Parallel Computing 44 (2015) 37–52

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures

João V.F. Lima [a,*], Thierry Gautier [d,b,c], Vincent Danjean [b,c,d], Bruno Raffin [d,b,c], Nicolas Maillard [a]

[a] Inst. of Informatics, UFRGS, CP 15064, 91501-970 Porto Alegre, RS, Brazil
[b] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[c] CNRS, LIG, F-38000 Grenoble, France
[d] Inria, France

ARTICLE INFO

Article history:
Received 5 April 2014
Received in revised form 28 January 2015
Accepted 1 March 2015
Available online 6 March 2015

Keywords:
Parallel programming
Accelerators
Task parallelism
Data-flow dependencies
Work stealing

ABSTRACT

In this paper, we present a comparison of scheduling strategies for heterogeneous multi-CPU and multi-GPU architectures. We designed and evaluated four scheduling strategies on top of XKaapi runtime: work stealing, data-aware work stealing, locality-aware work stealing, and Heterogeneous Earliest-Finish-Time (HEFT). On a heterogeneous architecture with 12 CPUs and 8 GPUs, we analysed our scheduling strategies with four benchmarks: a BLAS-1 AXPY vector operation, a Jacobi 2D iterative computation, and two linear algebra algorithms Cholesky and LU. We conclude that the use of work stealing may be efficient if task annotations are given along with a data locality strategy. Furthermore, our experimental results suggests that HEFT scheduling performs better on applications with very regular computations and low data locality.

1. Introduction

With the recent evolution of processor design, future generations of processors will contain hundreds of cores. To increase the performance per watt ratio, the cores will be non-symmetric with few highly powerful cores and numerous, but simpler, cores. The success of these machines will rely on the ability to schedule the workload at runtime, even for small problem instances.

One of the main challenges is to design a scheduling strategy that may be able to exploit all potential parallelism on heterogeneous architectures composed of multi-CPUs and multi-GPUs. Previous works demonstrate the efficiency of strategies such as static distribution [1–4], centralized list scheduling with data locality [5], cost models [6,7] based on Earliest-Finish-Time scheduling [8], and dynamic for a specific application domain [9,10].

In our previous works, we state that the classic work stealing is cache-unfriendly and does not consider data locality [11,12]. We described a locality-aware work stealing with annotations that improves significantly the performance of compute-bound linear algebra problems [11]. However, it does not consider the processing power of available resources. Few studies have compared performance of different scheduling strategies for heterogeneous multi-CPU and multi-GPU platforms.

* Corresponding author.
E-mail addresses: jvlima@inf.ufsm.br (J.V.F. Lima), thierry.gautier@inrialpes.fr (T. Gautier), vincent.danjean@imag.fr (V. Danjean), bruno.raffin@inria.fr (B. Raffin), nicolas@inf.ufrgs.br (N. Maillard).

http://dx.doi.org/10.1016/j.parco.2015.03.001
0167-8191/© 2015 Elsevier B.V. All rights reserved.

The purpose of this paper is to compare scheduling strategies based on dynamic scheduling and cost models for data-flow task programming on heterogeneous architectures. We designed and evaluated four scheduling strategies on top of XKaapi runtime: work stealing [12,13], data-aware work stealing [5,12], locality-aware work stealing [11,14], and Heterogeneous Earliest-Finish-Time (HEFT) [6,8].

We analysed our scheduling strategies with four benchmarks: a BLAS-1 AXPY vector operation, a Jacobi 2D iterative computation, and two linear algebra algorithms (Cholesky and LU). We conclude that the use of work stealing may be efficient if task annotations are given (such as `SetArch`) along with a data locality strategy. Experiments with LU showed that our locality-aware work stealing improved performance over HEFT by 13.24% and reduced data footprint by 17.29% on 4CPUs-8GPUs. Furthermore, our experimental results suggest that HEFT performs better on applications with very regular computations and low data locality. Jacobi 2D results showed that HEFT reduced iteration time by 53.19% and data footprint by 73.11% over locality-aware work stealing on 4 CPUs-8 GPUs.

The remainder of the manuscript is organized as follows. Section 2 presents the related work on runtime systems and scheduling algorithms for heterogeneous architectures. Section 3 gives an overview of XKaapi runtime and extensions from our previous works [11,12] for multi-GPU systems. Section 4 details the contribution of this paper on scheduling strategies for multi-CPU and multi-GPU architectures. Our experimental results are presented in Section 5. Finally, Section 6 and Section 7, respectively, present the discussion and conclude the paper.

2. Related work

In this section, we present runtime systems and scheduling algorithms for heterogeneous systems.

Charm++ has two extensions to support GPUs. Charm++ GPU Manager [15] allowed the overlap of chare objects (or computations) with transfers to GPUs. G-Charm [16] schedules at runtime chare objects between CPUs and GPUs based on the current loads and the estimated execution time provided by previous executions.

KAAPI [17] was specialized for multi-CPU and multi-GPU systems [10]. Each CPU or GPU implementation of a given task was encapsulated in a functor object, which provided a clear separation between a task definition and its various implementations, *i.e.*, one to each architecture.

StarPU is a runtime system providing a data management facility and an unified execution model over heterogeneous architectures including GPUs and Cell BE processors [6,7]. Its programming model relies on explicit parallelism by tasks with data dependencies and a memory layer to abstract transfers among disjoint address spaces.

StarSs proposes a programming model to exploit task-level parallelism by OpenMP-like pragmas [18] and a runtime system to schedule tasks while preserving dependencies. OmpSs [5] is a continuation of StarSs and extends SMPSs [19] and GPUSs [18] by providing simpler code annotations with the capacity to have recursive tasks. Recent versions of OmpSs include specific multi-versions of the same task [20] and regions of strided and/or overlapped data [21].

In the context of scheduling, many works in the literature propose strategies involving GPUs. There are related works in scheduling restricted to GPUs [22] and based on runtime systems [5,7,10,18,20]. Hermann et al. [10] propose a static and dynamic scheduling for iterative computations on multi-CPU and multi-GPU systems. The task graph partitioning (static phase) and work stealing (dynamic phase), respectively, enforce data locality and reduce the total number of steals.

Augonnet et al. [7] study strategies based on the Heterogeneous Earliest-Finish-Time (HEFT) scheduling algorithm [8]. They compare the impact of execution time (*heft-tm*) in addition to data transfers (*heft-tmdp*) and data prefetch (*heft-tmdp-pr*).

Ayguadé et al. [18] describe the GPUSs centralized list scheduling to minimize data transfers. Recent versions of GPUSs, called OmpSs [5], depict two scheduling policies: centralized with first-in first-out (FIFO) and locality-aware. The strategies proposed in GPUSs and OmpSs have the limitation of strict usage of GPUs, while CPUs are involved only in runtime routines. Planas et al. [20] study a scheduling strategy with task versioning similar to StarPU on multi-CPU and multi-GPU systems.

In our previous works, we state that the classic work stealing is cache-unfriendly and does not consider data locality [11,12]. We described a locality-aware work stealing with annotations that improves significantly the performance of compute-bound linear algebra problems [11]. However, it does not consider the processing power of available resources. In this sense, our strategy without any annotation is not aware of the efficiency in a certain architecture type. Since CPUs and GPUs are heterogeneous in computing power, a bad decision will affect tasks outside the critical path, which are candidates to be offloaded to accelerators, and may impact performance.

3. XKaapi runtime system overview

In this section, we introduce the XKaapi programming model and runtime system. We overview its software stack (3.1), programming model (3.2), scheduling algorithm (3.3), and multi-GPU runtime support (3.4). The aspects described in this section are previous contributions on XKaapi [10–12,17,23] and provide the basis for this paper's contribution on scheduling strategies for heterogeneous architectures.

3.1. XKaapi software stack

XKaapi[1] is a novel implementation of the KAAPI runtime developed by the INRIA MOAIS[2] team. The proposal of XKaapi is to target multicore architectures and various accelerators such as GPU and Intel Xeon Phi. More than a runtime, XKaapi is a fully featured software stack to program parallel architectures. The core stack is written in C and is designed using a bottom-up approach: each layer is kept as specialized as possible to fit a specific need.

Currently, the runtime stack includes: a runtime supporting multicores and multiprocessors; a set of ABIs (QUARK [24], OpenMP runtime libGOMP [23]); and a set of high level APIs such as C++ API **Kaapi++** and C API **Kaapic** [25]. Throughout this paper, we focus on the Kaapi++ API, which is a C++ interface derived from the Athapascan API [26] but with modifications at data sharing and task signature.

3.2. Programming model

The parallelism in XKaapi is explicit, while the detection of synchronizations is implicit: data dependencies between tasks and memory transfers are automatically managed by the runtime. A XKaapi program is composed of sequential code and some annotations or runtime calls to create tasks. Its task model enables non-blocking task creation: the caller creates tasks and continues the program execution [17].

XKaapi uses the concept of task multi-versioning [10] in order to have a clear separation between the task definition and its implementations. A task is a function call that returns no value except through the list of its effective parameters. Each task is associated with a signature that includes the number of effective parameters and their access modes. The implementation of this task signature for a given architecture corresponds to a template specialization of the `TaskBodyCPU` or (not exclusive) `TaskBodyGPU` classes. The types and the number of effective parameters must match the task's signature.

Fig. 1 shows an example of a task signature (`struct UserTask`) with CPU (`TaskBodyCPU`) and GPU (`TaskBodyGPU`) implementations conforming to its signature. We note that XKaapi expects at least the CPU implementation of a task signature. At task execution the runtime will execute the signature's implementation based on the current worker type, *i.e.*, CPU or GPU.

In XKaapi runtime, tasks share data if they have access to the same memory region. A memory region is defined as a set of addresses in the process virtual address space. The user is responsible for indicating the access mode of each task parameter. The main access modes are *read*, *write*, *reduction* or *exclusive* (read and write). Data dependencies between tasks unfold a Data-Flow Graph (DFG) [26].

3.3. Scheduling by work stealing

The XKaapi runtime implements work stealing inspired by Cilk [13]. The work stealing principle can be synthesized as follows. An idle thread, called a thief, initiates a steal request to a random selected victim. On reply, the thief receives a copy of one ready task, leaving the original task marked as stolen. Coherency between a thief and its victim is ensured by a Dijsktra-like protocol [13].

The runtime creates a system thread for each worker, which is in general a processor core. A thread creates tasks recursively and pushes them on its own work queue, which is represented as a stack. Once a task ends, the thread executes its children following a FIFO order by popping tasks from its own work queue. During task execution, if a thread finds a stolen task, it suspends its execution and switches to the work stealing scheduler that waits for dependencies to be met before resuming the task. Otherwise, tasks are performed in FIFO order without computation of data-flow dependencies since sequential execution is a valid order of execution [17,26].

The main difference between XKaapi and other software [5,6,24] is that XKaapi computes data-flow dependencies only when an idle thread searches for a ready task. This technique reduces the overhead of normal task execution in recursive programs where the number of steals depends on the length of the critical path, not on the number of tasks. This concept follows the work-first principle [13]: at the expense of a larger critical path, XKaapi moves the cost of computing ready tasks from the work performed by the victim during task's creation to the steal operations performed by thieves.

Thanks to our approach, the classical fine-grained recursive Fibonacci in a data-flow implementation shows an overhead T_1/T_{serial} of about 10 [23], which is of the same order as Cilk or TBB that do not handle data-flow dependencies.

3.4. Multi-GPU support

In this section we overview the XKaapi runtime extensions for multi-CPU and multi-GPU architectures. The extensions implement a programming model that offers asynchronous execution of GPU tasks and abstracts memory details [12]. Our current version has support for recent GPUs from NVIDIA by CUDA programming.

[1] http://kaapi.gforge.inria.fr
[2] http://moais.imag.fr

Kaapi++ task signature

```
struct UserTask: public ka::Task<1>::Signature<
  ka::RW<int>,     /* input and output parameter */ > {};
```

Kaapi++ CPU version

```
template<>
struct TaskBodyCPU<UserTask> {
  void operator() (
    ka::pointer_rw<int> data )
  { /* CPU implementation */ }
};
```

Kaapi++ GPU version

```
template<>
struct TaskBodyGPU<UserTask> {
  void operator() (
    ka::pointer_rw<int> data )
  { /* CPU code that launches
      GPU kernels */ }
};
```

Fig. 1. Example of multi-versioning with CPU and GPU implementations.

The main features on XKaapi are task annotations (Section 3.4.1), GPU workers and task execution (Section 3.4.2), concurrent GPU operations (Section 3.4.3), and memory management (Section 3.4.4).

3.4.1. Task annotations

XKaapi provides for programmers the concept of *task annotation* to pass scheduling hints through the Kaapi++ API. Its goal consists in "advising" the scheduler that a task should execute on a certain processor type (CPU or GPU), since CPUs and GPUs have different attributes such as processing power.

The main annotation for scheduling strategies on heterogeneous architectures is the `SetArchitecture` (or `SetArch`). The `SetArch` annotation restricts a task to a specific architecture type (CPU or GPU) and the runtime shall comply with this condition. Thus, a task with attribute CPU (`ka::ArchHost`) or GPU (`ka::ArchCUDA`) will not be executed by a worker of different type, allowing CPU-only and GPU-only tasks.

An example of task annotations is illustrated in Fig. 2. It creates two independent tasks in which the first executes on any CPU (line 2) and the second on any GPU (line 5). We note that only CPU tasks have support for recursive task creation.

3.4.2. GPU workers and task execution

Our runtime for multi-GPU systems dedicates a CPU core to manage a target GPU in the same way as other runtime tools such as StarPU [6] and OmpSs [5]. Fig. 3 illustrates the execution mechanism of XKaapi over three computing units (workers): two CPUs and one GPU. Two CPU cores become CPU workers and execute CPU computations (`TaskBodyCPU` code), while the third CPU core becomes a GPU worker and does not compute CPU tasks. This GPU worker is dedicated to find ready tasks and to send computations to the GPU. In addition, the core executes all host code to manage the GPU such as memory management and execution control.

3.4.3. Concurrent operations between CPU and GPU

XKaapi has an execution strategy for GPUs that avoids CUDA's implicit synchronizations and exploits concurrent memory transfers in two ways (host-to-device and device-to-host) along with kernel execution. It splits the execution of a GPU task in three basic operations: host-to-device input transfers (H2D), `TaskBodyGPU` execution (*i.e.* launch of CUDA kernels) (K), and device-to-host output transfers (D2H).

Fig. 4 illustrates the way XKaapi allows to pipeline concurrent operations on a GPU card with two way transfers (host-to-device and device-to-host). XKaapi uses a *sliding window* strategy to limit the number of enqueued operations at each stream in the GPU. In our example, the sliding window of Fig. 4 has two operations for each computing stream.

3.4.4. Memory management

XKaapi memory management enables the use of different address spaces offering an abstraction layer similar to a distributed shared-memory (DSM). It divides a heterogeneous system in memory nodes composed of main memory (or host memory) and GPU device memory of each card.

XKaapi manages GPU memory through a software cache, based on the Least Recently Used (LRU) replacement policy. Each GPU worker maintains a FIFO queue in order to keep track of allocated blocks. When a GPU task needs to access a memory block that is not present on the GPU, the runtime will allocate memory and insert it into the GPU queue. If the GPU memory is full, the software cache tries to evict the least recently used memory block from its own queue (LRU policy). If possible, unused blocks are reused without being freed.

Data consistency is guaranteed in a lazy fashion by a write-back policy. Data transfers to or from GPU occur only to ensure an up-to-date copy of data to a task, *i.e.*, when a task accesses data and when the data is in an invalid state in the target address space. This policy avoids unnecessary transfers and would reduce the stress of the memory bus, unlike write-through policy [5,6,27].

```
/* CPU-only task */
ka::Spawn<TaskOnlyCPU>( ka::SetArch(ka::ArchHost) )( /* */ );

/* GPU-only task */
ka::Spawn<TaskOnlyGPU>( ka::SetArch(ka::ArchCUDA) )( /* */ );
```

Fig. 2. Example of task annotations in the Kaapi++ API.

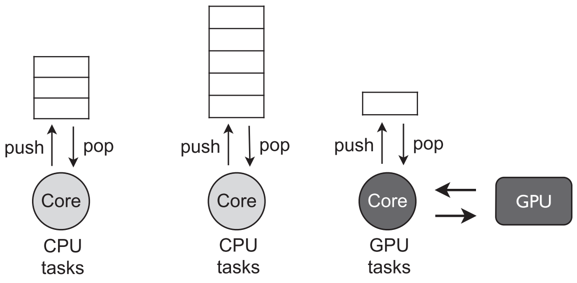


Fig. 3. Runtime structure of XKaapi with two CPU workers and one GPU worker.

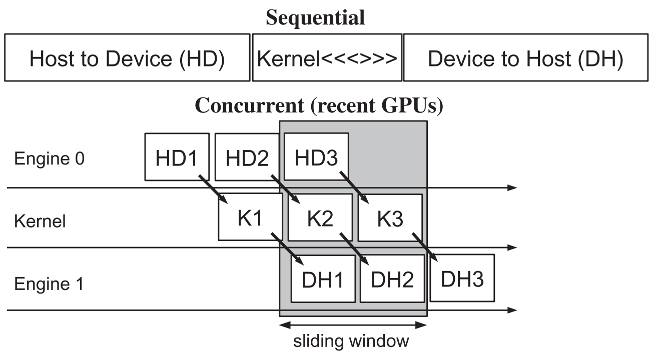


Fig. 4. Sequential and concurrent operations in a GPU card.

3.5. Summary

In our previous work, we demonstrated that our asynchronous approach of concurrent GPU operations achieved an almost ideal overlapping of data transfer and kernel execution with DGEMM algorithm for single-GPU [12]. Besides, we state that the classic work stealing is cache-unfriendly and does not consider data locality. In [11] we describe a locality-aware work stealing with annotations that improves significantly the performance of compute-bound linear algebra problems.

Nonetheless, our scheduling strategy based on work stealing lacks of more sophisticated decisions in order to consider processing power of available resources. We originally assumed that some tasks of a certain algorithm are more efficient on GPUs than CPUs. The obtained results with scheduling annotations seem consistent with our hypothesis. On the other hand, it is unlikely to achieve similar results for unknown tasks without empirical observations.

4. Scheduling strategies for XKaapi

In this section, we describe one of the main contributions of this paper: the design of four scheduling strategies on top of XKaapi runtime for heterogeneous multi-CPU and multi-GPU architectures.

Three scheduling strategies are based on dynamic scheduling and one is based on cost models. Our four different scheduling strategies for data-flow task programming applications are: work stealing [12,13], data-aware work stealing [5,12], locality-aware work stealing [11,14], and Heterogeneous Earliest-Finish-Time (HEFT) [6,8]. The strategies are designed on top of the XKaapi scheduling framework with performance models for task and transfer prediction.

4.1. Scheduling framework

In most runtime systems, the scheduler is designed as a plug-in that interfaces with an API able to manage a list of tasks. Most of list algorithms consider a centralized management of the list. However, the cost of concurrent list access induces synchronization overhead that can not be ignored. A suitable approach is to distribute the list among workers and each manages its own list of tasks. For instance, work stealing and work pushing are popular decentralized list scheduler algorithms.

We designed a framework in XKaapi in order to implement scheduling strategies based on decentralized list scheduling. Our interface is mainly inspired in work stealing and is composed of three operations: *pop*, *push* and *steal*.

Let us denote the operation's scope as *local*, whose manipulated list belongs to the current worker, and *remote* to a list not owned by the current worker. All three operations contain two parameters: a task list to manipulate (*local* or *remote*) and a task as input or output.

4.1.1. A general scheduling loop

Algorithm 1 illustrates a general scheduling loop of our scheduling framework. At each iteration, either the own queue is not empty and the worker uses it or the worker emits a steal request to a randomly selected worker in order to get a task to execute. In Fig. 5 we show a flowchart to represent the scheduling loop. Due to dependencies, once a worker executes a task, it calls the *activate* operation in order to activate its successors.

Algorithm 1. General scheduling loop of a worker w_j

```
while Execution not terminated do
    if Worker own queue is empty then
        $T \leftarrow$ steal from a random selected worker
        if $T \neq \emptyset$ then
            local_push $T$ into worker own queue
        end
    else
        $T \leftarrow$ pop from the worker own queue
        Execute $T$
        activate the task's successors of $T$
    end
end
```

Prologue and epilogue hooks give support to perform actions before and after task execution at line 9 of Algorithm 1. Augonnet et al. [6] employ hooks to deal with inaccuracy or missing performance prediction in the context of the HEFT strategy. Our scheduling strategies also apply these hooks to calibrate performance models (Section 4.2) and correct erroneous predictions due to unpredictable or unknown behavior, such as operating system jitter or I/O disturbance.

All of our scheduling strategies follow this algorithm. The workers terminate their execution when all tasks are completed.

4.1.2. Basic framework operations: pop, push, and steal

A framework interface for scheduling strategies is not a new concept in heterogeneous systems. Previous works described a minimal interface to design scheduling strategies and selection at runtime [5,6]. We designed a framework based on work stealing to design scheduling algorithms derived from list scheduling. Our framework is composed of three basic operations: *pop*, *push*, and *steal*.

A *pop* removes one task from the head of a task list for execution on the current worker. It is restricted to tasks capable of execution in the current worker, *i.e.*, tasks with an implementation to the current architecture type (CPU or GPU). Its scope is local to the current worker and may perform load balancing in centralized strategies. Thus, a pop can be issued to a remote task list with mutual exclusion.

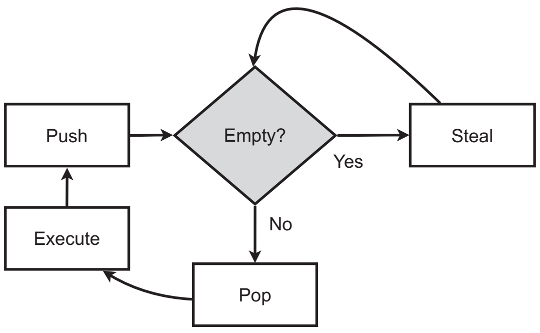


Fig. 5. General scheduling loop of the XKaapi scheduling framework.

The *push* method inserts one task at the head or the tail of a list. We designed two push versions depending on the list's scope: *local* and *remote*. A *local push* inserts a task at list's head and is commonly employed in the scope of a local task list to the current worker. Task insertion and removal from the same list position, in this case the list head, is a well-known technique to improve locality of tasks [28]. An exception of the *local push* scope is centralized strategies in which there is one task list shared by a number of workers. Whereas, a *remote push* inserts tasks onto the tail of non-local list worker. This operation provides support for a number of additional list scheduling based strategies such as heuristics [14] and work pushing from cost models. In addition to both versions of *push*, a *push_activated* inserts task successors into the list. It provides a way to apply cost model strategies over a set of ready tasks for execution.

A *steal* removes a task from the list's tail of a remote worker, or *victim*. Our *steal* operation is based on the classic work stealing algorithm [13]. An idle thread, called a thief, initiates a steal request to a random selected victim. To find a ready task, a thief thread calls the *steal* operation from the framework passing the victim's task list as parameter. On reply, the *steal* returns a reference (or memory pointer) of one ready task. In our framework, XKaapi *steal* does not require the removal of the stolen task from the victim's list. The runtime only expects the selection of a task, which will be marked as stolen by the XKaapi runtime. A scheduling strategy can disable *steal* by not specifying a steal function.

4.2. Performance model

Cost models depend on a certain knowledge of both application algorithm and the underlying architecture to predict performance at runtime. They are associated with scheduling strategies to predict task completion time such as HEFT. In order to predict performance, we designed a performance model for task execution time and communication, similar to StarPU [6]. We present here our transfer time estimation that is based on asymptotic bandwidth, and our task prediction that is based on history-based model.

XKaapi performance model for data transfer can predict communication transfer by asymptotic bandwidth, which is benchmarked through offline *sampling* of the PCIe latency and bandwidth. This sampling detects all available resources (CPUs or GPUs) and performs a series of *ping-pong* benchmarks by measuring both the bandwidth $B_{i\to j}$ and the latency $L_{i\to j}$ between each pair of resource (from a CPU to a GPU for instance). As all CPUs share the same memory, the estimated transfer time between two CPUs is always null.

Let us assume that $t_{i\to j}$ is the predicted transfer time from worker i to worker j given n bytes of data to transfer, $B_{i\to j}$ and $L_{i\to j}$ the stored bandwidth and latency, and $n_{accel.}$ the number of accelerators used by the program. We estimate the transfer time between workers i and j as:

$$t_{i\to j} = \frac{n}{B_{i\to j}} \times n_{accel.} + L_{i\to j} \tag{1}$$

The $n_{accel.}$ factor was introduced to model the share of PCIe links between all accelerators. It is a simplistic model but it seems enough with respect to our experiments.

The XKaapi performance model for task prediction relies on a history-based model for regular computations. We use three task fields to identify a performance entry: *task name*, *input size* (footprint), and *processor type*. The key to identify a unique hash entry of tasks is then composed of $(task_{name},\ task_{footprint},\ p_{type})$. At each executed task, the scheduling strategy can make use of *prologue* or *epilogue* hooks to update a task entry in order to be applied by the next execution. Still, we note that enabling task model sampling on GPUs may affect performance since the runtime has to disable asynchronous execution in order to know its actual completion time.

4.3. Scheduling strategies on top of XKaapi

4.3.1. XKaapi work stealing

In comparison with original multi-CPU work stealing, multi-GPU work stealing has little modifications. Since GPU operations are asynchronous, the GPU worker polls regularly the completion of previous asynchronous GPU operations [12]. For instance, the begin of a `TaskBodyGPU` code should wait for data transfers of input parameters.

The end of a `TaskBodyGPU` code does not guarantee the conclusion of its launched kernels. A task that completes its execution, when the launch of asynchronous kernels have completed, activates the successor tasks (according to the data-flow dependencies). Activation of successor tasks only occurs after the dispatch (or launch) of all kernels from a GPU task. These new ready tasks are pushed on the tasks' queue attached to the current GPU and they may be stolen by one CPU or another GPU.

The implementation of XKaapi work stealing is straightforward since all three operations of our framework are also basic work stealing operations (pop, push, and steal) [13].

4.3.2. Data-aware work stealing

The goal of our data-aware strategy is to reduce memory transfers between host and devices in order to execute a ready task. It is similar to the classic work stealing but considers meta-data information to reduce memory transfers. Bueno et al. [5] proposed a related scheduling strategy named *locality-aware*, but over a centralized scheduler.

In our strategy, for each ready task to be pushed, the algorithm first goes through every shared data argument of the task and searches for the workers where this data is in valid state. If the argument is valid, it keeps track of the amount of valid data (bytes) in this worker. The worker that owns the maximum number of data bytes in valid state for this task is then chosen as target to run the task. The ready task will be pushed onto the *mailbox* of the target worker, which would execute tasks from its mailbox before becoming a "thief". We note that a ready task pushed into the mailbox of a worker may be stolen if another worker becomes idle.

In Algorithm 2, we illustrate our designed algorithm. It accumulates for each shared argument the total number of valid bytes on each worker p_j. At line 8 it sorts workers by decreasing order of bytes and selects worker p_j that minimizes data transfer to execute task n.

Algorithm 2. Data-aware work stealing

```
   Input  : ready task $n$
   Output: target worker $p_i$
1  foreach shared argument $d_k$ of task $n$ do
2      foreach worker $p_j$ do
3          if $d_k$ is in valid state on the memory node of worker $p_j$ then
4              $total_j \leftarrow total_j +$ size in bytes of $d_k$
5          end
6      end
7  end
8  $p_i \leftarrow$ worker $j$ such that $total_j = \max_k \{total_k\}$
9  return $p_i$
```

The design of Algorithm 2 over our framework is also straightforward since it is similar to classic work stealing. The search of a target worker to reduce data transfer is performed at activation of the successor tasks (*push_activated*). Thus, a newly activated task is pushed to a target worker in order to reduce data transfers. In our previous work, we demonstrated that a data-aware strategy may reduce transfers between the host and GPUs compared to classic work stealing [11].

4.3.3. Locality-aware work stealing

The goal of our locality-aware strategy is to reduce invalidations of data replicas based on an owner-computes rule (OCR). This strategy is similar to locality-guided work stealing proposed by Acar et al. [14], but with an automatic scheme to (locally) reduce the number of cache invalidations instead of explicit code annotation. Guo et al. [29] also propose a similar locality strategy.

Our locality-aware strategy searches a shared data argument that has *write* or *exclusive* access mode. It pushes a ready task to the *mailbox* of a worker (CPU or GPU) that has a valid copy of this argument (*i.e.* output argument). If more than one worker is eligible, then the scheduler simply selects a worker at random. We note that a ready task pushed into the mailbox of a worker may be stolen if another worker becomes idle.

We show our algorithm in Algorithm 3. It goes through each shared argument of a task n and tests the access mode. If the argument has *write* access, it queries to the memory management which worker has a valid copy of d_i. The algorithm returns the local (current) worker p_{local} if no worker or no write argument are found.

Algorithm 3. Locality-aware work stealing

```
   Input  : ready task $n$
   Output: target worker $p_i$
1  foreach shared argument $d_i$ of task $n$ do
2      $a_i \leftarrow$ access mode for $d_i$
3      if $a_i$ has write access then
4          $p_i \leftarrow$ a worker with $d_i$ in valid state
5          return $p_i$
6      end
7  end
8  return $p_{local}$
```

Similarly, the design of Algorithm 3 over our framework is straightforward since it is similar to classic work stealing. The search of a target worker to reduce cache invalidations is performed at activation of the successor tasks (*push_activated*). In our previous work, we demonstrated that a locality-aware strategy may reduce data transfers and improve performance on multi-GPUs [11].

4.3.4. Heterogeneous Earliest-Finish-Time

The Heterogeneous Earliest-Finish-Time (HEFT) is a scheduling algorithm for a bounded number of heterogeneous processors [8]. It has two major phases: *task prioritizing* for computing the priorities of all tasks and a *worker selection* phase to select the "best" worker, which minimizes the task's finish time. The HEFT algorithm complexity is $O(v^2 \times p)$ for v tasks and p workers (CPUs plus GPUs).

Our HEFT scheduler is based on the algorithm presented in [6,8] and implements both phases (task prioritizing and worker selection) at activation of the task's successors (*push_activated* operation). The *task prioritizing* phase calculates for all ready tasks i a speedup $S_i = \frac{t_i^{CPU}}{t_i^{GPU}}$ relative to a GPU execution. Next, it sorts the list of ready tasks by S_i in decreasing order. In the *worker selection* phase, the algorithm selects tasks in the order of their speedup S_i and schedules each task on its "best" worker, which minimizes the task's finish time. Algorithm 4 describes the basic steps of the HEFT strategy over XKaapi.

Algorithm 4. Heterogeneous Earliest-Finish-Time (HEFT)

```
Input  : A finished task task
Output: A list of ready tasks tasklist
1 foreach task $i$ activated by finished task task do
2     $S_i \leftarrow \frac{t_i^{CPU}}{t_i^{GPU}}$
3 end
4 Sort activated tasks by decreasing speedup $S_i$
5 foreach task $i$ activated by task task do
6     Schedule task $i$ to minimize finish time on a worker $p_j$
7     Remote push of task $i$ into tasklist of worker $p_j$
8     Update dates of worker $p_j$
9 end
```

Although the original HEFT considers that the appropriate time slot on a worker p_j starts when all input data of a task T_i is available at p_j, we define the appropriate time slot when the worker p_j completes the execution of its last assigned task. Our HEFT algorithm incorporates data communications onto the search of an appropriate idle time slot by time prediction of data transfer for each worker (CPU or GPU) according to the state of task's arguments.

4.4. Summary

Previous works demonstrate the efficiency of strategies such as static distribution [1–4], centralized list scheduling with data locality [5], cost models [6,7] based on Earliest-Finish-Time scheduling [8], and dynamic for a specific application domain [9,10]. Nevertheless, few studies have reported on performance of different scheduling strategies for heterogeneous multi-CPU and multi-GPU platforms.

We described the design of four scheduling strategies on top of XKaapi runtime for heterogeneous multi-CPU and multi-GPU architectures. Three scheduling strategies are based on work stealing scheduler and one is based on Heterogeneous Earliest-Finish-Time (HEFT) cost model. These strategies were designed on top of the XKaapi scheduling framework with performance models for task and transfer prediction.

In our dynamic strategies, the main difference between locality-aware and data-aware relies on the heuristic to improve data locality. While data-aware considers the amount of valid data (*i.e.* containing its last version) per worker, our locality-aware only takes into account output data in valid state on a worker. We believe that data-aware strategy may benefit coarse-grained tasks with data intensive computations. On the other hand, locality-aware may favor fine-grained tasks with data arguments in read and write access (*i.e.* update mode) following our OCR-based rule.

In addition, we presented HEFT strategy that considers data transfer and computing power of resources to estimate execution time of tasks. Our hypothesis is that HEFT may favor coarse-grained tasks and very regular computations.

5. Experiments

The goal of our experiments is to evaluate the XKaapi scheduling strategies for heterogeneous architectures composed of multi-CPU and multi-GPU. Our objectives are:

1. evaluate the runtime overhead (T_1/T_{serial}) of the parallel algorithm with 1 CPU or 1 GPU (T_1) by the serial algorithm on CPU or GPU (T_{serial}), which is new compared to our previous runtime experiments [11,12];
2. compare work stealing and data-oriented strategies (data-aware and locality-aware), which schedule based on idle resources and data locality, against an algorithm based on cost models such as HEFT, which considers the processing power of available resources;
3. analyse the impact of different scheduling strategies on data footprint (total transfers) and cache hit ratio.

In our graphics we denote *work stealing* for default work stealing of XKaapi, *data-aware* for data-aware work stealing, *locality-aware* for locality-aware work stealing, and *heft + data* for HEFT scheduling with data transfer prediction. We use the suffix `SetArch` for algorithms using a task annotation to execute tasks only in GPUs or CPUs. In this paper, we only apply `SetArch` to restrict tasks on GPUs since our previous experiments showed its efficiency at highly parallel GPU tasks [11,12].

In each experiment, we show in the x-axis the number of resources as the number computing CPUs and GPUs for each execution. We employ this notation to clearly distinguish the number of computing CPUs and GPUs at runtime. Since XKaapi dedicates a CPU to manage a GPU, the number of computing CPUs is the total number of CPUs on the platform minus the number of selected GPUs (see Section 3.4 for details). This strategy aims to study the scalability of the runtime as long as we increase the number of computing GPUs.

Each result is a mean of 30 executions. The 95% confidence interval is represented on the graphs by a gray band around the mean values.

5.1. Platform and environment

All experiments have been conducted on a heterogeneous, multi-GPU system, named “Idgraf”. Idgraf is composed of two hexa-core Intel Xeon X5650 CPUs (12 CPU cores total) running at 2.66 GHz with 72 GB of memory. It is enhanced with 8 NVIDIA Tesla C2050 GPUs (Fermi architecture) of 448 GPU cores (scalar processors) running at 1.15 GHz each (2688 GPU cores total) with 3 GB GDDR5 per GPU (18 GB total). Fig. 6 illustrates the hardware topology of Idgraf.

The machine has 4 PCIe switches to support up to 8 GPUs. When 2 GPUs share a switch, their aggregated PCIe bandwidth is bounded to the one of a single PCIe 16×. Experiments using up to 4 GPUs always use 1 GPU per PCIe switch to avoid this bandwidth constraint. On the other hand, experiments using more than 4 GPUs have to share some pairs of GPUs through the PCIe switch.

We used as software environment GNU/Linux Debian *squeeze* ×86/64, the compiler GCC 4.4, CUDA 5.0, and the library ATLAS 3.9.39 (BLAS and LAPACK).

5.2. Benchmarks

We designed four benchmarks using XKaapi Kaapi++ interface. A BLAS-1 AXPY vector operation ($y \leftarrow \alpha * x + y$) in single precision (SAXPY) that aims to evaluate our contributions by a very regular, memory-bound, problem. A Jacobi 2D iterative computation in double precision to assess the capacity to scheduling data-flow tasks on stencil applications. Finally, two linear algebra algorithms Cholesky ($A = LL^T$) and LU ($A = LU$) in double precision to measure performance capacity of different scheduling strategies. Both benchmarks (LU and Cholesky) are parallel blocked algorithms [30] in which Cholesky is a right-looking implementation, and LU uses partial pivoting based on a modified version from StarPU [6].

Almost all tasks have a GPU version (`TaskBodyGPU`) except one in Cholesky and one in LU. These two tasks correspond to BLAS-2 dominated operations that may not be efficient on GPU. Cholesky does not have a GPU version for the diagonal panel factorization (task *POTRF*), and LU does not have a GPU version for row permutations.

5.3. Runtime overhead

In this experiment, we evaluate the overhead T_1/T_{serial} of the parallel algorithm T_1 on one CPU or GPU with XKaapi over the serial algorithm T_{serial} in order to study the XKaapi overhead on task creation. We measured T_1 as the execution time of

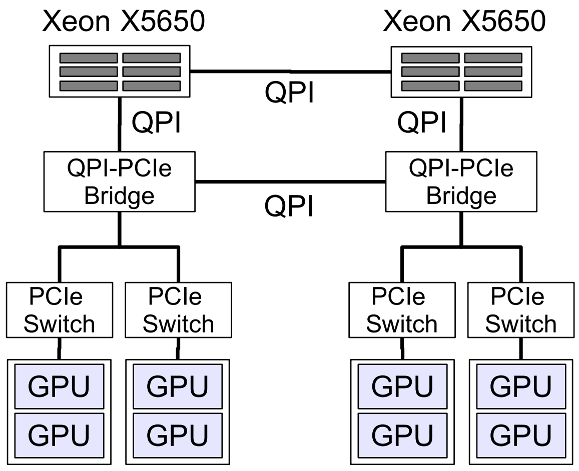
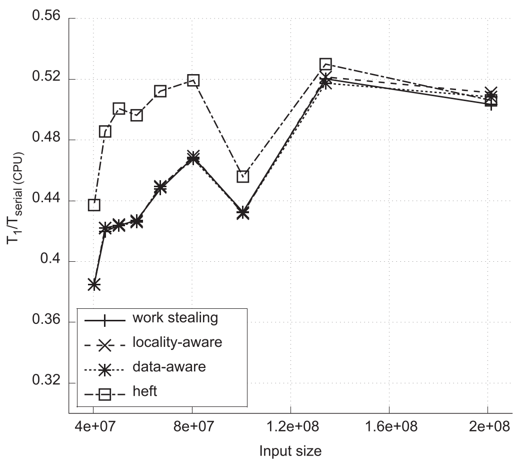
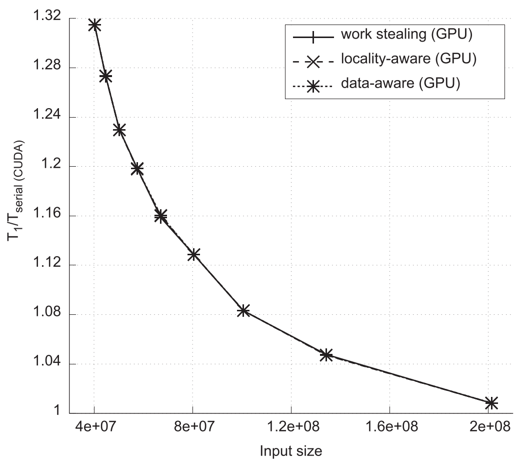


Fig. 6. Idgraf hardware topology with 12 CPUs and 8 Tesla C2050 GPUs.

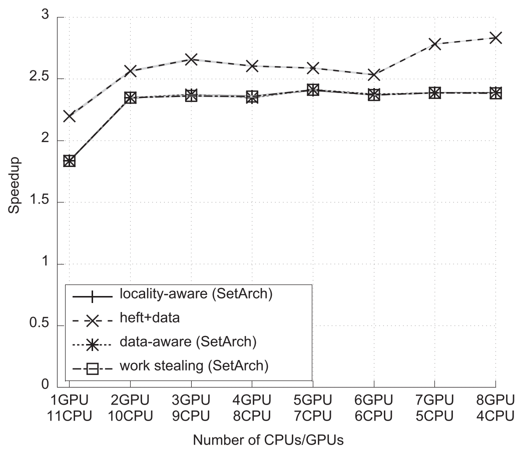


(a) T_1/T_{serial} on one CPU.

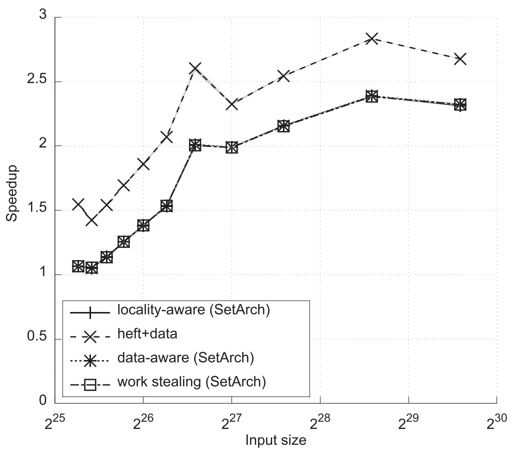


(b) T_1/T_{serial} on one GPU.

Fig. 7. SAXPY parallel overhead T_1/T_{serial} with XKaapi runtime.



(a) Input size 402653184.



(b) 4 CPUs 8 GPUs.

Fig. 8. Performance results for SAXPY in terms of speedup.

parallel SAXPY on 1CPU (only CPU tasks) or 1GPU (only GPU tasks) obtained with XKaapi. The execution time T_{serial} was obtained using the serial CPU version (CBLAS function `cblasSaxpy`) or the serial GPU version (CUBLAS function `cublasSaxpy`) including memory transfers, but without memory allocations.

Fig. 7 illustrates the experimental results. The T_1/T_{serial} overhead on one CPU was between 0.38 and 0.53 (Fig. 7(a)), which means that the parallel CPU version was faster than the serial CPU version. All three work stealing strategies had similar overhead, while HEFT showed an overhead greater than others by an insignificant difference. These results can be explained by the processor cache effect of the parallel algorithm, which is also called *superlinear speedup*.

The T_1/T_{serial} overhead over one GPU was greater than 1 for all input sizes (Fig. 7b), which means that the parallel GPU version was slower than the GPU serial version. The lowest input size had the maximum GPU overhead (1.32), and all three work stealing strategies showed similar GPU overhead. It seems that XKaapi offers low runtime overhead on multi-CPU and multi-GPU executions. We note that we were not able to measure HEFT overhead since it does not have support for `SetArch` annotation on GPUs.

5.4. Performance results

In order to evaluate the performance of our scheduling strategies, we performed two experiments for each benchmark: increasing the number of GPUs (up to 8 GPUs) and increasing the input size.

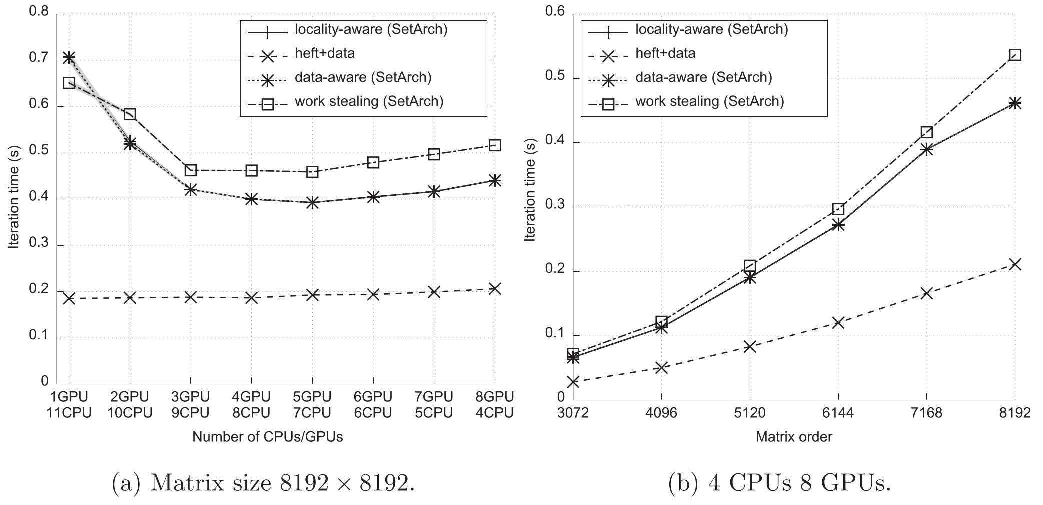


Fig. 9. Performance results for Jacobi 2D in terms of iteration time.

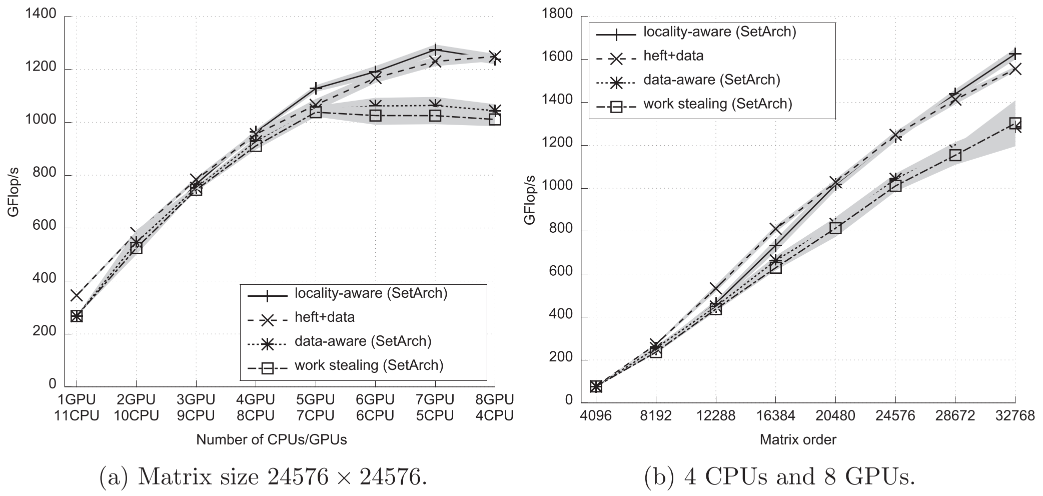


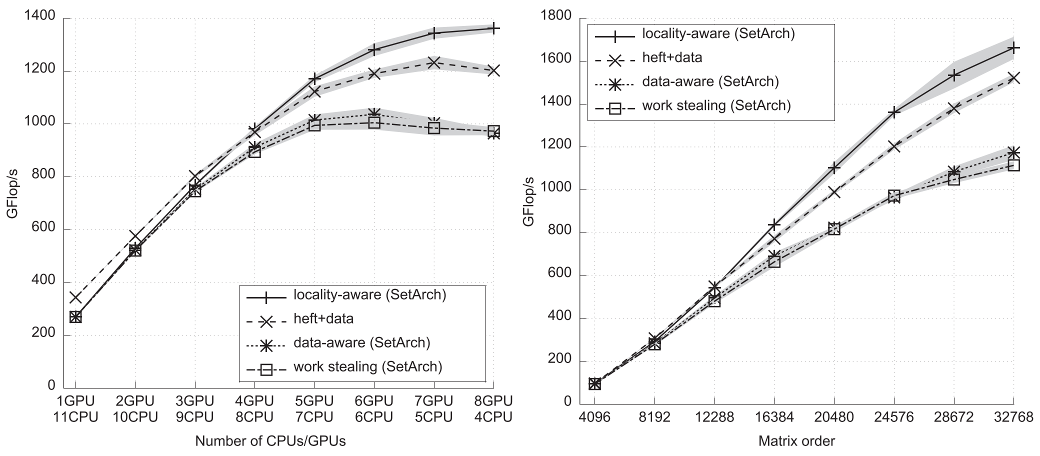
Fig. 10. Performance results for Cholesky in GFlop/s.

SAXPY – Fig. 8 shows the speedup T_{serial}/T_p over the serial version on CPU with SAXPY computation. On both experiments the maximum speedup was obtained with *heft + data* (about 2.8 of speedup). The other three strategies had similar performance. It seems that SAXPY benefited of the additional CPUs with *heft + data* and outperformed other strategies. Besides, these results suggest that *heft + data* is efficient on very regular computations such as SAXPY.

Jacobi 2D – Fig. 9 shows the performance results with Jacobi 2D in terms of mean iteration time, *i.e.*, total execution time (in seconds) divided by number of iterations. HEFT scheduling (*heft + data*) outperformed other strategies for almost all cases and was 113.63% better than *locality-aware* on 4 CPUs-8 GPUs (Fig. 9(a)). *locality-aware* and *data-aware* had similar results in both experiments and outperformed pure work stealing (*work stealing*). It appears that scheduling strategies based on processing power are more suitable to iterative computations than data locality algorithms. As predicted by our hypothesis, the *work stealing* had the lowest performance for almost all cases, except for 11 (Fig. 9(a)).

Cholesky – Fig. 10 shows the performance results with Cholesky factorization. The *locality-aware* strategy outperformed other strategies from 4 GPUs to 7 GPUs (Fig. 10(a)), while *heft + data* had better performance on 8 GPUs and matrices smaller than or equal to 24768 × 24768 (Fig. 10(b)). The peak performance was obtained by *locality-aware* with 1.27 TFlop/s on 5 CPUs-7 GPUs (Fig. 10a) and 1.62 TFlop/s at matrix size 32768 × 32768 (Fig. 10(b)). *locality-aware* was 1.93% and 4.51% better than *heft + data* at matrices 28672 × 28672 and 32768 × 32768 respectively (Fig. 10(b)). However, *heft + data* outperformed work stealing strategies from 1 GPU to 4 GPUs.

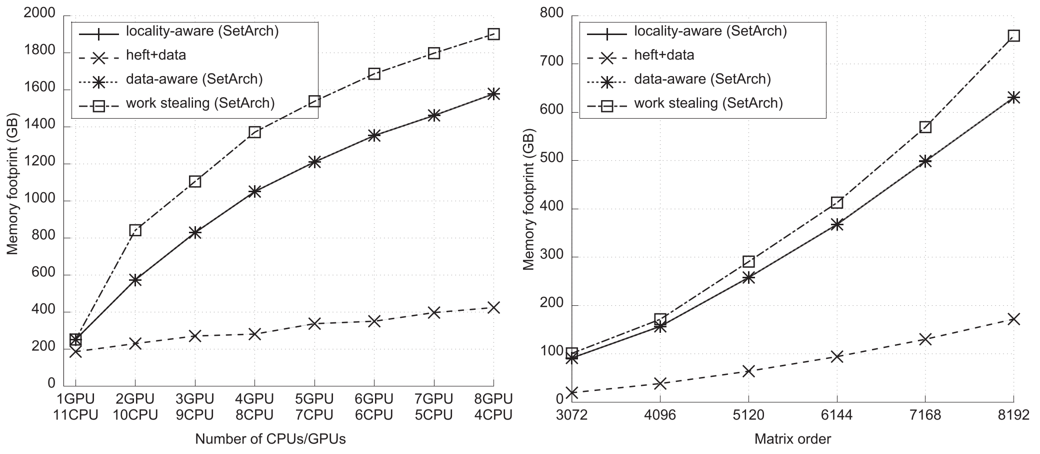
It appears that HEFT scheduling performs better by using additional CPUs in conjunction with GPUs. The results with *work stealing* are consistent with earlier findings suggesting that work stealing with locality improves performance significantly



(a) Matrix size 24576 × 24576.

(b) 4 CPUs 8 GPUs.

Fig. 11. Performance results for LU in GFlop/s.



(a) Matrix size 8192 × 8192.

(b) 4 CPUs 8 GPUs.

Fig. 12. Memory footprint metrics for Jacobi 2D in GB.

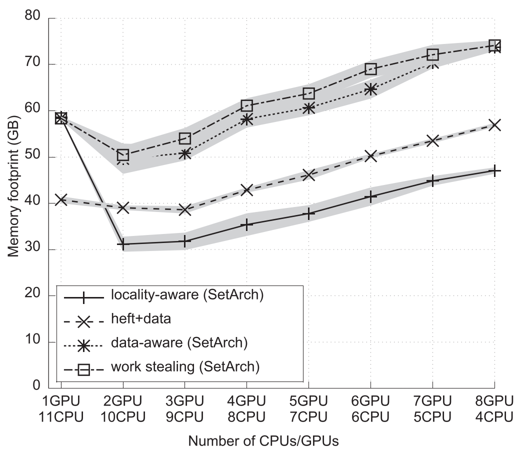
[11]. Besides, *data-aware* results where slightly better than *work stealing* for both experiments. As predicted by our hypothesis, the *work stealing* had the lowest performance due to its cache-unfriendly nature [14].

LU – Fig. 11 shows the performance results with LU factorization. The *locality-aware* strategy outperformed other strategies from 4 GPUs (Fig. 11(a)) and from matrices of size 16384 × 16384 (Fig. 11(b)). The peak performance was attained by *locality-aware* with 1.36 TFlop/s on 4 CPUs-8 GPUs (Fig. 11(a)) and 1.66 TFlop/s at matrix size 32768 × 32768 (Fig. 11(b)). The *heft + data* strategy performed better for some cases than *locality-aware* (about 27.34% better on 11 CPUs-1 GPUs) and outperformed other strategies (*work stealing* and *data-aware*) for all cases. In all cases (both experiments), *data-aware* outperformed work stealing (*work stealing*).

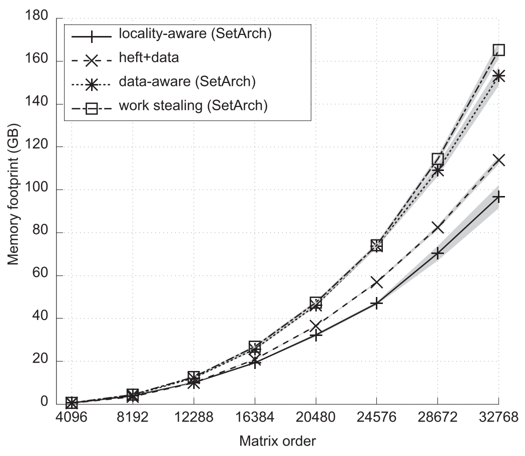
Compared to Cholesky's findings, *locality-aware* had better results than *heft + data* on LU. It seems that our locality-aware work stealing is efficient on irregular computations such as LU with pivoting. Besides, it appears that HEFT scheduling performs better using additional CPUs over executions dominated by CPUs and smaller matrices.

5.5. Software cache analysis

In this section we evaluate the impact of different scheduling strategies on the XKaapi software cache for benchmarks Jacobi 2D and LU. Trace metrics were collected at each experiment in order to analyse data footprint (all transfers between host and GPUs) and cache hit ratio.

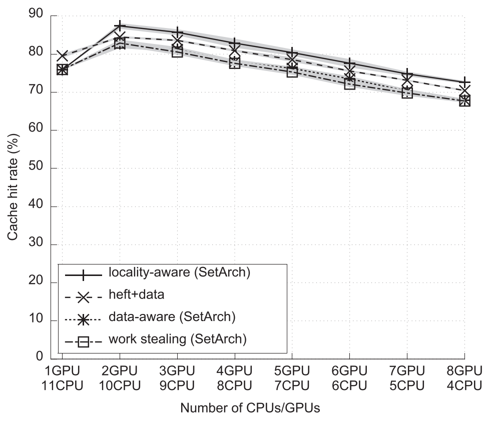


(a) Matrix size 24576 × 24576.

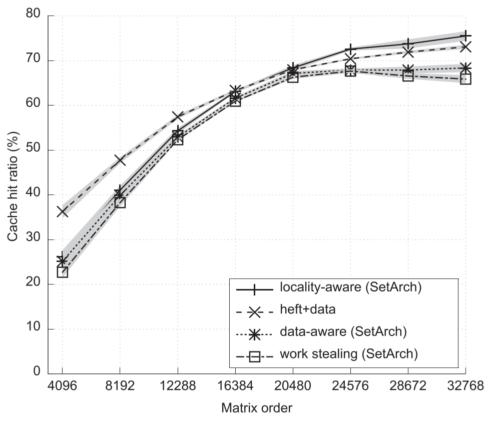


(b) 4 CPUs 8 GPUs.

Fig. 13. Memory footprint metrics for LU in GB.



(a) Matrix size 24576 × 24576.



(b) 4 CPUs 8 GPUs.

Fig. 14. Memory cache hit ratio for LU.

Fig. 12 reports data footprint of Jacobi 2D for all iterations from our performance experiments (Fig. 9). HEFT scheduling had significant lower footprint for all cases with 73.11% lower footprint than *locality-aware* on 4 CPUs-8 GPUs (Fig. 12(a)) and 72.72% lower footprint than *locality-aware* at matrix size 32768 × 32768 (Fig. 12). In a similar way to performance metrics, *data-aware* and *locality-aware* had near data footprint that was lower than work stealing (*work stealing*) for all cases. We do not show the Jacobi 2D hit ratio because its was near 96% at the end of all iterations.

Fig. 13 shows data footprint of LU factorization from our performance experiments (Fig. 11). The *locality-aware* strategy had the lowest data footprint for almost all cases with 47.07 GB on 4 CPUs-8 GPUs (Fig. 13) and 96.85 GB with matrix size 32768 × 32768 (Fig. 13(b)). In other words, data footprint with *locality-aware* was lower than *heft + data* by 17.29% (Fig. 13(a)) and by 14.96% (Fig. 13(b)). These results suggests that locality-aware work stealing reduces data footprint compared to HEFT cost model. We did not expect the high data footprint of *data-aware* that was slightly lower than pure work stealing footprint (*work stealing*).

Finally, Fig. 14 illustrates cache hit ratio from XKaapi with LU factorization. The *locality-aware* strategy attained the maximum hit ratio of 87.36% on 2CPUs-10GPUs (Fig. 14(a)) and 75.51% at matrix size 32768 × 32768 (Fig. 14(b)). It appears that locality-aware improves cache hit ratio by a small amount even at significant footprint reduction. As expected, pure work stealing (*work stealing*) had the lowest hit ratio for all cases due to its cache-unfriendly nature [14].

6. Discussion

We first measured the runtime overhead of XKaapi using the SAXPY operation for T_1 on one CPU and T_1 on one GPU. Our experimental results showed that XKaapi had low overhead on CPUs and GPUs, which included other low level operations such as memory consistency and allocation on GPUs. This is consistent with our earlier findings on XKaapi efficiency at concurrent GPU operations with low overhead [12].

The performance results of the four benchmarks had a performance behavior that can be divided in two groups. The first is the group that did not scale as expected by increasing the number of GPUs but scaled by increasing the input size. In this group, SAXPY (Fig. 8) and Jacobi 2D (Fig. 9) had this tendency. The second group scaled by increasing the number of GPUs and the input size. Both linear algebra algorithms, Cholesky (Fig. 10) and LU (Fig. 11), are in this group.

It seems that scheduling SAXPY tasks is mainly influenced by considering the computing power of the available resources. Data locality may not affect its performance since it has one input and output argument. Besides, there are no data dependencies in SAXPY due to its high data parallel pattern.

On the other hand, the Jacobi 2D results suggest that both data locality and computing power of resources are essential on task scheduling. Experimental results with Jacobi 2D showed that HEFT reduced iteration time by 53.19% and data footprint by 73.11% over locality-aware on 4CPUs-8 GPUs. Comparing the results over pure work stealing, locality-aware improved its performance by 16.98% and reduced data footprint by 14.70% on 4 CPUs-8 GPUs. These results can be explained considering the locality of each task border, which is one of main characteristics of stencil computations, and computing power of tasks in different resources.

The performance results with Cholesky and LU suggest that linear algebra algorithms benefit significantly of dynamic scheduling with data locality. In addition to the high computing cost of Cholesky and LU, they have many data dependencies that may benefit of data locality. For instance, on Cholesky over 4 CPUs-8 GPUs, locality-aware improved performance by 2.04% and reduced data footprint by 6.39% over HEFT. On LU at 4 CPUs-8 GPUs, locality-aware improved performance by 13.24% and reduced data footprint by 17.29% over HEFT. LU had better performance gain than Cholesky on locality-aware work stealing because it has more irregular computations such as pivoting.

Comparing the four different strategies, two of the previous benchmarks benefited of decisions based on the computing power of resources (SAXPY and Jacobi 2D), while the other two had better results using work stealing with data locality (Cholesky and LU). One possible explanation is that HEFT allows to reduce footprint and exploit the target architecture over very regular computations. Nevertheless, the knowledge of the underlying architecture comes at the cost of tuning and building a performance model at runtime or offline. We note that such a model can not be applied to irregular applications since their workload is unknown until execution.

The locality-aware work stealing with `SetArch` annotation for GPUs showed significant performance on Cholesky and LU. In our experiments, there was no need to tune our executions but annotate certain tasks highly efficient on GPUs. Our experimental results led us to infer that decreasing the invalidation of data replicas would improve performance and reduce data footprint.

A limitation of our locality-aware strategy relies on the lack of knowledge about the application algorithm. In this sense, the strategy without any annotation is not aware of the efficiency in a certain architecture type. Since CPUs and GPUs are heterogeneous in computing power, a bad decision will affect tasks outside the critical path, which are candidates to be offloaded to accelerators, and may impact performance.

An exception to our data locality hypothesis is the data-aware strategy that outperformed pure work stealing by a small factor. The results seem inconsistent with our hypothesis that data-aware would be more efficient than HEFT for Cholesky and LU as sustained by other works [5,21].

7. Conclusion

In this paper, we presented a comparison of different scheduling strategies based on cost models and dynamic scheduling by work stealing for heterogeneous multi-CPU and multi-GPU architectures. We designed and evaluated four scheduling strategies on top of XKaapi runtime: work stealing, data-aware work stealing, locality-aware work stealing, and Heterogeneous Earliest-Finish-Time (HEFT). We conducted experimental results with four benchmarks (SAXPY, Jacobi 2D, Cholesky, and LU) on a heterogeneous architecture composed of 8 GPUs and 12 CPUs.

We conclude that the use of work stealing may show efficient performance provided that task annotations are given (such as `SetArch`) along with a data locality strategy. Experiments with LU factorization showed that locality-aware work stealing improved performance over HEFT by 13.24% and reduced data footprint by 17.29% on 4 CPUs-8 GPUs. Furthermore, our experimental results suggests that HEFT scheduling performs better on applications with very regular computations and low data locality. Jacobi 2D results showed that HEFT reduced iteration time by 53.19% and data footprint by 73.11% over locality-aware on 4 CPUs-8 GPUs. Finally, on the context of runtime systems, we conclude that our scheduling framework can provide a basic support for different scheduling strategies.

Future works include more experimental evaluations on cost models with different performance schemes such as energy consumption, and dynamic scheduling strategies with a performance model built at runtime.

Acknowledgments

This work has been partially supported by CAPES/Brazil and CNPq/Brazil.

References

[1] J. Dongarra, J. Kurzak, P. Luszczek, S. Tomov, Dense linear algebra on accelerated multicore hardware, in: M.W. Berry, K.A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, F. Saied (Eds.), High-Performance Scientific Computing, Springer, London, 2012, pp. 123–146, http://dx.doi.org/10.1007/978-1-4471-2437-55.
[2] M. Horton, S. Tomov, J. Dongarra, A class of hybrid lapack algorithms for multicore and GPU architectures, in: Proc. of the 2011 SAAHPC, IEEE, Washington, DC, USA, 2011, pp. 150–158, http://dx.doi.org/10.1109/SAAHPC.2011.18.
[3] F. Song, J. Dongarra, A scalable framework for heterogeneous GPU-based clusters, in: Proc. of ACM SPAA, ACM, New York, NY, USA, 2012, pp. 91–100, http://dx.doi.org/10.1145/2312005.2312025.
[4] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (5–6) (2010) 232–240, http://dx.doi.org/10.1016/j.parco.2009.12.005.
[5] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguadé, J. Labarta, Productive Programming of GPU Clusters with OmpSs, in: Proc. of the 26th IEEE IPDPS, 2012, pp. 557–568.
[6] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput.: Pract. Exp. 23 (2) (2011) 187–198, http://dx.doi.org/10.1002/cpe.1631.
[7] C. Augonnet, J. Clet-Ortega, S. Thibault, R. Namyst, Data-aware task scheduling on multi-accelerator based platforms, in: Proc. of the 16th IEEE ICPADS, 2010, pp. 291–298, http://dx.doi.org/10.1109/ICPADS.2010.129.
[8] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274, http://dx.doi.org/10.1109/71.993206.
[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: a generic distributed DAG engine for high performance computing, Parallel Comput. 38 (1–2) (2012) 37–51, http://dx.doi.org/10.1016/j.parco.2011.10.003.
[10] E. Hermann, B. Raffin, F.c. Faure, T. Gautier, J. Allard, Multi-GPU and multi-CPU parallelization for interactive physics simulations, Proc. of the 2010 Euro-Par, vol. 6272, Springer, 2010, pp. 235–246, http://dx.doi.org/10.1007/978-3-642-15291-723.
[11] T. Gautier, J.V. Lima, N. Maillard, B. Raffin, XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures, in: Proc. of the 27th IEEE IPDPS, 2013, pp. 1299–1308, http://dx.doi.org/10.1109/IPDPS.2013.66.
[12] J.V.F. Lima, T. Gautier, N. Maillard, V. Danjean, Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs, in: Proc. of the 24th SBAC-PAD, IEEE, New York, USA, 2012, pp. 75–82.
[13] M. Frigo, C.E. Leiserson, K.H. Randall, The implementation of the cilk-5 multithreaded language, SIGPLAN Not. 33 (5) (1998) 212–223, http://dx.doi.org/10.1145/277652.277725.
[14] U.A. Acar, G.E. Blelloch, R.D. Blumofe, The data locality of work stealing, in: Proc. of the ACM SPAA, ACM, New York, NY, USA, 2000, pp. 1–12, http://dx.doi.org/10.1145/341800.341801.
[15] P. Jetley, L. Wesolowski, F. Gioachin, L.V. Kalé, T.R. Quinn, Scaling hierarchical N-body simulations on GPU clusters, in: Proc. of the 2010 ACM/IEEE SC, 2010, pp. 1–11, http://dx.doi.org/10.1109/SC.2010.49.
[16] R. Vasudevan, S.S. Vadhiyar, L.V. Kalé, G-charm: an adaptive runtime system for message-driven parallel applications on hybrid systems, in: Proc. of the 27th ACM ICS, ACM, New York, NY, USA, 2013, pp. 349–358, http://dx.doi.org/10.1145/2464996.2465444.
[17] T. Gautier, X. Besseron, L. Pigeon, KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors, in: Proc. of the PASCO'07, ACM, London, Canada, 2007, pp. 15–23.
[18] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, E. Quintana-Ortí, An extension of the StarSs programming model for platforms with multiple GPUs, Proc. of the 2009 Euro-Par, vol. 5704, Springer, 2009, pp. 851–862.
[19] R.M. Badia, J.R. Herrero, J. Labarta, J.M. Pérez, E.S. Quintana-Ortí, G. Quintana-Ortí, Parallelizing dense and banded linear algebra libraries using SMPSs, Concurr. Comput.: Pract. Exp. 21 (2009) 2438–2456.
[20] J. Planas, R. Badia, E. Ayguade, J. Labarta, Self-adaptive OmpSs tasks in heterogeneous environments, in: Proc. of the 27th IEEE IPDPS, 2013, pp. 138–149. doi:10.1109/IPDPS.2013.53.
[21] J. Bueno, X. Martorell, R.M. Badia, E. Ayguadé, J. Labarta, Implementing ompss support for regions of data in architectures with multiple address spaces, in: Proc. of the 27th ACM ICS, ACM, New York, NY, USA, 2013, pp. 359–368, http://dx.doi.org/10.1145/2464996.2465017.
[22] J. Toss, T. Gautier, A New Programming Paradigm for GPGPU, Proc. of the 2012 Euro-Par, vol. 7484, Springer, 2012, pp. 895–907, http://dx.doi.org/10.1007/978-3-642-32820-688.
[23] F. Broquedis, T. Gautier, V. Danjean, libKOMP, an efficient OpenMP runtime system for both fork-join and data flow paradigms, in: IWOMP, Rome, Italy, 2012, pp. 102–115.
[24] A. YarKhan, J. Kurzak, J. Dongarra, Quark users' guide: Queueing and runtime for kernels, Tech. Rep. ICL-UT-11-02, University of Tennessee, 2011.
[25] F. Lementec, V. Danjean, T. Gautier, X-Kaapi C programming interface, Rapport Technique RT-0417, INRIA, Dec. 2011.
[26] F. Galilee, G. Cavalheiro, J.-L. Roch, M. Doreille, Athapascan-1: on-line building data flow graph in a parallel language, in: Proc. of the 1998 PACT, 1998, pp. 88–95, http://dx.doi.org/10.1109/PACT.1998.727176.
[27] G. Quintana-Ortí, F.D. Igual, E.S. Quintana-Ortí, R.A. van de Geijn, Solving dense linear systems on platforms with multiple hardware accelerators, SIGPLAN Not. 44 (4) (2009) 121–130, http://dx.doi.org/10.1145/1594835.1504196.
[28] R.D. Blumofe, C.E. Leiserson, Space-efficient scheduling of multithreaded computations, SIAM J. Comput. 27 (1998) 202–229, http://dx.doi.org/10.1137/S0097539793259471.
[29] Y. Guo, J. Zhao, V. Cave, V. Sarkar, SLAW: a scalable locality-aware adaptive work-stealing scheduler, in: Proc. of the 24th IEEE IPDPS, 2010, pp. 1–12, http://dx.doi.org/10.1109/IPDPS.2010.5470425.
[30] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Comput. 35 (1) (2009) 38–53, http://dx.doi.org/10.1016/j.parco.2008.10.002.