Data-Aware Task Scheduling on Multi-Accelerator based Platforms

Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, Raymond Namyst
University of Bordeaux – LaBRI – INRIA Bordeaux Sud-Ouest
Bordeaux, France

***Abstract*—To fully tap into the potential of heterogeneous machines composed of multicore processors and multiple accelerators, simple offloading approaches in which the main trunk of the application runs on regular cores while only specific parts are offloaded on accelerators are not sufficient. The real challenge is to build systems where the application would permanently spread across the entire machine, that is, where parallel tasks would be dynamically scheduled over the full set of available processing units.**

To face this challenge, we previously proposed StarPU, a runtime system capable of scheduling tasks over multicore machines equipped with GPU accelerators. StarPU uses a software virtual shared memory (VSM) that provides a high-level programming interface and automates data transfers between processing units so as to enable a dynamic scheduling of tasks. We now present how we have extended StarPU to minimize the cost of transfers between processing units in order to efficiently cope with multi-GPU hardware configurations. To this end, our runtime system implements data prefetching based on asynchronous data transfers, and uses data transfer cost prediction to influence the decisions taken by the task scheduler.

We demonstrate the relevance of our approach by benchmarking two parallel numerical algorithms using our runtime system. We obtain significant speedups and high efficiency over multicore machines equipped with multiple accelerators. We also evaluate the behaviour of these applications over clusters featuring multiple GPUs per node, showing how our runtime system can combine with MPI.

I. Introduction

The High Performance Computing community has recently witnessed a major evolution of parallel architectures. The invasion of multicore chips has impacted almost all computer architectures, going from laptops to high-end parallel computers. While researchers are still having a hard time bridging the gap between the theoretical performance of multicore machines and the sustained performance achieved by current software, they now have to face yet another architecture trend: the use of specific-purpose processing units as side accelerators to speed up computation-intensive applications.

Currently, the most widespread accelerators are Graphical Processing Units (GPU), which are massively parallel devices that can run SIMD programs very efficiently. They are mostly used as co-processors to speed up data parallel computations. The gap with classical multiprocessor architectures is so huge that simply enhancing existing solutions is not an option. GPU accelerators feature their own local storage that is not kept consistent with the main host memory. Thus, all data transfers between the main memory and the accelerators must be done explicitly. Furthermore, because they come with a specific instruction set following a SIMD execution model, the code executed by these accelerators must be generated by a specific compiler, which makes it impossible to migrate tasks between processors and accelerators during their execution.

Recently, several parallel computer manufacturers have released machines featuring multiple GPU racks, with several GPUs attached to each shared-memory node. As a result, an increasing part of the HPC community is currently trying to understand how to design or to adapt applications to be able to exploit heterogeneous multicore architectures such as multi-GPU clusters. This requires to revisit almost the complete software stack, from the parallel languages down to the runtime systems. The main challenge is to build environments where the application would spread across the entire machine, that is, where parallel tasks would be *scheduled* over the full set of available processing units, as opposed to simply be *offloaded* to accelerators.

The StarPU [3] runtime system was designed to meet this goal, and is typically intended to serve as a target for numerical kernel libraries and parallel compilers. It is capable of scheduling tasks over heterogeneous machines in a very efficient manner, thanks to the use of auto-tuned performance models [1]. To perform the necessary data transfers between accelerators in a transparent way, StarPU features a software virtual shared memory subsystem in charge of transparently transferring input data to accelerators before tasks can start and access them.

By experimenting with StarPU on several numerical kernel applications (*i.e.* LU decomposition and Grid Stencil), we observed that these memory transfers had a highly negative impact over the overall execution time.

The contributions of this paper are the following. We have extended the StarPU runtime system to efficiently deal with multi-accelerator hardware configurations in which data transfers are a key issue. We present how fully asynchronous data transfers can be used to implement efficient data prefetching on GPU accelerators. We also introduce a new data aware scheduling policy, based on this prefetching mechanism, that uses an auto-tuned data transfer cost prediction engine to improve scheduling accuracy while reducing the occupancy of the memory buses. Finally, we have also

implemented a library that provides an MPI-like semantic on top of StarPU. The integration of MPI transfers within task parallelism is done in a very natural way by the means of asynchronous interactions between the application and StarPU. More generally, these interactions allow StarPU to accelerate legacy codes or third-party libraries written using various programming paradigms quite easily.

II. Scheduling tasks over multi-GPU heterogeneous machines

To deal with accelerator programming, a wide spectrum of new languages, techniques and tools have been designed, ranging from low-level development kits to new parallel languages. Although these efforts cover a wide range of functionality, most of them are still unable to exploit multi-GPU machines efficiently because of their lack of dynamic scheduling capabilities. This is what motivated the design of the StarPU runtime system.

A. The StarPU runtime system

StarPU is a runtime system for scheduling a graph of tasks onto a heterogeneous set of processing units, and is meant to be used as a back-end for *e.g.* parallel language compilation environments and High-Performance libraries. The two basic principles of StarPU is firstly that tasks can have several implementations, for some or each of the various heterogeneous processing units available in the machine, and secondly that transfers of data pieces to these processing units are handled transparently by StarPU, relieving application programmers from explicit data transfers.

To transfer data between processing units automatically, StarPU implements a software virtual shared memory using a relaxed consistency model and featuring data replication capabilities. The application just has to register the different pieces of data by giving their addresses and sizes in the main memory. StarPU tasks contain explicit references to all their input and output data, using handles returned by the virtual shared memory system. This way, the scheduler can automatically fetch the input data of a task before it gets executed, and write back the output whenever appropriate.

From the point of view of the application, tasks are just *pushed* to the scheduler, which then dispatches them onto the different processing units. StarPU was designed as a testbed for developing, tuning and experimenting with various scheduling policies in a portable way. Defining such a scheduling policy actually consists in creating a set of queues and associating them with the different processing units, and defining the code that will be triggered each time a new task gets ready to be executed, or each time a processing unit is about to go idle. Various designs can be used to implement the queues (*e.g.* FIFOs or stacks), and queues can be organized according to different topologies, which makes it possible to develop a wide range of scheduling policies independently from the hardware technologies.

An example of code written on top of StarPU is shown on Figure 2, and a more complete description of StarPU's programming interface as well as its internal design are available as a research report [2].

B. About the impact of data movements on overall performance

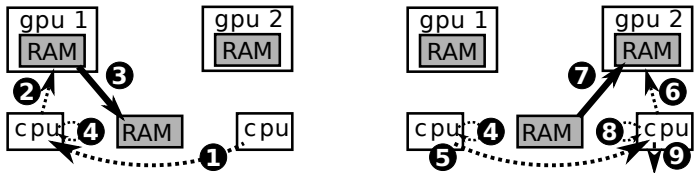
For non-trivial parallelism, communication is usually needed between the different processing units, to *e.g.* exchange intermediate results. This not only means data transfers between CPUs and GPUs, but also potentially between GPUs themselves, or even between CPUs, GPUs, and other machines on the network in the case of clusters.

The RAM embedded in GPUs usually provides a trade-off of a very high bandwidth but a non-negligible latency. The main I/O bus of the machine typically has a much lower bandwidth, but also a very high latency, due to quite huge overheads in software stacks like CUDA. NUMA factors also affect transfers, mostly their bandwidth, which can be seen cut by half, while the latency penalty is negligible compared to the software overhead. Eventually, cluster network interface cards (NICs) have a quite good latency, but their bandwidth is yet lower. As a result, with the increasing number of processing units and their increasing performance, data transfers become a critical performance concern since the memory bus can easily be a bottleneck.

StarPU already keeps track of where data have already been transferred to avoid spuriously consuming memory bandwidth by sending them again. To further optimize memory transfers, we extend StarPU to not only take benefit from asynchronous transfers supported by recent accelerators, but to also save yet more memory transfers by extending scheduling policies so as to improve their task placement decisions according to data locality and transfer costs. We also extend StarPU to improve the efficiency of the unavoidable data transfers by automatically trying to overlap them with computations.

III. Efficient data management within multi-accelerator based platforms

As the number of accelerators keeps growing, and even if an appropriate schedule should avoid superfluous data transfers, there usually remains some unavoidable transfers which can be critical for performance. Ideally, the different processing units should never stall waiting for a resource. A common solution to minimize the impact of communications is to overlap them with computations, which needs asynchronous support. In this section, we therefore present the new mechanisms implemented in StarPU to manage data asynchronously, and we extend the load-balancing capabilities previously offered by StarPU to take data movements into account in the scheduler.



(a) Step 1: GPU1 to RAM

(b) Step 2: RAM to GPU2

Figure 1. Transfer between two GPUs are implemented by the means of chained requests.

A. Asynchronous data requests within StarPU

Similarly to task execution requests, **data movement requests** can now be submitted asynchronously in the StarPU runtime system. Adding such *data requests* made it possible to run StarPU on multi-accelerator machines. When a processing unit needs to fetch some data into its local memory, it queries the data management facilities to locate every data replicate. Once the most appropriate source (*i.e.* owner) is chosen (*e.g.* the closest), an asynchronous *data request* is inserted in a queue attached to the source (there is one queue per memory node). When a data request is submitted to a memory node, any processing unit attached to this node can perform the data transfer. Usually, processing units poll for these requests each time they complete a task. An optional callback function may also be executed when a request is completed. For instance, this callback can be used to submit another data request.

Figure 1 shows that this permits **chained requests** which make it possible to transfer some piece of data D from GPU1 to GPU2. It should indeed be noted that because of technical constraints, it is currently impossible to transfer D directly from a NVIDIA GPU to another, so that an intermediate transfer through the main memory is still necessary. On Figure 1(a), the GPU2 worker (running on a dedicated CPU), which needs some data D, first posts a data request to fetch D into main memory (1). Once D is not busy any more, the GPU1 worker asks the GPU driver (*e.g.* CUDA) (2) to asynchronously copy D to main memory (3) [1]. On Figure 1(b), once the termination of the transfer is detected (4), a callback function posts a second data request between main memory and GPU2 (5). Steps (6), (7) and (8) are respectively equivalent to (2), (3) and (4). Finally, the GPU2 worker is notified once the data transfer is done (9).

This mechanism can deal with different types of accelerators (*e.g.* ATI and NVIDIA at the same time). If one has both a synchronous GPU and an asynchronous GPU, the transfer between main memory and the asynchronous GPU will still be done asynchronously. In the future, the use of data requests could be extended to implement more complex types of data transfer, such as direct transfers between a network card and a GPU when this is technically doable.

Data requests are a convenient abstraction for the scheduling policies: by keeping track of the on-going requests, it is possible to evaluate the activity of the memory subsystem. This can also be leveraged by scheduling policies by the means of simple yet powerful techniques.

B. Integrating data management and task scheduling

While our previous papers deal with the problem of task scheduling for machines equipped with a single accelerator and multiple CPUs, such platforms are typically accelerator-centric, so that most data remain on the single accelerator most of the time. Having multiple accelerators requires that we take care of data much better than before, for instance to reconsider whether it is worth moving data or not, we must have an idea of the actual overhead introduced by data transfers. In order to get the best from recent accelerators, we need to be able to overlap data transfers with computations as much as possible, so that we introduce prefetch mechanisms within the scheduler.

Data transfer overhead prediction. Getting an estimation of the time required to perform a data transfer is important in order to decide whether is it better to move data or to migrate computation to another processing unit. Since StarPU keeps track of the different data duplicates, it knows whether accessing some data requires a transfer or not. When StarPU is initialized for the first time on a machine, it evaluates both the bandwidth and the latency between each pair of memory nodes (*e.g.* GPU RAM to main memory). A rough estimation of data transfer times can then be derived from those numbers. Scheduling policies can thus now estimate the overhead introduced by data movements when assigning a task to some processing unit. This also permits to select the least expensive transfer when multiple duplicates are available over a non-uniform machine.

Data prefetching. The input data of a task should ideally be already available when it is about to start. To ensure that, once the scheduling policy has decided where a task should be run, prefetch requests are scheduled to transfer input data in the background while the previous tasks are still being executed. As StarPU keeps track of all the resources

[1]The GPU2 worker does not do this itself because switching CUDA contexts is very costly

Table I
SCHEDULING STRATEGIES BASED ON PERFORMANCE MODELS IMPLEMENTED IN STARPU

Name	Policy description
heft-tm	HEFT based on Task duration Models
heft-tm-pr	**heft-tm** with data PRefetch
heft-tmdp	**heft-tm** with remote Data Penalty
heft-tmdp-pr	**heft-tmdp** with data PRefetch

it allocated, a prefetch request will not be considered if it would prevent pending tasks from executing properly due to GPU memory usage.

C. Examples of scheduling strategies taking data transfers into accounts

Table I shows strategies that are implemented in StarPU. In previous work [3], we have shown that load balancing can greatly be improved by the means of performance models by implementing the HEFT scheduling algorithm (Heterogeneous Earliest Finish Time [15]) in a **heft-tm** policy. We now extend it to take data into account.

The rationale behind HEFT is to maintain an estimation of the dates when each processing unit should be available and to assign a new task to the worker which minimizes termination date. In the **heft-tm** policy, data transfers were not considered so only computation was taken into account. To let the scheduler enhance data locality, scheduling strategies should also consider data movements. In the **heft-tmdp** strategy, termination time is computed by considering both the duration of the computations and the data transfer overhead which StarPU can now evaluate. **heft-tmdp** thus makes it possible to improve data locality while enforcing load balancing.

The **heft-tm-pr** policy (resp. **heft-tmdp-pr**), extends **heft-tm** (resp. **heft-tmdp**) by using prefetching: when a task is assigned to a processing unit, and provided there is enough memory to handle the pending tasks, the input buffers are prefetched into the memory of the device.

IV. HYBRID PROGRAMMING MODELS

Requiring every application or library to be (re)written using the same programming model is not realistic. More generally, a hybrid programming model permits to select the paradigms that are most suited to the different parts of the application: if some heavily optimized kernel is already available in OpenMP or in TBB, we should not have to re-implement it in a less efficient way.

Hybrid models are not only needed for performance or to support legacy codes: when it comes to clusters of machines equipped with accelerators, it is necessary to be able to perform communications (typically with MPI). We therefore built a small library on top of StarPU which provides an MPI-like semantic to make it easier to accelerate MPI applications with StarPU.

Table II
FUNCTIONS PROVIDED BY OUR MPI-LIKE LIBRARY

Type of call	Examples of functions
Blocking	`starpu_mpi_{send,recv}`
Non-Blocking	`starpu_mpi_{isend,irecv}` `starpu_mpi_{test,wait}`
Array	`starpu_mpi_array_{isend,irecv}`
Detached	`starpu_mpi_{send,recv}_detached`

A. Adding support for MPI on top of StarPU

Data managed by StarPU are not only accessible from tasks, but also from the main loop of the application which can explicitly access them synchronously with a simple *acquire/release* semantic. For instance, the application can require StarPU to make sure the data is available in the main memory (acquire), then send it with MPI, and then release it so that StarPU tasks can modify it again.

Mixing MPI and task parallelism however remains a difficult problem. It is tedious to describe synchronous MPI transfers *a priori* while constructing a task graph in an asynchronous way. While the *acquire/release* semantic provides an easy way to integrate MPI blocking calls in an application written with StarPU, it is not sufficient for asynchronous MPI transfers. We therefore extended the data management library with a *non-blocking acquire* method. Instead of blocking while the piece of data is unavailable, this function immediately returns, and a user-provided callback is executed when it becomes available. Conceptually, with such asynchronous user interactions, non-blocking MPI transfers can be handled similarly to asynchronous tasks. Adapting an application described as a DAG of asynchronous StarPU tasks for a cluster therefore becomes natural.

To facilitate the integration of MPI codes, we have developed a small library that implements a MPI-like semantic summarized by Table II. This for instance permits to directly send or to receive a piece of data that is described as a StarPU data descriptor (handle) to or from a MPI process running another instance of StarPU. It is also possible to directly transfer multiple handles at the same time with the functions of type *array*. Moreover, the MPI semantic implies that any non-blocking transfer should be completed with a call to `MPI_Test` or `MPI_Wait`. Such synchronization points hardly fit into an asynchronous task-based approach, so that we extended our MPI-like semantic with **detached** functions which need not be completed by an explicit call to test or wait from the application, but will still release tasks depending on the data.

Implementing functions such as `starpu_mpi_isend` is simple. Given the data handle, which contains a full description of the piece of data, it is possible to automatically create a MPI datatype. An asynchronous data request is posted to StarPU so that when the data handle is available (in read-only mode), the `MPI_Isend` function gets called. This call is performed in a dedicated thread to prevent thread-

```
__global__ void incrementer_cuda_kernel(unsigned *token)
{
  (*token)++;
}

void increment_cuda(void *descr[], void *cl_arg)
{
  unsigned *tokenptr = STARPU_VARIABLE_GET_PTR(descr[0]);
  incrementer_cuda_kernel<<<1,1>>>(tokenptr);
}

void increment_cpu(void *descr[], void *cl_arg)
{
  unsigned *tokenptr = STARPU_VARIABLE_GET_PTR(descr[0]);
  (*tokenptr)++;
}

starpu_codelet increment_codelet = {
  .where = STARPU_CPU|STARPU_CUDA,
  .cuda_func = increment_cuda,
  .cpu_func = increment_cpu,
  .nbuffers = 1
};
```

```
unsigned token = 0;
starpu_data_handle token_handle;
starpu_variable_data_register(&token_handle,
                  0, &token, sizeof(token));

for (unsigned loop = 0; loop < NLOOPS; loop++)
{
  if ((loop > 0) || (rank > 0))
    starpu_mpi_recv_detached(token_handle,
              prev_rank, TAG, MPI_COMM_WORLD, NULL, NULL);

  struct starpu_task *task = starpu_task_create();
    task->cl = &increment_codelet;
    task->buffers[0].handle = token_handle;
    task->buffers[0].mode = STARPU_RW;
  starpu_task_submit(task);

  if ((loop < NLOOPS) && (rank < size))
    starpu_mpi_send_detached(token_handle,
          next_rank, TAG, MPI_COMM_WORLD, NULL, NULL);
}

starpu_task_wait_for_all();
```

Figure 2. MPI Ring

safety issues. This thread also ensures progression by calling `MPI_Test` until all pending requests are completed.

B. A complete example with MPI

Figure 2 shows the code of an MPI ring implemented with StarPU. After being registered to StarPU on the different MPI nodes, the piece of data is transmitted from one node to the other following a logical ring, while being incremented at each step by using a StarPU task. This task can either be executed on a CPU or a CUDA device, and its implementations merely consists in grabbing a pointer to the local copy of the data and increment it. A barrier is put at the end of the program to wait for the termination of all tasks and data transfers which were all submitted asynchronously by the loop[2]. Even if this is a very simple example, the mechanisms involved are suitable for more realistic problems, an LU decomposition for instance, as evaluated in the next section.

[2]Dependencies are automatically inserted by StarPU which we modified to be able enforce sequential data consistency.

V. Evaluation

To validate our approach, we experimented the improvements of StarPU with two typical numerical applications, on a single multi-GPU machine and on a cluster of multi-GPU machines. A 3D stencil kernel is used to stress data transfer efficiency, and the LU decomposition is used to stress the load distribution of the scheduler for an unbalanced problem.

To the best of our knowledge, there is currently no generic system equivalent to StarPU, which permits to exploit hybrid platforms with an arbitrary number of CPUs and accelerators, so that we only present measurements performed with StarPU. However, in the case of a LU decomposition running on a single accelerator, we have shown [3] that we obtain results comparable to those of the MAGMA library for instance.

A. Experimental testbed

We used two platforms to evaluate our approach in different environments. Both platforms are running Linux and use CUDA 3.0 BETA.

HANNIBAL is composed of **two** X5550 (NEHALEM) hyper-threaded **quad-core** processors running at 2.67 GHz with 48 GB of memory divided in two NUMA nodes. It is equipped with **3 NVIDIA QUADRO FX5800** with 4 GB of memory.

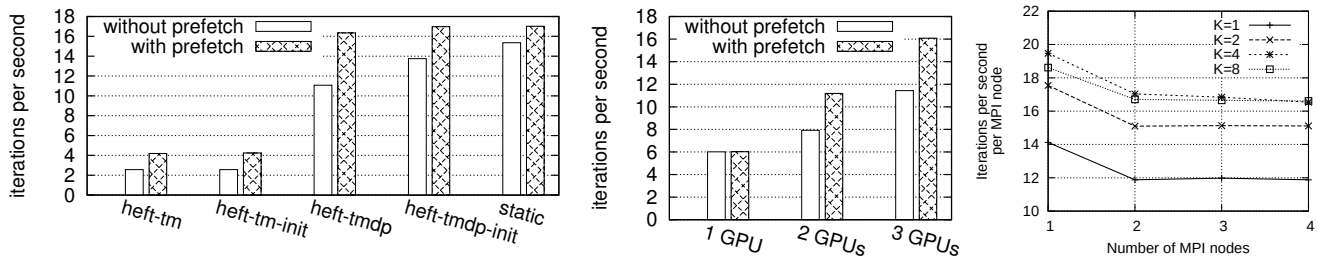
The **AC cluster** is composed of 32 nodes which are all equipped with **two dual-core** 2.4 GHz OPTERONS with 8 GB of memory divided in two NUMA nodes and an NVIDIA TESLA S1070 unit (**four GT200 GPUs** with 4 GB of memory per GPU).

B. Improvements on one node

We first experiment with various scheduling policies on a single node, HANNIBAL, to evaluate the benefits of taking data transfer time into account and automatically prefetching data.

1) Stencil: A stencil application puts a lot of pressure on data management because it is basically a BLAS1 operation, that *a priori* needs intermediate results transfer between Processing Units for each domain iteration. To get good performance, it is thus essential to properly overlap communication with computation and avoid the former as much as possible. This is also a good stress-test for the dynamic schedulers of StarPU since just statically binding all computations should *a priori* give the best performance. We implemented a simple 3D 256x256x4096 stencil split into 64 blocks along the z axis.

The left part of Figure 3(a) shows the performance obtained with the 3 GPUs of HANNIBAL using various schedulers. The *static* case uses a trivial static allocation



(a) Impact of scheduling on one node.

(b) Scalability within one node

(c) Throughput with MPI

Figure 3. Performance of a Stencil kernel over multiple GPUs and over MPI.

and thus gets the best performance of course. The *heft-tm* case does not use Data transfer Penalty and thus gets the worst performance, because it basically schedules all tasks quite randomly. Binding the tasks like in the *static* case for the first stencil iteration but not using Data transfer Penalty, the *heft-tm-init* case, does not bring any benefit. The *heft-tmdp* case does use Data transfer Penalties, without any particular initial placement. the achieved performance is already very close to the *static* case. We observed that the penalties actually tend to guide the scheduler into assigning adjacent tasks to the same GPU and keeping that assignment quite stable over iterations. This means that StarPU permits to just submit tasks without having to care how they could ideally be distributed, since the scheduler can dynamically find a distribution which is already very good. In the *heft-tmdp-init* case, the initial placement permits to achieve the same performance as the *static* case, thanks to data penalties guiding the scheduler into keeping that initial placement most of the time. We additionally observed that if for some external reason a task lags more than expected, the scheduler dynamically shifts the placement a bit to compensate the lag, which is actually a benefit over static allocation. Similarly, when exploiting a heterogeneous set of accelerators, StarPU automatically finds a good distribution of tasks according to the respective computation power, as explained in a previous paper [3].

The right part of figure 3(a) shows the scalability of the performance obtained by the completely dynamic *heft-tmdp* scheduler: it scales quite linearly.

It can also be noticed that the prefetch heuristic does provide a fairly good improvement, except of course when using only a single GPU since in that case data just always remains automatically inside that GPU.

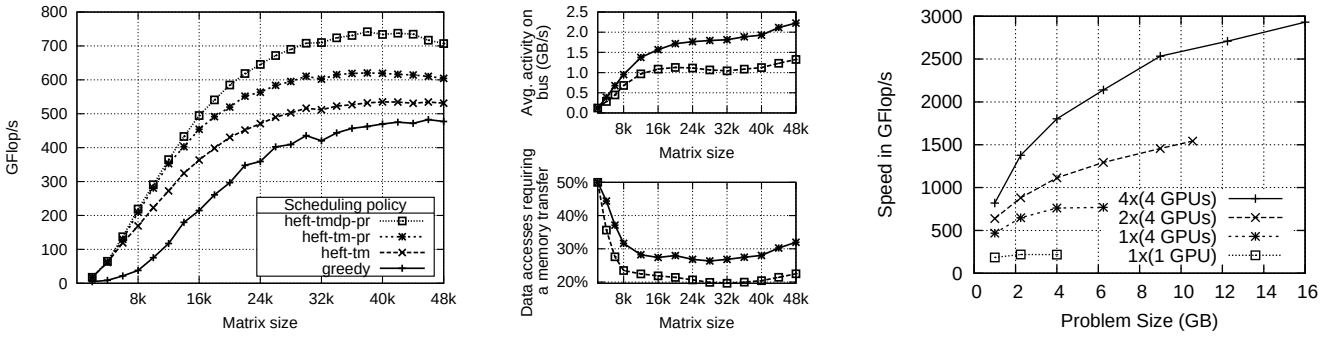
2) LU decomposition: Contrary to the stencil benchmark which is a steady state problem, for an LU decomposition the scheduler needs to maintain a sufficient amount of parallelism during the entire execution, which is therefore a good test case for load balancing. It also illustrates the use of hybrid platforms because not only GPUs but also CPUs are processing tasks.

With multiple accelerators, memory buses become a major bottleneck so that taking data transfers into account while scheduling tasks is critical. The *heft-tm-pr* policy attempts to mask the cost of memory transfers by overlapping them with computations thanks to the prefetch mechanism. Figure 4(a) shows a typical 20 % speed improvement over the *heft-tm* strategy which does not prefetch data. However, data prefetching does not reduce the total amount of memory transfers. The average activity on the bus is shown at the top of Figure 4(b): the *heft-tmda-pr* policy, by penalizing remote data accesses, and therefore favoring data locality, does however have a direct impact on the total amount of data transfers which drops almost by half. Likewise, the bottom of Figure 4(b) shows that the processing units tend to work more locally. In Figure 4(a), this translates into another 15 % reduction of the execution time.

C. Scalability over MPI

Thanks to the features detailed in section IV, we have integrated multi-GPU and MPI support in the Stencil and LU decomposition applications and experimented them with the AC cluster of multi-GPU machines. The distribution of data between machines of the cluster is static but tasks are dynamically scheduled within machines between their 4 GPUs. The transmission of intermediate results over MPI is achieved dynamically by StarPU as explained in section IV.

Stencil over MPI: Figure 3(c) shows how the stencil application scales over 4 machines, using the *heft-tmdp-pr* scheduling policy. K is the size of the overlapping border that is replicated between domain blocks. It needs to be big enough to facilitate overlapping communication with computation, without incurring too much duplicate computation due to the border. One can note that performance drops quite a bit between 1 MPI node and 2 nodes, due to the limitation of the network bandwidth, but using more nodes does not



(a) Impact of scheduling on Speed.

(b) Activity on the bus

(c) Speed over MPI

Figure 4. LU decomposition over multiple GPUs and over MPI.

really reduce the performance[3]. This shows that the StarPU execution runtime does not seem to have bad effects on the efficiency of the MPI library.

LU decomposition over MPI: Figure 4(c) shows the performance of the LU decomposition over up to 4 machines (*i.e.* 16 GPUs). We use a 2D-cyclic data layout to distribute the blocks between the different nodes. The LU decomposition already lacks parallelism, using more nodes implies that each GPU is given even less work to process. The scalability of our algorithm is therefore rather limited, but using multiple machines permits to solve larger problems. It however should be noted that we used a straightforward blocked implementation, and that improving scalability algorithm would require work at the algorithmic level which is out of the scope of this paper.

VI. Related work

Numerous projects intend to simplify the problem of code generation and code offloading onto accelerators, but many of them are centered on a single accelerator and offer little or no support for data management [5]. In the case of hybrid computing (*e.g.* using accelerators in addition to CPUs) or with multi-accelerators machines, it is usually not sufficient to simply load input data on the device during initialization and to get the result back at the end. A few libraries however consider the problem of data management with more care.

Software Distributed Shared Memory (SDSM) such as COMIC [11] and GMAC [7] respectively permit to access data transparently on Cell processors and on NVIDIA GPUs. Those SDSM boost productivity, but their efficiency highly depends on the workload. On the other hand, StarPU offers a high-level explicit data management which avoids the usual drawbacks of a DSM (*e.g.* false sharing). Its expressiveness enables optimizations such as reliable data prefetching mechanisms. Both approaches are complementary because real applications often consist of both legacy code which is hard to modify, as well as some performance-critical kernels which have to be rewritten for those accelerators. Data management is also made explicit in StarSs by the means of code annotations, and both its Cell and GPU [4] implementations take advantage of data caching as well as asynchronous data transfers.StarPU offers more precise hints to the scheduler, as it not only knows if some data is available, but it can also predict how expensive it is to move it, and take this into account while scheduling. Recent extensions permit to encapsulate MPI transfers in a SMPSs task, but these *taskified* transfers are not really asynchronous because they must be finished by the end of the task [13].

Accelerating applications over clusters is a natural concern for the HPC ecosystem. Rather than adopting a hybrid paradigm, message passing is often promoted as the single programming paradigm that should be used to cope with multiple accelerators. Various projects therefore provide libraries to have an MPI-like semantic either on Cell processors [14] or with GPUs [10]. Other models such as S_GPU [8] or CUDA wrapper [9] provide low level mechanisms to share GPUs between multiple processes on a machine. StarPU can use such virtualized processing resources when there are multiple processes on the same machine. On the other hand, higher-level approaches such as Sequoia directly consider a model which maps onto clusters of machines equipped with accelerators by focusing on data management [6]. Sequoia however maps data, and therefore computation, in a static fashion which could benefit from the flexibility of our dynamic scheduling.

VII. Conclusion and future work

We have shown how to support multiple GPUs in an efficient way by using asynchronous data management to perform data prefetch and overlap communication with computation. The design is robust enough to cope with

[3]Thanks to the 1D distribution, communication happens only between consecutive nodes, and the network switch is not (yet) saturated with the 4 available nodes.

e.g. limitations like lack of GPU-GPU transfer support in CUDA, but should still be able to express direct GPU-GPU transfers or even GPU-NIC transfers. Experimental results have shown that in addition to prefetch and overlap, performance can be dramatically improved by providing the scheduling policies with information to enhance locality by scheduling tasks according to data location, which reduces the required transfers. Last but not least, the asynchronous data management model permits to pave the way for hybrid models with the example of a library on top of StarPU to deal with MPI.

We plan to step further by extending StarPU's virtual shared memory over MPI in a way similar to a Distributed Shared Memory system (DSM), but at the StarPU data granularity level, to avoid most usual performance issues of DSMs. More generally we intend to provide tools to facilitate the generation of "data-flow algorithms" on top of StarPU and MPI. Combining this with GMAC's DSM would be interesting to make it easier to port existing applications over StarPU. Not only combining with MPI but also with CPU runtime environments or kernels such as OpenMP or MKL would also permit StarPU to take benefit from existing efficient parallel kernels. On another plan, we also consider using StarPU as a back-end for higher level libraries like MAGMA [12] or FFT, compilers (StarSs), or programming models, to handle scheduling and let them focus on their own algorithmic issues. Eventually, we plan on continuing the work in the scheduling area by extending our current simple greedy algorithms into heuristics which actually look at the task dependencies, to *e.g.* automatically detect priorities of tasks and data transfers according to the underlying graph.

ACKNOWLEDGMENTS

This work was partially supported by the NCSA under INRIA-UIUC Joint Laboratory for Petascale Computing project and utilized the AC cluster and by the ANR through the COSINUS (PROHMPT ANR-08-COSI-013 project). The research leading to these results has received funding from the European Union 7th Framework Programme (FP7/2007-2013) under grant agreement #24848 (PEPPHER Project, www.peppher.eu). We thank NVIDIA and its Professor Partnership Program for their hardware donations.

REFERENCES

[1] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *Proceedings of the Euro-Par Workshops, HPPC*, Delft, The Netherlands, August 2009.

[2] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report 7240, INRIA, March 2010.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010. Accepted for publication.

[4] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par*, pages 851–862, 2009.

[5] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment, 2007.

[6] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[7] Isaac Gelado, Javier Cabezas, John E. Stone, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *ASPLOS'10*, Pittsburgh, PA, USA, March 2010.

[8] Luigi Genovese, Matthieu Ospici, Thierry Deutsch, Jean-Francois Méhaut, Alexey Neelov, and Stefan Goedecker. Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *J Chem Phys*, 131(3):034103, 2009.

[9] Volodymyr V. Kindratenko, Jeremy Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, and Wen mei W. Hwu. GPU clusters for high-performance computing. In *CLUSTER*, pages 1–8, 2009.

[10] Orion Sky Lawlor. Message Passing for GPGPU Clusters: cudaMPI. In *IEEE Cluster PPAC Workshop*, 2009.

[11] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. COMIC: a coherent shared memory interface for Cell BE. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 303–314, New York, NY, USA, 2008. ACM.

[12] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators. Technical Report 223, LAPACK Working Note, November 2009.

[13] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 5–16, New York, NY, USA, 2010. ACM.

[14] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI Microtask for programming the Cell Broadband Engine Processor. *IBM Syst. J.*, 45(1), 2006.

[15] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.