Exploiting a Parametrized Task Graph model for the parallelization of a sparse direct multifrontal solver

Emmanuel Agullo[1], George Bosilca[5], Alfredo Buttari[2], Abdou Guermouche[3], and Florent Lopez[4]

[1] INRIA - LaBRI, Bordeaux (France)
[2] CNRS - IRIT, Toulouse (France)
[3] Université de Bordeaux - LaBRI, Bordeaux (France)
[4] RAL - STFC, Didcot (UK)
[5] University of Tennessee Knoxville (USA)

Abstract. The advent of multicore processors requires to reconsider the design of high performance computing libraries to embrace portable and effective techniques of parallel software engineering. One of the most promising approaches consists in abstracting an application as a directed acyclic graph (DAG) of tasks. While this approach has been popularized for shared memory environments by the OpenMP 4.0 standard where dependencies between tasks are automatically inferred, we investigate an alternative approach, capable of describing the DAG of task in a distributed setting, where task dependencies are explicitly encoded. So far this approach has been mostly used in the case of algorithms with a regular data access pattern and we show in this study that it can be efficiently applied to a higly irregular numerical algorithm such as a sparse multifrontal QR method. We present the resulting implementation and discuss the potential and limits of this approach in terms of productivity and effectiveness in comparison with more common parallelization techniques. Although at an early stage of development, preliminary results show the potential of the parallel programming model that we investigate in this work.
Keywords: multicore architectures, programming models, runtime system, parametrized task graph, numerical scientific library, sparse direct solver, multifrontal QR factorization.

1 Introduction

Since their introduction, multicore processors have become increasingly popular and are nowadays a commodity used beyond the high performance computing (HPC) community. However, there is no clear consensus on the best practices for programming such architectures and developers often have to make a trade-off between productivity (the pace at which a code may be written and maintained) and performance (the pace at which the code is eventually executed). For instance, some software developers may choose to limit the parallelization of their

code to the introduction of a few OpenMP pragma directives within the main computational-intensive loops of their algorithms. On the other end of the spectrum, highly optimized libraries such as linear algebra numerical kernels are often written with low-level synchronizations schemes relying on POSIX threads (pthread) primitives at a possible high cost in terms of development and maintenance. One of the most promising approach for enhancing the productivity while maintaining high performance consists in abstracting an application as a directed acyclic graph (DAG) of tasks and delegating the orchestration of the task to a runtime system.

Whereas task-based runtime systems were mainly research tools in the past years, their recent progress make them now a solid candidates for designing advanced scientific software. They provide programming paradigms that allow the programmer to express concurrency in a simple yet effective way and relieve her from the burden of dealing with low-level architectural details. Runtime systems offer a uniform programming interface for a specific subset of hardware or low-level software entities (e.g., pthread implementations). They are designed as thin user-level software layers that complement the basic, general purpose functions provided by the operating system. Applications then target these uniform programming interfaces in a portable manner and low-level, hardware dependent details are hidden inside runtime systems. The adaptation of runtime systems is commonly handled through drivers. Portability is thus enabled by the abstraction provided by the runtime system.

All the above mentioned efforts have contributed to proving the ease of use, the effectiveness and portability of general purpose runtime systems to the point where the OpenMP board has decided to include similar features since the 4.0 standard: the `task` construct was extended with the `depend` clause which enables the OpenMP runtime to automatically detect dependencies among tasks and consequently schedule them accordingly. While task-based programming has been popularized with OpenMP 4.0 where dependencies between tasks are automatically inferred, the concept itself is much older, and provided in varied forms by several research projects. In the context of this work we investigate an alternative approach consisting of explicitly encoding the dependencies between tasks. Many studies [3, 11, 19, 1] have shown the potential of the approach in the case of relatively regular algorithms such as dense linear algebra. On the other hand, the effort for assessing it on irregular algorithms is much more narrow [20, 21]. In this paper, we consider a highly irregular numerical algorithm, namely the sparse multifrontal QR method, and we show how we can turn it out into a DAG of tasks with explicit dependencies. We present the resulting code and discuss the potential and limits it delivers in terms of productivity and effectiveness in comparison with more common parallelization techniques.

The rest of the paper is organized as follows. Section 2 presents the related work on task-based programming models and runtime systems as well as numerical libraries that have been developed on top of them, including the model we want to highlight in this paper (consisting in explicitly defining the dependencies of the DAG) together with the runtime system we use (PaRSEC [10, 9]) to

support it. We then present the highly irregular numerical method we want to implement (namely, the multifrontal QR method) to illustrate our discussion in Section 3. We show how it can be written as a DAG of tasks with explicit dependencies in Section 4 and present preliminary (but encouraging!) performance results in Section 5. Section 6 concludes the paper and present perspectives.

2 Related work

2.1 Parallel programming models for task-based algorithms

The most common strategy for the parallelization of task-based algorithms consists in traversing the DAG sequentially and submit the tasks, as discovered, to the runtime system using a non blocking function call. The dependencies between tasks are automatically inferred by the runtime system through a data dependency analysis [4] and the actual execution of the task is then postponed to the moment when all its dependencies are satisfied. This programming model is known as a *Sequential Task Flow* (STF) model as it fully relies on `sequential consistency` for the dependency detection. This paradigm is also sometimes referred to as *superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies. As mentioned above, the popularity of this model encouraged the OpenMP board to include it in the 4.0 standard. The simplicity of the STF model facilitates the design of numerical algorithms in a concise manner and can be exploited to efficiently target multicore architectures [2].

One challenge in scaling to large scale distributed many-core systems is how to represent extremely large DAGs of tasks in a compact fashion. The *Parameterized Task Graph* (PTG) model introduced in [14] addresses this issue. In this model, tasks are not enumerated as in the STF model but parametrized and the dependencies between tasks are explicitly expressed. This property can be used to encode the DAG in a compact, size independent, way inducing a lower memory footprint for its representation as well as ensuring limited complexity for parsing it as the problem size grows. For this reason the memory consumption overhead in the runtime system for representing the DAG is much lower for the PTG model than for the STF model. In addition with a STF model the DAG has to be completely unrolled on all participating processes whereas with a PTG the DAG is only partially unfolded during the execution following the task progression. From this point of view, the advantage of the PTG approach over the STF can be crucial when exploiting processors with a very large number of cores. We address this particular model in the present paper.

2.2 Task-based runtime systems for modern architectures

Many initiatives have emerged in the past years to develop efficient task-based runtime systems for modern platforms. Their review is out of the scope of this paper. We mention two important projects supporting the STF model. The StarSs

project is actually an umbrella term that describes both the StarSs language extensions and a collection of runtime systems targeting different types of platforms [7, 6]. StarSs provides an annotation-based language which extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system. The StarPU runtime system provides a generic interface for developing parallel, task-based applications. It supports multicore architectures equipped with accelerator as well as distributed memory systems. This runtime is capable of transparently handling data and provides a rich panel of features.

The PaRSEC runtime system provides a distributed generic task scheduler supplemented by programming interface complying to the two main programming models presented in Section 2.1. In particular, it is one of the few (and certainly the most popular) runtime systems supporting the PTG model. The embedded scheduler is dynamic, designed to exploit the memory hierarchy of modern architectures and capable of maximizing computation to communication overlap, exploiting data locality and achieving load-balancing between the resources. PaRSEC provides a language called Job Data Flow (JDF) providing an extended PTG expressivity to parallel codes. During the compilation process, the files containing the JDF code are translated into C-code files by a specific compiler distributed with PaRSEC called *daguepp*. The DAG is defined by a set of *task types* that can be associated with several *parameters* defined on a given *range* of values. The tasks are associated with a list of *predecessors* and *successors* that define the dependencies in the DAG. These dependencies are generally based on data but may also represent precedence constraints. Tasks are associated with a code that will be executed for each task instance. This task code can have multiple instances, each tied to specific hardware resources (accelerators, FPGA, ...), and the runtime will select the most appropriate one dynamically depending on the availability of resources and the needs of the algorithm. For more information we redirect the interested reader to [10, 8, 9, 11].

3 Multifrontal QR method

The multifrontal method, introduced by Duff *et al.* [17] is a method for the factorization (either Cholesky, LDL^T, LU or QR) of sparse, linear systems. This algorithm is based on the concept of *elimination tree* [22] expressing the dependencies between the operations which eliminate the unknowns of the input matrix A, each vertex f of the tree being associated with k_f of these unknowns. The coefficients of the corresponding k_f columns and all the other coefficients concerned by their elimination are assembled together into a dense matrix, called *frontal matrix* or, simply, *front*, associated with the tree node. An edge of the tree represents a dependency between such fronts. The elimination tree is thus a topological order for the elimination of the unknowns: a front can only be eliminated after its children. The multifrontal QR factorization then consists in a tree traversal following a **topological order** for eliminating the fronts. When a front is visited, first, the **activation** operation allocates and initializes

the front data structure. Next, the front can then be **assembled**, i.e., filled up with the coefficients in the associated k_f rows of the matrix A and with coefficients resulting from the factorization of child nodes. Once assembled, the k_f unknowns are eliminated through a **complete, dense** QR **factorization** of the front. This produces k_f 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 coefficients that will be assembled into the parent front together with the contribution blocks from all the sibling fronts.

One distinctive feature of the multifrontal QR factorization is that frontal matrices are not entirely full but, prior to their factorization, can be permuted into a staircase structure that allows for moving many zero coefficients in the bottom-left corner of the front and for ignoring them in the subsequent computation; this allows for a considerable saving in the number of operations. It must be noted that when handling matrices from real-life applications, elimination trees can be quite large (i.e., contain up to $O(10^4)$ nodes), irregular and unbalanced, frontal matrices can be of varying sizes (from a few units up to $O(10^4)$ rows or columns) and shapes (either over or under-determined). We refer to [5, 15, 12] for further details on the multifrontal QR method.

Because of what said above, the multifrontal factorization results in an extremely irregular, heterogeneous and unpredictable workload even in the case where a regular partitioning is applied to fronts. Therefore its implementation on modern supercomputers is a challenging task. In this work we investigate the use of PTG based runtime systems for this method and assess their ease of use and effectiveness.

4 Design of a task-based multifrontal QR factorization with explicit dependencies

The multifrontal method provides two distinct sources of concurrency: **tree-level** and **node-level** parallelism. The first one stems from the fact that fronts in separate branches are independent and can thus be processed concurrently; the second one from the fact that, if a front is large enough, multiple processes can be used to assemble and factorize it.

In order to exploit both sources of parallelism; in the proposed implementation of our PTG-based parallel multifrontal factorization, which we refer to as `qrm_parsec`, we use an approach based on hierarchical DAGs. We consider a two-level hierarchy with an outer DAG and multiple inner DAGs spawned by the tasks in higher level DAG. The outer DAG contains tasks related to the activation, assembly and deactivation of fronts in the elimination tree whereas each inner DAG contains the tasks related to the factorization of the frontal matrix. This approach is illustrated in Figure 4 where three different DAGs denoted by 1, 2 and 3 are spawned by tasks in the outer DAG.

The PaRSEC implementation of our solver is split into three JDF files described in the next sections.

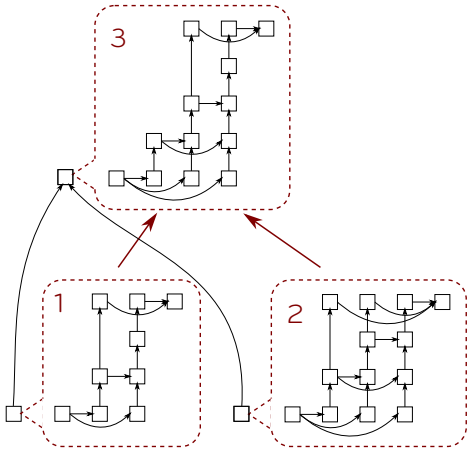

Fig. 1. Two levels hierarchical DAGs implemented in PaRSEC. The inner DAGs are spawned by tasks contained in the top level DAG.

4.1 The Factorization

This JDF file represents the DAG operating at the elimination tree level and contains the description of four tasks:

- `activate`: allocates the memory needed for assembling and factorizing a frontal matrix. The activation of a node depends on the activation of its children.
- `assemble`: spawns a lower level DAG of tasks performing the assembly of the frontal matrix in parallel; this DAG is defined in the `assembly.jdf` file described below. It depends on the activation of the related node and is completed when all the spawned tasks have been executed.
- `init`: initializes the frontal matrix data structure and spawns the lower level DAG performing the front factorization which is described below. This task depends on the assemble tasks as the front factorization can only start once it has been assembled. As for the assemble task, its completion is achieved when the DAG it spawn is completely executed.
- `deactivate`: stores apart the result of a front factorization and frees the memory allocated by the activate task. It can be executed only after its contribution block has been assembled into the parent node.

An excerpt of the `factorization.jdf` file is shown in Figure 2 where, for the sake of simplicity, we only describe the `init` task. This task needs a set of symbolic data denoted `S` in the data-flow which is provided by the `assemble` task. Note that in the case were the front has no children, the `assemble` task perform no operations apart from passing the symbolic data to the init task. When the `init` tasks is completed, the front is factorized and can be assembled into the parent node. Therefore we transfer the symbolic data to the assembly operation of the parent node.

```
init(n)

  n = 1 .. NN

  /* get info on frontal matrix */
  front = inline_c %{ return get_front(n); %}
  p     = inline_c %{ return get_front_parent(n); %}
  prio  = inline_c %{ return get_front_prio(front); %}

  RW S <- S assemble(n) /* initialize the front  assembly */
       -> (p != 0) ? S assembly(p)

BODY
{
   /* initialize frontal matrix*/
   _qrm_init_front(front);
   /* create qr factorization DAG for frontal matrix */
   qr_handle = qr_initialize(front);
   /* submit front factorization to PaRSEC */
   dague_enqueue(qr_handle);
}
END
```

Fig. 2. Excerpt of code for performing operation at elimination tree level with PaRSEC.

Each of these tasks are executed once for every node in the elimination tree.

4.2 `qr_1d.jdf, qr_2d.jdf`

Once assembled, a frontal matrix can be factorized using any QR factorization algorithm for dense matrices. For this operation, we have chosen two different variants, namely, a LAPACK-style factorization based on a 1D partitioning of the front in block-columns and a Communication Avoiding method based on a 2D partitioning into tiles [18, 13]. For a matter of conciseness we only present the 1D version (`qr_1d.jdf`) of the code. These implementations are based on the ones found in the DPLASMA library [8] which provide dense linear algebra kernels routine for distributed systems built on top of the PaRSEC runtime systems. We adapted these kernels to the specific staircase structure of frontal matrices described in Section 3.

The JDF code for the QR factorization with a 1D block-column partitioning is presented in Figure 3. In this JDF we have two type of task: the `geqrt` task corresponding to the panel operations and the `gemqrt` corresponding to update operations with respect to panel reductions. Note that this JDF is similar to the DPLASMA implementation except that we used the `_geqrt_stair` and `_gemqrt_stair` kernels, respectively for the panel and update operations, capable of exploiting the staircase structure of block-columns. The `geqrt` tasks are associated with the panel index represented by the parameter `p` which has values in the range `0..NP-1` where `NP` represents the number of panel operations in the front. Similarly, the `gemqrt` task is defined by two parameters. The first represents the panel operations and the second represents the subsequent update operations depending on each panel operation. For each panel operation `p` we

perform update operations on block-columns `p+1..NC-1` where `NC` is the total number of block-columns in the frontal matrix. Along with a R factor resulting from the `geqrt_stair` operation, the `geqrt` task produce a `V` and `T` data that are sent to the subsequent update tasks which are represented by `gemqrt(p, p+1..NC-1)` in the JDF code. Concerning the `gemqrt` tasks, for a given a block-column `u`, it retrieves the `V` and `T` matrices of the corresponding panel `p` along with the block-column issued by the update with respect to the previous panel task denoted `gemqrt(p-1,u)`. Once the update operation has been executed, the block-column is sent either to the next update operation denoted `gemqrt(p+1,u)` or to the panel operation denoted `geqrt(u)` if the block-column is up-to-date.

As for the 2D code, because of the fronts staircase structure, some tiles are equal to zero and must be skipped in the computation which alters the data-flow with respect to the methods described in the literature. In PaRSEC this can be conveniently handled by using conditional expressions in the JDF.

```
geqrt(p)

  p = 0 .. (NP-1)

  RW A_p <- (p==0) ?  A(0,p) : C_u gemqrt(p-1, p)
         -> (p < NC-1) ? V_p gemqrt(p, (p+1)..(NC-1))
         -> A(p)

  RW T_p <- T(p)                                              [type = LITTLE_T]
         -> (p < NC-1) ? T_p gemqrt(p, (p+1)..(NC-1))         [type = LITTLE_T]
         -> T(p)

BODY
{
   _geqrt_stair(&m, &n, &ib, &stair[off], &off, A_p + off,
                &lda, T_p, &ldt, work, &info);
}
END

gemqrt(p, u)

  p = 0..(NP-1)
  u = (p+1)..(NC-1)

  READ  V_p <- A_p geqrt(p)
  READ  T_p <- T_p geqrt(p)                                   [type = LITTLE_T]
  RW    C_u <- (p==0) ? A(0,u) : C_u gemqrt(p-1, u)
            -> ((u == p+1) && (u <= (NP-1))) ? A_p geqrt(u)
            -> ((u > p+1) && (p < (NP-1))) ? C_u gemqrt(p+1, u)

BODY
{
   _gemqrt_stair("l", "t", &m, &j, &k, &ib, &stair[off], &off, V_p + off,
                 &ldv, T_p, &ldt, C_u  + off, &ldc, work, &info);
}
END
```

Fig. 3. Code for the 1D block-column dense QR factorization with PaRSEC.

4.3 `assembly.jdf`

In the DAG instantiated by this assembly operations, each task corresponds to the assembly of a block from all the blocks in children's frontal matrices contributing to it. Note that in order to express the data-flow for these assemblies, we need to compute, for every block in a frontal matrix, a list of contributing blocks in children node. This mapping is computed upon front activation and is not required when using a STF model.

4.4 Discussion

It must be noted that it is possible to execute some of the factorization tasks related to a node before the handling of its child nodes is completed; this additional concurrency, which we refer to as *inter-level parallelism*, may lead to considerable benefits especially in the case of narrow and unbalanced elimination trees where tree parallelism is scarce. `qrm_parsec` cannot use inter-level parallelism because the factorization DAG is spawned only once the front is fully assembled due to the dependency between the init and the assemble tasks described above. Although technically possible, using inter-level parallelism is more complex to achieve with the PTG model than with the STF one where the expression of dependencies is simpler when the DAG is defined dynamically, as it's partially the case in the multifrontal method. This is the subject of ongoing research.

In practical cases the elimination tree may have thousands of nodes and thus the DAG may contain millions of tasks; this much concurrency is clearly useless for the systems targeted by this work which only include few cores. In order to reduce the size of the DAG, entire subtrees at the bottom of the elimination tree are handled at once within a single task. This technique is very well known in the domain of sparse, direct methods and provides considerable benefits in terms of reduced runtime overhead as well as improved data locality.

Most of the execution time is spent in BLAS-3 operations (like dense matrix multiplications). Because these have a favorable ratio between computations and data access, the whole multifrontal factorization can be considered as *compute bound* and thus the effects of memory contention can be considered light.

5 Early experimental results

We evaluate the PTG implementation of our solver on a set of test matrices presented in Table 1 from real world applications publicly available in the University of Florida Sparse Matrix Collection[16]. We also use the hirlam matrix, from the HIRLAM[6] research program. The PaRSEC runtime system is used to support the PTG model. The COLAMD fill-reducing column permutation was applied to all the matrices. The runs were performed on the Dude system which is a shared-memory machine equipped with four AMD Opteron(tm) Processor

[6] http://hirlam.org

id	Mat. name	m	n	nz	op. count (Gflop)	Time (sec)	Mem (GB)
1	karted	46502	133115	1770349	257	46.3	0.7
2	degme	185501	659415	8127528	591	103.2	1.4
3	cat_ears_4_4	19020	44448	132888	716	134.9	1.2
4	hirlam	1385270	452200	2713200	2339	392.0	3.5
5	e18	24617	38602	156466	3399	474.5	3.5
6	flower_7_4	27693	67593	202218	4261	774.7	3.6
7	Rucci1	1977885	109900	7791168	12764	1786.0	5.1
8	TF17	38132	48630	586218	38209	5185.0	15.3

Table 1. The set of matrices used for the experiments along with the associated sequential factorization time and memory consumption.

8431 (six cores) and 72 GB of memory. As a reference, we also report on the performance of the STF implementation of the solver from [2], which is supported with StarPU and named `qrm_starpu` below.

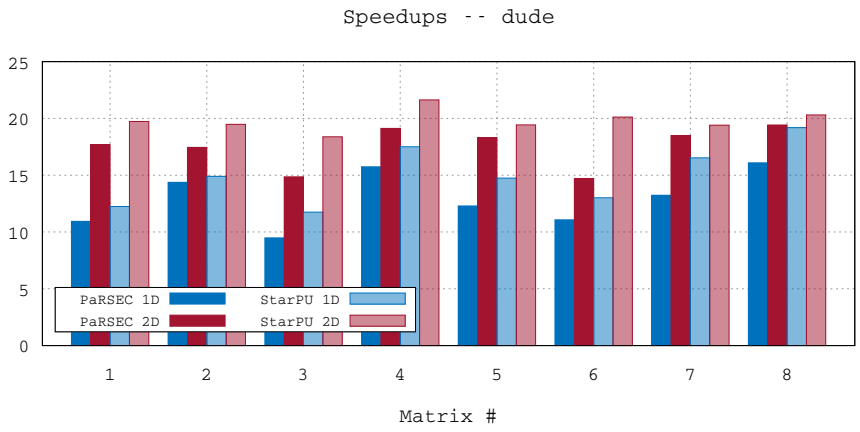

Fig. 4. Speedup of `qrm_starpu` and `qrm_parsec` on the Dude system (24 cores).

The experimental results are presented in Figure 4. They show the scalability of `qrm_parsec` using both the 1D and 2D front factorization algorithms; the speedups are computed with respect to the sequential running time reported in Table 1. These are compared to the results obtained with an equivalent implementation based on the Sequential Task Flow model and the StarPU runtime system [2]. The results show that `qrm_parsec` achieves a satisfactory performance on all the tested matrices, including the smallest ones (on the left side of the plot) with speedups close to 20 (out of 24) for the largest size ones. Figure 4 also

shows that the 2D Communication Avoiding front factorization variant achieves much better speedups than the 1D block-column one; this is expected since most of the frontal matrices in the multifrontal QR method are (strongly) overdetermined and thus the 1D method simply does not provide enough concurrency (especially for smaller size matrices).

Finally, the STF implementation consistently achieves better performance than the PTG-based one. This difference comes mostly from the fact that the STF code exploits the inter-level parallelism mentioned in Section 4.4. Note, also, that `qrm_parsec` is a proof of concept code whereas the StarPU-based implementation is fully optimized; therefore the performance gap could be partly reduced through code optimization.

6 Concluding remarks

In this paper, we have investigated the impact on programmability and discussed the potential in terms of performance of programming a highly irregular numerical algorithm as a task-based DAG of tasks with explicit dependencies. We have shown that providing the dependencies is not trivial and requires a deep understanding of the parallelism available in the algorithm. However, thanks to the task-based abstraction, this model provides an interesting alternative to STF as it allows to write the high-level algorithm independently of the underlying processor, delegating the burden of handling synchronizations to the runtime system. Furthermore, we have shown that the considered model provides a lot of flexibility at runtime to instantiate the most appropriate variant of an algorithm (such as 1D and 2D kernels in this study). Although the performance results are preliminary, they are very encouraging. We would be delighted to present them and discuss them in a workshop that aims at making parallelism available to a wide range of applications using systematic software engineering methodology, beyond the scope of numerical, scientific libraries.

References

1. E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A hybridization methodology for high-performance linear algebra software for GPUs. *in GPU Computing Gems, Jade Edition*, 2:473–484, 2011.
2. E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems. *ACM Transactions On Mathematical Software*, 2016. To appear.
3. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1):012037, 2009.
4. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.
5. 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.

6. E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par*, pages 851–862, 2009.
7. 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.
8. 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*, pages 1432–1441, Anchorage, United States, May 2011.
9. 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(6):36–45, 2013.
10. 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.
11. 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*, pages 699–733, 2013.
12. A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. *SIAM Journal on Scientific Computing*, 35(4):C323–C345, 2013.
13. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35:38–53, January 2009.
14. M. Cosnard and M. Loi. Automatic task graph generation techniques. In *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, volume 2, pages 113–122 vol.2, Jan 1995.
15. T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Softw.*, 38(1):8:1–8:22, Dec. 2011.
16. T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.
17. 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.
18. B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tile QR factorization with parallel panel processing for multicore architectures. In *IPDPS*, pages 1–10. IEEE, 2010.
19. 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(9):1134–1143, 2012.
20. K. Kim and V. Eijkhout. A parallel sparse direct solver via hierarchical DAG scheduling. *ACM Trans. Math. Softw.*, 41(1):3:1–3:27, Oct. 2014.
21. X. Lacoste. *Scheduling and memory optimizations for sparse direct solver on multi-core/multi-GPU cluster systems.* PhD thesis, LaBRI, Université Bordeaux, Talence, France, Feb. 2015.
22. R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions On Mathematical Software*, 8:256–276, 1982.