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. International Journal of Foundations of Computer Science, 2023, 34 (01), pp.51-80. 10.1142/s0129054122500186 . hal-03758021

HAL Id: hal-03758021

https://hal.science/hal-03758021

Submitted on 13 Sep 2022

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.

Distributed under a Creative Commons Attribution 4.0 International License

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

Loris Marchal*, Samuel McCauley†, Bertrand Simon‡ and Frédéric Vivien*

*CNRS, INRIA, ENS Lyon and University of Lyon
LIP, ENS Lyon, 46 allée d'Italie
Lyon, 69007, France
loris.marchal@ens-lyon.fr
frederic.vivien@inria.fr

†Williams College
880 Main St, Williamstown MA 01267 USA
sam@cs.williams.edu

‡IN2P3 Computing Center / CNRS
21 Av. Pierre de Coubertin, 69100, Villeurbanne, France
bertrand.simon@cnrs.fr

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

Scientific applications are usually described using directed acyclic graphs, where nodes represent tasks and edges represent dependencies between tasks. For some applications, this graph is a tree: each task produces a single result used solely by its parent. The temporary results of each task have to be stored between their production and their use.

We focus on the case when the data manipulated are very large. Then, during an execution, all data may not fit together in memory. In such a case, some data have to be temporarily written to disk and evicted from memory. These data are later read from disk when they are needed for computation.

These Input/Output operations are very expensive; hence, our goal is to minimize their total volume. The order in which the tasks are processed considerably influences the amount of such Input/Output operations. Finding the schedule which minimizes this amount is an open problem that we revisit in this paper.

We first formalize and generalize known results, and prove that existing solutions can be arbitrarily worse than the optimal. We then present an Integer Linear Program to solve it optimally. Finally, we propose a novel heuristic algorithm. We demonstrate its good performance through simulations on both synthetic and realistic trees built from actual scientific applications.

Keywords: scheduling, task tree, out-of-core, I/O minimization.

1. Introduction

Scientific applications are often modeled by directed 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 Input/Output (I/O) operations 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 scheduling of rooted in-trees. Such tree-shaped task graphs arise in several computational domains, such as the factorization of sparse matrices [8], or in computational physics code modeling electronic properties [15]. Dependencies between tasks, modeled by the edges of the tree, represent the tasks' 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.

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 [13, 19]. 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 child's data, or which part of a child's data, has to be written to disk. In a previous study [13], we have focused on the case when a child's 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 resides in memory, and write only part of it to the disk if needed. This is for instance what is done by operating systems using *paging*: all data is divided into same-size *pages*, which can be moved from main memory to secondary storage when needed. Since all modern computer systems implement paging, we consider here that data can be split into parts (corresponding to system pages), each of which can be written to disk.

Note that the present study focuses on sequential scheduling for task trees. Even if parallel processing is necessary for most large-scale scientific applications, we claim that such a sequential study is needed: (i) for memory-constrained applications, a sequential traversal of the task tree is preferred, as it lowers the memory pressure, but each task is performed in parallel on multiple cores; (ii) if the ultimate goal is to obtain a parallel schedule of the tree, it is important to first investigate the sequential problem before moving to its more complex, bi-criteria (I/O and makespan) parallel variant. 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 previous results from the literature.

- A proof of optimality of post-order traversals when trees are homogeneous (all output data has the same size). An algorithm to compute the best post-order traversal was previously proposed by E. Agullo [1]; our proof shows that this algorithm is optimal for homogeneous trees.
- A proof that neither the best post-order traversal nor the memory-peak minimization algorithms are approximation algorithms for minimizing the I/O volume.
- An Integer Linear Programming (ILP) formulation of the considered problem that allows us to compute an optimal solution for small scale problems.
- The design of a new heuristic that takes advantage of peak-memory-optimizing algorithms.
- An extensive experimental comparison of all available strategies (including the ILP for small test cases) through simulations on both synthetic and realistic trees built from actual sparse matrices. These simulations show the very good performance of the proposed solution.

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. An optimal ILP is presented in Section 5. We introduce a new heuristic in Section 6 and evaluate its performance through simulations in Section 7. We finally conclude and present future directions in Section 8.

2. Related work

As stated above, rooted trees are commonly used to represent task dependencies for scientific applications. This is, for example, the case for some computational physics codes modeling the electronic properties of semiconductors and metals [15, 17, 22], and for the accurate modeling of the electronic structure of atoms and molecules in quantum chemistry [5, 12]. In the domain of sparse linear algebra, Liu [18] gives a detailed description of the construction of the elimination tree, its use for Cholesky and LU (Lower-Upper) factorizations, and its role in multifrontal direct methods: during the factorization, the computations are organized as a task tree, and the huge size of the data involved makes it absolutely necessary to reduce the memory requirement of the factorization. Note that peak memory minimization is still a crucial question for direct solvers, as highlighted by Agullo et al. [2], who study the effect of processor mapping on memory consumption for multifrontal methods.

Memory and storage have always been a limited parameter for large computations, as outlined by the pioneering work of Sethi and Ullman [25] 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 [20] 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 [19] presented 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 [13]. However, they are very natural and straightforward solutions to this problem, as they allow us to fully process one subtree before starting a new one. Therefore, they are widely used in sparse matrix software like `MUMPS` [3, 4] (MUltifrontal Massively Parallel sparse direct Solver), and achieve good performance on actual elimination trees [13].

The problem of minimizing I/Os—and not only peak memory—has been formalized as the so-called *red-blue Pebble Game* [14], where pebbles have to be placed on the computation DAG and colors respectively represent the small and fast memory. This model is adapted to unit-size data and several hardness and inapproximation results have been developed for several variants, see [23] among the recent work. Our problem restricted to unit-size data falls under the ONESHOT model in the terminology adopted in [23]. On general task graphs (i.e., directed acyclic graphs), this problem is shown to be NP-hard and impossible to approximate within a factor less than 2 unless the Unique Games Conjecture is invalid. Note that the fact that we show an optimal algorithm for unit-size data (Section 4) directly implies a polynomial-time solution for the red-blue pebble game on trees.

As mentioned in the introduction, the problem of minimizing the I/O volume when traversing a tree has been studied in [13] with the constraint that each node's data either stays in the memory or has to be written wholly to disk. Here we study the case when we have the option to store part of the data, which is also the topic of E. Agullo's PhD. thesis [1]. In his thesis, Agullo exhibited the best postorder traversal for minimizing I/O volume, which we adapt to our model in Section 4.1. He also studied 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 presented an out-of-core version of the `MUMPS` solver.

Finally, out-of-core execution is a well-known approach for computing on large data, especially (but not exclusively) in linear algebra [24, 26].

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 the t-th task computed. The function τ represents the amount of I/Os for each task: $\tau(i) = m$ means that m units of the output data of task i are written to disk (see below). 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. We assume without loss of generality 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. Finally, since there are exactly the same number of *read* and *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 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:

  $$\forall i \in V,\ 0 \leq \tau(i) \leq w_i.$$

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

$$\forall i \in V, \quad \sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} (w_k - \tau(k)) \leq M - \bar{w}_i. \tag{1}$$

The problem we are considering in this paper, called MinIO, is to find a valid traversal that minimizes the total amount of I/Os, 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 [1, 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 an 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 that the offline cache replacement policy MIN is optimal [6, 16]. MIN evicts from the cache the data which will be used the latest.

Proof. Given a tree G, a memory bound M, a schedule σ, and an 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 data i that is not the last to be used among active data. Let j denote the last-used among active data (so FiF would evict j). 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. 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 6 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/Os, 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 not make the problem less combinatorial: there are $n!$ possible schedules and already 2^n functions τ 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 peak memory, denoted MinMem, is closely 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 maximum 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 proven by Liu [19]. We refer to this algorithm as OptMinMem.
- The best postorder traversal for peak-memory minimization can also be computed in polynomial time [20]. We refer to this algorithm as PostOrderMinMem.

4. Existing solutions are not satisfactory for heterogeneous data

We now detail two existing solutions for the MinIO problem. The first one is the best postorder traversal for MinIO proposed by Agullo [1]. We show that it is optimal if all data have unit size. The second uses the optimal traversal for MinMem proposed by Liu [19], and then applies Theorem 1 to obtain a valid traversal. After presenting these algorithms, we prove that neither of them is constant-factor competitive compared to the optimal traversal.

4.1. *Computing the best postorder traversal*

For the sake of completeness, we present the algorithm computing the best postorder traversal for MinIO from [1] and adapt it to our model. 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 of T_i when I/O operations are chosen 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 in decreasing order of $A_j - w_j$ achieves the minimum V_i.

Theorem 3 (Lemma 3.1 in [20]) *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.

Algorithm 1: PostOrderMinIO (G, r)

Input: 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 for each 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 in decreasing order of A_i − w_i do
6     ℓ_r ← ℓ_r ⊕ ℓ_i
7 ℓ_r ← ℓ_r ⊕ {r}
8 return ℓ_r
```

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 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 [25], which considers binary trees from arithmetic expressions and aims to minimize the number of store/load operations when evaluating these expressions with a limited number of registers. They considered different variants, and the one with commutative operators closely resembles our problem, where the registers are replaced by memory slots and load/store by read/write. However, in our work, we do not limit the model to binary trees, but consider any tree with homogeneous data sizes. In the case that the heterogeneity in data sizes is limited, our result provides a good strategy of minimizing the number of I/O operations.

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

In order to prove this theorem, we need first to define labels on the nodes of the 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, $v_1, \ldots, v_k$ 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:

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

Let Postorder be a 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 $c(root) = 0$. 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 right before 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:

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

Intuitively, $W(T(v))$ 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 5. *With infinite memory,* Postorder *uses $l(n)$ slots to compute the subtree rooted at node n.*

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

Lemma 6. *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 $\max_{1 \leq i \leq k}(l(v_i) + i - 1)$. 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 performed using the FiF policy).

Lemma 7. *With bounded memory M,* 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 new notation: for any node v of T we define $\mathcal{W}(v)$ as $\mathcal{W}(v) = W(T(v)) - c(v)$. In other words, $\mathcal{W}(v)$ intuitively 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 $\mathcal{W}(v) = W(T(v))$ if v is the root of T. We prove by induction on the size of $T(v)$ that at most $\mathcal{W}(v)$ I/Os are performed during the execution of $T(v)$.

Let us assume that v is a leaf. Then $\mathcal{W}(v) = 0$. 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 $W(T(v)) = \mathcal{W}(v) = c(v) = 0$.

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 $\mathcal{W}(v_i)$ 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 $W(T(v_i)) = \mathcal{W}(v_i) + c(v_i)$ I/Os.

Let us consider the $(i+1)$-th child of v. If $c(v_{i+1}) = 0$, then $l(v_{i+1}) + \sum_{1 \leq j < i+1}(1 - c(v_j)) \leq M$. Then, according to Lemma 5, no I/Os are required to execute $T(v_{i+1})$ under Postorder even after the processing of $T(v_1)$ through

Lemma 9.

$T(v_i)$. Indeed, before the start of the processing of $T(v_{i+1})$ the memory contains exactly $\sum_{1\leq j<i+1}(1-c(v_j))$ nodes. Therefore $\mathcal{W}(v_{i+1}) = c(v_{i+1}) = W(T(v_{i+1})) = 0$.

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

Lemma 9 relies on the following intermediate result.

Lemma 8. *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 $l'(a) \leq l(a)$ and $l'(a) \geq M$. Then $w'(v) = w(v)$.*

Proof. Let $v_1, \dots, v_k$ be the children of v, ordered so that $l(v_1) \geq \cdots \geq l(v_k)$. 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 $l'(v'_1) \geq \cdots \geq l'(v'_k)$. Let j' be the index of a in this ordering: $v'_{j'} = a = v_j$.

Note that $j' \geq j$. 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 $l(v_{j'}) = l'(v'_{j'-1}) \geq l'(v'_{j'}) = l'(a) \geq M$. However, for any $i \in [1; j']$, $l'(v'_i) \geq l'(v'_{j'}) \geq M$ and $l(v_i) \geq l(v_{j'}) \geq M$. Therefore, for any $i \in [2; j']$, $l'(v'_i) + m'(v'_i) > M$ (because $m'(v'_i) \geq 1 - c'(v'_1) = 1$) and, thus, $c'(v'_i) = 1$. Similarly, for any $i \in [2; j']$, $l(v_i) + m(v_i) > M$ (because $m(v_i) \geq m(v_1) = 1$) 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. Finally, we have $w'(v) = \sum_{i=2}^{k} c'(v'_i) = \sum_{i=2}^{k} c(v_i) = w(v)$. □

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

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

Here is a short summary of the proof, which is given in full detail in Appendix A. The result is proven by induction on the size of the tree. The case where no I/Os are required can be deduced from Lemma 6.

We then consider a tree T for which any schedule performs at least one I/O, and an optimal schedule P on this tree. We focus on the first node s to be stored under this schedule, and define the tree T' in which $T(s)$ is replaced by s. Using the induction hypothesis, we know that any schedule on T', including the restriction of P on T', performs at least $W(T')$ I/Os. Therefore, we deduce that P performs at least $W(T') + 1$ I/Os on T. Thus, it remains to prove that $W(T') \geq W(T) - 1$.

To do so, we focus on the closest ancestor of s to have a label l larger than M, and denote it as μ. We first prove that in the new tree T', we have $l(\mu) \geq M$. This means, by Theorem 8, that the w labels of the ancestors of μ are unchanged in T'. Then, we prove through an extensive case study that $w(\mu)$ in T' cannot be smaller than $w(\mu)$ in T minus one. Finally, we conclude that all the other w labels are equal in T and in T'; therefore, $W(T') \geq W(T) - 1$.

We are now ready to prove Theorem 4.

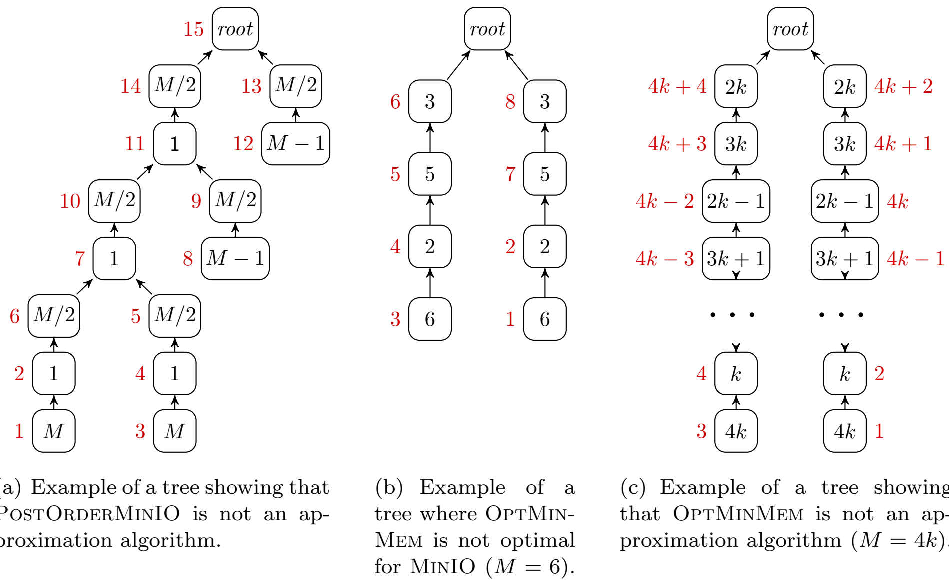
Proof of Theorem 4. From Lemmas 7 and 9, Postorder is optimal for homogeneous trees. Moreover, PostOrderMinIO is a post-order that minimizes the volume of I/O operations. Hence, it is also optimal for homogeneous trees. □

The Postorder algorithm designed in this proof is actually equivalent to PostOrderMinMem, the postorder algorithm minimizing the peak memory, when applied to homogeneous trees. The only difference with PostOrderMinIO is that the latter sorts the children by non-increasing $A_i = \min(M, l_i)$ whereas Postorder sorts them by non-increasing l_i. PostOrderMinIO is then less specific, as it does not specify the order among subtrees with $l_i \geq M$: there are more ties that can be arbitrarily broken. This difference also implies that the schedule given by Postorder does not depend on the value of M. It is thus *cache-oblivious* [11] and optimal (on homogeneous trees) for any memory size. Therefore, if we consider several levels of memory (e.g., a cache memory connected to a Random-Access Memory, itself connected to a disk), Postorder minimizes the memory transfers between every level (e.g., both cache-RAM and RAM-disk transfers). Note that with heterogeneous trees, this result does not hold anymore, as the optimal traversal depends on the memory size. Therefore, no algorithm can simultaneously minimize transfer between all levels of the memory hierarchy.

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 [13]. 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/Os than the optimal amount. We could exhibit an example where the optimal traversal does not perform any I/Os and PostOrderMinIO performs some I/Os, but we rather present a more general example where the optimal traversal does perform some I/Os: 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 1(a).



(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 1. The black label inside node i represents w_i. The red label next to the nodes indicates in (a) the optimal schedule, and in (b) and (c) the OptMinMem schedule.

It is possible to traverse the tree of Figure 1(a) with a memory of size M using only a single I/O, by executing the nodes in increasing order of the (red) 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.

On the other hand, the best postorder traversal must perform a volume of I/O equal to $M/2-1$ before processing any leaf, except for the very first processed leaf. This is because the least common ancestor of any two leaf nodes has two children of size $M/2$, and all leaves have size at least $M-1$. Thus, any postorder traversal performs at least $M/2-1$ I/Os for all but one leaf node, leading to at least $3M/2-3$ I/Os for the tree in Figure 1(a) (a best postorder starts from any of the two M leaves and performs $3M/2-2$ I/Os). We can extend this tree as follows: 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/Os in an out-of-core execution seems similar to minimizing the peak memory when the memory is 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 [19], 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 1(b) a tree showing that OptMinMem does not always lead to minimum I/Os in our model. Let $M = 6$. The tree of Figure 1(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 1(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 one entire chain, then the other. However, OptMinMem achieves a memory peak of $5k$ by alternating between chains, each time processing the chain until reaching a node with a weight smaller than $2k$, as represented by the labels besides the nodes. OptMinMem performs 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 $k/2$, and OptMinMem is not constant-factor competitive for the MinIO problem.

4.5. *Unknown complexity*

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 [13]. However, this proof highly depends on indivisible nodes, rather than on the recursive structure of trees. 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. ILP formulation of the problem

We now present an Integer Linear Program solving the MinIO problem.

The linear program relies on the boolean variables δ_{ij} to express the schedule constraints. δ_{ij} is equal to 1 if node i precedes node j in the corresponding schedule and 0 otherwise, as used previously in [7] for instance. All the variables considered in this linear program are nonnegative. The first constraints represent the antisymmetric (Equation (2)), acyclic (Equation (3)) and reflexive (Equation (4)) properties of the order, and the consistency with the precedence constraints (Equation (5)).

$$\forall i, j \in G, \quad \delta_{ij} + \delta_{ji} = 1 \tag{2}$$

$$\forall i, j, k \in G, \quad \delta_{ij} + \delta_{jk} + \delta_{ki} \geq 1 \tag{3}$$

$$\forall i \in G, \quad \delta_{ii} = 1 \tag{4}$$

$$\forall (i, j) \in E, \quad \delta_{ij} = 1 \tag{5}$$

$$\forall i, j \in G, \quad \delta_{ij} \in \{0, 1\} \tag{6}$$

We introduce the variable $\alpha_i \in [0, 1]$ which represents the fraction of node i written to disk.

$$\forall i \in G, \ 0 \leq \alpha_i \leq 1 \tag{7}$$

The memory constraint is then equivalent to the following nonlinear inequality (that will be linearized ultimately). Indeed, Equation (8) is the transposition of the memory constraint defined as Equation (1) in Section 3.1, noting that $\delta_{ki}\delta_{ip} = 1$ if and only if node k is active during the execution of node i.

$$\forall i \in G, \quad \max\left(w_i, \sum_{(j,i)\in E} w_j\right) + \sum_{\substack{(k,p)\in E,\\ k\neq i, p\neq i}} \delta_{ki}\delta_{ip}(1-\alpha_k)w_k \leq M \tag{8}$$

Finally, the objective function is to minimize the I/O cost:

$$\text{Minimize} \sum_{i\in G} \alpha_i w_i \tag{9}$$

We have now formalized the MinIO problem described in Section 3.1 through a quadratic program $\mathcal{P}_{quad}$ composed of Equations (2) to (8). It then remains to linearize Equation (8).

We define the variables x_{ik} and y_{ik}. They are constrained to satisfy the following: if node k is active during the execution of node i, then x_{ik} is equal to 1 and y_{ik} is not larger than α_k; otherwise they are both null. Note that in the special case when i is either k or its parent, both variables are forced to be 0.

$$\forall (k,p) \in E, i \notin \{k,p\}, \quad x_{ik} = \delta_{ki} + \delta_{ip} - 1 \tag{10}$$

$$\forall (k,p) \in E, i \notin \{k,p\}, \quad 0 \leq y_{ik} \leq \min(\delta_{ki}, \delta_{ip}, \alpha_k) \tag{11}$$

$$\forall (k,p) \in E, \quad x_{pk} = y_{pk} = x_{kk} = y_{kk} = 0 \tag{12}$$

$$\forall i \in G, \quad \max\left(w_i, \sum_{(j,i)\in E} w_j\right) + \sum_{(k,p)\in E} x_{ik} w_k - \sum_{(k,p)\in E} y_{ik} w_k \leq M \tag{13}$$

The final integer linear program $\mathcal{P}_{lin}$ is then composed of Equations (2) to (7) and (10) to (13), with the objective function described by Equation (9). It requires $O(n^2)$ variables and $O(n^3)$ constraints. We now prove that the linearization is correct: for any value X of the objective function, there exists a solution to $\mathcal{P}_{lin}$ of objective value X if and only if there exists a solution to $\mathcal{P}_{quad}$ of objective value X.

First, as an intermediate step, we show that if there exists a valuation of variables that satisfies Equations (2) to (7) and (10) to (12), then for $(k,p) \in E$ and $i \in G$ with $i \notin \{k,p\}$, we have $x_{ik} - y_{ik} \geq \delta_{ki}\delta_{ip}(1-\alpha_k)$. We consider such a valuation of the variables. By the precedence constraint (5), we have $\delta_{kp} = 1$. Hence, thanks to the antisymmetric (2) and acyclic (3) constraints, we deduce that we cannot have both $\delta_{ki} = 0$ and $\delta_{ip} = 0$. Therefore thanks to Equation (10), we have $x_{ik} = \delta_{ki}\delta_{ip}$. From Equation (11), we deduce $y_{ik} \leq \alpha_k$ and $y_{ik} = 0$ if $\delta_{ki}\delta_{ip} = 0$. Thus, we have $y_{ik} \leq \delta_{ki}\delta_{ip}\alpha_k$ and finally $x_{ik} - y_{ik} \geq \delta_{ki}\delta_{ip}(1-\alpha_k)$.

Assume that $\mathcal{P}_{lin}$ allows a feasible valuation of variables $\mathcal{V}$. $\mathcal{V}$ respects the conditions of the above paragraph, so that $x_{ik} - y_{ik} \geq \delta_{ki}\delta_{ip}(1-\alpha_k)$. Therefore, the left hand side of Equation (13) is not smaller than the left hand side of Equation (8). As Equation (13) is satisfied as part of $\mathcal{P}_{lin}$, Equation (8) is satisfied. $\mathcal{V}$ (restricted to the δ and α variables) is then a solution of $\mathcal{P}_{quad}$.

On the contrary, let us assume now that $\mathcal{P}_{quad}$ allows a feasible valuation of variables $\mathcal{V}$. Then, we complete $\mathcal{V}$ by setting, for $(k,p) \in E$ and $i \in G$ with $i \notin \{k,p\}$, $x_{ik} = \delta_{ki}\delta_{ip}$, $y_{ik} = \delta_{ki}\delta_{ip}\alpha_k$, and for $i \in \{k,p\}$, $x_{ik} = y_{ik} = 0$. Let $\mathcal{V}'$ be the completed valuation. In $\mathcal{V}'$, Equation (13) is then equivalent to Equation (8), which is thus also satisfied. We now show that $\mathcal{V}'$ satisfies Equations (10) and (11). Let $(k,p) \in E$ and $i \in G$ with $i \notin \{k,p\}$. By the precedence constraint (5), we have $\delta_{kp} = 1$. Hence, thanks to the antisymmetric (2) and acyclic (3) constraints, we deduce that we cannot have both $\delta_{ki} = 0$ and $\delta_{ip} = 0$. Therefore, $x_{ik} = \delta_{ki}\delta_{ip} = \delta_{ki} + \delta_{ip} - 1$, so Equation (10) is satisfied. Then, $y_{ik} = \delta_{ki}\delta_{ip}\alpha_k$ is equal to 0 if δ_{ki} or δ_{ip} is null, and to α_k otherwise, so Equation (11) is satisfied. Therefore, $\mathcal{V}'$ is a solution of $\mathcal{P}_{lin}$, achieving the same objective value as $\mathcal{V}$ in $\mathcal{P}_{quad}$.

6. 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 several times: when we detect that an 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 2). Under an I/O function τ, we define the ***expansion*** of a node i as the substitution of this node with 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 any 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 any 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 perform I/Os; if there are several such nodes, we choose the one whose parent is scheduled

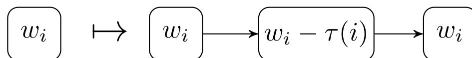


Figure 2. Example of node expansion.

the latest. After the expansion, the algorithm recomputes OptMinMem on the modified tree, and proceeds until no more I/Os 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 may achieve better performance than OptMinMem, as illustrated in several examples in Appendix B.

Unfortunately, the complexity of FullRecExpand is not polynomial, as the number of iterations of the while loop at Line 4 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 4 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 show in the next section that this variant gives results which are very similar to the original version.

7. 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.

Algorithm 2: FullRecExpand (G, r, M)

```
  Input: tree G, root of exploration r
  Output: Return a tree G_r which can be executed without any I/O,
          obtained from G by expanding several nodes
1 for each child i of r do
2     G_i ← FullRecExpand(G, i, M)
3 G_r ← tree formed by the root r and the G_i subtrees
4 while OptMinMem(G_r, r) needs more than a memory M do
5     τ ← I/O function obtained from OptMinMem(G_r, r) using the FiF
       policy
6     i ← node for which τ(i) > 0 whose parent is scheduled the latest in
       OptMinMem(G_r, r)
7     modify G_r by expanding node i according to τ(i)
8 return G_r
```

7.1. *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 used half-Catalan numbers in order to draw a tree, similarly to the method described at the beginning of [21]. The memory weight of each task is uniformly drawn from $[1; 100]$.

The second dataset, named SMALLSYNTH, is composed of 30,000 synthetic binary trees of 30 nodes, generated with the same method as the trees of SYNTH. This dataset contains trees small enough to allow the determination of the optimal solution by solving the ILP directly.

The last dataset, named TREES, is composed of 329 elimination trees of actual sparse matrices from the University of Florida Sparse Matrix Collection[a] (see [13] for more details on elimination trees and the data set). Our dataset corresponds to the 329 smallest of the 640 trees presented in [13], with trees ranging from 2000 to 40000 nodes.

For each tree of each dataset, 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 all trees from the TREES dataset where $Peak_{incore} = LB$ (i.e., trees for which out-of-core execution is useless whatever the memory bound M), 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 $[LB, Peak_{incore} - 1]$. The main memory bound we use in our simulation is the middle of this interval $M_{mid} = (LB + Peak_{incore} - 1)/2$. For a more complete analysis, we also perform the same simulations with the two extreme memory bounds $M_{min} = LB$ and $M_{max} = Peak_{incore} - 1$.

7.2. *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 vs. $M = 1000$ slots. Therefore, in this section, if a schedule performs k I/Os, we define its *relative I/O volume* as $(M + k)/M$. Then, a schedule with no I/O operations has a relative I/O volume of 1 while a schedule needing M I/Os has a relative I/O volume of 2.

In order to compare the performance of these algorithms, we use a generic tool called a *performance profile* [9]. For a given dataset, we compute the relative I/O volume of each algorithm on each tree and for each memory limit. Then, rather than computing an average across all the cases, a performance profile reports a

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

cumulative distribution function. We define the *deviation* of a heuristic on a given instance as the relative I/O volume of this heuristic divided by the best relative I/O volume achieved for this instance. We then use the deviation to the best heuristic for the datasets Synth and Trees, and the deviation to the optimal solution for the dataset SmallSynth, which is computed with the ILP. Given a heuristic and a deviation τ expressed in percentage, we compute the fraction of test cases for which the heuristic has a deviation not larger than τ, and plot these results. Therefore, the higher the curve, the better the method: for instance, for a deviation $\tau = 5\%$, the performance profile shows how often a given method lies within 5% of the smallest relative I/O volume obtained.

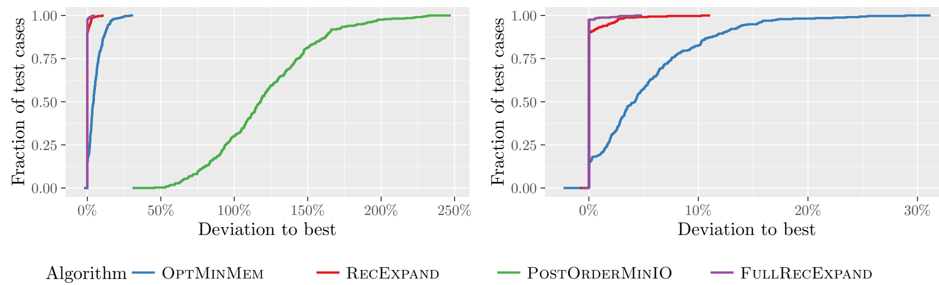


Figure 3. Performance profiles of FullRecExpand, RecExpand, OptMinMem and PostOrderMinIO on the Synth dataset with the M_{mid} memory bound (right: same performance profiles without PostOrderMinIO).

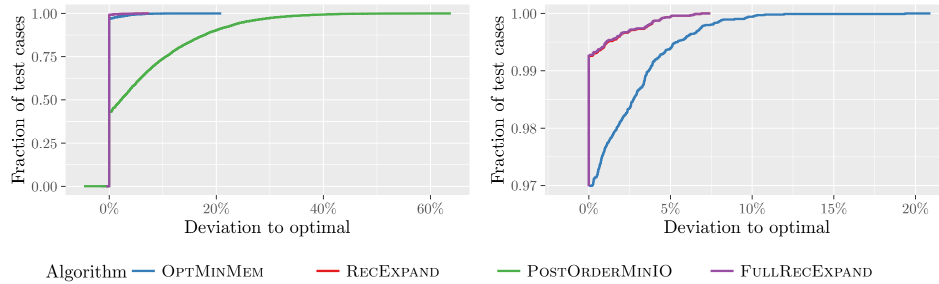


Figure 4. Performance profiles for the SmallSynth dataset with the M_{mid} memory bound (right: same performance profiles without PostOrderMinIO; note that we zoom in on the top part of the y-axis to better display instances where the heuristics are non-optimal).

The left plot of Figure 3 presents the performance profile of the four heuristics for the complete dataset Synth using the memory bound M_{mid}. The first result is the poor performance of PostOrderMinIO in this dataset: it almost always

has a deviation of at least 50%, and even of 100% in 75% of the cases. Thus, the right plot of the figure presents the performance profiles of exclusively OptMinMem, RecExpand, and FullRecExpand. RecExpand performs far better than OptMinMem: it produces strictly fewer I/Os than OptMinMem on 90% of the instances, and on half of them, OptMinMem has a deviation of at least 4%. 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 a deviation larger than 2% over FullRecExpand on only 3% of the instances.

We present the performance profiles for the dataset SmallSynth and the memory bound M_{mid} on Figure 4. In this figure, the deviation is computed using the optimal solution obtained via the ILP, which allows us to analyze the quality of the solutions returned by FullRecExpand and RecExpand. As the left graph shows, the heuristics observe the same hierarchy as in the Synth dataset with larger trees, but as one could expect, the differences of performance are less significant: PostOrderMinIO has a deviation of less than 10% over the optimal solution in 75% of the instances. OptMinMem is non-optimal in 3% of the instances. FullRecExpand and RecExpand achieve better performance as they are non-optimal in respectively 0.72% and 0.74% on the instances. The performance profile on the subset of trees where at least one of these heuristics is non-optimal is presented on the right graph.

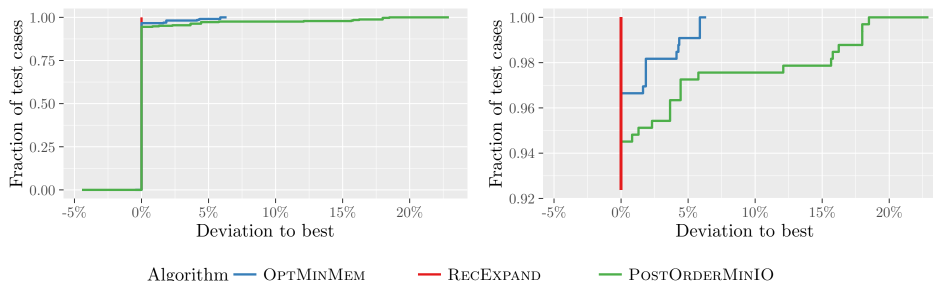


Figure 5. Performance profiles for the complete Trees dataset with the M_{mid} memory bound (left) and zoom on the top part corresponding to instances where the heuristic results differ (right).

The left plot of Figure 5 presents the performance profiles of the three heuristics PostOrderMinIO, RecExpand and OptMinMem for the complete dataset Trees using the memory bound M_{mid}. 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 top part of the same performance profile, corresponding to 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 out-

performed, and OptMinMem performs better than PostOrderMinIO) but with smaller discrepancies between the heuristics. We observe a deviation larger than 5% on only 3% of the instances for PostOrderMinIO and 1% of the instances for OptMinMem.

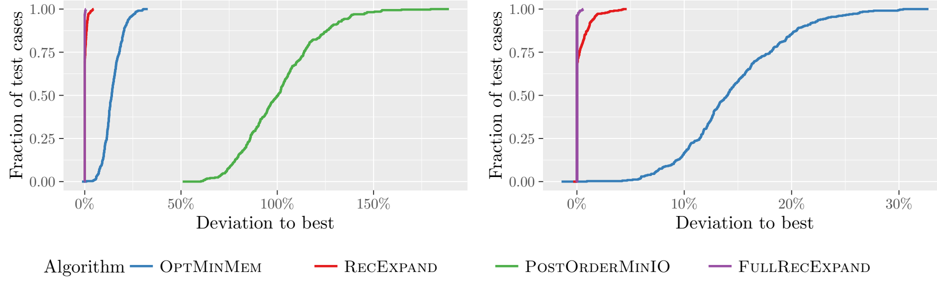


Figure 6. Performance profiles of FullRecExpand, RecExpand, OptMinMem and PostOrderMinIO on the Synth dataset with the M_{min} memory bound (right: same performance profiles without PostOrderMinIO).

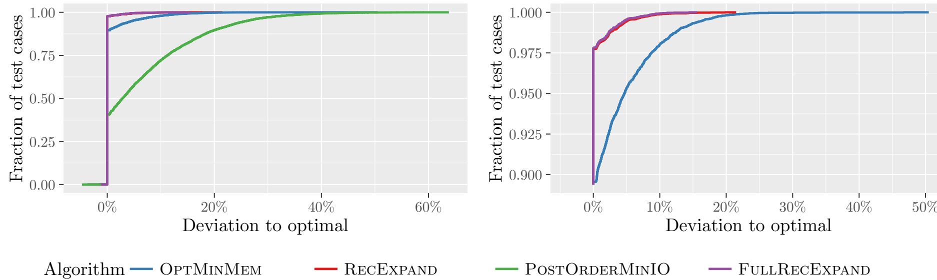


Figure 7. Performance profiles for the SmallSynth dataset with the M_{min} memory bound (right: same performance profiles without PostOrderMinIO, zoom on the top part corresponding to instances where the heuristic results differ).

We now consider the memory bound $M_{min} = LB$, which 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 6, the SmallSynth dataset in Figure 7, and the Trees dataset in Figure 8. The main conclusion that can be made in comparison to the previous results is that the difference between OptMinMem and RecExpand is significantly larger with this memory bound. Indeed, in the Synth dataset, there is a deviation of 10% for OptMinMem in 90% of the cases whereas such a deviation was reached in only 15% of the cases previously. This can be explained by the fact that the memory bound considered here is

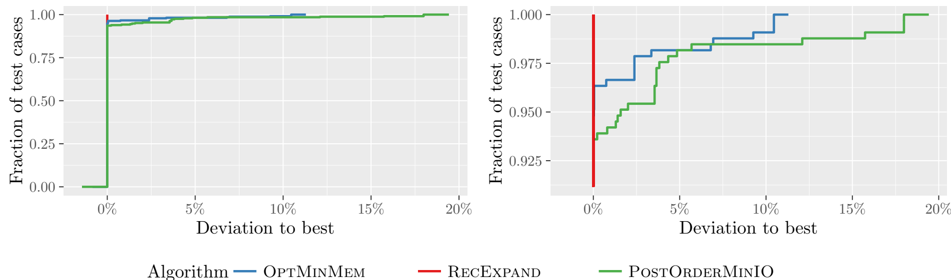


Figure 8. Performance profiles for the complete Trees dataset with the M_{min} memory bound (left) and zoom on the top part corresponding to instances where the heuristic results differ (right).

further from the memory required by MinMem. On the other hand, the difference between PostOrderMinIO and RecExpand is smaller in this case: there is a deviation of 100% 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 8, even if it is less significant. For the SmallSynth dataset, the proportion of non-optimal cases is around 3.5 times larger than with the previous memory bound for the three heuristics FullRecExpand, RecExpand, and MinMem, so they are also further from the optimal, but the modification of the memory bound did not significantly modify the behavior of PostOrderMinIO.

For the sake of completeness, we have also considered the memory bound $M_{max} = Peak_{incore} - 1$, which is the opposite case: the largest memory bound for which I/Os are required in order to compute a tree. With this memory bound, OptMinMem, RecExpand, and FullRecExpand are always equal (and even optimal for the SmallSynth dataset), and only PostOrderMinIO achieves worse performance. This can be explained by the fact that M_{max} is right below the memory required 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 deviation of PostOrderMinIO is smaller than with the other memory bounds.

8. 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 us to optimally solve the problem when all output data have identical size, but that, in the general case, none of them has a constant competitive factor compared to the optimal solution. In addition to an ILP formulation of the problem, which allows us to compute an optimal solution for small trees, we proposed a novel heuristic solution. Through simulations, we show that this new heuristic is very efficient in practice, achieves

better performance than existing solutions, and achieves near optimal performance on small trees. Despite our efforts, the complexity of the problem remains open. Determining this complexity would definitely be a major step, although our findings already lay the basis for more advanced studies. These include moving to parallel out-of-core execution (as was already done for parallel incore execution [10]) as well as designing competitive algorithms for the sequential problem.

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.

Bibliography

[1] E. Agullo, On the out-of-core factorization of large sparse matrices, PhD thesis, École normale supérieure de Lyon, France (2008).

[2] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent and F.-H. Rouet, Robust memory-aware mappings for parallel multifrontal factorizations, *SIAM Journal on Scientific Computing* **38**(3) (2016) C256–C279.

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

[4] 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) (2006) 136–156.

[5] W. G. Aulbur, Parallel implementations of quasiparticle calculations of semiconductors and insulators, PhD thesis, The Ohio State University (1996).

[6] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, *IBM Systems Journal* **5**(2) (1966) 78–101.

[7] J. R. Correa and A. S. Schulz, Single-machine scheduling with precedence constraints, *Mathematics of Operations Research* **30**(4) (2005) 1005–1021.

[8] T. A. Davis, *Direct Methods for Sparse Linear Systems*, Fundamentals of Algorithms,, Vol. 2 (Society for Industrial and Applied Mathematics, 2006).

[9] D. E. Dolan and J. J. Moré, Benchmarking optimization software with performance profiles, *Mathematical Programming* **91**(2) (2002) 201–213.

[10] L. Eyraud-Dubois, L. Marchal, O. Sinnen and F. Vivien, Parallel scheduling of task trees with limited memory, *TOPC* **2**(2) (2015) p. 13.

[11] M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran, Cache-oblivious algorithms, *Foundations of Computer Science, 1999. 40th Annual Symposium on*, IEEE, (IEEE Computer Society, New York, NY, 1999), pp. 285–297.

[12] M. S. Hybertsen and S. G. Louie, Electron correlation in semiconductors and insulators: Band gaps and quasiparticle energies, *Physical Review B* **34**(8) (1986) p. 5390.

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

[14] H. Jia-Wei and H. T. Kung, I/o complexity: The red-blue pebble game, *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, *STOC '81*, (Association for Computing Machinery, New York, NY, USA, 1981), p. 326–333.

[15] 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) (2011) 63–75.

[16] M. Lee, P. Michaud, J. S. Sim and D. Nyang, A simple proof of optimality for the MIN cache replacement policy, *Inf. Process. Lett.* **116**(2) (2016) 168–170.

[17] T. J. Lee and G. E. Scuseria, Achieving chemical accuracy with coupled-cluster theory, *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, ed. S. R. Langhoff (Springer, New York, NY, 1995), pp. 47–108.

[18] J. W. H. Liu, The role of elimination trees in sparse factorization, *SIAM Journal on Matrix Analysis and Applications* **11**(1) (1990) 134–172.

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

[20] J. W. Liu, On the storage requirement in the out-of-core multifrontal method for sparse factorization, *ACM Transactions on Mathematical Software (TOMS)* **12**(3) (1986) 249–264.

[21] E. Mäkinen, Generating random binary trees—a survey, *Information Sciences* **115**(1-4) (1999) 123–136.

[22] J. M. L. Martin, *Benchmark studies on small molecules*, *Encyclopedia of Computational Chemistry, Vol. 1* (Wiley, New York, NY, 1998), New York, NY, pp. 115–128.

[23] P. A. Papp and R. Wattenhofer, On the hardness of red-blue pebble games, *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, (Association for Computing Machinery, 2020), p. 419–429.

[24] 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, *Handbook on Data Centers*, eds. S. U. Khan and A. Y. Zomaya (Springer, New York, NY, 2015), pp. 647–667.

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

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

Appendix A. Omitted proof from Section 4.2

Proof of Theorem 9. 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 Theorem 6, "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 Theorem 5, 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 Theorem 6, we have $l(s) \leq M$ and, hence, no node of $T(s)$ has a label larger than M. Let 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 9 for an illustration of the tree.

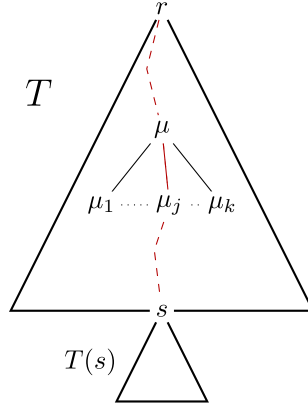


Figure 9. Scheme of the composition of the tree T.

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 equivalents in T.

We define, as in the proof of Theorem 8, 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 indices. 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 Theorem 8, 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. □

Appendix B. Illustration of *FullRecExpand* on some examples

The left-hand side of Figure 10 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 11 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 performs 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.

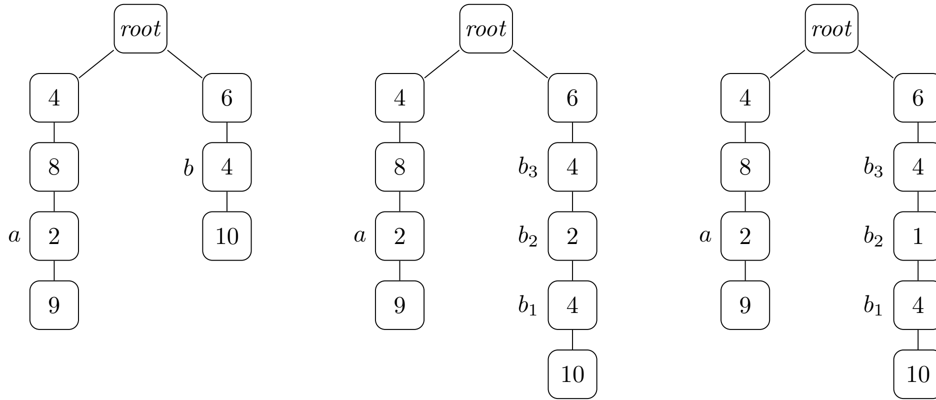


Figure 10. Example where FullRecExpand is optimal whereas OptMinMem and PostOrderMinIO are not. Let $M = 10$. The left tree is the original one, and the others are obtained during the execution of FullRecExpand after the expansion of b.

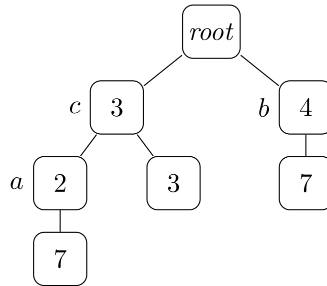


Figure 11. Example where FullRecExpand and OptMinMem are not optimal whereas PostOrderMinIO is. $M = 7$ in this example.