Parallel Scheduling of Task Trees with Limited Memory

LIONEL EYRAUD-DUBOIS, INRIA and University of Bordeaux
LORIS MARCHAL, CNRS and University of Lyon
OLIVER SINNEN, University of Auckland
FRÉDÉRIC VIVIEN, INRIA and University of Lyon

This article investigates the execution of tree-shaped task graphs using multiple processors. Each edge of such a tree represents some large data. A task can only be executed if all input and output data fit into memory, and a data can only be removed from memory after the completion of the task that uses it as an input data. Such trees arise in the multifrontal method of sparse matrix factorization. The peak memory needed for the processing of the entire tree depends on the execution order of the tasks. With one processor, the objective of the tree traversal is to minimize the required memory. This problem was well studied, and optimal polynomial algorithms were proposed.

Here, we extend the problem by considering multiple processors, which is of obvious interest in the application area of matrix factorization. With multiple processors comes the additional objective to minimize the time needed to traverse the tree—that is, to minimize the makespan. Not surprisingly, this problem proves to be much harder than the sequential one. We study the computational complexity of this problem and provide inapproximability results even for unit weight trees. We design a series of practical heuristics achieving different trade-offs between the minimization of peak memory usage and makespan. Some of these heuristics are able to process a tree while keeping the memory usage under a given memory limit. The different heuristics are evaluated in an extensive experimental evaluation using realistic trees.

Categories and Subject Descriptors: F.2.0 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*General*; F.2.3 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Tradeoffs between Complexity Measures*; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Approximation algorithms, memory usage, multicriteria optimization, pebble game, scheduling, task graphs

ACM Reference Format:
Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. 2015. Parallel scheduling of task trees with limited memory. ACM Trans. Parallel Comput. 2, 2, Article 13 (June 2015), 37 pages.
DOI: http://dx.doi.org/10.1145/2779052

1. INTRODUCTION

Parallel workloads are often modeled as task graphs, where nodes represent tasks and edges represent the dependencies between tasks. There is an abundant literature on task graph scheduling when the objective is to minimize the total completion time, or

Authors' addresses: L. Eyraud-Dubois, INRIA, 200 avenue de la Vieille Tour, 33400 Talence, France; email: Lionel.Eyraud-Dubois@labri.fr; L. Marchal and F. Vivien, LIP, ENS-Lyon, 46 allée d'Italie, 69007 Lyon, France; emails: loris.marchal@ens-lyon.fr, frederic.vivien@inria.fr; O. Sinnen, Department of Electrical and Computer Engineering, University of Auckland, Private Bag 92019, Auckland, New Zealand; email: o.sinnen@auckland.ac.nz.

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. This is best exemplified with an application that, depending on the way it is scheduled, will either fit in the memory or require the use of swap mechanisms or out-of-core techniques. There are very few existing studies on the minimization of the memory footprint when scheduling task graphs, and even fewer of them targeting parallel systems.

We consider the following memory-aware parallel scheduling 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[1]: each node takes as input several large files, one for each of its children, and it produces a single large file; the different files may have different sizes. Furthermore, the execution of any node requires its *execution* file to be present; the execution file models the program and/or the temporary data of the task. We are to execute such a set of tasks on a parallel system made of p identical processing resources sharing the same memory. The execution scheme corresponds to a schedule of the tree where processing a node of the tree translates into reading the associated input files and producing the output file. How can the tree be scheduled to optimize the memory usage?

Modern computing platforms exhibit a complex memory hierarchy ranging from caches to RAM and disks and even sometimes tape storage, with the classical property that the smaller the memory, the faster. Thus, to avoid large running times, one usually wants to avoid the use of memory devices whose IO bandwidth is below a given threshold: even if out-of-core execution (when large data are unloaded to disks) is possible, this requires special care when programming the application and one usually wants to stay in the main memory (RAM). This is why, in this article, we are interested in the question of minimizing the amount of *main memory* needed to completely process an application.

Throughout the article, we consider *in-trees,* where a task can be executed only if all of its children have already been executed (This is absolutely equivalent to considering *out-trees* as a solution, as an in-tree can be transformed into a solution for the corresponding out-tree by just reversing the direction of time, as outlined in Jacquelin et al. [2011]). A task can be processed only if all of its files (input, output, and execution) fit in currently available memory. At a given time, many files may be stored in the memory, and at most p tasks may be processed by the p processors. This is obviously possible only if all tasks and execution files fit in memory. When a task finishes, the memory needed for its execution file and its input files is released. Clearly, the schedule that determines the processing times of each task plays a key role in determining which amount of main memory is needed for a successful execution of the entire tree.

The motivation for this work comes from numerical linear algebra, and especially the factorization of sparse matrices using direct multifrontal methods [Davis 2006]. During the factorization, the computations are organized as a tree workflow called the *elimination tree*, and the huge size of the data involved makes it absolutely necessary to reduce the memory requirement of the factorization. The sequential version of this problem (i.e., with $p = 1$ processor) has already been studied. Liu [1986] discusses how to find a memory-minimizing traversal when the traversal is required to correspond to a postorder traversal of the tree. A follow-up study [Liu 1987] presents an optimal algorithm to solve the general problem, without the postorder constraint on the traversal. Postorder traversals are known to be arbitrarily worse than optimal traversals for memory minimization [Jacquelin et al. 2011]. However, they are very natural and straightforward solutions to this problem, as they allow the full

[1]The concept of *file* is used here in a very general meaning and does not necessarily correspond to a classical file on a disk. Essentially, a file is a set of data.

processing of one subtree before starting a new one. Therefore, they are widely used in sparse matrix software like MUMPS [Amestoy et al. 2001, 2006], and in practice, they achieve close to optimal performance on actual elimination trees [Jacquelin et al. 2011]. Note that we consider here that no numerical pivoting is performed during the factorization and thus that all characteristics of the task tree (length of tasks, size of the data) are known before the computation really happens. Moreover, even if some software (including MUMPS) are able to distribute the processing of each task to multiple processors, here we focus on the case where each task has to be processed by a single processor, and we keep the extension to parallel tasks for future work.

The parallel version of this problem is a natural continuation of these studies: when processing large elimination trees, we would like to take advantage of parallel processing resources. However, to the best of our knowledge, no theoretical study exists for this problem. A preliminary version of this work, with fewer complexity results and proposed heuristics, was presented at IPDPS 2013 [Marchal et al. 2013]. The key contributions of this work are as follows:

—A new proof that the parallel variant of the *pebble-game* problem is NP-complete (simpler than in Marchal et al. [2013]). This shows that the introduction of memory constraints, in the simplest cases, suffices to make the problem NP-hard (Theorem 4.2).
—The proof that no schedule can simultaneously achieve a constant-ratio approximation for the memory minimization and for the makespan minimization (Theorem 4.4); bounds on the achievable approximation ratios for makespan and memory when the number of processors is fixed (Theorems 4.5 and 4.6).
—A series of practical heuristics achieving different trade-offs between the minimization of peak memory usage and makespan; some of these heuristics are guaranteed to keep the memory under a given memory limit.
—An exhaustive set of simulations using realistic tree-shaped task graphs corresponding to elimination trees of actual matrices; the simulations assess the relative and absolute performance of the heuristics.

The rest of this article is organized as follows. Section 2 reviews related studies. The notation and formalization of the problem are introduced in Section 3. Complexity results are presented in Section 4, whereas Section 5 proposes different heuristics to solve the problem, which are evaluated in Section 6.

2. BACKGROUND AND RELATED WORK

2.1. Sparse Matrix Factorization

As mentioned earlier, determining a memory-efficient tree traversal is very important in sparse numerical linear algebra. The elimination tree is a graph-theoretical model that represents the storage requirements, and computational dependencies and requirements, in the Cholesky and LU factorization of sparse matrices. In a previous study [Jacquelin et al. 2011], we described how such trees are built and how the multifrontal method [Liu 1992] organizes the computations along the tree. This is the context of the founding studies of Liu [1986, 1987] on memory minimization for postorder or general tree traversals presented in the previous section. Memory minimization is still a concern in modern multifrontal solvers when dealing with large matrices. Efforts have been made to design dynamic schedulers that take into account dynamic pivoting (which impacts the weights of edges and nodes) when scheduling elimination trees with strong memory constraints [Guermouche and L'Excellent 2004], or to consider both task and tree parallelism with memory constraints [Agullo et al. 2012]. Although these studies try to optimize memory management in existing parallel solvers, we aim at designing a simple model to study the fundamental underlying scheduling problem.

2.2. Scientific Workflows

The problem of scheduling a task graph under memory constraints also appears in the processing of scientific workflows whose tasks require large I/O files. Such workflows arise in many scientific fields, such as image processing, genomics, or geophysical simulations. The problem of task graphs handling large data has been identified in Ramakrishnan et al. [2007], which proposes some simple heuristic solutions. Surprisingly, in the context of quantum chemistry computations, Lam et al. [2011] have recently rediscovered the algorithm published in 1987 in Liu [1987].

2.3. Pebble Game and Its Variants

On the more theoretical side, this work builds on the many papers that have addressed the pebble game and its variants. The pioneering work of Sethi and Ullman [1970] on register allocation has been formalized as a pebble game on directed graphs in Gilbert et al. [1980] with the following rules:

(i) A pebble may be removed from a vertex at any time.
(ii) A pebble may be placed on a source node at any time.
(iii) If all predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.

In Sethi and Ullman [1970], the authors seek to minimize the number of registers that are needed to compute an arithmetic expression, which is naturally described as a tree. In this context, pebbling a node corresponds to loading an input (rule (ii)) or computing a particular subexpression (rule (iii)). They show how to compute, in polynomial time, a pebbling scheme that uses a minimum number of pebbles for in-trees. The problem of determining whether a general DAG can be executed with a given number of pebbles has been shown NP-hard by Sethi [1973] if no vertex is pebbled more than once. The general problem allowing recomputation—that is, repebbling a vertex that has been pebbled before—has been proven PSPACE complete [Gilbert et al. 1980].

The pebble-game problem translates into a simple instance of our problem when all I/O files have size 1 and all execution files have size 0. To the best of our knowledge, there have been no attempts to extend these results to parallel machines, with the objective of minimizing both memory and total execution time. We present such an extension in Section 4.

3. MODEL AND OBJECTIVES

3.1. Application Model

In this paper, we consider a tree-shaped task graph T composed of n nodes, or tasks, numbered from 1 to n. Nodes in the tree have an output file, an execution file (or program), and several input files (one per child). More precisely,

—Each node i in the tree has an execution file of size n_i, and its processing on a processor takes time w_i.
—Each node i has an output file of size f_i. If i is not the root, its output file is used as input by its parent $parent(i)$; if i is the root, its output file can be of size zero or contain outputs to the outside world.
—Each non–leaf node i in the tree has one input file per child. We denote by $Children(i)$ the set of the children of i. For each child $j \in Children(i)$, task j produces a file of size f_j for i. If i is a leaf node, then $Children(i) = \emptyset$ and i has no input file: we assume that the initial data of the task either resides in its execution file or is read from disk (or received from the outside word) during the execution of the task.

During the processing of a task i, the memory must contain its input files, the execution file, and the output file. The memory needed for this processing is thus

$$\left(\sum_{j \in Children(i)} f_j\right) + n_i + f_i.$$

After i has been processed, its input files and execution file (program) are discarded, whereas its output file is kept in memory until the processing of its parent.

3.2. Platform Model and Objectives

In this work, our goal is to design a simple platform model that allows the study of memory minimization on a parallel platform. We thus consider p identical processors sharing a single memory.

Any sequential optimal schedule for memory minimization is obviously an optimal schedule for memory minimization on a platform with any number p of processors. Therefore, memory minimization on parallel platforms is only meaningful in the scope of multicriteria approaches that consider trade-offs between the following two objectives:

—*Makespan*: The classical makespan, or total execution time, which corresponds to the time span between the beginning of the execution of the first leaf task and the end of the processing of the root task.

—*Memory*: The amount of memory needed for the computation. At each timestep, some files are stored in the memory and some task computations occur, inducing a memory usage. The *peak memory* is the maximum usage of the memory over the whole schedule, hence the memory that needs to be available, which we aim to minimize.

4. COMPLEXITY RESULTS IN THE PEBBLE-GAME MODEL

Since there are two objectives, the decision version of our problem can be stated as follows.

Definition 4.1 (*BiObjectiveParallelTreeScheduling*). Given a tree-shaped task graph T with file sizes and task execution times, p processors, and two bounds $B_{C_{\max}}$ and B_{mem}, is there a schedule of the task graph on the processors whose makespan is not larger than $B_{C_{\max}}$ and whose peak memory is not larger than B_{mem}?

This problem is obviously NP-complete. Indeed, when there are no memory constraints ($B_{mem} = \infty$) and when the task tree does not contain any inner node—that is, when all tasks are either leaves or the root—then our problem is equivalent to scheduling independent tasks on a parallel platform that is an NP-complete problem as soon as tasks have different execution times [Lenstra et al. 1977]. Conversely, minimizing the makespan for a tree of same-size tasks can be solved in polynomial time when there are no memory constraints [Hu 1961]. In this section, we consider the simplest variant of the problem. We assume that all input files have the same size ($\forall i, f_i = 1$) and no extra memory is needed for computation ($\forall i, n_i = 0$). Furthermore, we assume that the processing of each node takes unit time: $\forall i, w_i = 1$. We call this variant of the problem the *pebble-game model* because it perfectly corresponds to the pebble-game problems introduced earlier: the weight $f_i = 1$ corresponds to the pebble that one must put on node i to process it; this pebble must remain there until the parent of node i has been completed, as the parent of node i uses as input the output of node i. Processing a node is done in unit time.

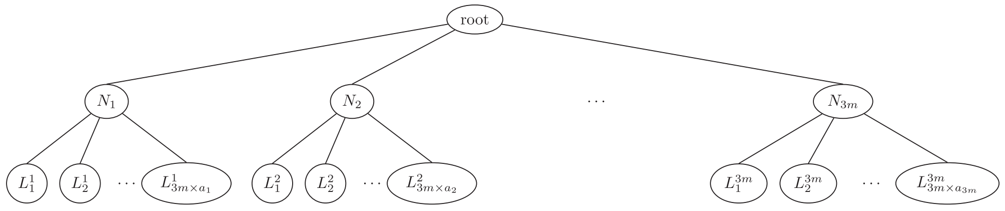

Fig. 1. Tree used for the NP-completeness proof.

In this section, we first show that, even in this simple variant, the introduction of memory constraints (a limit on the number of pebbles) makes the problem NP-hard (Section 4.1). Then, we show that when trying to minimize both memory and makespan, it is not possible to get a solution with a constant approximation ratio for both objectives, and we provide tighter ratios when the number of processors is fixed (Section 4.2).

4.1. NP-Completeness

THEOREM 4.2. *The BiObjectiveParallelTreeScheduling problem is NP-complete in the pebble-game model (i.e., with $\forall i, f_i = w_i = 1, n_i = 0$).*

PROOF. First, it is straightforward to check that the problem is in NP: given a schedule, it is easy to compute its peak memory and makespan.

To prove the problem NP-hard, we perform a reduction from 3-PARTITION, which is known to be NP-complete in the strong sense [Garey and Johnson 1979]. We consider the following instance $\mathcal{I}_1$ of the 3-PARTITION problem: let a_i be $3m$ integers and B an integer such that $\sum a_i = mB$. We consider the variant of the problem, also NP-complete, where $\forall i, B/4 < a_i < B/2$. To solve $\mathcal{I}_1$, we need to solve the following question: does there exist a partition of the a_i's in m subsets $S_1, \ldots, S_m$, each containing exactly three elements, such that for each S_k, $\sum_{i \in S_k} a_i = B$? We build the following instance $\mathcal{I}_2$ of our problem, illustrated in Figure 1. The tree contains a root r with $3m$ children, the N_i's, each one corresponding to a value a_i. Each node N_i has $3m \times a_i$ children, $L_1^i, \ldots, L_{3m\times a_i}^i$, which are leaf nodes. The question is to find a schedule of this tree on $p = 3mB$ processors, whose peak memory is not larger than $B_{mem} = 3mB + 3m$ and whose makespan is not larger than $B_{C_{\max}} = 2m + 1$.

Assume first that there exists a solution to $\mathcal{I}_1$—in other words, that there are m subsets S_k of three elements with $\sum_{i \in S_k} a_i = B$. In this case, we build the following schedule:

—At step 1, we process all of the nodes $L_x^{i_1}$, $L_y^{j_1}$, and $L_z^{k_1}$ with $S_1 = \{a_{i_1}, a_{j_1}, a_{k_1}\}$. There are $3mB = p$ such nodes, and the amount of memory needed is also $3mB$.
—At step 2, we process the nodes N_{i_1}, N_{j_1}, N_{k_1}. The memory needed is $3mB + 3$.
—At step $2n + 1$, with $1 \le n \le m - 1$, we process the $3mB = p$ nodes $L_x^{i_n}$, $L_y^{j_n}$, $L_z^{k_n}$ with $S_n = \{a_{i_n}, a_{j_n}, a_{k_n}\}$. The amount of memory needed is $3mB + 3n$ (counting the memory for the output files of the N_t nodes previously processed).
—At step $2n + 2$, with $1 \le n \le m - 1$, we process the nodes N_{i_n}, N_{j_n}, N_{k_n}. The memory needed for this step is $3mB + 3(n + 1)$.
—At step $2m + 1$, we process the root node and the memory needed is $3m + 1$.

Thus, the peak memory of this schedule is B_{mem} and its makespan $B_{C_{\max}}$.

Reciprocally, assume that there exists a solution to problem $\mathcal{I}_2$—that is, there exists a schedule of makespan at most $B_{C_{\max}} = 2m+1$. Without loss of generality, we assume that the makespan is exactly $2m+1$. We start by proving that at any step of the algorithm, at most three of the N_i nodes are being processed. By contradiction, assume that four (or more) such nodes $N_{i_s}, N_{j_s}, N_{k_s}, N_{l_s}$ are processed during a certain step s. We recall that $a_i > B/4$ so that $a_{i_s} + a_{j_s} + a_{k_s} + a_{l_s} > B$ and thus $a_{i_s} + a_{j_s} + a_{k_s} + a_{l_s} \geq B + 1$. The memory needed at this step is thus at least $(B+1)3m$ for the children of the nodes N_{i_s}, N_{j_s}, N_{k_s}, and N_{l_s} and 4 for the nodes themselves, hence a total of at least $(B+1)3m+4$, which is more than the prescribed bound B_{mem}. Thus, at most three N_i nodes are processed at any step. In the considered schedule, the root node is processed at step $2m+1$. Then, at step $2m$, some of the N_i nodes are processed, and at most three of them from what precedes. The a_i's corresponding to those nodes make the first subset S_1. Then, all nodes L_x^j such that $a_j \in S_1$ must have been processed at the latest at step $2m-1$, and they occupy a memory footprint of $3m \sum_{a_j \in S_1} a_j$ at steps $2m-1$ and $2m$. Let us assume that a node N_k is processed at step $2m-1$. For the memory bound B_{mem} to be satisfied, we must have $a_k + \sum_{a_j \in S_1} a_j \leq B$. (Otherwise, we would need a memory of at least $3m(B+1)$ for the involved L_x^j nodes plus 1 for the node N_k). Therefore, node N_k can as well be processed at step $2m$ instead of step $2m-1$. We then modify the schedule to schedule N_k at step $2m$ and thus add k to S_1. We can therefore assume, without loss of generality, that no N_i node is processed at step $2m-1$. Then, at step $2m-1$, only the children of the N_j nodes with $a_j \in S_1$ are processed, and all of them are. So, none of them have any memory footprint before step $2m-1$. We then generalize this analysis: at step $2i$, for $1 \leq i \leq m-1$, only some N_j nodes are processed and they define a subset S_i; at step $2i-1$, for $1 \leq i \leq m-1$, the only processed nodes are the nodes L_x^k that are children of the nodes N_j such that $a_j \in S_i$.

Because of the memory constraint, each of the m subsets of a_i's built earlier sum to at most B. Since they contain all a_i's, their sum is mB. Thus, each subset S_k sums to B, and we have built a solution for $\mathcal{I}_1$. □

4.2. Joint Minimization of Both Objectives

As our problem is NP-complete, it is natural to wonder whether approximation algorithms can be designed. In this section, we prove that there does not exist any scheduling algorithm that approximates both the minimum makespan and the minimum peak memory with constant factors. This is equivalent to saying that there is no *Zenith* (also called *simultaneous*) approximation. We first state a lemma, valid for any tree-shaped task graph, which provides lower bounds for the makespan of any schedule.

LEMMA 4.3. *For any schedule S on p processors with a peak memory M, we have the two following lower bounds on the makespan $C_{\max}$:*

$$C_{\max} \geq \frac{1}{p}\sum_{i=1}^{n} w_i$$

$$M \times C_{\max} \geq \sum_{i=1}^{n}\left(n_i + f_i + \sum_{j \in Children(i)} f_j\right) w_i.$$

In the pebble-game model, these equations can be written as

$$C_{\max} \geq n/p$$

$$M \times C_{\max} \geq 2n-1.$$

PROOF. The first inequality is a classical bound stating that all tasks must be processed before $C_{\max}$.

Similarly, each task i uses a memory of $n_i + f_i + \sum_{j \in Children(i)} f_j$ during a time w_i. Hence, the total memory usage (i.e., the sum over all time instants t of the memory used by

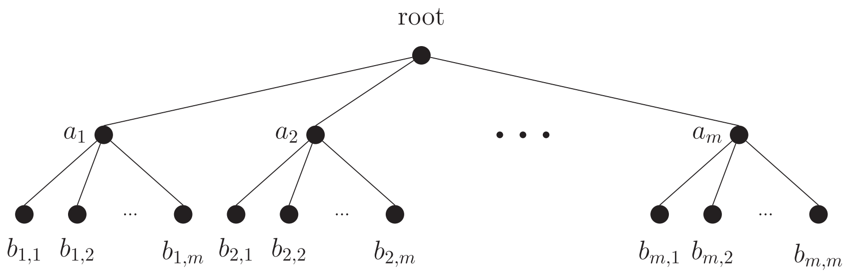

Fig. 2. Tree used for establishing Theorem 4.4.

S at time t) needs to be at least equal to $\sum_{i=1}^{n}(n_i + f_i + \sum_{j\in Children(i)} f_j)w_i$. Because S uses a memory that is not larger than M at any time, the total memory usage is upper bounded by $M \times C_{\max}$. This gives us the second inequality. In the pebble-game model, the right-hand term of the second inequality can be simplified:

$$\sum_{i=1}^{n}\left(n_i + f_i + \sum_{j\in Children(i)} f_j\right)w_i = \left(\sum_{i=1}^{n} f_i\right) + \left(\sum_{i=1}^{n}\sum_{j\in Children(i)} f_j\right) = n + (n-1). \quad \square$$

In the next theorem, we show that it is not possible to design an algorithm with constant approximation ratios for both makespan and maximum memory (i.e., approximation ratios independent of the number of processors p). In Theorem 4.5, we will provide a refined version that analyzes the dependence on p.

THEOREM 4.4. *For any given constants α and β, there does not exist any algorithm for the pebble-game model that is both an α-approximation for makespan minimization and a β-approximation for peak memory minimization when scheduling in-tree task graphs.*

PROOF. In this proof, we consider the tree depicted in Figure 2. The root of this tree has m children $a_1, \dots, a_m$. Any of these children, a_i, has m children $b_{i,1}, \dots, b_{i,m}$. Therefore, overall, this tree contains $n = 1 + m + m \times m$ nodes. On the one hand, with a large number of processors (namely, m^2), this tree can be processed in $C^*_{\max} = 3$ timesteps: all of the leaves are processed in the first step, all of the a_i nodes in the second step, and finally the root in the third and last step. On the other hand, the minimum memory required to process the tree is $M^* = 2m$. This is achieved by processing the tree with a single processor. The subtrees rooted at the a_i's are processed one at a time. The processing of the subtree rooted at node a_i requires a memory of $m + 1$ (for the processing of its root a_i once the m leaves have been processed). Once such a subtree is processed, there is a unit file that remains in memory. Hence, the peak memory usage when processing the j-th of these subtrees is $(j-1)+(m+1) = j+m$. The overall peak $M^* = 2m$ is thus reached when processing the root of the last of these subtrees.

Let us assume that there exists a schedule S that is both an α-approximation for the makespan and a β-approximation for the peak memory. Then, for the tree of Figure 2, the makespan $C_{\max}$ of S is at most equal to 3α, and its peak memory M is at most equal to $2\beta m$. Because $n = 1 + m + m^2$, Lemma 4.3 implies that $M \times C_{\max} \geq 2n - 1 = 2m^2 + 2m + 1$. Therefore, $M \geq \frac{2m^2+2m+1}{3\alpha}$. For a sufficiently large value of m, this is larger than $2\beta m$, the upper bound on M. This contradicts the hypothesis that S is a β-approximation for peak memory usage. $\square$

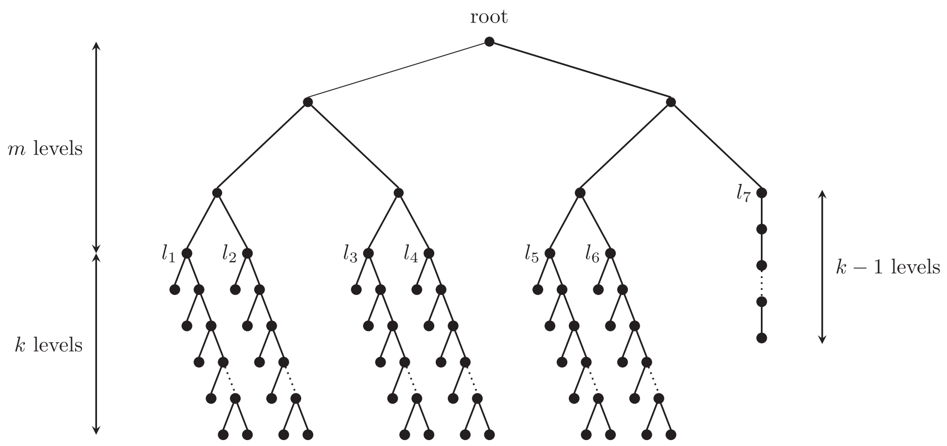

Fig. 3. Tree used to establish Theorem 4.5 for $p = 13$ processors.

Theorem 4.4 only considers approximation algorithms whose approximation ratios are constant. In the next theorem we consider algorithms whose approximations ratios may depend on the number of processors in the platform.

THEOREM 4.5. *When scheduling in-tree task graphs in the pebble-game model on a platform with $p \geq 2$ processors, there does not exist any algorithm that is both an $\alpha(p)$-approximation for makespan minimization and a $\beta(p)$-approximation for peak memory minimization, with*

$$\alpha(p)\beta(p) < \frac{2p}{\lceil \log(p) \rceil + 2}.$$

PROOF. We establish this result by contradiction. We assume that there exists an algorithm that is an $\alpha(p)$-approximation for makespan minimization and a $\beta(p)$-approximation for peak memory minimization when scheduling in-tree task graphs, with $\alpha(p)\beta(p) = \frac{2p}{(\lceil \log(p) \rceil + 2)} - \epsilon$ with $\epsilon > 0$.

The proof relies on a tree similar to the one depicted in Figure 3 for the case $p = 13$. The top part of the tree is a complete binary subtree with $\lceil \frac{p}{2} \rceil$ leaves, $l_1, \ldots, l_{\lceil \frac{p}{2} \rceil}$, and of height m. Therefore, this subtree contains $2\lceil \frac{p}{2} \rceil - 1$ nodes, all of its leaves are at depth either m or $m - 1$, and $m = 1 + \lceil \log(\lceil \frac{p}{2} \rceil) \rceil = \lceil \log(p) \rceil$. To prove the last equality, we consider whether p is even:

—p is even: $\exists l \in \mathbb{N}$, $p = 2l$. Then, $1 + \lceil \log(\lceil \frac{p}{2} \rceil) \rceil = \lceil \log(2\lceil \frac{2l}{2} \rceil) \rceil = \lceil \log(2l) \rceil = \lceil \log(p) \rceil$.
—p is odd: $\exists l \in \mathbb{N}$, $p = 2l + 1$. Then, $1 + \lceil \log(\lceil \frac{p}{2} \rceil) \rceil = \lceil \log(2\lceil \frac{2l+1}{2} \rceil) \rceil = \lceil \log(2l + 2) \rceil$. Since $2l + 1$ is odd, $\lceil \log(2l + 2) \rceil = \lceil \log(2l + 1) \rceil = \lceil \log(p) \rceil$.

Each node l_i is the root of a *comb* subtree[2] of height k (except the last node if p is odd), and each comb subtree contains $2k - 1$ nodes. If p is odd, the last leaf of the binary top subtree, $l_{\lceil \frac{p}{2} \rceil}$, is the root of a chain subtree with $k - 1$ nodes. Then, the entire tree contains $n = pk - 1$ nodes (be careful not to count the roots of the comb subtrees twice):

[2]A *comb* tree is recursively defined either as a single node or a tree whose root has two children: one is a leaf and the other is the root of a comb tree.

—p is even: $\exists l \in \mathbb{N}$, $p = 2l$. Then,

$$n = \left(2\left\lceil\frac{p}{2}\right\rceil - 1\right) + \left(\left\lceil\frac{p}{2}\right\rceil(2k-2)\right) = (2l-1) + l(2k-2) = pk - 1.$$

—p is odd: $\exists l \in \mathbb{N}$, $p = 2l+1$. Then,

$$\begin{aligned} n &= \left(2\left\lceil\frac{p}{2}\right\rceil - 1\right) + \left(\left(\left\lceil\frac{p}{2}\right\rceil - 1\right)(2k-2)\right) + (k-2) \\ &= (2(l+1)-1) + (l+1-1)(2k-2) + (k-2) = (2l+1)k - 1 = pk - 1. \end{aligned}$$

With the p processors, it is possible to process all comb subtrees (and the chain subtree if p is odd) in parallel in k steps by using two processors per comb subtree (and one for the chain subtree). Then, $m-1$ steps are needed to complete the processing of the binary reduction (the l_i nodes have already been processed at the last step of the processing of the comb subtrees). Thus, the optimal makespan with p processors is $C^*_{\max} = k + m - 1$.

We now compute the optimal peak memory usage, which is obtained with a sequential processing. Each comb subtree can be processed with three units of memory, if we follow any postorder traversal starting from the deepest leaves. We consider the sequential processing of the entire tree that follows a postorder traversal that processes each comb subtree as described previously, that processes first the left-most comb subtree, then the second left-most comb subtree, the parent node of these subtrees, and so on, and finishes with the right-most comb subtree (or the chain subtree if p is odd). The peak memory is reached when processing the last comb subtree. At that time, either $m-2$ or $m-1$ edges of the binary subtree are stored in memory (depending on the value of $\lceil\frac{p}{2}\rceil$). The processing of the last comb subtree itself uses three units of memory. Hence, the optimal peak memory is not greater than $m+2$: $M^* \leq m+2$.

Let $C_{\max}$ denote the makespan achieved by the studied algorithm on the tree, and let M denote its peak memory usage. By definition, the studied algorithm is an $\alpha(p)$-approximation for the makespan: $C_{\max} \leq \alpha(p)C^*_{\max}$. Thanks to Lemma 4.3, we know that

$$M \times C_{\max} \geq 2n - 1 = 2pk - 3.$$

Therefore,

$$M \geq \frac{2pk-3}{C_{\max}} \geq \frac{2pk-3}{\alpha(p)(k+m-1)}.$$

The approximation ratio of the studied algorithm with respect to the peak memory usage is thus bounded by

$$\beta(p) \geq \frac{M}{M^*} \geq \frac{2pk-3}{\alpha(p)(k+m-1)(m+2)}.$$

Therefore, if we recall that $m = \lceil\log(p)\rceil$,

$$\alpha(p)\beta(p) \geq \frac{2pk-3}{(k+m-1)(m+2)} = \frac{2pk-3}{(k+\lceil\log(p)\rceil-1)(\lceil\log(p)\rceil+2)} \xrightarrow[k\to\infty]{} \frac{2p}{\lceil\log(p)\rceil+2}.$$

Hence, there exists a value k_0 such that for any $k \geq k_0$,

$$\alpha(p)\beta(p) \geq \frac{2p}{\lceil\log(p)\rceil+2} - \frac{\epsilon}{2}.$$

This contradicts the definition of ϵ and hence concludes the proof. □

Readers may wonder whether the bound in Theorem 4.5 is tight. This question is especially relevant because the proof of Theorem 4.5 uses the average memory usage

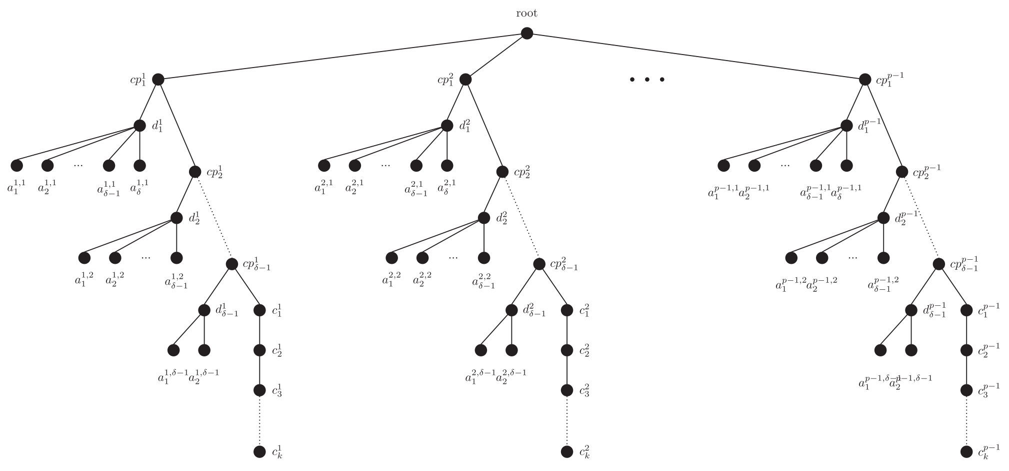

Fig. 4. Tree used for establishing Theorem 4.6.

as a lower bound to the peak memory usage. This technique enables one to design a simple proof, which may however be very crude. In fact, in the special case where $\alpha(p) = 1$—that is, for makespan-optimal algorithms—a stronger result holds. For that case, Theorem 4.5 states that $\beta(p) \geq \frac{2p}{\lceil \log(p) \rceil + 2}$. Theorem 4.6 states that $\beta(p) \geq p - 1$ (which is a stronger bound when $p \geq 4$). This result is established through a careful, painstaking analysis of a particular task graph. Using the average memory usage argument on this task graph would not enable one to obtain a nontrivial bound.

THEOREM 4.6. *There does not exist any algorithm for the pebble-game model that is both optimal for makespan minimization and that is a $(p - 1 - \epsilon)$-approximation algorithm for the peak memory minimization, where p is the number of processors and $\epsilon > 0$.*

PROOF. To establish this result, we proceed by contradiction. Let p be the number of processors. We then assume that there exists an algorithm $\mathcal{A}$ that is optimal for makespan minimization and is a $\beta(p)$-approximation for peak memory minimization, with $\beta(p) < p - 1$. Thus, there exists $\epsilon > 0$ such that $\beta(p) = p - 1 - \epsilon$.

We start by presenting the tree used to establish this theorem. Then, we compute its optimal peak memory and its optimal makespan. Finally, we derive a lower bound on the memory usage of any makespan-optimal algorithm to obtain the desired contradiction.

The tree. Figure 4 presents the tree used to derive a contradiction. This tree is made of $p - 1$ identical subtrees whose roots are the children of the tree root. The value of δ will be fixed later on.

Optimal peak memory. A memory-optimal sequential schedule processes each subtree rooted at cp_1^i sequentially. Each of these subtrees can be processed with a memory of $\delta + 1$ by processing first the subtree rooted at d_1^i, then the one rooted at d_2^i, and so forth, until the one rooted at $d_{\delta-1}^i$, and then the chain of c_j^i nodes, and the remaining cp_j^i nodes. The peak memory, reached when processing the last subtree rooted at cp_1^i, is thus $\delta + p - 1$.

Optimal execution time. The optimal execution time with p processors is at least equal to the length of the critical path. The critical path has a length of $\delta + k$, which is the length of the path from the root to any c_k^i node, with $1 \le i \le p-1$. We now define k for this lower bound to be an achievable makespan, with an overall schedule as follows:

—Each of the first $p-1$ processors processes one of the critical paths from end to end (except obviously for the root node that will only be processed by one of them).
—The last processor processes all of the other nodes. We define k so that this processor finishes processing all of the nodes it is allocated at time $k+\delta-2$. This allows the other processors to process all $p-1$ nodes cp_1^1 through cp_1^{p-1} from time $k+\delta-2$ to time $k+\delta-1$.

To find such a value for k, we need to compute the number of nodes allocated to the last processor. In the subtree rooted in cp_1^i, the last processor is in charge of processing the $\delta-1$ nodes d_1^i through $d_{\delta-1}^i$, and the descendants of the d_j^i nodes, for $1 \le j \le \delta-1$. As node d_j^i has $\delta-j+1$ descendants, the number of nodes in the subtree rooted in cp_1^i that are allocated to the last processor is equal to

$$(\delta-1)+\sum_{j=1}^{\delta-1}(\delta-j+1) = \delta-2+\frac{\delta(\delta+1)}{2} = \frac{\delta^2+3\delta-4}{2} = \frac{(\delta+4)(\delta-1)}{2}.$$

All together, the last processor is in charge of the processing of $(p-1)\frac{(\delta+4)(\delta-1)}{2}$ nodes. As we stated earlier, we want this processor to be busy from time 0 to time $k+\delta-2$. This gives the value of k:

$$k+\delta-2 = (p-1)\frac{(\delta+4)(\delta-1)}{2} \quad\Leftrightarrow\quad k = \frac{(p-1)\delta^2+(3p-5)\delta+4(2-p)}{2}.$$

Note that, by looking at the first equality, the expression on the right-hand side of the second equality is always an integer; therefore, k is well defined.

To conclude at the optimal makespan with p processors is $k+\delta-1$, we just need to provide an explicit schedule for the last processor. This processor processes all of its allocated nodes, except node d_1^1, and nodes $a_1^{1,1}$ through $a_{\delta-3}^{1,1}$, in any order between the time 0 and k. Then, between time k and $k+\delta-2$, it processes the remaining $a_j^{1,1}$ nodes and then node d_1^1.

Lower bound on the peak memory usage. We first note that, by construction, under any makespan-optimal algorithm, the $p-1$ nodes c_j^i are processed during the time interval $[k-j, k-j+1]$. Similarly, the $p-1$ nodes cp_j^i are processed during the time interval $[k+\delta-j-1, k+\delta-j]$. Without loss of generality, we can assume that processor P_i, for $1 \le i \le p-1$, processes nodes c_k^i through c_1^i and then nodes $cp_{\delta-1}^i$ through cp_1^i from time 0 to time $k+\delta-1$. The processor P_p processes the other nodes. The only freedom an algorithm has is in the order processor P_p is processing its allocated nodes.

To establish a lower bound, we consider the class of schedules that are makespan optimal for the studied tree, whose peak memory usage is minimum among makespan-optimal schedules, and that satisfy the additional properties we just defined. We first show that, without loss of generality, we can further restrict our study to schedules that, once one child of a node d_j^i has started being processed, process all other children of that node and then the node d_j^i itself before processing any children of any other node $d_{j'}^{i'}$. We also establish this property by contradiction by assuming that there is no schedule (in the considered class) that satisfies the last property. Then, for any schedule $\mathcal{B}$, let $t(\mathcal{B})$ be the first date at which $\mathcal{B}$ starts the processing of a node $a_m^{i,j}$ while some node $a_{m'}^{i',j'}$ has already been processed, but node $d_{j'}^{i'}$ has not been processed yet.

We consider a schedule $\mathcal{B}$, which maximizes $t(\mathcal{B})$ (note that $t(\mathcal{B})$ can only take values no greater than $\delta + k - 1$ and that the maximum and $\mathcal{B}$ are thus well defined). We then build from $\mathcal{B}$ another schedule $\mathcal{B}'$, whose peak memory is not greater and that does not overlap the processing of nodes d_j^i and $d_{j'}^{i'}$. This schedule is defined as follows. It is identical to $\mathcal{B}$ except for the time slots at which node d_j^i, node $d_{j'}^{i'}$ or any of their children was processed. If, under $\mathcal{B}$ node d_j^i was processed before node $d_{j'}^{i'}$ (respectively, node $d_{j'}^{i'}$ was processed before node d_j^i), then under the new schedule, in these time slots, all children of d_j^i are processed first, then d_j^i, then all children of $d_{j'}^{i'}$ and finally $d_{j'}^{i'}$ (respectively, all children of $d_{j'}^{i'}$ are processed first, then $d_{j'}^{i'}$, then all children of d_j^i and finally d_j^i). The peak memory due to the processing of nodes d_j^i and $d_{j'}^{i'}$ and of their children is now $\max\{\delta - j + 2, \delta - j' + 3\}$ (respectively $\max\{\delta - j' + 2, \delta - j + 3\}$). In the original schedule, it was no smaller than $\max\{\delta - j + 3, \delta - j' + 3\}$ because at least the output of the processing of one of the children of node $d_{j'}^{i'}$ (respectively d_j^i) was in memory while node d_j^i (respectively $d_{j'}^{i'}$) was processed. Hence, the peak memory of the new schedule $\mathcal{B}'$ is no greater than the one of the original schedule. The new schedule satisfies the desired property at least until time $t(\mathcal{B}) + 1$. Hence, $t(\mathcal{B}')$ is larger than $t(\mathcal{B})$, which contradicts the maximality of $\mathcal{B}$ with respect to the value $t(\mathcal{B})$.

From the preceding discussion, to establish a lower bound on the peak memory of any makespan-optimal schedule, it is sufficient to consider only those schedules that do not overlap the processing of different nodes d_j^i (and of their children). Let $\mathcal{B}$ be such a schedule. We know that under schedule $\mathcal{B}$, processor P_p processes nodes without interruption from time 0 until time $k + \delta - 2$. Therefore, in the time interval $[k + \delta - 3, k + \delta - 2]$, processor P_p processes a node d_1^i, say d_1^1. Then, we have shown that we can assume, without loss of generality, that processor P_p exactly processes the δ children of d_1^1 in the time interval $[k - 3, k + \delta - 3]$. Therefore, at time k, processor P_p has processed all nodes allocated to it except node d_1^1 and $\delta - 3$ of its children. Therefore, during the time interval $[k, k + 1]$, the memory must contain

(1) The output of the processing of all d_j^i nodes, for $1 \leq i \leq p - 1$ and $1 \leq j \leq \delta - 1$, except for the output of node d_1^1 (which has not yet been processed). This corresponds to $(p - 1)(\delta - 1) - 1$ elements.
(2) The output of the processing of three children of node d_1^1 and an additional unit to store the result of a fourth one. This corresponds to four elements.
(3) The result of the processing of the c_1^i nodes, for $1 \leq i \leq p - 1$ and room to store the results of the $cp_{\delta-1}^i$ nodes, for $1 \leq i \leq p - 1$. This corresponds to $2(p - 1)$ elements.

Overall, during this time interval, the memory must contain

$$((p - 1)(\delta - 1) - 1) + 4 + 2(p - 1) = (p - 1)\delta + p + 2$$

elements. As the optimal peak memory is $\delta + p - 1$, this gives us a lower bound on the ratio ρ of the memory used by $\mathcal{B}$ with respect to the optimal peak memory usage:

$$\rho \geq \frac{(p - 1)\delta + p + 2}{\delta + p - 1} \xrightarrow[\delta \to \infty]{} p - 1.$$

Therefore, there exists a value δ_0 such that

$$\frac{(p - 1)\delta_0 + p + 2}{\delta_0 + p - 1} > p - 1 - \frac{1}{2}\epsilon.$$

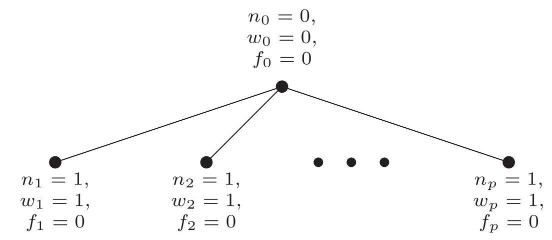

Fig. 5. Tree used for the proof of Lemma 4.7.

As algorithm $\mathcal{A}$ cannot have a strictly lower peak memory than algorithm $\mathcal{B}$ by definition of $\mathcal{B}$, this proves that the ratio for $\mathcal{A}$ is at least equal to $p - 1 - \frac{1}{2}\epsilon$, which contradicts the definition of ϵ. □

Furthermore, a similar result can also be derived in the general model (with arbitrary execution times and file sizes), but without the restriction that $\alpha(p) = 1$. This is done in the next lemma.

LEMMA 4.7. *When scheduling in-tree task graphs in the general model on a platform with $p \geq 2$ processors, there does not exist any algorithm that is both an $\alpha(p)$-approximation for makespan minimization and a $\beta(p)$-approximation for peak memory minimization, with $\alpha(p)\beta(p) < p$.*

PROOF. Consider the tree drawn in Figure 5. This tree can be scheduled in time $C^*_{\max} = 1$ on p processors if all non–root nodes are processed simultaneously (by using a peak memory of p), or sequentially in time p by using only $M^* = 1$ memory. Lemma 4.3 states that for any schedule with makespan $C_{\max}$ and peak memory M, we have $MC_{\max} \geq p$. This immediately implies that any algorithm with approximation ratio $\alpha(p)$ for makespan and $\beta(p)$ for peak memory minimization must verify $\alpha(p)\beta(p) \geq p$. This bound is tight because in this model, any memory-optimal sequential schedule is an approximation algorithm with $\beta(p) = 1$ and $\alpha(p) = p$. □

5. HEURISTICS

Given the complexity of optimizing the makespan and memory at the same time, we have investigated heuristics and propose six algorithms. The intention is that the proposed algorithms cover a range of use cases, where the optimization focus wanders between the makespan and the required memory. The first heuristic, PARSUBTREES (Section 5.1), employs a memory-optimizing sequential algorithm for each of its subtrees, the different subtrees being processed in parallel. Hence, its focus is more on the memory side. In contrast, PARINNERFIRST and PARDEEPESTFIRST are two list scheduling–based algorithms (Section 5.2), which should be stronger in the makespan objective. Nevertheless, the objective of PARINNERFIRST is to approximate a postorder in parallel, which is good for memory in a sequential processing. The focus of PARDEEPESTFIRST is fully on the makespan. Then, we move to memory-constrained heuristics (Section 5.3). Initially, we adapt the two list scheduling heuristics to obtain bounds on their memory consumption. Finally, we design a heuristic, MEMBOOKINGINNERFIRST (Section 5.3.3), that proposes a parallel execution of a sequential postorder while satisfying a memory bound given as input. We conclude this section by gathering all approximation results for the proposed heuristics in Table I.

5.1. Parallel Execution of Subtrees

The most natural idea to process a tree T in parallel is arguably to split it into subtrees, to process each of these subtrees with a sequentially memory-optimal algorithm

[Jacquelin et al. 2011; Liu 1987], and to have these sequential executions happen in parallel. The underlying idea is to assign to each processor a whole subtree to enable as much parallelism as there are processors while allowing the use of a single-processor memory-optimal traversal on each subtree. Algorithm 1 outlines such an algorithm, using Algorithm 2 for splitting T into subtrees. The makespan obtained using ParSubtrees is denoted by $C_{\max}^{\text{ParSubtrees}}$.

ALGORITHM 1: ParSubtrees(T, p)

```
1 Split tree $T$ into $q$ subtrees ($q \leq p$) and a set of remaining nodes, using SplitSubtrees($T$, $p$).
2 Concurrently process the $q$ subtrees, each using a memory minimizing algorithm (e.g.,
  [Jacquelin et al.]).
3 Sequentially process the set of remaining nodes, using a memory minimizing algorithm.
```

In this approach, q subtrees of T, $q \leq p$, are processed in parallel. Each of these subtrees is a maximal subtree of T. In other words, each of these subtrees includes all descendants (in T) of its root. The nodes not belonging to the q subtrees are processed sequentially. These are the nodes where the q subtrees merge, the nodes included in subtrees that were produced in excess (if more than p subtrees were created), and the ancestors of these nodes. An alternative approach, as discussed later, is to process all produced subtrees in parallel, assigning more than one subtree to each processor when $q > p$. The advantage of Algorithm 1 is that we can construct a splitting into subtrees that minimizes its makespan, established shortly in Lemma 5.1.

As w_i is the computation weight of node i, W_i denotes the total computation weight (i.e., sum of weights) of all nodes in the subtree rooted in i, including i. SplitSubtrees uses a node priority queue PQ in which the nodes are sorted by nonincreasing W_i, and ties are broken according to nonincreasing w_i. *head*(PQ) returns the first node of PQ while *popHead*(PQ) also removes it. $PQ[i]$ denotes the i-th element in the queue.

SplitSubtrees starts with the root of the entire tree and continues splitting the largest subtree (in terms of the total computation weight W) until this subtree is a leaf node ($W_{head(PQ)} = w_{head(PQ)}$). The execution time of Step 2 of ParSubtrees is that of the largest of the q subtrees of the splitting, hence $W_{head(PQ)}$ for the solution found by SplitSubtrees. Splitting subtrees that are smaller than the largest leaf ($W_j < \max_{i \in T} w_i$) cannot decrease the parallel time, but only increase the sequential time. More generally, given any splitting s of T into subtrees, the best execution time for s with ParSubtrees is achieved by choosing the p largest subtrees for the parallel Step 2. This can be easily derived, as swapping a large tree included in the sequential part with a smaller tree included in the parallel part cannot increase the total execution time. Hence, the value $C_{\max}^{\text{ParSubtrees}}(s)$ computed in Step 12 is the makespan that would be obtained by ParSubtrees on the splitting computed so far. At the end of algorithm SplitSubtrees (Step 2), the splitting that yields the smallest makespan is selected.

Lemma 5.1. SplitSubtrees *returns a splitting of T into subtrees that results in the makespan-optimal processing of T with* ParSubtrees.

Proof. The proof is by contradiction. Let $s_{\min}$ be the splitting into subtrees selected by SplitSubtrees. Assume now that there is a different splitting s_{opt} that results in a strictly shorter processing with ParSubtrees.

Because of the termination condition of the *while*-loop, SplitSubtrees splits any subtree that is heavier than the heaviest leaf. Therefore, any such tree will be at one time at the head of the priority queue. Let r be the root node of a heaviest subtree in s_{opt}. From what precedes, there always exists a step t, which is the first step in SplitSubtrees where a node, say r_t, of weight W_r, is the head of PQ at the end of the

ALGORITHM 2: SplitSubtrees(T, p)

```
1  foreach node i do compute $W_i$ (the total processing time ofthe tree rooted at $i$);
2  Initialize priority queue $PQ$ with the tree root;
3  $seqSet \leftarrow \emptyset$;
4  $Cost(0) = W_{root}$;
5  $s \leftarrow 1$;                                              /* splitting rank */
6  while $W_{head(PQ)} > w_{head(PQ)}$ do
7      $node \leftarrow popHead(PQ)$;                                /* Remove $PQ[1]$ */
8      $seqSet \leftarrow seqSet \cup node$;
9      Insert all children of $node$ into priority queue $PQ$;
10     $p' \leftarrow \min(p, |PQ|)$;
11     $LargestSubtrees[s] \leftarrow \{PQ[1], \ldots, PQ[p']\}$;
       /* All nodes not in LargestSubtrees will be processed sequentially.   */
12     $C_{\max}^{\text{ParSubtrees}}[s] = W_{PQ[1]} + \sum_{i \in seqSet} w_i + \sum_{i=p'+1}^{|PQ|} W_{PQ[i]}$;
13     $s \leftarrow s + 1$;
14 Select subtree set $LargestSubtrees[s_{\min}]$ such that $C_{\max}^{\text{ParSubtrees}}[s_{\min}] = \min_{t=0}^{s-1} C_{\max}^{\text{ParSubtrees}}[t]$
   (break ties in favor of smaller $t$) to be processed in parallel;
```

step (r_t is not necessarily equal to r, as there can be more than one subtree of weight W_r). Let s_t be the solution built by SplitSubtrees at the end of step t. By definition of r, there cannot be any leaf node in the entire tree that is heavier than W_r. The cost of the solution s_t is equal to the execution time of the parallel processing of the $\min\{p, |PQ|\}$ subtrees plus the execution time of the sequential processing of the remaining nodes. Therefore, $C_{\max}^{\text{ParSubtrees}}(t) = W_r + Seq(t)$, where $Seq(t)$ is the total weight of the sequential set *seqSet* at step t, denoted $seqSet(t)$, plus the total weight of the surplus subtrees (i.e., of all subtrees in PQ except the p subtrees of largest weights). The cost of s_{opt} is $C^*_{\max} = W_r + Seq(s_{opt})$, given that r is the root of a heaviest subtree of s_{opt} by definition.

SplitSubtrees splits subtrees by nonincreasing weights. Furthermore, by definition of step t, all subtrees split by SplitSubtrees, up to step t included, were subtrees whose weights were strictly greater than W_r. Therefore, because r is the weight of the heaviest subtree in s_{opt}, all subtrees split by SplitSubtrees up to step t included must have been split to obtain the solution s_{opt}. This has several consequences. First, $seqSet(t)$ is a subset of $seqSet(s_{opt})$ because for any solution s, $seqSet(s)$ is the set of all nodes that are roots of subtrees split to obtain the solution s. Second, either a subtree of s_t belongs to s_{opt} or this subtree has been split to obtain s_{opt}. Therefore, the sequential processing of the $\max\{|PQ| - p, 0\}$ exceeding subtrees is no smaller in s_{opt} than in the solution built at step t. It directly follows from the two preceding consequences that $Seq(t) \leq Seq(s_{opt})$. However, s_{opt} and s_t have the same execution time for the parallel phase W_r. It follows that $C_{\max}^{\text{ParSubtrees}}(t) \leq C^*_{\max}$, which is a contradiction to s_{opt}'s shorter processing time. □

Complexity. We first analyze the complexity of SplitSubtrees. Computing the weights W_i costs $O(n)$. Each insertion into PQ costs $O(\log n)$ and calculating $C_{\max}^{\text{ParSubtrees}}(s)$ in each step costs $O(p)$. Given that there are $O(n)$ steps, SplitSubtrees's complexity is $O(n(\log n + p))$. The complexity of the sequential traversal algorithms used in Steps 2 and 3 of ParSubtrees is at most $O(n^2)$ (e.g., Jacquelin et al. [2011] and Liu [1987]), or $O(n \log n)$ if the optimal postorder is sufficient. Thus, the total complexity of ParSubtrees is $O(n^2)$ or $O(n(\log n + p))$, depending on the chosen sequential algorithm.

Memory.

Lemma 5.2. ParSubtrees *is a p-approximation algorithm for peak memory minimization: the peak memory, M, verifies $M \leq pM_{seq}$, where M_{seq} is the memory required for the complete sequential execution.*

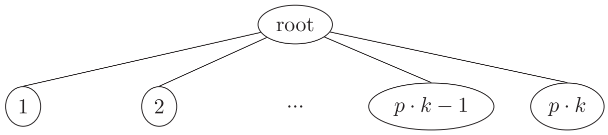

Fig. 6. ParSubtrees is at best a p-approximation for the makespan.

Proof. We first note that during the parallel part of ParSubtrees, the total memory used, M_p, is not more than p times M_{seq}. Indeed, each of the p processors executes a maximal subtree, and the processing of any subtree does not use, obviously, more memory (if done optimally) than the processing of the whole tree. Thus, $M_p \leq p \cdot M_{seq}$.

During the sequential part of ParSubtrees, the memory used, M_S, is bounded by $M_{seq} + \sum_{i \in Q} f_i$, where the second term is for the output files produced by the root nodes of the $q \leq p$ subtrees processed in parallel (Q is the set of the root nodes of the q trees processed in parallel). We now claim that at least two of those subtrees have a common parent. More specifically, let us denote by X the node that was split last (i.e., it was split in the step s_{min}, which is selected at the end of SplitSubtrees). Our claim is that at least two children of X are processed in the parallel part. Before X was split (in step $s_{min} - 1$), the makespan as computed in Step 12 of SplitSubtrees is $C_{\max}(s_{min} - 1) = W_X + Seq(s_{min} - 1)$, where $Seq(s_{min} - 1)$ is the work computed in sequential ($\sum_{i \in seqSet} w_i + \sum_{i=PQ[p'+1]}^{|PQ|} W_i$). Let D denote the set of children of X that are not executed in parallel, then the total weight of their subtrees is $W_D = \sum_{i \in D} W_i$. We now show that if at most one child of X is processed in the parallel part, X was not the node that was split last:

—If exactly one child C of X is processed in the parallel part, then $C_{\max}(s_{min}) = W_{X'} + Seq(s_{min} - 1) + w_X + W_D$, where X' is the new head of the queue, and thus verifies $W_{X'} \geq W_C$. And since $W_X = w_X + W_C + W_D$, we can conclude that $C_{\max}(s_{min}) \geq C_{\max}(s_{min} - 1)$.

—If no child of X is processed in the parallel part, then $C_{\max}(s_{min}) = W_{X'} + Seq(s_{min} - 1) - W_Y + w_X + W_D$, where X' is the new head of the queue and Y is the newly inserted node in the p largest subtrees in the queue. Since $W_{X'} \geq W_Y$ and $W_X = w_X + W_D$, we obtain once again $C_{\max}(s_{min}) \geq C_{\max}(s_{min} - 1)$.

In both cases, we have $C_{\max}(s_{min}) \geq C_{\max}(s_{min} - 1)$, which contradicts the definition of X (the select phase, Step 2 of SplitSubtrees, would have selected step $s_{min} - 1$ rather than step s_{min}). Let us now denote by C_1 and C_2 two children of X that are processed in the parallel phase. Remember that the memory used during the sequential part is bounded by $M_S \leq M_{seq} + f_{C_1} + f_{C_2} + \sum_{i \in Q \setminus \{C_1, C_2\}} f_i$. Since a sequential execution must process node X, we obtain $f_{C_1} + f_{C_2} \leq M_{seq}$. And since $\forall i$, $f_i \leq M_{seq}$, we can bound the memory used during the sequential part by $M_S \leq 2M_{seq} + (p-2)M_{seq} \leq pM_{seq}$.

Furthermore, given that up to p processors work in parallel, each on its own subtree, it is easy to see that this bound is tight if the sequential peak memory can be reached in each subtree. □

Makespan. ParSubtrees delivers a p-approximation algorithm for makespan minimization, and this bound is tight. Because at least one processor is working at any time under ParSubtrees, ParSubtrees delivers, in the worst case, a p-approximation for makespan minimization. To prove that this bound is tight, we consider a tree of height 1 with $p \cdot k$ leaves (a fork), where all execution times are equal to 1 ($\forall i \in T$, $w_i = 1$), and where k is a large integer (this tree is depicted in Figure 6). The optimal makespan for such a tree is $C^*_{\max} = kp/p + 1 = k + 1$ (the leaves are processed in

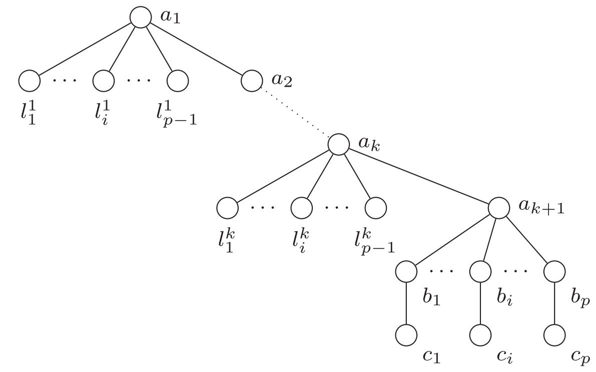

Fig. 7. No memory bound for ParSubtreesOptim.

parallels, in batches of size p, and then the root is processed). With ParSubtrees, p leaves are processed in parallel, then the remaining nodes are processed sequentially. The makespan is thus $C_{\max} = (1 + pk - p) + 1 = p(k-1) + 2$. When k tends to $+\infty$, the ratio between the makespans tends to p.

Optimization. Given the just observed worst case for the makespan, a makespan optimization for ParSubtrees is to allocate all produced subtrees to the p processors instead of only p subtrees. This can be done by ordering the subtrees by nonincreasing total weight and allocating each subtree in turn to the processor with the lowest total weight. Each of the parallel processors executes its subtrees sequentially. This optimized form of the algorithm is named ParSubtreesOptim. Note that this optimization should improve the makespan, but it will likely worsen the peak memory usage.

Indeed, we can prove that ParSubtreesOptim does not have an approximation ratio with respect to memory usage. Consider the tree depicted in Figure 7, assuming the pebble-game model, to be scheduled on p processors. This tree has k levels with $p-1$ leaves at each level, and an additional level with p chains of length 2, and includes a total of $(k+2)p+1$ nodes. The algorithm SplitSubtrees will split the subtrees rooted at a_1, then a_2, and so forth, until it reaches a_{k+1}, and then $b_1, \ldots, b_p$. We first prove that SplitSubtrees will select a splitting where a_{k+1} is split. By contradiction, assume this is not the case—in other words, the selected splitting contains a subtree rooted at a_j among the *LargestSubtrees*. This is obviously the largest subtree, and only $p-1$ other subtrees can be processed in parallel in ParSubtrees, each of them contains a single leaf. Thus, the makespan of this splitting is $(k+1)p+2$. On the contrary, splitting the subtree rooted at a_{k+1} (but not the ones below) provides a solution with makespan $kp+3$, which is smaller as soon as $p \geq 2$. Thus, SplitSubtrees selects a splitting that splits the subtree rooted at a_{k+1}.

Furthermore, the optimal memory usage to compute this tree is $p+1$ (this is achieved with a sequential postorder traversal scheduling the tree from right to left). ParSubtreesOptim schedules all leaves l_i^j in parallel on all p processors, and the internal nodes $a_1, \ldots, a_k$ are started only once all of these leaves are finished. After completing all of the leaves, the memory usage is at least $k(p-1)$. When k tends to $+\infty$, the ratio between the memory requirements also tends to $+\infty$.

5.2. List Scheduling Algorithms

ParSubtrees is a high-level algorithm employing sequential memory-optimized algorithms. An alternative, explored in this section, is to design algorithms that directly work on the tree in parallel. We first present two such algorithms that are event-based list scheduling algorithms [Hwang et al. 1989]. One of the strong points of list scheduling algorithms is that they are $(2 - \frac{1}{p})$-approximation algorithms for makespan minimization [Graham 1966].

Algorithm 3 outlines a generic list scheduling, driven by node finish time events. At each event at least one node has finished, so at least one processor is available for processing nodes. Each available processor is given the respective head node of the priority queue. The priority of nodes is given by the total order O, a parameter to Algorithm 3.

ALGORITHM 3: List scheduling(T, p, O)

```
1  Insert leaves in priority queue $PQ$ according to order $O$;
2  $eventSet \leftarrow \{0\}$;                                   /* ascending order */
3  while $eventSet \neq \emptyset$ do                             /* event: node finishes */
4      $t \leftarrow popHead(eventSet)$;
5      $NewReadyNodes \leftarrow$ set of nodes whose last children completed at time $t$;
6      Insert nodes from $NewReadyNodes$ in $PQ$ according to order $O$;
7      $\mathcal{P} \leftarrow$ available processors at time $t$;
8      while $\mathcal{P} \neq \emptyset$ and $PQ \neq \emptyset$ do
9          $proc \leftarrow popHead(\mathcal{P})$;
10         $node \leftarrow popHead(PQ)$;
11         Assign $node$ to $proc$;
12         $eventSet \leftarrow eventSet \cup finishTime(node)$;
```

5.2.1. Heuristic ParInnerFirst. From the study of the sequential case, one knows that a *postorder* traversal, while not optimal for all instances, provides good results [Jacquelin et al. 2011]. Our intention is to extend the principle of postorder traversal to the parallel processing. For the first heuristic, called ParInnerFirst, the priority queue uses the following ordering O: (1) inner nodes in an arbitrary order, and (2) leaf nodes ordered according to a given postorder traversal. Although any postorder may be used to order the leaves, it makes heuristic sense to choose an optimal sequential postorder so that memory consumption can be minimized (this is what is done in the experimental evaluation discussed later). We do not further define the order of inner nodes, as it has absolutely no impact. Indeed, because we target the processing of tree-shaped task graphs, the processing of a node makes at most one new inner node available, and the processing of this new inner node can start right away on the processor that freed it by completing the processing of its last unprocessed child.

Complexity. The complexity of ParInnerFirst is that of determining the input order O and that of the list scheduling. Computing the optimal sequential postorder is $O(n \log n)$ [Liu 1986]. In the list scheduling algorithm, there are $O(n)$ events and n nodes are inserted and retrieved from PQ. An insertion into PQ costs $O(\log n)$, so the list scheduling complexity is $O(n \log n)$. Hence, the total complexity is also $O(n \log n)$.

Memory. ParInnerFirst is not an approximation algorithm with respect to peak memory usage. This is derived considering the tree in Figure 8. All output files have size 1, and the execution files have size 0 ($\forall i \in T : f_i = 1, n_i = 0$). Under an optimal sequential processing, leaves are processed in a deepest first order. The resulting optimal memory requirement is $M_{seq} = p + 1$, reached when processing a join node. With p processors, all leaves have been processed at the time the first join node $(k - 1)$ can be executed. (The longest chain has length $2k - 2$.) At that time, there are $(k - 1) \cdot (p - 1) + 1$ files in memory. When k tends to $+\infty$, the ratio between the memory requirements also tends to $+\infty$.

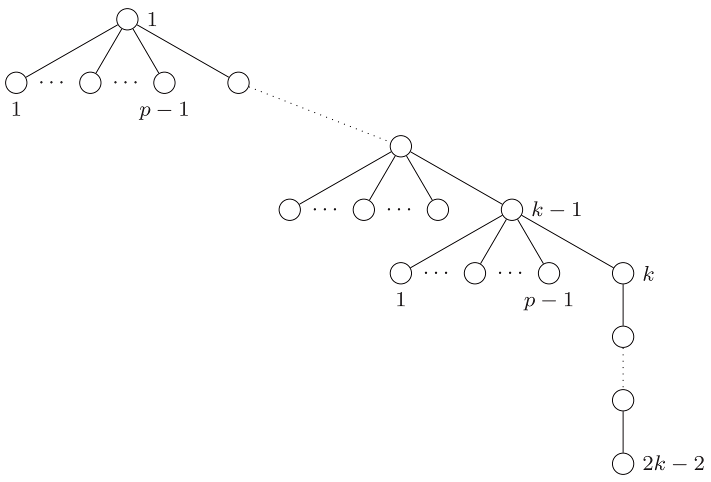

Fig. 8. No memory bound for ParInnerFirst.

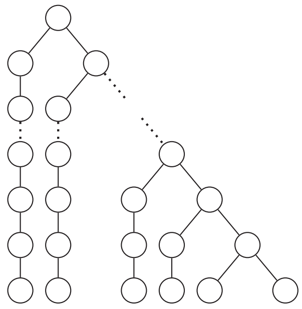

Fig. 9. Tree with long chains.

5.2.2. Heuristic ParDeepestFirst. The previous heuristic, ParInnerFirst, tries to take advantage of the memory performance of optimal sequential postorders. Going in the opposite direction, another heuristic objective can be the minimization of the makespan. For trees, an inner node depends on all nodes in the subtree it defines. Therefore, it makes heuristic sense to try to process the deepest nodes first to try to reduce any possible waiting time. For the parallel processing of a tree, the most meaningful definition of the depth of a node i is the w-weighted length of the path from i to the root of the tree, including w_i (therefore, the depth of node i is equal to its top-level plus w_i [Casanova et al. 2008]). A deepest node in the tree is a deepest node in a critical path of the tree.

ParDeepestFirst is our proposed list scheduling deepest-first heuristic. ParDeepestFirst is defined by Algorithm 3, called with the following node ordering O: nodes are ordered according to their depths, and in case of ties, inner nodes have priority over leaf nodes; remaining ties are broken according to an optimal sequential postorder.

Complexity. The complexity is the same as for ParInnerFirst, namely $O(n \log n)$. See ParInnerFirst's complexity analysis.

Memory. The memory required by ParDeepestFirst is unbounded with respect to the optimal sequential memory M_{seq}. Consider the tree in Figure 9 with many long chains, assuming the pebble-game model (i.e., $\forall i \in T : f_i = 1, n_i = 0, w_i = 1$). The optimal sequential memory requirement is 3. The memory usage of ParDeepestFirst will be proportional to the number of leaves, because they are all at the same depth, the deepest one. As we can build a tree like the one in Figure 9 for any predefined number of chains, the ratio between the memory required by ParDeepestFirst and the optimal one is unbounded.

5.3. Memory-Constrained Heuristics

From the analysis of the three algorithms presented so far, we have seen that only ParSubtrees gives a guaranteed bound on the required peak memory. The memory behavior of the two other algorithms, ParInnerFirst and ParDeepestFirst, will be analyzed in the experimental evaluation presented in Section 6. In a practical setting, it might be very desirable to have a strictly bounded memory consumption to be certain that the algorithm can be executed with the available limited memory. In fact, a guaranteed upper limit might be more important than a good average behavior, as the system needs to be equipped with sufficient memory for the worst case. ParSubtrees's guarantee of at most p times the optimal sequential memory seems high, and thus an obvious goal would be to have a heuristic that minimizes the makespan while keeping the peak memory usage below a given bound. To approach this goal, we first study how to limit the memory consumption of ParInnerFirst and ParDeepestFirst. Our study relies on some reduction property on trees, presented in Section 5.3.1, where we also show how to transform any tree into one that satisfies the reduction property. We then develop memory-constrained versions of ParInnerFirst and ParDeepestFirst (Section 5.3.2). The memory bounds achieved by these new variants are rather lax. Therefore, we design our last heuristic, MemBookingInnerFirst, with stronger memory properties (Section 5.3.3). In the experimental section (Section 6), we will show that these three heuristics achieve different trade-offs between makespan and memory usage.

5.3.1. Simplifying Tree Properties. To design our memory-constrained heuristics, we make two simplifying assumptions. First, the considered trees do not have any execution files. In other words, we assume that for any task i, $n_i = 0$.

Eliminating execution files. To still be able to deal with general trees, we can transform any tree T with execution files into a strictly equivalent tree T' where all execution files have a null size. Let i be any node of T. We add to i a new leaf child i', whose execution time is null ($w_{i'} = 0$), whose execution file is of null size ($n_{i'} = 0$), and whose output file has size n_i ($f_{i'} = n_i$). Then, we set n_i to 0. Any schedule S for the original tree T can be easily transformed into a schedule S' for the new tree T' with the exact same memory and execution-time characteristics: S' schedules a node from T at the same time than S, and a node i from $T' \setminus T$ at the same time than the father of i is scheduled by S (because i has a null execution time).

The second simplifying assumption is that all considered trees are reduction trees.

Definition 5.3 (*Reduction Tree*). A task tree is a *reduction tree* if the size of the output file of any inner node i is not more than the sum of its input files:

$$f_i \le \sum_{j \in Children(i)} f_j. \qquad (1)$$

This reduction property is very useful, because it implies that executing an inner node does not increase the amount of memory needed (this will be used, for instance, later in Theorem 5.4).

For convenience, we sometimes use the following notation to denote the sum of the sizes of the input files of an inner node i:

$$inputs(i) = \sum_{j \in Children(i)} f_j.$$

We now show how general trees can be transformed into reduction trees.

Turning trees into reduction trees. We can transform any tree T that does not satisfy the reduction property stated by Equation (1) into a tree where each (inner) node satisfies it. Let i be any inner node of T. We add to i a new leaf child i', whose execution time is null ($w_{i'} = 0$), whose execution file is of null size ($n_{i'} = 0$), and whose output file has the following size:

$$f_{i'} = \max\left\{0, f_i - \left(\sum_{j \in Children(i)} f_j\right)\right\} = \max\{0, f_i - inputs(i)\}.$$

The new tree is not equivalent to the original one. Let us consider an inner node i that did not satisfy the reduction property. Then, $f_{i'} = f_i - inputs(i) > 0$. The memory used to execute node i in the tree T is $inputs(i) + n_i + f_i$. In the new tree, the memory needed to execute this node is $(inputs(i) + (f_i - inputs(i)) + n_i + f_i > inputs(i) + n_i + f_i$. Any schedule of the original tree can be transformed into a schedule of the new tree with an increase of the memory usage bounded by

$$p \times \max_i\{0, f_i - inputs(i)\}.$$

Obviously, a more clever approach is to transform a tree first into a tree without execution files and then to transform the new tree into a tree with the reduction property. Under this approach, the increase of the memory usage is bounded by

$$p \times \max_i\{0, f_i - inputs(i) - n_i\}.$$

Transforming schedules. The algorithms proposed in the following sections produce schedules for reduction trees without execution files, which might have been created from general trees that do not possess our simplifying properties. The schedule S' produced by an algorithm for a reduction tree without execution files T' can readily be transformed into a schedule S for the original tree T. To create schedule S, we simply remove all (leaf) nodes from the schedule S' that were introduced in the simplification transform ($i' \in T' \setminus T$). Because those nodes have zero processing time ($\forall i' \in T' \setminus T$: $w_{i'} = 0$), there is no impact on the ordering and on the starting time of the other nodes of T. In terms of memory consumption, the peak memory for schedule S is never higher than that for schedule S'. A leaf i' that was added to eliminate an execution file might use memory earlier in S' than the execution file n_i in S, but it is the same amount and freed at the same time. In terms of leaf nodes introduced to enforce the reduction property, they might only increase the memory needed for tree T' (as discussed earlier); hence, removing these nodes cannot increase the peak memory needed for schedule S. In summary, the schedule S for tree T has the same makespan as S' and a peak memory that is not greater than that of S'.

5.3.2. Memory-Constrained PARINNERFIRST *and* PARDEEPESTFIRST. Both PARINNERFIRST and PARDEEPESTFIRST are based on the list scheduling approach presented in Algorithm 3. To achieve a memory-constrained version of these algorithms for reduction trees, we modify Algorithm 3 to obtain Algorithm 4. The code common to both algorithms is shown in gray in Algorithm 4, and the new code is printed in black.

We use the same event concept as previously. However, we only start processing a node if (1) it is an inner node, or (2) it is a leaf node and the current memory consumption plus the leaf's output file (f_c) is less than the amount M of available memory. Once a node is encountered that fulfills neither of these conditions, the node assignment is stopped ($\mathcal{P} \leftarrow \emptyset$) until the next event. Therefore, Algorithm 4 may deliberately keep some processors idle when there are available tasks and thus does not necessarily produce a list schedule (hence, the name of "pseudo" list schedules). Subsequently, the only approximation guarantee on the makespan produced by these heuristics is that they are p-approximations, the worst case for heuristics that always use at least one processor at any time before the entire processing completes.

Algorithm 4 may be executed with any sequential node ordering O and any memory bound M as long as the peak memory usage of the corresponding sequential algorithm with the same node order O is no greater than M. From Algorithm 4, we design two new heuristics: ParInnerFirstMemLimit and ParDeepestFirstMemLimit. ParInnerFirstMemLimit uses, for the order O, an optimal sequential postorder with respect to peak memory usage. For ParDeepestFirstMemLimit, nodes are ordered by nonincreasing depths, and in case of ties, inner nodes have priority over leaf nodes; remaining ties are broken according to an optimal sequential postorder.

ALGORITHM 4: Pseudo list scheduling with memory limit (T, p, O, M)

```
1  Insert leaves in priority queue $PQ$ according to order $O$;
2  $eventSet \leftarrow \{0\}$;                                   /* ascending order */
3  $M_{used} \leftarrow 0$;                                       /* amount of memory used */
4  while $eventSet \neq \emptyset$ do                             /* event: node finishes */
5      $t \leftarrow popHead(eventSet)$;
6      $NewReadyNodes \leftarrow$ set of nodes whose last children completed at time $t$;
7      Insert nodes from $NewReadyNodes$ in $PQ$ according to order $O$;
8      $\mathcal{P} \leftarrow$ available processors at time $t$;
9      $Done \leftarrow$ nodes completed at time $t$;
10     $M_{used} \leftarrow M_{used} - \sum_{j \in Done} inputs(j)$;
11     while $\mathcal{P} \neq \emptyset$ and $PQ \neq \emptyset$ do
12         $c \leftarrow head(PQ)$;
13         if $|Children(c)| > 0$ or $M_{used} + f_c \leq M$ then
14             $M_{used} \leftarrow M_{used} + f_c$;
15             $proc \leftarrow popHead(\mathcal{P})$;
16             $node \leftarrow popHead(PQ)$;
17             Assign $node$ to $proc$;
18             $eventSet \leftarrow eventSet \cup finishTime(node)$;
19         else
20             $\mathcal{P} \leftarrow \emptyset$
```

Theorem 5.4. *The peak memory requirement of Algorithm 4 for a reduction tree without execution files processed with a memory bound M and a node order O is at most* $2M$*, if* $M \geq M_{seq}$*, where* M_{seq} *is the peak memory usage of the corresponding sequential algorithm with the same node order O.*

Proof. We first show that the required memory never exceeds $2M$ and then show that the algorithms completely process the considered tree T.

We analyze the memory usage at the time a new candidate node c is considered for execution (line 12 of Algorithm 4). The amount of currently used memory is then $M_{used} = In_{\text{IN}} + Out_{\text{IN}} + Out_{\text{LF}} + InIdle$, where

—In_{IN} is the size of the input files of the currently processed inner nodes,
—Out_{IN} is the size of the output files of the currently processed inner nodes,
—Out_{LF} is the size of the output files of the currently processed leaves, and
—$InIdle$ is the size of the input files stored in memory but not currently used (because they are input files of inner nodes that are not yet ready).

There are two cases—the candidate node c can be either a leaf node or an inner node:

(1) c is a leaf node. The processing of a leaf node only starts if $M_{used} + f_c \leq M$. Therefore, the processing of a leaf node never provokes the violation of the memory bound of M, and thus a fortiori, of a memory limit of $2M$.
(2) c is an inner node. The processing of a candidate inner node always starts right away, regardless of the amount of available memory. When the processing of c starts, the amount of required memory becomes $M_{new} = In_{\text{IN}} + Out_{\text{IN}} + Out_{\text{LF}} + InIdle + f_c$. T is by hypothesis a reduction tree. Therefore, the size of the output file f_c does not exceed $InIdle$—that is, the size of all possible input files stored in memory right before the start of the processing of inner node c, but not used at that time, because this includes all input files of inner node c. In addition, the total size of the output files of the processed inner nodes, Out_{IN}, cannot exceed the total size of the input files of the processed inner nodes, In_{IN}. Therefore, $M_{new} = In_{\text{IN}} + Out_{\text{IN}} + Out_{\text{LF}} + InIdle + f_c \leq In_{\text{IN}} + Out_{\text{IN}} + Out_{\text{LF}} + 2InIdle \leq 2In_{\text{IN}} + Out_{\text{LF}} + 2InIdle \leq 2(In_{\text{IN}} + Out_{\text{LF}} + InIdle)$.
 So the new memory requirement M_{new} is not greater than twice the memory occupied by all *input* files and all *output* files of leaf nodes. Because the tree is a reduction tree by hypothesis, executing an inner node never increases the total size of all input files and all output files of leaves. This can only happen by starting a leaf, but that is not done if it would exceed the required memory M. Therefore, $In_{\text{IN}} + Out_{\text{LF}} + InIdle$ never exceeds M and $M_{new} \leq 2M$.

We now prove that when the algorithm ends, the entire input tree has been processed. We reason by contradiction and assume that this is not the case. Ready inner nodes are processed without checking the amount of available memory. Therefore, when the algorithm terminates without having completed the processing of the tree, *eventSet* is empty but some leaves have not been processed. Then, let l be the first unprocessed leaf, according to the order O. At the time Algorithm 4 terminates, it has processed exactly the same leaves as the sequential algorithm when it starts processing leaf l. Because *eventSet* is empty, there are no remaining ready inner nodes and no node is processed at the time of the algorithm termination. Because of the hypothesis that T is a reduction tree, the amount of available memory when Algorithm 4 terminates is not smaller than the amount of available memory under the sequential algorithm right before it starts processing leaf l. Because the sequential algorithm can process the whole tree with a peak memory usage of $M_{seq} \leq M$, the processing of leaf l can be started by Algorithm 4. This contradicts the assumption of early termination. □

We define a variant ParDeepestFirstMemLimitOptim of ParDeepestFirstMemLimit, and a variant ParInnerFirstMemLimitOptim of ParInnerFirstMemLimit, by being more aggressive about starting leaves. Instead of checking for the condition $M_{used} + f_c \leq M$ before starting a leaf node c (line 13 of Algorithm 4), it is in fact sufficient to check that $In_{\text{IN}} + \frac{1}{2}Out_{\text{LF}} + InIdle + f_c \leq M$ (using the notation of the proof of Theorem 5.4). For Case (1) of the proof, one just needs to remark that after leaf c is started, $M_{new} = In_{\text{IN}} + Out_{\text{IN}} + Out_{\text{LF}} + InIdle + f_c$. Then, because the tree is a reduction tree, $Out_{\text{IN}} \leq In_{\text{IN}}$. Therefore, $M_{new} \leq 2In_{\text{IN}} + Out_{\text{LF}} + f_c + InIdle \leq 2In_{\text{IN}} + Out_{\text{LF}} + 2f_c + 2InIdle$, which, in turn, is no greater than $2M$ because of the new condition. The modified condition has

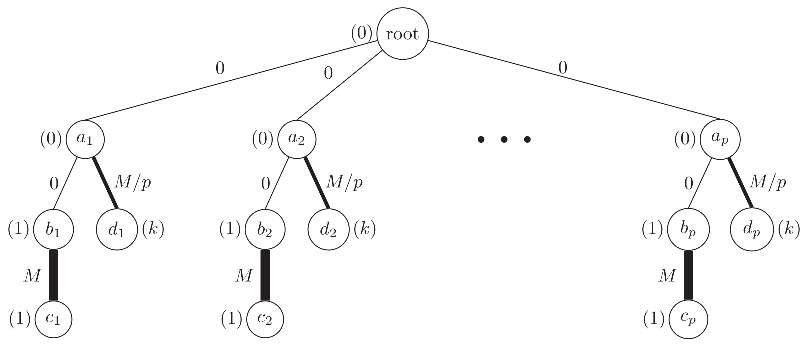

Fig. 10. Tree used to establish the worst-case performance of most memory-constrained heuristics. Node labels in parentheses are processing times; edge labels are memory weights. All nodes have null-size processing files.

no impact on the study of Case (2), because the inequality $In_{\text{IN}} + \frac{1}{2}Out_{\text{LF}} + InIdle \leq M$ is sufficient to conclude that case.

Memory. Theorem 5.4 establishes that the peak memory required by ParInnerFirstMemLimit and ParDeepestFirstMemLimit with p processors is at most twice that of their sequential execution ($p = 1$) with the same order. It should be noted that this peak requirement M_{seq} does not correspond in general to the memory requirement of an *optimal* sequential algorithm. In particular, the sequential execution of ParInnerFirstMemLimit corresponds to a postorder traversal, which is not optimal for all instances but generally provides good results [Jacquelin et al. 2011]. We propose using ParInnerFirstMemLimit with a node order O that corresponds to an optimal sequential postorder, such as with Liu [1986]. The memory requirement of the sequential ParDeepestFirstMemLimit is unbounded compared to the optimal sequential memory requirement, because the same arguments apply as the ones discussed for ParDeepestFirst in Section 5.2.2.

Makespan. We have stated that the preceding heuristics are p-approximation algorithms for makespan minimization. The following lemma refines this result.

Lemma 5.5. ParInnerFirstMemLimit *and* ParInnerFirstMemLimitOptim *are both p-approximation algorithms for makespan minimization, and this bound is tight.*

Proof. At any time under ParInnerFirstMemLimit and ParInnerFirstMemLimitOptim, there is at least one processor that is not idle and is processing a task. Therefore, both ParInnerFirstMemLimit and ParInnerFirstMemLimitOptim are p-approximation algorithms for makespan minimization. We establish that this bound is tight by studying the tree in Figure 10. This tree can be processed with a peak memory usage of M. We assume that ParInnerFirstMemLimit is called with this memory limit. The key observation is that in any schedule, among the three descendants of an a_i node, the nodes c_i and b_i must be processed before the d_i node: otherwise, keeping in memory the output file of size M/p of node d_i makes it impossible to start processing the leaf node c_i because of its output file of size M. And since under ParInnerFirstMemLimit leaf nodes are processed according to a postorder traversal, the processing of the subtrees is sequentialized and the overall processing takes time $p(2 + k)$. On the other hand, with respect to the makespan, it would be better to first sequentially process in that order c_1, b_1, c_2, b_2, $\ldots$, c_p, and b_p, which would take a time $2p$, and then process in parallel the d_i's for an overall makespan of $2p + k$. Hence, in this example, the approximation ratio of ParInnerFirstMemLimit is no smaller than $\frac{p(2+k)}{2p+k}$, which tends to p when k tends to infinity. □

We do not have a similar result for the memory-limited deepest first algorithms, as already the sequential traversal with these algorithms (which determines the given memory limit) can require significantly more memory than a postorder traversal. For the example in Figure 10, the minimum sequential memory for a deepest first traversal is equal to pM. The additional memory buys a lot of freedom for ParDeepestFirstMemLimit and makes the comparison harder.

5.3.3. Memory Booking Heuristic MemBookingInnerFirst*.* The two heuristics described in the previous section satisfy an achievable memory bound, M, in a relaxed way: the guarantee is that they never use more than twice the memory limit. Here, we aim at designing a heuristic that satisfies an achievable memory bound M in the strong sense: the heuristic never uses more than a memory of M.

To achieve such a goal, we want to ensure that whenever an inner node i becomes ready, there is enough memory to process it. Therefore, we book in advance some memory for its later processing. Our goal is to book as little memory as possible, and to do so as late as possible. The algorithm then relies on a sequential postorder schedule, denoted PO: for any node k in the task graph, $PO(k)$ denotes the step at which node k is executed under PO. Let j be the last child of i to be processed. If the total size of the input files of j is larger than (or equal to) f_i, then only that last child will book some memory for node i. In this case, (part of) the memory that was used to store the input files of j will be used for f_i. If the total size of the input files of j is smaller than f_i, then the second to last child of i will also have to book some memory for f_i, and so on. The following recursive formula states the amount of memory $Contrib[j]$ a child j has to book for its parent i:

$$Contrib[j] = \min\left(inputs(j),\ f_i - \sum_{\substack{j' \in Children(i) \\ PO(j') > PO(j)}} Contrib[j'] \right).$$

If j is a leaf, it may also have to book some memory for its parent. However, the behavior for leaves is quite different than for inner nodes. A leaf node cannot transfer some of the memory used for its input files (because it does not have any) to its parent for its parent output file. Therefore, the memory booked by a leaf node may not be available at the time of the booking. However, this memory will eventually become available (after some inner nodes are processed); booking the memory prevents the algorithm from starting the next leaf if it would use too much memory: this ensures that the algorithm completes the processing without violating the memory bound. The contribution of a leaf j for its parent i is

$$Contrib[j] = f_i - \sum_{\substack{j' \in Children(i) \\ PO(j') > PO(j)}} Contrib[j'].$$

Note that the value of $Contrib$ for each node can be computed before starting the algorithm in a simple tree traversal. Using these formulas, we are able to guarantee that enough memory is booked for each inner node i:

$$\sum_{j \in Children(i)} Contrib[j] = f_i.$$

ALGORITHM 5: MEMBOOKINGINNERFIRST(T, p, PO, M)

```
Input: tree $T$, number of processor $p$, postorder $PO$, memory limit $M$ (not smaller than
       the peak memory of the sequential traversal defined by $PO$)
1  foreach node $i$ do $Booked[i] \leftarrow 0$;
2  $M_{used} \leftarrow 0$;
3  while the whole tree is not processed do
4      Wait for an event (task finish time or starting point of the algorithm);
5      foreach finished non–leaf node $j$ with parent $i$ do
6          $M_{used} \leftarrow M_{used} - inputs(j)$;
7          $Booked[i] \leftarrow Booked[i] + Contrib[j]$;
8      $NewReadyNodes \leftarrow$ set of nodes whose last children completed at event;
9      Insert nodes from $NewReadyNodes$ in $PQ$ according to order $O$;
10     $WaitForNextTermination \leftarrow false$;
11     while $WaitForNextTermination = false$ and there is an available processor $P_u$ and $PQ$
       is not empty do
12         $j \leftarrow pop(PQ)$;
13         if $j$ is an inner node and $M_{used} + f_j \leq M$ then
14             $M_{used} \leftarrow M_{used} + f_j$;
15             $Booked[j] \leftarrow 0$;
16             Make $P_u$ process $j$;
17         else if $j$ is a leaf and $M_{used} + f_j + \sum_{k \notin Ancestors(j)} Booked[k] \leq M$ then
18             $M_{used} \leftarrow M_{used} + f_j$;
19             $Booked[\text{parent of } j] \leftarrow Booked[\text{parent of } j] + Contrib[j]$;
20             Make $P_u$ process $j$;
21         else
22             $push(j, PQ)$;
23             $WaitForNextTermination \leftarrow true$;
```

Using these definitions, we design a new heuristic, MEMBOOKINGINNERFIRST, which is described in Algorithm 5. In this algorithm, $Booked[i]$ denotes the amount of memory currently booked for the processing of an inner node i. We make use of a new notation: we denote by $Ancestors(i)$ the set of nodes on the path from i to the root node (excluding i itself)—that is, all ancestors of i.

Note that contrarily to PARINNERFIRSTMEMLIMIT, MEMBOOKINGINNERFIRST does not guarantee that there is always enough memory available to process an inner node i as soon as it becomes ready. This is why Lemma 5.6 only guarantees that an inner node i will *eventually* be processed if a leaf j with $PO(j) > PO(i)$ is started by MEMBOOKINGINNERFIRST. This corresponds to the case when some leaf j, which is processed after i in the sequential postorder PO, is started earlier than i in a parallel schedule: this happens when some processor is available but task i is not ready yet, as its children have not yet completed.

LEMMA 5.6. *Consider any inner node i. If some leaf j with $PO(j) > PO(i)$ has been started by Algorithm 5, then at some point there will be enough memory to process i.*

PROOF. By contradiction, assume that an available inner node i can never be processed because of memory constraints—that is, Algorithm 5 stops without processing i, and some leaf j with $PO(j) > PO(i)$ has been started (in case of several such leaves, we consider the one with largest $PO(j)$). Note that i cannot be a parent of j (otherwise, we would have $PO(i) > PO(j)$). We consider the amount $A = M - M_{used} - \sum_{k \notin Ancestors(j)} Booked[k]$ and its evolution. Before starting j, we check that $A \geq f_j$. When starting j, the amount of available memory is decreased by f_j, and thus we have $A \geq 0$. The following events may happen after the beginning of j:

—Some inner node u not in *Ancestors*(j) is terminated. Let us call v its parent. When u completes, M_{used} decreases by *inputs*(u), whereas *Booked*$[v]$ increases by *Contrib*$[u] \leq$ *inputs*(u). Thus, A does not decrease.
—Some inner node k not in *Ancestors*(j) is started. In that case, the booked memory f_k is traded for used memory, and the total memory amount A is preserved.
—An inner node u in *Ancestors*(j) is started. In this case, the amount of available memory may temporarily decrease. However, because of the reduction property, the amount of memory freed when u completes is not smaller than the amount of additional memory temporarily used for the processing of u.
—A leaf node has completed: this modifies neither the amount of available or booked memory, and so A is left unchanged.

Therefore, when the algorithm stops with i available, $A \geq 0$. Thus, $M - M_{used} \geq$ *Booked*$[i] = f_i$: there is enough memory to process i. □

Using the previous lemma, we now prove Algorithm MemBookingInnerFirst.

Theorem 5.7. MemBookingInnerFirst *called with a postorder PO and a memory bound M processes the whole tree with memory M if M is not smaller than the peak memory of the sequential traversal defined by PO.*

Proof. By contradiction, assume that the algorithm stops while some nodes are unprocessed. We consider two cases:

—There is at least one available unprocessed inner node i (if there are several, we choose the one with the smallest *PO* value). Consider the step *PO*(i) when this node i is processed in the sequential postorder schedule. At this time, the set $\mathcal{S}$ of the leaves processed by the sequential postorder is exactly the set of the leaves j such that *PO*$(j) <$ *PO*(i). Thanks to Lemma 5.6, we know that MemBookingInnerFirst has not processed any leaf j with *PO*$(j) >$ *PO*(i). Therefore, the set of the leaves processed by MemBookingInnerFirst is a subset of $\mathcal{S}$. Node i being available, MemBookingInnerFirst has processed all of the leaves in the subtree *ST* rooted at i. MemBookingInnerFirst cannot start a leaf k if a leaf j with *PO*$(j) <$ *PO*(k) has not been started. Therefore, any leaf that precedes any leaf of *ST* in the postorder has also been processed by MemBookingInnerFirst. By definition of a postorder, there is no leaf that does not belong to *ST* that is scheduled after the first leaf of *ST* and before i. Therefore, MemBookingInnerFirst has processed the exact same set of leaves as the sequential postorder at step *PO*(i). We now prove that the same set of inner nodes have been processed by both algorithms:
  —Assume that an inner node k has been processed by MemBookingInnerFirst but not by the sequential postorder at time *PO*(i). Since k has not yet been processed by the sequential postorder, *PO*$(k) >$ *PO*(i). Since no leaf j with *PO*$(j) >$ *PO*(i) has been processed by MemBookingInnerFirst, since *PO*$(k) >$ *PO*(i), and since *PO* is a postorder, then k can only be a parent of i, which contradicts the fact that i is not processed.
  —Assume that an inner node k has been processed by the sequential postorder at time *PO*(i) but not by MemBookingInnerFirst. Since it has been processed before i in the sequential postorder, *PO*$(k) <$ *PO*(i). This node, or one of its inner node predecessor, must be available in MemBookingInnerFirst (note that it cannot be a leaf, as all leaves with *PO* values smaller than *PO*(i) are already processed). This contradicts the fact that i is the available inner node with smallest *PO* value.

Thus, there is no difference in the state of the sequential postorder when it starts i and MEMBOOKINGINNERFIRST when it stops, including in the amount of available memory. This contradicts the fact that i cannot be started because of memory issues.

—There is no unprocessed available inner node. Thus, some leaf is available and cannot be processed. Let j be the first of these leaves according to PO. None of the inner nodes for which some memory has been booked is available, and thus they are all parents of j (because PO is a postorder and because the processing of all leaves that precede j in PO has been completed). Thus, the memory condition that prevents j to be executed can be rewritten: $M - M_{used} < f_j$. However, since no inner node is available, this is the same situation as right before j is processed in the sequential postorder, which contradicts the fact that j can be processed in the sequential postorder. □

LEMMA 5.8. MEMBOOKINGINNERFIRST *is a p-approximation algorithm for makespan minimization, and this bound is tight.*

This result is proved following the exact same arguments than for the bound on the performance of PARINNERFIRSTMEMLIMIT, including the tree in Figure 10.

Complexity. Algorithm 5 can be implemented with the same complexity as the other heuristics, namely $O(n\log(n))$ (which comes from the management of the PQ queue). The only operations added to this algorithm that could increase this complexity is the test executed on line 17 to make sure that a new leaf can be started—that is, the computation of $\sum_{k\notin Ancestors(j)} Booked[k]$ for each leaf might take $O(n^2)$ time if not done carefully. However, it is possible to avoid recomputing the values too many times. We first note the following property: when leaf j has not been started,

$$\sum_{k\notin Ancestors(j)} Booked[k] = \sum_{PO(k)<PO(j)} Booked[k].$$

Indeed, if leaf j has not been started, the postorder property ensures that any $k \notin Ancestors(j)$ with $PO(k) \geq PO(j)$ has $Booked[k] = 0$, because none of its children have started their execution.

For an efficient implementation, we keep a record of $R = \sum_{PO(k)<PO(j)} Booked[k]$ for the leaf j that was tested on the last execution of line 17. To keep this record, it is enough to

—decrease R by $Booked[i]$ each time an inner node i with $PO(i) < PO(j)$ begins execution,
—increase R by $Contrib[i]$ each time an inner node i with $PO(i) < PO(j)$ finishes,
—and increase R by $\sum_{PO(j)\leq PO(k)<PO(j')} Booked[k]$ if a new leaf j' is being considered in the test of line 17.

In total, the number of updates to R over the course of the whole algorithm is bounded by $2n$: each *Contrib* value is added at most once to R, and each *Booked* value is subtracted at most once. Furthermore, the cost of computing the sums $\sum_{PO(j)\leq PO(k)<PO(j')} Booked[k]$ is also bounded by n since each node is considered only once. Hence, these updates do not increase the total complexity of $O(n\log(n))$ of the whole algorithm.

6. EXPERIMENTAL VALIDATION

In this section, we experimentally compare the heuristics proposed in the previous section and compare their performance to lower bounds.

Table I. Approximation Ratios of the Different Heuristics

	Approximation Ratios with Respect to	
Heuristics	Memory	Makespan
ParSubtrees	$\leq p$	$= p$
ParSubtreesOptim	$+\infty$	$\leq p$
ParInnerFirst	$+\infty$	$= 2 - \frac{1}{p}$
ParDeepestFirst	$+\infty$	$= 2 - \frac{1}{p}$
ParInnerFirstMemLimit	$\leq 2 \times \frac{M}{M_{opt}}$	$= p$
ParInnerFirstMemLimitOptim	$\leq 2 \times \frac{M}{M_{opt}}$	$= p$
ParDeepestFirstMemLimit	$\leq 2 \times \frac{M}{M_{opt}}$	$\leq p$
ParInnerFirstMemLimitOptim	$\leq 2 \times \frac{M}{M_{opt}}$	$\leq p$
MemBookingInnerFirst	$= \frac{M}{M_{opt}}$	$= p$

Note: "$\leq \alpha$" and "$= \alpha$" mean that the heuristic is an α-approximation algorithm; the second expression means that the bound is tight. For memory-constrained heuristics, the ratio is expressed as a function of the optimal peak memory M_{opt} and of the memory budget M. M is, by assumption, greater than the peak memory usage of the order O used by the studied heuristic ($M \geq M_{seq} \geq M_{opt}$).

6.1. Setup

All heuristics have been implemented in C. Special care has been devoted to the implementation to avoid complexity issues. Especially, priority queues have been implemented using binary heap to allow for $O(\log n)$ insertion and minimum extraction. We have also implemented Liu's algorithm [Liu 1987] to obtain the minimum sequential memory peak, which is used as a lower bound on memory for comparing the heuristics.

6.2. Dataset

The dataset 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 chosen matrices satisfy the following assertions: not binary, not corresponding to a graph, square, having a symmetric pattern, a number of rows between 20,000 and 2,000,000, a number of nonzeros per row at least equal to 2.5, and a number of nonzeros at most equal to 5,000,000; each chosen matrix has the largest number of nonzeros among the matrices in its group satisfying the previous assertions. With these criteria, we automatically select a set of medium to large matrices from different application domains with a nontrivial number of nonzeros. At the time of testing, there were 76 matrices satisfying these properties. We first order the matrices using MeTiS [Karypis and Kumar 1998] (through the MeshPart toolbox [Gilbert et al. 1998]) and `amd` (available in Matlab), then build the corresponding elimination trees using the `symbfact` routine of Matlab. We also perform a relaxed node amalgamation [Liu 1992] on these elimination trees to create assembly trees. We have created a large set of instances by allowing 1, 2, 4, and 16 (if more than 1.6×10^5 nodes) relaxed amalgamations per node.

At the end, we compute memory weights and processing times to accurately simulate the matrix factorization: we compute the memory weight n_i of a node as $\eta^2 + 2\eta(\mu - 1)$, where η is the number of nodes amalgamated, and μ is the number of nonzeros in the column of the Cholesky factor of the matrix that is associated with the highest node (in the starting elimination tree); the processing time w_i of a node is defined as $2/3\eta^3 + \eta^2(\mu - 1) + \eta(\mu - 1)^2$ (these terms corresponds to one Gaussian elimination,

Table II. Proportions of Scenarios When Heuristics Reach Best (or Close to Best) Performance, and Average Deviations from Optimal Memory and Best Achieved Makespan

Heuristic	Best Memory (%)	Within 5% of Best Memory (%)	Normalized Memory	Best Makespan (%)	Within 5% of Best Makespan (%)	Normalized Makespan
ParSubtrees	81.1	85.2	2.34	0.2	14.2	1.40
ParSubtreesOptim	49.9	65.6	2.46	1.1	19.1	1.33
ParInnerFirst	19.1	26.2	3.79	37.2	82.4	1.07
ParDeepestFirst	3.0	9.6	4.13	95.7	99.9	1.04

two multiplications of a triangular $\eta \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix, and one multiplication of a $(\mu - 1) \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix). The memory weights f_i of edges are computed as $(\mu - 1)^2$.

The resulting 608 trees contains from 2,000 to 1,000,000 nodes. Their depth ranges from 12 to 70,000, and their maximum degree ranges from 2 to 175,000. Each heuristic is tested on each tree using $p = 2$, 4, 8, 16, and 32 processors. Then the memory and makespan of the resulting schedules are evaluated by simulating a parallel execution.

6.3. Results for Heuristics without Memory Bound

The comparison of the first set of heuristics (without memory bounds) is summarized in Table II. It presents the fraction of the cases where each heuristic reaches the best memory (respectively makespan) among all heuristics, or when its memory (respectively makespan) is within 5% of the best one. It also shows the average normalized memory and makespan. For each scenario (consisting in a tree and a number of processors), the memory obtained by each heuristic is normalized by the optimal (sequential) memory, and the makespan is normalized using a classical lower bound, since makespan minimization is NP-hard even without memory constraint. The lower bound is the maximum between the total processing time of the tree divided by the number of processors and the maximum weighted critical path.

Table II shows that ParSubtrees and ParSubtreesOptim are the best heuristics for memory minimization. On average, they use less than 2.5 times the amount of memory required by the optimal sequential traversal, when ParInnerFirst and ParDeepestFirst need 3.79 and 4.13 times this amount of memory, respectively. ParInnerFirst and ParDeepestFirst perform best for makespan minimization, having makespans very close on average to the best achieved ones, which is consistent with their 2-approximation ratio for makespan minimization. Furthermore, given the critical-path–oriented node ordering, we can expect that ParDeepestFirst makespan is close to optimal. ParDeepestFirst outperforms ParInnerFirst for makespan minimization, at the cost of a noticeable increase in memory. ParSubtrees and ParSubtreesOptim may be better trade-offs, as they use (on average) almost only half the memory of ParDeepestFirst for at most a 35% increase in makespan.

Figure 11 presents the evolution of the performance of these heuristics with the number of processors. The figure displays average normalized makespan and memory, and vertical bars represent 95% confidence intervals for the mean.[3] On this figure, we plot the results for all 608 trees except 76 of them, for which the results are so different that it does not make sense to compute average values anymore. These outliers belong to four different classes of applications, and the specific results for these graphs are shown in Figure 12. Note that in both figures, values of the different plots are

[3]The confidence intervals are obtained with the nonparametric bootstrap method.

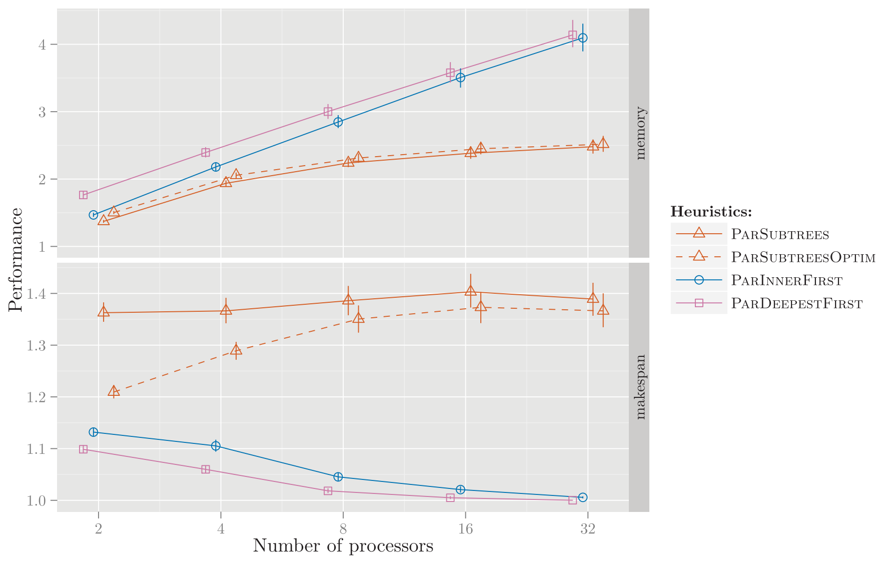

Fig. 11. Performance (makespan and memory) to the respective lower bounds for the first set of heuristics, excluding the trees with extreme performance. Vertical bars represent 95% confidence intervals.

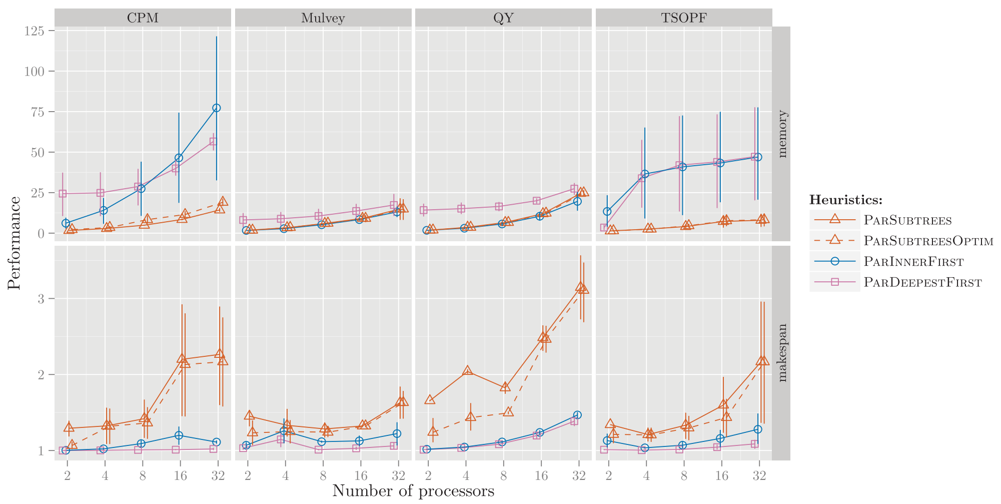

Fig. 12. Performance (makespan and memory) to the respective lower bounds for the first set of heuristics for four specific classes of trees that show specific behavior (CPM: closest point method; Mulvey: closest point method; QY: transient stability-constrained interior point optimal power flow; TSOPF: transient stability-constrained optimal power flow; see details on http://www.cise.ufl.edu/research/sparse/matrices/groups.html). Vertical bars represent 95% confidence intervals.

slightly offset on the x axis for better readability. Figure 11 shows that ParDeepestFirst and ParInnerFirst have a similar performance evolution, just like ParSubtrees and ParSubtreesOptim. The performance gap between these two groups, both for memory and makespan, increases with the number of processors. With a large number of processors, ParDeepestFirst and ParInnerFirst are able to decrease the normalized makespan (at the cost of an increase of memory), whereas ParSubtrees has an almost constant normalized makespan with the number of processors.

Despite the very different values for makespan and memory utilization, and a much higher variability, the results for the outliers presented in Figure 12 give the same conclusions about the relative performance of the heuristics. Furthermore, this graph also exhibits the absence of approximation ratios for ParDeepestFirst and ParInnerFirst for memory minimization. Indeed, even though the trees used in this set are taken from real-life applications, in contrast with the carefully crafted counterexamples of Section 5, the memory usage of ParDeepestFirst and ParInnerFirst on those trees can reach up to 100 times the optimal memory usage.

6.4. Results for Memory-Constrained Heuristics

In addition to the previous heuristics, we also test the memory-constrained heuristics. Since they can be applied only to reduction trees with null processing sizes, we transform the trees used in the previous tests into reduction trees as explained in Section 5.3.1. For a given scenario (tree, number of processors, memory bound), the memory obtained by each heuristic is normalized by the optimal memory on the original tree (not the reduction one). Thus, the normalized memory represents the actual memory used by the heuristic compared to the one of a sequential processing. In particular, this allows a fair comparison between memory-constrained heuristics and the previous unconstrained heuristics.

To compare the memory-constrained heuristics, we have applied them on the previous dataset using various memory bounds. For each tree, we first compute the minimum sequential memory M_{seq} obtained by a postorder traversal of the original tree. Then, each heuristic is tested on the corresponding tree with a memory bound $B = xM_{seq}$ for various ratios $x \geq 1$. Sometimes the heuristic cannot run because the amount of available memory is too small. This is explained by the following factors:

—The memory-constrained heuristics use a reduction tree that may well need more memory than the original tree. In general, however, the transformation from the original tree does not significantly increase the minimum amount of memory needed to process the tree.
—ParInnerFirstMemLimit has a minimum memory guarantee that is twice the sequential memory of a postorder traversal, and thus it cannot run with a memory smaller than $2M_{seq}$.
—ParDeepestFirstMemLimit has a minimum memory guarantee that is twice the sequential memory of a deepest first sequential traversal of the tree. A deepest first traversal uses much more memory than a postorder traversal, and thus ParDeepestFirstMemLimit needs much more memory than ParInnerFirstMemLimit to process a tree.

Figure 13 presents the results of these simulations. In this figure, points are shown only when a heuristic succeeds in more than 95% of the cases. The intuition is that with a success rate larger than 95%, the heuristic is presumably useful for this ratio. This figure shows that when the memory is very limited ($B < 2M_{seq}$), MemBookingInnerFirst is the only heuristic that can be run, and it achieves reasonable makespans. For a less strict memory bound ($2M_{seq} \leq B < 5M_{seq}$ or $2M_{seq} \leq B < 10M_{seq}$ depending on the number of processors), ParInnerFirstMemLimit is able to process the tree and achieves better makespans, especially when B is large. Finally, when memory is abundant, ParDeepestFirstMemLimit is the best among all heuristics. In the figure, we also tested the two variants ParInnerFirstMemLimitOptim and ParDeepestFirstMemLimitOptim presented in Section 5.3.2 that are more aggressive when starting leaves, but with the

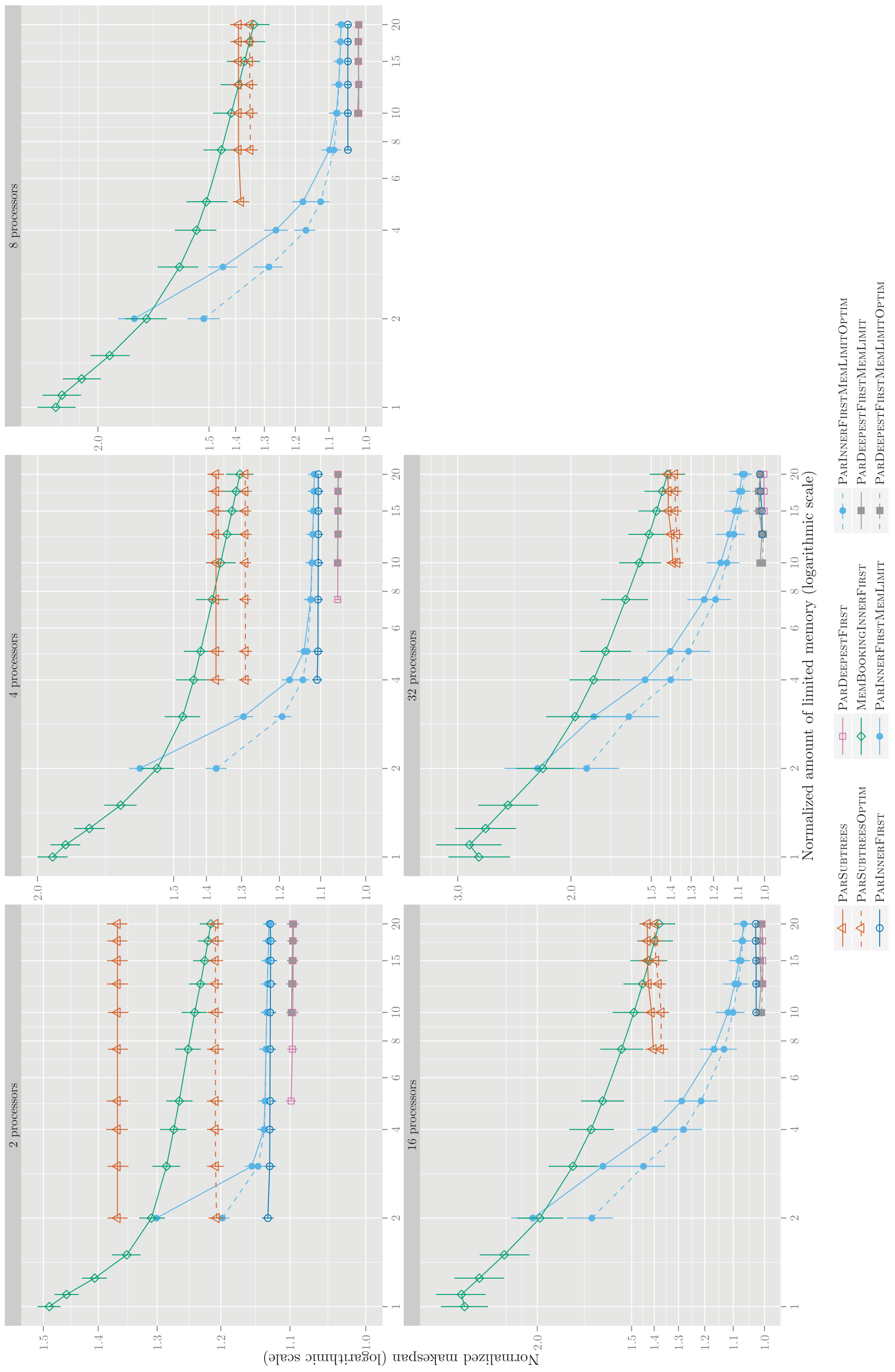

Fig. 13. Memory and makespan performance of memory-constrained heuristics (logarithmic scale).

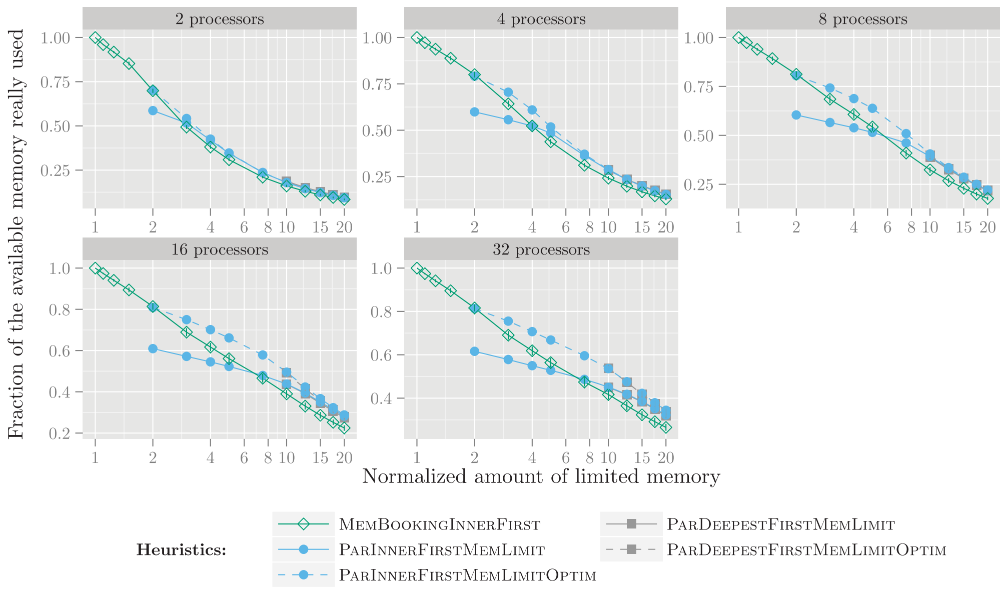

Fig. 14. Real use of limited memory for memory-constrained heuristics.

same memory guarantee as ParInnerFirstMemLimit and ParDeepestFirstMemLimit. We see that these strategies are able to better reduce the makespan in the case of a very limited memory (B close to $2M_{seq}$).

Finally, Figure 14 shows the ability of memory-constrained heuristics to make use of the limited amount of available memory. In this figure, points corresponding to ParDeepestFirstMemLimit (respectively ParDeepestFirstMemLimitOptim) are hardly distinguishable from ParInnerFirstMemLimit (respectively ParInnerFirstMemLimitOptim). We notice that MemBookingInnerFirst is able to fully use the very limited amount of memory when B is close to M_{seq}. The good use of memory is directly correlated with good makespan performance: for a given and limited amount of memory, heuristics giving the best makespans are the ones that use the largest fraction of available memory. Especially, we can see that ParInnerFirstMemLimitOptim and ParDeepestFirstMemLimitOptim are able to use much more memory than their nonoptimized counterpart, especially when memory is very limited.

7. CONCLUSION

In this study, we have investigated the scheduling of tree-shaped task graphs onto multiple processors under a given memory limit and with the objective to minimize the makespan. We started by showing that the parallel version of the pebble game on trees is NP-complete, hence stressing the negative impact of the memory constraints on the complexity of the problem. More importantly, we have proved that there does not exist any algorithm that is simultaneously a constant-ratio approximation algorithm for both makespan minimization and peak memory usage minimization when scheduling tree-shaped task graphs. We have also established bounds on the achievable approximation ratios for makespan and memory when the number of processors is fixed. Based on these complexity results, we then designed a series of practical heuristics; some of these heuristics are guaranteed to keep the memory under a given memory limit. Finally, we have assessed the performance of our heuristics using real task graphs arising from sparse matrices computation. These simulations demonstrated that the different heuristics achieve different trade-offs between the minimization of peak memory usage and makespan; hence, the set of designed heuristics provide an efficient solution for each situation.

This work represents an important step toward a comprehensive theoretical analysis of memory/makespan minimization for applications organized as trees of tasks, as it provides both complexity results and memory-constrained heuristics. Multifrontal sparse matrix factorization is an important application for this work and a good incentive to refine the computation model. In a second step, we should consider trees of parallel tasks rather than of pure sequential tasks, as the computations corresponding to large tasks (at the top of the tree) are usually distributed across processors. Of course, one would need a proper computation model to derive relevant complexity results. To get even closer to reality, one would also need to consider distributed memory rather than shared memory, or a mix of both. Furthermore, all of our scheduling policies are static, as they construct a final schedule before the computation starts. In practice, lightweight dynamic schedulers are needed that are able to cope with inaccurate timing predictions and to react to changes in the application: for example, using numerical pivoting will slightly alter the duration of tasks and the size of data. Hence, many important but challenging findings remain to be discovered.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their careful reading and helpful comments. We thank Bora Uçar for his help in creating and managing the dataset used in the experiments. We gratefully acknowledge that this work is partially supported by the Marsden Fund Council from government funding, grant 9073-3624767, administered by the Royal Society of New Zealand, and by the ANR *RESCUE* and *SOLHAR* projects, funded by the French National Research Agency.

REFERENCES

E. Agullo, P. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet. 2012. Robust memory-aware mappings for parallel multifrontal factorizations. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing (PP'12)*.

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

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

H. Casanova, A. Legrand, and Y. Robert. 2008. *Parallel Algorithms*. Chapman and Hall, London, UK.

T. A. Davis. 2006. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

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

J. R. Gilbert, T. Lengauer, and R. E. Tarjan. 1980. The pebbling problem is complete in polynomial space. *SIAM Journal on Computing* 9, 3, 513–524.

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

R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal* XLV, 9, 1563–1581.

A. Guermouche and J.-Y. L'Excellent. 2004. Memory-based scheduling for a parallel multifrontal solver. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE, Los Alamitos, CA, 71–81. DOI:http://dx.doi.org/10.1109/IPDPS.2004.1303001

T. C. Hu. 1961. Parallel sequencing and assembly line problems. *Operations Research* 9, 6, 841–848.

J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. 1989. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing* 18, 2, 244–257.

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

G. Karypis and V. Kumar. 1998. *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 Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN.

C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan. 2011. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems and Structures* 37, 2, 63–75.

J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. 1977. Complexity of machine scheduling problems. *Annals of Discrete Mathematics* 1, 343–362.

J. Liu. 1992. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review* 34, 1, 82–109. DOI:http://dx.doi.org/10.1137/1034004

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

J. W. H. Liu. 1987. An application of generalized tree pebbling to sparse matrix factorization. *SIAM Journal on Algebraic Discrete Methods* 8, 3, 375–395.

L. Marchal, O. Sinnen, and F. Vivien. 2013. Scheduling tree-shaped task graphs to minimize memory and makespan. In *Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'13)*. IEEE, Los Alamitos, CA, 839–850.

A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. 2007. Scheduling data-intensive workflows onto storage-constrained distributed resources. In *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'07)*. IEEE, Los Alamitos, CA, 401–409.

R. Sethi. 1973. Complete register allocation problems. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*. ACM, New York, NY, 182–195.

R. Sethi and J. D. Ullman. 1970. The generation of optimal code for arithmetic expressions. *Journal of the ACM* 17, 4, 715–728.

Received January 2014; revised April 2015; accepted May 2015