In Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue.

StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier

University of Bordeaux – LaBRI – INRIA Bordeaux Sud-Ouest

Abstract. In the field of HPC, the current hardware trend is to design multiprocessor architectures that feature heterogeneous technologies such as specialized coprocessors (*e.g.,* Cell/BE SPUs) or data-parallel accelerators (*e.g.,* GPGPUs). Approaching the theoretical performance of these architectures is a complex issue. Indeed, substantial efforts have already been devoted to efficiently offload parts of the computations. However, designing an execution model that unifies all computing units and associated embedded memory remains a main challenge.
We have thus designed StarPU, an original runtime system providing a high-level, unified execution model tightly coupled with an expressive data management library. The main goal of StarPU is to provide numerical kernel designers with a convenient way to generate parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand.
We have developed several strategies that can be selected seamlessly at run time, and we have demonstrated their efficiency by analyzing the impact of those scheduling policies on several classical linear algebra algorithms that take advantage of multiple cores and GPUs at the same time. In addition to substantial improvements regarding execution times, we obtained consistent *superlinear* parallelism by actually *exploiting* the heterogeneous nature of the machine.

1 Introduction

Multicore processors are now mainstream. To face the ever-increasing demand for more computational power, HPC architectures are not only going to be *massively* multicore, they are going to feature *heterogeneous* technologies such as specialized coprocessors (*e.g.,* Cell/BE SPUs) or data-parallel accelerators (*e.g.,* GPGPUs). As illustrated by the currently Top500-leading IBM RoadRunner machine, which is composed of a mix of Cells and Opterons, accelerators have indeed gained a significant audience. In fact, heterogeneity not only affects the design of computing units itself (CPU *vs* Cell's SPU), but also memory design (cache and memory banks hierarchy) and even programming paradigms (MIMD *vs* SIMD).

But is HPC, and its usual momentum, ready to face such a revolution? At the moment, this is clearly not the case, and progress will only come from our ability to harness such architectures while requiring minimal changes to programmers' habits. As none of the main programming standards (*i.e.,* MPI and OpenMP) currently address all the requirements of heterogeneous machines, important standardization efforts are needed. Unless such efforts are made, accelerators *will* remain a niche. In this respect,

the OpenCL initiative is clearly a valuable attempt in providing a common programming interface for CPUs, GPGPUs, and possibly other accelerators. However, the OpenCL API is a very low-level one which basically offers primitives for explicitely offloading tasks or moving data between coprocessors. It provides no support for task scheduling or global data consistency, and thus can not be considered as a true "runtime system", but rather as a virtual device driver.

To bridge the gap between such APIs and HPC applications, one crucial step is to provide optimized versions of computation kernels (BLAS routines, FFT, and other numerical libraries) capable of running seemlessly over heterogenenous architectures. However, since no performance model that would allow to use a static hardware resource assignment is likely to emerge in the near future, these kernels must be designed to dynamically adapt themselves to the available resources and the application load.

As an attempt to provide a runtime system allowing the implementation of such numerical kernels, we have recently developed a data-management library that seamlessly enforces a coherent view of all hardware memory banks (*e.g.,* main memory, SPU local store, GPU on-board memory, etc.) [1]. This library was successfully used to implement some simple numerical kernels quite easily (using a straightforward scheduling scheme), the next step is now to abstract the concept of task on heterogeneous hardware and to provide expert programmers with scheduling facilities. Integrated within high-level programming environments such as OPENMP, this would indeed help application developers to concentrate on high-level algorithmic issues, regardless of the underlying scheduling issues.

We here propose STARPU, a simple tasking API that provides numerical kernel designers with a convenient way to execute parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand. STARPU is based on the integration of the data-management facility with a task execution engine. We demonstrate the relevance of our approach by showing how heterogeneous parallel versions of some numerical kernels were developed together with advanced scheduling policies.

Section 2.2 presents the unified model we propose to design in STARPU. In section 3, we enrich our model with support for scheduling policies. We study how scheduling improves the performance of our model in section 4. We compare our results with similar works in section 5, and section 6 draws a conclusion and plans for future work.

2 The STARPU runtime system

Each accelerator technology usually has its specific execution model (*e.g.,* CUDA for NVIDIA GPUs), and its proper interface to manipulate data (*e.g.,* DMA on the CELL). Porting an application to a new platform therefore often boils down to rewriting a large part of the application, which severely impairs productivity. Writing portable code that runs on multiple targets is currently a major issue, especially if the application needs to exploit multiple accelerator technologies, possibly at the same time.

To help tackling this issue, we designed STARPU, a runtime layer that provides an interface unifying execution on accelerator technologies as well as multicore processors. Middle layers tools (such as programming environments and HPC libraries) can

build up on top of StarPU (instead of directly using low-level offloading libraries) to keep focused on their specific role instead of having to handle efficient simultaneous use of offloading libraries. That allows programmers to make existing applications efficiently exploit different accelerators with limited effort. We now present the main components of StarPU: a high level library that takes care of transparently performing data movements, already described in a previous article [1], and a unified execution model.

2.1 Data Management

As accelerators and processors usually cannot transparently access the memory of each other, performing computations on such architectures implies explicitly moving data between the various computational units. Considering that multiple technologies may interact, and that specific knowledge is required to handle the variety of low-level techniques, in previous work [1] we designed a high level library that efficiently automates data transfers throughout heterogeneous machines. It uses a software MSI caching protocol to minimize the number of transfers, as well as partitioning functions and eviction heuristics to overcome the limited amount of memory available on accelerators.

2.2 An accelerator-friendly unified execution model

The variety of technologies makes accelerator programming highly dependant on the underlying architecture, so we propose a uniform approach for task and data parallelism on heterogeneous platforms. We define ***codelets*** as an abstraction of a task (*e.g.,* a matrix multiplication) that can be executed on a core or offloaded onto an accelerator using an asynchronous continuation passing paradigm. Programmers supply implementations of *codelets* for each of the architectures that can execute them, using their respective usual programming languages (*e.g.,* CUDA) or libraries (*e.g.,* BLAS routines). An application can then be described as a set of *codelets* with data dependencies.

Declaring tasks and data dependencies A *codelet* includes a high level description of the data and the type of access (*i.e.,* read, write or both) that is needed. Since *codelets* are launched asynchronously, this allows StarPU to reorder tasks in case this improves performance. By declaring dependencies between tasks, programmers can just let StarPU automatically enforce the actual dependencies between *codelets*.

Designing *codelet* drivers As the aim of *codelets* is to offer a uniform execution model, it is important that it be simple enough to fit to the various architectures that might be targeted. In essence, adding the support of *codelets* for a new architecture means writing a driver that continuously does the following: request a *codelet* from StarPU, fetch its data, execute the implementation of the *codelet* for that architecture, perform its callback function and tell StarPU to unlock tasks waiting for this *codelet*'s completion. That model has been successfully implemented on top of CUDA, on multicore multiprocessors, and we are porting it to Cell's coprocessors (as we have already successfully used a similar approach in previous work in the Cell Runtime Library [12]).

3 A generic scheduling framework for heterogeneous architectures

The previous section has shown how tasks can be executed on the various processing units of a heterogeneous machine. However, we did not specify how they should be distributed efficiently, especially with regards to load balancing. It should be noted that nowadays architectures have gotten so complex that it is very unlikely that writing portable code which efficiently maps tasks statically is either possible or even productive.

3.1 Scheduling tasks in a heterogeneous world

Data transfers have an important impact on performance, so that a scheduler favouring locality may increase the benefits of caching techniques by improving data reuse. Considering that multiple problems may be solved concurrently, and that machines are not necessarily fully dedicated (*e.g.,* when coupling codes), dynamic scheduling becomes a necessity. In the context of heterogeneous platforms, performance vary a lot according to architectures (*i.e.,* in terms of raw performance) and according to the workload (*e.g.,* SIMD code *vs.* irregular memory access). It is therefore crucial to take the specificity of each computing unit into account when assigning work.

Similarly to the problems of data transfers or task offloading, heterogeneity makes the design and the implementation of portable scheduling policies a challenging issue. So we propose to extend our uniform execution model with a uniform interface to design *codelet* schedulers. StarPU offers low level scheduling mechanisms (*e.g.,* work stealing) so that scheduler programmers can use them in a high level fashion, regardless of the underlying (possibly heterogeneous) target architecture. Since all scheduling strategies have to implement the same interface, they can be programmed independently from applications, and the user can select the most appropriate strategy at runtime.

In our model, each **worker** (*i.e.,* each computation resource) is given an *abstract* queue of *codelets*. Two operations can be performed on that queue: task submission (`push`), and request for a task to execute (`pop`). The actual queue may be shared by several workers provided its implementation takes care of protecting it from concurrent accesses, thus making it totally transparent for the *codelet* drivers. All scheduling decisions are typically made within the context of calls to those functions, but there is nothing that prevents a strategy from being called in other circumstances or even periodically.

In essence, defining a scheduling policy consists in creating a set of queues and associating them with the different workers. Various designs can be used to implement the queues (*e.g.,* FIFOs or stacks), and queues can be organized according to different topologies (*e.g.,* a central queue, or per-worker queues). Differences between strategies typically result from the way one of the queue is chosen when assigning a new *codelet* after its submission by the means of a `push` operation.

3.2 Writing portable scheduling algorithms

Since they naturally fit our queue-based design, all the strategies that we have written with our interface (see Table 1) implement a greedy *list scheduling* paradigm: when a

ready task (*i.e.,* all its dependencies are fulfilled) is submitted, it is directly inserted in one of the queues, and former scheduling decisions are not reconsidered. Contrary to DAG scheduling policies, we do not schedule tasks that are not yet ready: when the last dependency of a task is executed, StarPU schedules it by the means of a call to the usual `push` function.

Restricting ourselves to list scheduling may somehow reduce the generality of our scheduling engine, but this paradigm is simple enough to make it possible to implement portable scheduling policies. This is not only transparent for the application, but also for the drivers which request work. This simplicity allows people working in the field of scheduling theory to branch higher level tools to use StarPU as an experimental playground.

Moreover, preliminary results confirm that using *codelet* queues in the scheduling engine is powerful enough to efficiently exploit the specificities of the Cell processor; and the design of OpenCL is also based on task queues: list scheduling with our `push` and `pop` operations is a simple, yet expressive paradigm.

3.3 Scheduling hints

To investigate the possible scope of performance improvements thanks to better scheduling, we let the programmer add some extra optional scheduling hints within the *codelet* structure. One of our objective is to fill the gap between the tremendous amount of work that has been done in the field of scheduling theory and the need to benefit from those theoretical approaches on actual machines.

Declaring prioritized tasks The first addition is to let the programmer specify the level of priority of a task. Such priorities typically prevent crucial tasks from having their execution delayed too much. While describing which tasks should be prioritized usually makes sense from an algorithmic point of view, it could also be possible to infer it provided an analysis of the task DAG.

Guiding scheduling policies with performance models Many theoretical studies of scheduling problems often assume to have a *weighted* DAG of the tasks [2]. Whenever it is possible, we thus propose to let the programmer specify a performance model to extend the dependency graph with weights. Scheduling policies can subsequently eliminate the source of load imbalance by distributing work with respect to the amount of computation that has already been attributed to the various processing units.

The use of performance models is actually fairly common in high performance libraries. Various techniques are thus available to allow the programmer to make performance predictions. Some libraries exploit the performance models of computation kernels that have been studied extensively (*e.g.,* BLAS). This for instance makes it possible to select an appropriate granularity [17] or even a better (static) scheduling [15]. It is also possible to use sampling techniques to automatically determine such model costs, provided actual measurements. In StarPU that can be done either by the means of a pre-calibration run, using the results of previous executions, or even by dynamically adapting the model with respect to the running execution.

The heterogeneous nature of the different workers makes performance prediction even more complex. Scheduling theory literature often assumes that there is a mathematical model for the amount of computation (*i.e.,* in Flop), and that execution time

Table 1. Scheduling policies implemented using our interface

Policy	Category	Queue design	Load balancing
default	greedy	central FIFO	n/a
priority	greedy	central deque or priority FIFO	n/a
work-stealing	greedy	per-worker deque	steal
weighted random	directed	per-worker FIFO	model
cost model	directed	per-worker FIFO	model

may be computed according to the relative speed of each processor (*i.e.,* in FLOP/S) [2]. However, it is possible that a *codelet* is implemented using different algorithms (with different algorithmic complexities) on the various architectures. As the efficiency of an algorithm heavily depends of the underlying architecture, another solution is to create performance models for each architecture. In the following section, we compare both approaches and analyze the impact of model accuracy on performances.

3.4 Predefined scheduling policies

We currently implemented a set of common queue designs (stack, FIFO, priority FIFO, deques) that can be directly manipulated within the different methods in a high level fashion. The policy can also decide to create different queue topologies, for instance a central queue or per-worker queues. The `push` and the `pop` methods are then responsible for implementing the load balancing strategy. Defining a policy with our model only consists in defining a couple of methods. On the one hand, a method called at the initialization of STARPU. On the other hand, the `push` and the `pop` methods that implement the interaction with the abstract queue.

Table 1 shows a list of scheduling policies that were designed usually in less than 100 lines of C code, which shows the conciseness of our approach.

4 Experimental validation

To validate our approach, we present several scheduling policies and experiment them in STARPU on a few applications. To investigate the scope of improvements that scheduling can offer, we gradually increase the quality of the hints given to STARPU by the programmer. We then analyze in more details how STARPU takes advantage of proper scheduling to exploit heterogeneous machines efficiently.

4.1 Experimental testbed

Our experiments were performed on an E5410 XEON quad-core running at 2.33 GHZ with 4 GB of memory and an NVIDIA QUADRO FX4600 graphic card with 768 MB of embedded memory. This machine runs LINUX 2.6 and CUDA 2.0. We used the ATLAS 3.6 and the CUBLAS 2.0 implementations of the BLAS kernels. All measurements were performed a significant number of times and unless specified otherwise, the standard deviation is never above 1 % of the average value which we show. Given

CUDA requirements, one core is dedicated to controlling the GPU efficiently [1] so that we compute on three cores and a GPU at the same time.

We have implemented several (single precision) numerical algorithms that use *codelets* in order to analyze the behaviour of StarPU and the impact of scheduling on their performance. **A blocked matrix multiplication** which will help us demonstrate that greedy policies are not always effective, even on such simple algorithms. **A blocked Cholesky decomposition** (without pivoting) which emphasizes the need for priority-based scheduling. **A blocked LU decomposition** (without pivoting) which is similar to Cholesky but performs twice as much computation, and thus parallelism. This demonstrates how our system tackles load balancing issues while actually taking advantage of a heterogeneous platform.

All these algorithms are compute-bound as they mostly involve BLAS 3 kernels ($\mathcal{O}(n^3)$ operations against $\mathcal{O}(n^2)$ memory accesses). Performance figures are shown in synthetic GFlop/s as this gives an evaluation of the efficiency of the computation since speedups are not relevant on such heterogeneous platforms.

4.2 Impact of the design of the queues

The choice of the design and the organization of the queues that compose a strategy is important when writing a scheduling strategy. The choice between a FIFO and a stack may also be important: a stack may for instance help to improve locality and thus data reuse, especially with divide-and-conquer algorithms, but implementing priority is easier with a FIFO. As all our benchmarks naturally tend to have a FIFO task ordering, measurements are not performed on stack-based strategies since that is irrelevant.

Even if it may require some load balancing mechanisms, decentralising queues helps to reduce contention and makes it possible to handle each worker specifically. This is also interesting when accessing a global shared queue is expensive (*e.g.,* on the Cell which needs expensive DMA transfers).

Scheduling policies with support for priority tasks helps reducing load imbalance for algorithms that suffer insufficient parallelism. Figure 1 shows that the Cholesky algorithm benefits from priorities by up to $10\,GFlops$ on large problems. However, it does not affect LU decomposition a lot on Figure 2 as the FIFO ordering naturally fits the natural priorities of the algorithm. On Figure 2, priority tasks are either appended at the end of a global FIFO (*greedy* policy) or put into dedicated ones (*priority* FIFO). Small problems benefit from a strict FIFO ordering, but large ones perform better with the greedy algorithm. This confirms that selecting the best scheduling policy is not obvious. That is why we made it possible to select the scheduling policy at runtime.

4.3 Policies based on performance models

Figure 3 demonstrates that the *greedy* policy delivers around 105 GFlops on medium-sized problems, which is about $90\,\%$ of the sum of the results achieved on a single GPU (91.3 GFlops) and 3 cores (25.5 GFlops). That relatively low efficiency is explained by the important load imbalance. Intuitively, that issue can be solved if a GPU that is n times faster than a core is given n times more tasks: next section shows how that can be achieved using performance models.

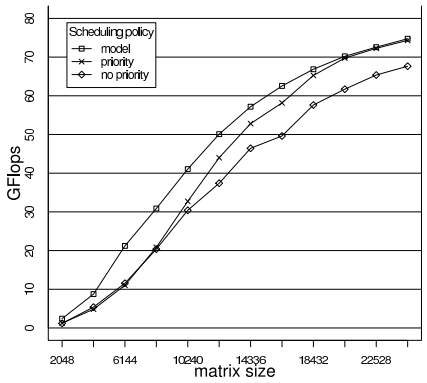

Fig. 1. Cholesky decomposition

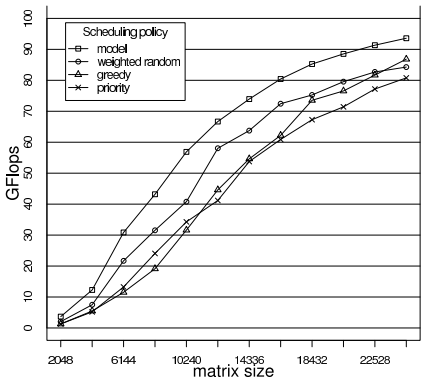

Fig. 2. LU decomposition

Average acceleration-based performance model In the *weighted random* strategy, each worker is associated with a ratio that can be interpreted as an *acceleration* factor. Each time a task is submitted, a random number is generated to select one of the workers with a probability proportional to its ratio. That ratio can for instance be set by the programmer once for all on the machine, or be measured thanks to reference benchmarks (*e.g.,* BLAS kernels). The *weighted random* strategy is typically be suited to independent tasks of equal size.

Still, the shaded area on Figure 3 shows that this policy produces extremely variable schedules for which the lack of load balancing mechanism explains why the average value is worse than the *greedy* policy. Unexpectedly, this strategy also gives really interesting improvement of an order of 10 GFLOPS on LU and Cholesky decompositions even though the latter is not shown on Figure 1 for readability reasons. It is interesting to note that this optimization is effective on the entire spectrum of sizes, especially on medium ones for which it is especially interesting to obtain performance improvements without any effort from programmers.

Per-task accurate performance models Evenly distributing tasks over workers with regard to their respective speed does not necessarily make sense for tasks that are not equally expensive, or if there are dependencies between them. Hence, programmers can specify a *cost model* for each *codelet*. This model can for instance represent the amount of work and be used in conjunction with the relative speed of each workers. It can also directly model the execution time on each of the architectures. We implemented the *Earliest Task First* strategy using those models. Each worker is assigned a queue. Given their expected duration, tasks are then assigned to the queues which minimize termination time.

By severely reducing load balancing issues, we obtained substantial improvements over all our previous strategies, for all benchmarks. Even though there is a limited

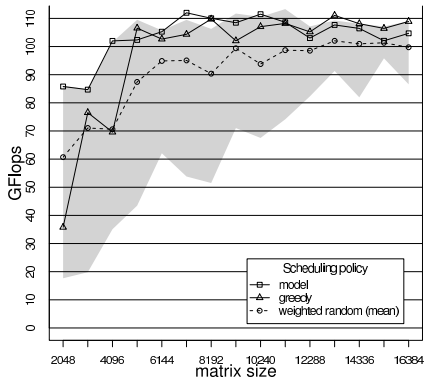

Fig. 3. Blocked Matrix Multiplication

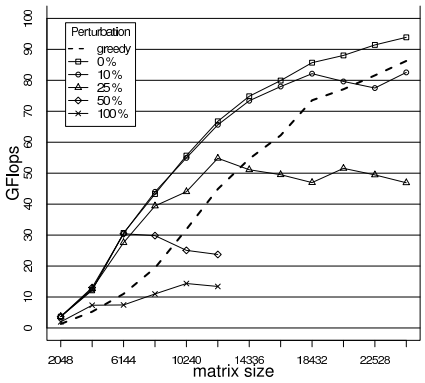

Fig. 4. Impact of the model accuracy

Table 2. Superlinear acceleration on LU decomposition (30720×30720)

	Simple performance model			Per-architecture performance model		
Measured speed (GFlops)	3 CPUs + 1 GPU	3 CPUs	1 GPU	3 CPUs + 1 GPU	3 CPUs	1 GPU
	95.41	21.24	75.04	98.21	21.68	75.07
Efficiency	$95.41 =$ **99.1 %** $(21.24 + 75.04)$			$98.21 =$ **101.5 %** $(21.68 + 75.07)$		

number of independent tasks, we always obtain almost the best execution for matrix multiplication. On the CHOLESKY decomposition, performance models outperform our other strategies on any sizes. This strategy improves the performance of medium and small sized problems up to twofold, but we observe equivalent results for large ones because the *priority* strategy does not suffer too much load imbalance on large inputs. The *performance modeling* strategy is also handicapped by a limited support for priority tasks, which is especially useful on the Cholesky benchmark. Likewise, the LU decomposition obtains even more important improvements as we achieve up to 25 GFLOPS improvement, thus reducing execution time by a factor of 2 on medium size problems. It also increases asymptotic speed from 80 GFLOPS to more than 95 GFLOPS.

On Figure 4, we applied a random perturbation of the performance prediction to study how model accuracy affects scheduling. Even with large miss-predictions (*e.g.,* 25 %), the *performance modeling* strategy still outperforms other ones for medium size problems: the importance of accuracy depends on the size of the problem. But since the execution time is usually within less than 1 % of the prediction for BLAS kernels, it is worth paying a attention to an accurate modeling, even if it needs not give exact prediction to be useful.

4.4 Taking advantage of heterogeneity

In our *heterogeneous* context, we define **efficiency** as the ratio between the sum of the speeds obtained separately on each architecture and the speed obtained while using all

architectures at the same time. This indeed expresses how well we manage to add up the speeds of the different architectures. Table 2 shows the efficiency of our parallel code. While heterogeneity could impact programmability, our implementation of the LU decomposition is actually not affected by the use of various architectures at the same time: the speed measured with three CPUs and a GPU amounts to $99.1\,\%$ of the sum of the speeds measured separately with three cores on the one hand, and with one GPU on the other hand. This result is contrasted by the need to dedicate one core to the accelerator, but it demonstrates that the overhead is rather low, mostly caused by parallelization rather than heterogeneity itself.

In addition to that, Table 2 shows an interesting *superlinear* efficiency of $101.5\,\%$ when using a per-architecture performance model instead of a mere *accelerating factor* with a single performance model common to all architectures. This illustrates the impossibility to model the actual capabilities of the various computation units by the means of a mere *ratio*, even though this model is fairly common in theoretical scheduling literature [2]. Some tasks indeed suit GPUs while others are relatively more efficient on CPUs: matrix multiplication may be ten times faster on a GPU than on a core while a CPU may be only five times slower on matrix additions. The rationale behind this *superlinear* efficiency is that it is better to execute the tasks you are good at, and to let others perform those for which you are not so good.

5 Related work

If accelerators have received a lot of attention in the last years, most people program them directly on top of constructors' API at a low level, with little attention to code portability. While GPGPUs were historically programmed using standard graphical APIs [14], AMD's FIRESTREAM and especially NVIDIA's CUDA are by far the most common way to program GPUs nowadays. Likewise, CELL is usually programmed directly on top of the low level LIBSPE interface even if IBM ALF targets both CELL and multicore processors. FPGA, CLEARSPEED and all other accelerating boards still need specific vendor interfaces. Most efforts however tend to be around writing fast computation kernels rather than designing a generic programming model.

In contrast, multicore (and SMP) programming is getting more mature and *standards* such as OPENMP, which has gained substantial audience in spite of MPI which still remains the most commonly used standard in HPC. Besides the OPENCL standardization effort which not only attempts to unify programming paradigms, but also proposes a low-level device interface, a lot of projects thus try to implement the MPI standard on the CELL [13] while it does not seem to be adapted for GPUs even if it becomes common to use hybrid models (*e.g.,* CUDA with MPI processes). DURAN *et al.* propose to enrich OPENMP with directives to declare data dependencies [8], which is particularly useful for all accelerator technologies. OPENMP therefore seems to be a promising programming interface for both CELL [4] and GPUs provided compiling environments are offered sufficient support. STARPU could thus be used as a back-end for CELLSS or for the HMPP [7] which generate *codelets* resp. for the CELL and for GPUs.

A lot of efforts have also been devoted to design or to extend languages with a proper support for data and task parallelism, but most of them actually re-implement a streaming paradigm [11] which does not necessarily capture all applications that may exploit accelerators. Various projects intend to implement libraries with computation kernels that are actually offloaded [5], but STARPU avoids for instance the limitation of the size of problems solved by BARRACHINA *et al.* [3] while preserving the benefits of their work at the algorithmic level.

Some runtime systems were designed to address multicore and accelerators architectures [6, 12, 16]. Most approaches adopt an interface similar to CHARM++'s asynchronous OFFLOAD API. The well established CHARM++ runtime system actually offers support for both CELL [10] and GPUs [16] (even though there are no performance evaluation available yet for GPUs to the best of our knowledge). But its rather low level interface only has limited support for data management: offloaded tasks only access blocks of data instead of our high level arbitrary data structures, and they do not benefit from our caching techniques. JIMENEZ *et al.* use performance prediction to schedule tasks between a CPU and a GPU [9], but their approach is not applicable to scheduling inter-dependent tasks since data transfers are explicit.

6 Conclusion and future work

We presented STARPU, a new runtime system that efficiently exploits heterogeneous multicore architectures. It provides a uniform execution model, a high-level framework to design scheduling policies and a library that automates data transfers. We have written several scheduling strategies and observed how they perform on some classical numerical algebra problems.

In addition to improving programmability by the means of a high level uniform approach, we have shown that applying simple scheduling strategies can significantly reduce load balancing issues and improve data locality. Since there exists no ultimate scheduling strategy that addresses all algorithms, programmers who need to hard-code task scheduling within their hand-tuned code may experiment important difficulties to select the most appropriate strategy. Many parameters may indeed influence which policy is best suited for a given input. Empirically selecting at runtime the most efficient one makes it possible to benefit from scheduling without putting restrictions or making excessive assumptions. We also demonstrated that given a proper scheduling, it is possible to exploit the specificity of the various computation units of a heterogeneous platform and to obtain a consistent superlinear efficiency.

It is crucial to offer a uniform abstraction of the numerous programming interfaces that result from the advent of accelerator technologies. Unless such a common approach is adopted, it is very unlikely that accelerators will evolve from a *niche* with dispersed efforts to an actual mainstream technique. While the OPENCL standard also does provide task queues and an API to offload tasks, our work shows that such a programming model needs to offer an interface that is simple but also expressive.

We plan to implement our model on additional accelerator architectures, such as the CELL by the means of a driver for the CELL RUNTIME LIBRARY [12], or on top of generic accelerators with an OPENCL driver. In the future, we expect STARPU to offer

support for the HMPP compiling environment [7] which could generate our *codelets*. We are also porting real applications such as the PASTIX [15] and the MUMPS solvers. STARPU could be a high-level platform to implement some of the numerous policies that exist in the scheduling literature. This would reduce the gap between HPC applications and theoretical works in the field of scheduling.

References

1. Cédric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multicore architectures. In *Euro-Par 2008 Workshops - Parallel Processing*, Las Palmas de Gran Canaria, Spain, August 2008.
2. Cyril Banino, Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, and Yves Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):319–330, 2004.
3. Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, and Enrique S. Quintana-Ort. Solving Dense Linear Systems on Graphics Processors. Technical report, Universidad Jaime I, Spain, February 2008.
4. Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.
5. Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures, 2007.
6. C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF '08*, 2008.
7. R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment, 2007.
8. A. Duran, J. M. Perez, E. Ayguade, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependant tasks. In *IWOMP Proceedings*, 2008.
9. Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC*, pages 19–33, 2009.
10. David Kunzman. Charm++ on the Cell Processor. Master's thesis, Dept. of Computer Science, University of Illinois, 2006.
11. M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *GSPx Multicore Applications Conference*, 2006.
12. Maik Nijhuis, Herbert Bos, Henri E. Bal, and Cédric Augonnet. Mapping and synchronizing streaming applications on cell processors. In *HiPEAC*, pages 216–230, 2009.
13. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1), 2006.
14. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 03 2007.
15. Pierre Ramet and Jean Roman. Pastix: A parallel sparse direct solver based on a static scheduling for mixed 1d/2d block distributions. In *Proceedings of Irregular'2000, Cancun, Mexique*, pages 519–525. Springer Verlag, 2000.
16. Lukasz Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008.
17. R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.