Model and complexity results for tree traversals on hybrid platforms

Julien Herrmann[1], Loris Marchal[1], and Yves Robert[1,2]

[1] École Normale Supérieure de Lyon, CNRS & INRIA, France
[2] University of Tennessee Knoxville, USA

Abstract We study the complexity of traversing tree-shaped workflows whose tasks require large I/O files. We target a heterogeneous architecture with two resources of different types, each equipped with its own memory, such as a multicore node equipped with a dedicated accelerator (FPGA or GPU). Tasks in the workflow are tagged with the type of resource that is needed for their processing. Besides, a task can be processed on a given resource only if all its input files and output files can be stored in the corresponding memory. At a given execution step, the amount of data stored in each memory strongly depends upon the ordering in which the tasks are executed, and upon when communications between both memories are scheduled. The objective is to determine an efficient traversal that minimizes the maximum amount of memory of each type needed to traverse the whole tree. In this paper, we establish the complexity of this two-memory scheduling problem, provide inapproximability results, and show how to determine the optimal depth-first traversal. Altogether, these results lay the foundations for memory-aware scheduling algorithms on heterogeneous platforms.

1 Introduction

Modern computing platforms are heterogeneous: a typical node is composed of a multi-core processor equipped with a dedicated accelerator, such as a FPGA or a GPU. Our goal is to study the execution of a computational workflow, described by an out-tree, onto such a heterogeneous platform, with the objective of minimizing the amount of memory of each resource needed for its processing. The nodes of the workflow tree correspond to tasks, and the edges correspond to the dependencies among the tasks. The dependencies are in the form of input and output files: each node accepts a (potentially large) file as input, and produces a set of files, each of them to be processed by a different child node. We consider in this paper that we have two different processing units at our disposal, such as a CPU and a GPU. For sake of generality, we designate them by a color (namely *blue* and *red*). Each task in the workflow is best suited to a given resource type (say a core or a GPU), and is *colored* accordingly. To execute a task of a given color, the input file and all the output files of the task must fit within the corresponding memory. As the workflow tree is traversed, tasks of different colors are processed, and capacity constraints on both memory types must be met. In addition, when a child of a task has a different color than its parent, say for example that a blue task has a red child, a communication from the blue memory to the red memory must be scheduled before the red child can be processed (and again, the input file and all output files of this red child must fit within the red memory). All these constraints require to carefully orchestrate the scheduling of the tasks, as well as the communications between memories, in order to minimize the maximum amount of each memory that is needed throughout the tree traversal.

Memory-aware scheduling is an important problem that has been the focus of many papers (see Section 2 for related work). This work mainly builds upon the pioneering work of Liu, who has studied tree traversals that minimize the peak amount of memory used on a homogeneous system, hence with a single memory type. Liu first restricted to depth-first traversals in [5], before dealing with an optimal algorithm for arbitrary traversals in [5]. The main objective of this paper is to extend these results to colored trees with two memory types, and tasks belonging to a given type. Clearly, the traversal, i.e., the order chosen to execute the tasks, and to perform the communications, plays a key role in determining which amount of each memory is needed for a successful execution of the whole tree. The interplay between both memories dramatically complicates the scheduling: it is no surprise that the complexity of the problem, that was polynomial with a unique memory, now becomes NP-complete.

In this paper, we concentrate on memory usage, but we are fully aware that performance aspects are important too, and that even more difficult trade-offs are to be found between parallel performance and memory consumption. One could envision a fully general framework, where tasks have different execution-times for each resource type (instead of being tied to a given resource as in this paper), and where concurrent execution of several tasks on each resource type is possible (instead of the fully sequential processing of the task graph that is assumed in this paper). Altogether, this study is only a first step towards the design of memory-aware schedules on modern heterogeneous platforms with two memory types. However, despite the apparent simplicity of the model, our results show that we already face a difficult bi-criteria optimization problem when dealing with two different memory types. We firmly believe that the results presented in this paper will help to lay the foundations for memory-aware scheduling algorithms on modern heterogeneous platforms such as those equipped with multicores and GPUs. Indeed, one key contribution of the paper is the derivation of several complexity results: NP-completeness of the problem, and inapproximability within a constant (α, β) factor pair of both absolute minimum memory amounts. Here the absolute minimum memory of a given type is computed when assuming an infinite amount of memory of the other type. Another important contribution is the determination of the optimal depth-first traversal, which turns out to minimize both memories simultaneously (among all possible depth-first traversals).

The rest of the paper is organized as follows. We start with an overview of related work in Section 2. Then we detail the framework in Section 3. We deal with complexity results in Section 4, which constitutes the heart of the paper. Finally we provide some concluding remarks and hints for future work in Section 5.

2 Related Work

The work presented in this paper builds upon previous results related to memory-aware scheduling, but its applications are relevant to the field of sparse matrix factorization and of hybrid computing.

2.1 Sparse matrix factorization

Determining a memory-efficient tree traversal is very important in sparse numerical linear algebra. The elimination tree is a graph theoretical model that represents the storage requirements, and computational dependencies and requirements, in the Cholesky and LU factorization of sparse matrices. In a previous study, we have described how such trees are built, and how the multifrontal method organizes the computations along the tree [4]. This is the context of the founding studies of Liu [5,6] on memory minimization for postorder or general tree traversals mentioned in Section 1. Recently, still in the context of a single memory type, an extension of these results to parallel machines base been proposed in [7].

2.2 Scientific workflows

The problem of scheduling a task graph under memory constraints also appears in the processing of scientific workflows whose tasks require large I/O files. Such workflows arise in many scientific fields, such as image processing, genomics or geophysical simulations. The problem of task graphs handling large data has been identified in [8] which proposes some simple heuristic solutions.

2.3 Pebble game and its variants

On the more theoretical side, this work builds upon the many papers that have addressed the pebble game and its variants. Scheduling a graph on one processor with the minimal amount of memory amounts to revisiting the I/O pebble game with pebbles of arbitrary sizes that must be loaded into main memory before *firing* (executing) the task. The pioneering work of Sethi and Ullman [10] deals with a variant of the pebble game that translates into the simplest instance of the problem with a unique memory and where all files have weight 1. The concern in [10] was to minimize the number of registers that are needed to compute an arithmetic expression. The problem of determining whether a general DAG can be traversed with a given number of pebbles has been shown NP-hard by Sethi [9]. However, this problem has a polynomial complexity for tree-shaped graphs [10].

2.4 Hybrid computing

Hybrid computing consists in the simultaneous use of CPUs and GPUs to optimize performance for high performance computing. Since CPUs and GPUs are powerful for specific and different tasks, its is natural to schedule a task on its "favorite" resource, that is, the resource where its execution time is minimal. This has been done successfully to increase performance in linear algebra libraries [11,3]. There also exist software tools that schedule an application composed of tasks with both CPU and GPU implementations on hybrid platforms: for instance, StarPU [1] optimizes the execution time of an application by scheduling its tasks on various resources based on predictions of execution and data transfer times.

3 Framework

As stated above, we deal with tree traversals on a two-memory system where each task belongs to a specific memory. Dependencies are in the form of input and output files: each task accepts a file as input from its parent node in the tree, and produces a set of files to be consumed by each child node.

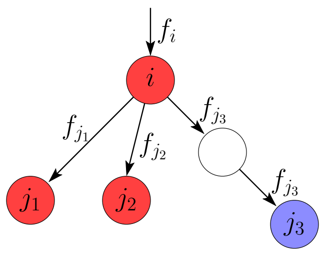

Figure 1: Illustration of a tree node with its children.

The tree work-flow $\mathcal{T}$ is composed of n nodes, or tasks, numbered from 1 to n, where $\mathit{Children}(i)$ denotes the set of the children of i. We consider here out-trees, where a parent node has to be processed before its children. As illustrated on Figure 1, each task (or node) i in the tree is characterized by the size f_i of its input file (data needed before the execution and received from its parent), which is the weight of the edge between the node and its parent, and by its *color*, which represents the specific memory where the task has to be executed. We let $\mathit{color}(i) \in \{\mathit{red}, \mathit{blue}\}$ represent the memory type of task i. If $\mathit{color}(i) = \mathit{red}$, then i is a computational node which operates in the *red* memory, which it uses to load its input file, execute its program and produce the set of output files for its children. Similarly, if $\mathit{color}(i) = \mathit{blue}$, then i is a computational node which operates in the *blue* memory. Each communication from one memory to the other is achieved through a communication node, which is uncolored. Hence, there are three types of nodes in the tree, *red* or *blue* computational nodes (or tasks), and uncolored communication nodes. Each time there is a data dependence between two tasks assigned to different memories, the output file of the source task has to be loaded from one memory into the other, using a communication node. Thus, in the model, the tree $\mathcal{T}$ does not contain direct edges between *blue* and *red* nodes; memory loads from one memory to the other occur only when processing a communication node. A *valid traversal* σ of the tree $\mathcal{T}$ is an ordered list of the nodes of $\mathcal{T}$ (including communication nodes) such that all node dependences in $\mathcal{T}$ are enforced by the schedule. Here are further details on the processing of each node type:

- Computational nodes: they represent a task executed on a specific memory. During the processing of a computational task i, the associated memory must contain the input file and its output files. Assuming that i is a *blue* task, the amounts of memory *BlueMemReq*(i) and *RedMemReq*(i) that are needed for this processing are thus:

$$\mathit{BlueMemReq}(i) = \left(\sum_{j \in \mathit{Children}(i)} f_j\right) + f_i, \qquad \mathit{RedMemReq}(i) = 0$$

  After task i has been processed, the input file is discarded, while its output files are kept in memory until the processing of its children. Thus, for a traversal σ of $\mathcal{T}$, the actual amounts of memory used to process the *blue* node i are:

$$\mathit{BlueMemUsed}(\sigma, i) = \left(\sum_{j \in \mathit{Children}(i)} f_j\right) + f_i + \sum_{j \in S_{blue} \setminus \{i\}} f_j,$$
$$\mathit{RedMemUsed}(\sigma, i) = \sum_{j \in S_{red}} f_j$$

  where S_{blue} (respectively S_{red}) denotes the set of files stored in the *blue* (respectively *red*) memory when the scheduler decides to execute task i. Note that S_{blue} must contain the input file of task i. After processing the *blue* node i, we have:

$$S_{blue} \leftarrow (S_{blue} \setminus \{i\}) \cup \mathit{Children}(i), \quad S_{red} \leftarrow S_{red}$$

  Initially, S_{blue} contains the input file of the root and $S_{red} = \emptyset$ if the root is a *blue* node, and conversely if the root is a *red* node.
- Communication nodes represent communications between one memory and the other. Each communication node i has an input file of size f_i and an output file of the same size. It loads f_i units of memory from one memory to the other. During the processing of a communication task i from the *blue* memory to the *red* memory, both memories must contain the file of size f_i. Thus, the amount of *blue* and *red* memory needed for this processing is f_i:

$$\mathit{BlueMemReq}(i) = f_i, \quad \mathit{RedMemReq}(i) = f_i$$

After i has been processed, the input file from the *blue* memory is discarded, while the output file is kept in the *red* memory until the processing of its child. Thus, for a traversal σ of $\mathcal{T}$, the actual amounts of memory used to process the communication node i are:

$$\mathit{BlueMemUsed}(\sigma, i) = f_i + \sum_{j \in S_{blue} \setminus \{i\}} f_j, \quad \mathit{RedMemUsed}(\sigma, i) = f_i + \sum_{j \in S_{red}} f_j$$

Note that S_{blue} must contain the input file of task i. Letting j denote the unique child of communication node i, we have after the execution of i that:

$$S_{blue} \leftarrow S_{blue} \setminus \{i\}, \quad S_{red} \leftarrow S_{red} \cup \{j\}$$

It is important to stress that a communication node need not be processed right after the execution of its parent. The only constraint is that its processing must precede the execution of its unique child. This flexibility in the schedule severely complicates the search for efficient traversals.

As stated above, we face a multi-criteria optimization problem: how to minimize the amount of both memories needed for the tree traversal? The *peak memory* is the maximum usage of each memory over the whole schedule σ of the tree $\mathcal{T}$, and is defined for the *blue* and the *red* memory by:

$$M^{\sigma}_{\text{blue}}(\mathcal{T}) = \max_i \ \mathit{BlueMemUsed}(\sigma, i), \quad M^{\sigma}_{\text{red}}(\mathcal{T}) = \max_i \ \mathit{RedMemUsed}(\sigma, i)$$

Thus, we define the optimal peak for each memory needed to process a tree $\mathcal{T}$ as:

$$M^{\text{opt}}_{\text{blue}}(\mathcal{T}) = \min_{\sigma} \ M^{\sigma}_{\text{blue}}(\mathcal{T}), \quad M^{\text{opt}}_{\text{red}}(\mathcal{T}) = \min_{\sigma} \ M^{\sigma}_{\text{red}}(\mathcal{T})$$

We point out that $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ can be seen as the minimum amount of *blue* memory required to traverse the tree when there is an unbounded amount of *red* memory available: a schedule which reaches $M^{\sigma}_{\text{blue}}(\mathcal{T}) = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ can use an arbitrary amount of *red* memory. Intuitively, one may ask what are trade-offs between the *blue* and *red* memory requirements of feasible schedules. One major objective of this paper is to provide quantitative answers to this question.

Top-down vs. bottom-up traversals. We conclude this section with two remarks on the model. First, we can handle the case where a node in the tree needs an execution file (in addition to input and output files) by adding an extra child to the node, whose input file has the size of the execution file. Second, there is a complete equivalence with top-down traversals of out-trees (the problem addressed in this paper) and bottom-up traversals of in-trees (as used in sparse matrices factorization). In a nutshell, one only needs to reverse the direction of the edges, and to execute the schedule backwards, to move from one variant to another[3]. In fact, the literature deals with both variants. The seminal paper of Liu [5] originally deals with post-order bottom-up traversals for in-trees, while we speak of depth-first top-down traversals for out-trees in this paper, but there is no actual difference.

[3] This equivalence has been formally proven in [4] for single-memory platforms, and it is straightforward to extend the proof for two-memory systems.

4 Complexity results

This section presents several important complexity results. We start with the NP-completeness of the two-memory minimization problem in Section 4.1. Next we show in Section 4.2 that the problem reduces to traversing uncolored trees when one memory is unbounded. Then, we prove in Section 4.3 that it is impossible to approximate both minimum memories within arbitrary constant factors. Finally, we determine the optimal depth-first traversal (the equivalent of post-order traversals for in-trees). Due to lack of space, only the inapproximability proof is detailed in Section 4.3. All other proofs are available in the companion research report [2].

4.1 Hardness of the problem

Our first result assesses the complexity of the problem, as formulated in the following definition.

Definition 1 (TwoMemoryTraversal). *Given a tree $\mathcal{T}$ with n nodes, and two fixed memory amounts M_{red} and M_{blue}, does there exist a traversal σ of the tree such that $M^{\sigma}_{\text{blue}}(\mathcal{T}) \leq M_{\text{blue}}$ and $M^{\sigma}_{\text{red}}(\mathcal{T}) \leq M_{\text{red}}$?*

Theorem 1. *The* TwoMemoryTraversal *problem is NP-complete.*

The proof relies on a reduction from the 2-partition problem: consider an instance of 2-partition with n integers a_i of sum S . The reduction uses the tree illustrated in Figure 2, with maximum memory amounts set to $M_{red} = 3S$ and $M_{blue} = 2S$. Assuming without loss of generality that R_{root} is processed before $R^{(2)}_{root}$, it is possible to prove that if the processing of the tree does not exceed the prescribed memory bounds, then a subset I of the C_i such that $\sum_{i \in I} a_i = S/2$ has to be processed before R_{big}. The detailed proof of this result is available in [2].

4.2 When one memory is unbounded

In this section, we focus on the computation of $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ (or $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$) which represents the minimal peak memory reachable when there is no constraint on the other memory. We show that the computation of $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ and $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ for a bi-colored tree $\mathcal{T}$ reduces to the computation of the minimal peak memory for an uncolored tree.

Definition 2. *Given a bi-colored tree $\mathcal{T}$, we construct the corresponding uncolored (or for convenience, single-colored) tree $\mathcal{T}_{\text{blue}}$ by turning every communication node and* red *node into a* blue *node, and by turning every* red *edge of weight f_i into a* blue *edge of weight 0. We construct the single-colored tree $\mathcal{T}_{\text{red}}$ in a similar way. We let M^{∞}_{blue} denote the minimal amount of memory needed to process $\mathcal{T}_{\text{blue}}$ (and similarly, M^{∞}_{red} for $\mathcal{T}_{\text{red}}$).*

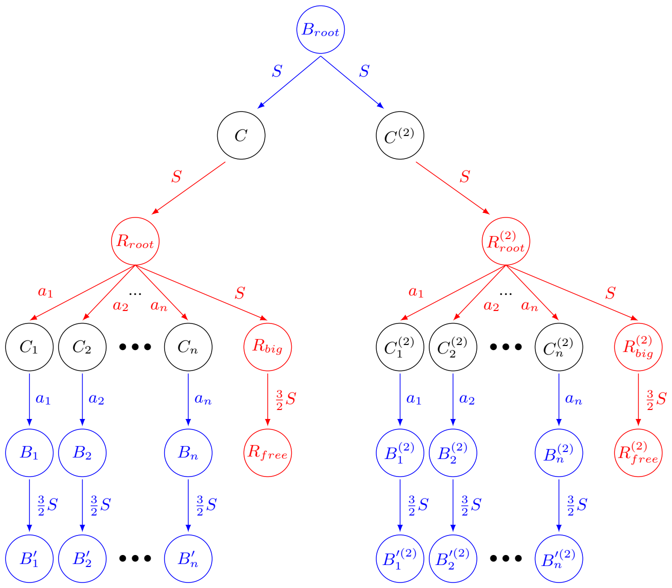

Figure 2: Tree used in the proof of Theorem 1

The following result is straightforward.

Theorem 2. *For any bi-colored tree $\mathcal{T}$, we have $M^{\infty}_{\text{red}} = M^{\text{opt}}_{\text{red}}(\mathcal{T})$ and $M^{\infty}_{\text{blue}} = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$.*

4.3 Joint minimization of both objectives

Since the traversal problem is NP-complete, it is natural to wonder whether there exist approximation algorithms. In this section, we prove that there does not exist schedules that approximates both minimum memories $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ and $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ within arbitrary constant factors for any bi-colored tree $\mathcal{T}$. Since the (usually unfeasible) point of the Pareto diagram with coordinates $(M^{\text{opt}}_{\text{blue}}(\mathcal{T}),M^{\text{opt}}_{\text{red}}(\mathcal{T}))$ is sometimes called the *Zenith*, this result amounts to proving that there exists no *Zenith*-approximation.

Definition 3. *Given a bi-colored tree $\mathcal{T}$, we can construct the corresponding uncolored tree $\mathcal{T}_{\text{unco}}$ by turning every colored node of $\mathcal{T}$ into an uncolored node. We let $M^{\text{opt}}_{\text{unco}}(\mathcal{T}_{\text{unco}})$ be the minimal amount of memory needed to process $\mathcal{T}_{\text{unco}}$.*

The following lemma, whose simple proof can be found in [2], is helpful to prove the following theorem.

Lemma 1. *Given a bi-colored tree $\mathcal{T}$ with n nodes, consider an arbitrary traversal σ of $\mathcal{T}$ that requires an amount of* red *memory equal to $M^{\sigma}_{\text{red}}(\mathcal{T})$ and an amount of* blue *memory equal to $M^{\sigma}_{\text{blue}}(\mathcal{T})$. Then necessarily:*

$$M^{\sigma}_{\text{red}}(\mathcal{T}) + M^{\sigma}_{\text{blue}}(\mathcal{T}) \geq M^{\text{opt}}_{\text{unco}}(\mathcal{T}_{\text{unco}})$$

Theorem 3. *Given two constants α and β, there exists no algorithm that is both an α-approximation for* blue *memory peak minimization and a β-approximation for* red *memory peak minimization, when scheduling bi-colored trees.*

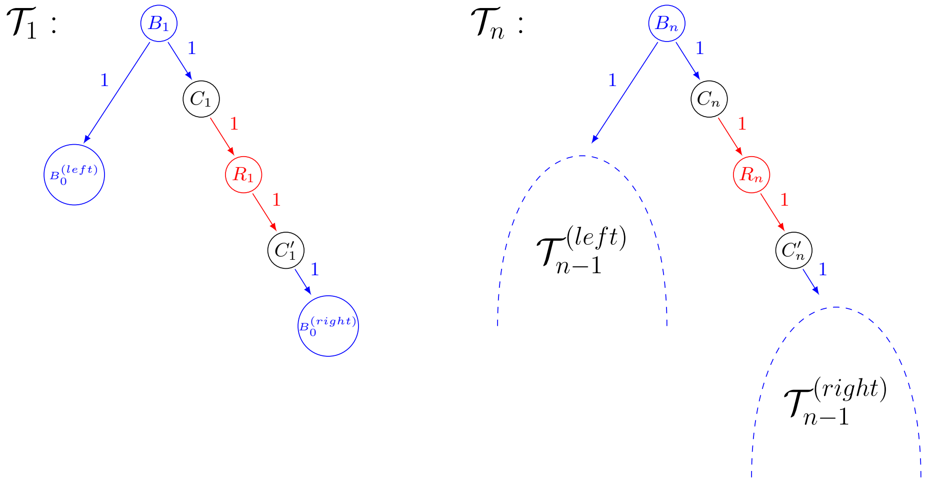

Figure 3: Recursive definition of $\mathcal{T}_n$ in the proof of Theorem 3

Proof. To establish this result, we proceed by contradiction. We therefore assume that there is a constant α, a constant β, and an algorithm $\mathcal{A}$ that processes any bi-colored tree $\mathcal{T}$ using a *blue* peak memory that is not greater than α times the optimal *blue* peak memory $M_{\text{blue}}^{\text{opt}}(\mathcal{T})$ and using a *red* peak memory that is not greater than β times the optimal *red* peak memory $M_{\text{red}}^{\text{opt}}(\mathcal{T})$. To derive the contradiction, we use the family of tree $(\mathcal{T}_n)_{n\in\mathbb{N}}$ depicted on Figure 3. $\mathcal{T}_n$ is defined recursively using $\mathcal{T}_{n-1}$.

We prove the following statements:

– $\mathbf{\forall n \geq 2, M_{\text{blue}}^{\text{opt}}(\mathcal{T}_n) = 3}$
  Consider the traversal σ_{blue} that processes $\mathcal{T}_n$ as follows:
  - If $n = 0$, σ_{blue} processes the node B_0
  - If $n > 0$, σ_{blue} processes the nodes B_n and C_n. Then $\mathcal{T}_{n-1}^{(left)}$ is processed recursively. Nodes R_n and C'_n follow. And finally $\mathcal{T}_{n-1}^{(right)}$ is processed recursively.

  At each step of this process, the traversal σ_{blue} does not use more than 3 units of *blue* memory. Since *BlueMemReq*$(B_{n-1}) = 3$, this proves that $M_{\text{blue}}^{\text{opt}}(\mathcal{T}_n) = 3$.

– $\mathbf{\forall n \geq 1, M_{\text{red}}^{\text{opt}}(\mathcal{T}_n) = 2}$
  Consider the traversal σ_{red} that processes $\mathcal{T}_n$ as follows. At step k:
  - If $k = 0$, σ_{red} processes the node B_0

- If $k > 0$, σ_{red} processes the nodes B_k. Then $\mathcal{T}_{k-1}^{(left)}$ is processed recursively. Nodes C_k, R_k and C'_k follow. And finally $\mathcal{T}_{k-1}^{(right)}$ is processed recursively.

At each step of this process, the traversal σ_{red} does not use more than 2 units of *red* memory. Since $\mathit{RedMemReq}(R_n) = 2$, this proves that $M_{\text{red}}^{\text{opt}}(\mathcal{T}_n) = 2$.

– Let $\mathcal{T}_n^{\text{unco}}$ be the uncolored tree corresponding to $\mathcal{T}_n$ as describe in Definition 3 and $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_n^{\text{unco}})$ the minimum amount of memory required to execute it. We now prove by induction that $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_n^{\text{unco}}) = n + 2$ for $n \geq 2$. As show in [6], depth-first traversals (called *post-order* traversals in [6]) traversals are optimal for peak memory minimization of uncolored trees with unit costs. Besides, all depth-first traversals of $\mathcal{T}_n^{\text{unco}}$ require the same amount of memory. Thus $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_n^{\text{unco}}) = M_{\text{unco}}^{\text{opt}}(\mathcal{T}_{n-1}^{\text{unco}}) + 1$ for $n \geq 2$. Since $M_{\text{unco}}^{\text{opt}}(\mathcal{T}_1^{\text{unco}}) = 2$, we have the result.

By hypothesis, algorithm $\mathcal{A}$ can process any $\mathcal{T}_n$ with $M_{\text{blue}}^{\mathcal{A}}(\mathcal{T}_n) \leq \alpha . M_{\text{blue}}^{\text{opt}}(\mathcal{T}_n) = 3\alpha$ and $M_{\text{red}}^{\mathcal{A}}(\mathcal{T}_n) \leq \beta . M_{\text{red}}^{\text{opt}}(\mathcal{T}_n) = 2\beta$. Let $n_0 = \lceil 3\alpha + 2\beta \rceil$, we have:

$$\begin{aligned} M_{\text{blue}}^{\mathcal{A}}(\mathcal{T}_{n_0}) + M_{\text{red}}^{\mathcal{A}}(\mathcal{T}_{n_0}) &\leq 3\alpha + 2\beta \\ &< \lceil 3\alpha + 2\beta \rceil + 2 \\ &= M_{\text{unco}}^{\text{opt}}(\mathcal{T}_{n_0}^{\text{unco}}) \end{aligned}$$

This contradicts Lemma 1, which means that such an algorithm $\mathcal{A}$ cannot exist.

4.4 Depth-first traversals

Definition 4. *A depth-first traversal is a feasible traversal that processes all nodes of a tree $\mathcal{T}$ by processing the root and, then, recursively processing all sub-trees. Hence, in a post-order traversal, after processing a node i, the whole sub-tree rooted at i is completely processed before any other node that does not belong to this sub-tree. Formally, a feasible traversal σ of the tree $\mathcal{T}$ with n nodes is a depth-first traversal if and only if for each node $r \in \mathcal{T}$, with two children $i \in \mathit{Children}(r)$ and $j \in \mathit{Children}(r)$, we have:*

$$\sigma(i) < \sigma(j) \Rightarrow (\forall u \in T_i, \sigma(u) < \sigma(j))$$

where T_i is the sub-tree rooted at the node i.

In the context of single-memory trees, depth-first traversals are known to be sub-optimal [6]: worse, their memory usage can be arbitrarily high as compared to that of the optimal solution [4]. Clearly, these negative results remain true in a two-memory framework (simply assume that one memory is infinite). Still, depth-first traversals are a natural heuristic for traversing tree graphs, and they enjoy a simple implementation and memory management. As such, they are the most commonly used traversals in actual sparse solvers. Algorithm 1 computes the optimal depth-first traversal: when it encounters a blue node (respectively a red node), it applies the rule for minimizing the blue (resp. red) memory in depth-first traversals, which does not impact the amount of red (resp. blue) memory. It turns out that this traversal is optimal among all depth-first traversals for both memory usages (see proof in [2]).

Theorem 4. *Algorithm 1 returns the best depth-first traversal σ of $\mathcal{T}$ for both the* blue *and the* red *memories and the amount of memory M^{blue} and M^{red} used by σ.*

Algorithm 1: BestDepthFirstTraversal($\mathcal{T}$)

```
output: Schedule $\sigma$ with peak blue memory $M^{blue}$ and peak red memory $M^{red}$
root ← the root of $\mathcal{T}$ ;
$CurrentMem \leftarrow 0$;
$(\sigma, M^{blue}, M^{red}) \leftarrow ([\text{root}], 0, 0)$;
for $i \in Children(root)$ do
    $(\sigma_i,\ M_i^{blue},\ M_i^{red}) \leftarrow$ BestDepthFirstTraversal($T_i$);
    $CurrentMem \leftarrow CurrentMem + f_i$
if color($root$) = blue then
    for $i \in Children(root)$ in the increasing order of $M_i^{\text{blue}} - f_i$ do
        $\sigma \leftarrow [\sigma; \sigma_i]$;
        $CurrentMem \leftarrow CurrentMem - f_i$;
        $M^{blue} \leftarrow \max(M^{blue}, CurrentMem + M_i^{blue})$;
    $M^{red} \leftarrow \max_{i \in Children(root)} M_i^{red}$;
if color($root$) = red then
    for $i \in Children(root)$ in the increasing order of $M_i^{\text{red}} - f_i$ do
        $\sigma \leftarrow [\sigma; \sigma_i]$;
        $CurrentMem \leftarrow CurrentMem - f_i$;
        $M^{red} \leftarrow \max(M^{red}, CurrentMem + M_i^{red})$;
    $M^{blue} \leftarrow \max_{i \in Children(root)} M_i^{blue}$;
if the root node is an uncolored communication node then
    i ← the unique child of root; $\sigma \leftarrow [\sigma; \sigma_i]$;
    if color($i$) = blue then
        $M^{blue} \leftarrow M_i^{blue}$;
        $M^{red} \leftarrow \max(f_i, M_i^{red})$;
    if color($i$) = red then
        $M^{red} \leftarrow M_i^{red}$;
        $M^{blue} \leftarrow \max(f_i, M_i^{blue})$;
return $(\sigma,\ M^{blue},\ M^{red})$;
```

5 Conclusion

In this paper, we have studied the bi-criteria memory minimization problem that arises when traversing a task tree for a system composed of two different computing units with their own memory. After relating this problem to the well-studied one-memory problem, we have proved that the search for an optimal solution is NP-complete, and that it was impossible to approximate both memories by any pair of constant factors. In addition, we have determined the optimal depth-first traversal, which turns out to minimize both memories simultaneously.

Admittedly, the platform model used in this paper is a simplified one, but this was the key to derive complexity results in this initial study. In future work, the model should be refined in several directions, so as to more accurately account for all the characteristics of hybrid platforms (using both CPUs and GPUs); however, this is not expected to change the NP-completeness results. A first step towards a more realistic model would be to include computation times for the tasks, and to try to minimize both the processing time of the total tree, and the amount of blue and red memories needed. A second step would consist in providing each task with two different running times rather than a color, and to give the ability for the scheduler to choose the computing unit for each task based on running time and memory. Given the complexity of the problem in the simple case, we do not expect to find approximation algorithms, but rather to design simple heuristics (as BestDepthFirst) that may be optimal under restrictive conditions, either on the traversal type or on the tree structure.

References

1. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.
2. J. Herrmann, L. Marchal, and Y. Robert. Tree traversals with task-memory affinities. Research report 8226, INRIA, 2013.
3. M. Horton, S. Tomov, and J. Dongarra. A class of hybrid lapack algorithms for multicore and gpu architectures. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 150 –158, july 2011.
4. M. Jacquelin, L. Marchal, Y. Robert, and B. Ucar. On optimal tree traversals for sparse matrix factorization. *IPDPS'11*, 2011.
5. J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Software*, 12(3):249–264, 1986.
6. J. W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM J. Algebraic Discrete Methods*, 8(3), 1987.
7. L. Marchal, O. Sinnen, and F. Vivien. Scheduling tree-shaped task graphs to minimize memory and makespan. Research report 8082, INRIA, 2012. Accepted for publication in IPDPS 2013.
8. 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*. IEEE, 2007.
9. R. Sethi. Complete register allocation problems. In *STOC'73*, pages 182–195. ACM Press, 1973.
10. R. Sethi and J. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970.
11. S. Tomov, R. Nath, P. Du, and J. Dongarra. *MAGMA version User's guide*. available at `http://icl.eecs.utk.edu/magma/(2009)`.