Science & Technology Facilities Council

Technical Report
RAL-TR-2016-016

Experiments with sparse Cholesky using a sequential task-flow implementation

I Duff, J Hogg, F Lopez

November 2016

Enquiries concerning this report should be addressed to:

RAL Library
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358-6254

NLAFET-WorkingNote-7

Experiments with sparse Cholesky using a sequential task-flow implementation

Iain Duff, Jonathan Hogg, and Florent Lopez

ABSTRACT

We describe the development of a prototype code for the solution of large sparse symmetric positive definite systems that is efficient on parallel architectures. We implement a DAG-based Cholesky factorization that offers good performance and scalability on multicore architectures. Our approach uses a runtime system to execute the DAG. The runtime system plays the role of a software layer between the application and the architecture and handles the management of task dependencies as well as the task scheduling. In this model, the application is expressed using a high-level API, independent of the hardware details, thus enabling portability across different architectures. Although widely used in dense linear algebra, this approach is nevertheless challenging for sparse algorithms because of the irregularity and variable granularity of the DAGs arising in these systems. We assess the ability of two different Sequential Task Flow implementations to address this challenge: one implemented with the OpenMP standard, and the other with the modern runtime system StarPU. We compare these implementations to the state-of-the-art solver HSL_MA87 and demonstrate comparable performance on a multicore architecture.

Keywords: sparse Cholesky, SPD systems, runtime systems, StarPU, OpenMP

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

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

Correspondence to: florent.lopez@stfc.ac.uk
This work is supported by the NLAFET Project funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement 671633.

November 1, 2016

Contents

1 Introduction

We investigate the use of a runtime system for implementing a sparse Cholesky decomposition for solving the linear system

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

where A is a large sparse symmetric positive-definite matrix. We focus on exploiting multicore architectures that are now ubiquitous in high performance computing. DAG-based algorithms have been shown to be extremely efficient in terms of performance and scalability and have been widely used in dense linear algebra as in the PLASMA software package [2]. They have been adapted to sparse algorithms in, for example, the `HSL_MA87` solver [14] implementing a supernodal Cholesky factorization, and the `qr_mumps` solver [8] implementing a multifrontal QR method.

The traditional approach for implementing a task-based solver includes the development of an ad hoc scheduler which relies on the knowledge of the algorithm to manage synchronisations and enforce dependencies between processes and is implemented using a low-level multithreading library such as pthreads (POSIX threads). This approach, however, lacks portability as it is developed for a specific target architecture. It can then be costly in terms of programming effort to port the code to different architectures and adapt it to emerging ones. This would be particularly challenging when targeting heterogeneous architectures equipped with different types of processing units with different capabilities such as GPU-accelerated multicore systems. Instead, in this work, we explore an alternative approach based on the use of a runtime system that consists of a software layer between the architecture and the application. The application is implemented using a high-level API provided by the runtime system, and low level details such as data consistency and task scheduling are delegated to the runtime system.

Several dense linear algebra software packages have been built using this approach such as DPLASMA [6], which uses the PaRSEC [7] runtime system and Chameleon which supports several runtime systems including StarPU [5] and PaRSEC. Both packages are designed for distributed memory systems equipped with accelerators. For sparse linear algebra, however, relatively few libraries have adopted this approach. Two examples of runtime-based sparse solvers are `qr_mumps` [1], that implements a multifrontal QR method, and `PaSTIX` [13] that implements a supernodal Cholesky method. Compared to dense linear algebra, the difficulty with employing this approach in the sparse case stems from the fact that the DAGs are extremely irregular with a large variability of task granularity and irregularities in the dependency pattern.

We focus on a Sequential Task Flow (STF) programming model for expressing the DAG. This model offers a simple way to define the parallel code from the sequential one. Although this work is closely related to that used in the StarPU version of `PaSTIX`, the expression of dependencies differs between these two solvers. In the `PaSTIX` solver, dependencies are explicitly expressed and therefore it does not take advantage of the STF features available in StarPU. We show that our codes, implemented with a task-based runtime system supporting the STF model, can lead to an implementation of a sparse matrix factorization that is as efficient as a state-of-the-art solver on a multicore system.

2 Task-based sparse Cholesky factorization

We first describe the supernodal Cholesky factorization method that we use for solving sparse symmetric positive-definite systems. In particular, we will focus on a DAG-based variant of this algorithm that has been proven to be efficient on multicore architectures in the HSL_MA87 solver [14]. The factorization is one of the three main phases for solving a linear system that includes an analysis phase preceding the factorization, and a solve phase following it. We specifically focus on the factorization because it corresponds to the most computationally expensive phase.

The supernodal Cholesky method is a factorization algorithm for sparse matrices where the input matrix A is decomposed as

$$PAP^T = LL^T, \tag{2.1}$$

where P is a permutation matrix and the factor L is a lower triangular matrix. The factorization is then followed by a solve phase for computing x 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 [3, 4, 18, 20] or Nested Dissection [12] or variants of these methods.

The dependencies between the coefficients in the factor L during the factorization can be expressed by a tree structure called an *elimination tree* where each node of this tree represents a column in the factor. In order to increase the efficiency of operations by exploiting Level 3 BLAS routines, the elimination tree is transformed into an *assembly tree* where columns having a similar nonzero pattern are amalgamated into a dense matrix that is referred to as a *nodal* matrix or *supernode*.

The factorization is effected by traversing the assembly tree in a topological order and performing two main operations at each supernode: a dense Cholesky factorization of the current supernode and an update of the ancestor supernodes using the resulting factors. The update operations may be performed using either *right-looking* updates where ancestors nodes are updated as soon as the nodal factorization is done, or *left-looking* updates where the current supernode is updated just before being factorized. We use the software package HSL_MC78 [15] to compute the assembly tree during the analysis phase.

There are two main sources of parallelism that can be exploited in the assembly tree: *tree-level* and *node-level* parallelism. Tree-level parallelism is due to the fact that supernodes located in separate branches of the tree can be processed independently and node-level parallelism is exploited when supernodes are large enough to be efficiently processed in parallel. One approach for the parallelization of the supernodal algorithm consists in exploiting these two levels of parallelism independently. For example, several processes can be used to handle the factorization of independent supernodes in the tree and these processes can exploit node-level parallelism by using multithreaded routines. Note that, with this strategy, there is a synchronisation point between the processing of a node and the processing of its children therefore potentially limiting parallelism. Instead, in our work, we follow the approach proposed in [14] where supernodes are partitioned into square blocks of order `nb` and operations are performed on these blocks. Figure 2.1 shows a simple assembly tree that consists of three supernodes where the dashed lines represent the block partitioning of supernodes. Figure 2.2 represents the DAG associated with the factorization of the tree in Figure 2.1.

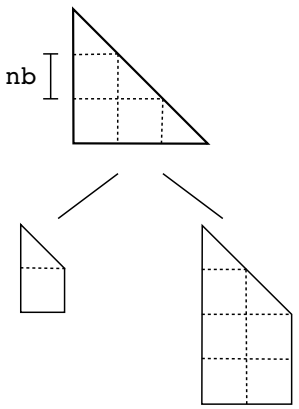

Figure 2.1: Simple assembly tree with three supernodes partitioned into square blocks of order `nb`.

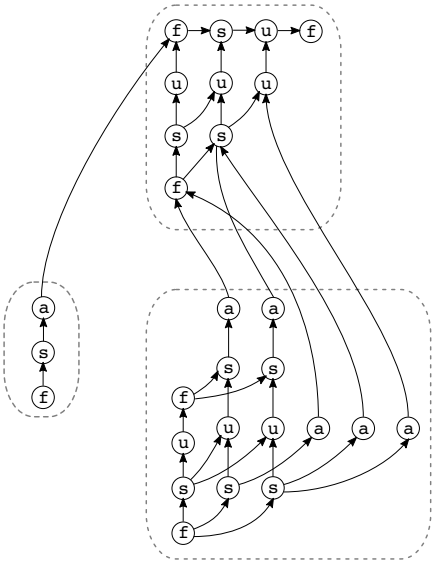

Figure 2.2: DAG corresponding to the factorization of the tree in Figure 2.1.

As shown in the DAG represented in Figure 2.2 there are several kernels involved in the factorization of the supernodes:

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 2.2, replaces the elimination tree for expressing the dependencies during the computation of the factors. Note that when exploiting the node and tree 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 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 2.3. Note that this is the sequential algorithm that is used as a basis for the implementation of 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`.

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.

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

forall nodes snode in post-order

  ! factorize node
  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 2.3: Pseudo-code for the sequential version of the task-based sparse Cholesky factorization.

3 Sequential Task Flow parallel programming model

In this work we exploit a *Sequential Task Flow (STF)* programming model for the implementation of a parallel task-based Cholesky factorization on top of a runtime system. In this 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 [2] and FLAME [16] 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.

```
for (i = 1; i < N; i++) {
  x[i] = f(x[i]);
  y[i] = g(x[i], y[i-1]);
}
```

Figure 3.1: Simple example of a sequential code.

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

Figure 3.2: STF code corresponding to Figure 3.1 example.

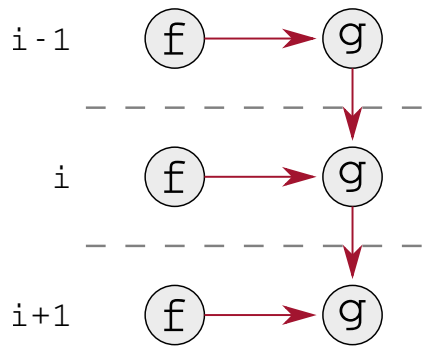

Figure 3.3: DAG corresponding to the sequential code presented in Figure 3.1.

As an example, we consider the sequential code in Figure 3.1 for which the corresponding DAG is shown in Figure 3.3. Based on a STF model, the parallel version of this code is illustrated in Figure 3.2. In the sequential code, the two functions f and g manipulate arrays x and y. The STF code is obtain 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 introduced in [9]. In this model, task dependencies are explicitly encoded with the dataflow of each task so that the whole DAG can be expressed in a compact format.

4 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 board 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 3. In this work we use an OpenMP implementation of our Cholesky solver and show advantages of using this 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.

```
#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 4.1: Simple example of a parallel version of the sequential code in Figure 3.1 using a STF model with OpenMP.

We present in Figure 4.1 an example of a parallel implementation for the sequential code in Figure 3.1 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 4.1 and in order to introduce the features provided by the StarPU API, we show in Figure 4.2 an example of a StarPU-based implementation for the simple example presented in Figure 3.1. The task submission is done through the `starpu_insert_task` 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. In our example the codelet `g_cl` in line 10 describes a task that can be executed on a CPU and a CUDA device (`STARPU_CPU | STARPU_CUDA`) respectively with the kernels `g_cpu_func` and `g_cuda_func`. The data handles, declared in line 21 in our example, 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 the memory nodes during the 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`.

Both OpenMP and StarPU implementations rely on a dynamic scheduler for scheduling the ready tasks during the 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 4.3 where

```
/* Codelet definition for kernel f */
struct starpu_codelet f_cl =
  {
    .where = STARPU_CPU,
    .cpu_funcs = { f_cpu_func },
    .nbuffers = 1
  };

/* Codelet definition for kernel g */
struct starpu_codelet g_cl =
  {
    .where = STARPU_CPU | STARPU_CUDA,
    .cpu_funcs = { g_cpu_func },
    .cuda_funcs = { g_cuda_func },
    .nbuffers = 3
  };

starpu_init(); /* initialization of StarPU */

/* declaration of data handles */
starpu_data_handle_t x_handle[N], y_handle[N];

for (i = 0; i < N; i++) {
  starpu_variable_data_register(&x_handle[i], STARPU_MAIN_RAM,
                                (uintptr_t) &x[i], sizeof(double));
  starpu_variable_data_register(&y_handle[i], STARPU_MAIN_RAM,
                                (uintptr_t) &y[i], sizeof(double));
 }

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

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

  starpu_insert_task(&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();

starpu_shutdown(); /* shutdown StarPU */
```

Figure 4.2: Simple example of a parallel version of the sequential code in Figure 3.1 using a STF model with StarPU.

the scheduler is placed between the runtime core where the DAG is built and the workers which can, for example, 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

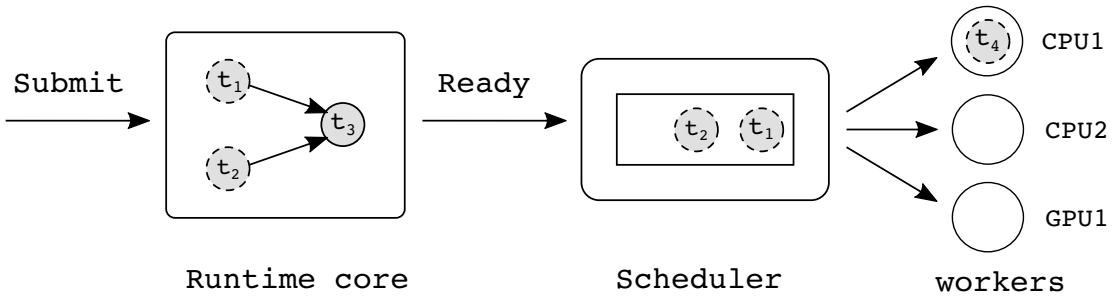

Figure 4.3: Illustration of the dynamic scheduling strategy of tasks in the runtime system.

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 Parallelization of a task-based Cholesky factorization using an STF programming model

In this section we describe the implementation of our DAG-based Cholesky solver using the STF programming parallel model presented in Section 3. We developed two different versions of our code; first that using OpenMP, and secondly that using the StarPU runtime system.

A pseudo-code for our solver is shown in Figure 5.1. Following the sequential algorithm shown in Figure 2.3, 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 2. 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 3, 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.

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

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

```
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 5.1: Pseudo-code for the sparse Cholesky factorization using a STF model presented in Section 3.

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 5.1 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 5.2, and its equivalent in the OpenMP version is given in Figure 5.3. 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.

```
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);
```

Figure 5.2: Submission routine used for the solve tasks in the StarPU code.

```
!$omp task firstprivate(m, n)
!$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
```

Figure 5.3: Submission routine used for the solve tasks in the OpenMP code.

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 4, 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 do not have a compiler for this version so that we cannot choose either the scheduler or give priorities to the tasks.

6 Strategy of parallelization and task scheduling in MA87

We use the HSL solver HSL_MA87 as a benchmark reference when studying the performance of our code. In this section, we briefly introduce the strategy used by this reference solver for the parallelization of a DAG-based sparse Cholesky factorization. The representation of the DAG in HSL_MA87 is similar to that used in the PTG model in that the DAG is implicitly represented and progressively unrolled during execution following the output dependencies of completed tasks. However, the scheduler is hand-coded and is designed specifically for the solver relying on knowledge of the algorithm. This is implemented within OpenMP but was coded using a version of OpenMP without the tasking capabilities of version 4.0 of the standard.

During the analysis phase, every block is associated with a counter representing the number of tasks that manipulate the associated data. The factorization starts by processing blocks associated with a counter equal to zero and when a given thread finishes the execution of a task, it performs two operations: it decreases the counter associated with blocks involved in the computation and puts into the scheduler the tasks that become ready for execution as a result of the completion of the current task.

Thus the strategy used by HSL_MA87 has much in common with the PTG model in that it follows the data-flow associated with a task to submit subsequent tasks to the scheduler. This approach offers several

advantages over the STF model and can impact the performance as we will show in our experimental results in Section 7. First, unlike the STF model where one single master thread is involved in unrolling, every thread is responsible for submitting tasks to the scheduler which means that the DAG is built in parallel and the cost for setting up the DAG is shared between all resources. In addition, the only tasks that are instantiated are either the tasks being executed or the tasks ready for execution that are stored in the scheduler. Instead, in our solver, every task in the DAG is instantiated and submitted to the runtime system and stored in memory which produces a higher memory footprint for representing the DAG.

The scheduling strategy used in HSL_MA87 is similar to the LWS scheduler presented in the previous section where the scheduler tries to enforce data reuse and takes into account task priorities which depend on their position in the DAG.

7 Experimental results

In this study, the tests were made 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 are provided by the Intel MKL v11.3 library and we used the version 1.3 of the StarPU runtime system.

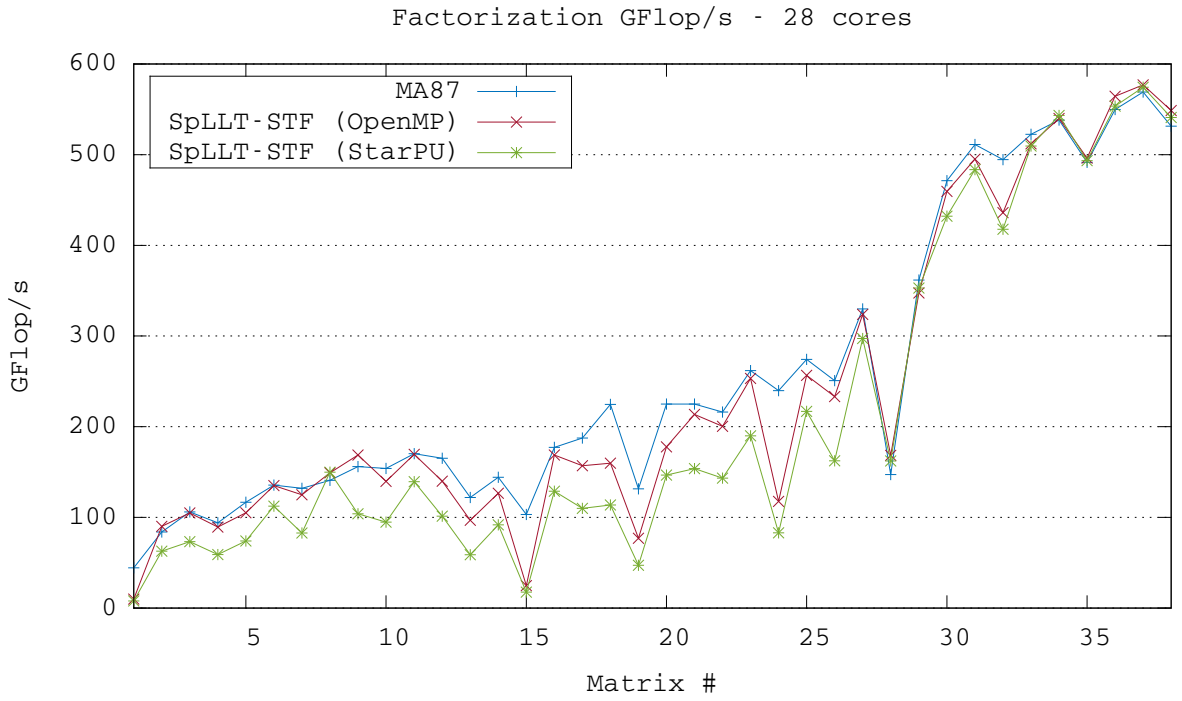

Figure 7.1: Performance results for both OpenMP and StarPU versions of the SpLLT and HSL_MA87 solvers on 28 cores for the test matrices presented in Table A.

In our experiments we use a set of matrices taken from the University of Florida Matrix Collection [10]. From this collection, we selected a set of symmetric positive-definite matrices with a wide range of applications and sparsity structures. They are listed in Table A along with their orders and number of entries. In this table, we also indicate the number of entries in the factor L and the flop count for the factorization when using the nested-dissection ordering METIS [17]. Note that, in this table, matrix

#	Name	MA87		spLLT			
		MA87		OpenMP (gnu)		StarPU	
		nb	factor. (s)	nb	factor. (s)	nb	factor. (s)
1	Schmid/thermal2	1024	0.391	1024	1.742	512	2.215
2	Rothberg/gearbox	256	0.253	384	0.236	512	0.339
3	DNVS/m_t1	256	0.206	256	0.208	1024	0.298
4	Boeing/pwtk	768	0.249	768	0.262	768	0.396
5	Chen/pkustk13	256	0.218	256	0.242	384	0.344
6	GHS_psdef/crankseg_1	256	0.233	256	0.234	384	0.281
7	Rothberg/cfd2	256	0.242	256	0.256	384	0.386
8	DNVS/thread	256	0.236	256	0.223	384	0.222
9	DNVS/shipsec8	256	0.253	256	0.234	384	0.380
10	DNVS/shipsec1	256	0.245	256	0.270	384	0.398
11	GHS_psdef/crankseg_2	256	0.267	256	0.268	384	0.326
12	DNVS/fcondp2	256	0.294	384	0.347	384	0.479
13	Schenk_AFE/af_shell3	256	0.437	512	0.550	512	0.906
14	DNVS/troll	256	0.381	384	0.434	512	0.599
15	AMD/G3_circuit	256	0.607	768	2.534	768	3.561
16	GHS_psdef/bmwcra_1	256	0.339	256	0.356	512	0.466
17	DNVS/halfb	256	0.370	256	0.442	384	0.631
18	Um/2cubes_sphere	256	0.321	384	0.451	512	0.634
19	GHS_psdef/ldoor	256	0.620	512	1.058	768	1.726
20	DNVS/ship_003	256	0.361	384	0.457	512	0.554
21	DNVS/fullb	256	0.445	256	0.469	384	0.651
22	GHS_psdef/inline_1	256	0.659	384	0.711	768	0.995
23	Chen/pkustk14	256	0.557	256	0.576	512	0.768
24	GHS_psdef/apache2	256	0.710	768	1.448	512	2.053
25	Koutsovasilis/F1	384	0.776	384	0.829	768	0.981
26	Oberwolfach/boneS10	256	1.102	256	1.185	768	1.702
27	ND/nd12k	384	1.452	384	1.479	768	1.611
28	JGD_Trefethen/Trefethen_20000	768	4.233	512	3.700	768	3.843
29	ND/nd24k	384	5.350	512	5.570	768	5.485
30	Janna/Flan_1565	384	7.722	512	7.921	768	8.419
31	Oberwolfach/bone010	384	7.117	768	7.350	768	7.525
32	Janna/StocF-1465	768	8.337	768	9.455	768	9.869
33	GHS_psdef/audikw_1	384	10.419	768	10.630	768	10.680
34	Janna/Fault_639	384	14.392	768	14.350	768	14.260
35	Janna/Hook_1498	512	17.019	384	16.860	768	16.960
36	Janna/Emilia_923	768	23.221	384	22.620	768	23.060
37	Janna/Geo_1438	768	29.654	768	29.250	1024	29.380
38	Janna/Serena	768	52.830	384	51.170	768	51.900

Table 7.1: Factorization times (seconds) obtained with MA87 and SpLLT with both OpenMP and StarPU versions. The factorizations were run with the block sizes `nb=(256, 384, 512, 768, 1024)` on 28 cores and `nemin=32`.

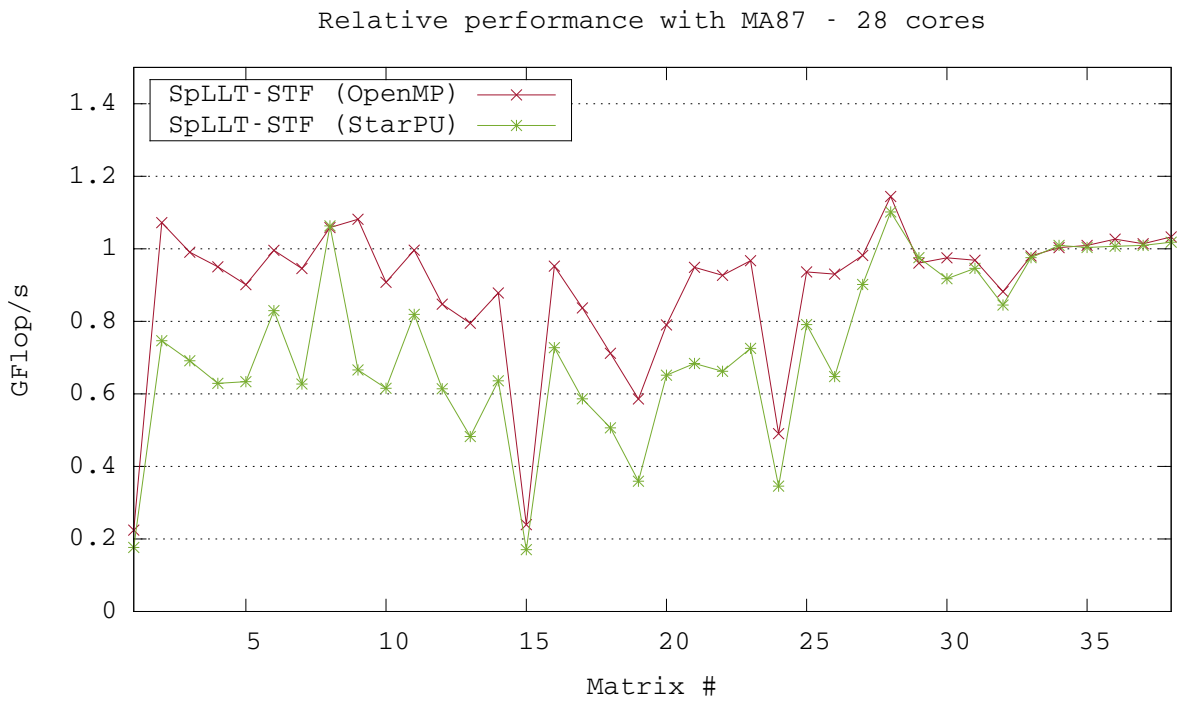

Figure 7.2: Performance results obtained with SpLLT, OpenMP and StarPU relative to the performance obtained with HSL_MA87.

characteristics are obtained without node amalgamation. This means that the number of entries in L as well as the operation count is minimized. However, in our experiments we use node amalgamation to obtain better efficiency of operations at the cost of an increase in the operation count and the number of entries in L. Node amalgamation is controlled by a parameter `nemin` used during the analysis phase. The elimination tree is traversed using a post-order and, when a node is visited, it is merged with its parents if the column count in both nodes is lower than `nemin` or if the merging generates no additional fill-in in L. In our experiments, we use the analysis routine HSL_MC78 and set the `nemin` value to 32. This corresponds to a good trade-off between sparsity and efficiency of floating-point computation.

The choice for the parameter `nb` in the parallel execution is not trivial as it impacts several aspects of the execution. Although a small value for this parameter increases the number of tasks in the DAG and thus the parallelism, it also reduces the performance of the Level 3 BLAS routines used in the tasks. The optimal value for this parameter is thus a trade-off between a sufficient amount of parallelism to feed the resources and good kernel efficiency. In addition, as we have seen in Section 5, the parameter `nb` influences the overhead for managing the tasks because when the number of tasks in the DAG increases, it also increases the time for handling the DAG including the task submission, dependency detection and scheduling. Finally, the optimal value for `nb` depends on a large number of parameters such as processing unit capabilities and number of resources and cannot be easily determined without a precise performance model for the application which is extremely difficult to establish. Instead we empirically determine a good `nb` value for each problem by running multiple tests on a range of values for `nb`. We used `(256, 384, 512, 768, 1024)` as the range in these experiments.

The best factorization times obtained with SpLLT and HSL_MA87 for every problem listed in Table A are reported in Table 7.1 along with the corresponding value of `nb` used in the run. The Gflop/s rates corresponding to the best factorization times are presented in Figure 7.1. Additionally, Figure 7.2 illustrates the relative performance of the SpLLT codes compared to HSL_MA87.

For matrices from #27 to #38, corresponding to the bigger problems of our test set, the performance

behaviour of the OpenMP and StarPU versions of SpLLT is very similar and comparable to that obtained with our reference solver HSL_MA87. On smaller problems, with factorization times generally smaller than a second, it appears that the OpenMP version gives better results than the StarPU one, with results competitive with HSL_MA87 except on a few problems. Matrices # 1 and # 15, for example, give extremely poor results with both versions of our code. We will explore reasons for this in the next section.

#	Name	DAG stats		SpLLT (StarPU)		
		# task	time / task avg. (ms)	task insert (s)	factor. (s)	time (s)
1	Schmid/thermal2	166695	0.071	2.213		2.215
2	Rothberg/gearbox	19392	0.417	0.326		0.339
8	DNVS/thread	7481	0.746	0.155		0.222
13	Schenk_AFE/af_shell3	65195	0.250	0.906		0.906
15	AMD/G3_circuit	290235	0.154	3.558		3.561
19	GHS_psdef/ldoor	126786	0.230	1.724		1.726
24	GHS_psdef/apache2	163804	0.264	2.052		2.053
32	Janna/Flan_1565	226486	1.007	3.452		8.419
35	Janna/Hook_1498	639060	0.867	8.441		16.960
38	Janna/Serena	795367	1.857	10.920		51.900

Table 7.2: DAG information along with the times (seconds) spent by the master thread to submit all the tasks to the runtime system on a subset of our test matrices.

In order to understand the behaviour of the STF-based implementations and to identify the main limiting factors for performance on smaller problems, we gathered information on the DAG and measured the time spent for submitting tasks during execution on a subset of our test matrices. These results are shown in Table 7.2 where we put the number of tasks in the DAGs, the average time spent in tasks for each problem and the task insert times. As expected, the DAG size generally gets bigger with the problem size except for the two matrices # 1 and # 15 for which we noticed very low performance. Consequently these matrices have the lowest granularity which causes relatively more time to be spent for task submission in the runtime system. This behaviour occurs for the matrices # 1, # 2, # 13, # 15, # 19 and # 24 where the time for submitting the DAG is close to or equal to the factorization time. This means that the DAG unrolling is too slow for feeding the resources and is a constraint on the factorization time and therefore limits the scalability of the code. On the matrices with a bigger granularity of tasks in the DAG, the task submission time is no longer an issue. We see this on matrices # 8, # 32, # 35 and # 38 where our code gives comparable results to HSL_MA87. As explained in Section 6, in HSL_MA87 the DAG is progressively unrolled by all the threads, which means that the cost for the task submission is distributed among the resources and therefore does not constrain the execution time. Finally, we see that, when using a STF model, it is crucial for the runtime system to keep the overhead associated with the task management as low as possible in order to achieve good performance. Although OpenMP seems to be less costly in terms of overheads than StarPU on smaller problems, it should be noted that, as explained in Section 4, the latter offers more features and flexibility than the former. For this reason StarPU appears to be more adapted to handle large problems where tasks have a sufficiently large granularity in the DAG.

7.1 Performance analysis

In this section we present a detailed performance analysis on a subset of the test matrices. This performance analysis, introduced in [1] and [19], allows us to identify the limiting factors for achieving performance on our machine. The idea of the approach is to measure the time spent by workers in the different parts of the execution such as: the time spent in tasks, denoted by $t_t(p)$ where p corresponds to the number of

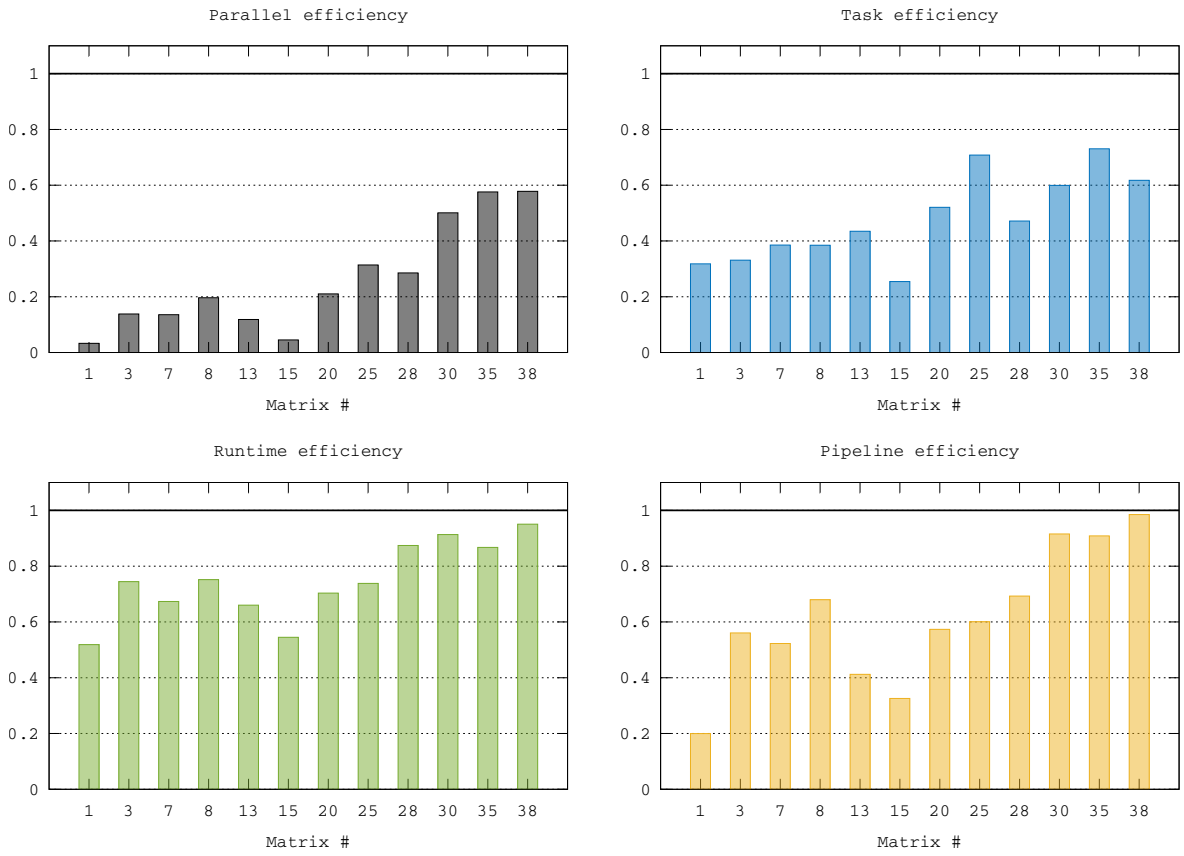

Figure 7.3: Efficiency analysis for a subset of the test matrices including the parallel efficiency (top-left), the task efficiency (top-right), the runtime efficiency (bottom-left) and the pipeline efficiency (bottom-right).

workers, the time spent in the runtime system, denoted by $t_r(p)$, and the time spent idle denoted by $t_i(p)$. In addition we define the parallel efficiency of the code as:

$$e(p) = \frac{t(1)}{t(p) \times p}$$

where $t(p)$ corresponds to the factorization time when using p resources. Then, using the times previously introduced and measured during the execution, we decompose the parallel efficiency as a product of efficiencies as:

$$e(p) = \frac{\tilde{t}_t(1)}{t_t(p) + t_r(p) + t_i(p)} = \overbrace{\frac{t_t(1)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_i(p)}}^{e_p},$$

where each efficiency corresponds to a specific identified effect that impacts the performance:

- e_t is the *task efficiency* which measures the impact of the loss of data locality in the kernels when going from serial to parallel execution. It also measures the impact of data partitioning which creates parallelism but decreases the granularity of operations and thus the efficiency of floating-point computation.
- e_r is the *runtime efficiency* corresponding to the overhead induced by the runtime system for handling the DAG and scheduling tasks on the machine.
- e_p is the *pipeline efficiency* which measures the resource utilization during the parallel execution. This quantity depends on the shape of the DAG and the quality of the task scheduling.

We show the results in Figure 7.3. Similarly to what we found in the previous section, the parallel efficiency tends to increase when the problem size increases. By computing e_t, e_r and e_p, we can identify the limiting factors for the parallel efficiency. First, as expected, the pipeline efficiency is lower for smaller matrices as their associated DAG generally contains less tasks and potentially less parallelism. In addition, as explained in the previous section, the time spent by the master thread in the task submission limits the parallelism in some cases. This explains, for example, why we obtain such a low pipeline efficiency on matrices #1 and # 15. As for the pipeline efficiency, the task efficiency is lower on smaller matrices. This is because when there is little parallelism in the DAG, there are few ready tasks in the scheduler and there is thus a large amount of work-stealing between the workers in the scheduler and thus a large amount of data movement. Moreover, the processors used in these experiments are equipped with Turbo Boost technology that is capable of dynamically increasing the clock rate when the number of cores involved is low. This is, for example, the case when running the solver in a serial mode and so, by accelerating the execution of tasks in serial mode, this feature decreases the task efficiency when the number of resources increases. Finally, we can see from the runtime efficiency that smaller problems perform worse on the runtime system and are associated with a greater runtime overhead. This is the case for matrices # 1 and # 15, for example.

Note that, in the case of the OpenMP version of our solver, it is not possible to perform such a detailed analysis as OpenMP does not have the functionality for measuring the time spent in the runtime system.

7.2 Reducing the impact of DAG unrolling

In order to reduce the impact of the time spent in unrolling the DAG for the factorization of the STF codes, we considered two different approaches: moving to a *nested STF* model and using *tree pruning*.

The idea of the *nested STF* model is to use tasks to submit other tasks in order to distribute the cost for task submission. This is done by splitting the main submission loop presented in Figure 5.1 into the two codes shown in Figures 7.4 and 7.5. The first code corresponds to the main submission loop which is executed in serial by the master thread, and the latter corresponds to a task executed by a worker and responsible for submitting the node factorization task. This approach differs from a pure STF model as not all the tasks are submitted by the master thread and is similar to a *recursive* model that we refer to as the *nested STF* model.

```
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

    call submit(insert_factorize_node, snode:R)
end do
```

Figure 7.4: Main submission routine executed by the master thread. This performs both the initialisation and factorization tasks for the nodes in the assembly tree.

The strategy however suffers from several limitations. In the case of OpenMP, nested tasks as well as tasks that are submitted by other tasks belong to different execution contexts and, in the standard, it is not possible to express dependencies between tasks that are not in the same context. In the strategy presented above, it is therefore not possible to exploit inter-node parallelism which greatly reduces parallelism. In the case of the StarPU version, it is possible to implement the proposed strategy and have the same fine-grained parallelism as in the original code but our tests showed lower performance than the basic algorithm. The reason for this is that, although we were able to reduce the time spent by the master thread in the main

```
! 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
```

Figure 7.5: Kernel for the `insert_factorize_node` task in charge of submitting the factorization tasks for a given node.

loop, the submission of factorization tasks in the `insert_factorize_node` becomes a significant overhead because the runtime system cannot handle the concurrent submission of tasks efficiently.

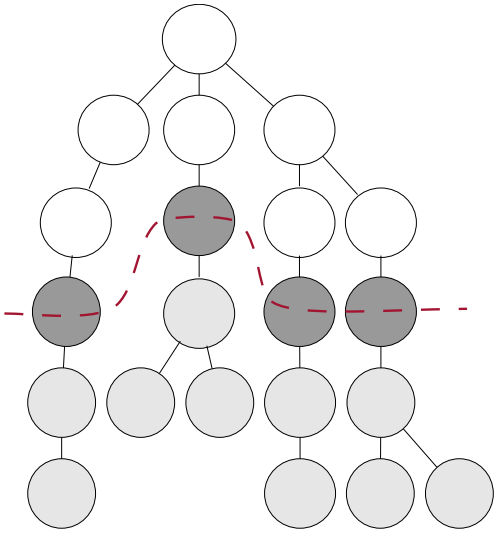

Figure 7.6: Illustration of the tree pruning strategy used by SpLLT.

The second strategy that we investigated consists in a logical pruning of the assembly tree similar to the strategy used by the `qr_mumps` solver [1]. This technique consists in grouping the small nodes at the bottom of the tree into subtrees that are processed in serial. Our algorithm, inspired by [11], 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 an approximation of the computational cost for processing a subtree. In Figure 7.6 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 red dashed 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

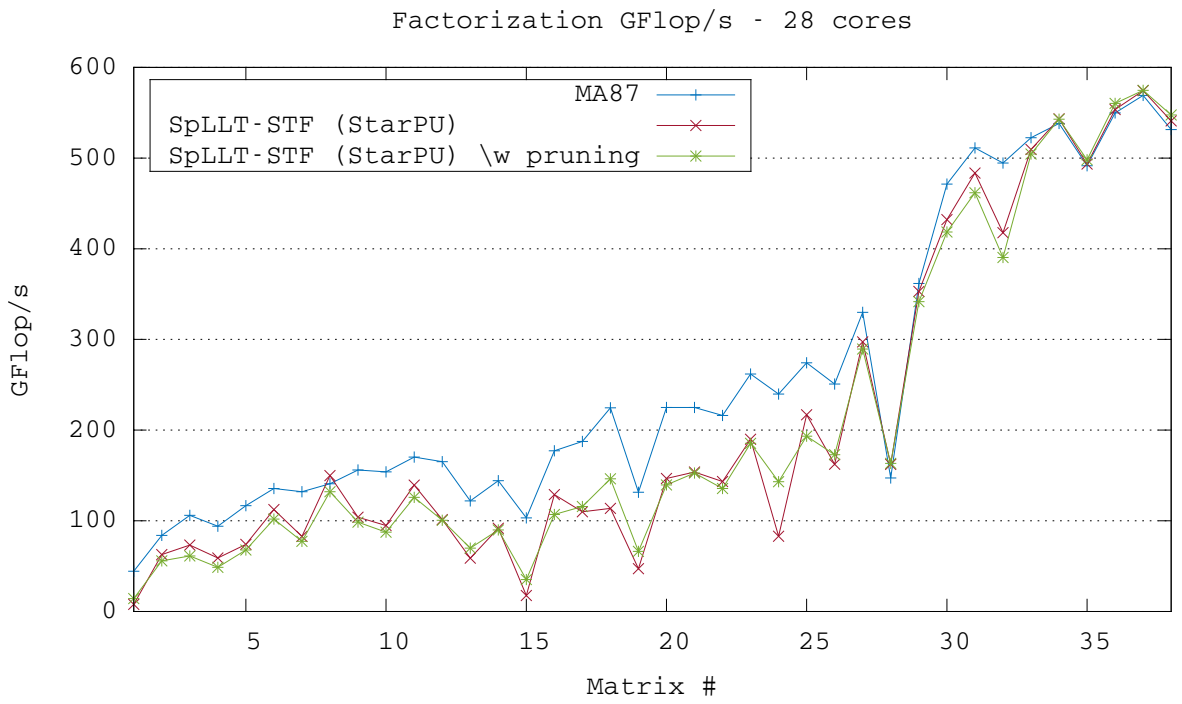

Figure 7.7: Performance results for StarPU versions of the SpLLT and HSL_MA87 solvers on 28 cores for the test matrices presented in Table A. We also show the impact of tree pruning on the performance of SpLLT.

this. 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.

We have implemented this strategy in SpLLT using both StarPU and OpenMP. We compare the StarPU version to the reference solver in Figure 7.7 and the OpenMP version to the reference solver in Figure 7.8. As we see from these results, the tree pruning strategy does not necessarily have great impact on performance, especially in the case of the StarPU version. Interestingly, for both matrices # 1 and #15 the performance results are greatly improved when exploiting subtrees especially for the OpenMP version of SpLLT. Using subtrees generally increases performance when solving large problems where parallelism is plentiful but limits it in the case of smaller problems and thus reduces the performance. Additionally, when the number of tasks is large and the task granularity is small, the proposed strategy reduces time spent by the master thread for DAG unrolling.

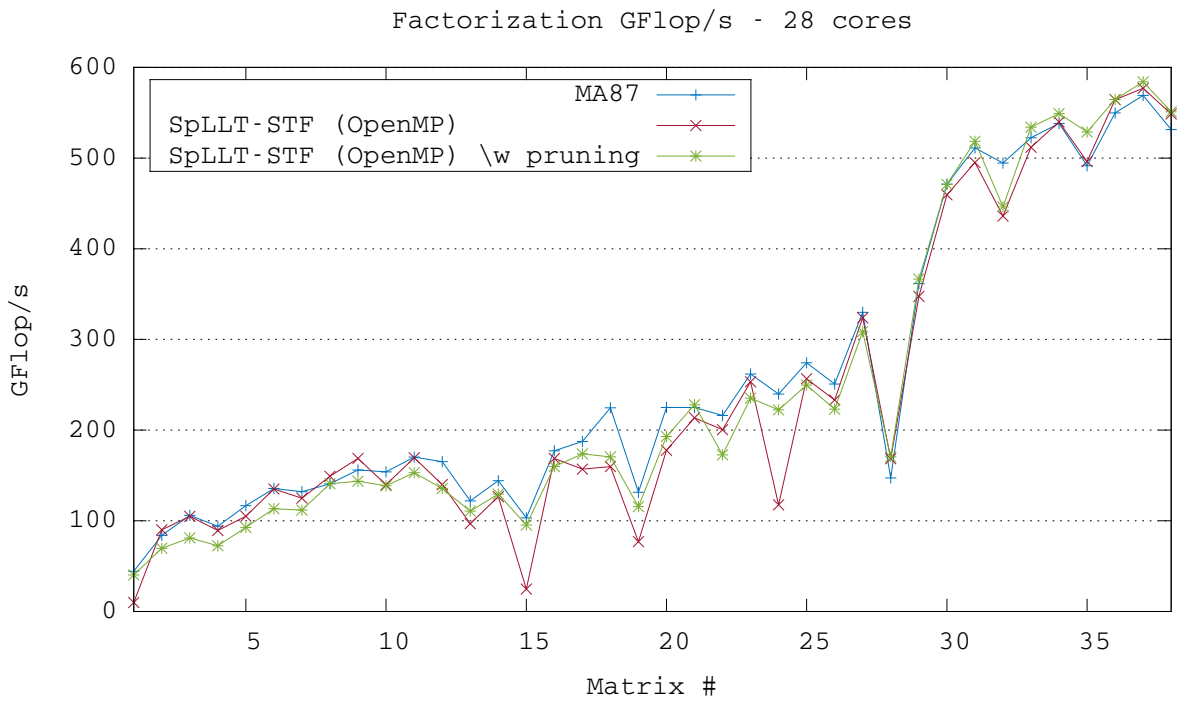

Figure 7.8: Performance results for OpenMP versions of the SpLLT and HSL_MA87 solvers on 28 cores for the test matrices presented in Table A. We also show the impact of tree pruning on the performance of SpLLT.

8 Concluding remarks

This report has described in detail the development of a new Cholesky solver implemented with a runtime system using a Sequential Task Flow model. As shown in our experimental results, our SpLLT solver gives competitive results compared to the reference solver HSL_MA87 on a large subset of our tested matrices and especially the largest problems. We have seen that smaller problems present a difficult challenge for the runtime systems, and we observe that when the task granularity in the DAG decreases the efficiency of the runtime may decrease. Although the performance of the OpenMP version generally lies within 10% of the performance achieved by HSL_MA87, the StarPU version can perform poorly on small problems with low-granularity tasks. This behaviour might be expected as the StarPU runtime system offers more functionality than OpenMP and so incurs more overhead. It provides, for example, features that allow us to extend the current version of our code to heterogeneous machines such as GPU-based systems and distributed-memory architectures. We will show in later work that the StarPU version of our code constitutes a good basis for the development of a Cholesky solver for heterogeneous and distributed-memory architectures.

References

[1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, *Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems*, Tech. Rep. IRI/RT–2014-03–FR, IRIT, November 2014. Submitted to ACM Transactions On Mathematical Software.

[2] 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.

[3] 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.

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

[5] 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.

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. J. Dongarra, *Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA*, in Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011, Anchorage, United States, May 2011, pp. 1432–1441.

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] J. D. Hogg and J. A. Scott, *A modern analyse phase for sparse tree-based direct methods*, Tech. Rep. RAL-TR-2010-031, STFC Rutherford Appleton Lab., 2010.

[16] 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.

[17] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

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

[19] F. Lopez, *Task-based multifrontal QR solver for heterogeneous architectures*, thèse de doctorat, Université Paul Sabatier, Toulouse, France, décembre 2015.

[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	JGD_Trefethen/Trefethen_20000	20.0	0.3	90.7	652.6	Integer matrix
29	ND/nd24k	72.0	14.4	321.6	2054.4	3D mesh problem
30	Janna/Flan_1565	1565	59.5	1477.9	3859.8	3D mechanical problem
31	Oberwolfach/bone010	987	36.3	1076.4	3876.2	Bone micro-FEM
32	Janna/StocF-1465	1465	11.2	1126.1	4386.6	Underground aquifer
33	GHS_psdef/audikw_1	944	39.3	1242.3	5804.1	Automotive crankshaft
34	Janna/Fault_639	639	14.6	1144.7	8283.9	Gas reservoir
35	Janna/Hook_1498	1498	31.2	1532.9	8891.3	Steel hook
36	Janna/Emilia_923	923	21.0	1729.9	13661.1	Gas reservoir
37	Janna/Geo_1438	1438	32.3	2467.4	18058.1	Underground deformation
38	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, $nz(A)$ represent the number entries in the matrix A, $nz(L)$ represent the number of entries the factor L and *Flops* correspond to the operation count for the matrix factorization.