Resource aggregation in task-based applications over accelerator-based multicore machines

Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, Pierre-André Wacrenier

Resource aggregation in task-based applications over accelerator-based multicore machines

Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, Pierre-André Wacrenier
INRIA, LaBRI, University of Bordeaux, Talence, France
Email: firstname.lastname@inria.fr

***Abstract*—Computing platforms are now extremely complex providing an increasing number of CPUs and accelerators. This trend makes balancing computations between these heterogeneous resources performance critical. In this paper we tackle the task granularity problem and we propose *aggregating several CPUs* in order to execute larger parallel tasks and thus find a better equilibrium between the workload assigned to the CPUs and the one assigned to the GPUs. To this end, we rely on the notion of *scheduling contexts* in order to isolate the parallel tasks and thus delegate the management of the task parallelism to the inner scheduling strategy. We demonstrate the relevance of our approach through the dense Cholesky factorization kernel implemented on top of the StarPU task-based runtime system. We allow having parallel elementary tasks and using Intel MKL parallel implementation optimized through the use of the OpenMP runtime system. We show how our approach handles the interaction between the StarPU and the OpenMP runtime systems and how it exploits the parallelism of modern accelerator-based machines. We present experimental results showing that our solution outperforms state of the art implementations to reach a peak performance of 4.6 TFlop/s on a platform equipped with 20 CPU cores and 4 GPU devices.**

***Keywords*—Multicore; accelerator; GPU; heterogeneous computing; task DAG; runtime system; dense linear algebra; Cholesky**

I. Introduction

Due to recent evolution of High Performance Computing architectures toward massively parallel heterogeneous multicore machines, 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 [1], OpenMP or Intel TBB [2] for multicore machines, PaRSEC [3], Charm++ [4], Harmony [5], KAAPI [6], Qilin [7], StarPU [8] or StarSs [9] for heterogeneous configurations) has allowed programmers to rely on task-based runtime systems and develop efficient implementations of parallel libraries (e.g. Intel MKL [10], FFTW [11]).

Since the increase of cores is a clear trend to boost theoretical performance of processors, the gap between the processing power of individual CPU cores and accelerators (e.g. GPU) is increasing. The main issue encountered when trying to exploit both CPUs and accelerators lies in the fact that these devices have very different characteristics and requirements. Compared to regular CPUs, a GPU for instance is composed of many lightweight cores and requires massive parallelism to hide memory latencies and thus fully tap into its potential performance. As a result, GPUs typically exhibit better performance when executing kernels featuring numerous threads, which we call *coarse grain kernels* in the remainder of the paper. On the contrary, regular CPU cores typically reach their peak performance with fine grain tasks working on a reduced memory footprint. To illustrate this claim, we provide in Figure 1 a performance profile of the matrix product kernel (DGEMM) on two different devices: an Intel CPU Haswell Xeon E5-2680 v3 with 12 cores, and a GPU Nvidia K40M. We can observe that the sequential MKL implementation of the DGEMM kernel reaches its peak performance for matrix sizes greater than 200 while in the case of the cuBLAS kernel, the GPU reaches its peak performance for sizes above 2000.

Unfortunately, runtime systems often consider GPUs or Xeon Phi accelerators as single devices, and treat individual cores equally. And because many applications are parallelized using homogeneous block or tile decomposition, runtime systems' schedulers have to cope with very different durations when executing tasks over single cores or over accelerators, resulting in situations where only a few tasks are assigned to CPUs because of bad scores computed by the performance prediction-based heuristics. As a consequence, task-based applications running on such heterogeneous platforms typically adopt an intermediate granularity, chosen as a trade-off between coarse-grain and fine-grain tasks. A small granularity would indeed lead to poor performance on the GPU side, whereas large kernel sizes would dramatically increase the cost of bad load balancing decisions.

The basic solution used by dense linear algebra libraries [12], [13], [14], [15] is to compute a unique size which will represent the best trade-off. Some approaches relax this constraint and are able to split coarse-grain tasks at run time to generate fine-grain tasks for CPUs [16] and provide a more flexible and efficient approach. In this paper, we investigate a dual approach: instead of splitting tasks, we aggregate computing resources in a way such that coarse grain kernels can efficiently be executed by CPU pools and by exploiting inner task parallelism. We provide in Figure 1 a performance of the DGEMM kernel when using the multithreaded Intel MKL implementation. We can see that by doing so, the performance of the kernel scales almost perfectly for matrix sizes larger than 960.

The approach we propose in this paper to tackle the granularity problem is based on resource aggregation: instead of dynamically splitting tasks, we rather aggregate resources to process coarse grain tasks in a parallel fashion. To do so, we introduce a generalization of the concept of scheduling contexts [17] to handle any type of parallel tasks. We illustrate how state of the art scheduling heuristics are upgraded to deal

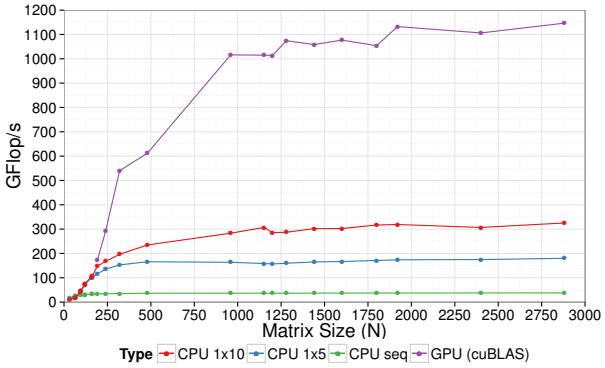


Fig. 1. Performance of the matrix product kernel (DGEMM) on CPU and GPU devices.

with parallel tasks. We then evaluate our solution on a dense Cholesky factorization kernel and show that it outperforms state of the art implementations to reach a peak performance of 4.6 TFlop/s on a platform equipped with 20 CPU cores and 4 GPU devices.

II. Related Work

A number of research efforts have recently been focusing on redesigning HPC applications to use dynamic runtime systems. As an example, several sparse direct solvers have been redesigned on top of task-based runtime systems [18], [19] and exhibit high performance and improved portability. Similar efforts have also been undertaken in the field of fast multipole method computations [20], [21]. The dense linear algebra community has strongly adopted this modular approach, by relying on task-based runtime systems, over the past few years [12], [13], [14], [15] and delivered production-quality software relying on it. For example, The `MAGMA` library [14], provides Linear Algebra algorithms over heterogeneous hardware and can optionally use the StarPU runtime system to perform dynamic scheduling between CPUs and GPUs, illustrating the 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. However, these approaches require that accelerators process a large share of the total workload to ensure a good load balancing between resources. Additionally, all these approaches require an identical tile size, consequently, all tasks have the same granularity independently from where they are executed leading to a loss of efficiency of both the CPUs and the accelerators.

There are a few prior studies trying to resolve the granularity issue between regular CPUs and accelerators in the specific context of dense linear algebra. Most of these efforts rely on heterogeneous tile sizes [22], [23] which may involve extra memory copies when split data need to be coalesced again [24]. However, in most of these efforts, the decision to split a task is made statically at submission time. More recently, a more dynamic approach has been proposed in [16] where the large granularity tasks are hierarchically split at runtime when they are executed on CPUs. Although this paper succeeds at tackling the granularity problem, the proposed solution is specific to linear algebra kernels. In the context of this paper, we tackle the granularity problem with the opposite point of view and a more general approach: rather than splitting coarse grained tasks, we aggregate resources which cooperate to process the task in parallel. By doing so, our runtime system does not only support sequential tasks but also parallel ones. Actually, our runtime system is able to cope with several flavors of inner parallelism (OpenMP, Pthreads, StarPU) simultaneously. However, calling simultaneously several parallel procedures is a difficult matter because usually parallel libraries make the assumption that each procedure has exclusive access to the architecture of the machine. They are not aware of the resource utilization of one another and they may thus oversubscribe threads to the processing units. This problem is referred to as the parallel *composability* problem. This issue has been first tackled within the Lithe framework [25] which is a runtime system enabling interoperability between different parallel runtime systems, e.g. Intel TBB and OpenMP. It is a resource sharing management interface that defines how *harts* (i.e. abstraction of hardware threads) are transferred between parallel libraries within an application. Some efforts have been made within Intel TBB [25], [26], [27] to ensure a good behavior when concurrent parallel TBB kernels are used. These contributions suffered either from the fact that they do not allow to dynamically change the number of resources associated with a parallel kernel or from the fact that they are restricted to a given inner runtime system. Our contribution in this study is a generalization of a previous work [28] where we introduced the so-called scheduling contexts which aim at structuring the parallelism for complex applications relying on concurrent parallel StarPU kernels. We will present in the next section how we extend the scheduling contexts to be able to manage any inner runtime system.

III. Background

Tackling the composability problem at the runtime system level can ensure both the portability and the reusability of the solution. Indeed, such an approach benefits from low level information regarding the architecture of the machine and allows the programmer to have full control on the resource allocation of the application. We integrate our solution inside the StarPU runtime system as it provides a flexible platform to deal with heterogeneous architectures.

A. The StarPU runtime system

StarPU [8] 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 and GPUs). The two basic principles of StarPU are firstly that tasks can have several implementations, for some or each of the various heterogeneous processing units available in the machine, and secondly that necessary data transfers to these processing units are handled transparently by the runtime system. StarPU tasks are defined as multi-version kernels, gathering the different implementations available for CPUs and GPUs, associated to a set of input/output data. To avoid unnecessary data transfers, StarPU allows multiple copies of the same registered data to reside at the same time in different memory locations as long as it is not modified. Asynchronous data prefetching is also used to hide memory latencies.

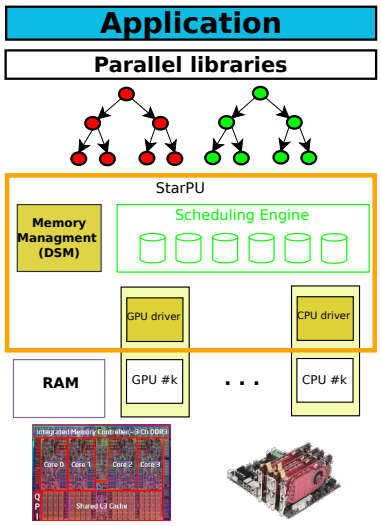


Fig. 2. The architecture of the StarPU runtime system.

StarPU is a platform for developing, tuning and experimenting with various task scheduling policies in a portable way (see Figure 2). It provides an abstract view of the machine by relying on the notion of worker to describe the computing resource (e.g. CPU, GPU) in charge of executing the tasks. Implementing a scheduler consists in creating a set of queues, associating them with the different processing units, and defining the code that is triggered each time a new task becomes ready to be executed, or each time a processing unit is about to go idle. Various designs can be used to implement queues (e.g. FIFOs or stacks), and they can be organized according to different topologies. Several built-in schedulers are available, ranging from greedy and work-stealing based policies to more elaborate schedulers implementing variants of the Minimum Completion Time (MCT) policy [29]. This latter family of schedulers is based on auto-tuned history-based performance models that provide estimations of the expected durations of tasks and data transfers.

StarPU can co-schedule multiple applications concurrently on the same node using *Scheduling Contexts* [17] that act as lightweight virtual machines: each context manages its own computing resources and its own scheduler. These scheduling contexts can typically be used to encapsulate other runtime systems such as OpenMP, and thus allow StarPU to spawn OpenMP tasks over a controlled set of computing resources.

IV. A RUNTIME SOLUTION TO DEAL WITH NESTED PARALLELISM

We introduce a set of mechanisms which aim at managing nested parallelism (i.e. task inner parallelism) within the StarPU runtime system. To do so, we consider the general case where a parallel task may be implemented on top of any runtime system. We present in Figure 3(a) the standard architecture of a task-based runtime system where the task-graph is provided to the runtime and the ready tasks (in purple) are dynamically scheduled on queues associated with the underlying computing resources. We propose a more flexible scheme where tasks may feature *internal parallelism* implemented using any other runtime system. This idea is represented in Figure 3(b) where multiple CPU devices are grouped to form *virtual resources* which will be referred to as *clusters*: in this example, each cluster contains 3 CPU cores. We will refer to the main runtime system as the *external runtime system* while runtime system used to implement parallel tasks will be denoted as *inner runtime system*. The main challenges regarding this architecture are: 1) how to constrain the internal runtime system to restrain his execution to the selected set of resources, 2) how to extend the existing scheduling strategies to this new type of computing resources, and 3) how to define the number of *clusters* and their corresponding resources. In this paper, we will focus on the first two problems since the latter is strongly related to online moldable/malleable task scheduling problems which are out of the scope of this paper.

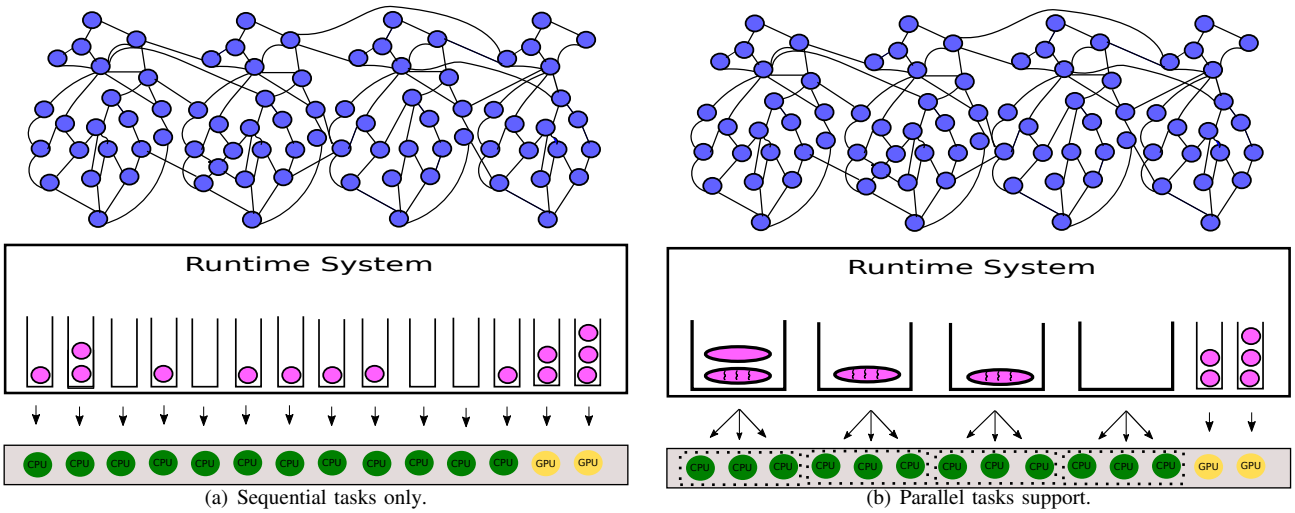
Forcing a parallel task to run on the set of resources corresponding to a cluster depends on wether or not the internal runtime system has its own pool of threads. More precisely, if the inner runtime system offers a multithreaded interface, that is to say the execution of the parallel task requires a call that has to be done by each thread, the internal runtime system can directly use the StarPU workers assigned to the cluster. We show in Figure 4(a) how we manage internal SPMD runtime systems. In this case, the parallel task is inserted in the local queue of each StarPU worker.

In the other hand, if the inner runtime system features its own pool of threads (for example when using OpenMP for parallelizing the task), the StarPU workers corresponding to the cluster need to be paused until the end of the parallel task. This is done to avoid oversubscribing threads over the underlying resources. We describe in Figure 4(b) how the interaction is managed. We allow only one StarPU worker to keep running. This latter called the *master worker* of the cluster, is in charge of popping the tasks assigned to the cluster by the scheduler. The master worker takes the role of a regular application thread with respect to the inner runtime system when tasks have to be executed. In Figure 4(b), the black threads represent the StarPU workers and the pink ones the inner runtime system (e.g. OpenMP) threads. The master worker joins the team of inner threads while the other StarPU threads are paused.

Depending on the inner scheduling engine, the set of computing resources assigned to a cluster may be dynamically adjusted during the execution of a parallel task. This obviously requires the inner scheduler (resp. runtime system) to be able to support such an operation. For instance, parallel kernels implemented on top of runtime systems like OpenMP will not allow removing a computing resource during the execution of the parallel task. In this case we refer to the corresponding parallel task as a *moldable* one and we consider resizing the corresponding scheduling context only at the end of such a task or before starting a new one.

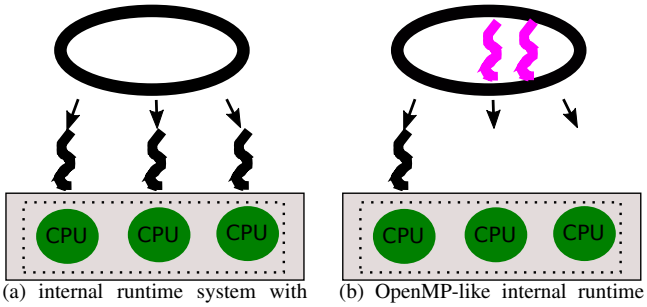
A. Adapting MCT and performance models for parallel tasks

As presented in III-A MCT is a scheduling policy implemented in StarPU. It is based on task performance predictions: each ready task is assigned to the resource which is expected



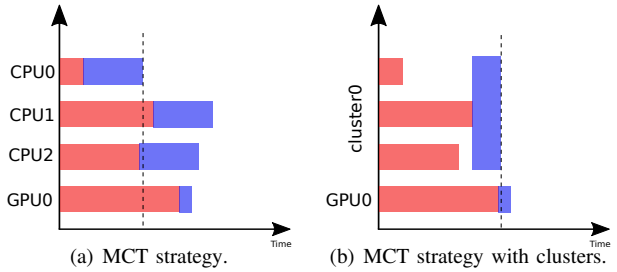
(a) Sequential tasks only. (b) Parallel tasks support.

Fig. 3. Managing internal parallelism within StarPU .



(a) internal runtime system with multithreaded interface. (b) OpenMP-like internal runtime system.

Fig. 4. Management of the pool of threads within a cluster.



(a) MCT strategy. (b) MCT strategy with clusters.

Fig. 5. Adaptation of the MCT scheduling strategy cluster.

to complete it the sooner. Thus, this scheduling policy is particularly well adapted to heterogeneous architectures as it takes into account hardware characteristics in its estimations (e.g. some tasks may be faster on GPU than CPU). By doing so, this scheduling strategy is able to capture the speedup and the affinities between the types of tasks composing the task graph and the types of computing resources. As an illustration, we provide in Figure 5(a) an example showing the behavior of the MCT strategy. In this scenario, we can see that the blue task corresponding to the task that has to be scheduled has different durations on CPU and GPU devices. The choice is then made to schedule it on the CPU0 which will complete it first (the expected duration of the task is based on the performance model previously mentioned). In this section, we propose an adaptation of the MCT scheduler suited for parallel tasks.

In StarPU, the task's estimated length and transfer time used for MCT decisions is computed thanks to performance models. These models are actually based on performance history tables dynamically built during the application execution. It is then possible for the runtime system to predict the execution time of each type of tasks on each type of worker. Therefore, even without the programmer's involvement, the runtime can provide a relatively accurate performance estimation of the expected requirements of the tasks allowing the scheduler to take appropriate decisions when assigning tasks to computing resources.

We adapt the MCT strategy and the underlying performance models to be able to select not only a worker but also a cluster as the resource which will execute the task. In a way, we introduce a new type of resources in the scheduler similarly to regular resources types (CPU, GPU, etc). This time the performance model is parametrized not only by the size and type of the task together with the candidate resource but also by the size of the resource if the latter is a cluster. Thus, tasks can be assigned to a cluster either explicitly by the user or by the policy depending on where it would finish first. This is illustrated in Figure 5(b), where this time the three CPUs composing our platform are grouped in a cluster. We can see that the expected duration of the parallel task on the cluster is used to choose the resource having the minimum completion time for the task. Note that in this scenario, we chose to illustrate a cluster with an OpenMP-like internal runtime system.

This approach allows dealing with a heterogeneous architecture made of different types of processing units as well as clusters grouping different sets of processing units. Therefore, our approach is able to deal with multiple sizes of clusters together with common workers and take appropriate decisions. In fact, it is helpful to think of the clusters as mini-accelerators.

In this work, we let the user define a set of such clusters (mini-accelerators) and schedule tasks dynamically on them, without changing automatically the configuration. While our approach allows changing the size of those clusters on the fly, defining a scheduler able to dimension clusters is still an ongoing research.

B. Case Study: parallel version of the Intel MKL library

In this context, we illustrate how the proposed framework can deal with parallel tasks by taking a specific example, namely the parallel multithreaded version of the Intel MKL library implemented on top of OpenMP. First of all, we recall that regular scheduling contexts allow the schedulers to run unmodified as guests in an isolated manner. In fact they are only informed on which resources (cluster) they are allowed to execute and then they assign the workload according to their algorithm. However, Intel MKL parallel library requires executing an inner runtime specific code in order to enforce the required resource allocation.

In order to deal with this requirement we rely on a StarPU functionality that allows a code provided by the programmer, which will be referred to as a *callback*, to be executed at different points of the execution of a given task. One such callback is the *prologue callback* which is executed by the master worker when creating the cluster. We integrate in this callback the specific code required by Intel MKL and OpenMP such that it is transparently triggered before starting executing any sequence of parallel tasks.

We show in Figure 6 an example of how this tool can be used. Our aim is to typically isolate and bind the execution of Intel MKL parallel tasks on specific parts of the machine. Therefore, we provide the implementation of the prologue callback that consists in creating the OpenMP team of threads needed by the kernel and binding them to the logical ids on which the corresponding scheduling context is allowed to execute. Further on, when executing the Intel MKL implementation of the parallel task, the inner scheduler reuses the previously created threads, that are correctly fixed on the required computing resources.

This approach can be generalized to other runtime systems as the programmer can provide the implementation of the prologue callback and thus use the necessary functions in order to indicate the resource allocation required for the corresponding scheduling context. Such a runtime should however respect certain properties: be able to be executed on a restricted set of resources and allow the privatization of its global and static variables.

V. Experimental Results

We evaluate our method on a Cholesky factorization, an application widely used in linear algebra. This algorithm is present in multiple linear algebra libraries [14], [15], which represent the computations as a DAG of tasks. We present in Figure 7 the DAG of the Cholesky factorization on a matrix containing 5x5 tiles.

For our evaluation, we use the Cholesky factorization of `Chameleon`[1] [30], a dense linear algebra library for heterogeneous platforms based on the StarPU runtime system. Similar to most task-based linear algebra libraries, `Chameleon` relies on a BLAS library for optimized implementations of the kernels (cuBLAS for NVidia GPUs, MKL for Intel CPUs and Co-Processors). Our adaptation of `Chameleon` does not change the high level task-based algorithms and subsequent DAGs. We simply extend the prologue of each task so as to use the OpenMP implementation of MKL inside contexts, as depicted in section IV-B and manage the clusters creations. We call `pt-Chameleon` this adapted version of `Chameleon` that handles parallel tasks.

We conducted our experiments on the PlaFRIM experimental testbed[2]. The machine we use is heterogeneous and composed by **two 12-cores Intel Xeon CPU E5-2680 v3** (@2.5 GHz equipped with 30 MB of cache each) and enhanced with **four NVidia K40m GPUs**. Each CPU possess a theoretical peak of 480 GFlop/s in double precision while each GPU boasts a theoretical peak of 1.43 TFlop/s.

All experiments have been carried out using the latest version of the StarPU runtime system. We use a recent version of the baseline `Chameleon` library and its adaptation `pt-Chameleon`. Everything was compiled using the Intel compiler version 2015.3.187. We use the same version of Intel MKL for the CPU BLAS implementation and CUDA 7.0.28 for the cuBLAS GPU kernels. Our `Chameleon` and `pt-Chameleon` experiments are conducted using the MCT scheduler implementation of StarPU adapted to the use of parallel tasks as described in section IV-A.

The rest of this section will consist in three main parts. First, we will present our `pt-Chameleon` implementation's behavior and illustrate its benefits compared to the `Chameleon` library in Figure 8. We will then present a more detailed study of the behavior of the Cholesky factorization using parallel-tasks in Figure 9. Finally, we will propose a scheduling optimization and compare our work to the existing reference implementations in Figures 11 and 10. We show in those last plots reference lines (horizontal solid lines in the plots) representing the peak performance of the most efficient BLAS3 kernel: DGEMM. The numbers come from the experiment depicted on Figure 1, where DGEMM reaches 700 GFlop/s on 20 cores; and 1.1 TFlop/s on each GPU. These are clearly upper bounds for a Cholesky factorization, which involves several other less efficient BLAS kernels.

It is important to note that on all figures we report our performance with 2 clusters aggregating 10 cores each. Indeed, empirical experiments showed that it is the configuration providing the best performance. We remind that the platform is composed of two CPU sockets having 12 cores each, from which two are used to control the GPU devices.

We report in figure 8 the performance behavior of our implementation of the Cholesky factorization of, `pt-Chameleon` framework, compared to the existing `Chameleon` library. This figure is set out in two dimensions in order to make the study more readable both in terms of scalability and granularity, which are two major benefits of our approach. We can observe that for the single GPU case, `pt-Chameleon` greatly outperforms `Chameleon` on

[1] https://project.inria.fr/chameleon/

[2] https://plafrim.bordeaux.inria.fr/

```c
int main()
{
  int cpus[3] = {CPU_1, CPU_2, CPU_3};

  /* define the table of resource ids */
  ctx = starpu_create_sched_ctx(cpus,3);

  /* submit the set of tasks to the context*/
  for( i = 0; i < ntasks; i++)
  {
    /* indicate the prologue callback */
    tasks[i].prologue = cl_prologue;

    /* indicate the codelet of the task*/
    tasks[i].codelet = codelet;

    /* submit the task to the context */
    starpu_task_submit_to_ctx(tasks[i], ctx);
  }
}
```

```c
#include "mkl.h"

void cl_prologue()
{
  /* context on which the task executes */
  int ctx = starpu_get_current_ctx();

  /* get the CPUs of the current context */
  int cpus[MAX_WORKERS];
  int ncpus = starpu_get_cpus(ctx, cpus);

 /* bind openmp threads to CPUs */
#pragma omp parallel num_threads(ncpus)
  bind_to_cpu(cpus[omp_get_thread_num()]);
}

void codelet_cpu_func()
{
  /* call the mkl parallel kernel */
  DGEMM(...);
}
```

Fig. 6. Executing Intel MKL parallel codes within a scheduling context

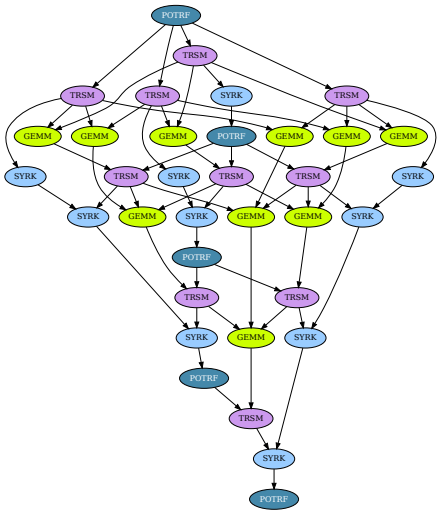


Fig. 7. Task diagram of a Cholesky factorization

		DPOTRF	DTRSM	DSYRK	DGEMM
960	1 core	1.7	8.8	27.5	29.8
	10 core	0.34	1.3	3.7	3.6
1920	1 core	5.8	17.2	31.3	31.1
	10 core	0.78	2.0	3.5	3.6

TABLE I. ACCELERATION FACTOR OF CHOLESKY FACTORIZATION KERNELS ON A GPU COMPARED TO SEVERAL CPU CONFIGURATIONS. WITH TILE SIZES OF 960 AND 1920.

relatively small matrix sizes. Moreover, this gap widens as we increase both the number of GPUs and the tile size. Indeed, the use of `pt-Chameleon` allows the StarPU runtime system to view the machine with less resources to give work to, the difference is two clusters to schedule on for the CPUs against 20 single cores for the `Chameleon` case. Since the amount of parallelism is limited with quite small matrix sizes, our parallel task framework is able to maximize CPU utilization with a smaller number of tasks. By relying on the multi-threaded Intel MKL our implementation exploits the internal parallelism of the tasks in an efficient way using 10 threads for all BLAS kernels even on tile sizes of 960 leading to an overall good performance. We are able to reach our peak performance on a matrix of 15K order with 1 GPU, 25K using 2 GPUs, 35K with 3 GPUs and 40K with 4 GPUs. It is interesting to note that as matrix size increases, the baseline `Chameleon` implementation is able to catch up with `pt-Chameleon`. This is explained by the high amount of parallelism available using tile sizes of 960, rendering the scheduling decisions easier. The sheer amount of parallelism allows the single core `Chameleon` library to reach a good overall occupancy of the machine. Another benefit of our approach is the flexibility regarding the granularity (tile size). Indeed, a tile size of 960 is usually used because it is a good trade-off for libraries such as `Chameleon` between reaching maximum GPU performance and providing enough parallelism for every CPU core. However, this trade-off limits the performance on architectures like the one we are using for this study. In comparison, our `pt-Chameleon` implementation is able to keep a sufficient amount of parallelism for our clusters even by increasing the tile size. In this figure, we can see a peak performance increase of over 200 GFlop/s for 4 GPUs between the tile sizes of 960 and 1920. This is mainly due to the fact that by increasing the tile size, GPU devices are more efficient while clusters provide enough computing power to not slow down the whole execution. This is illustrated in Table I which provides for different granularities, the relative performance of two cluster configurations for all the kernels used during the Cholesky factorization. This table also highlights we are able to accelerate the critical path (mainly composed of DPOTRF and DTRSM) in the clusters case, helping shorten the whole execution.

We report in Figure 9 the behavior of `pt-Chameleon` and `Chameleon` in two different cases in order to provide a detailed analysis of the interest of exploiting internal parallelism. To this end, we show the performance of both implementations using 4 GPUs and a tile size of 960 with two different settings for the data allocation scheme (by using the `numactl` tool). The solid lines with round points represent the performances of Figure 8 where we set the allocation policy as a round robin on all available memory nodes (two per socket) called “interleave all” in the legend, whereas the dashed lines with triangle-shaped points represent the performance when allocating all

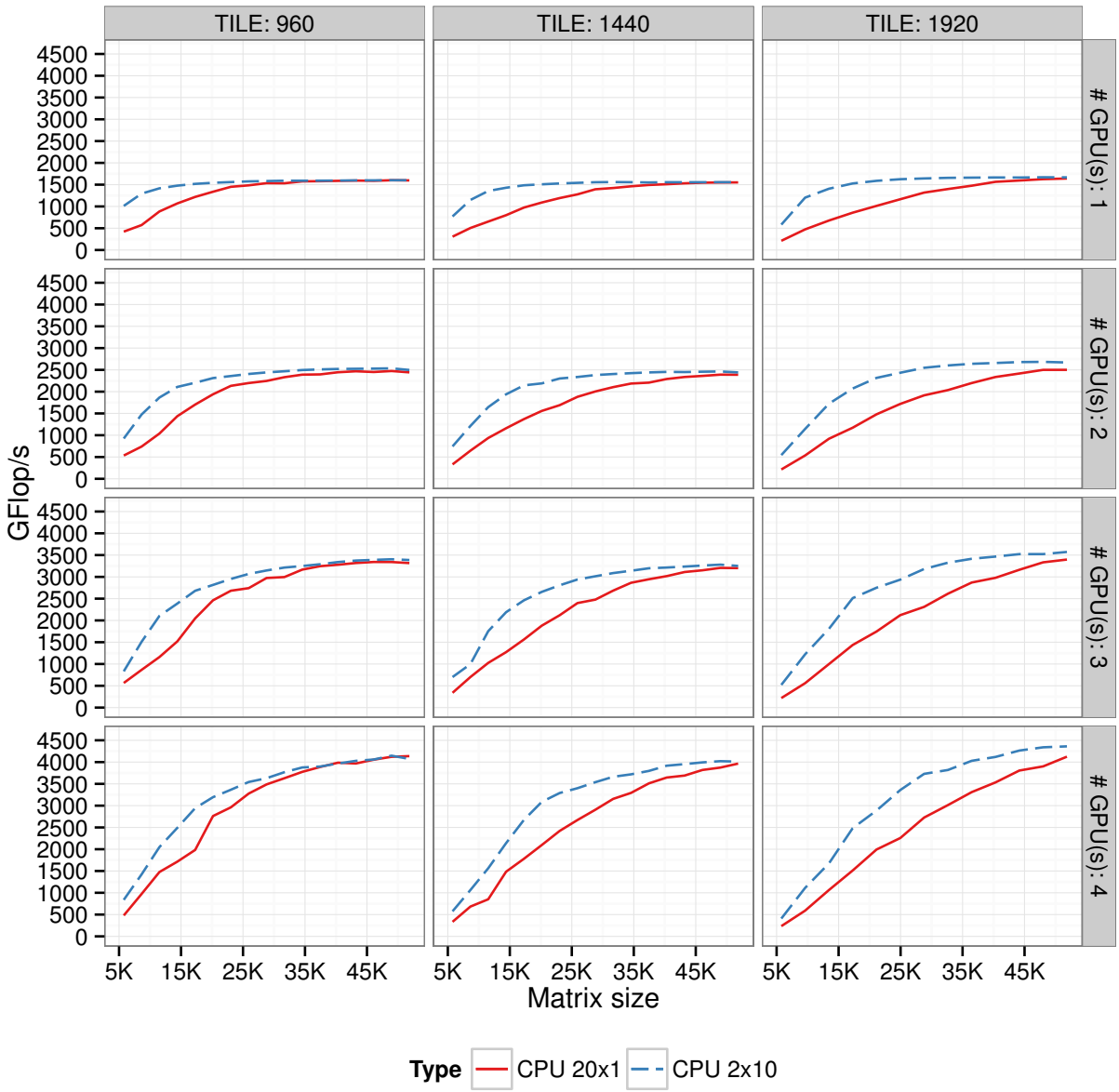


Fig. 8. Performance of the Cholesky factorization with `pt-Chameleon` and `Chameleon` with varying number of GPUs and tile sizes.

of the matrix on a single NUMA node, here the NUMA node 0, called "membind 0" in the legend. We have also added in this figure error bars showing the performance variations over 4 runs for each point. As this figure shows, there is a greater performance variation for the `Chameleon` library compared to the `pt-Chameleon` library. This is explained by two different issues: 1) performance gap between a single CPU core and a GPU leading to a CPU under-utilization, 2) bandwidth and cache efficiency of parallel tasks against sequential ones.

Firstly, we see a wide performance gap between `Chameleon` and `pt-Chameleon` together with a high instability of the `Chameleon` library with matrix of sizes 15K and 20K. These are explained by the difficulty of finding an efficient dynamic scheduling on this configuration together with the low amount of tasks available for scheduling and is highlighted by the numbers shown in Table I. This table shows the acceleration factor on all the kernels of the Cholesky factorization when using GPUs compared to two cluster configurations: one single core and a cluster of 10 cores. By relying on the factors provided in the table, MCT needs to schedule 30 DGEMM tasks on each GPU before it is beneficial to assign DGEMM tasks to CPU cores. In opposition, when using a cluster of 10 cores the acceleration factor is much smaller which eases the scheduling decision by creating a smaller hetereogeneity. In practice, we saw that for a matrix size of 15K for the `Chameleon` run the scheduler assigns all DPOTRF to CPUs, 10% DTRSM tasks to CPUs and under 1% DSYRK and DGEMM to the CPUs. In the

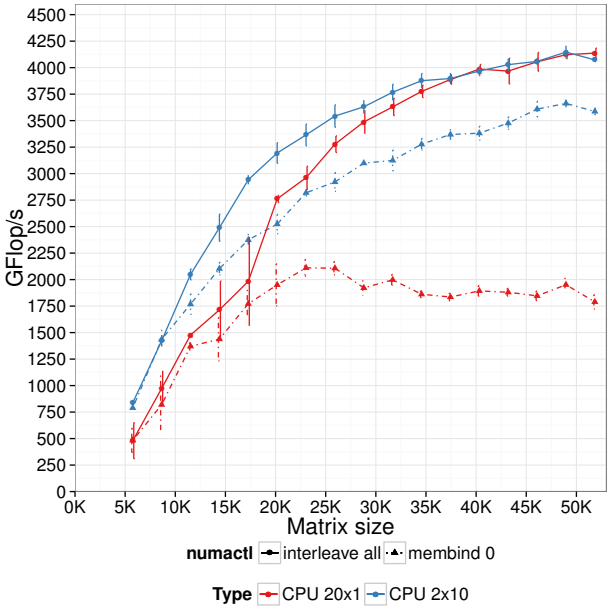


Fig. 9. Comparison of the `pt-Chameleon` Cholesky factorization with different NUMA settings. 20 CPUs and 4 GPUs are used.

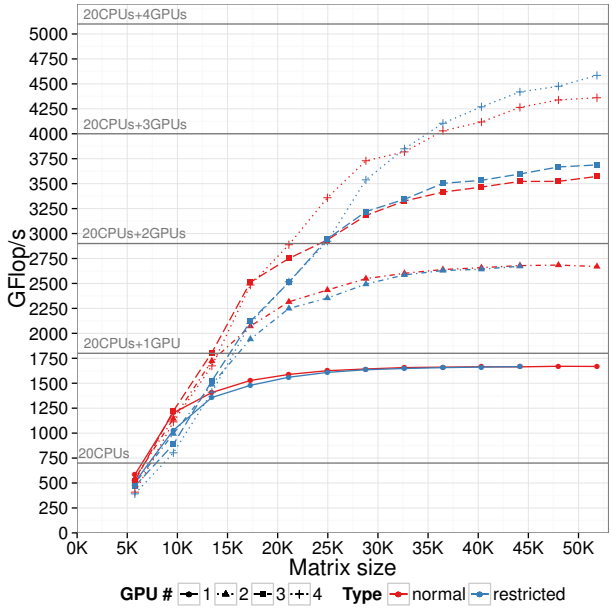


Fig. 10. Comparison of the regular `pt-Chameleon` Cholesky factorization and the constrained `pt-Chameleon`. 20 CPUs and 4 GPUs are used. The tile size is equal to 1920.

other hand, in the `pt-Chameleon` case when using clusters of 10 cores we saw that all DPOTRF kernels are executed on CPUs as this provides 3 times better performance than on GPUs, exactly 50% of TRSMs are executed on CPUs and finally 10% DGEMM and DSYRK are executed on the CPUs. In summary, the systematically higher performance on smaller matrix sizes comes from being able to make use of the full machine in the `pt-Chameleon` case, whereas the difficulty of managing such a high level of heterogeneity forces the scheduler to mostly use the GPU performance for the `Chameleon` case. Furthermore, with higher tile sizes such as 1920 the GPU acceleration factors against a single core increases, which explains the performance gap on Figure 8 with those tile sizes, while the use of a 10 core cluster allows to keep a close homogeneity between GPU and CPU performance grain.

Secondly, we are able to see performance instability for the `Chameleon` library on 4 GPUs with very large matrix sizes when using interleave settings of numactl. On these matrix sizes, scheduling on CPUs becomes easier thanks to the degree of parallelism available at all times. The performance instability seen for example for matrix sizes of 43K is explained by another constraint, the cache behaviour and bandwidth use. To highlight this claim, we show performance results when constraining the allocated data on one NUMA node. We therefore possess in this case less available DRAM access bandwidth, around 30GB/s instead of 120GB/s when using all the memory nodes. This plot shows a wide gap between `Chameleon` and `pt-Chameleon`. A detailed profiling analysis using the VTune[3] framework with matrix sizes of 43K shows that `Chameleon` has twice the amount of cache misses on critical kernels and uses the full available bandwidth almost all the time. More precisely, when considering the DGEMM kernel (which is a compute intensive kernel), the profiling study shows that the overall performance of these kernels is 59% (resp. 13%) bounded by memory bandwidth for `Chameleon` (resp. `pt-Chameleon`). Similar numbers for the "interleave all" case show that on some executions compute intensive kernels become slightly more memory bound for `Chameleon` on this platform (around 5%).

In Figure 10 we consider the execution of the Cholesky factorization on 2 clusters of 10 cores each with a tile size of 1920. We compare the performance of the application when using two different versions of the MCT scheduler: 1) the version described in Section IV-A, 2) a constrained version where some specific tasks, namely DPOTRF and DTRSM, are executed only on CPU workers, the two other tasks, DSYRK and DGEMM are scheduled on any resources according to the MCT scheduler. This second strategy is comparable to what is done in [16] where only DGEMM kernels are executed on GPU devices. We can observe that for small matrices, using the regular MCT scheduler leads to a better performance since in the constrained version the amount of work done by CPUs is too large. However, when the matrix size gets larger, the constrained version starts to be efficient and leads to a 5% increase in performance on average which leads to a peak performance of 4.6 TFlop/s on our test platform.

Finally in Figure 11 we compare `pt-Chameleon` using the constrained MCT scheduler to multiple reference dense linear algebra libraries: `MAGMA`, sequential `Chameleon` and `DPLASMA` using hierarchical the adaptive granularity scheme presented in [16]. We selected the setup providing the absolute highest peak performance for each library: a tile size of 960 for `Chameleon`, a tile size of 1920 for `pt-Chameleon`, the default settings with multi-threading for `MAGMA`, a tile size of 1920 with a CPU granularity of 320 for `DPLASMA`.

[3] https://software.intel.com/en-us/intel-vtune-amplifier-xe

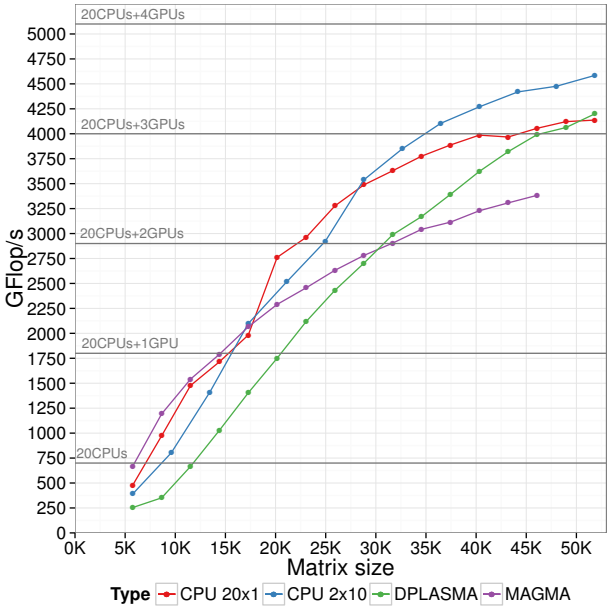


Fig. 11. Comparison of the constrained `pt-Chameleon` with baseline `Chameleon`, `MAGMA` and hierarchical `DPLASMA`. 20 CPUs and 4 GPUs are used.

We see that the absolute peak performance is obtained by `pt-Chameleon` that outperforms all the other implementations. We can see also that the adaptive granularity scheme provided by the `DPLASMA` allows is able to reach better peak performance than `MAGMA` and `Chameleon` but is still 300 Gflops/s below `pt-Chameleon` on the largest configuration.

VI. Conclusion

One of the biggest challenges raised by the design of high performance task-based applications on top of heterogeneous accelerator-based machines lies in choosing the adequate granularity of tasks.

Our approach consists in reducing the performance gap between accelerators and single cores by scheduling parallel tasks over clusters of CPUs. For this purpose, we have extended the concept of scheduling context, which we introduced in a previous work, so that the main runtime system only sees virtual computing resources on which it can schedule parallel tasks (e.g. BLAS kernels). The execution of tasks inside such clusters can virtually rely on any thread-based runtime system, and does not interfere with the outer scheduler. As a proof of concept we allow the interoperability of StarPU and OpenMP to co-manage task parallelism.

We present an evaluation of our approach on a regular dense linear algebra kernel. We show that our approach is able to outperform the `MAGMA`, `DPLASMA` and `Chameleon` state-of-the-art dense linear algebra libraries on all matrix sizes by using the same task granularity on accelerators and clusters.

In the near future, we intend to further extend this work along several directions. First, we plan to investigate how our approach could be applied to more complex irregular applications (e.g. sparse direct solvers or Fast Multipole Methods). On the hardware side, we also have conducted preliminary experiments that suggest our approach can also benefit to manycore accelerators such as the Intel Xeon Phi processor. Indeed, forming small clusters of cores on such a processor decreases cache pressure and helps placing kernels in situations where their performance is actually maximal.

In the long term, we plan to investigate dynamic task granularity adaptation, which is complementary to our approach. We believe mixing both approaches will allow the algorithms and applications designers to have more flexibility regarding the ability to use multiple granularities simultaneously.

Acknowledgment

We are grateful to Mathieu Faverge for his help for the comparison of `pt-Chameleon` and `DPLASMA`. This work is supported by the Agence Nationale de la Recherche, under grant ANR-13-MONU-0007. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PLaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS.

References

[1] 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.

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

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Computing*, vol. 38, no. Issues 1, pp. 37 – 51, 2012.

[4] 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.

[5] 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.

[6] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-gpu and multi-cpu parallelization for interactive physics simulations," in *Euro-Par 2010 - Parallel Processing*, 2010, vol. 6272, pp. 235–246.

[7] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, dec. 2009, pp. 45 –55.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 187–198, Feb. 2011.

[9] 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 2009*, 2009, pp. 851–862.

[10] 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

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

[12] 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.

[13] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, no. 3, 2009.

[14] 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*, vol. 180, no. 1.

[15] 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.

[16] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical dag scheduling for hybrid distributed systems," in *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Hyderabad, India, May 2015.

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

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

[19] 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.

[20] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM J. Scientific Computing*, vol. 36, no. 1, 2014. [Online]. Available: http://dx.doi.org/10.1137/130915662

[21] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 11, pp. 1935–1946, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3132

[22] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems," in *Proceedings of ICS'12*, 2012, pp. 365–376.

[23] J. V. F. Lima, F. Broquedis, T. Gautier, and B. Raffin, "Preliminary experiments with xkaapi on intel xeon phi coprocessor," in *25th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2013, Porto de Galinhas, Pernambuco, Brazil, October 23-26, 2013*, 2013, pp. 105–112. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2013.28

[24] K. Kim, V. Eijkhout, and R. A. van de Geijn, "Dense matrix computation on a heterogenous architecture: A block synchronous approach," Texas Advanced Computing Center, The University of Texas at Austin, Tech. Rep. TR-12-04, 2012.

[25] 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

[26] A. Marochko, "Tbb 3.0 task scheduler improves composability of tbb based solutions." 2012, http://software.intel.com/en-us/blogs/2010/05/13/tbb-30-task-scheduler-improves-composability-of-tbb-based-solutions-part-1/.

[27] Z. Majo and T. R. Gross, "A library for portable and composable data locality optimizations for numa systems," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688509

[28] A. Hugo, A. Guermouche, P. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: A supervised approach," *IJHPCA*, vol. 28, no. 3, pp. 285–300, 2014.

[29] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260–274, Mar 2002.

[30] 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," *GPU Computing Gems, Jade Edition*, vol. 2, pp. 473–484, 2011.