Parallel scheduling of DAGs under memory constraints

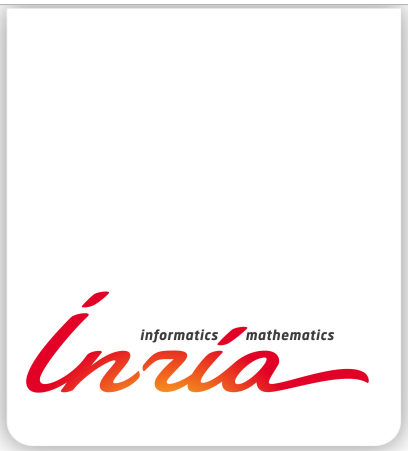
Loris Marchal, Hanna Nagy, Bertrand Simon, Frédéric Vivien

Parallel scheduling of DAGs under memory constraints

Loris Marchal, Hanna Nagy, Bertrand Simon, Frédéric Vivien

RESEARCH REPORT
N° 9108
October 2017
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9108--FR+ENG



Parallel scheduling of DAGs under memory constraints

Loris Marchal*, Hanna Nagy†, Bertrand Simon‡, Frédéric Vivien§

Project-Team ROMA

Research Report n° 9108 — October 2017 — 37 pages

Abstract: Scientific workflows are frequently modeled as Directed Acyclic Graphs (DAG) of tasks, which represent computational modules and their dependencies in the form of data produced by a task and used by another one. This formulation allows the use of runtime systems which dynamically allocate tasks onto the resources of increasingly complex computing platforms. However, for some workflows, such a dynamic schedule may run out of memory by exposing too much parallelism. This paper focuses on the problem of transforming such a DAG to prevent memory shortage, and concentrates on shared memory platforms. We first propose a simple model of DAGs which is expressive enough to emulate complex memory behaviors. We then exhibit a polynomial-time algorithm that computes the maximum peak memory of a DAG, that is, the maximum memory needed by any parallel schedule. We consider the problem of reducing this maximum peak memory to make it smaller than a given bound by adding new fictitious edges, while trying to minimize the critical path of the graph. After proving this problem NP-complete, we provide an ILP solution as well as several heuristic strategies that are thoroughly compared by simulation on synthetic DAGs modeling actual computational workflows. We show that on most instances we are able to decrease the maximum peak memory at the cost of a small increase in the critical path, thus with little impact on quality of the final parallel schedule.

Key-words: Scheduling, Task graph, Bounded Memory

* Loris Marchal is with CNRS, France.
† Hanna Nagy is with Technical University of Cluj-Napoca.
‡ Bertrand Simon is with ENS de Lyon, France.
§ Frédéric Vivien is with Inria, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Ordonnancement parallèle de DAGs sous contraintes mémoire

Résumé : Les applications de calcul scientifique sont souvent modélisées par des graphes de tâches orientés acycliques (DAG), qui représentent les tâches de calcul et leurs dépendances, sous la forme de données produites par une tâche et utilisées par une autre. Cette formulation permet l'utilisation d'API qui allouent dynamiquement les tâches sur les ressources de plateformes de calcul hétérogènes de plus en plus complexes. Cependant, pour certaines applications, un tel ordonnancement dynamique peut manquer de mémoire en exploitant trop de parallélisme. Cet article porte sur le problème consistant à transformer un tel DAG pour empêcher toute pénurie de mémoire, en se concentrant sur les plateformes à mémoire partagée. On propose tout d'abord un modèle simple de graphe qui est assez expressif pour émuler des comportements mémoires complexes. On expose ensuite un algorithme polynomial qui calcule le pic mémoire maximum d'un DAG, qui représente la mémoire maximale requise par tout ordonnancement parallèle. On considère ensuite le problème consistant à réduire ce pic mémoire maximal pour qu'il devienne plus petit qu'une borne donnée en rajoutant des arêtes fictives, tout en essayant de minimiser le chemin critique du graphe. Après avoir prouvé ce problème NP-complet, on fournit un programme linéaire en nombres entiers le résolvant, ainsi que plusieurs stratégies heuristiques qui sont minitieusement comparées-sur des graphes synthétiques modélisant des applications de calcul réelles. On montre que sur la plupart des instances, on arrive à diminuer le pic mémoire maximal, au prix d'une légère augmentation du chemin critique, et donc avec peu d'impact sur la qualité de l'ordonnancement parallèle final.

Mots-clés : Ordonnancement, Graphe de tâches, Mémoire limitée

1 Introduction

Parallel workloads are often described by Directed Acyclic task Graphs, where nodes represent tasks and edges represent dependencies between tasks. The interest of this formalism is twofold: it has been widely studied in theoretical scheduling literature [10] and dynamic runtime schedulers (e.g., StarPU [2], XKAAPI [13], StarSs [22], and PaRSEC [5]) are increasingly popular to schedule them on modern computing platforms, as they alleviate the difficulty of using heterogeneous computing platforms. Concerning task graph scheduling, one of the main objectives that have been considered in the literature consists in minimizing the makespan, or total completion time. However, with the increase of the size of the data to be processed, the memory footprint of the application can have a dramatic impact on the algorithm execution time, and thus needs to be optimized [23, 1]. This is best exemplified with an application which, depending on the way it is scheduled, will either fit in the memory, or will require the use of swap mechanisms or *out-of-core* execution. There are few existing studies that take into account memory footprint when scheduling task graphs, as detailed below in the related work section.

Our focus here concerns the execution of highly-parallel applications on a shared-memory platform. Depending on the scheduling choices, the computation of a given task graph may or may not fit into the available memory. The goal is then to find the most suitable schedule (e.g., one that minimizes the makespan) among the schedules that fit into the available memory. A possible strategy is to design a static schedule before the computation starts, based on the predicted task durations and data sizes involved in the computation. However, there is little chance that such a static strategy would reach high performance: task duration estimates are known to be inaccurate, data transfers on the platform are hard to correctly model, and the resulting small estimation errors are likely to accumulate and to cause large delays. Thus, most practical schedulers such as the runtime systems cited above rely on *dynamic* scheduling, where task allocations and their execution order are decided at runtime, based on the system state.

The risk with dynamic scheduling, however, is the simultaneous scheduling of a set of tasks whose total memory requirement exceeds the available memory, a situation that could induce a severe performance degradation. Our aim is both to enable dynamic scheduling of task graphs with memory requirements and to guarantee that at no time during the execution the available memory is exceeded. We achieve this goal by adding fictitious dependencies in the graph to cope with memory constraints: these additional edges will restrict the set of valid schedules and in particular forbid the concurrent execution of too many memory-intensive tasks. This idea is inspired by [24], which applies a similar technique to graphs of smaller-grain tasks. The main difference with the present study is that they focus on homogeneous data sizes: all the data have size 1, which is also a classical assumption in instruction graphs produced by the com-

pilation of programs. On the contrary, our approach is designed for larger-grain tasks appearing in scientific workflows whose sizes are highly irregular.

The rest of the paper is organized as follows:

- We first briefly review the existing work on memory-aware task graph scheduling (Section 2).
- We propose a very simple task graph model which both accurately describes complex memory behaviors and is amenable to memory optimization (Section 3).
- We introduce the notion of the maximum peak memory of a workflow: this is the maximum peak memory of any (sequential or) parallel execution of the workflow. We then show that the maximum peak memory of a workflow is exactly the weight of a special cut in this workflow, called the maximum topological cut. Finally, we propose a polynomial-time algorithm to compute this cut (Section 4).
- In order to cope with limited memory, we formally state the problem of adding edges to a graph to decrease its maximum peak memory, with the objective of not harming too much the makespan of any parallel execution of the resulting graph. We prove this problem NP-hard and propose both an ILP formulation and several heuristics to solve it on practical cases (Section 5). Finally we evaluate the heuristics through simulations on synthetic task graphs produced by classical random workflow generators (Section 6). The simulations show that the two best heuristics have a limited impact on the makespan in most cases, and one of them is able to handle all studied workflows.

2 Related Work

Memory and storage have always been a limited parameter for large computations, as outlined by the pioneering work of Sethi and Ullman [27] on register allocation for task graphs. It was later translated to the problem of scheduling a task graph under memory or storage constraints for scientific workflows whose tasks require large I/O data. Such workflows arise in many scientific fields, such as image processing, genomics, and geophysical simulations. The problem of task graphs handling large data has been identified by Ramakrishnan et al. [23] who introduce clean-up jobs to reduce the memory footprint and propose some simple heuristics. Their work was continued by Bharathi et al. [4] who develop genetic algorithms to schedule such workflows. This problem also arises in sparse direct solvers, as highlighted by Agullo et al. [1] who study the effect of processor mapping on memory consumption for multifrontal methods. In some cases, such as for sparse direct solvers, the task graph is a tree, for which specific methods have been proposed, both to reduce the minimum peak memory [20] and to design memory-aware parallel schedulers [3].

As explained in the introduction, our study is inspired by the work of Sbîrlea at al. [24]. This study focuses on a different model, in which all data have the same size. They target smaller-grain tasks in the Concurrent Collections (CnC) programming model [6], a stream/dataflow programming language. Their objective is, as ours, to schedule a DAG of tasks under a limited memory. For this, they associate a color to each memory slot and then build a coloring of the data, in which two data with the same color cannot coexist. If the number of colors is not sufficient, additional dependency edges are introduced to prevent two data to coexist. These additional edges respect a pre-computed sequential schedule to ensure acyclicity. An extension to support data of different sizes is proposed, which conceptually allocates several colors to a single data, but is only suited for a few distinct sizes.

In the realm of runtime systems, memory footprint is a real concern. In StarPU, attempts have been made to reduce memory consumption by throttling the task submission rate [25].

Compared to the existing work, the present work studies graphs with arbitrary data sizes, and it formally defines the problem of transforming a graph to cope with a strong memory bound: this allows the use of efficient dynamic scheduling heuristics at runtime with the guarantee to never exceed the memory bound.

3 Problem modeling

3.1 Formal description

As stated before, we consider that the targeted application is described by a workflow of tasks whose precedence constraints form a DAG $G = (V, E)$. Its nodes $i \in V$ represent tasks and its edges $e \in E$ represent precedence, in the form of input and output data. The processing time necessary to complete a task $i \in V$ is denoted by w_i. In our model, the memory usage of the computation is modeled only by the size of the data produced by the tasks and represented by the edges. Therefore, for each edge $e = (i, j)$, we denote by m_e or $m_{i,j}$ the size of the data produced by task i for task j. We assume that G contains a single source node s and a single sink node t; otherwise, one can add such nodes along with the appropriate edges, all of null weight. For the sake of simplicity, we define the following sizes of inputs and outputs of a node i:

$$Inputs(i) = \sum_{j|(j,i)\in E} m_{j,i} \qquad Outputs(i) = \sum_{j|(i,j)\in E} m_{i,j}$$

We propose here to use a very simple memory model, which might first seem unrealistic, but will indeed prove itself very powerful both to model complex memory behaviors and to express the peak memory usage. In the proposed model, at the beginning of the execution of a task i, all input data of i are immediately deleted from the memory, while all its output data are allocated to the

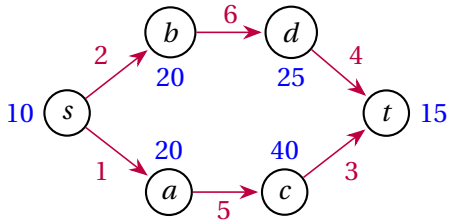


Figure 1: Example of a workflow, (red) edge labels represent the size $m_{i,j}$ of associated data, while (blue) node labels represent their computation weight w_i.

memory. That is, the amount of used memory M_{used} is transformed as follows:

$$M_{\text{used}} \leftarrow M_{\text{used}} - \mathit{Inputs}(i) + \mathit{Outputs}(i)\,.$$

This model, called the SIMPLEDATAFLOWMODEL, is extremely simple, and in particular does not allow a task to have both its inputs and outputs simultaneously in memory. However, we will see right below that it is expressive enough to emulate other complex and more realistic behaviors.

Before considering other memory models, we start by defining some terms and by comparing sequential schedules and parallel execution of the graph. We say that the data associated to the edge (i, j) is *active* at a given time if the execution of i has started but not the one of j. This means that this data is present in memory. A *sequential schedule* of a DAG G is defined by an order of its tasks. The *memory used* by a sequential schedule at a given time is the sum of the sizes of the active data. The *peak memory* of such a schedule is the maximum memory used during its execution. A *parallel execution* of a graph on p processors is defined by:

- An allocation μ of the tasks onto the processors (task i is computed on processor $\mu(i)$);
- The starting times σ of the tasks (task i starts at time $\sigma(i)$).

As usual, a valid schedule ensures that data dependencies are satisfied ($\sigma(j) \geq \sigma(i) + w_i$ whenever $(i, j) \in E$) and that processors compute a single task at each time step (if $\mu(i) = \mu(j)$, then $\sigma(j) \geq \sigma(i) + w_i$ or $\sigma(i) \geq \sigma(j) + w_j$). Note that when considering parallel execution, we assume that all processors use the same shared memory, whose size is limited.

A very important feature of the proposed SIMPLEDATAFLOWMODEL is that there is no difference between *sequential schedules* and *parallel execution* as far as memory is concerned, which is formally stated in the following theorem.

Theorem 1. *For each parallel execution (μ, σ) of a DAG G, there exists a sequential schedule with equal peak memory.*

Proof. We consider such a parallel execution, and we build the corresponding sequential schedule by ordering tasks in non decreasing starting time. Since in the SIMPLEDATAFLOWMODEL, there is no difference in memory between a task being processed and a completed task, the sequential schedule has the same amount of used memory as the parallel execution after the beginning of each task. Thus, they have the same peak memory. □

This feature will be very helpful when computing the maximum memory of any parallell execution, in Section 4: thanks to the previous result, it is equivalent to computing the peak memory of a sequential schedule.

3.2 Emulation of other memory models

3.2.1 Classical workflow model

As we explained above, our model does not allow inputs and outputs of a given task to be in memory simultaneously. However, this is a common behavior, and some studies, such as [16], even consider that in addition to inputs and outputs, some temporary data t_i has to be in memory when processing task i. The memory needed for its processing is then $Inputs(i) + t_i + Outputs(i)$. Although this is very different to what happens in the proposed SIMPLEDATAFLOWMODEL, such a behavior can be simply emulated, as illustrated on Figure 2. For all task i, we split it into two nodes i_1 and i_2. We transform all edges (i, j) by edges (i_2, j), and edges (k, i) by edges (k, i_1). We also add an edge (i_1, i_2) with an associated data of size $Inputs(i) + t_i + Outputs(i)$. Task i_1 represents the allocation of the data needed for the computation, as well as the computation itself, and its work is thus $w_{i_i} = w_i$. Task i_2 stands for the deallocation of the input and temporary data and has work $w_{i_2} = 0$.

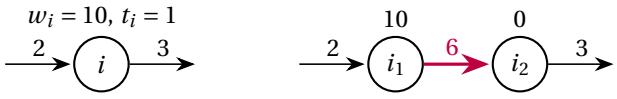


Figure 2: Transformation of a task as in [16] (left) to the SIMPLEDATAFLOWMODEL (right).

3.2.2 Shared output data

Our model considers that each task produces a separate data for every of its successors. However, it may well happen that a task i produces an output data d, of size $o_{i,d}$, which is then used by several of its successors, and can be freed after the completion of these successors. The output data is then shared among successors, contrarily to what is considered in the SIMPLEDATAFLOWMODEL. Any task can then produce several output data, some of which can be shared among

several successors. Again, such a behavior can easily be emulated in the proposed model, as illustrated on Figure 3.

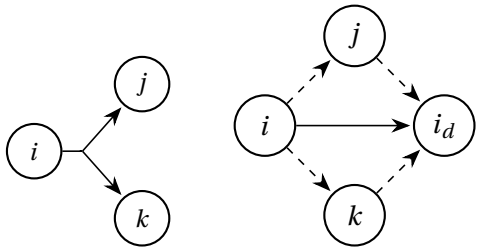


Figure 3: Transformation of a task with a single shared output data into SIMPLEDATAFLOWMODEL. The plain edge carries the shared data size, while dashed edges have null size.

Such a task i with a shared output data will first be transformed as follows. For each shared output data d of size $o_{i,d}$, we add a task i_d which represents the deallocation of the shared data d (and thus has null computation time w_{i_d}). An edge of size $o_{i,d}$ is added between i and i_d: $m_{i,i_d} = o_{i,d}$. Data dependency to a successor j sharing the output data d is represented by an edge (i, j) with null data size ($m_{i,j} = 0$) (if it does not already exist, due to an other data produced by i and consumed by j). Finally, for each such successor j, we add an edge of null size (j, i_d) to ensure that the shared data will be freed only when it has been used by all the successors sharing it. The following result states that after this transformation, the resulting graph correctly models the memory behavior.

Theorem 2. *Let G be a DAG with shared output data, and G' its transformation into* SIMPLEDATAFLOWMODEL. *There exists a schedule σ of G with peak memory M if and only if there exists a schedule σ' of G' with peak memory at most M.*

Proof. First, consider a schedule σ which executes the graph G with a memory of size M. We assume that σ frees shared output data as soon as possible (otherwise we first transform it into a schedule freeing shared output data as soon as possible, which does not increase the peak memory). We transform σ into a schedule σ' of G. When σ schedules a node i of G, σ' schedules the same node i of G'. When σ frees a shared data d output by node i, σ' schedules node i_d. We now show by induction on σ that σ' is a valid schedule on G' and that both schedules use the same amount of memory at any time. Suppose σ' valid for the first k operations of σ, and consider the following one. When σ schedules a node i of G, σ' schedules the same node of G'. Its predecessors are then completed. The sum of the sizes of the output data of i in G and G' are equal, as when the transformation removes a shared output data, it adds a single edge of the same size, along with null-weight edges. If a shared data d output by a task j is freed in σ, then task j_d is executed in σ', which reduces the memory consumption by the size of the data d. Therefore, σ and σ' have the same memory consumption for an additional operation. By induction, we get the result.

Now, suppose there exists a schedule σ' of G' with a peak memory equal to M. We transform σ' into a schedule σ of G. When σ' schedules a node i of G', σ schedules the same node i of G. When σ' schedules node i_d of G', σ frees the shared data d output by node i. As in the previous case, We now show by induction on σ' that σ is a valid schedule on G and that both schedules use the same amount of memory at any time. Suppose σ valid for the first k operations of σ', and consider the following one. If σ' schedules a node i of G', σ schedules the same node of G. As previously, the precedence is respected, and the memory consumed is the same in both schedules. If σ' schedules a node i_d of G', σ frees the shared data d output by task i. The nodes consuming this data are completed, so this operation is authorized, and the memory consumption is reduced by the size of data d in both cases. By induction, we get the result.

□

3.2.3 Pebble game

One of the pioneer work dealing with the memory footprint of a DAG execution has been conducted by Sethi [26]. He considered what is now recognized as a variant of the PEBBLEGAME model. We now show that the proposed SIMPLEDATAFLOWMODEL is an extension of PEBBLEGAME. The pebble game is defined on a DAG as follows:

- A pebble can be placed on a node with no predecessor at any time;
- A pebble can be placed on a node if all its predecessors have a pebble;
- A pebble can be removed from a node at any time;
- A pebble cannot be placed on a node that has been previously pebbled.

The objective is to pebble all the nodes of a given graph, using a minimum number of pebbles. Note that the pebble of a node should be removed only when all its successors are pebbled. This is the main difference with our model, where a node produces a different output data for each of its successors. Thus, the PEBBLEGAME model ressembles the model with shared output data presented above, with all data of size one. We thus apply the same transformation and consider that a pebble is a shared output data used for all the successors of a node. In addition, we add fictitious successor to all nodes without successors. Hence, the pebble placed on such a node can be considered as the data consumed by this successor. Then, we are able to prove that the memory behavior of the transformed graph under SIMPLEDATAFLOWMODEL corresponds to the pebbling of the original graph, as outlined by the following theorem.

Theorem 3. *Let P be DAG representing and instance of a* PEBBLEGAME *problem, and G its transformation into* SIMPLEDATAFLOWMODEL. *There exists a pebbling scheme τ of P using at most B pebbles if and only if there exists a schedule σ' of G' with peak memory at most B.*

Proof. In the PEBBLEGAME model, we can consider that every node outputs a single data of size one consumed by all its successors. Recall that for a node with no successor, the transformation acts as if a fictitious successor existed for this node. Therefore, the transformation adds one node for each node u in the graph P. In order to clarify whether we consider the graph P or G, we call u_1 the node of G that corresponds to the node u of P and u_2 the node of G that corresponds to the data output by node u of P. If u has no successor in P, we denote f_u the fictitious node added in G, successor of u_1 and predecessor of u_2.

First, we consider a traversal τ which traverses P with B pebbles. We transform τ into a schedule σ of G: when τ pebbles a node u of P, σ executes the node u_1 of G, when τ removes a pebble from a node u of P, σ executes the node u_2 of G. If u has no successor in P, σ first executes f_u then u_2. We now show by induction on τ that σ is a valid schedule on G. Suppose σ valid for the first k operations of τ, and consider the following one. If τ pebbles a new node u, this means that u_1 was not executed before by σ, as recomputations are forbidden, and that all the predecessors v^i of u have been pebbled by τ, so that the nodes v_1^i have been executed by σ in G. These nodes v_1^i correspond to the predecessors of u_1 in G, so the execution of u_1 is valid. If τ unpebbles a node u, this means that its successors v^i have already been pebbled. Indeed, otherwise, as recomputations are not allowed, τ would not be a valid schedule. Therefore, as the predecessors of u_2 in G are u_1 and the v_1^i, these nodes have been executed by σ, so the execution of u_2 is valid. If u has no successor in P, then the execution f_u and u_2 is valid. Finally, σ is a valid schedule. At any time, the memory used by σ is equal to the numbers of nodes u of P such that u_1 is executed but not u_2. This is equal to the number of pebbles required by τ, so σ is a valid schedule of G using a memory of size B.

Now, we consider a schedule σ of G with a peak memory equal to B. We transform σ into a traversal τ of P: when σ executes a node u_1, τ pebbles the node u, and when σ executes a node u_2, τ removes the pebble of node u. Nothing is done when σ executes a node f_u. We now show by induction on σ that τ is a valid traversal of P. Suppose τ valid for the first k operations of σ, and consider the following one. First, suppose that σ executes a node u_1. Let v^i be the predecessors of u in P. By the precedence constraints, we know that the nodes v_1^i have already been executed by σ, and that the nodes v_2^i have not. Therefore, τ has pebbled the nodes v^i, but have not unpebbled them. So τ is allowed to pebble u. Now, suppose that σ executes a node u_2. The node u_1 has already been pebbled by the precedence constraints, so removing this pebble is a valid move. Therefore, τ is a valid traversal of P. As above, at any time, the memory used by σ is equal to the numbers of nodes u of P such that u_1 is executed but not u_2. This is equal to the numbers of pebbles used by τ. □

3.3 Peak memory minimization in the proposed model

The emulation of the PEBBLEGAME problem, as proposed above, allows us to formally state the complexity of minimizing the memory of a DAG, as expressed by the following theorem.

Theorem 4. *Deciding whether an instance of* SIMPLEDATAFLOWMODEL *can be scheduled with a memory of limited size is NP-complete.*

Proof. The problem of deciding whether an instance of PEBBLEGAME can be traversed with a given number of pebbles is NP-complete [26]. Then, thanks to Theorem 3, we know that an instance of PEBBLEGAME can be transformed into an instance of SIMPLEDATAFLOWMODEL (with twice as many nodes), which then inherits of this complexity result. □

4 Computing the maximal peak memory

In this section, we are interested in computing the *maximal peak memory* of a given DAG $G = (V, E)$, that is, the largest peak memory that can be reached by a sequential schedule of G. Our objective is to check whether a graph can be safely executed by a dynamic scheduler without exceeding the memory bound.

We first define the notion of *topological cut*. We recall that G contains a single source node s and a single sink node t.

Definition 1. *A topological cut* (S, T) *of a DAG* G *is a partition of* G *in two sets of nodes* S *and* T *such that* $s \in S$, $t \in T$, *and no edge is directed from a node of* T *to a node of* S. *An edge* (i, j) *belongs to the cut if* $i \in S$ *and* $j \in T$. *The weight of a topological cut is the sum of the weight of the edges belonging to the cut.*

For instance, in the graph of Figure 1, the cut $(\{s, a, b\}, \{c, d, t\})$ is a topological cut of weight 11. In the SIMPLEDATAFLOWMODEL, the memory used at a given time is equal to the sum of the sizes of the active output data, which depends solely on the set of nodes that have been executed. Therefore, the maximal peak memory of a DAG is equal to the maximum weight of a topological cut.

Definition 2. *The* MAXTOPCUT *problem consists in computing a topological cut of maximum weight for a given DAG.*

We first prove that this problem is polynomial, by providing a linear program over the rationals solving it, and then propose an explicit algorithm which does not rely on linear programming.

4.1 Complexity of the problem

The MAXTOPCUT problem belongs to the family of problems in which we are interested in computing a weighted cut in a graph that optimizes some quantity.

The problem of finding a cut of *minimum* weight (when edge weights are nonnegative) has been thoroughly studied in the literature, and many polynomial-time algorithms have been proposed to solve it, both undirected and directed graphs [19]. On the opposite, computing a maximal cut is in general much more difficult. It is well-known that this problem is NP-complete on general graphs, both undirected and directed [17], and with unit weights [11]. In 2011, Lampis et al. even extend this result to DAGs [18], which are our scope of interest. However, our problem is more restrictive, as we are only interested in maximal *topological cuts* on DAGs, which means that all the edges of the cut have the same direction. This constraint actually heavily reduces the set of possible cuts. There are 2^n possible cuts for any DAG with n nodes: the number of ways to partition the nodes in two sets. However, the number of topological cuts can be much lower: only $n-1$ possibilities for a chain graph on n nodes. The problem of finding a maximal topological cut is then intuitively easier than finding a maximal cut in a DAG.

We show that MAXTOPCUT is actually polynomial by exhibiting a Linear Program solving it. This proof is adapted form [21].

Theorem 5. *The problem of finding a maximal topological cut in a DAG is polynomial.*

Proof. We consider a DAG G, where each edge (i,j) has a weight $m_{i,j}$. We assume that it has a single source vertex s and a single target vertex t (otherwise, add these nodes with null-weight edges).

We now consider the following linear program $\mathscr{P}$.

$$\max \sum_{(i,j)\in E} m_{i,j} d_{i,j} \tag{1}$$

$$\forall (i,j) \in E,\ \ d_{i,j} = p_i - p_j \tag{2}$$

$$\forall (i,j) \in E,\ \ d_{i,j} \geq 0 \tag{3}$$

$$p_s = 1 \tag{4}$$

$$p_t = 0 \tag{5}$$

Intuitively, an integer solution of $\mathscr{P}$ corresponds to a valid topological cut (S,T). The variable p_i represents the potential of vertex i: if it is equal to 1 then $i \in S$ and if it is equal to 0 then $i \in T$. Then, $d_{i,j}$ is equal to 1 if the edge (i,j) belongs to the cut (S,T) and 0 otherwise. Finally, the objective function represents the weight of the cut. However, a general solution of $\mathscr{P}$ consists of rational numbers and not integers, so does not correspond directly to a topological cut. Nevertheless, we show that for this particular program, a naive rounding algorithm exhibits a topological cut, which can then be computed in polynomial time.

Note that $\mathscr{P}$ is similar to the classic linear program computing the minimal $s-t$ cut [19]. The only differences are Equation (2) being an equality instead of an inequality, and the direction of the objective function.

We begin by proving that if G admits a topological cut of weight M, there is a solution of the linear program for which the objective function equals M. Let (S,T) be a topological cut of G. For every node i, we define $p_i = 1$ if $i \in S$ and $p_i = 0$ if $i \in T$. Then, for each edge (i,j) belonging to the cut, we have $p_i - p_j = 1$ and for the remaining edges (i,j), we have $p_i - p_j = 0$. Indeed, no edge can be directed from T to S by definition. Therefore, we have for all $(i,j) \in E$, $d_{i,j} = p_i - p_j \geq 0$ so the proposed valuation satisfies $\mathscr{P}$, and the objective function is equal to the weight of (S,T).

Now, suppose that $\mathscr{P}$ admits a valid rational solution of objective function M^*. We prove that there exists a topological cut (S^*,T^*) of G of weight at least M^*. First, note that for any edge (i,j), we have $d_{i,j} \geq 0$ so $p_i \geq p_j$. Then, every node of G belongs to a directed path from s to t by definition of s and t. Therefore, every p_i belongs to $[0,1]$. Indeed, for a given $i \in V$, let $v_1, v_2, \ldots, v_k$ be the vertices of a directed path from i to t, with $i = v_1$ and $t = v_k$. Then, we deduce that $p_i = p_{v_1} \geq p_{v_2} \geq \cdots \geq p_{v_k} = p_t = 0$. A similar proof with a path from s to i shows that p_i is not larger than 1.

In order to prove the existence of (S^*,T^*), we consider a random topological cut (S,T) defined as follows: draw w uniformly in $]0,1[$, and let (S,T) be the cut (S_w,T_w), with $S_w = \{i \mid p_i > w\}$ and $T_w = \{j \mid p_j \leq w\}$. This partition is valid as for any $i \in S_w$ and $j \in T_w$, we have $p_i > p_j$, so the edge (j,i) cannot belong to E: this would imply $d_{j,i} < 0$ which violates a constraint of $\mathscr{P}$. Now, let us compute the expected cost $M(S,T)$ of (S,T). The probability for a given edge (i,j) to belong to (S,T) is exactly $d_{i,j} = p_i - p_j$, as w is drawn uniformly in $]0,1[$ and all p_i belong to $[0,1]$. Therefore, the expected cost of (S,T) is given by

$$\begin{aligned} E\left(M(S,T)\right) &= \sum_{(i,j)\in E} m_{i,j} \Pr\big((i,j) \text{ belongs to } (S,T)\big) \\ &= \sum_{(i,j)\in E} m_{i,j} d_{i,j} = M^*. \end{aligned}$$

Therefore, there exists $w \in]0,1[$ such that $M(S_w,T_w) \geq M^*$, which proves the existence of a topological cut (S^*,T^*) of weight at least M^*. Note that an algorithm could then find such a topological cut by computing $M(S_{p_i},T_{p_i})$ for every $i \in V$.

We now show that it is not necessary, as, if M^* is the optimal objective function, then the weight of any cut (S_w,T_w) is equal to M^*. First, note that no cut (S_w,T_w) can have a weight larger than M^* by definition. So, for all w, we have $M(S_w,T_w) \leq M^*$. As $E\left(M(S,T)\right) = M^*$, we conclude that $\Pr\left(M(S,T) < M^*\right) = 0$. It remains to show that no single value of w can lead to a suboptimal cut. Assume by contradiction that there exists $w_0 \in]0,1[$ such that $M(S_{w_0},T_{w_0}) < M^*$. Let $w_1 = \min\left\{p_i \mid p_i > w_0\right\}$, which is defined as $p_t = 1 > w_0$, and consider any $w \in [w_0,w_1[$. For every $i \in V$, if $p_i > w_0$ then $p_i \geq w_1 > w$, and if $p_i \leq w_0$ then $p_i \leq w$, so, by definition of S_w and T_w, we have $(S_w,T_w) = (S_{w_0},T_{w_0})$. Therefore, we get

$$\Pr\big(M(S,T) < M^*\big) \geq \Pr\big((S,T) = (S_{w_0},T_{w_0})\big) \geq w_1 - w_0 > 0.$$

This inequality contradicts the fact that $\Pr(M(S,T) < M^*)) = 0$.

To conclude, a maximal topological cut can be computed by first solving the linear program $\mathscr{P}$ in rationals, then selecting any cut (S_w, T_w), for instance by taking $w = 1/2$.

□

4.2 Explicit algorithm

In the previous section, we have exhibited a linear program solving the MAXTOPCUT problem. We present here a more direct way of computing the maximum topological cut, through Algorithm 1. We first consider a problem related to the dual version of MAXTOPCUT, which we call MINFLOW:

Definition 3. *The* MINFLOW *problem consists in computing a flow of minimum value where the amount of flow that passes through each edge is not smaller than its weight.*

We recall that the value of a flow f is defined as $\sum_{j,\,(s,j)\in E} f_{s,j}$. In this problem the edge weights do not represent *capacities* as in a traditional flow, but rather *demands*: the minimum flow must be larger than these demands on all edges[1]. We recall that the MAXFLOW problem consists in finding a flow of maximum value where the amount of flow that passes through each edge is not larger than its weight. Its dual version, the MINCUT problem, consists in computing the st-cut (S,T) of minimum weight, where $s \in S$ and $t \in T$. Note that this cut may not be topological. See [7, Chapter 26] for more details. The MINFLOW problem is described by the following linear program.

$$\begin{aligned} \min \quad & \sum_{j \mid (s,j)\in E} f_{s,j} \\ \forall\, j \in V \setminus \{s,t\}, \quad & \left(\sum_{i \mid (i,j)\in E} f_{i,j}\right) - \left(\sum_{k \mid (j,k)\in E} f_{j,k}\right) = 0 \\ \forall (i,j) \in E, \quad & f_{i,j} \geq m_{i,j} \end{aligned}$$

We propose in Algorithm 1 an explicit algorithm to resolve the MAXTOPCUT problem. We first need an upper bound f_{max} on the value of the optimal flow solving the dual MINFLOW problem on G. We can take for instance f_{max} equal to one plus the sum of the $m_{i,j}$'s. The algorithm builds a flow f with a value at least f_{max} on all edges. Intuitively, the flow f can be seen as an optimal flow f^* solving the MINFLOW problem, on which has been added an arbitrary flow f^+. In order to compute f^* from f, the algorithm explicitly computes f^+, by solving a MAXFLOW instance on a graph G^+. Intuitively, this step consists in maximizing the flow that can be subtracted from f^*. Finally, the maximum topological cut

[1] This must not be mistaken with the demands of vertices (i.e., the value of the consumed flow) as in the Minimum Cost Flow problem.

associated to the flow f^* is actually equal to the minimum st-cut of G^+ that can be deduced from the residual network induced by f^+. We recall that the residual network of G^+ induced by f^+ contains the edge (i,j) such that either $(i,j) \in E$ and $f^+_{i,j} < m^+_{i,j}$ or $(j,i) \in E$ and $f^+_{j,i} > 0$, as defined for instance in [7, Chapter 26].

The complexity of Algorithm 1 depends on two implementations: how we compute the first flow f and how we solve the MAXFLOW problem. The rest is linear in the number of edges. Computing the starting flow f can be done by looping over all edges, finding a simple path from s to t containing a given edge, and adding a flow going through that path of value f_{max}. Note that this method succeeds because the graph is acyclic, so every edge is part of a simple path (without cycle) from s to t. This can be done in $O(|V||E|)$. Solving the MAXFLOW problem can de done in $O\big(|V||E|\log\big(|V|^2/|E|\big)\big)$ using Goldberg and Tarjan's algorithm [14]. Therefore, Algorithm 1 can be executed in time $O\big(|V||E|\log\big(|V|^2/|E|\big)\big)$.

Algorithm 1: Resolving MAXTOPCUT on a DAG G

1 Construct a flow f for which $\forall (i,j) \in E,\ f_{i,j} \geq f_{max}$, where $f_{max} = 1 + \sum_{(i,j)\in E} m_{i,j}$ (see the description)
2 Define the graph G^+ equal to G except that $m^+_{i,j} = f_{i,j} - m_{i,j}$
3 Compute an optimal solution f^+ to the MAXFLOW problem on G^+
4 $S \leftarrow$ set of vertices reachable from s in the residual network induced by f^+ ; $T \leftarrow V \setminus S$
5 **return** the cut (S,T)

Theorem 6. *Algorithm 1 solves the* MAXTOPCUT *problem.*

Proof. First, we show that the cut (S,T) is a topological cut. We have $s \in S$ and $t \in T$ by definition. We now show that no edge exist from T to S in G. By definition of S, no edge exist from S to T in the residual network, so if there exists an edge (j,i) from T to S in G, it verifies $f^+_{j,i} = 0$. We then show that every edge of G has a positive flow going through it in f^+, which proves that there is no edge from T to S.

Assume by contradiction that there exists an edge (k,ℓ) such that $f^+_{k,\ell}$ is null. Let $S_k \subset V$ be the set of ancestors of k, including k. Then, S_k contains s but not t nor ℓ as G is acyclic. Denoting $T_k = V \setminus S_k$, we get that (S_k, T_k) is a topological cut as no edge goes from T_k to S_k by definition. The weight of the cut (S_k, T_k) is at most the value of the flow f, which is $|f|$. As $f^+_{k,\ell} = 0$, the amount of flow f^+ that goes through this cut is at most $|f| - f_{k,\ell} \leq |f| - f_{max}$. Therefore, the value of f^+ verifies $|f^+| \leq |f| - f_{max}$.

Now, we exhibit a contradiction by computing the amount of flow f^+ passing through the cut (S,T). By definition of (S,T), all the edges from S to T are saturated in the flow f^+: for each edge $(i,j) \in E$ with $i \in S$ and $j \in T$, we have $f^+_{i,j} = m^+_{i,j} = f_{i,j} - m_{i,j}$. The value of the flow f^+ is equal to the amount of flow

going from S to T minus the amount going from T to S. Let $E_{S,T}$ (resp. $E_{T,S}$) be the set of edges between S and T (resp. T and S). We have the following (in)equalities:

$$\begin{aligned}
|f^+| &= \left(\sum_{(i,j)\in E_{S,T}} f^+_{i,j}\right) - \left(\sum_{(j,i)\in E_{T,S}} f^+_{j,i}\right) \\
&\geq \left(\sum_{(i,j)\in E_{S,T}} \left(f_{i,j} - m_{i,j}\right)\right) - \left(\sum_{(j,i)\in E_{T,S}} f_{j,i}\right) \\
&\geq |f| - \left(\sum_{(i,j)\in E_{S,T}} m_{i,j}\right) \quad > \quad |f| - f_{max}
\end{aligned}$$

Therefore, we have a contradiction on the value of $|f^+|$, so no edge exists from T to S and (S,T) is a topological cut.

Now, we define the flow f^* on G, defined by $f^*_{i,j} = f_{i,j} - f^+_{i,j} \geq m_{i,j}$. We show that f^* is an optimal solution to the MINFLOW problem on G. It is by definition a valid solution as $f^+_{i,j} \leq m^+_{i,j} = f_{i,j} - m_{i,j}$ so $f^*_{i,j} = f_{i,j} - f^+_{i,j} \geq f_{i,j} + m_{i,j} - f_{i,j} = m_{i,j}$. Let g^* be an optimal solution to the MINFLOW problem on G and g^+ be the flow defined by $g^+_{i,j} = f_{i,j} - g^*_{i,j}$. By definition, $g^*_{i,j} \geq m_{i,j}$ so $g^+_{i,j} \leq f_{i,j} - m_{i,j} = m^+_{i,j}$. Furthermore, we know that $g^*_{i,j} \leq f_{max}$ because there exists a flow, valid solution of the MINFLOW problem, of value $\sum_{(i,j)\in E} m_{i,j} \leq f_{max}$: simply add for each edge (i,j) a flow of value $m_{i,j}$ passing through a path from s to t containing the edge (i,j). Then, we have $g^*_{i,j} \leq f_{max} \leq f_{i,j}$ so $g^+_{i,j} \geq 0$ and g^+ is therefore a valid solution of the MAXFLOW problem on G^+, but not necessarily optimal.

So the value of g^+ is not larger than the value of f^+ by optimality of f^+, and therefore, the value of f^* is not larger than the value of g^*. Finally, f^* is an optimal solution to the MINFLOW problem on G.

Now, we show that (S,T) is a topological cut of maximum weight in G. Let (S_0,T_0) be any topological cut of G. The total amount of flow of f^* passing through the edges belonging to (S_0,T_0) is equal to the value of f^*. As for all $(i,j) \in E$ we have $f^*_{i,j} \geq m_{i,j}$, the weight of the cut (S_0,T_0) is not larger than the value of f^*. It remains to show that this upper bound is reached for the cut (S,T). By the definition of (S,T), we know that for $(i,j) \in (S,T)$, we have $f^+_{i,j} = m^+_{i,j} = f_{i,j} - m_{i,j}$. Therefore, on all these edges, we have $f^*_{i,j} = f_{i,j} - f^+_{i,j} = m_{i,j}$, so the value of the flow f^* is equal to the weight of (S,T).

Therefore, (S,T) is an optimal topological cut. □

5 Lowering the maximal peak memory of a graph

In Section 4, we have proposed a method to determine the maximal topological cut of a DAG, which is equal to the maximal peak memory of any (sequential or parallel) traversal. We now move to the problem of scheduling such a

graph within a bounded memory M. If the maximal topological cut is at most M, then any schedule of the graph can be executed without exceeding the memory bound. Otherwise, it is possible that we fail to schedule the graph within the available memory. One solution would be to provide a complete schedule of the graph onto a number p of computing resources, which never exceeds the memory. However, using a static schedule can lead to very poor performance if the task duration are even slightly inaccurate, or if communication times are difficult to predict, which is common on modern computing platforms. Hence, our objective is to let the runtime system dynamically choose the allocation and the precise schedule of the tasks, but to restrict its choices to avoid memory overflow.

In this section, we solve this problem by transforming a graph so that its maximal peak memory becomes at most M. Specifically, we aim at adding some new edges to G to limit the maximal topological cut. Consider for example the toy example of Figure 1. Its maximal topological cut has weight 11 and corresponds to the output data of tasks a and b being in memory. If the available memory is only $M = 10$, one may for example add an edge (d,a) of null weight to the graph, which would result in a maximal topological cut of weight 9 (output data of a and d). Note that on this toy example, adding this edge completely serializes the graph: the only possible schedule of the modified graph is sequential. However, this is not the case of realistic, wider graphs. We formally define the problem as follows.

Definition 4. *A* partial serialization *of a DAG $G = (V,E)$ for a memory bound M is a DAG $G' = (V,E')$ containing all the edges of G (i.e., $E \subset E'$), on which the maximal peak memory is bounded by M.*

In general, there exist many possible partial serializations to solve the problem. In particular, one might add so many edges that the resulting graph can only be processed sequentially. In order to limit the impact on parallel performance of the partial serialization, we use the critical path length as the metric. The critical path is defined as the path from the source to the sink of the DAG whose total processing time is maximum. By minimizing the increase in critical path when adding edges to the graph, we expect that we limit the impact on performance, that is, the increase in makespan when scheduling the modified graph.

We first show that finding a partial serialization of G for memory M is equivalent to finding a sequential schedule executing G using a memory of size at most M. On the one hand, given a partial serialization, any topological order is a valid schedule using a memory of size at most M. On the other hand, given such a sequential schedule, we can build a partial serialization allowing only this schedule (by adding edge (i, j) if i is executed before j). Therefore, as finding a sequential schedule executing G using a memory of size at most M is NP-complete by Theorem 4, finding a partial serialization of G for a memory bound of M is also NP-complete.

However, in practical cases, we know that the minimum memory needed to process G is smaller than M. Therefore, the need to find such a minimum memory traversal adds an artificial complexity to our problem, as it is usually easy to compute a sequential schedule not exceeding M on actual workflows. We thus propose the following definition of the problem, which includes a valid sequential traversal to the inputs.

Definition 5. *The* MINPARTIALSERIALIZATION *problem consists, given a DAG $G = (V,E)$, a memory bound M, and a sequential schedule σ of G not exceeding the memory bound, in computing a partial serialization of G for the memory bound M that has a minimal critical path length.*

5.1 Complexity analysis

We now show that the MINPARTIALSERIALIZATION problem is NP-complete. As explained above, this complexity does not come from the search of a sequential traversal with minimum peak memory. To prove this result, we first propose the following lower bound on the makespan.

Lemma 1. *Let $G = (V,E)$ be a DAG. Any schedule $\mathscr{S}$ of peak memory $M_{\mathscr{S}}$ and makespan $T_{\mathscr{S}}$ verifies:*

$$T_{\mathscr{S}} M_{\mathscr{S}} \geq \sum_{i \in V} Outputs(i)\, w_i.$$

As a corollary, if G has a maximal peak memory of $M_{\max}$, then the length T_∞ of its critical path satisfies:

$$T_\infty M_{\max} \geq \sum_{i \in V} Outputs(i)\, w_i.$$

Proof. To prove this result, we consider the function which associates to each time step the memory usage using schedule $\mathscr{S}$ at this time. Its maximum is $M_{\mathscr{S}}$ and it is define between $t = 0$ and $t = T_{\mathscr{S}}$, so the area under the curve is upper bounded by $T_{\mathscr{S}} M_{\mathscr{S}}$. Now, for each task, its output data must be in memory for at least the execution time of this task, hence $\sum_{i \in V} Outputs(i)\, w_i$ is a lower bound of the area under the curve, which proves the result. □

We now consider the decision version of the MINPARTIALSERIALIZATION problem, which amounts to finding a partial serialization of a graph G for a memory M with critical path smaller than CP, and prove that it is NP-complete.

Theorem 7. *The decision version of the* MINPARTIALSERIALIZATION *problem is NP-complete, even for independent paths of length two.*

Proof. First, this problem is in NP as given a partial serialization of a graph G for a memory bound M, one can check in polynomial time that it is valid: simply

compute its maximum peak memory (using Algorithm 1) and the length of its critical path.

To prove the problem NP-hard, we perform a reduction from 3-PARTITION, which is known to be NP-complete in the strong sense [12]. We consider the following instance $\mathscr{I}_1$ of the 3-PARTITION problem: let a_i be $3m$ integers and B an integer such that $\sum a_i = mB$. We consider the variant of the problem, also NP-complete, where $\forall i, B/4 < a_i < B/2$. To solve $\mathscr{I}_1$, we need to solve the following question: does there exist a partition of the a_i 's in m subsets $A_1, \ldots, A_m$, each containing exactly 3 elements, such that, for each A_k, $\sum_{i \in A_k} a_i = B$. We build the following instance $\mathscr{I}_2$ of our problem. We define a DAG G with $6m$ vertices denoted by u_i and v_i for $1 \leq i \leq 3m$. G contains $3m$ edges, each pair (u_i, v_i), which have weights equal to a_i. Each vertex u_i has a unit work and v_i has a null work. The memory bound is equal to B and the problem asks whether there exists a partial serialization of G for B with critical path length at most m. A schedule σ executing sequentially the pairs u_i, v_i does not exceed the memory bound B (not even $B/2$), so the instance (G, B, σ) is a valid instance of the MINPARTIALSERIALIZATION problem.

Assume first that $\mathscr{I}_1$ is solvable, let $A_1, \ldots, A_m$ be a solution. We build a solution to $\mathscr{I}_2$. Define the graph G' from the graph G with the following additional edges. For $i \in [1, m-1]$, add edges of null weight between every v_j for $a_j \in A_i$ and every u_k for $a_k \in A_{i+1}$. The critical path of G' is then equal to m. Let $S, \bar{S}$ be a topological partition of the graph G', with no edge from $\bar{S}$ to S, and C be the set of edges between S and $\bar{S}$. Assume that C contains an edge (u_j, v_j): $u_j \in S$ and $v_j \in \bar{S}$. Then let k be such that a_j and a_k do not belong to the same set A_i. There is a directed path connecting either v_j to u_k or v_k to u_j, so $(u_k, v_k) \notin C$. Therefore, as $A_1, \ldots, A_m$ is a solution to $\mathscr{I}_1$, the weight of the cut C is equal to B, so G' solves $\mathscr{I}_2$.

Now, assume that $\mathscr{I}_2$ is solvable, let G' be a partial serialization of G for B whose critical path has length T_∞ at most m. Note that the following bound due to Lemma 1 is tight on G':

$$T_\infty M_{\max} \geq \sum_{i \in V} \mathit{Outputs}(i)\, w_i.$$

Indeed, the length of the critical path verifies $T_\infty \leq m$, the maximal peak memory $M_{\max}$ verifies $M_{\max} \leq B$, for any $i \in [1, m]$ u_i has a unit weight and v_i a null one and $\mathit{Outputs}(u_i) = a_i$. Therefore, $\sum_{i \in V} \mathit{Outputs}(i)\, w_i = mB$ so $T_\infty = m$ and $M_{\max} = B$.

Let U_1 be the set of nodes u_i without predecessors in G'. There cannot be more than three nodes in U_1 because the cut $(U_1, \bar{U_1})$ would have a weight larger than B. Assume by contradiction that its weight is less than B. Consider the graph G'_1 equal to G' except that the nodes in U_1 have a null work. The critical path of G'_1 is equal to $m-1$ and in G'_1, we have $\sum_{i \in V} \mathit{Outputs}(i)\, w_i > mB - B = (m-1)B$, so the bound of Lemma 1 is violated. Therefore, the weight of the cut

$(U_1, \bar{U}_1)$ is equal to B, so U_1 is composed of three vertices that we will denote by $u_{i_1}, u_{j_1}, u_{k_1}$, and we have $a_{i_1} + a_{j_1} + a_{k_1} = B$.

Suppose by contradiction that there exists a node u_i not in U_1 such that there is no path from v_{i_1}, v_{j_1} or v_{k_1} to u_i. Then, $(U_1 \cup \{u_i\}, V \setminus (U_1 \cup \{u_i\}))$ is a topological cut of G'_1 of weight strictly larger than B, which is impossible by definition of $\mathscr{I}_2$. Therefore, in G'_1, the nodes that have no ancestors are $U_1 = \{u_{i_1}, u_{j_1}, u_{k_1}\}$, and the nodes whose ancestors belong in U_1 are $\{v_{i_1}, v_{j_1}, v_{k_1}\}$.

We can then apply recursively the same method to determine the second set U_2 of three vertices $u_{i_2}, u_{j_2}, u_{k_2}$ without ancestors of positive work in G'_1. We now define G'_2 as equal to G'_1 except that nodes of U_2 have a null work, and continue the induction.

At the end of the process, we have exhibited m disjoint sets of three elements a_i that each sum to B, so $\mathscr{I}_1$ is solvable. □

5.2 Finding an optimal partial serialization through ILP

We present in this section an Integer Linear Program solving the MINPARTIALSERIALIZATION problem. This formulation combines the linear program determining the maximum topological cut and the one computing the critical path of a given graph.

We consider an instance of the MINPARTIALSERIALIZATION problem, given by a DAG $G = (V, E)$ with weights on the edges, and a memory limit M. The sequential schedule σ respecting the memory limit is not required. First, for any $(i, j) \notin E$, we set $m_{i,j} = 0$. We furthermore assume that there is a single source vertex s and a single target vertex t, as explained above.

We first consider the $e_{i,j}$ variables, which are equal to 1 if edge (i, j) exists in the associated partial serialization, and to 0 otherwise.

$$\forall (i, j) \in V^2, \quad e_{i,j} \in \{0, 1\} \tag{6}$$

$$\forall (i, j) \in E, \quad e_{i,j} = 1 \tag{7}$$

We need to ensure that no cycle has been created by the addition of edges. For this, we compute the transitive closure of the graph: we enforce that the graph contains edge (i, j) if there is a path from node i to node j. Then, we know that the graph is acyclic if and only if it does not contain any self-loop. This corresponds to the following constraints:

$$\forall (i, j, k) \in V^3, \quad e_{i,k} \geq e_{i,j} + e_{j,k} - 1 \tag{8}$$

$$\forall i \in V, \quad e_{i,i} = 0 \tag{9}$$

Then, we use the flow variables $f_{i,j}$, in a way similar to the formulation of the MINFLOW problem. If $e_{i,j} = 1$, then $f_{i,j} \geq m_{i,j}$, and $f_{i,j}$ is null otherwise. Now, the flow going out of s is equal to the maximal cut of the partial serialization, see the proof of Theorem 6, so we ensure that it is not larger than M. Now, note that each $f_{i,j}$ can be upper bounded by M without changing the solution space.

Therefore, Equation (11) ensures that $f_{i,j}$ is null if $e_{i,j}$ is null, without adding constraints on the others $f_{i,j}$. This leads to the following inequalities:

$$\forall (i,j) \in V^2, \quad f_{i,j} \geq e_{i,j} m_{i,j} \tag{10}$$

$$\forall (i,j) \in V^2, \quad f_{i,j} \leq e_{i,j} M \tag{11}$$

$$\forall j \in V \setminus \{s,t\}, \quad \sum_{i \in V} f_{i,j} - \sum_{k \in V} f_{j,k} = 0 \tag{12}$$

$$\sum_{j \in V} f_{s,j} \leq M \tag{13}$$

This set of constraints defines the set of partial serializations of G with a maximal cut at most M. It remains to compute the length of the critical path of the modified graph, in order to formalize the objective. We use the p_i to represent the top-level of each task, that is, their earliest completion time in a parallel schedule with infinitely many processors. The completion time of task s is w_s, and the completion time of another task is equal to its processing time plus the maximal completion time of its predecessors:

$$p_s \geq w_s$$
$$\forall (i,j) \in V^2, \quad p_j \geq w_j + p_i e_{i,j}$$

The previous equation is not linear, so we transform it by using W, the sum of the processing times of all the tasks and the following constraints.

$$\forall i \in V, \quad p_i \geq w_i \tag{14}$$

$$\forall (i,j) \in V^2, \quad p_j \geq w_j + p_i - W(1 - e_{i,j}) \tag{15}$$

If $e_{i,j}$ is null, then Equation (15) is less restrictive than Equation (14) as $p_i < W$, which is expected as there is no edge (i,j) in the graph. Otherwise, we have $e_{i,j} = 1$ and the constraints on p_j are the same as above.

Finally, we define the objective as minimizing the top-level of t, which is the critical path of the graph.

$$\text{Minimize} \quad p_t \tag{16}$$

We denote $\mathscr{P}$ the resulting ILP. We now prove that there exists a solution to $\mathscr{P}$ of objective at most L if and only if there exists a partial serialization PS of G with memory bound M of critical path length at most L.

Consider a solution of $\mathscr{P}$ of objective cost at most L. Let PS be the directed graph composed of the edges (i,j) for every $i,j \in V^2$ such that $e_{i,j} = 1$. The weight of such edges is $m_{i,j}$. First, PS is acyclic. This can be shown by induction on the size of a potential cycle. No self-loop can exist as all $e_{i,i}$ are null. If a cycle contains more than one edge, Equation (8) ensures the existence of a strictly smaller cycle, while Equation (9) forbids self-loops. Then, the equations concerning $f_{i,j}$ model the MINFLOW problem already studied, and ensure that

the minimum flow is smaller than M. The only difference being that each $f_{i,j}$ is bounded by M, which is already the case in any solution. Finally, consider a critical path $(s, i_1, i_2, \ldots, i_k, t)$ of PS. The equations concerning the variables p_i ensure that $p_t \geq w_s + w_{i_1} + \cdots + w_{i_k} + w_t$. Therefore, L is not smaller than the critical path length. Therefore, PS is a partial serialization for M of critical path length at most L.

Now, consider a partial serialization PS of G for M, of critical path length at most L. We set $e_{i,j} = 1$ if and only if there exists a path from i to j in PS. This respects the acyclicity constraints as PS is a DAG by definition. The maximum peak memory of PS is at most M, therefore the maximum cut of the graph induced by the variables $e_{i,j}$ is at most M, so there exists a valuation of the variables $f_{i,j}$ satisfying the flow constraints. Finally, we set the variables p_i equal to the top-level of task i in PS:

$$\forall i \in V,\ \ p_i = w_i + \max_{j \in V} \left\{ e_{j,i} p_j \right\}.$$

This valuation satisfies the last constraints and the objective function is then equal to L.

5.3 Heuristic strategies to compute a partial serialization

We now propose several heuristics to solve the MINPARTIALSERIALIZATION problem. These heuristics are based on the same framework, detailed in Algorithm 2. The idea of the algorithm, inspired by [24], is to iteratively build a partial serialization G' from G. At each iteration, the topological cut of maximum weight is computed via Algorithm 1. If its weight is at most M, then the algorithm terminates, as the obtained partial serialization is valid. Otherwise, another edge has to be added in order to reduce the maximum peak memory. We rely on a subroutine in order to choose which edge to add. In the following, we propose four possible subroutines. If the subroutine succeeds to find an edge that does not create a cycle in the graph, we add the chosen edge to the current graph. Otherwise, the heuristic fails. Such a failure may happen if the previous choices of edges have led to a graph which is impossible to schedule without exceeding the memory.

We propose four possibilities for the subroutine $\mathscr{A}(G, M, C)$, which selects an edge to be added to G. They all follow the same structure: two vertices u_S and u_T are selected from the maximum cut $C = (S, T)$, where $u_S \in S$ and $u_T \in T$ and no path exists from u_S to u_T. The returned edge is then (u_T, u_S). For instance, in the toy example of Figure 1, only two such edges can be added: (c, b) and (d, a). Note that adding such an edge prevents C from remaining a valid topological cut, thus it is likely that the weight of the new maximum topological cut will be reduced.

We first define some classical attributes of a graph:

- The *length* of a path is the sum of the work of all the nodes is the path, including its extremities;

Algorithm 2: Heuristic for MINPARTIALSERIALIZATION

Input: DAG G, memory bound M, subroutine $\mathscr{A}$
Output: Partial serialization of G for memory M

while *G has a topological cut of weight larger than M* **do**
  Compute a topological cut $C = (S, T)$ of maximum weight using Algorithm 1
  if *the call $\mathscr{A}(G, M, C)$ returns (u_T, u_S) with no path from node u_S to node u_T* **then**
    Add edge (u_T, u_S) of weight 0 to G
  else
    return Failure
return the modified graph G

- The *bottom level* of an edge (i, j) or a node i is the length of the longest path from i to t (the sink of the graph);
- The *top level* of an edge (i, j) or a node j is the length of the longest path from s (the source of the graph) to j.

We now present the four subroutines. The MINLEVELS heuristic, as well as the two following ones, generates the set P of vertex couples $(j, i) \in T \times S$ such that no path from i to j exist. Note that P corresponds to the set of candidate edges that might be added to G. Then, it returns the couple $(u_T, u_S) \in P$ that optimizes a given metric. MINLEVELS tries to minimize the critical path of the graph obtained when adding the new edge, by preventing the creation of a long path from s to t. Thus, it returns the couple $(j, i) \in P$ that minimizes $top\_level(j) + bottom\_level(i)$.

The MAXSIZE heuristic aims at minimizing the weight of the next topological cut. Thus, it selects a couple (j, i) such that outgoing edges of i and incoming edges of j contribute a lot to the weight of the current cut. Formally, it returns the coupe $(j, i) \in P$ that maximizes $\sum_{k \in T} m_{i,k} + \sum_{k' \in S} m_{k',j}$ (considering that $m_{i,j} = 0$ if there is no edge from i to j).

The MAXMINSIZE heuristic is a variant of the previous heuristic and pursues the same objective. However, it selects a couple of vertices which both contribute a lot to the weight of the cut, by returning the couple $(j, i) \in P$ that maximizes $\min\left(\sum_{k \in T} m_{i,k}\ ,\ \sum_{k' \in S} m_{k',j}\right)$.

Finally, the last heuristic is the only one that is guaranteed to never fail. To achieve this, it relies on a sequential schedule σ of the graph that does not exceed the memory M. Such a sequential schedule needs to be precomputed, and we propose a possible algorithm below.

Given such a sequential schedule σ, this heuristic, named RESPECTORDER, always adds an edge (j, i) which is compatible with σ (i.e., such that $\sigma(j) \leq \sigma(i)$), and which is likely to have the smallest impact on the set of valid schedules for the new graph, by maximizing the distance $\sigma(i) - \sigma(j)$ from j to i in σ. Let u_T

be the node of T which is the first to be executed in σ, and u_S be the node of S which is the last to be executed in σ. First, note that u_S must be executed after u_T in σ, because otherwise, the peak memory of σ will be at least the weight of C which is a contradiction. The returned couple is then (u_T, u_S). Note that no path from u_S to u_T can exist in the graph if all the new edges have been added by this method. Indeed, all the added edges respect the order σ by definition. Then, no failure is possible, but the quality of the solution highly depends on the input schedule σ.

5.4 Computing a sequential schedule for MINLEVELS

In this section we discuss the generation of the schedule σ, which is used as an input for heuristic RESPECTORDER. By definition, this sequential schedule executes the DAG G using a memory at most M. As proven in Theorem 4, deciding if such a schedule exists is NP-complete. However, most graphs describing actual workflows exhibit a high level of parallelism, and the difficulty is not in finding a sequential schedule fitting in memory. As a consequence, we assume that a Depth First Search (DFS) schedule, which always completes a parallel branch before starting a new one, never exceeds the memory bound.

The problem with a DFS schedule is that applying RESPECTORDER using such a schedule is likely to produce a graph with a large critical path. For this objective, a Breadth First Search (BFS) schedule is more appropriate, but it is not likely to respect the memory bound.

As proposed in [24], a way to solve this problem is to "mix" DFS and BFS schedules, and tune the proportion of each one to get a schedule respecting the memory bound but still offering good opportunities for parallelism. Formally, we define the α-BFSDFS schedule, which depends on the parameter $\alpha \in [0,1]$ and two schedules, a DFS and a BFS. A 0-BFSDFS schedule is equal to the BFS and a 1-BFSDFS schedule is equal to the DFS. For a given task i, we note $DFS(i)$ and $BFS(i)$ the rank of task i according to each schedule (i.e., the number of tasks executed before task i). Then, the α-BFSDFS schedules the tasks of G in non-decreasing order of

$$\alpha DFS(i) + (1-\alpha) BFS(i).$$

The α-BFSDFS schedule respects the precedence constraints: indeed, if task i has a successor j, then i is scheduled before j in both BFS and DFS. Then, as α and $1-\alpha$ are non-negative, α-BFSDFS schedules i before j.

The idea consists in starting from the 0-BFSDFS schedule, and then to increase the α parameter until the memory of the resulting schedule is not larger than M. As we assumed that DFS (1-BFSDFS) does not exceed M, this process is guaranteed to success. In practice, we chose in the experiments to increment α by step of 0.05 until we find an appropriate schedule.

	DAGGEN dense | sparse	LIGO	MONTAGE	GENOME
Nb. of test cases	572 | 572	220	220	220
MINLEVELS	1 | 12	20	1	0
RESPECTORDER	0 | 0	0	0	0
MAXMINSIZE	2 | 5	3	0	0
MAXSIZE	6 | 12	13	0	17
ILP	26 | 102			

Table 1: Number of failures for each dataset

6 Simulation results

We now compare the performance of the proposed heuristics through simulations on synthetic DAGs. All heuristics are implemented in C++ using the igraph library.

We generated the first dataset, named DAGGEN, using the DAGGEN software [28]. Five parameters influence the generation of these DAGs. The number of nodes belongs to $\{25, 50, 100\}$. The width, which controls how many tasks may run in parallel, belongs to $\{0.2, 0.5, 0.8\}$. The regularity, which controls the distribution of the tasks between the levels, belongs to $\{0.2, 0.8\}$. The density, which controls how many edges connect two consecutive levels, belongs to $\{0.2, 0.8\}$. The jump, which controls how many levels an edge may span, belongs to $\{1, 2, 4\}$. Combining all these parameters, we obtain a dataset of 108 DAGs. This dataset has already been used to model workflows in the scheduling literature [15, 9]. We split it in two parts in the representations: the sparse DAGGEN dataset contains the DAGs with a density of 0.2 and the dense DAGGEN dataset contains the DAGs with a density of 0.8. Indeed, this parameter leads to significant differences in the results, hence the distinction.

The three other datasets represent actual applications and have been generated with the Pegasus Workflow Generator [8]. We consider three different datasets, named LIGO, MONTAGE, and GENOME, each containing 20 graphs of 100 nodes.

The sizes given for each file are incoherent, as their value changes if the file is read by several nodes. Hence, we assumed that the size of a file is the one given by the node that produced it. Several nodes can produce data which share the same name. In this case, we assumed that these data are different, which is coherent with the precedence relations. We assumed that the memory needed during the execution of a node is negligible compared to the size of the input and output data, which must be kept in memory during this process. We then apply the transformation presented in Section 3 to treat data that are shared

among several tasks, and duplicate the nodes to cope with the memory model of SIMPLEDATAFLOWMODEL.

The heuristics have been simulated for eleven memory bounds per DAG, evenly spread between two bounds. The smallest bound corresponds to the memory required for a DFS schedule, while the largest bound corresponds to the maximal peak memory of the DAG. In the results, a normalized memory of 0 corresponds to the lowest bound, while 1 corresponds to the largest bound. One may argue that the range of memory considered can be small for some graphs, and will then be of little interest. We therefore computed the ratio of the largest memory considered divided by the lowest for each graph, and we present the statistic summary in Table 2. We can see that this ratio is very high for the LIGO and GENOME dataset: finding a partial serialization achieving the lowest memory bound means that the maximal memory consumption is divided by more than 20 for most of these graphs. This ratio has a median of 6 for the MONTAGE, which is also a high potential improvement. It is lower for the sparse DAGGEN dataset, with a median of 2, and especially for the sparse DAGGEN dataset, with a median of 1.3. Note that 4 DAGs of the DAGGEN dataset have been discarded because the minimum memory equals the maximum memory.

	DAGGEN dense | sparse	LIGO	MONTAGE	GENOME
First quartile	1.2 | 1.7	21.2	5.5	20.1
Median	1.3 | 2	21.7	6.2	21.5
Third quartile	1.4 | 2.5	22.1	6.8	22

Table 2: Statistic summary of the ratio maxmem/minmem for each dataset.

In order to assess the performance of the heuristics, we first examine the critical path length of the obtained partial serialization. We first normalize each critical path by the critical path of the original graph. Therefore, for the largest memory bounds, the original graph being itself a valid partial serialization, all the normalized critical paths equal 1. When a method fails to find a solution, we say that the critical path achieved is infinite. As we focus on the statistical summary of the results, this allows to fairly compare two heuristics with different success rate, as only the outlier points are not displayed. Failure rates are reported in Table 1.

We plot the results obtained for the sparse and dense DAGGEN dataset in Figures 4 and 5 respectively. For each heuristic and memory bound, we display the 108 results as a Tukey boxplot. The box presents the median, the first and third quartiles. The whiskers extend to up to 1.5 times the box height, and points outside are plotted individually. The first trend that can be observed, is that, as expected, the lower the memory bound, the larger the critical path. The difference between the minimal and the maximal memory bound is smaller for

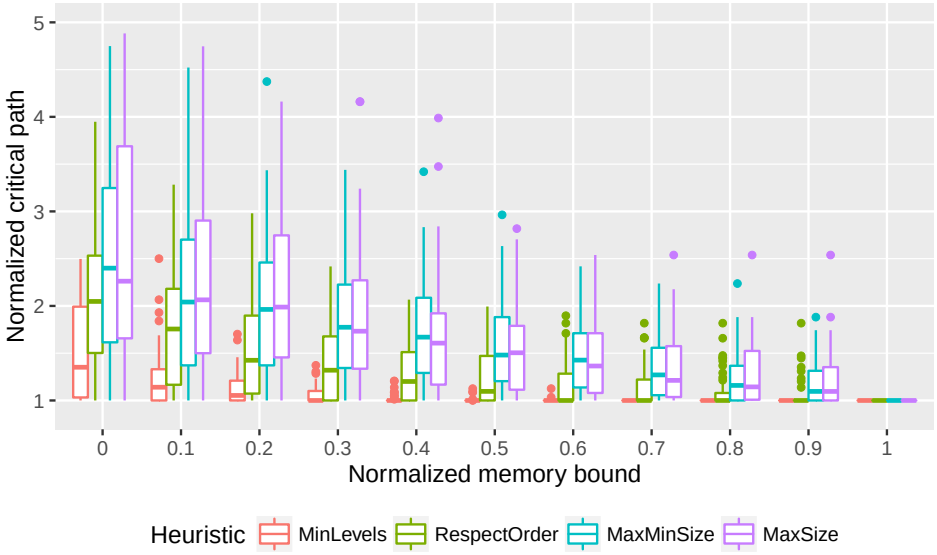


Figure 4: Critical path length obtained by each method for the sparse DAGGEN dataset.

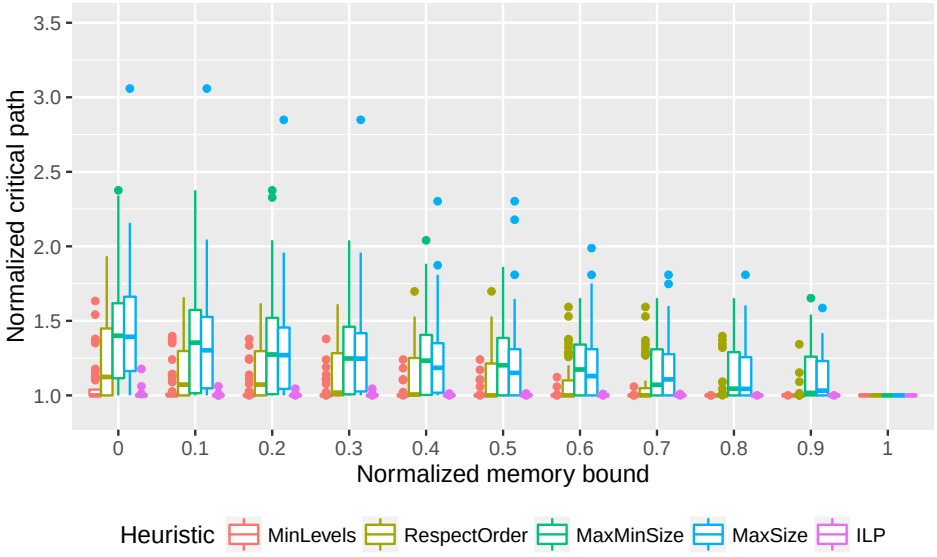


Figure 5: Critical path length obtained by each method for the dense DAGGEN dataset.

dense graphs. Therefore, it is logical that the heuristics lead to a larger increase of the critical path in sparse graphs. Comparing the heuristics, we can see that MINLEVELS clearly outperforms the other ones for any value of the memory bound. Then, RESPECTORDER obtains better performance than MAXMINSIZE and MAXSIZE, except when the memory bound is the lowest, where these three heuristics are comparable. Note that no significant difference appears when restricting the dataset to specific values of the generation parameters. The results are widely spread as the graphs differ in several parameters. We remark therefore that MINLEVELS is highly robust considering the variety of the graphs. On this dataset, we have also computed the optimal solution by using the Integer Linear Program presented in Section 5. We implemented the ILP using CPLEX with a time limit of one hour of computation on a standard laptop computer (8 cores Intel i7). When it was unable to provide a solution within the time limit, we assume a failure. This happens on sparse graphs, especially for low memory bounds, which is why it is omited on Figure 4.

The second criterion we use to compare the heuristics consists in evaluating the makespan achieved by a simple scheduling heuristic on the partial serialization returned by each heuristic on a simulated platform. The chosen scheduling heuristic is the traditional list-scheduling algorithm, in which whenever a task terminates, the available task with the highest bottom level is executed. This corresponds to the well-known HEFT scheduler [29] when adapt to dynamic schedulers, as for example done in the *dmda* scheduler of StarPU [2].

We simulated a platform of 2 processors for the dataset DAGGEN, and the results are presented in Figures 6 and 7. We can notice that the differences between the heuristics are smaller than previously, while the hierarchy is not modified. On Figure 8, we plotted for each DAG of the DAGGEN dataset and for each memory bound, the makespan obtained by each heuristic in function of the critical path obtained.

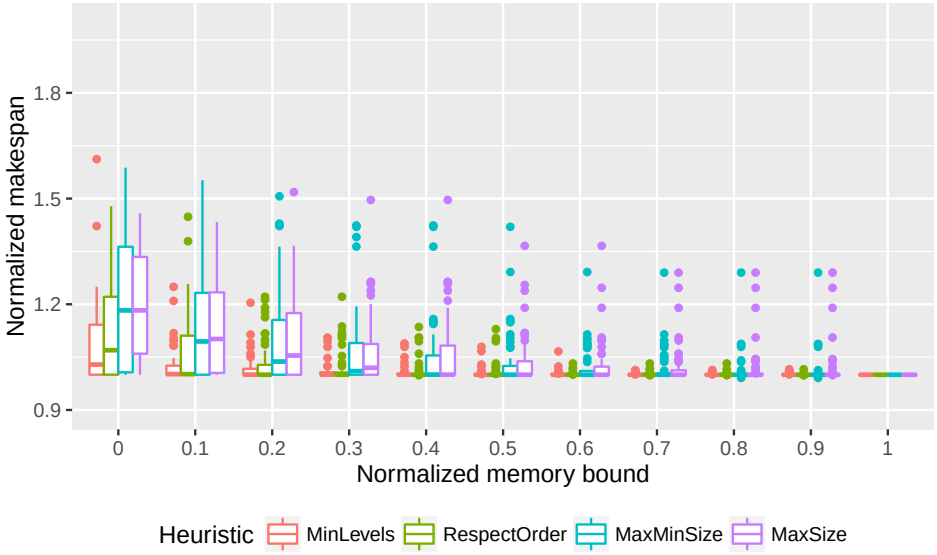


Figure 6: Makespan obtained by each method for the sparse DAGGEN dataset.

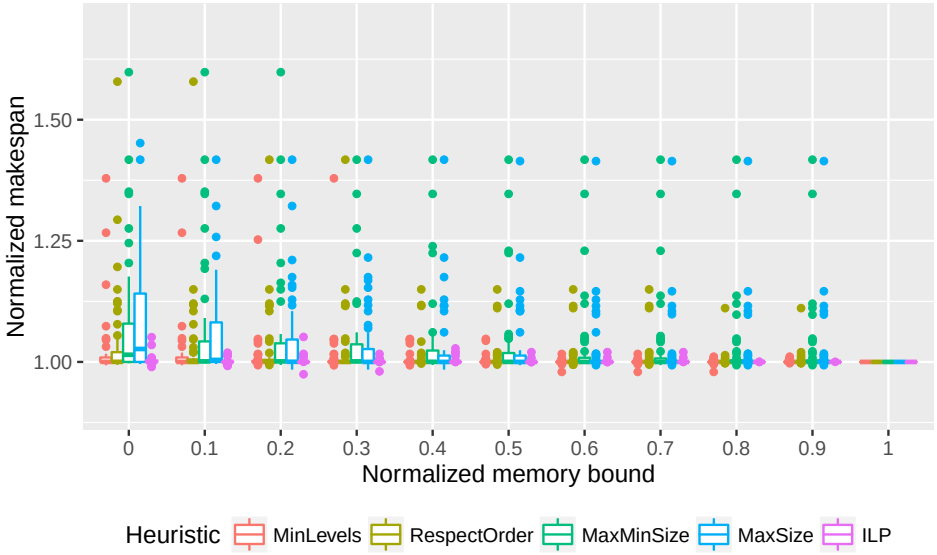


Figure 7: Makespan obtained by each method for the dense DAGGEN dataset.

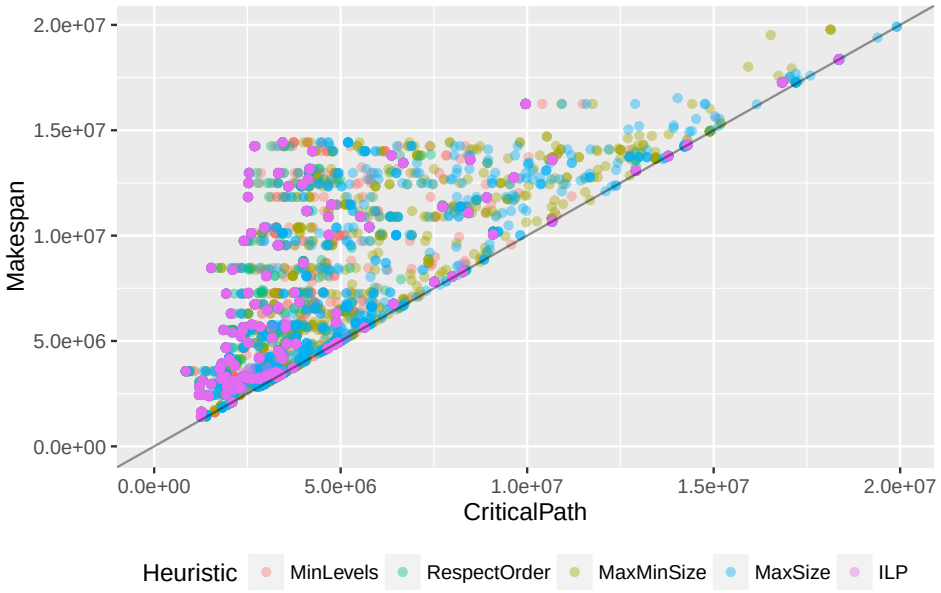


Figure 8: Makespan in function of the critical path length obtained by each method for the DAGGEN dataset.

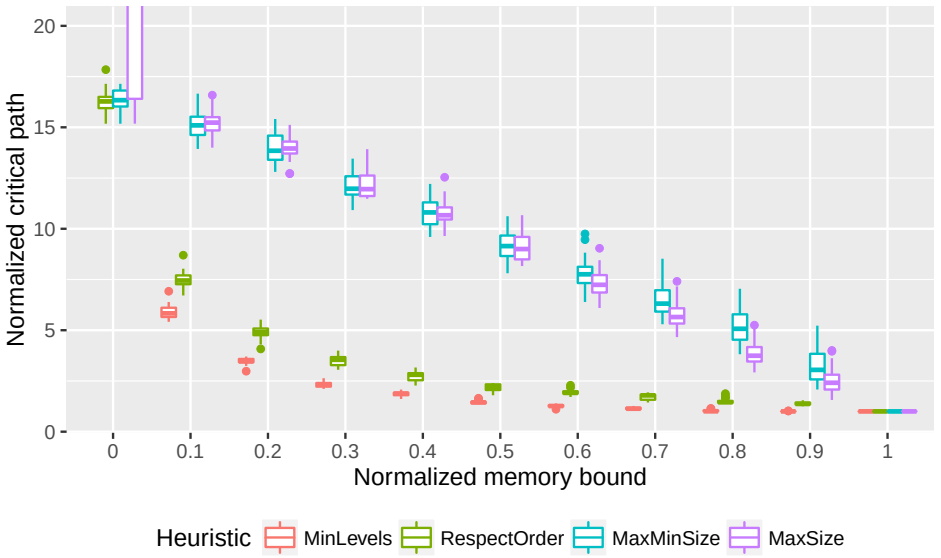


Figure 9: Critical path length obtained by each method for the LIGO dataset.

We plot the results obtained for the LIGO dataset on Figure 9, showing the critical path lengths achieved by each heuristic for each memory bound. The similar structure of all graphs in this dataset explains that the results lie in a smaller interval. The hierarchy of the heuristics is the same as in the DAGGEN dataset: MINLEVELS presents the best performance, RESPECTORDER leads to slightly longer critical paths, and MAXSIZE and MAXMINSIZE achieve similar results, several times higher than the first two heuristics. Note that for the lowest memory bound, MINLEVELS never succeeds in this dataset (hence, it does not appear in the plot), MAXSIZE also presents a high failure rate, whereas RESPECTORDER and MAXMINSIZE have comparable results.

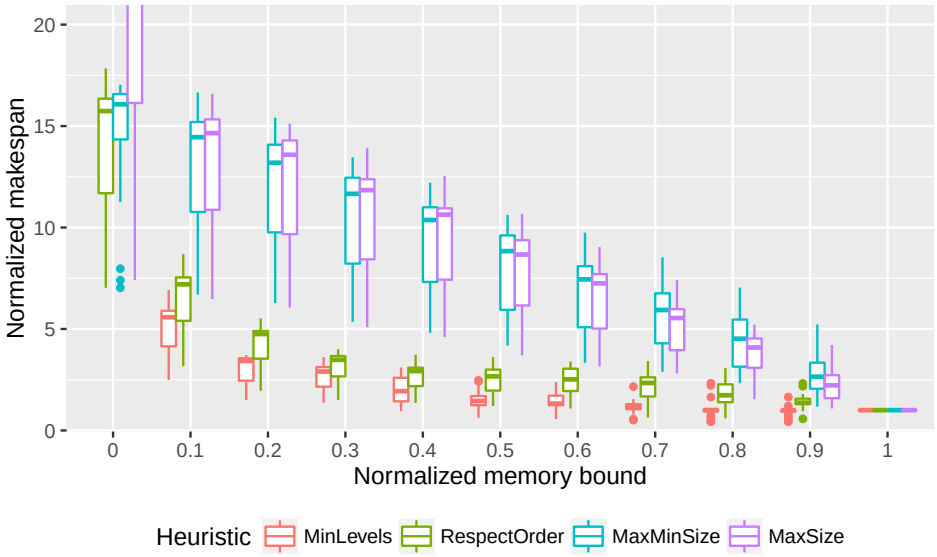


Figure 10: Makespan obtained by each method for the LIGO dataset.

Figure 10 presents the simulation on 5 processors. Except the slightly more scattered results, the ranking of the heuristics is very similar than the ones obtained with the critical path. Therefore, even if the final objective is to obtain a graph that we can schedule within a small makespan, our objective of minimizing the critical path is completely relevant.

On Figure 11, we plotted for each DAG of the LIGO dataset and for each memory bound, the makespan obtained by each heuristic in function of the critical path obtained. We can see that when the critical path achieved is large, the makespan obtained is very close to the critical path length. On the opposite, for smaller values of the critical path length, we can obtain a makespan several times higher, because the partial serialization kept more parallelism in the graph.

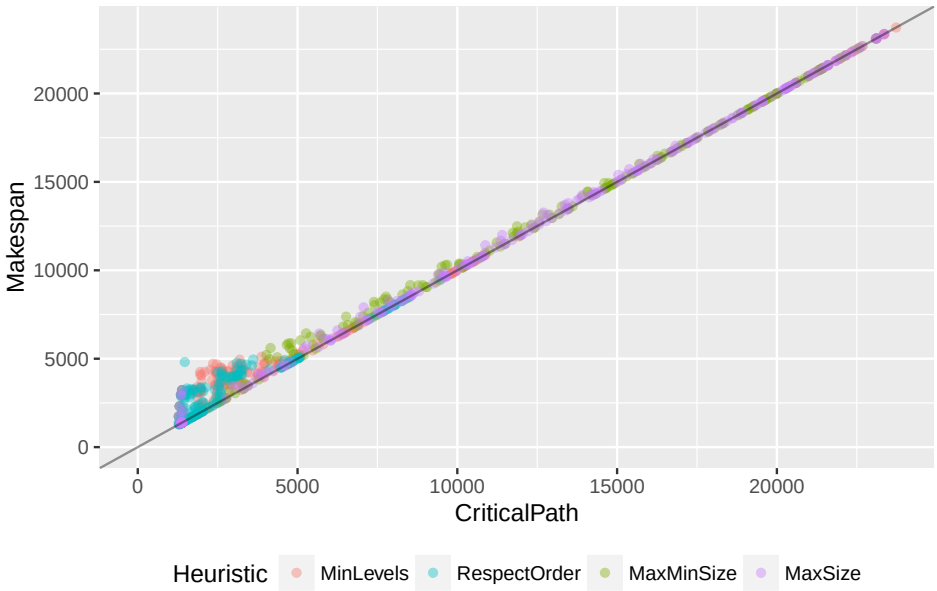


Figure 11: Makespan in function of the critical path length obtained by each method for the LIGO dataset.

In Figures 12 to 14, we present the same results for the GENOME dataset. We observe a trend similar to the results on the LIGO dataset, except that MINLEVELS never fails, even for the lowest memory bound.

In Figures 15 to 17, we present the same results for the MONTAGE dataset. We observe a trend similar to the results on the LIGO dataset, except that MINLEVELS and RESPECTORDER always present better results that the other heuristics, even for the lowest memory bound.

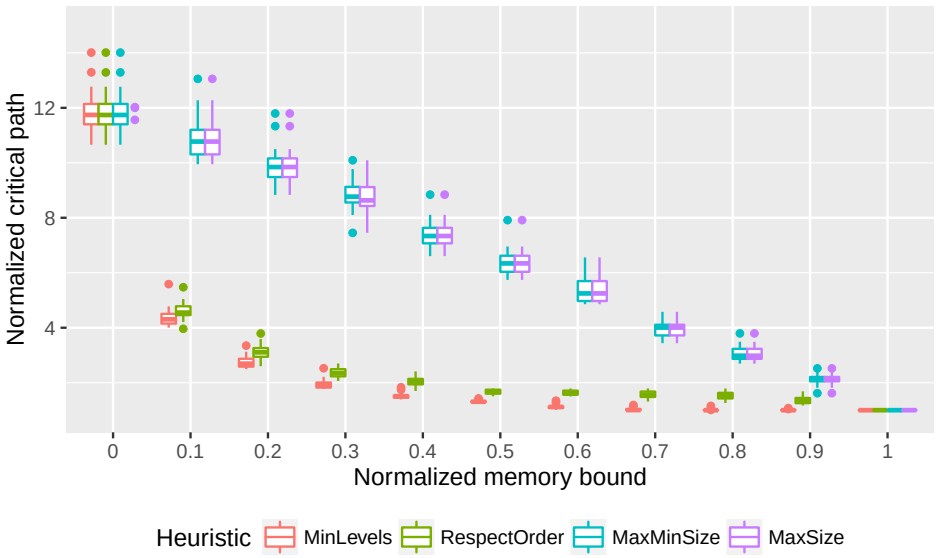


Figure 12: Critical path length obtained by each method for the GENOME dataset.

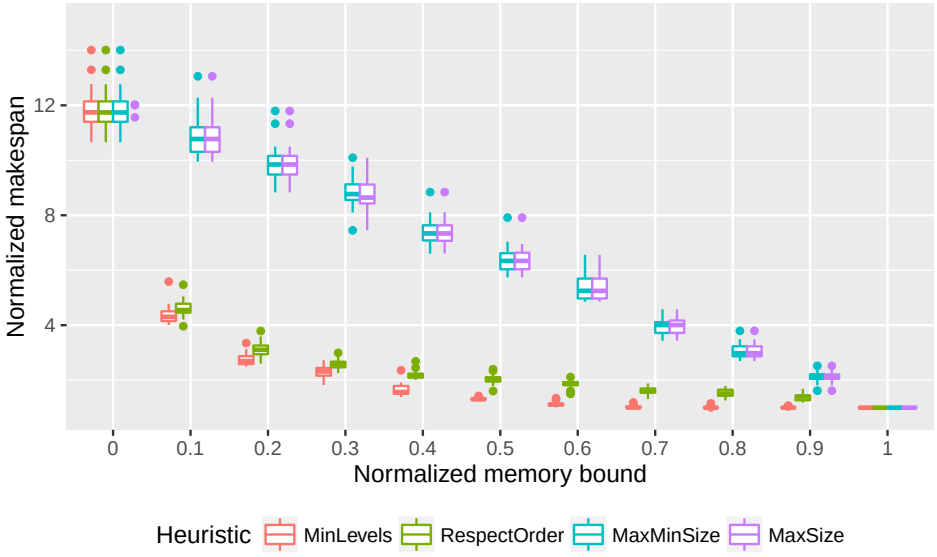


Figure 13: Makespan obtained by each method for the GENOME dataset.

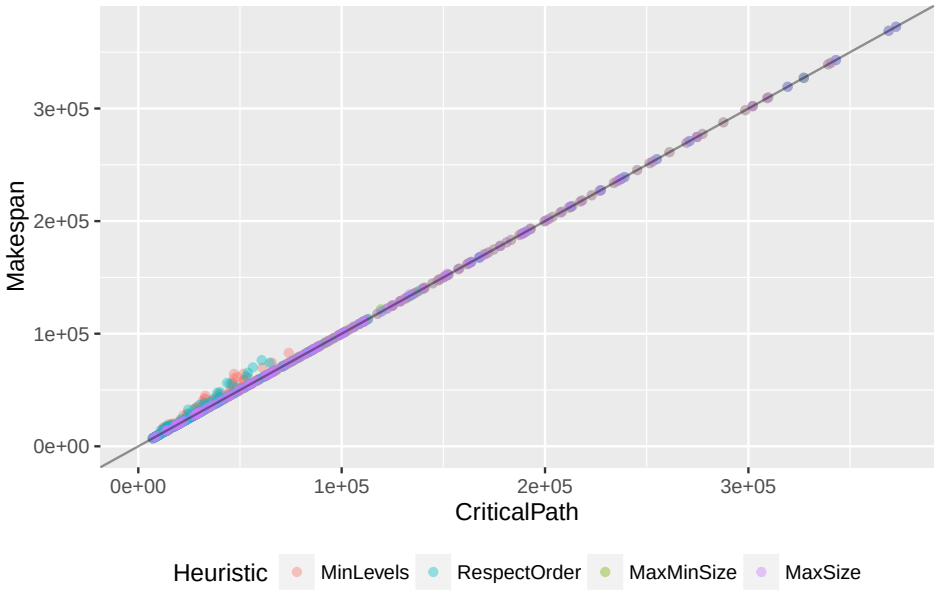


Figure 14: Makespan in function of the critical path length obtained by each method for the GENOME dataset.

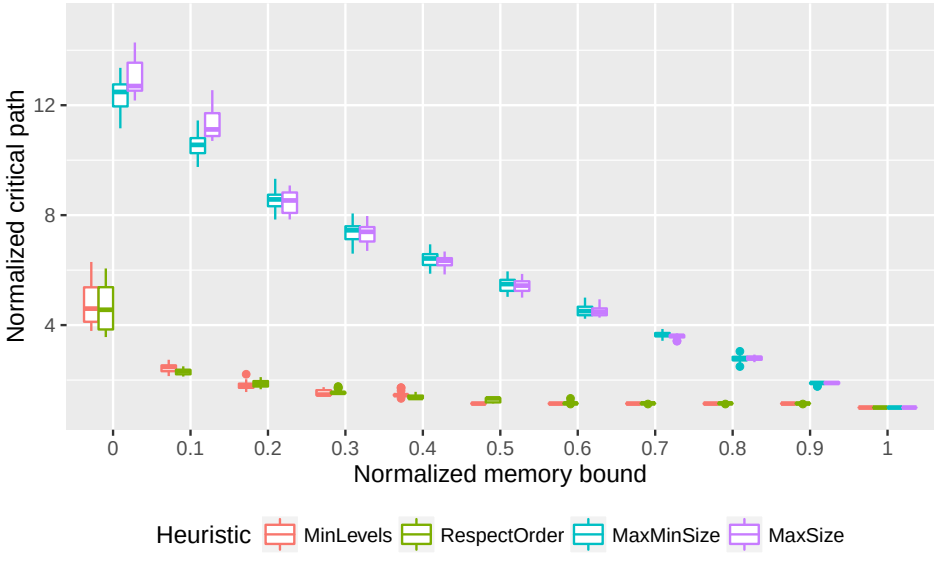


Figure 15: Critical path length obtained by each method for the MONTAGE dataset.

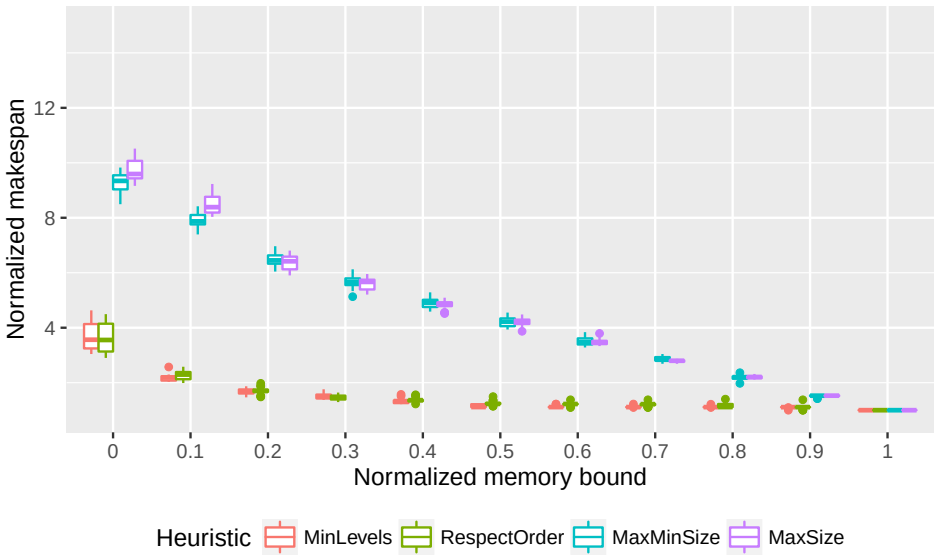


Figure 16: Makespan obtained by each method for the MONTAGE dataset.

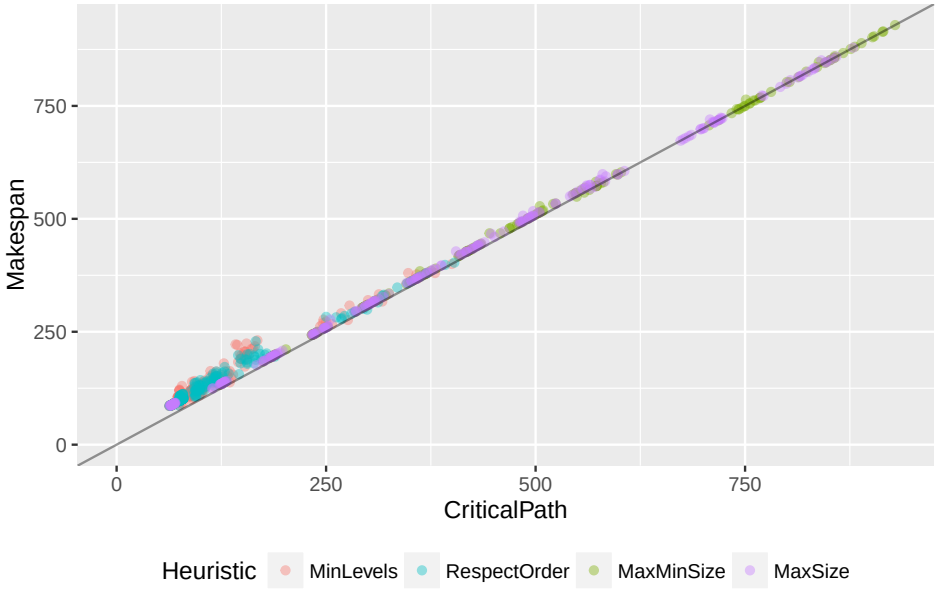


Figure 17: Makespan in function of the critical path length obtained by each method for the MONTAGE dataset.

We have shown in these experiments that we can partially serialize realistic graphs so that any schedule fits a given memory bound, for a reasonable cost in terms of the critical path and makespan augmentation. One may argue that the maximal peak memory considered does not reflect the actual memory consumption of a traditional algorithm. In order to address this problem, we measured the peak memory achieved by the scheduling heuristic on every graph of the datasets. Then, we normalized it in the same way as in the plots above: a value of 1 means that the maximal peak memory is actually achieved, and a value of 0 means that the peak memory reached is the same as the Depth First Search considered. Note that we can obtain negative values, which happened only for some graphs of the DAGGEN datasets, if the DFS requires a larger memory. The statistical summary is presented in Table 3. We note that the scheduling heuristic uses the maximal peak memory for most of the graphs of the LIGO and GENOME datasets, and a normalized memory larger than 0.88 for most of the graphs of the MONTAGE dataset. Therefore, on these realistic graphs, the gain in memory is high. On the DAGGEN dataset, the partial serialization is not as beneficial, as we obtain a median of 0.53. However, note that the MINLEVELS heuristic does not lead to a high augmentation of makespan: less than a 5% increase for the lowest memory bound for 75% of the graphs. Therefore, it is logical that the memory consumption can not be reduced by a larger factor.

	DAGGEN dense | sparse	LIGO	MONTAGE	GENOME
First quartile	-0.03 | 0.39	0.99	0.88	1
Median	0.31 | 0.6	1	0.9	1
Third quartile	0.71 | 0.75	1	0.93	1

Table 3: Normalized memory used by EFT

7 Conclusion

In this paper, we have focused on lowering the memory footprint of task graphs representing computational workflows. As we recognize the need for dynamic schedules (such as in runtime systems), we have focused on the transformation of the graphs prior to the scheduling phase. Adding fictitious edges that represent "memory dependencies" prevents the scheduler to run out of memory. After formally modeling the problem, we have shown how to compute the maximal peak memory of a graph, we have proven the problem of adding edges to cope with limited memory while minimizing the critical path to be NP-complete, and proposed both an ILP formulation of the problem and several heuristics. Our simulations show that our best heuristics, RESPECTORDER and MINLEVELS, either never fail, or are able to limit the memory footprint with lim-

ited impact of the parallel makespan for most task graphs. Our future work consists in implementing the proposed heuristics in a runtime system and evaluate them on actual graphs.

References

[1] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J. L'Excellent, and F. Rouet. Robust memory-aware mappings for parallel multifrontal factorizations. *SIAM J. Scientific Computing*, 38(3), 2016.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] G. Aupy, C. Brasseur, and L. Marchal. Dynamic memory-aware task-tree scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 758–767. IEEE, 2017.

[4] S. Bharathi and A. Chervenak. Scheduling data-intensive workflows on storage constrained resources. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*. ACM, 2009.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: Exploiting heterogeneity for enhancing scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[6] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] R. F. Da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman. Community resources for enabling research in distributed scientific workflows. In *e-Science (e-Science), 2014 IEEE 10th International Conference on*, volume 1, pages 177–184. IEEE, 2014.

[9] F. Desprez and F. Suter. A bi-criteria algorithm for scheduling parallel task graphs on clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 243–252. IEEE, 2010.

[10] M. Drozdowski. Scheduling parallel tasks – algorithms and complexity. In J. Leung, editor, *Handbook of Scheduling*. Chapman and Hall/CRC, 2004.

[11] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[13] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *International Workshop on Parallel Symbolic Computation*, pages 15–23, 2007.

[14] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 136–146, New York, NY, USA, 1986. ACM.

[15] S. Hunold. One step toward bridging the gap between theory and practice in moldable task scheduling with precedence constraints. *Concurrency and Computation: Practice and Experience*, 27(4):1010–1026, 2015.

[16] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. On optimal tree traversals for sparse matrix factorization. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 556–567. IEEE, 2011.

[17] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[18] M. Lampis, G. Kaouri, and V. Mitsou. On the algorithmic effectiveness of digraph decompositions and complexity measures. *Discrete Optimization*, 8(1):129–138, 2011.

[19] E. L. Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

[20] J. W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM J. Algebraic Discrete Methods*, 8(3):375–395, 1987.

[21] Peter Shor (http://cs.stackexchange.com/users/198/peter-shor). Minimum s-t cut in weighted directed acyclic graphs with possibly negative weights. Computer Science Stack Exchange. URL: http://cs.stackexchange.com/q/6498 (version: 2012-11-05).

[22] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *IJHPCA*, 23(3):284–299, 2009.

[23] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 401–409, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[24] D. Sbîrlea, Z. Budimlić, and V. Sarkar. Bounded memory scheduling of dynamic task graphs. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 343–356. ACM, 2014.

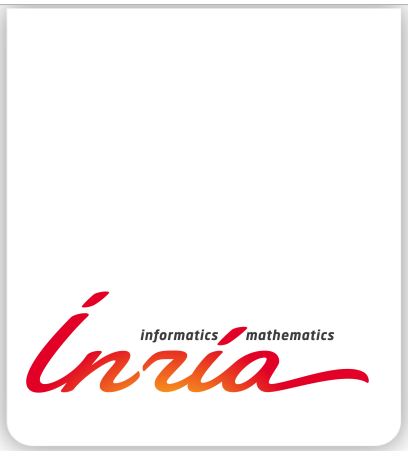
[25] M. Sergent, D. Goudin, S. Thibault, and O. Aumage. Controlling the memory subscription of distributed applications with a task-based runtime system. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*, pages 318–327. IEEE, 2016.

[26] R. Sethi. Complete register allocation problems. *SIAM journal on Computing*, 4(3):226–248, 1975.

[27] R. Sethi and J. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970.

[28] F. Suter. Daggen: A synthetic task graph generator. `https://github.com/frs69wq/daggen`.

[29] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399