A runtime approach to dynamic resource allocation for sparse direct solvers

A.-E Hugo A. Guermouche P.-A. Wacrenier R. Namyst
INRIA, LaBRI, University of Bordeaux
Talence, France

***Abstract*—To face the advent of multicore processors and the ever increasing complexity of hardware architectures, programming models based on DAG-of-tasks parallelism regained popularity in the high performance, scientific computing community. In this context, enabling HPC applications to perform efficiently when dealing with graphs of parallel tasks that could potentially run simultaneously is a great challenge. Even if a uniform runtime system is used underneath, scheduling multiple parallel tasks over the same set of hardware resources introduces many issues, such as undesirable cache flushes or memory bus contention. In this paper, we show how runtime system-based scheduling contexts can be used to dynamically enforce locality of parallel tasks on multicore machines. We extend an existing generic sparse direct solver to use our mechanism and introduce a new decomposition method based on proportional mapping that is used to build the scheduling contexts. We propose a runtime-level dynamic context management policy to cope with the very irregular behavior of the application. A detailed performance analysis shows significant performance improvements of the solver over various multicore hardware.**

I. 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 most of the underlying hardware performance, HPC applications need to be strongly optimized. This usually leads to hand-tuned complex source code that copes both with algorithmic and architecture concerns. Thus, high performance is achieved at the price of a tremendous development effort and a very poor maintainability. Typically, hardware- and topology-dependent optimizations impact both algorithms and data layouts of applications, and lead to poor performance portability.

For these reasons, an increasing number of parallel libraries or applications follow a more flexible approach by adopting a layered architecture. For instance, many numerical algorithms are described at a high level independently of the hardware architecture as a Directed Acyclic Graph (DAG) of tasks where each vertex represents a task and each 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. The MAGMA library [44], that provides Linear Algebra algorithms on heterogeneous hardware by relying on the StarPU runtime system to perform dynamic scheduling between CPUs and GPUs, well illustrates this trend toward delegating scheduling to the underlying runtime system. Moreover, such libraries often exhibit state-of-the-art performance, resulting from heavy tuning and strong optimization efforts.

Many research efforts have recently been devoted to the design of runtime systems able to provide programmers with portable techniques and tools to exploit such complex hardware. The availability of mature implementation of such runtime systems (e.g. Cilk [21], OpenMP or Intel TBB [39] for multicore computers, Anthill [42], DAGuE [11], Charm++ [29], Harmony [18], KAAPI [25], Qilin [35], StarPU [6] or StarSs [8] for heterogeneous configurations) has allowed programmers to rely on thread/task facilities to develop efficient implementations of parallel libraries (e.g. Intel MKL [16], FFTW [20]).

A number of recent efforts have been focusing on redesigning HPC applications to use such runtime systems. As an example, several sparse direct solvers have been redesigned on top of task-based runtime systems [3], [32] and exhibit high performance and improved portability. Similar efforts have also been undertaken in the field of fast multipole method computations [2].

In this paper, we investigate how to push further the interaction between the application and the runtime system. By allowing the application to provide informations about the structure of its task graph to the runtime, the latter is able to perform a better mapping on the underlying topology and the whole stack behaves better. We consider a sparse direct solver, namely the `qr_mumps` solver [3], and provide an improved interaction scheme which enhances locality and guides the behavior of the underlying runtime system. We believe that this approach can easily be extended to a larger application domain.

The contributions of this paper are as follows:

- We introduce a new partial static mapping strategy for sparse direct solvers based on scheduling contexts.
- We propose a new dynamic management strategy for the scheduling contexts aiming at improving the performance of applications having hierarchical task graphs.
- We present performance results that show how our solution proves great potential in improving the behavior of the parallel sparse direct solver.

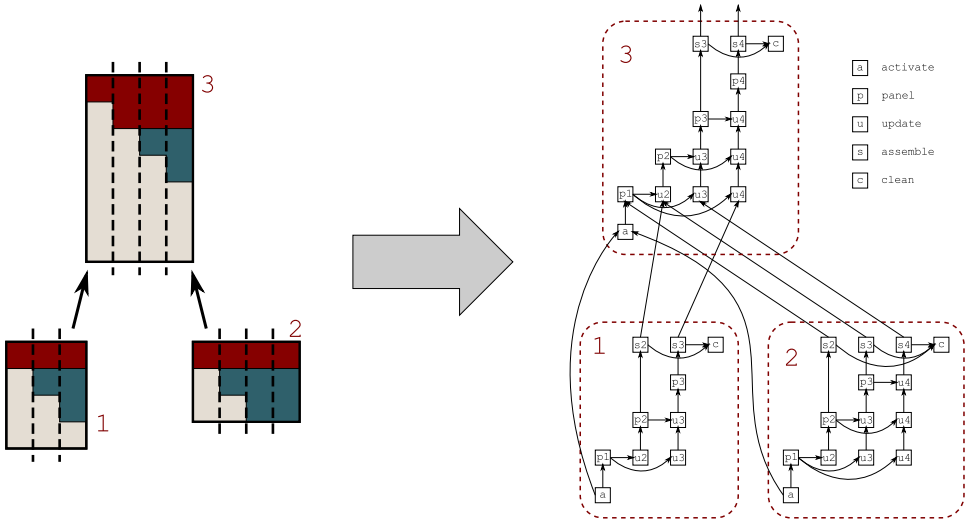


Fig. 1: An example of how a simple elimination tree with three nodes is transformed into a DAG in the `qr_mumps` code. Vertical, dashed lines show the partitioning of fronts into block-columns. Dashed-boxes group together all the tasks related to a front.

II. Motivating application: Sparse QR factorizations

The sparse QR factorization method which we consider in this section is the multifrontal method. Like other direct methods, the multifrontal algorithm is based on an elimination tree [41], which is the transitive reduction of the filled matrix graph and represents the dependencies between the elimination operations. The task graph is built during the so-called analysis phase where all the preprocessing algorithms are applied. This phase is then followed by the actual numerical factorization and the solve steps. This graph, which has a number of nodes which is typically one order of magnitude 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.* [5].

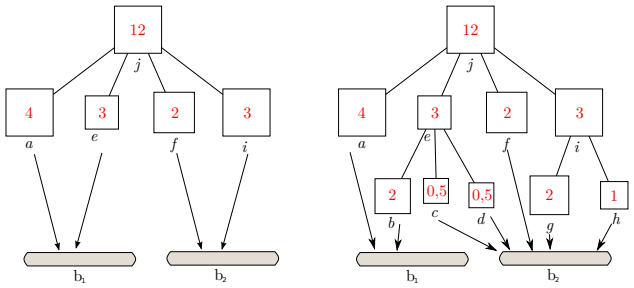
The baseline of the `qr_mumps` solver, is the parallelization model proposed by Buttari [13] which is based on the approach presented earlier in related work on dense matrix factorizations by Buttari *et al.* [14] and extended to the supernodal Cholesky factorization of sparse matrices by Hogg *et al.* [26]. 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 [13] from which this example was taken. The execution of the tasks is guided by a dynamic scheduler which allows the tasks to work asynchronously.

Recently, Buttari *and al.* have proposed a modified version of the `qr_mumps` software [3] which was designed on top of the StarPU runtime system. The idea was mainly to delegate all the parallelism management to StarPU (e.g. scheduling, task dependencies, etc.). In the following parts of the paper, we will extend this work and illustrate how to improve the interaction between the runtime system and the application will improve the general behavior of the solver.

III. A partial task graph mapping strategy

In this section we present a partial mapping strategy based on the use of scheduling contexts to ensure a good locality and an improved load-balancing among the computational

resources. We remind that the task graph considered in this paper is tree-shaped and composed of parallel tasks (see Section II) and that the algorithm proposed below is done during the analysis phase of the sparse direct solver. The idea is to use a classical static scheduling algorithm, namely *proportional mapping* [37], for tree-shaped task graphs and to adapt it to our context. This algorithm, uses a "local" mapping of the processors to the nodes of the task graph. To be more precise, starting from the root node to which all the processors are assigned, the algorithm splits the set of processors recursively among the branches of the tree according to their relative workload until all the processors are assigned to at least one subtree (these number of processors correspond to the red numbers associated to the nodes of the tree given in Figure 2). This algorithm is characterized by both a good workload balance and a good locality. It is important to note that the approach described below can be adapted to other scheduling algorithms for trees of malleable tasks like the ones presented in [38], [33].



(a) First step .

(b) Second step .

Fig. 2: Mapping algorithm.

In this paper, we propose to upgrade the stopping criterion for the top-down process of the proportional mapping so that the locality will be enhanced on multicore systems. Our improved version needs an additional input which represents a number of "bundles" (each bundle represents a group of computational resources). The bundle concept is linked to the architecture of the underlying platform: a bundle corresponds to the set of resources sharing a given level of the memory hierarchy. Generally the number of bundles may be the number of actual processor sockets or even the number of memory banks. During the top-down process, the current layer of nodes and their corresponding number of computational resources are assigned to the bundles in a sorted-item bin-packing way: we traverse the layer in the topological order and associate the nodes to the bundles until they are full. For example, in Figure 2(a), we use as input to the algorithm two bundles (b_1 and b_2) containing 6 processors each. At the first step of the algorithm, the bundle b_1 is associated with nodes a and e while the bundle b_2 is associated with f and i. This leads the bundle b_1 to exceed its capacity of 6 and thus the algorithm needs to go further using the top-down scheme (note that node a is a leaf in this example). In the next step (see Figure 2(b)), b_1 is associated to nodes a and b and b_2 to nodes c, d, f, g and h leading all the bundles to be perfectly filled and thus stopping the top-down process. More generally, during the top-down process, a layer of nodes in the tree is accepted if the projection of the nodes of the tree over the bundles satisfies the constraint of each bundle. Naturally, for trees which can be met in sparse direct solvers, the constraint (i.e. the number of resources) associated with each bundle needs to be slightly relaxed to ensure that a feasible configuration can be found: a tolerance parameter may be used to check the acceptance criterion.

Once the top-down process has been completed, we associate each subtree rooted at a node above the accepted layer with an abstract context associated to the set of resources resulting from the proportional mapping. This produces a hierarchy of contexts which are given to the runtime system together with the amount of work (resp. the number of resources) associated with each context before the factorization begins. Note that if the number of resources is rational, the resource located at a border of a scheduling context is shared with the neighboring one (the runtime system uses time-sharing of the resource among the contexts it belongs to). From the implementation point of view, this represents a slight modification of the factorization in the sense that the scheduling context to which a given task belongs is an attribute of the task (see [27] for more details). We introduce in the next section a dynamic management strategy for this hierarchy of scheduling contexts. Note that from a pure software point of view, the fact that the tasks are now assigned to a context represents a very marginal modification.

IV. A RUNTIME SYSTEM LEVEL TO DYNAMICALLY MANAGE THE RESOURCE ALLOCATION

StarPU is a C library that provides programmers with a portable interface for scheduling dynamic graphs of tasks onto a heterogeneous set of processing units (*i.e.* CPUs, GPUs and Intel Xeon-Phi). In a previous work [27], we have introduced an extension of StarPU, that allows multiple parallel codes to run concurrently with minimal interference. Such parallel codes run within *scheduling contexts* that provide confined execution environments which can be used to partition computing resources. Moreover, a *hypervisor* that automatically expands or shrinks contexts using feedback from the runtime system (e.g. resource utilization) was introduced. In this paper we present a hierarchical resizing strategy for scheduling contexts that enforces the locality of the parallel tasks of application. Our work borrows some ideas from the Bubble Scheduling approach [43].

In the previous section we described the algorithm we implemented in order to structure the parallelism hierarchically and enforce locality. Branches of the scheduling context tree are isolated on sets of processing units which share a given level of the memory hierarchy. The execution is a top-down traversal of the task graph where a parent node cannot be treated before its children have been processed. Thus, from the scheduling contexts hierarchy point of view, the active scheduling contexts correspond to a layer at the bottom of the

tree. This layer goes up towards the root of the tree during the execution.

In the ideal situation, mapping the tree of contexts on the hierarchical architecture of the underlying platform leads to a perfect exploitation of the resources. However, this is not the case for real-life applications. Indeed, applications like `qr_mumps` usually deal with very irregular task-graphs where predicting the actual execution time is challenging. For this reason, mechanisms to dynamically step in whenever imbalance appears are indispensable.

To this effect we used the hypervisor introduced in [27] in order to dynamically adapt the resource allocation over the scheduling contexts such that unexploited processing units can be used by other parallel tasks. Our main constraint in this situation is to keep the computation local in the sense of the memory hierarchy.

Hierarchical resizing the Scheduling Contexts

Our approach to the locality problem is to reallocate the resources hierarchically and to take the resizing decisions locally at each level of the tree of the scheduling contexts. Contexts with the same parent access nearby data and as soon as they finish executing they provide contribution blocks (see Section II) to their parent. The hypervisor enforces then the locality and allocates the resources such that this group of contexts finishes in the minimum amount of time. In other words, each level of contexts has its own deadline and the execution of the branches of the application progresses locally on the corresponding group of processing units. In Figure 3 each group of sibling contexts can exchange processing units as long as they finish their execution time before a certain deadline (e.g. D5, D6, etc.)

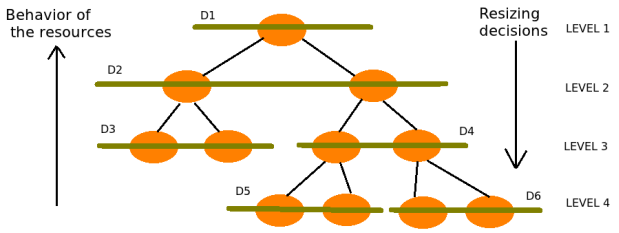


Fig. 3: Resizing hierarchical contexts by having local deadlines

In order to keep the decisions consistent, resizing information (e.g. the speed of the processing units) is transmitted upwards from the leaves to the root and resizing decisions are taken at the higher level and then propagated downwards.

Allocation of processing units

The hypervisor uses the linear program described by the Equation (1) in order to resize a set of scheduling contexts. By using the workload value of the kernels provided by the application, in our case `qr_mumps` solver, this algorithm distributes proportionally the resources over the scheduling contexts. We obtain a rough computation of the number of processing units needed by each context such that the program ends its execution in a minimal amount of time.

$$\max\left(\frac{1}{t_{max}}\right) \text{subject to} \left(\begin{array}{l}\left(\forall c \in C, n_{\alpha,c} v_{\alpha,c} \geq \frac{W_c}{t_{max}}\right) \\ \wedge \left(\sum_{c \in C} n_{\alpha,c} = n_\alpha\right) \\ \wedge \left(\forall c \in C, n_{\alpha,c} < max_{\alpha,c}\right)\end{array}\right) \quad (1)$$

In this linear program C denotes the set of contexts, $n_{\alpha,c}$ represents the unknown of the system, that is the number of processing units that are assigned to a context c, W_c is the total amount of work associated to the context c, t_{max} represents the maximum amount of time spent by a context to process its amount of work, $v_{\alpha,c}$ represents the speed (i.e. floating point operations per second) of the processing units belonging to the context c, n_α is the total number of processing units. Equation (1) expresses that each context should have the appropriate number of CPUs and GPUs such that it finishes its assigned amount of work before the deadline t_{max}. max_α represents a constraint value that indicates a rough information concerning the parallelism of the kernels. Of course, this linear program can be easily generalized to platforms with more than one type of resources.

The hypervisor resolves this equation several times during the execution such that it can consider and insert new collected information like: the speed of the processing units when executing a certain kernel, more precise values of the workload of the kernels or bounds for the number of allocated processing units. A good reactivity of the hypervisor is required as the `qr_mumps` solver may dynamically update the workload information associated with the scheduling contexts.

Triggering the reallocation of resources

An important aspect in improving the execution time of an application is determining when resources are no longer efficiently used in their scheduling context, they are slow or even idle. We use the hypervisor as a tool that collects information concerning on one hand the behavior of the application with the provided distribution of resources and on the other hand the efficiency of the processing units when executing the parallel tasks. Therefore, the runtime is in charge with monitoring the application and providing the hypervisor information like: the moment a task started/finished executing, its workload, the time a worker spent without executing any task in a certain context. Meanwhile, the application is in charge with providing and adjusting dynamically the information concerning the workload of the application and of each task.

The hypervisor synthesizes this information and computes at some period of time the interval in which the processing units were idle and also what we call the “instant speed” of a scheduling context or of a type of processing unit. This latter represents the number of flops executed by a context respectively a type of worker (CPU, GPU, etc.) in a certain sample of time. The hypervisor uses this information in order to compare it with the ideal speed resulting from the previous

solution of the linear program (1). Whenever the difference between the actual speed and the ideal one exceeds a threshold or PUs are in an idle mode for longer than a given limit of time we consider that the current distribution of processing units is not valid and the resizing process is triggered. The linear program (1) is then solved using the instant speed values computed, leading thus to a more balanced distribution. The reactivity of the hypervisor is adapted to the irregularity of the problem. We have evaluated the best trade off between the need to resize the contexts and the overhead it implies.

The hypervisor monitors the scheduling contexts hierarchically and triggers the resizing of the scheduling contexts at a certain level. Verifications are then going up towards the root level and stop at the level where there is no need to resize (i.e. at this given level the speed of each context is consistent with the ideal speed). Hence, the resizing decisions at higher levels of the tree of scheduling contexts are propagated toward the layer of active contexts.

Upper bounds to the allocation of resources

Determining that a kernel does not scale on a certain number of processing units is an information the runtime can easily provide. By computing the time a resource was idle in a context in a previous period of time we can predict how many resources that contexts needs in the future (a maximum value). However, this task is more complicated when resources behaved well in the past. We can consider that the parallelism of the kernel is determined by the number of ready tasks available or we can impose an upper bound to the allocation of the resources only at certain moments. For `qr_mumps` we implemented an algorithm that considers this max only when the speed of the processing units is very far from an average speed value computed from the beginning of the execution of the application. This solution is adapted to the behavior of `qr_mumps`, that usually has a good speed at the beginning of the execution (tree parallelism is sufficient to ensure performance at that moment).

The maximum number of processing units needed by the leaf contexts is computed locally and this information is propagated hierarchically until the root where it is used as an upper bound in the linear equation (1).

V. Experimental results

In this section we evaluate the behavior of our hierarchical approach and illustrate how it improves locality and performance. This is done on a set of sparse linear systems solved on two types of architectures: SMP and NUMA. The experimental evaluation illustrates the gains in terms of execution time and enhancement of the locality. Moreover, the cost of the hierarchical strategy is evaluated.

A. Experimental environment

As stated above, we evaluate the behavior of our approach on two platforms:

- `MachineA` which has uniform access to the memory. It is composed of 4 Intel E7-4870 processors having 10 cores clocked at 2.40 GHz and having 30 MB of L3 cache for a total of 40 cores. The platform is equipped with 1 TB of memory.
- `MachineB` is a cache coherent Non Uniform Memory Access (ccNUMA) platform containing 8 Intel E7-8837 processors having 8 cores clocked at 2.67 GHz and having 24 MB of L3 cache for a total of 64 cores. The platform is equipped with 300 GB of memory organized in groups of 100 GB each interconnected with a slower memory bus.

Both platform are shared memory memory ones in order to match the requirements of `qr_mumps`, which is a solver designed for shared memory systems.

#	Mat. name	m	n	nz	op. count (Gflops)
1	TF15	7742	6334	80057	93.90
2	tp-6	142752	1014301	11537419	381.82
3	esoc	37830	327062	6019939	891.58
4	Rucci1	1977885	109900	7791168	5316.94
5	pre2	659033	659033	5834044	777.67
6	ultrasound80	531441	531441	33076161	64777.40
7	conv3d64	836550	836550	12548250	108491.50

TABLE I: Matrices test set. The operation count is related to the matrix factorization with METIS column permutation.

The experiments were conducted on a set of matrices mainly from the University of Florida Sparse Matrix Collection[1] presented in Table I. The exceptions being the ultrasound80 matrix (Propagation of 3D ultrasound waves, provided by M. Sosonkina) and the conv3d64 matrix (provided by CEA-CESTA and generated using AQUILON[2]). All the matrices have been reordered using a fill-reducing matrix permutation produced by METIS[3] (version 5.0.2). We divided this set of matrices in two groups: the so called small problems: TF15, tp-6, esoc, Rucci1, pre2 and the large problems: ultrasound80 and conv3d64. The behavior of the small problems has not been evaluated on more than 40 cores, as they are not able to scale on so many processing units. All codes were compiled with the GNU v. 4.7.2 suite and linked to the Intel MKL sequential BLAS and LAPACK libraries. All the tests were run with real data in double precision. Finally, it is important to mention that for a small number of processing units, the cores used for the experiments are chosen according to a compact strategy according to the memory hierarchy.

B. Experimental evaluation

We begin the evaluation section by measuring the cost of the dynamic algorithm used to distribute the processing units to the hierarchical structure of the scheduling contexts. We measure the time spent calling the hypervisor and trying to redistribute the resources in order to match the structure of the application and the machine. In the table II we can see that the cost of the hypervisor is more important on smaller matrices, because they do not have enough computation in

[1] http://www.cise.ufl.edu/research/sparse/matrices
[2] http://www.enscpb.fr/master/aquilon
[3] http://glaros.dtc.umn.edu/gkhome/views/metis

order to compensate the time spent to improve the execution time. However, larger problems like conv3d64 have enough workload in order to make profitable the hypervisor. Moreover, conv3d64 has a relatively regular assembly tree, therefore the hypervisor is less needed and implicitly less called.

	8 cores	16 cores	24 cores	32 cores	40 cores
TF15	0.03	0.02	0.02	0.04	0.10
tp-6	0.48	0.11	0.06	0.10	0.24
ESOC	0.02	0.01	0.04	0.09	0.12
rucci1	0.03	0.02	0.03	0.03	0.04
pre2	0.01	0.01	0.03	0.06	0.06
ultrasound80	0.04	0.02	0.03	0.02	0.007
conv3d64	0.04	0.03	0.04	0.03	0.03

(a) Cost of the resizing process on `MachineA`.

	8 cores	16 cores	24 cores	32 cores	40 cores	64 cores
TF15	0.80	0.07	0.17	0.22	0.27	-
tp-6	0.89	0.66	0.29	0.78	0.53	-
ESOC	0.05	0.04	0.13	0.21	0.23	-
rucci1	0.09	0.05	0.08	0.10	0.13	-
pre2	0.04	0.03	0.10	0.16	0.17	-
ultrasound80	0.09	0.03	0.08	0.05	0.07	0.14
conv3d64	0.15	0.03	0.09	0.05	0.08	0.16

(b) Cost of the resizing process on `MachineB`.

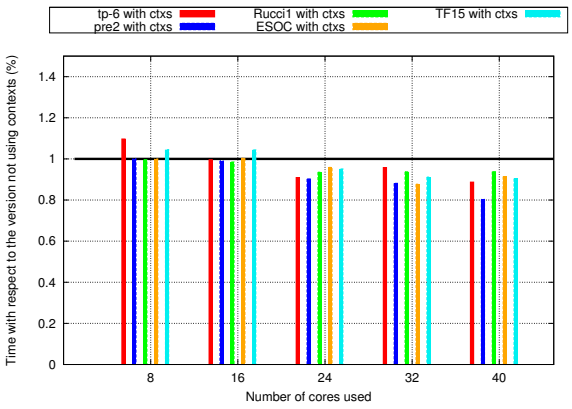
TABLE II: Cost of the resizing process with respect to the total execution time (%)

On the `MachineB` platform the overall overhead is more important than on the `MachineA` platform. This is mainly due to the NUMA aware architecture of the platform `MachineB`. Therefore, when monitoring the applications the hypervisor detects an important number of cases of slow contexts or idle resources which require its help in order to adjust to this architecture. We can see that for small problems like tp-6 we spend 0.89% of the time in the hypervisor even when running on 8 cores. This shows that making profit of the hypervisor for this problem is more difficult due to its costs compared to its execution time. The overhead of the hypervisor varies also with the structure of the graph of tasks of the problem. According to its structure we may need more or less contexts and implicitly the resizing may be more or less expensive.

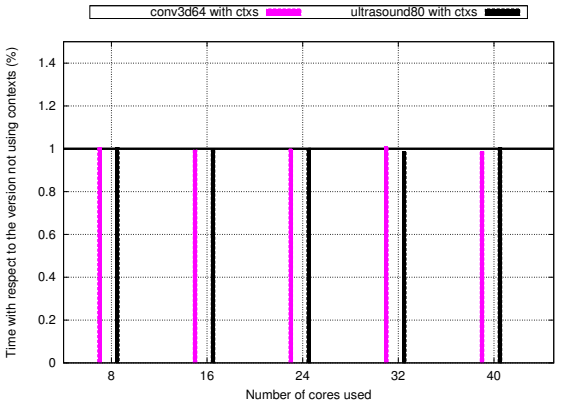
Further on, we evaluate the behavior of the hierarchical contexts approach to structure the parallelism of different problems. In previous work [3] Buttari *and al.* have studied the performance behavior of `qr_mumps` on top of StarPU and they compared it with different solution from the state of art. Therefore, in the following section we compare the execution time of our approach to the one using StarPU without any contexts.

In Figure 4 we show the ratio between the execution time of the version using the contexts and the basic StarPU one not using the contexts. In table III we can see the referenced execution time of the version without contexts. This complements the data provided by Figure 4 and 5. We can see that the scalability of the solver is satisfactory when the problem is large enough to be treated on the considered number of cores.

We can observe in Figure 4 that the execution time of



(a) Small problems



(b) Large problems

Fig. 4: Execution time of the hierarchical version of `qr_mumps` with respect to the non contexts StarPU version on the `MachineA` platform

the hierarchical version of the solver has a comparable performance with the regular StarPU implementation on small number of processors. However, when we increase the number of processes, we can observe that the hierarchical version starts to take advantage of the locality and thus improves the performance, such that we obtain a decrease of the execution time of up to 15% The hierarchical version does not always outperform the regular version, especially when the problems have a regular form of the assembly tree. There is more room to improve performance if the assembly tree is irregular and unbalanced. One side effect of our hierarchical algorithm is that it will assign a lot of resources to the branch of the tree corresponding to the critical path. Thus, on such irregular trees, this algorithm may reduce the length of the critical path (by increasing the number of resources) leading to a decrease of

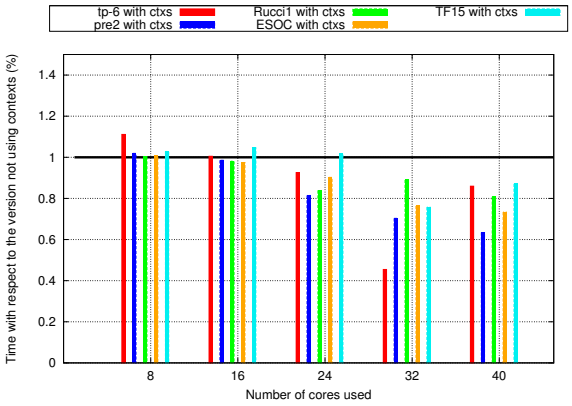
	8 cores	16 cores	24 cores	32 cores	40 cores
TF15	2.27	1.82	2.05	2.37	2.28
tp-6	11.06	9.08	9.69	10.14	11.43
ESOC	21.65	13.19	14.83	16.27	16.92
rucci1	99.10	55.26	43.74	40.51	41.07
pre2	19.64	10.95	9.64	13.38	14.48
ultrasound80	1066.59	584.92	421.30	345.62	304.79
conv3d64	1779.88	966.80	680.49	546.78	463.84

(a) Execution time on `MachineA`.

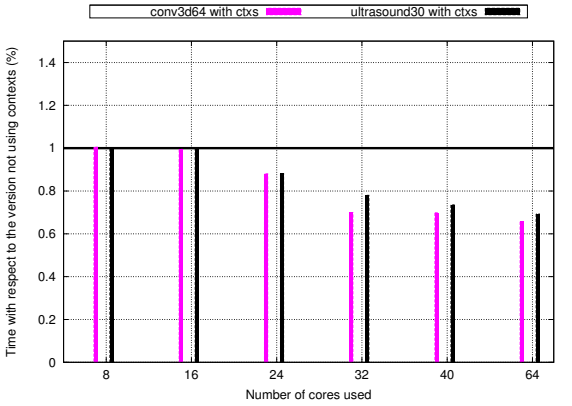
	8 cores	16 cores	24 cores	32 cores	40 cores	64 cores
TF15	2.65	2.30	2.85	3.93	3.88	-
tp-6	11.23	9.39	11.18	16.02	14.05	-
ESOC	21.27	13.43	20.85	24.73	27.50	-
rucci1	93.29	52.55	53.25	53.48	59.45	-
pre2	18.98	10.99	13.43	22.84	25.44	-
ultrasound80	1172.22	751.47	550.13	502.73	510.29	527.16
conv3d64	2166.15	1405.10	1032.14	890.85	813.71	784.87

(b) Execution time on `MachineB`.

TABLE III: Execution time in seconds of different test problems of the regular `qr_mumps` implementation on top of StarPU



(a) Small problems



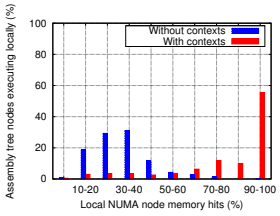
(b) Large problems

Fig. 5: Execution time of the hierarchical version version of `qr_mumps` with respect to the non contexts StarPU version on the `MachineB` platform

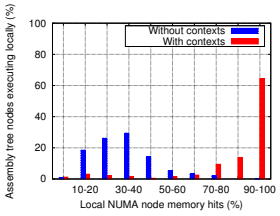
the execution time.

In Figure 5 we can notice a similar behavior for the small problems. However, on larger test problems (see Figure 5(b)), the execution time gain obtained from the use of our hierarchical approach grows with the number of resources. This is mainly due to the fact that the hierarchical strategy enhances data locality and isolates branches of the assembly on a specific set of cores taking advantage of the strongly non-uniform memory hierarchy. We can observe gains going up to 30% on some cases like the conv3d64 on 64 cores. On highly NUMA architectures like `MachineB` the locality is an important matter. We can see that using our hierarchical approach on top of 32 cores or 64 cores for the large matrices improves the behavior of the applications. This is mainly due to the fact that the `MachineB` has 4 NUMA groups of 16 cores and by isolating sections of the assemble tree on different NUMA nodes we avoid data transfers between the memory nodes. Using 40 cores, for example, implies executing on two complete groups and another additional 8 cores in another group. The increase in processing units does not compensate the costs of data transfers to those isolated cores. We can see this behavior especially for small matrices for which the computations do not counterbalance the data transfers.

To push further the analysis of the results we present in Figures 6 and 7 a plot illustrating the improvement of the locality of memory access when using the hierarchical approach for two of our test problems on the two platforms. First of all, to be consistent with the underlying architecture, we considered executions on 40 (resp. 32) cores for `MachineA` (resp. `MachineB`). Moreover, we remind that memory allocations are done during the execution of the activation task corresponding to each node of the assembly tree (see Section II). This plot considers the amount (percentage) of assembly tree nodes for which the amount of corresponding StarPU tasks were executed on the same socket as the activation task. For

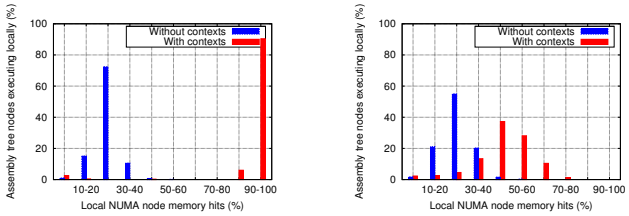


(a) Rucci1 on the `MachineA` platform using 40 cores.



(b) Rucci1 on the `MachineB` platform using 32 cores.

Fig. 6: Locality of data references for the Rucci1 problem.



(a) conv3d64 on the `MachineA` platform using 40 cores.

(b) conv3d64 on the `MachineB` platform using 32 cores.

Fig. 7: Locality of data references for the conv3d64 problem.

example, if we consider the plot presented in Figure 7(a), we can observe that for the hierarchical approach (red bars) almost 90% of the nodes of the assembly had between 90% and 100% of their corresponding tasks executed on the same socket as the activation task. On the other side, the regular StarPU implementation has around 70% of the nodes of the tree having between 20% and 30% of their tasks executed on the same socket as the memory allocation. The locality has an important impact on large problems like conv3d64. The improvement on its execution time on `MachineB` is mainly due to the fact that at least 40% of the assembly nodes face between 40% and 60% memory hits, compared to the non context version that mainly faces between 20% and 30% memory hits.

If we reconsider the results showed in Figure 5, we can see that indeed conv3d64 and Rucci1 both improve their execution time when using the hierarchical strategy on `MachineB`. Therefore, we can can confirm that by enforcing the locality on non-uniform memory access platforms we can improve the overall execution time of the application.

VI. Related Work

A lot of initiatives have emerged in the past years to develop efficient runtime systems for modern heterogeneous platforms. Most of these runtime systems use a task-based paradigm to express concurrency and dependencies by employing a task dependency graph to represent the application to be executed. Qilin [35] provides an interface to submit kernels that operate on arrays which are automatically dispatched between the different processing units of an heterogeneous machine. Moreover, Qilin dynamically compiles parallel code for both CPUs (by relying on the Intel TBB [39] technology) and for GPUs, using CUDA. Another relevant framework is Charm++ [30] which is a parallel variant of the C++ language that provides sophisticated load balancing and a large number of communication optimization mechanisms. Charm++ has been extended to provide support for accelerators such as the Cell processors as well as GPUs [31]. Many runtime systems propose a task-based programming paradigm. Runtime systems like KAAPI/XKAAPI [25] or APC+ [24] offer support for hybrid platforms mixing CPUs and GPUs. Their data management is based on a DSM-like mechanism: each data block is associated with a bitmap that permits to determine whether there is already a copy locally available to a specific processing unit or not. Moreover, task scheduling within KAAPI is based on work-stealing mechanisms or on graph partitioning. 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 [9], [10], [7]. 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. Finally, the DAGuE/PaRSEC [11] runtime system dynamically schedules tasks within a node using a rather simple strategy driven by locality. It takes advantage of the specific shape of the task graphs (in the sense that there are few types of tasks) to represent the task dependency graph in an algebraic fashion. Some of these runtime systems tackle the locality issues by having greedy scheduling strategies which aim at enhancing the locality. For instance, the DAGuE runtime system tries to avoid data movements by having a data driven scheduling approach, while in the KAAPI framework, some work-stealing strategies may take advantage of the memory hierarchy to select the victim. In opposition, our approach considers the problem at a higher level, requiring some input from application to drive the dynamic decisions of the runtime system.

Regarding resource sharing, Lithe [36] is a runtime system that enables interoperability between different parallel runtimes, e.g. Intel TBB and OpenMP. Lithe is a resource sharing management interface that defines how *harts* (i.e. abstraction of hardware threads) are transferred between parallel libraries within an application. Lithe imposes a hierarchical organization between libraries as well as a specific implementation of multitasking. Thus, this kind of frameworks are not well-adapted to the context of a complex irregular application which needs to dynamically resize the groups of resources.

From the linear algebra solvers point of view, a lot of effort has been spent to improve the behavior of the existing solvers on emerging architectures. By taking into account both task and data affinity and by relying on a two-level hybrid parallelization approach mixing multithreading and message passing, numerous solvers are now able to efficiently exploit the features of these new platforms [19], [22], [23], [40]. In other cases, new solvers have been designed and implemented from scratch for these new computer platforms. The chosen scheme is mostly what has been done for dense linear algebra solvers (fine-grained parallelism, thread-based parallelization and advanced data management to deal with complex memory hierarchies). Examples of this kind of solvers are HSL-MA87/HSL-MA86 [26] and SuperLU-MT [34] for sparse LU or Cholesky factorizations and SPQR [17] and `qr_mumps` [13] for sparse QR factorizations.

A successful approach to deal with the complexity of modern architecture is centered around the use of runtime systems to manage tasks dynamically, these runtime systems being either generic or specific to the application. As a result, higher performance portability is also achieved thanks to the hardware abstraction layer introduced by runtime systems [1]. These efforts resulted in the design of the MAGMA library [4] on top of StarPU, the DPLASMA library [12] on top of DAGuE

and the adaptation of the existing FLAME library [28] to heterogeneous multicore systems using the SuperMatrix [15] runtime system. More recently, this approach has been used for more irregular applications. Sparse direct solvers have been redesigned on top of task-based runtime systems [3], [32] leading to a good behavior and an improved portability. Finally, this methodology starts to be used in other application fields like what has been done for the fast multipole method in [2].

VII. Conclusion

The main objective of this work was to evaluate a new way of handling resource allocation and memory affinity within HPC applications using powerful runtime system mechanisms. We believe a tight interaction between the application and the runtime system can improve performance with only slight modifications on the application side.

Our approach consists in capturing the parallel structure of the application in a hierarchical manner and projecting it on an abstract tree. This hierarchy is eventually mapped on scheduling contexts at the runtime system level which is responsible for dynamic resource management. Using information coming from the application together with hardware metrics captured at runtime, our runtime system is able to better adjust the number of processing units allocated to each parallel task.

We demonstrate the relevance and effectiveness of our approach on a complex irregular sparse linear algebra solver over modern multicore architectures. Our experiments show that by continuously enforcing locality between related tasks, our approach exhibits a gain of up to 35% on test cases coming from real-life applications.

In the near future, we plan to further extend this work to heterogeneous platforms equipped with accelerators. We also plan to generalize our work to several other task-based runtime systems, such as OpenMP or Intel TBB-powered parallel libraries.

Acknowledgments

This work was supported by the European Commission as part of the FP7 Project PEPPHER under grant 248481, by the ANR through the MN (Solhar ANR-13-MONU-007 project) program.

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*, vol. 2, pp. 473–484.

[2] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-Based FMM for Multicore Architectures," *SIAM Journal on Scientific Computing*, 2013. [Online]. Available: http://hal.inria.fr/hal-00911856

[3] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Multifrontal qr factorization for multicore architectures over runtime systems," in *Euro-Par 2013 Parallel Processing - 19th International Conference*, 2013, pp. 521–532.

[4] 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," vol. Vol. 180.

[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.*, vol. 3(4), pp. 275–300, 1996.

[6] 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*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[7] 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*, ser. Lecture Notes in Computer Science, H. J. Sips, D. H. J. Epema, and H.-X. Lin, Eds., vol. 5704. Springer, 2009, pp. 851–862.

[8] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Euro-Par*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862. [Online]. Available: http://portal.acm.org/citation.cfm?id=1616772.1616863

[9] 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*, vol. 21, no. 18, pp. 2438–2456, 2009.

[10] P. Bellens, J. M. Pérez, F. Cabarcas, A. Ramírez, R. M. Badia, and J. Labarta, "CellSs: Scheduling techniques to better exploit memory hierarchy," *Scientific Programming*, vol. 17, no. 1-2, pp. 77–95, 2009.

[11] 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*, vol. 38, no. 1-2, pp. 37–51, 2012.

[12] 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.

[13] A. Buttari, "Fine-grained multithreading for the multifrontal QR factorization of sparse matrices," 2013, to appear on the SIAM Journal on Scientific Computing.

[14] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Par. Comp.*, vol. 35, no. 1, pp. 38–53, 2009.

[15] E. Chan, F. G. V. Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. A. van de Geijn, "Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *PPOPP*, S. Chatterjee and M. L. Scott, Eds. ACM, 2008, pp. 123–132.

[16] I. Corporation, "MKL reference manual," http://software.intel.com/en-us/articles/intel-mkl. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl

[17] T. A. Davis, "Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 8, 2011.

[18] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2008, pp. 197–200.

[19] M. Faverge and P. Ramet, "Dynamic scheduling for sparse direct solver on NUMA architectures," in *Proceedings of PARA'2008*, Trondheim, Norway, May 2008.

[20] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[21] M. Frigo, C. Leiserson, and K. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, 1998.

[22] A. Gupta, "A shared- and distributed-memory parallel general sparse direct solver," *Appl. Algebra Eng. Commun. Comput.*, vol. 18, no. 3, pp. 263–277, 2007.

[23] A. Gupta, S. Koric, and T. George, "Sparse matrix factorization on massively parallel computers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 1:1–1:12.

[24] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek, "Improving performance of adaptive component-based dataflow middleware," *Parallel Computing*, vol. 38, no. 6-7, pp. 289–309, 2012.

[25] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU parallelization for interactive physics simulations," in *Euro-Par (2)*, ser. Lecture Notes in Computer Science, P. D'Ambra, M. R. Guarracino, and D. Talia, Eds., vol. 6272. Springer, 2010, pp. 235–246.

[26] J. D. Hogg, J. K. Reid, and J. A. Scott, "Design of a multicore sparse Cholesky factorization using DAGs," *SIAM J. Scientific Computing*, vol. 32, no. 6, pp. 3627–3649, 2010.

[27] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: A supervised approach," in *IPDPS Workshops*. IEEE, 2013, pp. 1050–1059.

[28] 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.*, vol. 72, no. 9, pp. 1134–1143, 2012.

[29] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling hierarchical n-body simulations on gpu clusters," in *SC*. IEEE, 2010, pp. 1–11.

[30] L. V. Kalé and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on c++," in *OOPSLA*, 1993, pp. 91–108.

[31] D. M. Kunzman and L. V. Kalé, "Programming heterogeneous clusters with accelerators using object-based programming," *Scientific Programming*, vol. 19, no. 1, pp. 47–62, 2011.

[32] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca, "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes," INRIA, Rapport de recherche RR-8446, Jan. 2014. [Online]. Available: http://hal.inria.fr/hal-00925017

[33] R. Lepere, G. Mounie, and D. Trystram, "An approximation algorithm for scheduling trees of malleable tasks," *European Journal of Operational Research*, vol. 142, pp. 242–249, 2002.

[34] X. S. Li, "Evaluation of sparse LU factorization and triangular solution on multicore platforms," in *VECPAR*, ser. Lecture Notes in Computer Science, J. M. L. M. Palma, P. Amestoy, M. J. Daydé, M. Mattoso, and J. C. Lopes, Eds., vol. 5336. Springer, 2008, pp. 287–300.

[35] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO*, D. H. Albonesi, M. Martonosi, D. I. August, and J. F. Martínez, Eds. ACM, 2009, pp. 45–55.

[36] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with lithe," *SIGPLAN Not.*, vol. 45, pp. 376–387, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806639

[37] A. Pothen and C. Sun, "A mapping algorithm for parallel sparse Cholesky factorization," *SIAM Journal on Scientific Computing*, vol. 14(5), pp. 1253–1257, 1993.

[38] G. N. S. Prasanna and B. R. Musicus, "The optimal control approach to generalized multiprocessor scheduling." *Algorithmica*, vol. 15, no. 1, pp. 17–49, 1996.

[39] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[40] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Future Generation Comp. Syst.*, vol. 20, no. 3, pp. 475–487, 2004.

[41] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Transactions on Mathematical Software*, vol. 8, pp. 256–276, 1982.

[42] G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira, "Coordinating the use of gpu and cpu for improving performance of compute intensive applications," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-sept. 4 2009, pp. 1 –10.

[43] S. Thibault, R. Namyst, and P.-A. Wacrenier, "Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework," in *Euro-Par*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds., vol. 4641. Springer, 2007, pp. 42–51.

[44] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, april 2010, pp. 1 –8.