factorization. Such pivots are said to be *delayed*. Note that, delaying pivots generally increases the floating-point operation and memory footprint of the factorization. It is therefore crucial to limit the number of delayed pivots and this can be done by using scaling techniques such as weighted matching scaling approach implemented in `MC64` [12, 13].

In Figure 3 we use the assembly tree from Figure 2 to illustrate how pivots might be delayed during the factorization process in order to preserve stability. In Figure 3(a) we show the case where the entry a_{33} cannot be used as a pivots to eliminate column 3 because of the entry shown in red. It is therefore necessary to delayed this column as there is no alternative for eliminating it. As a result the delayed pivot is sent to the parent supernode and put at the end of the fully-summed variable contained at this supernode. This procedure generate extra memory storage and floating-point operation because of the two new coefficients that are represented in blue.

3 Task-based sparse Cholesky factorization

As mentioned in the previous section, it is possible to identify several levels of parallelism in the assembly tree such as node-level and tree-level parallelism. In the parallel implementation of sparse factorization algorithms, these are generally exploited separately. In this case, a common approach consists in having several processes for handling independent branches in the tree thus taking advantage of the tree-level



(a) Failed entry.

(b) Delayed pivot.

Figure 3: Delayed pivot in the assembly tree (b) resulting from a failed pivot (a) when trying to eliminate column 3. The failed entry is represented in red and the blue parts correspond to the extra entries in the sparse structure due to the delayed pivot.

parallelism while using multithreaded BLAS kernels for processing supernodes in order to exploit node-level parallelism. In our work we instead follow the DAG-based approach taken by Hogg et al. [17] for the implementation of the solver `HSL_MA87`. As we explained during the introduction, this approach offers a significant improvement over the use of the traditional *fork-join* parallelism model in the dense case. By using the simple assembly tree depicted in Figure 4(a) we illustrate how we use the task-based Cholesky factorization in our assembly tree. In this tree, the supernodes are partitioned into square blocks of size `nb` and we operate on these blocks to achieve the factorization. The tasks, corresponding to the factorization of our simple assembly tree are illustrated in the DAG shown in Figure 4(b). The tasks in this DAG execute the following kernels:

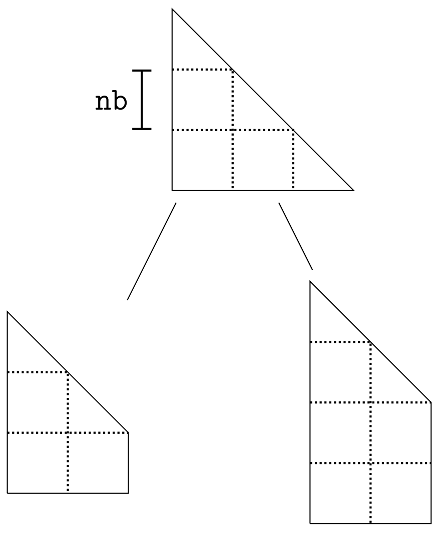
1. tasks denoted `f` correspond to the computation of the Cholesky factor of a diagonal block,

2. tasks denoted `s` perform a triangular solve of a subdiagonal block using a factor computed with a task `f`,

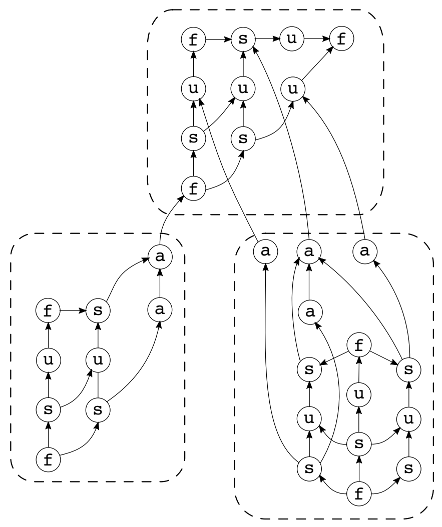
3. tasks denoted `u` perform an update of a block within a supernode corresponding to the previous factorization of blocks, and

4. tasks denoted `a` represent the update between supernodes with respect to the factorization blocks computed at a given node.

In our code, the DAG, such as that shown in Figure 4(b), replaces the elimination tree for expressing the dependencies during the computation of the factors. Note that when exploiting the node and tree level parallelism separately, it is not possible to start factorizing a supernode before all of its descendant nodes have been processed. However, when using the DAG, it is possible that some tasks in a given node become ready for execution and can then be scheduled while its descendants are still being processed. Using this DAG-based parallelism it is therefore possible to pipeline the processing of a given



(a) Simple assembly tree.



(b) Cholesky factorization DAG.

Figure 4: Simple assembly tree in which supernodes are partition into square block of size `nb` (a) and the DAG associated to the Cholesky factorization of this tree (b).

node with the processing of its ancestors. This additional level of parallelism allowed by the use of a DAG-based algorithm is referred to as *inter-node* parallelism.

The pseudo-code corresponding to the task-based Cholesky factorization is presented in Figure 5. Note that this is the sequential algorithm that is used as a basis for the implementation of our parallel code. In this code we have the following kernels:

- `alloc(snode)`: partitions the supernode `snode` into blocks and allocates the data structures.
- `init(snode)`: initializes the blocks by copying the coefficients from the original matrix into them.
- `factorize(bc_kk)`: computes the Cholesky factor of the diagonal block `bc_kk`.
- `solve(bc_kk, bc_ik)`: performs the triangular solve of an off-diagonal block `bc_ik` with the block resulting from the factorization of the diagonal block `bc_kk` in its column.
- `update(bc_ik, bc_jk, bc_ij)`: performs the update operation of a block `bc_ij` within a supernode using the blocks `bc_ik` and `bc_jk` from a previously processed column.
- `update_btw(snode, bc_ik, bc_jk, anode, bc_ij)`: performs the update operation between the factors computed in the blocks `bc_ik` and `bc_jk` in the supernode `snode` and the block `bc_ij` located in the ancestor supernode `anode`. In the pseudo-code, `p` and `q` represent the number of subdiagonal blocks involved in the update of the ancestor node `anode` in the k-th column of `snode`. Arrays `rmap` and `cmap` give respectively the row mapping and column mapping between the rows and columns in `snode` and in `anode`.

```
forall nodes snode in post-order
   ! Allocate data structures
   call alloc(snode)
   ! Initialize node structure
   call init(snode)
end do

forall nodes snode in post-order

  ! Factorize supernode with 'm' rows and 'n' columns
  do k=1..n in snode
    call factorize(blk(k,k))

    do i=k+1..m in snode
        call solve(blk(k,k), blk(i,k))
    end do

    do j=k+1..n in snode
       do i=k+1..m in snode
          call update(blk(j,k), blk(i,k), blk(i,j))
       end do
    end do

    forall ancestors(snode) anode
      do j=k+1..p in snode
         do i=j..q in snode
            call update_btw(blk(j,k), blk(i,k), a_blk(rmap(i),
   cmap(j)))
         end do
      end do
    end do

  end do
end do
```

Figure 5: Pseudo-code for the sequential version of the task-based sparse Cholesky factorization.

In this algorithm, we perform the update using a right-looking scheme. Although left and right-looking schemes can lead to different performance in serial mode, neither is considered better because their behaviour depends on the characteristics of the architecture. In a parallel mode, this code is used to create the task-graph corresponding to factorization and both left and right-looking schemes produce the same DAG. Although these two schemes might influence the order in which tasks are submitted to the runtime systems, in our approach these tasks are dynamically scheduled and prioritised depending on their position in the DAG and so are therefore independent of the submission order.

4 Programming models

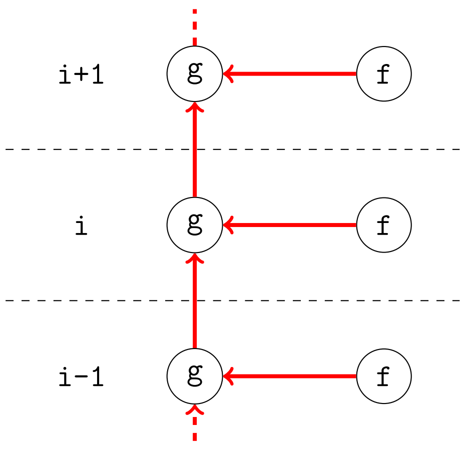
In this deliverable we focus on two different programming models for implementing task algorithms. In this section we first introduce the *Sequential Task-Flow* (STF) model which is an intuitive model largely based on the sequential algorithm and therefore very intuitive to use. Then we present a data-flow model called *Parametrized Task Graph* (PTG) which is extremely scalable but is much less easy to use than the STF model. In order to illustrate these two programming models, we will use the simple sequential code presented in Figure 6(a). An extract of the DAG associated with this example is shown in Figure 6(c).

```
for (i = 1; i < N; i++) {
  // Read and write x[i] with kernel f
  x[i] = f(x[i]);
  // Read x[i] and y[i-1] and write
  // in y[i] using kernel g
  y[i] = g(x[i], y[i-1]);
}
```

(a) Simple example of a sequential code.

```
for (i = 1; i < N; i++) {
  // Submit task executing kernel f
  submit(f, x[i]:RW);
  // Submit task executing kernel g
  submit(g, x[i]:R, y[i-1]:R, y[i]:W);
}
```

(b) STF code.



(c) DAG extract.

Figure 6: Simple example of a sequential code (a) and an extract of its corresponding DAG (c). In addition we propose an STF implementation of the sequential code (b).

4.1 The sequential task flow model

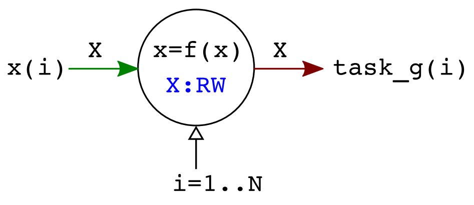
In the sequential task flow model the detection of dependencies between tasks relies on a data analysis of input and output data in order to guarantee the *sequential consistency* of operations during parallel execution. This analysis is often referred to as *superscalar* analysis in deference to the dependency detection between instructions that are performed in superscalar processors. In this context, the dependency graph is used to allow the parallel execution of independent instructions and is referred to as instruction-level parallelism. The STF model is the most commonly used paradigm for the parallelization of DAG-based algorithms. For example, several dense linear algebra software packages such as PLASMA [3] and FLAME [18] use this model in their implementation. One reason for its popularity is its ease of use: the parallel code is very similar to the sequential one. Essentially, for a given sequential algorithm, the function calls (i.e. the execution of tasks in the case of a DAG-based algorithm) are replaced by the asynchronous submission of the task to a runtime system for scheduling. Depending on the data access provided (read, write, or read/write), the runtime system automatically detects the dependencies between the tasks. The sequential consistency is then ensured by the fact that the order of submission of tasks corresponds to the sequential order.

As an example to illustrate the model, we show in Figure 6(b) the parallel code corresponding to our sequential example in Figure 6(a). In the sequential code, the two functions f and g manipulate arrays x and y. The STF code is obtained by submitting the tasks that consist of a kernel function (f or g in this example) together with data which are associated with a data access which can be R when the data is read, W when the data is written, and RW when the data is read and modified.

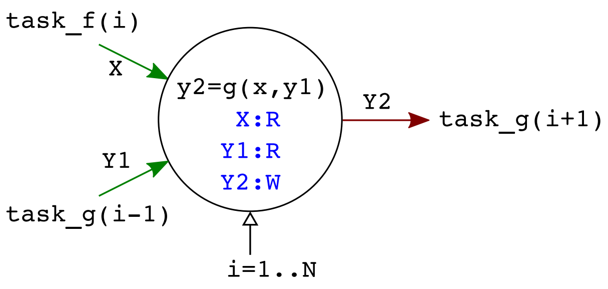
While easy to use, this model has several drawbacks that may affect performance and scalability. The tasks are issued and submitted to the runtime system sequentially. If the time to execute a task is small compared to the time needed for building and submitting a task, the parallel execution might be constrained by the time spent in the submission loop. To avoid this a *recursive* model could be used where intermediate tasks submit other tasks, enabling the parallelization of task submission. This could be implemented, for example, by using *callback* functions to trigger the submission of tasks that are executed on task completion. Another issue arising with the STF model comes from the fact that the whole DAG is unrolled during the parallel execution and every task in the DAG is stored in order to track task dependencies. In the case where the DAG is extremely large, handling and storing the DAG might represent a large overhead in terms of computational cost and memory. Although the recursive model allows us to mitigate the problem, it doesn't remove it, and it may be necessary to consider a different model such as the Parametrized Task Graph (PTG) model.

4.2 The parametrized task graph model

The PTG model is a dataflow programming model for representing a DAG and was introduced in [11]. In this model, the DAG is represented using a compact format, independent of the problem size, where dependencies are explicitly encoded. We introduce the PTG model by using the simple sequential code shown in Figure 6(a).



(a) Task executing kernel `f`.



(b) Task executing kernel `g`.

Figure 7: PTG representation for task types `task_f` (a) and `task_g` (b) as shown in the DAG presented in Figure 6(c).

The example in Figure 7(a) corresponds to a possible PTG representation for describing a DAG using a diagram language. It illustrates the fact that the PTG representation is independent of the size of the DAG (which depends on the parameter `N` in our example) and therefore has a limited memory footprint. In comparison, when using an STF model, the memory footprint for representing the DAG grows with the size of the

DAG because every task instance has to be kept in memory at least until its completion. Another interesting aspect of the PTG model comes from the fact that when the DAG is traversed in parallel every process involved in the execution only needs to traverse the portion of the DAG related to the tasks being executed in that process. Therefore, the DAG is handled in a distributed fashion which constitutes an advantage over the STF model where every process is required to unroll the whole DAG which could limit the scalability of the application on large systems.

5 Runtime systems

In this deliverable we mainly experiment with three runtime systems: StarPU[2] which is developed at the STORM team at INRIA Bordeaux Sud-ouest, PaRSEC[3] from the ICL at University of Tennessee and the OpenMP standard which includes features for implementing task-based algorithms since Version 3.0 of its API.

5.1 The StarPU and OpenMP runtime systems

The popularity of task-based algorithms persuaded the OpenMP board to introduce the *task* construct in Version 3.0 of its API. Then, motivated by the popularity of the STF model, the OpenMP committee decided to include the *depend* construct in Version 4.0 allowing users to express dependencies between tasks in a similar way to the STF model presented in Section 4.1. In this work we use an OpenMP implementation of our Cholesky solver and show advantages in terms of performance, scalability and productivity. However, because many features are still unavailable in the OpenMP standard, we also developed a version based on the StarPU runtime system. As shown in the next section, both implementations of our solver rely on a STF model, but the StarPU-based implementation can benefit from a wider range of features that are available with StarPU. For example, although we focus on shared-memory architectures in this paper, the StarPU version can be extended to a distributed-memory version whereas OpenMP can't be used on such architectures. In addition, OpenMP, unlike StarPU, does not give users any control over the scheduling of tasks. Every implementation of OpenMP provides a default scheduler which does not take into account the application. This can be very limiting especially when the application is executed in a heterogeneous context such as a GPU-accelerated multicore architecture.

We present in Figure 8 an example of a parallel implementation for the sequential code in Figure 6(a) using OpenMP. In this example we first create the parallel section using the omp construct *parallel* and then we put the master thread in charge of the task submission using the *master* construct. As previously explained, tasks are created with the *task* construct and data access is given to the runtime system using the *depend* construct. In the OpenMP standard, read-only data access is indicated by the parameter *in*, write-only by the parameter *out* and read-write by the parameter *inout*. Finally the task submission loop finishes with the *taskwait* clause indicating that the master thread should wait for the completion of the tasks previously submitted.

Similarly to the OpenMP example given in Figure 8 and in order to introduce the features provided by the StarPU API, we show in Figure 9 an example of a StarPU-based

[2]http://starpu.gforge.inria.fr/

[3]https://bitbucket.org/icldistcomp/parsec/

```
#pragma omp parallel
{
#pragma omp master
   {
      for (i = 1; i < N; i++) {
#pragma omp task depend(inout:x[i:1])
         x[i] = f(x[i:1]);
#pragma omp task depend(in:x[i], y[i-1:1]) depend(out:y[i:1])
         y[i] = g(x[i:1], y[i-1:1]);
      }
#pragma omp taskwait
   }
}
```

Figure 8: Simple example of a parallel version of the sequential code in Figure 6(a) using a STF model with OpenMP.

```
/* tasks submission */
for (i = 1; i < N; i++) {

  starpu_task_insert(&f_cl,
                     STARPU_RW, x_handle[i],
                     0);

  starpu_task_insert(&g_cl,
                     STARPU_R, x_handle[i],
                     STARPU_R, y_handle[i-1],
                     STARPU_W, y_handle[i],
                     0);
}

/* wait for all submitted tasks to be executed */
starpu_task_wait_for_all();
```

Figure 9: Simple example of a parallel version of the sequential code in Figure 6(a) using a STF model with StarPU.

implementation for the simple example presented in Figure 6(a). The task submission is done through the `starpu_task_insert` function that takes as input a *codelet* and a set of *handles*. A codelet corresponds to the description of a task and includes a list of computational resources where the task can be executed as well as the corresponding computational kernels. The data handles represent a piece of data that is accessed in the task and can be read (`STARPU_R`), written (`STARPU_W`), or read and written (`STARPU_RW`). In order to be used, a data handle must be *registered* to the runtime system by providing information such as a pointer to the data, its size and type. This information allows StarPU to automatically perform the data transfer between memory nodes during execution. For example, when data needs to be accessed on a GPU device, the runtime system automatically transfers it to the device memory node. As a result, StarPU is capable of ensuring data consistency over multiple nodes. When all the tasks have been submitted to the runtime system, we wait for their completion by calling the routine `starpu_task_wait_for_all`.

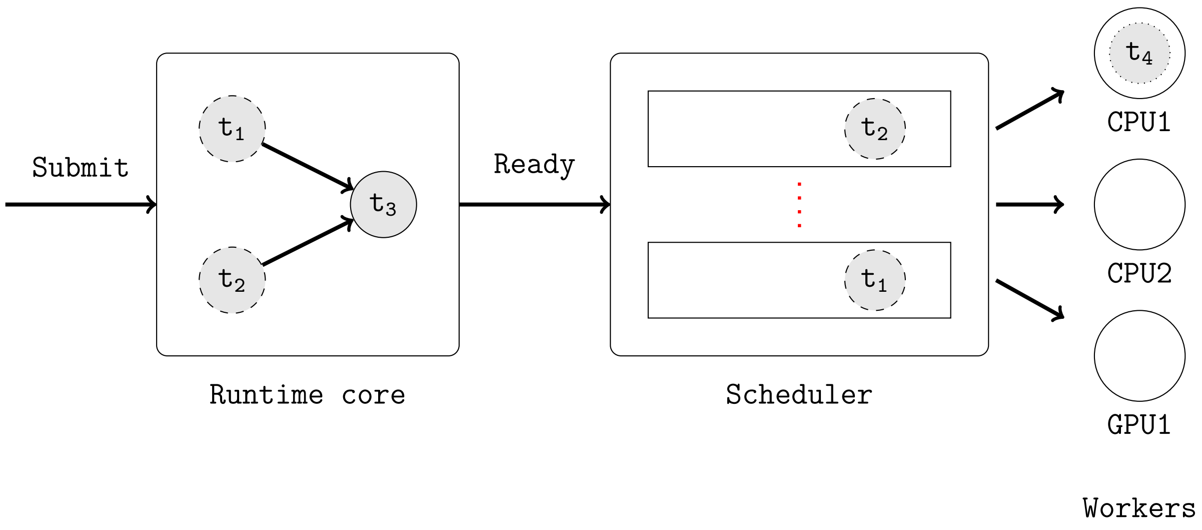


Figure 10: Illustration of the dynamic scheduling strategy of tasks in the runtime system.

Both OpenMP and StarPU implementations rely on a dynamic scheduler for scheduling the ready tasks during execution. In this model, a task is put in the scheduler as soon as it becomes ready for execution, which is when all of its dependencies are satisfied. Workers try to retrieve a task from the scheduler when they become idle. This dynamic scheduling strategy is illustrated in Figure 10 where the scheduler is placed between the runtime core where the DAG is built. The workers can be CPUs and GPUs, the scheduler is responsible for storing the ready tasks in scheduling queues and distributing them to idle workers. Although OpenMP Version 4.0 doesn't provide any features to control the scheduling policy, Version 4.5 allows users to provide a priority along with a submitted task so critical tasks are scheduled sooner. StarPU not only supports the use of task priorities but also makes it possible to use different scheduling strategies and to implement a new one if necessary.

5.2 The PaRSEC runtime system

The PaRSEC runtime system is one of the few libraries providing an interface for implementing PTG-based parallel codes. This is done by using a dedicated high-level language called Job Data Flow (JDF) for describing DAGs. The JDF code is translated into a C-code at compile time by the *parse_ptgpp* source-to-source compiler distributed with the PaRSEC library. The JDF codes contain a collection of task types, usually one for each kernel, associated with a set of parameters. These parameters are associated with a range of values and each value corresponds to a task instance. Tasks are associated with one or more data, and the dataflow is explicitly encoded for each task type. Several kernels can be attached to each task type depending on the resources available on the architecture.

In the context of distributed memory systems, users must provide the data distribution to the runtime system in addition to the JDF code which is used to map the task instances to the compute nodes during the execution.

```
 1 N      [type = int]
 2 
 3 task_f(i) /* Task name */
 4 
 5 i = 1..N-1 /* Execution space declaration for parameter i */
 6 
 7 : x(i) /* Task must be executed on the node where x(i) is stored
     */
 8 
 9   /* Task reads x(i) from memory ... */
10   RW X <- x(i)
11   /* ... and sends it to task_g(i) */
12        -> X task_g(i)
13 BODY
14 
15   X = f(X)    /* Code executed by the task */
16 
17 END
18 
19 task_g(i) /* Task name */
20 
21 i = 1..N-1 /* Execution space declaration for parameter i */
22 
23 : y(i) /* Task must be executed on the node where y(i) is stored
     */
24 
25   /* Task reads x(i) from task_f(i)... */
26   R X  <- X task_f(i)
27   /* ... y(i-1) from task_g(i-1)... */
28   R Y1 <- (i > 1) ? Y2 task_g(i-1) : y(i-1)
29   /* ... and sends y(i) to task_g(i+1) */
30   W Y2 -> (i < N-1) ? Y2 task_g(i+1)
31 
32 BODY
33 
34   Y2 = g(X, Y1) /* Code executed by the task */
35 
36 END
```

Figure 11: Simple example of a parallel version of the sequential code in Figure 6(a) using a PTG model with PaRSEC.

In Figure 11, we illustrate the use of a PTG model using the JDF language with PaRSEC by implementing a parallel version of the simple example shown in Figure 6(c). In this JDF code we have the representation for the two types of task, `task_f` and `task_g`, that each have with one parameter, `i`. This parameter is defined on the range `1..N-1` where `N` is defined at the beginning of the JDF code and initialised when the DAG is instantiated. As illustrated by the diagram in Figure 7(a), the dataflow for `task_f` contains two edges that are expressed in lines 10 and 12 of the JDF code using the symbols `<-` for the input dataflow and `->` for the output dataflow. Similarly, the three edges for the dataflow for task_g are expressed in lines 26, 28 and 30. Note that the last two dataflows are conditional and depend on the value of i. The instance of task_g associated with the parameter i=1 reads the data denoted by Y1 in memory because it is the first task to touch this data. The following instances, however, will get this data from the previously executed tasks. The kernel associated with each task is contained between the BODY and END keywords. As we mentioned previously, multiple kernels, one for each type of architecture for example, can be associated with these tasks. In our example we only provide the implementation for CPUs. In a distributed-memory context, the node selected to execute a given task depends on the memory location of the associated data. In our example the data affinity is defined with the instructions on lines 7 and 23 and depends on data x for task_f and data y for task_g. Note that during the execution, data might not be up-to-date on a given node in which case PaRSEC handles the transfer from one node to another before executing the task.

6 The SpLLT solver

In this section, we present the sparse Cholesky solver SpLLT that we developed for this deliverable. In this solver, we implemented the DAG-based sparse Cholesky factorization using the StarPU and PaRSEC runtime systems as well as the standard OpenMP leading to three different versions of the code. On the one hand we have SpLLT-STF that is based on a STF model and implemented with StarPU and OpenMP and on the other hand we have SpLLT-PTG based on a PTG model and implemented with PaRSEC.

6.1 The STF implementation using OpenMP and StarPU

In this section we describe the implementation of our DAG-based Cholesky solver using the STF parallel programming model presented in Section 4.1. We developed two different versions of our code; the first using OpenMP, and the second using the StarPU runtime system.

A pseudo-code for our solver is shown in Figure 12. Following the sequential algorithm shown in Figure 5, it consists in a bottom-up traversal of the assembly tree where at each node the tasks for the factorization and update operations are submitted to the runtime system. The kernels used in the tasks are the same as those presented in Section 3. Note that the task submission is done using a right-looking scheme meaning that every node in the tree must be allocated and partitioned before the submission of the numerical tasks. In addition, the alloc task is executed sequentially because we need to allocate the data structures and partition the supernodes in order to submit the numerical tasks.

As explained in Section 4.1, when using the STF model to submit a task, we need to provide the access mode along with the data so that the runtime system can ensure the sequential consistency of the parallel algorithm. For this reason, in the submission of factorize tasks, the diagonal block blk(k,k) is associated with a read-write access mode indicating that the kernel will read and modify this block when computing the Cholesky factor of the block. Similarly, because the solve operations need the diagonal block to compute the subdiagonal blocks of the factors, we have to indicate that the diagonal block is read when submitting the solve by associating it with a read-only access mode. With this information, the runtime detects the dependencies between the factorize and solve tasks and allows the parallel execution of the solve tasks within a block-column.

```
forall nodes snode in post-order
   ! allocate data structures
   call alloc(snode)
   ! initianlize node structure
   call submit(init, snode:W)
end do

forall nodes snode in post-order

  ! factorize node
  do k=1..n in snode
    call submit(factorize, snode:R, blk(k,k):RW)

    do i=k+1..m in snode
        call submit(solve, blk(k,k):R, blk(i,k):RW)
    end do

    do j=k+1..n in snode
       do i=k+1..m in snode
          call submit(update, blk(j,k):R, blk(i,k):R, blk(i,j):RW
   )
       end do
    end do

    forall ancestors(snode) anode
      do j=k+1..p(anode) in snode
         do i=k+1..q(anode) in snode
            call submit(update_btw, anode:R, blk(j,k):R, blk(i,k)
   :R,
                 a_blk(rmap(i), cmap(j)):RW)
         end do
      end do
    end do

  end do
end do
```

Figure 12: Pseudo-code for the sparse Cholesky factorization using an STF model presented in Section 4.1.

In order to ensure that the supernode is initialized before the factorization starts, we use a symbolic handle called `snode` and pass it to the `init` tasks using a write access mode. Then we also pass it to the `factorize` tasks in read access mode. Because all the subsequent factorization tasks in a supernode depend on the first `factorize` task, we thus guarantee that the numerical task cannot start before the supernode is initialized. For the same reason, the `update_btw` task takes the `anode` handle as input with read access mode because it modifies a block in an ancestor node and the task should not be executed before the node is initialized. The specific nature of this symbolic handle is that it represents a set of blocks instead of a single block.

One issue arises with the dependency detection of the `update` tasks that are applied to a given block. This task takes as input two blocks L_{ik} and L_{jk} and performs the operation

$$L_{ij} = L_{ij} - L_{ik} L_{jk}^T$$

on a third block L_{ij}. These update operations are commutative in infinite precision arithmetic. However, when two `update` tasks are performed on the same block, the runtime system detects that these tasks modify the same data and will ensure that the order of execution follows the order of submission. With StarPU it is possible to use the `STARPU_COMMUTE` flag to avoid this unnecessary dependency that potentially limits the parallelism. This flag indicates that operations performed by a kernel are commutative. The OpenMP standard still does not provide such a functionality.

The STF code that is presented in Figure 12 is independent of the runtime system used for the implementation. In practice only the implementation of the `submit` routines are specific to the runtime system. This illustrates the fact that the expression of the algorithm is strictly separated from the task scheduling and data management. An example of the implementation of this `submit` routine in the StarPU version is given in Figure 13(a), and its equivalent in the OpenMP version is given in Figure 13(b). In this example we show the submission of the solve tasks. In the OpenMP version, blocks are identified using data pointers and these pointers are associated with a data access when submitting a task. It is thus necessary to allocate the blocks before being able to submit the tasks that use these blocks. In the case of StarPU, blocks are associated with a handle that is set up in the `alloc` routine. Tasks are then associated with this handle instead of using a pointer as is done by OpenMP. There are several advantages associated with the use of a handle. For example, StarPU is capable of detecting when data are written for the first time and will perform the allocations using the information contained in the handle. We don't use this feature for the allocation of blocks, but we use it for the management of scratch memory needed by the `update_btw` task. We do not include this in the pseudo-code for the sake of clarity.

```c
struct starpu_codelet cl_solve_block = {
 .where = STARPU_CPU,
 .cpu_funcs = {spllt_solve_block, NULL},
};

starpu_task_insert(
      &cl_solve_block,
      STARPU_R, blk_kk_handle,
      STARPU_RW, blk_ik_handle,
      STARPU_PRIORITY, prio,
      0);
```

(a) StarPU code.

```fortran
!$omp task firstprivate(m, n)
!$omp    & depend(in:bc_kk%c) &
!$omp    & depend(inout:bc_ik%c)

call spllt_solve_block(m, n, bc_kk%c, bc_ik%c)

!$omp end task
```

(b) OpenMP code.

Figure 13: Routines used to submit solve tasks with StarPU (a) and OpenMP (b).

Note that the efficiency of these submission routines may be critical to the performance of the execution and, as shown in our tests, the submission of tasks in the DAG may be sometimes a limiting factor for the performance. This happens when there is a large number of tasks and the task granularity is small. In such cases, especially when the number of resources increases, the unrolling of the DAG may be too slow to feed all the resources and it therefore bounds the execution time. In that respect, the partition parameter nb may influence the performance because a small value for this parameter increases the number of tasks in the DAG and therefore the overhead associated with task submission and task management.

As mentioned in Section 5, although StarPU provides a complete API for designing new scheduling policies, it also provides a number of common scheduling strategies. In our experiment we choose the LWS (Locality Work Stealing) scheduler which takes into account data locality and priority and provides good results on multicore architectures. In the scheduler, tasks are prioritized according to a priority value provided by the user. In our case, the priority depends on the position of the task in the DAG. For example, the `factorize` tasks are given the highest priority because they lie on the critical path. With OpenMP, although setting priorities is in the 4.5 standard, we did not use a compiler for implementing this version in our experiment so that we cannot give priorities to the tasks.

6.2 Tree pruning strategy

In order to reduce the impact of the time spent in unrolling the DAG when using a STF model to implement the factorization, we use a *pruning* of the assembly tree. This is similar to the pruning strategy used in the `qr_mumps` solver [2] and consists in grouping small nodes at the bottom of the tree into subtrees that are processed in serial. Our algorithm, inspired by [14], is done by traversing the nodes with a top-bottom tree traversal starting from the root node and balancing the workload across the subtrees until we reach a desired load balance while preserving enough parallelism to feed all the resources. The workload is represented by the amount of floating-point operations required to process the subtree which is a rather accurate estimate of the computational cost for processing a subtree. In Figure 14 we illustrate the pruning of a simple elimination tree where the grey nodes belong to a subtree rooted at the dark-grey nodes lying on the dashed red line. The advantage of such pruning is that it reduces the number of tasks to be handled by the runtime system and thus the overhead associated with it. In addition, the tasks that are removed from the DAG correspond to the smaller granularity tasks. On the other hand, this algorithm decreases the amount of parallelism in the DAG which might become too low on the smaller problems when the number of resources is large.

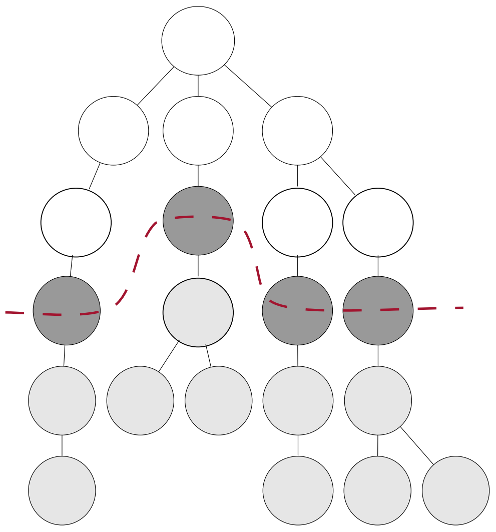


Figure 14: Illustration of the tree pruning strategy used by `SpLLT`.

```
task_factor_block(diag_idx)

  diag_idx = 0..(ndiag-1) /* Index of diag block*/

  /* Global block index */
  id_kk = %{return get_diag_blk_idx(diag_idx);%}
  /* Index of current supernode */
  snode = %{return get_blk_node(id_kk);%}
  /* Index of block in current block-column*/
  last_blk = %{return get_last_blk(id_kk);%}
  /* id of prev diag block */
  prev_id_kk = %{return get_diag_blk_idx(diag_idx-1);%}
  /* Number of input contribution for current block*/
  dep_in_count = %{return get_dep_in_count(id_kk);%}
  /* Number of out contribution for current block*/
  dep_out_count = %{return get_dep_out_count(id_kk);%}

  : blk(id_kk)

  RW bc_kk <- (is_first(snode, id_kk) && dep_in_count==0) ?
                bc task_init_block(id_kk)
           <- (is_first(snode, id_kk) && dep_in_count > 0) ?
                bc_ij task_update_btw(id_kk, dep_in_count)
           <- (!is_first(snode, id_kk)) ?
                bc_ij task_update_block(diag_idx-1, prev_id_kk+1,
   prev_id_kk+1)
           -> (id_kk == last_blk) ?
                blk(id_kk) : bc_kk task_solve_block(diag_idx, (
   id_kk+1)..last_blk)
           -> (dep_out_count > 0) ?
                bc task_update_btw_aux(id_kk, 1..dep_out_count)

  ; FACTOR_PRIO /* Task priority */

BODY

  factor_block(bc_kk);   /* Cholesky factorization kernel */

END
```

Figure 15: Extract of the JDF code for the sparse Cholesky factorization in which the `task_factor_block` task type is shown.

6.3 The PTG implementation using PaRSEC

In the study [1] the authors investigated the implementation of a sparse factorization using a PTG model. This implementation was based on a two-level approach where the processing of the assembly tree and the multifrontal matrix factorization are coded in two different JDFs which split the exploitation of tree-level and node-level parallelism. Even if this hierarchical approach facilitated the construction of the dataflow representation, it incorporated unnecessary synchronisation, prevented the exploitation of inter-node parallelism and therefore drastically impacted the scalability of the code. For this reason, in `SpLLT`, we choose to express the whole DAG in one JDF file that includes all the task types and dependencies. This enables the exploitation of all the parallelism available in the DAG but increases the complexity of the dataflow representation.

In Figure 15 we present an extract of this JDF code with the description of the `task_factor_block` task type associated with the `factor_block` kernel. As shown in Figure 4(b), the factorization DAG contains one `task_factor_block` task for every block on the diagonal of our matrix. We thus associate this task type with the parameter `diag_idx`, ranging from `0` to `ndiag-1`, where `ndiag` is the total number of diagonal blocks. A task instance manipulates a single block, referred to as `bc_kk`, in a RW mode as it computes its Cholesky factor. The instructions on lines 6-16 retrieve the information on the current supernode and block being processed that is necessary to determine the data flow associated with the task. This information is obtained from the structure of the problem which is built during the analysis phase.

The instruction on line 18 indicates to the runtime system the location where the task should be executed. In this example, the notation `blk(id_kk)` means that the task should be executed on the compute node where the block is stored. This location depends on the data distribution given to the runtime system by the user. Note that in our implementation the data distribution is straightforward as we focus on multicore machines for which the data is located on one compute node.

The input dataflow, expressed on lines 20-25, is split into three different cases: if the processed block corresponds to the first block in the current supernode, then either the supernode has received a contribution from a descendent supernode and thus the data is received from an `task_update_btw` task or we read the data from the initialization task of type `task_init_block`; if the current block is not the first in the supernode, then the data necessary comes from an `update_block` task resulting from the factorization of previous block-column. The output dataflow, expressed on lines 26-29, shows that the data is sent to several tasks: the `task_solve_block` tasks that compute the factors on the subdiagonal blocks and the `task_update_btw_aux` tasks that update the blocks in the ancestor nodes. Note that, for every block, we need the number of contributions received (`dep_in_count`) and sent (`dep_out_count`) to other blocks located in other supernodes. This information is computed during the analysis phase by traversing the assembly tree and is added to the data structure associated with each block.

Whenever a task is completed during the execution of the DAG, the data associated with this task become available and the runtime system checks the output dataflow for tasks which are ready for execution. The new ready tasks are then scheduled using the task priority `FACTOR_PRIO` provided on line 31 and as well as data locality information.

7 Experimental results with the `SpLLT` solver

We tested our `SpLLT` solver on a multicore machine equipped with two Intel(R) Xeon(R) E5-2695 v3 CPUs with fourteen cores each (twenty eight cores in total). Each core, clocked at 2.3 GHz and equipped with AVX2, has a peak of 36.8 Gflop/s corresponding to a total peak of 1.03 Tflop/s in real, double precision arithmetic. The code is compiled with the GNU compiler (`gcc` and `gfortran`), the BLAS and LAPACK routines from by the Intel MKL v11.3 library. We used version 1.3 of StarPU and used the latest development version (version v1.1.0-2771-g7a4cb0d) of the PaRSEC runtime system.

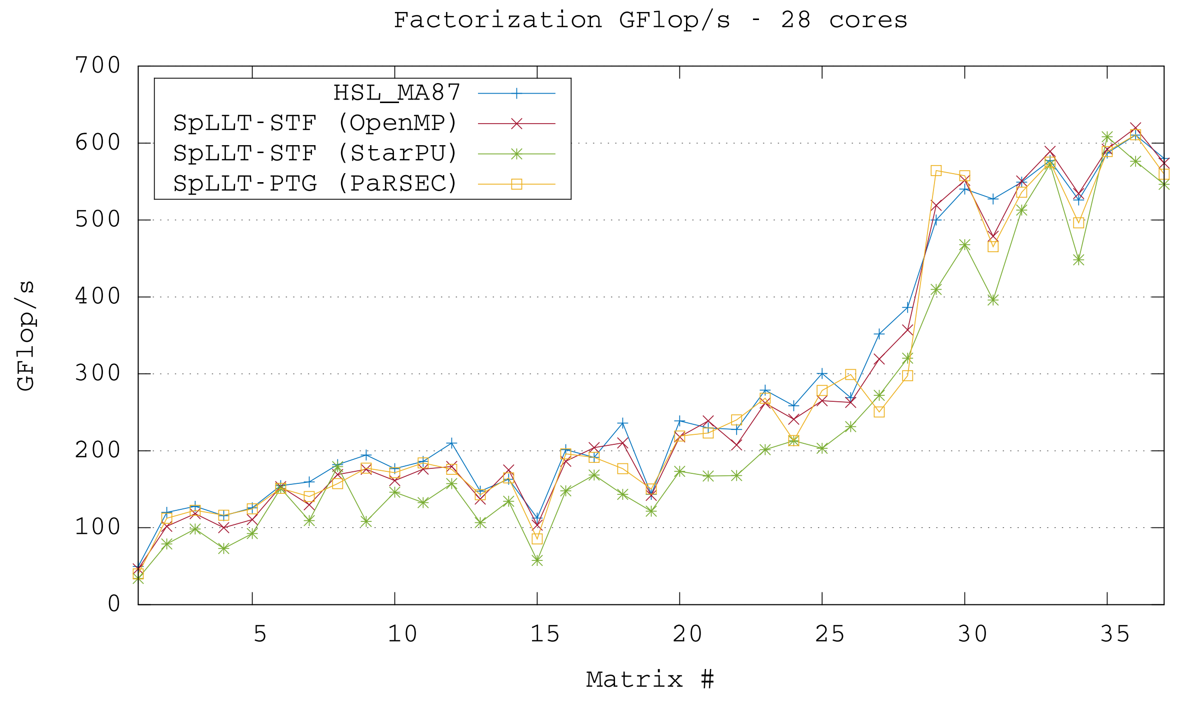


Figure 16: Performance results for `SpLLT`, using OpenMP, StarPU and PaRSEC, compared against the `HSL_MA87` solvers on 28 cores for the test matrices presented in Table A.1.

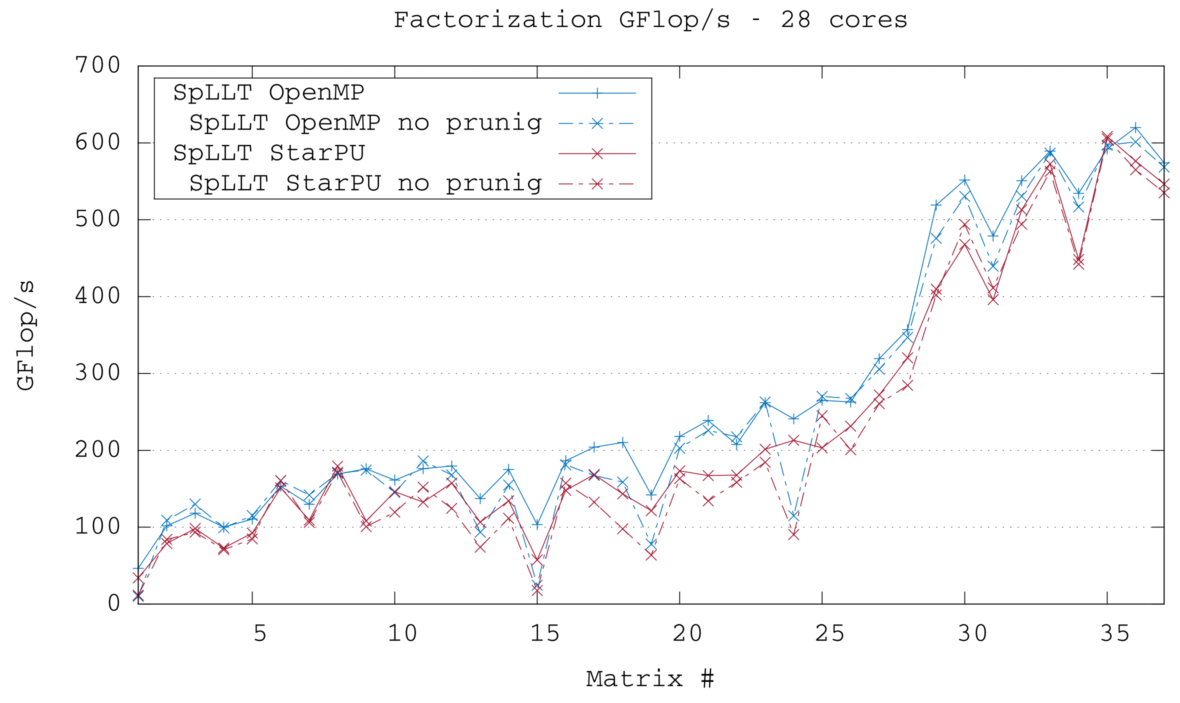


Figure 17: Effect of tree pruning on `SpLLT` performance using a STF model (OpenMP and StarPU) for the test matrices presented in Table A.1.

In Figure 16 we show the performance of the `SpLLT` solver using the STF model (with the StarPU and OpenMP runtime systems) and the PTG model (with the PaRSEC runtime system). In addition we present the performance results obtained with our reference solver `HSL_MA87`. In our experiments, the OpenMP and PaRSEC versions match the performance of `HSL_MA87` whereas the StarPU version is generally slower. In Figure 17 we show the effect of the tree pruning strategy on the performance. We can see that, although the tree pruning has generally a small effect on the performance, it significantly improves the performance on the matrices #15, #19 and #24 for the OpenMP and StarPU codes.

8 Availability of the `SpLLT` software

The `SpLLT` code including the latest development version is available on the NLAFET GitHub project in the repository NLAFET/SpLLT that can be found at `https://github.com/NLAFET/SpLLT`. The compilation is handled by CMake[4] tools and the Makefile for building `SpLLT` can be created using the `cmake` command as shown in Figure 18. By default this command will build the sequential version of the code and the `-DRUNTIME` option can be used to build the parallel version. This option selects one of the supported runtime systems: `-DRUNTIME=OMP` for the OpenMP version, `-DRUNTIME=StarPU` for the StarPU version and `-DRUNTIME=Parsec` for the PaRSEC version.

```
# Get latest development version from GitHub
git clone https://github.com/NLAFET/SpLLT
# Move to source directory
cd SpLLT
# Create build directory
mkdir build
cd build
# Create Makefile with cmake command. The -DRUNTIME option can be
# used to select a runtime system.
cmake <path-to-source> -DRUNTIME=<StarPU|OMP|Parsec>
# Build SpLLT software
make

```

Figure 18: Compilation of the `SpLLT` solver.

9 Pivoting strategies for LDL^T factorization

In this section we focus on the indefinite case and, as we explained in the introduction, we need to include pivoting to ensure the stability of the factorization, contrary to the positive definite case. This pivoting adds complexity to the algorithms by introducing synchronisations and increasing floating-point operations and memory footprint in the factorization. We describe the A Posteriori Threshold Pivoting (APTP) strategy developed in conjunction with this deliverable to improve the performance and scalability of the LDL^T factorization while ensuring stability.

[4]`https://cmake.org/`

9.1 Threshold partial pivoting

Threshold Partial Pivoting (TPP) is the method of choice for selecting pivots for performing the LDL^T factorization of supernode as illustrated in Equation 3. It uses a fixed stability criterion u ($0 < u \leq 1$) and ensures that all entries in the factor L are such that $|l_{ij}| \leq u^{-1}$. The factorization algorithm is backward stable and gives the possibility to compute the solution with a good accuracy [6].

9.2 A posteriori threshold pivoting

In this section, we introduce a LDL^T factorization algorithm relying a new the pivoting strategy called A posteriori Threshold Pivoting (APTP) . It is based on a 2D partitioning of matrices into square blocks (similarly to the Cholesky algorithm presented in Section 3) thus increasing the parallelism compared to the state-of-the-art TPP strategy. This new algorithm is based the following two techniques:

Fail-in-place approach which consists in keeping the failed columns in place and handling them at the end of the factorization. In this case, these columns must be updated during the factorization.

Speculative execution which consists in speculatively running a task assuming that no numerical issues have occurred in other tasks that might affect the current one. This requires doing a backup of entries and implementing a backtracking strategy if numerical instability is detected.

Using these two principles it is possible to design a task-based LDL^T algorithm with better scalability than the TPP strategy. Speculative execution makes it possible to process block columns in parallel using 2D partitioning. In addition, the fail-in-place approach avoids communications because we do not permute rows and columns outside their blocks. However, using speculative execution comes at the cost of storing backups for the tasks that are speculatively executed. Also, the fail-in-place strategy has two main drawbacks: first, we need a fallback strategy for handling the failed columns; secondly, keeping the failed columns up to date introduces small granularity tasks in the DAG. As a result, using speculative execution and a fail-in-place approach improves the scalability of the LDL^T factorization so long as the number of failed pivots remains relatively low.

The LDL^T factorization with APTP strategy is shown as Algorithm 1. Tasks are presented as subroutine calls that update (i.e. have an *inout* dependency on) arguments before the semicolon, and have an input dependency on all blocks after it. The variable $nelim_j$ is special as we do not express task dependencies involving it, but instead use atomic updates. The kernels are as follows:

Factor($A_{jj}, nelim_j$) computes the LDL^T factorization of a block on the diagonal as follow:

1. Stores a backup of block A_{jj};
2. Performs a pivoted factorization $A_{jj} = P_j L_{jj} D_{jj} L_{jj}^T P_j$, where P_j is a permutation;
3. Initialises $nelim_j$ to the block size of A_{jj}.

Algorithm 1 A posteriori threshold pivoting LDL^T factorization.

```
1: for $j = 1$ to nblk do
2:    Factor( $A_{jj}$, $nelim_j$ )
3:    for $i = 1$ to $j - 1$ do
4:       ApplyT( $A_{ji}$, $nelim_j$; $A_{jj}$ )
5:    end for
6:    for $i = j + 1$ to mblk do
7:       ApplyN( $A_{ij}$, $nelim_j$; $A_{jj}$ )
8:    end for
9:    Adjust($nelim_j$; All blocks $A_{:j}$ and $A_{j:}$ )
10:   for $k = 1$ to $j - 1$ do
11:      for $i = k$ to $j - 1$ do
12:         UpdateTT( $A_{ik}$; $A_{ji}$, $A_{jk}$, $nelim_j$ )
13:      end for
14:      for $i = j$ to mblk do
15:         UpdateNT( $A_{ik}$; $A_{ij}$, $A_{jk}$, $nelim_j$ )
16:      end for
17:   end for
18:   for $k = j$ to nblk do
19:      for $i = k$ to mblk do
20:         UpdateNN( $A_{ik}$; $A_{ij}$, $A_{kj}$, $nelim_j$ )
21:      end for
22:   end for
23: end for
```

There are several options for performing the pivoted factorization of the block A_{jj} including using complete pivoting or an implementation of TPP. In our implementation, a two-level approach involving an outer block size *nb* (a parameter available to the user) and an inner block size *nbi* (fixed at compile time). APTP is performed in parallel using the outer block size, and recurses to perform APTP in serial with the smaller inner block size. In the inner block APTP then uses complete pivoting for full blocks and a simple implementation of TPP for partial blocks.

ApplyN($A_{ij}, nelim_j$**;** A_{jj}**)** applies the LDL^T factorization from the diagonal block A_{jj} to a subdiagonal block A_{ij}. This is done by:

1. Stores a backup of block A_{ij};
2. Performs the operation $L_{ij} = P_j A_{ij} (L_{jj} D_{jj})^{-T}$;
3. Find the least column $nelim_{ij}$ in L_{ij} that contains a failed entry i.e. $l_{pq} > u^{-1}$ and performs the atomic reduction $nelim_j = \min(nelim_j, nelim_{ij})$

ApplyT($A_{ji}, nelim_j$**;** A_{jj}**)** applies the LDL^T factorization from the block A_{jj} to the failed entries in the block A_{ji} on its left. This is done by:

1. Store a backup of block A_{ji};
2. Performs the operation $L_{ji} = (L_{jj} D_{jj})^{-1} A_{ji} P_j^T$ on the columns corresponding to the failed pivots;
3. Find the least row $nelim_{ji}$ in L_{ji} that contains a failed entry i.e. $l_{pq} > u^{-1}$ and performs the atomic reduction $nelim_j = \min(nelim_j, nelim_{ji})$.

Note that if there are no failed entries in block A_{ji} then this operation becomes a no-op.

Adjust($nelim_j$**; All blocks** $A_{:j}$ **and** $A_{j:}$**)** ensures $nelim_j$ has been atomically reduced in all the blocks of $A_{:j}$ and $A_{j:}$. Decrements $nelim_j$ down by one if we would otherwise accept only the first column of a 2×2 pivot.

UpdateNN(A_{ik}**;** A_{ij}**,** A_{kj}**,** $nelim_j$**)** performs the update operations on a block that is on the right of the eliminated block column j.

1. If block A_{ik} is in the same column as the eliminated column, i.e. $k = j$, then restores the failed entries from the backup;
2. Performs the update operation $A_{ik} = A_{ik} - L_{ij} D_{jj} L_{kj}^T$. If the task operates on column j only the uneliminated entries (that have just been restored) are updated, otherwise the whole block is updated.

UpdateNT(A_{ik}**;** A_{ij}**,** A_{jk}**,** $nelim_j$**)** performs the update operations on a block that is on the bottom left of the eliminated block column j.

1. If block A_{ik} is in the same row as the eliminated column i.e. $i = j$, then restores the failed entries from the backup;
2. Performs the update operation: $A_{ik} = A_{ik} - L_{ij} D_{jj} L_{jk}$ on the uneliminated entries.

UpdateTT(A_{ik}; A_{ji}, A_{jk}, $nelim_j$) performs the update operations on a block that is on the top left of the eliminated block column j. The operation $A_{ik} = A_{ik} - L_{ji}^T D_{jj} L_{jk}$ is performed on the uneliminated entries.

After completion of this algorithm, failed entries are permuted to the back of the matrix. At this stage the entries can be refactorized (they have likely been updated since they failed) using either APTP or TPP, or passed directly to the parent. Figure 19 visualises the data structure part-way through the factorization in which some pivots have failed.

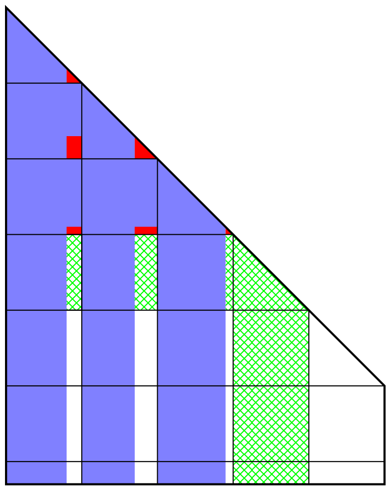


Figure 19: Part-way through the execution of Algorithm 1. Iterations $j = 1, 2, 3$ have completed, and iteration $j = 4$ is about to begin. Blue entries have been eliminated. Red entries have failed to be eliminated. Green entries indicate uneliminated entries considered for elimination on iteration $j = 4$.

10 The SSIDS solver

The first version of SSIDS, released in 2014, was described in [16] and was meant to run on a single GPU. In this section we describe the new release of SSIDS that adds a CPU implementation to the solver. It implements a multifrontal LDL^T factorization with the APTP strategy introduced in Section 9.2. As we show in the experimental section, this new code performs better than other state-of-the-art direct solvers and is more reliable in terms of accuracy.

The CPU version of SSIDS has been implemented using Fortran and C++, and the parallelization of the factorization is done with OpenMP. This implementation makes use of the `task` construct and `depend` clause and follows a STF model as presented in Section 4.1. In Figure 20, we show a pseudo-code corresponding to the factorization of the assembly tree in SSIDS. In this pseudo-code, the main loop iterates over the nodes in a post-ordering and each node factorization is contained in a task that depends on the tasks for the node factorization of the children nodes. The factorization of the multifrontal matrices at each node is handled via the three main routines:

assemble_pre()

```
#pragma omp taskgroup
{
  for (n = 0; n < nnodes; n++) {

    #pragma omp task
            depend(in: children(n))
            depend(inout: node(n))
    {
      // Allcate memory for both fully-summed columns and
      // contribution block and assemble contribution
      // into fully-summned columns
      assemble_pre(n, children(n));
      // Factorize fully-summed columns
      factor(n);
      // Assemble contribution into contribution block
      assemble_post(n, children(n));
    }
// Wait for the completion of tasks
}
```

Figure 20: Pseudo-code for the multifrontal sparse LDL^T factorization implemented in SSIDS.

1. Allocates memory for both the fully summed columns and the contribution block. Note that we use two different memory allocators: we use a *stack* allocator for the fully-summed columns as they are never moved whereas the contribution blocks are allocated using a *buddy system* allocator. The advantage of using a buddy system allocator is that it is faster at allocating and deallocating memory but this is done at the expense of requiring more space than some other methods;
2. Assembles contribution from children into the fully summed columns only.

factor() Factorizes the fully summed columns and calculates the contribution block. This routines implements the LDL^T factorization with APTP detailed in Section 9.2.

assemble_post() Assembles contribution from children into the contribution block.

Note that, in this case a node factorization might start only when all of its children nodes have been factorized. As a consequence there is no inter-level parallelism unlike the case of the Cholesky factorization. This limits the parallelism in the assembly tree.

11 Experimental results with the SSIDS solver

We present experimental results with the SSIDS solver on a the same multicore machine described in Section 7. It is equipped with two Intel(R) Xeon(R) E5-2695 v3 CPUs with fourteen cores each (twenty eight cores in total). Each core, clocked at 2.3 GHz and equipped with AVX2, has a peak of 36.8 Gflop/s corresponding to a total peak of 1.03 Tflop/s in real, double precision arithmetic. The code is compiled with the GNU compiler (gcc and gfortran) version 6.1.0[5], the BLAS and LAPACK routines are provided by the Intel MKL v11.3 library. The SSIDS solver is included in the SPRAL library and we used the version with the git tag PAPER_20160922.

	Factor (s)			ndelays			bwerr		
#	SSIDS	MA97	PARDISO	SSIDS	MA97	PARDISO	SSIDS	MA97	PARDISO
1	0.08	0.06	0.05	997	1125	-	9.20e-15	1.29e-14	4.67e-15
2	0.07	0.08	0.03	401	381	-	3.41e-15	7.60e-15	8.13e-16
3	0.09	0.12	0.05	2312	2640	-	1.10e-14	3.91e-15	4.45e-14
4	0.10	0.10	0.08	3464	3434	-	6.53e-15	6.24e-14	1.92e-15
5	0.12	0.11	0.12	5031	4910	-	2.27e-14	4.66e-14	3.80e-15
6	0.16	0.16	0.17	0	0	-	3.16e-14	4.17e-14	3.93e-14
7	0.11	0.15	0.12	0	0	-	1.41e-15	1.62e-15	3.88e-03
8	0.08	0.12	0.08	1531	1353	-	8.09e-15	1.06e-14	5.22e-16
9	0.10	0.12	0.07	1685	1523	-	5.95e-15	1.40e-14	5.36e-16
10	0.24	0.63	0.24	78	196	-	1.49e-14	1.30e-14	4.71e-17
11	0.19	0.26	0.26	13	89	-	3.63e-15	1.55e-15	1.31e-16
12	0.15	0.23	0.29	1	0	-	4.74e-14	1.35e-13	4.59e-14
13	0.21	0.31	0.25	27	389	-	1.40e-13	3.66e-13	3.38e-15
14	0.29	1.52	0.33	26	69	-	2.04e-11	3.11e-11	1.07e-06
15	0.31	0.40	0.16	1235	1199	-	2.41e-15	8.36e-16	1.53e-13
16	0.34	0.40	0.17	1599	1633	-	1.66e-15	2.03e-13	5.12e-15
17	0.18	0.25	0.24	30	345	-	4.65e-16	1.69e-16	1.95e-16
18	0.19	0.27	0.22	7	103	-	5.49e-16	1.81e-15	1.71e-16
19	0.42	0.60	0.27	2818	2831	-	3.91e-15	1.27e-13	3.07e-14
20	0.72	1.17	0.35	4405	4295	-	4.36e-15	2.36e-13	1.24e-14
21	0.59	2.52	0.59	60	102	-	1.41e-12	1.91e-12	5.19e-12
22	1.04	5.21	1.04	79	190	-	6.01e-10	5.69e-10	8.16e-10
23	1.67	8.18	1.50	164	554	-	7.33e-09	4.37e-09	1.40e-07

Table 1: Timing, number of delayed pivots (ndelay) and backward errors (bwerr) obtained with SSIDS when running matrices from the hard indefinite test set on a Haswell compute node.

For testing SSIDS, we used two test sets of matrices. The first set, shown in Table A.2, and referred to as *Easy Indefinite* is a set of matrices that require few delayed pivots. No scaling is performed on these problems. The second test set, shown in Table A.3 and referred to as *Hard Indefinite* contains matrices that require significant scaling and pivoting. Matrices are scaled and ordered using a matching-based ordering and scaling.

In our experiments we compared SSIDS with several sparse direct solvers for symmetric indefinite systems:

HSL_MA86 is a solver from the HSL library. It implements a task-based supernodal method and uses a block-column partitioning of supernodes. We used version 2.4.0.

HSL_MA97 is a solver from the HSL library. It is based on a multifrontal method and uses a recursive parallel factorization at the node level. We used version 2.4.0.

[5] The flags "-g -O2 -march=native" were used for the compilation.

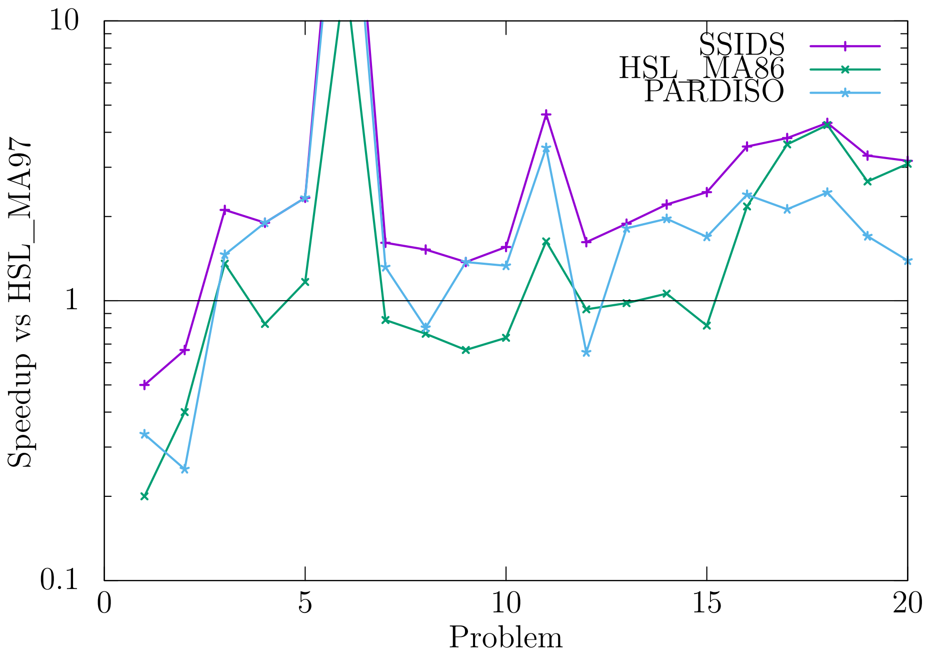


Figure 21: Timing results obtained with `SSIDS` when running the easy indefinite test set on a Haswell compute node.

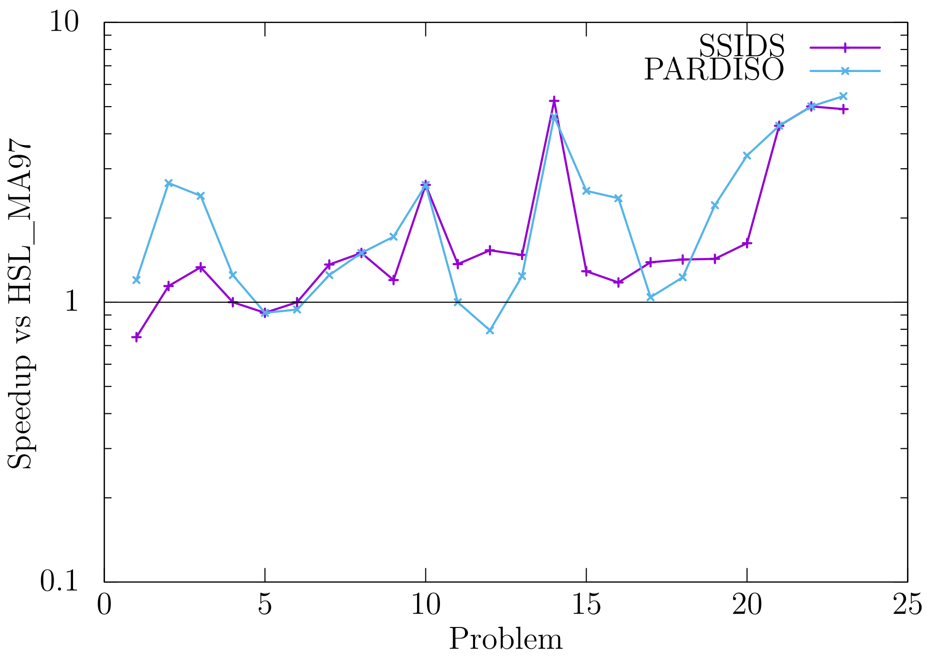


Figure 22: Timing results obtained with `SSIDS` when running the hard indefinite test set on a Haswell compute node.

`PARDISO` is a solver included in the MKL library. It implements a supernodal method and uses a Bunch-Kaufman and a static pivoting scheme.

The performance results that we obtained on the multicore machine are presented in Figure 21 for the easy indefinite set and in Figure 22 and Table 1 for the hard indefinite set. We observe from these experiments that `SSIDS` systematically performs better than the other solvers in the case of easy indefinite matrices. In the case of hard indefinite problems, SSIDS performs better than HSL_MA97 expect in rare cases. In comparison with PARDISO, SSIDS performs worse for roughly half of the problems (10 of them) from our the hard indefinite set, whereas in the other half (13 problems) it performs better. However, as we see in Table 1 SSIDS always gives small backward errors compared to PARDISO which, for some of the test problems gives very inaccurate results even though iterative refinement is used.

12 Availability of the SSIDS software

The SSIDS code including the latest development version is available as part of the SPRAL library that can be found on the GitHub repository at https://github.com/ralna/spral. The compilation process is handled by GNU Autotools packages[6] and an example of the instructions for compiling SSIDS is sown in Figure 23.

```
# Get latest development version from github and run dev scripts
git clone --depth=1 https://github.com/ralna/spral.git
cd spral
./autogen.sh

# Build and install library
BUILDDIR=build; mkdir $BUILDDIR; cd $BUILDDIR
../configure --with-metis='-L/path/to/metis -lmetis'
make
sudo make install # Optional

# Link against library
cd /path/to/your/code
gfortran -o myprog myobj.o -lspral -lmetis -lblas
```

Figure 23: Compilation of the SSIDS solver.

13 Conclusions and future work

This deliverable introduces two sparse direct solvers for solving linear systems: a Cholesky solver SpLLT for positive-definite systems and an LDL^T solver SSIDS for solving indefinite systems. In our approach, we exploit task-based algorithms for performing matrix factorization that we implement using a runtime system. In the experimental sections we show that our approach offers good performance compared to the state-of-the-art solvers and, in the case of indefinite systems, we show that using the new APTP strategy, we don't need to sacrifice performance for reaching good accuracy of the computed solution. Using a runtime system for implementing a sparse direct solver yields several advantages. It facilitates the implementation of parallel algorithms because we are using a high-level API, and it also improves the maintainability and portability of the codes.

In the context of SpLLT our current work consists in porting the StarPU version of the code to GPU-based heterogeneous architectures. The StarPU runtime system provides

[6]https://www.lrde.epita.fr/~adl/autotools.html

many features for targeting such systems including the possibility of associating GPU kernels to the tasks and of implementing new scheduling strategies which is crucial for achieving performance on such platforms.

Although `SSIDS` is very competitive with the state-of-the-art LDL^T solvers, it suffers from the lack of features in the OpenMP library compared to other runtime systems. In particular it remains quite difficult to exploit GPUs in this context. For this reason, we are currently developing a new code called `SpLDLT` with the same purpose as `SSIDS` but implemented with the StarPU runtime system. It relies on the same APTP strategy for the LDL^T factorization.

Finally, in our work we only focused on symmetric systems so far that represent a large proportion of applications for our solvers. Symmetric systems constitute the most difficult case among symmetrically structured problems because we exploit symmetry for reducing memory footprint and computational costs. In addition, it is challenging to preserve both numerical stability and symmetry in the case of indefinite systems. However, our algorithms and pivoting strategy can be adapted to the case of non-symmetric problems without major difficulties so we do not anticipate any problems in generating algorithms and code for this case.

References

[1] E. Agullo, G. Bosilca, A. Buttari, A. Guermouche, and F. Lopez, *Exploiting a Parametrized Task Graph model for the parallelization of a sparse direct multifrontal solver*, in Euro-Par 2016: Parallel Processing Workshops, Grenoble, France, Aug. 2016.

[2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, *Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems*, ACM Trans. Math. Softw., 43 (2016), pp. 13:1–13:22.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, *Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects*, Journal of Physics: Conference Series, 180 (2009), p. 012037.

[4] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.

[5] ——, *Algorithm 837: Amd, an approximate minimum degree ordering algorithm*, ACM Trans. Math. Softw., 30 (2004), pp. 381–388.

[6] C. Ashcraft, R. G. Grimes, and J. G. Lewis, *Accurate symmetric indefinite linear equation solvers*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 513–561.

[7] 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, 23 (2011), pp. 187–198.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, *Parsec: Exploiting heterogeneity to enhance scalability*, Computing in Science and Engineering, 15 (2013), pp. 36–45.

[9] A. Buttari, *Fine-grained multithreading for the multifrontal QR factorization of sparse matrices*, SIAM Journal on Scientific Computing, 35 (2013), pp. C323–C345.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Comput., 35 (2009), pp. 38–53.

[11] M. Cosnard and M. Loi, *Automatic task graph generation techniques*, in System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on, vol. 2, Jan 1995, pp. 113–122 vol.2.

[12] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 973–996.

[13] I. S. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. Matrix Anal. Appl., 27 (2005), pp. 313–340.

[14] G. A. Geist and E. Ng, *Task scheduling for parallel sparse cholesky factorization*, Int. J. Parallel Program., 18 (1990), pp. 291–314.

[15] A. George and J. W. H. Liu, *An automatic nested dissection algorithm for irregular finite element problems*, SINUM, 15 (1978), pp. 1053–1069.

[16] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, *A sparse symmetric indefinite direct solver for GPU architectures*, ACM Trans. Math. Softw., 42 (2016), pp. 1:1–1:25.

[17] J. D. Hogg, J. K. Reid, and J. A. Scott, *Design of a multicore sparse cholesky factorization using dags*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3627–3649.

[18] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. V. Zee, *The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations*, J. Parallel Distrib. Comput., 72 (2012), pp. 1134–1143.

[19] J. W. H. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Trans. Math. Softw., 11 (1985), pp. 141–153.

[20] W. F. Tinney and J. W. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proceedings of the IEEE, 55 (1967), pp. 1801–1809.

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 set for positive-definite matrices and their characteristics without node amalgamation. n is the matrix order, $nz(A)$ represents the number entries in the matrix A, $nz(L)$ represents the number of entries the factor L and *Flops* correspond to the operation count for the matrix factorization.

#	Name	n	nz(A)	nz(L)	Flops
		(10^3)	(10^6)	(10^6)	(10^9)
1	Oberwolfach/t2dal	4.26	0.02	0.28	0.02
2	GHS_indef/dixmaanl	60.00	0.18	1.58	0.05
3	Oberwolfach/rail_79841	79.84	0.32	4.43	0.33
4	GHS_indef/dawson5	51.54	0.53	5.69	0.90
5	Boeing/bcsstk39	46.77	1.07	9.61	2.66
6	Boeing/pct20stif	52.33	1.38	12.60	5.63
7	GHS_indef/copter2	55.48	0.41	12.70	6.10
8	GHS_indef/helm2d03	392.26	1.57	33.00	6.16
9	Boeing/crystk03	24.70	0.89	10.90	6.26
10	Oberwolfach/filter3D	106.44	1.41	23.80	8.71
11	Koutsovasilis/F2	71.50	2.68	23.70	11.30
12	McRae/ecology1	1000.00	3.00	72.30	18.20
13	Cunningham/qa8fk	66.13	0.86	26.70	22.10
14	Oberwolfach/gas_sensor	66.92	0.89	27.00	22.10
15	Oberwolfach/t3dh	79.17	2.22	50.60	70.10
16	Lin/Lin	256.00	1.01	126.00	285.00
17	GHS_indef/sparsine	50.00	0.80	207.00	1390.00
18	PARSEC/Ge99H100	112.98	4.28	669.00	7070.00
19	PARSEC/Ga10As10H30	113.08	3.11	690.00	7280.00
20	PARSEC/Ga19As19H42	133.12	4.51	823.00	9100.00

Table A.2: Easy Indefinite test set. Statistics as reported by the analyse phase of SSIDS with default settings, assuming no delays.

#	Name	n	nz(A)	nz(L)	Flops
		(10^3)	(10^6)	(10^6)	(10^9)
1	TSOPF/TSOPF_FS_b39_c7	28.22	0.37	2.61	0.26
2	TSOPF/TSOPF_FS_b162_c1	10.80	0.31	1.89	0.36
3	QY/case39	40.22	0.53	3.87	0.40
4	TSOPF/TSOPF_FS_b39_c19	76.22	1.00	7.28	0.75
5	TSOPF/TSOPF_FS_b39_c30	120.22	1.58	11.10	1.10
6	GHS_indef/cont-201	80.59	0.24	7.12	1.11
7	GHS_indef/stokes128	49.67	0.30	6.35	1.16
8	TSOPF/TSOPF_FS_b162_c3	30.80	0.90	6.37	1.41
9	TSOPF/TSOPF_FS_b162_c4	40.80	1.20	7.32	1.43
10	GHS_indef/ncvxqp1	12.11	0.04	3.56	2.52
11	GHS_indef/darcy003	389.87	1.17	23.20	3.01
12	GHS_indef/cont-300	180.90	0.54	17.20	3.58
13	GHS_indef/bratu3d	27.79	0.09	7.49	4.72
14	GHS_indef/cvxqp3	17.50	0.07	6.33	5.27
15	TSOPF/TSOPF_FS_b300	29.21	2.20	13.40	6.92
16	TSOPF/TSOPF_FS_b300_c1	29.21	2.20	13.50	7.01
17	GHS_indef/d_pretok	182.73	0.89	24.80	7.42
18	GHS_indef/turon_m	189.92	0.91	24.70	7.60
19	TSOPF/TSOPF_FS_b300_c2	56.81	4.39	27.00	14.10
20	TSOPF/TSOPF_FS_b300_c3	84.41	6.58	40.50	21.40
21	GHS_indef/ncvxqp5	62.50	0.24	22.90	24.30
22	GHS_indef/ncvxqp3	75.00	0.27	39.30	63.70
23	GHS_indef/ncvxqp7	87.50	0.31	51.00	101.00

Table A.3: Hard indefinite matrices test set. Statistics as reported by the analyse phase of SSIDS with default settings, using matching-based ordering, assuming no delays.