Dynamic Memory-Aware Task-Tree Scheduling

Guillaume Aupy, Clément Brasseur, Loris Marchal

Dynamic Memory-Aware Task-Tree Scheduling

Guillaume Aupy*
Vanderbilt University,
Nashville TN, USA
guillaume.aupy@inria.fr
* *Now with Inria and Université de Bordeaux, France*

Clément Brasseur, Loris Marchal
CNRS, LIP,
École Normale Supérieure de Lyon
Lyon, France
loris.marchal@ens-lyon.fr

***Abstract*—Factorizing sparse matrices using direct multifrontal methods generates directed tree-shaped task graphs, where edges represent data dependency between tasks. This paper revisits the execution of tree-shaped task graphs using multiple processors that share a bounded memory. A task can only be executed if all its input and output data can fit into the memory. The key difficulty is to manage the order of the task executions so that we can achieve high parallelism while staying below the memory bound. In particular, because input data of unprocessed tasks must be kept in memory, a bad scheduling strategy might compromise the termination of the algorithm. In the single processor case, solutions that are guaranteed to be below a memory bound are known. The multi-processor case (when one tries to minimize the total completion time) has been shown to be NP-complete. We present in this paper a novel heuristic solution that has a low complexity and is guaranteed to complete the tree within a given memory bound. We compare our algorithm to state of the art strategies, and observe that on both actual execution trees and synthetic trees, we always perform better than these solutions, with average speedups between 1.25 and 1.45 on actual assembly trees. Moreover, we show that the overhead of our algorithm is negligible even on deep trees (10^5), and would allow its runtime execution.**

***Keywords*-scheduling; memory; tree;**

I. Introduction

Parallel workloads are often modeled as task graphs, where nodes represent tasks and edges represent the dependencies between tasks. There is an abundant literature on task graph scheduling when the objective is to minimize the total completion time, or makespan. However, with the increase of the size of the data to be processed, the memory footprint of the application can have a dramatic impact on the algorithm execution time, and thus needs to be optimized. This is best exemplified with an application which, depending on the way it is scheduled, will either fit in the memory, or will require the use of swap mechanisms or out-of-core. There are very few existing studies that take into account the memory footprint when scheduling task graphs, and even fewer of them targeting parallel systems.

In the present paper, we consider the parallel scheduling of rooted in-trees. The vertices of the trees represent computational tasks, and its edges represent the dependencies between these tasks, which are in the form of input and output data: each task requests for its processing all the data produced by its children tasks to be available in memory, and outputs a new data for its parent. We want to process the resulting task tree on a parallel system made of p computing units, also named processors, sharing a global memory of limited size M. At any time, the size of all the data currently produced but not yet consumed cannot exceed M. Our objective is to minimize the makespan, that is, the total time needed to process the whole task tree, under the memory constraint.

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

In this paper, we mainly build on two previous results. On the theoretical side, we have previously studied the complexity of the bi-criteria problem which considers both makespan minimization and peak memory minimization [2], and we have proposed a few heuristic strategies to schedule task trees under hard memory constraints. However, these strategies requires strong reduction properties on the tree. An arbitrary tree can be turned into a reduction tree, but this increases its memory footprint, which limits the performance of the scheduler under memory constraint. On the practical side, Agullo et al. [3] uses a simple activation strategy to ensure the correct termination of a multifrontal QR factorization, whose task graph is an in-tree. Both approaches have drawbacks: the first one artificially increases the peak memory of the tree, and the second one overestimates the memory booked to process a subtree. Our objective is to take inspiration from both to design a better scheduling algorithm.

Note that we are looking for a *dynamic* scheduling algorithm, that is, a strategy that dynamically reacts to task terminations to activate and schedule new nodes. We suppose that only the tree structure and the data sizes are known

before the execution, not the task processing times, so that one cannot rely on them to build a perfect static schedule. Finally, the scheduling complexity should be kept as low as possible, since scheduling decision need to be taken during the computation without delaying the task executions.

Our contributions are as follows:

- We provide a novel heuristic along with a proof of its termination for memory bounds.
- We provide data-structure optimizations to improve its computational complexity.
- We provide a thorough experimental study, both on actual and synthetic trees to show its dominance over state of the art algorithms.
- We propose a new makespan lower bound for memory-constrained parallel platforms.

The rest of the paper is organized as follows. We first present the problem, its notation and formalize our objective in Section II. We then review related work and the two existing approaches listed above in Section III. Next, we present our new scheduling algorithms, as well as the proof of its correctness in Section IV. Then, we propose a memory-aware makespan lower bound in Section V. Finally, we present a set of comprehensive simulations to assess the benefit of the new algorithm VI, before presenting concluding remarks in Section VII

II. Model and objectives

A. Application model

Let T be a rooted in-tree (dependencies point toward the root) composed of n nodes, the tasks, denoted by their index $1, \ldots, n$. A node i is characterized by its input data (one per child), its execution data (of size n_i), and its output data (of size f_i). When processing node i, all input, execution and output data must be allocated in memory. At the termination of node i, input and execution data are deallocated, and only the output data stays in memory. We denote by $Children(i)$ the set of children of node i, which is empty is i as a leaf. The memory needed for the processing of node i, illustrated on Figure 1, is given by:

$$MemNeeded_i = \left(\sum_{j \in Children(i)} f_j \right) + n_i + f_i. \quad (1)$$

B. Platform model

We consider a shared-memory parallel platform, composed of p homogeneous processors onto which each task can be computed. Those processors share a limited memory of size M.

C. Objectives

Our objective here is to minimize the makespan, that is the total execution time, while keeping the size of the data stored in memory below the bound M. This problem is a

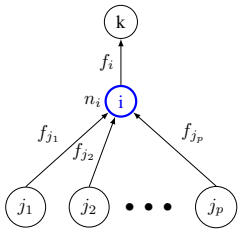

Figure 1. Input, execution and output data of a node i with children $j_1, \ldots j_p$ and parent k.

variant of the bi-criteria problem which aims at minimizing both the makespan and the peak memory. Note that those two objective are antagonist: the best way to minimize the makespan is to parallelize as much as possible without regard to the memory used, while the best way to minimize the peak memory used is to execute the whole schedule on a single processor which would give the worst makespan. In a previous study [2], we have proven that this bi-criteria problem was NP-complete and inapproximable within constants factors of both the optimal memory and the optimal makespan. Our variant clearly inherits this complexity and in this study, we are mainly looking for heuristic solutions.

Note that the algorithms under consideration are natural candidates to replace the simple activation scheme of [3]. Thus, their runtime complexity is critical, as we want to take scheduling decisions very fast. While a notable, say quadratic, complexity is acceptable in the initial preprocessing phase, we are looking for $O(1)$ complexity at each task termination, or eventually logarithmic.

III. Related work and existing algorithms

Memory and storage have always been a limited parameter for large computations, as outlined by the pioneering work of Sethi and Ullman [4] on register allocation for task graphs. In the realm of sparse direct solvers, the problem of scheduling a tree so as to minimize peak memory has first been investigated by Liu [5] in the sequential case: he proposed an algorithm to find a peak-memory minimizing traversal of a task tree when the traversal is required to correspond to a postorder traversal of the tree. A follow-up study [6] 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 [7]. However, they are very natural and straightforward solutions to this problem, as they allow to fully process one subtree before starting a new one. Therefore, they are widely used in sparse matrix software like `MUMPS` [8], [9], and achieve good performance on actual elimination trees [7].

The problem of scheduling a task graph under memory or storage 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, and geophysical simulations. The problem of task graphs handling large data has been identified by Ramakrishnan et al. [10] who propose some simple heuristics. In the context of quantum chemistry computations, Lam et al. [11] also consider task trees with data of large size.

Note that peak memory minimization is still a crucial question for direct solvers, as highlighted by Agullo et al. [12], who study the effect of processor mapping on memory consumption for multifrontal methods.

We now review the two scheduling strategies from the literature that target our problem.

A. Simple activation heuristic

Agullo et al.[3] use a simple activation strategy to ensure that a parallel traversal of a task tree will process the whole tree without running out of memory. The first step is to compute a postorder traversal, such as the memory-minimizing traversal of [5]. This postorder traversal, denoted by AO, will serve as an order to activate tasks. This solution requires that the available memory M is not smaller than the peak memory M_{AO} of the activation order. The strategy is summarized in Algorithm 1. The activation of a task i consists in allocating all the memory needed for this tasks. Then, only tasks that are both activated and whose dependency constraints are satisfied (i.e., all predecessors in the tree are already processed) are available for execution. Another scheduling heuristic may be used to choose which tasks the among available ones are executed: we denote by EO the order giving the priority of the tasks for execution. Note that *available nodes* are nodes whose children have been completed.

This simple procedure is efficient to schedule task trees without exceeding the available memory. However, it may book too much memory, and thus limit the available parallelism in the tree. Consider for example a chain of tasks $T_1 \rightarrow T_2 \rightarrow T_3$. Algorithm 1 will first book $n_1 + f_1$ for task T_1, then $n_2 + f_2$ for T_2 and finally $n_3 + f_3$ for T_3 (assuming all this memory is available). However, no two tasks of this chain can be scheduled simultaneously because of their precedence order. Thus, it is not necessary to book n_1, n_2 and n_3 at the same time, nor is it necessary to book memory for f_1 and f_3: the memory used for T_1 can be later reused for the processing of T_2 and T_3. By booking memory in a very conservative way, this heuristic may prevent nodes from other branches to be available for computation, and thus delay the processing of these nodes.

B. Booking strategy for reduction trees

In a previous publication [2], we have proposed a novel activation policy based on a refined memory booking strategy. However, our strategy is limited to special trees, also called reduction trees, who exhibit the following two properties:

- There is no execution data, i.e., $n_i = 0$ for each node i;
- The size of the output of each node is smaller than the size of its inputs, that is, $f_i \leq \sum_{j \in Children(i)} f_j$.

Using these two properties, we are able to prove that if the memory has been successfully booked for all the leaves of a subtree, then all nodes in this subtree can be processed without additional memory. Moreover, we know how to compute the amount of booked memory that each completing node has to transmit to its parent.

Contrarily to the previous algorithm, this complex strategy allows to correctly predict the amount of memory that needs to be booked for a given subtree. However, it only applies to special trees, namely reduction trees. General trees may be transformed into reduction trees by adding fictitious edges. However this increases the overall peak memory needed for any traversal, which limits its interest. We have indeed noticed that it does not give better performance than Algorithm 1 on general trees. Furthermore, in some cases it makes it a key limitation for general trees with limited memory as it does not always allow for the completion of those trees.

Algorithm 1: ACTIVATION(T, p, AO, EO, M)

```
1  M_Booked ← 0
2  ACT ← ∅
3  while the whole tree is not processed do
4      Wait for an event (task finishes or t = 0)
       // Free the memory booked by j
5      foreach just finished node j do
           M_Booked ← M_Booked − n_j − Σ_{j∈Children(i)} f_j
6      continueActivation ← true
7      while continueActivation do
           // Activate the first node i of AO if
              possible
8          i ← pop(AO)
9          if M_Booked + n_i + f_i ≤ M then
10             M_Booked ← M_Booked + n_i + f_i
11             push(i, ACT)
12         else
13             push(i, AO)
14             continueActivation ← false
       // Process available nodes in ACT following
          priority order EO
15     while there is an available processor P_u and there is an
       available node in ACT do
16         Let i be the available node in ACT with maximal
           priority in EO: Remove(i, ACT)
17         Make P_u process i
```

IV. A DYNAMIC FAST ALGORITHM

In this section, we propose a novel algorithm, named MEMBOOKING to schedule trees under limited memory, that

overcomes the limitations of the previous two strategies. Similarly to Algorithm 1, we rely on the activation of nodes, following an activation AO which is guaranteed to complete the whole tree within the prescribed memory in the case of a linear execution. However, activating a node does not correspond here to booking the exact memory $n_i + f_i$ needed for this node: some of this memory will be transfered by some of its descendant in the tree, and if needed, we only book what is absolutely needed. The core idea of the algorithm is the following: when a node completes its execution, we want (i) to reuse the memory that is freed on one of its ancestors and (ii) perform these transfers of booked memory in an As Late As Possible (ALAP) fashion. More precisely, the memory freed by a completed node j will only be transfered to one of its ancestors i if (a) all the descendants of i have enough memory to be executed (that is, they are activated), and (b) if this memory is necessary and cannot be obtained from another descendent of i that will complete its execution later. Finally, an execution order EO states which of the activated and available nodes should be processed whenever a processor is available.

In order to keep track of all nodes, we use *five states* to descrive them (a node can only be in one state), which we present in reverse order of their use for a given node:

1) Finished (FN): This corresponds to nodes which have completed their execution.
2) Running (RUN): This corresponds to nodes being executed.
3) Activated (ACT): This corresponds to nodes for which we have booked enough memory (some of this memory might be booked by some descendant in the subtree).
4) Candidate for activation ($CAND$): This corresponds to nodes which are the next to be activated, that is, all their descendant have been activated but they are not activated yet. This is the initial state for all leaves.
5) Unprocessed (UN): This corresponds to nodes which have not yet been considered; it is the initial state for all interior nodes.

Because a node can only have be in one state at a given time, we write $j \in UN$ (resp. $CAND$, ACT, RUN, FN) if node j is in the corresponding state.

We now present the MEMBOOKING algorithm (Algorithm 2), as well as its proof of correctness. Some optimizations and data-structures used to reduce the time complexity are detailed in Section IV-D and in the companion report [13].

At the beginning of the schedule, or each time a task completes, the MEMBOOKING algorithm performs these three consecutive operations:

1) Memory re-allocation: DISPATCHMEMORY (Algorithm 3) reallocates the memory used by a node that just finished its execution. We present this algorithm in Section IV-A.

Algorithm 2: MEMBOOKING(T, p, AO, EO, M)

```
   // initialization of the memory
1  foreach node i do
2      Booked[i] ← 0
3      BookedBySubtree[i] ← −1
4  M_Booked ← 0
5  UN ← T \ Leaves(T)
6  CAND ← Leaves(T)
7  ACT ← ∅

   // main loop
8  while the whole tree is not processed do
9      Wait for an event (task finishes or t = 0)
10     foreach just finished node j do
11         DispatchMemory(j)
12     UpdateCAND-ACT(CAND, ACT)
13     while there is an available processor P_u and ACT ≠ ∅
       do
14         Let i be the available node in ACT with maximal
           priority in EO: Remove(i, ACT)
15         Make P_u process i
```

Available nodes are nodes whose children have been completed.

2) Node activation: UPDATE$CAND$-ACT (Algorithm 4) allocates the available memory following the activation order AO. We present this algorithm in Section IV-B.
3) Node scheduling: the schedule is done following the execution order EO amongst the nodes that are both activated and ready to be executed.

Note that while AO is a topological order, we do not have any constraint on EO.

To be able to keep track of the memory allocated to each node, we use two arrays of data that are updated during the computation, namely:

- `Booked`$[1..n]$, which contains the memory that is currently booked in order to process nodes 1 to n. We further call $M_{Booked} = \sum_i$ `Booked`$[i]$;
- `BookedBySubtree`$[1..n]$, which sums the memory that is currently booked by the subtree rooted in $i \in \{1, \ldots, n\}$.

A. Memory re-allocation

When a node finishes its computation, the memory that was used during its computation can be allocated to other nodes. Our memory allocation works in two steps:

1) First we free the memory that was used by the node that has just finished its execution. Note that we cannot free all the memory: if the node that finished is not the root of the tree, then its output needs to be saved. In that case we allocate this memory to its parent.
2) Then we allocate the memory freed to its ancestors in ACT following an As Late As Possible strategy (meaning that if there is already enough memory booked in the unfinished part of the subtree, we do not allocate

it to the root of the subtree but keep it for later use). We thus compute the contribution $C_{i,j}$ of a terminated node j to its parent i as the difference between what is needed by node i and what can be provided later by its subtree.

Algorithm 3: DISPATCHMEMORY(j)

```
   /* First we free the memory used by j                          */
1  B = Booked[j]
2  { Booked[j]              ← 0
   { M_Booked               ← M_Booked − B
   { BookedBySubtree[j]     ← BookedBySubtree[j] − B
3  i ← parent(j)
4  if i ≠ NULL then
   |  { Booked[i]           ← Booked[i] + f_j
5  |  { M_Booked            ← M_Booked + f_j
   |  { BookedBySubtree[i]  ← BookedBySubtree[i] + f_j
6  |  B = B − f_j
   /* Then we dispatch the memory used by j
      between its ancestors which are in ACT, if
      it is neccessary                                            */
7  while i ≠ NULL and i ∈ ACT ∪ RUN and B ≠ 0 do
8  |  C_{j,i} =
   |  max(0, MemNeeded_i − (BookedBySubtree[i] − B))
   |  { Booked[i]           ← Booked[i] + C_{j,i}
9  |  { M_Booked            ← M_Booked + C_{j,i}
   |  { BookedBySubtree[i]  ← BookedBySubtree[i]
   |                          −(B − C_{j,i})
10 |  B = B − C_{j,i}
11 |  i ← parent(i)
```

B. Node activation

Our second algorithm, UPDATE$CAND$-ACT, updates both ACT and $CAND$. The key point of this sub-algorithm is that it is conceived such that nodes are activated following the AO order. We formally show this result in the companion report [13].

C. Proof of correctness

In this section, we show that the implementation of this algorithm will guarantee a correct execution if the memory bound is large enough. Namely, we show the following result:

Theorem 1. *If T can be executed with a memory bound of M_0 using the sequential schedule AO, then for all $M \geq M_0$, for all p and EO,* MEMBOOKING(T, p, AO, EO, M) *processes T entirely.*

Note that because of space limitations, we only draft the proof here. The full proof (along with intermediary results) is available in the companion report [13].

To prove Theorem 1, we need to verify that the following conditions are respected:

Algorithm 4: UPDATE$CAND$-$ACT(CAND, ACT)$

```
1  WaitForMoreMem ← false
2  while !(WaitForMoreMem) and CAND ≠ ∅ do
3  |  Let i be the node of CAND with maximal priority in
   |  AO
4  |  MissMem_i = max(0, MemNeeded_i −
   |  (Booked[i] + Σ_{j∈Children(i)} BookedBySubtree[j]))
5  |  if M_Booked + MissMem_i ≤ M then
   |  |  { Booked[i]           ← Booked[i] + MissMem_i
6  |  |  { M_Booked            ← M_Booked + MissMem_i
   |  |  { BookedBySubtree[i] ← Booked[i]+
   |  |                         Σ_{j∈Children(i)} BookedBySubtree[j]
7  |  |  remove(i, CAND); insert(i, ACT)
8  |  |  if ∀j ∈ Children(parent(i)), j ∉ UN ∪ CAND
   |  |  then
9  |  |  |  remove(parent(i), UN);
   |  |  |  insert(parent(i), CAND)
10 |  else
11 |  |  WaitForMoreMem ← true
```

1) The memory used never exceed the limit M;
2) Each running task has enough memory to run;
3) No data is lost, that is, a result that was computed will not be overwritten until it has been used;
4) All tasks are executed.

To prove items 1, 2 and 3, we use Booked$[1..n]$ introduced earlier as a memory counter. We first show that no node i will use at anytime more than Booked$[i]$ memory slots for its execution or input storage. We then can show that at all time $\sum_i$ Booked$[i] \leq M$ which guarantees that we never use more memory than what we need. Furthermore, we show that if a task $i \in RUN$, then Booked$[i] = MemNeeded_i$. This guarantees that a task always has enough memory available to run. Furthermore, to be sure that we never erase important information, we show that unless a task is moved to FN, Booked$[i]$ is never decreased, and increases by the sufficient amount when an input of i is created.

Item 4 is proved by contradiction: we show that if not all tasks are executed, then there exists a time t where $RUN = \emptyset$. Then we show that at that time, necessarily there exists a task in ACT such that (i) all its children are in FN, and (ii) that has enough memory booked to be able to be executed. Hence this task could be moved to RUN, contradicting the property that $RUN = \emptyset$.

D. Complexity analysis

In this section we give a complexity analysis of the algorithm presented in the previous section. We have chosen to separate the idea of the main algorithm (Section IV) from the optimizations presented here and used to lower the execution cost so that the original algorithm is more understandable. A complete version of algorithm MEMBOOKING

with structural optimizations is available in the companion report [13].

Theorem 2. *Let T a tree with n nodes, and H be its height, AO an activation order, EO an execution order, M a memory bound and p processors, then* MEMBOOKING(T, p, AO, EO, M) *runs in* $O\left(n(H + \log n)\right)$.

Proof: First let us define some data structures that we use and update during the execution to reach this time complexity.

First we introduce some informative arrays:

- one that keeps track of the number of children of each nodes that are still in UN or $CAND$;
- another one that keeps track of the number of children of each nodes that are not finished;
- another one that keeps track of nodes not in $UN \cup CAND$.

These arrays are updated during the execution of the algorithm. The total time complexity of updating each of them throughout execution is $O(n)$.

We now introduce the main structures used in the computation:

- We implement $CAND$ as a heap whose elements are sorted according to the activation order (AO). All elements are inserted and removed (with complexity $O(\log n)$) at most once from $CAND$, hence a time complexity of $O(n \log n)$. Furthermore, in UPDATE$CAND$-ACT, extracting i from $CAND$ (on line 3) is done in constant time.
- We use a data structure $ACTf$ to remember the subset of ACT such that all its elements children have finished their execution. This is implemented as a heap whose elements are sorted according to the activation order (EO). We show in [13] how to update this structure throughout the execution. All elements are inserted and removed at most once from $ACTf$, hence the time complexity of elements going through $ACTf$ is $O(n \log n)$. Finally, in MEMBOOKING, extracting i from $ACTf$ (line 14) is done in constant time.

We review in details all operations performed by MEMBOOKING:

- DISPATCHMEMORY is called exactly once per node (every time a node finishes). For a given node j of depth h_j, it does at most $O(h_j)$ operations (the test $i \in ACT \cup RUN$ on line 7 can now be done in constant time with the arrays define above), which gives a total cumulative time complexity of $O(nH)$.
- UPDATE$CAND$-ACT is called for each event. Note that we have already accounted for removing or inserting all elements from the different sets. Finding the element to remove from $CAND$ on line 3 is done in constant time because $CAND$ is a heap sorted according to AO. Similarly, the test on line 8 can now be done in constant time.
  The most time consuming event is the computation of $MissMem_i$ on line 4 which could be computed up to $O(n)$ times for a given node if the condition on M_{Booked} and M on line 5 is not satisfied (hence giving a time complexity of $O(n^2)$). To avoid this case, we simply make sure that we can compute `BookedBySubtree`$[i]$ and keep the information to avoid recomputation. Details on this are available in [13].
- Finally, the last "while" loop of MEMBOOKING (line 13) is entered once per element contained in ACT, that is exactly n times. Furthermore, because $ACTf$ is a heap sorted according to EO, removing one element is done in $O(\log n)$, which gives a cumulative complexity of $O(n \log n)$ for this last loop.

Finally accounting for all operations, the total time complexity of this optimized algorithm is $O(n(log(n) + H))$. ∎

V. NEW MAKESPAN LOWER BOUND

It is usual in scheduling problems to compare the makespan of proposed algorithms to lower bounds, as the optimal makespan is usually out of reach (NP-complete). The classical lower bound for scheduling task graphs considers the maximum between the average workload (total computation time divided by the number of processors) and the longest path in the graph. In a memory-constrained environnement, the memory bound itself may prevent the simultaneous execution of too many tasks. We propose here a new lower bound that takes this into account.

Theorem 3. *Let $C_{\max}$ be the makespan of any correct schedule of a tree whose peak memory is at most the memory bound M, and t_i the processing time of task i. Then*

$$C_{\max} \geq \frac{1}{M} \sum_i MemNeeded_i \times t_i.$$

Proof: Consider a task i as described in the model of Section II: its processing requires a memory of $MemNeeded_i$ (see Equation (1)). As stated in the theorem, we denote by t_i its processing time. Consider the total memory usage of a schedule, that is, the sum over all time instants t of the memory used by this schedule. Then, task i contributes to at least $MemNeeded_i \times t_i$ to this total memory usage. For a schedule of makespan $C_{\max}$, the total memory usage cannot be larger than $C_{\max} \times M$, where M is the memory bound. Thus, $\sum_i MemNeeded_i \times t_i \leq C_{\max} \times M$ which concludes the proof. ∎

We have noticed in the simulations described in the next section that with eight processors, this new lower bound improved the classical lower bound in 22% of the case for on actual assembly trees, and in these cases the average increase in the bound was 46%. For the simulations on synthetic trees,

it has improved the lower bound in 33% of the cases, with an average improvement of 37%. Contrarily to the previous lower bound, this new lower bound does not depend on the number of processors, hence the improvement is even greater with more processors.

It is important to understand that the more precise the lower bound, the more information is available for a possible improvement of the considered heuristics.

VI. Simulations

We report here the results of the simulations that we performed to compare our new booking strategy (MemBooking) to the two other scheduling heuristics presented above: the basic Activation policy [3] presented in Section III-A and the booking strategy [2] for reduction trees, denoted MemBookingRedTree, from Section III-B. In the latter, the tree is first transformed into a "reduction tree" [2] by adding some fictitious nodes and edges before the scheduling strategy can be applied.

A. Data sets

The trees used for the simulations come from two data sets, which we briefly describe below.

The first data set, also called *assembly trees* are trees corresponding to the multifrontal direct factorization of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices/). This data sets is taken from [2], where more information can be found on multifrontal factorization and on how the trees are constructed. This data sets consists in 608 trees which contains from 2,000 to 1,000,000 nodes. Their height ranges from 12 to 70,000 and their maximum degree ranges from 2 to 175,000.

The second data set is synthetic. The node degree is taken randomly in $[1; 5]$, with a higher probability for small values to avoid very large and short trees, on which we already observed with the previous data set that our algorithm outperforms other strategies. Edges weights follow a truncated exponential distribution of parameter 1. The size of a node is 10% of its outgoing edge weight and it processing time is proportional to its outgoing edge degree. We generated 50 synthetic trees of 1.000, 10.000 and 100.000 nodes, which results in trees of respective average height of 63, 95 and 131. More details on the data set can be found in [13].

B. Simulation setup

All three strategies were implemented in C, with special care to avoid complexity issues. These strategies have been applied to the two tree families described above, with the following parameters:

- We tested 5 different number of processors (2,4,8,16,32). The results were quite similar, expect for the extreme case (too large or too small parallelism), so we report here only the results for 8 processors. Results for other numbers of processors are available in the companion research report [13].
- For each tree, we first computed the post-order traversal that minimizes the peak memory. This gives the minimum amount of memory needed for both Activation and MemBooking (MemBookingRedTree is likely to use more memory as it works on a transformed tree). The heuristics are then tested with a factor of this minimal memory, which we call below *normalized memory bound*. We only plot an average result when a given strategy was able to schedule at least 95% of the trees within the memory bound.
- The previous post-order was used as input for both the activation order AO and the execution order EO for Activation and MemBooking. We also tested other order for activation and execution, such as other postorders, critical path ordering, or even optimal (non-postorder) ordering for peak memory [6]. As we will detail later, this only results in slightly noticeable change in performance.

During the simulations of the parallel executions, we reported the makespan (total completion time), which is normalized by the maximum of the classical lower bound and our new memory related lower bound (see Section V). We also reported the peak memory of the resulting schedules, as well as the time needed to compute the schedule. This scheduling time does not include the computation of the activation or execution order, which may be done beforehand.

C. Results on the assembly trees

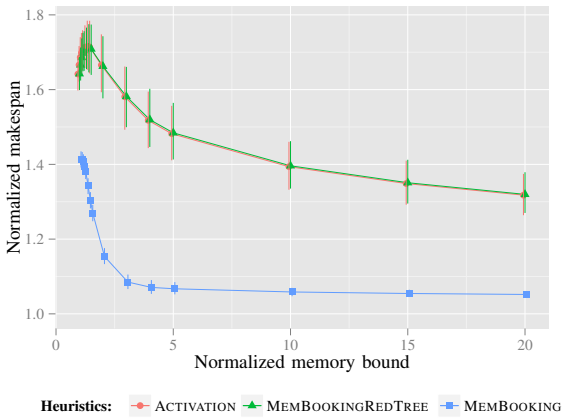

Figure 2. Makespan of assembly trees with all heuristics depending on the memory bound

Figure 2 plots the average normalized makespan of all strategies on various memory constraints. We notice that for a memory bound twice the minimum memory, MemBooking is 1.4 faster than Activation on average. However, even this particular speedup spans a wide interval (between

1 and 6) due to the large heterogeneity of the assembly trees. Note that the two heuristics from the literature give very similar results: this is explained by the fact that MEMBOOKINGREDTREE first transforms the trees before applying a smart booking strategy: on these trees, adding fictitious edges has the same effect than booking to much memory (as ACTIVATION does) and hinders the benefit of the booking strategy. We also note that MEMBOOKING is able to take advantage of very scarce memory conditions: as soon as the available memory increases from its minimum value, its makespan drops and reaches only 10% above the lower bound for 3x the minimum memory, leaving very little room to hope for better algorithms. This is also

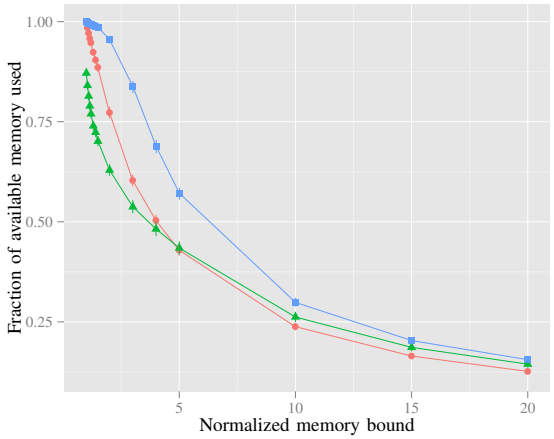

Figure 3. Fraction of memory used by all heuristics on assembly trees (same legend as Figure 2)

illustrated by Figure 3, which plots the real use of the memory by the heuristics: while ACTIVATION and MEMBOOKINGREDTREE are very conservative, MEMBOOKING is able to use larger fraction of the available memory when it is limited.

Changing the activation or execution order: On Figure 4, we present the average makespan of the ACTIVATION heuristic for various activation and execution order. This figure is similar to Figure 2 for the other scheduling heuristics. The activation and execution order used are the following:

- memPO (memory PostOrder): the sequential postorder traversal that minimizes the peak memory (NB: this is the order chosen activation and execution order of both ACTIVATION and MEMBOOKING in all other plots of this section);
- CP (Critical Path): nodes orders by decreasing bottom-level;
- OptSeq (Optimal Sequential): the sequential (non postorder) traversal that minimizes the peak memory, computed as in [6];
- perfPO (performance PostOrder): another postorder traversal, designed for parallel performance (subtrees with larger critical path are scheduled first, which, in a parallel execution, is supposed to give higher priority to nodes with large critical path).

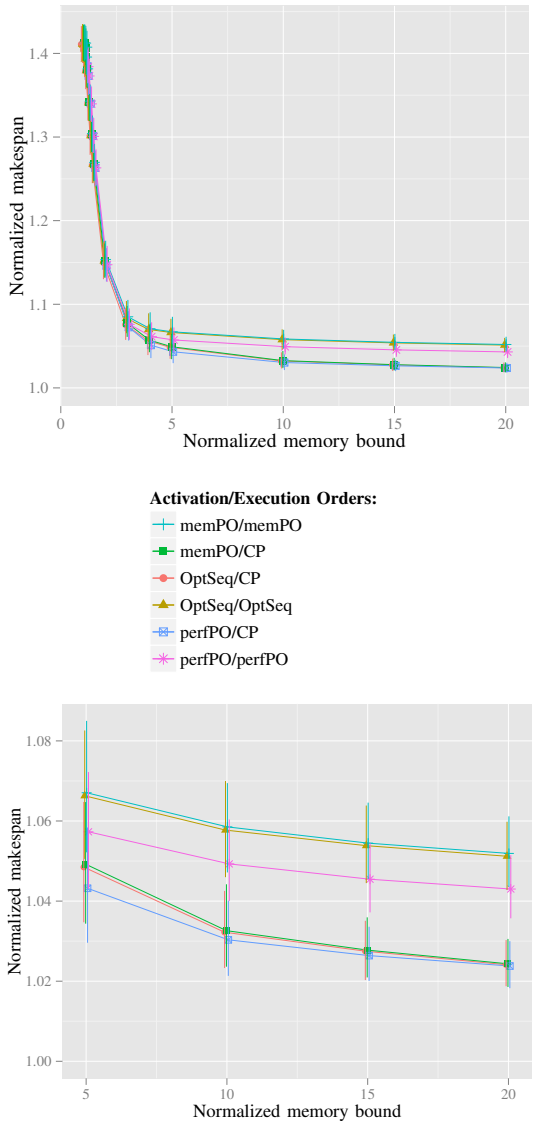

Figure 4. Makespan of assembly trees for the proposed MEMBOOKING strategy using different activation and execution order. The bottom plot corresponds to a zoom on the part where the memory is less limited.

We notice that the results of using different orders for activation and/or execution slightly change the results: using CP as an execution order always gives a small but noticeable improvement over the other strategies. On the contrary, the choice of the activation order has little effect on the final makespan. The same effects can be seen on ACTIVATION (when changing the activation/execution orders) and MEMBOOKING (when changing its priority order). However, the gap between the performance of different orders is much smaller than the gap of using different scheduling strategy:

changing the activation/execution order does not change the ranking of the scheduling policies

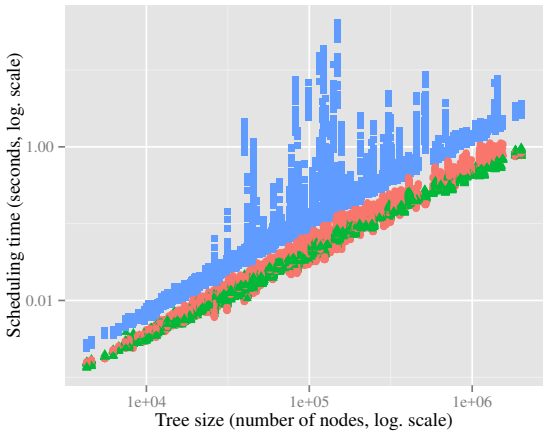

Figure 5. Running times of the heuristics on assembly trees (same legend as Figure 2)

Discussion on execution time: From the complexity analysis (Section IV-D) we expect our algorithm to add significant overhead when trees are very deep (nH term of the worst-case complexity). We first study the cumulative running time in Figure 5 of the various strategy as a function of the number of nodes in the trees. All strategies have similar running times, except on a subset of trees for which our heuristic is much slower ($\approx$ 10s). We verify that those

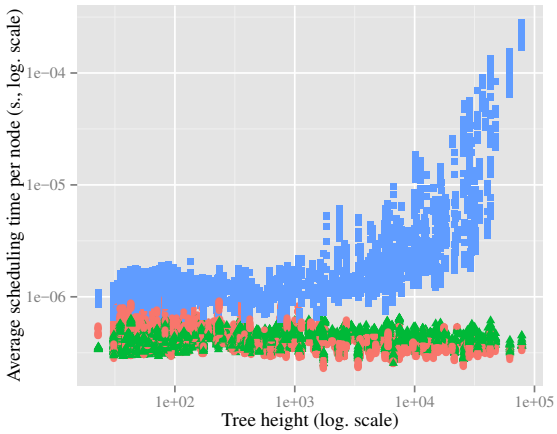

Figure 6. Running times of the heuristics on assembly trees (same legend as Figure 2)

running times indeed depend on the height of the tree in Figure 6. Another noticeable fact from this figure is that overall the average overhead for each node remains negligible (below 1ms per node with height $H = 10^5$!).

As future work, it may be interesting to get rid of this height factor in the complexity of the algorithm especially for cases when $H = O(n)$.

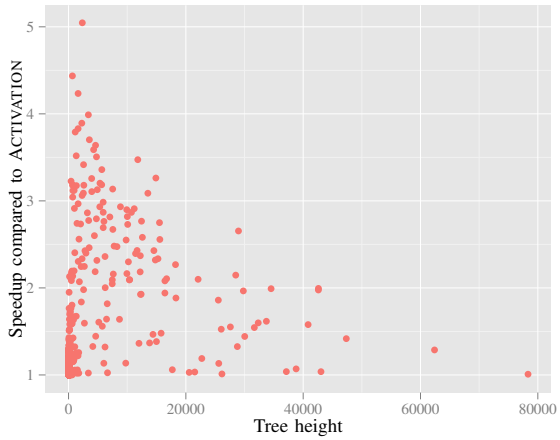

Figure 7. Speedup of MEMBOOKING compared to ACTIVATION on assembly trees when the normalized memory bound is 2 for all 608 trees.

To give some hindsight on the importance of this factor, we decided to study the speedup of MEMBOOKING compared to that of ACTIVATION as a function of the tree height. In particular, one can see from the correlations between Figures 5 and 6 that the trees of large height are trees where $H = O(n)$. We report these results in Figure 7 where we show the relation between achievable speedup and tree height: very deep trees usually correspond to thin ones and do not provide big opportunities for increasing the parallelism. This is why our strategy achieves best speedups on shallow trees. With this in mind, an interesting study would be to derive a good measure on trees which may hint whether the use of a sophisticated strategy such as MEMBOOKING is needed. Finding such a good measure would need a particular research effort and is out of scope of this paper.

D. Results on the synthetic trees

The simulations on synthetic trees show the same general trends as what we notices on assembly trees, and thus we only review briefly their results.

Figure 8 shows that MEMBOOKING is once again able to schedule trees faster in a memory-constrained environnement. Synthetic trees are more regular and homogeneous that assembly trees, so that the speedup of MEMBOOKING over ACTIVATION is more regular. It reaches an average 1.3 when the memory is twice the bound.

Finally, the scheduling time of all strategies is always below 0.1 seconds due to the smaller height of the trees. Note that MEMBOOKINGREDTREE is not able to schedule most trees in a very constrained memory environment: when the memory bound is smaller that 1.4 times the minimum memory peak of a sequential postorder processing, MEMBOOKINGREDTREE is unable to schedule 33% of the trees (or more) and thus is not included in the plot.

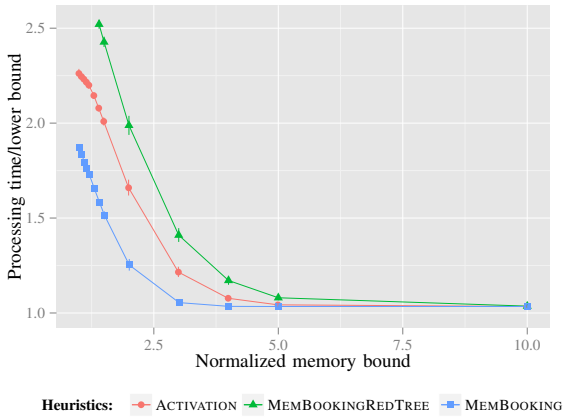

Figure 8. Makespan of synthetic trees with all heuristics depending on the memory bound

VII. CONCLUSION

In this paper, we have proposed a novel algorithm for scheduling task trees on computing platforms with bounded shared memory. The proposed algorithm carefully allocates memory to activated nodes, and accurately predicts how much memory can be recycled from the processing of a node's ancestors. We have shown that it is always able to schedule a tree under an admissible memory bound, and that its complexity is sufficiently small to allow its implementation in actual runtime schedulers. By performing simulations on both actual assembly trees of sparse direct multifrontal solvers and on a broader class of synthetic trees, we have proved that it outperforms its two competitors from the literature, especially when memory is a scarce resource. Incidentally, we have proposed a new makespan lower bound that takes into account a bound on the shared memory, which, to the best of our knowledge, is the first of its kind.

This study is the first step in the design of realistic schedulers for task trees handling large data, such as assembly trees. One major extension would be to consider parallel tasks rather than only sequential ones. To this goal, one would need to make several adaptations to cope with the extra memory needed for a parallel processing, and to solve the unavoidable trade-off between allocating many processors to big tasks (and losing on tree parallelism) and allocating many tasks in parallel (and threatening the memory bound). Nevertheless, we are confident that the algorithm presented in this paper (or its adaptation) would still provide an improvement over the classical ACTIVATION algorithm. Another necessary extension would be to consider distributed memories, or even a mix of distributed/shared memory (as in clusters of cores sharing a dedicated memory).

ACKNOWLEDGMENTS

This work was supported by the ANR SOLHAR project funded by the French National Research Agency and by the Keystone Associate Team funded by Inria.

REFERENCES

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

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

[3] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Multifrontal QR factorization for multicore architectures over runtime systems," in *Euro-Par 2013 Parallel Processing - 19th International Conference*, 2013, pp. 521–532.

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

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

[6] ——, "An application of generalized tree pebbling to sparse matrix factorization," *SIAM J. Algebraic Discrete Methods*, vol. 8, no. 3, pp. 375–395, 1987.

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

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

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

[10] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling data-intensiveworkflows onto storage-constrained distributed resources," in *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'07)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 401–409.

[11] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan, "Memory-optimal evaluation of expression trees involving large objects," *Computer Languages, Systems & Structures*, vol. 37, no. 2, pp. 63–75, 2011.

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

[13] G. Aupy, C. Brasseur, and L. Marchal, "Dynamic memory-aware task-tree scheduling," INRIA, France, Research Report 8966, Oct. 2016.