A generic scheduler to foster data locality for GPU and out-of-core task-based applications

Maxime Gonthier, Samuel Thibault, Loris Marchal

A generic scheduler to foster data locality for GPU and out-of-core task-based applications

Maxime Gonthier
maxime.gonthier@ens-lyon.fr
LaBRI, LIP, Inria & ENS-Lyon
France

Loris Marchal
loris.marchal@ens-lyon.fr
LIP, CNRS, ENS-Lyon, Inria
France

Samuel Thibault
samuel.thibault@u-bordeaux.fr
LaBRI, CNRS, Inria & Univ. Bordeaux
France

ABSTRACT

Hardware accelerators, such as GPUs, now provide a large part of the computational power used for scientific simulations. GPUs come with their own (limited) memory and are connected to the main memory of the machine via a bus with limited bandwidth. Scientific simulations often operate on very large data, to the point of not fitting in the limited GPU memory. In this case, one has to turn to *out-of-core* computing: data are kept in the CPU memory, and moved back and forth to the GPU memory when needed for the computation. This out-of-core situation also happens when processing on multicore CPUs with limited memory huge datasets stored on disk. In both cases, data movement quickly becomes a performance bottleneck.

Task-based runtime schedulers have emerged as a convenient and efficient way to manage large applications on such heterogeneous platforms. They are in charge of choosing which tasks to assign on which processing unit and in which order they should be processed.

In this work, we focus on this problem of scheduling for a task-based runtime to improve data locality in an out-of-core setting, in order to reduce data movements. We design a data-aware strategy for both task scheduling and data eviction from limited memories. We compare this to existing scheduling techniques in runtime systems. Using the StarPU runtime, we show that our strategy achieves significantly better performance when scheduling tasks on multiple GPUs with limited memory, as well as on multiple CPU cores with limited main memory.

ACM Reference Format:
Maxime Gonthier, Loris Marchal, and Samuel Thibault. 2023. A generic scheduler to foster data locality for GPU and out-of-core task-based applications. In *Proceedings of* . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High-performance computing applications, such as simulations for aeronautics, material strength, or seismology, continue to require more and more intensive computing power on ever larger amounts of data. This increasing computing demand is being met by using increasing parallelism, with large multicore processors and hardware accelerators such as GPUs. However, recent technology trends show a widening gap between peak compute speed and communication bandwidth as well as decrease in memory per gigaflop [16, 17]. With users trying to solve larger systems, it becomes common to encounter a situation where all input data of the problem cannot fit into the memory of the computing units (either CPUs or GPUs).

To avoid running out of memory, *out-of-core* computing has emerged, allowing the input data to stay on slow but large storage (typically disk) and to be loaded into the memory of the processing units (typically CPUs) whenever needed for the computation [4]. All cores of a multicore CPU share a common (limited) memory, as depicted on Figure 1. The bandwidth between the disk and the CPU memory is only of a few hundred MB/s, which has a strong impact on performance if data movements are not carefully handled to overlap them with computation. Similarly, one should avoid loading several times the same data but favor data reuse, by improving temporal data locality, i.e., by performing all computations on the same data before its eviction from the memory.

Recently, the trend has been to leverage GPUs in addition to CPUs, to achieve unforeseen computation speed as well as energy requirements efficiency. GPU can achieve multiple teraflops in performances but are limited by a shared bandwidth of a few thousands MB/s. In this case, *out-of-core* computing consists in keeping the whole input data in the (larger) memory of the CPU, and loading data in the memory of a GPU only when it is needed for computation (see Figure 2). In such a distributed memory context, data reuse is even more crucial as one has also to focus on spatial data locality, that is to aim at gathering all computations on the same data on the same GPU. Hence, the gap between communication speed and computation is a strong performance bottleneck when computing on large data.

In this paper, we consider that these these two out-of-core situations (multicore CPU with limited shared memory and GPUs with limited distributed memory) can be treated similarly, and we focus on any system with two levels of memory: one fast and limited memory devoted to computations (in red in Figures 1 and 2), and one large but slower memory devoted to storing the whole dataset (in green in Figures 1 and 2).

To harness the power of complex heterogeneous computing platforms, it has become very common to use task-based programming, i.e. to express the application computation as a Directed Acyclic Graph (DAG), and let a dynamic runtime system such as OmpSs [15], PaRSEC [14], or StarPU [7] manage the execution of the task graph over such distributed and heterogeneous platforms. The burden of allocating data in memory, choosing task processing order and mapping is thus offloaded from the application programmer to the

, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

runtime system, in the form of a task scheduling problem. As the runtime system handles both data and tasks, it has the opportunity to minimize data movement. The runtime scheduler must solve the complex task of deciding the mapping of the tasks to the processing units to improve spatial data locality, as well as the ordering of the tasks itself, to privilege the temporal locality of data, thus favoring data reuse and saving duplicate data transfers. In addition, this scheduler has to account for task priorities (some tasks may be more crucial to compute early than others) and task affinities (in case of a heterogeneous platform, some tasks have a larger acceleration factor on GPUs than others). Besides, it is also essential to trigger data transfers ahead of task execution (data prefetches) so that they can be overlapped by the computation of the previous tasks. Last but not least, when the embedded memory of the processing unit is full, the runtime has to carefully decide which data should be evicted from it, to make room for further data.

In this paper, our aim is to solve this challenging problem, that is we propose a **task scheduler for runtime systems managing several computing units with shared or distributed limited memory**. We also aim at managing data movements (loads and evictions). More precisely, we want to determine (i) the assignment of the tasks to the processing units to reach a good load balance and spatial data locality and (ii) the order in which tasks must be processed on each processing unit to optimize temporal data locality as well as maximize overlap between computations and data movements. Our solution is *data-centric*, trying to re-use data as much as possible: a processing unit is assigned all the "free" tasks, for which it already owns all input data. Then, when no more free tasks can be performed on a processing unit, we look for the data that, if transferred to its memory, would allow to assign many tasks on this processing unit, for a minimal transfer time. Our scheduling strategy also looks for a good trade-off between data locality and priority in the task graph. Finally, we design a custom eviction policy that takes advantage of the (limited) vision of tasks to be processed on a processing unit to avoid evicting data that will be useful in a near future.

In this paper, we make the following contributions:

- We review existing schedulers in task-based runtime systems.
- We propose DARTS, a Data-Aware Reactive Tasck Scheduler for out-of-core computing in task-based runtime systems with a custom eviction policy;
- We implemented DARTS within the StarPU runtime system;
- We compare the performance of DARTS and existing schedulers on two classical linear-algebra benchmarks (namely Cholesky and LU factorization), on two out-of-core settings: (i) multiple GPUs with distributed memory and (ii) a multicore CPU with shared memory.
- Our experiments demonstrate that our scheduler is able to achieve good performance even when the memory is very limited, whereas all existing schedulers see their performance drop. Besides, DARTS performs comparably to the best existing schedulers when memory is not limited.

Note that the present paper extends a previous version of DARTS [9]. The previous version was limited to independent tasks sharing data, hence it was unable to handle complete task graphs as required by runtime systems, and was only tested on restricted task sets.

The rest of this paper is organized as follows. Section 2 presents previous work in the area. Section 3 details the scheduling problem in runtime systems. Section 4 shows existing schedulers for runtime systems, and our scheduler is presented in Section 5. Section 6 presents the experiments to test our scheduler both on multiple GPUs and on CPU.

2 RELATED WORK

We review here existing work both on specific out-of-core algorithms and on scheduling for runtime systems.

Out-of-core algorithms. Out-of-core computation has been proposed to process large datasets for specific operations, mainly in linear algebra. Toledo [24] surveys such out-of-core for dense linear algebra operations. This appears also in sparse linear algebra: Demmel et al. [18] propose to reduce the amount of data transfers both between processing elements and from/to the main memory. Direct sparse solvers are known to produce large amount of temporary data, which makes out-of-core computing the only solution to factorize very large sparse matrices [4]. Marchal et al. [21] also focus on reducing data transfers for task trees arising in sparse direct solvers. Such algorithms are however tailored to specific application cases, while task-based runtime systems aim for generic-purpose designs.

Scheduling and data locality in runtime systems. As outlined in the introduction, runtime systems like OmpSs [15], PaRSEC [14], or StarPU [7] are increasingly popular to cope with the complexity of modern computing platforms. In this context, some runtimes have been striving to improve data locality for better performance. Many runtime systems such as XKaapi [19] rely on work-stealing for load balancing, which also gives some guarantee on data locality [1]. On the contrary, the StarPU runtime system automatically calibrates performance models to predict task execution times. Based on these predictions, the DMDAS scheduler (presented below in Section 4.2) schedules tasks on the resource on which they are expected to complete at the earliest, which also takes data transfers into account. These predictions, however, only rely on the current state of the memory, and do not take into account its limited size: when some new data are loaded in memory, other data may be evicted, which is not taken into account and may lead to incorrect predictions. Legion [10] allows the user to express locality thanks to data regions, and to provide a data mapping strategy to ensure that data is not moved around when it is not necessary.

For the specific case of computing matrix products on multiple GPUs, the PaRSEC runtime pays attention to the memory limitation and implements a control-flow in order to avoid critically overflowing the GPUs' memory [20].

Hence, no existing runtime system is able to automatically deal with both data locality and limited memory for general computations.

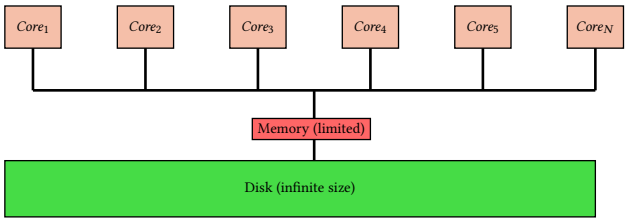


Figure 1: Platform topology of a multicore CPU with shared memory.

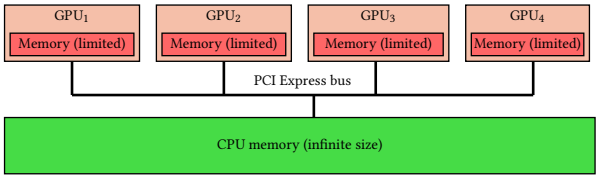


Figure 2: Platform topology of multiple GPUs with distributed memory.

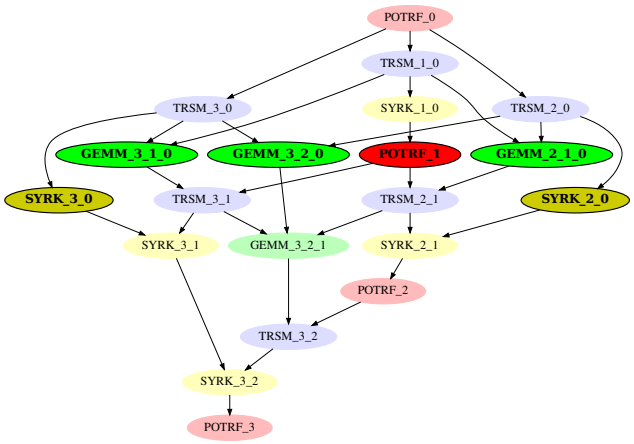


Figure 3: Taskset from the Cholesky factorization. Tasks with bright colors are tasks ready to be computed at a given time.

3 PROBLEM STATEMENT

We present here the model of the computing platform used when presenting the algorithms below, and we precisely state the objectives of a memory-aware scheduler for a runtime system.

3.1 Computing platform model

We concentrate in this paper on two different memory-limited architectures, namely a shared limited memory (Figure 1) and a distributed limited memory (Figure 2). We consider that the shared-memory setting is a special case of the distributed setting, as it includes a single memory, used by multiple cores which can be viewed as a single more efficient, parallel processing element. Hence, we concentrate the description of the problem and the algorithms on the more general distributed case, where the processing units are the GPUs and the large and slow storage is the main memory of the CPU. However, the problem and proposed algorithms can be easily translated for the shared memory case, as in the experiments reported in Section 6.4.

Each of the GPUs have their own memory with the same limited size M. They are all connected to the main memory of the machine through a shared communication link (PCI Express bus) whose bounded bandwidth is shared by all the communications taking place simultaneously between the main memory and some GPUs. All input data originally reside on the main memory, whose size is assumed sufficient to store all the original and temporary data of the computation.

Note that in addition to the PCI Express bus connecting all GPUs to the main memory, direct and faster connections may be available on some pairs of GPUs (such as NVidia NVLink or NVSwitch). This may allow to load data in a GPU faster than from the main memory if the data is already on another GPU.

3.2 Task-based runtime scheduler

The application is described by the programmer as a task graph, where vertices represent tasks and edges represent data dependencies between tasks. The programmer provides the code for each task as well as the description of its input and output data, allowing the runtime system to assign tasks to processing units and to move data around when needed. Figure 3 provides a small example of such a task graph. At a given time of the computation, some of the tasks have been completed, and some tasks are not available for computation as their input data has not been computed yet (tasks with pale colors on Figure 3). The tasks available for computation, i.e., not yet processed but with all their predecessors completed, form a subset of independant tasks in the graph, called the *ready tasks* (bright tasks in the figure). Several of these available tasks depend on common predecessors (e.g., both GEMM_3_1_0 and GEMM_3_2_0 depend on the result of TRSM_3_0), which means that they share a common input data produced by the common predecessor. The runtime scheduler has the knowledge of this dependency pattern and can take advantage of this to improve data locality while mapping and scheduling tasks (e.g., by mapping GEMM_3_1_0 and GEMM_3_2_0 to the same GPU). We denote by $\mathcal{D}(T)$ the input data of task T. Data may have different sizes, and tasks may have different processing times: this information is available to the scheduler through profiling.

The runtime system takes care of prefetching (loading a data a bit earlier so as to avoid waiting for unavailable data), marking tasks as available when their predecessors complete and managing task queues. The scheduler has the responsibility to map and order task on each processing unit. Given a set of ready tasks, when a processing unit is idling, the scheduler must send a task to this processing unit. If a processing unit needs to load a data in order to compute a task and the memory is saturated, the scheduler has to evict a data from the processing unit's memory.

In this paper, we are focusing on optimizing the scheduler for data locality, hence we propose algorithms to **map tasks to processing units**, **order tasks on each processing unit** and **evict data** from the limited memory when needed.

In a previous study, we proved that the simpler problem of ordering tasks sharing input data to minimize data movement on a single processing unit was NP-complete [8], hence the more general problem studied in the present paper is also NP-complete.

Note that we concentrate here only on tasks' input data: in the case of linear algebra for instance, the output data of task is most often smaller than the input data and can be transferred concurrently with data input. Data output is then not the driving constraint for efficient execution, as our experiments show. Our model could however be extended to integrate large task outputs if needed.

4 EXISTING RUNTIME SCHEDULERS

In this section, we present various algorithmic solutions that already exist in the literature to solve the partitioning and scheduling problem presented above. Some of these methods are greedy (Section 4.1), some first solve the partitioning problem, and then schedule the tasks on each processing unit (Section 4.2), some use work stealing to deal with load balancing and locality (Section 4.3), and some uses priority as a main focus point (Section 4.4).

4.1 The baseline: EAGER

The EAGER scheduler allows processing units (PUs) to pick up tasks on demand from a shared queue. The queue is filled with tasks as they get released by dependencies. Hence, task are processed in the order of their release.

4.2 Dynamic scheduler of StarPU: DMDAS

DMDA or "Deque Model Data Aware" (Algorithm 1) is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the StarPU runtime [6] (also called tmdp-pr). It computes for each processing unit PU_k the expected completion time $C(T_i, \mathrm{PU}_k)$ of the first task T_i in the queue of ready tasks, based on a prediction of the time required to transfer the data to the processing unit (*comm*) and of the task computation time (*comp*):

$$C(T_i, \mathrm{PU}_k) = \sum_{\substack{D_j \in \mathcal{D}(T_i) \\ D_j \notin memory(\mathrm{PU}_k)}} comm(D_j, \mathrm{PU}_k) + comp(T_i, \mathrm{PU}_k) \quad (1)$$

Note that the data transfer time is counted only if the data is not already in the memory of the processing unit. The task is then assigned to the processing unit PU_k which minimizes $C(T_i, \mathrm{PU}_k)$. Communication and computation times are predicted using a performance profile of the communication bus and processing unit which is calibrated beforehand. Tasks are assigned to processing units with this rule, one by one in the order of their release by the completion of their input dependencies.

DMDAS is a variant of DMDA that sorts tasks in the queue by priority order (as provided by the application). It is the default state-of-the-art scheduler used by the Chameleon library [3]. DMDAS also includes an additional *Ready* strategy (Algorithm 2): tasks are reordered at runtime in order to favor tasks with the most input data already loaded into memory[1].

Algorithm 1 Deque Model Data Aware heuristic (DMDA)

```
InMem ← ∅
while not all ready tasks have been assigned do
    T_i ← pop(readyTasks)
    Compute C(T_i, PU_k) using Eq. 1 for each PU_k
    Assign T_i to PU_k with smallest C(T_i, PU_k)
    for each D_j ∈ D(T_i) do
        Request data prefetch for D_j in PU_k
        Add D_j to memory(PU_k)
```

Algorithm 2 Ready reordering heuristic, called by PUs

Require: List L of assigned tasks

```
while L ≠ ∅ do
    Search first T ∈ L requiring the lowest amount of data transfers
    Wait for all data in D(T) to be in PU's memory
    Start processing T
```

In contrast to EAGER, DMDAS is a more advanced policy in terms of scheduling. It provides optimized execution thanks to its *Ready* strategy coupled with the DMDA algorithm and task priorities.

4.3 A work stealing policy: LWS

Locality Work Stealing (or LWS) is a scheduler that combines work stealing for load balancing, and locality to minimize communication.

Each worker has a queue of tasks planned for future execution. LWS can deal with locality in two ways. First, when a task is released, it is queued on the worker that released it. Thus, by default a task and its descendants are all scheduled on the same worker. In most cases, the descendants of a task all share at least one input data, so scheduling them on the same worker favors data reuse. Secondly, when a worker becomes idle, it steals a set of tasks from neighboring workers. It steals tasks from the end of their queue. There is a high chance that the inputs of these tasks have not been prefetched yet. This encourages workers to work on different input data. It thus favors distribution of different data to different workers, which increases the locality within each worker's task queue.

Work Stealing has proven to be efficient in situations where partitioning and reducing communication is critical (see [23], LWS is similar to HWS).

4.4 A priority-based scheduler from the PaRSEC runtime

DPLASMA [13] is a well-known implementation of dense linear algebra operations for heterogeneous architectures. It uses PaRSEC [14], a dynamic runtime for architecture-aware scheduling of

[1] https://files.inria.fr/starpu/testing/master/doc/html/Scheduling.html#DMTaskSchedulingPolicy

tasks on heterogeneous architectures. PaRSEC comes with different schedulers that all share two common aspects. First, they use the theoretical performance of CPUs and GPUs to balance the load assigned to each processing unit. Second, they use the knowledge of the DAG [22] to evaluate the cost of a task relative to its input, i.e. if multiple tasks use the same input data, the cost of a task will only be its computation time (without counting the data loading time) as its input data can be reused. Thus, multiple tasks that share input data have a higher probability of being computed on the same processing unit.

In our experiments, we use the Absolute Priority scheduler (AP) because of it affinities with priorities, an important feature for the applications we will be using. With AP, all processing units share a task waiting queue that is sorted by priority, and every time a processing unit becomes available, it takes for execution a task from the head of the waiting queue. We also evaluated the default scheduler of PaRSEC: LFQ. We found that AP always get slightly better performance on our applications. For this reason we only show AP in the following experiments.

5 PROPOSED STRATEGY FOR OUT-OF-CORE SITUATIONS: DARTS

We present here our main contribution, the DARTS scheduler (for Data-Aware Reactive Task Scheduling). A preliminary version of DARTS limited to tasks without dependencies has already been proposed [9]. For completeness, we outline here the whole scheduler, but we detail the new features.

5.1 Intuition

The principle of DARTS is to consider data locality before task assignation. The scheduling decision is based on the data (the data already loaded into the memory of the processing units and the data used by the ready tasks) and an ordered list of tasks is derived from it. The goal is to perform as many tasks as possible with the data available in the processing unit. Strategies presented in the previous section were favoring priority over locality: DMDAS and AP sort tasks by priority, favoring progress on the task graph critical path over data reuse. On the contrary, DARTS can contradict priority when useful to favor data reuse.

5.2 Task flow

In order to grasp DARTS' strategy, we need to understand the flow of tasks within the StarPU runtime as described in Figure 4. This figure uses the distributed memory case with GPUs because it is the more complete case, but the behavior is similar for shared memory with CPUs. The pink box represents the common StarPU core, and the boxes above it represent the specific scheduler, here DARTS. Tasks ready for execution are provided to DARTS through the *readyTasks* queue. When a GPU_k is idling, it will pull the head of the $taskBuffer_k$ queue of tasks. If that is empty, it will pull tasks from $plannedTasks_k$. And if that is empty, DARTS will be called to fill $plannedTasks_k$ using tasks from *readyTasks*. Note that tasks in $taskBuffer_k$ are in the StarPU common core and their data are being prefetched into the GPU memory. Thus, DARTS cannot access those tasks anymore. However, tasks in $plannedTasks_k$ can still be removed by DARTS as we will see in the eviction case (see section 5.4).

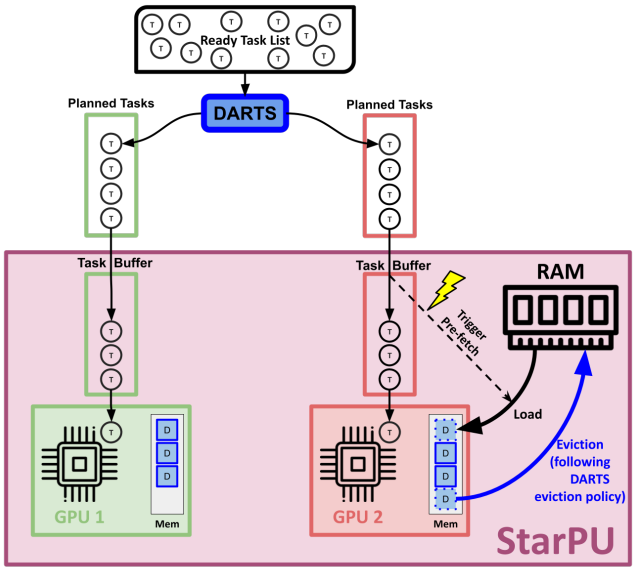


Figure 4: Task flow in StarPU when using DARTS

5.3 Strategy

DARTS is detailed in Algorithm 3. PU_k denotes the k-th processing unit. The principle of DARTS is, when requested for tasks to put in $plannedTasks_k$ for PU_k, to first look for the best new data to load, that is, a data that will allow as much work to be performed on PU_k as possible. Our first improvement over the previous DARTS design [9] comes from the selection of the best data to load. Whenever some PU_k requests for some task to execute, we first look in the $dataNotInMem_k$ set (which initially contains all data) for the optimal data D_{opt} that, if moved into the memory of PU_k, would minimize the ratio between the transfer duration of D_{opt} and the computation time of tasks that become "free", i.e., tasks that can be assigned and processed on PU_k without any additional data movement. Considering the transfer duration allows to favor direct communication between GPUs (using NVLinks), which is much faster than using the PCI Express bus. In case of equality between several optimal data, we break ties by considering task priorities and the remaining work associated with a data. Once such data is found, all the corresponding free tasks are assigned to $plannedTasks_k$. If we cannot find a data which enables free tasks, we then look for the data D_{opt} that will enable a high-priority task T provided that we perform one more data transfer: T depends on D_{opt}, on another data D not in memory, and possibly other data already in memory. It may happen that we do not find any such data, for example if all ready tasks depend on at least three data not in memory. In this case, the highest-priority task from *readyTasks* is assigned to PU_k.

We describe here the values required to select the optimal data D_{opt} (see Algorithm 3):

- $S_0(D)$: the set of tasks that depend only on D and some data already loaded in the memory of PU_k.

Algorithm 3 DARTS scheduler on PU_k

When PU_k requests a new task
if $plannedTasks_k = \emptyset$ **then** ▷ We need to fill $plannedTasks_k$
  for each data $D \in dataNotInMem_k$ **do**
    Compute $S_0(D)$, $S_1(D)$, $max\_prio(D)$, $computation\_time(D)$, $task\_left(D)$ and $transfer\_duration(D)$
    Compute $ratio = \frac{transfer\_duration(D)}{computation\_time(D)}$
  Choose the data D_{opt} with the smallest $ratio$, tiebreak in this order with $|S_0(D)|$, max_prio(D), $|S_1(D)|$ and $task\_left$(D)
  if $|S_0(D_{opt})| > 0$ **then**
    Append $S_0(D_{opt})$ to $plannedTasks_k$
    Remove D_{opt} from $dataNotInMem_k$
  else if $|S_1(D_{opt})| > 0$ **then**
    Choose task T with highest priority from $S_1(D_{opt})$
    Append T to $plannedTasks_k$
    Remove the inputs of T from $dataNotInMem_k$
  else
    Choose task T with highest priority from $readyTasks$
    Append T to $plannedTasks_k$
    Remove the inputs of T from $dataNotInMem_k$
Return head of $plannedTasks_k$

- $S_1(D)$: the set of tasks that depend only on D, some data already loaded in the memory of PU_k, and 1 additional data.
- $max\_prio(D)$: the highest priority of a task in $S_0(D)$ if $|S_0(D)| > 0$, otherwise the highest priority of a task in $S_1(D)$.
- $computation\_time(D)$: the sum of the durations of the tasks in $S_0(D)$.
- $task\_left(D)$: the sum of the durations of the tasks in $readyTasks$ that use D as an input.
- $transfer\_duration(D)$: time required to load D to PU_k.

The previous DARTS design only supported independent tasks. The support of task graphs with dependencies requires supporting the dynamic addition of tasks in *readyTasks* when their dependencies are resolved. When such addition occurs, we dynamically update each $dataNotInMem_k$, to include data used by this new ready task but which was not used by any other ready task, and not loaded or planned for load on PU_k. The $dataNotInMem_k$ sets thus always contain exactly the set of data used by tasks that can be started (either in *readyTasks*, in some $plannedTasks_k$, or in some $taskBuffer_k$), which are not yet loaded on PU_k.

Besides, for some newly-released ready tasks, we can bypass *readyTasks* and directly push them to $plannedTasks_k$ when they are already "free". Indeed, since the scheduler controls the loading of data, when a new task becomes ready we can check if it is already free on PU_k, i.e., if it can be already be processed by some PU_k without any additional data load. In such a case, we directly assign the new ready task to the corresponding $plannedTasks_k$. If several PUs qualify, we first consider the one with the fewest-queued tasks, to balance the load.

5.4 Eviction policy

In order to perform under memory constraints, DARTS needs a custom eviction policy, that matches its strategy. The goal is to keep "free" tasks in *taskBuffer* and *plannedTasks* so as not to contradict the task order. The eviction policy considers all data currently in memory, and first tries to evict data that is not useful for any task in $taskBuffer_k$ (as those tasks are close to being executed on PU_k), and which is an input of few tasks in $plannedTasks_k$. If this is not possible, we apply Belady's rule [12]: evict the input data whose next usage in $taskBuffer_k$ is the furthest in the future.

We have updated this eviction policy to update $dataNotInMem_k$ for each PU_k after the eviction. This is essential for tasksets with dependencies, as the dataset must be dynamically managed to reduce the computational complexity of finding D_{opt}. If the evicted data is not used by any task in *readyTasks*, we remove it from all $dataNotInMem_k$. It will be added there again when a new ready task with this input becomes available. Otherwise, if any task of *readyTasks* uses the evicted data, we add it to $dataNotInMem_k$ for each PU_k that does not hold it in memory (and is not scheduled to load it). This keeps $dataNotInMem_k$ up to date with the data that is currently required by ready tasks.

6 EXPERIMENTAL EVALUATION

We present below the experimental evaluation conducted to compare the strategies presented above.[2] The paper presenting the preliminary version of DARTS [9] only evaluated it on independent tasks, namely variants of the Matrix Multiplication. Our study is more in-depth and takes into account more complex applications, equipped with dependencies.

6.1 Settings

All strategies mentioned above have been implemented in the StarPU runtime system [7], and evaluated both in distributed and shared memory (see details below). We have most often limited the processing units memory to 2000 MB in order to better distinguish the performance of different strategies even on small datasets. All schedulers use the LRU eviction policy except for DARTS. The various schedulers have been tested on two linear algebra applications: the Cholesky decomposition ($A = L \times L^T$) and the LU factorization ($A = L \times U$) without pivoting. When the scheduler requires task priorities (which is the case for DMDAS, LWS and DARTS), they are computed as the bottom-level of the task in the task graph, which is the minimum time needed from the start of the task to the completion of the whole graph, assuming unbounded resources [5]. The PaRSEC AP scheduler uses its own set of priorities (although we noticed no difference when using bottom-level priorities).

We measure the obtained performance as the throughput of elementary computational operations performed per time unit (in GFlop/s), as well as the total volume of data transferred either between CPU and GPUs in the case of distributed memory with GPUs, or between CPU and disk in the case of shared memory. When measuring the throughput, the cost of computing the schedule is considered. Each point in the following plots is the average of the performance obtained over 10 iterations. For most of the results, the deviance is less than 2%, thus we do not show error bars in the following graphs. We plot the performance obtained with various

[2] The code to reproduce the results of this paper, including DARTS and the applications we used, is available at: https://shorturl.at/ghORU

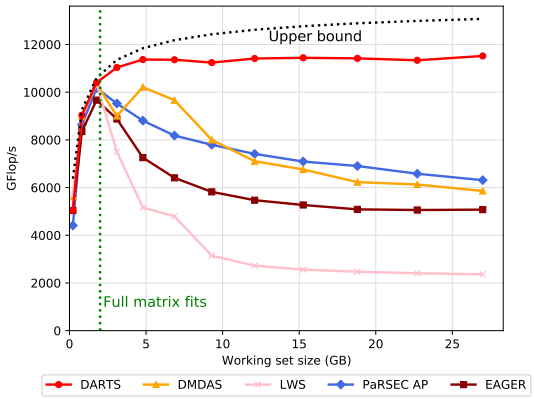


Figure 5: Performance on the Cholesky factorization with one Tesla V100 GPU. Memory limited to 2000 MB.

Schedulers	EAGER	LWS	AP	DMDAS	DARTS	bus limit
Transfers (GB)	449	1440	776	700	192	403

Table 1: Amount of data transfers on the last point of Figure 5.

problem sizes, ranging from a few hundreds MB up to 106 GB in order to test all strategies on various memory conditions.

6.1.1 Distributed memory with GPUs. We performed experiments on Tesla V100 GPUs (using cuBLAS 10.2 GPU kernels with single precision), equipped with a 12 GB/s PCI bus. Our application scenarios are the Cholesky decomposition and the LU factorization. We use tiles of size 1920 as it allows to achieve best peak performance on GPUs. Although being separate implementations for flexibility, our applications are identical to those of the Chameleon linear algebra library [2]. In the case of the PaRSEC-AP scheduler, we use the DPLASMA implementation of the Cholesky decomposition.

6.1.2 Shared memory with CPU. We used an AMD EPYC 7642 48 cores/CPU with disks that sustain a 350 MB/s bandwidth. The application scenario is the LU factorization. We use tiles of size 320, which is an optimized size for CPU cores.

6.2 Cholesky factorization with GPUs

Single GPU case. Figure 5 shows the results obtained by the various algorithms using one GPU. The dotted horizontal black line represents the maximum throughput that the GPU can achieve when processing the tasks for Cholesky without I/O, and is thus our asymptotic goal. The green dotted vertical line denotes the largest input that fits into the GPU memory.

From Figure 5 we can see that LWS and EAGER greatly suffer from the memory constraint, as their performance plummet after the green line. With only 1 GPU, they both process tasks in their natural order of arrival, resulting with a poor progress on the critical path for EAGER and a large number of data loads for LWS (see Table 1). On this table, "bus limit" is the amount of transfers that can be done during the minimum time for computation (given by the bound on the throughput), thus the hard limitation induced by the PCI bus bandwidth: a strategy exceeding this amount necessarily requires more time for the data transfers than the optimal time for computation. LWS and EAGER are both above it.

DMDAS has more sustained performance but is far from the asymptotic goal on large working set sizes. This is due to a conflict between data prefetching and eviction. Once the GPU is filled with data, it is not clear for DMDAS whether some data should be evicted in order to perform more prefetches. So it will rather stop prefetching data as long as all the data currently in the GPU will be useful for the subsequent tasks to be executed there. DMDAS does not reassign tasks according to the new data loaded on the GPU because it does not have a global view of the set of data and tasks, and thus cannot strike a balance between prefetching and eviction.

To better understand those results we can have a look at Figure 6. It corresponds to the 8th point of Figure 5. It shows the processing order of each task on the first 4 iterations of the Cholesky factorization. Each small square is a task. The first 1000 processed task are in red, the next 1000 in green, then blue, yellow, magenta, cyan and orange. Within each color, the fade (from light to dark) represents the processing order. The black square represents the amount of tiles the GPU can load in its memory. We can see that DMDAS processes tasks following anti diagonals (because of task priorities). If we look at the 1000 blue tasks for DMDAS, we can see that these diagonals do not allow much data reuse (there are only data shares between the k iterations), and if the memory cannot hold more tasks, the affinity on the rows and columns cannot be found, resulting in multiple loads of the same rows or columns. AP behaves similarly to DMDAS. It uses the expected completion time of tasks and sorts them by priority, resulting in similar results.

DARTS is able to maintain good performance while the working set size increases. DARTS also processes task in a diagonal while it can fit in memory (task in red on Figure 7) but then switches to *triangle blocks*. We can notice such blocks on the blue and magenta tasks (a triangle on the main diagonal preceded by a square of tasks of the same color on the first few columns). Those triangles and the associated square fit exactly in memory and share a lot of common data and are replicated over multiple iteration (up to k=8) in order to progress on the critical path, while maximizing data reuse. It has been proven that accessing the result matrix by *triangle blocks* is the most efficient in terms of data transfers [11], although in DARTS' scheduling, those triangles are only a result of a local greedy strategy to maximize data reuse. On Table 1 DARTS has the lowest amount of data transfers on the largest working set size and is the only strategy under the bus limit, meaning that theoretically, all transfers can be overlapped by a computation, which explains our good performance.

With multiple GPUs. Figure 8 shows the performance with 8 GPUs and Figure 9 shows the amount of data transfers. On Figure 8, the green dotted line now corresponds to the largest matrix that can be stored in the *aggregated* memory of all GPUs. We first notice that compared to the performance with a single GPU, all strategies are much further from the upper bound. The bus limit value is 250 GB on the last point of Figure 9, and all strategies generate more transfers. With multiple GPUs, some data has to be replicated on various GPU memories, which greatly increases the total amount

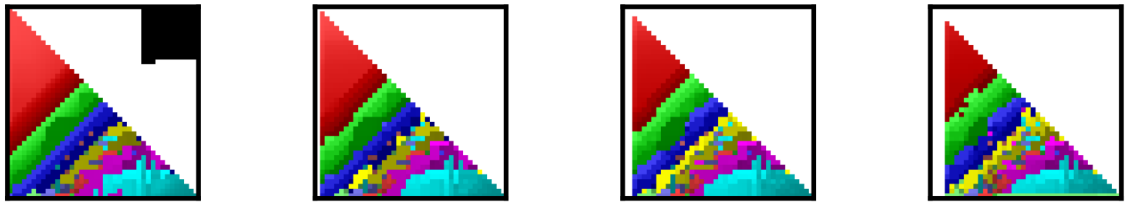


Figure 6: Visualization of iterations 1 to 4 of the Cholesky factorization with DMDAS and 1 GPU.

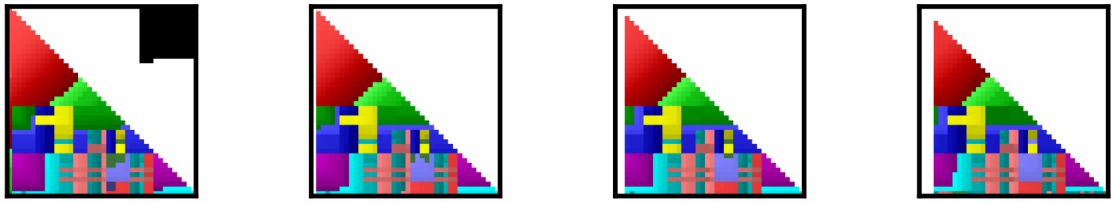


Figure 7: Visualization of iterations 1 to 4 of the Cholesky factorization with DARTS and 1 GPU.

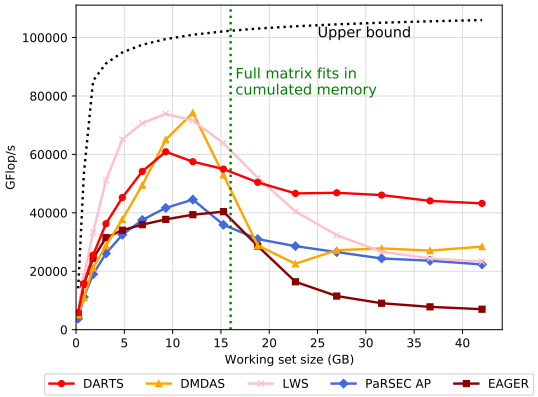


Figure 8: Performance on the Cholesky factorization with 8 Tesla V100 GPUs. Memory limited to 2000 MB per GPU.

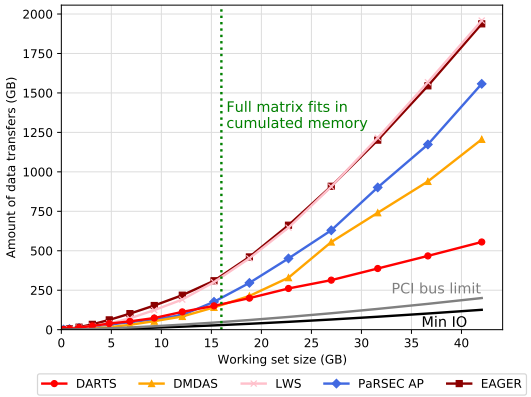
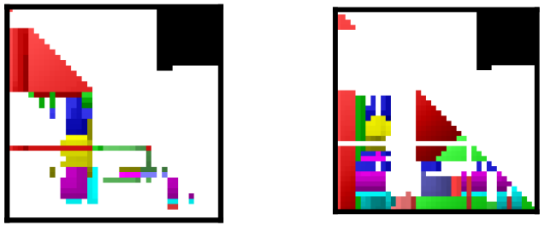


Figure 9: Amount of data transfers on the Cholesky factorization with 8 Tesla V100 GPUs. Memory limited to 2000 MB per GPU.

of data transfers. For matrices smaller than the limit size (green dotted line), LWS achieves the best performance. By stealing work from neighbor GPUs, LWS is able to largely increase transfers using NVLinks, which are much faster than transfers with the CPU memory. With 8 GPUs, the opportunity for such transfers are much more important which explains those performance. This explains why on Figure 9, even with more transfers than DARTS, LWS achieves a better throughput on the left of the green line. For larger matrices, LWS and EAGER have much more data transfers and thus much smaller throughput. They do not consider memory limitation and thus do not favor data reuse. Similarly for AP and DMDAS, the low throughput is associated with high transfer rates on Figure 9, as in the case with one GPU. However, before the constraint, as they consider the performance model to schedule, they are able to distribute tasks in a way that reduces the completion time. DARTS always gives a task to idling GPUs. With a lot of GPUs and few tasks, it is sometimes more efficient to assign many tasks sharing data to a single GPU, even if another one is idling, as it can lead to a smaller completion time overall.

DARTS has the best performance once the memory becomes a constraint. Figure 10 represents the task order with 2 GPUs (for clarity) using DARTS on the first iteration of the outer-loop of the Cholesky algorithm. We can still find those *triangle blocks* that allows us to reduce data transfers, as can be seen on Figure 9. Moreover, there is a good distribution of the data load on the 2 GPUs. When choosing the data to load D_{opt}, DARTS uses a ratio between transfer time and computation time of task using D_{opt}. This favors the selection of a data associated with as many unprocessed tasks as possible. Thus two distinct processing units have a low probability to select the same data as the total work associated with a data is reduced after some of its task is scheduled on another processing



(a) Task processed by GPU 1. (b) Task processed by GPU 2.

Figure 10: Visualization of the first iteration of the Cholesky factorization with DARTS and 2 GPUs.

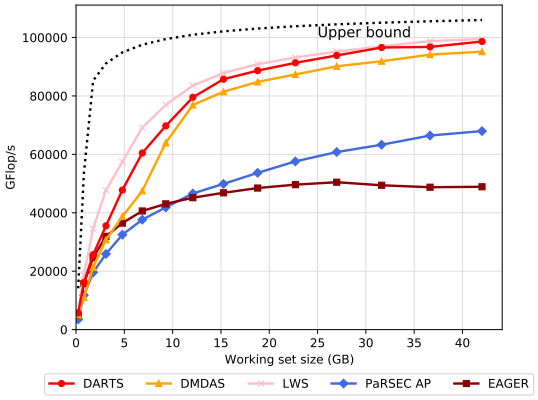


Figure 11: Performance on the Cholesky factorization with 8 Tesla V100 GPU. Hardware limitation of each GPU memory at 32 GB.

unit. This encourages GPUs to work on different datasets, further reducing the total amount of data to load by minimizing the replication of data on multiple GPUs. Moreover, once the workload is big enough, keeping some GPU idle is not beneficial. The sustained performance of DARTS with both 1 and 8 GPUs shows that it is generic enough to adapt to various numbers of processing units.

With multiple GPUs and no memory limitation. Figure 11 shows the results obtained with 8 GPUs without imposing a memory limit (each GPU is equipped with 32 GB of RAM). As mentioned before, DARTS is unable to keep some GPU idle, which leads to slightly worse results than LWS on the first few points. Apart from this, DARTS performs very similarly to LWS, which is the best strategy here. Our scheduling strategy can thus also be used in situations where memory is not a constraint, making DARTS generic enough to adapt to various memory sizes.

6.3 LU factorization with GPUs

Results on 4 GPUs. Figure 12 presents performance using the LU factorization with 4 GPUs. This figure does not present results for PaRSEC-AP, as the DPLASMA implementation of LU does not make use of the cuSolver library to solve getrf kernels of the LU factorization. Hence, AP uses a much slower version of this kernel, which makes it impossible for us to compare AP on fair grounds.

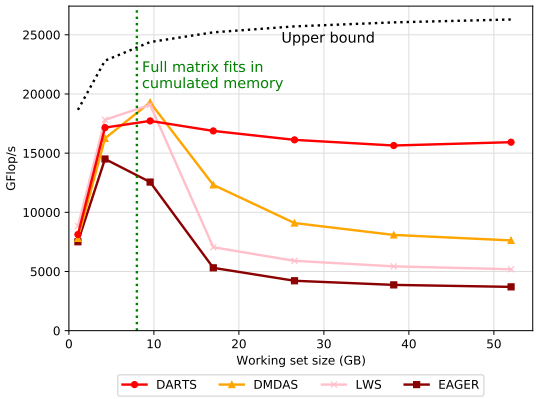


Figure 12: Performance on the LU factorization with 4 Tesla V100 GPU. Memory limited to 2000 MB per GPU.

Schedulers	EAGER	LWS	DMDAS	DARTS	bus limit
Data transfers (GB)	1312	1347	871	383	85

Table 2: Amount of data transfers on the last point of Figure 12

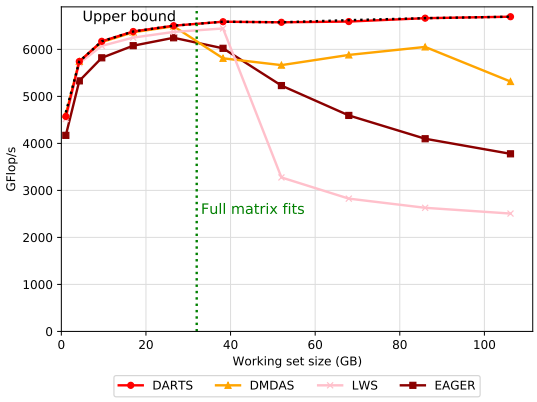


Figure 13: Performance on the LU factorization with 1 Tesla V100 GPU. Hardware memory limitation at 32 GB.

LWS and EAGER process tasks in their submission order (sorted by priorities for LWS), which makes it impossible to reuse data on consecutive tasks when memory is limited. Hence, these two schedulers end up with 3 times more data transfers than DARTS (see Table 2). DARTS has more sustained performance, and similarly to the Cholesky case, is able to reduce data transfers by distributing the data on multiple GPUs and reusing data for consecutive tasks.

Results on a single GPU and no memory limitation. Figure 13 shows the results for the LU factorization on a single GPU without limiting the memory. The GPU embeds 32GB of RAM, we can see this value on the plot with the green line. DMDAS has more sustained performance than LWS and EAGER. The *Ready* strategy

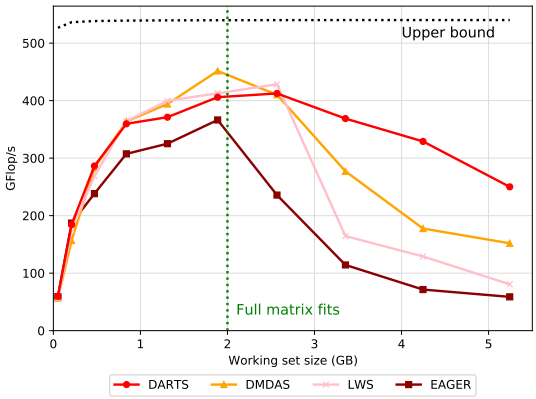


Figure 14: Performance on the LU factorization with AMD EPYC 7642 48 cores/CPU. Memory limited to 2000 MB.

(see Algo 2), allows DMDAS to re-order task so as to compute first tasks for which the input data is already in memory, thus reducing data transfers. DARTS stays very close to the maximum obtainable GFlop/s for the whole range of matrix sizes. LU factorization has a higher rate of computation-per-byte compared to Cholesky, so that schedulers can more easily stay under the limit induced by the PCI bus. Here DARTS has two times less transfers than this limit (210 GB compared to 447 GB for the bus limit on the last point of Figure 13) and it is able to completely overlap communications with computations, even after the memory limitation, induced by the hardware itself in this case (only 1 GPU).

6.4 LU factorization on a multicore CPU

As outlined before, our scheduler is able to consider any memory-limited system, and StarPU manages disk-CPU transfers exactly like CPU-GPU transfers. Figure 14 reports experiments on a 48-core CPU with 2 GB of shared memory. In this shared-memory setting, DARTS behaves in a similar way as it does with one GPU: it uses a single *plannedTasks* queue as well as a single *taskBuffer*. Again, DARTS is able to achieve good performance even when the memory is limited, while other schedulers see their throughput drop whenever the whole dataset cannot be stored in memory.

7 CONCLUSION

In this paper, we have focused on the problem of scheduling tasks in runtime systems to handle very large datasets that do not fit into the memory of the processing units. We have largely improved a scheduler for the StarPU runtime system, named DARTS, to cope with task graphs. Our scheduler is mainly focused on data locality but also takes task priorities into account. We have perform experiments with two classical linear algebra operations: Cholesky and LU factorizations. Thanks to its modularity, DARTS reaches very good performance in a large variety of situations, from multicore CPU with shared memory to multiple GPUs with distributed memory. Among available competitors, DARTS in the only scheduler to reach good performance in memory-limited scenarios, and it also ranks among the best ones in all settings. Future works include taking advantage of the modularity of DARTS to design hierarchical schedulers for large distributed platforms, and to combine it with other schedulers (such as work stealing) to benefit from all their features, as (locality-aware) work stealing is efficient to balance load among distributed nodes, while DARTS allows to cope with limited memories.

REFERENCES

[1] Acar, U. A., Blelloch, G. E., and Blumofe, R. D. The data locality of work stealing. *Theory Comput. Syst. 35*, 3 (2002), 321–347.
[2] Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., and Tomov, S. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, vol. 2. Morgan Kaufmann, 2010.
[3] Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., and Thibault, S. P. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems* (2017).
[4] Agullo, E., Guermouche, A., and L'Excellent, J.-Y. A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory. *Parallel Computing 34*, 6 (2008), 296–317.
[5] Alias, C., Thibault, S., and Gonnord, L. A Compiler Algorithm to Guide Runtime Scheduling. Research Report RR-9315, INRIA Grenoble ; INRIA Bordeaux - Sud-Ouest, Dec. 2019.
[6] Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *Int. Conf. on Parallel and Distributed Systems* (2010).
[7] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23* (2011).
[8] Authors removed for double-blind submission, A. Locality-Aware Scheduling of Independent Tasks for Runtime Systems. In *COLOC - 5th workshop on data locality of EuroPAR* (2021).
[9] Authors removed for double-blind submission, A. Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems. In *36th IEEE International Parallel and Distributed Processing Symposium* (2022).
[10] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012).
[11] Beaumont, O., Eyraud-Dubois, L., Vérité, M., and Langou, J. I/O-Optimal Algorithms for Symmetric Linear Algebra Kernels. In *ACM Symposium on Parallelism in Algorithms and Architectures* (2022).
[12] Belady, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal 5*, 2 (1966).
[13] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemariner, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011).
[14] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., and Dongarra, J. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering 15*, 6 (2013).
[15] Bueno, J., Planas, J., Duran, A., Badia, R. M., Martorell, X., Ayguade, E., and Labarta, J. Productive programming of GPU clusters with OmpSs. In *International Parallel and Distributed Processing Symposium* (2012).
[16] Cheng, Q., Glick, M., and Bergman, K. Chapter 20 - Optical interconnection networks for high-performance systems. In *Optical Fiber Telecommunications VII*, A. E. Willner, Ed. Academic Press, 2020.
[17] Council, N. R., et al. *Getting up to speed: The future of supercomputing*. National Academies Press, 2005.
[18] Demmel, J., Hoemmen, M., Mohiyuddin, M., and Yelick, K. Avoiding communication in sparse matrix computations. In *IEEE International Symposium on Parallel and Distributed Processing* (2008).
[19] Ferreira Lima, J. V., Gautier, T., Danjean, V., Raffin, B., and Maillard, N. Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures. *Parallel Computing 44* (2015), 37–52.
[20] Herault, T., Robert, Y., Bosilca, G., and Dongarra, J. Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC. In *Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)* (2019).
[21] Marchal, L., McCauley, S., Simon, B., and Vivien, F. Minimizing I/Os in Out-of-Core Task Tree Scheduling. *International Journal of Foundations of Computer Science 34*, 01 (2023), 51–80.
[22] Moustafa, S., Faverge, M., Plagne, L., and Ramet, P. 3D cartesian transport

sweep for massively parallel architectures with PARSEC. In *IEEE International Parallel and Distributed Processing Symposium* (2015), pp. 581–590.

[23] Quintin, J.-N., and Wagner, F. Hierarchical work-stealing. In *Euro-Par 2010 - Parallel Processing* (2010), pp. 217–229.

[24] Toledo, S. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*. American Mathematical Society Press, 1999, pp. 161–180.