Memory-aware tree partitioning on homogeneous platforms

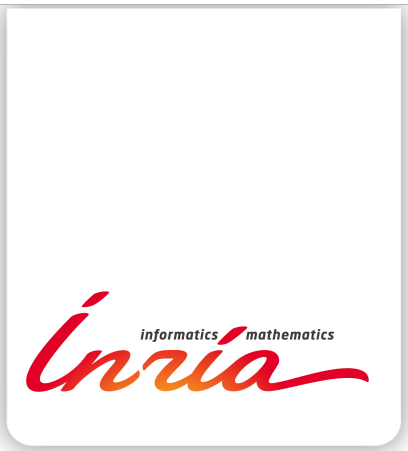
Changjiang Gou, Anne Benoit, Loris Marchal

Memory-aware tree partitioning on homogeneous platforms

Anne Benoit, Changjiang Gou, Loris Marchal

RESEARCH REPORT
N° 9115
November 2017
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9115--FR+ENG



Memory-aware tree partitioning on homogeneous platforms

Anne Benoit, Changjiang Gou, Loris Marchal

Project-Team ROMA

Research Report n° 9115 — November 2017 — 25 pages

Abstract: Scientific applications are commonly modeled as the processing of directed acyclic graphs of tasks, and for some of them, the graph takes the special form of a rooted tree. This tree expresses both the computational dependencies between tasks and their storage requirements. The problem of scheduling/traversing such a tree on a single processor to minimize its memory footprint has already been widely studied. Hence, we move to parallel processing and study how to partition the tree for a homogeneous multiprocessor platform, where each processor is equipped with its own memory. We formally state the problem of partitioning the tree into subtrees such that each subtree can be processed on a single processor and the total resulting processing time is minimized. We prove that the problem is NP-complete, and we design polynomial-time heuristics to address it. An extensive set of simulations demonstrates the usefulness of these heuristics.

Key-words: Scheduling, graph partitioning, memory-aware, makespan minimization

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Partitionnement d'arbres de tâches orienté mémoire pour les plates-formes homogènes

Résumé : Les applications scientifiques sont couramment modélisées par des graphes de tâches. Pour certaines d'entre elles, le graphe prend la forme particulière d'un arbre enraciné. Cet arbre détermine à la fois les dépendances entre tâches de calcul et les besoins en stockage. Le problème d'ordonnancer (ou parcourir) un tel arbre sur un seul processeur pour réduire son empreinte mémoire a déjà été largement étudié. Dans ce rapport, nous considérons le traitement parallèle d'un tel arbre et étudions comment le partitionner pour une plate-forme de calcul formée de processeurs homogènes disposant chacun de sa propre mémoire. Nous formalisons le problème du partitionnement de l'arbre en sous-arbres de telle sorte que chaque sous-arbre puisse être traité sur un seul processeur et que le temps de calcul total soit minimal. Nous montrons que ce problème est NP-complet et proposons des heuristiques polynomiales. Un ensemble exhaustif de simulations permet de montrer l'utilité de ces heuristiques.

Mots-clés : Ordonnancement, partitionnement de graphe, algorithmes orientés mémoire

1 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 [3], or in computational physics code modeling electronic properties [12]. 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. The potentially large size of the output files makes it crucial to find a traversal that reduces the memory requirement. In the case where even the minimum memory requirement is larger than the local memory capacity, a good way to solve the problem is to partition the tree and map the subtrees onto a multiprocessor computing system in which each processor has its own private memory. Partitioning makes it possible to both reduce memory requirement and to improve the processing time (or makespan) by doing some processing in parallel, but it also incurs communication costs. On modern computer architectures, the impact of communications between processors on both time an energy is non negligible.

The problem of scheduling the nodes of a tree on a single processor with minimum memory requirement has been studied before, and memory optimal traversals have been proposed [15, 10]. The problem of scheduling such a tree on a single processor with limited memory is also discussed in [10]: 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. These files will be retrieved later, when the corresponding node is scheduled. The total volume of data written to (and read from) the secondary storage is called the Input/Output volume (or I/O volume), and the objective is then to find a traversal with minimum I/O volume.

In this work, we consider that the target platform is a multi-processor platform, each processor being equipped with its own memory. The platform is homogeneous, as all processors have the same computing power and the same amount of memory. In the case of memory shortage, rather than performing I/O operations, we send some files to another processor that will handle the processing of the corresponding subtree. If the tree is a linear chain, this will

only slow down the computation since communications need to be paid. However, if the tree is a fork graph, it may end up in processing different subtrees in parallel, and hence potentially reducing the makespan. The time needed to execute a subtree is the sum of the time for the communication of the input file of its root and the computation time of each task in the subtree. The MinMakespan problem then consists in dividing the tree into subtrees, each subtree being processed by a separate processor, so that the makespan is minimized. The memory constraint states that we must be able to process each subtree within the limited memory of a single processor.

The main contributions of this paper are the following:

- We formalize the MinMakespan problem, and in particular we explain how to express the makespan given a decomposition of the tree into subtrees;
- We prove that MinMakespan is NP-complete;
- We design several polynomial-time heuristics aiming at obtaining efficient solutions;
- We validate the results through a set of simulations.

The paper is organized as follows. Section 2 gives an overview of related work. Then, we formalize the model in Section 3. In Section 4, we show that MinMakespan is NP-complete. All the heuristics are presented in Section 5, and the experimental evaluation is conducted in Section 6. Finally, we give some concluding remarks and hints for future work in Section 7.

2 Related work

As stated above, rooted trees are commonly used to represent task dependencies for scientific applications. In the domain of sparse linear algebra, Liu [16] gives a detailed description of the construction of the elimination tree, its use for Cholesky and LU factorizations and its role in multifrontal methods. In [14], Liu introduces two techniques for reducing the memory requirement in post-order tree traversals. In the subsequent work [15], the post-order constraint is dropped and an efficient algorithm to find a possible ordering for the out-of-core multifrontal method is given. Building upon Liu's work, some of us [10] proposed a new exact algorithm for exploring a tree with the minimum memory requirement, and studied how to minimize the I/O volume when out-of-core execution is required. The problem of general task graphs handling large data has also been identified by Ramakrishnan et al. [19], who propose some simple heuristics. Their work was continued by Bharathi et al. [1], who develop genetic algorithms to schedule such workflows.

On the more theoretical side, this work builds upon many papers that have addressed the pebble game and its variants. Scheduling a graph on one processor with the minimum amount of memory amounts to revisiting the Input/Output pebble game with pebbles of arbitrary sizes that must be loaded into main memory before *firing* (executing) the task. The pioneering work of Sethi and Ullman [22] deals with a variant of the pebble game that translates into the

simplest instance of our problem where all input/output files have weight 1, all execution files have weight 0 and the graph is a rooted tree, where the aim is to minimize the number of registers that are needed to compute an arithmetic expression.

Partitioning a tree, or more generally a graph into separate subsets to optimize some metric has been thoroughly studied. Graph partitioning has varied applications in parallel processing, complex networks, image processing, etc. Generally, these problems is NP-hard. Exact algorithms have been proposed, which mainly rely on branch-and-bound framework [13], and are appropriate only for very small graphs and small number of resulting subgraphs. A large variety of heuristics and approximation algorithms to this problem have been presented. Some of them directly partition the entire graph, such as spectral partitioning that uses using eigenvector from Laplacian matrix to infer the global structure of a graph [5, 6], geometric partitioning that considers coordinates of graph nodes and projection to find a optimal bisecting plane [23, 18], streaming graph partitioning that uses much less memory and time, applied mainly in big data processing [24]. Their results can be iteratively improved by different types of strategies: node-swapping between adjacent subgraphs [11, 8, 20], graphing growing from some carefully selected nodes [4, 21], randomly choosing nodes to visit according to transition probabilities [17]. A multi-level scheme that consists of contraction, partitioning on the smaller graphs and mapping back to the original graph and improvement, can give a high quality results in a short execution time [2]. Compared to the classical graph partitioning studies which tend towards balanced partitions (subgraphs with approximately the same weight), our problem considers a more complex memory constraint on each subtree, which makes the previous work on graph partitioning unsuitable to find a good partitioning strategy.

3 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. Note that we may similarly consider precedence constraints oriented towards the root by reversing all schedules, as outlined in [10]. A precedence constraint $i \rightarrow j$ 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. More formally, let M be the size of the main memory of the processor, and let S be the set of files stored in this memory when the scheduler decides to execute task i. Note that S must contain the input file of task i. The processing of task i is

possible if we have:

$$MemReq(i) = f_i + m_i + \sum_{j \in children(i)} f_j \leq M - \sum_{j \in S, j \neq i} f_j,$$

where $MemReq(i)$ denotes the memory requirement of task i, and $children(i)$ are its children nodes in the tree. Intuitively, M should exceed the largest memory requirement over all tasks (denoted as $MaxOutDeg$ in the following), so as to be able to process each task:

$$MaxOutDeg = \max_{1 \leq i \leq n} (MemReq(i)) \leq M.$$

However, this amount of memory is in general not sufficient to process the whole tree, as input files of unprocessed tasks must be kept in memory until they are processed.

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. Of course, it must fit in memory to be executed. If the whole tree fits in memory and is executed sequentially on a single processor, 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 [15] or some of the authors' previous work [10]. 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

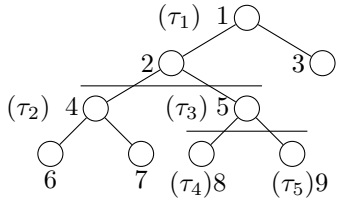


Figure 1: Recursive computation of makespan.

$i = r$). 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 $MS(r)$. Figure 1 exhibits an example of such a tree decomposition, where the horizontal lines represent the edges cut to disconnect the tree τ into five subtrees. Subtree τ_1 will be executed first after receiving its input file of size $f_1 = 0$, and it includes tasks 1, 2 and 3. Then, subtrees τ_2 and τ_3 will be processed in parallel. The final makespan for τ_1 is thus:

$$MS(1) = \frac{f_1}{\beta} + w_1 + w_2 + w_3 + \max(MS(4), MS(5)),$$

where $MS(5)$ recursively calls $\max(MS(8), MS(9))$, since τ_4 and τ_5 can also be processed in parallel.

We are now ready to formalize the optimization problem that we consider:

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.*

Given a tree τ and its partition into subtrees τ_i , we consider its quotient graph Q given by the partition: vertices from a same subtree are represented by a single vertex in the quotient tree, and there is an edge between two vertices $u \to v$ of the quotient graph if and only if there is an edge in the tree between two vertices $i \to j$ such that $i \in \tau_u$ and $j \in \tau_v$. Note that since we impose a partition into subtrees, the quotient graph is indeed a tree. This quotient tree will be helpful to compute the makespan and to exhibit the dependencies between the subtrees.

4 Problem complexity

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

Proof. First, it is easy to check that the problem belongs to NP: given a partition of the tree into $k \leq p$ subtrees, we can check in polynomial time that (i) the memory needed for each subtree does not exceed M, and that (ii) the obtained makespan is not larger than a given bound.

To prove the completeness, we use a reduction from 2-partition [9]. We consider an instance $\mathcal{I}_1$ of 2-partition: given n positive integers $a_1, \ldots, a_n$, does there exist a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i = S/2$, where $S = \sum_{i=1}^{n} a_i$. We consider the 2-partition-equal variant of the problem, also NP-complete, where both partitions have the same number of elements ($|I| = n/2$, and thus, n is even). Furthermore, we assume that $n \geq 4$, since the problem is trivial for $n = 2$. From $\mathcal{I}_1$, we build an instance $\mathcal{I}_2$ of MinMakespan as follows:

- The tree τ consists of $2n+1$ nodes, and it is described on Figure 2, where the weights on edges represent the size of input files f, and the computation time and memory requirements are indicated unless they are equal to 0. Formally, the tree consists of a linear chain $n+1 \to n+2 \to \cdots \to 2n+1$, and there are n additional tasks $1, \ldots, n$ such that $parent(i) = n+i$, for $1 \leq i \leq n$. For $1 \leq i \leq n$, we have $w_i = S - a_i$, $m_i = M - \sum_{j=1}^{i} a_j$, and $f_i = a_i$. Furthermore, $w_{n+k} = m_{n+k} = 0$ for $1 \leq k \leq n$, $w_{2n+1} = 0$, $m_{2n+1} = S/2$, and $f_{n+1+k} = M - \sum_{j=1}^{k} a_j$ for $1 \leq k \leq n$. Finally, $f_{n+1} = 0$ since $n+1 = r$ is the root of the tree.
- The makespan bound is $C_{\max} = (n+1)\frac{S}{2}$.
- The memory bound is $M = C_{\max} + S + 1$.
- The bandwidth is $\beta = 1$.
- The number of processors is $p = \frac{n}{2} + 1$.

Consider first that $\mathcal{I}_1$ has a solution, I, such that $I \subseteq \{1, \ldots, n\}$ and $|I| = n/2$ (i.e., I contains exactly $n/2$ elements). We execute sequentially tasks $n+1$ to $2n+1$, plus the tasks in I, and we pay communications and execute in parallel tasks not in I. We can execute each of these tasks in parallel since there are $n/2+1$ processors and exactly $n/2$ tasks not in I. We execute tasks $n+1, \ldots, 2n+1$ in this order. Since we have cut nodes not in I, there remains exactly files of size $S/2$ in memory, plus $f_{2n+1} = M - S$, and we also need to accommodate $m_{2n+1} = S/2$, hence we use exactly a memory of size M. We can then execute nodes in I starting from the bottom of the tree, without exceeding the memory bound. Indeed, once task $2n+1$ has been executed, there remains only some of the $f_i = a_i$'s in memory, and they fit together with m_i in memory. The makespan is therefore $\frac{n}{2}S - \frac{S}{2}$ for the sequential part (executing all tasks in I), and each of the tasks not in I can be executed within a time S (since $\beta = 1$), all of them in parallel, hence a total makespan of $(n-1)\frac{S}{2} + S = C_{\max}$. Hence, $\mathcal{I}_2$ has a solution.

Consider now that $\mathcal{I}_2$ has a solution. First, because of the constraint on the makespan, tasks $n+1$ to $2n+1$ must be in the same subtree, otherwise we would pay a communication of at least $M - S = C_{\max} + 1$, which is not acceptable. Let I be the set of tasks that are executed on the same subtree as these tasks $n+1, \ldots, 2n+1$. I contains at least $\frac{n}{2}$ tasks since the number of processors is

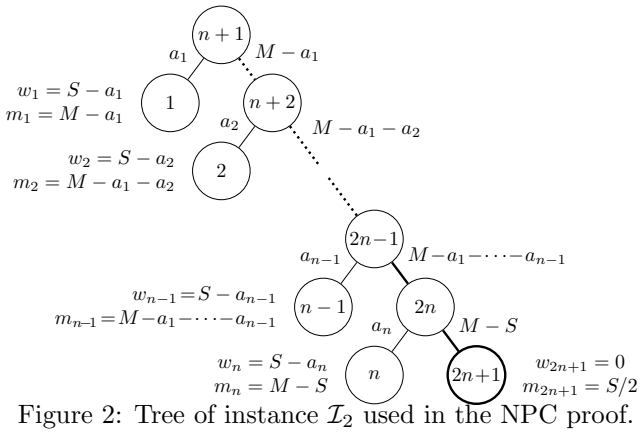


Figure 2: Tree of instance $\mathcal{I}_2$ used in the NPC proof.

$\frac{n}{2} + 1$. If I contains more than $\frac{n}{2}$ tasks, then the makespan is strictly greater than $(\frac{n}{2} + 1)S - S$ for the sequential part, plus S for all other tasks done in parallel, that is $(\frac{n}{2} + 1)S > C_{\max}$. Therefore, I contains exactly $\frac{n}{2}$ tasks.

The constraint on makespan requires that $\frac{n}{2}S - \sum_{i \in I} a_i + S \leq C_{\max}$, and hence $\sum_{i \in I} a_i \geq \frac{S}{2}$. Furthermore, as noted before, one cannot execute task $2n + 1$ unless enough memory has been freed from excluding some files from memory and paying some communication. The only files remaining in memory are the files from tasks in I, and hence the memory constraint writes $\sum_{i \in I} a_i + M - S + \frac{S}{2} \leq M$, hence $\sum_{i \in I} a_i \leq \frac{S}{2}$. Therefore, we must have $\sum_{i \in I} a_i = \frac{S}{2}$, and we have found a solution to $\mathcal{I}_1$. □

5 Heuristic strategies

In this section, we design polynomial-time heuristics to solve the MinMakespan problem. We first propose heuristics that do not take memory constraints into account (Section 5.1). Then, we move to the design of memory-aware heuristics, which always produce subtrees that do not exceed the memory constraint (Section 5.2).

5.1 Tree partitioning without memory constraints

In this section, we focus on the case where $\mathit{MinMemory}(\tau) \leq M$, 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.

5.1.1 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 [7], 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.

The **SplitSubtrees** heuristic relies on a splitting algorithm, which maintains a set of subtrees and iteratively *splits* the subtree with the largest total computation time until it reaches a leaf. A priority queue PQ is used to store all the current subtree roots, sorted by non-increasing total computation plus communication cost $W_i + \frac{f_i}{\beta}$.

At the beginning, PQ only contains the root of τ. Then, **SplitSubtrees** splits the largest subtree: it moves the head of PQ to the sequential set $seqSet$, and inserts all its children in PQ. Nodes in $seqSet$ will be first executed sequentially on a processor, and afterwards, if there are more than $p-1$ subtrees in PQ, the $|PQ|-(p-1)$ smallest subtrees (in terms of $W_i+\frac{f_i}{\beta}$) will be executed in the same processor as $seqSet$. The $p-1$ largest subtrees in PQ will be executed in parallel, each using a processor. Therefore, the makespan of splitting s is $MS_s = \sum_{i\in seqSet} w_i + W_{More} + \max_{k\in PQ}(\frac{f_k}{\beta} + \sum_{j\in\tau_k} w_j)$, where W_{More} is the sum of the W_i's of the smallest subtrees, if there are more than $p-1$ subtrees. This process is repeated until the head of PQ is a leaf node, in which case we cannot split it, and this node will take the longest time for parallel subtree execution. Finally, we select the splitting that results in the minimum makespan MS_s.

Note that each subtree is executed with a memory-optimal algorithm [15, 10] to ensure that the memory constraint is not exceeded. There are at most n possible splits, hence the algorithm is polynomial.

Algorithms 1 and 2 provide the details for this heuristic.

5.1.2 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, as illustrated on Figure 3, it is beneficial to split the tree into more

Algorithm 1 SplitSubtrees (τ, p)

1: Split tree τ into $q \leq p-1$ subtrees and a sequential set, using Algorithm 2.
2: Sequentially process the sequential set, using a memory minimizing algorithm, e.g., [10].
3: Concurrently process the q subtrees, each using a memory minimizing algorithm.

Algorithm 2 SplitSubtrees (τ, p)

1: **for all** nodes $i \in \tau$ **do**
2: compute $W_i = w_i + \sum_{k \in desc(i)} w_k$ (the total processing time of the subtree rooted at i)
3: **end for**
4: $PQ \leftarrow \{r\}$ (the priority queue consists of the tree root)
5: $seqSet \leftarrow \emptyset$
6: $MS_0 = W_r$
7: $s \leftarrow 1$ (splitting rank)
8: **while** $W_{head(PQ)} > w_{head(PQ)}$ **do**
9: $node \leftarrow popHead(PQ)$
10: $seqSet \leftarrow seqSet \cup node$
11: $PQ \leftarrow PQ \setminus \{node\} \cup children(node)$
12: **if** $|PQ| > p-1$ **then**
13: Let $More$ denote the $|PQ| - (p-1)$ smallest nodes, in terms of W_i, in PQ
14: $W_{More} = \sum_{i \in More} W_i$
15: **else**
16: $W_{More} = 0$
17: **end if**
18: $MS_s = \sum_{i \in seqSet} w_i + W_{More} + \max_{k \in PQ}(\frac{f_k}{\beta} + \sum_{j \in \tau_k} w_j)$
19: $s \leftarrow s+1$
20: **end while**
21: select splitting s' that leads to the smallest MS_s
22: put all edges cut in splitting s' to a set C
23: **return** C

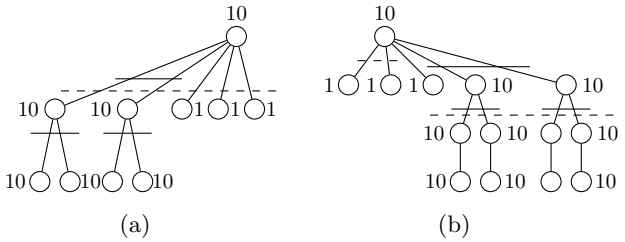


Figure 3: Two cases where **SplitSubtrees** is suboptimal ($p = 7$). Node labels denote their computational weight. The horizontal dashed lines represent the edges cut in the solution of **SplitSubtrees**, while solid lines represent the optimal partition.

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, as in Figure 3a. 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 *seqSet*. 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, as illustrated on Figure 3b. Thus, the initial sequential set can be split again into a number of subtrees.

Note that in this algorithm, because some parts of the tree are removed, we need to use a special version of **SplitSubtrees** (on line 10 of Algorithm 4), which takes into account the removed part to compute the actual makespan of the current splitting. Please refer to Algorithm 4 for details.

5.1.3 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 (see Algorithm 5 for details). Therefore, it creates 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 (line 1 and line 10 of Algorithm 5). 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

Algorithm 3 ImprovedSplitV1 (τ, p)

$i = 0$
$C_i \leftarrow$ **SplitSubtrees**(τ, p) (C_i: roots of subtrees in parallel part)
Let MS_i be the makespan of τ according to partition C_i
$q = |C_i|$, $p_{idle} = p - |C_i| - 1$
$C \leftarrow C_i$
while $q + p_{idle} \geq 3$ **do**
  $i = i + 1$
  $\tau' \leftarrow$ the leaf subtree on critical path
  $C_i \leftarrow$ **SplitSubtrees**$(\tau', 3)$
  Let MS_i be the makespan of τ with the partition $C \cup C_i$
  $C \leftarrow C \cup C_i$
  remove the $\max(0, 2 - p_{idle})$ smallest nodes from C_i
  $q = q - \max(0, 2 - p_{idle})$
  if $parent(\tau') = root$ of τ **then**
    $q = q - 1$
  end if
  if $p_{idle} > 0$ **then**
    $p_{idle} = p_{idle} - \min(2, p_{idle})$
  end if
end while
denote j the splitting so that MS_j is minimum
$C \leftarrow C_0 \cup C_1 \cup \cdots \cup C_{j-1} \cup C_j$
return C

Algorithm 4 ImprovedSplitV2 (τ, p)

$< C_0, MS_0 > \leftarrow SplitSubtrees(\tau, p)$
construct the quotient tree Q according to τ and C_0
$i = 0$, $C \leftarrow \emptyset$
repeat
  $S \leftarrow \emptyset$
  move all nodes with maximum $W + \frac{f}{\beta}$ in C_i to S
  remove subtrees that are rooted at nodes in S from τ
  $p = p - |S|$
  $i = i + 1$
  $< C_i, MS_i > \leftarrow SplitSubtrees(\tau, p, Q)$
until $p = 1$
denote i' the splitting so that $MS_{i'}$ is minimum
$C \leftarrow C_{i'}$
return C

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. Then, the this new root and its ancestors are removed from PQ at line 9.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.

Algorithm 5 ASAP (τ, p)

1: $PQ \leftarrow$ children of the root of τ, sorted by non-increasing W
2: $i = 0, C_i \leftarrow \emptyset$
3: Let MS_i be the makespan of τ with partition C_i
4: **repeat**
5: $\quad i = i + 1$
6: $\quad$ **if** PQ is empty **then**
7: $\quad\quad$ break;
8: $\quad$ **end if**
9: $\quad u \leftarrow popHead(PQ)$
10: $\quad$ insert $Children(u)$ into PQ (and keep it sorted)
11: $\quad C_i \leftarrow C_{i-1} \cup \{u\}$ (the edges we cut)
12: $\quad$ Let MS_i be the makespan of the tree with partition C_i
13: **until** $|C_i| = p - 1$
14: select j with minimum MS_j
15: **return** C_j

5.1.4 Avoiding chains of subtrees

The tree in Figure 4 provides an example in which **ASAP** cuts four edges (edges between red nodes) and maps each subtree to one processor. Compared to executing the tree on one processor, the parallel executing scheme from **ASAP** costs more due to the communication between processors. More formally, a partition that creates a chain in the quotient tree, as defined below, wastes processors and communication time.

Definition 2 (Chain)**.** *Given a tree τ, its partition into subtrees and the result-*

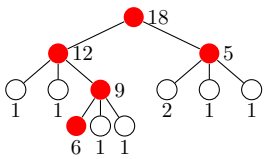


Figure 4: Example showing that a chain always wastes processors. Node labels represent their subtree computational weight. All edges have weight 10, and $p = 6$. Red nodes denote subtrees' roots as determined by **ASAP**.

ing quotient tree Q, a chain is a set of nodes $u_1, \dots u_k$ of Q such that u_i is the only child of u_{i-1} ($i > 1$).

We propose an algorithm to avoid this shortcoming, called **AvoidChain** (see Algorithm 6). 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.

Algorithm 6 AvoidChain (τ, C)

```
1: Construct a Q from τ and C
2: Initialize a queue que with root of Q
3: i ← 0
4: while que is not empty do
5:    (to find all Chains)
6:    node ← popHead(que)
7:    if node has only one child then
8:       i ← i + 1, node ← children(node)
9:       push node to the end of Chain_i
10:   end if
11:   while node has only one child do
12:      push its child to the end of Chain_i
13:      node ← children(node)
14:   end while
15:   push children(node) to the end of que
16: end while
17: merge subtrees in Chains, remove these restored edges from C
18: return C
```

5.1.5 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 Q to model dependencies among subtrees, and finds its critical path. A critical path is a set of nodes of Q that defines the makespan of τ. All subtrees on the critical path except leaves of Q 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, hence the name of the heuristic. This is formalized as Algorithm 7.

Algorithm 7 LarSav(τ, C)

Compute the quotient tree Q and its critical path $CriPat$
Let ms be the makespan of τ with partition C
$n \leftarrow p - |C| - 1$ (number of idle processors)
while $n \geq 1$ **do**
  for all subtree τ_i on $CriPat$ **do**
    $C_i \leftarrow \emptyset$
    if subtree τ_i is a leaf of Q and $n \geq 2$ **then**
      $u \leftarrow$ *root* of τ_i
      while u has a single child v **do**
        $u \leftarrow v$
      end while
      if u has more than one child **then**
        $C_i \leftarrow$ largest two children of u
      end if
    end if
    if subtree τ_i is not a leaf **then**
      Let L be the set of nodes of τ_i with all their descendants in τ_i, and H be the subset of nodes of L whose parents are not in L
      $C_i \leftarrow \{$some node $k \in H$ with maximal $W_k\}$
    end if
    Let ms_i be the makespan of τ with partition $C \cup C_i$
  end for
  Select k such that $ms_k = \min_i ms_i$
  if $ms_k < ms$ **then**
    $C \leftarrow C \cup C_k$
    $n \leftarrow n - |C_k|$
    $ms \leftarrow ms_k$
  else
    return C
  end if
  update Q and $CriPat$
end while
return C

5.2 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. A first try could be to apply the previous strategies that do not take memory into account: in some cases, we might get lucky and end up with subtrees that all fit into the available memory of the processors. However, it is very likely that in some other cases, a subtree will require more memory than available. In this section, we focus on the design of memory-aware partitioning strategies. We consider two-step heuristics:

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;
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.

5.2.1 Partitioning trees for the memory

Our objective is now to partition the tree into a number k of subtrees such that each of these subtrees fits into the memory. For now, we are not really concerned about the value of k, except that it must not be larger than p.

We first note the proximity of this problem with the MinIO problem [10]. 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 [10], 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 [10]. 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. The amount of freed memory should be at least $Need(i) = (MemReq(j) - f_j) - M^{avail}$, where M^{avail} is the currently available memory when we try to execute node j. In that case, **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 $\sigma(i_1) > \sigma(i_2) > \cdots > \sigma(i_j)$, where $\sigma(i)$ is the step of processing node i in the traversal (f_{i_1} is the data that will be used for processing the latest) and selects the first data from S until their total size exceeds or equals $Need(j)$.

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

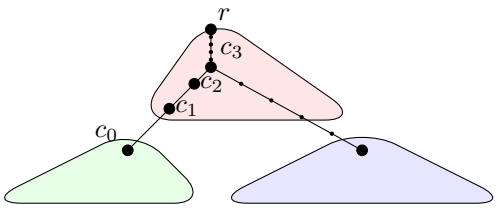


Figure 5: Example to show how **Upper** works: we first grow the green (left) subtree.

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 may result into edges with larger weights being cut, and thus an increased communication time, but it is likely to reduce the number of subtrees. 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**.

5.2.2 Optimizing a partition for makespan

We now move to the second step of the process detailed above. Once we have obtained a partition into subtrees that is valid for the memory, we strive to optimize its makespan.

All previous three heuristics return a partition in which each subtree fits in local memory. Note that it may fail when returning more subtrees than the amount of processors.

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. We illustrate this procedure on the example in Figure 5, where c_0 is the root of the first subtree in this order. We try to grow

this green subtree by including the parent c_1 of c_0, as well the other children of c_1. We iteratively repeat this process until the memory needed for the updated green subtree is larger than the bound M, or until we create some dependency with another child. In the example, this would happen if we include c_3 in the green subtree, as the blue subtree would become its child. Creating such a chain would dramatically impact the makespan, and we thus stop the process before it happens. We then select the extension of the current subtree that reduces the most the makespan, before trying to grow other subtrees (the blue one in the example). 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. This algorithm is detailed as Algorithm 8.

Finally, note that it is also possible to apply the **LarSav** heuristic 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.

Algorithm 8 Upper (τ, C, M)

Let ms be the makespan of τ with partition C
CandidateRoots $\leftarrow \{r\}$ (root of the tree)
repeat
  gain $\leftarrow 0$
  root $\leftarrow popHead($*CandidateRoots*$)$
  Let *ChildrenRoots* be the set of roots of the children of the subtree rooted at *root* in the quotient tree given by the C partition
  Sort *ChildrenRoots* by the non decreasing makespan
  while *ChildrenRoots* $\neq \emptyset$ **do**
    $c \leftarrow popHead($*ChildrenRoots*$)$
    $i \leftarrow 0$, $C_0 \leftarrow C$, $ms_0 \leftarrow ms$, $c_0 \leftarrow c$
    repeat
      $i \leftarrow i + 1$
      Let c_i be the parent of c_{i-1}, p the parent of c_i
      $C_i \leftarrow C_{i-1} \cup \{(p, c_i)\} \backslash \{(c_i, c_{i-1})\}$
      Let ms_i be the makespan of τ with partition C_i
    until $MinMemory(\tau_{c_i}) > M$ or c_i is an ancestor of some node $c' \in$ *ChildrenRoots*
    Select k in $0 \dots i-1$ which minimizes ms_k
    if $ms_k < ms$ **then**
      gain $\leftarrow$ *gain* $+ (ms - ms_k)$
      $C \leftarrow C_k$, $ms \leftarrow ms_k$, $c \leftarrow c_k$
    end if
    Push c at the end of *CandidateRoots*
  end while
until *gain* ≤ 0 or *CandidateRoots* $= \emptyset$
return C

6 Experimental validation through simulations

In this section, we compare the performance of the proposed heuristics on a wide range of computing platform settings. In particular, we evaluate them on the two cases previously mentioned, without memory constraints and with memory constraints. Note that the code of the simulations, as well as the datasets, will be made available on a public repository after the double-blind review process.

6.1 Dataset and simulation setup

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. 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 $\mathit{MaxOutDeg} = \max_{1\leq i\leq n}(\mathit{MemReq}(i))$, where $\mathit{MemReq}(i)$ 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.

In the memory-constrained case, the memory bound for each processor is set to the *MaxOutDeg*, the minimum memory needed to process any single task. This is thus a very strict scenario. The sequential tree traversal used in **FirstFit**, **LargestFirst** and **Immediately** is given by *MinMem* as described in [10], which has a minimum memory cost.

6.2 Without memory constraints

In Section 5.1.3, we proposed the **ASAP** heuristic, as well as its **ASAPc10** variant, which considers more choices for candidate edges to be cut. Here, we first compare these two heuristics on Figure 6, for $p = 0.10n$. Note that the makespan obtained by **ASAPc10** is smaller on more than 75% of the cases where communication is expensive ($CCR = 16$), while both solutions have almost the same makespan for cheap communications ($CCR = 0.0625$). Therefore, in the following simulations, we choose to only use the **ASAPc10** variant.

The results of the various heuristics are shown on Figure 7a, where **BestImprovedSplit** corresponds to running the two improvements for **SplitSubtrees**

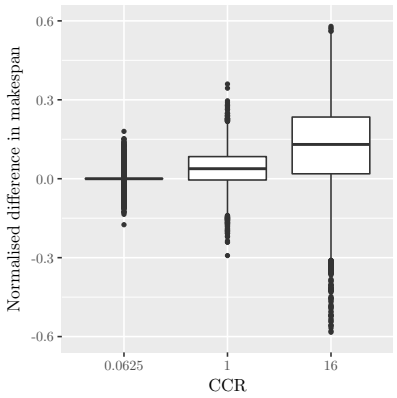


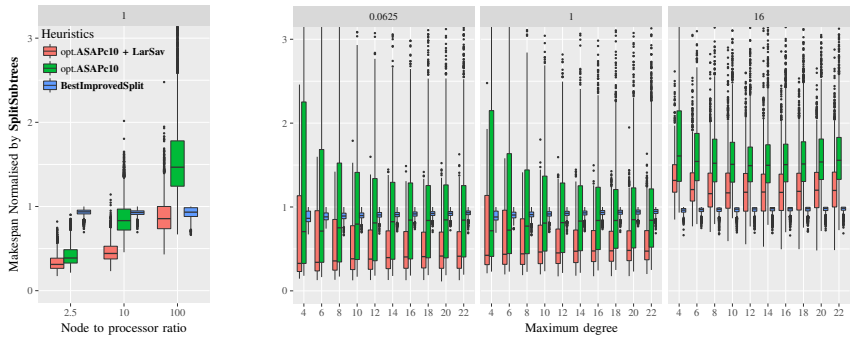
Figure 6: Normalized difference of makespan between **ASAP** and **ASAPc10** ((**ASAP**−**ASAPc10**)/**ASAP**) for various CCRs, without memory constraints.

(**ImprovedSplitV1** and **ImprovedSplitV2**) and selecting the variant with the smallest makespan. In this figure, the lower is the better, since it corresponds to smaller makespan. Hence, **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. Finally, **BestImprovedSplit** allows us to reduce the makespan of **SplitSubtrees**, but only by a small factor.

When communication is expensive ($CCR = 16$), as Figure 7b depicts, **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. Unsurprisingly, **BestImprovedSplit** is always better, even though it does not reduce makespan a lot. We also find that the structure of trees does not have an obvious influence on the relative performance of heuristics.

6.3 With memory constraints

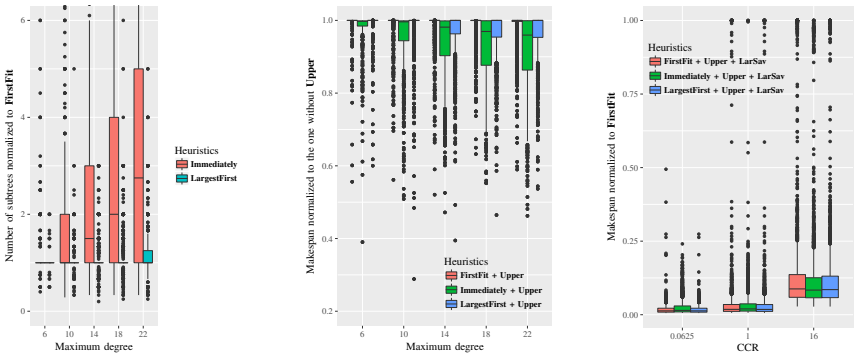
In the memory-constrained case, the tree has to be decomposed into some subtrees by either **FirstFit**, **LargestFirst** or **Immediately** such that none of them exceeds the bound M. Each subtree is then mapped onto a processor. First, we compare how many subtrees these algorithms produced, since using less subtrees in this first phase leaves more room for makespan improvement in the second phase. **FirstFit** produces less subtrees, as shown in Figure 8a, 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. It also leaves more room for other algorithms to reduce the makespan afterwards.



(a) For all cases, $CCR = 1$.

(b) Different values of the CCR. Same legend and y-axis as in Figure 7a.

Figure 7: Performance of the proposed heuristics without memory constraint.



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

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

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

Figure 8: Performance of the heuristics with memory constraint.

Upper is designed to reduce the makespan without using more processors. Figure 8b 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. In the solution returned by **Upper**, 90% of processors are idle on at least 75% of the cases, which gives **LarSav** many opportunities to improve the partition. As one can notice in Figure 8c, the makespan returned by **LarSav** is much smaller. However, there are some cases where no more processors are left idle, so that the makespan cannot be reduced any further. In a few extreme cases of our simulation, the partition returned needs more processors than we have, which results in a failure. In conclusion, **FirstFit** is the best option for computing a first partition, **Upper** and **LarSav** are then very helpful to reduce its makespan.

7 Conclusion

We have studied the tree partitioning problem, targeting at a multiprocessor computing system in which each processor has its own local memory. The tree represents dependencies between tasks, and it can be partitioned into subtrees, where each subtree is executed by a distinct processor. The goal is to minimize the time required to compute the whole tree (makespan), given some memory constraints: the minimum memory requirement of traversing each subtree should not be more than the local memory capacity. We have proved that the problem above, MinMakespan, is NP-complete, and we have designed several heuristics to tackle it with or without memory constraints. 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.

Building upon these promising results, an interesting direction for future work would be to consider partitions that do not necessarily rely on subtrees, but where a single processor may handle several subtrees. Also, we plan to extend this work to general directed acyclic graphs of task, while we have restricted the approach to trees so far.

References

[1] S. Bharathi and A. Chervenak. Scheduling data-intensive workflows on storage constrained resources. In *Proc. of the 4th Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*. ACM, 2009.

[2] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

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

[4] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive {FEM}. *Parallel Computing*, 26(12):1555 – 1581, 2000. Graph Partitioning and Parallel Computing.

[5] W. Donath and A. Hoffman. Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.

[6] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, Sept 1973.

[7] 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.

[8] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, June 1982.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[10] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. On optimal tree traversals for sparse matrix factorization. In *IPDPS 2011, 25th IEEE International Symposium on Parallel and Distributed Processing*, pages 556–567, Anchorage, Alaska, USA, 2011. IEEE Computer Society.

[11] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[12] 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.

[13] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *50 Years of Integer Programming 1958-2008*, pages 105–132, 2010.

[14] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Software*, 12(3):249–264, 1986.

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

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

[17] L. Lovász. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2:1–46, 1993.

[18] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 538–547, Washington, DC, USA, 1991. IEEE Computer Society.

[19] 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 *CCGrid'07*, pages 401–409, 2007.

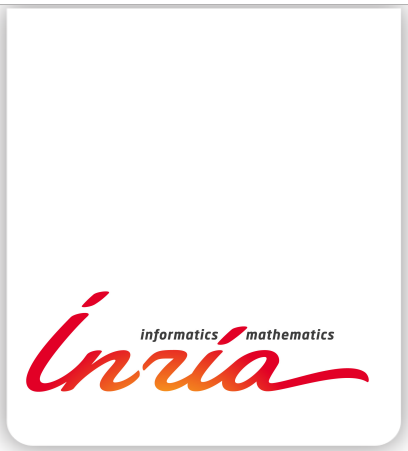
[20] L. A. Sanchis. Multiple-way network partitioning. *IEEE Trans. Comput.*, 38(1):62–81, Jan. 1989.

[21] S. Schamberger. On partitioning fem graphs using diffusion. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 277–, April 2004.

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

[23] H. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135 – 148, 1991.

[24] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399