Locality-Aware Scheduling of Independent Tasks for Runtime Systems

Maxime Gonthier[1], Loris Marchal[1], and Samuel Thibault[2]

[1] LIP, CNRS, ENS de Lyon, Inria & Université Claude-Bernard Lyon 1,
maxime.gonthier@ens-lyon.fr loris.marchal@ens-lyon.fr
[2] LaBRI, University of Bordeaux, CNRS, Inria Bordeaux – Sud-Ouest
samuel.thibault@u-bordeaux.fr

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 able to order tasks and prefetch their input data in the GPU memory (after possibly evicting some previously-loaded data), while aiming at minimizing data movements, so as to reduce the total processing time. In this paper, we focus on how to schedule tasks that share some of their input data (but are otherwise independent) on a 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. These strategies have been implemented in the StarPU runtime system. We present their performance on tasks from tiled 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, Runtime systems.

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 [4] 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.

Data movement is an important problem to consider when scheduling tasks on GPUs, as those 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 GPUs 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 computations 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 GPUs for the processing of all tasks with a constraint on the memory size. We start focusing on independent tasks sharing 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.

Because of space limitation, the complete review of related work is devoted to the extended version of the paper [8]. 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 2).
- 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 3).
- We implement all four heuristics into the StarPU runtime and study the performance (amount of data transfers and total processing time) obtained on both 2D and 3D blocked matrix multiplications (Section 4). Overall, our evaluation shows that our heuristic generally surpasses previous strategies, in particular in the most constrained situations.

Note that while we focus our experimental validation on GPUs, the optimization problem studied in this paper is not specific to the use of such accelerators:

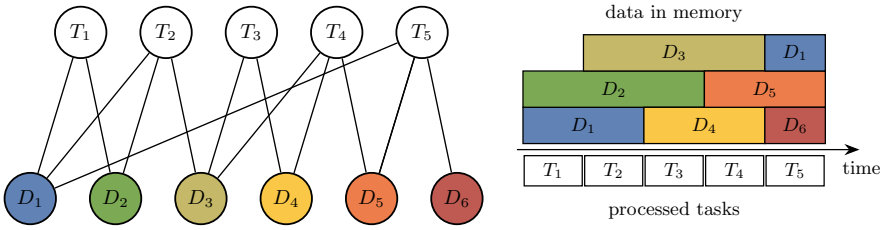


Fig. 1: Example with 5 tasks and 6 data, with a memory holding at most $M = 3$ data. The graph of input data dependencies is shown on the left. The schedule on the right corresponds to processing the tasks in the natural order with the following eviction policy: $\mathcal{V}(1) = \mathcal{V}(2) = \emptyset$,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

it appears as soon as tasks sharing data must be processed on a system with limited memory and bandwidth. For example, it is also relevant for a computer made of several CPUs with restricted shared memory, and limited bandwidth for the communication between memory and disk.

2 Problem modeling and complexity

We consider the problem of scheduling independent tasks on one GPU with memory size M. As proposed in previous work [9], 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 here 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 could however easily be extended to integrate task output.

All m tasks must be processed. Our goal is to determine in **which order** to process them, and **when each data must be loaded or evicted**, in order to **minimize the amount of data movement**. 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):

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, however, 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 defined recursively:

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

Our 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:

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

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:

Definition 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 [3]. We prove in the following theorem that this rule can be extended to our problem, with tasks requiring multiple data (see proof in the extended version of the paper [8]).

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 $\mathcal{V}$,*

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

For cache management, Belady's rule has little practical impact, as the stream of future requests is generally unknown; simple online policies such as LRU (Least Recently Used [7]) 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. Thanks to the previous result, we can restrict our problem to finding the optimal task order σ. Unfortunately, this problem is NP-complete. The proof, available in [8], consists in a reduction from the cutwidth minimization problem on graphs.

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.*

3 Algorithms

We present here several heuristics to solve the MinLoadsForTasksSharingData 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.

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 Cuthill–McKee algorithm, 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 [6]. 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 Cuthill–McKee algorithm, which we also notice in our experiments. The straightforward adaptation of the Reverse-Cuthill–McKee algorithm to our model is available in the extended version [8].

Maximum Spanning Tree (MST) Yoo [10] et al. 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 and then order the

Algorithm 1 Hierarchical Fair Packing heuristic

Let $P_i \leftarrow [T_i]$ for $i = 1 \dots m$ and $\mathbb{P} = \{P_1, \dots, P_m\}$
SizeLimit $\leftarrow$ *true*, *MaxSizeReached* $\leftarrow$ *false*,
while $|\mathbb{P}| > 1$ **do**
    while (*MaxSizeReached* = *false* or *SizeLimit* = *false*) and $|\mathbb{P}| > 1$ **do**
        MaxSizeReached $\leftarrow$ *true*
        for all packages P_i with the smallest number of tasks **do**
            Find a package P_j such that $|\mathcal{D}(P_i) \cap \mathcal{D}(P_j)|$ is maximal
            if $weight(P_i \cup P_j) \leq M$ or *SizeLimit* = *false* **then**
                Merge P_i and P_j (append P_j at then end of P_i and remove P_j from $\mathbb{P}$)
                MaxSizeReached $\leftarrow$ *false*
            end if
        end for
    end while
    SizeLimit $\leftarrow$ *false*
end while
Return the only package in $\mathbb{P}$

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 the extended version [8].

Deque Model Data Aware Ready (DMDAR) DMDA or "Deque Model Data Aware" is a dynamic scheduling heuristic designed to schedule tasks on heterogeneous processing units in the StarPU runtime. It takes data transfer time into account and schedules tasks where their completion times is expected to be minimal [1] (also called tmdp). We focus here on a variant, DMDAR, which additionally uses a *ready* strategy at runtime, to favor tasks whose data has already been loaded into memory. If at some point the next task T_i planned for execution requires some data which is not yet loaded in the GPU memory, then it looks further in the list of scheduled tasks. If it finds a task T_j that needs to load strictly less data than task T_i, it will first opportunistically compute that task T_j (see the extended version for details [8]). 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.

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 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

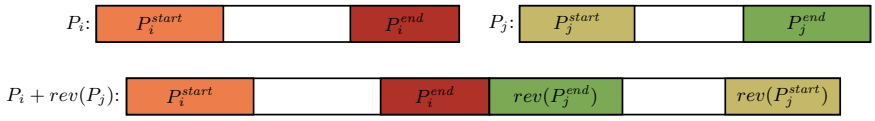


Fig. 2: Flipping packages to improve HFP. Here we assume that the pair of sub-packages (P_i^{end}, P_j^{end}) is the one with the most shared input data, so that only P_j is reversed before merging packages.

proven that building the minimum number of packages is NP-complete [8], hence we rely on a greedy heuristic to build them, described in Algorithm 1. We start with packages containing a single task. Then we consider all packages with fewest tasks and try to merge each of them with another package with whom it shares the most input data. When it is not possible to merge packages without exceeding the M bound any more, we perform a second step where we gather packages in the same way but ignore the M bound on the input size. The intuition is to create meta-packages that express the data affinity between packages already built. Note that we do not modify the order of tasks within packages when merging them, hence keeping the good data locality inside packages. Eventually, the last remaining package after all merges is the list of tasks for the schedule.

We note $\Delta = \max_i |\mathcal{D}(T_i)|$ the maximal number of data for any task. For linear algebra applications, it is most often a very small constant number. The worst-case complexity of HFP (detailed in the extended version [8]) is $O(m^3\Delta^2)$.

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 weight of their input data is smaller than M but their cardinal is maximal, as illustrated on Figure 2. 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.

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.

HFP's packing and package flipping allows it to be applicable and have good performance with other classes of problem such as the Cholesky factorization or random tasks graphs.

4 Experimental evaluation

We present below a subset of the experimental evaluation conducted to compare the strategy presented above.[3] We refer the interested reader to the extended version of the paper [8] for a more thorough discussion of these results, as well as experiments on other datasets (Cholesky and randomized 2D multiplication tasks sets). We used cuBLAS 10.2 GPU kernels with single precision.

4.1 Settings

All strategies mentioned above have been implemented in the StarPU runtime system [2]. This allows us to test them on a variety of applications expressed as sets of tasks. We performed both real experiments on a tesla V100 GPU as well as simulations using the ability to run StarPU code over the SimGrid simulator [5] to test our strategies in various experimental conditions. The use of simulation is motivated both by the fidelity of the simulated results as well as the saving of energy consumption. Even on the actual GPU, we have divided the original 12000 MB/s PCI bandwidth by two (by generating traffic between the CPU memory and another GPU) to represent the bandwidth share typically available for a given GPU in a multi-GPU platform. We have limited the GPU memory to 500 MB in order to better distinguish the performance of different strategies even on small datasets.

The scheduling algorithms receive the whole set of tasks of the application in a natural order (row by row for a matrix multiplication for instance), 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 (in GFlop/s) as well as the total volume of data transferred between CPU and GPU. When measuring GFlop/s, the cost of computing the MST, RCM, and HFP heuristics is not considered, to only observe their benefit as a first approach.

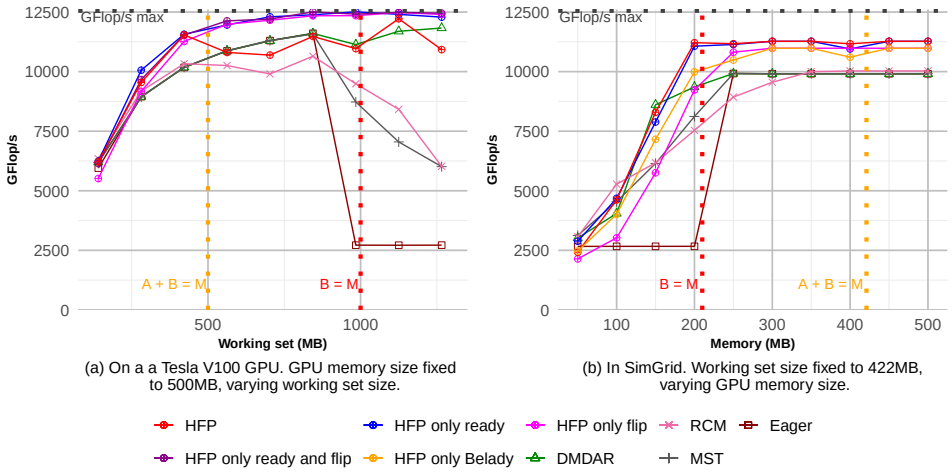
We use two sets of tasks for these experiments (see [8] for more datasets).

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 rows of A and columns of B.

Square 3D Matrix multiplication All matrices (A, B, C) are tiled, 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.

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. Unless specified otherwise, for HFP we enable all of the *Ready* dynamic task reordering of DMDAR (see Section 3), the package flipping (called flip on the plots), and Belady's optimal eviction policy (called Belady on the plots). We also show results when enabling only one of them.

[3] The code used to reproducibly obtain the results of this paper is available at `https://gitlab.inria.fr/starpu/locality-aware-scheduling/-/tree/coloc2021`



(a) On a a Tesla V100 GPU. GPU memory size fixed to 500MB, varying working set size.

(b) In SimGrid. Working set size fixed to 422MB, varying GPU memory size.

Fig. 3: Performance on the 2D matrix multiplication.

4.2 Results on the 2D matrix multiplication

On Figure 3, we plot the performance of each scheduling heuristic when varying either the size of the problem, or conversely the size of the available memory. On these graphs, the dotted horizontal black line represents the maximum GFlop/s (12557) that the GPU can achieve when processing elementary matrix product (without I/Os) and is 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 Eager, MST and RCM heuristics switch to pathological behavior at the red line. Indeed, they tend to process tasks along the rows of C. This allows us to reuse the same block-row of matrix A for tasks that compute tiles of the same row of C, but requires reloading the whole matrix B for each new block-row of A, which is a well-known pathological case of the LRU eviction policy.

DMDAR does not suffer from this pathological case because its *Ready* strategy allows it to rather process tasks that need the block-column of B already in memory instead of reloading the whole matrix.

The HFP heuristic gets performance very close to ideal. Indeed, it tends to gather tasks that compute a square part of C that require parts of A and B, that can fit in memory size M. This allows us to execute a lot of tasks with very few data to load. On figure 3a which shows native execution measurements, we notice that, with larger working sets, the cost of our implementation of the Belady rule brings significant overhead. On other figures which show simulated execution, this overhead is not included, which allows to observe its benefit. Here are the percentage of improvement of HFP with only *Ready* and *flip* over the other heuristics, averaged on the nine points:

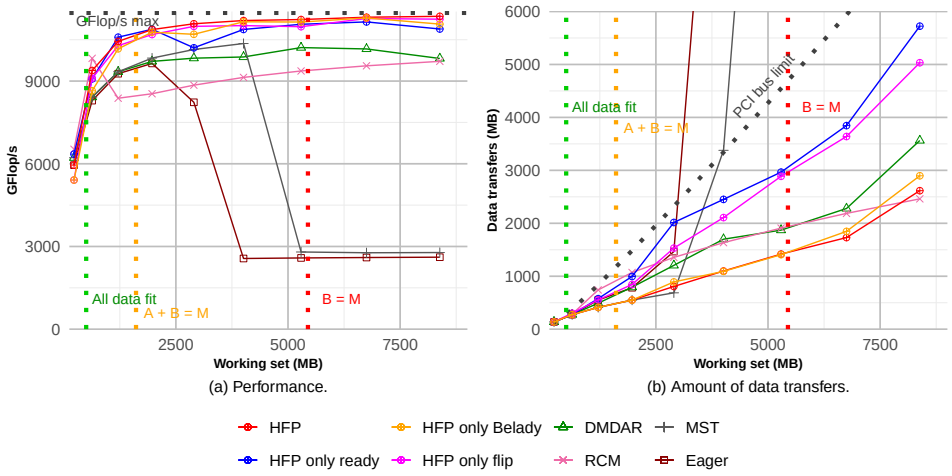


Fig. 4: Results on 3D matrix multiplication on SimGrid: GPU memory size fixed to 500MB, varying working set size.

Reference	Eager	MST	RCM	DMDAR	HFP only flip	HFP only ready
Improvement	51.5%	25.6%	26.0%	8.3%	1.9%	-0.2%

Figure 3b shows the dual view of Figure 3a: the working set is now set to 422MB and we simulate varying amounts of available GPU memory. The measurements at 500MB on Figure 3b are the same as the measurements at 422MB on Figure 3a. We can observe the same results as on 3a 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 here. The Belady rule or package flipping alone do not provide the same amount of improvement.

4.3 Results on the 3D matrix multiplication

On Figure 4, we plot the performance and amount of data transfers for all heuristics on the 3D matrix multiplication. On this set of tasks, matrix C now plays a role in affinities, which is why we added a vertical green dotted line to denote the situation when all A, B, and C matrices fit in memory. On Figure 4b, the black dotted line represents the maximum number of transfers that can be done during the minimum time for computation (given by the bound on the GFlop/s), thus the hard limitation induced by the PCI bus bandwidth: a heuristic exceeding this amount necessarily requires more than the optimal time for computation.

MST keeps ordering tasks along the rows of C, and thus still gets pathological performance when memory can not fit matrix B. This is confirmed on Figure 4b: the number of loads gets dramatically high. RCM and DMDAR, however, do not

have the same problem. RCM (resp. DMDAR) computes tasks along columns (resp. rows) of C but alternates between tasks of a few consecutive columns (resp. rows). This allows them to improve data reuse: Figure 4b shows that they exhibit a limited number of transfers, even with a large working set.

HFP keeps gathering tasks forming a square part of C, which provides better locality. Here are the percentages of average improvement of HFP over the other heuristics:

Reference algorithm	Eager	MST	RCM	DMDAR	HFP only flip	HFP only ready	HFP only Belady
Improvement	79.4%	48.1%	16.2%	11.0%	2.0%	1.9%	2.7%

As the 3D matrix multiplication already exhibits a better data locality than the 2D multiplication, the differences in performance between heuristics is less pronounced than on Figure 3a, but HFP is still better on average. It is worth noticing that HFP without the Belady rule gets higher performance than RCM and DMDAR, even if it triggers a larger number of transfers. The latter heuristics indeed tend to periodically require a sudden burst of data loads, while HFP tends to require loads that are nicely distributed over time, and thus well overlapped with computation. We however notice that HFP without *Ready* gets a number of transfers very close to the PCI bus limit in the 3014 MB working set case, which translates into lower performance. We can also see on Figure 4b that the Belady rule significantly reduces the quantity of data transfers.

5 Conclusion and Future Work

To take the best performance out of GPUs, it is crucial to avoid moving data as much as possible. We provided in this paper a formalization of the problem of ordering independent tasks sharing input data in order to minimize the amount of data transfers, and showed that this problem is NP-complete. We also exhibited an optimal eviction scheme, based on Belady's rule. We adapted three heuristics for the ordering problem, based on the state of the art, and compared them with a new algorithm gathering tasks with similar input data into packages of increasing size, called HFP. We also present an improvement of HFP based on package flipping. All four ordering strategies have been implemented in the StarPU runtime and tested on various sets of tasks. In all cases, the proposed HFP heuristic provides significant speedups. For instance, it allows on average a 8.3% (resp. 11%) improvement over the most advanced StarPU scheduler for 2D (resp. 3D) matrix multiplication. 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. The Belady rule reduces drastically the number of data transfers. Without this rule, HFP may entail much more data transfers than other heuristics, but achieves better performance, which shows that HFP is also good at distributing data transfer over time to increase transfer/computation overlap. Studying this final problem (minimizing computation time with overlap) is one of our future directions. We also plan to focus on the very beginning of the execution, where it is crucial

to first schedule tasks with few input data. Optimizing the implementation of Belady's rule and adapting it to the *Ready* dynamic task reordering will allow to integrate it in native executions. 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. Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In: 16th International Conference on Parallel and Distributed Systems. Shangai, China (Dec 2010)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23 (2011)
3. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2) (1966)
4. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.: PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. Computing in Science and Engineering 15(6), 36–45 (Nov 2013)
5. Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. Journal of Parallel and Distributed Computing 74(10) (Jun 2014)
6. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th National Conference. ACM '69 (1969)
7. Denning, P.J.: The working set model for program behavior. Communications of the ACM 11(5), 323–333 (1968)
8. Gonthier, M., Marchal, L., Thibault, S.: Locality-Aware Scheduling of Independant Tasks for Runtime Systems. Research report, Inria (2021), `https://hal.inria.fr/hal-03144290`
9. Kaya, K., Uçar, B., Aykanat, C.: Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories. J. Parallel Distributed Comput. 67(3) (2007)
10. Yoo, R.M., Hughes, C.J., Kim, C., Chen, Y.K., Kozyrakis, C.: Locality-aware task management for unstructured parallelism: A quantitative limit study. In: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (2013)