H2020-FETHPC-2014: GA 671633

NLAFET Working Note 14

Experiments with sparse Cholesky using a parametrized task graph implementation

Iain Duff and Florent Lopez

June 2017

Document information

This preprint report is also published as Technical Report RAL-TR-2017-006, Science & Technology Facilities Council, UK.

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

Experiments with sparse Cholesky using a parametrized task graph implementation

Iain Duff[†] and Florent Lopez[†]

ABSTRACT

We describe the design of a sparse direct solver for symmetric positive-definite systems using the PaRSEC runtime system. In this approach the application is represented as a DAG of tasks and the runtime system is in charge of running the DAG on the target architecture. Portability of the code across different architectures is enabled by delegating to the runtime system the task scheduling and data management. Although runtime systems have been exploited widely in the context of dense linear algebra, the DAGs arising in sparse linear algebra algorithms remain a challenge for such tools because of their irregularity. In addition to overheads induced by the runtime system, the programming model used to describe the DAG impacts the performance and the scalability of the code. In this study we investigate the use of a Parametrized Task Graph (PTG) model for implementing a task-based supernodal method. We discuss the benefits and limitations of this model compared to the popular Sequential Task Flow model (STF) and conduct numerical experiments on a multicore system to assess our approach. We also validate the performance of our solver SpLLT by comparing it to the state-of-the-art solver MA87 from the HSL library.

Keywords: sparse Cholesky, SPD systems, runtime systems, PaRSEC

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

NLAFET Working Note 14. Also published as RAL Technical Report RAL-TR-2017-006.

This work is supported by the NLAFET Project funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement 671633. June 5, 2017

Contents

1 Introduction **1**

2 Task-based sparse Cholesky factorization **1**

3 The Parametrized Task Graph model **2**

4 Runtime systems **4**

5 Expressing a parallel Cholesky factorization using a PTG model **5**

6 Experimental results **6**

7 Concluding remarks **9**

A Test problems **11**

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. In this approach the runtime system acts as a software layer between our application and the target architecture and thus enables portability of our code across different architectures. In our solver we use the task-based supernodal method implemented in the state-of-the-art HSL_MA87 [11] solver that has been shown to be efficient for exploiting multicore architectures. The HSL_MA87 solver is designed following a traditional approach where the task scheduler is implemented using a low-level API specific to a target architecture. In our solver, we express the DAG using a high-level API and the runtime system handles the management of task dependencies, scheduling and data coherency across the architecture.

Many dense linear algebra software packages have already exploited this approach and have shown that it is efficient for exploiting modern architectures ranging from multicores to large-scale machines including heterogeneous systems. Two examples of such libraries are DPLASMA [4] built with the PaRSEC [5] runtime system and Chameleon which has an interface to several runtime systems including StarPU [3] and PaRSEC. As a result of these research efforts for demonstrating the effectiveness of this approach, the OpenMP board decided to include tasking features in version 3.0 of the standard to facilitate the implementation of DAG-based algorithms. The library includes, for example, the `task` directive for creating tasks and the `depend` clause for declaring dependencies between them. The PLASMA package [10], which used to rely on the QUARK runtime system, has been ported to OpenMP using the tasking features offered in the latest versions of the standard. This transition not only improved the portability and maintainability of this library but also didn't impact the performance of the code [15].

In contrast, sparse linear algebra algorithms represent a bigger challenge for runtime systems because the DAGs arising in this context are usually irregular with an extremely variable task granularity. In [8] we studied this case using both the StarPU runtime system and the OpenMP standard for implementing the sparse Cholesky solver SpLLT. We exploited a Sequential Task Flow (STF) model and obtained competitive performance compared to HSL_MA87. However, we identified potential limitations of the STF model in terms of performance and scalability. For this reason we now investigate the use of an alternative paradigm, the Parametrized Task Graph (PTG) for implementing the factorization algorithm. We use the PaRSEC runtime system to implement the PTG and compare it with our existing OpenMP implementation and the HSL_MA87 solver.

2 Task-based sparse Cholesky factorization

In the context of direct methods, the solution of equation (1.1) is generally achieved in three main phases: the *analysis*, the *factorization* and the *solve* phases. The analysis is responsible for computing the structure of the factors and the data dependencies during the factorization. These dependencies can be represented by a tree structure called an *elimination tree*. Note that the nonzero structure of the factors differs from the original matrix because some zero entries become nonzero during the factorization. This phenomenon is referred to as *fill-in*. Moreover, sets of consecutive columns that have the same structure are amalgamated and the elimination tree is replaced by an *assembly tree* where nodes in this tree are referred to as *supernodes*. Although this amalgamation generally results in a higher fill-in and therefore a higher floating point operation count, it enables the use of efficient Level-3 BLAS operations in the factorization. We use the software package SSIDS from SPRAL[1] to compute the assembly tree during in the analysis phase.

The factorization phase computes the Cholesky decomposition of the input matrix as:

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

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

where P is a permutation matrix and the factor L is a lower triangular matrix. The two main techniques for finding the permutation matrix are Minimum Degree [2,13,14] or Nested Dissection [9].

The factorization is effected by traversing the assembly tree in a topological order and performing two main operations at each supernode: compute a dense Cholesky factorization of the supernode and update the ancestor supernodes with these factors. 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.

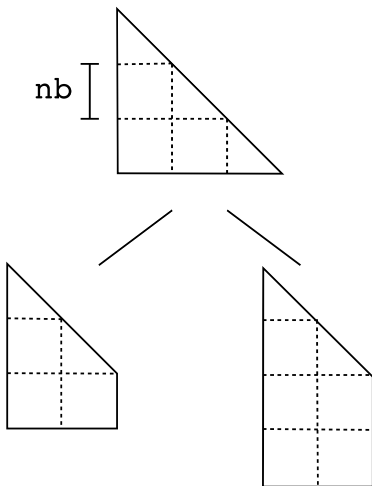

Figure 2.1: Simple assembly tree with three supernodes partitioned into square blocks of order `nb`.

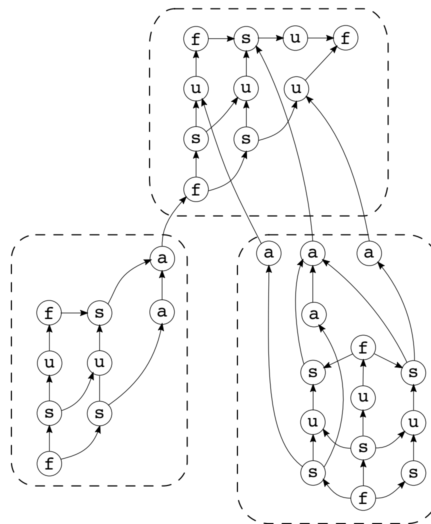

Figure 2.2: DAG corresponding to the factorization of the tree in Figure 2.1.

Two levels of parallelism are commonly exploited in the assembly tree: *tree-level* and *node-level* parallelism. Tree-level parallelism comes from the fact that coefficients in independent branches of the tree may be processed in parallel and node-level parallelism corresponds to the fact that multiple resources can be used to process a supernode. In our work we implement a DAG-based supernodal method in which supernodes are partitioned into square blocks of order `nb` and operations are performed on these blocks. In Figure 2.2 we illustrate the DAG for the factorization of the simple assembly tree shown in Figure 2.1 containing three supernodes. The tasks in this DAG execute the following kernels:

- `factor_block` (denoted `f`) that computes the Cholesky factor of a block on the diagonal;
- `solve_block` (denoted `s`) that performs a triangular solve of a subdiagonal block using the factor computed in `factor_block`;
- `update_block` (denoted `u`) that performs an update of a block within a supernode corresponding to the previous factorization of blocks;
- `update_btw` (denoted `a`) that computes the update between supernodes.

In this algorithm the exploitation of parallelism no longer relies on the assembly tree but is replaced by the DAG where tree-level and node-level parallelism are exploited without distinction. Moreover, tasks in a supernode might become ready for execution during the processing of its child nodes. This brings an additional opportunity for concurrency referred to as *internode-level* parallelism.

3 The Parametrized Task Graph model

The PTG model is a dataflow programming model for representing a DAG and was introduced in [6]. It is an alternative to the Sequential Task Flow (STF) paradigm that we presented and used in previous

work [8]. In the STF model, the DAG is sequentially traversed and tasks are submitted to the runtime system along with data access information used by the runtime system to infer dependencies and guarantee the correctness of the parallel execution. In the PTG 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 3.1. A part of the DAG associated with this algorithm is illustrated in Figure 3.2 and shows the dependencies between the tasks executing the kernels `f` and `g`.

```
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.

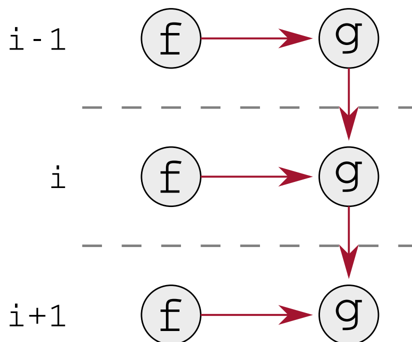

Figure 3.2: Extract of the DAG corresponding to the sequential code presented in Figure 3.1.

Figure 3.3 illustrates a compact representation of the DAG presented in Figure 3.2 where tasks are divided into two classes: `task_f` and `task_g` executing kernels `f` and `g` respectively. Tasks of type `task_f` manipulate data `X` in Read/Write (RW) mode and tasks of type `task_g` manipulate three pieces of data, `X` and `Y1` in Read (R) mode and `Y2` in Write (W) mode. Each task instance is identified by a parameter `i` ranging from `1` to `N`. In `task_f` tasks, the input data is directly read from the array `x` in memory and the output data is given to the task instance `task_g(i)`. In `task_g` the input `X` comes from the output of `task_f(i)` task, the input `Y1` comes from the output of `task_g(i-1)` and the output `Y2` is given to task `task_g(i+1)`.

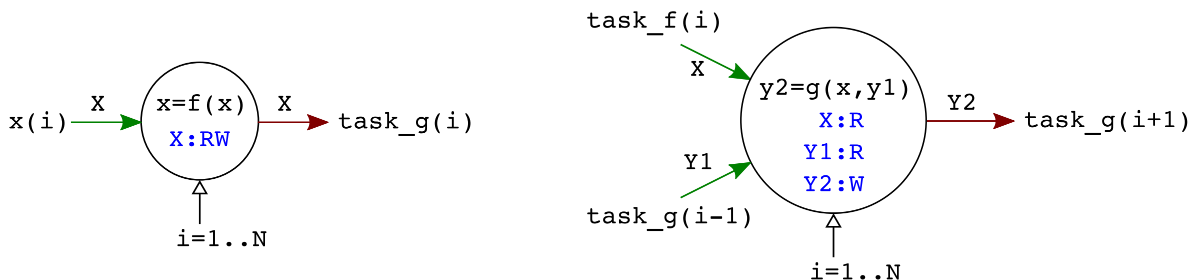

Figure 3.3: PTG representation for task types `task_f` (left) and `task_g` (right) as shown in the DAG presented in Figure 3.2.

The example in Figure 3.3 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.

4 Runtime systems

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 such as CPUs and GPUs.

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 on the compute nodes during the execution.

```
N     [type = int]

task_f(i) /* Task name */

i = 1..N-1 /* Execution space declaration for parameter i */

: x(i) /* Task must be executed on the node where x(i) is stored */

  /* Task reads x(i) from memory ... */
  RW X <- x(i)
  /* ... and sends it to task_g(i) */
       -> X task_g(i)
BODY

  X = f(X)   /* Code executed by the task */

END

task_g(i) /* Task name */

i = 1..N-1 /* Execution space declaration for parameter i */

: y(i) /* Task must be executed on the node where y(i) is stored */

  /* Task reads x(i) fron task_f(i)... */
  R X  <- X task_f(i)
  /* ... y(i-1) from task_g(i-1)... */
  R Y1 <- (i > 1) ? Y2 task_g(i-1) : y(i-1)
  /* ... and sends y(i) to task_g(i+1) */
  W Y2 -> (i < N-1) ? Y2 task_g(i+1)

BODY

  Y2 = g(X, Y1) /* Code executed by the task */

END
```

Figure 4.1: Simple example of a parallel version of the sequential code in Figure 3.1 using a PTG model with PaRSEC.

In Figure 4.1, we illustrate the use of a PTG model using PaRSEC by implementing a parallel version of the simple example shown in Figure 3.2 using the JDF language. In this JDF code we have the representation for the two types of task `task_f` and `task_g` that are associated with one parameter `i` each. This parameter is defined on the range `1..N-1` where `N` is defined at the beginning of the JDF code, associated with the type `int`, and initialised when the DAG is instantiated. As illustrated by the diagram in Figure 3.3, the dataflow for `task_f` contains two edges that are expressed 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 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 `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 provided 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.

5 Expressing a parallel Cholesky factorization using a PTG model

We use PaRSEC to implement our SpLLT solver by expressing the DAG of the factorization algorithm presented in Section 2 in the JDF language. In a previous study [1] we investigated the use of a PTG model for implementing a multifrontal QR method and used a two-level approach where the processing of the assembly tree and the node factorization are split in two different JDFs thus separating 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 internode-level 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 5.1 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 2.2, the factorization DAG contains one `task_factor_block` task for every block on the diagonal in 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

```
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 5.1: Extract of the JDF representation implemented with PaRSEC for the supernodal algorithm.

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 in the output dataflow which tasks become 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.

6 Experimental results

We tested the PaRSEC implementation of 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 are provided by the Intel MKL v11.3 library and we used the latest version (version v1.1.0-2771-g7a4cb0d) of the PaRSEC runtime system.

In our experiments, we use a set of matrices taken from the SuiteSparse Matrix Collection [7]. From this collection, we selected a set of symmetric positive-definite matrices from different applications with varying 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 [12]. Note that, in this table, matrix 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 SSIDS and set the `nemin` value to 32. This corresponds to a good trade-off between sparsity and efficiency of floating-point computation.

For each tested problem it is not theoretically possible to determine an optimal value for the parameter `nb` because it depends on a huge number of factors including the number of resources and the amount of parallelism available in the DAG. For this reason, the best value for the parameter `nb` is empirically chosen by running multiple tests on the range `(256, 384, 512, 768, 1024, 1536)`.

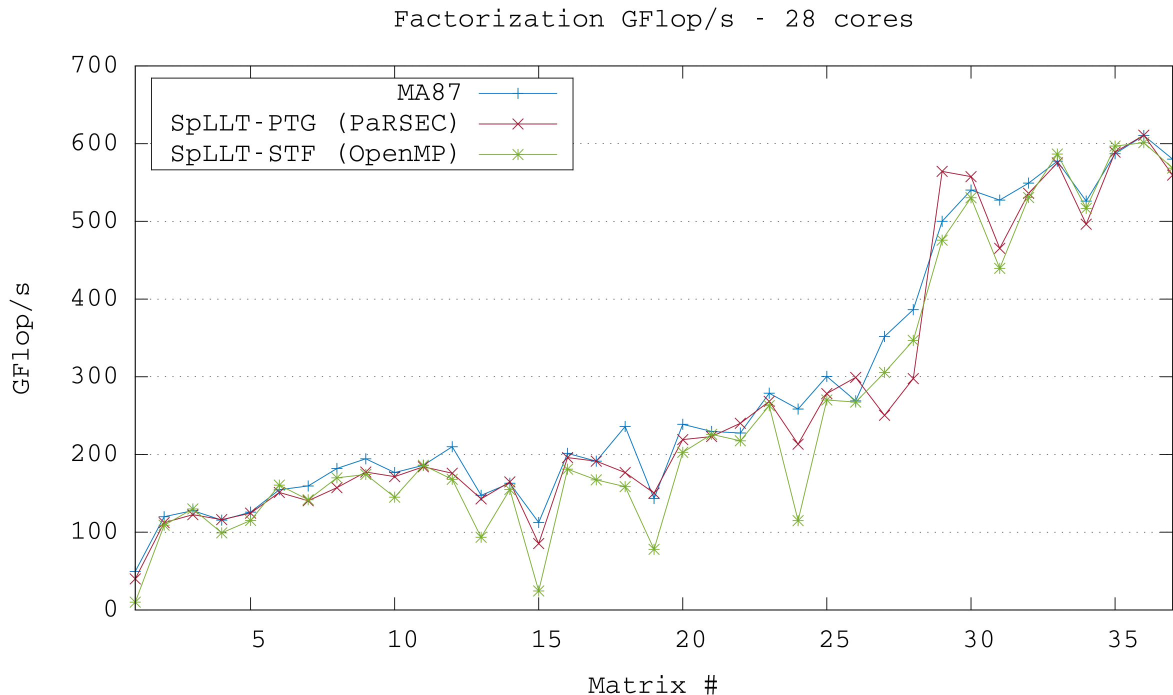

Figure 6.1: Performance results for both PaRSEC and OpenMP versions of the SpLLT and HSL_MA87 solvers on 28 cores for the test matrices presented in Table A.

In order to assess the performance results obtained with our PaRSEC implementation, we run the STF-based OpenMP implementation of our solver, presented in [8], and the HSL_MA87 solver on the same set of test matrices. The factorization times are presented in Table 6.1 along with the value of the parameter `nb` for which these times are obtained. The performance results associated with the factorization

#	Name	SpLLT				MA87	
		PaRSEC		OpenMP			
		nb	Time (s)	nb	Time (s)	nb	Time (s)
1	Schmid/thermal2	384	0.465	768	1.831	768	0.375
2	Rothberg/gearbox	384	0.203	384	0.209	256	0.190
3	DNVS/m_t1	256	0.191	256	0.180	256	0.183
4	Boeing/pwtk	384	0.216	768	0.253	256	0.217
5	Chen/pkustk13	384	0.219	384	0.237	256	0.217
6	GHS_psdef/crankseg_1	384	0.224	256	0.211	256	0.219
7	Rothberg/cfd2	384	0.244	384	0.242	256	0.215
8	DNVS/thread	256	0.227	256	0.210	256	0.196
9	DNVS/shipsec8	256	0.239	384	0.243	256	0.218
10	DNVS/shipsec1	256	0.236	384	0.279	256	0.229
11	GHS_psdef/crankseg_2	256	0.265	256	0.262	256	0.262
12	DNVS/fcondp2	256	0.296	384	0.310	256	0.248
13	Schenk_AFE/af_shell3	384	0.400	768	0.612	256	0.388
14	DNVS/troll	256	0.358	512	0.381	256	0.362
15	AMD/G3_circuit	384	0.788	768	2.760	256	0.598
16	GHS_psdef/bmwcra_1	384	0.329	384	0.356	256	0.320
17	DNVS/halfb	384	0.389	384	0.445	256	0.389
18	Um/2cubes_sphere	384	0.438	512	0.488	256	0.328
19	GHS_psdef/ldoor	384	0.582	512	1.120	256	0.610
20	DNVS/ship_003	384	0.398	384	0.430	256	0.365
21	DNVS/fullb	384	0.484	384	0.478	256	0.470
22	GHS_psdef/inline_1	384	0.637	512	0.703	256	0.672
23	Chen/pkustk14	384	0.585	384	0.597	256	0.563
24	GHS_psdef/apache2	384	0.858	384	1.593	256	0.708
25	Koutsovasilis/F1	384	0.819	384	0.844	384	0.759
26	Oberwolfach/boneS10	512	0.993	384	1.110	384	1.104
27	ND/nd12k	512	2.052	512	1.682	384	1.461
28	ND/nd24k	768	6.986	768	5.999	384	5.383
29	Janna/Flan_1565	768	6.929	384	8.218	384	7.820
30	Oberwolfach/bone010	768	7.013	512	7.373	384	7.238
31	Janna/StocF-1465	768	9.523	768	10.078	384	8.400
32	GHS_psdef/audikw_1	768	10.894	768	11.004	384	10.634
33	Janna/Fault_639	1024	14.464	768	14.185	768	14.407
34	Janna/Hook_1498	1024	18.096	1024	17.378	768	17.074
35	Janna/Emilia_923	768	23.263	1536	22.957	768	23.337
36	Janna/Geo_1438	1024	29.629	1536	30.103	768	29.651
37	Janna/Serena	1536	53.805	1536	52.954	768	51.888

Table 6.1: Factorization times (seconds) obtained with MA87 and SpLLT code using PaRSEC and OpenMP. The factorizations were run with the block sizes `nb=(256, 384, 512, 768, 1024, 1536)` on 28 cores and `nemin=32`.

are shown in Figure 6.1. It is interesting to note that the value for the parameter `nb` is generally bigger for SpLLT than for HSL_MA87. This is because of a bigger overhead required by a general purpose runtime system for managing the tasks compared to the lightweight scheduler in HSL_MA87 specifically optimized for the target architecture. From these factorization rates we can see that SpLLT is competitive with HSL_MA87 with both the OpenMP and PaRSEC versions. In addition the PaRSEC version generally performs better in our tests and we observe a big difference between the OpenMP and PaRSEC versions for some particular problems. For example results obtained with matrices #1, #13, #15, #19 and#24 indicate that the SpLLT-OpenMP version performs poorly on these particular problems. In [8] we identified this issue and determined that when using the STF model the performance could be limited by the time spent in unrolling the DAG which causes resource starvation. This occurs either when the task granularity in the DAG is relatively small or the number of resources is big compared to the amount of parallelism available. In the PTG version, this issue no longer appears because the traversal of the DAG is handled in a distributed fashion by all the workers involved in the computation.

7 Concluding remarks

In this study we presented the design of a task-based sparse Cholesky solver using a PTG model and implemented with the PaRSEC runtime system. In our experiments, we have shown that the PTG model is an interesting alternative to the popular STF model as our PaRSEC implementation offered competitive performance against our STF-based OpenMP implementation and the state-of-the art HSL_MA87 solver on a multicore machine. We explained the potential benefits of this model over the STF model for large-scale systems and we pointed out the challenge of implementing the PTG representation of a DAG. In the case of irregular DAGs, such as those arising in the context of sparse linear algebra algorithms, we show that it becomes particularly difficult to produce the dataflow representation of this DAG. The encouraging performance results obtained with our PaRSEC implementation indicate that it is a good candidate to target distributed memory systems. The high level of abstraction used to describe the DAG and the portability provided by the runtime system allows us to use the same JDF code for targeting such architectures. However, a new challenge arises for establishing a proper data distribution for limiting the data movements between the compute nodes.

The future work for this PaRSEC version of our SpLLT solver includes the design of a data distribution for the supernodes and the blocks within supernodes capable of limiting the communication cost when running on a distributed-memory machine. Moreover in the supernodal algorithm the updates between supernodes can be performed in a different order that impacts the communication pattern and we want to investigate the effect of the updating scheme on the performance of our code.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] I. S. Duff, J. Hogg, and F. Lopez, *Experiments with sparse cholesky using a sequential task-flow implementation*, Tech. Rep. RAL-TR-16-016, Rutherford Appleton Laboratory, Oxfordshire, England, 2016. NLAFET Working Note 7.

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

[10] A. Haidar, J. Kurzak, and P. Luszczek, *An improved parallel singular value algorithm and its implementation for multicore hardware*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, New York, NY, USA, 2013, ACM, pp. 90:1–90:12.

[11] 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.

[12] 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.

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

[14] 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.

[15] A. YarKhan, J. Kurzak, P. Luszczek, and J. Dongarra, *Porting the plasma numerical library to the openmp standard*, International Journal of Parallel Programming, 45 (2017), pp. 612–633.

A Test problems

#	Name	n (10^3)	nz(A) (10^6)	nz(L) (10^6)	Flops (10^9)	Application/Description
1	Schmid/thermal2	1228	4.9	51.6	14.6	Unstructured thermal FEM
2	Rothberg/gearbox	154	4.6	37.1	20.6	Aircraft flap actuator
3	DNVS/m_t1	97.6	4.9	34.2	21.9	Tubular joint
4	Boeing/pwtk	218	5.9	48.6	22.4	Pressurised wind tunnel
5	Chen/pkustk13	94.9	3.4	30.4	25.9	Machine element
6	GHS_psdef/crankseg_1	52.8	5.3	33.4	32.3	Linear static analysis
7	Rothberg/cfd2	123	1.6	38.3	32.7	CFD pressure matrix
8	DNVS/thread	29.7	2.2	24.1	34.9	Threaded connector
9	DNVS/shipsec8	115	3.4	35.9	38.1	Ship section
10	DNVS/shipsec1	141	4.0	39.4	38.1	Ship section
11	GHS_psdef/crankseg_2	63.8	7.1	43.8	46.7	Linear static analysis
12	DNVS/fcondp2	202	5.7	52.0	48.2	Oil production platform
13	Schenk_AFE/af_shell3	505	9.0	93.6	52.2	Sheet metal forming
14	DNVS/troll	214	6.1	64.2	55.9	Structural analysis
15	AMD/G3_circuit	1586	4.6	97.8	57.0	Circuit simulation
16	GHS_psdef/bmwcra_1	149	5.4	69.8	60.8	Automotive crankshaft
17	DNVS/halfb	225	6.3	65.9	70.4	Half-breadth barge
18	Um/2cubes_sphere	102	0.9	45.0	74.9	Electromagnetics
19	GHS_psdef/ldoor	952	23.7	144.6	78.3	Large door
20	DNVS/ship_003	122	4.1	60.2	81.0	Ship structure
21	DNVS/fullb	199	6.0	74.5	100.2	Full-breadth barge
22	GHS_psdef/inline_1	504	18.7	172.9	144.4	Inline skater
23	Chen/pkustk14	152	7.5	106.8	146.4	Tall building
24	GHS_psdef/apache2	715	2.8	134.7	174.3	3D structural problem
25	Koutsovasilis/F1	344	13.6	173.7	218.8	AUDI engine crankshaft
26	Oberwolfach/boneS10	915	28.2	278.0	281.6	Bone micro-FEM
27	ND/nd12k	36.0	7.1	116.5	505.0	3D mesh problem
28	ND/nd24k	72.0	14.4	321.6	2054.4	3D mesh problem
29	Janna/Flan_1565	1565	59.5	1477.9	3859.8	3D mechanical problem
30	Oberwolfach/bone010	987	36.3	1076.4	3876.2	Bone micro-FEM
31	Janna/StocF-1465	1465	11.2	1126.1	4386.6	Underground aquifer
32	GHS_psdef/audikw_1	944	39.3	1242.3	5804.1	Automotive crankshaft
33	Janna/Fault_639	639	14.6	1144.7	8283.9	Gas reservoir
34	Janna/Hook_1498	1498	31.2	1532.9	8891.3	Steel hook
35	Janna/Emilia_923	923	21.0	1729.9	13661.1	Gas reservoir
36	Janna/Geo_1438	1438	32.3	2467.4	18058.1	Underground deformation
37	Janna/Serena	1391	33.0	2761.7	30048.9	Gas reservoir

Table A.1: Test matrices and their characteristics without node amalgamation. n is the matrix order, $nz(A)$ 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.