Mapping tree-shaped workflows on systems with different memory sizes and processor speeds

Svetlana Kulagina, Henning Meyerhenke, Anne Benoit

▶ To cite this version:

Svetlana Kulagina, Henning Meyerhenke, Anne Benoit. Mapping tree-shaped workflows on systems with different memory sizes and processor speeds. Concurrency and Computation: Practice and Experience, 2023, 35 (25), 10.1002/cpe.7842 . hal-04397633

HAL Id: hal-04397633
https://inria.hal.science/hal-04397633

Submitted on 16 Jan 2024

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.

Distributed under a Creative Commons Attribution 4.0 International License

Received: 23 December 2022 | Revised: 21 May 2023 | Accepted: 6 June 2023
DOI: 10.1002/cpe.7842

RESEARCH ARTICLE

WILEY

Mapping tree-shaped workflows on systems with different memory sizes and processor speeds

Svetlana Kulagina[1] | Henning Meyerhenke[1] | Anne Benoit[2]

[1]Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Germany

[2]LIP laboratory, ENS Lyon, Lyon, France

Correspondence
Svetlana Kulagina, Department of Computer Science, Humboldt-Universität zu Berlin, Berlin, Germany.
Email: svetlana.kulagina@hu-berlin.de

Funding information
Deutsche Forschungsgemeinschaft; CRC: "CRC1440 Fonda"

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 large trees, it is crucial to schedule them efficiently, so that their execution time (or makespan) is minimized. Furthermore, it is usually 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 two types of target architecture models: in AM1, each processor can have a different memory size, and in AM2, each processor can also have a different speed (in addition to a different memory size). We design a three-step heuristic for AM1, which adapts and extends previous work for homogeneous clusters [Gou C, Benoit A, Marchal L. Partitioning tree-shaped task graphs for distributed platforms with limited memory. IEEE Trans Parallel Dist Syst 2020; 31(7): 1533–1544]. The changes we propose concern the assignment to processors (accounting for the different memory sizes) and the availability of *suitable* processors when splitting or merging subtrees. For AM2, we extend the heuristic for AM1 with a two-phase local search approach. Phase A is a swap-based hill climber, while (the optional) Phase B is inspired by iterated local search. We evaluate our heuristics for AM1 and AM2 with extensive simulations, and we demonstrate that exploiting the heterogeneity in the cluster significantly reduces the makespan, compared to the state of the art for homogeneous processors.

KEYWORDS
mapping, memory constraint, tree partitioning, workflows

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

A preliminary version of this paper was presented at the *20th Intl. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar)*. It is scheduled for publication in the Proceedings of the 2022 Euro-Par Workshops.

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs License, which permits use and distribution in any medium, provided the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.
© 2023 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

graph representing the tasks and the edges their dependencies. One common form of such DAGs is a rooted directed tree,[1] 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.[2,3]

These workflows are usually very large (thousands of tasks), and they cannot always be executed on a single processor. Indeed, their memory needs may exceed the memory of a single processor, and also the time to execute sequentially such a workflow may be very large. A solution is therefore to execute the workflow in parallel, for example, on a compute cluster where processors have their own local memory and communicate via the network. One must then decide how to map the workflow on the target architecture, that is, on which processor should each task of the workflow be executed. Hence, the workflow is partitioned into subtrees, and each processor executes a part of the tree that must fit into its memory. The quality of a mapping is expressed by the corresponding *makespan*, that is, the total execution time required to run the workflow entirely. Since several processors are used, some parts of the execution can be executed in parallel. In this study, we assume the workflow and its properties to be known before the mapping is decided. Previous work by Gou et al.[3] for completely homogeneous clusters (or other homogeneous platforms) proposed a detailed model to formalize this optimization problem, showed the corresponding decision problem to be $\mathcal{NP}$-complete, and proposed several variants of a successful three-step heuristic. Their goal is to minimize the makespan, while ensuring that each processor can process the subtree assigned to it, that is, that it has enough memory for processing this subtree. The heuristics work as follows: (i) partition the tree into subtrees, minimizing the makespan and not taking the memory limit into account; (ii) further partition subtrees that are currently too large for the memory limit; and (iii) ensure that the number of subtrees is less than or equal to the number of processors of the target platform.

While this previous study provided solutions for homogeneous clusters, more and more compute clusters have different memory sizes, for example, 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 size. In terms of processor speeds, "fat" nodes tend to have slower processor speeds, while "thin" nodes with small memory sizes have faster processors. Of course, there can be configurations in between as well. Thus, adapting the mapping algorithm to different memory sizes and processor speeds is very relevant. Yet, maybe with the exception of He et al.,[4] there are no algorithms in the literature tailored to the problem of mapping tree-shaped workflows on memory- or speed-heterogeneous architectures (i.e., where the memory sizes/processor speeds differ). And, while He et al.[4] design their algorithm with heterogeneity in mind, their experimental setup and results do not consider architectures with different memory sizes or different processor speeds, which are in our focus.

In this article, we focus on two types of target architecture models. In AM1, all processors have the same speed but different memory sizes. In AM2, each processor can also have a different speed (in addition to a different memory size). We hence add up to two levels of heterogeneity compared to a fully homogeneous platform. Our main contributions are

1. the formalization of the problems in the AM1 and AM2 models, that is, how to account for different memory sizes in AM1, and also for different processor speeds in AM2;
2. the design of two partitioning and mapping heuristics (HETPART1 and HETPART2—short for *heterogeneous tree partitioning*) for tree-shaped workflows, respectively, for the AM1 and AM2 models; and
3. an extensive set of simulations on real-world and randomly generated trees to assess the performance of the proposed heuristics.

The heuristics work in three steps, building upon the work by Gou et al.[3] for the homogeneous case. We adapt two of these steps for HETPART1: (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, that is, the processors with sufficient memory sizes. HETPART2 further improves over HETPART1 by considering different processor speeds. Its Phase A starts from a solution computed by HETPART1. In a hill-climbing fashion, it then performs

swaps of processor assignments to further improve the makespan. After a locally optimal assignment has been achieved, one can execute (an optional) Phase B, in which HETPART2 borrows ideas from iterated local search (ILS): it perturbs the solution by swapping subtrees between processors of different speeds, making special effort to swap subtrees to and from processors with the highest speeds. Then, HETPART2 repeats Phases A and B until no further makespan improvement can be achieved.

To evaluate the benefit from exploiting heterogeneity for different inputs, we perform extensive simulations on real-world trees coming from sparse matrix factorization, as well as randomly generated trees. The standard of reference consists of the state-of-the-art algorithm, HOMPART, by Gou et al.[3] For a fair comparison, we use the same heterogeneous target architecture and let the reference algorithm HOMPART work on it in different scenarios regarding hardware consumption. Our experimental results on a cluster with four different memory sizes show that HETPART1 reduces the makespan on average by 15.8% and 25.5%, respectively, compared to the two best homogeneous scenarios. Where the improvement by HETPART1 is only 15.8%, the corresponding homogeneous scenario has the major drawback of not producing a valid solution for more than 20% of the instances. In a cluster with four different processor speeds (in addition to different memory sizes),

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

HetPart2 reduces the makespan found by HetPart1, on average by 53.3%. For six categories of trees out of nine, the improvement amounts to more than 56.9%.

After discussing related work in Section 2, we formalize the models and scheduling problems in Section 3. The algorithmic contributions can be found in Section 4. Extensive simulations are conducted in Section 5. Finally, we conclude and give hints of the future work directions in Section 6.

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.[5] 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[6] and bags of tasks.[7] However, current consensus seems to be that a workflow is best described with a directed acyclic graph (DAG),[8,9] 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.[2,10]

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,[6] focusing on fault tolerance,[11] and also energy efficiency.[12] Recently, an important focus is put on memory optimization, since memory and I/O become a bottleneck.[13,14] 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.[6]

The architecture of the expected underlying platform is a difference in approaches. Some works[11,15] focus on homogeneous computing platforms that can be represented in a more straightforward way in the model (as noted by Alam and Khan[16]). In their systematic literature review, da Silva and Gabriel[17] find that 29 out of 58 works they reviewed assume a homogeneous cluster. Heterogeneous scenarios have been explored in the past[18] and have gained even more traction lately, both in algorithmic research and from the systems perspective.[19] In our paper, we focus on tree workflows, but also findings in related areas support our claim. For example, reviews of scheduling algorithms for cloud computing like that done by Bittencourt et al.[20] note that the difference between a homogeneous and heterogeneous architecture is one of the most crucial ones for the algorithm.

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 by Jacquelin et al.[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.

Our heuristics split the input trees into smaller subtrees. The more general case of partitioning graphs is a well-studied problem.[21,22] For arbitrary graphs and reasonably balanced parts, the corresponding decision problem is $\mathcal{NP}$-complete.[23] The same holds true for partitioning DAGs into acyclic parts, for which Herrmann et al.[15] recently proposed multilevel heuristics. For the case when the strict condition of balanced weights of parts of a graph is relaxed, approaches to its partitioning were proposed by Feldmann and Foschini.[24] In particular, for edge-weighted trees, they present an algorithm that computes a near-balanced partition in polynomial time. Its solution quality is not worse than the optimal balanced partition. Note that their algorithm is not applicable in our scenario since our subtrees may be significantly larger than size-balanced ones.

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 by Gou et al.[3] As pointed out in Section 1, recent work by He et al. has attempted to extend this approach to heterogeneous architectures.[4] 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 memories with different sizes, while He *et al.*use the homogeneous merge from Gou et al.[3]

3 | MODEL

We first describe the application model in Section 3.1 and the platform model in Section 3.2. The constraints and scheduling objectives are then formalized in Section 3.3. Finally, we discuss the problem complexity in Section 3.4.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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 Gou et al.[3]). The tree vertices, numbered from 1 to n, correspond to the tasks, where each task is the smallest non-changeable workflow entity. Each task $v_i \in V$ $(1 \leq i \leq n)$ requires w_i operations to be performed. Vertex $r \in V$ is used to denote 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.[3] 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. Finally, the total memory requirement of the task v_i consists in the input file, the output files (size of the files to be sent to the children), and the memory size m_i: Task v_i requires a memory of $f_i + m_i + \sum_{j:(v_i,v_j)\in E} f_j$ for its execution.

Hence, given a tree workflow, $D_{\max}$ is the maximum memory requirement of a node in this tree:

$$D_{\max} = \max_{v_i \in V} \left\{ f_i + m_i + \sum_{j:(v_i,v_j)\in E} f_j \right\}.$$

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 $P = \{p_1, \ldots, p_l\}$, with $|P| = l$. Each pair of processors can communicate with each other via some network, and communication operations can happen in parallel.

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 $(v_i, v_j) \in E$, 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 bandwidth β. Hence, the time to send the file from v_i to v_j is $\frac{f_j}{\beta}$.

We consider two kinds of heterogeneity of the target platforms.

Architecture model AM1: Memory-heterogeneous nodes

This model adds one level of heterogeneity to a homogeneous cluster, that is, each processor may have a different size of private memory, but the processors are otherwise identical. Hence, for $1 \leq u \leq l$, M_u is the size of the main memory of processor p_u. Each processor p_u is aware of its memory size. 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, that is, $M_u \geq f_i + m_i + \sum_{j:(v_i,v_j)\in E} f_j$. All processors compute 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 $\frac{w_i}{s}$.

Architecture model AM2: Memory- and processor-heterogeneous nodes

In this next level of heterogeneity, processors additionally have different execution speeds—as well as different memory sizes expressed in the same way as in AM1. The time to execute a task differs depending on the processor on which it is mapped, but the communication network remains homogeneous. The heterogeneity in the running times of task executions on different processors is modeled by the speed s_u of a corresponding processor p_u, $1 \leq u \leq l$. The execution time of a single task v_i on different processors now varies and is expressed as w_i/s_u, where p_u is the processor on which task v_i is being executed.

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. Consider Figure 1, where for the subtree τ_1 we have $root(\tau_1) = 1$, and $tasks(1) = \{1, 2, 4, 5, 9\}$. Similarly, as another example, for τ_3 we have $root(\tau_3) = 6$ and $tasks(6) = \{6, 10, 11\}$.

The notation $u = proc(i)$ means that processor p_u handles the subtree rooted at v_i; it must 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 rooted at v_i,

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 1 Partition of a tree.

to obtain its minimum memory requirement and the corresponding traversal (in which order tasks should be executed), using the MinMemory algorithm.[13] Indeed, the MinMemory algorithm computes a traversal that requires the minimum amount of memory for a tree rooted at vertex v_i. It does so by computing an optimal traversal for the subtrees rooted at the children of v_i, then merges these optimal traversals and finally appends v_i to the end. Based on that, we denote by $M_{\min}(i)$ the minimum memory required to execute the subtree τ_ℓ rooted in v_i. This leads to the important **memory constraint**: for each subtree τ_ℓ rooted at v_i, $M_{\min}(i) \leq M_{proc(i)}$.

Given a valid partition 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. Finally, the makespan for the whole tree is $MS(r)$. Recall that for the subtree rooted in v_r, we have $f_r = 0$.

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

The first term corresponds to the incoming communication.

The second term is the time to process all tasks on processor $u = proc(i)$ with the speed s_u (no communication to be paid within the same processor). In case of AM1, all s_u's are equal and can be simplified as s. 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 HetPartMap1 problem is the first problem targeted in this paper, and it is expressed as follows. Given a task tree and a platform with (**het**erogeneous) memories with different sizes, 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, $M_{\min}(i) \leq M_{\mathrm{proc(i)}}$), and the makespan $MS(r)$ is minimized. The HetPartMap2 problem extends HetPartMap1 by further considering different processor speeds in the computation of the makespan.

The partitioning/mapping problem was shown to be $\mathcal{NP}$-complete for a fully homogeneous platform in Gou et al.[3] Considering platforms with different memory sizes and/or different processor speeds only makes it more difficult (there is a direct reduction from the fully homogeneous version of the problem to HetPartMap1 and HetPartMap2). Furthermore, both HetPartMap1 and HetPartMap2, in their decision version ("Given a bound on the makespan, does the problem have a solution with a makespan less than this bound?"), remain in $\mathcal{NP}$. Indeed, given an instance of the decision problem and a solution, we can compute in polynomial time the memory requirement of each subtree and the makespan of the whole tree accounting for different processor speeds; hence we can check whether it is a valid solution.

To deal with the $\mathcal{NP}$-completeness of HetPartMap1 and HetPartMap2, we focus on the design of efficient polynomial-time heuristics for the two optimization problems.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4 | HEURISTIC STRATEGIES

We first describe in Section 4.1 HETPART1, a polynomial-time heuristic for the HETPARTMAP1 problem with different memory sizes. Then, we explain in Section 4.2 how the HETPART1 heuristic can be extended, using local search, to further account for different speeds to address the HETPARTMAP2 problem. The corresponding new two-phase heuristic for this latter problem is called HETPART2.

4.1 | HETPART1: Dealing with different memory sizes

Following the idea of Gou et al.,[3] the heuristic works in three steps: (1) partition the tree into subtrees to minimize the makespan, without paying attention to the memory constraint; (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 Gou et al.,[3] 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. Each step of HETPART1 is repeated only once. However, the steps themselves include recursive computations.

4.1.1 | Step 1: Minimizing the 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 Gou et al.[3] The memory constraint is not the focus in this step yet.

Three heuristics proposed by Gou et al.[3] are SPLITSUBTREES, IMPROVEDSPLIT, and ASAP. The first one produces a two-level partition of the tree with one subtree, *seqSet*, that contains the root of that tree, and $p-1$ independent subtrees. It starts with moving the root of the original tree into *seqSet* and creating the set of independent subtrees from the children of that root. It later operates by successively splitting the biggest (in terms of makespan) independent subtree, moving its root into *seqSet* and adding the root's children to the list of independent subtrees. IMPROVEDSPLIT improves SPLITSUBTREES by building a multi-level solution. It splits not only the independent subtrees, but both *seqSet* and independent subtrees until no makespan improvement is possible. Finally, ASAP tries to cut edges that are as close to the root as possible, to parallelize the potential execution of the tree.

For HETPART1, we use ASAP since it performed best in configuration experiments (see Section 5.1.4).

4.1.2 | Step 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.[3] 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 (Line 2). The processors are sorted by memory sizes (from largest to smallest) in a dynamic array M. In each iteration of the while loop (Line 4), we fit the currently largest subtree s into the processor with the currently biggest available memory m. This is done using any memory fitting algorithm (referred to as MEMFIT), and currently we use FIRSTFIT.[3] This algorithm checks the memory required by subtree s (using the result from the MINMEMORY algorithm), and if it does not fit entirely within memory m, it further splits the subtree. This splitting is done according to the optimal memory traversal provided by MINMEMORY. If the traversal of an edge exceeds the memory m, then this edge is cut, as well as edges going to tasks that have not yet been traversed. Hence, the result is a subtree that fits within m (denoted as S_{fitted}, Line 6), 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 (Line 13). The priority queue Q of the subtrees allows us to manage incoming subtrees with different memory requirements with the least overhead. All these trees are left unassigned, and we will merge subtrees in Step 3 in order to be able to assign each subtree to a processor

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 2 Attempting to allocate tree τ_1.

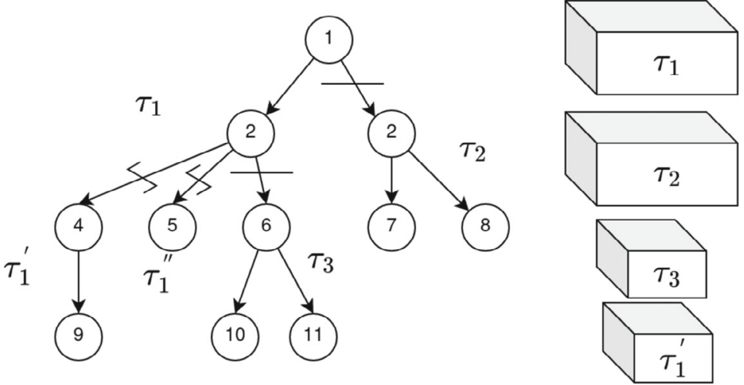

FIGURE 3 Tree τ_1 had to be further split, so that new trees τ_1' and τ_1'' were added.

As an illustrative example, the biggest subtree τ_1 in Figure 2 is too big to be placed on the first processor. Therefore, we further split it, as shown in Figure 3. Now, τ_1 is small enough to be placed there, but two new subtrees join the queue: τ_1' and τ_1'' (their cut edges are marked with zigzags). Let us assume that they are the smallest ones in terms of memory requirement. The remaining subtrees, respectively, fit on the processors that are free for them; however, there is no processor left for τ_1''. This subtree will have to be merged with another one in the next step.

Unlike Step 1, makespan optimization is not the main focus of BIGGESTFIT. Instead, it focuses on fitting the subtrees in memory. However, we expect it to achieve better makespans than the memory-fitting methods in Gou et al.,[3] by exploiting memory heterogeneity. Note that cutting the subtrees may increase the overall makespan, since these subtrees were returned by the first step, optimizing for makespan. HETPART1 is able to recognize that some processor memories are bigger than the others, and that bigger subtrees fit on them without any further action. When fitting, HETPART1 may perform fewer cuts than HOMPART since the latter will attempt to reduce all subtrees to the same (smaller) size. A partition with fewer cuts resembles the original solution better than one with more cuts; hence, the former is likely to have a better makespan. Moreover, fewer cuts require fewer additional tree traversals, thus taking less running time for mapping.

4.1.3 | Step 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, this step may also want to increase the number of subtrees instead—in case there remain some idle processors and the makespan can be improved by splitting some of the subtrees.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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
```

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 the original tree τ. In a quotient tree, each subtree in the original tree is contracted to a single vertex—with the memory requirement of this subtree attached; resulting multi-edges between vertices in T are merged and (re)weighted accordingly. The general idea of the HeterMerge algorithm is very similar to that of Gou et al.[3]: 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 the 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 Equation (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.

Note that this step ensures that merging happens not with arbitrary subtrees, but, among all possibilities, with those subtrees that lead to the smallest overall makespan. This way, we undo some "damage" that we may have caused to the makespan in Step 2, when we were forced to cut some subtrees to make them fit into memory. Distinguishing between different memory sizes (potentially) allows us here to achieve a smaller makespan than what HomPart would have achieved in the same situation, because we might be able to merge a subtree to another subtree that still has empty space in the memory of its processor. HomPart, in turn, would often not be able to notice this available (heterogeneous) memory—and the valid improving merge.

Increasing the number of subtrees

If all subtrees have already been assigned to processors but there are still some idle processors, then some subtrees can be further broken down if it improves the makespan. We employ the SplitAgain algorithm from Gou et al.[3] 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.

4.1.4 | Time complexity

To establish an upper bound on the time complexity of HetPart1, we look at each of the three phases separately. The dominant part in BiggestFit is the first while-loop. In the worst case, it may be executed $\mathcal{O}(|V| + |P|)$ times, where $|V|$ is the number of tasks in the tree and $|P|$ the number of

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 2. Merge for memories with different sizes

```
1: function HETERMERGE(τ, C, S, P)
2:                          ▷ Input: tree τ, cut edges C, subtrees S, and set of processors P
3:   T ← quotient tree according to τ and C;
4:   A ← binary array of length |P|, initialized with 1s;   ▷ A[u] = 1 ↔ proc. u has been assigned a subtree
5:   toMerge ← |S| − |P|;                          ▷ Number of subtrees not yet assigned to a proc.
6:   while toMerge > 0 do
7:     Δ_min ← −∞;
8:     for each node i ∈ T except the root do
9:       j ← parent(i);
10:      if i is a leaf and i has only one sibling k then                              ▷ Case 1
11:        p ← CHOOSEPROCESSOR(i, j, k, A);
12:        Δ_i ← estimated increase in MS(r) if i, j and k are merged onto p;
13:      else                                                                           ▷ Case 2
14:        p ← CHOOSEPROCESSOR(i, j, 0, A);
15:        Δ_i ← estimated increase in MS(r) if i and j are merged onto p;
16:      end if
17:      if p ≠ −1 and Δ_i < Δ_min thenΔ_min ← Δ_i; p_min ← p; i_min ← i;
18:      end if
19:    end for
20:    if Δ_min = −∞ then break;                          ▷ No further improvement possible
21:    end if
22:        ▷ Now, i_min, p_min, Δ_min correspond to a possible merge,leading to the smallest increase in makespan
23:    if i_min is a leaf and i_min has only one sibling then                            ▷ Case 1
24:      Merge i_min to its parent j and sibling k in τ; Update T and C;
25:      Assign the merged subtree to p_min;                    ▷ And free other procs. next
26:      if 0 < proc(i) ≠ p_min then A[proc(i)] ← 0;
27:      else if 0 < proc(j) ≠ p_min then A[proc(j)] ← 0;
28:      else if 0 < proc(k) ≠ p_min then A[proc(k)] ← 0;
29:      end if
30:      toMerge ← toMerge −2;
31:    else                                                                             ▷ Case 2
32:      Merge i_min to its parent j in τ; Update T and C;
33:      Assign the merged subtree to p_min;                     ▷ And free other proc. next
34:      if 0 < proc(i) ≠ p_min then A[proc(i)] ← 0;
35:      else if 0 < proc(j) ≠ p_min then A[proc(j)] ← 0;
36:      end if
37:      toMerge ← toMerge −1;
38:    end if
39:  end while
40:  return (MS(r), C);
41: end function
```

processors. Since we assume $|V| \geq |P|$, this is $\mathcal{O}(|V|)$ (i.e., $O(n)$ since $|V| = n$). Within the loop, the dominant part is the call to MEMFIT, in our case FIRSTFIT. The latter can be bounded by $\mathcal{O}(|V|^2)$ time per call.[13]

The algorithm HETERMERGE has two nested loops, each of which runs at most $|V|$ times. The makespan computation and processor selection within the inner loop takes $\mathcal{O}(|V|)$ and $\mathcal{O}(|P|)$ time, respectively. This leads to an upper bound of $\mathcal{O}(|V|^3)$ in total for HETERMERGE. Finally, SPLITAGAIN does not exceed the cubic bound, leading to $\mathcal{O}(n^3)$ time for the whole HETPART1 algorithm in total.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 3. Choose the most suitable processor for the merge

```
1: function CHOOSEPROCESSOR(i, j, k, A)
2:                                   ▷ Input: node i, its parent~j, its only sibling k if k ≠ 0, processor array A
3:   if proc(i) ≠ −1 or proc(j) ≠ −1 or proc(k) ≠ −1 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 ≥ 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 ≥ M_min(i + j + k); If several candidates, pick p with minimum memory; If no candidates, p ← −1;
     ▷ Look for smallest possible fitting processor; p = −1 if no proc. found
8:   return p;
9: end function
```

Algorithm 4. HETPART2 PHASE A: SWAP THE BEST PAIR UNTIL NO FEASIBLE IMPROVING SWAP EXISTS

```
1: function SWAPUNTILBEST(Quotient tree T, set of processors P, mapping map)
2:   best ← pair(map, makespan(T, P, map));
3:   while true do
4:     curr ← best;
5:     for all pairs (p, p′) of P × P do
6:       if isSwapFeasible(best.first, p, p′) then
7:         map ← swap(best.first, (p, p′));
8:         next ← pair(map, makespan(T, P, map));
9:         if next.second<curr.second then
10:          curr ← next;                                   ▷ better solution is stored
11:        end if
12:      end if
13:    end for
14:    if curr.second<best.second then
15:      best ← curr;                                       ▷ execute improving swap
16:    else
17:      return best;                          ▷ stop because no improving swap exists
18:    end if
19:  end while
20: end function
```

4.2 | HETPART2: Dealing with different memory sizes and different processor speeds

To provide a heuristic for the architecture model AM2, we first observe that a solution computed by HETPART1 is a valid solution for HETPARTMAP2 as well. This solution includes the partition of the tree τ and the assignment of each resulting subtree to a processor. We consider the quotient tree T of τ corresponding to this solution (hence, each node of T corresponds to a subtree of τ). Each node of T is assigned to a processor in P. We propose a heuristic in two phases, where the second one is optional.

4.2.1 | Phase A: Hill-climbing to a local optimum

We use the solution from HETPART1 as a starting solution for a hill-climbing local search heuristic called SWAPUNTILBEST. This heuristic constitutes the first phase (Phase A) of HETPART2. In an iterative fashion, we gradually improve the current solution by swapping the two subtree assignments that yield the best (positive) makespan improvement in the current iteration. In fact, the approach shows some resemblance to the Kernighan-Lin algorithm[25] for graph partitioning—in that it swaps the pair with the highest improvement (but here only real improvements). Algorithm 4 shows the pseudocode.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Before the local search loop, we store the initial solution as best one so far and compute its makespan (Line 2). The while-loop runs until no further improvement can be found (Line 17). Of course, one could also set a maximum number of iterations or running time. Even in its current form, it is guaranteed to converge since each iteration leads to an improvement and the makespan has to be nonnegative.

In each of its iterations, we compute for each feasible swap of processor pairs the makespan of the resulting solution (Lines 5 to 13). The procedure `isSwapFeasible` called in Line 6 checks if the memories of each processor in the pair are big enough to hold the subtree located on the other processor (recall that each processor p_u is aware of the size of its memory M_u). The currently best candidate pair is kept in *curr*—if it improves the best solution seen so far (Line 10). Executing a swap (e.g., Line 7) means to swap the assignment of the two subtrees and their processors. If executed twice on the same pair, a swap returns the mapping into its previous state.

If the best swap found in this *while* iteration improves the solution from the previous iteration, then we replace the old solution with it (Line 15) and start the next iteration—otherwise we return the current solution (Line 17). Note that it may be possible to avoid the recomputation of some feasible swaps and their makespan in the next iteration. We choose this simple approach for two reasons: (i) swaps often affect the makespan of many other future swaps anyway and (ii) the algorithm's empirical running time is small compared to HETPART1.

Note that each iteration of the *while* loop takes $\mathcal{O}(|P|^2)$ time, where $|P|$ is the number of processors. Bounding the number of iterations is challenging, though. A slight adaptation of Algorithm 4 can be shown to have overall time complexity $\mathcal{O}(|P|^2)$. Observing that $|T| \leq |P|$, one could achieve this by allowing each pair of quotient tree nodes to be swapped at most once. Note, however, that this would potentially degrade the solution quality, so that we do not pursue this option.

In summary, we do not modify the partition of the tree in this heuristic, but instead we swap already computed subtrees between processors. Swaps allow us to try out different mappings, but only the best one observed makes its way to the next round. By swapping this way, we take different processor speeds into account, which often leads to better makespans.

4.2.2 | Phase B: Perturbations

If the running time of the mapping algorithm is not of highest concern, then HETPART2 can be executed with an optional Phase B to further improve the solution obtained from Phase A with SWAPUNTILBEST. To escape the local optimum computed by SWAPUNTILBEST, we use techniques from the metaheuristic iterated local search (ILS).[26] More precisely, we perturb a locally optimal solution in order to diversify and escape to probably unexplored areas of the search space.

A natural approach would be to swap some subtrees between processors in a randomized way. However, some swaps are more valuable for the perturbation than others. Firstly, if there are several processors with the same characteristics (memory size and processor speed), then swapping subtrees between them is of no effect to the makespan. Such swaps are thus excluded. Secondly, preliminary experiments showed that the processors with the highest speeds have the biggest impact on the makespan. If they also have the smallest memory sizes, then not many subtrees can fit on them. Giving them new subtrees with different memory sizes first is crucial to achieve a promising perturbed solution.

The main idea of the algorithm is that swaps that are more valuable for the perturbation are executed earlier, before competing swaps (involving the same vertices of the quotient tree) invalidate them. Algorithm 5 shows the pseudocode for this heuristic. We keep two data structures: a map M (Line 2) that stores possible swaps for each vertex in the quotient tree T; and the list of potential swaps SW (Line 3). We use the former to execute some swaps earlier than the others. We first build all potential swaps, that is, swaps that are feasible and change subtrees between processors of different speeds (Line 6). We add these swaps into SW; at the same time we update M, where we add each new potential swap into the sets of swaps of both subtrees involved in this swap (Line 8).

Then, we execute the swaps, starting from those that include the fastest processors (Line 12). Each time a swap is executed, it invalidates itself, so that the same swap stored for the other vertex in M is also invalidated. This prevents subtrees from swapping back and forth. After no swaps on the fastest processor group are available any more, we start executing swaps for the subtrees that have the least number of possible swaps (Line 19). If the next swap is still feasible and not invalidated, then we execute it. Note that if a vertex has no potential swaps or if all of them have been invalidated previously, we simply skip this vertex.

Building upon PERTURB, we can now formulate the HETPART2 heuristic in its two-phase version (Algorithm 6Algorithm 6). It executes SWAPUNTILBEST (Line 8), perturbs the result with PERTURB (Line 7), and repeats the two until no further makespan improvement can be achieved (Line 9).

5 | EXPERIMENTAL EVALUATION

In this section, we describe the experimental settings in Section 5.1 and a representative subset of the experimental results in Section 5.2. All the results have been obtained via simulations on the target cluster platforms.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 5. PERTURBATION

```
1:  procedure PERTURB(Quotient tree T with the set of vertices V, set of processors P, mapping map)
2:      Init map M with pairs {v ∈ V, ∅};            ▷ Map of vertices to the potential swaps involving this vertex
3:      SW ← [];                                       ▷ The list of potential swaps
4:      for all pairs (p, p′) of P × P do
5:          Swap sw ← (p, p′);
6:          if isSwapFeasible(map, p, p′) and s_p ≠ s_p′ then
7:              SW.add(sw);
8:              M[map(p)].add(sw); M[map(p′)].add(sw);
9:          end if
10:     end for
11:     Let s_max be the highest processor speed in P;
12:     for all swaps sw ∈ SW do
13:         if (s_sw.first = s_max or s_sw.second = s_max ) and isSwapFeasible(map, sw) then
14:             map ← swap(map, sw);
15:             Invalidate sw;
16:         end if
17:     end for
18:     Init PQ Q with M;            ▷ Build a priority queue of the vertices by the number of potential swaps
19:     while not Q.empty() do
20:         (v, SW) ← Q.extractMin();                  ▷ Take the subtree v with the least available swaps
21:         for sw ∈ SW do
22:             if isSwapFeasible(map, sw) and isSwapValid(sw) then
23:                 map ← swap(map, sw);
24:                 Invalidate sw;
25:             end if
26:         end for
27:     end while
28:     return map;
29: end procedure
```

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". All experiments are executed on workstations with 192 GB RAM and 2x 12-Core Intel Xeon 6126 @3.2 GHz and CentOS 8 as OS. The code can be downloaded at https://box.hu-berlin.de/d/e01d496a8a18429e849b/ with password "het-sched". The input data that support the findings of this study are openly available under the same url, under the "data" folder. The baseline algorithm by Gou et al.,[3] 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 trees. 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.[3] in the homogeneous setting.

To avoid overfitting to one particular instance set, we also generate a set of random trees, derived from Prüfer sequences[27] and with random node and edge weights; see Table 1 for detailed parameters.

Prüfer sequences are sequences of numbers corresponding to the labels of the nodes in the tree. For each node, its degree (the number of its neighbours) is equal to the number of times this node appears in the sequence. Thus, generating sequences with more repetitions of numbers in them means building trees with a higher fanout (due to larger average degrees in the resulting tree).

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 6. HETPART2 A+B

1: **procedure** HETPART2 A+B(Quotient tree T with the set of vertices V, set of processors P, mapping map)
2: $MS_{min} \leftarrow$ SWAPUNTILBEST(T, P, map); $MS_{new} \leftarrow \infty$;
3: **repeat**
4: **if** $MS_{new} < MS_{min}$ **then**
5: $MS_{min} \leftarrow MS_{new}$;
6: **end if**
7: $map \leftarrow$ PERTURB(T, P, map);
8: $MS_{new} \leftarrow$ SWAPUNTILBEST(T, P, map);
9: **until** ($MS_{new} \geq MS_{min}$)
10: return MS_{min};
11: **end procedure**

TABLE 1 Random trees and their generation parameters.

		Node weights			Edge weights	
Category	**# Children**	**Mean**	**Stddev**	**Makespan weights**	**Mean**	**Stddev**
Random	Prüfer sequence	(11,200)	10	(0.01,0.9)	(1000,5000)	500
Largem_i, w_i, f_i	Prüfer sequence	(1100, 20,000)	1000	(1.0, 90.0)	(100,000, 500,000)	50,000
Smallm_i, w_i, f_i	Prüfer sequence	(1,20)	1	(0.001,0.09)	(100, 500)	5
Large m_i	Prüfer sequence	(1100, 20,000)	1000	(0.01, 0.9)	(1000,5000)	500
Largew_i	Prüfer sequence	(11,200)	10	(1.0, 90.0)	(1000,5000)	500
Largef_i	Prüfer sequence	(11, 200)	10	(0.01, 0.9)	(100,000, 500,000)	50,000
Larger fanout	3 +-1	(11, 200)	10	(0.01, 0.9)	(1000,5000)	500
Largest fanout	20 +-4	(11, 200)	10	(0.01, 0.9)	(1000, 5000)	500

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 "random" derives its fanout from a Prüfer sequence. Its node / makespan / edge weights all result from a uniform random distribution. For "large m_i, w_i, f_i", the expected values of these weights are all multiplied by 100, while for "small m_i, w_i, f_i", they are divided by 10. The categories "large m_i", "large w_i", and "large f_i" increase only one of these respective weights. Finally, "larger fanout" and "largest fanout" have an expected fanout of 3 (standard deviation 1) and 20 (standard deviation 4), respectively; their other weights are as in "random" 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 heuristics HETPART1 and HETPART2 exploit 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 sizes and processor speeds) among these nodes. To make the algorithms' job difficult, we use a modest size: a cluster with 36 nodes. As for resource distribution, the cluster is 4-fold with four kinds of nodes (9 nodes of each kind), see Table 2.

The cluster nodes at the two ends of the spectrum are called "fat" and "extra-light", while "moderate" and "light" nodes lie between these two extremes. The "fat" nodes have three 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. As for the speeds, the situation is reversed: the "extra-light" nodes are the fastest, having three times the speed of "moderate" nodes, while "light" nodes have 1.5 their speeds and "fat" ones only half their speed. In their design, the "light" nodes are given just enough memory to handle the largest task of the workflow instances, hence a memory size of $D_{\max}$. 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

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Node number and memory allocation on clusters.

Cluster configuration		Extra-light	Light	Moderate	Fat
4-fold	Memory	$0.5D_{max}$	D_{max}	$1.5D_{max}$	$3D_{max}$
	Speeds	3	1,5	1	0,5
	# Nodes	9	9	9	9

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 Gou et al.[3]

5.1.4 | Setup for algorithmic comparison

The two major criteria for comparing HetPart1 with the baseline HomPart, as well as for comparing HetPart1 to its extension HetPart2, 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 nine 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, hence a total of 18 nodes. 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”.

As HetPart1 and HomPart ignore processor speeds, all processor speeds are assumed equal (normalized to 1) when comparing these heuristics. For HetPart2, in turn, the actual speeds are used. When comparing HetPart1 and HetPart2, we compute the speed-aware makespan for both. This approach allows us to investigate the benefit of HetPart2 by taking processor speeds into account. For all heuristics, the bandwidth is set to 500.

Both HetPart1 (and thus implicitly also HetPart2) and HomPart may use different subheuristics in each phase—in particular for the initial partitioning for makespan, where heuristics SplitSubtrees, ASAP, and ImprovedSplit were proposed by Gou et al.[3] We select the best combinations (regarding solution quality, on average) for our setup, both for HetPart1 and for HomPart, in order to be as fair as possible. We do not consider ImprovedSplit due to its excessive running time. It turns out that for HetPart1, the best results are obtained with ASAP for the first step, followed by BiggestFit and HeterMerge or SplitAgain. For HomPart, however, using SplitSubtrees in the first step gives (surprisingly) better results when combined with FirstFit in the second step and finally Merge or SplitAgain. Figure 4 shows the aggregated results for HomPart using SplitSubtrees and ASAP, hence confirming that the former is better in this case. In the following, we use the combinations that respectively returned the best results. Since HetPart2 starts with a HetPart1 solution, we indirectly use for it the same heuristics as for HetPart1.

5.2 | Results

We first compare the makespan difference between using HetPart1 and using HomPart. To this end, we use HetPart1 as baseline and report the increase in makespan when HomPart is used in various configurations (ML, SM, FF). If HomPart could not find a solution, no bar is reported. Note that the higher the ratio, the more beneficial HetPart1 is compared to HomPart. For the comparison between HetPart1 and HetPart2, we study by how much HetPart2 further decreases the makespan of HetPart1, when accounting for processor speeds. For the runtime of HetPart2 we only take the time to execute the swaps, without computing the initial solution, and compare it with the runtime of HetPart1 that would provide the initial solution. The geometric mean is used when aggregating several ratios. In our computation, we filter out the trees when a relation could not be computed (e.g., if HomPart was unable to find a solution or if the runtimes were recorded as zeros). We present the results in form of barplots, where the height of the bar represents the mean of the values, and the whisker shows the confidence interval.

5.2.1 | Makespan

HetPart1

Figure 5 displays the average makespan of the three HomPart scenarios compared to the new algorithm HetPart1 as baseline (100%). Each bar represents an instance group. As most bars are above 1, HetPart1 performs best overall: averaged over all instance groups, the best homogeneous variant HomPart-ML still increases the makespan by 15.8%. At the same time, note that HomPart-ML is not able to produce results

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 4 Makespan produced by all variants of HOMPART in relation to the baseline ASAP-HOMPART-ML. Lower is better.

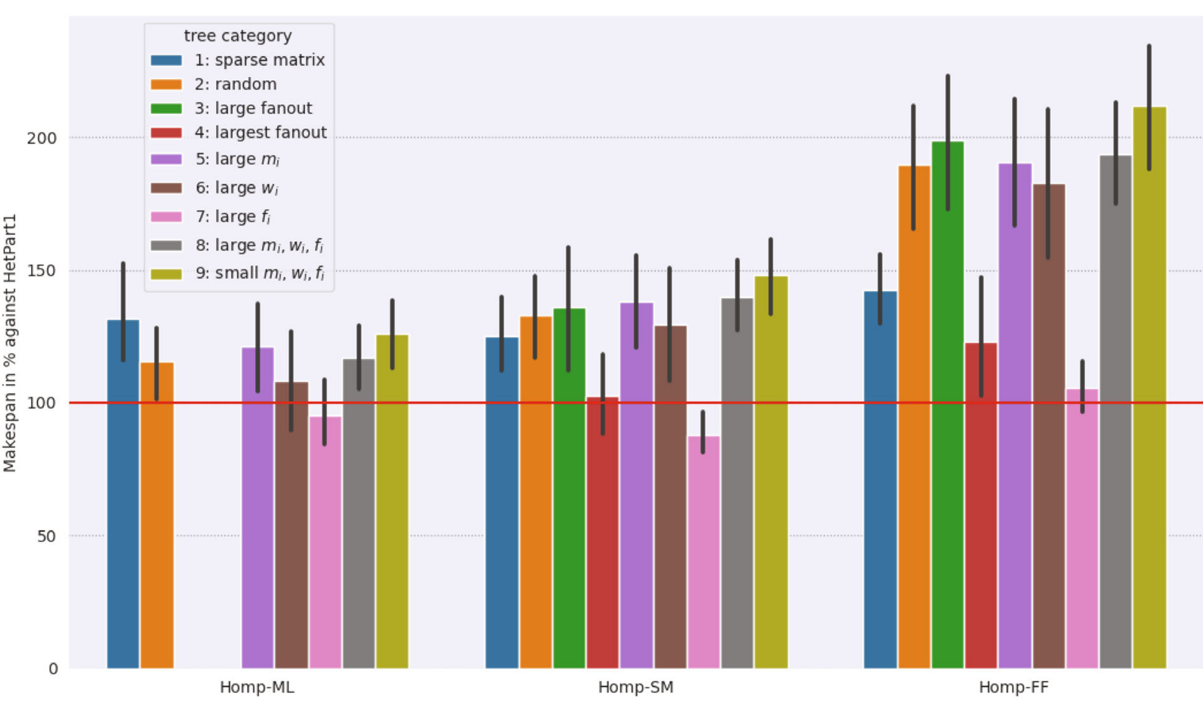

FIGURE 5 Makespan of HOMPART-ML, HOMPART-SM, and HOMPART-FF in relation to the new algorithm HETPART1 as baseline (100%). Lower is better. The two missing bars for HOMPART-ML indicate unsuccessful runs without solutions.

for two instance groups (3 and 4). 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 HETPART1 to HOMPART-SM, the next best scenario that is able to solve all instances, HETPART1 is on average 25.5% better. Overall, HETPART1 achieves high improvements in most cases but two. In case of "7: large f_i" (high communication costs) and "4: highest fanout", 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 "1: sparse matrix trees", HETPART1 is 25.0% better than the best homogeneous scenario HOMPART-SM. HOMPART-ML fares comparably to HOMPART-SM (31% increase), while HOMPART-FF is clearly the worst (42.2% increase). On "2: random" trees, HETPART1 improves by at least 15.4% (against HOMPART-ML). The other two homogeneous

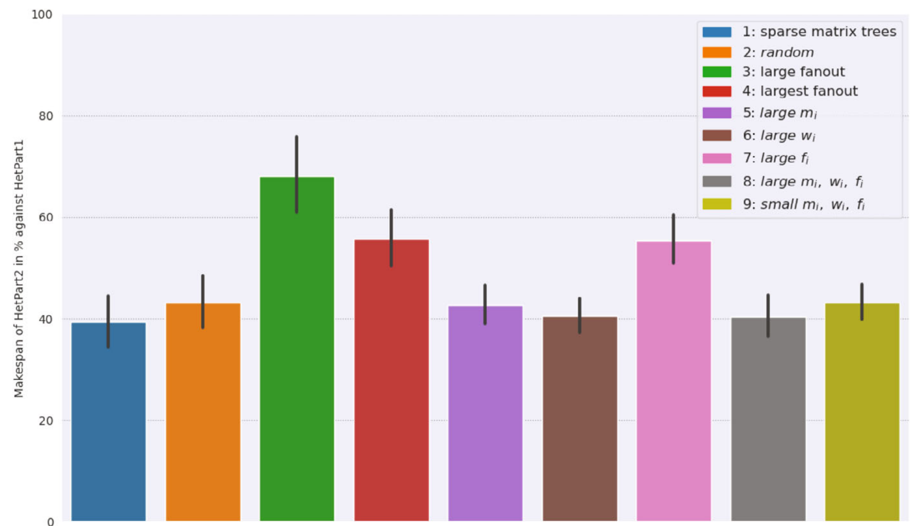

FIGURE 6 Makespan of HETPART2 A+B in relation to the baseline HETPART1 (100%). Lower is better.

variants perform significantly worse (HOMPART-SM: 33.0%, HOMPART-FF: 86.7%). For the categories where only weights change (and not the tree topology – "8: large m_i, w_i, f_i" and "9: small m_i, w_i, f_i"), the improvement of HETPART1 compared to HOMPART-ML is 17.2% and 26.1%, respectively. Similar results can be observed when node weights (memory consumption per task, "5: large m_i") are large: HETPART1 improves on HOMPART-ML by 21.2%.

HETPART1 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 "4: largest fanout" and "7: large f_i". 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 12.5% better than HETPART1 for "7: large f_i".

As indicated before, on trees with fixed fanouts ("3: large fanout" and "4: largest fanout"), 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 HETPART1. Trees with large w_i (6) fall between the two poles: HETPART1 yields the best results again; the improvement on HOMPART-ML is rather modest with 8.3%. However, HETPART1 fares significantly better than HOMPART-SM (29.4%) and HOMPART-FF (82.7%).

Finally, note that overall, looking at all categories, HOMPART-ML is the best competitor, 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: large fanouts", no tree could be solved. For "4: largest fanout", half of the trees were unsolved. The other categories have 2 to 5 unsolved trees out of 30, except for the sparse matrix trees, where all trees could be solved.

HETPART2: *Both phases vs.* HETPART1 *and comparison between phases*

HETPART2 with both phases manages to achieve a remarkable average improvement of 53.3% in makespan. As demonstrated by Figure 6, for all categories of trees except for "3: large fanout", "4: largest fanout", and "7: large f_i", the improvement by HETPART2 of the HETPART1 makespan ranges from 57.9% to 60.7%. For the two categories of trees with high fanout, the improvement is smaller (31.9% and 44.4% for "3: large" and "4: largest fanout", respectively). The categories with higher fanout are difficult for HETPART1 to deal with, and continue to be so for HETPART2. Also, a higher communication load (category "7: large f_i") restricts the ability of both algorithms to exploit the parallelism of the cluster. This might have to do with more subtrees being dependent on other subtrees to finish.

The makespan improvement specifically obtained in Phase B is shown in Figure 7. The blue+dark-purple (top) bars represent the makespans achieved by only Phase A, while dark-purple only (bottom) bars are the makespans achieved after both phases A and B. Interestingly enough, the category "4: largest fanout" is the one most amenable to perturbation. The improvement of only one round of swaps without perturbations was only 30.1% for this category, making the perturbations achieve additional 16.2% of improvement. "1: sparse matrix trees" was the second most sensitive, with 9.0% improvement. For other categories, the additional improvement achieved by perturbations was smaller with values between 2.5% (on "7: large f_i") and 5%. The susceptibility towards perturbation is tree-specific, but averages itself out on most categories. "7: large f_i" is the least susceptible to perturbations, because high communication costs lower the potential for makespan minimization. If the application running time is dominated by the time it takes to transfer files in the network, then different mappings of subtrees to processors will have little impact on

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

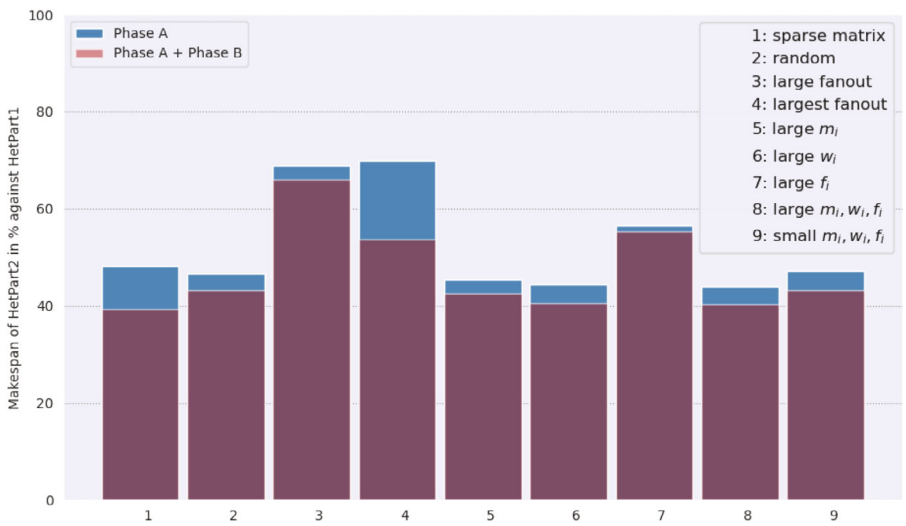

FIGURE 7 Makespans of HETPART2 A and HETPART2 A+B in relation to the baseline HETPART1 (100%). Lower is better.

the resulting makespan. On the contrary, for "4: largest fanout", a category that was difficult for HETPART1, HETPART2 is able to unleash unused potential with regard to execution on faster processors. The same can be observed for "1: sparse matrix trees", a category that has some notably thin and big trees.

In Phase B, the number of perturbations until convergence is quite low, fluctuating between 2 and 5. For over half of the trees, Phase B even stops after only one perturbation (without improvement over Phase A). More perturbation iterations mean an improvement over Phase A, but the properties of the tree have more impact on the efficiency of Phase B than the number of iterations, as illustrated by the two extremes: "4: largest fanout' and "7: large f_i."

5.2.2 | Running times

HETPART1 *compared to* HOMPART.

As shown in Figure 8, the running time of HETPART1 is faster than that of HOMPART-ML, HOMPART-SM, HOMPART-FF (averaged over all instance groups). More precisely, HETPART1 is 1.4 times faster than HOMPART-FF, 1.6 times faster than HOMPART-SM and 3.98 times faster than HOMPART-ML. (Note that this comparison does not consider the three largest matrix trees due to their excessive running time.) As expected, HOMPART-ML, the variant that provides comparably good makespan values, also takes the longest to execute. Our experiments indicate that most time is spent merging. Smaller memory sizes as in HOMPART-ML produce trees that require extensive merging, explaining the longer running time.

HETPART2 *A+B compared to* HETPART1.

The relative running times of HETPART2 A+B in comparison to HETPART1 are shown in Figure 9. For all categories except four, HETPART2 is faster than HETPART1. For "6: large f_i", "7: large w_i", "9: small m_i, w_i, f_i"; however, the (relative) running time is on average 128%, 116.5%, and 122%, respectively. For the other categories, the running times of HETPART1 and HETPART2 are comparable: 97% on "1: sparse matrix trees", 105% on '2: random' and 75% on "4: largest fanout", and "5: large m_i" (all percentages refer to HETPART2 in relation to HETPART1).

Considering that we execute all possible swaps, the running time investment is acceptable. This effect mostly stems from search space reduction: while the partitioning operates on the tree, the swaps are performed between the much fewer cluster nodes. Finding best combinations of swaps does not necessarily depend on the size of a tree. This also explains the notably big variations in the running times. The upper outliers are small trees (2000 vertices) with a 3× increase for the worst instances. The lower outliers are big trees with 30 000 to 50 000 vertices—for them, HETPART2 takes less than 20% of the HETPART1 running time.

HETPART2 *B compared to* HETPART2 *A*

Of further interest is the running time of the optional Phase B, compared to Phase A. The results are shown in Figure 10. On most categories, Phase B is up to 20% faster or comparable to Phase A. This indicates that the perturbation and later reswapping leads to reasonably fast convergence.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

FIGURE 8 Runtime increase in % (lower is better) compared to HETPART1 by three homogeneous scenarios: upper - HOMPART-FF, middle- HOMPART-SM, lower- HOMPART-ML. Instances: 1: sparse matrix trees, 2: random, 3: large fanout, 4: largest fanout, 5: large m_i, 6: large w_i, 7: large f_i, 8: large m_i, w_i, f_i, and 9: small m_i, w_i, f_i.

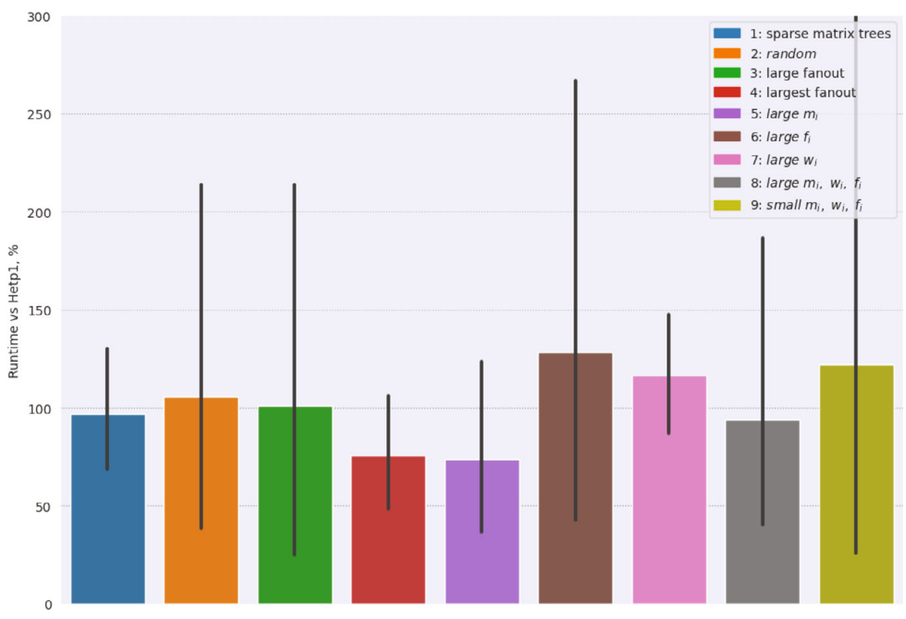

FIGURE 9 Running time of HETPART2 relative to the baseline (100%).

Only on "1: sparse matrix trees" Phase B is slower than Phase A (by 11.7%) and has a higher variation. This may be due to longer makespan computation times on large trees that make even a few iterations of reswapping slower.

Absolute running times

While we deem the comparison between the heuristics most important, we also report absolute running times for a complete picture, see Table 3. In the first two lines, we show geometric means of all tree running times, per tree category. In the next lines, we show the performance of HETPART1 and HETPART2 on smallest and biggest trees, respectively. For the small trees, absolute running times are around one second or less for HETPART1 and fluctuate between one and almost three seconds for HETPART2. On these trees, HETPART1 continuously outperforms HETPART2, especially on categories "6: large f_i", "7: large w_i", and "9: small m_i, w_i, f_i". This is because the complexity of the tree itself is relatively low (due to

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

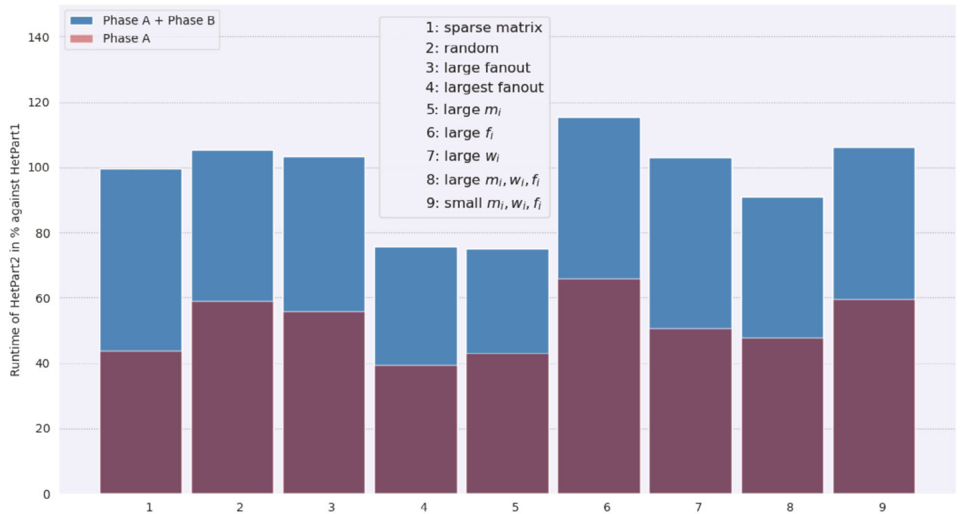

FIGURE 10 Running time of HETPART2 B relative to the baseline HETPART2 A (100%).

TABLE 3 Absolute runtimes, average over all, smallest and biggest trees, in seconds.

Algorithm	Tree size	1: sparse matrix trees	2: random	3: large fanout	4: largest fanout	5: large m_i	6: large w_i	7: large f_i	8: large m_i, w_i, f_i	9: small m_i, w_i, f_i
HETPART1	All	13.5	23.5	26.4	4.3	27.2	26.5	1.4	23.2	34.5
HETPART2	All	8.9	8.0	5.2	1.9	8.1	9.0	1.1	8.3	8.0
HETPART1	Small 10	1.4	1.1	1.2	0.2	0.9	1.2	0.1	1.1	1.9
HETPART2	Small 10	1.35	2.04	1.5	0.21	1.7	2.4	0.33	1.8	2.9
HETPART1	Big 10	984.7	305.2	407.3	430.4	448.8	238.4	120.4	305.1	462.1
HETPART2	Big 10	189.2	30.0	15.7	10.2	32.9	34.6	14.1	30.9	31.3

the small number of vertices), but the complexity of HETPART2 is bound to the size of the cluster. This insight is also reflected in the absolute running times of the biggest trees. For all generated tree categories, the running times lie in hundreds of seconds, while the runtimes of HETPART2 lie in dozens of seconds. The growth of the absolute running time of HETPART2 between the smallest and the biggest tree is mostly due to the costlier makespan computations in the bigger trees. Sparse matrix trees are somewhat special in that its biggest trees are very big and produce running times close to 1000 seconds. Makespan computations in these trees are also very costly, so that also HETPART2 takes around 200 seconds on average.

In general, it seems to make sense to omit the phase HETPART2 B on small trees (2000 vertices in our experiments). Given the very short running times of HETPART1 on these trees and comparably high running times of HETPART2, an additional 10% makespan improvement may not make a big difference for a small tree. The larger the tree, however, the bigger is the impact of Phase B of HETPART2. At the same time, HETPART2 as a whole takes only a small part of the overall running time in case of large trees (50,000 vertices). It makes sense for the user to wait just a little longer to achieve an additional makespan improvement.

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 and processor speed. Taking heterogeneity into account when partitioning these trees into subtrees pays off: our new memory-heterogeneous heuristic HETPART1 clearly improves the makespan compared to the homogeneous state of the art and is also faster. 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. HETPART2, a two-phase local search heuristic, which is aware of different memory sizes *and* different processor speeds, improves upon HETPART1 regarding makespan even more than HETPART1 on HOMPART–with an average decrease of makespan of nearly 60% in

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

case of Phases A and B. When using only Phase A, the makespans become worse by 3% – 16% compared to using both phases, but the running time is roughly cut in half as well.

Future works include the further increase of the heterogeneity level by adding different bandwidths. Overall, we expect similar findings: when the compute platform is sufficiently heterogeneous, a heuristic taking this heterogeneity into account should pay off. Furthermore, it would be interesting to further validate the results with real experimentations on parallel sparse matrix factorizations, even though we believe that the current study already demonstrates the potential of the proposed algorithms. Finally, as current and future applications may show variability and uncertainty, an interesting research direction would be to consider dynamic allocation strategies that adapt to the workload.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments and suggestions, which helped to improve the paper. Open Access funding enabled and organized by Projekt DEAL.

FUNDING INFORMATION

This work is partially supported by Collaborative Research Center (CRC) 1404 FONDA—Foundations of Workflows for Large-Scale Scientific Data Analysis, which is funded by German Research Foundation (DFG).

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in HU-Box at https://box.hu-berlin.de/d/e01d496a8a18429e849b/.

ORCID

Svetlana Kulagina https://orcid.org/0000-0002-2108-9425
Henning Meyerhenke https://orcid.org/0000-0002-7769-726X

REFERENCES

1. Kulagina S, Meyerhenke H, Benoit A. Mapping tree-shaped workflows on memory-heterogeneous architectures. *Proceedings of HeretoPar'22: Conjunction with EuroPar'22*, Glasgow, Scotland. Springer; 2022.
2. Liu JWH. The role of elimination trees in sparse factorization. *SIAM J Matrix Anal Appl*. 1990;11(1):134-172.
3. Gou C, Benoit A, Marchal L. Partitioning tree-shaped task graphs for distributed platforms with limited memory. *IEEE Trans Parallel Dist Syst*. 2020;31(7):1533-1544.
4. He S, Wu J, Wei B, Wu J. Task tree partition and subtree allocation for heterogeneous multiprocessors. *2021 IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computation & Networking (ISPA/BDCloud/SocialCom/SustainCom), IEEE*; 2021:571-577.
5. Sinnen O. *Task Scheduling for Parallel Systems*. Vol 60. John Wiley & Sons; 2007.
6. Benoit A, Rehn-Sonigo V, Robert Y. Multi-criteria scheduling of pipeline workflows. *2007 IEEE International Conference on Cluster Computing. IEEE*; 2007:515-524.
7. Benoit A, Marchal L, Pineau JF, Robert Y, Vivien F. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *IEEE Trans Comput*. 2009;59(2):202-217.
8. Adhikari M, Amgoth T, Srirama SN. A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Comput Surv*. 2019;52(4):1-36.
9. Liu J, Pacitti E, Valduriez P. A survey of scheduling frameworks in big data systems. *Int J Cloud Comput*. 2018;7:103-128.
10. Davis TA. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. Society for Industry and Applied Mathematics; 2006.
11. 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 Trans Comput*. 2021;1696-1710.
12. Aupy G, Benoit A, Renaud-Goud P, Robert Y. Energy-aware algorithms for task graph scheduling, replica placement and checkpoint strategies. *Handbook on Data Centers*. Springer; 2015:37-80.
13. Jacquelin M, Marchal L, Robert Y, Uçar B. On optimal tree traversals for sparse matrix factorization. *2011 IEEE International Parallel & Distributed Processing Symposium. IEEE Computer Society*; 2011:556-567.
14. Eyraud-Dubois L, Marchal L, Sinnen O, Vivien F. Parallel scheduling of task trees with limited memory. *ACM Trans Parallel Comput*. 2015;2(2):13.
15. Herrmann J, Kho J, Uçar B, Kaya K, Çatalyürek ÜV. Acyclic partitioning of large directed acyclic graphs. *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE; 2017:371-380.
16. Alam M, Khan A, Varshney AK. A review of dynamic scheduling algorithms for homogeneous and heterogeneous systems. *Syst Arch*. 2018;73-83.
17. da Silva EC, Gabriel PH. A comprehensive review of evolutionary algorithms for multiprocessor DAG scheduling. *Computation*. 2020;8(2):26.
18. Benoit A, Hakem M, Robert Y. Contention awareness and fault tolerant scheduling for precedence constrained tasks in heterogeneous systems. *Parallel Comput*. 2009;23:171-187.
19. Bader J, Lehmann F, Thamsen L, Will J, Leser U, Kao O. Lotaru: Locally estimating runtimes of scientific workflow tasks in heterogeneous clusters. *Proceedings of the 34th International Conference on Scientific and Statistical Database Management (SSDBM). Association for Computing Machinery*; 2022.
20. Bittencourt LF, Goldman A, Madeira ER, Fonseca dNL, Sakellariou R. Scheduling in distributed systems: a cloud computing perspective. *Comput Sci Rev*. 2018;30:31-54.
21. Buluç A, Meyerhenke H, Safro I, Sanders P, Schulz C. *Recent advances in graph partitioning*. Springer; 2016:117-158.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

22. Çatalyürek UV, Devine KD, Faraj MF, et al. More recent advances in (hyper) graph partitioning. *ACM Comput Surv*. 2022;55(12):1-38.
23. Garey MR, Johnson DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Series of Books in the Mathematical Sciences. 1st ed. W. H. Freeman; 1979.
24. Feldmann AE, Foschini L. Balanced partitions of trees and applications. *Algorithmica*. 2015;71(2):354-376.
25. Kernighan BW, Lin S. An efficient heuristic procedure for partitioning graphs. *Bell Syst Tech J*. 1970;49(2):291-307.
26. Talbi EG. *Metaheuristics: From Design to Implementation.* John Wiley & Sons; 2009.
27. Prüfer H. Neuer Beweis eines Satzes über Permutationen. *Archiv Math Phys*. 1918;27:142-144.

How to cite this article: Kulagina S, Meyerhenke H, Benoit A. Mapping tree-shaped workflows on systems with different memory sizes and processor speeds. *Concurrency Computat Pract Exper*. 2023;e7842. doi: 10.1002/cpe.7842

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7842 by Bibliothèque Diderot De Lyon, Wiley Online Library on [07/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License