Memory-aware tree partitioning on homogeneous platforms

Changjiang Gou, Anne Benoit, Loris Marchal

Memory-aware tree partitioning on homogeneous platforms

Changjiang Gou, Anne Benoit, Loris Marchal
Laboratoire LIP, École Normale Supérieure de Lyon, France & ECNU, Shanghai, China
Firstname.Lastname@ens-lyon.fr

I. Introduction

Parallel workloads are often modeled as directed acyclic graphs of tasks. In this paper, we aim at scheduling some of these graphs, namely rooted tree-shaped workflows, onto a set of homogeneous computing platforms, so as to minimize the makespan. Such tree-shaped workflows arise in several computational domains, such as the factorization of sparse matrices [2], in computational physics code modeling electronic properties [5], or in sparse linear algebra [7]. The nodes of the tree typically represent computation tasks and the edges between them represent dependencies, in the form of output and input files. In this paper, we consider out-trees, where there is a dependency from a node to each of its child nodes, but the case of in-trees is similar. For such out-trees, each node (except the root) has to receive an input file from its parent and produces a set of output files (except leaf nodes), each of them being used as an input by a different child node. All its input file, execution data and output files have to be stored in local memory during its execution. The input file is discarded after execution, while output files are kept for the later execution of the children.

The way the tree is traversed influences the memory behavior: different sequences of node execution demand different amounts of memory, and memory optimal traversals have been proposed [6], [4]. The problem of scheduling such a tree on a single processor with limited memory is also discussed in [4]: in case of memory shortage, some input files need to be moved to a secondary storage (such as a disk), which is larger but slower, and temporarily discarded from the main memory.

We focus here on a homogeneous multi-processor platform and aim at partitioning the tree and mapping different subtrees on different processors, each having its own private memory. Hence, we are able to both reduce memory requirement and to improve the processing time (or makespan) by doing some processing in parallel. However, this incurs communication costs, which may be non negligible.

II. Model

We consider a tree-shaped task graph τ, where the nodes of the tree, numbered from 1 to n, correspond to tasks, and the edges correspond to precedence constraints among the tasks. The tree is rooted (node r is the root, where $1 \leq r \leq n$), and all precedence constraints are oriented towards the leaves of the tree. A precedence constraint $i \to j$ means that task j needs to receive a file (or data) from its parent i before it can start its execution. Each task i in the rooted tree is characterized by the size f_i of its input file, and by the size m_i of its temporary execution data (and for the root r, we assume that $f_r = 0$). A task can be processed by a given processor only if all the task's data (input file, output files, and execution data) fit in the processor's currently available memory. The memory requirement of task i is thus $MemReq(i) = f_i + m_i + \sum_{j \in children(i)} f_j$, where $children(i)$ are the children nodes of task i in the tree.

Task i can be executed once its parent, denoted $parent(i)$, has completed its execution, and the execution time for task i is w_i, if it fits in memory. If the whole tree fits in memory and is executed sequentially, the execution time, or *makespan*, is $\sum_{i=1}^{n} w_i$. In this case, the task schedule, i.e., the order in which tasks of τ are processed, plays a key role in determining how much memory is needed to execute the whole tree in main memory. When tasks are scheduled sequentially, such a schedule is a topological order of the tree, also called a traversal. One can figure out the minimum memory requirement of a task tree τ and the corresponding traversal using the work of Liu [6] or some of the authors' previous work [4]. We denote by $MinMemory(\tau)$ the minimum amount of memory necessary to complete task tree τ.

The target platform consists of p identical processors, each equipped with a memory of size M. The aim is to benefit from this parallel platform both for memory, by allowing the execution of a tree that does not fit within the memory of a single processor, and also for makespan, since several parts of the tree could then be executed in parallel. The goal is therefore to partition the tree workflow τ into $k \leq p$ connected subtrees $\tau_1, \ldots, \tau_k$, which can be each executed within the memory of a single processor, i.e., $MinMemory(\tau_\ell) \leq M$, for $1 \leq \ell \leq k$. We are to execute such k subtrees on k processors. Let $root(\tau_\ell)$ be the task at the root of subtree τ_ℓ. If $root(\tau_\ell) \neq r$, the processor in charge of tree τ_ℓ needs to receive some data from the processor in charge of the tree containing $parent(root(\tau_\ell))$, and this data is a file of size $f_{root(\tau_\ell)}$. This can be done within a time $\frac{f_{root(\tau_\ell)}}{\beta}$, where β is the available bandwidth between each couple of processors.

We denote by $alloc(i)$ the set of tasks included in subtree τ_ℓ rooted in $root(\tau_\ell) = i$, and by $desc(i)$ the set of tasks that have a parent in $alloc(i)$: $desc(i) = \{j \mid parent(j) \in alloc(i)\}$. The makespan can then be expressed with a recursive formula. Let $MS(i)$ denote the makespan required to execute the whole subtree rooted in i, given a partition into subtrees. Note that the whole subtree rooted in i may contain several subtrees of the partition (it is τ for $i = r$). The goal is hence to express $MS(r)$, which is the makespan of τ. We have $MS(i) = \frac{f_i}{\beta} +$

$\sum_{j \in alloc(i)} w_j + \max_{k \in desc(i)} MS(k)$. We assume that the whole subtree τ_ℓ is computed before initiating communication with its children. The goal is to find a decomposition of the tree into $k \leq p$ subtrees that all fit in the available memory of a processor, so as to minimize the makespan $MS(r)$:

Definition 1 (MINMAKESPAN)**.** *Given a task tree τ with n nodes, a set of p processors each with a fixed amount of memory M, partition the tree into $k \leq p$ subtrees $\tau_1, \ldots, \tau_k$ such that $MinMemory(\tau_i) \leq M$ for $1 \leq i \leq k$, and the makespan is minimized.*

In the companion research report [1], we establish the problem complexity with the following theorem. Its proof relies on a sophisticated reduction from the Partition problem.

Theorem 1. *The (decision version of)* MINMAKESPAN *problem is NP-complete.*

III. HEURISTIC STRATEGIES

In this section, we design polynomial-time heuristics to solve the MINMAKESPAN problem. The pseudo-codes and more detailed descriptions are available in the companion research report [1].

A. Tree partitioning without memory constraints

In this section, we focus on the case where $MinMemory(\tau) \leq M$, i.e., it is possible to process the whole tree on a single processor without exceeding the memory constraint. The objective is to split the tree into a number of subtrees, each processed by a single processor, in order to minimize the makespan.

We first consider the case where the tree is a linear chain, and prove that its optimal solution uses a single processor.

Lemma 1. *Given a tree τ such that all nodes have at most one child (i.e., it is a linear chain), the optimal makespan is obtained by executing τ on a single processor, and the optimal makespan is $\sum_{i=1}^{n} w_i$.*

Proof. If more than one processor is used, all tasks are still executed sequentially because of dependencies, but we further need to account for communicating the f_i's between processors. Therefore, the makespan can only be increased. □

More generally, if the decomposition into subtrees form a linear chain, then the subtrees must be executed one after the other and no parallelism is exploited, so that the makespan can only be increased compared to executing the whole tree on a single processor.

1) Two-level heuristic: The first heuristic, **SplitSubtrees**, builds upon Lemma 1 and creates a two-level partition with a set of nodes executed sequentially, and a set of subtrees that can all be executed in parallel, so that we do not have any linear chain of subtrees. A similar idea was proposed in [3], where the goal was to reduce the makespan while limiting the memory in a shared-memory environment. **SplitSubtrees** adapts these ideas to the present model, which includes communications.

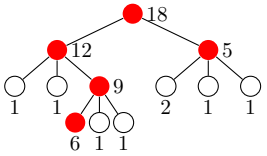


Figure 1: Example of a chain with **ASAP**.

*2) Improving the **SplitSubtrees** heuristic:* There are two main limitations of **SplitSubtrees**. First, it produces only a "two-level" solution: in the provided decomposition, all subtrees except one are the children of the subtree containing the root. In some cases, it is beneficial to split the tree into more levels. We thus designed a variant **ImprovedSplitV1**, which builds a multi-level solution. From the solution of **SplitSubtrees**, it tries to split again each of the children subtrees. To do this, it either uses processors initially left idle, or processors devoted to small subtrees that are then put back into the sequential set. It then outputs the solution that decreases the most the makespan.

The second limitation is the possibly too large size of the first subtree, containing the sequential set. As its execution is sequential, it may lead to a large resource waste. **ImprovedSplitV2** attempts to overcome this limitation by removing the largest subtree from the solution, and iteratively calling **SplitSubtrees** on the remaining trees. Thus, the initial sequential set can be split again into a number of subtrees.

*3) **ASAP** heuristic (as soon as possible):* The main idea of this heuristic is to parallelize the processing of tree τ as soon as possible, by cutting edges that are close to the root of the tree. **ASAP** uses a node priority queue PQ to store all the roots of subtrees produced. Nodes in PQ are sorted by non-increasing W_i, where W_i is the total computation weight of the subtree rooted at node i. Iteratively, it cuts the largest subtree, until there are as many subtrees as processors. Therefore, it is able to create a multi-level partition of the tree.

We also introduce a variant of the **ASAP** heuristic, named **ASAPc10**, which puts in PQ not only the children of the node that was selected as the root of a subtree the latest, but also all its descendants up to a depth of 10. All these nodes are candidates to be selected as subtree's root, and are sorted by non-increasing values of $W_i - \frac{f_i}{\beta}$. Here, we take into account the communication cost, as it corresponds to the gain of moving this subtree to another processor (we remove some computation from the tree, but add a communication cost). At each step, the best candidate in this set is selected to become the root of a new subtree. The value of 10 was selected as it seems a good tradeoff between performance and running time, but this heuristic can be generalized for any depths.

4) Avoiding chains of subtrees: The tree in Figure 1 provides an example in which **ASAP** cuts four edges (edges between red nodes) and maps each subtree to one processor. Node labels represent their subtree computational weight, all edges have weight 10, and $p = 6$. Compared to executing the tree on one processor, the parallel executing scheme from **ASAP** costs more due to the communication between processors. We are hence looking for chains in the quotient

tree, where vertices from a same subtree are represented by a single vertex, and dependencies are added in the quotient tree if and only if there is a dependency between two of the original nodes.

We propose an algorithm to avoid this shortcoming, called **AvoidChain**. We first build the quotient tree and look for chains in it. The subtrees of a chain are then merged into a single subtree, which leaves some processors idle. These idle processors can then be used to improve the makespan thanks to Algorithm **LarSav**, as explained below.

5) Increasing the number of subtrees: If the number of subtrees is smaller than the number of processors, we can further reduce the makespan by repartitioning subtrees. Given a tree τ and a partition C, **LarSav** first builds the quotient tree, and finds its critical path, which defines the makespan of τ. All subtrees on the critical path, except leaves of the quotient tree, are candidates to be cut into two parts. For each subtree that is a leaf in the quotient tree, we try to cut its two largest children, which avoids producing a chain. We iteratively select the subtrees and the corresponding splittings that reduce the most the makespan. We repeat this process until no more idle processor is left, or when the makespan stops decreasing. We select the solution that gives the largest savings in makespan.

B. Tree partitioning with memory constraints

We now move to the general case where the memory needed to process the whole tree, $MinMemory(\tau)$, is larger than the available memory M. Again, we want to partition the tree so as to minimize the makespan. We consider two-step heuristics: (Step 1) We first partition the tree into k subtrees so that each of these fits in the local memory; k should not be larger than the number of processors p, otherwise we fail to find a solution; (Step 2) Then, if $k < p$, we try to make use of the remaining $(p - k)$ processors to reduce the makespan, by repartitioning some of the subtrees obtained in Step 1. We first present three heuristics for Step 1, and then move to Step 2.

1) Partitioning trees for the memory: We first note the proximity of this problem with the MINIO problem [4]. In this problem, a similar tree has to be executed on a single processor with limited memory. When the memory shortage happens, some data have to be evicted from the main memory and written to disk. The goal is to minimize the total volume of the evicted data while processing the whole tree. In [4], six heuristics are designed to decide which files should be evicted. In the corresponding simulations, the **FirstFit** heuristic demonstrated better results. It first computes the traversal (permutation σ of the nodes that specifies their execution sequence) that minimizes the peak memory, using the provided MINMEMORY algorithm [4]. Given this traversal, if the next node to be processed, denoted as j, is not executable due to memory shortage, we have to evict some data from the memory to the disk. **FirstFit** orders the set $S = \{f_{i_1}, f_{i_2}, \ldots, f_{i_j}\}$ of the data already produced and still residing in main memory, so that f_{i_1} is the data that will be used for processing the latest, and selects the first data from S until enough memory has been freed. We consider the simple adaptation of **FirstFit** to our problem: the final set of data F that are evicted from the memory defines the edges that are cut in the partition of the tree, thus resulting in $|F| + 1$ subtrees. This guarantees that each subtree can be processed without exceeding the available memory, but not that the number of subtrees is smaller than p.

It seems natural to minimize the number of subtrees in Step 1, as we deal with makespan minimization possibly by repartitioning subtrees, in Step 2. Thus, we propose a variant of the **FirstFit** strategy, which orders the set S of candidate data to be evicted by non-increasing sizes f_i, and selects the largest data until their total size exceeds the required amount. This heuristic is called **LargestFirst**.

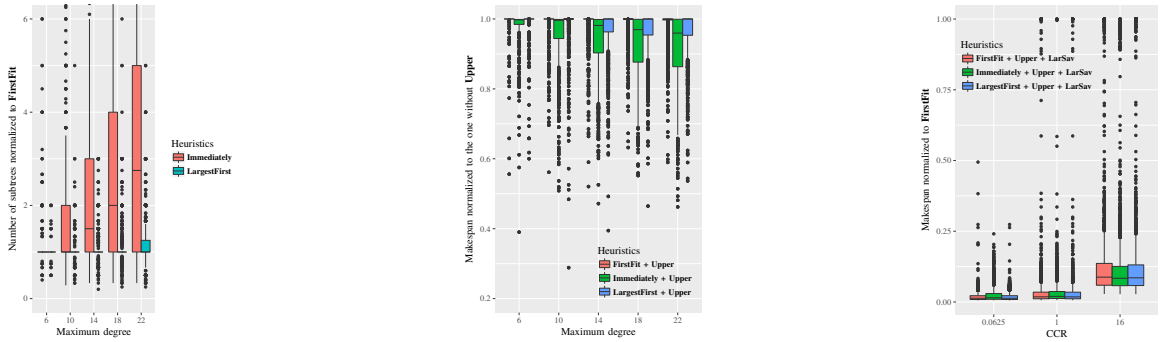
Finally, we propose a third and last heuristic to partition a tree into subtrees that fit into memory. As for the previous heuristic, we start from a minimum memory sequential traversal σ. We simulate the execution of σ, and each time we encounter a node that is not executable because of memory shortage, we cut the corresponding edge and this node becomes the root of a new subtree. We continue the process for the remaining nodes, and then recursively apply the same procedure on all created subtrees, until each of them fit in memory. This heuristic is called **Immediately**.

2) Optimizing a partition for makespan: When there are no more subtrees than processors, we first attempt to slightly modify this partition using heuristic **Upper**. The main idea is to reduce the workload of the sequential part, that is, the subtree that contains the root of the tree, since no other subtree can be executed simultaneously. For this purpose, we start by the subtree containing the root and look for its children in the quotient graph. We order these children by increasing makespan, so as to start with the less loaded subtrees, and try to grow the corresponding subtrees by including its parent. We iteratively repeat this process until the memory needed for the updated subtree is larger than the bound M, or until we create some dependency with another child. We then select the extension of the current subtree that reduces the most the makespan, before trying to grow other subtrees. All children subtrees are then put in the list of candidate subtrees to be later considered for optimization, so that the optimization propagates from the top to the bottom of the tree.

Finally, note that it is also possible to apply the **LarSav** heuristic described earlier if there are additional processors available. However, all heuristics will fail if the heuristic used in Step 1 returns a partition requiring too many processors.

IV. EXPERIMENTAL VALIDATION THROUGH SIMULATIONS

We generated ten groups of trees, each one with 3,000 trees, whose size ranges from 1,000 to 6,000. The maximum number of children of a node (called "maximum degree" in the following) is constant in a given group and ranges from 4 to 22. The trees with smaller maximum degree are thus generally deeper but narrower than the ones with larger maximum degree. The sizes of the nodes' input data (f_i) follow a truncated exponential distribution with mean value 100, where the values smaller than 10 are removed. The execution time (w_i) of each node is randomly generated in the same way.



(a) Number of subtrees produced by the heuristics, compared to **FirstFit**.

(b) Decrease in the makespan using **Upper**.

(c) **LarSav** further reduces the makespan ($n = 0.10p$).

Figure 2: Performance of the heuristics with memory constraint.

We then set the size of execution file (m_i) as three times its input data size. We only consider trees whose $MinMem$ is larger than its $MaxOutDeg$, others are discarded. Recall that $MaxOutDeg = \max_{1 \leq i \leq n}(MemReq(i))$, where $MemReq(i)$ denotes the memory requirement of task i.

To compare the performance of the proposed heuristics in different environments, we have selected three different options for the number p of processors: it is equal to 1%, 10%, or 40% of the tree size n. We also consider three scenarios for the relative cost of computations vs. communications. Given a tree, we select the communication bandwidth β such that the average communication to computation ratio (CCR), defined as the total time of the computations divided by the total time for communicating all data, is either $1/16 (= 0.0625)$, 1 or 16.

Due to lack of space, we do not detail results for the case without memory constraint, and we refer to the companion research report [1]. Note that **ASAPc10** generally performs better than the reference heuristic **SplitSubtrees**, except in the case of a too small number of processors ($n/p = 100$). Using **LarSav** on the solution produced by **ASAPc10** allows us to further reduce the makespan. **BestImprovedSplit** allows us to reduce the makespan of **SplitSubtrees**, but only by a small factor. When communication is expensive ($CCR = 16$), **ASAPc10** and **LarSav** are worse than **SplitSubtrees**: benefits from parallel execution are cancelled by communication costs. In other cases, **SplitSubtrees** is worse on more than half of the instances. We also find that the structure of trees does not have an obvious influence on the relative performance of heuristics.

In the memory-constrained case, the memory bound for each processor is set to the minimum memory needed to process any single task. This is thus a very strict scenario. The tree first has to be decomposed into some subtrees by either **FirstFit**, **LargestFirst** or **Immediately** such that none of them exceeds the bound M. The sequential subtree traversal is given by $MinMem$ as described in [4]. Each subtree is then mapped onto a processor. First, we compare how many subtrees these algorithms produced, since using less subtrees in this first phase leaves more room for makespan improvement in the second phase. **FirstFit** produces less subtrees, as shown in Figure 2a, and **LargestFirst** behaves almost the same on trees with a small maximum degree. So, **FirstFit** is the best heuristic that gives us a feasible partition when processors are limited.

Upper is designed to reduce the makespan without using more processors. Figure 2b shows that on average, it achieves only small makespan reductions, while it performs better on trees with larger maximum degree, especially for the partition built by **Immediately**. After **Upper**, **LarSav** can further reduce the makespan by using idle processors: 90% of processors are idle on at least 75% of the cases. Figure 2c shows that the makespan returned by **LarSav** is much smaller. In conclusion, **FirstFit** is the best option for computing a first partition, **Upper** and **LarSav** are then very helpful to reduce its makespan.

V. Conclusion

We have studied how to minimize the time required to compute a tree of tasks on a distributed computing platform with memory constraints. After proving the problem NP-complete, we designed several heuristics. Extensive simulations demonstrate the efficiency of these heuristics and provide guidelines about the heuristic that should be used. Without memory constraints, using a combination of **ASAPc10** and **LarSav** is the best choice in most settings, even though **BestImprovedSplit** may be useful with a few processors or expensive communications. With memory constraints, the combination of **FirstFit** with **Upper** and **LarSav** is the best choice.

References

[1] A. Benoit, C. Gou, and L. Marchal. Memory-aware tree partitioning on homogeneous platforms. Research Report RR-9115, INRIA, November 2017. Available at hal.inria.fr/hal-01644352.

[2] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. Society for Ind. and Applied Math., Philadelphia, 2006.

[3] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien. Parallel scheduling of task trees with limited memory. *ACM Transactions on Parallel Computing*, 2(2):13, 2015.

[4] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. On optimal tree traversals for sparse matrix factorization. In *IPDPS 2011*, pages 556–567, Anchorage, Alaska, USA, 2011.

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

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

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