Multifrontal QR factorization for multicore architectures over runtime systems

Emmanuel Agullo[1], Alfredo Buttari[2], Abdou Guermouche[3], and Florent Lopez[4]

[1] INRIA - LaBRI, Bordeaux
[2] CNRS - IRIT, Toulouse
[3] Université de Bordeaux - LaBRI, Bordeaux
[4] Université Paul Sabatier - IRIT, Toulouse

Abstract. To face the advent of multicore processors and the ever increasing complexity of hardware architectures, programming models based on DAG parallelism regained popularity in the high performance, scientific computing community. Modern runtime systems offer a programming interface that complies with this paradigm and powerful engines for scheduling the tasks into which the application is decomposed. These tools have already proved their effectiveness on a number of dense linear algebra applications. This paper evaluates the usability of runtime systems for complex applications, namely, sparse matrix multifrontal factorizations which constitute extremely irregular workloads, with tasks of different granularities and characteristics and with a variable memory consumption. Experimental results on real-life matrices show that it is possible to achieve the same efficiency as with an *ad hoc* scheduler which relies on the knowledge of the algorithm. A detailed analysis shows the performance behavior of the resulting code and possible ways of improving the effectiveness of runtime systems.
Keywords: sparse matrices, multifrontal method, QR factorization, runtime systems, heterogeneous architectures.

1 Introduction

The increasing degree of parallelism and complexity of hardware architectures requires the High Performance Computing (HPC) community to develop more and more complex software. To achieve high levels of optimization and fully benefit of their potential, not only the related codes are heavily tuned for the considered architecture, but the software is furthermore often designed as a single whole that aims to cope with both the algorithmic and architectural needs. If this approach may indeed lead to extremely high performance, it is at the price of a tremendous development effort and a very poor maintainability: At which price in terms of code refactoring can we extend a shared-memory software to handle distributed memory machines if it has been assumed some contiguity properties on data in memory at the algorithmic level? How to extend the same software to handle accelerators efficiently if the numerical algorithm itself has been designed to match a regular data distribution?

Alternatively, a modular approach can be employed. First, the numerical algorithm is written at a high level independently of the hardware architecture as a Directed Acyclic Graph (DAG) of tasks where a vertex represents a task and an edge represents a dependency between tasks. A second layer is in charge of taking the scheduling decisions. Based on these decisions, a runtime system is then in charge of performing the actual execution of the tasks, both ensuring that dependencies are satisfied at execution time and maintaining data consistency. The fourth layer consists of the optimized code for the related tasks on the underlying architectures. In most cases, the last three layers need not be written by the application developer. Indeed, it usually exists a very competitive state-of-the-art generic scheduling algorithm (such as work-stealing [4], Minimum Completion Time [19]) matching the algorithmic needs to efficiently exploit the targeted architecture (otherwise, a new scheduling algorithm may be designed, which will in turn be likely to apply to a whole class of algorithms). The runtime system only needs to be extended once for each new architecture. Finally, most of the time, the high-level algorithm can be cast in terms of standard operations (such as BLAS in dense linear algebra) for which vendors provide optimized codes. All in all, with such a modular approach, only the high-level algorithm has to be specifically designed, which ensures a high productivity. The maintainability is also guaranteed since the use of new hardware only requires (in principle) third party effort.

The dense linear algebra community has strongly adopted such a modular approach over the past few years [10, 17, 1, 8] and delivered subsequent production-level solvers. However, beyond this community, only few research efforts have been conducted to handle large scale codes. The main reason is that irregular problems are complex to design with a clear separation of the software layers without inducing performance loss. On the other hand, the runtime system community has strongly progressed, delivering very reliable and effective tools [6, 7, 14, 5] up to the point that the OpenMP board is reconsidering its tasking model [5] with respect to that approach.

This paper evaluates the usability of runtime systems and of the associated modular approach in the context of complex applications, namely, the multifrontal QR factorization of sparse matrices [3], which yields extremely irregular workloads, with tasks of different granularities and characteristics as well as a variable memory consumption. For that, we consider a heavily hand-tuned state-of-the-art solver for multicore architectures, `qr_mumps` [9], we propose an alternative modular design of the solver on top of the StarPU runtime system [5] and we present a thorough performance comparison of both approaches on the architecture for which the original solver has been tuned. The penalty of delegating part of the task management system to a third party software, the runtime system, is to be regarded with respect to the impact of the numerical algorithmic choices; for that purpose, we also discuss the relative performance with respect to another state-of-the-art multifrontal QR solver for multicore architectures, the SuiteSparseQR package [11], referred to as `spqr`.

[5] http://openmp.org/wp/presos/SC12/SC12_State_of_LC.2.pdf

2 Multifrontal QR factorization

The multifrontal method, introduced by Duff and Reid [12] as a method for the factorization of sparse, symmetric linear systems, can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix A^TA share the same structure. As in the Cholesky case, the multifrontal QR factorization is based on the concept of *elimination tree* [18]. This graph, which has a number of nodes that is typically one order of magnitude or more smaller than the number of columns in the original matrix, expresses the dependencies among the computational tasks in the factorization: each node i of the tree is associated with k_i unknowns of the original matrix and represents an elimination step of the factorization. The coefficients of the corresponding k_i columns and all the other coefficients affected by their elimination are assembled together into a relatively small dense matrix, called *frontal matrix* or, simply, *front*, associated with the tree node. The multifrontal QR factorization consists in a tree traversal in a topological order (i.e., bottom-up) such that, at each node, two operations are performed. First, the frontal matrix is **assembled** by stacking the matrix rows associated with the k_i unknowns with uneliminated rows resulting from the processing of child nodes. Second, the k_i unknowns are eliminated through a **complete** QR **factorization** of the front; this produces k_i rows of the global R factor, a number of Householder reflectors that implicitly represent the global Q factor and a *contribution block* formed by the remaining rows and that will be assembled into the parent front together with the contribution blocks from all the front siblings. A detailed presentation of the multifrontal QR method, including the optimization techniques described above, can be found in Amestoy *et al.* [3].

The classical approach to the parallelization of the multifrontal QR factorization [3, 11] consists in exploiting separately two distinct sources of concurrency: **tree** and **node parallelism**. The first stems from the fact that fronts in separate branches are independent and can thus be processed concurrently; the second from the fact that, if a front is big enough, multiple processes can be used to assemble and factorize it. The baseline of this work, instead, is the parallelization model proposed by Buttari [9] in the `qr_mumps` software which is based on the method presented earlier in related work on dense matrix factorizations by Buttari *et al.* [10] and extended to the supernodal Cholesky factorization of sparse matrices by Hogg *et al.* [15]. In this approach, frontal matrices are partitioned into block-columns, which allows one to decompose the workload into fine-grained tasks. Each task corresponds to the execution of an elementary operation on a block-column or a front; five elementary operations are defined: 1) the **activation** of a front consists in computing its structure and allocating the associated memory, 2) **panel** factorization of a block-column, 3) **update** of a block-column with respect to a previous panel operation, 4) **assembly** of the piece of contribution block in a block-column in the parent front and 5) **cleanup** of a front which amounts to storing the factors aside and deallocating the memory allocated in the corresponding activation. These tasks are then arranged into

a DAG where vertices represent tasks and edges the dependencies among them. Figure 1 shows an example of how a simple elimination tree (on the left) can be transformed into a DAG (on the right); further details on this transition can be found in the paper by Buttari [9] from which this example was taken.

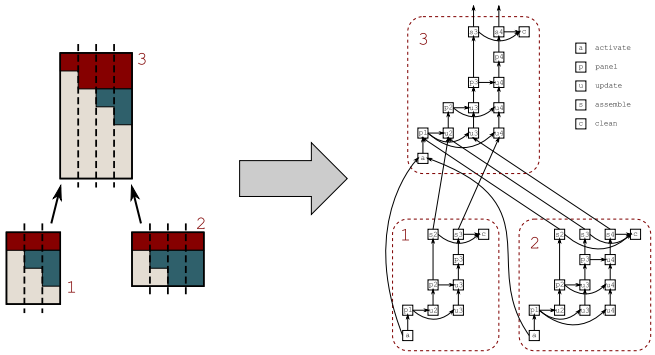

Fig. 1. An example of how a simple elimination tree with three nodes is transformed into a DAG in the `qr_mumps` code. Vertical, dashed lines show the partitioning of fronts into block-columns. Dashed-boxes group together all the tasks related to a front.

The execution of the tasks is guided by a dynamic scheduler which allows the tasks to work asynchronously. This approach is capable of achieving higher performance than the classical one thanks to the fact that tree and node types of parallelism are replaced by a single source, that is, DAG parallelism. This provides a higher amount of concurrency since dependencies are defined on a block-column basis rather than a front basis, which for instance allows one to start working on a front even if its children are not completely factorized. The execution mode, moreover, is more suited to multicore based architectures, as also shown in other related papers [10, 15], because, unlike classical approaches [11, 3], it does not suffer from the presence of heavy synchronizations.

3 The task-based StarPU runtime system

As most modern task-based runtime systems, StarPU aims at performing the actual execution of the tasks, both ensuring that the DAG dependencies are satisfied at execution time and maintaining data consistency. The particularity of StarPU is that it was initially designed to write a program independently of the architecture and thus requires a strict separation of the different software layers: high-level algorithm, scheduling, runtime system, actual code of the tasks. We refer to Augonnet *et al.* [5] for the details and present here a simple example containing only the features relevant to this work. Assume we aim at executing the sequence $fun_1(\underline{x}, y)$; $fun_2(\ x)$; $fun_1(\underline{z}, w)$, where $fun_{i,i\in\{1,2\}}$ are functions applied on w, x, y, z data; the arguments corresponding to data which are

submit_task(fun_1, $\underline{x}$, y, id=id_1)

submit_task(fun_2, x, id=id_2)

declare_dependency($id_3 \leftarrow id_1$)

submit_task(fun_1, $\underline{z}$, w, id=id_3)

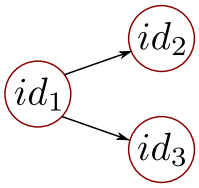

Fig. 2. Basic StarPU-like example (left) and associated DAG (right). Arguments corresponding to data that are modified by the function are underlined. The $id_1 \rightarrow id_2$ dependency is implicitly inferred with respect to the data hazard on x while the $id_1 \rightarrow id_3$ dependency is declared explicitly.

modified by a function are underlined. A task is defined as an instance of a function on a specific set of data. Because of possible data hazards [2] (here on x between fun_1 and fun_2), a so-called superscalar analysis [2] has to be performed to ensure that the parallelization does not violate dependencies. While (modern) CPUs implement such a superscalar analysis on chip at the instruction level [2], runtime systems implement it in a software layer on tasks. The set of tasks and related data they operate on are declared with the **submit_task** instruction. This is a non blocking call that allows one to add a task to the current DAG and postpone its actual execution to the moment when its dependencies are satisfied. Although the API of a runtime system can be virtually reduced to this single instruction, it may be convenient in certain cases to explicitly define extra dependencies. For that, identification tags can be attached to the tasks at submission time and dependencies are declared between the related tags with the **declare_dependency** instruction. For instance, an extra dependency is defined between the first and the third task in Figure 2 (left). Figure 2 (right) shows the resulting DAG built (and executed) by the runtime system. Optionally, a priority value can be assigned to each task to guide the runtime system in case multiple tasks are ready for execution at a given moment. In StarPU, the scheduling system is clearly split from the core of the runtime system (data consistency engine and actual task execution). Therefore, not only all built-in scheduling policies can be applied to any high-level algorithm, but new scheduling strategies can be implemented without having to interfere with low-level technical details of the runtime system.

4 Multifrontal QR factorization Based on StarPU

The execution of the `qr_mumps` software presented in Section 2 relies on a *ad hoc* scheduler which is extremely limited in features, relies on the knowledge of the algorithm and is, as a result, extremely lightweight. Replacing this scheduler with a complex, general purpose runtime system such as StarPU is not an easy task particularly because of several issues. First the DAG associated to the factorization of medium to large size matrices can have hundreds of thousands of tasks. Generating the whole DAG by submitting all the tasks to the runtime

system may overload it and may require too much memory (see, for example, Lacoste *et al.* [16]). Second because of contribution blocks, different traversals of the DAG may result in a different memory consumption. For this reason, the activation tasks have to be carefully scheduled in order to avoid an excessive memory consumption. Third StarPU automatically infers dependencies among tasks depending on data hazards. Because, for what said above, it is not possible to allocate at once the memory needed for all the fronts in the tree, the whole DAG cannot be submitted entirely unless all the dependencies are explicitly provided to StarPU, which is largely unpractical.

The first and the third issue can be overcome by submitting tasks progressively by means of other tasks. Because activation tasks are responsible for allocating the memory of the associated frontal matrices, in our StarPU based implementation they will also be in charge of submitting the tasks for their assembly and factorization i.e., panel, update, assembly and cleanup; this is shown in Algorithm 1 *(right)*. The dependencies among these tasks can be automatically inferred by StarPU. Activation tasks, instead, are submitted all at once at the beginning of the factorization and their mutual dependencies explicitly specified to StarPU as shown in Algorithm 1 *(left)*; because they are limited in number, the runtime system will not be overloaded. As a result of this technique, the size of the DAG that the runtime system has to handle is only proportional to the number of active fronts.

Algorithm 1. Task management

Main code
(submit activation tasks):

```
1: for all n in pre-computed post-order do
2:   for all children c of node n do
3:     declare_dependency(id_n ← id_c)
4:   end for
5:   /* submit activation of front f_n */
6:   submit_task(activation, f_n, prio.=−n, id=id_n)
7: end for
```

Code of the activation task
(submit other tasks):

```
1: allocate(f_n)
2: for all children c of n do
3:   for all block-columns b of f_c do
4:     /* submit assembly of b inside f_n */
5:     submit_task(assembly, b, f_n, prio.=3)
6:   end for
7:   submit_task(cleanup, f_c, prio.=4)
8: end for
9:
10: for all block-columns p in f_n do
11:   /* submit panel factorization of p */
12:   submit_task(panel, p, prio.=2)
13:   for all block-columns u > p in f_n do
14:     /* submit update of u wrt p */
15:     submit_task(update, p, u, prio.=1)
16:   end for
17: end for
```

The second issue, instead, can be overcome by conveniently assigning different priorities to the submitted tasks according to the idea that, as long as there is enough work to do on already activated fronts, no other front should be activated. This will keep the memory consumption under control while ensuring that there are always enough tasks for all the working threads. More precisely, each activation task is assigned a negative priority, whose value depends on a

specific tree traversal order which, in our specific case, has been computed as the post-order which minimizes the memory consumption [13]. Cleanup tasks are given the highest priority because they are responsible for freeing the memory allocated by activation tasks. The other tasks, instead, are given a fixed priority which depends on the number of out-going edges in the associated DAG vertex in order to maximize the degree of concurrency; therefore, assemblies have higher priority than panels which, in turn, have higher priority than updates.

For the sake of simplicity, in Algorithm 1 we assumed that assembly operations read a single block-column b but modify an entire front f_n bur in reality only a few block-columns of f_n are modified. It has to be noted that all the assembly operations at step 4 of Algorithm 1 *(right)* are independent from each other. In fact, even if multiple assemblies write on the same block-column of f_n, their modifications concern disjoint subsets of rows (not necessarily contiguous). This property is exploited by the `qr_mumps` scheduler, which was designed on purpose for this algorithm. StarPU, instead, will assume that these assemblies are dependent from each other. As a result, not only these operations cannot be performed in parallel but are forced to be executed in the same order as they have been submitted. As shown by the experimental results of Section 5, this may entail a slight performance loss.

5 Experimental Results

The native scheduler of the `qr_mumps` software was replaced with the StarPU runtime system according to the methods described in Section 4, leading to a software package that will be referred to as `qr_starpu`. This section aims at evaluating the effectiveness of the proposed techniques as well as the performance of the resulting code. For this purpose, the behavior of the `qr_starpu` code will be compared to the original `qr_mumps` one and also, briefly, to the SuiteSparseQR package (referred to as `spqr`) released by Tim Davis in 2009 [11].

#	Mat. name	m	n	nz	op. count (Gflops)
1	tp-6	142752	1014301	11537419	277.7
2	karted	46502	133115	1770349	279.9
3	EternityII_E	11077	262144	1572792	566.7
4	degme	185,501	659415	8127528	629.0
5	cat_ears_4_4	19020	44448	132888	786.4
6	Hirlam	1385270	452200	2713200	2401.3
7	e18	24617	38602	156466	3399.1
8	flower_7_4	27693	67593	202218	4261.1
9	Rucci1	1977885	109900	7791168	12768.1
10	sls	1748122	62729	6804304	22716.6
11	TF17	38132	48630	586218	38209.3

Table 1. Matrices test set. The operation count is related to the matrix factorization with COLAMD column permutation.

The experiments were conducted on a set of matrices from the the University of Florida Sparse Matrix Collection[6] presented in Table 1. The operation count is related to the factorization preceded by a COLAMD fill-reducing matrix permutation. The tests were run on the cache coherent Non Uniform Memory Access (ccNUMA) AMD Istanbul architecture equipped with 24 cores (6×4) clocked at 2.4 GHz. The codes were compiled with the GNU v. 4.4 suite and linked to the Intel MKL sequential BLAS and LAPACK libraries. All the tests were run with real data in double precision.

Table 2 shows the factorization times (in seconds) for the matrices of the test set presented in Table 1 using `qr_starpu`, `qr_mumps` and `spqr` with different numbers of cores. Both `qr_mumps` and `qr_starpu` clearly outperform the `spqr`

		Factorization time (sec.)										
	Matrix	1	2	3	4	5	6	7	8	9	10	11
	th.											
	1	51.8	49.0	97.5	104.8	137.5	417.6	496.1	733.6	1931.0	3572.0	5417.0
`qr_starpu`	12	6.9	6.2	10.9	12.4	16.2	43.4	50.4	92.4	190.3	439.3	525.8
	24	5.7	4.4	8.0	8.5	12.4	28.1	32.9	58.0	122.7	336.3	305.9
	speedup	9.1	11.1	12.2	12.3	11.1	14.9	15.1	12.6	15.7	10.6	17.7
	1	51.5	48.8	96.9	104.6	137.1	410.8	495.2	729.7	1928.0	3571.0	5420.0
`qr_mumps`	12	5.7	5.2	10.2	10.8	14.2	39.5	46.6	69.4	177.9	392.3	479.0
	24	5.0	4.3	7.9	8.0	11.0	26.5	30.5	48.8	120.9	337.0	282.0
	speedup	10.3	11.3	12.3	13.1	12.5	15.5	16.2	14.9	15.9	10.6	19.2
	1	52.9	49.9	99.5	111.0	123.3	406.3	538.3	687.5	2081	4276	5361
`spqr`	12	17.0	14.5	26.3	33.0	32.5	85.7	90.5	131.6	468	1644	770
	24	17.0	12.3	20.7	26.2	27.8	68.6	74.1	114.2	372	1389	589
	speedup	3.1	4.0	4.8	4.2	4.4	5.9	7.3	6.0	5.6	3.1	9.1

Table 2. Factorization times, in seconds, on an AMD Istanbul system for `qr_starpu` (*top*), `qr_mumps` (*middle*) and `spqr` (*bottom*). The first row shows the matrix number.

package by a factor greater than two thanks to the powerful programming and execution paradigm based on DAG parallelism. On the other hand, `qr_starpu` is consistently but only marginally less efficient than `qr_mumps`, by a factor below 10% for eight out of eleven matrices and still only below 20% in the worst case. As a conclusion, the parallelization scheme impacts performance much more than the underlying low-level layer, validating the thesis that modular approaches based on runtime systems can compete with heavily hand-tuned codes.

Memory consumption is an extremely critical point to address when designing a sparse, direct solver. As the building blocks for designing a scheduling strategy on top of StarPU differ (and are more advanced) than what is available in `qr_mumps` (which relies on an *ad hoc* lightweight scheduler) we could not reproduce exactly the same scheduling strategy. Therefore we decided to give higher priority to reducing the memory consumption in `qr_starpu`. This cannot easily be achieved in `qr_mumps` because its native scheduler can only handle two

[6] http://www.cise.ufl.edu/research/sparse/matrices

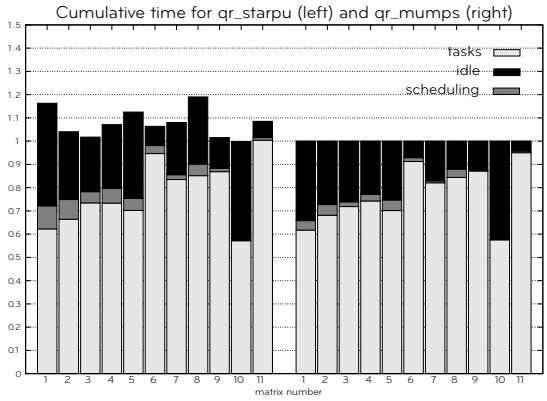

Fig. 3. Cumulative tasks time for `qr_starpu` (*left*) and `qr_mumps` (*right*) for the factorization with 24 cores. The times for `qr_starpu` are nromalized with respect to the time for `qr_mumps`

Mat.	Memory peak (MB)	
	`qr_starpu`	`qr_mumps`
1	1108.4	1426.5
2	970.7	1016.5
3	1315.1	1485.1
4	1818.8	2040.2
5	3111.2	3114.7
6	4418.2	5300.9
7	3845.1	3800.9
8	10031.0	13608.7
9	10070.8	8467.9
10	54296.0	53636.8
11	26212.7	26232.4

Table 3. Memory peak observed during the factorization of the tested matrices when using 24 cores.

levels of task priority; as a result, fronts are activated earlier in `qr_mumps`, almost consistently leading a higher memory footprint (see Table 3).

In order to explain in more details the performance behavior of `qr_starpu` and `qr_mumps`, a detailed analysis of the execution times is shown in Figure 3 and Table 4. The figure shows the cumulative times spent by all threads in the three main phases of the execution of both solvers: the time spent in tasks, the time for scheduling the tasks (this includes computing the DAG in `qr_starpu`) and the idle time spent waiting for dependencies to be satisfied; these timings will be referred to as $t_c(p)$, $t_s(p)$ and $t_i(p)$, respectively, p being the number of threads (24 in Figure 3). The efficiency $e(p)$ of the parallelization can then be defined in terms of these cumulative timings as follows:

$$e(p) = \frac{t_c(1)}{t_c(p)+t_s(p)+t_i(p)} = \overbrace{\frac{t_c(1)}{t_c(p)}}^{e_l} \cdot \overbrace{\frac{t_c(p)}{t_c(p)+t_s(p)}}^{e_s} \cdot \overbrace{\frac{t_c(p)+t_s(p)}{t_c(p)+t_s(p)+t_i(p)}}^{e_p}.$$

This expression allows us to decompose the efficiency as the product of three well identified effects: e_l which measures the impact of data locality issues on the efficiency of the tasks, e_s which measures the cost of the scheduler management with respect to the actual work done and e_p which measures how well the tasks have been pipelined as a result of the scheduling decisions. Table 4 shows the corresponding values for our matrix collection.

The cumulative tasks times $t_c(1)$ (not reported here for the sake of space) are nearly identical for the two codes and stay the same when the number of threads increases as shown in Figure 3 and by the fact that the e_l values in Table 4 are comparable. The difference in the overall execution time can be explained by the higher overhead imposed by the runtime system management and by the idle time. The overhead imposed by StarPU is higher (inducing a lower efficiency e_s

in Table 4) because the dependencies between tasks have to be inferred based on the data access modes whereas in `qr_starpu` they are all defined explicitly based on the knowledge of the algorithm. However, the cost of the scheduling grows moderately with the number of threads and, in any case, only accounts for a very small part of the overall execution time, especially for large scale matrices. The increased cumulative idle time, and the resulting lower pipeline e_p efficiency, are, instead, due to two factors. First, the constraints imposed in `qr_starpu` to limit the memory consumption yield slightly lower concurrency (and, therefore, more idle time). Second, as explained in the previous section, assembly operations are serialized in `qr_starpu`; although these tasks only account for a small portion of the overall execution time, their serialization may induce delays in the pipeline that lead some threads to starvation. This second issue could be overcome by specifying to the runtime system that assembly tasks can be executed in any order and, possibly, in parallel, but this feature is not currently available in StarPU. Finally, Table 5 shows the maximum number of

qr_starpu												
Matrix		1	2	3	4	5	6	7	8	9	10	11
	th.											
e_l	12	0.816	0.779	0.789	0.820	0.809	0.808	0.865	0.828	0.844	0.822	0.885
	24	0.711	0.680	0.669	0.730	0.694	0.666	0.768	0.689	0.733	0.752	0.774
e_p	12	0.860	0.856	0.906	0.877	0.876	0.976	0.918	0.797	0.958	0.802	0.952
	24	0.621	0.720	0.769	0.744	0.671	0.923	0.792	0.758	0.870	0.575	0.936
e_s	1	0.984	0.985	0.994	0.987	0.987	0.990	0.996	0.987	0.998	0.999	0.997
	12	0.915	0.930	0.970	0.951	0.953	0.974	0.977	0.966	0.991	0.997	0.993
	24	0.863	0.887	0.938	0.921	0.931	0.965	0.976	0.944	0.983	0.996	0.989
e	12	0.642	0.620	0.693	0.684	0.675	0.768	0.776	0.637	0.801	0.657	0.837
	24	0.381	0.434	0.482	0.500	0.433	0.593	0.594	0.493	0.627	0.431	0.716
qr_mumps												
Matrix		1	2	3	4	5	6	7	8	9	10	11
	th.											
e_l	12	0.851	0.844	0.844	0.854	0.832	0.862	0.891	0.850	0.882	0.881	0.921
	24	0.711	0.661	0.678	0.716	0.690	0.688	0.780	0.695	0.727	0.737	0.808
e_p	12	0.915	0.915	0.898	0.936	0.922	0.977	0.937	0.985	0.963	0.812	0.992
	24	0.658	0.727	0.739	0.771	0.747	0.929	0.829	0.880	0.874	0.578	0.957
e_s	1	0.998	0.996	0.999	0.997	0.998	0.999	0.999	0.999	1.000	1.000	1.000
	12	0.949	0.973	0.989	0.985	0.963	0.977	0.995	0.982	0.997	0.998	0.997
	24	0.939	0.937	0.973	0.963	0.939	0.982	0.990	0.959	0.996	0.996	0.993
e	12	0.739	0.751	0.749	0.787	0.738	0.823	0.830	0.822	0.847	0.714	0.910
	24	0.439	0.450	0.487	0.532	0.484	0.628	0.640	0.586	0.633	0.424	0.768

Table 4. Efficiency measures e_l, e_p, e_s and e for `qr_starpu` and `qr_mumps` ($e = e_l.e_p.e_s$).

tasks that the runtime system handles during the factorization versus the total number of tasks executed. The first being between 3 and 11 times smaller than the second, these data prove that the technique proposed in Section 4 is effective in reducing the runtime system overhead.

	DAG size										
Matrix	1	2	3	4	5	6	7	8	9	10	11
Max.	1640	1899	2023	2969	5063	4442	6965	12773	6846	11592	32978
Total	8610	10202	6058	14901	26579	49013	21192	136412	41023	33211	187101

Table 5. Maximum DAG size handled by StarPU during the factorization of the test matrices when using 24 threads.

6 Conclusions and future work

The main objective of this work was to evaluate the usability and effectiveness of general-purpose runtime systems for parallelizing sparse factorization methods which constitute a complex and irregular workload. This was assessed implementing a new package software, `qr_starpu`, derived from `qr_mumps` by relying on a modern robust runtime system, StarPU, instead of the original *ad hoc* scheduler. Due to the original features of the considered algorithm, special attention had to be paid to the submission of tasks in order to contain the memory consumption, to limit the overhead imposed by the runtime system and to circumvent some limitations of StarPU (common to many other modern runtime environments). As a result, we managed to achieve an excellent memory behavior (even better than the original `qr_mumps` solver) and a very competitive performance, the overhead on elapsed time being most of the time below 10% and in any case never higher than 20%. A detailed analysis has revealed that this difference can be explained with a higher overhead imposed by the runtime system (which, however, only accounts for a very small part of the total execution time) and a more conservative scheduling of tasks to achieve a lower memory consumption.

All in all, the marginal performance loss conjugated with the excellent memory behavior show that general purpose runtime systems are very well suited for the parallelization of sparse direct methods. These powerful tools, moreover, provide several features that are likely to offer better performance and portability on architectures with higher core counts or equipped with accelerating devices (such as GPUs or MICs). These features has been evaluated in this paper and their usage is the object of ongoing research. At the same time, this document provides guidelines for the improvement of both sparse direct methods and runtime environments. Because, already on 24 cores, a considerable fraction of time is spent waiting for dependencies to be satisfied, it may be beneficial to adopt algorithms by tiles [10] for the processing of fronts in order to improve the amount of concurrency. Runtime systems, instead, can be improved by adding features that allow to cope with memory-consuming tasks or that allow to infer dependencies based on the access to memory that has not been allocated yet.

References

1. 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(1):012037, 2009.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.
3. P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Int. Journal of Num. Linear Alg. and Appl.*, 3(4):275–300, 1996.
4. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.
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, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.
6. R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.
7. G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.
8. G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra. Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. *Scalable Computing and Communications: Theory and Practice*, 2013.
9. A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices, 2013. To appear on the SIAM Journal on Scientific Computing.
10. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Par. Comp.*, 35(1):38–53, 2009.
11. T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Softw.*, 38(1):8:1–8:22, December 2011.
12. I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
13. A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.
14. E. Hermann, B. Raffin, F. Faure, T. Gautier, and Jérémie Allard. Multi-GPU and multi-CPU parallelization for interactive physics simulations. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 2010.
15. J. Hogg, J. K. Reid, and J. A. Scott. A DAG-based sparse Cholesky solver for multicore architectures. Technical Report RAL-TR-2009-004, RAL, 2009.
16. X. Lacoste, P. Ramet, M. Faverge, I. Yamazaki, and J. Dongarra. Sparse direct solvers with accelerators over DAG runtimes. Research report RR-7972, INRIA, 2012.
17. G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3), 2009.
18. R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.
19. H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.