HAL
open science

Revisiting dynamic DAG scheduling under memory constraints for shared-memory platforms

Gabriel Bathie, Loris Marchal, Yves Robert, Samuel Thibault

To cite this version:

Gabriel Bathie, Loris Marchal, Yves Robert, Samuel Thibault. Revisiting dynamic DAG scheduling under memory constraints for shared-memory platforms. IPDPS - 2020 - IEEE International Parallel and Distributed Processing Symposium Workshops, May 2020, New Orleans / Virtual, United States. pp.1-10, 10.1109/IPDPSW50202.2020.00102 . hal-03024626

HAL Id: hal-03024626

https://inria.hal.science/hal-03024626

Submitted on 25 Nov 2020

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

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

Revisiting dynamic DAG scheduling under memory constraints for shared-memory platforms

Gabriel Bathie*, Loris Marchal*, Yves Robert*†, Samuel Thibault‡
*Laboratoire LIP, ENS Lyon, France
†University of Tennessee Knoxville, TN, USA
‡Laboratoire LaBRI, Univ. Bordeaux, France

***Abstract*—This work focuses on dynamic DAG scheduling under memory constraints. We target a shared-memory platform equipped with p parallel processors. We aim at bounding 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 provide an extensive set of experiments, both with randomly-generated graphs and with graphs arising form practical applications, which demonstrate the impact of resource constraints on peak memory usage.**

***Index Terms*—Workflow; task graph; dynamic scheduler; memory constraint; complexity.**

I. 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 (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 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 [1], [2]. 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 [3], [4], [5], [6], [7], [8] 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 III 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 [9], [10].

Consider a task graph whose internal nodes require a large volume of temporary data, such as graphs arising from multifrontal solvers [11]. 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 II. In our previous work [12], [13], 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, 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 III.

However, this previous work [12], [13] 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 was 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 provide an extensive set of experiments, both with randomly-generated graphs and with graphs arising form 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 II. We provide background on memory-aware scheduling in Section III. Then, Section IV is the core of the paper: we introduce the new model, assess its complexity, provide an optimal algorithm for Series Parallel Graphs, and discuss extensions. Section V is devoted to simulations both with randomly-generated graphs and with graphs arising form practical applications; we compare the solution compute 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 VI.

II. Related work

Memory and storage have always been limiting parameters for large computations, as outlined by the pioneering work of Sethi and Ullman [14] 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 [15] 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 [16]).

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 dependences oriented towards the root, which notably simplifies the problem: this is the case for sparse direct solvers [17] but also in quantum chemistry computations [18]. For such trees, memory-aware parallel schedulers have been proposed in [19], and the impact of processor mapping on memory consumption has been studied in [10].

The problem of general task graphs handling large data has been identified by Ramakrishnan et al. [9] who introduced clean-up jobs to reduce the memory footprint and propose some simple heuristics. Their work was continued by Bharathi et al. [20] 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 [21].

As explained in the introduction, we have previously proposed a way to restrict the potentially large memory needed for the traversal of a task graphs by adding fictitious edges [12], [13]. 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. [22]. 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 [23], 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 [12], [13] 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 [24]. 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).

III. Background

In Section III-A, we introduce the SimpleDataFlowModel [12], [13] to study memory usage for general DAGs. This model is a natural extension of the original pebble game [14], and of the model introduced by Liu for tree graphs [17]. Then in Section III-B, we discuss how to emulate more realistic models, and outline the limitations of the current approach.

A. 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 zero weight. An example of such a graph is illustrated in Figure 1.

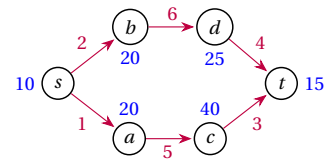

Fig. 1. Example of a workflow, (red) edge labels represent the size $m_{i,j}$ of associated data, while (blue) node labels represent their computation weight w_i.

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 $i \in V$:

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

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

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

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 the SimpleDataFlowModel and its usage in the literature [17], [12], [13].

We detail elementary transformations to account for more complex memory consumption rules in Section III-B. 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 dependences 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:

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

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 [13]). 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 (S, T) of a DAG G as 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 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 $O\left(|V||E| \log\left(|V|^2/|E|\right)\right)$[13]. 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 [12], [13]: we may need to insert $O(|V|^2)$ edges, each at a cost $O(|V|^3)$ if the DAG is dense (with $|E| = \Theta(|V|^2)$).

B. 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 $Inputs\,(i) + m_i^{temp} + Outputs\,(i)$. 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 $Inputs\,(i) + m_i^{temp} + Outputs\,(i)$. 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 $w_{i_2} = 0$.

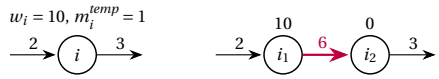

Fig. 2. Transformation of a task as in [25] (left) to the SIMPLEDATAFLOWMODEL (right).

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.

IV. RESOURCE CONSTRAINTS

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

A. Optimization problem

As outlined in Section III-B, when we transform an edge-weighted DAG G to the SIMPLEDATAFLOWMODEL, the resulting graph contains two different types of edges: those that correspond to edges of G, and those that correspond to computations (vertices of G). In terms of graph properties, this can be modeled as a 2-coloring of the edges. In what follows, computation edges are referred to as red edges, and communication edges as black edges. 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, if 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.

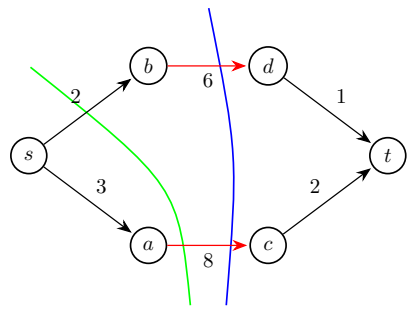

Fig. 3. Example of DAG for which the maxcut is an overestimation of the maximum memory used. The weight of the maxcut (in blue) is 14. For $p = 1$, the max cut with at most 1 computation edge (red edges, green cut) has weight 10.

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.

B. Complexity

As discussed in Section III-A, 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 very combinatorial:

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. For the completeness, we use a reduction from the MAX-K-SUBSETINTERSECTION (MSI) problem, which is NP-Complete [26]. The MSI problem is the following:

Definition 1. *Given a set X, $\mathcal{C} = \{S_i\}_{i \in [1,\ldots,l]}$ a set of l subsets of X, two integers $k \leq l$ and q, find a subset $I \subseteq [1, \ldots, l]$ such that $|I| = k$ and $\left|\bigcap_{i \in I} S_i\right| \geq q$. In other words, find k subsets S_i such that the cardinality of their intersection is greater 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 $n = |X|$. We build the following instance of P-MAXTOPCUT: $G = (V, E)$, where

$$V = \{s, t\} \cup \{u_i | i = 1, \ldots, l\} \cup \{v_j | j = 1, \ldots, n\}$$
$$E = \{(s, u_i) | i = 1, \ldots, l\} \cup \{(v_j, t) | j = 1, \ldots, n\}$$
$$\cup \{(u_i, v_j) | x_j \notin S_i\}$$

where the edges from s to the u_i are red and have weight $n + 1$, the other 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.

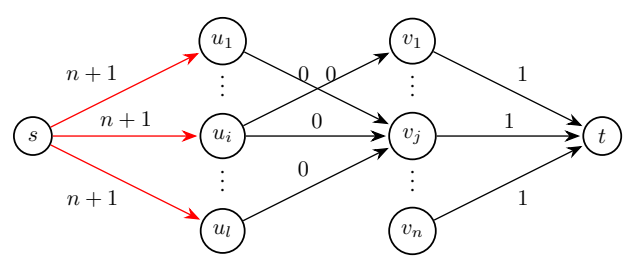

Fig. 4. DAG for the reduction: $(u_i, v_j) \in E \Leftrightarrow x_j \notin S_i$

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

$$S = \{s\} \cup \{u_i | i \notin I\} \cup \{v_j \mid \text{no predecessor of } v_j \text{ is in } S\}$$

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 $\mathcal{I}_2$.

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 that $(n+1)p+q$, 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, $\forall i, y \in C_i$. Hence, $y \in \bigcap_{i \in I} C_i$, and we get $Y \subseteq \bigcap_{i \in I} C_i$, and therefore $\left|\bigcap_{i \in I} C_i\right| \geq q$, therefore $\mathcal{I}_1$ has a solution.

Last, we show that this reduction is polynomial. The size of $\mathcal{I}_1$ is $n + l + \log(q)$. We do not need to count $\log(k)$ as $k \leq l$. The created instance $\mathcal{I}_2$ has $|V| = n + l + 2$ nodes and $|E| \leq n + l + nl$ edges, and weight $W = np + q$, therefore $\log(W) = \mathcal{O}(\log(n) + \log(p) + \log(q))$. Therefore $\mathcal{I}_2$ has size polynomial in the size of $\mathcal{I}_1$. Therefore, P-MAXTOPCUT is NP-complete. □

C. Integer Linear Program and Heuristic

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

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

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

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

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

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

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

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

The p variables are used to assign vertices to either S ($p_i = 1$) or T ($p_i = 0$). We consider that $isred_{i,j} = 1$ if $c(i,j) = red$ and $isred_{i,j} = 0$ otherwise. This ILP is adapted from the one from [13] 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 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 V: the heuristic always found a solution; furthermore, that solution was close to the optimal value in most cases (see Section V for details).

D. 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 Model) [27], [28]. 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.*

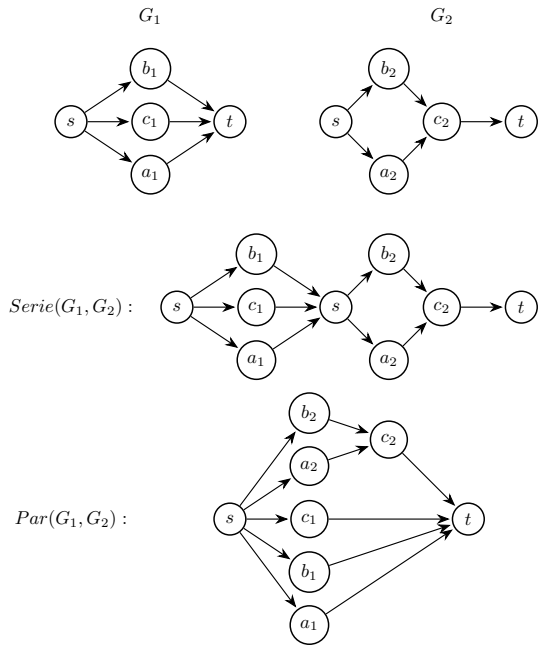

Fig. 5. Example of series and parallel composition of SPGs

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 [29], [30].

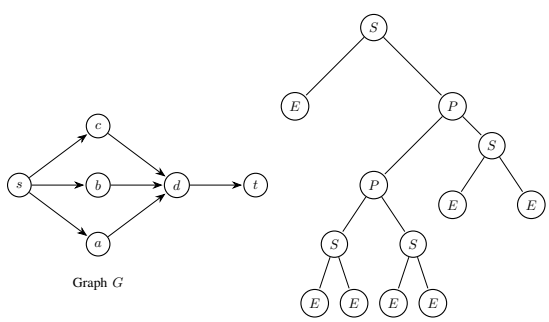

Fig. 6. Example of SP Graph (left) and its decomposition tree (right). $S =$ Series constructor, $P =$ Parallel, $E = Edge$.

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 $(\{s\}, \{t\})$.

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

$$M(E(m, r), k) = m, \forall k \geq 1, \forall r \in \{True, False\} \tag{9}$$

$$M(E(m, True), 0) = -\infty \tag{10}$$

$$M(E(m, False), 0) = m \tag{11}$$

$$M(Serie(G_1, G_2), k) = \max\{M(G_1, k), M(G_2, k)\} \tag{12}$$

$$M(Par(G_1, G_2), k) = \max_{j=0\ldots k}\{M(G_1, j) + M(G_2, k - j)\} \tag{13}$$

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…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 of leaf or has two children) with l leaves has exactly $2l - 1$ nodes[1]. 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 $\mathcal{O}(|E|p^2)$. □

V. 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 IV-C succeeds to compute a p-maxcut close to the optimal one.

A. 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 [31]. We use the same parameters that were used to produce a dataset widely used in the scheduling literature [32], [33], [13]. 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 [34]. We consider three different applications,

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

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 [35] application, when applied on matrices from the University of Florida Sparse Matrix Collection [36]. These matrices were ordered using either the `colamd` [37] or `scotch`[38] ordering. The 24 resulting task graphs are indeed trees of tasks whose size vary 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[39], and the solution returned by the heuristic. The C++ code used for the simulation is publicly available online at https://github.com/GBathie/PMaxcut.

B. Results

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 to the maximum topological cut without any bound on resources ($p = \infty$). We plot in Figure 7 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. For the Pegasus datasets, the value of the cut is reduced at least by a factor 1.6 (LIGO with $p = 10$) and at most by a factor 17 (LIGO with $p = 1$). For QR-Mumps, the value of the cut is reduced at least by 5% ($p = 10$) and at most by a factor 1.38 ($p = 1$). For the DAGGEN datasets, this ratio goes from 1.10 tup o 5.5. In most cases, the ratio p-maxcut/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 8 presents the results of the heuristic for the MONTAGE and LIGO datasets, normalized to the optimal p-maxcut computed with the ILP. We use the same boxplots, except that outliers are drawn and appear separately as empty circles. 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.

VI. 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 a NP-hard problem. We have introduced an Integer Linear Program (ILP) to solve it, together with an heuristic based on rounding the rational solution of the ILP. Furthermore, we provide an exact polynomial algorithm for the particular case of serial-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.

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. The third direction is to study the behavior of a restricted class of dynamic schedulers which try and select low memory-consuming tasks. For instance, instead of progressing to execute any ready task, these restricted schedulers would only select ready-tasks whose memory requirements keeps total memory consumption below a given threshold. The algorithmic complexity of such approaches will however probably remain very high. In particular, it is not clear how to fix the global threshold so that the restricted schedulers have a good chance to execute the whole task graph without exceeding the memory constraint. Finally, the fourth 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 application at scale within a task-based runtime system.

REFERENCES

[1] M. Drozdowski, "Scheduling multiprocessor tasks — an overview," *European Journal of Operational Research*, vol. 94, no. 2, pp. 215 – 230, 1996.

[2] 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*, vol. 13, no. 3, pp. 260–274, 2002.

[3] A. S. Grimshaw and W. A. Wulf, "The legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, 1997.

[4] 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*, vol. 23, no. 2, pp. 187–198, 2011.

[5] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. 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)*. IEEE, 2016, pp. 1–7.

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

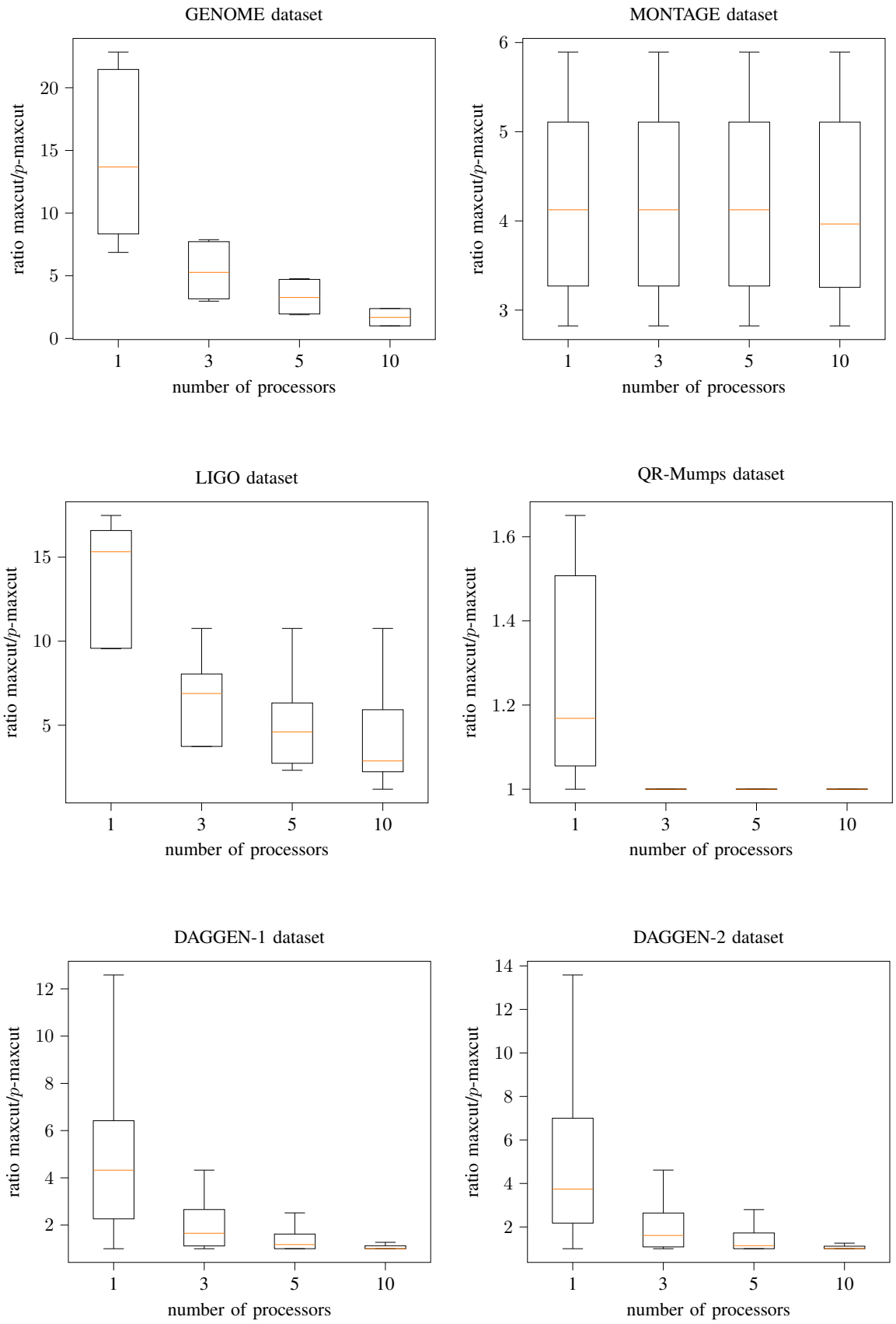

Fig. 7. Influence of p when computing the p-maxcut for all datasets.

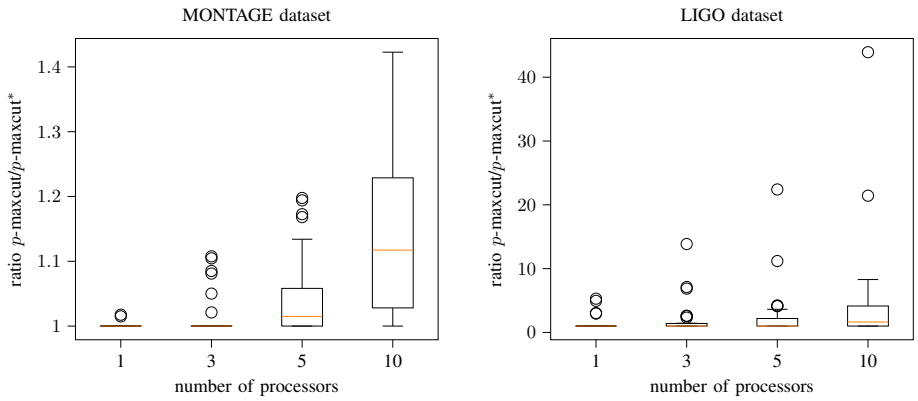

Fig. 8. Result of the heuristic for the MONTAGE and LIGO datasets.

[7] 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*, 2007, pp. 15–23.

[8] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *IJHPCA*, vol. 23, no. 3, pp. 284–299, 2009.

[9] 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 *CCGrid'07*, 2007, pp. 401–409.

[10] 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*, vol. 38, no. 3, 2016.

[11] 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*, vol. 23, no. 1, pp. 15–41, 2001.

[12] L. Marchal, H. Nagy, B. Simon, and F. Vivien, "Parallel scheduling of dags under memory constraints," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 204–213.

[13] L. Marchal, B. Simon, and F. Vivien, "Limiting the memory footprint when dynamically scheduling dags on shared-memory platforms," *J. Parallel Distrib. Comput.*, vol. 128, pp. 30–42, 2019. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.01.009

[14] R. Sethi and J. Ullman, "The generation of optimal code for arithmetic expressions," *Journal of the ACM*, vol. 17, no. 4, pp. 715–728, 1970.

[15] R. Sethi, "Complete register allocation problems," in *STOC'73*. ACM Press, 1973, pp. 182–195.

[16] J. R. Gilbert, T. Lengauer, and R. E. Tarjan, "The pebbling problem is complete in polynomial space," *SIAM J. Comput.*, vol. 9, no. 3, 1980.

[17] J. W. H. Liu, "An application of generalized tree pebbling to sparse matrix factorization," *SIAM J. Alg. Discrete Methods*, vol. 8, no. 3, pp. 375–395, 1987.

[18] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan, "Memory-optimal evaluation of expression trees involving large objects," *Computer Languages, Systems & Structures*, vol. 37, no. 2, pp. 63–75, 2011.

[19] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, "Parallel scheduling of task trees with limited memory," *ACM Transactions on Parallel Computing*, vol. 2, no. 2, p. 13, 2015.

[20] S. Bharathi and A. 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.

[21] M. Sergent, D. Goudin, S. Thibault, and O. Aumage, "Controlling the memory subscription of distributed applications with a task-based runtime system," in *Proc. of IPDPS Workshops*. IEEE, 2016, pp. 318–327.

[22] D. Sbîrlea, Z. Budimlić, and V. Sarkar, "Bounded memory scheduling of dynamic task graphs," in *Proc. of PACT*. ACM, 2014, pp. 343–356.

[23] 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*, vol. 18, no. 3-4, pp. 203–217, 2010.

[24] T. Kaler, W. Kuszmaul, T. B. Schardl, and D. 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.

[25] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar, "On optimal tree traversals for sparse matrix factorization," in *Proc. of the Int. Par. & Dist. Processing Symposium (IPDPS)*. IEEE, 2011, pp. 556–567.

[26] E. C. Xavier, "A note on a maximum k-subset intersection problem," *Information Processing Letters*, vol. 112, no. 12, pp. 471–472, 2012.

[27] G. Cordasco and A. L. Rosenberg, "On scheduling series-parallel dags to maximize area," *Int. J. Found. Comput. Sci.*, vol. 25, no. 5, pp. 597–622, 2014.

[28] L. Finta, Z. Liu, I. Mills, and E. Bampis, "Scheduling uet-uct series-parallel graphs on two processors," *Theoretical Computer Science*, vol. 162, no. 2, pp. 323–340, 1996.

[29] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *Proceedings of the eleventh annual ACM symposium on Theory of computing*, 1979, pp. 1–12.

[30] H. L. Bodlaender and B. van Antwerpen-de Fluiter, "Parallel algorithms for series parallel graphs and graphs with treewidth two 1," *Algorithmica*, vol. 29, no. 4, pp. 534–559, 2001.

[31] F. Suter, "Daggen: A synthetic task graph generator," https://github.com/frs69wq/daggen.

[32] 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*, vol. 27, no. 4, pp. 1010–1026, 2015.

[33] F. Desprez and F. Suter, "A bi-criteria algorithm for scheduling parallel task graphs on clusters," in *CCGrid*. IEEE, 2010, pp. 243–252.

[34] R. F. Da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling research in distributed scientific workflows," in *10th Int. Conf. on e-Science*, vol. 1. IEEE, 2014, pp. 177–184.

[35] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," *ACM Trans. Math. Softw.*, vol. 43, no. 2, p. 13, 2016. [Online]. Available: http://doi.acm.org/10.1145/2898348

[36] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[37] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, "Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 377–380, 2004. [Online]. Available: http://doi.acm.org/10.1145/1024074.1024080

[38] F. Pellegrini and J. Roman, "Sparse matrix ordering with scotch," in *International Conference on High-Performance Computing and Networking*. Springer, 1997, pp. 370–378.

[39] L. Gurobi Optimization, "Gurobi optimizer reference manual," 2020. [Online]. Available: http://www.gurobi.com