On optimal tree traversals for sparse matrix factorization

Mathias Jacquelin, Loris Marchal, Yves Robert, Bora Uçar

On optimal tree traversals for sparse matrix factorization

Mathias Jacquelin, Loris Marchal, Yves Robert and Bora Uçar
LIP, Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Email: {Mathias.Jacquelin|Loris.Marchal|Yves.Robert|Bora.Ucar}@ens-lyon.fr

LIP Research Report RRLIP2010-30

***Abstract*—We study the complexity of traversing tree-shaped workflows whose tasks require large I/O files. Such workflows typically arise in the multifrontal method of sparse matrix factorization. We target a classical two-level memory system, where the main memory is faster but smaller than the secondary memory. A task in the workflow can be processed if all its predecessors have been processed, and if its input and output files fit in the currently available main memory. The amount of available memory at a given time depends upon the ordering in which the tasks are executed. What is the minimum amount of main memory, over all postorder schemes, or over all possible traversals, that is needed for an in-core execution? We establish several complexity results that answer these questions. We propose a new, polynomial time, exact algorithm which runs faster than a reference algorithm. Next, we address the setting where the required memory renders a pure in-core solution unfeasible. In this setting, we ask the following question: what is the minimum amount of I/O that must be performed between the main memory and the secondary memory? We show that this latter problem is NP-hard, and propose efficient heuristics. All algorithms and heuristics are thoroughly evaluated on assembly trees arising in the context of sparse matrix factorizations.**

***Keywords*-Sparse matrix factorization, Multifrontal method, Assembly tree, Tree traversal, Postorder tree traversal, I/O minimization.**

I. Introduction

We consider the following memory-aware traversal problem for rooted trees. The nodes of the tree correspond to tasks, and the edges correspond to the dependencies among the tasks. The dependencies are in the form of input and output files: each node accepts a large file as input, and produces a set of large files, each of them to be accepted by a different child node. We are to execute such a set of tasks on a two-level memory system. The execution scheme corresponds to a traversal of the tree where visiting a node translates into reading the associated input file and producing output files. How one can traverse the tree so as to optimize memory usage? For convenience we refer to the two-levels of storage as the *main memory* and the *secondary memory*, and also as *in-core* and *out-of-core*. Many combinations such as cache and RAM, or RAM and disk, or even disk and tape, lead to the same association of a faster but smaller storage device together with a larger but slower device. The difficulty remains the same for all combinations: find an execution scheme that makes the best use of the *main memory*, and minimizes accesses to the *secondary memory*.

Throughout the paper, we consider *out-trees* where a task can be executed only if its parent has already been executed. However, we show in Section III that all results equivalently apply to *in-trees*, where tasks are processed from the leaves up to the root. Each task (or node) i in the tree is characterized by the size f_i of its input file (data needed before the execution and received from its parent), and by the size n_i of its execution file.

During execution, non-leaf nodes generate several output files, one for each child, which can have different sizes. A task can be processed only in-core; its execution is feasible only if all its files (input, output, and execution) fit in currently available memory. More formally, let M be the size of the main memory, and S the set of files stored in this memory when the scheduler decides to execute task i. Note that S must contain the input file of task i. The processing of task i is possible if we have:

$$MemReq(i) = f_i + n_i + \sum_{j \in Children(i)} f_j \leq M - \sum_{j \in S, j \neq i} f_j \quad (1)$$

where $MemReq(i)$ denotes the memory requirement of task i (and $Children(i)$ its child nodes in the tree). Once i has been executed, its input file and execution file can be discarded, and replaced by other files in main memory; the output files can either be kept in main memory, in order to execute some child of the task, or they can temporarily be stored into secondary memory (and retrieved later when the scheduler decides to execute the corresponding child of i). The volume of accesses (reads or writes) to secondary memory is referred to as the *I/O volume*.

Clearly, the traversal, i.e., the order chosen to execute the tasks, plays a key role in determining which amount of main memory and I/O volume are needed for a successful execution of the whole tree. More precisely, there are two main problems which the scheduler must address:

MinMemory Determine the minimum amount of main memory that is required to execute the tree without any

access to secondary memory.

MINIO Given the size M of the main memory, determine the minimum I/O volume that is required to execute the tree.

Obviously, a necessary condition for the execution to be successful is that the size M of main memory exceeds the largest memory requirement over all tasks:

$$\max_i MemReq(i) \leq M$$

However, this condition is not sufficient, and a much larger main memory size may be needed for the MINMEMORY problem.

The main motivation for this work comes from numerical linear algebra. Tree workflows (assembly or elimination trees) arise during the factorization of sparse matrices, and the huge size of the files involved makes it absolutely necessary to reduce the memory requirement of the factorization. The trees arising in this context are in-trees (as said before, and as we will discuss later, there is no difference between in-trees and out-trees). We build upon two key results from the literature [1], [2]. Liu [1] discusses how to find the best traversal for the MINMEMORY problem when the traversal is required to correspond to a postorder traversal of the tree. In the follow-up study [2], an exact algorithm is proposed to solve the MINMEMORY problem, without the postorder constraint on the traversal.

In this paper, we propose a new exact algorithm called $MinMem$ for the MINMEMORY problem. The $MinMem$ algorithm is based upon a novel approach that systematically explores the tree with a given amount of memory. This approach is quite different from the techniques used in [2]. Although the worst-case complexity of the proposed $MinMem$ algorithm is the same as that of Liu's, i.e., quadratic in the number of nodes in the tree, it turns out that it is much more efficient in practice, as demonstrated by our experiments with elimination trees arising in sparse matrix factorization (see Section VI for details). We also compare $MinMem$ and Liu's algorithms with the best postorder traversal (common in sparse matrix factorization packages), in terms of both quality (memory needed) and execution time. We report that the best postorder traversals result in only a little additional memory requirement than the optimal one in practice, which is good news for the current sparse matrix factorization libraries. However, we show that there exist trees where postorder based traversals require arbitrarily larger amounts of main memory than the optimal one.

As for the MINIO problem, we show that it is NP-hard, both for postorder based and for arbitrary traversals, even for simple harpoon graphs, while it is polynomial for arbitrary trees with unit-size files (in which case MINIO reduces to the *I/O pebble game* introduced by Hong and Kung [3]). This shows that introducing files of different sizes does add a level of difficulty in memory-aware scheduling of tree workflows. We provide a set of heuristics to solve the MINIO problem. Our heuristics use various greedy criteria to select the next node to be scheduled, and those files to be temporarily written to secondary memory. All these heuristics are evaluated using assembly trees arising in sparse matrix factorization methods.

The paper is organized as follows. We start with an overview of related work in Section II. Then we describe the framework in Section III. The next three sections constitute the heart of the paper. We deal with the MINMEMORY problem in Section IV, presenting complexity results for postorder traversals and proposing the exact $MinMem$ algorithm. Then we consider the MINIO problem in Section V, assessing the NP-hardness of this problem, and designing heuristics. The experimental evaluation of all MINMEMORY algorithms and MINIO heuristics is conducted in Section VI. Finally we provide some concluding remarks and hints for future work in Section VII.

II. BACKGROUND AND RELATED WORK

A. Elimination tree and the multifrontal method

As mentioned above, determining a memory-efficient tree traversal is very important in sparse numerical linear algebra. The elimination tree is a graph theoretical model that represents the storage requirements, and computational dependencies and requirements, in the Cholesky and LU factorization of sparse matrices. Here we give a brief description of such trees; we refer to [4] for the first formalization of elimination trees, and to [5] for an excellent survey on the subject.

There are at least two interpretations of elimination trees [5]. Among those, the one describing the dependencies of numerical values among the columns of the Cholesky factor serves well for our purposes in this paper. Assume that A is an $n \times n$ sparse, symmetric, positive definite matrix with a lower triangular Cholesky factor L such that $A = LL^T$. It is known that for $i > j$, the numerical values of column i of L depend on column j of L if and only if $\ell_{ij} \neq 0$. Consider building a directed graph on n vertices with edges representing the column dependencies, i.e., we add an edge from the vertex v_j to the vertex v_i whenever the column i of L depends on the column j. The transitive reduction (if there is a directed path of length at least two from v_j to v_i, then the edge (v_j, v_i) is discarded) of this graph yields the elimination tree. Given such a model, it is clear that the column i of L can only be computed after all the columns corresponding to the children of v_i in the elimination tree.

The multifrontal method of sparse matrix factorization [6], [7] organizes the computations of sparse factorizations as a sequence of dense matrix operations using the elimination tree. The method associates a block 2×2 matrix with each node of the elimination tree—the block matrix contains a diagonal element and the nonzeros in the corresponding row and column of the matrix currently being eliminated. The $(1, 1)$-block of a node can be eliminated (it is called

fully summed) only if all the updates to the corresponding diagonal entry have been computed. The Schur complement formed by the elimination of the fully summed variable on the $(2,2)$-block of a node cannot be eliminated until later in the factorization. This Schur complement is called the contribution block, and it is passed to the father node for the *assembly* operation. Therefore the operations that are at the heart of the multifrontal method are as follows. The first one is to assembly the contribution blocks from the children nodes, and the original entries from the matrix (if we are at a leaf, there is no contribution block); the second one is to eliminate the fully summed variable; and the third one is to compute and send the contribution block to the father. This leads to an in-tree where the computations proceed from the leaves to the root.

Since the elimination tree is defined with one variable (row/column) per node, it only allows one elimination per node and the (1,1) block would be of order one. Therefore, there would be insufficient computation at a node for efficient implementation. It is thus advantageous to combine or amalgamate nodes of the elimination tree. The amalgamation can be restricted so that two nodes of the elimination tree are amalgamated only if the corresponding columns of the L factor have the same structure below the diagonal [6]. As even this technique may not give a large enough (1,1) block, a threshold based amalgamation strategy can be used in which the columns to be amalgamated are allowed to have discrepancies in their patterns [8]. The resulting tree is often called the assembly tree.

B. Pebble game and its variants

On the more theoretical side, this work builds upon the many papers that have addressed the pebble game and its variants. The MINMEMORY problem amounts to revisiting the I/O pebble game with pebbles of arbitrary sizes that must be loaded into main memory before *firing* (executing) the task. The pioneering work of Sethi and Ullman [9] deals with a variant of the pebble game that translates into the simplest instance of MINMEMORY, with $f_i = 1$ and $n_i = 0$ for any task i. The concern in [9] was to minimize the number of registers that must be used while computing an arithmetic expression. The problem of determining whether a general DAG can be executed with a given number of pebbles has been shown NP-hard by Sethi [10] if no vertex is pebbled more than once (the general problem allowing recomputation, that is, re-pebbling a vertex which have been pebbled before, has been proven PSPACE complete by Gilbert, Lengauer and Tarjan [11]). However, this problem has a polynomial complexity for tree-shaped graphs [9].

A variant of the game with two levels of storage has been introduced by Hong and Kung [3] under the name of *I/O pebble game*, which was used to derive lower bounds on I/O operations and study the trade-off between I/O operation and main memory size for particular graphs. A comprehensive summary of results for pebble games can be found in the book by Savage [12].

In [9], the algorithm proposed by Sethi and Ullman for processing tree-shaped graphs and minimizing the number of allocated registers also has a minimum number of *store* instructions, which makes it optimal both for memory and for I/O minimization. It is quite interesting to see that the classical pebble game problem with trees remains polynomial with files of arbitrary sizes instead of pebbles (this is the MINMEMORY problem,) while the I/O pebble game becomes NP-hard (see Theorem 2).

On the application side, there are many variants of MINMEMORY, some of which being discussed in Section III-C. The execution model summarized by Equation (1) applies to a large variety of scenarios, including divide-and-conquer algorithms. For high-degree trees, simultaneously loading all children files into main memory may be a bottleneck requirement. While some applications could allow for processing the children one after the other, like in map-reduce problems, other scenarios call for generating all children data concurrently. Along the same line, a relaxation of the MINIO problem would allow to write *fractions* of files into secondary memory, leading to a *divisible* version of the problem. Again, while this may make sense in some cases (e.g., when the main memory is naturally divided into small pages, and if it is possible to unload some pages containing fractions of files), it is not always possible (e.g., when the main memory is a complex file system).

III. MODELS AND PROBLEMS

A. Application model

The tree workflow $\mathcal{T}$ is composed of p nodes, or tasks, numbered from 1 to p. Nodes in the tree have an input file, an execution file (or program), and several output files (one per child). More precisely:

- Each node i has an input file of size f_i. If i is not the root, its input file is produced by its parent $parent(i)$; if i is the root, its input file can be of size zero, or contain input from the outside world.
- Each node i in the tree has an execution file of size n_i.
- Each non-leaf node i in the tree, when executed, produces a file of size f_j for each $j \in Children(i)$. Here $Children(i)$ denotes the set of the children of i. If i is a leaf-node, then $Children(i) = \emptyset$ and i produces a file of null size: we then consider that the terminal data produced by leaves are directly written to the secondary memory or sent to the outside world, independently from the I/O mechanism.

The memory requirement $MemReq(i)$ of node i is the total amount of main memory that is needed to execute node i, as underlined in Equation (1). After i has been processed, its input file and program can be discarded, while its output files can either be kept in main memory (to process the children of i) or be stored in secondary memory temporarily.

Algorithm 1: Checking an in-core traversal.

Input: tree $\mathcal{T}$ with p nodes, available memory M, ordering σ of the nodes
Output: whether the traversal is feasible

$Ready \leftarrow \{root\}$
$M_{avail} \leftarrow M - f_{root}$
for $step = 1$ *to* p **do**
  Let i be the task such that $\sigma(i) = step$
  if $i \notin Ready$, *or* $MemReq(i) > M_{avail} + f_i$ **then**
    return FAILURE
  $M_{avail} \leftarrow M_{avail} + f_i - \sum_{j \in Children(i)} f_j$
  $Ready \leftarrow Ready \setminus \{i\} \cup \bigcup_{j \in Children(i)} \{j\}$
return SUCCESS

B. In-core traversals and the MINMEMORY problem

For the MINMEMORY problem, we are given a tree $\mathcal{T}$ with p nodes and an initial amount of memory M. A traversal is an ordering of the p nodes that specifies at which step whey are executed. A traversal must obey precedence constraints (a node is always scheduled after its parent) and must never exceed the available memory. Algorithm 1 checks if a given traversal is feasible: it computes the memory M_{avail} that is available at each step, which corresponds to the original memory M minus the size of the files of ready nodes (nodes which are not executed yet, but whose parents have been processed). A formal definition of a traversal is given below.

Definition 1 (INCORETRAVERSAL)**.** *Given a tree $\mathcal{T}$ and a amount M of available memory, the problem* INCORETRAVERSAL$(\mathcal{T}, M)$ *consists in finding a feasible in-core traversal σ described by a permutation of the nodes of a tree $\mathcal{T}$ such that:*

$$\forall i \neq root, \quad \sigma(parent(i)) < \sigma(i) \tag{2}$$

$$\forall i, \quad \sum_{\sigma(j)<\sigma(i)} \left(\sum_{k \in Children(j)} f_k - f_j \right) + n_i + \sum_{k \in Children(i)} f_k \leq M \tag{3}$$

In this definition, Equation (2) accounts for precedence constraints and Equation (3) deals with memory constraints. A postorder traversal is a traversal where nodes are visited according to some top-down postorder ordering of the tree nodes. Hence, in a postorder traversal, after processing a vertex i, the whole subtree rooted in i is completely processed.

Definition 2 (MINMEMORY)**.** *Given a tree $\mathcal{T}$, determine the minimum amount of memory M such that* INCORETRAVERSAL$(\mathcal{T}, M)$ *has a solution.* MINMEMORYPOSTORDER *is the same problem restricted to postorder traversals.*

C. Model variants

In this section, we discuss three variants of the model.

Bottom-up traversals for in-trees: Let $\mathcal{T}$ be an in-tree with p nodes and M the amount of main memory. As the tasks have to be executed from the leaves to the root, a task now has many input files and a single output file. We do not change the notations and assume that the output file has size f_i (*to* the parent, instead of *from* the parent in an out-tree), and the input files have the size f_j for each child j of i. A valid traversal of such an in-tree should respect the order of the tasks (from the leaves to the root) and should satisfy Equation (1) for each task i. Suppose $\sigma(\mathcal{T}, p, M)$ is a valid traversal of the in-tree $\mathcal{T}$. Then $\widetilde{\sigma}(\widetilde{\mathcal{T}}, p, M)$ is a valid traversal of the out-tree $\widetilde{\mathcal{T}}$ where $\widetilde{\sigma}$ denotes the reverse permutation of σ, defined as $\widetilde{\sigma}(i) = p - \sigma(i) + 1$ for all i. This is easy to verify as the reverse permutation guarantees the order of the tasks for $\widetilde{\mathcal{T}}$, and the memory constraint is satisfied for any task. The relation between a valid traversal of an in-tree $\mathcal{T}$ and the inverted traversal of the out-tree $\widetilde{\mathcal{T}}$ holds for the other way round too.

Model with replacement: In some variants of the pebble game, the player is allowed to move a pebble from one pebbled node to an unpebbled node. Extending the game to pebbles with non-unit costs, this amounts to the variant of the model where the memory occupied by the input file of node i (of size f_i) is *replaced* by the memory occupied by the output files of node i (of size $\sum_{j \in Children(i)} f_j$). The amount of memory needed to process node i is $\max(f_i, \sum_{j \in Children(i)} f_j)$ (note that in the pebble game, there is no cost n_i). This variant can be simulated by our model as follows: given an instance of the problem with the replacement policy, we add a negative weight $n_i = -\min(f_i, \sum_{j \in Children(i)} f_j)$ to each node i (an example is given in Figure 1).

Model with replacement: $\max(f_i, \sum_{j \in Children(i)} f_j) \leq M$ — Current model: $f_i + n_i + \sum_{j \in Children(i)} f_j \leq M$

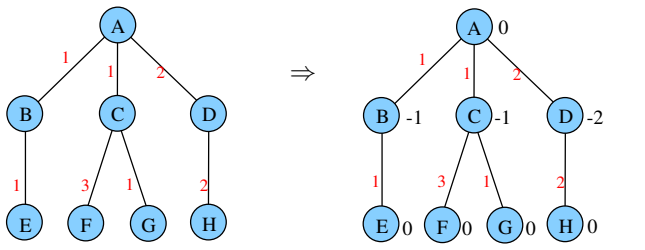


Figure 1. Transformation from the model with replacement.

Liu's model: In [2], the author introduces a bottom-up framework modelling sparse matrix LU factorization.

In this framework, the tree T modelling the application is modified as follows: each original node x of T is expanded in two nodes x^+ and x^-. Here x^+ represents x during the processing of a column, x^- being x after its processing. Note that in this model, parameter f_x is not used.

The cost n_{x^+} associated to node x^+ represents the number of nonzeros in columns of the matrix L from the subtree of T rooted in x, that are required during the processing of the column x in the factorization: in other words the memory peak associated to node x. Conversely, the cost n_{x^-} is the number of nonzeros in columns of matrix L associated with the subtree of T rooted in x that are still required after the processing of x, which is the storage requirement of the subtree of T rooted in x.

This variant can be simulated in our framework by merging back each pair of nodes (i^+,i^-) into node i, with an input file of size $f_i = n_{i^-}$ and an extra memory cost during processing $n_i = n_{i^+} - n_{i^-} - \sum_{j^- \in Children(i^+)} n_{j^-}$. An example is given in Figure 2.

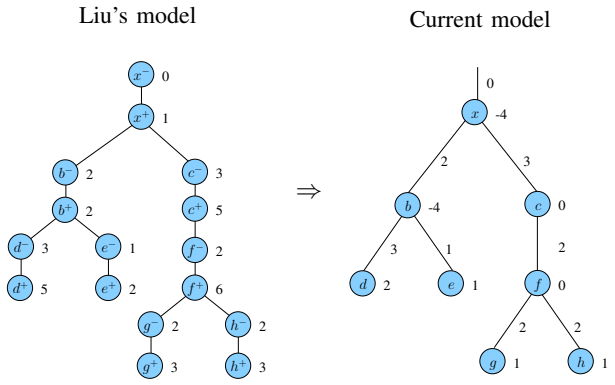


Figure 2. Reduction for Liu's model.

D. Out-of-core traversals and the MINIO problem

Out-of-core processing enables solving large problems, when the size of the data cannot fit into the main memory. In this case, some temporary data are copied into the secondary memory, and unloaded from the main memory, so as to leave room for other computations. Since secondary memory has a smaller access rate, the usual objective is to limit the volume of I/O operations.

Defining traversals that perform I/O operations is more complicated than defining in-core traversals: in addition to determining the ordering of the nodes (the permutation σ), at each step we have to identify which files are written into secondary memory (if necessary). When a task i is scheduled for execution but its input file was moved to secondary memory, that file must be read and loaded back into the main memory before processing task i. Thus, a given file is written at most once in the main memory. The ordering of the I/O operations is done via a second function τ, such that $\tau(i)$ is the step when the input file of task i (of size f_i) should be moved to secondary memory ($\tau(i) = \infty$ means that this file is never moved to the secondary memory).

Algorithm 2: Checking an out-of-core traversal.

Input: tree $\mathcal{T}$ with p nodes, available memory M, ordering σ of the nodes, ordering τ of the output transfers to secondary memory
Output: whether the traversal is feasible, and the amount of I/O

$Ready \leftarrow \{root\}$
$M_{avail} \leftarrow M - f_{root}$
$IO \leftarrow 0$
$Written \leftarrow \emptyset$
for $step = 1$ *to* p **do**
  foreach i *such that* $\tau(i) = step$ **do**
    if $\sigma(i) \geq step$ **then**
      return FAILURE
    $Written \leftarrow Written \cup \{i\}$
    $M_{avail} \leftarrow M_{avail} + f_i$
    $IO \leftarrow IO + f_i$
  Let i be the task such that $\sigma(i) = step$
  if $i \in Written$ **then**
    $Written \leftarrow Written \setminus \{i\}$
    $M_{avail} \leftarrow M_{avail} - f_i$
  if $i \notin Ready$, *or* $MemReq(i) > M_{avail} + f_i$ **then**
    return FAILURE
  $M_{avail} \leftarrow M_{avail} + f_i - \sum_{j \in Children(i)} f_j$
  $Ready \leftarrow Ready \setminus \{i\} \cup \bigcup_{j \in Children(i)} \{j\}$
return (SUCCESS, IO)

Algorithm 2 is used to check whether an out-of-core traversal is feasible. It makes use of $Written$, the set of files that have been moved to secondary memory. Similarly to the in-core case, M_{avail} denotes the memory which is available at a current step, and $Ready$ the set of ready nodes. The algorithm also computes IO, the total amount of data transferred from/to main memory. Note that each data written (once) to the secondary memory is read only once. At each step, the algorithm checks that the files written to secondary memory have been produced earlier, that precedence constraints are satisfied, and that there is enough memory to process the chosen node. More formally, a valid out-of-core traversal can be defined as follows.

Definition 3 (OUTOFCORETRAVERSAL)**.** *Given a tree $\mathcal{T}$ and a fixed amount of main memory M, the problem* OUTOFCORETRAVERSAL$(\mathcal{T}, M)$ *consists in finding an out-of-core traversal, described by a permutation σ of the nodes in $\mathcal{T}$ (corresponding to the schedule of computations), and a function $\tau : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\} \cup \{\infty\}$*

(corresponding to the schedule of I/O operations), such that:

$$\forall i \neq root, \quad \sigma(parent(i)) < \sigma(i) \tag{4}$$

$$\forall i \neq root, \quad \sigma(parent(i)) < \tau(i) \tag{5}$$

$$\forall i \neq root, \quad \text{if } \tau(i) < \infty \text{ , then } \tau(i) < \sigma(i) \tag{6}$$

$$\forall i, \quad \sum_{\sigma(j)<\sigma(i)} \left(\sum_{k \in Children(j)} f_k - f_j \right) - \sum_{\substack{\tau(j)<\sigma(i) \\ \sigma(j)>\sigma(i)}} f_j + n_i + \sum_{k \in Children(i)} f_k \leq M \tag{7}$$

Then the amount of data written in secondary memory is given by

$$IO = \sum_{\tau(i) \neq \infty} f_i$$

In Equation (7), the term $\sum_{\substack{\tau(j)<\sigma(i) \\ \sigma(j)>\sigma(i)}} f_j$ corresponds to the files that have been written into secondary memory at step $\sigma(i)$. We now define the MINIO problem, which asks for an out-of-core traversal with the minimum amount of I/O volume.

Definition 4 (MINIO)**.** *Given a tree $\mathcal{T}$, and a fixed amount of main memory M, determine the minimum I/O volume IO needed by a solution of* OUTOFCORETRAVERSAL$(\mathcal{T}, M)$.

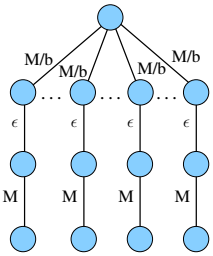
IV. THE MINMEMORY PROBLEM

In this section, we present algorithms for the MINMEMORY problem. We first present the best possible postorder traversal, and show that its performance can be arbitrarily bad. Then we propose an alternative to the optimal algorithm introduced by Liu [2].

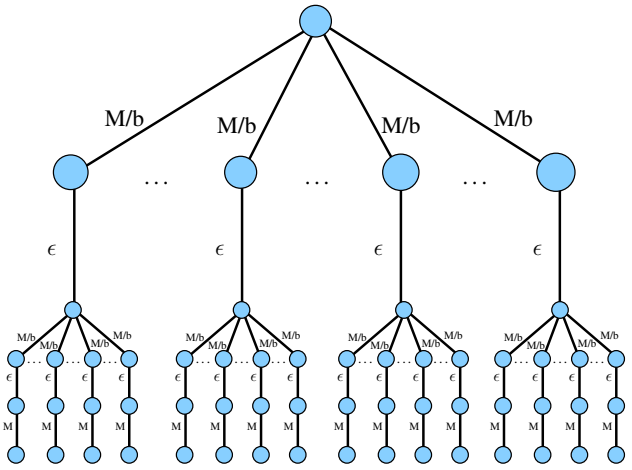
A. Postorder traversals

Postorder traversals are very natural for the MINMEMORY problem, and they are widely used in sparse matrix software like `MUMPS` [13], [14]. Liu [1] has characterized the best postorder traversal, leading to a fast but sub-optimal solution for MINMEMORY. In a nutshell, the best postorder is obtained by guaranteeing that in the resulting order, the children of a node are listed in the increasing order of the memory requirement of their respective subtrees. The algorithm is called $PostOrder$. In another study, Liu [2] has also provided an optimal algorithm for MINMEMORY whose worst case execution time is $O(p^2)$, where p is the number of tree nodes. The algorithm that finds the best postorder runs in $O(p \log(p))$ time, which calls for a tradeoff between speed and performance. But while postorder traversals are widely used in practice, their efficiency has never been thoroughly assessed (to the best of our knowledge). We now show that the best postorder may require arbitrarily more main memory than the optimal traversal.

Theorem 1. *Given any arbitrarily large integer K, there exist trees for which the best postorder traversal requires at least K times the amount of main memory needed by the optimal traversal for* MINMEMORY.



(a) One level.



(b) Two levels.

Figure 3. First levels of the graph for the proof of Theorem 1. Here b is the number of children of the nodes with more than one child.

Proof: Consider the harpoon graph with b branches in Figure 3(a). All branches are identical and all tasks have a zero length execution file. Any postorder traversal requires an amount of $M + \varepsilon + (b-1)M/b$ main memory, while the optimal traversal (which alternates between branches) only requires $M_{\min} = M + \varepsilon$. Now replace each leaf by a copy of the harpoon graph, as shown in Figure 3(b). The value of $M_{\min}$ is unchanged, while a postorder traversal requires $M + \varepsilon + 2(b-1)M/b$. Iterating the process K times leads to the desired result. ■

B. The Explore and MinMem algorithms

Liu [2] proposes an algorithm for MINMEMORY which is optimal among all possible traversals, not only postorder ones. It is a recursive bottom-up traversal of the tree which, at each node of the tree, combines the optimal traversals built for all subtrees. The combination is based on the notion of *Hill-Valley Segments* and requires some sophisticated

multi-way merging algorithm, in order to reach the $O(p^2)$ complexity. In this section, we introduce *MinMem*, another exact algorithm which proceeds top-down and maintains the best reachable cut of the tree at each step. While the worst-case complexity of *MinMem* is the same as Liu's exact algorithm, it runs faster in practical cases resulting from multifrontal methods (see Section VI). The *MinMem* algorithm is based on an advanced tree exploration routine: the *Explore* algorithm.

Algorithm 3: *Explore* $(T, i, M^{\text{avail}}, L_{\text{init}}, Tr_{\text{init}})$

Input: tree T, root i of the subtree to explore, available memory M^{avail}, initial set of nodes L_{init}, initial traversal Tr_{init}
Output: $\langle M_i, L_i, Tr_i, M_i^{\text{peak}} \rangle$, where:
M_i: the minimum memory requirement in the subtree rooted in i, reachable with memory M,
L_i: set of input files related to M_p,
Tr_i: the traversal from node i to L
M_i^{peak}: minimum memory to be able to visit a new node

if *node i is a leaf and* $n_i + f_i \leq M^{\text{avail}}$ **then**
  return $\langle 0, \emptyset, [i], \infty \rangle$
if $n_i + f_i + \sum_{j \in Children(i)} f_j > M^{\text{avail}}$ **then**
  $M_i^{\text{peak}} \leftarrow n_i + f_i + \sum_{j \in Children(i)} f_j$
  return $\langle \infty, \emptyset, [\,], M_i^{\text{peak}} \rangle$
if $L_{\text{init}} \neq \emptyset$ **then**
  $L_i \leftarrow L_{\text{init}}$
  $Tr_i \leftarrow Tr_{\text{init}}$
else
  $L_i \leftarrow Children(i)$
  $Tr_i \leftarrow [i]$
$Candidates \leftarrow L_i$
while $Candidates \neq \emptyset$ **do**
  foreach $j \in Candidates$ **do**
    $\langle M_j, L_j, Tr_j, M_j^{\text{peak}} \rangle \leftarrow$ $Explore(T, j, M^{\text{avail}} - \sum_{k \in L_i \setminus \{j\}} f_k, \emptyset, \emptyset)$
    `/* Process j */`
    if $M_j \leq f_j$ **then**
      $L_i \leftarrow L_i \setminus \{j\} \cup L_j$
      $Tr_i \leftarrow Tr_i \oplus Tr_j$ `/* append traversal` Tr_j `to the end of` Tr_i `*/`
  $Candidates \leftarrow$ $\left\{ j \in L_i \text{ such that } M^{\text{avail}} - \sum_{k \in L_i \setminus \{j\}} f_k \geq M_j^{\text{peak}} \right\}$
$M_i \leftarrow \sum_{j \in L_i} f_j$
$M_i^{\text{peak}} \leftarrow min_{j \in L_i} \left(M_j^{\text{peak}} + \sum_{k \in L_i \setminus \{j\}} f_k \right)$
return $\left\langle M_i, L_i, Tr_i, M_i^{\text{peak}} \right\rangle$

The *Explore* algorithm requires a tree T, a node i to start the exploration, and an amount of available memory M_{avail}. The last two parameters (L_{init} and Tr_{init}) are optional, and useful to speed-up the algorithm by avoiding the repeated exploration of some parts of the tree. With these parameters, the algorithm computes the minimal memory consumption that can be reached. If the whole tree can be processed, then the minimal memory is zero. Otherwise, the algorithm stops before reaching the bottom of the tree, because some parts of the tree require more memory than what is available. In this case, the state with minimal memory corresponds to a *cut* in the tree: some subtrees are not yet processed, and the input files of their root nodes are still stored in memory. The *Explore* algorithm outputs the cut with minimal memory occupation, as well as a possible traversal to reach this state with the provided memory. In the case where the whole subtree cannot be executed, it also gives the minimum amount of memory (called memory *peak*) which is needed to explore an additional node in the subtree.

When called on a node i, the algorithm first checks if the current node can be executed. If not, the algorithm stops and returns the current requirement as memory peak. Otherwise, it recursively proceeds in its subtree. The optimal cut is initialized with its children, and iteratively improved. All the nodes in the cut are explored: if the cut L_j found in the subtree of a child j has a smaller memory occupation than the child itself, the cut is updated by removing child j, and by adding the corresponding cut L_j. When no more nodes in the cut can be improved (which is easily tested using their respective memory peak), then the algorithm outputs the current cut.

Algorithm 4: *MinMem* (T)

Input: tree T
Output: minimum memory M needed to process the whole tree, traversal Tr

$M^{\text{peak}} \leftarrow \max_{i \in T} MemReq(i)$ `/* lower bound */`
$M^{\text{avail}} \leftarrow 0$
$L \leftarrow \emptyset$
$Tr \leftarrow [\,]$
while $M^{\text{peak}} < \infty$ **do**
  $M^{\text{avail}} \leftarrow M^{\text{peak}}$
  $\langle M, L, Tr, M^{\text{peak}} \rangle \leftarrow$ $Explore(T, root, M^{\text{avail}}, L, Tr)$
return $\langle M^{\text{avail}}, Tr \rangle$

The *Explore* algorithm can be used to check whether a given tree can be processed using a given memory. If not, it provides a refined lower bound on the necessary memory. The *MinMem* algorithm makes use of this information to solve the MINMEMORY problem.

To assess the complexity of the MINMEMORY problem, we consider the moment when each node is first visited

by *Explore*, that is, for each node i, the first call on $Explore(T, i)$. There are p such events, which we denote as $F_1, \ldots, F_p$. We observe that between two such events, no node is visited more than twice by *Explore*. Firstly, in *Explore*, a subtree is re-visited only if the available memory is larger than its peak, which induces that a new node will be visited. Secondly, *MinMem* asks *Explore* to re-visit the whole tree with its peak value, which similarly leads to visit a new node. Thus, there are at most $2p$ calls to *Explore* between two events F_i and F_{i+1}. Altogether, the overall complexity of the algorithm is $O(p^2)$.

V. The MinIO Problem

Contrarily to MinMemory, the MinIO problem turns out to be combinatorial. The difficulty goes beyond finding the best traversal. Indeed, even when the traversal is given, it is hard to determine which files should be transferred into secondary memory at each step.

A. NP-completeness

We prove that the following three variants of the problem are NP-complete.

Theorem 2. *Given a tree $\mathcal{T}$ with p nodes, and a fixed amount of main memory M, consider the following problems:*
(i) given a postorder traversal σ of the tree, determine the I/O schedule so that the resulting I/O volume is minimized,
(ii) determine the minimum I/O volume needed by any postorder traversal of the tree,
(iii) determine the minimum I/O volume needed by any traversal of the tree.
The (decision version of) each problem is NP-complete.

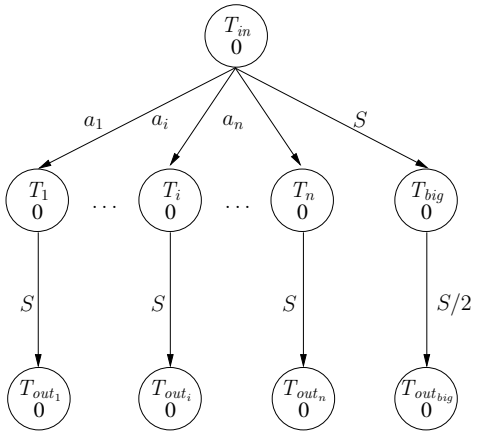


Figure 4. Graph corresponding to $Inst_2$ in the proof of Theorem 2.

Note that (iii) is the original MinIO problem. Also note that the NP-completeness of (i) does not a priori imply that of (ii), because the optimal postorder traversal could have a particular structure. The same comment applies for (ii) not implying (iii).

Proof: We use the same reduction for the three problems, which clearly all belong to NP. Consider an instance $Inst_1$ of 2-Partition [15], with n integers $\{a_1, a_2, \ldots, a_n\}$. The instance $Inst_2$, common for all three problems, consists in the harpoon graph depicted on Figure 4, with $2n+3$ nodes. We let $M = 2S$, which is the largest memory requirement of a node (the root node T_{in}). We let the I/O bound be $IO = S/2$. The construction of $Inst_2$ is clearly polynomial in the size of $Inst_1$.

Note that all traversals are postorder traversals. Any traversal must start with the root T_{in}. After it has been processed, $2S$ units of memory are occupied. In order to process the rest of the tree, one has two main choices:

- either execute one of the n tasks T_i first, with $1 \leq i \leq n$. This requires loading the output file of T_i of size S into main memory, hence to transfer some files whose total sizes are at least S into secondary memory. This violates the I/O bound.
- either execute task T_{big} first. This requires to load its output file of size $S/2$ into main memory, hence to transfer some files whose total sizes are at least $S/2$ into secondary memory.

For (ii) and (iii), the reduction from *2-Partition* goes as follows. Suppose first that $Inst_1$ has a solution, with $\sum_{i \in I} a_i = \sum_{i \notin I} a_i = S/2$. Then, after unloading files of size a_i with $i \in I$ (thus increasing the available memory by $S/2$), one is able to process the entire branch of T_{big} up to the root. This means $Inst_2$ has a solution.

Suppose now that $Inst_2$ has a solution. That means that some files were unloaded in order to process T_{big}. The amount of memory to free is at least $S/2$. It is also at most $S/2$ so as to meet the bound IO. Therefore, exactly $S/2$ units of memory were unloaded to be able to host T_{big}. If I is the set of the unloaded files, then $\sum_{i \in I} a_i = S/2$, which means that $Inst_1$ has a solution and concludes our proof.

Now for (i), take any ordering of the nodes σ which executes T_{big} just after the root task T_{in}. The proof is the same, independently of the rest of the ordering. ■

B. Heuristics

The NP-completeness of problem (i) in Theorem 2 shows that it is difficult to select which files to unload to secondary memory, even when the traversal is given. We introduce six heuristics that greedily choose such files. In the following, j denotes the next node to be processed. First, if f_j has been previously unloaded, it must be stored back into main memory. Then an amount of $MemReq(j) - f_j$ of main memory must be available to execute node j. Let M^{avail} be the currently available memory. If $MemReq(j) - f_j \leq M^{\text{avail}}$, then node j can be processed without I/O. Otherwise, we have to unload a volume $IOReq(j) = M^{\text{avail}} - (MemReq(j) - f_j)$. In that case, we order the set $S = \{f_{i_1}, f_{i_2}, \ldots, f_{i_j}\}$ of the

files already produced and still residing in main memory, so that $\sigma(i_1) > \sigma(i_2) > \cdots > \sigma(i_j)$. Hence f_{i_1} is the file which will be used at the latest iteration in the traversal, and so on. We greedily select the first files from S according to various criteria which we describe below.

Last Scheduled Node First (LSNF): We select the first files from S until their total size is at least $IOReq(j)$. The rationale is to unload the files that will be used the latest in the traversal, in order to avoid swapping intermediate files. This heuristic can easily be shown to be optimal for the divisible version of MINIO, where fractions of file can be written from and to secondary memory (see Section II-B.)

First Fit: This heuristic writes out the first file in S whose size is at least $IOReq(j)$. If no such file exists, the LSNF strategy is used.

Best Fit: This heuristic writes out the file in S whose size is the closest of $IOReq(j)$: it chooses i_k such that $|IOReq(j) - f_{i_k}|$ is minimal. This step is repeated until enough space has been freed.

First Fill: This heuristic writes out the first file in S whose size is smaller than $IOReq(j)$. This step is repeated until enough space has been freed. If not enough space can be freed, the LSNF strategy is then used. The rationale here is to avoid unduly writing big files out to secondary memory, thus significantly increasing I/O volume. Instead this heuristic tries to ''fill'' out the required I/O volume with the first eligible files.

Best Fill: This heuristic writes out the file whose size is the closest to $IOReq(j)$ among those files in S whose size is smaller than $IOReq(j)$. This step is repeated until enough space has been freed. If not enough space can be freed, the LSNF strategy is then used. The rationale here is to ''fill'' out the required I/O volume, but this time with the best eligible files.

Best K Combination: This last heuristic considers the first K files in S (we use $K = 5$ in the experiments) and selects the best combination, i.e., the subset whose size is the closest to $IOReq(j)$. This step is then repeated until enough memory has been freed.

VI. EXPERIMENTS

In this section, we experimentally compare the three algorithms for MINMEMORY, namely $PostOrder$ (which finds the best postorder traversal of the tree) and the two optimal variants Liu (exact algorithm of [2]) and $MinMem$. We evaluate the deviation of $PostOrder$ form the optimal solution, and we study the execution cost of each algorithm. Next we report on the performance of the heuristics for MINIO.

A. Setup

Each algorithm has been implemented in highly optimized C++ versions. The $PostOrder$ and Liu algorithms are written as iterative codes while $MinMem$ is a recursive code. Their behavior has been validated on a platform based on an Intel Xeon 5250 processor. Source code for all the algorithms, heuristics and experiments is publicly available at http://graal.ens-lyon.fr/~mjacquel/minmem.html.

Experiments were conducted within the Matlab environment for commodity reasons, especially ease of access to various data sets. We use a generic tool called *performance profiles* [16] to assess the proposed algorithms and heuristics. The main idea behind performance profiles is to use a cumulative distribution function as the performance metric, instead of taking averages over all test cases. We investigate the performance of the algorithms and heuristics in terms of running times and the quality of the solution (the memory requirement, or the total I/O volume). For a given metric, a profile plot shows the fraction of cases where a specific method gives results which are within some value τ of the best result reached by all algorithms. Therefore the higher the fraction, the more preferable the method. For example, for the runtime metric, a τ value shows the fraction of cases where the running time of the target algorithm is within τ of the fastest algorithm shown in the same plot. Similarly, for the memory requirement metric, a τ value shows the fraction of cases where the memory requirement of the target algorithm is within τ of the best result found by any algorithm shown in the same plot.

B. The Data Set

The data set contains assembly trees of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices/). The matrices satisfy the following assertions: square, number of rows is between 2×10^4 and 2×10^5, the number of nonzeros per row is at least 2.5, and the number of nonzeros is at most 5×10^6. At the time of testing there were 291 matrices satisfying these properties. We use the symmetrized pattern of the matrices, e.g., the pattern of $|A|+|A|^T+I$. We first order the matrices using MeTiS [17] (through MeshPart toolbox [18]) and `amd` (available in Matlab), and then build the corresponding elimination trees using the `symbfact` routine of Matlab. We also perform relaxed node amalgamation on these elimination trees to create assembly trees. We have created a large set of instances by allowing 1, 2, 4, and 16 (if $n > 1.6 \times 10^5$) relaxed amalgamations per node. We always realize perfect amalgamations, e.g., when a node is the only child of its parent and the parent has only one less entry in the associated column in L, the two nodes are amalgamated. When the current amalgamated node does not contain more than the allowed amalgamation per node, we amalgamate the node with its densest child. At the end we compute the weight of a node as $\eta^2+2\eta(\mu-1)$, where η is the number of nodes amalgamated, and μ is the number of nonzeros in the column of L which is associated with the highest node (in the starting elimination tree). Edge weights are computed as

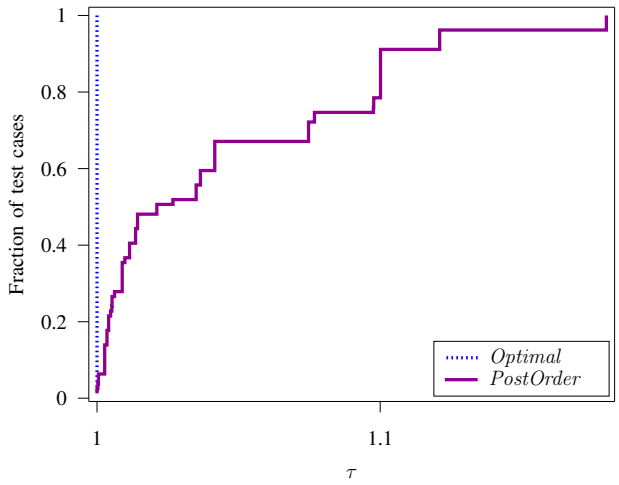


Figure 5. Performance profile for comparing the memory requirement obtained by *PostOrder* with the optimum values for the assembly trees for which *PostOrder* does not find an optimal solution.

Non optimal *PostOrder* traversals	4.2%
Max. *PostOrder* to opt. cost ratio	1.18
Avg. *PostOrder* to opt. cost ratio	1.01
Std. Dev. of *PostOrder* to opt. cost ratio	0.01

Table I
STATISTICS ON MEMORY COST OF *PostOrder* FOR ASSEMBLY TREES.

$(\mu - 1)^2$. These numbers correspond respectively to n_i and f_i as described in Section III.

C. Results for MINMEMORY

The first objective is to evaluate the performance of *PostOrder* in terms of the memory requirement of the resulting traversal with respect to the optimal value. In 95.8% of the cases, *PostOrder* is optimal. Only the non-optimal cases are depicted on Figure 5, *PostOrder* requiring up to 18% more memory than the optimal solution. Detailed statistics are given in Table I. As a conclusion, *PostOrder* statistically gives very good results for assembly trees, except in rare cases where it can require up to 20% more main memory than the optimal traversal.

The second objective is to compare the running times of the three algorithms. We observe on Figure 6 that *MinMem* is the fastest algorithm in 80% of the cases, and clearly outperforms *Liu*.

Altogether, these experiments show that in the context of sparse matrix assembly trees, *PostOrder* frequently offers optimal or near-optimal results. When *PostOrder* is not optimal, it is reasonably close to the minimum memory required to process the tree. Nevertheless, whenever memory becomes a key resource, *MinMem* can compete with *PostOrder* in terms of running time, and it always produces the optimum memory requirement, therefore constituting an interesting alternative.

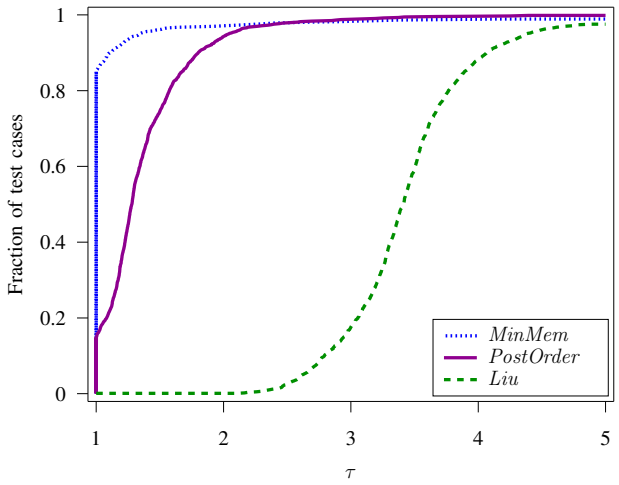


Figure 6. Performance profiles for comparing the running time of the three algorithms for the MINMEMORY problem on the assembly trees.

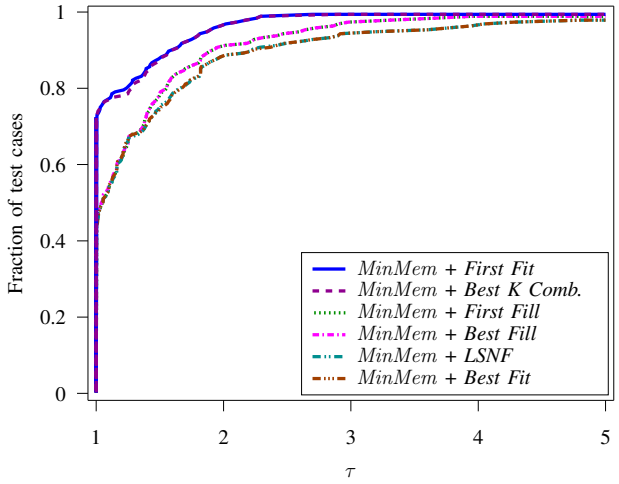


Figure 7. Performance profiles for comparing the resulting I/O volume of the heuristics for the *MinMem* algorithm on the assembly trees.

D. Results for MINIO

This experiment aims at evaluating the six heuristics introduced in Section V-B for the MINIO problem. Tree traversals are obtained using *PostOrder*, *Liu* and *MinMem* for the MINMEMORY problem. The available memory ranges from $\max_{i \in T} MemReq(i)$, to the minimal memory required for the traversal.

On Figure 7, the performance profile of all heuristics applied on traversals produced by *MinMem* is depicted. The best heuristic is clearly First Fit, which is almost tied by Best K Combination. Then Best Fill and First Fill provide almost the same I/O volume, and in turn perform better than Last Scheduled Node First and Best Fit, which are very close. As a consequence, because of its good behavior and low complexity, First Fit represents the best alternative among the six policies. This conclusion remains true when applying

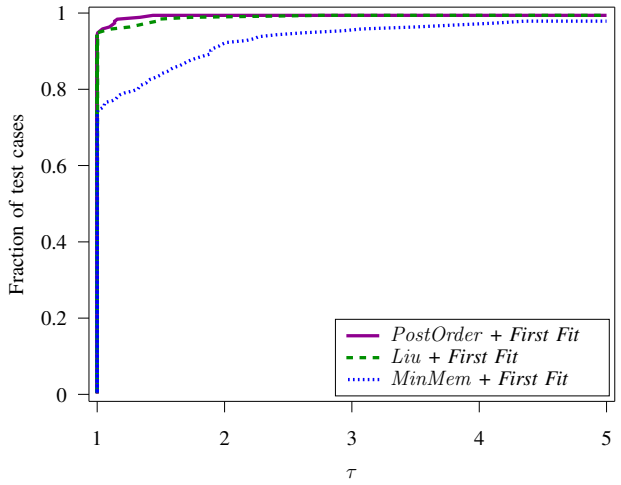


Figure 8. Performance profiles for comparing the resulting I/O volume of the three algorithms equipped with the First Fit heuristic on the assembly trees.

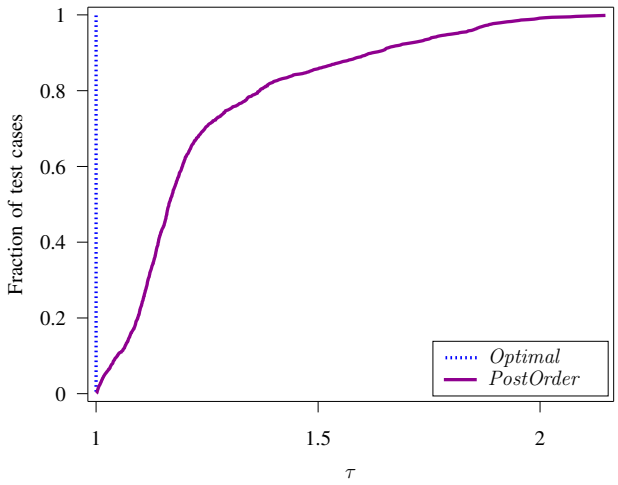


Figure 9. Performance profile for comparing the memory requirement obtained by *PostOrder* with the optimum values on the random trees.

Non optimal *PostOrder* traversals	61%
Max. *PostOrder* to opt. cost ratio	2.22
Avg. *PostOrder* to opt. cost ratio	1.12
Std. Dev. of *PostOrder* to opt. cost ratio	0.13

Table II
STATISTICS ON MEMORY COST OF *PostOrder* FOR RANDOM TREES.

the heuristics to traversals produced by *PostOrder* or *Liu*.

The next experiment aims at characterizing the behavior of the algorithms designed for MINMEMORY in the context of out-of-core traversals. The policy used for I/O is First Fit. The performance profile of every traversal is reported on Figure 8. The best results are provided by *PostOrder*. This experiment also shows that *MinMem* does not produce good out-of-core tree traversals, and is outperformed by *Liu* which provides better traversals for MINIO. This interesting result is due to the fact that contrarily to *MinMem*, *Liu* produces long chains of dependent tasks by construction. These chains reduce the pressure on main memory since files produced by a task will be consumed soon, thereby reducing the I/O volume. *PostOrder* also benefits from this phenomenon.

E. More on *PostOrder* Performance

This last experiment comes almost as a digression, because we do not use assembly trees here. The objective is to further assess the performance of *PostOrder* in terms of the resulting memory requirement. While the theory tells us that *PostOrder* can be arbitrarily bad (see Theorem 1), it turns out that its performance on assembly trees is very good (see Table I). We wanted to assess the performance of *PostOrder* on randomly generated trees. We keep the structure of every actual assembly tree from the data set discussed above, and assign random integers ranging from 1 to $N/500$ to the node weights and from 1 to N for the edge weights (N denoting the number of tree nodes). This leads to a comprehensive data set containing more than 3200 trees, and allows for a more refined performance evaluation of *PostOrder*.

The experiment (see Figure 9) shows that *PostOrder* requires more than the minimum memory in 61% of the cases. In some cases, *PostOrder* may require more than twice as much memory as the optimal solution. More details are given in in Table II. All in all, this experiment shows that when dealing with general trees, it is mandatory to use an optimal algorithm if main memory is a scarce resource.

VII. CONCLUSION

We have discussed how to traverse the nodes of a tree-shaped workflow so as to optimize the memory used in a two-level memory system. We have investigated two main problems. In the MINMEMORY problem, the aim is to minimize the memory requirement, while in the MINIO problem, the aim is to minimize the I/O volume, given a limited memory. Our motivating application was the multifrontal method of sparse matrix factorization in which elimination (or assembly) trees are used to reorganize the computations. The MINMEMORY problem corresponds to the problem of minimizing the memory requirement of an in-core execution of the multifrontal method, while the MINIO problem corresponds to the problem of minimizing the I/O requirement in an out-of-core execution.

For the MINMEMORY problem, we have proposed an exact algorithm which runs faster than the reference alternative of Liu [2]. The current state of the art software for sparse matrix factorization finds the best postorder as a solution to the MINMEMORY problem. This is done both for convenience and for the in-core memory requirement. We have investigated how good this choice is, and concluded that in

most practical cases, the minimum memory requirement due to a postorder is usually close to the optimal one (in a large set of instances we have seen at most 18% increase with respect to the memory minimizing traversal). However, we also showed that on general trees, the best postorder can result in memory requirements that are arbitrarily large.

We have shown that the MINIO problem is NP-complete, as well as some of its variations (for example, we have shown that finding the postorder traversal that minimizes the I/O volume is NP-complete). We have designed heuristics for the problem and have performed thorough experimental comparisons. Our experiments are based on highly optimized versions of three tree traversal algorithms, and precisely assess the quality of each proposed algorithm and I/O heuristic. We have shown that our $MinMem$ algorithm outperforms the running time of Liu's exact algorithm, but we have also observed that it was less suited for out-of-core execution.

With respect to the MINIO problem, there remain several challenging problems. Future research involves finding a lower bound for the minimum I/O volume when a fixed amount of main memory is permitted. This would allow to help assessing the absolute performance of the heuristics, rather than only comparing their relative performance. Even better than a bound, establishing a guarantee on the performance of the heuristics (showing that their achieved I/O volume always remain within a certain factor of the optimal) would be a very interesting contribution. Such a result seems out of reach for general traversals, but there is hope to derive an approximation algorithm for postorder traversals, which are simpler to analyze than arbitrary traversals.

More generally, we observe that the development of multicore platforms with non-uniform memory access introduces new levels of hierarchy in the whole memory system, from distributed caches to shared caches, to main memory, and to disk. Such platforms call for re-designing the whole computational chain of sparse matrix factorization, by introducing memory-aware computational kernels at every level. This paper is only a small step in this important direction.

ACKNOWLEDGEMENT

Loris Marchal and Bora Uçar are with CNRS. Yves Robert is with the Institut Universitaire de France. All authors are with Université de Lyon. This work was supported in part by the ANR Rescue project, and by the INRIA-Illinois Joint Laboratory for Petascale Computing.

REFERENCES

[1] J. W. H. Liu, "On the storage requirement in the out-of-core multifrontal method for sparse factorization," *ACM Trans. Math. Software*, vol. 12, no. 3, pp. 249–264, 1986.

[2] ——, "An application of generalized tree pebbling to sparse matrix factorization," *SIAM J. Algebraic Discrete Methods*, vol. 8, no. 3, 1987.

[3] J.-W. Hong and H. Kung, "I/O complexity: the red-blue pebble game," in *STOC'81: Proceedings of the 13th ACM symposium on Theory of Computing*. ACM Press, 1981, pp. 326–333.

[4] R. Schreiber, "A new implementation of sparse Gaussian elimination," *ACM Transactions on Mathematical Software*, vol. 8, no. 3, pp. 256–276, 1982.

[5] J. W. H. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 1, pp. 134–172, 1990.

[6] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Transactions on Mathematical Software*, vol. 9, pp. 302–325, 1983.

[7] J. W. H. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Review*, vol. 34, no. 1, pp. 82–109, 1992.

[8] C. Ashcraft and R. Grimes, "The influence of relaxed supernode partitions on the multifrontal method," *ACM Transactions on Mathematical Software*, vol. 15, no. 4, pp. 291–309, 1989.

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

[10] R. Sethi, "Complete register allocation problems," in *STOC'73: Proceedings of the fifth annual ACM symposium on Theory of computing*. ACM Press, 1973, pp. 182–195.

[11] J. R. Gilbert, T. Lengauer, and R. E. Tarjan, "The pebbling problem is complete in polynomial space," *SIAM J. Comput.*, vol. 9, no. 3, pp. 513–524, 1980.

[12] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1997.

[13] 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.

[14] 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.

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

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

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

[18] J. R. Gilbert, G. L. Miller, and S.-H. Teng, "Geometric mesh partitioning: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 19, no. 6, pp. 2091–2110, 1998.