HAL
open science

Partitioning tree-shaped task graphs for distributed platforms with limited memory

Anne Benoit, Changjiang Gou, Loris Marchal

To cite this version:

Anne Benoit, Changjiang Gou, Loris Marchal. Partitioning tree-shaped task graphs for distributed platforms with limited memory. [Research Report] RR-9115, Inria Grenoble Rhône-Alpes. 2019, pp.1-34. hal-01644352v4

HAL Id: hal-01644352

https://inria.hal.science/hal-01644352v4

Submitted on 5 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Partitioning tree-shaped task graphs for distributed platforms with limited memory

Anne Benoit, Changjiang Gou, Loris Marchal

RESEARCH REPORT
N° 9115
March 2019
Project-Team ROMA

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

Partitioning tree-shaped task graphs for distributed platforms with limited memory

Anne Benoit, Changjiang Gou, Loris Marchal

Project-Team ROMA

Research Report n° 9115 — March 2019 — 34 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. The present paper considers the parallel processing of such a tree and study how to partition it for a homogeneous multiprocessor platform, where each processor is equipped with its own memory. We formally state the problem of partitioning the tree into connected parts such that each part 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, tree partitioning, memory-aware, makespan minimization, parallel computing

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Partitionnement d'arbres de tâches pour des plates-formes distribuées avec limitation de mémoire

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 dépendance 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 largement été 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, minimisation du temps d'exécution, calcul parallèle

1 Introduction

Parallel workloads are often modeled as directed acyclic graphs of tasks. 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 [1], or in computational physics code modeling electronic properties [2]. The vertices (or 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 (the case of in-trees is similar). For such out-trees, each node (except the root) receives an input file from its parent, and it 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 parts onto a multiprocessor computing system in which each processor has its own private memory and is responsible for a single part. 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 and energy is non negligible, furthermore in sparse solvers it can be the bottleneck at even a small core counts [3]. The problem of scheduling a tree of tasks on a single processor with minimum memory requirement has been studied before, and memory optimal traversals have been proposed [4, 5]. The problem of scheduling such a tree on a single processor with limited memory is also discussed in [5]: 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 (MinIO problem).

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. We propose to partition the tree into parts that are connected components, such each part correspond to

a tree (which is embedded in the whole task tree). The time needed to execute such a part 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 part. The MinMakespan problem then consists in dividing the tree into parts that are connected components, each part being processed by a separate processor, so that the makespan is minimized. The memory constraint states that we must be able to process each part 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 connected components;
- We prove that MinMakespan is NP-complete;
- We design several polynomial-time heuristics aiming at obtaining efficient solutions;
- We evaluate the proposed heuristics 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. For instance, in dense linear algebra libraries such as SuperMatrix [6] and Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) [7], the dependencies between tasks are well identified, leading to an efficient asynchronous parallel execution of tasks. However, in sparse linear algebra, scheduling trees is more difficult because of enormous tasks' amount and their irregular weights [8]. Liu [9] 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 [10], Liu introduces two techniques for reducing the memory requirement in post-order tree traversals. In the subsequent work [4], the post-order constraint is dropped and an efficient algorithm to find a possible ordering for the multifrontal method is given. Building upon Liu's work, some of us [5] 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. [11], who propose some simple heuristics. Their work was continued by Bharathi et al. [12], who develop genetic algorithms to schedule such workflows.

Recently, many studies have been published on parallel sparse direct solver on multicore shared memory or distributed memory model to reduce the communication and execution time. Kim et al. [8] propose a two-level task parallelism algorithm, which first partitions the tree into many subtrees, and then further decomposes subtrees into regular fine-grained tasks. In this work, the scheduling of executing tasks of the first level is handled by OpenMP dynamically, which however may cause an arbitrarily bad memory consumption. In a later work, Kim et al. [13] take memory bound into consideration through Kokkos's [14]

dynamic task scheduling and memory management. Agullo et al. [15] also take advantage of two-level parallelism and discussed the ease of programming and the performance of the program. Targeting at distributed memory systems, Sao et al. [3] partition the tree into two levels, a common ancestor with its children, and then replicate the ancestor to processors that are in charge of children; both communication time and makespan are reduced by this method, at the expense of a larger memory consumption.

Partitioning a tree, or more generally a graph into separate subsets to optimize some metric has been thoroughly studied. Graph partitioning has various applications in parallel processing, complex networks, image processing, etc. Generally, these problems are NP-hard. Exact algorithms have been proposed, which mainly rely on branch-and-bound framework [16], and are appropriate only for very small graphs and small number of resulting subgraphs. A large variety of heuristics and approximation algorithms for this problem have been presented. Some of them directly partition the entire graph, such as spectral partitioning that uses eigenvector from Laplacian matrix to infer the global structure of a graph [17, 18], geometric partitioning that considers coordinates of graph nodes and projection to find an optimal bisecting plane [19,20], streaming graph partitioning that uses much less memory and time, applied mainly in big data processing [21]. Their results can be iteratively improved by different types of strategies: node-swapping between adjacent subgraphs [22–24], graphing growing from some carefully selected nodes [25,26], randomly choosing nodes to visit according to transition probabilities [27]. 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 [28]. For a concise review of graph partitioning, see [28].

When focusing on trees rather than general graphs, the balanced partitioning problem is still difficult [29]. It is APX-hard to approximate the cut size within any finite factor if subtrees are strictly balanced, some studies hence approximate the cut size as well as the balance, known as bicriteria-approximation [30]. When near-balance is allowed, tree partitioning is promising. Feldmann and Foschini [31] give a polynomial-time algorithm that cuts no more edges than an optimal perfectly balanced solution.

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 component, 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 vertices (or 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 [5]. 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 [4] or some of the authors' previous work [5]. 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 $k \leq p$ parts $\tau_1, \ldots, \tau_k$, which are connected components of the original tree. Hence, each part τ_i is itself a tree. We refer to these connected components as *subtrees* of τ. Note that τ can also be viewed as a tree made of these subtrees. Such a partition is illustrated on Figure 1, where the tree is decomposed into five subtrees: τ_1 with nodes 1, 2, and 3; τ_2 with nodes 4, 6, and 7; τ_3 with node 5; τ_4 with node 8; and τ_5 with node 9. We require that each subtree τ_i 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

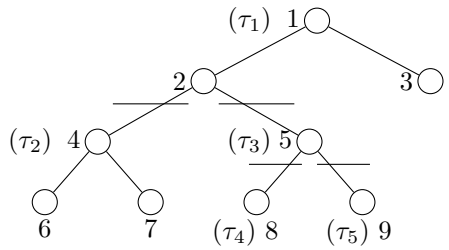

Figure 1: Tree partition and recursive computation of makespan.

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, not in $alloc(i)$, that have a parent in $alloc(i)$:

$$desc(i) = \{j \notin alloc(i) \mid parent(j) \in alloc(i)\}.$$

The makespan can then be expressed with a recursive formula. Let $MS(i)$ denote the time (or 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 (recall that $f_r = 0$ by convention):

$$MS(i) = \frac{f_i}{\beta} + \sum_{j \in alloc(i)} w_j + \max_{k \in desc(i)} MS(k).$$

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 is 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 are 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.

For convenience, we also denote by W_i the sum of the weights of all nodes in the subtree rooted in i (hence, for a leaf node, $W_i = w_i$):

$$W_i = w_i + \sum_{j \in children(i)} W_j.$$

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 [32]. 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 $n + 2$ nodes, and it is described on Figure 2: it is a fork graph (a root with $n+1$ children). The weights on edges represent the size of input files f_i, and the computation time and memory requirements are indicated respectively by w_i and m_i ($0 \leq i \leq n + 1$, where $root = 0$).
- For $1 \leq i \leq n$, $w_i = S - a_i$, $m_i = M - \sum_{j=1}^{i} a_j$, and $f_i = a_i$.
- For the last child, $w_{n+1} = 0$, $m_{n+1} = S/2$, and $f_{n+1} = M - S$.
- For the root, $w_{root} = 0$, $m_{root} = 0$, and $f_{root} = 0$.
- 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 root and task $n+1$, 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. 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 to execute task $n + 1$, we also need to accommodate $m_{2n+1} = S/2$, hence we

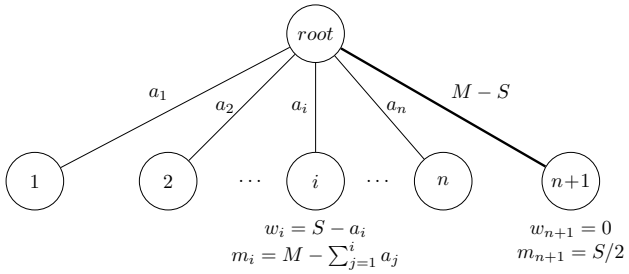

Figure 2: Tree of instance $\mathcal{I}_2$ used in the NPC proof.

use exactly a memory of size M. We can then execute nodes in I starting from the right of the tree, without exceeding the memory bound. Indeed, once task $n+1$ 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, *root* and task $n+1$ must be in the same subtree, otherwise we would pay a communication of $M - S = C_{\max} + 1$, which is not acceptable. Let I be the set of tasks that are executed on the same subtree as root and task $n+1$. I contains at least $\frac{n}{2}$ tasks, since the number of processors is $\frac{n}{2} + 1$. 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 $\sum_{i \in I} a_i \geq \frac{S}{2}$. After executing *root*, the files remaining in memory are the files from tasks in I and f_{n+1}, since other files are communicated to other processors. As long as f_{n+1} is in memory, no task of I can be executed due to the memory constraint, hence to execute task $n+1$, 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. The heuristics work in three steps: (1) partition the tree into subtrees in order to minimize the makespan, without accounting for the memory constraint; (2) partition subtrees that do not fit in memory, i.e., such that $MinMemory(\tau_i) > M$; (3) ensure that we do have the correct number of subtrees, i.e., merge some subtrees if there are more subtrees than processors, or further split subtrees if there are extra processors and the makespan can be reduced. We now detail the three steps, focusing on makespan, then memory, then number of processors.

5.1 Step 1: Minimizing the makespan

In the first step, 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 will consider the memory constraint on each subtree at the next step (Section 5.2).

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, as defined below, then the subtrees must be executed one after the other, no parallelism is exploited and unnecessary communication may occur.

Definition 2 (Chain)**.** *Given a tree τ, its partition into subtrees τ_i and the resulting 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} ($i > 1$).*

Therefore, having several subtrees as a linear chain can only increase the makespan, compared to an execution of the whole tree on a single processor.

We now propose four heuristics that aim at minimizing the makespan, and hence avoid having linear chains of subtrees.

5.1.1 Two-level heuristic

The first heuristic, **SplitSubtrees** is adapted from [33], where the goal was to reduce the makespan while limiting the memory in a shared-memory environment. It creates a two-level partition with one connected component containing the root, executed first on a single processor (and called the *sequential set*), followed by the parallel processing of $p-1$ independent subtrees. In the context of shared memory, this heuristic has been proven the two-level partition with best makespan [33, Lemma 5.1]. We adapt it to our context, in order to take communications into account.

The **SplitSubtrees** heuristic relies on a splitting algorithm, which maintains a set of subtrees and iteratively *splits* the subtree with the largest makespan. Initially, the only subtree is the whole tree. When a subtree is split, its root is moved to the sequential set (denoted *seqSet*) and all its children subtrees are added to the current set of subtrees. Algorithm 1 formalizes the heuristic, in which the current set of subtrees is stored in a priority queue PQ sorted by non-increasing makespan: $MS(i) = W_i + \frac{f_i}{\beta}$ (which accounts for communications).

For a given state of the algorithm (i.e., a partition of the tree between *seqSet* and subtrees in PQ), we consider the following mapping: the $p-1$ largest subtrees in PQ (in terms of total computation weight W) are allocated to distinct processors, while the remaining subtrees are processed by the same processor in charge of the sequential set. Note that all these nodes (*seqSet* plus the smallest subtrees of PQ) form a connected component of the original tree: *seqSet* is a

Algorithm 1 SplitSubtrees (τ, p)

for all nodes $i \in \tau$ **do**
  $MS(i) = W_i + \frac{f_i}{\beta}$;
end for
$PQ_0 \leftarrow \{r\}$; (the priority queue consists of the tree root)
$seqSet \leftarrow \emptyset$;
$MS_0 = MS(r)$;
$s \leftarrow 1$; (splitting rank)
while $head(PQ_{s-1})$ is not a leaf **do**
  $i \leftarrow popHead(PQ_{s-1})$;
  $seqSet \leftarrow seqSet \cup \{i\}$;
  $PQ_s \leftarrow PQ_{s-1} \setminus \{i\} \cup children(i)$;
  if $|PQ_s| > p - 1$ **then**
    Let $\mathcal{S}$ denote the $|PQ_s| - (p-1)$ smallest nodes, in terms of W_i, in PQ_s;
  else
    $\mathcal{S} = \emptyset$;
  end if
  $MS_s = \sum_{i \in seqSet} w_i + \sum_{i \in \mathcal{S}} W_i + \max_{k \in PQ_s \setminus \mathcal{S}}(MS(k))$;
  $s \leftarrow s + 1$;
end while
select splitting s^* that leads to the smallest MS_{s^*};
return PQ_{s^*};

connected component containing the root, and each root of a subtree in PQ has its parent in $seqSet$.

We iteratively consider the solutions obtained by the successive splitting operations and finally select the one with the best makespan. We stop splitting subtrees when the largest subtree in PQ is indeed a leaf. Thus, there are at most n iterations, hence the algorithm is polynomial. The algorithm returns the set of nodes that are the root of a subtree, which corresponds to a *cut* of the tree, i.e., the set of edges that are cut to partition the tree into subtrees.

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

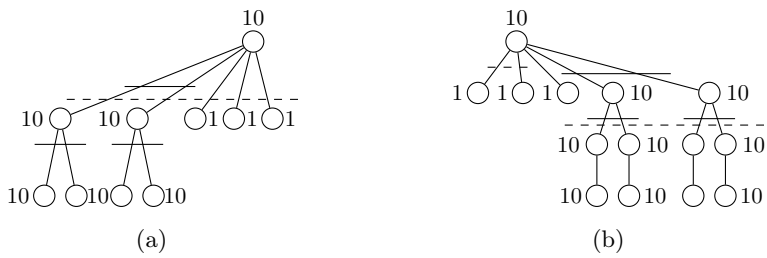

Figure 3: Two cases where **SplitSubtrees** is suboptimal. Dashed edges represent the solution of **SplitSubtrees**, plain edges give the optimal partition.

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 levels. In these examples, we have $p = 7$ processors. Node labels denote their computational weights (10 for all nodes, except three of them per tree), and there are no communication costs. The horizontal dashed lines represent the edges cut in the solution of **SplitSubtrees**, while solid lines represent the optimal partition. On the example of Figure 3(a), a two-level solution cannot achieve a makespan better than 40. If the cut was made at a lower level, the makespan would be even greater. It is however possible to achieve a makespan of 33 by cutting at two levels.

The second limitation is the possibly too large size of the first subtree, containing the sequential set $seqSet$. Since its execution is sequential, it may lead to a large resource waste. This is for instance the case in the example of Figure 3(b), where the optimal two-level solution has a sequential set whose execution time is 31, while further parallelism could have been used: the optimal solution cuts this sequential set in order to minimize the makespan.

To address these limitations, we design a new heuristic, **ImprovedSplit**, which improves upon **SplitSubtrees** by building a multi-level solution. Since we aim at further cutting the tree to obtain a multi-level solution, **ImprovedSplit** does not set a limit on the number of subtrees in a first step, but rather tries to create as many subtrees as possible, while the makespan can be improved. From the solution PQ of **SplitSubtrees**(τ, n) (with n processors, where n in the number of nodes in the tree), **ImprovedSplit** recursively tries to split again the largest children subtrees (subtrees whose roots are in PQ) until the makespan cannot be reduced further. Also, **ImprovedSplit** tries to further split the sequential set with a recursive call. Note that since the tree may include several levels of subtrees as **ImprovedSplit** is performed, the makespan of each subtree $MS(i)$ in **SplitSubtrees** must be computed with its definition of Section 3.

Finally, once all splits have been done, if there are more subtrees than processors, some of them will be merged by heuristic **Merge** (see Section 5.3), without accounting for the memory constraint.

5.1.3 ASAP heuristic

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's (recall that W_i is the total computation weight of the subtree rooted at node i). Iteratively, the heuristic cuts the largest subtree, if it has siblings, until there are as many subtrees as processors (see Algorithm 3 for details). Therefore, it creates a multi-level partition of the tree.

5.1.4 ASAP without chains of subtrees

One problem with **ASAP** is that it might create chains of subtrees (as defined above), which increases the makespan compared to a sequential execution of these subtrees. Figure 4 provides an example where this happens: the makespan is 11+2+12+2+10+(2+10) = 49, since the three leaf tasks of weight 10 are

Algorithm 2 ImprovedSplit (τ, p)

$PQ \leftarrow$ **SplitSubtrees**(τ, n);
for all i in PQ **do**
  τ_i is the subtree of τ rooted in i;
  $MS(i) = \sum_{j \in \tau_i} w_j + \frac{f_i}{\beta}$;
end for
$\tau_{seq} = \tau \setminus \cup_{i \in PQ} \{\tau_i\}$;
$C_p \leftarrow \emptyset$; $C_{temp} \leftarrow \emptyset$;
repeat
  $i \leftarrow popHead(PQ)$; $W \leftarrow MS(i)$;
  $C_{temp} \leftarrow$ **ImprovedSplit**(τ_i, n); (partition subtrees of parallel parts)
  Recompute $MS(i)$ with the new cut $C_p \cup C_{temp}$;
  if $MS(i) < W$ **then**
    $C_p \leftarrow C_p \cup C_{temp}$;
  end if
  Insert i into PQ (sorted by non-increasing makespan);
until $MS(i) \geq W$ or $head(PQ) = i$
$C_s \leftarrow$ **ImprovedSplit**(τ_{seq}, n); (partition the sequential part)
$C \leftarrow PQ \cup C_p \cup C_s$;
Merge$(\tau, C, p, MinMemory(\tau))$;
return C

executed in parallel. Four units of communication time could however be saved by executing all other nodes on the same processor, reaching a makespan of 45 and using only four processors.

To avoid this shortcoming, we therefore propose an algorithm called **ASAPnochain** (see Algorithm 4), used on top of **ASAP**. After calling **ASAP**, we first build the quotient tree in which, except the root, other nodes that have no siblings are elements of chains. Their input edges are then restored, i.e., subtrees are merged into a single subtree, which leaves some processors idle. These idle processors will be used, if possible, to improve the makespan, during the last step of the heuristics, see Section 5.3.

5.2 Step 2: Fitting into memory

After partitioning a tree into many subtrees by **SplitSubtrees**, **ImprovedSplit**, **ASAP**, or **ASAPnochain**, we propose three heuristics in this section

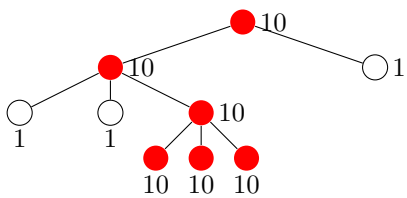

Figure 4: Example showing that a chain always wastes processors. Node labels represent their weight. All edges have weight 2, and $p = 6$. Red nodes denote subtrees' roots as determined by **ASAP**.

Algorithm 3 ASAP (τ, p)

```
PQ ← children of the root of τ, sorted by non-increasing W_i's;
s = 0; C_s ← ∅; (s is the step)
Let MS_s be the makespan of τ with partition C_s;
repeat
   if PQ is empty then
      break;
   end if
   i ← popHead(PQ);
   insert Children(i) into PQ (and keep it sorted);
   if i is not the only child of its parent then
      s ← s + 1;
      C_s ← C_{s−1} ∪ {i}; (the edge we just cut)
      Let MS_s be the makespan of τ with partition C_s;
   end if
until |C_s| = p − 1;
select step s* that minimizes MS_{s*};
return C_{s*};
```

Algorithm 4 ASAPnochain (τ, p)

```
C ← ASAP(τ, p);
Construct a quotient tree Q from τ and C;
for all node i on Q do
   if node i has only one child then
      remove input edge of i's child from C;
   end if
end for
return C
```

to check each subtree's minimum memory requirement and further partition those such that $MinMemory(\tau_i) > M$.

5.2.1 FirstFit heuristic

We first note the proximity of this problem with the MinIO problem [5]. 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 [5], 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 [5]. 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 each subtree can be processed without exceeding the available memory, but may lead to numerous subtrees.

5.2.2 LargestFirst heuristic

For our problem, we want to end up with a total of not more than p subtrees from the original tree (one subtree per processor), and since we may have already created p subtrees in Step 1 (Section 5.1), we do not want to create too many additional subtrees. Otherwise, subtrees will have to be merged in Step 3 (Section 5.3), possibly resulting in an increase of makespan. Therefore, 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**.

5.2.3 Immediately heuristic

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**.

5.3 Step 3: Reaching an acceptable number of subtrees

Now that we have first minimized the makespan, and then made sure that each subtree fits in local memory, we need to check how many subtrees have been generated. During this step, we either decrease the number of subtrees if it is greater than the number of processors p, or we increase it by further splitting subtrees if we have idle processors and the makespan may be improved.

5.3.1 Decreasing the number of subtrees

If there are more subtrees than processors, some of them have to be merged, and the resulted subtrees should also fit in local memory.

For subtrees that are leaves and have only one sibling, merging only theirselves to their parents will lead to a chain, which wastes processors. Thus, they are also merged with their siblings. In all combinations that fit in memory, we greedily merge subtrees that lead to the minimum increase in makespan. We compute the increase in makespan as follows. We denote the subtree to be

merged as node i of the quotient tree. Sometimes (when i is not on the critical path), $MS(i)$ can be increased without changing the final makespan $MS(r)$. We define d_i as the *slack* in $MS(i)$, that is, the threshold such that $MS(r)$ is not impacted by the increase of $MS(i)$ up to $MS(i) + d_i$. It can be recursively computed from the root: $d_i = d_t + MS(k) - MS(i)$, in which t is i's parent in the quotient tree and k is the sibling of i that has the maximum makespan. For the root, d_r is set to 0.

We then compute the increase of makespan of merging i to its parent t in the quotient tree as follows. We first estimate the increase Δ_t of $MS(t)$. If i is a leaf and has only one sibling, denoted j, the increase in makespan of their parent t is $\Delta_t = W_i + W_j - \max(\frac{f_i}{\beta} + w_i, \frac{f_j}{\beta} + w_j)$. For other subtrees, the makespan of t before the merge is $MS(t) = \frac{f_t}{\beta} + W_t + \max(MS(k), MS(i))$, and after merging i to t, $MS(t) = \frac{f_t}{\beta} + W_t + W_i + \max(MS(k), MS(j))$, where j is the child of i that has the maximum makespan. Therefore, the increase of $MS(t)$ is $\Delta_t = W_i + \max(MS(k), MS(j)) - \max(MS(k), MS(i))$. Finally, taking the slack into consideration, the increase of $MS(r)$ is $\Delta_t - d_i$.

This algorithm is formalized as Algorithm 5. There are initially at most n subtrees, and we decrease this number by one or two at each step, hence the algorithm runs in polynomial time.

Algorithm 5 Merge (τ, C, p, M)

```
Construct the quotient tree Q according to τ and C;
shortage ← p − |C| − 1; (amount of processors' shortage)
r ← root of τ;
while shortage > 0 do
  for all nodes i of Q except the root do
    if subtree i is a leaf and has only one sibling then
      Δ_i ← estimation of increase in MS(r) if subtree i and its sibling are
        merged with their parent;
      m_i ← subtree made of i, its sibling and their parent fits in memory
        size M;
    else
      Δ_i ← estimation of increase in MS(r) if merge subtree i with its
        parent;
      m_i ← subtree made of i and its parent fits in memory size M;
    end if
  end for
  set S ← {i s.t. m_i = true};
  j ← combination in S that has the minimum Δ_i;
  if subtree j is a leaf and has only one sibling then
    merge subtree j and its sibling with their parent; shortage = shortage−
      2;
  else
    merge subtree j with its parent;
    shortage = shortage − 1;
  end if
end while
```

5.3.2 Increasing the number of subtrees

If there are more processors than subtrees, we may be able to further reduce the makespan by splitting some of the subtrees. Given a tree τ and a partition C, **SplitAgain** 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. In the example of Figure 5, the critical path consists of three nodes of the quotient tree. Each subtree on the critical path is a candidate to be cut into two (or three) parts by cutting some edges. The set L (black nodes in Figure 5) contains the nodes whose input edge could be cut. If the subtree is a leaf in the quotient tree, we always split into three parts, otherwise we would create a chain and only increase the makespan. At each step, we greedily select the option (within nodes of L) that has the maximum potential decrease in makespan of τ. We compute the potential makespan decrease as follows: let i be the node whose input edge is considered to be cut. It currently lies in the subtree τ_t rooted at node t. After cutting the input edge of node i, it produces a new subtree τ_i of weight W_i.

The makespan of τ_t after cutting is given by:

$$\max(MS(t) - W_i, W_t + \frac{f_i}{\beta} + \max_{\tau_j \in Children(\tau_i)} MS(j)),$$

where $MS(j)$ is the makespan of a subtree rooted at j before cutting any new edge. Indeed, either the critical path does not include τ_i, or it now includes the communication to τ_i and the makespan of the largest children of τ_i in the new quotient tree. Note that if the child of τ_t that is in the critical path is also a child of τ_i (for instance, in Figure 5, when we try to cut the input edge of node a), the makespan will only be increased, and hence we will never cut edge i.

The decrease of the makespan of τ_t when cutting the input edge of node i is thus given by:

$$\Delta_i = \min(W_i, MS(t) - W_t - \frac{f_i}{\beta} - \max_{\tau_j \in Children(\tau_i)} MS(j)).$$

If we cut two edges in the last subtree on the critical path, say i and j, the makespan after cutting is

$$MS(t) - W_i - W_j + \max(\frac{f_i}{\beta} + W_i, \frac{f_j}{\beta} + W_j),$$

and the decrease of $MS(t)$ is:

$$\Delta_i = \min(W_j - \frac{f_i}{\beta}, W_i - \frac{f_j}{\beta}).$$

This process is repeated until there are no more idle processors or no further decrease in makespan. It is formalized in Algorithm 6.

Algorithm 6 SplitAgain(τ, C, p)

1: Compute the quotient tree Q and its critical path $CriPat$;
2: $n \leftarrow p - 1 - |C|$; (number of idle processors)
3: **while** $n \geq 1$ **do**
4: $L \leftarrow$ nodes of subtrees on $CriPat$;
5: Remove from L the roots of subtrees;
6: **for all** nodes i in L **do**
7: **if** i is in the last subtree on $CriPat$ and $n \geq 2$ **then**
8: Let j be the largest sibling of i, in terms of W;
9: $C_i \leftarrow$ input edges of nodes i and j;
10: **else**
11: $C_i \leftarrow$ input edge of node i;
12: **end if**
13: $\Delta_i \leftarrow$ makespan decrease when edges in C_i are cut;
14: **end for**
15: $k \leftarrow$ the node in L which leads to the largest Δ_k;
16: **if** $\Delta_k \geq 0$ **then**
17: $C \leftarrow C \cup C_k$; (cut edges in C_k)
18: $n \leftarrow n - |C_k|$;
19: Recompute Q and $CriPat$;
20: **else**
21: break;
22: **end if**
23: **end while**

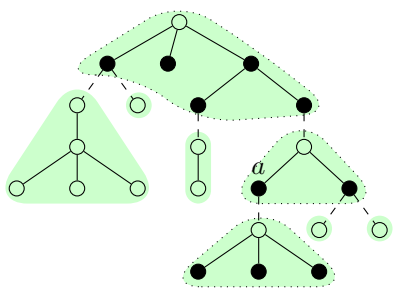

Figure 5: Example to illustrate **SplitAgain**: green areas surrounded with dotted line belong to the critical path; black nodes are candidates to be cut after line 5 (set L).

6 Experimental validation through simulations

In this section, we compare the performance of the proposed heuristics on a wide range of computing platform settings. We evaluate the results of the three stages: partition for reducing makespan, fitting in the memory constraint and constraint on the number of processors.

6.1 Dataset and simulation setup

The dataset contains assembly trees of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection (`http://www.cise.ufl.edu/research/sparse/matrices/`), which correspond to the computational requirements of sparse direct solvers on these matrices. For trees' construction details, we refer to [5]. To test heuristics proposed for fitting in local memory, we only keep trees whose *MinMem* is larger than its *MaxOutDeg*, others are discarded. Overall, there are 31 trees in the data set.

To compare the performance of the proposed heuristics in different environments, and since these trees exhibit very different number of nodes (from thousands to several millions), we have selected three different processor to node ratios (PNR): the number of processors p can be set to $1e-04$, 0.001, or 0.01 times the tree size n (while ensuring $p \geq 3$). 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 for communicating all data divided by the total computation time, is either 0.1, 1 or 10.

We consider two scenarios for the memory constraint: (i) in the *loose* scenario, the memory bound for each processor is set to *MinMemory*, hence there is no memory constraint; (ii) then, the *strict* scenario sets the memory bound to *MaxOutDeg*, the minimum memory needed to process any single task. The sequential tree traversal used in **FirstFit**, **LargestFirst** and **Immediately** is given by *MinMem* as described in [5], which has a minimum memory cost. All codes and trees can be found on github.com/gouchangjiang.

6.2 Step 1: Minimizing the makespan

The results of heuristics for reducing makespan on different computing scenarios are shown on Figure 6. We consider all combinations of CCRs and PNRs, and we normalize the makespan of **SplitSubtrees**, **ImprovedSplit**, **ASAP**, and **ASAPnochain** to the makespan obtained with a sequential execution of the tree, denoted by **Sequence**. Hence, a smaller ratio indicates a better relative performance. Note that there is no memory constraint in this step, hence **Sequence** returns a valid solution, using only one processor.

As expected, all heuristics achieve significant gain compared to the sequential execution, and the gain increases with the number of processors, achieving more than four times better makespan.

As expected, all heuristics are better than the reference sequential schedule **Sequence**: the makespan is reduced by at least 45% on more than 50% of the cases. With more processors, they behave even better than **Sequence**, four times better on more than 50% of the cases. Increasing the number of processors generally allows to reduce the makespan, except for **SplitSubtrees**.

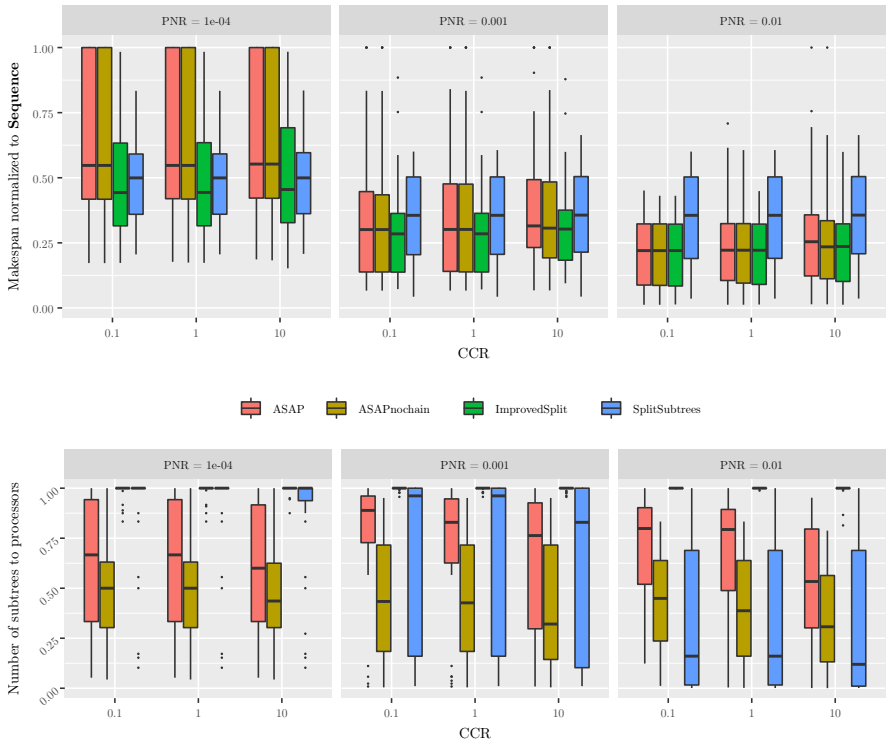

Figure 6: Makespan (top, normalized to **Sequence**) and number of generated subtrees (bottom) after Step 1 with different CCRs and PNRs.

All heuristics behave better than **SplitSubtrees** for all CCR values. Also, note that **ImprovedSplit** always surpasses **ASAPnochain** with few processors (PNR=$1e-04$ or 0.001).

Figure 6 presents the number of subtrees that are generated compared to the number of processors provided. Only **ImprovedSplit** takes fully advantage of processor resources in all cases. **SplitSubtrees** uses all processors only with few processors (PNR=$1e-04$) and not with more processors because it only splits in two levels. For instance, for PNR=0.01, **SplitSubtrees** uses 16% of the processors on more than 50% of the cases. **ASAPnochain** uses on average 22% less processors than **ASAP** by removing processors in chains. It uses much fewer processors than **ImprovedSplit**, using only half of the processors in around 50% of the cases.

Overall, the improvement of **ASAP** with **ASAPnochain** is always beneficial (it can only improve the makespan), even more with expensive communication costs (CCR=10). Therefore, we only consider **ASAPnochain** in the following (and not **ASAP**), since it results in lower makespan and produces less subtrees than **ASAP**. Indeed, we hope to be able to reuse the unused processors to further decrease the makespan in Step 3.

6.3 Step 2: Fitting into memory

At the end of Step 1, some subtrees may exceed the maximum available memory M when we consider the *strict* memory scenario. As expected, there are less subtrees not fitting into memory when there are many processors, since subtrees are smaller, and also when using **ImprovedSplit**, since it generates more subtrees, and hence smaller subtrees. The subtrees that do not fit into memory are further decomposed with either **FirstFit**, **LargestFirst** or **Immediately**, so that all subtrees fit in memory at the end of this step. We may then have more subtrees than processors, and Step 3 will later merge subtrees if needed.

In order to assess the performance of the heuristics from Step 2, we execute them in the *strict* memory scenario both on the original tree (**Sequence**, i.e., no heuristic from Step 1 is used) and after running the heuristics from Step 1. We report the average ratio of number of subtrees to processors NtoP, and the average percentage of gain on execution time

$$\mathrm{ET} = 100 \times \left(1 - \frac{MS^2_\infty}{MS^1}\right),$$

where MS^2_∞ is the makespan after Step 2 with an infinite number of processors (since splitting subtrees in Step 2 may generate more subtrees than available processors), and MS^1 is the makespan after Step 1. If ET is positive, it means that the new makespan is better, and it is feasible only if NtoP is smaller than or equal to 1.

Table 1 presents all results, for the three heuristics of Step 2 (**FirstFit**, **LargestFirst** and **Immediately**) and the four possibilities for Step 1, for CCR=1. Overall, **FirstFit** generates the smallest amount of subtrees, hence there is more chance that this heuristic will succeed to map all subtrees to processors while fitting in memory. **LargestFirst** has close results in terms of subtrees, and it is interesting to see that it can reduce the makespan even more than **FirstFit**. **Immediately** generates much more subtrees, and in some cases it is able to further decrease the makespan, but not always.

Results for other values of CCR are reported in Tables 2 and 3 of Appendix A.1. They lead to the same conclusions, even though there is less gain in terms of makespan (and sometimes even an increase in makespan) for CCR=10, since creating additional subtrees to fit into memory may generate expensive communications.

In the *loose* scenario, all subtrees fit in memory and hence Step 2 does not do anything, and in the following, we apply **LargestFirst** for Step 2 in the *strict* scenario. Indeed, **LargestFirst** obtains convincing results with a reasonable number of subtrees and interesting improvement in execution time. We refer readers interested in the results with **FirstFit** (resp. **Immediately**) to Appendix A.2 (resp. Appendix A.3).

6.4 Step 3: Reaching an acceptable number of subtrees

In this section, we examine the performance of **Merge** and **SplitAgain**, which are designed for reducing the number of subtrees so that we have enough processors, or for further optimizing the makespan if there are some remaining processors. As seen in Table 1, **ImprovedSplit** is the heuristic generating the most subtrees when combined with **LargestFirst**, and hence it requires to merge

	PNR	1e-04		0.001		0.01	
		NtoP	ET	NtoP	ET	NtoP	ET
FirstFit	ASAPnochain	1.34	7%	0.51	4%	0.40	0%
	ImprovedSplit	1.81	7%	1.09	2%	1.00	0%
	SplitSubtrees	1.66	8%	0.70	0%	0.33	0%
	Sequence	1.49	13%	0.20	13%	0.02	13%
LargestFirst	ASAPnochain	1.59	8%	0.51	4%	0.40	0%
	ImprovedSplit	2.07	9%	1.10	4%	1.00	0%
	SplitSubtrees	1.86	10%	0.71	0%	0.33	0%
	Sequence	2.17	13%	0.28	13%	0.03	13%
Immediately	ASAPnochain	5.97	9%	0.90	6%	0.41	2%
	ImprovedSplit	5.61	5%	1.38	1%	1.00	0%
	SplitSubtrees	5.13	9%	0.97	4%	0.33	0%
	Sequence	6.24	18%	0.90	18%	0.09	18%

Table 1: After Step 2, NtoP is the ratio of number of subtrees to processors, and ET is the gain on execution time. CCR=1.

some subtrees to obtain a feasible solution. The other heuristics leave many processors idle when there are many processors (PNR=0.001 or PNR=0.01), and we may be able to further improve the makespan by using **SplitAgain** in these cases.

Figure 7 shows the performance of **SplitAgain** or **Merge**. We compare the makespan after Step 3 to the execution time that was achieved at the end of Step 2 (with an infinite number of processors), using **LargestFirst** during Step 2 (*strict* memory scenario), with CCR=0.1. Each tile in the figure represents a testing case, and F represents a failure, i.e., we were not able to obtain a solution with less subtrees than processors. As expected, the less processors, the more failures we have. Since **Sequence** did nothing in Step 1, it starts from a sequential execution of the tree and hence it obtains important gains in makespan after using **SplitAgain** in Step 3. **SplitAgain** also allows us to improve the makespan with **ASAPnochain** and **SplitSubtrees**. As noted before, **ImprovedSplit** usually generates more subtrees than processors after Step 2, and hence we must use **Merge** to obtain a feasible solution, as well as for other heuristics when there are few processors (PNR=1e-04). We observe some failures in these cases, in particular when using **ImprovedSplit**, while **ASAPnochain** and **Sequence** succeed in most cases. Overall, the failure rate after Step 3 is 7.26%.

Still in the *strict* memory scenario, we finally compare the makespan of all our heuristics to **FirstFit**, since it is a simple adaptation from [5]. Indeed, **FirstFit** is likely to give a feasible solution in most cases, since it consumes least processors, as shown in Table 1. Furthermore, we consider the heuristic **Select**, which runs all possible heuristics at Step 1 (followed by **LargestFirst**, and then **SplitAgain** or **Merge**), and keeps the best solution for each input tree. This allows us to analyze whether there is at least one heuristic that outperforms others in all situations. Figure 8 presents the final makespan obtained after all three steps, excluding cases on which no solution was found. Recall that failure rates can be found in Figure 7. Overall, **ImprovedSplit**

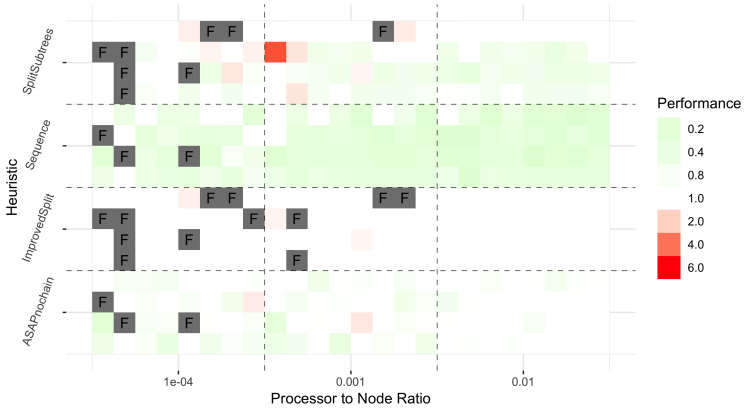

Figure 7: Increase or decrease of makespan after Step 3 (using **LargestFirst** at Step 2). F represents a failure. CCR=0.1.

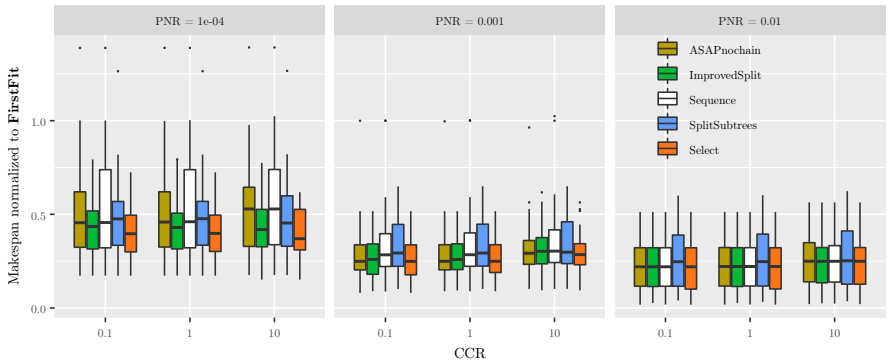

Figure 8: Final makespan (using Step 1 heuristic followed by **LargestFirst** and **SplitAgain**/**Merge**) normalized to **FirstFit**.

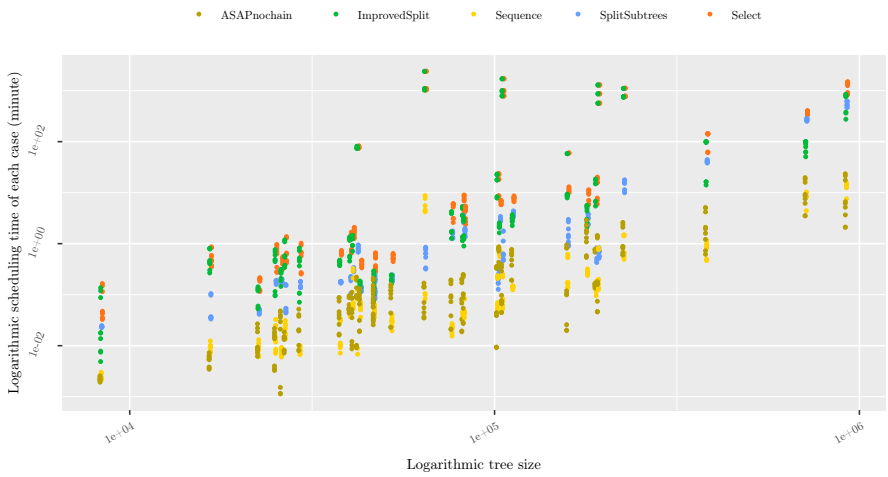

Figure 9: Logarithmic scheduling time in minutes of different allocation policies, all followed by **LargestFirst** and **SplitAgain** or **Merge**.

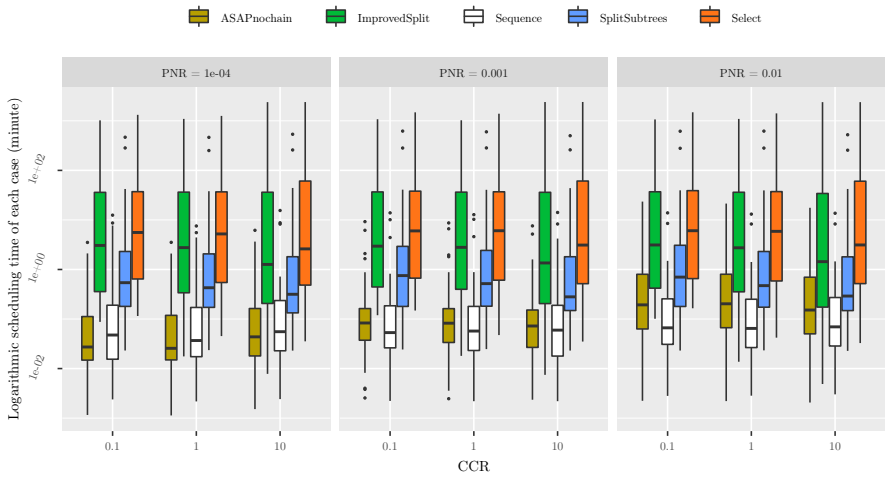

Figure 10: Logarithmic scheduling time in minutes of different allocation policies, all followed by **LargestFirst** and **SplitAgain** or **Merge**.

is the best heuristic when there are a few processors (PNR=1e-04), but other heuristics outperform it in several cases, since **Select** is even better achieving a makespan 2.5 times faster than the reference **FirstFit**. With more processors, **ASAPnochain** is slightly better, in particular for PNR=0.001. Skipping Step 1 (**Sequence**) gives reasonable results as soon as there are many processors (PNR=0.01, or even PNR=0.001), which shows that the heuristics from Step 3 (**SplitAgain** and **Merge**) are very efficient in these cases (in particular **SplitAgain**). With PNR=0.01, all heuristics achieve a makespan four times smaller than the reference, in average. Finally, note that we may get a makespan worse than the reference on some cases, in particular with **Sequence** and **SplitSubtrees** (1.39 or 1.26 times worse than the makespan from **FirstFit**), but these outlier cases are avoided by using **Select**.

Of course, **Select** is always the best pick, but it may come at the price of a higher scheduling time, since it implies to run all four variants. We report the execution times in Figure 9. To ease the reading, we plot the logarithm of the scheduling time, in minutes, against the logarithm of the number of nodes in the tree. **ASAPnochain** and **Sequence** are the fastest heuristics, and it is interesting to note that **ASAPnochain** can sometimes be even faster than **Sequence**, even though **Sequence** does not do anything in Step 1: the tree obtained at the end of Step 1 with **ASAPnochain** has then a faster scheduling time for Steps 2 and 3 than starting from the original tree. As expected, **ImprovedSplit** takes more time than **SplitSubtrees**, since it refines the solution from **SplitSubtrees** to cut in several levels. It has to be noted that very long scheduling times (above 10 minutes) only happen for very large trees (above 100.000 nodes), except for a few extreme cases. Overall, **Select** is only slightly longer than **ImprovedSplit**, since the scheduling times of all other heuristics are small in comparison to the one of **ImprovedSplit**. Finally, note that different processor to node ratios (PNR) only slightly impact the scheduling time (see Figure 10 for detailed results).

To summarize, we recommend using **Select**, unless the scheduling time is very important or the tree is very large, in which cases **ASAPnochain** is a good option for Step 1 (efficient makespan obtained with a fast execution of the heuristic).

Finally, we present results in the *loose* memory scenario, where the memory bound for each processor is set to *MinMemory*, hence there is no memory constraint. In this case, we only consider the use of Step 1 directly followed by Step 3. The reference heuristic becomes **SplitSubtrees**, which was directly adapted from ideas from [33], resulting in a two-level split of the tree.

Figure 11 reports the final makespan, after applying one heuristic of Step 1 followed by **SplitAgain**. Indeed, at the end of Step 1, there are always less subtrees than processors. With few processors (PNR=1e-04), there is little room for improvement over the two-level partitioning of the tree. However, when the number of processors increase, **SplitAgain** achieves good results in using available processors to reduce the makespan, even without going through a heuristic from Step 1 (**Sequence** variant). Using only **SplitAgain** on the original tree is a good option in this case.

Figure 12 shows the performance of **SplitAgain**, after heuristics of Step 1, no heuristics of Step 2 is used since the local memory is set as *MinMemory*. Obviously, there is no failure case. Indeed, heuristics of Step 1 guarantee that

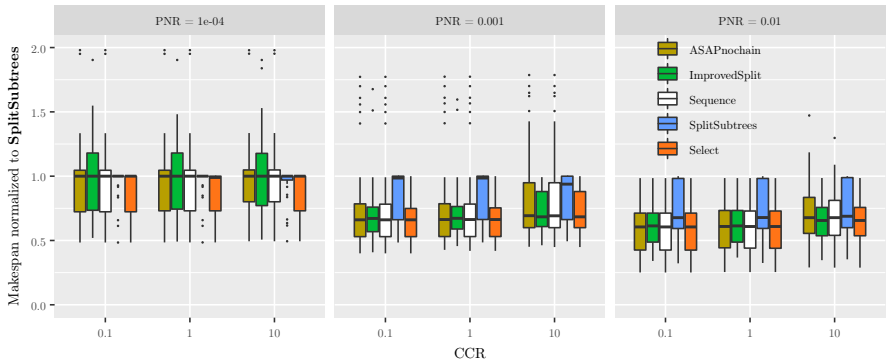

Figure 11: Final makespan of each Step 1 heuristic followed by **SplitAgain**, normalized to **SplitSubtrees** in the *loose* memory scenario.

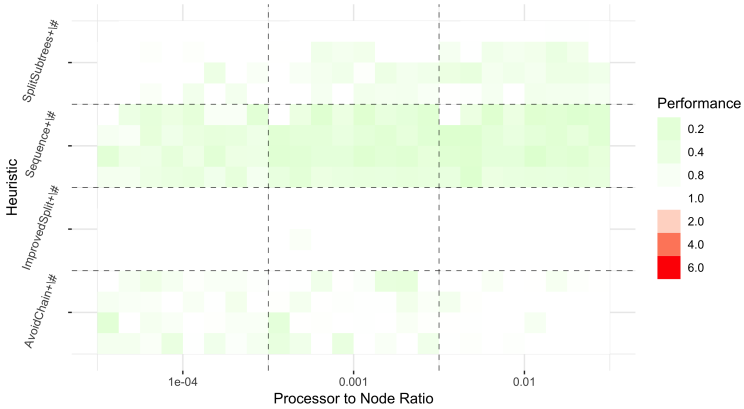

Figure 12: Increase or decrease of makespan after Step 3, in the *loose* memory scenario. CCR= 0.1.

no more subtrees are generated than processors. Makespan is decreased by **SplitAgain** on most cases except **ImprovedSplit**, as it always takes fully advantage of processors, leaves no idle processors for improvement.

Finally, Figure 13 reports the execution times of a heuristic of Step 1 followed by **SplitAgain**. **Sequence** is the fastest one, then closely followed by **ASAPnochain**. **ImprovedSplit** still takes far more time than **SplitSubtrees**. **Select** is only slightly longer than **ImprovedSplit**.

7 Conclusion

We have studied a 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

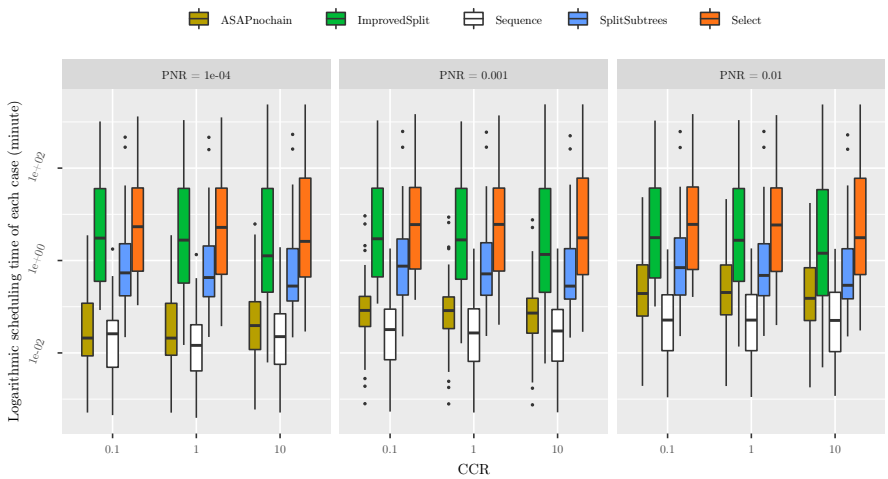

Figure 13: Logarithmic scheduling time in minutes of different allocation policies, all followed by **SplitAgain**, in the *loose* memory scenario.

above, MinMakespan, is NP-complete, and we have designed several heuristics to tackle it. We propose a three-step approach: (i) minimize the makespan; (ii) fit into memory if needed; and (iii) make sure that we have less subtrees than processors, and use as many processors as required to further minimize the makespan.

Extensive simulations demonstrate the efficiency of these heuristics and provide guidelines about the heuristics that should be used. Without memory constraint, the heuristic from Step 3, **SplitAgain**, is efficiently splitting the tree to minimize the makespan, and achieves results 1.5 times better than the reference heuristic, **SplitSubtrees**, when there are many processors available (processor to node ratio PNR$\geq$ 0.001). When there are memory constraints, one must make sure that each subtree fits into memory, and the reference heuristic is **FirstFit**, which partitions the tree for memory. In this case, using the best combination of a heuristic of Step 1, a heuristic to fit into memory, and finally **SplitSubtrees** or **Merge**, allows us to drastically improve the makespan (two to four times better, depending on the processor to node ratio). The use of **ASAPnochain** in Step 1 may be selected for a smaller scheduling time, since **ImprovedSplit** may lead to a smaller makespan, but at the price of a longer scheduling time.

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

[2] 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.

[3] P. Sao, X. S. Li, and R. Vuduc, "A communication-avoiding 3d lu factorization algorithm for sparse matrices," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 908–919.

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

[5] 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.* Anchorage, Alaska, USA: IEEE Computer Society, 2011, pp. 556–567.

[6] E. Chan, F. G. V. Zee, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Satisfying your dependencies with supermatrix," in *2007 IEEE International Conference on Cluster Computing*, Sept 2007, pp. 91–99.

[7] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *CoRR*, vol. abs/0709.1272, 2007. [Online]. Available: http://arxiv.org/abs/0709.1272

[8] K. Kim and V. Eijkhout, "A parallel sparse direct solver via hierarchical dag scheduling," *ACM Trans. Math. Softw.*, vol. 41, no. 1, pp. 3:1–3:27, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629641

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

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

[11] 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 *CC-Grid'07*, 2007, pp. 401–409.

[12] 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.

[13] K. Kim, H. C. Edwards, and S. Rajamanickam, "Tacho: Memory-scalable task parallel sparse cholesky factorization," in *IPDPS Workshops.* IEEE Computer Society, 2018, pp. 550–559.

[14] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[15] E. Agullo, G. Bosilca, A. Buttari, A. Guermouche, and F. Lopez, "Exploiting a parametrized task graph model for the parallelization of a sparse direct multifrontal solver," in *Euro-Par 2016: Parallel Processing Workshops*. Springer International Publishing, 2017, pp. 175–186.

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

[17] W. Donath and A. Hoffman, "Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices," *IBM Technical Disclosure Bulletin*, vol. 15, no. 3, pp. 938–944, 1972.

[18] W. E. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," *IBM Journal of Research and Development*, vol. 17, no. 5, pp. 420–425, Sept 1973.

[19] H. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering*, vol. 2, no. 2, pp. 135 – 148, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/095605219190014V

[20] 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*, ser. SFCS '91. Washington, DC, USA: IEEE Computer Society, 1991, pp. 538–547. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1991.185417

[21] 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*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 1222–1230. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339722

[22] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970. [Online]. Available: http://dx.doi.org/10.1002/j.1538-7305.1970.tb01770.x

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

[24] L. A. Sanchis, "Multiple-way network partitioning," *IEEE Trans. Comput.*, vol. 38, no. 1, pp. 62–81, Jan. 1989. [Online]. Available: http://dx.doi.org/10.1109/12.8730

[25] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw, "Shape-optimized mesh partitioning and load balancing for parallel adaptive {FEM}," *Parallel Computing*, vol. 26, no. 12, pp. 1555 – 1581, 2000, graph Partitioning and Parallel Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819100000430

[26] S. Schamberger, "On partitioning fem graphs using diffusion," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pp. 277–.

[27] L. Lovász, "Random walks on graphs," *Combinatorics, Paul erdos is eighty*, vol. 2, pp. 1–46, 1993.

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

[29] A. E. Feldmann, "Fast balanced partitioning of grid graphs is hard," *CoRR*, vol. abs/1111.6745, 2011.

[30] K. Andreev and H. Räcke, "Balanced graph partitioning," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04. New York, NY, USA: ACM, 2004, pp. 120–124. [Online]. Available: http://doi.acm.org/10.1145/1007912.1007931

[31] A. E. Feldmann and L. Foschini, "Balanced partitions of trees and applications," *Algorithmica*, vol. 71, no. 2, pp. 354–376, Feb 2015. [Online]. Available: https://doi.org/10.1007/s00453-013-9802-3

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

[33] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, "Parallel scheduling of task trees with limited memory," *ACM Transactions on Parallel Computing*, vol. 2, no. 2, p. 13, 2015.

A Appendix: Additional simulation results

This section presents additional simulation results, with more CCR options in Steps 1 and 2 (Appendix A.1), and by using **FirstFit** (Appendix A.2) or **Immediately** (Appendix A.3) at Step 2 instead of **LargestFirst**.

A.1 More CCR options in Steps 1 and 2

Tables 2 and 3 are counterparts of Table 1. They present the relative performance of all combinations of heuristics from Step 1 and Step 2 with CCR= 0.1 and CCR=10 respectively.

	PNR	1e-04		0.001		0.01	
		NtoP	ET	NtoP	ET	NtoP	ET
FirstFit	ASAPnochain	1.34	7%	0.52	4%	0.42	0%
	ImprovedSplit	1.81	7%	1.10	2%	1.00	0%
	SplitSubtrees	1.66	8%	0.70	0%	0.33	0%
	Sequence	1.49	13%	0.20	13%	0.02	13%
LargestF.	ASAPnochain	1.59	8%	0.52	4%	0.42	0%
	ImprovedSplit	2.03	8%	1.10	0%	1.00	0%
	SplitSubtrees	1.86	10%	0.71	0%	0.33	0%
	Sequence	2.17	13%	0.28	13%	0.03	13%
Imm.	ASAPnochain	5.97	9%	0.91	6%	0.43	2%
	ImprovedSplit	5.55	5%	1.39	0%	1.00	0%
	SplitSubtrees	5.13	9%	0.97	4%	0.33	0%
	Sequence	6.24	18%	0.90	18%	0.09	18%

Table 2: After Step 2, NtoP is the ratio of number of subtrees to processors, and ET is the gain on execution time. CCR=0.1.

	PNR	1e-04		0.001		0.01	
		NtoP	ET	NtoP	ET	NtoP	ET
FirstFit	ASAPnochain	1.34	6%	0.48	3%	0.34	-1%
	ImprovedSplit	1.75	10%	1.09	4%	0.99	0%
	SplitSubtrees	1.64	7%	0.67	-1%	0.33	-1%
	Sequence	1.49	12%	0.20	12%	0.02	12%
LargestF.	ASAPnochain	1.97	7%	0.48	3%	0.34	-1%
	ImprovedSplit	2.02	11%	1.10	7%	0.99	0%
	SplitSubtrees	1.84	10%	0.68	-1%	0.33	-1%
	Sequence	2.17	12%	0.28	12%	0.03	12%
Imm.	ASAPnochain	5.97	8%	0.87	5%	0.35	2%
	ImprovedSplit	6.08	4%	1.38	1%	0.99	0%
	SplitSubtrees	5.11	8%	0.94	3%	0.33	0%
	Sequence	6.24	17%	0.90	17%	0.09	17%

Table 3: After Step 2, NtoP is the ratio of number of subtrees to processors, and ET is the gain on execution time. CCR=10.

FirstFit generates the smallest amount of subtrees. **LargestFirst** behaves

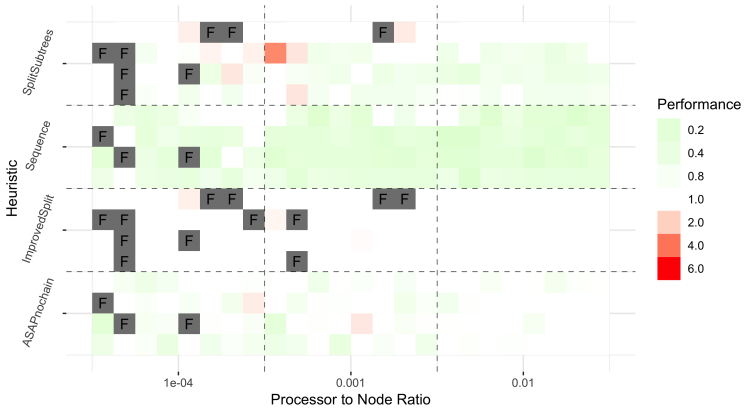

Figure 14: Increase or decrease of makespan after Step 3 (using **FirstFit** at Step 2). F represents failures. CCR=0.1.

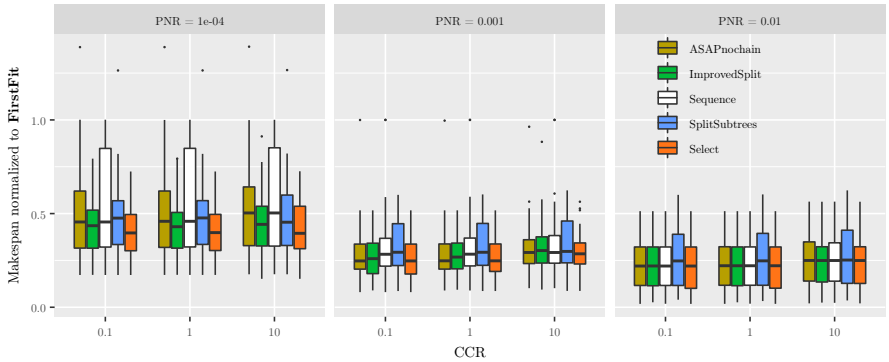

Figure 15: Final makespan (using Step 1-heuristic followed by **FirstFit** and **SplitAgain**/**Merge**) normalized to **FirstFit**.

close to **FirstFit**, it consumes a little more processors with the potential of reducing the makespan. **Immediately** produces far more subtrees than other two heuristics. The execution time of **FirstFit** and **LargestFirst** could be worse than makespan of Step 1 when communication is expensive (i.e. CCR=10). In conclusion, **FirstFit** is still the best choice when processors are limited (PNR=$1e-04$), **LargestFirst** and **Immediately** are also good options when many processors (PNR$\geq$ 0.001) are available.

A.2 Results with FirstFit at Step 2

Figure 14 shows the performance of **SplitAgain** or **Merge**, using **FirstFit** at Step 2 (*strict* memory scenario). The same as in Figure 7, less processors, more failures. **ImprovedSplit** is more likely to fail, while **Sequence** works well with **SplitAgain**. Compared to using **LargestFirst** at Step 2, using **FirstFit** has less cases who has an increase in makespan (i.e., less red rectangles).

As Figure 15 shows, using **FirstFit** at Step 2 gets almost the same result than using **LargestFirst**. The biggest difference is that the makespan of **Se-**

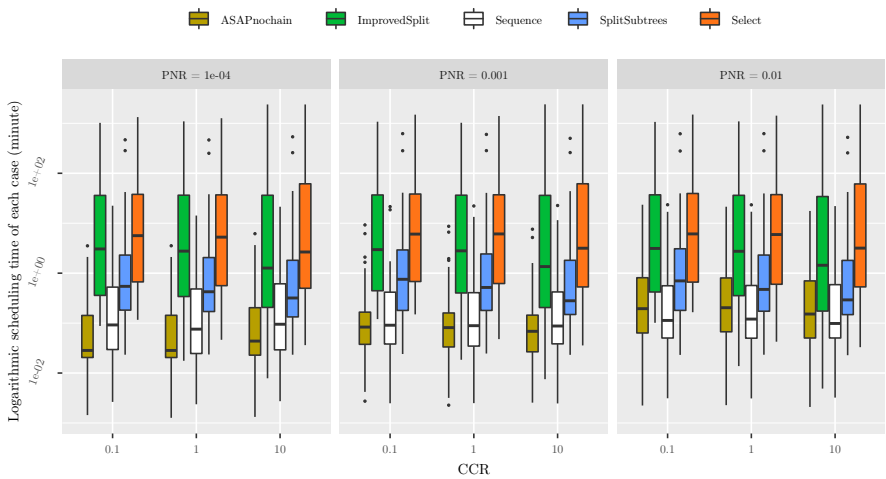

Figure 16: Logarithmic scheduling time in minutes of different allocation policies, all followed by **FirstFit** and **SplitAgain** or **Merge**.

quence (followed by **FirstFit** and **SplitAgain**/**Merge**) gets worse, when there are a few processors (PNR=$1e-04$).

Figure 16 reports the execution time of all heuristics (a heuristic from Step 1, followed by **FirstFit** + **SplitAgain**/**Merge**). As before using **LargestFirst** at Step 2, **ASAPnochain** is the fastest one, then followed by **Sequence** when there are a few processors (PNR= $1e-04$), otherwise **Sequence** is the best one. **ImprovedSplit** still costs far more time than others, which can be found between the little difference between itself with **Select**.

A.3 Results with Immediately at Step 2

Using **Immediately** at Step 2, as shown in Figure 17, there are more failures than using **LargestFirst** or **FirstFit**. Even using **Sequence** at Step 1 and PNR= 0.001, it may fail. Overall, the failure rate is 14.52%, two times larger than when using **LargestFirst**.

Final makespan results are depicted in Figure 18. Results are slightly different than with **LargestFirst**, and overall, **ASAPnochain** becomes the best solution, then followed by **ImprovedSplit** and **Sequence**. When there are a few processors (PNR= $1e-04$), no heuristic always win over the others, since the median of **Select** is around 0.08 lower than the others. Excluding failure cases, the final makespan using **Immediately** at Step 2 is slightly better than when using **LargestFirst**. This is more obvious when there are few processors (PNR= $1e-04$): compared to using **LargestFirst**, **ASAPnochain** decreases 8% in average when using **Immediately**.

Figure 19 shows the execution times in minutes. **ASAPnochain** is the fastest one when there are a few processors (PNR= $1e-04$). With more processors provided, **Sequence** becomes relatively faster than before, it is the fastest one when PNR= 0.01. **SplitSubtrees** costs more time than them. **ImprovedSplit** is far costly than others.

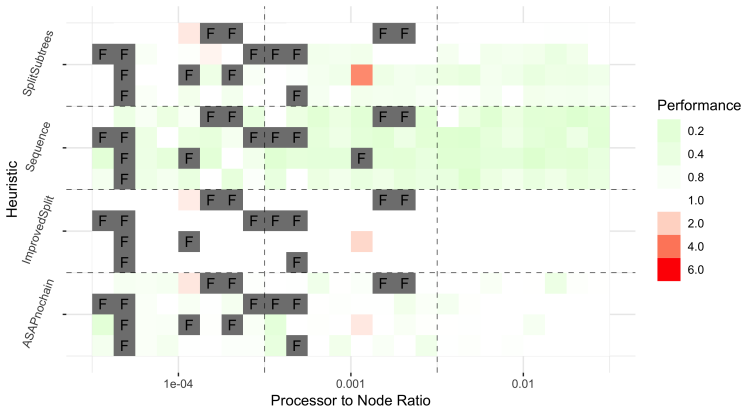

Figure 17: Increase or decrease of makespan after Step 3 (using **Immediately** at Step 2). F represents failures. $CCR = 0.1$.

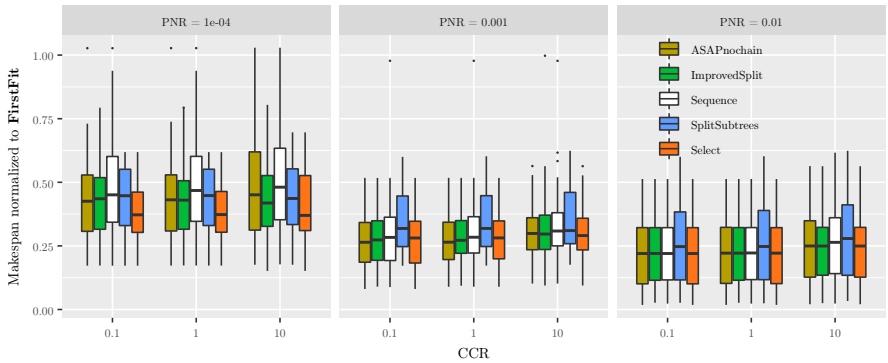

Figure 18: Final makespan (using Step 1-heuristic followed by **Immediately** and **SplitAgain**/**Merge**) normalized to **FirstFit**.

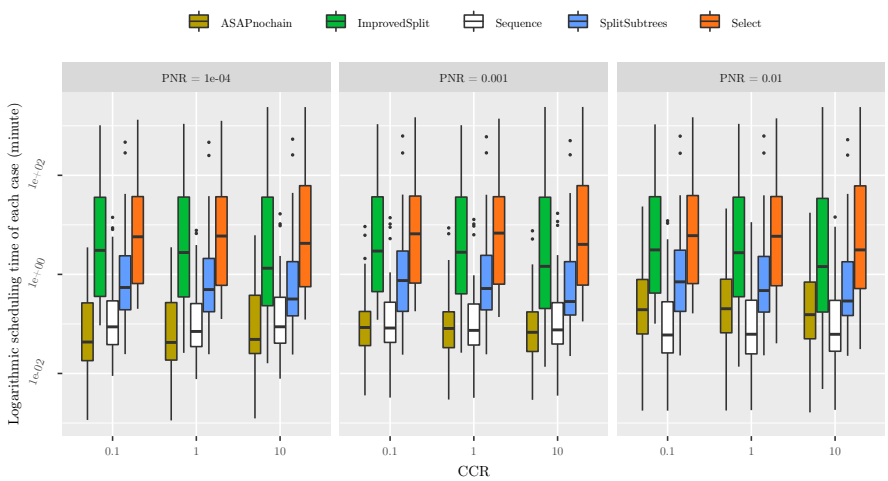

Figure 19: Logarithmic scheduling time in minutes of different allocation policies, all followed by **Immediately** and **SplitAgain** or **Merge**.

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