International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 11, Number 1, pages 27–49, January 2021

Dynamic DAG Scheduling Under Memory Constraints for Shared-Memory Platforms[1]

Gabriel Bathie
Laboratoire LIP, ENS Lyon, France

Loris Marchal
Laboratoire LIP, ENS Lyon, France

Yves Robert
ENS Lyon, France
University of Tennessee, Knoxville, TN, USA

Samuel Thibault
Laboratoire LaBRI, Univ. Bordeaux, France

Received: July 23, 2020
Revised: September 12, 2020
Accepted: October 6, 2020
Communicated by Masahiro Shibata

Abstract

This work focuses on dynamic DAG scheduling under memory constraints. We target a shared-memory platform equipped with p parallel processors. The goal is to bound the maximum amount of memory that may be needed by any schedule using p processors to execute the DAG. We refine the classical model that computes maximum cuts by introducing two types of memory edges in the DAG, black edges for regular precedence constraints and red edges for actual memory consumption during execution. A valid edge cut cannot include more than p red edges. This limitation had never been taken into account in previous works, and dramatically changes the complexity of the problem, which was polynomial and becomes NP-hard. We introduce an Integer Linear Program (ILP) to solve it, together with an efficient heuristic based on rounding the rational solution of the ILP. In addition, we propose an exact polynomial algorithm for series-parallel graphs. We further study the extension of the approach where the scheduler is dynamically constrained to select tasks (among ready tasks) so that the total memory used does not exceed some threshold. We provide an extensive set of experiments, both with randomly-generated graphs and with graphs arising from practical applications, which demonstrate the impact of resource constraints on peak memory usage.

Keywords: Workflow, task graph, dynamic scheduler, memory constraint, complexity.

1 Introduction

In the last decade, task systems have become ubiquitous to deploy scientific applications on large-scale parallel platforms. In such systems, the application is represented by a Directed Acyclic Graph

[1]A short version of this work [5] has appeared in the proceedings of the APDCM'20 workshop (colocated with IPDPS'20)

(DAG) of tasks, where the nodes represent the *tasks* (a computational kernel composed of a sequential set of operations to be applied to the input data), and the edges represent the *dependencies* between the tasks. The set of dependencies defines a partial order of execution. The problem is to map the tasks onto a set of p computing processors. In this paper, we target shared-memory platforms, where available processors consist of dozens of cores that share a main memory. A traditional objective is to determine a scheduling that minimizes the total execution time, or makespan. The makespan minimization problem has received considerable attention in the scheduling literature. On the theoretical side, many complexity results establish NP-hardness and inapproximability results. On the more practical side, several list heuristics[2] have been developed to achieve close-to-optimal makespans. These heuristics typically aim at minimizing the critical path of the schedule, and use estimations of task priorities such as bottom levels [16, 40]. However, all these heuristics are designed statically, meaning that they assign tasks to processors in a pre-determined ordering, before the beginning of the parallel execution. It turns out such static strategies are unlikely to reach their expected performance, and this for many reasons: (i) task duration estimates are known to be inaccurate and may be affected by unexpected preemptions by the system; (ii) data transfer costs on the platform are hard to correctly model and significantly vary from one execution to another, because they strongly depend upon link contention; and (iii) the resulting small estimation errors are likely to accumulate and to cause large delays. Altogether, static heuristics end up making wrong decisions!

This explains why most runtime systems [22, 4, 31, 8, 20, 33] rely on *dynamic* scheduling, where task allocations and their execution ordering are decided at runtime, based on the system state and unexpected events. These runtime systems dynamically maintain the list of tasks that are ready for execution, and assign them on-the-fly to processors, thereby accurately balancing the workload. However, not all dynamic schedules are equally good, because of memory constraints. Intuitively, a dynamic scheduling can be seen as a parallel traversal of the task graph, with all processors progressing simultaneously on different paths. At any time-step in the execution, the amount of memory needed for the traversal depends upon the input and output data of the tasks that are active at that step (see Section 3 for a detailed description), and this memory amount should never exceed the maximum memory made available to the application. Otherwise, the traversal will require the use of swap mechanisms or *out-of-core* execution, which will dramatically (and negatively) impact the achieved makespan [34, 1].

Consider a task graph whose internal nodes require a large volume of temporary data, such as graphs arising from multifrontal solvers [3]. Improper scheduling decisions may lead dynamic schedules to hit a memory wall at some step while everything was going fine in the previous steps; the dynamic schedule suddenly reaches a state where any further decision (any choice of the next task to execute) will exceed the amount of available memory. This unfortunate scenario arises because dynamic schedules usually consider only tasks that are ready for execution, and have thus a very limited insight into the fraction of the task graph that is yet to be discovered and processed. To avoid such a pitfall, some global information on the task graph is required to guide the dynamic schedule and enforce safe execution paths.

In summary, dynamic scheduling is needed for performance, but one should ensure that any dynamic schedule that can be produced by the runtime system will never exceed the total amount of memory available to the application. There are few existing studies that take dynamic memory footprint into account when scheduling task graphs, as detailed below in Section 2. In our previous work [29, 30], we have proposed an approach to ensure that any dynamic schedule never exceeds the available memory. In a nutshell, the idea is to introduce fictitious dependencies in the task graph to cope with memory constraints: these additional edges restrict the set of valid schedules and, in particular, forbid the concurrent execution of too many memory-intensive tasks. Formally, the additional edges are introduced to decrease the value of the maximal directed cut of the task graph,

[2] A list heuristic is a greedy scheduling heuristic that never keeps processors idle voluntarily; at any time-step, if there is a task ready to execute and an idle processor, then the task is assigned to that processor. In the general case, there are more ready tasks than idle processors, and the heuristic has to make choices. Obviously, this greedy approach is not always optimal, and there are cases where any list-scheduling heuristic achieves a makespan almost as twice the optimal. See [10] for a survey.

where the cut represents the total memory currently used after executing some tasks (those on one side of the cut) and before executing the rest of the tasks (those on the other side of the cut). There is a price to pay: each additional edge adds a fictitious dependence constraint, thereby limiting the degree of parallelism in the execution. We provide a detailed overview of this approach in Section 3.

However, this previous work [29, 30] does not account for resource limitation: there are only p processors, hence no more than p tasks can be processed concurrently. In terms of memory usage, ignoring resource limitation translates into considering too many potential cuts, thereby requiring too many fictitious edges, which unduly constraints the dynamic schedules. In this paper, we refine the standard model for memory-aware scheduling and introduce the first mechanism to take resource limitation into account. Our new model involves two types of memory edges in the DAG, black edges for regular precedence constraints, and red edges for actual memory consumption during execution. Then a valid edge cut cannot include more than p red edges. This limitation dramatically changes the complexity of the problem, which is polynomial with a single edge type and becomes NP-hard with two edge types. We provide an optimal solution for series-parallel graphs and an efficient heuristic for arbitrary graphs. The main contributions of this paper are the following:

- We introduce a new model with colored edges to account for resource constraints when computing peak memory;
- We show that the optimization problem becomes NP-complete, but we introduce an Integer Linear Program (ILP) to solve it, together with an efficient heuristic based on rounding the rational solution of the ILP. We also propose an exact polynomial algorithm for series-parallel graphs (SPGs);
- We further study the extension for the approach where the scheduler is dynamically constrained to select tasks (among ready tasks) such that the total memory used does not exceed some memory amount ;
- We provide an extensive set of experiments, both with randomly-generated graphs and with graphs arising from practical applications, that demonstrate the impact of resource constraints on peak memory usage.

The rest of the paper is organized as follows. We first briefly review the existing work on memory-aware task graph scheduling in Section 2. We provide background on memory-aware scheduling in Section 3. Then, Section 4 is the core of the paper: we introduce the new model, assess its complexity, provide an optimal algorithm for Series Parallel Graphs, and a heuristic for general graphs. Section 5 studies the complexity of the approach where the scheduler is dynamically constrained to select tasks (among ready tasks) so that the total memory used does not exceed some threshold. Section 6 is devoted to simulations both with randomly-generated graphs and with graphs arising from practical applications; we compare the solution computed by an ILP solver together with the solution found by an efficient polynomial-time heuristic. Finally, we conclude and give hints for future work in Section 7.

2 Related Work

Memory and storage have always been limiting parameters for large computations, as outlined by the pioneering work of Sethi and Ullman [38] on register allocation for task trees, modeled as a pebble game. The problem of determining whether a directed acyclic graph can be pebbled with a given number of pebbles (i.e., executed with a given number of registers) has been shown NP-complete by Sethi [37] if no vertex is pebbled more than once (the general problem allowing recomputation, that is, re-pebbling a vertex which have been pebbled before, has been proven PSPACE complete [21]).

This model 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. In several cases, the underlying task graph is a tree, with all dependencies oriented towards the root, which notably simplifies the problem: this is the case for sparse direct solvers [28] but also in quantum chemistry computations [27]. For such trees, memory-aware parallel schedulers have been proposed in [17], and the impact of processor mapping on memory consumption has been studied in [1].

The problem of general task graphs handling large data has been identified by Ramakrishnan et al. [34] who introduced clean-up jobs to reduce the memory footprint and propose some simple heuristics. Their work was continued by Bharathi et al. [6] who developed genetic algorithms to schedule such workflows. More recently, runtime schedulers have also been confronted to the problem: in the StarPU task-based runtime system, attempts have been made to reduce memory consumption by throttling the task submission rate [36].

As explained in the introduction, we have previously proposed a way to restrict the potentially large memory needed for the traversal of a task graph by adding edges that correspond to fictitious dependencies [29, 30]. Our method consists in first computing the worst achievable memory of any parallel traversal, using either a linear program or a min-flow algorithm. Then if the previous computation detects a potential situation when the memory exceeds what is available on the platform, we add a fictitious edge in order to make this situation impossible to reach in the new graph. This study is inspired by the work of Sbîrlea et al. [35]. In that study, the authors focus on a different model, in which all data have the same size (as for register allocation). They target smaller-grain tasks in the Concurrent Collections (CnC) programming model [9], a stream/dataflow programming language. Their objective is, just as ours, to schedule a DAG of tasks using a limited memory. To this purpose, they associate a color to each memory slot and then build a coloring of the data, in which two data items with the same color cannot coexist. If the number of colors is not sufficient, additional dependence edges are introduced to prevent too many data items 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 suitable for a few distinct sizes.

While our previous study [29, 30] is a first step towards the design of efficient memory-bounded dynamic schedulers, it suffers from major shortcomings that prevents its use in actual runtime schedulers:

- First, the running time of the algorithm is too high: computing the worst possible memory, while done in polynomial time, is expensive ($O(n^3)$ for a dense graph with n vertices), and it has to be called after each edge insertion, so potentially $O(n^2)$ times.
- Second, the algorithm assumes an unlimited number of processors, and thus the simultaneous execution of infinitely many tasks. Thus, it dramatically overestimates the amount of memory that may actually be needed by a parallel processing of the DAG.

In the present work, we alleviate both problems, through a new model to finely take the number of processors into account, and a new algorithm with much reduced complexity for a special case of task graphs (series-parallel graphs).

Finally, a recent paper studies the problem of computing the maximum memory of a multithreaded computation [26]. Their model is more complex and dedicated to Cilk programs, with the objective to derive low-complexity algorithms for this problem (typically linear-time algorithms).

3 Background

In Section 3.1, we introduce the SimpleDataFlowModel [29, 30] to study memory usage for general DAGs. This model is a natural extension of the original pebble game [38], and of the model introduced by Liu for tree graphs [28]. Then in Section 3.2, we discuss how to emulate more realistic models, and outline the limitations of the current approach.

3.1 The SimpleDataFlowModel

The target application is described by a workflow of tasks whose precedence constraints form a DAG $G = (V, E)$. Each node $i \in V$ represents a task and each edge $e \in E$ represents a precedence constraint, expressed in the form of output and input data. The processing time necessary to complete a task $i \in V$ is denoted by w_i. The memory usage of the computation is modeled only by the size of the data produced by the tasks and represented by the edges. Specifically, 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 appropriate edges of weight zero. An example of such a graph is illustrated in Figure 1.

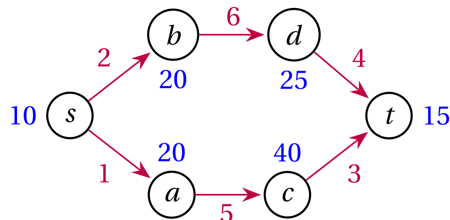

Figure 1: Example of a workflow, (red) edge labels represent the size $m_{i,j}$ of associated data, while (blue) node labels represent their computation weight w_i.

Memory consumption rules are remarkably simple in the SimpleDataFlowModel. In the 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 memory. We introduce the following definitions for the total input and output size of a node $i \in V$:

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

Now, the total amount of memory M_{used} needed to store all necessary data is transformed as follows when task i is executed:

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

The SimpleDataFlowModel may seem unrealistic, because when we start executing a task, its inputs are immediately deleted and we only allocate memory for its outputs. In many scientific applications, it is required to store both the inputs and the outputs throughout the execution of the task, and maybe to allocate space for some temporary data internal to the task. Fortunately, many complex memory behaviors, including the latter one with input, output and temporary data co-existing in memory, can be emulated in the SimpleDataFlowModel, via some elementary transformations of the input DAG. Together with its simplicity, this versatility explains the appeal of the SimpleDataFlowModel and its usage in the literature [28, 29, 30].

We detail elementary transformations to account for more complex memory consumption rules in Section 3.2. Beforehand, we explain how to estimate peak memory usage in the SimpleDataFlowModel. A *schedule* or *parallel execution* of a DAG with 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 $\sigma(i) \geq \sigma(j) + w_j$). When considering parallel executions, we assume that all processors use the same shared memory, whose size is limited. We say that the data associated to the edge (i,j) is *active* at a given time-step if the execution of i has started but not that of j. This means that the (output) data of i is present in memory.

We now compare parallel and sequential schedules. A *sequential schedule* $\mathcal{S}$ of a DAG G is defined by a total order σ of its tasks. Clearly, the *memory used* by a sequential schedule at a given time-step 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, which is given by:

$$M_{\text{peak}}(\sigma) = \max_i \sum_{j \text{ s.t. } \sigma(j)\leq\sigma(i)} \mathit{Outputs}\,(j) - \mathit{Inputs}\,(j) \tag{1}$$

where the set $\{j \text{ s.t. } \sigma(j) \leq \sigma(i)\}$ represents the set of tasks started before task i, including itself. Equation (1) demonstrates the simplicity of the SimpleDataFlowModel, where input data are replaced by output data as the execution progresses.

Furthermore, Equation (1) allows us to state a prominent feature of the SimpleDataFlowModel: there is no difference between *sequential schedules* and *parallel executions* as far as memory is concerned! More precisely, for each parallel execution (μ, σ), there exists a sequential schedule with equal peak memory: simply consider a sequential schedule that starts tasks in the same order as the parallel execution (see the detailed proof in [30]). A key consequence is that we can bound the maximum memory of any parallel execution: it is equivalent to computing the peak memory of a sequential schedule. Then, to compute the peak memory of a sequential schedule, we define a topological cut $C = (S, T)$ of a DAG G as a partition of G in two sets of nodes S and T such that either S or T is empty (degenerate case), or otherwise $s \in S$, $t \in T$, and no edge is directed from a node of T to a node of S (regular case). An edge (i, j) belongs to the cut if $i \in S$ and $j \in T$. The weight $M(C)$ of a topological cut C is the sum of the weights 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. Note that this cut would not be a topological cut if the edge (d, a) was present in the graph. 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 or initiated. Therefore, the maximal peak memory of a DAG is equal to the maximum weight of a topological cut. It turns out that there exists an algorithm to compute a maximal topological cut with polynomial complexity $O\left(|V||E|\log\left(|V|^2/|E|\right)\right)$[30]. As stated in the introduction, if the maximal topological cut exceeds the total memory available, we have proposed in our previous work to add fictitious edges that will go backwards (from T to S) and will decrease the weight of the cut. Unfortunately, the approach is very costly [29, 30]: we may need to insert $O(|V|^2)$ edges, each at a cost $O(|V|^3)$ if the DAG is dense (with $|E| = \Theta(|V|^2)$).

3.2 Emulation of More Realistic Models

As explained above, the SimpleDataFlowModel does not account for the fact that inputs and outputs of a given task often reside in memory simultaneously. However, this is a common behavior for scientific applications, and some studies [25] further account for some temporary data m_i^{temp} that has to be in memory when processing task i (in addition to task inputs and outputs). The memory needed for processing task i becomes $\mathit{Inputs}\,(i) + m_i^{temp} + \mathit{Outputs}\,(i)$. Such a behavior can be emulated in the SimpleDataFlowModel, as illustrated on Figure 2. Each task i is split into two nodes i_1 and i_2. We transform all edges (i, j) in edges (i_2, j), and edges (k, i) in edges (k, i_1). We also add an edge (i_1, i_2) with an associated data of size $\mathit{Inputs}\,(i) + m_i^{temp} + \mathit{Outputs}\,(i)$. Task i_1 represents the allocation of the data needed for the computation, as well as the computation itself, and its weight is thus $w_{i_1} = w_i$. Task i_2 stands for the deallocation of the input and temporary data and has weight $w_{i_2} = 0$.

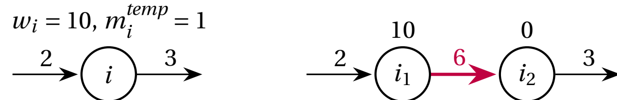

Figure 2: Transformation of a task as in [25] (left) to the SimpleDataFlowModel (right).

After this transformation, the graph includes two types of edges. The edges that were originally present in the graph and stand for regular dependencies between tasks are called the *black edges*. The edges that have been added to represent computations are called the *red edges*. Both edge types have different roles. In particular, there cannot be more than p red edges in a cut representing an actual state of a parallel computation of the graph with p processors. We now understand another limitation of the SimpleDataFlowModel: while it can emulate parallel executions with realistic memory rules, computing the maximum cut of the transformed graph will only provide a loose upper bound of the maximum memory needed by any dynamic schedule. In other words, we can still compute the maximum cut of the transformed graph, but it will overestimate the amount of memory that may actually be needed during a parallel execution of the DAG. One major contribution of this paper is to introduce a new framework which distinguishes between black and red edges to

account for resource constraints.

4 Resource Constraints

We formally state the optimization problem in Section 4.1 and assess its complexity in Section 4.2 for general graphs. We also formulate the problem as the solution of an Integer Linear Program (ILP) in Section 4.3, and we introduce an efficient heuristic. Finally, we give an efficient algorithm to solve the problem for series-parallel graphs, or SPGs, in Section 4.4.

4.1 Optimization Problem

As outlined in Section 3.2, when we transform an edge-weighted DAG G to the SimpleDataFlowModel, the resulting graph contains two different types of edges: black edges, that correspond to precedence constraints (edges of G), and red edges, that represent computations (vertices of G). Recall that the memory weight of computation edges is the sum of the memory used by the input, the output and temporary data of the computation. Therefore, the weight of red edges will likely be larger than that of black edges, which only carry the weight of input or output data.

The max-cut of the graph may well go through an arbitrary number of red edges. However, the program is scheduled on a platform with p processors, hence at most p computations can be executed in parallel. Therefore, the max-cut is an overestimation of maximum memory usage of the program, and the difference may be quite large especially because red edges have larger weights. Figure 3 illustrates this scenario.

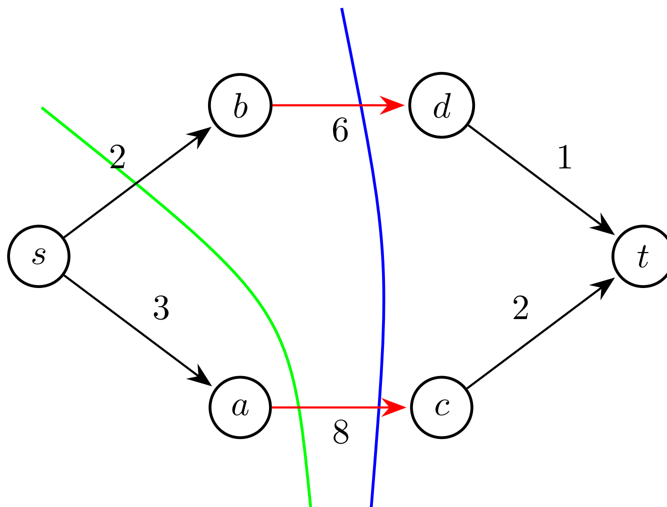

Figure 3: Example of DAG for which the maxcut is an overestimation of the maximum memory used. The weight of the maxcut (shown in blue) is 14. For $p = 1$, the max cut with at most 1 red edge (shown in green) has weight 10.

The natural question that arises is how to compute the maximum topological cut of a DAG cutting at most p computation edges. We state this question formally:

Problem 1. p-MaxTopCut *(optimization)*

Input: a DAG $G = (V, E)$, a weight function $m : E \to \mathbb{N}$, a coloring of the edges $c : E \to \{red, black\}$, a number of processors $p \in \mathbb{N}^*$.

Output: A topological cut $C = (S, T)$ of G, with maximum weight $M^*(C) = \sum_{e\in(S\times T)\cap E} m(e)$, crossing at most p red edges, i.e. $\sum_{e\in(S\times T)\cap E} \mathbb{1}_{c(e)=red} \leq p$.

and the corresponding decision problem:

Problem 2. p-MaxTopCut

Input: a DAG $G = (V, E)$, a weight function $m : E \to \mathbb{N}$, a coloring of the edges $c : E \to \{red, black\}$, a number of processors $p \in \mathbb{N}^*$, a memory bound $W \in \mathbb{N}$.

Question: Is there a topological cut $C = (S, T)$ in G, with weight at least W, crossing at most p red edges?

In what follows, we will use the term "p-cut" to refer to a *topological* cut crossing at most p red edges, and "p-maxcut" for a *topological* cut with maximum weight among those crossing at most p red edges.

4.2 Complexity

As discussed in Section 3.1, computing the maximum-weight topological cut (without colored edges) of a graph can be done in polynomial time. We show that adding the constraint on colors of edges makes the problem computationally hard:

Theorem 1. p-MaxTopCut *is NP-Complete*

Proof. The p-MaxTopCut problem is in NP: the set S of the cut (S, T) is a polynomial certificate. One can check in polynomial time that the cut is topological, has weight at least W and includes at most p red edges. In order to prove hardness, we do a reduction from the Max-k-SubsetIntersection (MSI) problem, which is NP-Complete [42]. The MSI problem is the following:

Definition 1. *Given a set X, $\mathcal{C} = \{S_i\}_{i\in[1,\dots,l]}$ a set of l subsets of X, two integers $k \le l$ and q, find a subset $I \subseteq [1, \dots, l]$ such that $|I| = k$ and $\left|\bigcap_{i\in I} S_i\right| \ge q$. In other words, find k subsets S_i such that the cardinality of their intersection is greater than or equal to q.*

Consider an instance $\mathcal{I}_1$ of MSI: a set X, $\mathcal{C}$ a collection of l subsets of X, two integers k and q. Let $n = |X|$ and $x_1, \dots x_n$ denote the elements of X. We build the following instance $\mathcal{I}_2$ of p-MaxTopCut: $G = (V, E)$, where

$$\begin{aligned} V &= \{s, t\} \cup \{u_i | i = 1, \dots, l\} \cup \{v_j | j = 1, \dots, n\} \\ E &= \{(s, u_i) | i = 1, \dots, l\} \cup \{(v_j, t) | j = 1, \dots, n\} \\ &\quad \cup \{(u_i, v_j) | x_j \notin S_i\} \end{aligned}$$

where the edges from s to the u_i are red and have weight $n + 1$, and the other edges are black. The edges from the v_j to t have weight 1, and the edges from the u_i to the v_j have weight 0. Finally, let $p = k$ and $W = (n + 1)p + q$. See Figure 4. If a node v_j has no predecessor (respectively a node u_i has no successor), we can add a black edge (s, v_j) (respectively (u_i, t)) with weight 0. This allows us to consider the case with only one source and target, but does not change the rest of the proof, hence we will omit these edges in the rest of the proof.

Now, assume that $\mathcal{I}_1$ has a solution, i.e. there are p subsets $(S_i)_{i\in I}$ of X whose intersection has cardinality at least q. Then consider the cut (S, T) where

$$\begin{aligned} S_0 &= \{s\} \cup \{u_i | i \notin I\} \\ S &= S_0 \cup \{v_j \mid \text{every predecessor of } v_j \text{ is in } S_0\} \end{aligned}$$

and $T = V \setminus S$: it goes through the edges (s, u_i) for $i \in I$ and through the edges (v_j, t) for $x_j \in \bigcap_{i\in I} S_i$. It is a topological cut, has exactly p red edges and by construction of G, all the v_j corresponding to the x_j that are in the intersection of the S_i are not linked to the corresponding u_i. Therefore, we can put at least q of them in S, and the cut crosses at least q edges (v_j, t) of weight 1. Hence, the cut has weight at least $p \cdot (n + 1) + q \cdot 1$ (the first term counts the weight of the red edges, the second term counts the weight of the (v_j, t) edges), and therefore it is a solution to $\mathcal{I}_2$.

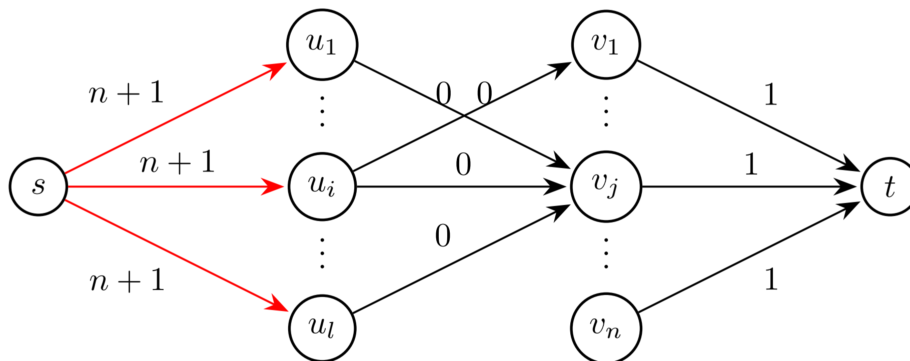

Figure 4: DAG for the reduction: $(u_i, v_j) \in E \Leftrightarrow x_j \notin S_i$.

Conversely, assume that $\mathcal{I}_2$ has a solution, i.e. there exists a topological cut (S, T) with at most p red edges and weight greater than $(n+1)p+q$. It goes through exactly p red edges, otherwise if it goes through less that p red edges, it can have weight at most $(p-1)(n+1)+n \cdot 1$ as the other edges carrying weight are the edges of weight 1, and there are only n of them. As the weight is greater than $(n+1)p+q$, we get that the cut crosses at least q edges (v_j, t) of weight 1.

Let $I = \{i | u_i \in T\}$, the set of the indices of the subsets corresponding to the (s, u_i) edges crossed by the cut. As remarked above, $|I| = p = k$, therefore we have selected exactly k subsets. To show that I is a solution to $\mathcal{I}_1$, we need to show that $\left|\bigcap_{i \in I} S_i\right| \geq q$.

Let $Y = \{x_j | v_j \in S\}$ be the set of elements x_j such that the edge (v_j, t) is crossed by the cut. As mentioned above, the cut crosses at least q such edges, therefore $|Y| \geq q$. For all $y \in Y$, as the cut is topological, we have that they are not linked to any of the $C_i, i \in I$. Therefore, by construction of G, we have $y \in C_i$ for all i, which implies that $y \in \bigcap_{i \in I} C_i$, hence $Y \subseteq \bigcap_{i \in I} C_i$. This implies in turn that $\left|\bigcap_{i \in I} C_i\right| \geq q$, therefore $\mathcal{I}_1$ has a solution.

Last, we show that this reduction is polynomial. The graph $G = (V, E)$ of $\mathcal{I}_2$ has $|V| = n + l + 2$ nodes and $|E| \leq n + l + nl$ edges, and can be constructed in polynomial time by a simple inspection of the sets $S_i, i \in [1, \dots, l]$. Furthermore, $W = np + q$ can also be computed in polynomial time from n, k and q. Therefore, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. This concludes the proof that p-MaxTopCut is NP-complete. □

4.3 Integer Linear Program and Heuristic

The following Integer Linear Program (ILP) can be used to compute the p-maxcut:

$$\max \sum_{(i,j) \in E} m_{i,j} d_{i,j} \tag{2}$$

$$\forall (i,j) \in E, \quad d_{i,j} = p_i - p_j \tag{3}$$

$$\forall (i,j) \in E, \quad d_{i,j} \geq 0 \tag{4}$$

$$p_s = 1 \tag{5}$$

$$p_t = 0 \tag{6}$$

$$\sum_{(i,j) \in E} \mathit{isred}_{i,j} d_{i,j} \leq p \tag{7}$$

$$\forall i, p_i \in \{0, 1\} \tag{8}$$

The p variables are used to assign vertices to either S ($p_i = 1$) or T ($p_i = 0$). We consider that $\mathit{isred}_{i,j} = 1$ if $c(i,j) = \mathit{red}$ and $\mathit{isred}_{i,j} = 0$ otherwise. This ILP is adapted from the one from [30]

which computes the maximum topological cut of G. A single constraint has been added: Equation (7) limits the number of red edges from S to T to at most p.

In the case of the maximum topological cut without resource constraints, there is a simple way to solve this ILP by solving it over the rational numbers and rounding to integers. Unfortunately, due to the additional constraint (Equation (7)), the rounding procedure does not always give a valid optimal value in the case of p-MaxTopCut. However, this gives the intuition for a heuristic. Starting from a fractional solution of the above linear program and a threshold value $w \in [0,1]$, we can derive an integer solution as follows: we take the p_is returned by the rational solution, and set p_i to 0 in the integer solution if and only if we had $p_i \leq w$ in the rational solution (and we let $p_1 = 1$ otherwise). This describes a topological cut, which might use more than p red edges. We propose to apply this rounding procedure to all possible values of w. In practice, we only have to consider all p_i rational values for $i = 1, \ldots, |V|$ as well as $w = 1$. Among these $|V| + 1$ values of w, we return the topological cut with at most p red edges with maximum weight (if any). Note that this procedure may fail if no rounding produces a cut with less that p red edges. However, considering all the $|V| + 1$ rounding values makes this very unlikely. In particular, it never happened in all the simulations reported in Section 6: the heuristic always found a solution; furthermore, that solution was close to the optimal value in most cases (see Section 6 for details).

4.4 Series-Parallel Graphs

Series-Parallel Graphs, or SPGs, are widely used in the literature because they nicely model fork-join types of computations such as BSP (Bulk Synchronous Parallel model) [11, 18]. SPGs are defined inductively as follows:

Definition 2. *A series-parallel graph (SPG) is either:*

- *the "Edge" graph $E(m, r) = (\{s,t\}, \{(s,t)\})$: two nodes, the source and the target, linked by an edge. m is the weight of that edge, $r \in \{true, false\}$ is true if and only if $c(s,t) = red$,*
- *the series composition of two SPGs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ (with respective sources and targets (s_1, t_1) and (s_2, t_2)):*

$$Series(G_1, G_2) = (V_1 \cup V_2, E_1 \cup E_2)$$

  with source $s = s_1$, target $t = t_2$, with $t_1 = s_2$ in the resulting graph,
- *the parallel composition of two SPGs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$:*

$$Par(G_1, G_2) = (V_1 \cup V_2, E_1 \cup E_2)$$

  with source $s = s_1 = s_2$ and target $t = t_1 = t_2$.

Series and parallel composition are illustrated on Figure 5.

Theorem 2. *The* p-MaxTopCut *problem can be solved in time $\mathcal{O}(|E|p^2)$ for a SPG with $|E|$ edges on a platform with p processors.*

Proof. A SPG is a binary tree of its constructors, called its decomposition tree (see Figure 6): leaves of the tree are the edges of the SPG, internal nodes are the series and parallel constructors. Note that every internal node has exactly two children, thus the tree is a full binary tree. Furthermore, given a series-parallel graph, its decomposition tree can be built in linear time [41, 7].

Furthermore, if G is the series composition of G_1 and G_2, then a topological cut of G is either a topological cut of G_1 or of G_2: the topological constraints forbid a cut that goes through both. Similarly, if $G = Par(G_1, G_2)$, then any cut of G that goes through G_1 goes through G_2 as well. Therefore, a topological cut of G with p red edges will cross k red edges in G_1 and $p - k$ red edges in G_2, for some $k, 0 \leq k \leq p$. Finally, if G is a red edge (s,t), it has no topological cut with zero red edges, and one nonempty topological cut: $(\{s\}, \{t\})$. If G is a black edge, then its maxcut is $(\{s\}, \{t\})$.

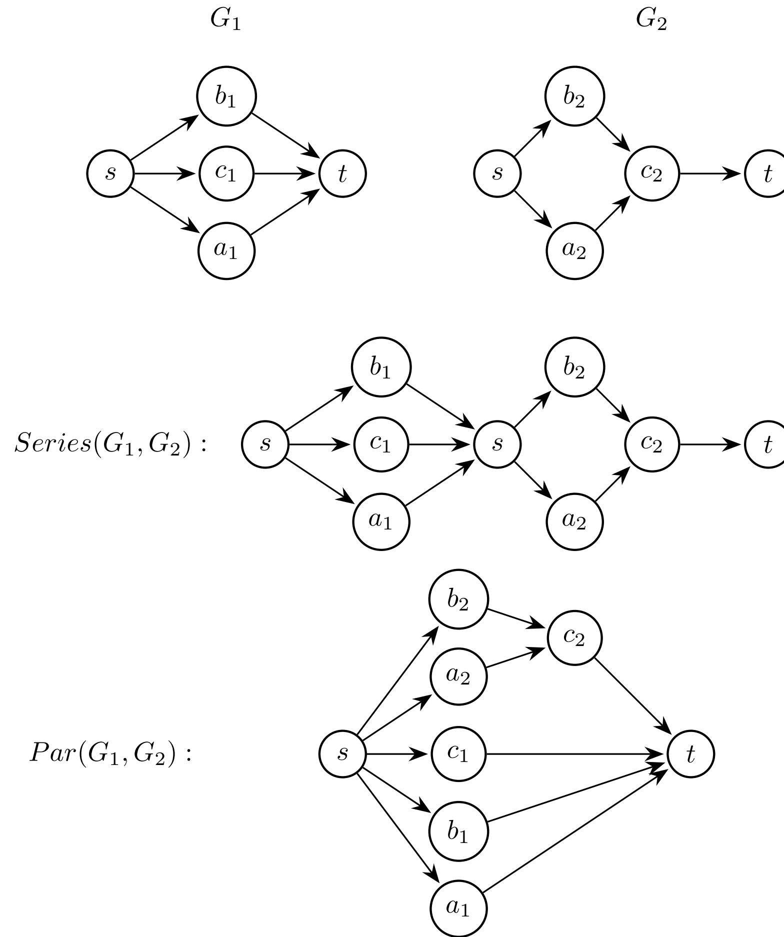

Figure 5: Example of series and parallel composition of SPGs.

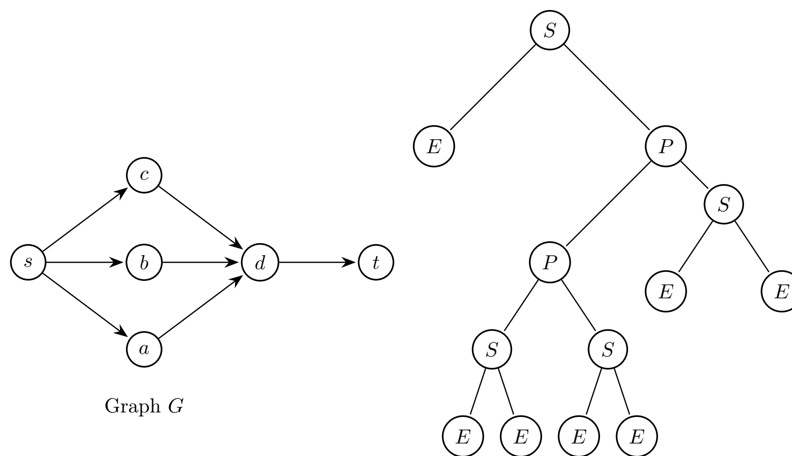

Figure 6: Example of SP Graph (left) and its decomposition tree (right). S = Series constructor, P = Parallel, E = *Edge*.

Let $M(G,k)$ denote the weight of the k-maxcut of a SPG G. The previous remarks lead to the following formulas:

$$M(E(m,r),k) = m, \forall k \geq 1, \forall r \in \{True, False\} \tag{9}$$
$$M(E(m,True),0) = -\infty \tag{10}$$
$$M(E(m,False),0) = m \tag{11}$$
$$M(Series(G_1,G_2),k) = \max\{M(G_1,k), M(G_2,k)\} \tag{12}$$
$$M(Par(G_1,G_2),k) = \max_{j=0\ldots k}\{M(G_1,j) + M(G_2,k-j)\} \tag{13}$$

Using these formulas, one can compute $M(G,k)$ using the values of $M(G_1,i), i = 1\ldots p$ and $M(G_2,j), j = 1\ldots p$ in time $\mathcal{O}(p)$ for each $k = 1\ldots$p, hence in total time $\mathcal{O}(p^2)$. Using dynamic programming and storing the values of $M(G',i), i = 1\ldots p$ for all G' in the decomposition tree of G, one can compute the p-maxcut of G in time $\mathcal{O}(p^2 \cdot N)$, where N is the number of nodes in the decomposition tree of G. To conclude on the complexity, we need to show that $N = \mathcal{O}(|E|)$. It is well-known that for any $l \geq 1$, a full binary tree (i.e. each node is either a leaf or has two children) with l leaves has exactly $2l-1$ nodes[3]. Using the fact that the leaves of the decomposition tree of G are exactly the edges of G, we obtain that $N = 2|E|-1$, and therefore the algorithm runs in time $\mathcal{O}(|E|p^2)$. □

5 Scheduling with Runtime Constraints

In this section, we discuss extensions that go beyond bounding the maximum memory peak of a dynamic schedule by computing the p-maxcut of the colored DAG. Indeed, this approach aims at guaranteeing that *any* graph traversal by the scheduler would not require more than some memory amount. But in practice, a dynamic scheduler that schedules tasks on the fly could be programmed to avoid deliberately scheduling tasks which would make the memory used by the parallel execution larger than the memory of the machine. Therefore, even if the p-MaxTopCut is larger than the memory of the machine, the scheduler might still be able to find a scheduling of the graph that does not exceed the available memory. We point out that the scheduling remains completely dynamic in this extension, but the idea is to guide it on the fly, according to the memory requests of the tasks that are discovered ready for execution.

In this context, we would like to know whether the scheduler will always be able to keep the memory of the computation under some threshold M, using the new rule of avoiding any task whose execution would exceed the memory currently available. A sufficient condition is that, for any stage of the computation that uses memory not greater than M, we can schedule another task such that the memory usage stays below M. If the opposite happens, then there exists a scheduling that reaches a situation where any choice leads the computation to use an amount of memory larger than M, which would cause the computation to stall. We state the problem formally as follows:

Definition 3. *Successor of a topological cut*
Let G be a DAG, $\mathcal{C} = (S,T)$ a topological cut of G. A topological cut $\mathcal{C}' = (S',T')$ of G is a successor of $\mathcal{C}$ if $S \subset S'$ and $|S' \setminus S| = 1$, i.e. S' is equal to S with an additional vertex.

Problem 3. TopologicalTraversability
Input: a DAG $G = (V,E)$, an integer $M \in \mathbb{N}$
Question: Does every topological cut $\mathcal{C}$ of G of weight $M(\mathcal{C}) \leq M$ have a successor $\mathcal{C}'$ of weight at most M?

The TopologicalTraversability belongs to $Co-NP$, hence we consider its negation:

Problem 4. BlockingTopologicalCut
Input: a DAG $G = (V,E)$, an integer $M \in \mathbb{N}$

[3]See https://en.wikipedia.org/wiki/Binary_tree.

Question: Does there exist a topological cut $\mathcal{C}$ of G of weight $M(\mathcal{C}) \leq M$ such that every successor $\mathcal{C}'$ has weight larger than M?

Unfortunately, this latter problem is computationally hard:

Theorem 3. BLOCKINGTOPOLOGICALCUT *is NP-Complete.*

Proof. We first prove that the problem is in NP. The certificate is simply the list of all the vertices in one set of the cut C. One can check in time $\mathcal{O}(|E|)$ that the cut is topological and that its weight is smaller than M. Furthermore, the cut has at most $|V| - 1$ successors, as any successor has one vertex more than C. Hence, computing the values of all the successors can be done in $\mathcal{O}(|E| \cdot |V|)$, which is polynomial in the size of G. Therefore, BLOCKINGTOPCUT $\in$ NP.

To prove the hardness of BLOCKINGTOPCUT, we use a reduction from the NP-complete problem 2-PARTITION [19]: Given a family of $n \geq 2$ positive integers $(a_i)_{i=1,\dots,n}, \forall i, a_i > 0$, is there a subset $I \subseteq \{1, \dots, n\}$ such that $\sum_{i \in I} a_i = \frac{S}{2}$? Let $\mathcal{I}_1$ be an instance of 2-PARTITION. Consider the DAG $G = (V, E)$, where

$$V = \{1, \dots, n\} \cup \{s, a, b, t\}$$

and

$$E = \{(s, i), (i, t), i = 1, \dots, n\} \cup \{(s, a), (a, b), (b, t)\}$$

Let $S = \sum_{i=1}^{n} a_i$, and set the weight of the edges as follows:

$$\begin{aligned} w(s, i) &= 0, \forall i \in \{1, \dots, n\} \\ w(i, t) &= a_i, \forall i \in \{1, \dots, n\} \\ w(s, a) &= \frac{S}{2} \\ w(a, b) &= \frac{S}{2} + 1 \\ w(b, t) &= 0 \end{aligned}$$

The resulting graph is shown in Figure 7.

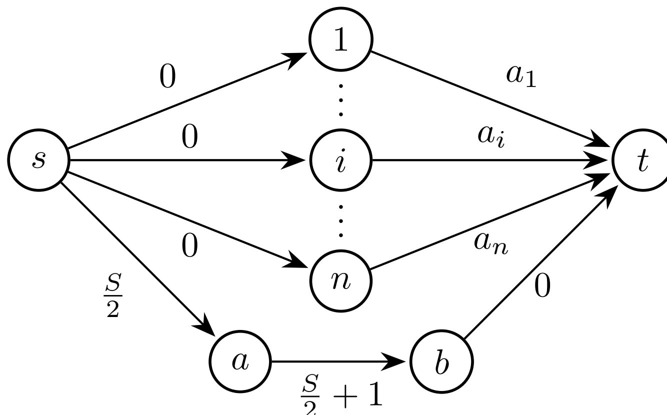

Figure 7: DAG for the reduction.

The instance $\mathcal{I}_2$ of BLOCKINGTOPCUT is G with bound $M = S$. The size of the graph $G = (V, E)$ created in $\mathcal{I}_2$ is polynomial in n: $|V| = n+4$ and $|E| = 2n+3$. $M = S = \sum_{i=1}^{n} a_i$ has size polynomial in those of the a_i. Therefore, the construction of $\mathcal{I}_2$ can be done in time polynomial in the size of $\mathcal{I}_1$, and the reduction is polynomial.

We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does. First, if $\mathcal{I}_1$ has a solution $I \subseteq \{1, \dots, n\}$ such that $\sum_{i \in I} a_i = \frac{S}{2}$, then we define $K = I \cup \{s\}$, and $C = (K, V \setminus K)$. It is a topological cut:

it contains vertices from $1, \ldots, n$ and s, and the only edges entering these vertices are from s. We then have

$$\begin{aligned} M(C) &= \sum_{u \in K} \sum_{v \notin K} w(u,v) \\ &= \left(\sum_{i \in I} \sum_{v \notin K} w(i,v) \right) + \sum_{v \notin K} w(s,v) \\ &= \left(\sum_{i \in I} w(i,t) \right) + \sum_{v \notin K} w(s,v) \\ &= \left(\sum_{i \in I} a_i \right) + w(s,a) \\ &= \left(\sum_{i \in I} a_i \right) + \frac{S}{2} \\ &= \frac{S}{2} + \frac{S}{2} \\ &= S \leq M \end{aligned}$$

Successors of C contain one additional vertex, either a or a vertex from $\{1, \ldots, n\}$. Adding any $j \in \{1, \ldots, n\}$ increases the value of the cut by $a_j > 0$, it would then be strictly larger than S and therefore, strictly larger than M. Adding a increases the weight by 1, and the resulting cut has weight $S + 1 > M$. Hence, C is a solution to $\mathcal{I}_2$, the instance of BlockingTopCut.

Conversely, if $\mathcal{I}_2$ has a solution, it is a cut $C = (K, V \setminus K)$ with $M(C) \leq M$, such that any successor C' has weight $M(C') > M$. We can then remark that the edge (a,b) cannot be in the cut (i.e. we cannot have $a \in K$ and $b \notin K$). Otherwise, such a cut $\tilde{C}$ would have a successor $K \cup \{b\}$ with value $M(\tilde{C}) - \frac{S}{2} - 1 \leq M(\tilde{C}) \leq M$, which would contradict the hypothesis that every successor must have value strictly greater than M.

Similarly, the edge (b,t) cannot be in the cut. Otherwise, either the cut contains all the vertices $i \in \{1, \ldots, n\}$, or it doesn't have all of them. If it has all of them, then the cut is $(V \setminus \{t\})$, and it has a successor with value $0 \leq M$, the cut $(V, \emptyset)$, which contradicts the hypothesis. On the other hand, if K contains only a set $I \subsetneq \{1, \ldots, n\}$ of the a_i, then let $j \in \{1, \ldots, n\} \setminus I$. The value of the cut is $\sum_{i \in I} a_i$. The cut then has a successor where we add the vertex j, with value $\sum_{i \in I} a_i + a_j \leq \sum_{i=1}^{n} a_i = S = M$. This means that the cut $K \cup \{j\}$ (successor of K) has value lower than M, which contradicts the hypothesis.

Hence, if $\mathcal{I}_2$ has a solution, K contains s but not a. It also contains some vertices i of $\{1, \ldots, n\}$: let $I = \{1, \ldots, n\} \cap K$. By hypothesis, we know that $M(K) \leq M = S$, and $M(K) = \sum_{i \in I} a_i + \frac{S}{2}$. Hence, $\sum_{i \in I} a_i + \frac{S}{2} \leq S \Leftrightarrow \sum_{i \in I} a_i \leq \frac{S}{2}$. K has a successor $K' = K \cup \{a\}$. By hypothesis, $M(K') > M = S$, and $M(K') = \sum_{i \in I} a_i + \frac{S}{2} + 1$. Therefore, we have

$$\begin{aligned} &\sum_{i \in I} a_i + \frac{S}{2} + 1 > S \\ \Leftrightarrow &\sum_{i \in I} a_i + 1 > \frac{S}{2} \\ \Leftrightarrow &\sum_{i \in I} a_i \geq \frac{S}{2} \text{ as the values are integers} \end{aligned}$$

Combining the last inequality with inequality above, we get that $\sum_{i \in I} a_i = \frac{S}{2}$. Hence, $\mathcal{I}_1$ has a solution, namely I. This concludes the proof. □

Because of this NP-hardness result, solving efficiently the BlockingTopologicalCut problem seems out of reach. The optimization problem associated with the BlockingTopologicalCut

and TopologicalTraversability problems is that of finding the smallest value $k \leq M$ such that there is no topological cut of weight lower than or equal to k that only has successors with weight greater than k. By using such a k to restrict the scheduler (i.e., the rule is that it cannot schedule tasks that would make the total amount of memory used to exceed k), it would be guaranteed that the computation would never use more than memory k , independently of the max cut of the task graph.

Unfortunately, it turns out that an approximate value that is within a constant factor of the optimal value still might not be an acceptable solution. Figure 8 shows a task graph whose max cut is $M = 2v$ (the graph can be traversed with maximum memory $2v$). Values $k = 2v$ and $k = v$ are acceptable solutions (the graph can be traversed with maximum memory $2v$ or v), but $\frac{3}{2}v$ is not a solution (there exists a cut with value $\frac{3}{2}v$, and all of its successors have value larger than $\frac{3}{2}v$). In other words, the set of acceptable k values is not connected! The example given in Figure 8 illustrates the difficulty of the optimization problem.

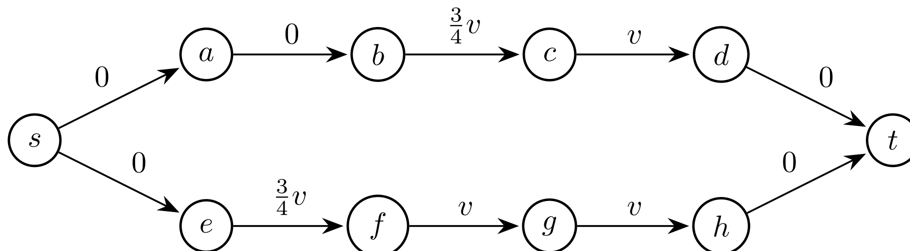

Figure 8: Example of DAG for which the set of admissible solutions is not connected: both v and $2v$ are solutions, but not $\frac{3}{2}v$.

6 Simulation Results

In this section, we perform simulations to assess the impact of resource constraints on the memory peak for dynamic schedulers. We also study whether the rounding heuristic described in Section 4.3 succeeds to compute a p-maxcut close to the optimal one.

6.1 Datasets

We used both synthetic task graphs and graphs from classical HPC applications. Specifically, we report experiments for five datasets. The first dataset is generated using the DAGGEN software [39]. We use the same parameters that were used to produce a dataset widely used in the scheduling literature [24, 15, 30]. These graphs count between 10 and 100 tasks. Five parameters influence the generation of these DAGs. The number of nodes belongs to $\{10, 25, 50, 100\}$. 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 get a dataset of 144 DAGs.

The next three datasets represent actual workflow applications and have been generated with the Pegasus Workflow Generator [12]. We consider three different applications, named Ligo, Montage, and Genome, each containing 20 graphs of 50 nodes and 20 graphs of 100 nodes. 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.

The last dataset consists in the task graphs of the qr_mumps [2] application, when applied on matrices from the University of Florida Sparse Matrix Collection [14]. These matrices were ordered

using either the `colamd` [13] or `scotch`[32] ordering. The 24 resulting task graphs are indeed trees of tasks whose size varies from 39 to 5900 nodes.

For all these graphs, we computed both the maximum topological cut (maxcut), the maximum topological cut with at most p red edges (p-maxcut) using the ILP Gurobi solver [23], and the solution returned by the heuristic described in Section 4.3. The C++ code used for the simulation is publicly available online at `https://github.com/GBathie/PMaxcut`.

6.2 Results

6.2.1 Comparing the Maxcut and the p-Maxcut

The first set of simulations studies the impact of the number of processors (the value of p) when computing the p-maxcut, comparing it with the maximum topological cut without any bound on resources ($p = \infty$). We plot in Figure 9 the ratios maxcut/p-maxcut obtained in all cases, using Tukey boxplots. The box presents the median, the first and third quartiles. The whiskers extend to up to 1.5 times the box height (interquartile range). While the results largely depend on the target, we observe globally that taking p into account when computing the maximum topological cut dramatically reduces its value in most cases. Note that, for better readability, we remove outliers from the plots, as they only concern special cases where the gain of using the p-maxcut instead of the maxcut was even higher (see below and Figure 10 for the outliers). For the Pegasus datasets, the value of the cut is reduced by a factor up to 24 (Genome with $p = 1$). For QR-Mumps, the value of the cut is reduced by a factor at most 1.7 ($p = 1$). For the DAGGEN datasets, this ratio goes up to 14. On most datasets, for low values of p, the p-maxcut yields an estimation of the maximum memory needed to schedule the task graphs that is much tighter than the maximum topological cut. However, in most cases, the ratio maxcut/p-maxcut decreases when the number of processors grows from 1 to 10, except for the MONTAGE graphs which exhibit a very large degree of parallelism.

Figure 10 complements the results of Figure 9 by showing outliers for the LIGO and QR-Mumps datasets. There are few of them, contrarily to the case of MONTAGE and GENOME where there are none. These results show that the approach is globally stable, but still can exhibit some unexpected behavior for some workflows.

6.2.2 Accuracy of the Heuristic

The heuristic presented in Section 4.3 is not guaranteed to give the optimal p-maxcut, and may output a cut with smaller value than the optimal one. Figure 11 presents the peak memory computed by the heuristic on all datasets, normalized to the optimal p-maxcut computed with the ILP. Thus, this value shows the ratio by which the p-maxcut is underestimated by the heuristic. We would like this ratio to be as close to 1 as possible. We use the same boxplots, except that outliers are drawn and appear separately as empty circles. For LIGO and MONTAGE, we observe that the heuristic is able to find a cut with a weight very close to optimal only for small values of p. For all the other datasets, the heuristic finds a p-maxcut which is at most 2% smaller than the optimal one in 99% of the cases. For the GENOME dataset, the heuristic always finds the optimal p-maxcut.

Table 1 provides a synthetic view of the results of the heuristic. The first two columns indicate the name of the dataset and number of processors p. The third column reports the average value of the ratio p-maxcut/p-maxcut*, where p-maxcut* is the estimation of the peak memory computed by the heuristic. In most cases, we observe that the heuristic always finds a value very close to the optimal value. On a global average, the difference is around 1%. The fourth and last column contains the number of simulations for which the heuristic failed to find the optimal result, over the size of the dataset. Overall, the heuristic failed to return the optimal value on less than 5% of the instances. Besides, even on datasets where the heuristic fails almost all the time to provide the optimal results, we notice that the average ratio between the result of the heuristic and the optimal p-maxcut is very small. Hence, we claim that in a large majority of cases, our heuristic is able to provide an accurate estimate of the p-maxcut with a very reasonable complexity.

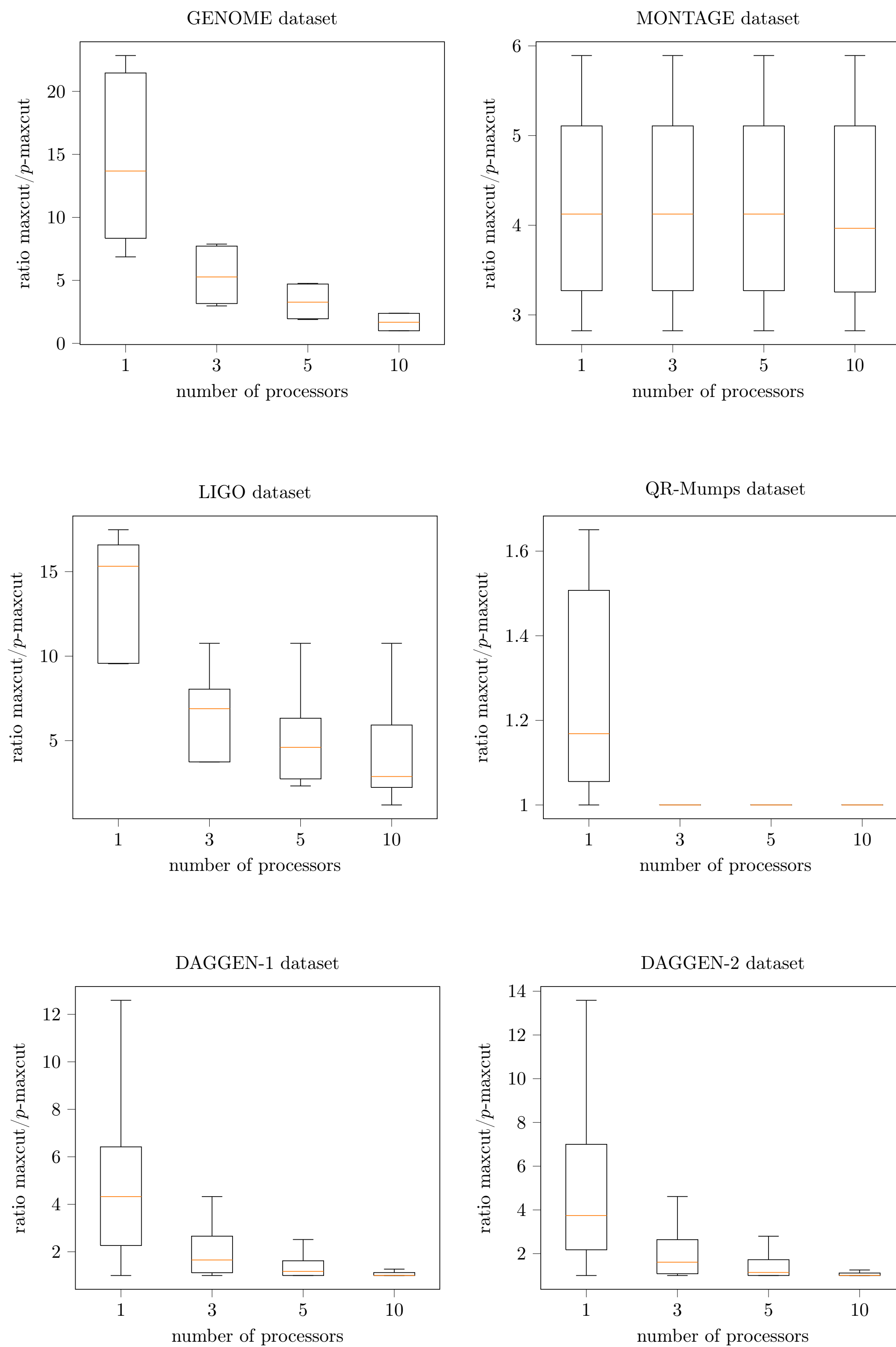

Figure 9: Influence of p when computing the p-maxcut for all datasets.

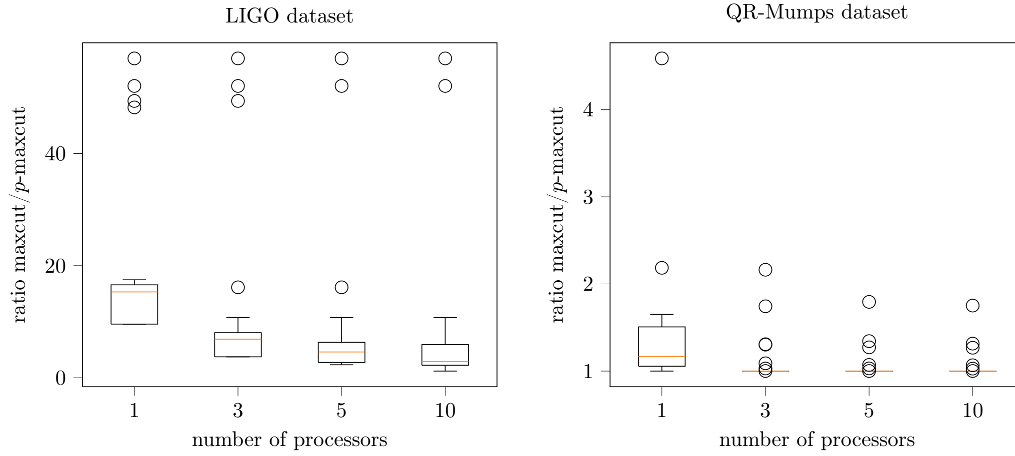

Figure 10: Complete results (with outliers) for LIGO and QR-Mumps.

Dataset	p	average value of p-maxcut/p-maxcut*	fraction of cases with p-maxcut$^* \neq p$-maxcut
GENOME	1	1.000	0/40
	3	1.000	0/40
	5	1.000	0/40
	10	1.000	0/40
LIGO	1	1.306	4/40
	3	1.836	13/40
	5	2.329	17/40
	10	4.001	29/40
MONTAGE	1	1.001	40/40
	3	1.011	40/40
	5	1.038	40/40
	10	1.148	39/40
QR_MUMPS	1	1.000	0/29
	3	1.006	3/29
	5	1.000	0/29
	10	1.003	2/29
DAGGEN-1	1	1.000	0/144
	3	1.031	8/144
	5	1.001	4/144
	10	1.001	3/144
DAGGEN-2	1	1.210	1/144
	3	1.023	6/144
	5	1.005	3/144
	10	1.001	2/144

Table 1: Summary of simulation results.

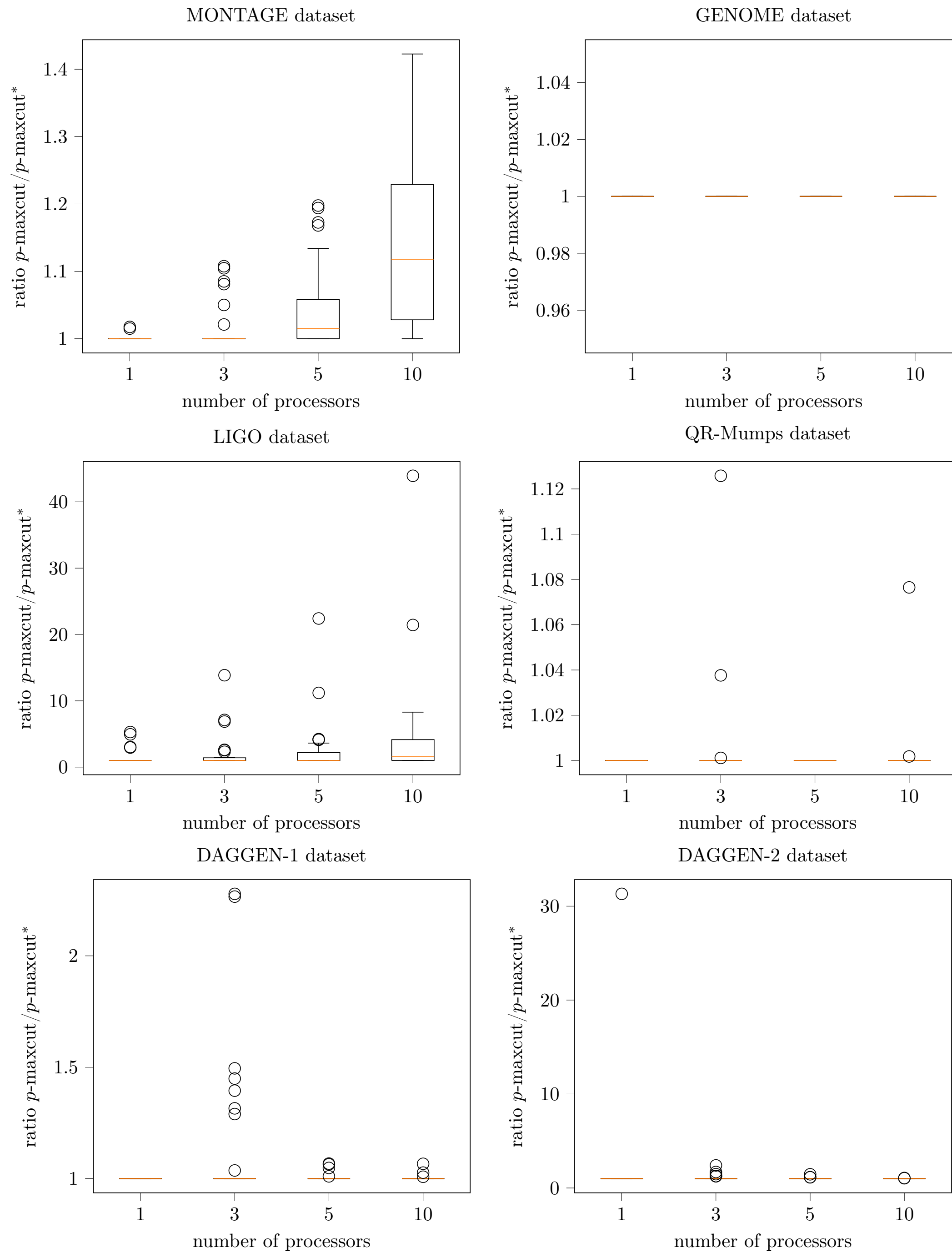

Figure 11: Complete results of the heuristic (with outliers) for all datasets. p-maxcut* is the estimation of the peak memory computed by the heuristic.

7 Conclusion and Future Work

In this paper, we have revisited dynamic DAG scheduling under memory constraints. We have introduced a new model that takes resource limitation into account when computing peak memory needs. By coloring those edges that represent temporary memory requirements during task execution, we bound the memory actually needed during an execution with p processors as a function of p, while previous work assumed unlimited resources. The additional constraints due to resource limitation turn an otherwise polynomial problem into an NP-hard problem. We have introduced an Integer Linear Program (ILP) to solve it, together with a heuristic based on rounding the rational solution of the ILP. Furthermore, we provide an exact polynomial algorithm for the particular case of series-parallel graphs. With an experimental study conducted over randomly-generated graphs and task graphs from actual applications, we show that our refined approach can significantly reduce the weight of the maximum topological cut. Finally, we have discussed an extended approach where the scheduler is dynamically constrained to select tasks (among ready tasks) so that the total memory used does not exceed some threshold. We have shown that this extension is not easier to deal with than the original problem.

Future work includes several promising directions. The first direction is to compare the ILP and the heuristic on task graphs of very large size, because we expect the ILP to fail providing a solution beyond a certain number of nodes. The second direction is to design efficient strategies to reduce peak memory in the refined model with colored edges, thereby extending previous approaches to the new model. Finally, the third direction would be to develop scheduling strategies that rely upon a coarse representation of the task graph instead of the complete graph, thereby allowing to deal with very large graphs while (hopefully) keeping a tight estimation of the total memory requirement. This would allow for an effective implementation of scientific applications at scale within a task-based runtime system.

Acknowledgement

We thank the reviewers for their helpful comments and suggestions.

References

[1] Emmanuel Agullo, Patrick R. Amestoy, Alfredo Buttari, Abdou Guermouche, Jean-Yves L'Excellent, and François-Henry Rouet. Robust memory-aware mappings for parallel multifrontal factorizations. *SIAM J. Scientific Computing*, 38(3), 2016.

[2] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Softw.*, 43(2):13, 2016.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[5] Gabriel Bathie, Loris Marchal, Yves Robert, and Samuel Thibault. Revisiting dynamic DAG scheduling under memory constraints for shared-memory platforms. In *22nd Workshop on Advances in Parallel and Distributed Computational Models APDCM 2020*. IEEE Computer Society Press, 2020.

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

[7] Hans L Bodlaender and Babette van Antwerpen-de Fluiter. Parallel algorithms for series parallel graphs and graphs with treewidth two 1. *Algorithmica*, 29(4):534–559, 2001.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering*, 15(6):36–45, 2013.

[9] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[10] H. Casanova, A. Legrand, and Y. Robert. *Parallel Algorithms.* Chapman and Hall/CRC Press, 2008.

[11] Gennaro Cordasco and Arnold L. Rosenberg. On scheduling series-parallel dags to maximize area. *Int. J. Found. Comput. Sci.*, 25(5):597–622, 2014.

[12] Rafael Ferreira Da Silva, Weiwei Chen, Gideon Juve, Karan Vahi, and Ewa Deelman. Community resources for enabling research in distributed scientific workflows. In *10th Int. Conf. on e-Science*, volume 1, pages 177–184. IEEE, 2014.

[13] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):377–380, 2004.

[14] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[15] Frédéric Desprez and Frédéric Suter. A bi-criteria algorithm for scheduling parallel task graphs on clusters. In *CCGrid*, pages 243–252. IEEE, 2010.

[16] Maciej Drozdowski. Scheduling multiprocessor tasks — an overview. *European Journal of Operational Research*, 94(2):215 – 230, 1996.

[17] Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. Parallel scheduling of task trees with limited memory. *ACM Transactions on Parallel Computing*, 2(2):13, 2015.

[18] Lucian Finta, Zhen Liu, Ioannis Mills, and Evripidis Bampis. Scheduling uet-uct series-parallel graphs on two processors. *Theoretical Computer Science*, 162(2):323–340, 1996.

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

[20] Thierry Gautier, Xavier Besseron, and Laurent 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.

[21] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM J. Comput.*, 9(3), 1980.

[22] A. S. Grimshaw and W. A. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[23] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020.

[24] Sascha 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.

[25] Mathias Jacquelin, Loris Marchal, Yves Robert, and Bora Uçar. On optimal tree traversals for sparse matrix factorization. In *Proc. of the Int. Par. & Dist. Processing Symposium (IPDPS)*, pages 556–567. IEEE, 2011.

[26] Tim Kaler, William Kuszmaul, Tao B. Schardl, and Daniele Vettorel. Cilkmem: Algorithms for analyzing the memory high-water mark of fork-join parallel programs. In *SIAM Symposium on Algorithmic Principles of Computer Systems*, 2020. Also available at `https://arxiv.org/abs/1910.12340`.

[27] Chi-Chung Lam, Thomas Rauber, Gerald Baumgartner, Daniel Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems & Structures*, 37(2):63–75, 2011.

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

[29] Loris Marchal, Hanna Nagy, Bertrand Simon, and Frédéric Vivien. Parallel scheduling of dags under memory constraints. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 204–213. IEEE, 2018.

[30] Loris Marchal, Bertrand Simon, and Frédéric Vivien. Limiting the memory footprint when dynamically scheduling dags on shared-memory platforms. *J. Parallel Distrib. Comput.*, 128:30–42, 2019.

[31] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2016.

[32] François Pellegrini and Jean Roman. Sparse matrix ordering with scotch. In *International Conference on High-Performance Computing and Networking*, pages 370–378. Springer, 1997.

[33] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with StarSs. *IJHPCA*, 23(3):284–299, 2009.

[34] Arun Ramakrishnan, Gurmeet Singh, Henan Zhao, Ewa Deelman, Rizos Sakellariou, Karan Vahi, Kent Blackburn, David Meyers, and Michael Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *CCGrid'07*, pages 401–409, 2007.

[35] Dragos Sbîrlea, Zoran Budimlić, and Vivek Sarkar. Bounded memory scheduling of dynamic task graphs. In *Proc. of PACT*, pages 343–356. ACM, 2014.

[36] Marc Sergent, David Goudin, Samuel Thibault, and Olivier Aumage. Controlling the memory subscription of distributed applications with a task-based runtime system. In *Proc. of IPDPS Workshops*, pages 318–327. IEEE, 2016.

[37] Ravi Sethi. Complete register allocation problems. In *STOC'73*, pages 182–195. ACM Press, 1973.

[38] Ravi Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

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

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

[41] Jacobo Valdes, Robert E Tarjan, and Eugene L Lawler. The recognition of series parallel digraphs. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12, 1979.

[42] Eduardo C Xavier. A note on a maximum k-subset intersection problem. *Information Processing Letters*, 112(12):471–472, 2012.