



HAL
open science

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

Maxime Gonthier, Samuel Thibault, Loris Marchal

► To cite this version:

Maxime Gonthier, Samuel Thibault, Loris Marchal. A generic scheduler to foster data locality for GPU and out-of-core task-based applications. 2024. hal-04146714v2

HAL Id: hal-04146714

<https://inria.hal.science/hal-04146714v2>

Preprint submitted on 13 Sep 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

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

Maxime Gonthier^a, Loris Marchal^b, Samuel Thibault^c

^aUniversity of Chicago, 5730 S Ellis Ave, Chicago, 60637, Illinois, United States of America

^bCNRS & ENS-Lyon, 46 alle d'Italie, Lyon, 69364, Auvergne-Rhone-Alpes, France

^cUniversity of Bordeaux, 200 Avenue de la Vieille
Tour, Talence, 33405, Nouvelle-Aquitaine, France

Abstract

Hardware accelerators like GPUs now provide a large part of the computational power used for scientific simulations. Despite their efficacy, GPUs possess limited memory and are connected to the main memory of the machine via a bandwidth limited bus. Scientific simulations often operate on very large data, that surpasses the GPU's memory capacity. Therefore, one has to turn to **out-of-core** computing: data are kept in a remote, slower memory (CPU memory), and moved back and forth from/to the device memory (GPU memory), a process also present for multicore CPUs with limited memory. 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. **We propose a scheduler for task-based runtimes** that improves **data locality** in an out-of-core setting, to reduce data movements. We design a data-aware strategy for both task scheduling and data eviction from limited memories. We compare this scheduler to existing schedulers in runtime systems. Using STARPU, we show that our new scheduling strategy achieves comparable performance when memory is not a constraint, and significantly better performance when application input data exceeds memory, on both GPUs and CPU cores.

Keywords: Scheduling, Memory-aware scheduling, Eviction policy, Tasks sharing data, Runtime systems, Data locality, GPUs, CPU

Email addresses: mgonthier@uchicago.edu (Maxime Gonthier),
loris.marchal@ens-lyon.fr (Loris Marchal), samuel.thibault@u-bordeaux.fr
(Samuel Thibault)

Preprint submitted to Journal of Parallel and Distributed Computing

September 12, 2024

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 [1, 2]. 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 [3]. 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 unprecedented 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 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).

39 To harness the power of complex heterogeneous computing platforms, it has
40 become very common to use task-based programming, i.e. to express the ap-
41 plication computation as a Directed Acyclic Graph (DAG), and let a dynamic
42 runtime system such as OmpSs [4], PaRSEC [5], or STARPU [6] manage the exe-
43 cution of the task graph over such distributed and heterogeneous platforms. The
44 burden of allocating data in memory, choosing task processing order and map-
45 ping is thus offloaded from the application programmer to the runtime system,
46 in the form of a task scheduling problem. As the runtime system handles both
47 data and tasks, it has the opportunity to minimize data movement. The runtime
48 scheduler must solve the complex task of deciding the mapping of the tasks to
49 the processing units to improve spatial data locality, as well as the ordering of
50 the tasks itself, to privilege the temporal locality of data, thus favoring data reuse
51 and saving duplicate data transfers. In addition, this scheduler has to account
52 for task priorities (some tasks may be more crucial to compute early than oth-
53 ers) and task affinities (in case of a heterogeneous platform, some tasks have a
54 larger acceleration factor on GPUs than others). Besides, it is also essential to
55 trigger data transfers ahead of task execution (data prefetches) so that they can
56 be overlapped by the computation of the previous tasks. Last but not least, when
57 the embedded memory of the processing unit is full, the runtime has to carefully
58 decide which data should be evicted from it, to make room for further data.

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

76 In this paper, we make the following contributions:

- 77 • We review existing schedulers in task-based runtime systems;
- 78 • We propose DARTS, a Data-Aware Reactive Task Scheduler for out-of-
79 core computing in task-based runtime systems with a custom eviction pol-
80 icy;
- 81 • We implement DARTS within the STARPU runtime;
- 82 • We compare the performance of DARTS and existing schedulers on two
83 classical linear-algebra benchmarks (namely Cholesky and LU factoriza-
84 tion) and on two out-of-core settings: (i) multiple GPUs with distributed
85 memory and (ii) a multicore CPU with shared memory;
- 86 • Our experiments demonstrate that our scheduler is able to achieve good
87 performance even when the memory is very limited, whereas all existing
88 schedulers see their performance drop. Besides, DARTS performs compa-
89 rably to the best existing schedulers when memory is not limited.

90 This paper extends a preliminary work on the DARTS scheduler [7]. Al-
91 though both versions use the same intuition for scheduling tasks, the earlier ver-
92 sion of DARTS was limited to independent tasks, and hence was only tested on
93 restricted datasets. The redesigned version of the DARTS scheduler is able to
94 handle general task graphs as required by runtime systems. It also uses an im-
95 proved eviction policy, and special care has been devoted to reducing its compu-
96 tational complexity. While the previous version of DARTS was only tested with
97 artificially limited memory for the GPUs and thus with much smaller datasets,
98 the present paper showcases experiments on significantly larger datasets using
99 the full memory of the devices, which corresponds to real-life scenarios. Be-
100 sides, the present paper also compares DARTS to many other schedulers from
101 the literature (LWS, DMDAS and AP from DPLASMA/ParSEC) and tests all
102 strategies on both CPUs and three different kind of GPUs. This large comparison
103 on large task graphs across various type of GPUs and CPUs was made possible
104 thanks to the new design of DARTS, its improved eviction policy and its reduced
105 complexity.

106 The paper is organized as follows. Section 2 presents previous work in the
107 area. Section 3 details the scheduling problem in runtime systems. Section 4
108 presents existing schedulers for runtime systems; our scheduler is introduced in
109 Section 5. Section 7 relates the experiments conducted to evaluate the perfor-
110 mance of the schedulers on both multiple GPUs and CPU cores.

111 2. Related work

112 To address the problem of fitting a working set in a given amount of mem-
113 ory [8], researchers explored various strategies that we review here.

114 *Out-of-core algorithms.* Out-of-core computation has been proposed to process
115 large datasets for specific operations, mainly in linear algebra. Toledo [9] sur-
116 veys such out-of-core for dense linear algebra operations. This appears also in
117 sparse linear algebra: Demmel et al. [10] propose to reduce the amount of data
118 transfers both between processing elements and from/to the main memory. Di-
119 rect sparse solvers are known to produce large amount of temporary data, which
120 makes out-of-core computing the only solution to factorize very large sparse ma-
121 trices [3]. Marchal et al. [11] also focus on reducing data transfers for task trees
122 arising in sparse direct solvers. Such algorithms are however tailored to specific
123 application cases, while task-based runtime systems aim for generic-purpose de-
124 signs.

125 *Scheduling and data locality in runtime systems.* As outlined in the introduction,
126 runtime systems like OmpSs [4], PaRSEC [5], or STARPU [6] are increasingly
127 popular to cope with the complexity of modern computing platforms.

128 In this context, some runtimes have been striving to improve data local-
129 ity for better performance. For example, runtime systems such as OmpSs and
130 XKaapi [12] rely on work-stealing for load balancing. XKaapi also gives some
131 guarantee on data locality [13] by providing a lower bound on the number of
132 data accesses required by its scheduler. They also show that their locality-guided
133 work-stealing policy performs significantly better than standard work-stealing.
134 We will use a similar work-stealing policy to evaluate our own strategies with
135 the LWS scheduler introduced in Section 4.3.

136 On the contrary, the STARPU runtime system automatically calibrates perfor-
137 mance models to predict task execution times. Based on these predictions, the
138 DMDAS scheduler (presented below in Section 4.2) schedules tasks on the re-
139 source on which they are expected to complete at the earliest, which also takes
140 data transfers into account. These predictions, however, only rely on the current
141 state of the memory, and do not take into account its limited size: when some
142 new data are loaded in memory, other data may be evicted, which is not taken
143 into account and may lead to incorrect predictions.

144 Legion [14] allows the user to express locality thanks to data regions, and to
145 provide a data mapping strategy to ensure that data is not moved around when
146 it is not necessary. PaRSEC, previously known as DAGuE [15], uses a different

147 DAG representation, compared to STARPU and other runtimes: it uses param-
148 eterized task graphs [16]. The advantage of such a model is the concise rep-
149 resentation of the DAG: each task is an algebraic representation that indicates
150 which type of task is to be executed after a task is finished. Thus, the memory
151 required for the DAG is only relative to the number of different task types. How-
152 ever, PaRSEC strategies do not specifically address the problem of scheduling
153 under memory constraints. We compare our proposed scheduler to a PaRSEC
154 scheduler presented in Section 4.4.

155 The Python-based Parla runtime [17] provides special data wrappers (Parla
156 Arrays) which allows flexible memory management. Data locality is then con-
157 sidered when scheduling computations through a cost function that mixes the
158 time required for moving data and the load of each node. This is very similar
159 to the DMDAS scheduler mentioned for STARPU. Additionally, Parla does not
160 specifically manage limited memory.

161 Data distribution can also be managed with modern high-level libraries. PHA-
162 ST [18] or SYCL coupled with the Celerity API [19] can automate parallelization
163 while leaving room to the programmer to direct data distribution on nodes. How-
164 ever, it must be manually programmed and does not specifically address memory
165 limitation.

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

168 **3. Problem statement**

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

172 *3.1. Computing platform model*

173 We concentrate in this paper on two different memory-limited architectures,
174 namely a shared limited memory (Figure 1) and a distributed limited memory
175 (Figure 2). We consider that the shared-memory setting is a special case of the
176 distributed setting, as it includes a single memory, used by multiple cores which
177 can be viewed as a single more efficient, parallel processing element. Hence,
178 we concentrate the description of the problem and the algorithms on the more
179 general distributed case, where the processing units are the GPUs and the large
180 and slow storage is the main memory of the CPU. However, the problem and
181 proposed algorithms can be easily translated for the shared memory case, as in
182 the experiments reported in Section 7.3.

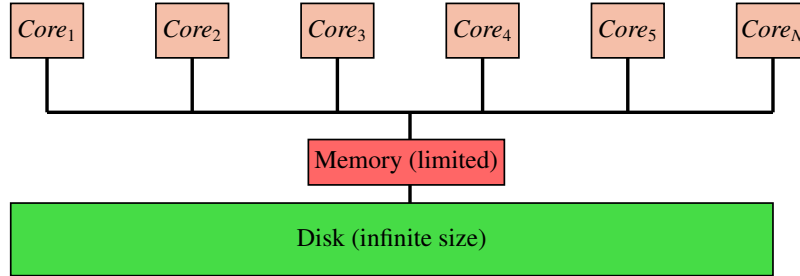


Figure 1: Platform topology of a multicore CPU with shared memory.

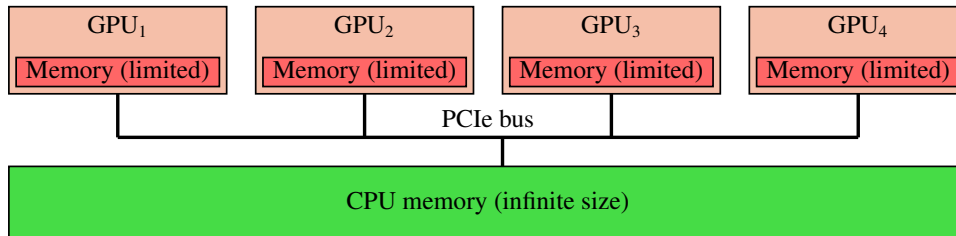


Figure 2: Platform topology of multiple GPUs with distributed memory.

183 Each of the GPUs have their own memory with the same limited size M .
 184 They are all connected to the main memory of the machine through a shared com-
 185 munication link (PCIe bus) whose bounded bandwidth is shared by all the com-
 186 munication taking place simultaneously between the main memory and some
 187 GPUs. All input data originally reside on the main memory, whose size is as-
 188 sumed sufficient to store all the original and temporary data of the computation.

189 Note that in addition to the PCIe bus connecting all GPUs to the main mem-
 190 ory, direct and faster connections may be available on some pairs of GPUs (such
 191 as NVidia NVLink). This may allow to load data in a GPU faster than from the
 192 main memory if the data is already on another GPU.

193 3.2. Task-based runtime scheduler

194 The application is described by the programmer as a task graph, where ver-
 195 tices represent tasks and edges represent data dependencies between tasks. The
 196 programmer provides the code for each task as well as the description of its input
 197 and output data, allowing the runtime system to assign tasks to processing units
 198 and to move data around when needed. Figure 3 provides a small example of
 199 such a task graph. At a given time of the computation, some of the tasks have
 200 been completed, and some tasks are not available for computation as their in-
 201 put data has not been computed yet (tasks with pale colors on Figure 3). The


```

Cholesky_decomposition(A):
for(k=0; k<N; k++)
  A[k][k]=POTRF(A[k][k])
  for(m=k+1; m<N; m++)
    A[m][k]=TRSM(A[k][k], A[m][k])
    for(n=k+1; n<N; n++)
      A[n][k]=SYRK(A[n][k], A[n][n])
      for(m=n+1; m<N; m++)
        A[m][n]+=GEMM(A[m][k], A[n][k])

```

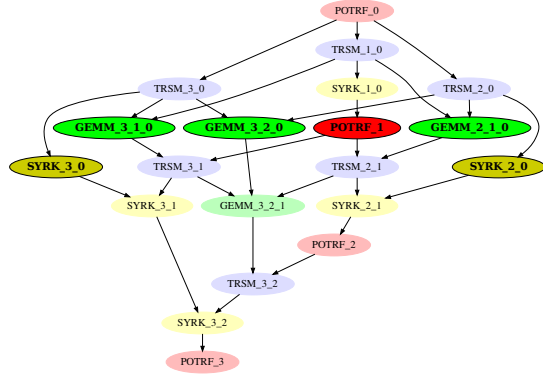


Figure 3: Pseudo-code for the Cholesky factorization (left) and corresponding task graph for $N = 3$ (right). Arrows represent data dependencies between tasks. Tasks with bright colors depicts tasks available at a given moment in the computation; they are independent of each other.

202 tasks available for computation, i.e., not yet processed but with all their prede-
 203 cessors completed, form a subset of independent tasks in the graph, called the
 204 *ready tasks* (bright tasks in the figure). Several of these available tasks depend
 205 on common predecessors (e.g., both GEMM_3_1_0 and GEMM_3_2_0 depend
 206 on the result of TRSM_3_0), which means that they share a common input data
 207 produced by the common predecessor. The runtime scheduler has the knowl-
 208 edge of this dependency pattern and can take advantage of this to improve data
 209 locality while mapping and scheduling tasks (e.g., by mapping GEMM_3_1_0
 210 and GEMM_3_2_0 to the same GPU). We denote by $\mathcal{D}(T)$ the input data of task
 211 T . Data may have different sizes, and tasks may have different processing times:
 212 this information is available to the scheduler through profiling.

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

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

223 In a previous study, we proved that the simpler problem of ordering tasks
 224 sharing input data to minimize data movement on a single processing unit was
 225 NP-complete [20], hence the more general problem studied in the present paper

226 is also NP-complete.

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

232 4. Existing runtime schedulers

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

239 4.1. The baseline: EAGER

240 The EAGER scheduler is a greedy scheduler that lets processing units (PUs)
241 pick up tasks on demand from a shared queue. The queue is filled with tasks as
242 they get released by dependencies. Hence, tasks are processed in the order of
243 their release. Like all other presented schedulers, it prefetches data of tasks to be
244 computed soon.

245 4.2. Dynamic scheduler of STARPU: DMDAS

DMDA or “Deque Model Data Aware” (Algorithm 1) is a dynamic schedul-
ing heuristic designed to schedule tasks on heterogeneous processing units in
the STARPU runtime [21] (also called tmdp-pr). DMDA is based on the HEFT
scheduler [22], which is already known to be an efficient scheduler. It computes
for each processing unit PU_k the expected completion time $C(T_i, PU_k)$ 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*):

$$C(T_i, PU_k) = \sum_{\substack{D_j \in \mathcal{D}(T_i) \\ D_j \notin \text{mem}(PU_k)}} \text{comm}(D_j, PU_k) + \text{comp}(T_i, PU_k)$$

246 Note that the data transfer time is counted only if the data is not already in the
247 memory of the processing unit. The task is then assigned to the processing unit
248 PU_k which minimizes $C(T_i, PU_k)$. Communication and computation times are

249 predicted using a performance profile of the communication bus and processing
 250 unit which is calibrated beforehand. Tasks are assigned to processing units with
 251 this rule, one by one in the order of their release by the completion of their input
 252 dependencies.

253 DMDAS is a variant of DMDA that sorts tasks in the queue by priority order
 254 (as provided by the application). It is the default state-of-the-art scheduler used
 255 by the Chameleon library [23]. DMDAS also includes an additional *Ready* strat-
 256 egy (Algorithm 2): tasks are reordered at runtime in order to favor tasks with the
 257 most input data already loaded into memory.

Algorithm 1 Deque Model Data Aware heuristic (DMDA)

```

1:  $InMem \leftarrow \emptyset$ 
2: while not all ready tasks have been assigned do
3:    $T_i \leftarrow pop(readyTasks)$ 
4:   Compute  $C(T_i, PU_k)$  using Eq. 4.2 for each  $PU_k$ 
5:   Assign  $T_i$  to  $PU_k$  with smallest  $C(T_i, PU_k)$ 
6:   for each  $D_j \in \mathcal{D}(T_i)$  do
7:     Request data prefetch for  $D_j$  in  $PU_k$ 
8:     Add  $D_j$  to  $memory(PU_k)$ 

```

Algorithm 2 Ready reordering heuristic, called by PUs

Require: List L of assigned tasks

```

1: while  $L \neq \emptyset$  do
2:   Search first  $T \in L$  requiring the lowest amount of data transfers
3:   Wait for all data in  $\mathcal{D}(T)$  to be in PU's memory
4:   Start processing  $T$ 

```

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

261 *4.3. A work stealing policy: LWS*

262 Work stealing policies exist on several runtimes systems like XKaapi [24],
 263 or libraries like TBB [25]. Work stealing has proven to be efficient in situations
 264 where partitioning and reducing communication is critical. We present here Lo-
 265 cality Work Stealing (or LWS), a scheduler that combines work stealing for load

266 balancing, and locality to minimize communication. LWS is similar to the HWS
267 scheduler introduced by Quintin et al. [26].

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

278 *4.4. A priority-based scheduler from the PaRSEC runtime: AP*

279 DPLASMA [27] is a well-known implementation of dense linear algebra op-
280 erations for heterogeneous architectures. It uses PaRSEC [5], a dynamic runtime
281 for architecture-aware scheduling of tasks. PaRSEC comes with different sched-
282 ulers that all share two common aspects. First, they use the theoretical perfor-
283 mance of CPUs and GPUs to balance the load assigned to each processing unit.
284 Second, they use the knowledge of the DAG [28] to evaluate the cost of a task
285 relative to its input, i.e. if multiple tasks use the same input data, the cost of a
286 task will only be its computation time (without counting the data loading time)
287 as its input data can be reused. Thus, multiple tasks that share input data have a
288 higher probability of being computed on the same processing unit.

289 In our experiments, we use the Absolute Priority scheduler (AP) because of
290 its affinities with priorities, an important feature for the applications we will be
291 using. With AP, all processing units share a task-waiting queue that is sorted by
292 priority, and each time a processing unit becomes available, it takes for execu-
293 tion a task from the head of the waiting queue. We also evaluated the default
294 scheduler of PaRSEC: LFQ. We found that AP always gets slightly better per-
295 formance on our applications. For this reason, we only show AP in the following
296 experiments.

297 **5. Improving the DARTS scheduler**

298 Here we present our re-designed DARTS scheduler. We highlight the new
299 features, but also include the main components of DARTS from [7].

300 5.1. Intuition

301 The principle of DARTS is to consider data locality before task allocation.
302 The scheduling decision is based on the data already loaded into the memory of
303 the processing units and the data used by the ready tasks. An ordered list of tasks
304 is derived from it. The goal is to perform as many tasks as possible with the data
305 available in the processing unit. Strategies presented in the previous section were
306 favoring priority over locality: DMDAS and ParSEC AP sort tasks by priority,
307 favoring progress on the task graph critical path over data reuse. On the contrary,
308 DARTS can contradict priority when useful to favor data reuse. Trading priority
309 for more locality is interesting when (i) a large number of tasks are available
310 for computation at a given time, and (ii) computation of tasks from the critical
311 path is not urgent because many tasks must be computed before the critical path
312 becomes really critical. In such a case, there is enough parallelism available to
313 occupy all the processing units, and favoring data locality can thus bring more
314 benefits. DARTS aims to take advantage of this fact by considering as a primary
315 factor the data that would bring the most data reuse if loaded, and as a secondary
316 factor, in case of a tie between two data to be loaded, the one associated with the
317 highest priority task.

318 5.2. Task flow

319 In order to grasp DARTS strategy, we need to understand the flow of tasks
320 within the STARPU runtime as described in Figure 4. This figure uses the dis-
321 tributed memory case with GPUs because it is the more complete case, but the
322 behavior is similar for shared memory with CPUs. The pink box represents
323 the common STARPU core, and the blue elements are the actions of the sched-
324 uler. Tasks ready for execution are provided to DARTS through the *readyTasks*
325 queue. When a GPU_k is idling, it will pull the head of the *taskBuffer_k*
326 of tasks. If that is empty, it will pull tasks from *plannedTasks_k*. And if that is
327 empty, DARTS will be called to fill *plannedTasks_k* using tasks from *readyTasks*.
328 Note that tasks in *taskBuffer_k* are in the STARPU common core and their data are
329 being prefetched into the GPU memory. Thus, DARTS cannot access those tasks
330 anymore. However, tasks in *plannedTasks_k* can still be removed by DARTS as
331 we will see in the eviction case (see section 5.4).

332 5.3. Strategy

333 The fundamental principle of DARTS is, when requested for tasks to put in
334 *plannedTasks_k* for PU_k, to first look for the best new data to load, that is, a data
335 that will maximize the work that can be performed on PU_k.

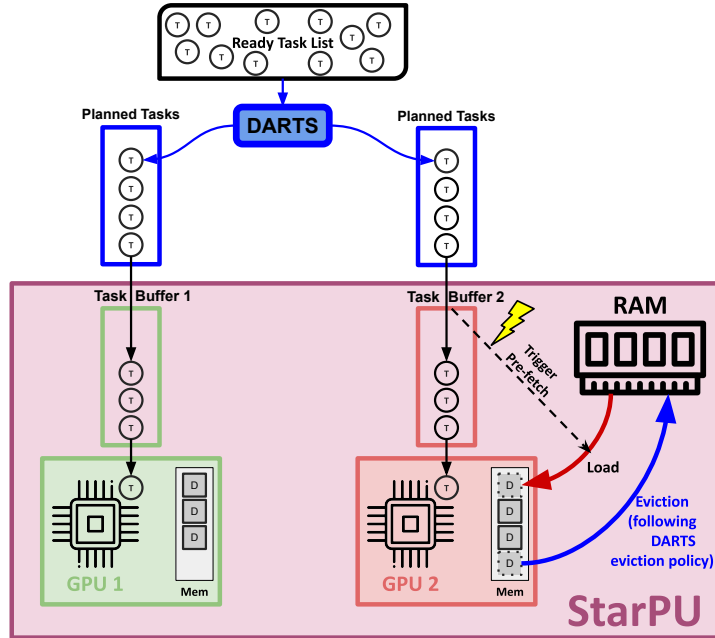


Figure 4: Flow of tasks in STARPU when using DARTS.

336 Our first improvement over the previous DARTS design comes from a complete
 337 re-design of the strategy of selection of the best data to load D_{opt} . We describe
 338 here the values required to select the optimal data D_{opt} (see Algorithm 3).

- 339 • $S_0(D)$: the set of tasks that depend only on D and some data already loaded
 340 in the memory of PU_k .
- 341 • $S_1(D)$: the set of tasks that depend only on D , some data already loaded in
 342 the memory of PU_k , and 1 additional data.
- 343 • $max_prio(D)$: the highest priority of a task in $S_0(D)$ if $|S_0(D)| > 0$, otherwise
 344 the highest priority of a task in $S_1(D)$.
- 345 • $computation_time(D)$: the sum of the durations of the tasks in $S_0(D)$.
- 346 • $task_left(D)$: the sum of the durations of the tasks in $readyTasks$ that use D
 347 as an input.
- 348 • $transfer_duration(D)$: time required to load D to PU_k .

349 Whenever some PU_k requests for some task to execute, we first look in the
 350 $dataNotInMem_k$ set (which initially contains all data) for the optimal data D_{opt}
 351 that, if moved into the memory of PU_k , would minimize the ratio between the
 352 time required to load D_{opt} on PU_k and the computation time of tasks that be-
 353 come “free”, i.e., tasks that can be assigned and processed on PU_k without any
 354 additional data movement. Considering the transfer duration favors direct com-
 355 munication between GPUs (using NVLinks), which is much faster than using
 356 the PCIe bus. It is common for several data to have a similar ratio. This is es-
 357 pecially the case when there is no data to enable “free” tasks. As a result, all
 358 data will have the same infinite ratio. To break ties, we first favor the data with
 359 the largest number of tasks in $S_0(D)$. Then, we break remaining ties by choos-
 360 ing the data that enables the computation of the highest-priority task T with one
 361 additional data load. In such a case, T depends on D_{opt} , on another data D not
 362 in memory, and possibly other data already in memory. Lastly, if we still have
 363 multiple candidate data for D_{opt} , we choose the data used by the task associated
 364 with the most remaining work by looking at both $S_1(D)$ and $task_left(D)$. It is
 365 preferable to load data that will be used by many subsequent tasks, even if it does
 366 not bring locality immediately, as it has the potential to bring more opportunity
 367 for data reuse for subsequent tasks. Once D_{opt} is found, all the corresponding
 368 “free” tasks are assigned to $plannedTasks_k$.

369 The support of task graphs with dependencies requires supporting the dy-
 370 namic addition of tasks in $readyTasks$ when their dependencies are resolved.
 371 When such addition occurs, we dynamically update each $dataNotInMem_k$, to
 372 include data used by this new ready task but which was not used by any other
 373 ready task, and not loaded or planned for load on PU_k . The $dataNotInMem_k$ sets
 374 thus always contain exactly the set of data used by tasks that can be started (ei-
 375 ther in $readyTasks$, in some $plannedTasks_k$, or in some $taskBuffer_k$), which are
 376 not yet loaded on PU_k .

377 Besides, for some newly-released ready tasks, we can bypass $readyTasks$ and
 378 directly push them to $plannedTasks_k$ when they are already “free”. Since we
 379 know which tasks have been queued to each $taskBuffer_k$ and $plannedTasks_k$, we
 380 know the next data loading operations performed on PU_k . When a new task
 381 becomes ready, we can check if it will be free on PU_k , i.e., if it can be already be
 382 processed by some PU_k without any additional data load (because its inputs are
 383 already loaded or are waiting to be loaded). In such a case, we directly assign
 384 the new ready task to the corresponding $plannedTasks_k$. If several PUs qualify,
 385 we first consider the one with the fewest-queued tasks, to balance the load.

Algorithm 3 DARTS scheduler on PU_k

When PU_k requests a new task

- 1: **if** $plannedTasks_k = \emptyset$ **then** ▷ We need to fill $plannedTasks_k$
- 2: **for each** data $D \in dataNotInMem_k$ **do**
- 3: Compute $S_0(D), S_1(D), max_prio(D),$
- 4: $computation_time(D), task_left(D)$ and $transfer_duration(D)$
- 5: Compute $ratio = \frac{transfer_duration(D)}{computation_time(D)}$
- 6: Choose the data D_{opt} with the smallest $ratio$, tiebreak in this order with $|S_0(D)|, max_prio(D), |S_1(D)|$ and $task_left(D)$
- 7: **if** $|S_0(D_{opt})| > 0$ **then**
- 8: Append $S_0(D_{opt})$ to $plannedTasks_k$
- 9: Remove D_{opt} from $dataNotInMem_k$
- 10: **else if** $|S_1(D_{opt})| > 0$ **then**
- 11: Choose task T with highest priority from $S_1(D_{opt})$
- 12: Append T to $plannedTasks_k$
- 13: Remove the inputs of T from $dataNotInMem_k$
- 14: **else**
- 15: Choose task T with highest priority from $readyTasks$
- 16: Append T to $plannedTasks_k$
- 17: Remove the inputs of T from $dataNotInMem_k$
- 18: Return head of $plannedTasks_k$

386 5.4. Eviction policy

387 In order to perform under memory constraints, DARTS needs a custom evic-
388 tion policy, that matches its strategy. The goal is to keep “free” tasks in *taskBuffer*
389 and *plannedTasks* so as not to contradict the task order. The eviction policy con-
390 siders all data currently in memory, and first tries to evict data that is not useful
391 for any task in *taskBuffer_k* (as those tasks are close to being executed on PU_k),
392 and which is an input of few tasks in *plannedTasks_k*. If this is not possible, we
393 apply Belady’s rule [29]: evict the input data whose next usage in *taskBuffer_k* is
394 the furthest in the future.

395 The improved version of DARTS that we propose also brings a novelty to this
396 eviction policy. We update *dataNotInMem_k* for each PU_k after an eviction. This
397 is essential for tasksets with dependencies, as the dataset must be dynamically
398 managed to reduce the computational complexity of finding D_{opt} . If the evicted
399 data is not used by any task in *readyTasks*, we remove it from all *dataNotInMem_k*.
400 It will be added there again when a new ready task with this input becomes
401 available. Otherwise, if any task of *readyTasks* uses the evicted data, we add the
402 data to *dataNotInMem_k* for each PU_k that does not hold it in memory (and is not
403 scheduled to load it). This keeps *dataNotInMem_k* up to date with the data that is
404 currently required by ready tasks.

405 6. Experimental settings

406 We detail below the experimental settings used to evaluate the schedulers
407 discussed above. All schedulers were implemented in the StarPU runtime sys-
408 tems. The code of the schedulers and the applications is available online for
409 reproducibility purpose ¹.

410 6.1. Applications

411 All strategies mentioned above have been implemented in the STARPU run-
412 time system [6], and evaluated both in distributed and shared memory. All sched-
413 ulers use the LRU (Least Recently Used) eviction policy except for DARTS. The
414 schedulers have been tested on two linear algebra applications: the Cholesky fac-
415 torization ($A = L \times L^T$) and the LU factorization ($A = L \times U$) without pivoting.

416 Cholesky and LU are composed of tiled matrix multiplications (GEMM, re-
417 quiring 3 input data), symmetric rank-k update for Cholesky only (SYRK, re-
418 quiring 3 input data), triangular matrix equation (TRSM, requiring 2 input data),

¹https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/JPDC_reproducibility

419 and Cholesky decomposition (POTRF, requiring 1 input data) or LU decompo-
420 sition (GETRF, requiring 1 input data). When the scheduler requires task pri-
421 orities (which is the case for DMDAS, LWS and DARTS), they are computed
422 as the bottom-level of the task in the task graph, which is the minimum time
423 needed from the start of the task to the completion of the whole graph, assuming
424 unbounded resources [30]. The PaRSEC AP scheduler uses its own set of pri-
425 orities (although we noticed no difference when using bottom-level priorities).
426 Although we are running tests on Cholesky and LU factorizations, we made sure
427 that our new DARTS performs as well as the previous version on the applications
428 from our previous study.

429 We measure the obtained performance as the throughput of elementary com-
430 putational operations performed per time unit (in GFlop/s), as well as the total
431 volume of data transferred either between CPU and GPUs and between GPUs
432 themselves in the case of distributed memory with GPUs, or between CPU and
433 disk in the case of shared memory. When measuring the throughput, the cost of
434 computing the schedule is considered. Each point in the following plots is the
435 average of the performance obtained over 10 iterations. For most of the results,
436 the standard deviation is less than 2%, thus we do not show error bars in the
437 following graphs.

438 Although they are separate implementations for flexibility, our applications
439 are identical to those of the Chameleon linear algebra library [31]. In the case
440 of the PaRSEC AP scheduler, we use the DPLASMA implementation of the
441 Cholesky decomposition. We made sure that both the STARPU implementation
442 and the DPLASMA implementation of the Cholesky factorization use the same
443 BLAS kernels with the same tile size.

444 6.2. Machines

445 We ran experiments using cuBLAS 11 single precision GPU kernels. Table 1
446 details the processing units used. We used three different generations of GPUs:
447 Nvidia P100, V100, and A100. Since they were manufactured at very different
448 times, we observe a gradual increase of their memory size, going from 16 GB
449 per GPU to 40 GB. Similarly, NVLinks have various bandwidths, from single
450 10GB/s NVLinks for the P100 to single and dual 24 and 48GB/s NVLinks for
451 the V100 and A100. The A100 has single 24GB/s NVLinks, but compensates
452 with better PCIe bandwidth. The PCIe bus connects the GPU to the NUMA
453 node, which contains the CPU and therefore access to main memory. While the
454 V100 has the same PCIe bus bandwidth as the A100, it only has 2 buses for 4
455 GPUs, which creates possible contention. The A100 has as many buses as there
456 are GPUs, so the full 12GB/s bandwidth is always available.

457 Figure 5 shows the topology of the node containing the V100 GPUs. We can
 458 see that the PCIe buses connecting the GPUs to the NUMA node have a band-
 459 width of 12GB/s for the connection to the first switch, and then 22GB/s for the
 460 link connecting directly to the NUMA node. This last link should theoretically
 461 have a bandwidth of 24 GB/s because it is connected to two 12 GB/s links, but
 462 the values we report in this figure and in Table 1 are the measured performance
 463 on the actual node and not the values reported by the vendor, which explains
 464 this anomaly. We can see that the bandwidth connecting the interconnect to the
 465 RAM is much higher, which makes the PCIe buses the main contention point.
 466 The PCIe buses are also the main bottleneck for the P100 and A100 nodes.

467 Table 1 also details the number of cores of each machine and their associated
 468 peak performance on the GEMM task. The shared memory settings of the CPU
 469 is detailed below in Section 7.3. We optimized the size of tiles used in the linear
 470 algebra computations to reach best peak performance for each architecture: we
 471 use tiles of size 2880 on the V100 and A100 GPUs, and a smaller size of 1920
 472 on the P100.

Table 1: Specifications of processing units used in the experimental evaluation.

	Nvidia P100	Nvidia V100	Nvidia A100	Intel Xeon E5-2620
Manufacturing year	2018	2019	2021	2012
Memory (GB)	16	32	40	32
NVLinks (GB/s)	10	24 or 48	24	/
PCIe Bus Bandwidth (GB/s)	13	12 (with contention)	12	/
Number of cores	3584	5120	6912	6 × 2 CPUs
Maximum GFlop/s on GEMM	6 700	14 000	18 000	15

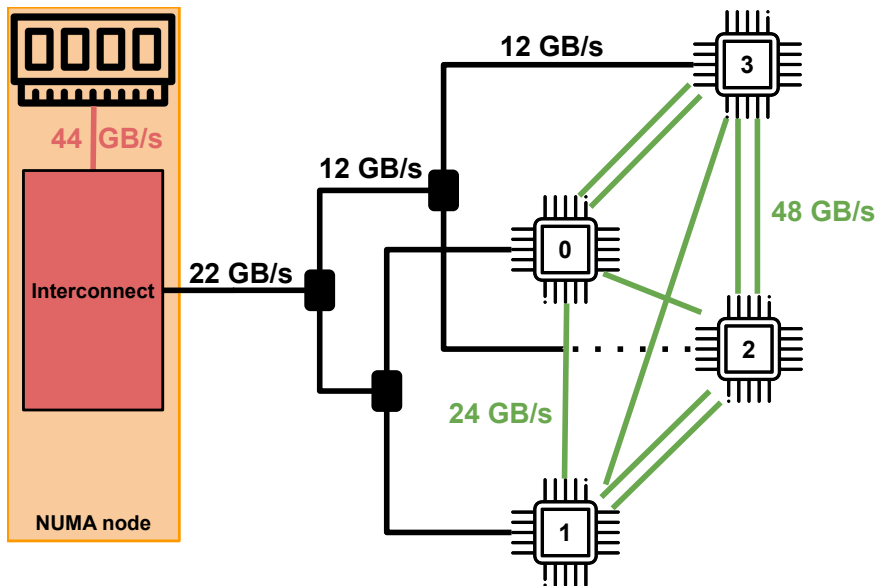


Figure 5: Topology of the node hosting the Nvidia V100 GPUs. The numbered processing units are V100 GPUs. The green lines are NVLinks, the black ones are PCIe busses. The bandwidths shown were measured on the actual node used for our experiments.

473 7. Experimental evaluation

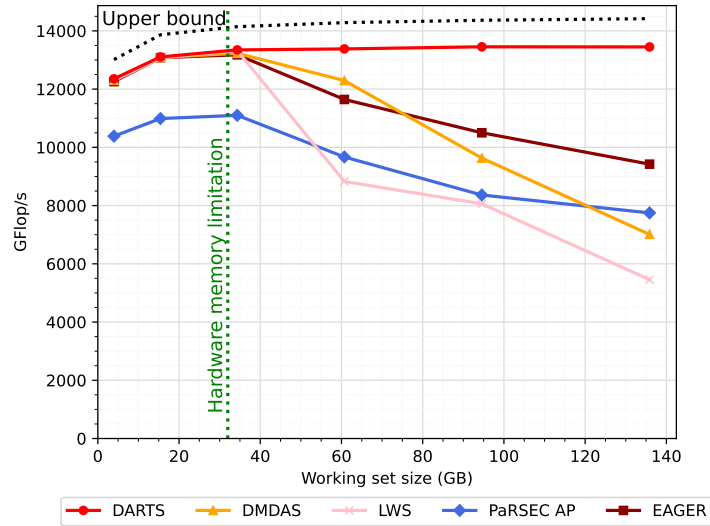
474 We present below the experimental evaluation conducted to compare the
 475 strategies presented above. We first detail results of the Cholesky factorization
 476 on a single V100 GPU. Then, we present results using three different types of
 477 GPUs: V100, A100, and P100, with and without memory constraint. Next, we
 478 discuss the results of LU decomposition on four GPUs, followed by the complete
 479 results with the three different GPUs. Finally, we discuss the results of the CPU
 480 experiments.

481 7.1. Cholesky factorization with GPUs

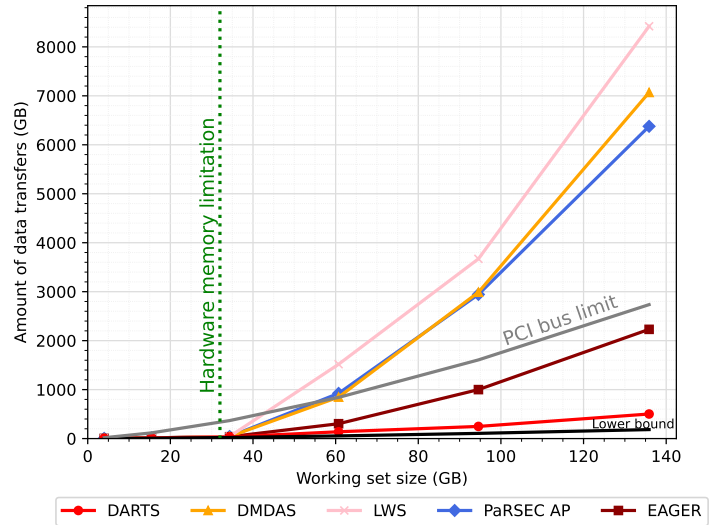
482 We present here a series of evaluations performed on GPUs using the Cholesky
 483 factorization.

484 7.1.1. Overview

485 Figure 6 shows the results obtained by the various algorithms using one V100
 486 GPU. Through calibration, STARPU creates history-based performance models
 487 for each kernel. STARPU also records which tasks are needed to compute an



(a) Performance. The black dotted line corresponds to the theoretical upper bound on performance computed by STARPU.



(b) Amount of data transfers from the CPU to the GPUs. The solid black line corresponds to the lower bound of communications as proven by [32]. The solid gray line is calculated from the amount of data to be transferred and the bandwidth of the PCIe bus, so it corresponds to the amount of data transfers that can theoretically be overlapped with computations.

Figure 6: Results on the Cholesky factorization with 1 Tesla V100 GPU. We vary the size of the working set on the x-axis by increasing the number of blocks of the tiled Cholesky factorization. For reference, the first point corresponds to a 15×15 matrix, while the last point corresponds to a 90×90 matrix. The green dotted line corresponds to the memory size of the V100 GPU.

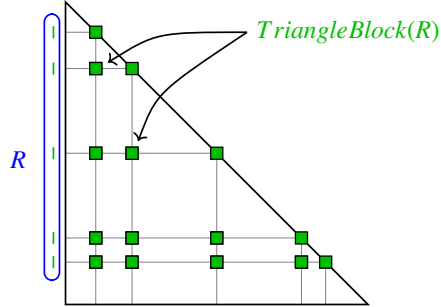


Figure 7: Representation of the tiles computed by a triangle block in an iteration of the communication optimal algorithm from [32]. R is a set of TRSM results that have already been computed. $\text{TriangleBlock}(R)$ is the corresponding set of tasks to be processed to minimize IOs. Figure derived from figures in [32].

488 application. Using the kernel’s performance model on the various available pro-
 489 cessing units and the application’s task set, it computes through linear program-
 490 ming a theoretical lower bound on the execution time where dependencies are
 491 disregarded [33]. This bound is thus not always achievable but provides an in-
 492 teresting asymptotic goal: we try to make the execution as parallel as possible
 493 to get close to the bound. We plot on Figure 6a the theoretical upper bound on
 494 performance simply derived from the theoretical lower bound on the execution
 495 time.

496 The hard memory limit is marked with a green dotted line. We can also see
 497 on Figure 6b, with the solid gray line, the PCIe bus limit: a strategy exceeding
 498 this amount necessarily requires more time for the data transfers than the optimal
 499 time for computation. Figure 6b also depicts, with a solid black line, the lower
 500 bound on the communication volume required by the Cholesky factorization of
 501 an $N \times N$ symmetric matrix, as proven by Beaumont et al. [32].

502 7.1.2. Optimal data access pattern

503 The authors of [32] prove (i) that it is not possible to have fewer IOs than
 504 the previous lower bound, and (ii) that there exists a sequential algorithm that
 505 matches such a result.

506 Figure 7 is an example of how the optimal schedule processes tasks on a
 507 given processing unit. The intuition is to partition the results of the TRSM tasks
 508 across multiple processing units and then compute as many SYRK and GEMM
 509 tasks as possible with such results. In Figure 7, we see on the left side, in blue,
 510 the results of TRSM operations that have already been computed on this process-
 511 ing unit. A node that has such results loaded into memory can compute the tiles

512 in green (which are SYRKs or GEMMs) with a minimal amount of IOs. The
513 communication-optimal algorithm would compute such *triangle blocks* associ-
514 ated with TRSM results already loaded on the GPU. Each GPU would compute
515 tiles associated with different TRSM results, thus not replicating the data. Fi-
516 nally, such triangle blocks must be computed over multiple iterations. A sched-
517 uler that demonstrates a data access pattern in a triangular block that effectively
518 reuses TRSM results across multiple iterations has the potential to significantly
519 reduce IOs.

520 7.1.3. Single GPU case

521 From Figure 6a, we can see that all schedulers except DARTS greatly suffer
522 from the limited memory of the GPU, as their performance plummets once the
523 memory becomes a constraint (symbolized by the green line). With only one
524 GPU, EAGER and LWS process tasks in their submission order. For EAGER,
525 this results in poor progress on the critical path, which explains the lower per-
526 formance. LWS has a large number of data loads, as shown in Figure 6b. LWS
527 suffers from a pathological case of the LRU eviction policy. It loads as much
528 data as possible into the GPU’s memory and processes tasks of the Cholesky
529 factorization one row of the matrix after the other. When the memory is full, it
530 evicts the least recently used data. For a large matrix, the evicted data will still
531 be used by subsequent tasks. This results in almost no data reuse, which leads to
532 a lot of data transfers, creating a performance bottleneck.

533 DMDAS has more sustained performance on the 60 GB point but is far from
534 the asymptotic goal on larger working set sizes. DMDAS does not reassign
535 tasks according to the new data loaded on the GPU because it does not have
536 a global view of the set of data and tasks, and thus cannot strike a balance be-
537 tween prefetching and eviction. The visualization of Figure 8 helps in under-
538 standing these results. This corresponds to a similar situation from the last point
539 of Figure 6a. Looking at the shading and color order, we can see that DMDAS
540 processes tasks along anti-diagonals (because of task priorities). As an example,
541 we can focus on the 1000 blue tasks. The first tasks to be processed are on the
542 first upper left anti diagonal of iteration 1, then on iterations 2, 3, and 4. Then the
543 tasks from the second blue diagonal of iteration 1 are computed, then those of it-
544 eration 2, 3 and 4. The tasks from the first anti diagonals have data affinity across
545 the different iterations. However, the affinity on the rows and columns within an
546 iteration can only be exploited if the memory can hold all the data needed for
547 the first anti diagonals of each iteration. In a very memory-constrained case,
548 after computing the first anti diagonal of each iteration, the GPU’s memory is
549 full. Thus, when the second blue anti diagonal is computed, all possible data that

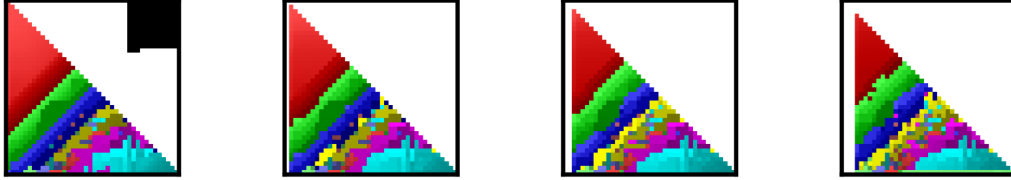


Figure 8: DMDAS's ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU, with $N = 40$. Each task is depicted by a colored squared at its position in the matrix being factorized. The first 1000 processed tasks are in red, the next 1000 in green, then blue, yellow, magenta, cyan and orange. The shading within each color represents the processing order. The black area represents the amount of tiles that can be loaded in memory. The GPU's memory is limited to 2 GB for the visualization in order to mimic a very memory-constrained situation similar to the last point of Figure 6a (but with reduced matrix size).

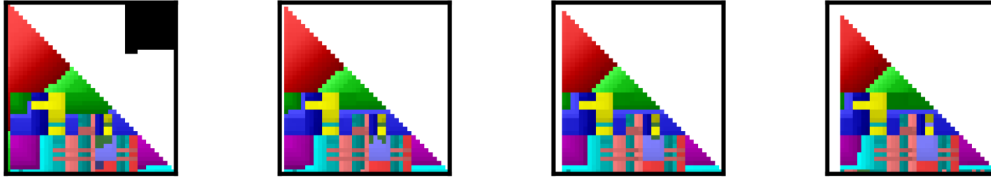


Figure 9: DARTS's ordering on iterations 1 to 4 of the Cholesky factorization with 1 Tesla V100 GPU, with $N = 40$. Same settings and visualization choices as Figure 8.

550 could have been reused has been evicted, resulting in multiple loads of the same
 551 rows or columns. This results in more data being transferred, as it can be seen in
 552 the Figure 6b. ParSEC AP schedules tasks similarly to DMDAS, as it uses the
 553 expected completion time of tasks and sorts them by priority.

554 DARTS maintains good performance with an increasing working set. DARTS
 555 also processes tasks in a diagonal while it can fit in memory (tasks in red on
 556 Figure 9) but then switches to what we call *DARTS triangle blocks*. We can
 557 notice such triangle blocks on the blue and magenta tasks (a triangle on the
 558 main diagonal preceded by a square of tasks of the same color on the first few
 559 columns). Note that DARTS triangle blocks are different from the optimal tri-
 560 angle blocks presented earlier because they do not necessarily group tasks from
 561 different zones, thus not forming a complete triangle. However, they have the
 562 benefit of reusing TRSM results to compute a maximum amount of tasks. Those
 563 triangles and the associated squares exactly fit in memory, share a lot of common
 564 data and are replicated over multiple iterations (up to $k = 8$) in order to progress
 565 on the critical path, while maximizing data reuse. DARTS forms these triangles
 566 because it can contradict priority with locality, as explained in Section 5.1. There

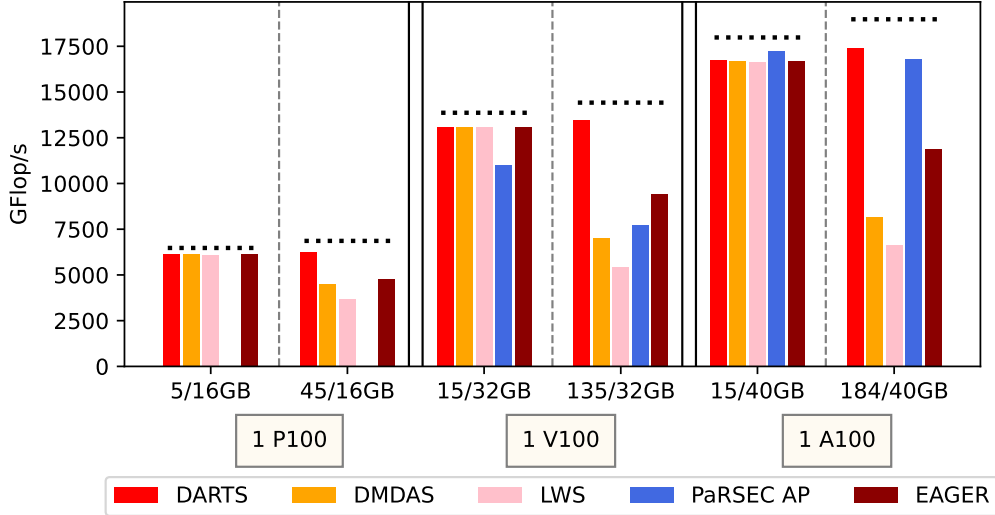


Figure 10: Performance with a single GPU on the Cholesky factorization. Each subfigure is a different experiment with a different GPU. The x-axis shows the working set size compared to the total memory of the GPU (e.g., 5/16GB denotes an experiment with a working set size of 5GB using a 16GB on-device memory). The solid vertical lines separate the experiments with different GPUs. The first half of each subfigure is a scenario where the working set size is smaller than the GPU memory. The second half is a memory-constrained situation. The black dotted line represents the theoretical upper bound on performance.

567 is enough parallelism to feed all the workers without having to progress on the
 568 critical path. Thus, DARTS focuses on data reuse with its strategy instead of task
 569 priority, which creates these triangle blocks on the visualization. As explained
 570 earlier, accessing data with such a pattern allows to reduce communications with
 571 the RAM as shown in Figure 6b: DARTS has the lowest amount of data trans-
 572 fers and is the only strategy under the bus limit. It means that theoretically, all
 573 transfers can be overlapped by a computation, which explains DARTS good per-
 574 formance.

575 7.1.4. Results with different types of GPU

576 Figure 10 illustrates the performance of the different schedulers across dif-
 577 ferent GPUs (P100, V100, and A100) and memory constraints. The experiments
 578 with P100 GPUs do not show ParSEC AP results: The P100 is a relatively slow
 579 GPU compared to its associated CPUs on the node, so ParSEC AP used CPUs
 580 in addition to GPUs for this experiment. Since we could not disable the CPUs,
 581 it was not a homogeneous configuration, so we could not compare the results.

582 Figure 10 illustrates that, for experiments where the full workload fits in

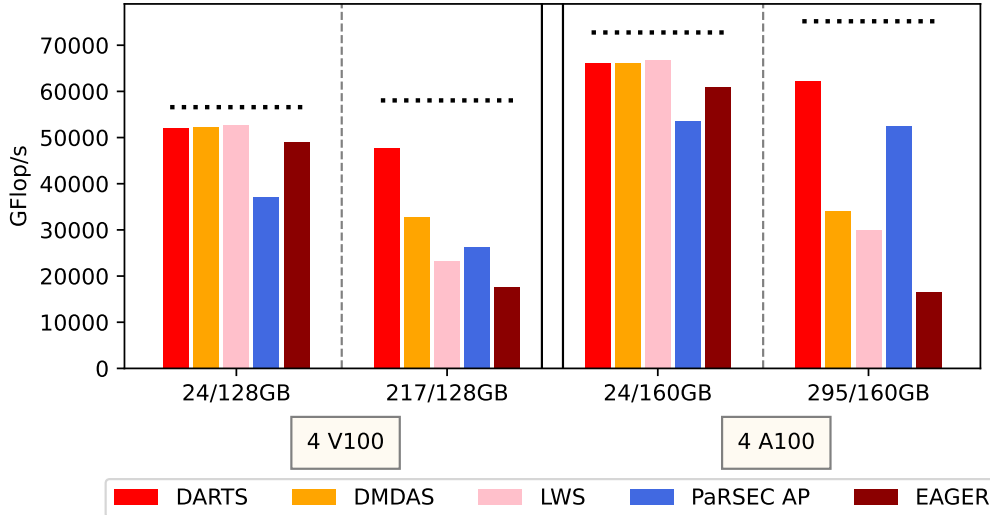


Figure 11: Performance with four GPUs on the Cholesky factorization. Same settings as Figure 10 excepts the memory size now considers the cumulated memory of all GPUs.

583 memory, the majority of schedulers yield comparable outcomes. This is to be ex-
 584 pected, since even the order in which tasks are submitted can lead to near-optimal
 585 performance as long as transfers are overlapped with computation, which is the
 586 case for all strategies here. For experiments with a working set size that exceeds
 587 the GPU memory, DARTS is proven to be the best strategy because it can reduce
 588 data transfers, similar to Figure 6b. However, on the A100 GPU, ParSEC AP
 589 and DARTS are almost equivalent, even after the memory constraint. This is due
 590 to two differences from the other compute nodes. First, the A100 GPU has more
 591 memory than the other GPUs. Second, the compute node containing the A100
 592 has as many PCI buses as there are GPUs. As a result, there is no contention on
 593 the buses to transfer data between the CPU and the GPUs, unlike on the compute
 594 node containing the V100 GPUs. (see Table 1). With these two improvements
 595 compared to the V100s, the benefit of data locality is reduced because transfers
 596 can be done quickly and the GPU’s memory can hold more data, leaving less
 597 opportunity for situations like the pathological case LWS suffered in Figure 6b.
 598 However, on the V100, with less memory and more contention, DARTS is better
 599 than ParSEC AP.

600 7.1.5. Results with multiple GPUs

601 Figure 11 illustrates the performance with four GPUs, while Figure 12 is the
 602 same experiment but shows the amount of data transfers. In Figure 12, we con-

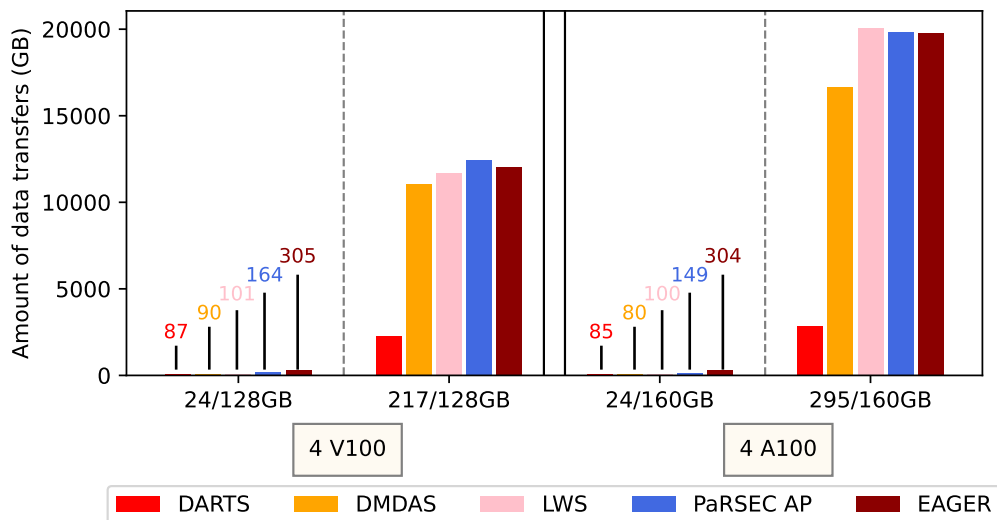


Figure 12: Amount of data transfers from CPU to the four GPUs and between GPUs on the Cholesky factorization for the experiments of Figure 11.

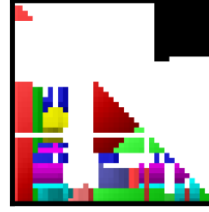
603 consider the transfers from CPUs to GPUs, as well as the transfers between GPUs
 604 via NVLinks. The memory size now refers to the cumulative memory of the four
 605 GPUs. The results cannot be displayed on the P100 GPUs because we did not
 606 have access to a machine with more than 2 P100 GPUs.

607 The performance gap between DARTS and the other strategies is more pro-
 608 nounced with four GPUs than with a single GPU configuration. This is partic-
 609 ularly clear when compared to ParSEC AP and EAGER. With multiple GPUs,
 610 some data must be replicated to different GPU memories, which greatly increases
 611 the total amount of data transfers, but it is necessary to increase parallelization.
 612 Normally this is not a problem, but with a memory constraint this replication
 613 greatly reduces performance because each GPU will suffer the same performance
 614 loss as seen with a single GPU. We can see that the amount of transfers on Fig-
 615 ure 12 is very important for the other strategies compared to DARTS.

616 For small matrices that fit in the cumulative GPU memory, we see in Fig-
 617 ure 11 that LWS achieves the best performance for both V100s and A100s by a
 618 small margin, but still notable for such small workloads. By stealing work from
 619 neighboring GPUs, LWS is able to greatly increase transfers using NVLinks,
 620 which are much faster than transfers using CPU memory. With four GPUs,
 621 the opportunity for such transfers is much more important, which explains this
 622 performance. This explains why in Figure 12, even with more transfers than
 623 DARTS, LWS achieves slightly better throughput on the experiments where the



(a) Task processed by GPU 1.



(b) Task processed by GPU 2.

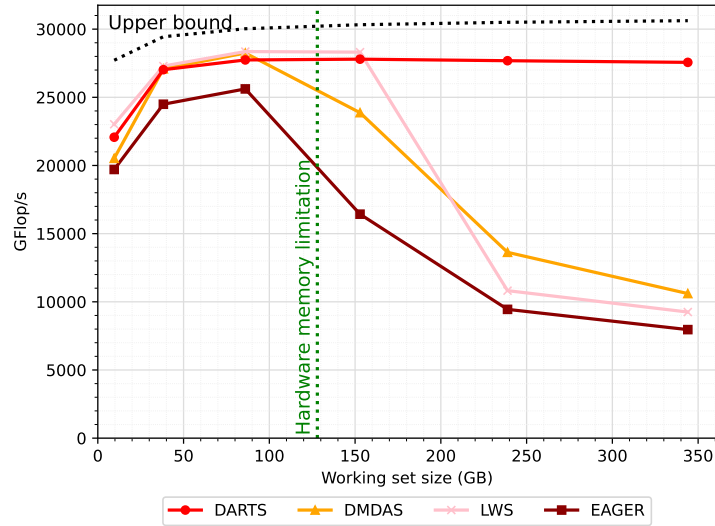
Figure 13: Same settings and visualization choices as Figure 8.

624 dataset fits in GPU memory. On both V100s and A100s, for larger matrices,
 625 LWS and EAGER have much more data transfers and thus much lower through-
 626 put. They do not take memory constraints into account and therefore do not
 627 favor data reuse. Similarly for ParSEC AP and DMDAS, the low throughput
 628 on the V100s and A100s experiment is associated with high transfer rates on
 629 Figure 12, as in the case of the single GPU experiment. Whenever memory is
 630 limited, DARTS consistently achieves the best performance.

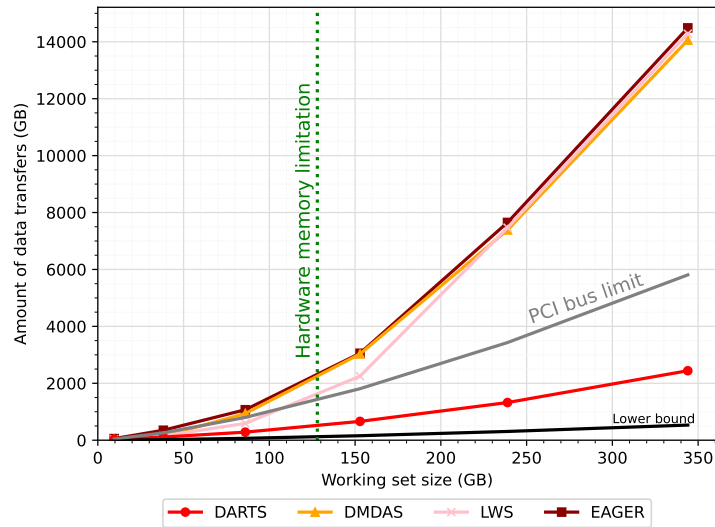
631 Figure 13 represents the task order with two V100 (for clarity) using DARTS
 632 on the first iteration of the outer-loop of the Cholesky algorithm. We can still
 633 find those *triangle blocks* that allows us to reduce data transfers, as can be seen
 634 on Figure 12. Moreover, there is a good distribution of the data load on the
 635 2 GPUs. When choosing the data to load D_{opt} , DARTS uses a ratio between
 636 transfer time and computation time of task using D_{opt} . This favors the selection of
 637 a data associated with as many unprocessed tasks as possible. Thus two distinct
 638 processing units have a low probability to select the same data as the total work
 639 associated with a data is reduced after some of its task is scheduled on another
 640 processing unit. This encourages GPUs to work on different datasets, further
 641 reducing the total amount of data to load by minimizing the replication of data
 642 on multiple GPUs. The sustained performance of DARTS with both one and four
 643 GPUs shows that it is generic enough to adapt to different numbers of processing
 644 units.

645 7.2. LU factorization with GPUs

646 We discuss here the results obtained on the LU factorization. Figures 14 de-
 647 picts the performance and amount of data transfers on the LU factorization with
 648 four GPUs. It should be noted that we do not present results for ParSEC AP as
 649 the DPLASMA implementation of LU does not make use of the cuSolver library
 650 to compute GETRF kernels of the LU factorization: ParSEC AP uses a much
 651 slower version of this kernel, and thus a fair comparison with other schedulers



(a) Performance. The black dotted line corresponds to the theoretical upper bound on performance computed by STARPU.



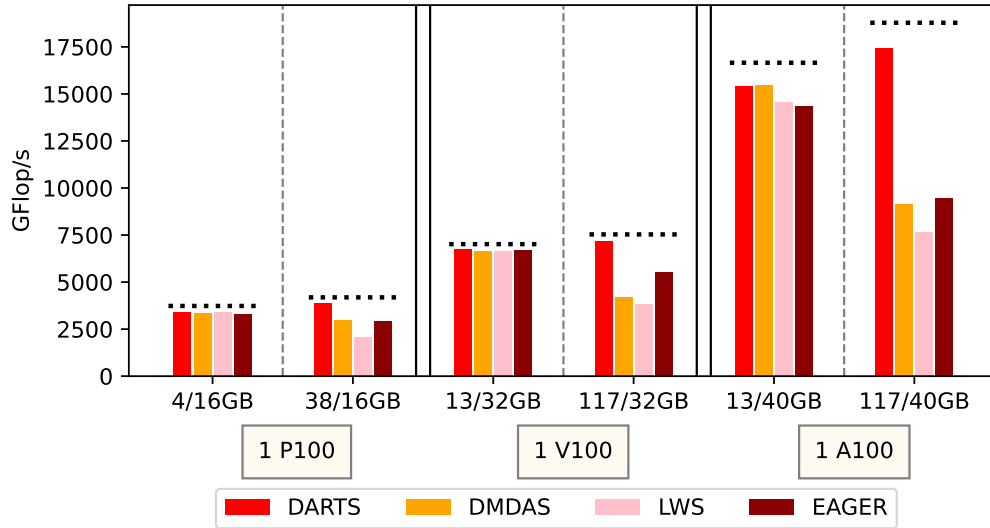
(b) Amount of data transfers. The solid black line corresponds to the lower bound of communications as proven by [34]. The solid gray line is derived from the bandwidth of the PCIe bus and the working set size and represents the theoretical maximum amount of data transfers that can be overlapped with computations.

Figure 14: Results on the LU factorization with four Tesla V100 GPUs. We vary the size of the working set on the x-axis by increasing the number of blocks of the tiled LU factorization. For reference, the first point corresponds to a 12×12 matrix, while the last point corresponds to a 72×72 matrix. The green dotted line corresponds to the cumulated memory of the V100 GPUs.

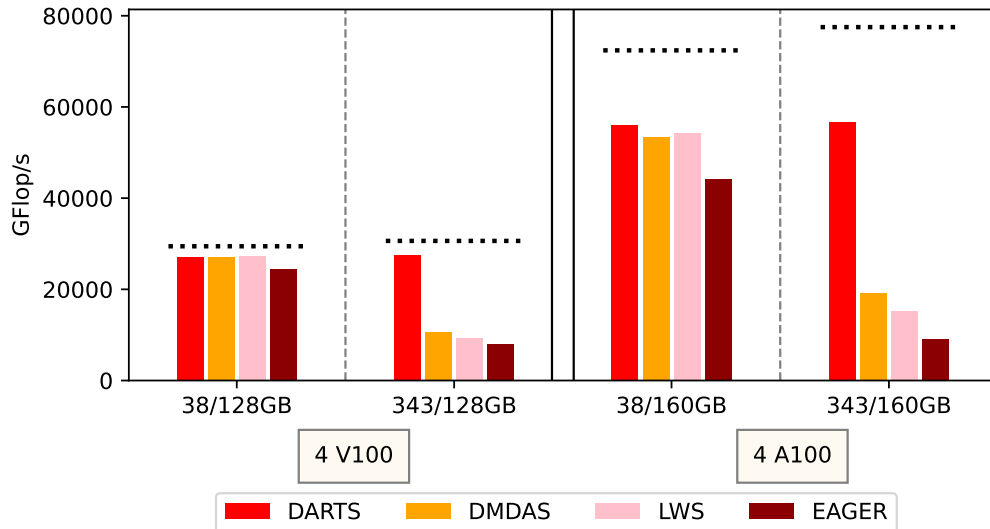
652 is not possible. The EAGER scheduler processes tasks in their submission or-
653 der, while LWS sorts them by priorities. In both case, a tile loaded for a given
654 computation cannot be reused on consecutive tasks when memory is limited.
655 Consequently, these two schedulers end up with at least three times more data
656 transfers than DARTS (see Figure 14b) once the memory becomes a constraint
657 (after the green vertical line). DMDAS suffers from the same issue as with the
658 Cholesky factorization: it is unable to balance prefetch and evictions, which re-
659 sults in a considerable amount of data transfers. DARTS exhibits more sustained
660 performance: as is the Cholesky case, it is able to reduce data transfers by dis-
661 tributing the data on multiple GPUs and reusing data for consecutive tasks. This
662 reaches 85% of the peak achievable performance, even with working sets twice
663 as large as the cumulated GPU memory.

664 The LU and Cholesky factorizations are very closely related computations:
665 LU can be considered a "double Cholesky". Because of the symmetry of the
666 matrix in the case of Cholesky, LU requires twice as much data as Cholesky, but
667 also includes twice as many computational tasks. Hence, both applications have
668 the same communication-to-computation ratio. Similar data reuse patterns can
669 be found for Cholesky and LU: building square blocks of GEMMs increases data
670 reuse. For Cholesky, however, one must exploit the symmetry of the matrix to
671 increase data reuse, which results in more complicated patterns. Such a solution
672 is much harder to create with a dynamic scheduler. Given that DARTS is able
673 to produce good data locality patterns on Cholesky, it is not surprising that it is
674 able to reduce even more data transfers with LU. Figure 14b plots the minimum
675 amount of IO required by LU factorization, as computed by Olivry et al. [34,
676 Table 2], and we see that DARTS can get close to the minimum amount of IOs:
677 DARTS has 4.8 times more data transfers than the lower bound and has 2.4 times
678 less transfers than the PCIe bus limit. As a result, it is able to completely overlap
679 communications with computations, resulting in near-optimal performance with
680 one or multiple GPUs. This explains why the performance difference with other
681 schedulers is more significant in the LU case than in the Cholesky case.

682 Figures 15a and 15b illustrate the outcome of the experiment with the differ-
683 ent GPUs. As previously observed, when the working set is smaller than the cu-
684 mulated memory of GPUs, DARTS yields comparable results to the other sched-
685 ulers. This is a satisfactory result, as it demonstrates that DARTS, despite its
686 initial design to address memory-limited scenarios, is capable of achieving com-
687 parable performance to state-of-the-art solutions in more traditional cases. When
688 the application dataset is significantly larger than the cumulated GPU memory,
689 DARTS largely outperforms other schedulers. Again, compared to the results on



(a) Performance with a single GPU on the LU factorization.



(b) Performance with four GPUs on the LU factorization.

Figure 15: Performance achieved with one or several GPUs on the LU factorization. X/YGB considers a scenario with a working set of size X GB for a (cumulative) GPU memory of Y GB. For each hardware setting, the left part presents a situation where the working set size is smaller than the GPU memory. The second half is a memory-constrained situation. The black dotted black represents the theoretical upper bound on performance.

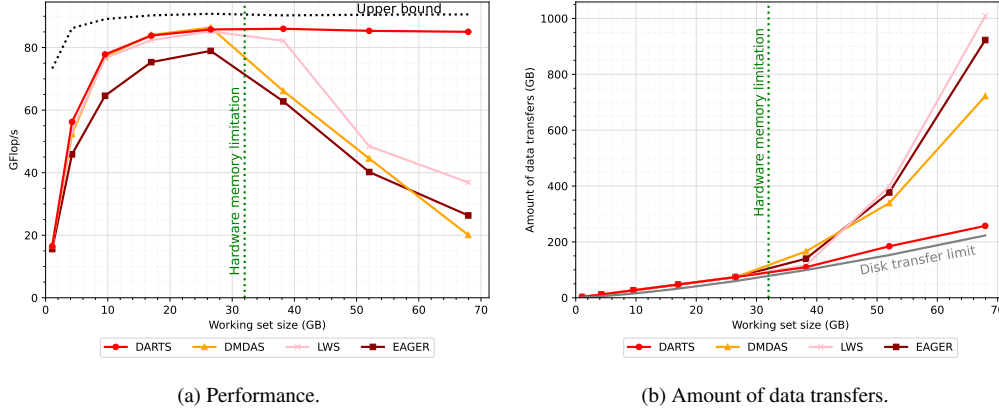


Figure 16: Results of the LU factorization with two Intel Xeon E5-2620 6-cores CPUs. Shared memory of 32 GB.

690 Cholesky, we observe that the difference between DARTS and the other sched-
 691 ulers is more significant.

692 7.3. LU factorization on a multi-core CPU

693 Our scheduler is able to consider any memory-limited system, and STARPU
 694 manages disk-CPU transfers as well as CPU-GPU transfers, which enables us
 695 to perform the experiments on CPUs. Figure 16 reports experiments on two Intel
 696 Xeon E5-2620 6 cores CPUs, with 32 GB of shared memory, and disks that
 697 sustain an approximately 94 MB/s bandwidth. Each core may need to transfer
 698 data from disk at the same time, placing a high demand on a limited bandwidth.
 699 One may argue that this particular CPU is slower and carries a smaller mem-
 700 ory than most standard CPUs. However, its 32 GB and 12-core configuration
 701 provides 2.6 GB of memory per core, which is a reasonable allocation in com-
 702 parison to contemporary CPUs that offer more memory but also a greater num-
 703 ber of cores. Hence, it allows us to run experiments of reasonable durations on
 704 a hardware setting with realistic memory-per-CPU ratio. The DARTS strategy,
 705 which is specifically designed to reduce data transfers, is used in this architecture
 706 to demonstrate that near-optimal performance can be achieved with a relatively
 707 modest memory size in comparison to the data input. The application scenario
 708 is the LU factorization and we use tiles of size 2880.

709 In this shared-memory setting, DARTS behaves in a similar way as it does
 710 with one GPU: it uses a single *plannedTasks* queue as well as a single *taskBuffer*.
 711 On both experiments, when the whole dataset fits in memory, DARTS has com-
 712 parable performance with DMDAS and LWS. For datasets larger than the avail-

713 able shared memory, DARTS is the only strategy able to obtain close-to-optimal
714 performance, whereas all other schedulers suffer from a large performance drop.
715 In terms of data transfers shown in Figure 16b, DARTS is able to stay close to
716 the disk bandwidth limit, symbolized by the gray curve. Thus, DARTS has time
717 to overlap most communications with computations, explaining its good results.

718 *7.4. Summary of the experimental evaluation*

719 The results demonstrated that when the cumulated memory of the processing
720 units is larger than the size of the application’s input data, DARTS exhibited
721 similar performance to those of state-of-the-art runtime schedulers. Further-
722 more, when the memory is too scarce to load all input data simultaneously,
723 existing schedulers experience large performance drops, while DARTS is able
724 to sustain close-to-peak performance, by scheduling tasks sharing data on the
725 same resource and reusing loaded data for successive tasks. This behaviour
726 has been acknowledged on both Cholesky and LU factorization, which involved
727 complex computation dependencies, on various GPUs and CPU settings. The
728 results demonstrate that DARTS is a generic scheduler that can be used in a di-
729 verse range of applications, memory configurations, and architectures. When the
730 memory is the primary limiting factor, DARTS is capable of achieving near-peak
731 performance.

732 **8. Conclusion**

733 In this paper, we have focused on the problem of scheduling tasks in run-
734 time systems to handle very large datasets that do not fit into the memory of the
735 processing units. We have largely improved a scheduler for the STARPU run-
736 time system, named DARTS, to cope with task graphs. Our scheduler is mainly
737 focused on data locality but also takes task priorities into account. We have per-
738 form experiments with two classical linear algebra operations: Cholesky and LU
739 factorizations. Thanks to its modularity, DARTS reaches very good performance
740 in a large variety of situations, from multicore CPU with shared memory to mul-
741 tiple GPUs with distributed memory. Among available competitors, DARTS is
742 the only scheduler to reach good performance in memory constrained scenar-
743 ios, and it also ranks among the best ones in all settings. Future works include
744 taking advantage of the modularity of DARTS to design hierarchical schedulers
745 for large distributed platforms, and to combine it with other schedulers (such as
746 work stealing) to benefit from all their features, as (locality-aware) work steal-
747 ing is efficient to balance load among distributed nodes, while DARTS allows to
748 cope with limited memories.

749 Acknowledgments

750 This work was supported by the SOLHARIS project (ANR-19-CE46-0009)
751 which is operated by the French National Research Agency (ANR).

752 Experiments presented in this paper were carried out using the Grid'5000
753 testbed, supported by a scientific interest group hosted by Inria and including
754 CNRS, RENATER and several Universities as well as other organizations (see
755 <https://www.grid5000.fr>).

756 We thank Lionel Eyraud-Dubois for his help in designing Figure 7 similar to
757 his work in [32] and for his insightful feedback.

758 References

- 759 [1] N. R. Council, et al., Getting up to speed: The future of supercomputing, National
760 Academies Press, 2005.
- 761 [2] Q. Cheng, M. Glick, K. Bergman, Chapter 20 - Optical interconnection networks for
762 high-performance systems, in: A. E. Willner (Ed.), Optical Fiber Telecommunications VII,
763 Academic Press, 2020. doi:<https://doi.org/10.1016/B978-0-12-816502-7.00020-8>.
764 URL [https://www.sciencedirect.com/science/article/pii/
765 B9780128165027000208](https://www.sciencedirect.com/science/article/pii/B9780128165027000208)
- 766 [3] E. Agullo, A. Guermouche, J.-Y. L'Excellent, A parallel out-of-core multifrontal method:
767 Storage of factors on disk and analysis of models for an out-of-core active memory, *Parallel*
768 *Computing* 34 (6) (2008) 296–317. doi:<https://doi.org/10.1016/j.parco.2008.03.007>.
769 URL [https://www.sciencedirect.com/science/article/pii/
770 S0167819108000495](https://www.sciencedirect.com/science/article/pii/S0167819108000495)
- 771 [4] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, J. Labarta, Produc-
772 tive programming of GPU clusters with OmpSs, in: International Parallel and Distributed
773 Processing Symposium, 2012.
- 774 [5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J. Dongarra, PaRSEC: A
775 programming paradigm exploiting heterogeneity for enhancing scalability, *Computing in*
776 *Science and Engineering* 15 (6) (2013).
- 777 [6] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task
778 Scheduling on Heterogeneous Multicore Architectures, *Concurrency and Computation:
779 Practice and Experience*, Special Issue: Euro-Par 2009 23 (2011). doi:10.1002/cpe.1631.
- 780 [7] M. Gonthier, L. Marchal, S. Thibault, Memory-Aware Scheduling of Tasks Sharing Data on
781 Multiple GPUs with Dynamic Runtime Systems, in: IPDPS 2022 - 36th IEEE International
782 Parallel and Distributed Processing Symposium, IEEE, Lyon, France, 2022, pp. 1–11.
783 URL <https://hal.inria.fr/hal-03552243>
- 784 [8] P. J. Denning, The working set model for program behavior, *Communications of the ACM*
785 11 (5) (1968) 323–333.
- 786 [9] S. Toledo, A survey of out-of-core algorithms in numerical linear algebra, in: *External*
787 *Memory Algorithms and Visualization*, American Mathematical Society Press, 1999, pp.
788 161–180.

- 789 [10] J. Demmel, M. Hoemmen, M. Mohiyuddin, K. Yelick, Avoiding communication in sparse
790 matrix computations, in: IEEE International Symposium on Parallel and Distributed Pro-
791 cessing, 2008. doi:10.1109/IPDPS.2008.4536305.
- 792 [11] L. Marchal, S. McCauley, B. Simon, F. Vivien, Minimizing I/Os in Out-of-
793 Core Task Tree Scheduling, International Journal of Foundations of Computer
794 Science 34 (01) (2023) 51–80. arXiv:<https://doi.org/10.1142/S0129054122500186>,
795 doi:10.1142/S0129054122500186.
796 URL <https://doi.org/10.1142/S0129054122500186>
- 797 [12] J. V. Ferreira Lima, T. Gautier, V. Danjean, B. Raffin, N. Maillard, Design and analysis of
798 scheduling strategies for multi-CPU and multi-GPU architectures, Parallel Computing 44
799 (2015) 37–52.
- 800 [13] U. A. Acar, G. E. Blelloch, R. D. Blumofe, The data locality of work stealing, Theory
801 Comput. Syst. 35 (3) (2002) 321–347.
- 802 [14] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and in-
803 dependence with logical regions, in: SC '12: Proceedings of the International Con-
804 ference on High Performance Computing, Networking, Storage and Analysis, 2012.
805 doi:10.1109/SC.2012.71.
- 806 [15] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, Dague:
807 A generic distributed dag engine for high performance computing, Parallel Com-
808 puting 38 (1), extensions for Next-Generation Parallel Programming Models (2012).
809 doi:10.1016/j.parco.2011.10.003.
- 810 [16] M. Cosnard, M. Loi, Automatic task graph generation techniques, in: Proceedings of the
811 Twenty-Eighth Annual Hawaii International Conference on System Sciences, Vol. 2, 1995.
812 doi:10.1109/HICSS.1995.375471.
- 813 [17] H. Lee, W. Ruys, I. Henriksen, A. Peters, Y. Yan, S. Stephens, B. You, H. Fingler,
814 M. Burtscher, M. Gligoric, et al., Parla: A python orchestration system for heterogeneous
815 architectures, in: SC '22: Proceedings of the International Conference on High Perfor-
816 mance Computing, Networking, Storage and Analysis, 2022.
817 URL <https://userweb.cs.txstate.edu/~burtscher/papers/sc22.pdf>
- 818 [18] B. Peccerillo, S. Bartolini, Phast - a portable high-level modern c++ programming library
819 for gpus and multi-cores, IEEE Transactions on Parallel and Distributed Systems 30 (1)
820 (2019) 174–189.
- 821 [19] P. Thoman, P. Salzmann, B. Cosenza, T. Fahringer, Celerity: High-level c++ for accelerator
822 clusters, in: R. Yahyapour (Ed.), Euro-Par 2019: Parallel Processing, Springer International
823 Publishing, Cham, 2019, pp. 291–303.
- 824 [20] M. Gonthier, L. Marchal, S. Thibault, Locality-Aware Scheduling of Independent Tasks
825 for Runtime Systems, in: COLOC - 5th workshop on data locality - 27th International
826 European Conference on Parallel and Distributed Computing, Lisbon, Portugal, 2021, pp.
827 1–12.
- 828 [21] C. Augonnet, J. Clet-Ortega, S. Thibault, R. Namyst, Data-Aware Task Scheduling on
829 Multi-Accelerator based Platforms, in: Int. Conf. on Parallel and Distributed Systems,
830 2010.
- 831 [22] H. Topcuoglu, S. Hariri, M.-Y. Wu, Task scheduling algorithms for heterogeneous pro-
832 cessors, in: Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99), 1999.
833 doi:10.1109/HCW.1999.765092.
- 834 [23] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S. P.

- 835 Thibault, Achieving high performance on supercomputers with a sequential task-based
836 programming model, *IEEE Transactions on Parallel and Distributed Systems* (2017).
837 doi:10.1109/TPDS.2017.2766064.
- 838 [24] T. Gautier, J. V. Ferreira Lima, N. Maillard, B. Raffin, Locality-Aware Work Stealing on
839 Multi-CPU and Multi-GPU Architectures, in: 6th Workshop on Programmability Issues for
840 Heterogeneous Multicores (MULTIPROG), Berlin, Germany, 2013.
841 URL <https://inria.hal.science/hal-00780890>
- 842 [25] A. Robison, M. Voss, A. Kukanov, Optimization via reflection on work stealing in tbb,
843 in: 2008 IEEE International Symposium on Parallel and Distributed Processing, 2008.
844 doi:10.1109/IPDPS.2008.4536188.
- 845 [26] J.-N. Quintin, F. Wagner, Hierarchical work-stealing, in: Euro-Par 2010 - Parallel Process-
846 ing, 2010, pp. 217–229.
- 847 [27] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Lan-
848 gou, P. Lemariner, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra, Flexible development
849 of dense linear algebra algorithms on massively parallel architectures with DPLASMA,
850 in: International Symposium on Parallel and Distributed Processing Workshops and Phd
851 Forum, 2011.
- 852 [28] S. Moustafa, M. Faverge, L. Plagne, P. Ramet, 3D cartesian transport sweep for massively
853 parallel architectures with PARSEC, in: IEEE International Parallel and Distributed Pro-
854 cessing Symposium, 2015, pp. 581–590.
- 855 [29] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, *IBM Sys-
856 tems Journal* 5 (2) (1966).
- 857 [30] C. Alias, S. Thibault, L. Gonnord, A Compiler Algorithm to Guide Runtime Scheduling,
858 Research Report RR-9315, INRIA Grenoble ; INRIA Bordeaux - Sud-Ouest (Dec. 2019).
859 URL <https://hal.inria.fr/hal-02421327>
- 860 [31] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster,
861 Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for
862 GPUs, in: GPU Computing Gems, Vol. 2, Morgan Kaufmann, 2010.
863 URL <https://hal.inria.fr/inria-00547847>
- 864 [32] O. Beaumont, L. Eyraud-Dubois, M. Vérité, J. Langou, I/O-Optimal Algorithms for Sym-
865 metric Linear Algebra Kernels, in: ACM Symposium on Parallelism in Algorithms and
866 Architectures, 2022.
867 URL <https://hal.inria.fr/hal-03580531>
- 868 [33] Theoretical Lower Bound On Execution Time, [https://files.
869 inria.fr/starpu/doc/html/OfflinePerformanceTools.html#
870 TheoreticalLowerBoundOnExecutionTime](https://files.inria.fr/starpu/doc/html/OfflinePerformanceTools.html#TheoreticalLowerBoundOnExecutionTime), accessed: 2024-06-27.
- 871 [34] A. Olivry, J. Langou, L.-N. Pouchet, P. Sadayappan, F. Rastello, Automated derivation of
872 parametric data movement lower bounds for affine programs (2019). arXiv:1911.06664.