HAL open science

Minimizing I/Os in Out-of-Core Task Tree Scheduling

Loris Marchal, Samuel Mccauley, Bertrand Simon, Frédéric Vivien

To cite this version:

Loris Marchal, Samuel Mccauley, Bertrand Simon, Frédéric Vivien. Minimizing I/Os in Out-of-Core Task Tree Scheduling. [Research Report] RR-9025, INRIA. 2017. hal-01462213

HAL Id: hal-01462213
https://inria.hal.science/hal-01462213

Submitted on 8 Feb 2017

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

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

Minimizing I/Os in Out-of-Core Task Tree Scheduling

Loris Marchal, Samuel McCauley, Bertrand Simon, Frédéric Vivien

RESEARCH REPORT
N° 9025
February 2017
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9025--FR+ENG

Minimizing I/Os in Out-of-Core Task Tree Scheduling

Loris Marchal, Samuel McCauley, Bertrand Simon, Frédéric Vivien

Project-Team ROMA

Research Report n° 9025 — February 2017 — 24 pages

Abstract: Scientific applications are usually described as directed acyclic graphs, where nodes represent tasks and edges represent dependencies between tasks. For some applications, such as the multifrontal method of sparse matrix factorization, this graph is a tree: each task produces a single output data, used by a single task (its parent in the tree).
We focus on the case when the data manipulated by tasks have a large size, which is especially the case in the multifrontal method. To process a task, both its inputs and its output must fit in the main memory. Moreover, output results of tasks have to be stored between their production and their use by the parent task. It may therefore happen, during an execution, that not all data fit together in memory. In particular, this is the case if the total available memory is smaller than the minimum memory required to process the whole tree. In such a case, some data have to be temporarily written to disk and read afterwards. These Input/Output (I/O) operations are very expensive; hence, the need to minimize them.
We revisit this open problem in this paper. Specifically, our goal is to minimize the total volume of I/O while processing a given task tree. We first formalize and generalize known results, then prove that existing solutions can be arbitrarily worse than optimal. Finally, we propose a novel heuristic algorithm, based on the optimal tree traversal for memory minimization. We demonstrate good performance of this new heuristic through simulations on both synthetic trees and realistic trees built from actual sparse matrices.

Key-words: Scheduling, Task tree, Tree traversal, I/O minimization, out-of-core

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Minimisation des entrées/sorties lors de l'exécution d'arbres de tâches

Résumé : Les applications de calcul scientifique sont souvent décrites comme des graphes de tâches dirigés et acycliques, où les nœuds représentent les tâches et les arêtes représentent les dépendances entre tâches. Pour certaines applications, comme les méthodes multifrontales de factorisation de matrices creuses, le graphe correspondant est un arbre: chaque tâche produit un unique fichier de données en sortie qui est utilisé par une unique tâche (son père dans l'arbre).

On s'intéresse ici dans le cas où les fichiers de données manipulés par les tâches sont de grande taille, ce qui est en particulier le cas dans les méthodes multifrontales. Pour traiter une tâche, ses fichiers d'entrée et de sortie doivent se trouver dans la mémoire principale. De plus, les fichiers de sortie doivent être stockés entre leur création et leur utilisation par la tâche père. Il peut donc arriver que, durant une exécution, la mémoire disponible soit inférieure à la mémoire minimum requise pour traiter l'arbre complet. Dans ce cas, certaines données doivent être temporairement transférées sur un disque pour être lues ultérieurement. Ces opérations d'Entrées/Sorties (E/S ou I/O en anglais) sont très coûteuses, d'où le besoin de les minimiser.

Nous revisitons dans cet article ce problème ouvert. Plus précisément, notre objectif est de minimiser le volume total d'Entrées/Sorties effectuées en traitant un arbre de tâche donné. Nous commençons par formaliser et généraliser des résultats connus, puis nous prouvons que les solutions existantes peuvent être arbitrairement loin de l'optimal. Finalement, nous proposons une nouvelle heuristique, basée sur le parcours d'arbre minimisant l'utilisation mémoire. Nous établissons les bonnes performances de cette heuristique à travers des simulations sur des arbres synthétiques ainsi que des arbres réalistes construits à partir de matrices creuses réelles.

Mots-clés : Ordonnancement, Arbre de tâches, Parcours d'arbres, Minimisation d'Entrées/Sorties

1 Introduction

Parallel workloads are often modeled as task graphs, where nodes represent tasks and edges represent the dependencies between tasks. There is an abundant literature on task graph scheduling when the objective is to minimize the total completion time, or makespan. However, with the increase of the size of the data to be processed, the memory footprint of the application can have a dramatic impact on the algorithm execution time, and thus needs to be optimized. When handling very large data, the available main memory may be too small to simultaneously handle all data needed by the computation. In this case, we have to resort to using disk as a secondary storage, which is sometimes known as *out-of-core* execution. The cost of the I/O operation to transfer data from and to the disk is known to be several orders of magnitude larger than the cost of accessing the main memory. Thus, in the case of out-of-core execution, it is a natural objective to minimize the total volume of I/O.

In the present paper, we consider the parallel scheduling of rooted in-trees. The vertices of the trees represent computational tasks, and the edges of the trees represent the dependencies between these tasks. Tasks are defined by their input and output data. Each task uses all the data produced by its children to output new data for its parent. In particular, a task must have enough available memory to fit the input from all its children.

The motivation for this work comes from numerical linear algebra, and especially the factorization of sparse matrices using direct multifrontal methods [1]. During the factorization, the computations are organized as a tree workflow called an elimination tree, and the huge size of the data involved makes it absolutely necessary to reduce the memory requirement of the factorization. Note that we consider here that no numerical pivoting is performed during the factorization, and thus that the structure of the tree, as well as the size of the data, are known before the computation really happens.

It is known that the problem of minimizing the peak memory M_{peak} of a tree traversal, that is, the minimum amount of memory needed to process a tree, is polynomial [2, 3]. However, it may well happen that the available amount of memory M is smaller than the peak memory M_{peak}. In this case, we have to decide which data, or part of data, have to be written to disk. In a previous study [3], we have focused on the case when the data cannot be partially written to disk, and we proved that this variant of the problem was NP-complete. However, it is usually possible to split data that reside in memory, and write only part of it to the disk if needed. This is for instance what is done using *paging*: all data are divided in same-size *pages*, which can be moved from main memory to secondary storage when needed. Since all modern computer systems implement paging, it is natural to consider it when minimizing the I/O volume.

Note that as in [3], the present study does not directly focus on parallel algorithms. However, parallel processing is the ultimate motivation for this work: complex scientific applications using large data such as multifrontal sparse matrix factorization always make use of parallel platforms. Most involved scheduling schemes combine data parallelism (a task uses multiple processors) and tree parallelism (several tasks are processed in parallel). We indeed have studied such a problem for peak memory minimization [4]. However, one cannot hope to achieve good results for the minimization of I/O volume in a parallel settings until the sequential problem is well understood, which is not yet the case. The present paper is therefore a step towards understanding the sequential version of this problem.

The main contributions of this work are:

- A formalization in a common framework of the results scattered in the literature;
- A proof of the dominance of post-order traversals when trees are homogeneous (all output data have the same size), knowing that an algorithm to compute the best post-order traversal has been proposed by E. Agullo [5].
- A proof that neither the best post-order traversal nor the memory-peak minimization algorithms are approximation algorithms for minimizing the I/O volume;

- A new heuristic that achieves good performance both on synthetic and actual trees as shown through simulations.

The rest of this paper is organized as follows. We give an overview of the related work in Section 2. Then in Section 3 we formalize our model and present elementary results. Existing solutions are studied in Section 4 before a new one is introduced in Section 5 and evaluated through simulations in Section 6. We finally conclude and present future directions in Section 7.

2 Related work

Memory and storage have always been a limited parameter for large computations, as outlined by the pioneering work of Sethi and Ullman [6] on register allocation for task trees. In the realm of sparse direct solvers, the problem of scheduling a tree so as to minimize peak memory has first been investigated by Liu [7] in the sequential case: he proposed an algorithm to find a peak-memory minimizing traversal of a task tree when the traversal is required to correspond to a postorder traversal of the tree. A *postorder* traversal requires that each subtree of a given node must be fully processed before the processing of another subtree can begin. A follow-up study [2] presents an optimal algorithm to solve the general problem, without the postorder constraint on the traversal. Postorder traversals are known to be arbitrarily worse than optimal traversals for memory minimization [3]. However, they are very natural and straightforward solutions to this problem, as they allow to fully process one subtree before starting a new one. Therefore, they are widely used in sparse matrix software like `MUMPS` [8, 9], and achieve good performance on actual elimination trees [3]. Note that peak memory minimization is still a crucial question for direct solvers, as highlighted by Agullo et al. [10], who study the effect of processor mapping on memory consumption for multifrontal methods.

As mentioned in the introduction, the problem of minimizing the I/O volume has been studied in [3] with the constraint that each data either stays in the memory or has to be written wholly to disk. We study here the case when we have the option to store part of the data, which is also the topic of E. Agullo's PhD. thesis [5]. In his thesis, Agullo exhibits the best postorder traversal for the minimization of the I/O volume, which we adapt to our model in Section 4.1. He also studies numerous variants of the model that are important for direct solvers, as well as other memory management issues—both for sequential and parallel processing. Based on these preliminaries, he finally presents an out-of-core version of the MUMPS solver.

Finally, out-of-core execution is a well-know approach for computing on large data, especially (but not only) in linear algebra [11, 12].

3 Problem modeling and basic results

3.1 Model and notation

As introduced above, we assume that we have an available memory (or primary storage) of limited size M, and a disk (or secondary storage) of unlimited size.

We consider a workflow of tasks whose precedence constraints are modeled by a tree of tasks $G = (V, E)$. Its nodes $v \in V$ represent tasks and its edges $e \in E$ represent dependencies. All dependencies are directed toward the root (denoted by *root*): a node can only be executed after the termination of all its children. The output data of a node i occupies a size w_i in the main memory. This data may be written totally or partially to the disk after task i produces it. In order for a node to be executed, the output data of all its children must be entirely stored in the main memory. An amount of memory m can be moved between the memory and the disk at a cost of m I/O operations, regardless of which data it corresponds to. We assume that all memory values (M, w_i) are given in an appropriate unit

(such as kilobytes) and are integers. We divide the main memory into *slots*, where each slot holds one such unit of memory.

At the beginning of the computation of a task i, the output data of i's children must be *in memory*, while at the end of its computation, its own output data must be in memory. The amount of memory needed in order to execute node i is thus

$$\bar{w}_i = \max\left(w_i, \sum_{(j,i)\in E} w_j\right).$$

We assume that M is at least as large as every $\bar{w}_i$, as otherwise the tree cannot be processed.

Our objective is to find a solution minimizing the total I/O volume. A solution needs to give the order in which nodes should be executed, and how much of each node should be written out during I/O operations. In particular, for a tree of n tasks, we define a solution to our problem as a permutation σ of $[1\dots n]$ and a function τ. We call such a solution a *traversal*. The permutation σ represents the *schedule* of the nodes, that is, $\sigma(i) = t$ means that task i is computed at step t, while the function τ represents the amount of I/O for each data: $\tau(i) = m$ means that m among w_i units of the output data of task i are written to disk (then we assume they are written as soon as task i completes). Note that we do not need to clarify which part of the data is written to disk, as our cost function only depends on the volume. Besides, we assume that when $\tau(i) \neq 0$, the *write* operation on the output data of task i is performed right after task i completes (and produces the data), and the *read* operation is performed just before the use of this data by task i's parent, as any other I/O scheme would use more memory at some time step for the same I/O volume. Finally, since there are as many *read* than *write* operations, we only count the *write* operations.

In order for a traversal to be valid, it must respect the following conditions:

- Tasks are processed in a topological order:

  $$\forall (i,j) \in E,\ \sigma(i) < \sigma(j);$$

  We say that a node i of parent j is considered *active* at step t under the schedule σ if $\sigma(i) < t < \sigma(j)$. This means that its output data is either partially in memory and/or partially written to disk at time t.
- The amount of data written to disk never exceeds the size of the data:

  $$\forall i \in V,\ 0 \le \tau(i) \le w_i;$$

- Enough memory remains available for the processing of each task (taking into account active nodes):

  $$\forall i \in G, \sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} (w_k - \tau(k)) \le M - \bar{w}_i.$$

The problem we are considering in this paper, called MinIO, is to find a valid traversal that minimizes the total amount of I/O, given by $\sum_{i\in G} \tau(i)$.

We formally define a *postorder* traversal as a traversal σ such that, for any node i and for any node k outside the subtree T_i rooted at i, we have either $\forall j \in T_i,\ \sigma(k) < \sigma(j)$ or $\forall j \in T_i,\ \sigma(j) < \sigma(k)$.

3.2 Towards a compact solution

Although a traversal is described by both the schedule σ and the I/O function τ, the following results show that one can be deduced from the other. The first result is adapted from [5, Property 2.1], which has the same result limited to postorder traversals (see Section 2). It states that given a schedule σ, it is easy to derive a I/O scheme τ which minimizes the I/O volume of the traversal (σ, τ).

Theorem 1. *We consider a tree G, a memory bound M, and a schedule σ. The I/O function τ following the* Furthest in the Future *policy achieves the best performance under σ.*

The I/O function τ following the *Furthest in the Future* (FiF) policy is defined as follows: during the execution of σ, whenever the memory exceeds the limit M, I/O operations are performed on the active nodes which will remain active the furthest in the future, i.e., whose execution come last in the schedule σ. This result is similar to Belady's rule which states the optimality of the offline MIN cache replacement [13, 14], that evicts from the cache the data which is used the latest.

Proof. Given a tree G, a memory bound M, a schedule σ and a I/O function τ that does not respect the FiF policy, it is straightforward to transform τ into another I/O function τ' following the rule. Consider the first step when an I/O is performed on a data i that is not the last one to be used among active data. Let j denote the last data used among active ones. We can safely increase $\tau'(j)$ and decrease $\tau'(i)$ until either $\tau'(j) = w_j$ or $\tau'(i) = 0$. As j is active longer than i is, the memory freed by τ' is available for a longer time than the one freed by τ, which keeps the traversal valid. Furthermore, by repeating this transformation, we produce an I/O function which respects the FiF policy. □

On the other hand, if we have an I/O function τ describing how much of each node is written to disk, we can compute a schedule σ such that (σ, τ) is a valid traversal (if such a schedule exists).

Theorem 2. *We consider a tree G, a memory bound M, and an I/O function τ for which there exists a valid schedule. Such a schedule can be computed in polynomial time.*

The proof of this result is delegated to Section 5 where we use a similar method to derive a heuristic: once we know where the I/O operations take place, we may transform the tree by *expanding some nodes* to make these I/O operations explicit within the tree structure. If a valid traversal using τ exists, the resulting tree may be completely scheduled without any additional I/O, and such a schedule can be computed using an optimal scheduling algorithm for memory minimization.

Both previous results allow us to describe solutions in a more compact format (as either a schedule or an I/O function). However, this does make the problem less combinatorial: there are $n!$ possible schedules and already 2^n τ functions if we restrict only to functions such that $\tau(i) = 0$ or w_i.

3.3 Related algorithms

As mentioned in Section 2, the problem of minimizing the peak memory, denoted MINMEM, is strongly related to our problem, and has been extensively studied. In this problem, the available memory is unbounded (which means no I/Os are required) and we look for a schedule that minimizes the peak memory, i.e., the maximal amount of memory used at any time during the execution. There are at least two important algorithms for this problem, which we use in the present paper:

- It is possible to compute a schedule minimizing the peak-memory in polynomial time, as proved by Liu [2]. We denote such an algorithm by OPTMINMEM.
- The best postorder traversal for peak-memory minimization can also be computed in polynomial time [7]. We will refer to this algorithm by POSTORDERMINMEM.

4 Existing solutions are not satisfactory

We now detail two existing solutions for the MINIO problem. The first one is the best postorder traversal proposed by Agullo [5]. The second consists in using the optimal traversal for MINMEM proposed by Liu [2] and then to apply Theorem 1 to obtain a valid traversal. After presenting these algorithms, we prove that none of them has a constant competitive factor compared to the optimal traversal.

4.1 Computing the best postorder traversal

For the sake of completeness, we present the adaption to our model of the algorithm computing the best postorder traversal for MINIO from [5]. Recall that in a postorder traversal, when a node is processed, its whole subtree must be processed before any other external node may be started. Given a node i and a postorder schedule σ, we first recursively define S_i as the storage requirement of the subtree T_i rooted at i. Let $Chil(i)$ be the children of i. Then:

$$S_i = \max\left(w_i,\ \max_{j\in Chil(i)}\left(S_j + \sum_{\substack{k\in Chil(i)\\ \sigma(k)<\sigma(j)}} w_k\right)\right).$$

This expression represents the maximum memory peak reached during the execution. If the peak is obtained at the end of the execution, it is then equal to w_i. Otherwise, it appears during the execution of the subtree of some child j. In this case, the peak is composed of the weights of the children already processed, plus the peak S_j of T_j.

We may now consider $A_i = \min(M, S_i)$, which represents the amount of main memory used for the out-of-core execution of the subtree T_i by σ. We recursively define V_i as the volume of I/Os performed by σ during the execution T_i when I/O operations are done using the FiF policy:

$$V_i = \max\left(0,\ \max_{j\in Chil(i)}\left(A_j + \sum_{k\in Chil(i)}^{\sigma(k)<\sigma(j)} w_k\right) - M\right) + \sum_{j\in Chil(i)} V_j.$$

The expression of V_i has a similar structure to the expression of S_i. No I/Os can be incurred when only the root i is in memory, hence w_i has no effect here. The second term accounts for the I/Os incurred on the children of i. Indeed, during the execution of node j, some parts of children of i must be written to disk if the memory peak exceeds M, and this quantity is at least $A_j + \sum_{k\in Chil(i)}^{\sigma(k)<\sigma(j)} w_k - M$. The last term accounts for the I/Os occurring inside the subtrees. Note that such I/Os can only happen if the memory peak of the subtree exceeds M.

It remains to determine which postorder traversal minimizes the quantity V_{root}. Note that the only term sensitive to the ordering of the children of i in the expression of V_i is $\max_{j\in Chil(i)}\left(A_j + \sum_{k\in Chil(i)}^{\sigma(k)<\sigma(j)} w_k\right)$. Theorem 3 states that sorting the children of i by decreasing values of $A_j - w_j$ achieves the minimum V_i.

Theorem 3 (Lemma 3.1 in [7])**.** *Given a set of values $(x_i, y_i)_{1\le i\le n}$, the minimum value of $\max_{1\le i\le n}\left(x_i + \sum_{j=1}^{i-1} y_j\right)$ is obtained by sorting the sequence (x_i, y_i) in decreasing order of $x_i - y_i$.*

Therefore, the postorder traversal that processes the children nodes by decreasing order of $A_i - w_i$ minimizes the I/O cost among all postorder traversals. This traversal is described in Algorithm 1, initially called with $r = root$, and will be referred to as POSTORDERMINIO. Note that in the algorithm $\oplus$ refers to the concatenation operation on lists.

4.2 POSTORDERMINIO is optimal on homogeneous trees

In this section we focus on homogeneous trees, that is on trees where all nodes have output data of size one. We will show that POSTORDERMINIO is optimal on these homogeneous trees, i.e., that it performs the minimum number of I/Os. This generalizes a result of Sethi and Ullman [6] for homogeneous binary trees.

Algorithm 1: POSTORDERMINIO (G, r)

Output: a tree G and a node r in G
Output: an ordered list ℓ_r of the nodes in the subtree rooted at r, corresponding to a postorder

```
1 foreach i child of r do
2   | ℓ_i ← PostOrderMinIO(G, i)
3   | Compute the A_i value using postorder ℓ_i
4 ℓ_r ← ∅
5 for i child of r by decreasing value of A_i − w_i do
6   | ℓ_r ← ℓ_r ⊕ ℓ_i
7 ℓ_r ← ℓ_r ⊕ {r}
8 return ℓ_r
```

Theorem 4. POSTORDERMINIO *is optimal for homogeneous trees.*

In order to prove this theorem, we need first to define some labels on the nodes of a tree. Let T be any homogeneous tree ($w_v = 1$ for all nodes v of T). In the following definitions, whenever v is a node of T with k children, $v_1, \dots, v_k$ will be its children.

Memory bound $l(v)$. For each node v of T, we recursively define a label $l(v)$ which represents the minimum amount of memory necessary to execute the subtree $T(v)$ rooted at v without performing any I/Os:

$$l(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \max_{1\le i\le k}(l(v_i) + i - 1) & \text{otherwise} \\ & \text{and ordering the children such that} \\ & l(v_i) \ge l(v_{i+1}) \text{ for } 1 \le i \le k-1 \end{cases}$$

We call POSTORDER one postorder schedule that executes the children of any node by non-increasing l-labels (ties being arbitrarily broken). Intuitively, under POSTORDER, while computing the i-th child, we have $i-1$ extra nodes in memory, each of size one, so we need $l(v_i) + (i-1)$ memory slots in total.

I/O indicator $c(v)$. If v_i is a child of v, intuitively, $c(v_i)$ represents the number of children of v written to disk by POSTORDER during the execution of $T(v_i)$. This number can be either 0 or 1. We set $c(v_1) = 0$ and

$$c(v_i) = \begin{cases} 0 & \text{if } l(v_i) + \sum_{1\le j<i}(1 - c(v_j)) \le M \\ 1 & \text{otherwise.} \end{cases}$$

We set $c(\mathit{root}) = 0$ where *root* is the root of T. To ease the writing of some proofs, we use the notation

$$m(v_i) = \sum_{1\le j<i}(1 - c(v_j)).$$

Thus $m(v_i)$ represents the number of children of v in memory when v_i is executed. Note that $m(v_1) = 0$ and $m(v_i) = (1 - c(v_1)) + \sum_{2\le j<i}(1 - c(v_j)) \ge (1 - c(v_1)) = 1$ for $2 \le i \le k$.

I/O volumes $w(v)$ and $W(T(v))$. $w(v)$ represents the total number of children of v stored by POSTORDER:

$$w(v) = \sum_{i=1}^{k} c(v_i) = \sum_{i=2}^{k} c(v_i).$$

Finally, for a given node v, we define $W(T(v))$ on the subtree rooted at v:

$$W(T(v)) = c(v) + \sum_{\mu \in T(v)} w(\mu).$$

$W(T(v))$ intuitively represents the total volume of communications performed during the execution of the tree $T(v)$ by POSTORDER.

We first state the correctness of the l-labels and the optimality of POSTORDER for the MINMEM problem.

Lemma 1. *With infinite memory,* POSTORDER *uses $l(n)$ slots to compute the subtree rooted at node n.*

Proof. This result follows from the definition of the labels $l(v)$. □

Lemma 2. *With infinite memory, any schedule uses at least $l(v)$ slots to compute the subtree rooted at v.*

Proof. We prove this result by induction on the size of $T(v)$. If v is a leaf, the result holds ($l(v) = 1$).

Otherwise, we assume the lemma to be true for the subtrees rooted at the children $v_1, \ldots, v_k$ of v. We consider the schedule returned by MINMEM. The memory peak inherent to the execution of a subtree $T(v_i)$ is equal to $l(v_i)$ by the induction hypothesis. Assume without loss of generality that the children of v are ordered such that MINMEM first computes a node of $T(v_1)$, then the next executed node not in $T(v_1)$ is in $T(v_2)$, then the next executed node neither in $T(v_1)$ nor in $T(v_2)$ is in $T(v_3)$, and so on. Then, the memory peak reached during the execution of $T(v_i)$ is at least $l(v_i) + (i-1)$ because, in addition to $T(v_i)$, at least $i-1$ subtrees have been partially executed: $T(v_1)$, ..., $T(v_{i-1})$. Finally, the total memory peak is at least equal to $\max_{1 \le i \le k}(l(v_i) + i - 1)$. By Theorem 3, this quantity is minimized when the nodes are ordered by non-increasing values of $l(v_i)$. Hence, the total memory peak is at least $l(v)$. □

We now state the performance of POSTORDER for the MINIO problem (I/Os are done following the FiF policy).

Lemma 3. POSTORDER *computes a given tree T using at most $W(T)$ I/Os.*

Proof. We prove this result by induction on the size of T. We introduce a new notation: for any node v of T we define $\mathscr{W}(v)$ as $\mathscr{W}(v) = W(T(v)) - c(v)$. In other words, $\mathscr{W}(v)$ represents the total volume of communications performed during the execution of the tree $T(v)$ *if* we had nothing to execute but $T(v)$ (in practice $T(v)$ may be a strict sub-tree of T and, therefore, the execution of $T(v)$ in the midst of the execution of T can induce more communications). Note that $\mathscr{W}(v) = W(T(v))$ if v is the root of T. We prove by induction on the size of $T(v)$ that POSTORDER performs at most $\mathscr{W}(v)$ I/Os during the execution of $T(v)$ *if* POSTORDER has nothing to execute but $T(v)$.

Let us assume that v is a leaf. Because we have assumed (in Section 3.1) that M was large enough for a single node to be processed without I/Os, $c(v) = 0$ and thus $W(T(v)) = 0 = \mathscr{W}(v) + c(v)$. On the other hand, POSTORDER performs 0 I/O during the execution of $T(v)$.

Now assume that v is not a leaf. By the induction hypothesis, for any $i \in [1;k]$, POSTORDER executes the tree $T(v_i)$ *alone* using at most $\mathscr{W}(v_i)$ I/Os. We prove that to process the tree $T(v_i)$, after the trees $T(v_1)$ through $T(v_{i-1})$ were processed, we need to perform at most $W(T(v_i)) = \mathscr{W}(v_i) + c(v_i)$ I/Os.

Let us consider the $(i+1)$-th child of v. If $c(v_{i+1}) = 0$, then $l(v_{i+1}) + \sum_{1 \le j < i+1}(1 - c(v_j)) \le M$. Then, according to Lemma 1, no I/Os are required to execute $T(v_{i+1})$ under POSTORDER even after the processing of $T(v_1)$ through $T(v_i)$. Indeed, before the start of the processing of $T(v_{i+1})$ the memory contains exactly $\sum_{1 \le j < i+1}(1 - c(v_j))$ nodes. Therefore $\mathscr{W}(v_{i+1}) = c(v_{i+1}) = W(T(v_{i+1})) = 0$ and we have the desired property.

We are now in the case $c(v_{i+1}) = 1$; thus $l(v_{i+1}) + \sum_{1\le j<i+1}(1 - c(v_j)) > M$. Recall that for $l \in [1;i]$, $l(v_l) \ge l(v_{i+1})$. Thus, if $l(v_{i+1}) \ge M$, then for $l \in [2;i]$, $l(v_l) \ge M$ and $c(v_l) = 1$ (because $m(v_l) \ge (1 - c(v_1)) = 1$). Therefore, after the completion of $T(v_i)$ there is only one node remaining in the memory: v_i. Then using a single I/O POSTORDER writes v_i to disk, the memory is empty, and $T(v_{i+1})$ can then be processed with $\mathscr{W}(v_{i+1})$ I/Os, giving a total of at most $\mathscr{W}(v_{i+1}) + c(v_{i+1}) = W(T(v_{i+1}))$ I/Os. The only remaining case is the case $l(v_{i+1}) < M$. The processing of $T(v_i)$ requires at least $l(v_{i+1})$ empty memory slots because $l(v_i) \ge l(v_{i+1})$. Hence, after the completion of $T(v_i)$ there are at least $l(v_{i+1}) - 1$ empty memory slots (the memory including the node v_i itself). Then using a single I/O POSTORDER writes v_i to disk and there are enough empty memory slots to process $T(v_{i+1})$ without any additional I/Os. Therefore we need to perform at most $W(T(v_{i+1})) = \mathscr{W}(v_{i+1}) + c(v_{i+1})$ I/Os. This concludes the proof. □

Lemma 5 relies on the following intermediate result.

Lemma 4. *Consider a node v of a tree T with a child, a, whose label $l(a)$ satisfies $l(a) > M$. Now, consider any tree T' identical to T, except that the subtree rooted at a has been replaced by any tree whose new label $l'(a)$ satisfies $l(a) \ge l'(a) \ge M$. Then $w'(v) = w(v)$.*

Proof. Let $v_1, \dots, v_k$ be the children of v, ordered so that $l(v_1) \ge \dots \ge l(v_k)$. Let j be the index of a: $a = v_j$. As the label of a in T', $l'(a)$, is not larger than $l(a)$, we can have $l'(a) < l'(v_{j+1})$. Therefore, we define another ordering of the children of v denoted by $v'_1, \dots, v'_k$ such that $l'(v'_1) \ge \dots \ge l'(v'_k)$. Let j' be the index of a in this ordering: $v'_{j'} = a = v_j$.

Note that $j' \ge j$. For $i \in [j+1; j']$, we have $v_i = v'_{i-1}$; at j, we have $v_j = v'_{j'}$; and for $i \notin [j; j']$, we have $v_i = v'_i$.

If $j' = 1$ then $j = 1$. This case means that a remains the node with the largest label. The labels of the other children of v remain unchanged. Because $c(v_1) = c'(v_1) = 0$ by definition, then $c'(v_i) = c(v_i)$ for any child v_i of v and, thus, $w(v)$ is equal to $w'(v)$.

Let us now consider the case $j' > 1$. From what precedes, $v'_{j'-1} = v_{j'}$. Then $l(v_{j'}) = l'(v'_{j'-1}) \ge l'(v'_{j'}) = l'(a) \ge M$. However, for any $i \in [1; j']$, $l'(v'_i) \ge l'(v'_{j'}) \ge M$ and $l(v_i) \ge l(v_{j'}) \ge M$. Therefore, for any $i \in [2; j']$, $l'(v'_i) + m'(v'_i) > M$ (because $m'(v'_i) \ge 1 - c'(v'_1) = 1$) and, thus, $c'(v'_i) = 1$. Similarly, for any $i \in [2; j']$, $l(v_i) + m(v_i) > M$ (because $m(v_i) \ge m(v_1) = 1$) and, thus, $c(v_i) = 1$. Therefore, for $i \in [1; j']$, $c(v_i) = c'(v'_i)$. Then, for $i \in [j'+1; k]$, $v_i = v'_i$, $m(v_i) = m'(v'_i)$, and $c(v_i) = c'(v'_i)$ by an obvious induction. Therefore, $w'(v) = \sum_{i=2}^{k} c'(v'_i) = \sum_{i=2}^{k} c(v_i) = w(v)$. □

The following lemma gives a lower bound on the I/Os performed by any schedule.

Lemma 5. *No schedule can compute a tree T performing strictly less than $W(T)$ I/Os.*

Proof. We proceed by induction on the number of nodes of T.

The base case consists of a tree T that can be scheduled without any I/O. For contradiction, assume that $W(T) > 0$. Then there exists a node v of T such that $w(v) > 0$ and a child v_i of v such that $c(v_i) = 1$. Then, by definition of $c(v_i)$ and of $l(v)$, $l(v) > M$. However, according to Lemma 2, "any schedule uses at least $l(v)$ slots to compute $T(v)$", so $T(v)$, and thus T, cannot be scheduled without I/Os. Hence, a contradiction; thus $W(T) = 0$.

Consider a tree T that cannot be scheduled without I/Os, and a schedule P on T that minimizes the total volume of I/Os.

First, by Lemma 1, there exists a node v such that $l(v) > M$. Otherwise, POSTORDER would be able to schedule T without I/Os, which would violate our assumption on T. Then, the label of the root r of T also satisfies $l(r) > M$.

Let s be the first node to be stored under P. Then, the subtree $T(s)$ has been scheduled without I/Os so, by Lemma 2, we have $l(s) \le M$ and, hence, no node of $T(s)$ has a label larger than M. Let

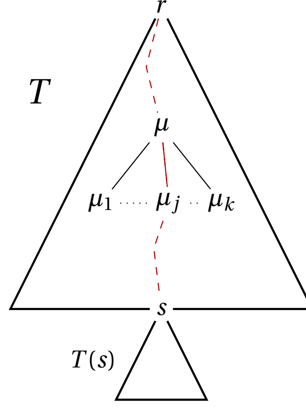


Figure 1: Scheme of the composition of the tree T.

μ be the closest ancestor of s to have a label larger than M. μ exists as $l(r) > M$ and $l(s) \leq M$. Let $\mu_1, \ldots, \mu_k$ be the children of μ, ordered such that $l(\mu_i) \geq l(\mu_{i+1})$. Let j be such that μ_j is either s or one of its ancestors. Let $t = \min\{i \in [1;k] \mid l(\mu_i) + i - 1 > M\}$ (t exists because, by definition, $l(\mu) > M$). See Figure 1 for an illustration of the tree.

Let T' be the tree obtained from T by replacing s by a leaf, therefore replacing the subtree $T(s)$ by a single node s. As $T(s)$ cannot be empty, T' contains fewer nodes than T. Consider a schedule P' on T' that executes the same operations as P on T and in the same order, except for the ones concerning $T(s)$.

We use the following notation: as above, l, m, c, w are defined on nodes of the tree T, whereas l', m', c', w' refer to the same values on the tree T'. The nodes in T' share the same names as their equivalent in T.

We define, as in the proof of Lemma 4, an ordering $\mu'_1, \ldots, \mu'_k$ on the children of μ, $l'(\mu'_i) \geq l'(\mu'_{i+1})$. Furthermore, we assume that this order is consistent with the original one, which means the following. Let j' be such that $\mu_j = \mu'_{j'}$. Note that $j' \geq j$. For $i \in [j+1; j']$, we have $\mu_i = \mu'_{i-1}$; at j, $\mu_j = \mu'_{j'}$; for $i \notin [j; j']$, we have $\mu_i = \mu'_i$. In particular, we have $\mu_{j'} = \mu'_{j'-1}$ if $j' > j$ and $\mu_{j'} = \mu'_{j'}$ if $j' = j$.

Note that, except s and its ancestors, every node v of T' satisfies $l(v) = l'(v)$ and $w(v) = w'(v)$. Our objective is to prove that $W(T') \geq W(T) - 1$. We first prove that $l'(\mu) \geq M$. We split into cases based on the value of t defined above:

1. $t < j$. The labels of $\mu_1, \ldots, \mu_t$ are left unchanged so $l'(\mu) \geq l'(\mu_t) + t - 1 > M$.

2. $t = j$. By definition of μ, we have $l(\mu_j) \leq M$, so we cannot have $t = j = 1$. The labels of $\mu_1, \ldots, \mu_{t-1}$ are left unchanged, and $l(\mu_{t-1}) \geq l(\mu_t)$, so

$$l'(\mu) \geq l'(\mu_{t-1}) + t - 2 \geq l(\mu_t) + t - 1 - 1 > M - 1.$$

3. $t > j$. Among $\mu_1, \ldots, \mu_t$, the only label that changed is μ_j. Therefore there are $t-2$ nodes that have a label l' larger than that of μ_t. Hence,

$$l'(\mu) \geq l'(\mu_t) + t - 2 > M - 1.$$

Now, we prove that $w'(\mu) \geq w(\mu) - 1$, by showing that there exists at most one index i such that $c(\mu_i) = 1$ and $c'(\mu_i) = 0$. Let I be the set of such indexes. Note that no index strictly smaller than j can be in I as the relevant labels are identical in both trees.

The following studies how the labels c and c' can differ. We consider two cases:

1. $c(\mu_j) = 0$. Thus $j \notin I$. Let $a = \min\{i \in [j+1, k] \mid c(\mu_i) = 1\}$. There are several cases; in each we show that I contains at most one element.

   (a) First, a does not exist. Then I is empty.

   (b) Assume $a > j'$. No index in $[1; j]$ can be in I, and thus no index in $[1; j']$. In particular, $c(v_l) = 0$ for $l \in [j; j']$ by definition of a. Because node μ_j appears right after node $\mu_{j'}$ in T', then $m'(\mu_j) = m'(\mu_{j'}) + (1 - c(\mu_{j'})) = (m(\mu_{j'}) - (1 - c(\mu_j))) + (1 - c(\mu_{j'})) = m(\mu_{j'}) + c(\mu_j) - c(\mu_{j'}) = m(\mu_{j'})$. Therefore, we have $m'(\mu_j) = m(\mu_{j'})$. As $l'(\mu_j) \leq l'(\mu_{j'})$, we get $m'(\mu_j) + l'(\mu_j) \leq m(\mu_{j'}) + l'(\mu_{j'})$. Then, because $l'(\mu_{j'}) = l(\mu_{j'})$, and by the definition of c, we conclude that $c'(\mu_j) \leq c(\mu_{j'})$.

   By definition, $j' \geq j$. Because $a > j'$, if $j' > j$, then $c(\mu_{j'}) = 0$ by definition of a. Otherwise $j' = j$ and we use the assumption $c(\mu_j) = 0$ to conclude that in all cases $c(\mu_{j'}) = 0$. Combined with $c'(\mu_j) \leq c(\mu_{j'})$ this gives us $c'(\mu_j) = 0$.

   Recall that the labels in $[1; j-1]$ are left unchanged, so $c(\mu_i) = c'(\mu_i)$ for $i \in [1; j-1]$. From what precedes, $c'(\mu_j) = c(\mu_j) = 0$. By definition of a and because $j' < a$, $c(\mu_i) = 0$ for $i \in [j+1; j']$. Thus, all these nodes have the same label l in T' and T, and all of them have $m'(\mu_i) \leq m(\mu_i)$ (by definition of m: they are preceded by the same nodes so their sums have the same terms, except node μ_j). Therefore, for all these nodes $c'(\mu_i) = 0$ and thus $c'(\mu_i) = c(\mu_i)$. Hence, $m(\mu_{j'+1}) = m'(\mu_{j'+1})$. Because $l(\mu_{j'+1}) = l'(\mu_{j'+1})$ we conclude that $c(\mu_{j'+1}) = c'(\mu_{j'+1})$. We then proceed by a simple induction on the nodes with a larger index to prove that I is empty.

   (c) Now, assume $a \leq j'$. Once again, because the labels in $[1; j-1]$ are left unchanged, and because $c(\mu_j) = 0$, no index in $[1; j]$ can be in I, and thus no index in $[1; a-1]$ can be in I.

   We have two cases to consider, depending on whether a is equal to 2 (recall that by definition $a \geq j+1 \geq 2$).

      i. $a = 2$. Then $j = 1$. Therefore, in T', μ_a is the first child and, by definition of c, $c'(\mu_a) = 0$.

      ii. $a > 2$. By definition of a, $c(\mu_a) = 1$. Then, either $a = j+1$ and then $a - 1 = j$ and $c(\mu_{a-1}) = c(\mu_j) = 0$, or $a > j+1$ and then $c(\mu_{a-1}) = 0$ by definition of a. In all cases, $c(\mu_{a-1}) = 0$. Therefore, $l(\mu_{a-1}) + m(\mu_{a-1}) \leq M$. Because $l(\mu_{a-1}) \geq l(\mu_a)$ and $m(\mu_a) = m(\mu_{a-1}) + 1$, $l(\mu_a) + m(\mu_a) \leq M + 1$. Because $c(\mu_a) = 1$ by definition of a, $l(\mu_a) = m(\mu_a) \leq M + 1$.
      Recall (for the third time) that the labels in $[1; j-1]$ are left unchanged, so $c(\mu_i) = c'(\mu_i)$ for $i \in [1; j-1]$. Moreover, by definition of a, $c(\mu_i) = 0$ for all $i \in [j+1; a-1]$. Therefore, because $c(\mu_j) = 0$, for all $i \in [j+1; a-1]$ $m'(\mu_i) = m(\mu_i) - 1$ and thus $c'(\mu_i) = c(\mu_i) = 0$. Also, $m'(\mu_a) = m(\mu_a) - 1$. Then $l'(\mu_a) + m'(\mu_a) = l(\mu_a) + m(\mu_a) - 1 = M$ from what precedes. Therefore, $c'(\mu_a) = 0$.

   Because $c'(\mu_a) = 0$, $m'(\mu_{a+1}) = m(\mu_{a+1})$. Then, by an immediate induction, $m'(\mu_i) = m(\mu_i)$ for $i \in [a+1; j']$. Therefore $[a+1; j'] \cap I = \emptyset$. In order to prove that $[j'+1; k] \cap I = \emptyset$, we have two cases to consider:

      i. $c'(\mu_j) = 1$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1})$. Indeed, the only nodes with an index not larger than j' that have different values for c and c' are μ_j and a. Therefore $c'(\mu_{j'+1}) = c(\mu_{j'+1})$. We can then proceed by induction to show that no index larger than j' belongs to I.

      ii. $c'(\mu_j) = 0$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1}) + 1$, and therefore $c'(\mu_{j'+1}) \geq c(\mu_{j'+1})$. We can then proceed by induction to show that for any index i larger than j' we have $m'(\mu_i) \geq m(\mu_i)$ and $c'(\mu_i) \geq c(\mu_i)$.

Therefore, we have $I = \{a\}$.

2. $c(\mu_j) = 1$. Recall that the labels in $[1; j-1]$ are left unchanged, so no index in $[1; j-1]$ can be in I. We now want to show that no index in $[j+1; k]$ can be in I. By definition of m and since $c(\mu_j) = 1$, we have $m(\mu_{j-1}) = m(\mu_j)$. Then for all $i \in [j+1; j']$, we have $l(\mu_i) = l'(\mu_i)$, and we get by an immediate induction that for all $i \in [j+1; j']$, we have $c(\mu_i) = c'(\mu_i)$. In order to prove the result on the interval $[j'+1; k]$, we have two cases to consider:

   (a) $c'(\mu_j) = 1$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1})$, and therefore $c'(\mu_{j'+1}) = c(\mu_{j'+1})$. We can then proceed by induction to show that no index larger than j' belongs to I.

   (b) $c'(\mu_j) = 0$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1}) + 1$, and therefore $c'(\mu_{j'+1}) \geq c(\mu_{j'+1})$. We can then proceed by induction to show that for any index i larger than j' we have $m'(\mu_i) \geq m(\mu_i)$ and $c'(\mu_i) \geq c(\mu_i)$.

   Therefore, $I \subseteq \{j\}$.

Putting things together, no node of $T(s)$ has a label l larger than M, so none has a positive label w. Between μ and s, no node had a label l larger than M. Therefore, except μ and its ancestors, all the nodes satisfy $w'(v) = w(v)$.

As $l'(\mu) \geq M$, all the ancestors v of μ satisfy $l'(v) \geq M$, so by Lemma 4, they also satisfy $w'(v) = w(v)$. Then, as $w'(\mu) \in \{w(\mu) - 1, w(\mu)\}$, we have $W(T') \geq W(T) - 1$.

By the induction hypothesis, P' executes at least $W(T') = W(T) - 1$ I/Os, so P executes at least $W(T)$ I/Os, which proves the lemma. □

We are now ready to prove Theorem 4.

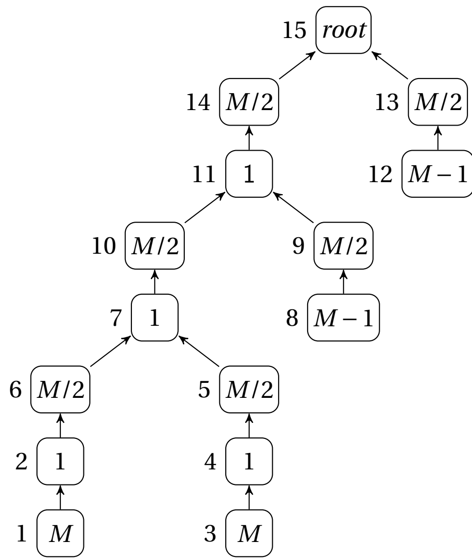
Proof of Theorem 4. Because of Lemma 3 and of Lemma 5, POSTORDER is optimal for homogeneous trees. However, POSTORDERMINIO is a post-order that minimizes the volume of I/O minimization. Hence, it is also optimal for homogeneous trees. □

Note that, on homogeneous trees, POSTORDER and POSTORDERMINIO are almost identical: POSTORDER sorts children by non-increasing l_i, while POSTORDERMINIO sorts them by non-increasing $A_i = min(M, l_i - 1)$. In particular, for children with $l_i > M$, the order is not significant for POSTORDERMINIO.

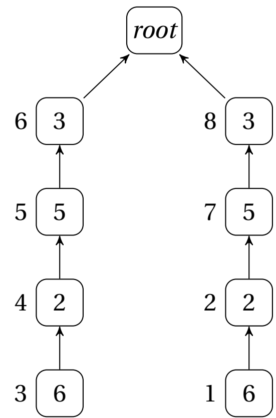
4.3 Postorder traversals are not competitive

Previous research has shown that the best postorder traversal for the MINMEM problem is arbitrarily far from the optimal traversal [3]. We prove here that postorder traversals may also have bad performance for the MINIO problem. More specifically, we prove that there exist problem instances on which POSTORDERMINIO performs arbitrarily more I/O than the optimal I/O amount. We could exhibit an example where the optimal traversal does not perform any I/O and POSTORDERMINIO performs some I/O, but we rather present a more general example where the optimal traversal performs some I/O: in the following example, the optimal traversal requires 1 I/O, when POSTORDERMINIO requires $\Omega(nM)$ I/Os. The tree used in this instance is depicted on Figure 2(a).

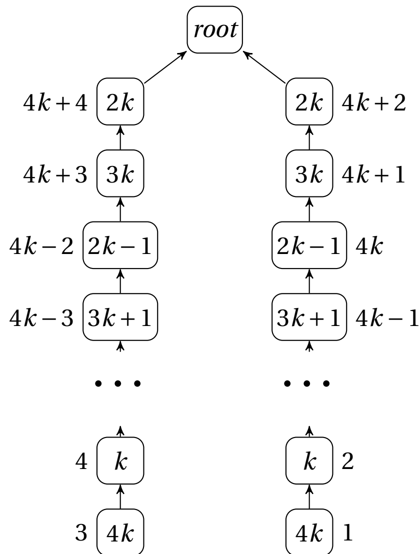
It is possible to traverse the tree of Figure 2(a) with a memory of size M using only a single I/O, by executing the nodes in increasing order of the labels next to the nodes. After processing the minimal subtree including the two leftmost leaves, our strategy is to process leaves from left to right. Before processing a new leaf, we complete the previous subtree up to a node of weight 1; this way the leaf and the actives nodes can both fit in memory.



(a) Example of a tree showing that POSTORDERMINIO is not an approximation algorithm.



(b) Example of a tree where OPTMINMEM is not optimal for MINIO ($M = 6$).



(c) Example of a tree showing that OPTMINMEM is not an approximation algorithm ($M = 4k$).

Figure 2: The label inside node i represents w_i. The label next to the nodes indicate in the leftmost figure the optimal schedule, and in the other two figures the OPTMINMEM schedule.

On the other hand, the best postorder traversal must perform a volume of I/O equal to $M/2-1$ before it can start any leaf except for the first leaf it processes. This is because the least common ancestor of any two leaf nodes has two nodes of size $M/2$ as children, and all leaves have size at least $M-1$. Thus, any postorder traversal incurs at least $M/2-1$ I/Os for all but one leaf node ($3M/2-2$ for the example here). We can extend the tree in Figure 2(a): we replace *root* by a node of size 1, add to it a parent of size $M/2$ which is the left child of the new root; the right child of the new root is then a chain containing a leaf of size $M-1$ and its parent of size $M/2$. Doing this repeatedly until n nodes are used gives the lower bound of $\Omega(nM)$. Therefore, POSTORDERMINIO is not constant-factor competitive.

4.4 OPTMINMEM is not competitive

Minimizing the amount of I/O in an out-of-core execution seems close to minimizing the peak memory when the memory in unbounded. Thus, in order to derive a good solution for MINIO, it seems reasonable to use an optimal algorithm for MINMEM, such as the OPTMINMEM algorithm presented by Liu [2], to compute a schedule σ and then to perform I/Os using the FiF policy. In the following, we also use OPTMINMEM to denote this strategy for MINIO. We prove here that there exist problem instances on which this strategy will also perform arbitrarily more I/Os than the optimal traversal.

We first exhibit in Figure 2(b) a tree showing that OPTMINMEM does not always lead to minimum I/Os in our model. Let $M = 6$. The tree of Figure 2(b) can be completed with 3 I/Os, by doing one chain after the other. This corresponds to a peak memory of 9. But OPTMINMEM achieves a peak memory of 8 at the cost of 4 I/Os by executing the nodes in increasing order of the labels next to the nodes.

This example can be extended to show that OPTMINMEM may perform arbitrarily more I/Os than the optimal strategy. The extended tree is illustrated on Figure 2(c). It contains two identical chains of length $2k+2$, for a given parameter k, and the memory size is set to $4k$. The weights of the tasks in each chain (in order from root to leaf) are defined by interleaving two sequences: $\{2k, 2k-1, \ldots, k\}$ and $\{3k, 3k+1, \ldots, 4k\}$. As above, it is possible to schedule this tree with only $2k$ I/Os, but with a memory peak of $6k$, by computing first one entire chain. However, OPTMINMEM achieves a memory peak of $5k$ by switching chains on each node with a weight smaller than $2k$, as represented by the labels besides the nodes. Doing so, OPTMINMEM incurs k I/Os on each of the $k+1$ smallest nodes, leading to a cost of $k(k+1)$ I/Os. The competitive ratio is then larger than $k/2$. Therefore, OPTMINMEM is not constant-factor competitive in the MINIO problem.

4.5 Complexity unknown

As shown above, polynomial-time approaches based on similar problems fail to even give a constant-competitive ratio. The main issue facing a polynomial approach is the highly nonlocal aspect of the optimal solution. For example, since postorder traversals are not optimal, it may be highly useful to stop at intermediate points of a subtree's execution in order to process entirely different subtrees.

We conjecture that this problem is NP-hard due to these difficult dependencies. As mentioned above, if we require entire nodes to be written to disk, the problem has been shown to be NP-hard by reduction to Partition [3]. However, this proof depends entirely on indivisible nodes, rather than on the tree's recursive structure. Taking advantage of the structure of our problem to give an NP-hardness result could lead to an interesting understanding of optimal solutions, and possibly further heuristics. We leave this as an open problem.

5 Heuristic

We now move to the design of a novel heuristic FULLRECEXPAND whose goal is to improve the performance of OPTMINMEM for the MINIO problem. The main idea of this heuristic is to run OPTMINMEM

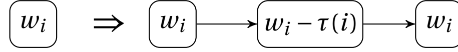


Figure 3: Example of node expansion.

several times: when we detect that some I/O is needed on some node, we force this I/O by transforming the tree. This way, the following iterations of OPTMINMEM will benefit from the knowledge of this I/O. We continue transforming the tree until no more I/Os are necessary.

In order to enforce I/Os, we use the technique of *expanding* a node (illustrated on Figure 3). Under an I/O function τ, we define the ***expansion*** of a node i as the substitution of this node by a chain of three nodes i_1, i_2, i_3 of respective weights w_i, $w_i - \tau(i)$ and w_i. The expansion of a node actually mimics the action of executing I/Os: the weight of the three tasks represent which amount of main memory is occupied by this node 1) when it is first completed ($w_{i_1} = w_i$), 2) when part of it is moved to disk ($w_{i_2} = w_i - \tau(i)$), and 3) when the whole data is transferred back to main memory ($w_{i_3} = w_i$).

This technique first allows us to prove Theorem 2, which states that given an I/O function τ, we can find a schedule σ such that (σ,τ) is a valid traversal if there exists one.

Proof of Theorem 2. Consider the tree G' obtained from G by expanding all the nodes for which τ is not null. Then, consider the schedule σ' obtained by OPTMINMEM on G', and let σ be the corresponding schedule on G. Then, the memory used by σ on G during the execution of a node i is the same as the one used by σ' on G' during the execution of the same node i, or of i_1 if i is expanded. Then, as OPTMINMEM achieves the optimal memory peak on G', we know that σ uses as little main memory as possible under the I/O function τ. Then, (σ,τ) is a valid traversal of G. □

The heuristic FULLRECEXPAND is described in Algorithm 2. The main idea of the heuristic is to expand nodes in order to obtain a tree that can be scheduled without I/O, which is equivalent to building an I/O function.

First, the heuristic recursively calls itself on the subtrees rooted at the children of the root, so that each subtree can be scheduled without I/O (but using expansions). Then, the algorithm computes OPTMINMEM on this new tree, and if I/Os are necessary, it determines which node should be expanded next. This selection is the only part where FULLRECEXPAND can deviate from an optimal strategy. Our choice is to select a node on which the FiF policy would incur I/Os; if there are several such nodes, we choose the one whose parent is scheduled the latest. After the expansion, the algorithm recomputes OPTMINMEM on the modified tree, and proceeds until no more I/O are necessary.

At the end of the computation, the returned schedule is obtained by running OPTMINMEM on the final tree computed by FULLRECEXPAND, and by transposing it on the original tree. The I/O performance of this schedule is then equal to the sum of the expansions.

FULLRECEXPAND is only a heuristic: it may give suboptimal results but also achieve better performance than OPTMINMEM, as illustrated on several examples Appendix A.

Unfortunately, the complexity of FULLRECEXPAND is not polynomial, as the number of iterations of the while loop at Line 3 cannot be bounded by the number of nodes, but may depend also on their weights. We therefore propose a simpler variant, named RECEXPAND, where the while loop at Line 3 is exited after 2 iterations. In this variant, the resulting tree G might need I/Os to be executed. The final schedule is computed as in FULLRECEXPAND, by running OPTMINMEM on this tree G. We later show that this variant gives results which are very similar to the original version.

Algorithm 2: FULLRECEXPAND (G, r, M)

Input: tree G, root of exploration r
Output: Return a tree G_r which can be executed without I/O, obtained from G by expanding several nodes

1 **foreach** *child i of r* **do**
2     $G_i \leftarrow$ FULLRECEXPAND(G, i, M)
3 $G_r \leftarrow$ tree formed by the root r and the subtrees G_i
4 **while** OPTMINMEM(G_r, r) *needs more than a memory M* **do**
5     $\tau \leftarrow$ I/O function obtained from OPTMINMEM(G_r, r) using the FiF policy
6     $i \leftarrow$ node for which $\tau(i) > 0$ whose parent is scheduled the latest in OPTMINMEM(G_r, r)
7   modify G_r by expanding node i according to $\tau(i)$
8 **return** G_r

6 Numerical results

In this section, we compare the performance of the two existing strategies OPTMINMEM and POSTORDERMINIO, and the two proposed heuristics FULLRECEXPAND and RECEXPAND. All algorithms are compared through simulations on two datasets described below. Because of its high computational complexity, FULLRECEXPAND is only tested on the first smaller dataset.

6.1 Datasets

The first dataset, named SYNTH, is composed of 330 instances of synthetic binary trees of 3000 nodes, generated uniformly at random among all binary trees. As we considered small trees, we simply used half-Catalan numbers in order to draw a tree, similarly to the method described at the beginning of [15]. The memory weight of each task is uniformly drawn from [1; 100].

The second dataset, named TREES, is composed of 329 elimination trees of actual sparse matrices from the University of Florida Sparse Matrix Collection[1] (see [3] for more details on elimination trees and the data set). Our datasets corresponds to the 329 smallest of the 640 trees presented in [3], with trees ranging from 2000 to 40000 nodes.

For each tree of the two datasets, we first computed the minimal memory size necessary to process the tree nodes: $LB = \max_i \bar{w}_i$. We also computed the minimal peak memory for an incore execution $Peak_{incore}$ (using OPTMINMEM). We eliminated some trees from the TREES dataset where $Peak_{incore} = LB$, leaving us with 133 remaining trees in this dataset. In all other cases, note that the possible range for the memory bound M such that some I/Os are necessary is $[LB, Peak_{incore} - 1]$. We chose to set M to the middle of this interval $M = (LB + Peak_{incore} - 1)/2$. Simulations using other values of M in this interval are presented in Appendix B.

6.2 Results

Our objective in this study is to minimize the total amount of I/Os needed to process the tree. In order to summarize and compare the performance of the different strategies we choose here to consider the number of I/Os and the memory bound M: performing 10 I/Os when the optimal only needs 1 does not have the same significance if the main memory consists of $M = 10$ slots or of $M = 1000$ slots. Therefore, in this section, if a schedule performs k I/Os, we define its performance as $(M + k)/M$.

[1] `http://www.cise.ufl.edu/research/sparse/matrices/`

Then, a schedule with no I/O operations has a performance of 1 while a schedule needing M I/Os has a performance of 2.

In order to compare the performance of these algorithms, we use a generic tool called *performance profile* [16]. For a given dataset, we compute the performance of each algorithm on each tree and for each memory limit. Then, instead of computing an average above all the cases, a performance profile reports a cumulative distribution function. Given a heuristic and a threshold τ expressed in percentage, we compute the fraction of test cases in which the performance of this heuristic is at most τ% larger than the best observed performance, and plot these results. Therefore, the higher the curve, the better the method: for instance, for an overhead $\tau = 5\%$, the performance profile shows how often a given method lies within 5% of the smallest performance obtained.

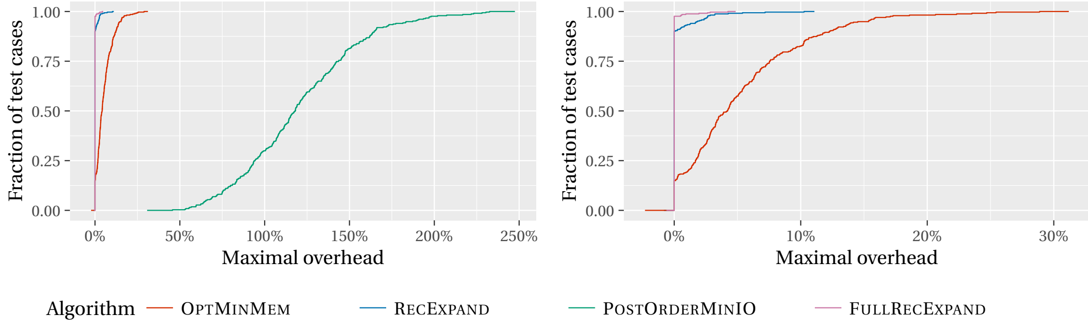


Figure 4: Performance profiles of FullRecExpand, RecExpand, OptMinMem and PostOrderMinIO on the Synth dataset (right: same performance profile without PostOrderMinIO).

The left plot of Figure 4 presents the performance profile of the four heuristics for the complete dataset Synth. The first result is the poor performance of PostOrderMinIO in this dataset: it almost always has at least 50% of overhead, and even a 100% overhead in 75% of the cases. Then, RecExpand performs far better than OptMinMem. The right plot of the figure presents the performance profiles of exclusively OptMinMem, RecExpand and FullRecExpand. RecExpand performs strictly less I/Os than OptMinMem on 90% of the instances, and on half of them, OptMinMem has a 4% overhead. We can also note that FullRecExpand performs only slightly better than RecExpand, but both heuristics are far ahead of OptMinMem, so the gain in the complexity of the algorithm is only balanced by a small loss of performance. For instance, RecExpand has an overhead of more than 2% over FullRecExpand on only 3% of the instances.

The left plot of Figure 5 presents the performance profiles of the three heuristics PostOrderMinIO, RecExpand and OptMinMem for the complete dataset Trees. The first remark is that the three heuristics are equal on more than 90% of the 329 instances. Therefore, we now focus on the right plot, which presents the same performance profile for the 25 cases where the heuristics do not all give equal performance. We can see that the hierarchy is the same as in the previous dataset (RecExpand is never outperformed, and OptMinMem performs better than PostOrderMinIO) but with smaller discrepancies between the heuristics. PostOrderMinIO and OptMinMem respectively have more than 5% of overhead on only 40% and 10% of these instances.

7 Conclusion

In this paper, we revisited the problem of minimizing I/O operations in the out-of-core execution of task trees. We proved that existing solutions allow to optimally solve the problem when all output data

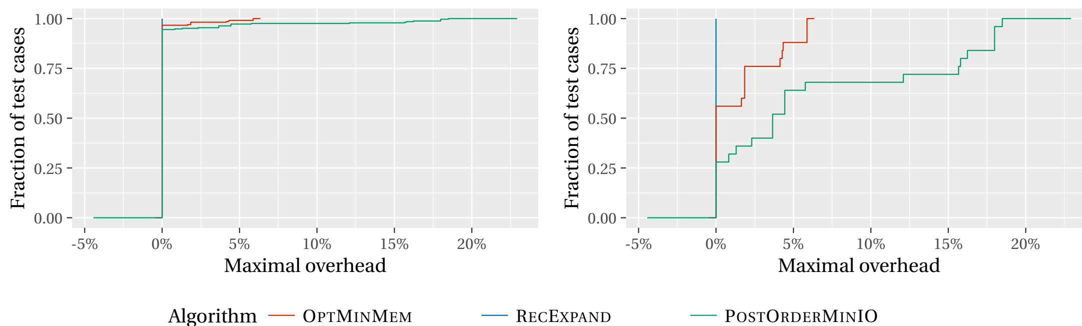


Figure 5: Performance profiles for the complete Trees dataset (left) and restricted to instances where the heuristics differ (right)

have identical size, but that none of them has a constant competitive factor compared to the optimal solution. We proposed a novel heuristic solution that improves on an existing strategy and proved very efficient in practice. Despite our efforts, the complexity of the problem remains open. Determining this complexity would definetely be a major step, although our findings already lays the bases for more advanced studies. This includes moving to parallel out-of-core execution, as we already did for parallel incore execution [4], but also designing competitive algorithm for the sequential problem.

8 Acknowledgment

This material is based upon research supported by the SOLHAR project operated by the French National Research Agency (ANR) and the Chateaubriand Fellowship of the Office for Science and Technology of the Embassy of France in the United States.

References

[1] T. A. Davis, *Direct Methods for Sparse Linear Systems*, ser. Fundamentals of Algorithms. Philadelphia: Society for Industrial and Applied Mathematics, 2006.

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

[3] M. Jacquelin, L. Marchal, Y. Robert, and B. Ucar, “On optimal tree traversals for sparse matrix factorization,” in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS’11)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 556–567.

[4] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, “Parallel scheduling of task trees with limited memory,” *TOPC*, vol. 2, no. 2, p. 13, 2015. [Online]. Available: http://doi.acm.org/10.1145/2779052

[5] E. Agullo, “On the out-of-core factorization of large sparse matrices,” Ph.D. dissertation, École normale supérieure de Lyon, France, 2008. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00563463

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

[7] J. W. H. Liu, “On the storage requirement in the out-of-core multifrontal method for sparse factorization,” *ACM Transaction on Mathematical Software*, 1986.

[8] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L’Excellent, “A fully asynchronous multifrontal solver using distributed dynamic scheduling,” *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.

[9] P. R. Amestoy, A. Guermouche, J.-Y. L’Excellent, and S. Pralet, “Hybrid scheduling for the parallel solution of linear systems,” *Parallel Computing*, vol. 32, no. 2, pp. 136–156, 2006.

[10] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J. L’Excellent, and F. Rouet, “Robust memory-aware mappings for parallel multifrontal factorizations,” *SIAM J. Scientific Computing*, vol. 38, no. 3, 2016.

[11] S. Toledo, “A survey of out-of-core algorithms in numerical linear algebra,” in *External Memory Algorithms, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-22, 1998*, 1998, pp. 161–180.

[12] E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, and Ü. V. Çatalyürek, “An out-of-core task-based middleware for data-intensive scientific computing,” in *Handbook on Data Centers*, S. U. Khan and A. Y. Zomaya, Eds. Springer, 2015, pp. 647–667. [Online]. Available: http://dx.doi.org/10.1007/978-1-4939-2092-1_22

[13] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[14] M. Lee, P. Michaud, J. S. Sim, and D. Nyang, “A simple proof of optimality for the MIN cache replacement policy,” *Inf. Process. Lett.*, vol. 116, no. 2, pp. 168–170, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.ipl.2015.09.004

[15] E. Mäkinen, “Generating random binary trees—a survey,” *Information Sciences*, vol. 115, no. 1-4, pp. 123–136, 1999.

[16] D. E. Dolan and J. J. Moré, “Benchmarking optimization software with performance profiles,” *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, 2002.

A Illustration of FULLRECEXPAND on some examples

The left-hand side of Figure 6 provides an example where FULLRECEXPAND performs better than OPTMINMEM. OPTMINMEM computes the left branch first until node a, then the right branch until node b, before completing the left branch. The memory peak reached is 12, but this schedule incurs 4 I/Os with a memory limit of 10: 2 on node a and 2 on node b. On this example, FULLRECEXPAND expands node b as specified on the middle diagram. With this expansion, OPTMINMEM schedules the right branch until b_2 first, then the whole left branch, using one more I/O on b_2. This node is expanded a second time on the right diagram, without changing the schedule obtained by OPTMINMEM, yielding to 3 I/Os on the original tree, all on b.

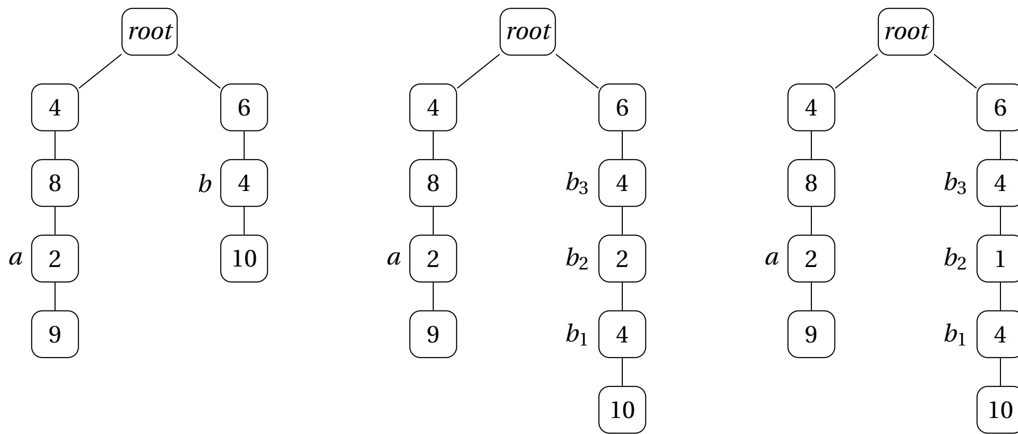


Figure 6: Example where FULLRECEXPAND is optimal whereas OPTMINMEM and POSTORDERMINIO are not. Let $M = 10$. The left tree is the original one, and the others are obtained during the execution of FULLRECEXPAND after the expansion of b.

Figure 7 provides an example where FULLRECEXPAND does not improve OPTMINMEM. On this instance, OPTMINMEM performs 4 I/Os, 2 on node a then 2 on node b, where POSTORDERMINIO executes first the left subtree and consumes only 3 I/Os on node c. This instance shows an example where no optimal solution perform an I/O on a node where OPTMINMEM performs an I/O. So the strategy of FULLRECEXPAND cannot be optimal, even if we used a different priority at Line 6.

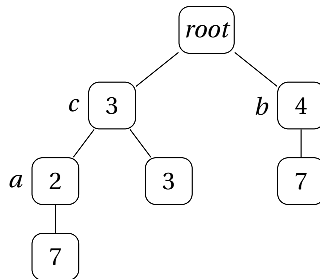


Figure 7: Example where FULLRECEXPAND and OPTMINMEM are not optimal whereas POSTORDERMINIO is. $M = 7$ in this example.

B Numerical results with other memory bounds

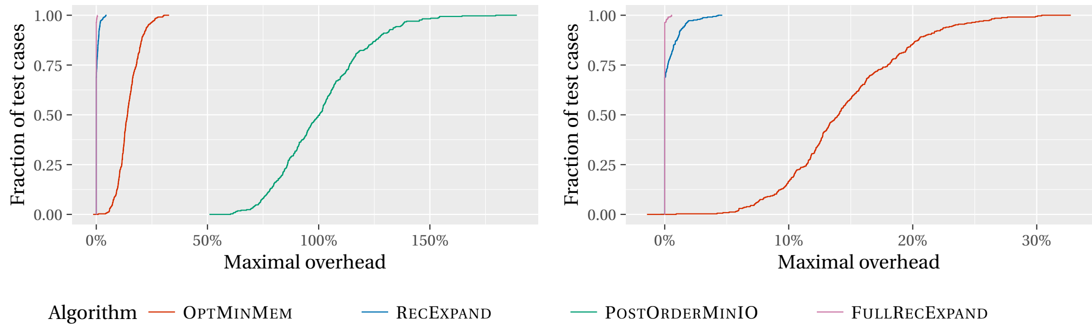


Figure 8: Performance profile of FullRecExpand, RecExpand, OptMinMem and PostOrderMinIO on the Synth dataset with the M_1 memory bound (right: same performance profile without PostOrderMinIO).

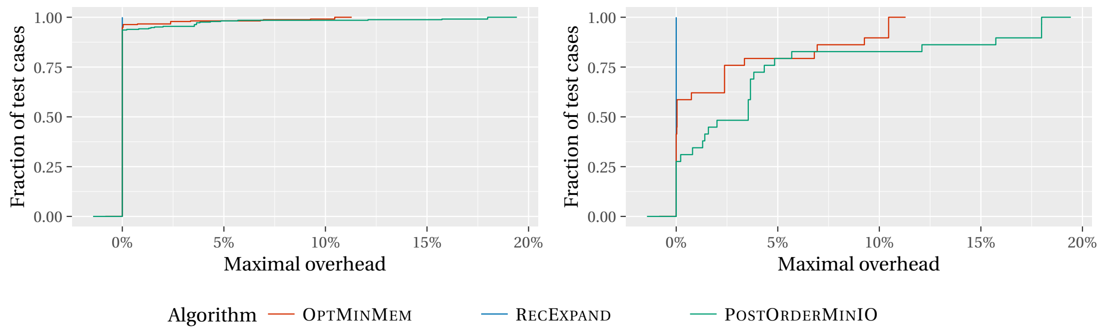


Figure 9: Performance profiles for the complete Trees dataset with the M_1 memory bound (left) and for the instances where the heuristics differ (right)

In this section, we present numerical results on the same datasets than the ones presented in Section 6, but with different memory bounds.

First, we use the memory bound $M_1 = LB$, which is defined in Section 6, and represents the minimum memory bound for which it is possible to compute a given tree. We plot the corresponding performance profiles for the Synth dataset in Figure 8 and the Trees dataset in Figure 9. The main conclusion that can be made comparing to the results of Section 6 is that the difference between OptMinMem and RecExpand is significantly larger with this memory bound. Indeed, there is a 10% of overhead for OptMinMem in 90% of the cases whereas such an overhead was reached in only 15% of the cases previously. This can be explained by the fact that the memory bound considered here is further from the memory required by MinMem. On the other hand, the difference between PostOrderMinIO and RecExpand are smaller in this case: there is a 100% of overhead for PostOrderMinIO in half of the cases whereas we had this property in 75% of the cases with a higher memory bound. The same tendency can be observed for the Trees dataset in Figure 9, even if it is less significant.

Second, we use the memory bound $M_2 = Peak_{incore} - 1$, which is on the opposite the largest memory bound for which I/Os are requires in order to compute a tree. The corresponding profiles for the

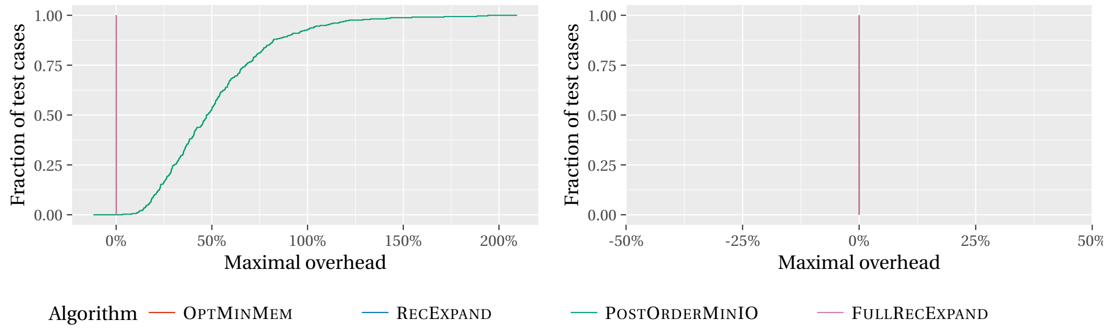


Figure 10: Performance profile of FULLRECEXPAND, RECEXPAND, OPTMINMEM and POSTORDER-MINIO on the SYNTH dataset with the M_2 memory bound (right: same performance profile without POSTORDERMINIO).

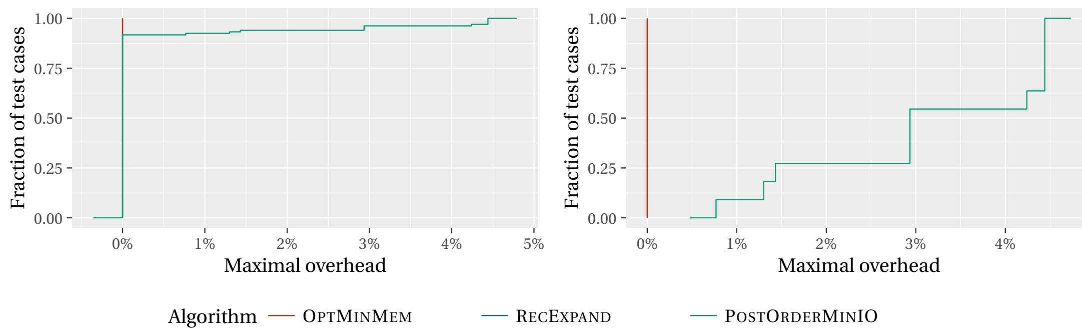


Figure 11: Performance profiles for the complete TREES dataset with the M_2 memory bound (left) and for the instances where the heuristics differ (right)

SYNTH and TREES dataset can be found in Figures 10 and 11. With this memory bound, OPTMINMEM, RECEXPAND and FULLRECEXPAND are always equal, and only POSTORDERMINIO has worse performances. This can be explained by the fact that M_2 is right below the memory requires by OPTMINMEM to compute a tree without I/Os. Therefore, we can argue that it is closer to the optimal algorithm and FULLRECEXPAND does not improve the few I/Os performed by MINMEM. Nevertheless, the overhead of POSTORDERMINIO is smaller than with the other memory bounds.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

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

ISSN 0249-6399