Memory-aware tree traversals with pre-assigned tasks

Julien Herrmann[a], Loris Marchal[a], Yves Robert[a,b]

[a]*Ecole Normale Supérieure de Lyon, CNRS/INRIA, France*
[b]*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 resource types, each with a different memory, such as a multicore node equipped with a dedicated accelerator (FPGA or GPU). The tasks in the workflow are colored according to their type and can be processed if all there input and output files can be stored in the corresponding memory. The amount of used memory of each type at a given execution step 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, and provide inapproximability results. In addition, we design several heuristics, based on both post-order and general traversals, and we evaluate them on a comprehensive set of tree graphs, including random trees as well as assembly trees arising in the context of sparse matrix factorizations.

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

Email addresses: julien.herrmann@ens-lyon.fr (Julien Herrmann), loris.marchal@ens-lyon.fr (Loris Marchal), yves.robert@ens-lyon.fr (Yves Robert)

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 [17], before dealing with an optimal algorithm for arbitrary traversals in [17]. In many situations, the optimal traversal is a depth-first traversal, but this is not always the case. An assessment of the relative performance of depth-first traversals versus optimal traversals is proposed by [14]. 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 major contribution is the study of depth-first traversals and related variants. We show how to extend Liu's algorithm to compute the best depth-first traversal, which simultaneously minimizes both memory usages. However, while depth-first traversals were natural algorithms with a single memory, they severely constrain the activation of communication nodes with two memories. We show that the optimization problem is still NP-complete when relaxing the firing of communication nodes in depth-first traversal, which leads us to go beyond depth-first traversals and to introduce general heuristics. These heuristics extends Liu's optimal algorithm along various (greedy) decision criteria to trade-off the usage of both memory types.

Finally, the third major contribution is a comprehensive assessment of all these heuristics using both randomly generated trees, and actual elimination trees that arise from the multifrontal factorization of sparse linear systems.

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. The next four sections constitute the heart of the paper. We deal with complexity results in Section 4. Section 5 is devoted to the study of depth-first traversals, a first class of (widely-used) heuristics. Then we introduce additional heuristics in Section 6. The experimental evaluation of all the heuristics is conducted in Section 7. Finally we provide some concluding remarks and hints for future work in Section 8.

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. In this section, we present related work for each domain.

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 [14]. This is the context of the founding studies of Liu [17, 18] on memory minimization for postorder or general tree traversals mentioned in Section 1. Memory minimization is still a concern in modern multifrontal solvers when dealing with large matrices. In particular, efforts have been made to design dynamic schedulers that takes into account dynamic pivoting (which impacts the weights of edges and nodes) when scheduling elimination trees with strong memory constraints [11], or to consider both task and tree parallelism with memory constraints [1]. Recently, still in the context of a single memory type, an extension of these results to parallel machines has been proposed in [**?**]. While these studies try to optimize memory management in existing parallel solvers, we aim at designing a simple model to study the fundamental underlying scheduling problem.

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 [21] which proposes some simple heuristic solutions. Surprisingly, in the context of quantum chemistry computations, Lam et al. [16] have recently rediscovered the algorithm published in 1987 in [18].

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 [23] 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 [23] 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 [22] 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 [9]). However, this problem has a polynomial complexity for tree-shaped graphs [23]. Recently, still in the contact of a single memory type, an extension of these results to parallel machines base been proposed in [19].

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 [24, 13]. There also exist software tools that schedule an application composed of tasks with both CPU and GPU implementations on hybrid platforms: many frameworks have recently been proposed for such hybrid task scheduling, such as StarPU [4], DAGuE and PaRSEC [6, 5] or StarSs [20]. However, these schedulers are dynamic, i.e., they make their decisions at runtime, based for example on the expected duration of the tasks on the different kind of processing units (CPUs, GPUs,...), and are mostly interested in execution time, and not memory footprint. On the contrary, in this paper, we study the offline problem of scheduling a tree of task to reduce memory on a hybrid system. Our study can be seen as a first theoretical step to take memory constraints into account in such dynamic hybrid schedulers.

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. We start this section by formally writing all the constraints that need to be satisfied during a traversal: for the convenience of the reader, we briefly review the constraints for uncolored trees (single memory) in Section 3.1, before dealing with those for bi-colored trees (two memories) in Section 3.2. Also, we work out a small example in Section 3.3. Finally, we state the target optimization problems in Section 3.4.

3.1. Uncolored trees

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. 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), and by the size n_i of its execution file. A *valid traversal* σ of the tree $\mathcal{T}$ is an ordered list of the nodes of $\mathcal{T}$ such that all precedence constraints in $\mathcal{T}$ are enforced by the schedule. Since the nodes of $\mathcal{T}$ are numbered from 1 to n, σ can be seen as a permutation of $[\![1, n]\!]$, where $\forall i \in [\![1, n]\!]$ and $\forall j \in \mathit{Children}(i)$, $\sigma(i) < \sigma(j)$.

- Each node i in the tree has an input file of size f_i. If i is not the root, its input file is produced by its parent $\mathit{parent}(i)$; if i is the root, its input file may be of size zero, or it may contain input from the outside world.
- Each node i in the tree has an execution file of size n_i. This execution file can be modeled by adding an extra child to the node, as depicted in Figure 1. Thus, from now on, we will assume w.l.o.g. that every node i has an execution file of size $n_i = 0$.
- Each non-leaf node i in the tree, when executed, produces a file of size f_j for each $j \in \mathit{Children}(i)$. If i is a leaf-node, then $\mathit{Children}(i) = \emptyset$ and i produces a file of null size (we consider that terminal data produced by leaves are directly sent to the outside world).

During the processing of a task i, the memory must contain its input file, and all its output files (including the execution file of the additional child whenever needed). The amount of memory $\mathit{MemReq}(i)$ that is needed for this processing is thus:

$$\mathit{MemReq}(i) = \left(\sum_{j \in \mathit{Children}(i)} f_j \right) + f_i$$

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 amount of memory used to process the node i is:

$$\mathit{MemUsed}(\sigma, i) = \left(\sum_{j \in \mathit{Children}(i)} f_j \right) + f_i + \sum_{j \in S \setminus \{i\}} f_j$$

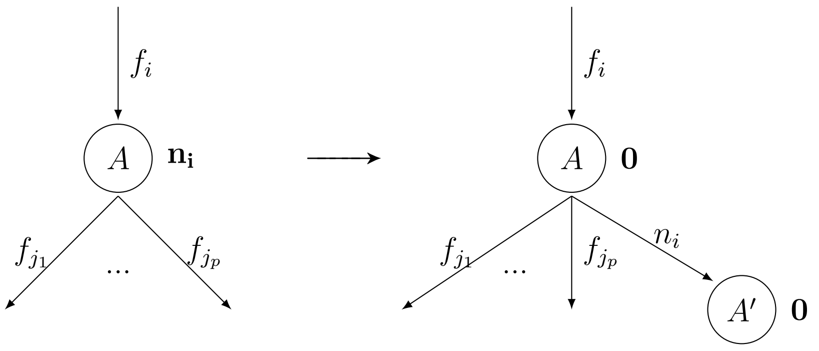

Figure 1: Modeling a node with an execution file of size $n_i \neq 0$

where S denotes the set of files stored in the memory when the scheduler decides to execute task i. Note that S must contain the input file of task i. After the processing of node i, we have:

$$S \leftarrow (S \backslash \{i\}) \cup \mathit{Children}(i)$$

Initially, S contains the input file of the root.

3.2. Bi-colored trees

When two memories are considered, each task (or node) i in the tree is now characterized by its *color*, which represents the specific memory where the task has to be executed, in addition to the size f_i of its input file (as before). 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:

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

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:

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

Initially, S_{blue} contains the input file of the root and $S_{\mathit{red}} = \emptyset$ 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:

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

  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:

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

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.

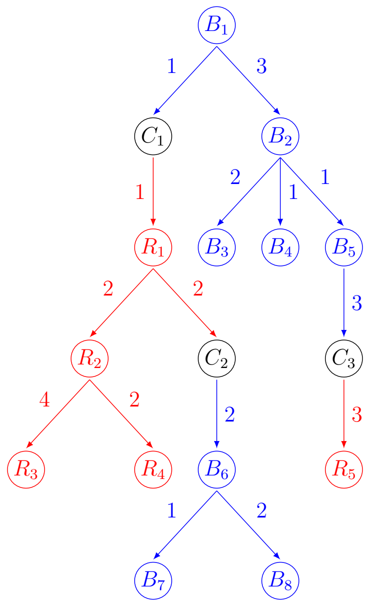

Figure 2: Bi-colored tree $\mathcal{T}$ for the example.

3.3. Example

Let consider the bi-colored tree $\mathcal{T}$ depicted in Figure 2. Any traversal of $\mathcal{T}$ has to start with the execution of the root B_1. After it has been processed, 4 units of the *blue* memory are occupied and the *red* memory is empty. We now have two choices:

- Either we process the right *blue* children B_2 first. This would use $BlueMemUsed(\sigma, B_2) =$ 8 units of the *blue* memory since the file of size 1 created by the root would still reside in the *blue* memory. Then, the sum of the file sizes stored in the *blue* memory after B_2 has been processed would be equal to $\sum_{j \in S_{blue}} f_j = 5$, and the *red* memory would be empty.

- Or we can process the communication node C_1 to load the file of size 1 from the *blue* memory to the *red* one. After that, if the left *blue* children B_2 is now processed, its execution would use $BlueMemUsed(\sigma, B_2) = 7$ units of the *blue* memory instead of 8 in the previous case, but the *red* memory would contain a file of size 1 wich will matter for a further execution of a *red* node.

A complete traversal σ_{ex} of $\mathcal{T}$ is described in Table 1, with the ordered list of the execution of tall nodes in $\mathcal{T}$, the amount of both memories required for each task, and the evolution of both memory usages after each execution. The maximum memory usage for the traversal σ_{ex} described in Table 1 is 7 units for the *blue* memory and 8 units for the *red* memory. As one can see, the ordered list of the execution of the computation and communication nodes of $\mathcal{T}$ will be the result of a trade-off between the usage of each memory. In fact, the memory-aware traversal problem for bi-colored rooted trees can naturally be cast into a two-criteria optimization problem.

$\sigma_{ex}(i)$	Fired Node	*BlueMemReq*(i)	*RedMemReq*(i)	*BlueMemUsed*(σ, i)	*RedMemUsed*(σ, i)	$\sum_{j \in S_{blue}} f_j$	$\sum_{j \in S_{red}} f_j$
1	B_1	4	0	4	0	$0 \rightarrow 4$	0
2	C_1	1	1	4	1	$4 \rightarrow 3$	$0 \rightarrow 1$
3	B_2	7	0	**7**	1	$3 \rightarrow 4$	1
4	B_3	2	0	4	1	$4 \rightarrow 2$	1
5	B_4	1	0	2	1	$2 \rightarrow 1$	1
6	B_5	4	0	4	1	$1 \rightarrow 3$	1
7	C_3	3	3	3	4	$3 \rightarrow 0$	$1 \rightarrow 4$
8	R_5	0	3	0	4	0	$4 \rightarrow 1$
9	R_1	0	5	0	5	0	$1 \rightarrow 4$
10	C_2	2	2	2	4	$0 \rightarrow 2$	$4 \rightarrow 2$
11	B_6	5	0	5	2	$2 \rightarrow 3$	2
12	B_7	1	0	3	2	$3 \rightarrow 2$	2
13	B_8	2	0	2	2	$2 \rightarrow 0$	2
14	R_2	0	8	0	**8**	0	$2 \rightarrow 6$
15	R_3	0	4	0	6	0	$6 \rightarrow 2$
16	R_4	0	2	0	2	0	$2 \rightarrow 0$

Table 1: Description of the traversal σ_{ex} in Section 3.3.

3.4. Objectives

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[1]. In fact, the literature deals with both variants. The seminal paper of Liu [17] 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.

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. Finally, we prove in Section 4.3 that it is impossible to approximate both minimum memories within arbitrary constant factors.

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 $M^{\sigma}_{\text{blue}}(\mathcal{T}) \leq M_{blue}$ and $M^{\sigma}_{\text{red}}(\mathcal{T}) \leq M_{red}$?

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

Proof. The problem clearly belongs to NP, and the certificate is the ordered list of tasks (of both colors and including uncolored communication nodes) executed by the schedule; it is easy to maintain the amount of each memory required by the schedule, and to check that neither M_{red} nor M_{blue} is exceeded.

To establish the completeness, we use a reduction from the 2-Partition problem [8]. Consider an instance $Inst_1$ of the 2-Partition problem, with n integers $\{a_1, a_2, ..., a_n \parallel \sum_{i=1}^{n} a_i = S\}$. Consider an instance $Inst_2$ of the TwoMemoryTraversal, consisting in the tree depicted on Figure 3. We set the bounds $M_{red} = 3S$ for the *red* memory and $M_{blue} = 2S$ for the *blue* memory. The construction of $Inst_2$ is polynomial in the size of $Inst_1$.

Assume first that $Inst_2$ has a solution. Any traversal must start with the root B_{root}. After it has been processed, $2S$ units of the *blue* memory are occupied, which means that it is full. Without loss of generality (by symmetry), assume that C is the next node to be executed. Then, we observe that if $C^{(2)}$ was the third executed node, we could never process R_{root} nor $R^{(2)}_{root}$ without violating the M_{red} bound for the *red* memory. Thus, the third executed node has to be R_{root}.

- We observe that the *red* tasks R_{big} and R_{free}, and each communication task C_i, all have to be processed before $R^{(2)}_{root}$, otherwise, since the execution of $R^{(2)}_{root}$ require $3S$

[1]This equivalence has been formally proven in [14] for single-memory platforms, and it is straightforward to extend the proof for two-memory systems.

units of memory, it would violate the M_{red} bound for the *red* memory. Besides, since we can not execute $R^{(2)}_{root}$ before R_{big}, if $C^{(2)}$ were processed before R_{big}, there would be at least S units of memory in the *red* memory and the execution of R_{big} (which requires $\frac{5}{2}S$ units of the *red* memory) would violate the M_{red} bound. Thus, the node R_{big} has to be processed before $C^{(2)}$.

- Besides, let i_0 be the index of the first processed task B_i in the traversal. Its execution requires $a_{i_0} + \frac{3}{2}S$ units of the *blue* memory, which implies that it can not be processed before $C^{(2)}$ without violating the M_{blue} bound for the *blue* memory. Thus, the node R_{big} has to be processed before B_{i_0}.

According to the previous arguments, the only tasks that can be processed right after R_{root} and before R_{big} are the communication tasks C_i. Let I be the set of the indices of the tasks C_i executed after R_{root} and before R_{big}.

$\to$ After the execution of R_{root}, there are $2S$ units occupied in the *red* memory and S units in the *blue* memory. Thus, to execute R_{big} without violating the M_{red} bound, the amount of *red* memory to free is at least $\frac{S}{2}$. This means that $\sum_{i \in I} a_i \geq \frac{S}{2}$.

$\to$ Besides, if $\sum_{i \in I} a_i > \frac{S}{2}$, the execution of B_{i_0} (which requires at least $\sum_{i \in I} a_i + \frac{3}{2}S$ units of the *blue* memory) will violate the M_{blue} bound.

Thus, $\sum_{i \in I} a_i = \frac{S}{2}$, which implies that $Inst_1$ has a solution.

Suppose now that $Inst_1$ has a solution I. According to the previous reasoning, the sequence of nodes B_{root}; C; R_{root}; $\forall i \in I$ C_i; R_{big} and R_{free} can be executed without violating the bounds on memories. After this sequence, there are $\frac{3}{2}S$ units occupied in the *blue* memory and the *red* one is empty. The node $C^{(2)}$ can be processed to load S units from the *blue* memory to the *red* one. Now, one of the *blue* node B_{i_0} with $i_0 \in I$ can be executed without violating the M_{blue} bound, followed by B'_{i_0}. Moreover, we can process every B_i and B'_i for all $i \in I$ to free the *blue* memory. Then, it is possible to execute every branch of C_i down to B'_i for all $i \notin I$. From this point on, we can process the sub-tree rooted at the node $R^{(2)}_{root}$ using the same pattern, which means that $Inst_2$ has a solution and concludes the proof. □

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, as depicted in Figure 4. We construct the single-colored tree $\mathcal{T}_{\text{red}}$ in a similar way. We let

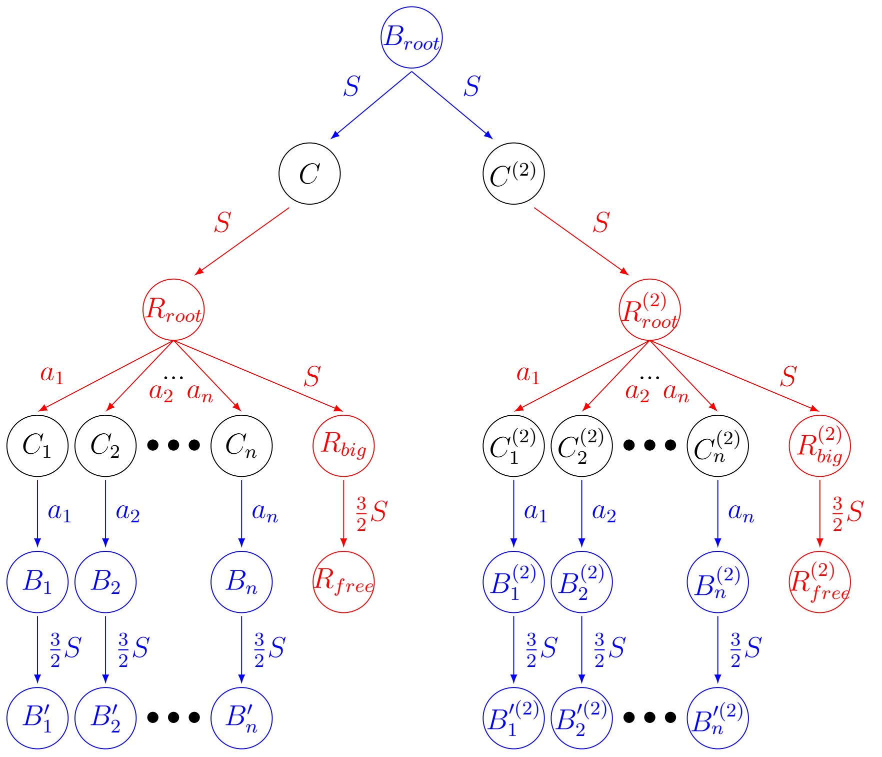

Figure 3: Tree used in the proof of Theorem 1

M^{∞}_{blue} denote the minimal amount of memory needed to process $\mathcal{T}_{\text{blue}}$ (and similarly, M^{∞}_{red} for $\mathcal{T}_{\text{red}}$).

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 $M^{\infty}_{\text{blue}} = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$.*

Proof. Given a bi-colored tree $\mathcal{T}$ with n nodes, consider $\mathcal{T}_{\text{blue}}$ and M^{∞}_{blue} as in Definition 2. We show here that $M^{\infty}_{blue} = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$. The proof for $M^{\infty}_{red} = M^{\text{opt}}_{\text{red}}(\mathcal{T})$ is similar.

First, $\mathcal{T}$ and $\mathcal{T}_{\text{blue}}$ have the same shape. The only differences between $\mathcal{T}$ and $\mathcal{T}_{\text{blue}}$ are some edge values and the color of some vertices and edges. Thus, to any feasible traversal σ of $\mathcal{T}$, we can associate the corresponding feasible traversal σ_{blue} of $\mathcal{T}_{\text{blue}}$, and reciprocally. For any node $i \in [\![1,n]\!]$ of $\mathcal{T}$, its corresponding node in $\mathcal{T}_{\text{blue}}$ will be referred at as $i_{blue} \in [\![1,n]\!]$, thus $\sigma(i) = \sigma_{blue}(i_{blue})$. Moreover we show that $BlueMemUsed(\sigma, i) = MemUsed(\sigma_{blue}, i_{blue})$ for each node $i \in [\![1,n]\!]$,:

- If $color(i) = blue$, node i is not changed in $\mathcal{T}_{\text{blue}}$ as described in Definition 2. Thus, $BlueMemReq(i) = MemReq(i_{blue})$ and the size of the files stored in the memory after i_{blue} has been processed is the same that the files stored in the *blue* memory after i has been processed.
- If $color(i) = red$, then $BlueMemReq(i) = 0$ and no file is stored in the *blue* memory after i has been processed. Besides, for the corresponding node i_{blue} in $\mathcal{T}_{\text{blue}}$, we have

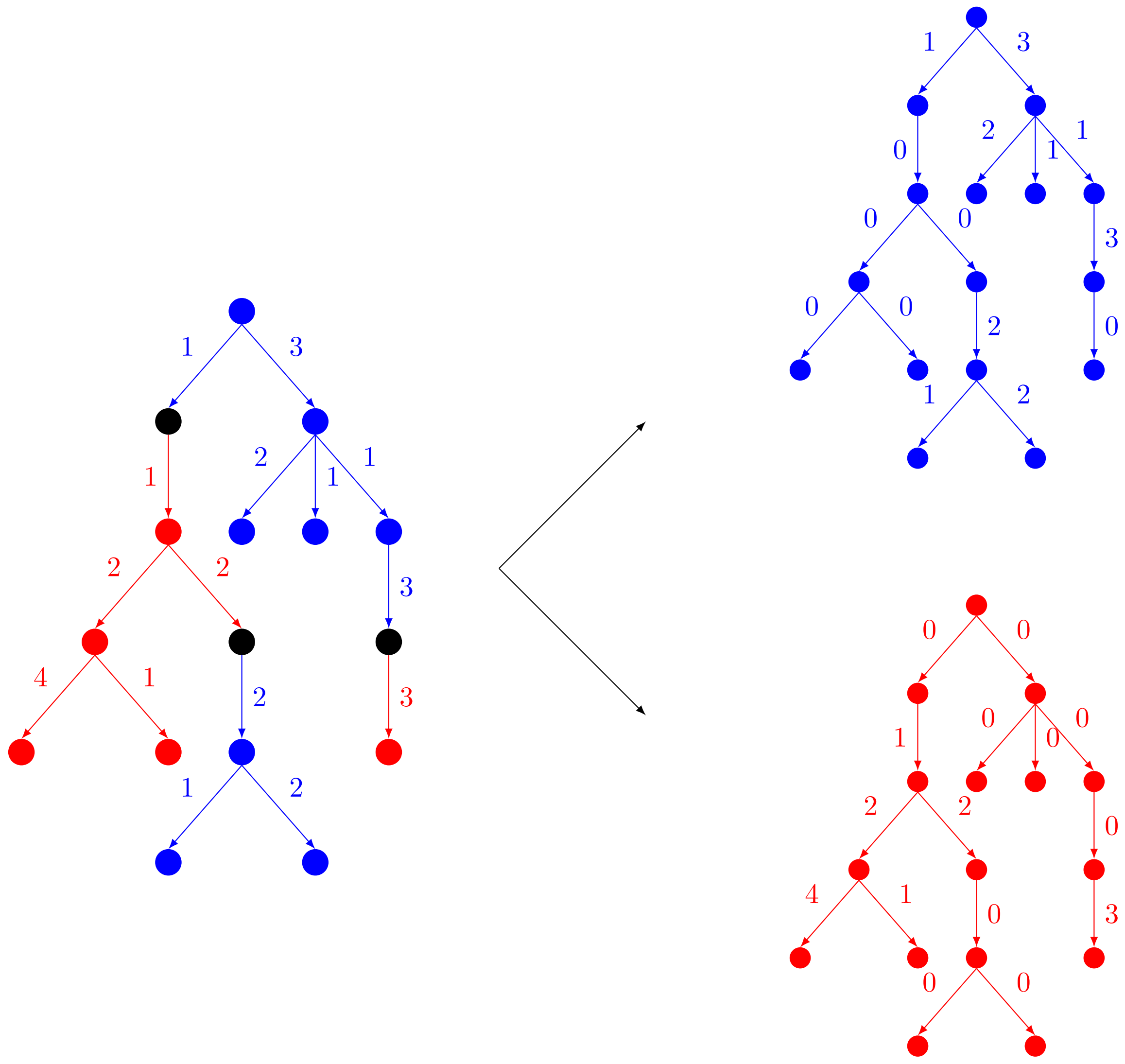

Figure 4: A bi-colored tree $\mathcal{T}$ and its corresponding single color trees $\mathcal{T}_{\text{blue}}$ and $\mathcal{T}_{\text{red}}$ in Definition 2.

$f_{i_{blue}} = 0$ and $f_j = 0$ for each $j \in \mathit{Children}(i_{blue})$,. Thus $\mathit{MemReq}(i_{blue}) = 0$ and no file is stored in the memory after i_{blue} has been processed.

- If i is uncolored (communication node), then $\mathit{BlueMemReq}(i) = f_i$. There are two sub-cases:

  - If i is a communication node from a *blue* node to a *red* node, its processing will store no file in the *blue* memory. According to the Definition 2, if j_{blue} denotes the child of i_{blue}, we have $f_{i_{blue}} = f_i$ and $f_{j_{blue}} = 0$. Thus, $\mathit{MemReq}(i_{blue}) = f_i$ and no file is stored in the memory after i_{blue} has been processed.
  - If i is a communication node from a *red* node to a *blue* node, its processing will store a file of size f_i in the *blue* memory. According to the Definition 2, if j_{blue} denotes the child of i_{blue}, we have $f_{i_{blue}} = 0$ and $f_{j_{blue}} = f_i$. Thus, $\mathit{MemReq}(i_{blue}) = f_i$ and a file of size f_i is stored in the memory after i_{blue} has been processed.

During the whole process $\mathit{BlueMemReq}(\sigma, i) = \mathit{MemReq}(\sigma_{blue}, i_{blue})$. Besides, the size of the files stored in the *blue* memory after i has been processed and the size of the files stored in the memory after i_{blue} has been processed are equal. Thus $\mathit{BlueMemUsed}(\sigma, i) = \mathit{MemUsed}(\sigma_{blue}, i_{blue})$ and $M^{\text{opt}}_{\text{blue}}(\mathcal{T}) = M^{\infty}_{blue}$. □

4.3. Joint minimization of both objectives

Since the traversal problem is NP-complete, it is natural to wonder whether there it is possible to get a schedule with guaranteed blue and red peak memories, compared to the optimal ones. In this section, we show that a trade-off must be enforced between these two objectives: indeed, if one wants a strong guarantee on one memory (blue or red), then the produced schedule may be arbitrarily bad for the other memory. More specifically, we prove that there does not exist schedules that can simultaneously approximate 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* in multi-objective optimization [7], 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, as depicted in Figure 5. 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 is helpful to prove the inapproximability 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:*

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

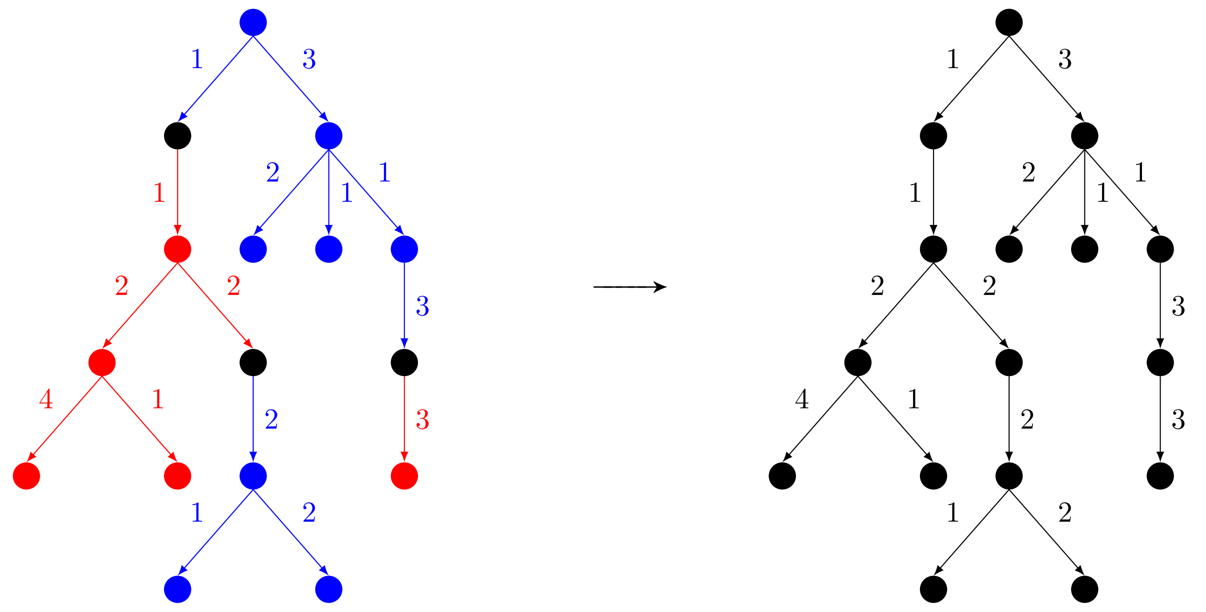

Figure 5: A bi-colored tree $\mathcal{T}$ and its corresponding uncolored tree $\mathcal{T}_{\text{unco}}$ in Definition 3.

Proof. Let $\mathcal{T}_{\text{unco}}$ be the uncolored tree corresponding to $\mathcal{T}$ as described in Definition 3. We observe that $\mathcal{T}$ and $\mathcal{T}_{\text{unco}}$ have the same tasks, hence to any feasible traversal σ of $\mathcal{T}$, we can associate the corresponding feasible traversal σ_u of $\mathcal{T}_{\text{unco}}$, and reciprocally. For any node $i \in [\![1, n]\!]$ of $\mathcal{T}$, its corresponding node in $\mathcal{T}_{\text{unco}}$ will be referred to as $i_u \in [\![1, n]\!]$, thus $\sigma(i) = \sigma_u(i_u)$.

We will show that

$$\forall i \in [\![1, n]\!], \quad \mathit{BlueMemUsed}(\sigma, i) + \mathit{RedMemUsed}(\sigma, i) = \mathit{MemUsed}(\sigma_u, i_u)$$

We proceed along the following case analysis:

- If $\mathit{color}(i) = \mathit{blue}$, then $\mathit{BlueMemReq}(i) = \mathit{MemReq}(i_u)$ and $\mathit{RedMemReq}(i) = 0$. Besides, no file is stored in the *red* memory after i has been processed; also, the size of the files stored in the *blue* memory after i has been processed is the same as that of the files stored in the memory after i_u has been processed.

- If $\mathit{color}(i) = \mathit{red}$, then $\mathit{RedMemReq}(i) = \mathit{MemReq}(i_u)$ and $\mathit{BlueMemReq}(i) = 0$. Besides, no file is stored in the *blue* memory after i has been processed; also, the size of the files stored in the *red* memory after i has been processed is the same as that the files stored in the memory after i_u has been processed.

- If i is uncolored (communication node), then $\mathit{BlueMemReq}(i) + \mathit{RedMemReq}(i) = 2 \times f_i = \mathit{MemReq}(i_u)$. Besides, a file of size f_i will be stored in one of the two memories after i has been processed, and a file of size f_i will be stored in the memory after i_u has been processed.

During the whole traversal, we thus have $\mathit{BlueMemReq}(\sigma, i)+\mathit{RedMemReq}(\sigma, i) = \mathit{MemReq}(\sigma_u, i_u)$. The sum of the size of the files stored in the *blue* memory and of the size of the files stored in the *red* memory after i has been processed is always equal to the size of the files stored in the memory after i_u has been processed. Thus $\mathit{BlueMemUsed}(\sigma, i) + \mathit{RedMemUsed}(\sigma, i) = \mathit{MemUsed}(\sigma_u, i_u)$. This means that:

$$\begin{aligned}
M^{\text{opt}}_{\text{unco}}(\mathcal{T}_{\text{unco}}) &\le M^{\sigma_u}_{\text{unco}}(\mathcal{T}_{\text{unco}}) \\
&= \max_i \ \mathit{MemUsed}(\sigma_u, i) \\
&= \max_i \{\mathit{BlueMemUsed}(\sigma, i) + \mathit{RedMemUsed}(\sigma, i)\} \\
&\le \max_i \{\mathit{BlueMemUsed}(\sigma, i)\} + \max_i \{\mathit{RedMemUsed}(\sigma, i)\} \\
&= M^{\sigma}_{\text{red}}(\mathcal{T}) + M^{\sigma}_{\text{blue}}(\mathcal{T})
\end{aligned}$$

which concludes the proof. □

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.*

Proof. To establish this result, we proceed by contradiction. We therefore assume that there is an integer α, an integer 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 $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ and using a *red* peak memory that is not greater than β times the optimal *red* peak memory $M^{\text{opt}}_{\text{red}}(\mathcal{T})$. To derive the contradiction, we use the family of tree $(\mathcal{T}_n)_{n\in\mathbb{N}}$ depicted on Figure 6. $\mathcal{T}_n$ is defined recursively using $\mathcal{T}_{n-1}$. To help the reader to visualize $\mathcal{T}_n$, Figure 7 represents $\mathcal{T}_2$.

- $\mathbf{\forall n \ge 2, M^{\text{opt}}_{\text{blue}}(\mathcal{T}_n) = 3}$
  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 $\mathcal{T}^{(left)}_{n-1}$ is processed recursively. Nodes R_n and C'_n follow. And finally $\mathcal{T}^{(right)}_{n-1}$ is processed recursively.

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

- $\mathbf{\forall n \ge 1, M^{\text{opt}}_{\text{red}}(\mathcal{T}_n) = 2}$
  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 $\mathcal{T}^{(left)}_{k-1}$ is processed recursively. Nodes C_k, R_k and C'_k follow. And finally $\mathcal{T}^{(right)}_{k-1}$ is processed recursively.

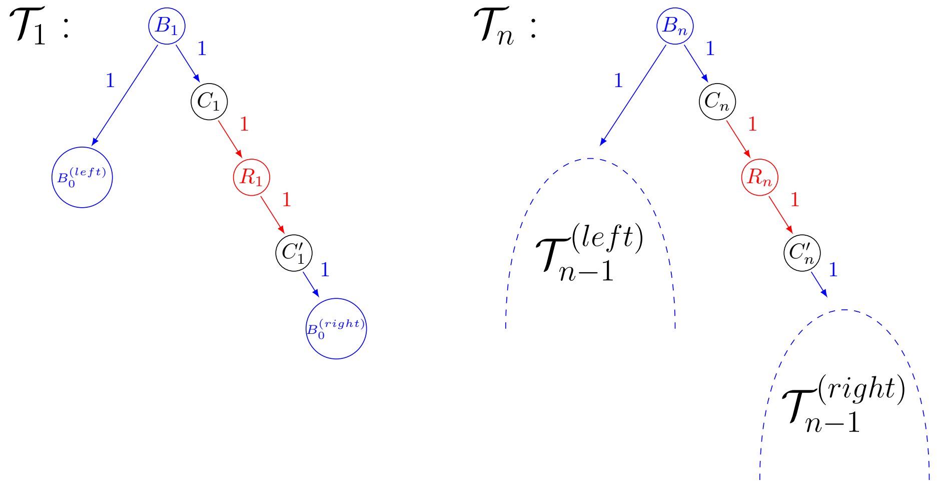

Figure 6: Recursive definition of $\mathcal{T}_n$ in the proof of Theorem 3

At each step of this process, the traversal σ_{red} does not use more than 2 units of *red* memory. Since $RedMemReq(R_n) = 2$, 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. $\mathcal{T}_2^{\text{unco}}$ is depicted in Figure 7. 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 [18], post-order traversals are optimal for peak memory minimization of uncolored trees with unit costs. Besides, all post-order 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. □

5. Depth-first traversals

In this section, we study depth-first traversals, which are the equivalent of post-order traversals for in-trees. In the context of single-memory trees, depth-first traversals are known to be sub-optimal [18]: worse, their memory usage can be arbitrarily high as compared to that of the optimal solution [14]. Clearly, these negative results remain true in a two-memory framework (simply assume that one memory is infinite). Still, depth-first

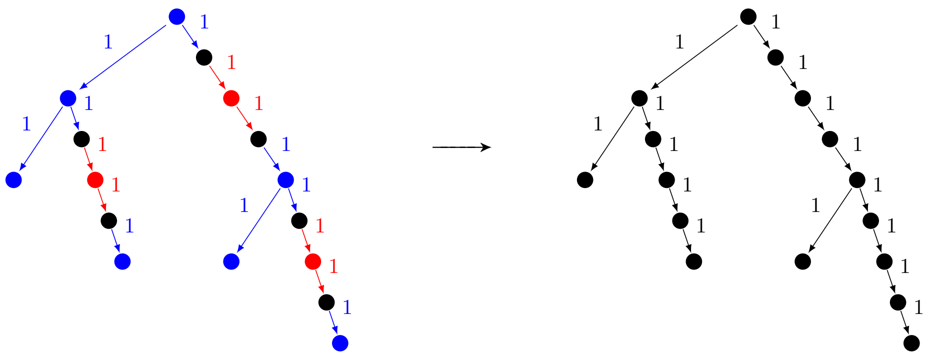

Figure 7: $\mathcal{T}_2$ and $\mathcal{T}_2^{\text{unco}}$ in the proof of Theorem 3.

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 like `MUMPS` [2, 3].

We show how to compute the optimal depth-first traversal in Section 5.1. It turns out that this traversal is optimal for both memory usages (among all depth-first traversals). However, depth-first traversals give no freedom on scheduling communication nodes. If we allow a communication node to be processed not immediately before its sub-tree, the ordering of the processing of the sub-trees and of the communication nodes will create a trade-off between both memory usages and will allow to decrease them. This leads us to define sloppy depth-first traversals, which we study in Section 5.2.

5.1. Strict 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 [18]: worse, their memory usage can be arbitrarily high as compared to that of the optimal solution [14]. 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.

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 $(\sigma, M^{blue}, M^{red})$;
```

Proof. Finding the best depth-first traversal of $\mathcal{T}$ amounts to find the best ordering to process every sub-tree. We prove that the order of the recursive processes at each step in Algorithm 1 is the best for both memories.

- At a given step, if the root of the sub-tree is a communication node, we have no choice, and we recursively process the sub-tree rooted at its unique child.
- At a given step, if the root r of the sub-tree is *blue*, then, the amount of *red* memory used to process this sub-tree will not depend on the order of the recursive processes to complete the sub-tree. Indeed, for each $i \in \mathit{Children}(r)$, after the recursive process of T_i, $S_{blue} \leftarrow S_{blue} \backslash \{i\}$ and S_{red} is unchanged. Then, independently of the order of the recursive processes of every T_i, the amount of *red* memory required to process $\mathcal{T}$ with a depth-first traversal will be $\mathit{RedMemReq}(\mathcal{T}) = \max_{i \in \mathit{Children}(root)} \mathit{RedMemReq}(T_i)$. Thus, at this step, we can only optimize the amount of *blue* memory. To do so, we use the optimal post-order traversal for uncolored trees provided by Liu [17]. This post-order traversal leads the best depth-first traversal for the *blue* memory at this step, and, thus, to the best depth-first traversal for both memories.
- At a given step, if the root of the sub-tree is *red*, the proof is similar.

□

5.2. Sloppy depth-first traversals

As explained in the previous section, the order of the sub-trees processed in a strict depth-first traversal does not influence the maximum usage of *red* memory for a tree rooted at a *blue* node, and vice versa. Thus, in a strict depth-first traversal, both memory usages are independent. This comes from the fact that strict depth-first traversals give no freedom on communications. If we allow a communication node to be processed not immediately before its sub-tree, the ordering of the processing of the sub-trees and of the communication nodes will create a trade-off between both memory usages. This leads us to define sloppy depth-first traversals.

Definition 5. A sloppy depth-first traversal is a feasible traversal similar to a depth-first traversal except that, after processing a communication node i, the whole sub-tree rooted at i is not necessarily processed immediately. We define $\mathit{SloppyChildren}(i)$ as being the set of thc *red* and *blue* children of i, together with the children of the uncolored children (these represent the set of the computational children of i). Formally, a feasible traversal σ of the tree $\mathcal{T}$ with n nodes is a sloppy depth-first traversal if and only if for each node $r \in \mathcal{T}$, and for any two nodes $i \in \mathit{Children}(r)$ and $j \in \mathit{SloppyChildren}(r) \cup \mathit{Children}(r)$, 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.

Definition 6 (TwoMemorySloppyDepthFirstTraversal)**.** Given a tree $\mathcal{T}$ with n nodes, and two fixed amount of memory M_{red} and M_{blue}, is there a sloppy depth-first traversal of the tree that need an amount of *red* memory inferior to M_{red} and an amount of *blue* memory inferior to M_{blue}?

Theorem 5. *The* TwoMemorySloppyDepthFirstTraversal *problem is NP-complete.*

Proof. The problem clearly belongs to NP, and the certificate is the ordered list of tasks (of both colors, and including communication nodes) executed by the schedule.

To establish the completeness, we use a reduction to the 2-Partition problem [8]. Consider an instance $Inst_1$ of the 2-Partition problem, with n integers $\{a_1,\ a_2,\ ...,\ a_n \parallel \sum_i a_i = S\}$. Consider an instance $Inst_2$ of the decision problem, consisting in the tree depicted on Figure 8. We set $M_{red} = 2S$ for the *red* tasks and $M_{blue} = 2S$ for the *blue* tasks. The construction of $Inst_2$ is polynomial in the size of $Inst_1$.

Assume first that $Inst_2$ has a solution. Any sloppy depth-first traversal must start with the root B_{root}. After it has been processed, $2S$ units of the *blue* memory are occupied, which means that this memory is full. Let i_0 be the index of the first *red* task R_i to be executed. We observe that B_{big} and B_{free} have to be processed before R_{i_0}, otherwise the process of C'_{i_0} (which occurs right after the process of R_{i_0} in a sloppy depth-first traversal) would violate the M_{blue} bound on the *blue* memory. Thus, the only tasks that can be processed right after B_{root} and before B_{big} are the communication tasks C_i. Let I be the set of the indices of the tasks C_i executed before B_{big}.

- If $\sum_{i \in I} a_i < \frac{S}{2}$, when the scheduler decides to execute B_{big}, the *blue* memory would be filled with $\sum_{i \notin I} a_i + S$ units. Thus the process of B_{big} will use $BlueMemUsed(B_{big}) = \sum_{i \notin I} a_i + S + \frac{3}{2}S > 2S$ units of *blue* memory, which violates the M_{blue} bound.
- If $\sum_{i \in I} a_i > \frac{S}{2}$, when the scheduler decides to execute R_{i_0}, the *red* memory would be filled with at least $\sum_{i \in I} a_i > \frac{S}{2}$ units. Thus the process of R_{i_0} will use at least $RedMemUsed(R_{i_0}) \geq \sum_{i \in I} a_i + \frac{3}{2}S > 2S$ units of *red* memory, which violates the M_{red} bound.

Thus, $\sum_{i \in I} a_i = \frac{S}{2}$, which implies that $Inst_1$ has a solution.

Suppose now that $Inst_1$ has a solution I. According to the previous reasoning, the sequence of nodes B_{root}; $\forall i \in I, C_i$; B_{big} and B_{free} can be executed without violating the bounds on memories. After this sequence, there are $\frac{S}{2}$ units occupied in the *blue* memory and in the *red* one. Now, one of the *red* node R_{i_0} with $i_0 \in I$ can be executed without violating the M_{red} bound, followed by C'_{i_0} and B_{i_0}. Moreover, we can process every R_i, C'_i and B_i $\forall i \in I$. Then, one is able to execute every branch of C_i down to B_i for all $i \notin I$, which means that $Inst_2$ has a sloppy depth-first solution and concludes our proof. □

6. Heuristics

In addition to depth-first traversals, in this section we present three traversal heuristics which aim at minimizing both the blue and red memories. All three heuristics are based on the seminal work by Liu [18] who considers a single memory. We proposed different adaptation for two memories. We start with the simplest heuristic and then proceed to more elaborate ones.

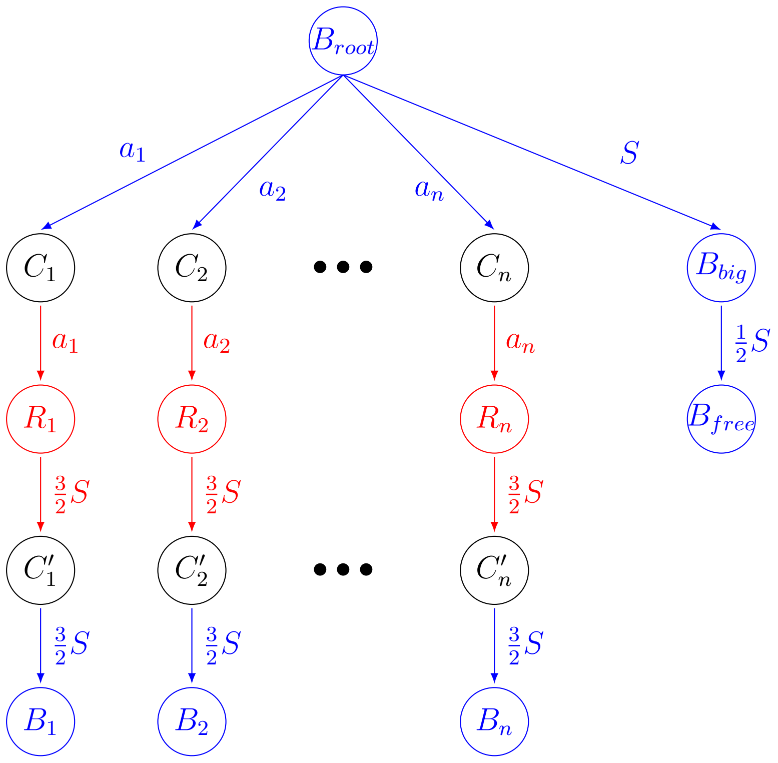

Figure 8: Tree corresponding to $Inst_2$ in the proof of Theorem 5

Working with the uncolored tree: LiuUncolored. We have shown that the problem TwoMemoryTraversal of finding a tree-traversal that minimizes both memory is NP-complete. However, when a single memory is considered, the problem becomes polynomial. It is thus natural to adapt the optimal algorithm for the single memory problem proposed by Liu [18], to bi-colored trees. The simplest adaptation amounts to considering the tree as uncolored, that is, as if all tasks were processed on the same computing unit with a single memory. On this uncolored tree, illustrated on Figure 4, we apply Liu's optimal algorithm. This heuristic computes an optimal traversal for the sum of the blue and the red memories. The intuition is that minimizing the sum of both memories will lead to a good memory usage for each of them. This heuristic is referred to as LiuUncolored in the following.

Refining the sum with weights: LiuWeightedSum. One problem with the previous heuristic is that both memories may not be equivalent. For example, it may well be the case that (input and output) files used by red tasks are much larger that those used by blue tasks. In such a case, minimizing the sum may lead to a much larger amount of blue memory that would be needed, for example, in an optimal traversal for the blue memory. This behavior is not desirable, and we can slightly change the heuristic to (try to) avoid this. We first compute the optimal amount of blue (respectively red) memory that is needed to traverse the tree, as described in Section 4.2, and we denote this amount by M^{∞}_{blue} (resp. M^{∞}_{red}).

Then, we normalize the memory weight of edges as follows: the memory weight f_i of the input edge of node i becomes f_i/M^{∞}_{blue} if this edge is blue, and f_i/M^{∞}_{red} if it is red. Then, the corresponding uncolored tree is considered and Liu's optimal algorithm is applied, as in the previous heuristic. This heuristic is called LiuWeightedSum in the following.

LiuWeightedMax. In the previous heuristics, when applying Liu's algorithm to modified trees, we minimize the sum (or the weighted sum) of both memory amounts. However, to get closer to the Zenith point, we would like to minimize the maximum, or rather the weighted maximum of both memories. It is possible to modify Liu's algorithm for this new goal. Of course, the resulting algorithm is not optimal anymore (which is coherent with the NP-completeness of the TwoMemoryTraversal problem), but it can be used as a heuristic. Liu's algorithm is a recursive algorithm which, at each node r, combines optimal traversals for the subtrees rooted at the children of r into an optimal traversal for the whole tree rooted in r. The combination relies on the definition of "hill-valley" segments: segments are defined by splitting a subtree schedule at different local minima (the "valley"). These segments are then sorted by non-increasing "hill" minus "valley" values (hill being the local peak memory of the segment). Liu [18] proves that such a combination of optimal subtree schedules leads to a global optimal schedule. In this heuristic, we replace the memory criterion used to define of the schedule by the maximum weighted memory: $\max(\frac{BlueMemUsed(\sigma,i)}{M^{\text{opt}}_{\text{blue}}(\mathcal{T})}, \frac{RedMemUsed(\sigma,i)}{M^{\text{opt}}_{\text{red}}(\mathcal{T})})$; we keep the same algorithm for combining subtree schedules. Of course, the proof of optimality does not hold for this new metric. This heuristic is called LiuWeightedMax in the following.

7. Experiments

In this section, we experimentally compare the memory usage of the heuristics proposed in the previous sections for TwoMemoryTraversal. For each heuristic among BestDepthFirst, LiuUncolored, LiuWeightedSum and LiuWeightedMax, we compute the amount of blue and red memory needed by the traversal. These values are compared to the minimum amount of blue (respectively red) memory needed when the red (resp. blue) memory is unbounded, as described in Section 4.2.

All heuristics have been implemented in C. The optimal value traversal for a single memory is computed using Liu's algorithm [18] written as a recursive code. Source code for all the algorithms, heuristics and experiments is publicly available at `http://perso.ens-lyon.fr/julien.herrmann/`.

7.1. Data Sets

We use four different sets of trees, ranging from actual trees arising in sparse matrix computations to random trees. We first describe the data set of uncolored trees which serves as a basis for our realistic colored trees.

Real uncolored trees for Cholesky factorization. The UNCOLOREDREALTREES data set contains assembly trees for a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices/). The chosen matrices satisfy the following assertions: not binary, not corresponding to a graph, square, having a symmetric pattern, a number of rows between 20,000 and 2,000,000, a number of non-zeros per row at least equal to 2.5, and a total number of non-zeros at most equal to 5,000,000; and each chosen matrix has the largest number of non-zeros among the matrices in its group satisfying the previous assertions. At the time of testing, there were 76 matrices satisfying these properties. We first order the matrices using MeTiS [15] (through the MeshPart toolbox [10]) and `amd` (available in Matlab), and then build the corresponding elimination trees using the `symbfact` routine of Matlab. We also perform a relaxed node amalgamation on these elimination trees to create assembly trees. We have created a large set of instances by allowing 1, 2, 4, and 16 (if more than 1.6×105 nodes) relaxed amalgamations per node. At the end we compute memory weights and processing times to accurately simulate the matrix factorization: we compute the memory weight n_i of a node as $\eta^2 + 2\eta(\mu - 1)$, where η is the number of nodes amalgamated, and μ is the number of non-zeros in the column of the Cholesky factor of the matrix which is associated with the highest node (in the starting elimination tree); the processing cost w_i of a node is defined as $2/3\eta^3 + \eta^2(\mu - 1) + \eta(\mu - 1)^2$ (these terms corresponds to one Gaussian elimination, two multiplications of a triangular $\eta \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix, and one multiplication of a $(\mu - 1) \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix). Edge weights f_i are computed as $(\mu - 1)^2$.

The resulting 644 trees contains from $2,000$ to $1,000,000$ uncolored nodes. Their depth ranges from 12 to $70,000$, and their maximum degree ranges from 2 to $175,000$.

Real colored trees for Cholesky factorization. The REALTREES data set is obtained by coloring every tree in UNCOLOREDREALTREES in a meaningful way. Every tree node in UNCOLOREDREALTREES represents a step of a $(\eta+\mu-1)\times(\eta+\mu-1)$ matrix factorization, with a panel of size η. In practice, at each step of the factorization, we aim at processing the GEMM routine (which corresponds to the multiplication of the $(\mu - 1) \times \eta$ matrix with the $\eta \times (\mu - 1)$ matrix) on the GPU. Indeed, GEMMs can reach up to 99% of the GPU's theoretical peak performance. Thus, we split every node into two tasks: a *red* one corresponding to the GEMM routine, and a *blue* one corresponding to the rest of the factorization.

Real trees with random colors. The RANDOMCOLOREDREALTREES data set is obtained by randomly coloring every node of every tree in UNCOLOREDREALTREES with an equiprobable choice in the set $\{$*red*, *blue*$\}$. Then, communication nodes are added between nodes of different colors.

Real trees with random weights and colors. The RANDOMWEIGHTEDREALTREES data set is obtained by randomly coloring every node of every tree in UNCOLOREDREALTREES with an equiprobable choice in the set $\{$*red*, *blue*$\}$ and by randomly changing the nodes and edges weight. Every n_i is set to a random integer value in $[\![1, \frac{N}{500}]\!]$ where N is the size of the tree, and every f_i is set to a random integer value in $[\![1, N]\!]$. Then, communication nodes are added between nodes of different colors.

Random trees. The RandomTrees data set is a set of 500 trees with random structure, random weights and random colors. Each tree has been generated as follows: the tree size N is randomly chosen in $[\![1, 32767]\!]$. Then, for each node $i \in [\![1, N]\!]$, its parent is randomly chosen in $[\![1, i-1]\!]$. The values of its n_i and f_i are uniformly chosen in $[\![1, 3276]\!]$, and its color is randomly chosen between *red* and *blue*. Then, communication nodes are added between nodes of different colors.

7.2. Results

In this section, we evaluate the performance of the four heuristics introduced above in terms of memory requirement. For every tree $\mathcal{T}$ in the data sets, and for every traversal σ returned by the heuristics, we compute the maximum relative overhead of each memory compared to the optimal value:
$MaxRelativeOverhead(\sigma, \mathcal{T}) = max(\frac{M^{\sigma}_{\text{blue}}(\mathcal{T}) - M^{\text{opt}}_{\text{blue}}(\mathcal{T})}{M^{\text{opt}}_{\text{blue}}(\mathcal{T})}, \frac{M^{\sigma}_{\text{red}}(\mathcal{T}) - M^{\text{opt}}_{\text{red}}(\mathcal{T})}{M^{\text{opt}}_{\text{red}}(\mathcal{T})})$.
As explained in Section 3, the optimal for both memories (also called *Zenith*) is a theoretical bound that may be not reachable. Thus, for a tree $\mathcal{T}$, there does not necessarily exist a traversal σ such that $MaxRelativeOverhead(\sigma, \mathcal{T}) = 0$. Detailed statistics for the four heuristics are given in Table 2. We make the following observations:

- For the RealTrees data set, BestDepthFirst statistically gives the best results, with an average relative overhead equal to 6.3%; it reaches the *Zenith* for 55.6% of the trees. This comes from the particular structure of the assembly trees. Indeed, most nodes in these assembly trees have an input file smaller than the sum of their output files: $f_i \leq \sum_{j \in Children(i)} f_j$. This means that when we execute a node, it is more likely to be profitable to execute the whole subtree straightaway. This is why BestDepthFirst turns out to be the best heuristic for the RealTrees and the RandomColoredRealTrees data sets. Besides, LiuUncolored is very close to the BestDepthFirst performances on the RealTrees data set, with an average relative overhead equal to 6.6%. On the contrary, LiuWeightedMax appears to be not well-designed for the structure of the assembly trees in RealTrees, with an average relative overhead equal to 8.4%; it can require up to 2.16 times the optimal memory for some trees.

- The structure of the trees in the RandomColoredRealTrees data set is close to the trees in RealTrees, and the results are similar. BestDepthFirst statistically gives the best results with an average relative overhead equal to 3.8%, and LiuUncolored provides the second best results with relative overhead equal to 5.2%.

- For the RandomWeightedRealTrees data set, file sizes are randomized, and BestDepthFirst is no longer adapted to such trees; it provides an average relative overhead equal to 20.9%. Much worse, LiuUncolored can require up to 5.13

times the optimal memory for some trees in RandomWeightedRealTrees. On the contrary, LiuWeightedMax appears to be well-designed for the trees in RandomWeightedRealTrees, with an average relative overhead two times lower than that of BestDepthFirst.

- The results for the RandomTrees data confirm that LiuWeightedMax is the best of the four heuristics when dealing with trees with random structure. It gives the best results with an average relative overhead equal to 3.4%, and exhibits a relative overhead inferior to 10% for 92% of the random trees.

Figures 10, 11 and 12 provide complete results of the simulations. In each figure, a point represents one scenario (one heuristic executed on one tree of the data set). To better visualize the distribution, we also plot a "cross" for each heuristic: the center of this cross is the average result, while the branches represent the scope of each objective between the $10th$ and $90th$ percentile of the distribution.

For the RealTrees data set, as explained above, we colored in *red* the nodes corresponding to the GEMM routine, and in *blue* the others nodes. Thus, every *red* nodes appears to have a communication node as father, and an unique communication node as child. With this structure, all of our heuristics gives the optimal memory usage for the *red* memory. This specification fits well with practice, where one aims at not overloading the GPU memory. Figure 9 provides the detailed distribution of the *blue* memory usage for the heuristics.

These figures exhibit the same trends for average values as observed in Table 2. For the RandomColoredRealTrees data set in Figure 10, and for the RandomTrees data set in Figure 12, we see that many traversals returned by the heuristics are optimal for at least one of the two memories, whereas for the RandomWeightedRealTrees data set in Figure 11, many more of the returned traversals are non-optimal for either memory. We also observe that LiuUncolored can require around 5 times the optimal red memory in two scenarios. These results show that the performance of the heuristics are strongly related to the structure of the trees. While BestDepthFirst achieves nice results for the realistic assembly trees, LiuWeightedMax appears to be a better solution when dealing with more random structures.

8. 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. This depth-first traversal achieves nice results for realistic assembly trees. We have also proposed several heuristics, based upon extensions of Liu's optimal

Data set	Algorithm	Avg.	Max.	Std. Dev.	Frac. of Opt.	Frac. ≤ 10%
RealTrees	BestDepthFirst	6.3%	64.4%	8.0%	55.6%	73.7%
	LiuWeightedMax	8.4%	116.5%	9.9%	49.8%	68.3%
	LiuWeightedSum	7.5%	76.0%	9.1%	52.8%	70.6%
	LiuUncolored	6.6%	60.0%	8.3%	55.0%	73.8%
RandomColoredRealTrees	BestDepthFirst	3.8%	44.0%	5.4%	67.2%	83.9%
	LiuWeightedMax	6.0%	52.3%	7.2%	51.4%	75.5%
	LiuWeightedSum	5.9%	52.6%	7.3%	54.1%	75.8%
	LiuUncolored	5.2%	52.6%	6.9%	59.7%	78.0%
RandomWeightedRealTrees	BestDepthFirst	20.9%	90.3%	18.6%	28.3%	44.6%
	LiuWeightedMax	10.2%	88.2%	13.6%	39.8%	72.7%
	LiuWeightedSum	13.4%	107.5%	16.3%	37.7%	65.2%
	LiuUncolored	15.4%	413.1%	17.0%	26.5%	60.2%
RandomTrees	BestDepthFirst	4.5%	28.2%	4.3%	33.4%	83.4%
	LiuWeightedMax	3.4%	23.5%	3.2%	26.0%	92.0%
	LiuWeightedSum	4.4%	21.4%	3.7%	20.6%	86.0%
	LiuUncolored	6.8%	32.9%	4.8%	14.6%	72.6%

Table 2: Statistics on the maximum relative overhead for each memory required by the four heuristics (comparison with the Zenith). Frac. of Opt. (respectively Frac $\leq$ 10%) counts the fractions of cases when the heuristics achieve the Zenith (resp. has a degradation not larger than 10%).

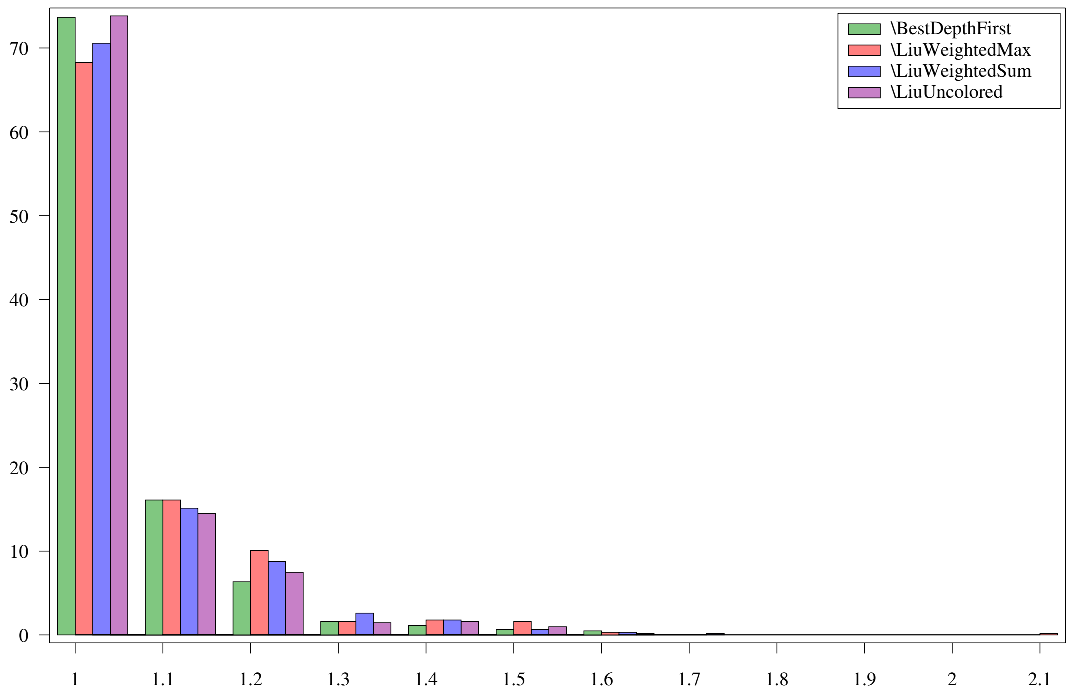

Figure 9: Percentage distribution of the *blue* memory usage for the RealTrees data set.

algorithm for the one-memory problem. These heuristics provide very good solutions when dealing with arbitrary tree graphs.

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

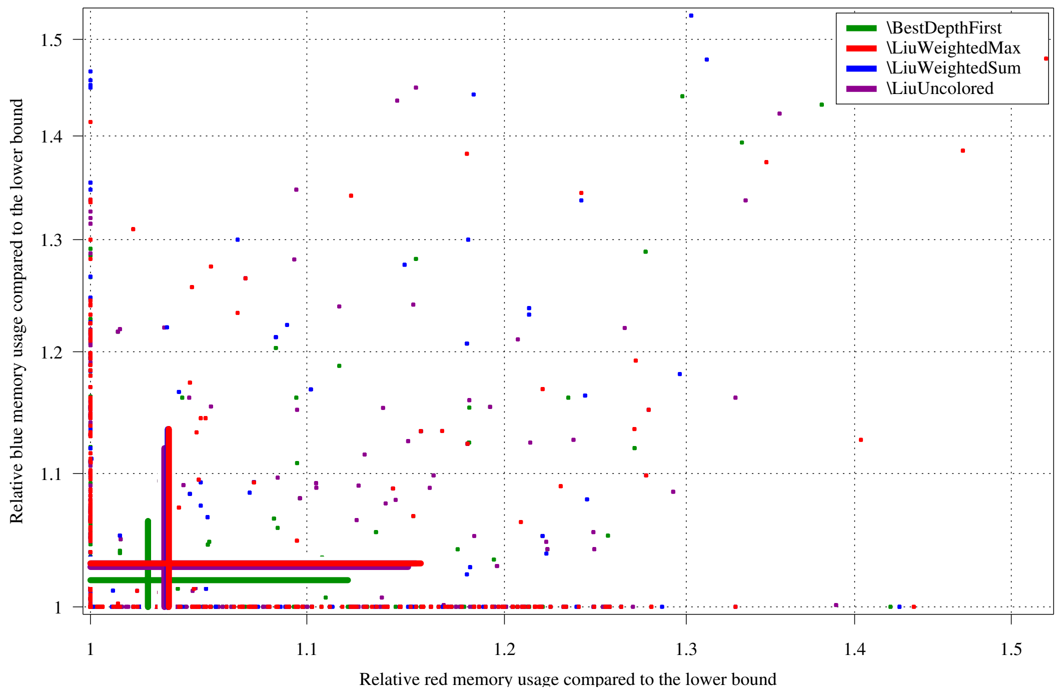

Figure 10: Distribution of each memory usage for the RandomColoredRealTrees data set.

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.

Acknowledgments.

The authors are with Université de Lyon, France. Y. Robert is with the Institut Universitaire de France. This work was supported in part by the ANR *RESCUE* and *SOLHAR* (ANR-13-MONU-0007) projects. A preliminary and much shorter version of this work appears in the proceedings of EuroPar [12]. We would like to thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper.

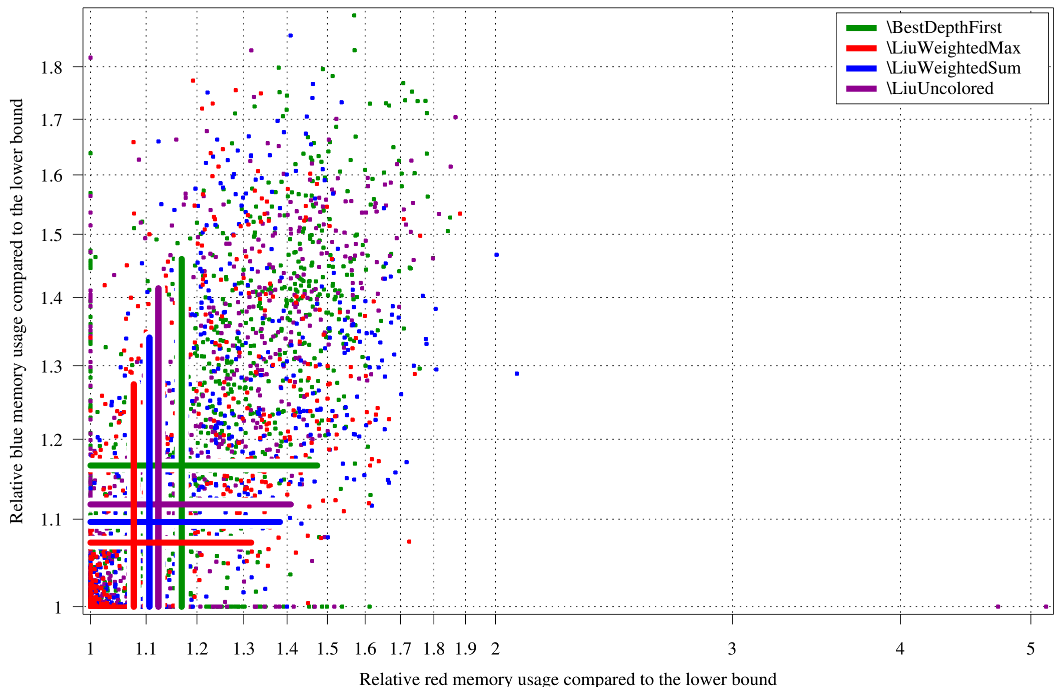

Figure 11: Distribution of each memory usage for the RandomWeightedRealTrees data set.

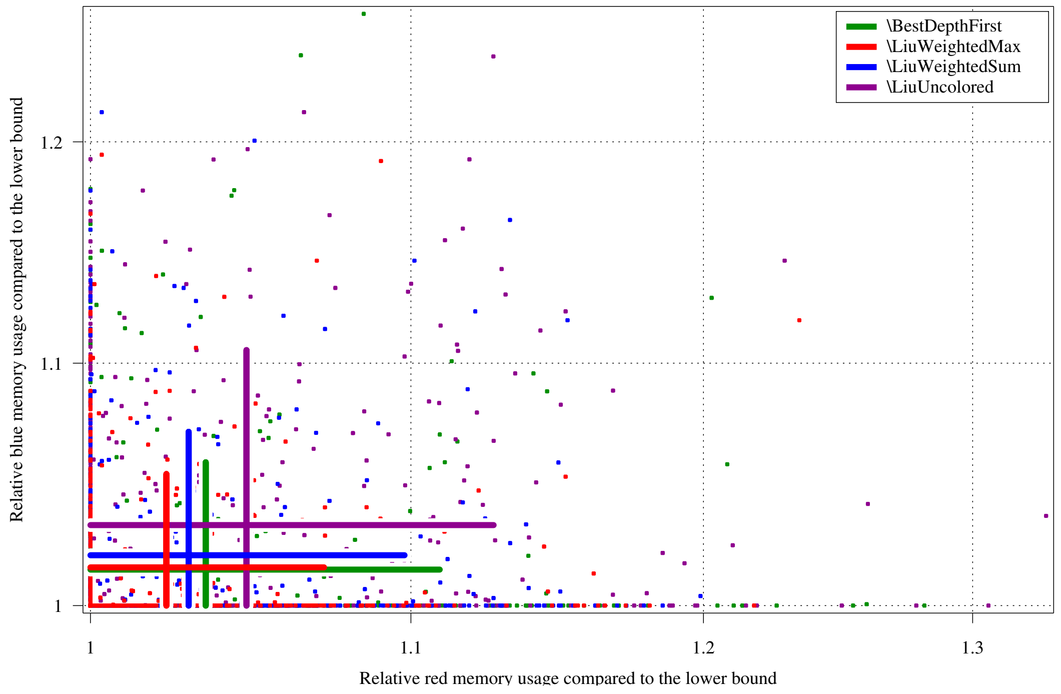

Figure 12: Distribution of each memory usage for the RandomTrees data set.

References

[1] E. Agullo, P. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet. Robust memory-aware mappings for parallel multifrontal factorizations, 2012. SIAM conf. on Parallel Processing for Scientific Computing (PP12).

[2] 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.

[3] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[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*, 23(2):187–198, 2011.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *IEEE Computing in Science and Engineering*, to appear. available online at `http://www.netlib.org/utk/people/JackDongarra/PAPERS/ieee_cise_submitted_2.pdf`.

[6] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[7] P.-F. Dutot, K. Rzadca, E. Saule, D. Trystram, et al. Multiobjective scheduling. In Y. Robert and F. Vivien, editors, *Introduction to Scheduling*. CRC Press, 2010.

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

[9] J. R. Gilbert, T. Lengauer, and R. E. Tarjan. The pebbling problem is complete in polynomial space. *SIAM J. Comput.*, 9(3), 1980.

[10] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[11] A. Guermouche and J.-Y. L'Excellent. Memory-based scheduling for a parallel multifrontal solver. In *IPDPS'04*, page 71, 2004.

[12] J. Herrmann, L. Marchal, and Y. Robert. Model and complexity results for tree traversals on hybrid platforms. In *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 647–658. Springer Berlin Heidelberg, 2013.

[13] 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.

[14] M. Jacquelin, L. Marchal, Y. Robert, and B. Ucar. On optimal tree traversals for sparse matrix factorization. *IPDPS'11*, 2011.

[15] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0.* U. of Minnesota, Dpt. of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[16] 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*, 37(2):63–75, 2011.

[17] 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.

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

[19] 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'13, 27th International Parallel and Distributed Processing Symposium.

[20] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[21] 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.

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

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

[24] S. Tomov, R. Nath, P. Du, and J. Dongarra. *MAGMA version User's guide.* available at `http://icl.eecs.utk.edu/magma/(2009)`.