Mapping Tree-shaped Workflows on Memory-heterogeneous Architectures

Svetlana Kulagina, Henning Meyerhenke, Anne Benoit

Mapping Tree-shaped Workflows on Memory-heterogeneous Architectures*

Svetlana Kulagina[1], Henning Meyerhenke[1], and Anne Benoit[2]

[1] Department of Computer Science, Humboldt-Universität zu Berlin, Germany
{kulagins,meyerhenke}@hu-berlin.de
[2] LIP laboratory, ENS Lyon, France
Anne.Benoit@ens-lyon.fr

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. Since typical workflows are modeled by huge trees, it is crucial to schedule them efficiently. 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. In particular, we design a novel algorithm to assign subtrees to processors with different memory sizes, and we show how to select appropriate processors when splitting or merging subtrees. The experiments demonstrate that exploiting the heterogeneity reduces the makespan significantly compared to the state of the art for homogeneous memories.

Keywords: Tree partitioning · Mapping · Heterogeneous memory

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 the class of rooted directed trees, which we consider in this paper. These tree-shaped workflows occur in a variety of scientific applications, most notably as elimination trees for sparse matrix factorizations [10,13,17] or in computational physics [15].

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

* 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). Corresponding author: Svetlana Kulagina.

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: (i) partition the tree into subtrees, minimizing the makespan while not taking the memory limit into account, (ii) further partition subtrees too big for the memory limit, and finally (iii) ensure that the number of subtrees is less than or equal to the number of processors. Yet, more and more compute clusters are heterogeneous, *i. e.*, have variable memory sizes. This can happen due to hardware updates, a combination of clusters, or an intentional configuration with fat and light nodes. 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 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 – for *heterogeneous tree partitioning*) for tree-shaped workflows that exploits memory heterogeneity (*i. e.*, different memory sizes). Our 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, the selection of processors considers their memory size. For the experiments (Section 5), we choose the homogeneous state-of-the-art algorithm by Gou *et al.* [10] as standard of reference with different resource consumption scenarios. Our experimental results show that HetPart reduces the makespan by better exploiting the heterogeneous memories. The average improvement is 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 does not produce a valid solution for more than 20% of the instances. Details omitted due to space constraints can be found in the companion research report [14].

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) [1,16], 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 [7,17].

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.* [11] has attempted to extend this approach to heterogeneous architectures. Their work leaves several important questions open, though: (i) the experiments seem to be on a system with homogeneous memories only, and (ii) the code is not available, but 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

Application model. We consider workflows that come in the form of rooted trees $\tau = (V, E)$, as in [13,10]. 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]. If $(v_j, v_i) \in E$ (i.e., $v_j \to v_i$), then 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 (see [13] for a very similar way of modeling a workflow).

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

Platform model. The target environment is a cluster consisting of a finite number l of processing units (processors), 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 [13]. 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., $M_u \geq f_i + m_i + \sum_{j:(v_i,v_j)\in E} f_j$. 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 $\frac{f_j}{\beta}$.

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 the subtree with root v_i. The processor handling this subtree τ_ℓ with root v_i is denoted by $proc(i)$; it is a processor p_u that 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. Yet, 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 can now 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 $desc(i) = \{j \mid v_j \notin tasks(i) \wedge (v_k, v_j) \in E \wedge v_k \in tasks(i)\}$ be the indices of tasks that are not in

the subtree rooted in v_i, but that have a parent in this subtree 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).

Problem statement and its 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 [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.

Minimizing makespan. In the first step, we split the tree into a number of subtrees with the aim to minimize the overall makespan. Neither the memory constraint nor the number of resulting trees is the focus of this step. The splitting continues as long as a better makespan can be achieved. Several heuristics are designed for this case in [10] (also see Section 5.1 for details).

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 the report [14] as Algorithm 1), which additionally considers the memory size of each processor. We use a max-priority queue Q to keep the current set of subtrees, S, "ordered" according to their memory consumption. Moreover, we sort the processors by memory size (from largest to smallest) in a dynamic array M. Then, in a while loop that terminates if Q or M become empty, we iteratively fit the currently largest subtree s into the processor with currently biggest memory m. This is done using any memory fitting algorithm (referred to as MemFit in the pseudocode); 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 while increasing the makespan as little as possible. The result is a subtree that fits within m (denoted as S_{fitted}, which is never empty but possibly equal to s), 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} fits within memory m, and we assign it to the corresponding processor, which is then removed from the array of available processors. If all processors have been assigned a subtree but there still remain some subtrees in Q, we take care of these subtrees in a second while loop that terminates when Q is empty. In the loop, we further split the subtrees with the memory m of the smallest processor as a threshold. All these subtrees are left unassigned and will be merged in the next step below.

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.

Decreasing the number of subtrees. Should the previous step yield more trees than there are processors, some trees need to be merged. To this end, we propose the HeterMerge heuristic (Algorithm 1). We first construct the quotient tree T of τ, where each subtree in τ becomes a vertex in T; there is an edge between two vertices $u \to v$ in T iff there is an edge from the corresponding subtree τ_u to τ_v in τ. The general idea now is similar to [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 [14]). If at

Algorithm 1 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

least one of the subtrees has been assigned already, then we select the processor with smallest memory that is able to hold the merged subtree. 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. 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 1, 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 1 breaks in Line 20.

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.

5 Experimental Evaluation

We now describe the experimental settings and a representative subset of the results. Additional results can be found in the companion research report [14]. All results have been obtained via a simulation of the target cluster platforms.

5.1 Experimental setup

All algorithms are implemented in C++ and compiled with g++ (v.11.2.0) with flags "-O2 -fopenmp". The code can be downloaded at this link: `https://box.hu-berlin.de/d/fe55a68653c74809b14d/` with password "het-sched". The baseline algorithm from [10], which we call HomPart, is also written in C++; it is compiled and executed with the same infrastructure.

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, see [14] for detailed parameters. We build eight categories, each containing 30 trees ranging in size from 2K to 50K nodes. The categories differ in the problem parameters (m_i, f_i, w_i) and the fanout, i.e., average number of children per node. By default, the fanout comes from a Prüfer sequence. For the "random" category, m_i, f_i, w_i all result

from a uniform random distribution. For "large f_i, w_i, m_i", the expected values of these weights are all multiplied by 100, while for "small f_i, w_i, m_i", they are divided by 10. The categories "large m_i", "large w_i", and "large f_i" increase only one of these respective values. Finally, "large 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. A detailed description of each tree category is given in [14].

Compute platforms. For evaluation purposes, we create synthetic compute platforms that resemble heterogeneous real-world configurations. To make the algorithms' job difficult, we use a modest 4-fold cluster with a total of 36 nodes of four different kinds (9 nodes of each kind): "extra-light" nodes with memory $D_{\max}/2$, "light" nodes with a memory $D_{\max}$, "moderate" nodes with memory $1.5D_{\max}$, and "fat" nodes with memory $3D_{\max}$. 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 $D_{\max}$. All processor speeds and bandwidths are assumed equal (normalized to 1 for speeds and to 500 for bandwidths).

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 (see [14]). The configurations of HomPart are suffixed with ML (*many light*, uses 27 nodes as "light"), SM (*some moderate*, uses 18 nodes as "moderate"), or FF (*few fat*, uses only the 9 "fat" nodes). Note that the memory would not suffice for the largest tasks if we took all 36 nodes and treated them as "extra-light".

We selected the best combinations of different heuristics in each phase (regarding solution quality, on average) for our setup, both for HetPart and for HomPart, in order to be as fair as possible (details in [14]). In the following, we use the combinations that respectively returned the best results. Detailed results supporting this claim can be found in the companion research report [14].

5.2 Results

We first 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. Lower values indicate a better quality.

Makespan. Figure 1 displays the average increase of the makespan (in %) of the three HomPart scenarios compared to HetPart. Each bar represents

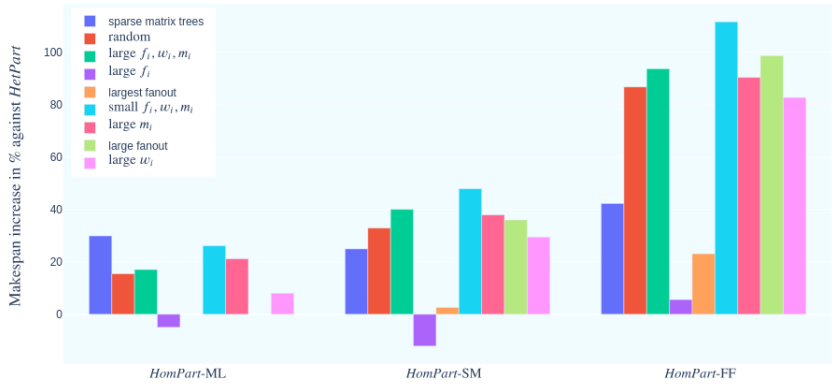

Fig. 1. Makespan increase of HomPart in % compared to HetPart in different cluster configurations (higher means that HetPart is better). The two missing bars for HomPart-ML indicate unsuccessful runs (no solution for HomPart in this setting).

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 (with fixed fanout). 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 f_i" and "largest fanout", HomPart-SM performs quite well – if it is able to find a solution.

In the following, we take a look at the respective instance groups. On sparse matrix trees, 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 random trees, HetPart improves by at least 15.4% (against HomPart-ML). The other two homogeneous variants perform significantly worse.

For the categories where only weights change (and not the tree topology – "large m_i, f_i, w_i" and "small m_i, f_i & w_i"), the improvement of HetPart compared to HomPart-ML is significant (17.1% and 26.1% respectively). Similar results can be observed with "large m_i": 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 "largest fanout" and "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. As indicated before, on trees with fixed fanouts ("large fanout, "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 solu-

tions, but they are much worse than those of HetPart. Trees with large w_i fall in 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 Figure 1), but it also leads to 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 "large fanout", no tree could be solved. For "largest fanout", half of the trees were unsolved. The other categories have two to five unsolved trees out of 30, except for the matrix trees, where all trees could be solved.

Comparison with a 2-fold cluster. We performed further experiments with a more homogeneous cluster with only "fat" and "light" nodes (18 nodes of each kind), and compared the results between the two clusters. Detailed results are available in [14], they are summarized below. With more heterogeneity to exploit in the 4-fold cluster, HetPart is able to provide a more tangible improvement. In the 2-fold cluster, HetPart wins by much smaller margins (2%-13%) and loses in 3 categories ("large f_i", "largest fanout", large "w_i"). For the sparse matrix trees (real-world instances), HetPart provides tangible improvements in both clusters (24.9% and 20.7%).

Running times. Here again, a summary is presented while detailed results are available in the companion research report [14]. 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 HomPart-FF. At the same time, as we saw above, HetPart provides a much better solution quality. The homogeneous scenario with the 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)
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 Int Conf 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 Trans. on Par. 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 Trans on Par and Dist 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 Par Distr Processing with Applic, Big Data, Cloud Comp, Sustainable Comp, Communications, Social Comp, Networking (ISPA/BDCloud/SocialCom/SustainCom). pp. 571–577 (2021)
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/hal-03581418`
15. Lam, C., Rauber, T., Baumgartner, G., Cociorva, D., Sadayappan, P.: Memory-optimal evaluation of expression trees involving large objects. Comput. Lang. Syst. Struct. **37**(2), 63–75 (2011), `https://doi.org/10.1016/j.cl.2010.09.003`
16. Liu, J., Pacitti, E., Valduriez, P.: A survey of scheduling frameworks in big data systems. International Journal of Cloud Computing **7** (01 2018)
17. Liu, J.W.H.: The role of elimination trees in sparse factorization. SIAM Journal on Matrix Analysis and Applications **11**(1), 134–172 (1990)