Taming data locality for task scheduling under memory constraint in runtime systems

Maxime Gonthier, Loris Marchal, Samuel Thibault

Taming data locality for task scheduling under memory constraint in runtime systems

Maxime GONTHIER[a,*], Loris MARCHAL[b], Samuel THIBAULT[c]

[a]*LIP, ENS de Lyon, LaBRI, Inria, maxime.gonthier@ens-lyon.fr 200 Av. de la Vieille Tour, Talence, 33405, Nouvelle-Aquitaine, France*
[b]*LIP, ENS de Lyon, Inria, CNRS, University Claude-Bernard Lyon 1, loris.marchal@ens-lyon.fr 46 allee d'Italie, Lyon, 69007, Auvergne - Rhone-Alpes, France*
[c]*LaBRI, Inria Bordeaux Sud-Ouest, CNRS, University of Bordeaux, samuel.thibault@u-bordeaux.fr 200 Av. de la Vieille Tour, Talence, 33405, Nouvelle-Aquitaine, France*

Abstract

A now-classical way of meeting the increasing demand for computing speed by HPC applications is the use of GPUs and/or other accelerators. Such accelerators have their own memory, which is usually quite limited, and are connected to the main memory through a bus with bounded bandwidth. Thus, particular care should be devoted to data locality in order to avoid unnecessary data movements. Task-based runtime schedulers have emerged as a convenient and efficient way to use such heterogeneous platforms. When processing an application, the scheduler has the knowledge of all tasks available for processing on a GPU, as well as their input data dependencies. Hence, it is possible to produce a tasks processing order aiming at reducing the total processing time through three objectives: minimizing data transfers, overlapping transfers and computation and optimizing the eviction of previously-loaded data. In this paper, we focus on how to schedule tasks that share some of their input data (but are otherwise independent) on a single GPU. We provide a formal model of the problem, exhibit an optimal eviction strategy, and show that ordering tasks to minimize data movement is NP-complete. We review and adapt existing ordering strategies to this problem, and propose a new one based on task aggregation. We prove that the underlying problem of this new strategy is NP-complete, and prove the reasonable complexity of our proposed heuristic. These strategies have been implemented in the StarPU runtime system. We present their performance on tasks from tiled 2D, 3D matrix products, Cholesky factorization, randomized task order, randomized data pairs from the 2D matrix product as well as a sparse matrix product. We introduce a visual way to understand these performance and lower bounds on the number of data loads for the 2D and 3D matrix products. Our experiments demonstrate that using our new strategy together with the optimal eviction policy reduces the amount of data movement as well as the total processing time.

Keywords: Memory-aware scheduling, Eviction policy, Tasks sharing data, GPUs, Runtime systems, Memory constraint.

1. Introduction

High-performance computing applications, such as physical simulations, molecular modeling or weather and climate forecasting, have an increasing demand in computer power to reach better accuracy. Recently, this demand has been met by extensively using GPUs, as they provide large additional performance for a relatively low energy budget. Programming the resulting heterogeneous architecture which merges regular CPUs with GPUs is a very complex task, as one needs to handle load balancing together with data movements and task affinity (tasks have strongly different speedups on GPUs). A deep trend which has emerged to cope with this new complexity is using task-based programming models and task-based runtimes such as PaRSEC [1] or StarPU [2]. These runtimes aim at scheduling scientific applications, expressed as directed acyclic graphs (DAGs) of tasks, onto distributed heterogeneous platforms, made of several nodes containing different computing cores.

Reducing data movement is an important optimization to consider when scheduling tasks sharing data on a system with limited memory and bandwidth. For instance, it is relevant for an out-of-core execution on a computer made of several CPUs with restricted shared memory, and limited bandwidth for the communication between memory and disk. In this article, we focus on solving this optimization problem using GPUs, as they are widely used in scientific computing and have a limited memory as well as a limited bandwidth to read/write data from/to the main memory of the system. Thus, it is crucial to carefully order the tasks that have to be processed on each GPU so as to increase data reuse and minimize the amount of data that needs to be transferred. It is also important to schedule the transfers soon enough (prefetch) so that data transfers can be overlapped with computation and all tasks can start without delay. We focus in this paper on the problem of **scheduling a set of tasks on one GPU with limited memory, where tasks share some of their input data but are otherwise independent**. More precisely, we want to determine the order in which tasks must be processed to optimize for locality, as well as when their input must be loaded/evicted into/from memory. Our objective is to minimize the total amount of data transferred to the GPU for the processing of all tasks with a constraint on the memory size. We start focusing on independent tasks sharing

*Corresponding author

input data because when using usual dynamic runtime schedulers, the scheduler is exposed at a given time to a fairly large subset of tasks which are independent of each others. This is in particular the case with linear algebra workflows, such as the matrix multiplication or Cholesky decomposition: except possibly at the very beginning or very end of the computation, a large set of tasks is available for scheduling. Thus, solving the optimization problem for the currently available tasks can lead to a large reduction in data transfers and hence a performance increase.

In this paper, we make the following contributions:

- We provide a formal model of the optimization problem, and prove the problem to be NP-complete. We derive an optimal eviction policy by adapting Belady's rule for cache management (Section 3).
- We review and adapt three heuristic algorithms from the literature for this problem, and propose a new one based on gathering tasks with similar data patterns into packages (Section 4).
- We implement all four heuristics into the StarPU runtime and study the performance (amount of data transfers and total processing time) obtained on various tasks sets coming from linear algebra operations (Section 5). Overall, our evaluation shows that our heuristic generally surpasses previous strategies, in particular in the most constrained situations.

As for the optimization problem, the algorithmic solution proposed in this paper can be applied to a wide range of systems that have limited bandwidth and memory, not just GPUs. The algorithmic solution could be extended to situations that present processing units needing to compute tasks sharing data on a system with limited resources.

2. Related Work

The problem under study covers several area, such as computing with large data on GPUs, improving data reuse, and cache management. We detail here some related studies in these areas.

Out-of-core computations on GPUs. The limited memory of GPUs have motivated many studies on how to efficiently access data stored outside the GPU memory for several specific applications such as stencil computations [3], sorting problems [4], large scale graph processing [5] or graphic computations [6]. The solution generally consists in building data blocks that each fit within the GPU memory. However, these studies focus on the case of computing on multiple GPUs, and hence concentrate on the mapping problem (which data to put on which GPU?) rather than the scheduling problem: they generally ignore how to order tasks within one GPU, which is the main focus of the present paper. A possible approach to solve the mapping problem is to rely on graph theory and to model the problem either as a matching problem in a bipartite graph composed of workers and tasks [7] or as a partitioning problem in a graph where edges represent the amount of shared data between two tasks [8].

Data locality in runtime systems. As outlined in the introduction, runtime systems like OmpSs [9], PaRSEC [1], or StarPU [2] 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 [10] rely on work-stealing for load balancing, which also gives some guarantee on data locality [11]. On the contrary, the StarPU runtime system automatically calibrates performance models to predict task execution times. Based on these predictions, the DMDAR scheduler (presented below in Section 4) 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 [12] 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. The Python based Parla runtime [13] provides special data wrappers (Parla Arrays) which allows flexible memory management. Data locality is then considered when scheduling computations through a cost function that mixes the time required for moving data and the load of each node.

Data distribution can also be managed with modern high-level libraries. PHAST [14] or SYCL coupled with the Celerity API [15] can automate parallelization while leaving room to the programmer to direct data distribution on nodes.

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 [16].

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

Partitioned global address space (PGAS). Some programming models like X10 [17] or the HPX [18] runtime makes locality explicit to the user with PGAS: one can control which objects are located close to each other. This communication model significantly improves the asynchronism and the overlap of communications and computations. The Chapel programming language [19] has recently been enhanced with locality based optimizations [20] on the compiler level. These PGAS however do not allow for a refined schedule that would manage evictions to cope with limited memory.

General studies on data locality. Among the numerous studies on managing data locality for better performance, the work of Yao et al. [21] is the closest to our problem, as they optimize the scheduling of independent tasks sharing input data. However, their target platforms are multi-core CPUs. Hence, a large part of their work is devoted to grouping tasks before ordering them,

so as to map a set of tasks with similar data access pattern on a given CPU. In our experiments, we include a heuristic named MST which we adapt from the strategy they use to order tasks within a group.

Cache management. Our work is also related to existing studies in cache management that focus on the following question: when a cache of limited size is used to store pages requested by an application, how long a page should stay in the cache, and when needed, which page should be evicted from the cache to make room for a new one? When the full sequence of page requests is know in advance, Belady's rule (explained in Section 3) allows to minimize the number of page loads [22]. In most cases, the sequence of future requests in not available. However, knowing a fraction of the sequence of future requests allows to improve the competitive ratio [23]. There have been some studies on how to reduce the page loads when one is allowed to reorder some of the next requests [24, 25]. However, in the present study, we are able to fully reorder the full set of requests (tasks), whereas cache studies usually consider that only a limited number of future cache requests are known in advance, and may possibly be reordered. Besides, in our study, each task typically requests not a single data (or page), but a series of them.

The present paper is an extended version of [26]. Compared to this first publication, it contains the proof of the theoretical results, more detailed pseudocodes of algorithms, as well as experiments on four new applications.

3. Problem Modeling and Complexity

We consider the problem of scheduling independent tasks on one GPU with memory size M. As proposed in previous work [27], tasks sharing their input data can be modeled as a bipartite graph $G = (\mathbb{T} \cup \mathbb{D}, E)$. The vertices of this graph are on one side the tasks $\mathbb{T} = \{T_1, \ldots, T_m\}$ and on the other side the data $\mathbb{D} = \{D_1, \ldots, D_n\}$. An edge connects a task T_i and a data D_j if task T_i requires D_j as input data. For the sake of simplicity, we denote by $\mathcal{D}(T_i) = \{D_j \text{ s.t. } (T_i, D_j) \in E\}$ the set of input data for task T_i. We consider that all data have the same size. The GPU is equipped with a memory of limited size, which may contain at most M data simultaneously. During the processing of a task T_i, all its inputs $\mathcal{D}(T_i)$ must be in memory.

For the sake of simplicity, we here do not consider the data output of tasks. In the case of linear algebra for instance, the output data is most often much smaller than the input data and can be transferred concurrently with data input. Data output is then not the driving constraint for efficient execution. Our model can however be extended to integrate task output.

All m tasks must be processed. Our goal is to **minimize the amount of data movement**. To fulfill this objective, we will determine in **which order** to process each task and **when** each data must be **loaded** and **evicted**. More formally, we denote by σ the order in which tasks are processed, and by $\mathcal{V}(t)$ the set of data to be evicted from the memory before the processing of task $T_{\sigma(t)}$. A schedule is made of m steps, each step being composed of the following three stages (in this order):

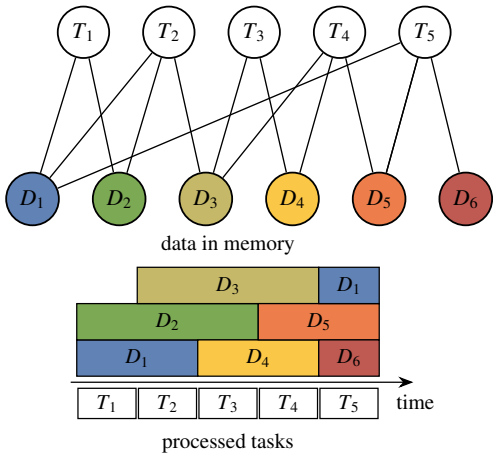


Figure 1: Example with 5 tasks and 6 data, with a memory holding at most $M = 3$ data. The graph of input data dependencies is shown on top. The schedule below corresponds to processing the tasks in the natural order with the following eviction policy: $\mathcal{V}(1) = \mathcal{V}(2) = \emptyset$, $\mathcal{V}(3) = \{1\}$, $\mathcal{V}(4) = \{2\}$, $\mathcal{V}(5) = \{3, 4\}$. This results in 7 loads (only D_1 is loaded twice).

1. All data in $\mathcal{V}(t)$ are evicted (unloaded) from the memory;
2. The input data in $\mathcal{D}(T_{\sigma(t)})$ that are not yet in memory are loaded;
3. Task $T_{\sigma(t)}$ is processed.

An example is shown in Figure 1. This example illustrates that input data are loaded in memory as late as possible: loading them earlier would be pointless and possibly trigger more data movements. In real computing systems, a pre-fetch is usually designed to load data a bit earlier so as to avoid waiting for unavailable data. For the sake of simplicity, we do not consider this in our model: if needed, we may simply book part of our memory for the pre-fetch mechanism.

Using the previous definition, we define the *live data* $L(t)$ as the data in memory during the computation of $T_{\sigma(t)}$, which can be recursively defined:

$$L(t) = \begin{cases} \mathcal{D}(T_{\sigma(1)}) & \text{if } t = 1 \\ L(t) = (L(t-1) \backslash \mathcal{V}(t)) \cup \mathcal{D}(T_{\sigma(t)}) & \text{otherwise} \end{cases}$$

The memory limitation can then be expressed as $|L(t)| \leq M$ for each step $t = 1, \ldots, m$. Our objective is to minimize the amount of data movement, i.e., to minimize the number of *load* operations: we consider that data are not modified so no *store* operation occurs when evicting a data from the memory. Assuming that no input data used at step t is evicted right before the processing ($\mathcal{V}(t) \cap \mathcal{D}(T_{\sigma(t)}) = \emptyset$), the number of loads can be computed as follows:

$$\#Loads(\sigma, \mathcal{V}) = \sum_t \left| \mathcal{D}(T_{\sigma(t)}) \backslash L(t) \right|$$

There is no reason for a scheduling policy to evict some data from memory if there is still room for new input data. We call *thrifty scheduler* such a strategy, formalized by the following constraints: if $\mathcal{V}(t) \neq \emptyset$, then $|L(t)| = M$. For this class of schedulers, the number of loads can be computed more easily: as soon as the memory is full, the number of loads is equal to

the number of evictions. That is, for the regular case when not all data fit in memory ($n > M$), we have:

$$\#Loads(\sigma, \mathcal{V}) = M + \sum_t |\mathcal{V}(t)|$$

Our optimization problem is stated below:

Problem 1 (MinLoadsForTasksSharingData). For a given set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$ according to $\mathcal{D}$, what is the task order σ and the eviction policy $\mathcal{V}$ that minimizes the number of loads #*Loads*?

A solution to this optimization problem consists in two parts: the order σ of the tasks and the eviction policy $\mathcal{V}$. Note that when each task requests a single data, finding an efficient eviction policy corresponds to the classical cache management policy problem. When the full sequence of data requests is known, the optimal policy consists in evicting the data whose next use is the furthest in the future. This is the well-known Belady MIN replacement policy [22] (see proof in [28]). We prove in the following theorem that this rule can be extended to our problem, with tasks requiring multiple data.

Theorem 1. We consider a task schedule σ for a MinLoadsForTasksSharingData problem. We denote by *MIN* the thrifty eviction policy that always evicts a data whose next use in σ is the latest (breaking ties arbitrarily). *MIN* reaches an optimal performance, i.e., for any eviction policy $\mathcal{V}$,

$$\#Loads(\sigma, MIN) \leq \#Loads(\sigma, \mathcal{V}).$$

Proof 1. We consider a given task order σ. We transform our problem so that each task depends on a single data. We replace each task T_i depending on data $\mathcal{D}(T_i) = \{D_1, \ldots, D_k\}$ by a series of $2k$ tasks: $T_i^{(1)}, T_i^{(2k)}$ such that $\mathcal{D}(T_i^{(j)}) = \mathcal{D}(T_i^{(j+k)}) = D_j$ for $j = 1, \ldots, k$. This transformation is performed both in the task set $\mathbb{T}$ (leading to $\mathbb{T}'$) and in the task order σ (leading to σ').

Let $\mathcal{V}$ be an optimal eviction policy for the original problem, i.e. for task set $\mathbb{T}$ and task order σ. We now transform it into an eviction policy for $\mathbb{T}'$ and σ' with the same number of loads and evictions. We group tasks by subsets of $2k$ tasks (as they were created above) and evict all data in $\mathcal{V}(t)$ before processing tasks $T_{\sigma(t)}^{(1)}, T_{\sigma(t)}^{(2k)}$ (and loading their missing inputs). We denote this strategy by $\mathcal{V}'$. Clearly, this is a valid strategy (we never exceed the memory if $\mathcal{V}$ did not on the original problem) and it has the same number of loads as $\mathcal{V}$:

$$\#Loads(\sigma, \mathcal{V}) = \#Loads(\sigma', \mathcal{V}').$$

Symmetrically, we consider an optimal eviction policy for the transformed problem ($\mathbb{T}'$ and σ') obtained with Belady's MIN replacement policy, denoted by *MIN*′: whenever some data must be evicted, it selects the one whose next use is the furthest in the future. We now prove that it can be transformed into an eviction policy *MIN* for the original problem with the same performance (loads and evictions), and that *MIN* also follows Belady's rule. We consider a subset of $2k$ tasks $T_i^{(1)}, T_i^{(2k)}$ coming from the expansion of task T_i scheduled at time t ($\sigma(t) = T_i$) and the set V_i of all data evicted by *MIN*′ before some task $T_i^{(j)}$. By property of *MIN*′ and as the memory is large enough for the inputs of task T_i ($M \geq k$), no input data of some $T_i^{(j)}$ belongs to V: during the first k tasks, their next occurrence is among the closest next tasks, and there is no eviction during the processing of the last k tasks. Thus, we can adapt *MIN*′ for the original problem by setting $MIN(t) = V$. It is easy to verify that *MIN* reaches the same performance as *MIN*′:

$$\#Loads(\sigma, MIN) = \#Loads(\sigma', MIN')$$

and that the data evicted at time t are (among the) ones whose next use is the furthest in the future.

As *MIN*′ is known to be optimal for the transformed problem, we have $\#Loads(\sigma', MIN') \leq \#Loads(\sigma', \mathcal{V}')$ and we conclude that $\#Loads(\sigma, MIN) \leq \#Loads(\sigma, \mathcal{V})$, which proves that *MIN* is optimal on the original problem. □

For cache management, Belady's rule has little practical impact, as the stream of future requests are generally unknown; simple online policies such as LRU (Least Recently Used [29]) are generally used. However in our case, the full set of tasks is available at the beginning. Hence, we can take advantage of this optimal offline eviction policy, as demonstrated below in our experiments, for example for the 3D matrix product (see Section 5.3). Thanks to the previous result, we can restrict our problem to finding the optimal task order σ. Unfortunately, this problem is NP-complete.

Theorem 2. Given a set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$ according to $\mathcal{D}$ and an integer B, finding a task order σ such that $\#Loads(\sigma, MIN) \leq B$ is NP-complete.

Proof 2. We first check that the problem is in NP. Given a schedule σ (and an eviction policy $\mathcal{V}$, which might be computed by *MIN*), it is easy to check in polynomial time that:

- The schedule is valid, that is, no more than M data are loaded in memory at any time step;
- The number of loads is not greater than the prescribed bound B.

The NP-completeness proof consists in a reduction from the cutwidth minimization problem (or *CMP*), proven NP-complete by Gavril in 1977 [30]. We denote by I_{CMP} an instance of *CMP* composed of a graph G, and by Δ the maximum degree of vertices in G. The question is to decide whether there exists a linear arrangement of the vertices such that the cutwidth is at most K. A linear arrangement α is a simple order of the vertices. The cutwidth $CUT_\alpha(v)$ of a vertex v under the linear arrangement α is the number of edges that connect vertices ordered before and after v in α, that is, the number of edges $(u, w) \in E$, such as $\alpha(u) < \alpha(v) < \alpha(w)$. The total cutwidth of G is the maximal cutwidth over all vertices : $CUT_\alpha(G) = \max_{v \in V} CUT_\alpha(v)$.

Given an instance I_{CMP} an instance of *CMP*, we create an instance $I_{MinLoads}$ of our problem as follows. For each vertex $v_i \in I_{CMP}$, we create a task T_i, and for each edge $e_k = (v_i, v_j)$, we create a data D_k such that D_k is a shared input of T_i and T_j. In addition, each task T_i with degree δ_i has $\Delta - \delta_i$ specific input

data, denoted by D_i^j for $j = 1, \ldots, \Delta - \delta_i$. Then,

$$\mathcal{D}(T_i) = \{D_k, \text{ s.t. } e_k \text{ is adjacent to } v_i\} \cup \{D_i^j \text{ for } j = 1, \ldots, \Delta - \delta_i\}.$$

Note that each task has exactly Δ input data. Finally, we set $M = K + \Delta$ and $B = |\mathbb{D}|$: we are looking for a solution where each data is loaded exactly once.

We now prove that if I_{CMP} has a solution, then $I_{MinLoads}$ has a solution. Let α be the linear arrangement solution of I_{CMP}. We consider the task order $\sigma = \alpha^{-1}$, and the optimal eviction policy *MIN*. We prove that

(i) A data is evicted only if it is not used anymore;

(ii) Each data is loaded exactly once.

Note that (ii) is a direct consequence of (i). We consider a step t when some data D_j is evicted and some task T_i is processed. We consider the set S of data in memory before starting step t together with the inputs of $T_{\sigma(t)}$ that are loaded in memory during step t. If D_j is evicted, this means that $|S| > M$ (*MIN* is a thrifty policy). We consider S', the subset of S containing the data that are used as inputs for a later step $t' > t$. By construction of $I_{MinLoads}$, each data $D_k \in S'$ corresponds to an edge $e_k = (v_a, v_b)$ in G such that $\sigma^{-1}(u_a) = \alpha(v_a) < t$ (the data was loaded for a task T_a scheduled before t) and $\sigma^{-1}(v_b) = \alpha(v_b) > t$ (the data is used for a task T_b scheduled after t). Hence, it corresponds to an edge counted in the cutwidth $CUT_\alpha(v_i)$. Since this cutwidth is bounded by K, there are at most K data in S'. Together with the Δ input data of the current tasks, no more than $K + \Delta = M$ in memory. Thus, the evicted data D_j is not used later than t. Since all data are loaded exactly once, the number of loads is not larger than B.

We now prove that if $I_{MinLoads}$ has a solution, then I_{CMP} has a solution. Let σ the task order in the solution of $I_{MinLoads}$. We construct the solution of I_{CMP} such that $\alpha = \sigma^{-1}$. We now prove that its cutwidth is not larger than K. By construction, the cutwidth $CUT_\alpha(v_i)$ at some vertex v_i (corresponding to a task T_i scheduled at time t) is the number of data which are used both before t and after t. Given the constraint on the number of loads, each data is loaded once, so such a data must be in memory during the processing of T_i, in addition to the Δ inputs of T_i, and there are at most $M - \Delta = K$ such data. This proves that $CUT_\alpha(v_i) \leq K$. Hence α is a solution for I_{CMP}. □

Extension to heterogeneous data. The previous model can be extended to cope with data having heterogeneous sizes. The NP-completeness result naturally extends to this variant of the problem. However, the optimality of the *MIN* eviction policy does not hold anymore (evicting a small data used before a large one may be beneficial). Some of the algorithms presented below extend to the heterogeneous variant; we state when it is the case.

4. Algorithms

We present here several heuristics to solve the MinLoads-ForTasksSharingData optimization problem. Two of them are adapted from the literature (Reverse-Cuthill-McKee and Maximum Spanning Tree), one of them is the actual dynamic strategy from the StarPU runtime (Deque Model Data Aware Ready) and we finally propose a new strategy: Hierarchical Fair Packing.

4.1. Reverse-Cuthill-McKee (RCM)

We have seen above that our problem is close to the cutwidth minimization problem, known to be NP-complete. This motivates the use of the CuthillMcKee algorithm [31], which concentrates on a close metric: the bandwidth of a graph. It permutes a sparse matrix into a band matrix so that all elements are close to the diagonal. If the resulting bandwidth is k, it means that vertices sharing an edge are not more than k edges away. We apply this algorithm on the graph of tasks $G^T = (\mathbb{T}, E^T, w^T)$ where there is an edge (T_i, T_j) if tasks T_i and T_j share some data, and where $w^T(T_i, T_j)$ is the number of such shared data. If the bandwidth of the graph is not larger than k, this means in our problem that any task T_i processed at time t has all its "neighbours" tasks (tasks sharing some data with T_i) processed in the time interval $[t - k; t + k]$. Hence, if k is low, this leads to a very good data locality. Reversing the obtained order is known to improve the performance of the CuthillMcKee algorithm [32], which we also notice in our experiments. The adaption of the Reverse-CuthillMcKee algorithm to our model is described in Algorithm 1.

Algorithm 1 Reverse-Cuthill-McKee heuristic

```
Build the graph G^T where vertices are tasks and edges are common data between tasks, weighted by the number of such data
σ ← [v] where v is the vertex of G^T with smallest weighted degree
i ← 0
while |σ| < m do
    Let N be the set of vertices adjacent to σ[i] in G^T not yet in σ
    Sort N by non-decreasing weighted degree
    Append N at the end of σ
    i ← i + 1
end while
Return σ in the reverse order
```

Differences between Cuthill-McKee (CM) and Reverse-Cuthill-McKee (RCM). We prove in the following theorem that both Cuthill-McKee and Reverse-Cuthill-McKee algorithms reach the same amount of data movement. More generally, reversing a schedule does not change the number of reads or eviction.

Theorem 3. For a given set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$ and a given task order σ : $\#Loads(\sigma, MIN) = \#Loads(\bar{\sigma}, MIN)$.

Proof 3. Given σ, an order of computation for $\mathbb{T}$, we know that data are used in the following order: $\mathcal{D}(T_{\sigma(1)}), \ldots, \mathcal{D}(T_{\sigma(m)})$. Together with the knowledge of the *MIN* eviction policy, we

can deduce the set of data that we need to load before computing task $T_{\sigma(t)}$, that we note S_t. It is the set of input data of T_i that were not in memory during the computation of the last task T_{i-1}, in other words: $S_t = \mathcal{D}(T_{\sigma(t)}) \backslash L(t-1)$. We denote by $\mathbb{S}$ the ordered list of data sets that we need to load before each task: $\mathbb{S} = [S_1, ..., S_m]$. Similarly, we build $\mathbb{V}$, the ordered list of data that we are evicted before each task= $\mathbb{V} = [\mathcal{V}(2), ..., \mathcal{V}(m), \mathcal{V}(m+1)]$. Note that we start at task 2 (no data is evicted before the first task) and we denote by $\mathcal{V}(m+1)$ the operation needed to completely empty the memory at the end of the execution. $\mathbb{S}$ and $\mathbb{V}$ totally describe the memory operations for an execution, and can be used to count the number of loads:

$$\begin{aligned} \#Loads(\sigma, MIN) &= \#Loads_{ordered\_list}(\mathbb{S}, \mathbb{V}) \\ &= \sum_{S_i \in \mathbb{S}} |S_i| = \sum_{\mathcal{V}(i) \in \mathbb{V}} |\mathcal{V}(i)| \end{aligned}$$

The last equality comes from the fact that each data is evict exactly as many times as it is loaded, thanks to the last eviction that totally frees the memory.

We consider the reversed order of σ: $\bar{\sigma}$, and similarly the reversed list of loads ($\bar{\mathbb{S}}$) and evictions ($\bar{\mathbb{V}}$). We consider $\mathbb{S}' = \bar{\mathbb{V}}$ and $\mathbb{V}' = \bar{\mathbb{S}}$ and notice that the pair $(\mathbb{S}', \mathbb{V}')$ describes correct lists of loading sets and eviction sets for $\bar{\sigma}$: this is what happens if we reverse the task order, and consider than each eviction for σ is transformed into a load, and each load for σ is transformed into an eviction. Hence, the total memory used by $(\mathbb{S}', \mathbb{V}')$ for $\bar{\sigma}$ is the same as the one used by $(\mathbb{S}, \mathbb{V})$ for σ, and not larger than M. Because $(\mathbb{S}', \mathbb{V}')$ is a correct loading/eviction scheme, we have

$$\#Loads_{ordered\_list}(\mathbb{S}', \mathbb{V}') \leq \#Loads(\bar{\sigma}, MIN)$$

We also have

$$\begin{aligned} \#Loads_{ordered\_list}(\mathbb{S}', \mathbb{V}') &= \#Loads_{ordered\_list}(\bar{\mathbb{V}}, \bar{\mathbb{S}}) \\ &= \sum_{S_i \in \mathbb{S}} |S_i| \\ &= \#Loads_{ordered\_list}(\mathbb{S}, \mathbb{V}) \\ &= \#Loads(\sigma, MIN) \end{aligned}$$

Hence, we have $\#Loads(\bar{\sigma}, MIN) \leq \#Loads(\sigma, MIN)$ By reversing once again the schedule (as well as the list of loading sets and eviction set), we obtain similarly that $\#Loads(\sigma, MIN) \leq \#Loads(\bar{\sigma}, MIN)$, proving the equality. □

We experimentally tested CM and RCM and concluded that the performance reached by both variants are not similar. We observed that in practice, RCM is always better than CM. Even if the total number of load is the same, the distribution of loads in time is not equal: there is more overlap between data movements and computation in RCM than in CM, which allows RCM to reach better performance.

Adaptation to heterogeneous data sizes. RCM can be adapted to heterogeneous data sizes by considering data weights instead of number of common data. We can thus modify line 1 of Algorithm 1 to: *Build the graph G^T where vertices are tasks and edges are common data between tasks, weighted by the weight of such data.*

4.2. Maximum Spanning Tree (MST)

As outlined in the related work, Yoo et al. [21] proposed another heuristic to order tasks sharing data to improve data locality. They first build a Maximum Spanning Tree in the graph G^T using Prim's algorithm [33] and then order the vertices according to their order of inclusion in the spanning tree. By selecting the incident edge with largest weight, they increase the data reuse between the current scheduled tasks and the next one to process. The direct adaption of the Maximum Spanning Tree to our model algorithm is described in Algorithm 2

Algorithm 2 Maximum Spanning Tree heuristic

```
For each vertex v_i set Key_Value(v_i) to 0
Key_Value(v_0) ← 1
while |σ| ≠ m do
    Choose v_i ∈ 𝕋 ∉ σ such that Key_Value(v_i) is maximum
    Add v_i at the end of the list σ
    For each couple (v_i, v_j), update Max_Path_Length(i, j)
    for each v_j adjacent to v_i∩ ∉ σ do
        if Key_Value(v_j) < Max_Path_Length(i, j) then
            Key_Value(v_j) ← Max_Path_Length(i, j)
        end if
    end for
end while
Return σ
```

4.3. Deque Model Data Aware Ready (DMDAR)

DMDA or "Deque Model Data Aware" (Algorithm 3) is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the StarPU runtime [34] (also called tmdp-pr). DMDA is a variant of DMDAS, the default state-of-the-art scheduler used by the Chameleon library [35]. This variant ignores task priorities that the user may set to denote that some tasks are more critical than others in the task graphs. As we consider independent tasks, such priorities are not useful. DMDA computes the expected completion time $C(T_i)$ of the first task T_i in the queue of tasks, based on a prediction of the time for transferring the data to the GPU (or communication time) *comm* and of the task computation time *comp*:

$$C(T_i) = \sum_{\substack{j \in \mathcal{D}(T_i) \\ D \notin InMem(GPU)}} comm(D_j) + comp(T_i) \qquad (1)$$

Note that the data transfer time is counted only if the data is not already in memory. The task is then allocated to the GPU which minimizes $C(T_i)$. Tasks are allocated to GPUs with this rule, one by one in their order of submission.

Here, we consider the DMDAR variant, which includes an additional *Ready* strategy (Algorithm 4): tasks are reordered at

runtime in order to favor tasks with the most input data already loaded into memory[1].

In our context with a single processing unit, DMDAR is reduced to selecting the next task with this strategy. DMDAR is a dynamic scheduler that relies on the actual state of the memory, it thus depends on the eviction policy, which is the LRU policy.

Algorithm 3 Deque Model Data Aware heuristic (DMDA)

```
1: InMem ← ∅
2: while all tasks have not been allocated do
3:     T_i ← pop(𝕋)
4:     Compute C(T_i) using Eq. 1
5:     Allocate T_i
6:     for each D_i ∈ 𝒟(T_i) do
7:         Request data prefetch for D_j
8:         Add D_i to InMem
9:     end for
10: end while
```

Algorithm 4 Ready reordering heuristic

Input: List L of tasks allocated

```
1: while L ≠ ∅ do
2:     Search first T ∈ L requiring the lowest amount of data transfers
3:     Wait for all data in 𝒟(T) to be in GPU's memory
4:     Start processing T
5: end while
```

DMDAR is already suited to heterogeneous data sizes, by taking them into account while computing $comm(D_j)$ and while selecting tasks in the *Ready* strategy.

4.4. Hierarchical Fair Packing (HFP)

HFP builds packages (denoted $P_1, P_2, ...$) of tasks, which are stored as lists of tasks, forming a partition of $\mathbb{T}$. To do so, it gathers tasks that share the most input data. By extension, we denote by $\mathcal{D}(P_k)$ the set of inputs of all tasks in package P_k. We aim at building the smallest number of packages so that the inputs of all tasks in each package fit in memory: $|\mathcal{D}(P_k)| \leq M$. The intuition is that once the data $\mathcal{D}(P_k)$ are loaded, all tasks in the package can be processed without any additional data movement. We have proven that building the minimum number of packages is NP-complete (see Theorem 4 and it's proof), hence we rely on a greedy heuristic to build them, described in Algorithm 5.

Theorem 4. We consider a set of tasks $\mathbb{T}$ sharing data in $\mathbb{D}$. Partitioning tasks into at most L packages $P_1, \ldots, P_L$ such that $|\mathcal{D}(P_i)| \leq M$ for each package P_i is an NP-complete problem.

Proof 4. Given a set of packages, it is easy to verify that they partition the task set and that the input size of each package is smaller than the M bound, hence the problem lies in NP.

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

We prove that the problem is NP-complete thanks to a reduction from the 3-partition problem: Given an integer B and $3n$ integer $a_1, a_2, \ldots, a_n$, such that $\sum_{i=1}^{3n} a_i = nB$ the problem is to decide whether we can partition $3n$ into n triplet whose sum is B. We consider the restricted version of the problem where for all i, $B/4 < a_i < B/2$, which is still NP-complete [36]. In this variant, each subset of integers reaching B has exactly three elements.

We consider an instance I_{3P} of the 3-partition problem and build an instance I_{MinP} of the package minimization problem as follows. For each $a_i \in I_{3P}$, we create a task T_i and a_i input data $\mathcal{D}(T_i) = \{D_{i,1} \ldots, D_{i,a_i}\}$ (no input data is shared among two tasks). We set the size limit of a package to $M = B$ and the maximum number of packages to $L = n$. Thus, in instance I_{MinP}, we try to solve the following question: Can we find at most n packages of input size at most M?

We know prove that if I_{3P} has a solution, then I_{MinP} has a solution. If I_{3P} has a solution, then we have n subsets of integers $S_1, S_2, \ldots, S_n$ which verify: $|S_i| = M$ for all i. We group tasks in n packages $P_1, P_2, \ldots, P_n$ such that $P_j = \{T_i, a_i \in S_j\}$ As $L = n$, we have exactly L packages. The input size of each package is

$$\left|\mathcal{D}(P_j)\right| = \sum_{T_i \in P_j} |\mathcal{D}(T_i)| = \sum_{a_i \in S_j} a_i = B = M.$$

Hence, this is a solution for I_{MinP}.

We now prove that if I_{MinP} has a solution, then I_{3P} has a solution. If I_{MinP} has a solution then there are at most L packages whose input size is at most M: $|\mathcal{D}(P_i)| \leq M$ for all i. We know that $\sum_{i=1}^{n} |S_i| = nM$, so $\sum_{i=1}^{n} |\mathcal{D}(P_i)| = nM$. We therefore have $L = n$ packages which must satisfy the following conditions:

$$\begin{cases} \sum_{i=1}^{n} |\mathcal{D}(P_i)| = nM \\ \forall i \, |\mathcal{D}(P_i)| \leq M \end{cases}$$

Any package with input size smaller than M would require that another package has a size larger than M, which is not possible. Therefore, we have $|\mathcal{D}(P_i)| = M$ for each package P_i. We denote by S_j the set of a_i corresponding to tasks T_i in P_j. Hence, $\sum_{a_i \in S_j} a_i = M$ for all S_j. We assume that $M/4 < a_i < M/2$, hence each S_j counts exactly three a_is, and the S_j are a solution to instance I_{3P}. □

Since building packages in an optimal way is NP-complete, we concentrate on a greedy heuristic to build them, described in Algorithm 5. We start with packages containing a single task. Then we iteratively consider all packages with fewest tasks and try to merge each of them with another package with whom it shares the most input data. To do so, we first compute the number of common data between each of the smallest packages (identified in $\mathbb{S}$) and all other packages. Then, for each P_i of the smallest packages, we select a package P_j such that the number of data shared by P_i and P_j is maximal and merge these two packages. Then, we mark P_j as not available for a merge at this step (to avoid all small packages merging with the same package in a single step). We also enforce that only pairs of packages with a maximal number of shared data are merged to

Algorithm 5 Hierarchical Fair Packing heuristic

Let $P_i \leftarrow [T_i]$ for $i = 1 \dots m$, $\mathbb{P} = \{P_1, \dots, P_m\}$ and $P_{\text{non\_connected}} \leftarrow \emptyset$
SizeLimit $\leftarrow$ *true*, *MaxSizeReached* $\leftarrow$ *false*,
while $|\mathbb{P}| > 1$ **do**
  MinPacketSize $\leftarrow \min_{P_i \in \mathbb{P}} |P_i|$
  $\mathbb{S} \leftarrow \{P_i \in \mathbb{P}$ with $|P_i| =$ *MinPacketSize*$\}$ ▷ The smallest packages that can be merged at this iteration
  for all packages $P_i \in \mathbb{S}$ and $P_j \in \mathbb{P}$ **do**
    SharedData$[i][j] \leftarrow \left|\mathcal{D}(P_i) \cap \mathcal{D}(P_j)\right|$
  end for
  MaxSharedData $\leftarrow 0$
  if *SizeLimit* = *true* **then** ▷ In case we are in the first phase
    MaxSharedData $\leftarrow \max_{P_i \in \mathbb{S}, P_j \in \mathbb{P}}$ *SharedData*$[i][j]$ such as $\left|\mathcal{D}(P_i \cup P_j)\right| \leq M$
    if *MaxSharedData* = 0 **then**
      SizeLimit $\leftarrow$ *false* ▷ End of first phase: lift the size constraint for second phase
    end if
  end if
  if *SizeLimit* = *false* **then** ▷ In case we are in the second phase
    MaxSharedData $\leftarrow \max_{P_i \in \mathbb{S}, P_j \in \mathbb{P}}$ *SharedData*$[i][j]$
    if *MaxSharedData* = 0 **then** ▷ We have identified subsets of tasks without any common data with other tasks
      for all packages $P_i \in \mathbb{S}$ **do**
        Merge P_i and $P_{\text{non\_connected}}$ ▷ This package gathers all non-connected task subsets
        Remove P_i from $\mathbb{P}$ and from $\mathbb{S}$
      end for
    end if
  end if
  $\mathbb{Q} \leftarrow \emptyset$ ▷ Set of packages that are not anymore available to merge with
  for all $P_i \in \mathbb{S}$ **do**
    Find j such that *SharedData*$[i][j]$ is maximal and $P_j \in \mathbb{P} \backslash \mathbb{Q}$
    if *MaxSharedData* = *SharedData*$[i][j]$ *and* $\left(\left|\mathcal{D}(P_i \cup P_j)\right| \leq M$ *or SizeLimit* = *false*$\right)$ **then**
      Merge P_i and P_j
      Add P_i and P_j to $\mathbb{Q}$
    end if
  end for
end while
if $|P_{non\_connected\_tasks}| > 0$ **then**
  Merge the only package in $\mathbb{P}$ and $P_{\text{non\_connected}}$ ▷ Retrieve all non-connected subsets
end if
Return the only package in $\mathbb{P}$

ensure good locality: if a small package cannot be merged at some step, it will be more successful in a later step. Merging two packages P_1 and P_2 consists in appending the list of tasks of P_2 at the end of the list of tasks of P_2: we never modify the order of tasks within an already built package, hence keeping a good data locality inside packages.

As presented above, we aim at building packages such that the size of the input data of a package is smaller than, or equal to M. This is done during the first phase. However, we do not stop here and we continue in a second phase to merge packages in the same way, without considering the bound on the size of input data anymore. The objective of this second phase is to create meta-packages that express the data affinity between packages already built, in order to schedule packages sharing many common input data close to each other.

Note that we may identified disconnected packages, that is, subsets of tasks that do not share any common data with other tasks: they correspond to connected components in the bipartite graph introduced in Section 3. We then merge the packages containing these tasks in a dedicated package, denoted $\mathbb{P}_{\text{non\_connected}}$, which is added to the last package in the very end of the algorithm. Eventually, the last remaining package after all merges provides the list of tasks of the final schedule.

Complexity of HFP. We now try to bound the complexity of HFP depending on the number m of tasks and on the maximal number of input data for any task: $\Delta = \max_i |\mathcal{D}(T_i)|$. We first remark that by keeping the input data of each package sorted, we can compute *SharedData*$[i][j]$ for any pair of packages P_i, P_j, with a complexity of $O(\mathcal{D}(P_i) + \mathcal{D}(P_j))$. Similarly, merging these packages can be done with the same linear complexity. The first part of an iteration of the outer while loop consists in computing *SharedData*$[i][j]$ for any packages $P_i \in \mathbb{S}$ and

$P_j \in \mathbb{P}$. The complexity of this phase is in $O(C)$ with

$$\begin{aligned} C &= \sum_{P_i \in \mathbb{S}} \sum_{j \in \mathbb{P}} \mathcal{D}(P_i) + \mathcal{D}(P_j) \\ &\leq \sum_{P_i \in \mathbb{P}} \sum_{j \in \mathbb{P}} \mathcal{D}(P_i) + \mathcal{D}(P_j) \\ &\leq 2\,|\mathbb{P}| \sum_{i \in \mathbb{P}} \mathcal{D}(P_i) \end{aligned}$$

Since all packages contain at least one task, we have $|\mathbb{P}| \leq m$. Moreover, we can bound the sum of all package inputs:

$$\sum_{i \in \mathbb{P}} \mathcal{D}(P_i) \leq \sum_{j=1}^{m} \mathcal{D}(T_j) \leq \Delta m$$

as each task has at most Δ inputs. Thus, the complexity of computing the shared data is in $O(\Delta m^2)$. Since there is at least one merge operation in each iteration of the while loop and there are at most m merge operations, the total complexity for computing shared data throughout the execution of the algorithm is in $O(\Delta m^3)$.

As outlined above merging two packages P_i, P_j can be done in complexity $O(\mathcal{D}(P_i) + \mathcal{D}(P_j))$, which is bounded by $O(\Delta m)$. Since there are at most m merge operations, the total complexity for merging is in $O(\Delta m^2)$. Hence the total complexity of HFP is dominated by the computation of the shared data and is in $O(\Delta m^3)$.

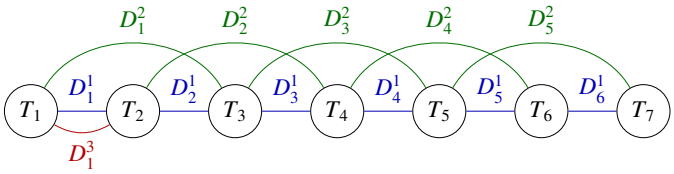


Figure 2: Data sharing among tasks that reach worst-case complexity for HFP (depicted for $m = 7$: an edge between two tasks represents a data shared by these tasks. Note that in addition to these data, tasks have private data so that each task exactly has Δ input data (except for T_m with $\Delta - 1$ data).

Note that this worst-case complexity can be reached on specific cases. We consider the following problem instance, depicted on Figure 2. Each task T_i, $3 \leq i \leq m$ has as input data D_{i-1}^1, D_{i-2}^2, and D_i^j for $j = 1, \ldots, \Delta - 2$. Task T_1 has input data D_1^j for $j = 1, \ldots, \Delta$; task T_2 has input data D_1^1,D_1^3, and D_2^j for $j = 1, \ldots, \Delta - 2$. In the very beginning, packages are made of single tasks, and they have at most one common data with another package, except for $\{T_1\}$ and $\{T_2\}$ which have two common data (D_1^1 and D_2^2), hence these packages are merged in the first step. In the second step, only package $\{T_3\}$ has two common data with the newly created package while all other packages have one or none common input data, hence $\{T_3\}$ is merge with the package created in the first step. Similarly, at step k, the package $\{T_{k+1}\}$ is merged with the package created at the previous step. At each step k, the shared input data must be recomputed. The first package contains k tasks, while the other $m - k$ packages contain a single task. Computing the shared data between the smallest packages and all other packages requires a complexity $O((m-k)^2\Delta + m\Delta)$. Summing over all $m-1$ steps, we reach a complexity $O((m-k)^3\Delta)$, corresponding to the previous bound.

In contrast with the previous pessimistic scenario, the complexity may be largely reduced in some cases. In an optimistic scenario, we merge all the packages by pairs at each iteration of the inner while loop, resulting in $\log_2 m$ iterations of this loop. At iteration i we have $\frac{m}{2^i}$ packages. At each iteration, the number of input data of a package at most doubles, thus this number is at most $2^i\Delta$ at iteration i. Since we need to compute many intersections of input data sets for a single merge, the cost of computing the intersections dominates the complexity. The complexity of this step for a single package at iteration i is thus $O(2^i\Delta)$. The total cost for iteration i is thus:

$$\left(\frac{m}{2^i}\right)^2 \times 2^i\Delta = \frac{m^2\Delta}{2^i}$$

When summing over all iterations, we get:

$$\sum_{i=1}^{\log_2 m} \frac{m^2 \times \Delta}{2^i} = O(\Delta \times m^2)$$

which gives the complexity of HFP in this optimistic scenario. Note that in linear algebra operations, all tasks have a very similar data access pattern and the pattern of data sharing is regular. Hence, in practice, this optimistic complexity is often reached.

Improving HFP with package flipping. A concern appears in the second step of HFP (when we merge packages without taking care of the M bound): if P_i is merged with P_j, the merged package contains the tasks of P_i followed by the ones of P_j. However, the last tasks of P_i might have very little shared data with the first tasks of P_j, leading to poor data reuse when starting P_j. Hence, for each package P_i, we consider two sub-packages P_i^{start} and P_i^{end} containing the first and last tasks so that the number of their input data is smaller than M but their number of tasks is maximal, as illustrated on Figure 3. Then, we count the common input data of each pair: $(P_i^{start}, P_j^{start})$, (P_i^{start}, P_j^{end}), (P_i^{end}, P_j^{start}), (P_i^{end}, P_j^{end}). We identify the pair with most common input data and selectively reverse the packages so that tasks in this pair of sub-packages are scheduled consecutively in the resulting package.

Flipping packages requires to go through the set of tasks of two packages. In the worst case, both packages together contain all of $\mathbb{T}$, so the complexity is $O(m)$. This complexity can be neglected compared to the original complexity of HFP.

Figure 4a and 4b show an example of the task processing order of C on a 2D matrix multiplication with and without package flipping with $M = 4$ (4 rows or columns can fit in memory at the same time). On these figures the numbers correspond to HFP's processing order of each block of C. When going from a colored set of tasks to another, at least 2 rows or columns must be loaded. Without package flipping, to process the blue set of tasks, 2 rows and 2 columns must be loaded because the memory is filled with rows number 2,3 and with columns number 0,1, which gives us a total of 12 data loads. In contrast, package flipping allows us to reuse the same rows (number 2,3)

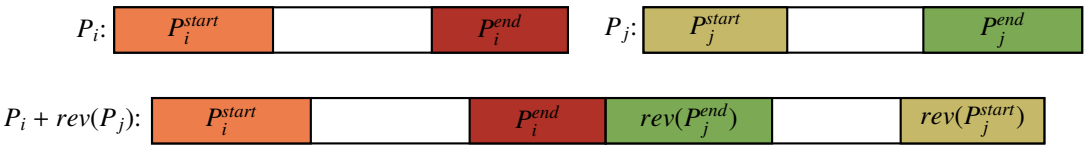
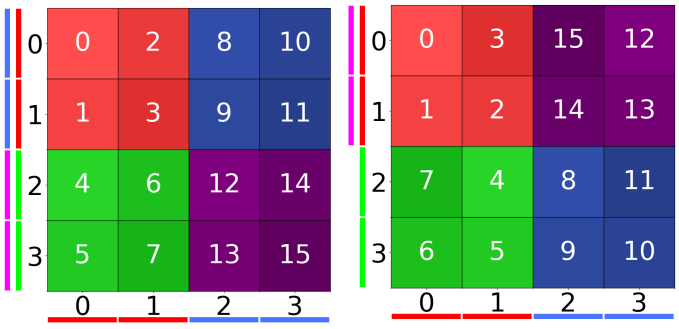


Figure 3: Flipping packages to improve HFP. Here we assume that the pair of sub-packages (P_i^{end}, P_j^{end}) is the one with the most shared input data, so that only P_j is reversed before merging packages.



(a) Without package flipping. (b) With package flipping.

Figure 4: HFP's processing order on a Square 2D Matrix Multiplication with $M = 4$. The colored lines on the sides represent data loads of rows or columns for the tasks of the corresponding color.

when going from the green to the blue set of tasks. We thus save 2 data loads for a total of 10 data loads. On a bigger scale, this "U-shape" formed by package flipping favors data reuse inside a package and thus reduces the total amount of data loads.

Optimal eviction policy. Lastly, we make another improvement to HFP: it is equipped with the optimal eviction policy adapted from Belady's rule (see Lemma 1). To make it compatible with dynamic runtimes, such as the StarPU runtime used in our experiments, we use a dynamic version of the eviction policy: whenever the runtime needs to evict some data, we choose the one whose next usage is the latest.

Adaptation to heterogeneous data sizes. It is possible to extend the HFP algorithm to deal with input data that have heterogeneous sizes. We assume that $weight(D)$ gives the weight of the input data D, and that this function is extended to tasks and subsets of tasks to give the size of their input data. Line 7 of Algorithm 5 needs to be modified to compute the weight of the data shared by two tasks instead of their numbers:

$$SharedData[i][j] \leftarrow \left|weight(\mathcal{D}(P_i)) \cap \mathcal{D}(P_j)\right|$$

Similarly, when testing if two packages P_i and P_j can be merged without exceeding the M bound, we need to replace the computation of the number of inputs data $\left|\mathcal{D}(P_i \cup P_j)\right|$ by the computation of the weight of these input data $weight(\mathcal{D}(P_i \cup P_j))$, on Lines 7 and 28 of Algorithm 5.

5. Experimental Evaluation

We present below the experimental evaluation conducted to compare the above schedulers to existing scheduling techniques in runtime systems [2]. The present study is limited to independent tasks, hence we concentrate on one of the major kernels in linear algebra made of independent tasks: matrix multiplication, as well as several variants. The state-of-the-art scheduling strategy is DMDAR, presented above: a version dedicated to independent tasks of the DMDAS scheduler used in the matrix multiplication of the Chameleon library [35].

5.1. Settings

All strategies mentioned above have been implemented using the StarPU runtime system [2]. This allows us to test them on a variety of applications expressed as sets of tasks. Experiments were conducted on a Tesla V100 GPU using cuBLAS 10.2 GPU kernels with single precision floating point numbers.

The GPU memory is limited to $M = 500\,MB$. This limitation allows us to distinguish the performance of different strategies even on small datasets. The V100 GPU has a memory of $M = 32\,GB$, thus we would have to use applications with a dataset 64 times larger in order to have the same ratio between the dataset size and the GPU memory. The time and power consumption costs would be much higher to make the same observations.

We use the following six benchmarks in our experiments: each benchmark is a set of independent tasks from a given application. All our applications are based on linear algebra operations and we use matrices composed of 960×960 tiles of single reals, a size that allows for a good tradeoff between performance and task density.

Square 2D Matrix Multiplication. To compute $C = A \times B$ in parallel, each task corresponds to the multiplication of one block-row of A per one block-column of B. Input data are thus the block-rows of A and block-columns of B. Matrix A is of size $N \times n$ (tiles) and matrix B of size $n \times N$, where N varies and $n = 4$.

Square 3D Matrix Multiplication. All matrices (A, B, C) are square and contains $N \times N$ tiles, and the computation of each tile of C is decomposed into multiple tasks, each of which requires one tile of A and one tile of B. Each tile of C is also used as input for all tasks on this tile but the first one.

Tasks of Cholesky Factorization. We consider the tasks of the tiled Cholesky decomposition [37] on a square $N \times N$ matrix, but remove all dependencies, as we are interested

[2] The code used to reproducibly obtain the results of this paper is available at `https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/FGCS2021`

only in independent tasks. Even if this does not compute the actual Cholesky decomposition, it allows to have dependencies with some regularity, but more complex than the 2D or 3D matrix multiplication.

Random Task Order from 2D Mat-Mult. We consider the set of tasks from the Square 2D Matrix Multiplication, but with a randomized submission order.

Randomized Pairs with 2D Inputs. We consider the set of tasks and data from the 2D matrix multiplication, but with a random dependency pattern between tasks and data: each task requires one (random) block-row of A and one (random) block-column of B. This allows us to test our algorithms on an unstructured dependency graph.

Sparse Square 2D Mat-Mult. Starting from the 2D Matrix Multiplication scenario above, we randomly remove 90% of the tasks, thus largely increasing the communication-to-computation ratio.

We use the four scheduling heuristics presented above, together with EAGER, a scheduler that processes tasks in the natural order (i.e. row major for matrix multiplications) as a baseline. DMDAR is considered as the state-of-the-art strategy as it is the one used in the Chameleon library. Each scheduling algorithm receives the whole set of tasks of an application in a natural order (row by row for a matrix multiplication for instance), and then output this same set of tasks in a new order, which is used in StarPU to process tasks on the GPU. We measure the obtained performance as the throughput of elementary computational operations performed per time unit (in GFlop/s, thus the higher, the better), as well as the total volume of data transferred between CPU and GPU (which we try to minimize). Each result 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 define the working set size as the size of all input data. For example, for the 2D matrix multiplications, we have:

$$working\_set\_size = 2 \times N \times n \times 960^2 \times 4$$

as we have 2 matrices of $N \times n$ tiles (where $n = 4$), each tile counting 960^2 elements of 4 bytes. We plot the performance obtained with various problem sizes, in order to test all strategies on various memory conditions.

Unless specified otherwise, for HFP we enable all three optimizations: *Ready* dynamic task reordering of DMDAR (see Section 4), package flipping (called flip on the plots), and Belady's optimal eviction policy (called Belady on the plots). All schedulers use LRU's eviction policy except for HFP (unless otherwise stated). In some cases, we take into account the scheduling's overhead of HFP. Our proposed heuristic is an offline scheduler, thus, if we know the size of the matrix as well as the size of the GPU memory, the processing order can be generated ahead of execution. For the same reasons, MST and RCM's scheduling time are not taken into consideration. On the contrary, DMDAR and EAGER are dynamic, so we cannot pre-compute their schedule and their scheduling time cannot be ignored. The overhead of the Belady's eviction policy (which is applied at runtime) is always taken into account in the results.

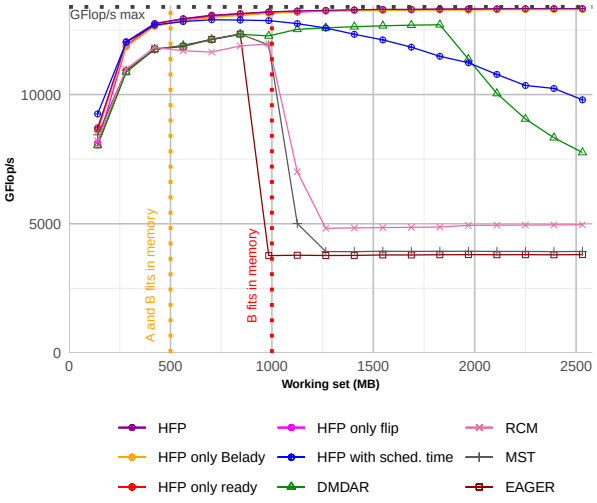


Figure 5: Performance on the Square 2D Matrix Multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90.

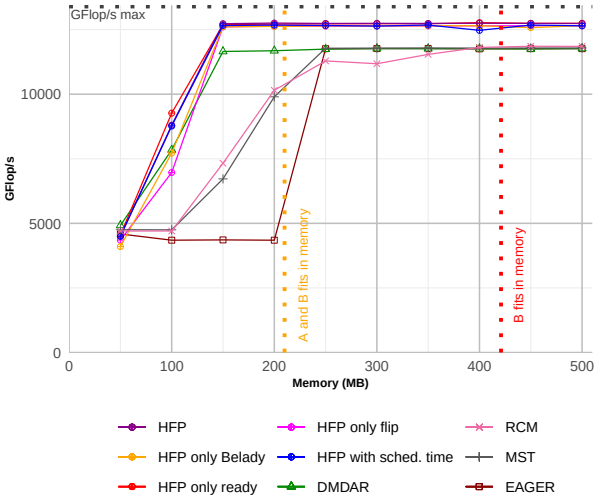


Figure 6: Performance on the Square 2D Matrix Multiplication in real execution with 1 Tesla V100 GPU while varying the memory size, N set to 15

5.2. Results on the 2D Matrix Multiplication

General overview. Figures 5 shows the performance of each scheduling heuristic when varying the size of the problem, while Figure 6 shows the performance while varying the available memory. On these graphs, the dotted horizontal black line represents the maximum throughput (13,393 GFlop/s) that the GPU can achieve when processing elementary matrix products (without I/Os) and is thus our asymptotic goal. The red dotted vertical line denotes the situation when the GPU memory can fit exactly only one of the two input matrices, and the orange line denotes the situation when it can accommodate both input matrices. The next figures use the same convention without the labels.

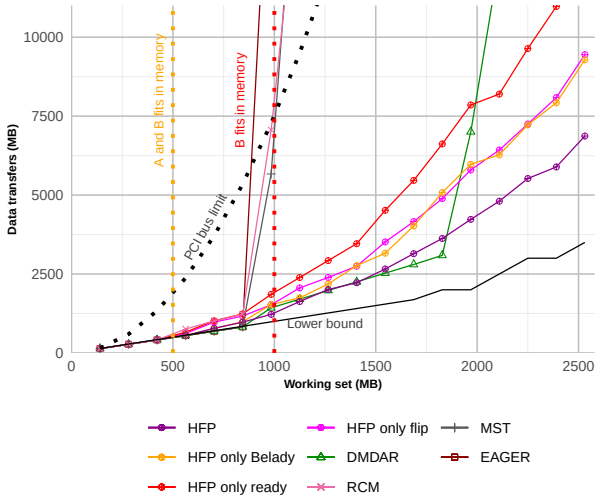


Figure 7: Amount of data transfers on the Square 2D Matrix Multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90.

Figure 7 shows the amount of data transfers. There, the black dotted curve represents the maximum number 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.

Communication lower bound for 2D matrix product. The solid black line on Figure 7 represents the lower bound of the amount of data transfers required to complete the matrix product. The derivation of this lower bound follows the ideas introduced by Hong and Kung [38] and later used by many other studies. Note that we cannot use general formulas for the communication lower bound of the matrix multiplication (such as [39]) as it deals with 3D multiplication, while we consider here the 2D multiplication.

The considered matrices have the following size: matrix A is $N \times n$, matrix B is $n \times N$ and matrix C is $N \times N$. All these sizes are in number of blocs, and the memory size of a block is S. We denote by $m = M/nS$ the maximum number of block rows of A (or block columns of B) that fits in memory. We define a *phase* of the computation as a time window during which at most m block columns of A (or block columns of B) are read (corresponding to a volume of I/O of M). Together with the m block rows/columns that may originally reside in memory, $2m$ block rows/columns are available for the computation. We are looking for an upper bound on the number of computations that may be performed using these $2m$ blocs rows/columns of data. The most optimistic situation is when m blocks rows of A and m blocks rows of B are available, leading to m^2 block computations. Since in total, we need to perform N^2 block computations, we have at least $\lfloor N^2/m^2 \rfloor$ full phases. Hence, a lower bound on the I/O is given by:

$$LB_{IO} = \left\lfloor \frac{N^2}{m^2} \right\rfloor M = \left\lfloor \frac{N^2 n^2 S^2}{M^2} \right\rfloor M = \left\lfloor \frac{input\_matrix\_size^2}{M^2} \right\rfloor M$$

where $input\_matrix\_size = nNS$ is the size of an input matrix (A or B). We may slightly refine this bound, by acknowledging the specificity of the first phase: no data is initially in memory when the computation starts. Hence, we define the first phase as the interval when the first $2m$ rows/columns are read (instead of m), which also leads to at most m^2 computations. We finally have two cases:

- Both input matrices fit in memory, leading to no full phases. In this case, the amount of I/O is $2 \times input\_matrix\_size$.
- Both input matrices do not fit in memory, leading to at least one full phase. In this case, the first phase leads to $2M$ I/Os.

This is summarized in the following formula:

$$LB_{IO} = \left\lfloor \frac{input\_matrix\_size^2}{M^2} \right\rfloor M + \min(M, 2 \times input\_matrix\_size)$$

Lastly, the minimum amount of data transfers cannot be smaller than the size of the inputs ($2 \times input\_matrix\_size$), so we take the maximum between this quantity and the previous bound.

A pathological matrix size for EAGER, MST and RCM. The EAGER, MST and RCM heuristics switch to pathological behavior at the red vertical line. We can both see the throughput plummeting (Figure 5) and the data transfers increasing (Figure 7) at the same working set size. These schedulers tend to process tasks along the rows of C. To explain the results we need to understand LRU's behavior when multiplying matrix. We multiply A by B to get C. For small matrices we can for example load all of B, a row of A and a piece of C to write the result in it. This results in few data transfers and thus good performance as we can see on Figure 5 before 1 000 MB. After 1 000 MB however, neither A nor B fits in memory. The scheduler is therefore forced to load a few columns of B, a row of A and a block of C. It computes the first row of C. Unfortunately it could not load all the columns from B, so when it wants to compute a block for which not all the data are in memory, it has to evict the first column from B (the one used least recently) in order to load the column of B it needs. But when it goes to the computation of the second row of C, we need the first columns of B that we just evicted. It must therefore again evict the last columns of B. This generates many additional data transfers as we can see on Figure 7. Consequently all the algorithms treating tasks row by row or column by column will suffer from this well-known pathological case of LRU. HFP aims to avoid this pathological case.

To help us understand the results we developed a visualization tool that represents the order produced by a scheduler as well as the resulting data loads by showing matrix C. Figure 8 shows the behavior of RCM on a matrix of size $N = 20$. For better readability, our visualizations uses a smaller memory ($M = 250\,MB$) and matrix size ($N = 20$). However it leads to the same behavior as the eighth point ($N = 40$ and $M = 500\,MB$) of Figure 5. The fade and the numbering represent the processing order. A tile is fitted with a horizontal (resp.

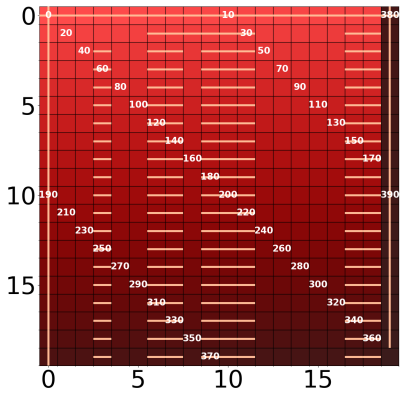


Figure 8: Visualization of RCM's processing order (represented by the fade from lighter to darker as well as the numbering) on the Square 2D Matrix Multiplication. An orange vertical (resp. horizontal) line in a square corresponds to a row (resp. column) load that was necessary to compute this tile. $N = 20$ and $M = 250\,MB$, which corresponds to the 8th point of Figure 5.

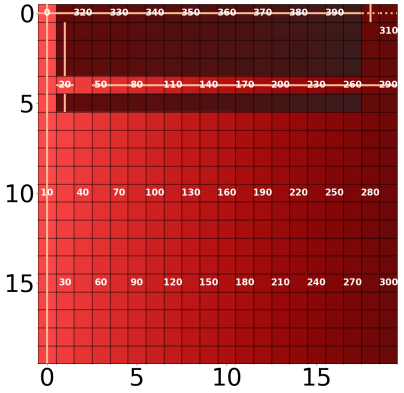


Figure 9: Visualization of DMDAR's processing order on the Square 2D Matrix Multiplication. $N = 20$ and $M = 250\,MB$, which corresponds to the 8th point of Figure 5.

vertical) orange line if a column (resp. row) load was necessary before computing the tile. This line may be solid, representing a blocking fetch operation, or dotted for a prefetch done in the background of previous computations. As we can see on Figure 8, by looking at the amount of solid horizontal lines in the squares, processing tasks rows by rows generates a lot of loads for the same columns of B. A similar phenomenon happens with EAGER and MST, and thus explains the amount of data transfers on Figure 7 for EAGER, MST and RCM.

DMDAR's results. As we can see on the visualization (Figure 9), DMDAR first processes the first column of C. Then instead of processing the first block from the second column of C, it will process a block from the second column on row number 4 because those data are already loaded on memory. Then it will continue with blocks from the second column. So, DMDAR does not suffer from LRU's pathological case because its *Ready* strategy allows it to rather process tasks that need the block-row of A already in memory instead of reloading the whole matrix. DMDAR's data transfers however start to rise for the last five working set sizes as we can see on Figure 7. That corresponds to the performance drop of the last five points of

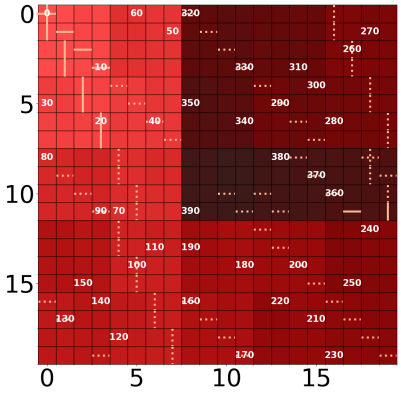


Figure 10: Visualization of HFP's processing order on the Square 2D Matrix Multiplication. $N = 20$ and $M = 250\,MB$, which corresponds to the 8th point of Figure 5.

Figure 5. The reason is a conflict between data prefetching and eviction. Indeed, once the GPU is filled with data, it is not clear for DMDAR whether some data should be evicted in order to perform more prefetches. It will thus 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. Also, when some data is actually evicted, DMDAR does not reconsider the task mapping according to the new set of data loaded on the GPU. The basic problem of DMDAR here is that it does not have a global view of the whole set of data and tasks, and thus cannot make a balance between prefetching and eviction. HFP aims to solve this issue.

HFP's results. As we observe on Figure 5, all HFP's variants (excluding HFP with sched. time) gets performance very close to ideal. Indeed, it tends to gather tasks that compute a square part of C that requires parts of A and B, that can fit in memory size M. On Figure 10, thanks to the color fade (and the numbering), we can distinguish that the processing order forms rectangles of blocks of C. This allows HFP to execute a lot of tasks with very few data to load. Moreover, we can see on the visualization that most of these data loads are done during a prefetch (dotted orange lines in the tiles), thus allowing to overlap computation and data transfers. As we can see on Figure 7, HFP with only Belady, flip or *Ready* induces more data transfers than DMDAR. However HFP performs better than DMDAR, even with these variants, thanks to better prefetching. It means that HFP is able to better distribute data transfers over time, while DMDAR has to transfer a lot of data at once when computing a new row of C. It is also worth noting that HFP with all optimizations (Belady, flip and *Ready*) greatly reduces data transfers compared to its variants with only one of them. Thus, HFP stays very close to the lower bound of the amount of data transfers required to complete the matrix product. This does not impact performance in 2D, but we will see in the following applications that combining all the improvements is crucial to achieve peak performance.

Table 1 offers us a summary of the performance of HFP with respect to the other algorithms.

The blue line in Figure 5 represents the performance of HFP

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	106.3 %	87.6 %	72.9 %	15.1 %

Table 1: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the Square 2D Matrix Multiplication in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90 (Figure 5).

when the scheduling time is not ignored and with a *fast start*, that is, with a reduced complexity for the first iteration of HFP. Indeed, during the first iteration of HFP, the search for the pair of tasks sharing the most data leads to a huge complexity (m^2 intersections to compute). For the 2D and 3D matrix products, the tasks are all identical, so we can predict the first merge of tasks of HFP. Thus, we merge together the tasks sharing at least one data without looking for the maximum of intersections. This leads to a reduced complexity without impacting the scheduling quality. HFP with scheduling time + *fast start* is outperformed by DMDAR only between 1 300 and 1 900 MB. The scheduling time of HFP greatly reduces it performance when the number of tasks becomes important, which may cancel out the benefit of a better locality. However, as the full set of task is available at the start of the computation, the whole schedule can be pre-processed offline, thus suppressing the scheduling cost. For the sake of readability, we do not show HFP with scheduling time on the next applications.

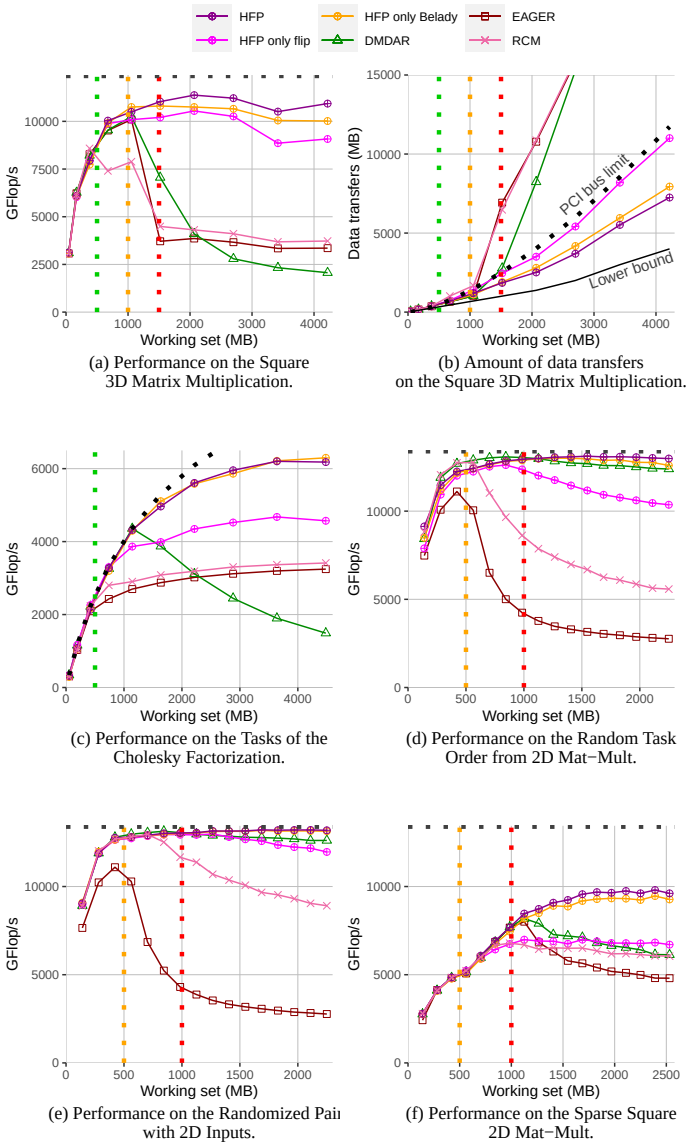
Figure 6 shows the dual view of Figure 5: The working set is now set to 422 MB and we vary the amount of available GPU memory. The measurements at 500 MB on Figure 6 (its last point) are the same as the measurements at 422 MB on Figure 5 (its third point). We can observe the same results as on Figure 5 but reversed: when the available memory is smaller than the working set, heuristics get pathological behavior. Since we strongly reduce the amount of available memory, we get a more restrictive situation, and the *Ready* task selection provides a large improvement on the second point. HFP with scheduling time is also always better than DMDAR. This graph confirms that HFP can achieve better performance than DMDAR, even under very constrained memory.

Figure 11 shows the results on the next five applications. On this figure, the MST heuristic is not plotted, as it is always slightly worse than RCM.

5.3. Results on the 3D Matrix Multiplication

General overview. On Figure 11.a, we plot the performance for all heuristics on the 3D matrix multiplication in real execution. On this set of tasks, matrix C now plays a role in affinities. Figure 11.b shows the amount of data transfer from the RAM to the GPU in these experiments. Remember that we do not count results written back to main memory in these data transfers.

Communication lower bound for the 3D matrix product. As for the 2D matrix product, we plot in Figure 11.b with a solid black line the amount of data loaded from the main memory to the GPU memory (we do not consider the communication of the results back to the main memory). We can apply the lower



(a) Performance on the Square 3D Matrix Multiplication.
(b) Amount of data transfers on the Square 3D Matrix Multiplication.
(c) Performance on the Tasks of the Cholesky Factorization.
(d) Performance on the Random Task Order from 2D Mat−Mult.
(e) Performance on the Randomized Pair with 2D Inputs.
(f) Performance on the Sparse Square 2D Mat−Mult.

Figure 11: Results on other datasets for a real execution on a Tesla V100 GPU.

bound on data reads from [39], which are here slightly tighten by computing the number of full phases with a floor function:

$$LB_{IO}^{3D} = 2M \left\lfloor \frac{N^3 S}{M\sqrt{M/S}} \right\rfloor$$

As previously for the 2D matrix product, we take the maximum between this quantity and the size of A and B ($2N^3S$) as both input matrices should be loaded at most once.

EAGER and RCM's results. One layer of C at a time, EAGER and RCM still compute tasks row by row. Thus, they still get pathological performance, but this time, when the memory cannot accommodate matrix A and B, as we can see on Figure 11.a. Indeed, their number of loads (on Figure 11.b) get dramatically high after the orange dotted line, from 1 100MB on the fifth point to 7 600 MB on the sixth point.

DMDAR's results. DMDAR suffer here from the same issue of balance between prefetching and eviction, mentioned in Section 5.2. It loads the full set of data from C in prefetches. Thus, when matrices A and B cannot fit in memory, we are in a situation where the memory is filled with data from C, but we still need to evict and load data from A and B. In this case, DMDAR will load a full column of B and then evict and load data from the rows of A, in order to avoid the eviction of prefetched data. This generates a lot of transfers as we can see on Figure 11.b. This affects the throughput, DMDAR achieves good performance for the first five points but, once the memory is constrained, the performance are diminished. On Figure 11.a, DMDAR has worse performance than EAGER for the last three points, because of the scheduling cost induced by looking at the full set of tasks in order to find a task that can load a minimum number of data.

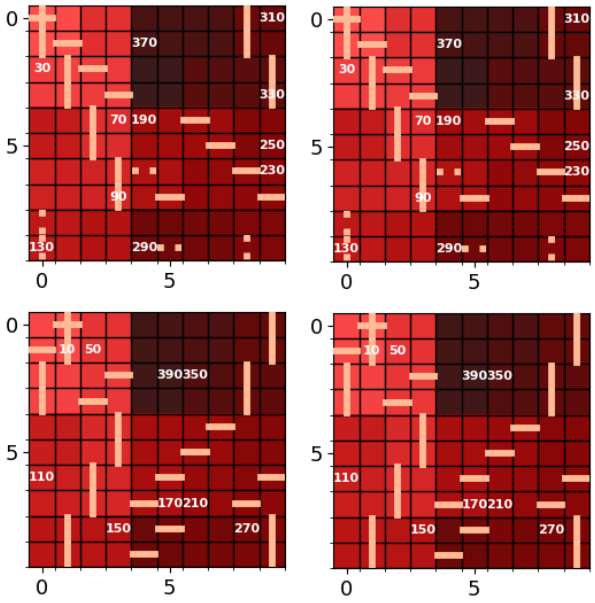


Figure 12: Visualization of HFP's processing order on the Square 3D Matrix Multiplication with only 4 layers.

HFP's results. For the sake of readability, Figure 12 shows visualization of a smaller version of the 3D matrix multiplication, when the size of inner loop is limited to 4 (matrix A is $N \times 4$ while B is $4 \times N$). The whole subset of tiles products (of size $N \times 4 \times N$) is plotted in 4 panels: the same tile on all four panels uses the same tile of C. HFP keeps gathering tasks forming rectangular blocks with all four layers of C, which provides a good locality between input matrices A, B and C, as well as maximizing data loads during a prefetch (as one can see with the orange dotted lines).

Table 2 shows us the percentages of improvement of HFP over the other heuristics.

HFP is still better on average. HFP without the Belady rule (HFP only flip) triggers a larger number of transfers (about twice as many transfers as HFP with Belady from the red dotted line as we see on Figure 11.b). The Belady rule reduces in a significant way the quantity of data transfers, which explains the better performance of HFP with this eviction policy.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	68.3 %	64.0 %	73.7 %	66.5 %

Table 2: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the Square 3D Matrix Multiplication in real execution with 1 Tesla V100 GPU, N ranging from 2 to 20 (from Figure 11.a).

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	66.8 %	57.5 %	56.3 %	65.8 %

Table 3: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the Tasks of the Cholesky Factorization in real execution with 1 Tesla V100 GPU, N ranging from 5 to 50 (from Figure 11.c).

5.4. Results on the Tasks of the Cholesky Factorization

Figure 11.c shows the performance of each scheduling heuristic on the task set of the Cholesky factorization. The green dotted line denotes the working set size allowing to load all the data needed for the computation on memory.

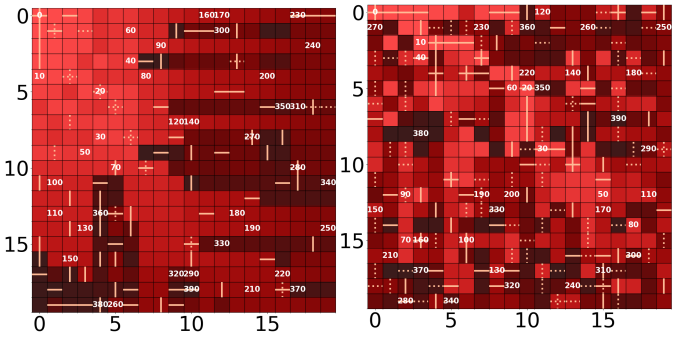
EAGER and RCM's results. We notice that the EAGER and RCM heuristics get pathological performance once the whole matrix cannot fit the memory. They indeed do not manage to reuse more than one tile between consecutive tasks, thus entailing a lot of tile reloads.

DMDAR and HFP's results. DMDAR has similar results with HFP for a working set inferior to 2.5 times the memory. DMDAR indeed takes advantage of the actual task submission order of the Cholesky algorithm, which starts with tasks which require few input data (POTRF and TRSM kernels). Meanwhile, it can load data for the subsequent tasks with more input dependencies (GEMM kernel). HFP, on the contrary, does not pay attention to the task submission order, and aims for data sharing as much as possible. It will thus introduce a lot of GEMM tasks at the beginning of the execution, and is thus producing more data transfers. As the working set increases, however, HFP achieves better performance than DMDAR. Belady's eviction policy is crucial in this case, indeed HFP with only Belady greatly increases the performance and allows us to stay close to our asymptotic goal. The average improvement of HFP over the other heuristics with Cholesky are available on table 3.

5.5. Results on the Random Task Order from 2D Mat-Mult

Results on the Random Task Order From the 2D Matrix Multiplication, shown on Figure 11.d, have an interesting impact on the results previously discussed from Figure 5.

EAGER and RCM's results. Due to the task order randomization, EAGER and RCM cannot take advantage of the natural order of submission of tasks, and their performance is greatly affected. Indeed for EAGER, once A or B cannot fit in memory, processing tasks row by row does not bring any improvement.



(a) DMDAR's processing order. (b) HFP's processing order.

Figure 13: Visualization of the processing order on the Random Task Order from 2D Mat-Mult. $N = 20$ and $M = 250\,MB$, which corresponds to the 8th point of Figure 11.d.

For RCM, its ability to process tasks row by row while modifying the processing order inside a row to favor data locality allows it to maintain better performance than EAGER.

DMDAR's results. The randomized task submission positively affects the performance of DMDAR. With the randomized submission order, DMDAR can no longer process tasks column by column, as it was the case in Section 5.2 and illustrated by Figure 9. Indeed with this random order, when DMDAR is planning the next task to process, it will naturally process blocks of C forming squares. It is very unlikely that random blocks from the same columns would be chosen. Moreover, once three blocks of C forming a square have been processed, DMDAR will necessarily find the fourth block, completing the square thanks to *Ready*, as it is looking for a data sharing the most data with what is loaded in memory. This is clearly visible on Figure 13a. The light red blocks form a perfect square representing the situation when both matrices fit in memory ($N = 8$ with $M = 250\,MB$ which corresponds to the third point on Figure 11.d). For the rest of the visualization we can still distinguish rectangles of tasks computed one after another. It therefore completely avoids the LRU's pathological case mentioned earlier. Thus the data transfers of DMDAR are much smoother with the random submission order, which improves its performance compared to Figure 5.

HFP's results. For the first seven points, HFP obtains poorer performance than DMDAR, which was not the case on Figure 5. As explained above, DMDAR is better with the random submission order. However, HFP suffers from an issue in this situation, which is illustrated by the visualization on Figure 13b. For the first 100 tasks (corresponding to the very light red squares), we observe that the tasks share the same rows and columns of data. However, if we want to merge this package with another, the data sharing will be much more limited than in the classic case of the 2D matrix, where we could merge packages using strictly the same rows or columns. Thus, the randomization forces us to form and merge packages corresponding to sparse matrices, which limits data reuse. This explains our performance on the first few points.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	143.5 %	109.4 %	50.1 %	1.2%

Table 4: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the Random Task Order from 2D Mat-Mult in real execution with 1 Tesla V100 GPU, N ranging from 5 to 80 (from Figure 11.d).

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	142.5 %	66.4 %	18.2 %	1.6%

Table 5: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the Randomized Pairs with 2D Inputs in real execution with 1 Tesla V100 GPU, N ranging from 5 to 80 (from Figure 11.e).

However, HFP is still able to form squares blocks of C that allows to compute a large number of tasks with a limited number of transfers. So even if we observe a large number of transfers on the visualization of Figure 13b, these are overlapped with the computation. Indeed there are a lot of transfers in orange dotted line on the visualization, meaning they are done in prefetch. This explains the good performance of HFP for the last points. In addition, we observe on Figure 11.d that HFP without Belady does not achieve optimal GFlop/s. It is therefore crucial to use Belady, which allows us to avoid evicting, and thus having to load again a data that will be used again in the future. Indeed, it is even more important to manage eviction when the submission order is random because we no longer rely on the natural order. Likewise, we can observe that HFP with all the improvements obtains better performance compared to HFP with only Belady. This shows that the combination of HFP's improvements is effective.

The average improvement of HFP over the other heuristics can be seen on table 4.

5.6. Results on the Randomized Pairs with 2D Inputs

The experiments on Randomized Pairs obtained from the Square 2D Matrix Multiplication, shown on Figure 11.e lead to very similar results than the previous experiments. The performance of EAGER decreases once a matrix does not fit in memory any more. Processing tasks along the rows of C is equivalent to a random processing order with a randomized matrix operation. RCM, gets more sustained performance: the randomization of dependencies actually decreases the effect of the classical LRU pathological case, since it does not tend to execute tasks rows by rows any more. For the same reasons as explained in Section 5.5, DMDAR gets performance close to ideal. HFP outperforms DMDAR from the eighth working set size. The randomization of data dependencies does not allow HFP to form perfect squares of tasks. Thus it can only beat DMDAR when the working set is very restrictive compared to the GPU's memory. The percentages of average improvement of HFP over the other heuristics averaged on the sixteen points is shown on Table 5.

5.7. Results on the Sparse Square 2D Mat-Mult.

Ref. Alg.	EAGER	MST	RCM	DMDAR
Impr.	65.9 %	55.6 %	46.6 %	41.6 %

Table 6: Percentages of improvement of HFP over the other heuristics averaged on all sizes. From results on the Sparse Square 2D Mat-Mult in real execution with 1 Tesla V100 GPU, N ranging from 5 to 90 (from Figure 11.f).

We can observe on Figure 11.f the performance obtained on a sparse matrix multiplication with only 10% of the tasks.

EAGER, RCM and DMDAR's results. Like on the previous experiments, EAGER's natural order does not carry enough locality to deal with the memory limitation. Both RCM and DMDAR manage to find affinities on the rows or columns, but the sparsity increases the amount of data transfers compared to the classical 2D matrix multiplication.

HFP's results. The way HFP groups packages does not depend on the ratio between the number of tasks and the number of different data. Even in this more data-intensive case, HFP is able to form clusters of data-sharing tasks. As we can see with HFP-only-flip on Figure 11.f, the package flipping optimization has no impact in this situation because we cannot form optimal orders in a sparse set of tasks. However, the packing heuristic coupled with Belady's eviction policy produces a good amount of data reuse, which explains why a good performance is kept even when memory is a constraint. The percentage of improvements of HFP are given on Table 6.

5.8. Summary of experimental results

On 2D and 3D matrix products, the proposed HFP algorithm outperforms its competitors: it is able to group together tasks of C forming rectangular blocks, as well as nicely ordering tasks among those blocks. Hence, it both minimizes the data transfers and allows a smooth overlap of communications with computations. In the case of the 3D matrix product, the Belady eviction policy further helps to improve the performance. On more heterogeneous applications such as tasks from the Cholesky factorization, HFP with all its optimizations (package flipping, *Ready* and Belady's eviction policy) is also able to reduce the amount of transfers and always reaches the best performance. On the randomized variants of the 2D matrix multiplication, HFP still manages to group tasks sharing data in order to obtain peak performance. Finally, on the sparse application, HFP is still able to form packages of tasks, increasing data reuse and thus leading to the best performance once the memory is a constraint.

The DMDAR scheduler, currently the best choice among available scheduling policies in StarPU, obtains good performance for 2D matrix multiplication as well as its randomized variants, but is unable to cope both with very large number of tasks and with more intricate data sharing patterns such as the ones from 3D matrix multiplication.

6. Conclusion and Future Work

Considering data movement is crucial to get the best performance out of GPUs. In this paper, we focus on the ordering of tasks sharing some of their input data on a GPU with limited memory and connected to the main memory through a bus with bounded bandwidth. We provide a formalization of the problem when trying to minimize the amount of data transfers and show that this problem is NP-complete. We also exhibit an optimal eviction scheme, based on Belady's rule. We adapt two heuristics from the literature (MST and RCM) and consider two schedulers used in runtime systems (EAGER and DMDAR). We design and implement a new algorithm gathering tasks with similar input data into packages of increasing size, called HFP. All five ordering strategies have been implemented using the StarPU runtime and tested for various sets of tasks on a V100 GPU. In all cases, HFP provides significant speedups. For instance, for 2D and 3D matrix products, HFP is on average 15.1% and 64.0% better than all other heuristics. HFP is very relevant and obtains important speedups particularly in the case when the memory is very constrained compared to the size of the total working set. Using the optimal eviction scheme allows HPF to reduce the number of data transfers and reach a total amount around twice the lower bounds than can be derived for the 2D and 3D matrix products.

Some data transfers are not costly, as they can be overlap by computation: we plan to focus on improving this overlap in future work. We also plan to concentrate on the very beginning of the execution, where it is crucial to first schedule tasks with few input data as they maximize the ratio between computations and required communications. Optimizing the implementation of Belady's rule and adapting it to the *Ready* dynamic task reordering is also key to integrate it in native executions. The complexity of HFP could also be further improved to make it usable on very large task sets. On a longer term, we want to tackle the general case with tasks not only sharing input data, but also with inter-task dependencies, as well as targeting multi-GPU platforms, for which our approach with packages seems particularly well suited.

Acknowledgments

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

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

References

[1] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J. Dongarra, PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability, Computing in Science and Engineering 15 (6) (Nov. 2013).

[2] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23 (2011). doi:10.1002/cpe.1631.

[3] G. Jin, T. Endo, S. Matsuoka, A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of gpus, in: 2013 IEEE International Conference on Cluster Computing (CLUSTER), 2013, pp. 1–8. doi:10.1109/CLUSTER.2013.6702633.

[4] Y. Ye, Z. Du, D. Bader, Q. Yang, W. Huo, Gpumemsort: A high performance graphics co-processors sorting algorithm for large scale in-memory data, GSTF INTERNATIONAL JOURNAL ON COMPUTING 1 (05 2011). doi:10.5176/2010-2283_1.2.34.

[5] K. Shirahata, H. Sato, S. Matsuoka, Out-of-core gpu memory management for mapreduce-based large-scale graph processing, in: 2014 IEEE International Conference on Cluster Computing (CLUSTER), 2014, pp. 221–229. doi:10.1109/CLUSTER.2014.6968748.

[6] J. Lee, H. Kang, H. ju Yeom, S. Cheon, J. Park, D. Kim, Out-of-core gpu 2d-shift-fft algorithm for ultra-high-resolution hologram generation, Opt. Express 29 (12) (2021) 19094–19112.

[7] Z. Fu, Z. Tang, L. Yang, C. Liu, An optimal locality-aware task scheduling algorithm based on bipartite graph modelling for spark applications, IEEE Transactions on Parallel and Distributed Systems 31 (10) (2020) 2406–2420. doi:10.1109/TPDS.2020.2992073.

[8] D. Tripathy, A. Abdolrashidi, L. N. Bhuyan, L. Zhou, D. Wong, Paver: Locality graph-based thread block scheduling for gpus, ACM Trans. Archit. Code Optim. 18 (3) (jun 2021).

[9] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, J. Labarta, Productive programming of GPU clusters with OmpSs, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, 2012, pp. 557–568.

[10] J. V. Ferreira Lima, T. Gautier, V. Danjean, B. Raffin, N. Maillard, Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures, Parallel Computing 44 (2015) 37–52.

[11] U. A. Acar, G. E. Blelloch, R. D. Blumofe, The data locality of work stealing, Theory Comput. Syst. 35 (3) (2002) 321–347.

[12] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, 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, pp. 1–11. doi:10.1109/SC.2012.71.

[13] H. Lee, W. Ruys, I. Henriksen, A. Peters, Y. Yan, S. Stephens, B. You, H. Fingler, M. Burtscher, M. Gligoric, et al., Parla: A python orchestration system for heterogeneous architectures, in: SC '22: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2022.
URL https://userweb.cs.txstate.edu/~burtscher/papers/sc22.pdf

[14] B. Peccerillo, S. Bartolini, Phast - a portable high-level modern c++ programming library for gpus and multi-cores, IEEE Transactions on Parallel and Distributed Systems 30 (1) (2019) 174–189.

[15] P. Thoman, P. Salzmann, B. Cosenza, T. Fahringer, Celerity: High-level c++ for accelerator clusters, in: R. Yahyapour (Ed.), Euro-Par 2019: Parallel Processing, Springer International Publishing, Cham, 2019, pp. 291–303.

[16] T. Herault, Y. Robert, G. Bosilca, J. Dongarra, Generic matrix multiplication for multi-gpu accelerated distributed-memory platforms over parsec, 2019, pp. 33–41. doi:10.1109/ScalA49573.2019.00010.

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: An object-oriented approach to non-uniform cluster computing, SIGPLAN Not. 40 (10) (2005) 519538.

[18] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, Hpx: A task based programming model in a global address space, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14, Association for Computing Machinery, New York, NY, USA, 2014.

[19] D. Callahan, B. Chamberlain, H. Zima, The cascade high productivity language, in: Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings., 2004, pp. 52–60. doi:10.1109/HIPS.2004.1299190.

[20] E. Kayraklioglu, E. Ronaghan, M. P. Ferguson, B. L. Chamberlain, Locality-based optimizations inthechapel compiler, in: X. Li, S. Chandrasekaran (Eds.), Languages and Compilers for Parallel Computing, Springer International Publishing, Cham, 2022, pp. 3–17.

[21] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, C. Kozyrakis, Locality-aware task management for unstructured parallelism: A quantitative limit study, in: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2013. doi:10.1145/2486159.2486175.

[22] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems Journal 5 (2) (1966).

[23] S. Albers, On the influence of lookahead in competitive paging algorithms, Algorithmica 18 (3) (1997) 283–305.

[24] T. Feder, R. Motwani, R. Panigrahy, S. Seiden, R. van Stee, A. Zhu, Combining request scheduling with web caching, Theoretical Computer Science 324 (2) (2004).

[25] S. Albers, New results on web caching with request reordering, Algorithmica 58 (2) (2010) 461–477.

[26] M. Gonthier, L. Marchal, S. Thibault, Locality-Aware Scheduling of Independent Tasks for Runtime Systems, in: COLOC - 5th workshop on data locality - 27th International European Conference on Parallel and Distributed Computing, Lisbon, Portugal, 2021, pp. 1–12.

[27] K. Kaya, B. Uçar, C. Aykanat, Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories, J. Parallel Distributed Comput. 67 (3) (2007). doi:10.1016/j.jpdc.2006.11.004.

[28] P. Michaud, (yet another) proof of optimality for min replacement (Oct. 2007).

[29] P. J. Denning, The working set model for program behavior, Communications of the ACM 11 (5) (1968) 323–333.

[30] Gavril, Some np-complete problems on graphs, in: Proceedings of the 11th conference on Information Sciences and Systems, 1977, pp. 91–95.

[31] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 1969 24th National Conference, ACM '69.

[32] W.-H. Liu, A. H. Sherman, Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices, SIAM Journal on Numerical Analysis 13 (2) (1976) 198–213.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.

[34] C. Augonnet, J. Clet-Ortega, S. Thibault, R. Namyst, Data-Aware Task Scheduling on Multi-Accelerator based Platforms, in: Int. Conf. on Parallel and Distributed Systems, 2010.

[35] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S. P. Thibault, Achieving high performance on supercomputers with a sequential task-based programming model, IEEE Transactions on Parallel and Distributed Systems (2017) 1–1doi:10.1109/TPDS.2017.2766064.

[36] M. R. Garey, D. S. Johnson, Computers and Intractability, A Guide to the Theory of NP-Completeness, W.H. Freeman and Co, London (UK), 1979.

[37] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, J. Roman, S. Thibault, S. Tomov, Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators., in: Symposium on Application Accelerators in High Performance Computing, 2010.

[38] J.-W. Hong, H. Kung, I/O complexity: The red-blue pebble game, in: STOC'81: Proceedings of the 13th ACM symposium on Theory of Computing, ACM Press, 1981, pp. 326–333.

[39] T. M. Smith, B. Lowery, J. Langou, R. A. van de Geijn, A tight i/o lower bound for matrix multiplication, available at https://arxiv.org/abs/1702.02017 (2019). arXiv:1702.02017.