Mapping Tree-shaped Workflows on Memory-heterogeneous Architectures

Svetlana Kulagina, Henning Meyerhenke, Anne Benoit

Mapping Tree-shaped Workflows on Memory-heterogeneous Architectures

Svetlana Kulagina, Henning Meyerhenke, Anne Benoit

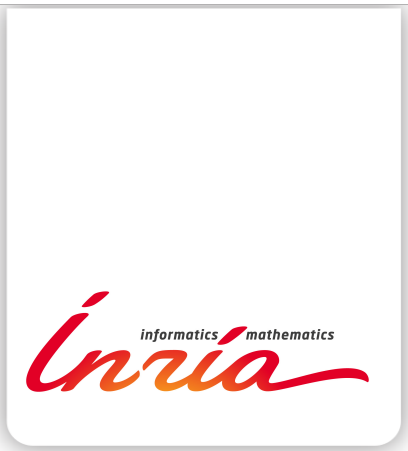
RESEARCH REPORT

N° 9458

February 2022

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9458--FR+ENG



Mapping Tree-shaped Workflows on Memory-heterogeneous Architectures

Svetlana Kulagina*, Henning Meyerhenke*, Anne Benoit†

Project-Team ROMA

Research Report n° 9458 — February 2022 — 20 pages

Abstract: Directed acyclic graphs are commonly used to model scientific workflows, by expressing dependencies between tasks, as well as the resource requirements of the workflow. As a special case, rooted directed trees occur in several applications, for instance in sparse matrix computations. Since typical workflows are modeled by huge trees, it is crucial to schedule them efficiently, so that their execution time (or makespan) is minimized. Furthermore, it might be beneficial to distribute the execution on several compute nodes, hence increasing the available memory, and allowing us to parallelize parts of the execution. To exploit the heterogeneity of modern clusters in this context, we investigate the partitioning and mapping of tree-shaped workflows on target architectures where each processor can have a different memory size. Our three-step heuristic adapts and extends previous work for homogeneous clusters [Gou et al., TPDS 2020]. The changes we propose concern the assignment to processors (which considers the different memory sizes) and the availability of *suitable* processors when splitting or merging subtrees. We evaluate our approach with extensive simulations and demonstrate that exploiting the heterogeneity in the cluster reduces the makespan significantly compared to the state of the art for homogeneous memory.

Key-words: Tree partitioning, Workflows, Mapping, Memory constraint

* Department of Computer Science, Humboldt-Universität zu Berlin, Germany
† Roma project-team, LIP laboratory, ENS Lyon, France

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

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

Placement de workflows de type arbre sur des architectures à mémoire hétérogène

Résumé : Les graphes acycliques dirigés sont couramment utilisés pour modéliser les applications de type *workflow*, en exprimant les dépendances entre les tâches, ainsi que les besoins en ressources. En cas particulier, les arbres dirigés enracinés apparaissent dans plusieurs applications, par exemple dans les calculs de matrices creuses. Comme les workflows typiques sont modélisés par de grands arbres, il est crucial de les ordonnancer efficacement, afin de minimiser leur temps d'exécution (ou makespan). En outre, il peut être avantageux de répartir l'exécution sur plusieurs nœuds de calcul, ce qui augmente la mémoire disponible et nous permet de paralléliser certaines parties de l'exécution. Afin d'exploiter l'hétérogénéité des clusters modernes dans ce contexte, nous étudions le partitionnement et le placement d'arbres sur des plates-formes où chaque processeur peut avoir une taille de mémoire différente. Notre heuristique en trois étapes adapte et étend les travaux précédents pour les clusters homogènes [Gou et al., TPDS 2020]. Les changements que nous proposons concernent l'affectation aux processeurs (qui prend en compte les différentes tailles de mémoire) et la disponibilité des processeurs convenables lors de la division ou de la fusion de sous-arbres. Nous évaluons notre approche à l'aide de simulations et démontrons que l'exploitation de l'hétérogénéité du cluster réduit considérablement le makespan par rapport à l'état de l'art pour la mémoire homogène.

Mots-clés : Partitionnement d'arbres, Workflows, Placement, Contrainte mémoire

1 Introduction

In many scientific disciplines, singular tasks revolving around the computation of one particular problem have made way to more complicated workflows that consist of many individual tasks. Such workflows are often represented as directed acyclic graphs (DAGs), with nodes of the graph representing the tasks and the edges their dependencies. One common form of such DAGs is a rooted directed tree, which we consider in this paper. These tree-shaped workflows occur in a variety of applications, for example in sparse matrix factorizations and computational physics [16, 10].

Running such workflows efficiently in parallel, *e. g.* on a compute cluster where processors have their own local memory and communicate via the network, requires a good scheduling strategy. Such a strategy would distribute singular tasks or whole subtrees to computing nodes in a way that fulfills a goal. Our focus regarding schedule quality is on the total execution time, expressed by the *makespan* of the schedule. To this end, we assume the workflow and its properties to be known before scheduling. Previous work [10] for completely homogeneous clusters (or other homogeneous platforms) showed the corresponding scheduling problem to be NP-complete and proposed several variants of a successful three-step heuristic. In summary, these steps are (i) to partition the tree into subtrees, minimizing the makespan and not taking the memory limit into account, (ii) to further partition subtrees too big for the memory limit, and finally (iii) to ensure that the number of subtrees is less than or equal to the number of processors.

Yet, more and more compute clusters are heterogeneous, *e. g.* due to hardware updates, a combination of clusters, or an intentional configuration where some compute nodes have special capabilities. It is for instance quite common in large clusters to have a few "fat" nodes with a particularly large memory.[1] Thus, to adapt the scheduling algorithm to variable memory constraints is very relevant. Yet, maybe with the exception of He *et al.* [11], there are no scheduling algorithms in the literature tailored to the problem of scheduling tree-shaped workflows on memory-heterogeneous architectures. And while He *et al.* [11] design their algorithm with heterogeneity in mind, their experimental setup and results do not consider memory-heterogeneous architectures, which are our focus.

In this paper, we present a partitioning and mapping heuristic (called HetPart – short for *heterogeneous tree partitioning*) for tree-shaped workflows that exploits memory heterogeneity. To this end, after discussing related work (Section 2), we formalize the scheduling problem in Section 3. The algorithmic contribution, described in Section 4, consists of a three-step heuristic that builds upon the work by Gou *et al.* [10] for the homogeneous case. We adapt two of these steps: (i) the assignment of tasks to processors, which now considers the different memory sizes, and (ii) when splitting or merging subtrees, we take the availability of suitable processors into account, i.e., those with sufficient mem-

[1] One of many possible examples is the JUWELS cluster of Forschungszentrum Jülich, `https://apps.fz-juelich.de/jsc/hps/juwels/configuration.html`.

ory. To evaluate the benefit from exploiting heterogeneity with HetPart, we employ simulations on real-world as well as randomly generated trees in Section 5. As standard of reference, we choose the state-of-the-art algorithm by Gou *et al.* [10]. For a fair comparison, we use the same heterogeneous target architecture and let the reference algorithm work on it in different scenarios regarding hardware consumption. Our experimental results on a cluster with four different memory sizes show that HetPart reduces the makespan on average by 15.5% and 25.0%, respectively, compared to the two best homogeneous scenarios. Where the improvement by HetPart is only 15.5%, the corresponding homogeneous scenario has the major drawback of not producing a valid solution for more than 20% of the instances.

2 Related Work

Scheduling and mapping collections of tasks on various types of computing platforms has been a focus of research interest since the 1990s. Many different kinds of applications have been considered over time, ranging from independent tasks to graphs of tasks, where tasks may have dependence constraints. Earlier works schedule various forms of workflows, such as pipeline workflows [5] and bags of tasks [4]. However, current consensus seems to be that a workflow is best described with a directed acyclic graph (DAG) [1, 15], which is the most general representation of dependence constraints. Rooted task trees are a common special case of DAGs, where each task (except the root) has a single parent node. Such trees arise in particular from sparse linear algebra applications [7, 16].

The goal is usually to be able to execute the whole application as fast as possible, hence minimizing the *makespan*, or total execution time. Several other objective functions have been studied, as for instance minimizing the throughput or latency of pipelined applications [5], focusing on fault tolerance [3], and also energy efficiency [2]. Recently, an important focus is put on memory optimization, since memory and I/O become a bottleneck [13, 8]. Some of these optimization goals may be antagonistic, and one may want to consider several of them simultaneously. This can be done either by finding Pareto-optimal solutions aiming at optimizing all objectives, or by fixing constraints on some objectives and optimizing only one. This latter approach is particularly suitable when objectives are of different nature, as in [5].

In the current work, the main optimization objective is to minimize the makespan. As each processor has a limited amount of memory, one must ensure that a constraint on memory is not violated, by carefully mapping parts of the applications on each processor such that a processor can handle its part within its own memory limit. Hence, one must partition the tree, map each subtree on its own processor, and then schedule the subtrees without exceeding the processor's memory. Given a tree, an exact scheduling algorithm with minimum memory requirement was designed [13]. An algorithm was also designed to minimize the I/O volume when parts of data need to be evicted from memory (MinIO problem). We choose not to evict data from memory in our case, but

rather we aim at using several processors to process the application. The focus of our work is hence on the partitioning of the tree, and mapping of subtrees onto processors. We then reuse, for each subtree, the optimal scheduling algorithm that minimizes the memory requirement.

The partitioning of various forms of graphs has been reviewed [6], and in particular, the partitioning of DAGs is difficult [12]. However, for the case when the strict condition of balanced weights of parts of the graph is relaxed, approaches to its partitioning were proposed [9].

Note that the problem of makespan minimization of a tree of tasks, by partitioning the tree so that each part fits (memory-wise) onto a processor, has already been tackled in the case of homogeneous processors [10]. As pointed out in Section 1, recent work by He *et al.* has attempted to extend this approach to heterogeneous architectures [11]. Their work leaves several important questions open, though: (i) the experiments seem to be on a system with homogeneous memories only, and (ii) the descriptions regarding the subroutine FitMemory are not sufficient for a reimplementation. Our work differs from theirs in several respects. As an example, one of our main contributions is a new merging procedure accounting for heterogeneous memories, while He *et al.* use the homogeneous merge from [10].

3 Model

3.1 Application model

We consider workflows that come in the form of rooted trees $\tau = (V, E)$, as motivated in the introduction (see also [10]). The tree vertices, numbered from 1 to n, correspond to the tasks, where each task is the smallest non-changeable workflow entity. Hence, each task $v_i \in V$ $(1 \leq i \leq n)$ requires w_i operations to be performed. Vertex $v_r \in V$ $(1 \leq r \leq n)$ is the root of the tree.

The edges, in turn, model precedence constraints between tasks. We assume all precedence constraints to be oriented towards the leaves, which is no limitation [10]. A precedence $v_j \rightarrow v_i$ (hence, $(v_j, v_i) \in E$) means that the task v_i cannot start before receiving an input file (or, more generally, input data) from its parent task v_j. The size of the (single) input file received by v_i is denoted as f_i (for the root, $f_r = 0$). The task also requires some memory to be executed; its size is denoted by m_i for task v_i.

Given a tree workflow, $D_{\max}$ is the maximum memory requirement of a node in this tree: $D_{\max} = \max_{v_i \in V} \left\{ f_i + m_i + \sum_{j:(v_i,v_j)\in E} f_j \right\}$. For each node, the memory requirement includes the size of all files to be sent to its children.

3.2 Platform model

The target computing environment is a cluster consisting of a finite number l of processing units, called processors, and denoted by $p_1, \ldots, p_l$. Each pair of

processors can communicate with each other via some network, and communication operations can happen in parallel. We assume that the system-specific bandwidth is always available for transferring input files to the responsible processor. All data generated during the execution of a task on processor p_u are stored on p_u, $1 \leq u \leq l$. Tasks are non-preemptive and atomic: a processor executes a single task at a time.

For $1 \leq u \leq l$, let M_u be the size of the main memory of processor p_u. Task v_i can be processed by p_u only if all the data required to execute the task fits into the processor's memory, i.e., $M_u \geq f_i + m_i + \sum_{j:(v_i,v_j)\in E} f_j + m_i$. While processors may have memories of different sizes, we consider a platform with processors computing at an identical speed s (number of operations per seconds), hence any processor can execute task v_i ($1 \leq i \leq n$) within time $\frac{w_i}{s}$.

For $(v_i, v_j) \in E$, if task v_i is mapped on processor p_u and task v_j is mapped on processor p_v, the input file for v_j is sent through the communication network, which has a bandwidth β. Hence, the time to send the file from v_i to v_j is $\frac{f_j}{\beta}$.

3.3 Constraints and scheduling objectives

In order to benefit from the parallel platform, the idea is to partition the tree τ into subtrees, and then map each subtree onto its own processor. Each subtree τ_ℓ is identified by its root $root(\tau_\ell) = v_i$, with $1 \leq i \leq n$. We denote by $tasks(i)$ the set of tasks included in subtree τ_ℓ. The processor handling τ_ℓ is p_u, with $u = proc(i)$, and it should be able to process the whole subtree within its own memory. Depending on the order in which tasks are processed, the required memory may differ. However, it is possible, given a subtree, to obtain its minimum memory requirement $M_{\min}$ and the corresponding traversal (in which order tasks should be executed), using the MinMemory algorithm [13]. Hence, we denote by $M_{\min}(i)$ the minimum memory required to execute the subtree τ_ℓ rooted in v_i. We are now ready to express the **memory constraint**: for each subtree τ_ℓ rooted in v_i, $M_{\min}(i) \leq M_{proc(i)}$.

Given a valid partitioning and mapping (i.e., a set of subtrees and a mapping of subtrees onto processors such that each subtree fits into the processor's memory), one can compute the corresponding execution time of the tree, or **makespan**. Let $desc(i)$ be the indices of tasks that are not in τ_ℓ (rooted in v_i), but that have a parent in τ_ℓ. These tasks are the root of subtrees that are descendants of τ_ℓ, and hence the processor in charge of τ_ℓ will need to send files to the processors in charge of these subtrees.

The makespan can then be computed recursively, where $MS(i)$ denotes the makespan of the subtree rooted in v_i. The makespan for the whole tree is then $MS(r)$. Note that for the subtree rooted in v_r, we have $f_r = 0$.

$$MS(i) = \frac{f_i}{\beta} + \sum_{k\in tasks(i)} \frac{w_k}{s} + \max_{j\in desc(i)} MS(j). \tag{1}$$

The first term corresponds to the incoming communication. The second term is the time to process all tasks on processor $proc(i)$ (no communication to be paid

within the same processor). Finally, the last term corresponds to the longest makespan of descendant subtrees, which are processed in parallel (and hence the longest one determines the makespan).

3.4 Problem complexity

The HetMemPartMap problem targeted in this paper is the following. Given a task tree and a platform with **het**erogeneous **mem**ories, the goal is to **part**ition the tree into subtrees, to **map** each subtree onto a processor, such that the memory constraint on each processor is respected (for the subtree rooted in v_i, $M_{\min}(i) \leq M_{proc(i)}$), and the makespan $MS(r)$ is minimized.

The problem was shown to be NP-complete for a fully homogeneous platform in [10], and considering platforms with heterogeneous memories only makes it more difficult. In the following, we focus on the design of an efficient heuristic for such platforms.

4 Heuristic Strategies

In this section, we describe HetPart, a polynomial-time heuristic for the HetMemPartMap problem. Following the idea of [10], the heuristic works in three steps: (1) partition the tree into subtrees to minimize the makespan; (2) assign the trees to fitting processors and further partition the subtrees that do not fit into memory; (3) adjust the number of subtrees to comply with the number of nodes in the target platform, and possibly reassign the new subtrees to different processors. Unlike the work of Ref. [10], we need to fix the assignment of each subtree to a specific processor, since processors have different memories. Furthermore, we need to consider which processors are still available when taking a partitioning decision in Step 2 or a merging decision in Step 3.

4.1 Minimizing Makespan

In the first step, the objective is to split the tree into a number of subtrees with the aim to minimize the overall makespan. Several heuristics are designed for this case in Ref. [10]. The memory constraint is not the focus in this step yet.

4.2 Fitting into Memory

After the tree has been partitioned with the aim to minimize the makespan, the subtrees need to be allocated to processors while respecting the memory constraints. Gou *et al.* [10] suggest three fitting methods that all cut the existing subtrees further until they reach the (unique) memory constraint. Building on the FirstFit method, we propose the new BiggestFit algorithm (shown in Algorithm 1), which additionally considers the memory size of each processor.

We use a max-priority queue Q to keep the subtrees in S "ordered" according to their memory consumption. For the processors, we sort them by memory sizes

Algorithm 1 BiggestFit

```
1: procedure BiggestFit(S, M)   ▷ Input: subtrees S and proc. memory limits M
2:     Init PQ Q with S;                                   ▷ max-priority queue
3:     M.sort(desc);                                ▷ Sort processors by mem size
4:     while not Q.empty() and not M.empty() do
5:         s ← Q.extractMax(); m ← M.head();
6:         (S_fitted, S_rem) ← MemFit(s, m);
7:         Q.add(S_rem);                                ▷ Reinsert remaining subtrees
8:         ScheduleOn(S_fitted, m);
9:         M.remove(m);                                 ▷ Remove assigned processor
10:    end while
11:    while not Q.empty() do   ▷ No more procs., but further split subtrees
12:        s ← Q.extractMax(); m ← min(M);  ▷ m is the memory of smallest proc.
13:        (S_fitted, S_rem) ← MemFit(s, m);
14:        Q.add(S_rem);                          ▷ Reinsert remaining subtrees in Q
15:    end while
16: end procedure
```

(from largest to smallest) in a dynamic array M. At each iteration of the while loop (Line 4), we fit the currently largest subtree s into the current processor with memory m. This is done using any memory fitting algorithm (referred to as MemFit), and currently we use FirstFit [10]. This algorithm checks the memory required by subtree s, and if it does not fit entirely within memory m, it splits the subtree in order to increase the makespan as little as possible. The result is a subtree that fits within m (denoted as S_{fitted}), and it may also generate new subtrees (denoted as S_{rem}) that are added to the set of subtrees still in need to be assigned to a processor (in the priority queue Q). If S_{rem} is empty (the original subtree fits within m, and hence $S_{fitted} = s$), then this step is ignored.

Thanks to this MemFit algorithm, S_{fitted} can now fit within memory m, and we assign it to the corresponding processor that is removed from the array of available processors (Lines 8 and 9). If all processors have been assigned a subtree but there still remain some subtrees in Q, we take care of them in the second while loop (Line 11). We further split the subtrees with the memory m of the smallest processor as a threshold. All these trees are left unassigned, and we will merge subtrees in the next step in order to be able to assign each subtree to a processor.

4.3 Adjusting the Number of Subtrees

After the tree has been partitioned into subtrees (for makespan minimization, Step 1) and after further splitting the subtrees to fit into the respective memories (Step 2), we need to adjust the number of the resulting subtrees to match the

number of processors. This is mandatory if there are still unassigned subtrees after BiggestFit has been applied on the tree: in this case, we need to decrease the number of subtrees so that each one can be assigned to a processor. However, note that this step may also increase the number of subtrees instead – in case all subtrees have been assigned and there remain some idle processors.

4.3.1 Decreasing the number of subtrees.

Should the previous step have yielded more trees than there are processors, some of them need to be merged. To this end, we propose the HeterMerge heuristic (Algorithm 2). We first construct the quotient tree T of τ, where each subtree in τ becomes a vertex in T and resulting multi-edges between vertices in T are merged and (re)weighted accordingly. The general idea is very similar to Ref. [10]: as candidate merge operations, we either try merging a leaf to its parent and only sibling (Case 1), or only to its parent (Case 2).

The main difference to the homogeneous case is that we need to choose the processor on which the resulting merged tree is to be executed. This choice is done through the ChooseProcessor procedure (see Algorithm 3). If at least one of the subtrees has been assigned already (Line 3), then we select the processor with smallest memory that is able to hold the merged subtree (Line 4). Otherwise, if we were not able to find a processor, we are looking for an available processor to handle the merged subtree. Such processors may have been released in a previous merge iteration. This processor must have enough memory to process the merged subtree, and if there are several candidates, we pick the one with the smallest memory to keep larger processors for further iterations (Line 7). If no suitable processor can be found, we return -1 and this merge is not possible.

Since the processors have identical computing speeds (and only memories of different size), the makespan after a merge can be computed by applying Eq. (1). More precisely, we compute the difference Δ_i between the makespans before and after the merge of node i. Finally, in Lines 23 to 38 of Algorithm 2, we perform the merge that results in the smallest increase of the makespan (if there is at least one valid merge), and we iterate as long as merges are possible, until all subtrees have been successfully assigned to processors. When no further merges are possible, Algorithm 2 breaks in Line 20.

4.3.2 Increasing the number of subtrees.

If all subtrees have already been assigned to processors but there are still some idle processors, some subtrees can be further broken down if it improves the makespan. We employ the SplitAgain algorithm from [10] with a single modification: we check if the resulting subtree fits into the memory of any free processor before assigning the subtree to this free processor.

Algorithm 2 Merge for heterogeneous memories

procedure HeterMerge(τ, C, S, P)
  ▷ Input: tree τ, cut edges C, subtrees S, and set of processors P
  $T \leftarrow$ quotient tree according to τ and C;
  $A \leftarrow$ binary array of length $|P|$, initialized with 1s; ▷ $A[u] = 1 \leftrightarrow$ proc. u has been assigned a subtree
  `toMerge` $\leftarrow |S| - |P|$; ▷ Number of subtrees not yet assigned to a proc.
  while `toMerge` > 0 **do**
    $\Delta_{\min} \leftarrow -\infty$;
    for each node $i \in T$ except the root **do**
      $j \leftarrow$ parent(i);
      if i is a leaf **and** i has only one sibling k **then** ▷ Case 1
        $p \leftarrow$ ChooseProcessor(i, j, k, A);
        $\Delta_i \leftarrow$ estimated increase in $MS(r)$ if i, j and k are merged onto p;
      else ▷ Case 2
        $p \leftarrow$ ChooseProcessor($i, j, 0, A$);
        $\Delta_i \leftarrow$ estimated increase in $MS(r)$ if i and j are merged onto p;
      end if
      if $p \neq -1$ and $\Delta_i < \Delta_{\min}$ **then** $\Delta_{\min} \leftarrow \Delta_i$; $p_{\min} \leftarrow p$; $i_{\min} \leftarrow i$;
      end if
    end for
    if $\Delta_{\min} = -\infty$ **then break**; ▷ No further improvement possible
    end if
    ▷ Now, $i_{\min}, p_{\min}, \Delta_{\min}$ correspond to a possible merge, leading to the smallest increase in makespan
    if $i_{\min}$ is a leaf **and** $i_{\min}$ has only one sibling **then** ▷ Case 1
      Merge $i_{\min}$ to its parent j and sibling k in τ; Update T and C;
      Assign the merged subtree to $p_{\min}$; ▷ And free other procs. next
      if $0 < proc(i) \neq p_{\min}$ **then** $A[proc(i)] \leftarrow 0$;
      else if $0 < proc(j) \neq p_{\min}$ **then** $A[proc(j)] \leftarrow 0$;
      else if $0 < proc(k) \neq p_{\min}$ **then** $A[proc(k)] \leftarrow 0$;
      end if
      `toMerge` $\leftarrow$ `toMerge` $- 2$;
    else ▷ Case 2
      Merge $i_{\min}$ to its parent j in τ; Update T and C;
      Assign the merged subtree to $p_{\min}$; ▷ And free other proc. next
      if $0 < proc(i) \neq p_{\min}$ **then** $A[proc(i)] \leftarrow 0$;
      else if $0 < proc(j) \neq p_{\min}$ **then** $A[proc(j)] \leftarrow 0$;
      end if
      `toMerge` $\leftarrow$ `toMerge` $- 1$;
    end if
  end while
  return ($MS(r)$, C);
end procedure

Algorithm 3 Choose the most suitable processor for the merge

1: **procedure** ChooseProcessor(i, j, k, A)
2: ▷ Input: node i, its parent j, its only sibling k if $k \neq 0$, processor array A
3: **if** $proc(i) \neq 0$ or $proc(j) \neq 0$ or $proc(k) \neq 0$ **then** ▷ At least one of the subtrees is assigned to a processor
4: Let p be the proc. with min. memory among those assigned to i, j, and k s.t. $M_p \geq M_{\min}(i+j+k)$;
5: If such a processor exists, **return** p;
6: **end if**
7: Select a processor p s.t. $A[p] = 0$ and $M_p \geq M_{\min}(i+j+k)$; If several candidates, pick p with minimum memory; If no candidates, $p \leftarrow -1$; ▷ Look for smallest possible fitting processor; $p = -1$ if no proc. found
8: **return** p;
9: **end procedure**

5 Experimental Evaluation

In this section, we describe the experimental settings and experimental results. All results have been obtained via a simulation of the target cluster platforms.

5.1 Settings

5.1.1 Code and Machine.

All algorithms are implemented in C++ and compiled with g++ (v.11.2.0) using the flags "-O2 -fopenmp". They are executed on a workstation with 192 GB RAM and 2x 12-Core Intel Xeon 6126 @3.2 GHz and CentOS 8 as OS. We will publish the code on github after paper acceptance. The baseline algorithm from Ref. [10], which we call HomPart, is also written in C++; it is compiled and executed with the same infrastructure.

5.1.2 Instances.

We evaluate the algorithms on two general sets of trees: elimination trees generated from real-world sparse matrices, and randomly generated ones. The real-world tree workflows were provided by Jacquelin *et al.* [13]; we consider the set of 31 trees that were already used by Gou *et al.* [10] in the homogeneous setting. To avoid overfitting to one particular instance set, we also generate a set of random trees, derived from Prüfer sequences [17] and with random node and edge weights.

We build eight "random" categories with 30 trees each. Within each category, we use six different tree sizes (2K, 4K, 10K, 20K, 30K, and 50K nodes) with five trees for each size. The categories differ in their parametrization w. r. t. node and edge weights as well as fanout and makespan. As most others, the category "Normal" derives its fanout from a Prüfer sequence. Its node /

Table 1: Random trees and their generation parameters.

Category	# Children	Node Weights		Makespan Weights	Edge Weights	
		Mean	Stddev		Mean	Stddev
Normal	Prüfer sequence	(11,200)	10	(0.01,0.9)	(1000,5000)	500
All large	Prüfer sequence	(1100, 20000)	1000	(1.0, 90.0)	(100000, 500000)	50000
All small	Prüfer sequence	(1,20)	1	(0.001,0.09)	(100, 500)	5
Large node weights	Prüfer sequence	(1100, 20000)	1000	(0.01, 0.9)	(1000,5000)	500
Large makespan weights	Prüfer sequence	(11,200)	10	(1.0, 90.0)	(1000,5000)	500
Large edge weights	Prüfer sequence	(11, 200)	10	(0.01, 0.9)	(100000, 500000)	50000
3 children	3 +-1	(11, 200)	10	(0.01, 0.9)	(1000,5000)	500
20 children	20 +-4	(11, 200)	10	(0.01, 0.9)	(1000, 5000)	500

makespan / edge weights all result from a uniform random distribution. For "All large", the expected values of these weights are all multiplied by 100, while for "All small", they are divided by 10. The categories "Large node weights", "Large makespan weights", and "Large edge weights" increase only one of these respective weights (i.e., the m_i's, w_i's or f_i's). Finally, "3 children" and "20 children" have an expected fanout of 3 (standard deviation 1) and 20 (standard deviation 4), respectively; their other weights are as in "Normal" in expectation.

To achieve trees with higher fanout, we changed the rate at which each number appears in the Prüfer sequence. The detailed parameters for the respective uniform distributions are shown in Table 1.

5.1.3 Compute Platforms.

To evaluate how well the new heuristic HetPart exploits heterogeneity, we create synthetic compute platforms that resemble real-world configurations and differ in the number of nodes as well as in the distribution of resources (in particular memory) among these nodes. To make the algorithms' job difficult, we use a modest size: two clusters with 36 nodes each. As for resource distribution, one cluster is 4-fold with four kinds of nodes (9 nodes of each kind), and the other 2-fold with two kinds of nodes (18 nodes of each kind), see Table 2.

Both clusters have "fat" and "light" nodes, whereas the 4-fold cluster additionally has "moderate" and "extra-light" nodes. The "fat" nodes have three

Table 2: Node number and memory allocation on clusters.

Cluster conf.		extra-light	light	moderate	fat
4-fold	mem	$0.5D_{\max}$	$D_{\max}$	$1.5D_{\max}$	$3D_{\max}$
	nodes	9	9	9	9
2-fold	mem	-	$D_{\max}$	-	$3D_{\max}$
	nodes	-	18	-	18

times the memory size of the "light" ones, while the "moderate" ones have 1.5 times the memory size of the "light" ones. Finally, the "extra-light" nodes have half the memory of the "light" ones. In their design, the "light" nodes are given just enough memory to handle the largest task of the workflow instances, hence a memory $D_{\max}$. Thus, the amount of memory given to a certain tree depends not only on the memory capacity of the cluster node, but also on the tree's requirements expressed by its $D_{\max}$. Note that in a real-world setting, the memory actually available to a workflow would also be adapted to its needs. Also, having only "light" nodes corresponds to the *strict* memory scenario in Ref. [10]. All processor speeds and bandwidths are assumed equal (normalized to 1 for speeds and to 500 for bandwidths).

5.1.4 Setup for Algorithmic Comparison.

The two major criteria for comparing HetPart with the baseline HomPart are solution quality (makespan of the produced schedules) and running time. To account for fluctuations in the running time, we perform three runs of each experiment and use the arithmetic mean.

Since the homogeneous algorithm cannot exploit varying memory sizes, the heterogeneous clusters need to be represented in a homogeneous way for HomPart. The main differences stem from the memory limit imposed on each compute node. The strictest memory limit leads to "Many Light" (ML), which takes the 27 nodes that are at least "light". Using only the fat nodes with their full memories is the "Few Fat" (FF) scenario. In between these two is "Some Moderate" (SM), which uses the "fat" and the "moderate" nodes at the memory limit of the latter. The respective configuration of HomPart is suffixed with ML, FF, or SM. Note that the memory would not suffice for the largest tasks if we took all 36 nodes and treated them as "extra-light". Figure 1 illustrates the different cluster configurations.

Both HetPart and HomPart may use different heuristics at each phase, in particular for the initial partitioning for makespan, where heuristics SplitSubtrees, ASAP, and ImprovedSplit are proposed in [10]. We selected the best combinations (regarding solution quality, on average) for our setup, both for HetPart and for HomPart, in order to be as fair as possible. We did not consider ImprovedSplit due to its long running time. It turns out that for HetPart, the best results are obtained with ASAP for the first step, followed

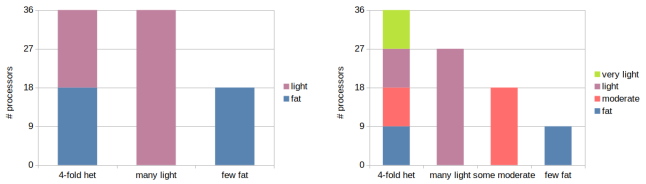


Figure 1: Cluster composition: the heterogeneous cluster and its corresponding homogeneous variants. (a) 2-fold cluster and (b) 4-fold cluster.

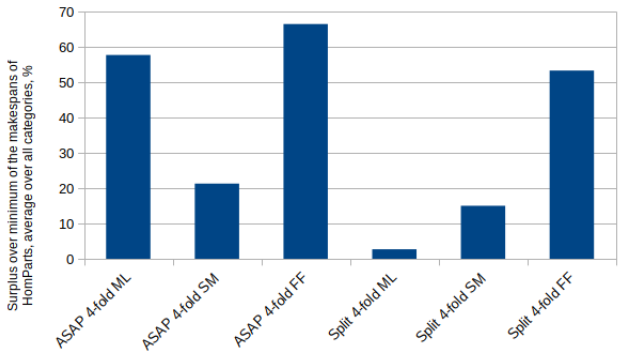


Figure 2: Among all makespans generated by all variants of HomPart per category, we chose the minimal one, and then computed the surplus of other ones over it. This diagram shows these surpluses, aggregated over all categories.

by BiggestFit and HeterMerge or SplitAgain. For HomPart however, surprisingly, using SplitSubtrees in the first step gives better results, when combined with FirstFit in the second step, and finally Merge or SplitAgain. In the following, we use these combinations that respectively returned the best results.

Figure 2 shows the aggregated results for HomPart using SplitSubtrees and ASAP, hence confirming that the former is better in this case.

5.2 Results

We first focus on the 4-fold cluster, and in particular we study the increase of makespan when using HomPart rather than HetPart. We report the percentage of increase in makespan when HomPart is used in various configurations (ML, SM, FF). If HomPart could not find a solution, no bar is reported. The geometric mean is used when aggregating several ratios. Note that the higher the ratio, the more beneficial HetPart is compared to HomPart.

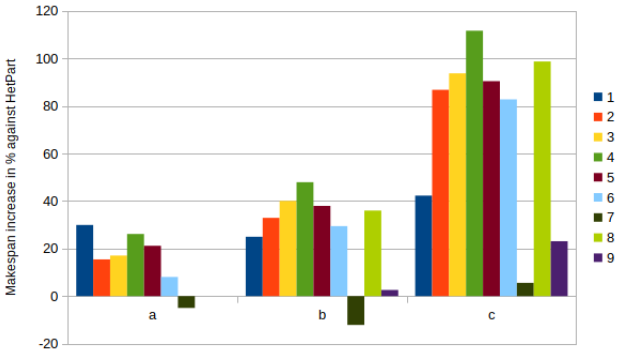


Figure 3: Makespan increase in % compared to HetPart by (a) HomPart-ML, (b) HomPart-SM, (c) HomPart-FF. The two missing bars for HomPart-ML indicate unsuccessful runs (no solution for HomPart in this setting). Instances: (1) sparse matrix trees, (2) normal (random trees), (3) all large, (4) all small, (5) large node weights, (6) large makespan weights, (7) large edge weights, (8) 3 children, (9) 20 children.

5.2.1 Makespan of 4-fold cluster

Figure 3 displays the average increase of the makespan (in %) of the three HomPart scenarios compared to HetPart. Each bar represents an instance group. As most bars are above 0, HetPart performs best overall: averaged over all instance groups, the best homogeneous variant HomPart-ML still increases the makespan by 15.5%. At the same time, note that HomPart-ML is not able to produce results for two instance groups. This robustness problem results from the fact that finding a valid solution can become more difficult if only light nodes are available. If we compare to the next best scenario, HomPart-SM, which is able to solve all instances, HetPart is 25.1% better on average. Overall, HetPart achieves high improvements in most cases but two. In case of large edge weights (7, high communication costs) and a high-fan out with 20 children (9), HomPart performs quite well – if it is able to find a solution.

In the following, we take an individual look at the respective instance groups. On sparse matrix trees (1), HetPart is 25.0% better than the best homogeneous scenario HomPart-SM. HomPart-ML fares comparably to HomPart-SM (29.9% increase), while HomPart-FF is clearly the worst (42.3% increase).

On normal random trees (2), HetPart improves by at least 15.4% (against HomPart-ML). The other two homogeneous variants perform significantly worse (HomPart-SM: 32.9%, HomPart-FF: 86.7%).

For the categories where only weights change (and not the tree topology – "all large" (3) and "all small" (4)), the improvement of HetPart compared to HomPart-ML is 17.1% and 26.1% respectively. Similar results can be observed when node weights (memory consumption per task, (5)) are large: HetPart improves on HomPart-ML by 21.2%.

HetPart works very well in these previous categories as the corresponding instances allow our heuristic to distribute the tasks across the whole cluster. The situation is somewhat different for the categories "20 children" (9) and "large edge weights" (7). Here, all heuristics use only a subset of the cluster since the trees cannot be parallelized and distributed that well. Evidently, the dominance of communication over computation in these trees yields this behavior. HomPart-SM performs best on both of these groups (and significantly better than for other groups) and it is even 12.7% better than HetPart for "large edge weights".

As indicated before, on trees with fixed fanouts ("3 children" (8) and "20 children" (9)), HomPart-ML cannot find a solution for the majority of the trees, hence no results are displayed in this case. The other two homogeneous scenarios do find solutions, but they are much worse than those of HetPart.

Trees with large makespan weights (6) fall between the two poles: HetPart yields the best results again; the improvement on HomPart-ML is rather modest with 8.4%. However, HetPart fares significantly better than HomPart-SM (29.4%) and HomPart-FF (82.7%).

Finally, note that overall, for all categories, HomPart-ML compares the most closely to HetPart (the increase in makespan is low in section (a) of Figure 3), but it also produces the largest number of unsolved trees. On average over all categories, 21.8% of the trees could not be solved by HomPart-ML. For the category "3 children", no tree could be solved. For "20 children", half of the trees were unsolved. The other categories have 2 to 5 unsolved trees out of 30, except for the matrix trees, where all trees could be solved.

5.2.2 Comparison between 4-fold and 2-fold cluster

To explore "how heterogeneous" a system needs to be for significant improvements by HetPart, we also consider the 2-fold cluster described in Section 5.1. The detailed results for this 2-fold cluster are available in Figure 4. Also, in Figure 5, we compare the respectively smallest improvement of HetPart against HomPart on both clusters.

Clearly, with more heterogeneity to exploit, HetPart is able to provide a more tangible improvement on the 4-fold cluster. In most categories, the HetPart improvement against the closest competitor in the 4-fold cluster (dark-blue) lies between 15% and 30%. HetPart performs worse than the closest HomPart only for one category ("large edge weights", 7). In the 2-fold cluster, however, HetPart wins by much smaller margins (2%-13%) and loses in 3 categories. For the matrix trees, HetPart provides tangible improvements in both clusters (24.9% and 20.7%).

5.2.3 Running Times

As shown in Figure 6, the running time of HetPart is comparable to that of HomPart-SM and HomPart-FF (averaged over all instance groups). More precisely, HetPart is 9.7% faster than HomPart-SM but 7.3% slower than

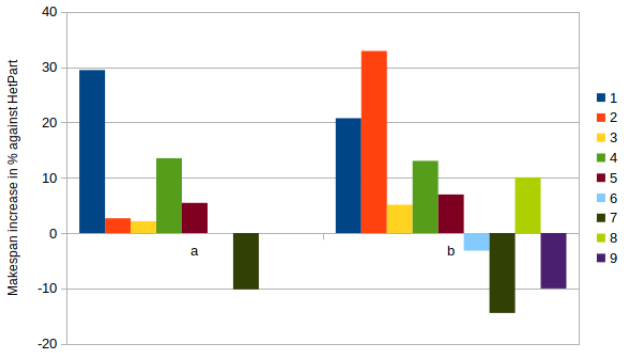


Figure 4: Makespan increase in % compared to HetPart by (a) HomPart-ML, and (b) HomPart-FF on a 2-fold cluster. The missing bars for HomPart-ML indicate unsuccessful runs (no solution for HomPart in this setting). Instances: (1) sparse matrix trees, (2) normal (random trees), (3) all large, (4) all small, (5) large node weights, (6) large makespan weights, (7) large edge weights, (8) 3 children, (9) 20 children.

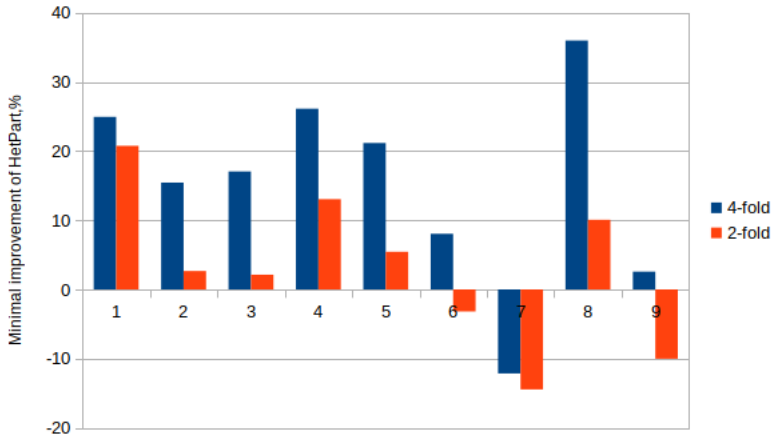


Figure 5: The comparison of the *minimal* improvement of HetPart against HomPart on the 4-fold cluster vs 2-fold cluster. Instances: (1) sparse matrix trees, (2) normal (random trees), (3) all large, (4) all small, (5) large node weights, (6) large makespan weights, (7) large edge weights, (8) 3 children, (9) 20 children.

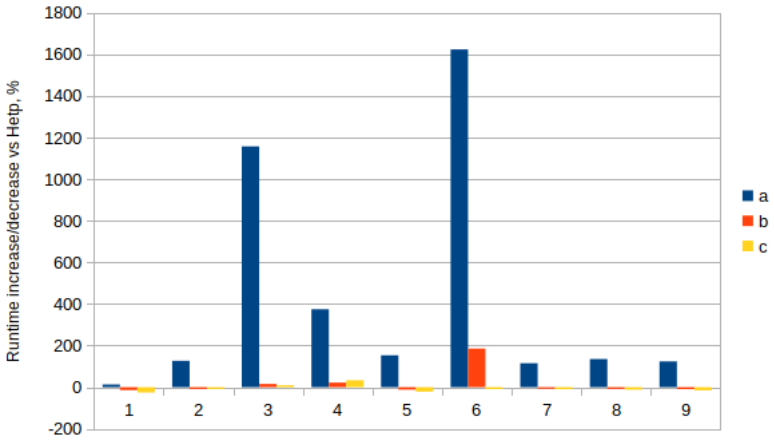


Figure 6: Runtime change in % (lower is better) compared to HetPart by three homogeneous scenarios: (a) HomPart-ML, (b) HomPart-SM, (c) HomPart-FF. Instances: (1) sparse matrix trees, (2) random trees, (3) all large, (4) all small, (5) large node weights, (6) large makespan weights, (7) large edge weights, (8) 3 children, (9) 20 children.

HomPart-FF. At the same time, as we saw above, HetPart provides a much better solution quality. The homogeneous scenario with best quality, HomPart-ML, is much slower. Its running time is 3.5× higher than HetPart's. Our experiments indicate that most time is spent merging. Smaller memory sizes as in HomPart-ML produce trees that require extensive merging, explaining the much longer running time. Note that we do not consider here the three largest matrix trees due to their very long runtime.

6 Conclusions and Future Work

We have studied the problem of tree partitioning for a heterogeneous multiprocessor computing system, where each processor can have a different memory size. Taking heterogeneity into account when partitioning these trees into subtrees pays off: our new heuristic HetPart clearly improves the makespan compared to the homogeneous state of the art. At the same time, the best homogeneous scenario, HomPart-ML, fails to produce valid solutions in many cases due to its inability to exploit the full memory of the cluster and it is 3.5× slower.

Future work includes the increase of the heterogeneity level. This should include different processor speeds and different bandwidths in the cluster. Overall, we expect similar findings for such cases: when the compute platform is sufficiently heterogeneous, a heuristic taking this heterogeneity into account should pay off. However, integrating processor speeds and bandwidths makes a corresponding heuristic significantly more complicated.

References

[1] Adhikari, M., Amgoth, T., Srirama, S.N.: A survey on scheduling strategies for workflows in cloud environment and emerging trends. ACM Computing Surveys (CSUR) **52**(4), 1–36 (2019)

[2] Aupy, G., Benoit, A., Renaud-Goud, P., Robert, Y.: Energy-aware algorithms for task graph scheduling, replica placement and checkpoint strategies. In: Handbook on Data Centers, pp. 37–80. Springer (2015)

[3] Benoit, A., Le Fevre, V., Perotin, L., Raghavan, P., Robert, Y., Sun, H.: Resilient scheduling of moldable parallel jobs to cope with silent errors. IEEE Transactions on Computers (2021). https://doi.org/10.1109/TC.2021.3104747

[4] Benoit, A., Marchal, L., Pineau, J.F., Robert, Y., Vivien, F.: Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. IEEE Transactions on Computers **59**(2), 202–217 (2009)

[5] Benoit, A., Rehn-Sonigo, V., Robert, Y.: Multi-criteria scheduling of pipeline workflows. In: 2007 IEEE International Conference on Cluster Computing. pp. 515–524. IEEE (2007)

[6] Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent advances in graph partitioning. Algorithm engineering pp. 117–158 (2016)

[7] Davis, T.A.: Direct Methods for Sparse Linear Systems. Fundamentals of Algorithms, Society for Ind. and Applied Math., Philadelphia (2006)

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

[9] Feldmann, A.E., Foschini, L.: Balanced partitions of trees and applications. Algorithmica **71**(2), 354–376 (2015)

[10] Gou, C., Benoit, A., Marchal, L.: Partitioning tree-shaped task graphs for distributed platforms with limited memory. IEEE Transactions on Parallel and Distributed Systems **31**(7), 1533–1544 (2020)

[11] He, S., Wu, J., Wei, B., Wu, J.: Task tree partition and subtree allocation for heterogeneous multiprocessors. In: 2021 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom). pp. 571–577 (2021). https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00084

[12] Herrmann, J., Kho, J., Uçar, B., Kaya, K., Çatalyürek, Ü.V.: Acyclic partitioning of large directed acyclic graphs. In: 2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID). pp. 371–380. IEEE (2017)

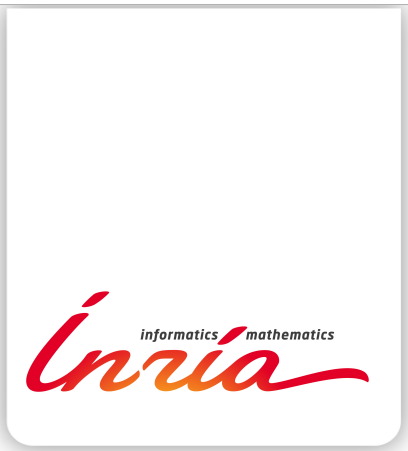
[13] Jacquelin, M., Marchal, L., Robert, Y., Uçar, B.: On optimal tree traversals for sparse matrix factorization. In: 2011 IEEE International Parallel & Distributed Processing Symposium. pp. 556–567. IEEE (2011)

[14] Kulagina, S., Meyerhenke, H., Benoit, A.: Mapping Tree-shaped Workflows on Memory-heterogeneous Architectures. Research report 9458, Inria (2022), `https://hal.inria.fr`

[15] Liu, J., Pacitti, E., Valduriez, P.: A survey of scheduling frameworks in big data systems. International Journal of Cloud Computing **7** (01 2018). https://doi.org/10.1504/IJCC.2018.10014859

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

[17] Prüfer, H.: Neuer Beweis eines Satzes über Permutationen. Archiv für Mathematik und Physik **27**, 142–144 (1918)



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