Mapping Large Memory-constrained Workflows onto Heterogeneous Platforms

Svetlana Kulagina, Henning Meyerhenke, Anne Benoit

Mapping Large Memory-constrained Workflows onto Heterogeneous Platforms*

Svetlana Kulagina
Humboldt Universitaet zu Berlin
Berlin, Germany
svetlana.kulagina@hu-berlin.de

Henning Meyerhenke
Humboldt Universitaet zu Berlin
Berlin, Germany
meyerhenke@hu-berlin.de

Anne Benoit
ENS Lyon and IUF
Lyon, France
Anne.Benoit@ens-lyon.fr

ABSTRACT

Scientific workflows are often represented as directed acyclic graphs (DAGs), where vertices correspond to tasks and edges represent the dependencies between them. Since these graphs are often large in both the number of tasks and their resource requirements, it is important to schedule them efficiently on parallel or distributed compute systems. Typically, each task requires a certain amount of memory to be executed and needs to communicate data to its successor tasks. The goal is thus to execute the workflow as fast as possible (i.e., to minimize its makespan) while satisfying the memory constraints.

Hence, we investigate the partitioning and mapping of DAG-shaped workflows onto heterogeneous platforms where each processor can have a different speed and a different memory size. We first propose a baseline algorithm in the absence of existing memory-aware solutions. As our main contribution, we then present a four-step heuristic. Its first step is to partition the input DAG into smaller blocks with an existing DAG partitioner. The next two steps adapt the resulting blocks of the DAG to fit the processor memories and optimize for the overall makespan by further splitting and merging these blocks. Finally, we use local search via block swaps to further improve the makespan. Our experimental evaluation on real-world and simulated workflows with up to 30,000 tasks shows that exploiting the heterogeneity with the four-step heuristic reduces the makespan by a factor of 2.44 on average (even more on large workflows), compared to the baseline that ignores heterogeneity.

KEYWORDS

Workflow mapping, DAG partitioning and scheduling, memory constraints, heterogeneous platforms.

ACM Reference Format:
Svetlana Kulagina, Henning Meyerhenke, and Anne Benoit. 2024. Mapping Large Memory-constrained Workflows onto Heterogeneous Platforms. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3673038.3673068

*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).

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673068

1 INTRODUCTION

Many large scientific applications, *e. g.*, for data analysis, are nowadays often comprised of different software components expressed as tasks within a workflow [21]. Examples for such tasks can be data acquisition, data cleansing, various data analysis tasks, and visualization of the results [27] – executed on each data set. Such workflows are typically represented as directed acyclic graphs (DAGs); the vertices in the DAG represent the task and the edges express (input/output) dependencies between them [1, 23]. Such workflows can be very large as well as memory- and compute-intensive, so that they exhaust the resources of a single machine and need to be executed on a parallel or distributed system.

For an efficient execution, the workflow should be mapped to the system appropriately; one must decide which task is executed on which processor. This requires to partition the workflow into smaller sub-DAGs and to assign each sub-DAG to one processor. When doing so, a common objective function to minimize is the makespan, *i. e.*, the total running time of the workflow [26]. Since parallel and distributed systems such as (networks of) compute clusters are often heterogeneous in terms of memory size and/or processor speed, these different properties of the individual compute nodes should be taken into account for a high-quality mapping. This leads to the following optimization problem: given a workflow in form of a weighted DAG $G = (V, E)$ and a compute system $\mathcal{S}$ with k processors, compute an acyclic k'-way partition of G, with $k' \leq k$, and a mapping onto $\mathcal{S}$, such that the execution of each sub-DAG on its assigned processor does not exceed the memory of the processor, and the makespan is minimized. This is an NP-hard problem, even with no memory constraints and a homogeneous platform [13], and hence large workflows are handled by heuristics in practice. While there certainly are heterogeneity-aware (*e. g.*, [2]) and memory-aware (*e. g.*, [4]) techniques, we do not know of algorithms nor tools that *simultaneously* handle large DAGs, minimize the makespan, and respect memory constraints on the tasks.

Contributions. In this paper, we investigate the memory-constrained partitioning and mapping of DAG-shaped workflows onto heterogeneous platforms, where each processor can have a different memory size and a different processor speed. To this end, in the absence of heterogeneity-aware tools that also adhere to the memory constraints, we first propose a baseline algorithm that produces valid solutions, building on a memory-efficient traversal of the DAG. As our main contribution, we then present a partitioning-based heuristic. To deal with the difficulty of the problem, it is divided into four steps: (i) partition the input DAG into smaller blocks with a DAG partitioner, (ii) adapt the blocks of the DAG to fit the processor memories, (iii) optimize for the overall makespan by further

splitting and merging workflow blocks, and (iv) block swaps to further improve the makespan. An extensive experimental evaluation on real-world and simulated workflows shows that (i) the heuristic scales to big workflows (we use up to 30 000 tasks), taking less than 11 minutes on average for them, and (ii) exploiting the heterogeneity in the cluster with the four-step heuristic reduces the makespan by a factor 2.44 on average compared to the baseline that only takes memory sizes into account. The improvement even reaches a factor of nearly 5 for big workflows and a large cluster configuration.

Outline. We survey related work in Section 2. Section 3 presents the model and formally introduces the optimization problem tackled in this paper. Heuristics are described in Section 4, and both the experimental setting and detailed results are presented in Section 5. We conclude and provide directions for future work in Section 6.

2 RELATED WORK

The problem of executing a collection of tasks on various types of computing platforms has received long-standing research attention [26]. With the increasing number of scientific workflows that need to be executed multiple times, this problem has gained renewed interest. Currently, the most common way of representing a workflow is as a directed acyclic graph (DAG) [1, 23], hence having the most general representation of dependence constraints.

Large-scale workflows are to be executed on parallel or distributed platforms, and the mapping problem consists in deciding where to execute each part of the workflow, guided by a target objective function. Usually, the goal is to execute the entire workflow as fast as possible, hence to minimize the *makespan*, which models the total execution time [7, 20, 26].

However, while performance is the objective, one must at the same time account for the constraints given by the execution platform. In particular, since memory and I/O become a bottleneck [11, 17], one must make sure that each processor's local memory is large enough to execute the part of the workflow that has been mapped on it. For shared-memory platforms (we assume distributed memory instead), Bathie *et al.* [4] propose, among others, a dynamic memory-aware DAG scheduling heuristic based on an ILP formulation. Their optimization objective is cut-based and the overall approach is limited to rather small workflows.

Scheduling for heterogeneous platforms has been investigated in the past already [2, 5, 12], of course. In particular, it has been remarked that the consideration of heterogeneity is a crucial aspect in scheduling algorithms for cloud platforms [6]. Astonishingly, despite their obvious importance for fault-free workflow execution, memory constraints are mostly ignored in the literature and also play no significant role in recent surveys [1, 22]. For scheduling tree-shaped workflows (a special case of the DAGs considered here) with memory constraints, we showed in [19] that taking different memory sizes and processor speeds into account can improve the makespan considerably. More precisely, we extended a partitioning-based heuristic that maps trees on homogeneous platforms [14] by making it heterogeneity-aware. A similar goal is pursued by He *et al.* [15] with their tree mapping algorithm.

When moving from tree-shaped workflows to the case of general DAGs, the relatively simple partitioning approaches above do not work any longer. That is why we employ DAG partitioning, for which several tools exist, among them DAGP [16]. Techniques for partitioning general undirected graphs are numerous [8], but in many cases not easily transferable to the DAG case. Note that partitioning into (nearly-)balanced (and in case of DAGs: acyclic) blocks is $\mathcal{NP}$-hard both for general graphs [13] and for DAGs [24].

We are not the first to employ DAG partitioning for scheduling. The problem of memory minimization for series-parallel DAGs has been tackled by Kayaaslan *et al.* [18]. The authors propose an algorithm that fits parts of the workflow into memories of adequate size, but they do not aim at makespan optimization. Also, Özkaya *et al.* [25] built a scheduler based on their partitioner DAGP. This scheduler uses the DAGP output and then employs a list-based scheduling approach to optimize for makespan. However, this scheduler does not take memory constraints into account, and thus does not produce valid solutions for our target problem in general.

3 MODEL

We first describe the target applications, which are (large scientific) workflows, in Section 3.1. Next, we define the execution environment, a heterogeneous system (in terms of processor speed and memory size), in Section 3.2. We provide details about the optimization objective, which is to minimize the makespan, in Section 3.3. Finally, we are ready to express the optimization problem in Section 3.4. Table 1 summarizes the main notation.

3.1 Workflow

A workflow is modeled as a directed acyclic graph $G = (V, E)$, where V is the set of vertices (tasks), and E is a set of directed edges of the form $e = (u, v)$, with $u, v \in V$, expressing precedence constraints between tasks. Each task $u \in V$ is performing w_u operations, and it also requires some amount of memory to be executed, denoted as m_u. Each edge $e = (u, v)$ has a cost $c_{u,v}$ that corresponds to the size of the (logical) output files written by task u and used as input by task v.

Symbol	Meaning
$G = (V, E)$	Workflow graph, set of vertices (tasks) and edges
Π_u, C_u	Parents of a task u, children of a task u
m_u	Memory weight of task u
w_u	Normalized execution time of task u (makespan weight)
$c_{u,v}$	Communication volume along the edge $(u, v) \in E$
F, $\mathcal{F}$	A partitioning function and the partition it creates
V_i	Block number i
$\mathcal{S}, k$	Computing system and its number of processors
p_j, proc(V_i)	Processor number j, processor of block V_i
M_j, s_j	Memory size and speed of processor p_j
β	Bandwidth in the compute system
l_u	Bottom weight of task u
μ_G, μ_i	Makespan of the entire workflow G and of a block V_i
$\Gamma = (\mathcal{V}, \mathcal{E})$	Quotient graph, its vertices and its edges
r_u, r_{V_i}	Memory requirement of task u and of block V_i
$\mathcal{P}$	Critical path in a workflow

Table 1: Notation.

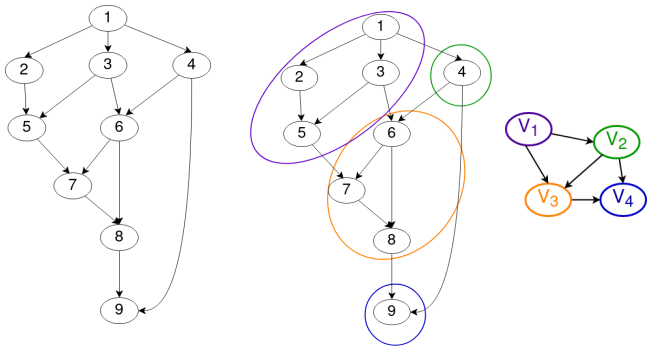

Figure 1: An example graph G, its possible acyclic partition $\mathcal{F}$ into four blocks, and a resulting quotient graph Γ.

Hence, the task memory requirement for the execution of a task u consists of the input files (size of the files to be received from the parents), the output files (size of the files to be sent to the children), and the memory size m_u:

$$r_u = \sum_{(v,u)\in E} c_{v,u} + \sum_{(u,v)\in E} c_{u,v} + m_u.$$

The parents of a task $u \in V$ are the directly preceding tasks that must be completed before u can be started, i.e., the set of parents is $\Pi_u = \{v \in V : (v,u) \in E\}$. A task without parents is called a *source task*. The children tasks of u are the tasks following u directly according to the precedence constraints, i.e., $C_u = \{v \in V : (u,v) \in E\}$. A task without children is called a *target task*. Each task may have multiple parents and children. Fig. 1 (left) shows an example DAG consisting of nine tasks, with one source task (1) and one target task (9). The parents of task 6 are 3 and 4, its children tasks 7 and 8.

3.2 Execution environment

The goal is to execute the workflow on a heterogeneous system, denoted as $\mathcal{S}$, which consists of k processors $p_1, \ldots, p_k$. Each processor p_j ($1 \le j \le k$) has an individual memory size M_j and a speed s_j. The execution time of a single task $u \in V$ on a processor p_j is expressed as $\frac{w_u}{s_j}$. We assume that all processors are connected with the same bandwidth β.

In general, more than just one task can be mapped on the same processor; hence we aim at partitioning the tasks into blocks, where each block will be mapped on a distinct processor. A *partitioning function* is a function $F : V \to \mathbb{N}$ that assigns each task a block number. The i-th block, denoted as V_i, contains all tasks that have been assigned number i: $V_i = \{u \in V : F(u) = i\}$. $\mathcal{F}$ is the partition, *i. e.*, the set of blocks that F creates. In Fig. 1 (middle), the partition $\mathcal{F}$, shown as colored circles, divides the graph into four blocks. If block $V_i \in \mathcal{F}$ has been assigned to a processor $p_j \in \mathcal{S}$, we say that $p_j = proc(V_i)$. Furthermore, the maximum memory requirement of a block V_i is r_{V_i}, and it depends on the order in which the block, which is a DAG itself, is executed. We use the MEMDAG algorithm from [18] to compute this memory requirement r_{V_i}, by transforming block V_i into a series-parallel graph, and then finding the traversal that leads to the minimum memory consumption.

Note that the number of blocks has to be at most k, the number of available processors. Some processors might be left unused, hence the DAG can be partitioned into $k' \le k$ blocks.

3.3 Makespan computation

In order to compute the makespan, given a DAG G and a partitioning function F, we consider the quotient graph $\Gamma = (\mathcal{V}, \mathcal{E})$ induced by the partitioning function F. Each quotient graph vertex $v_i \in \mathcal{V}$ corresponds to a block V_i of the *original* DAG G, as illustrated in Fig. 1 (right). The vertex weights of v_i ($v_i \in \mathcal{V}$) are defined as $w_{v_i} = \sum_{u\in V_i} w_u$ (total vertex weight in block V_i), while edge weights $c_{v_i,v_j} = \sum_{u\in V_i, v\in V_j} c_{u,v}$ are the sum of weights of all the edges between the two corresponding blocks. For instance, with unitary tasks and edge weights in Fig. 1, $w_{v_1} = 4$, $w_{v_2} = 1$, $w_{v_3} = 3$, and $w_{v_4} = 1$. Furthermore, the edge costs would be 1, except for $c_{v_1,v_3} = 2$. In this example, Γ is a DAG. However, note that if tasks 4 and 9 had been merged in a same block, the resulting quotient graph would have been cyclic, due to the edges $(4,6)$ and $(8,9)$. We restrict to partitions that lead to acyclic quotient graphs, otherwise it is not possible to orchestrate computations and the makespan is not defined, as both processors rely on the result of the other processor. Then, given an acyclic quotient graph Γ, the makespan computation builds on the notion of *bottom weight*. The bottom weight $\underline{l}_v$ of a node $v \in \mathcal{V}$ is defined as:

$$\underline{l}_v = \begin{cases} \frac{w_v}{s_v} & \text{if } C_v = \emptyset, \\ \frac{w_v}{s_v} + \max_{v'\in C_v} \left\{ \frac{c_{v,v'}}{\beta} + \underline{l}_{v'} \right\} & \text{otherwise,} \end{cases} \quad (1)$$

where C_v is the set of children of node v, and s_u is the speed of the processor that node v has been assigned to. If node v has not been assigned to any processor, then the speed is assumed to be 1. In the example, with unitary speeds and bandwidth, we have $\underline{l}_{v_4} = 1$ (no children), then $\underline{l}_{v_3} = 3 + 1 + 1 = 5$, $\underline{l}_{v_2} = 1 + \max\{1+5, 1+1\} = 7$, and finally $\underline{l}_{v_1} = 4 + \max\{1+7, 2+5\} = 12$.

We can now define the *makespan* of a graph G via its corresponding quotient graph Γ, assuming that Γ is a DAG. The makespan of Γ equals the maximum of the bottom weights of its tasks:

$$\mu(\Gamma) = \max_{v\in\mathcal{V}} \underline{l}_v. \quad (2)$$

In case of a single source, this maximum is achieved on the source task. Note that in a quotient DAG Γ, when the assignment to processors changes, the makespan weights and the critical path change as well (in general).

An unpartitioned DAG G corresponds to a quotient DAG Γ with only one task v_1, and it is always executed on a single processor, p_j. Therefore, its makespan is the sum of makespan weights of its vertices divided by this processor's speed: $\mu_G = \sum_{v\in V} \frac{w_v}{s_j}$. Because all generated files reside in the memory of the same processor, communication costs can be neglected.

The makespan is completely defined once we have a mapping of the workflow onto a computing system $\mathcal{S}$, and hence we can account for processor speeds. If the mapping is not complete yet, as mentioned in the bottom weight definition, a speed of 1 is assumed, so that the bottom weights can still be computed with this assumption, leading to an *estimated makespan*. If some workflow

tasks have already been assigned, the corresponding speed is used for these tasks.

Finally, note that in the definition, the finishing time of block V_i is equal to the finishing time of all the tasks within this block, since we sum up all task weights before doing the communication and moving up to the next block. In reality, some tasks may finish before the block finishes, and their successors could start earlier, but we do not consider this possibility, hence providing in fact an overestimation of the makespan.

3.4 Problem formulation

We formulate the DAGP-PM problem (**DAG P**artitioning with **P**roportionate blocks and minimized **M**akespan).

DAGP-PM problem. Let $\mathcal{S} = p_1, \ldots, p_k$ be the execution environment consisting of k processors. Each processor p_j (for $1 \leq j \leq k$) has an individual memory size M_j. Given a DAG $G = (V, E)$, find an acyclic k'-way partitioning function F (one that results in a quotient DAG) with $k' \leq k$, its corresponding partition $\{V_1, \ldots, V_{k'}\}$ of V, and a mapping of blocks V_i to processors such that

$$proc(V_i) \neq proc(V_j), \quad 1 \leq i \neq j \leq k'$$
$$r_{V_i} \leq M_i, \quad \text{where } p_i = proc(V_i)$$

and the induced makespan is minimized.

In Fig. 1, with $k = 4$, the problem would require each vertex in the quotient graph to be placed on a separate processor in such a way that the overall makespan is minimized.

Note that this DAGP-PM problem is (when formulated as decision problem) NP-complete, even without memory constraints and precedence constraints. Indeed, the problem of scheduling independent tasks with different execution times, even on a homogeneous platform, is well-known to be NP-complete (by reduction from 2-partition or 3-partition [7]). We are facing a complex problem, and the core contribution is to design a heuristic solution method, DagHetPart, which accounts for the heterogeneity of the target platform while respecting the memory constraints.

4 HEURISTICS

Before describing the DagHetPart heuristic accounting for platform heterogeneity (Section 4.2), we start in Section 4.1 with a baseline heuristic, building on related work, that aims at returning a valid mapping of the DAG. The goal is to adhere to the memory constraints, since no existing (makespan-oriented) mapping algorithm, to the best of our knowledge, is addressing these constraints and is able to return a valid solution to the DAGP-PM problem.

4.1 Memory-aware baseline – DagHetMem

The baseline heuristic DagHetMem is directly building upon the memDag algorithm [18]. memDag computes a memory-optimal traversal of the workflow graph G that minimizes the peak memory consumption. It however does not ensure that this traversal fits into one or multiple actual memories of the processors. It also does not compute the resulting makespan of the traversal, and does not try to optimize it.

DagHetMem first sorts the processors by decreasing memory sizes. Thus, if the peak memory is lower than the largest memory available, the whole DAG can be mapped on a single processor as a valid mapping. Usually, the entire workflow does not fit into the memory of a single processor; in this case, we perform the traversal returned by memDag and add tasks in the first part (block V_1) as long as the memory requirement of the current (largest) processor is not exceeded. The update to the memory requirement is done each time a new task is traversed, by adding its memory requirement, subtracting the memory requirement of the task executed beforehand, and updating edge weights that should remain in memory.

If the traversal of a given task $u \in V$ leads to a block whose memory requirement is exceeding the memory of the processor, we remove u from the block. We start by creating the next block on the next available processor, starting again from a memory of zero, and resuming the traversal from task u that initiates the new block.

Hence, the heuristic proceeds iteratively, following the order of the initial traversal returned by memDag, and creating a new part as necessary. It ends when all tasks have been traversed and assigned. Note that for some problem instances, if the platform does not have enough memory, this heuristic may not return any solution if there are some remaining tasks but no more processors available. This means that the workflow needs a larger platform to be executed.

Once the blocks have been generated, the quotient graph can be computed, based on the part numbers. Note that parts are not necessarily connected graphs, since we follow the traversal order. But given the quotient graph, it is possible to compute the makespan achieved by this mapping, as explained in Section 3.3. This baseline algorithm is not trying to optimize the makespan, but it produces a valid solution that respects the memory constraints if it is successful. It does not exploit the parallelism in the DAG, in particular not for workflows that fit on a single processor memory-wise.

Hence, we focus in the following on the design of a heuristic for the DAGP-PM problem, aiming at minimizing the makespan while respecting memory constraints.

Note that DagHetMem always computes the traversal of the entire workflow graph, which may take a lot of time. DagHetPart avoids this by partitioning the workflow graph first.

4.2 Partitioning-based heuristic – DagHetPart

The DagHetPart heuristic is going beyond DagHetMem and aims at minimizing the makespan. Since there are many constraints, the key idea is to decompose the heuristic into several steps; each step builds upon the previous one to improve the solution and adapt it to the heterogeneity of the computing system.

High-level overview. In the first step, we use a previously introduced DAG partitioning algorithm to obtain the initial blocks that will be used by our heuristic (hence its name DagHetPart, **DAG**-shaped workflow **HET**erogeneous environment **PART**itioning-based). This step ignores all the heterogeneity (memory size, processor speed) of the computing system and optimizes for the smallest edge cut. In the second step, we create a preliminary assignment of these first blocks to processors, changing their size if necessary to fit them into the memories of given processors. This partial solution has to be valid for the assigned blocks, but there may still be unassigned blocks left. This step respects the memory heterogeneity, but ignores the different processor speeds. The third step uses the processor speeds to optimize the makespan. Furthermore, it aims at

respecting the number of processors, by merging blocks if there are more blocks than processors, until it obtains a valid assignment of blocks to processors. Here, the algorithm may not be able to find a valid assignment for at least one block, if the platform does not have enough resources for the DAG. In that case, no valid assignment is returned. Finally, the fourth step starts with a valid assignment from Step 3 and tries to improve it by local search. Two blocks are swapped (between two processors) if the operation is possible memory-wise and if it brings the biggest improvement in makespan among all other possible swaps. If there are free processors, we also try to move blocks to one of these processors if it improves the makespan. In the worst case, no improvement over Step 3 is achieved. We now detail each of the four steps.

Step 1: Partitioning. We use the edge-cut-optimizing acyclic graph partitioner DAGP [16] for the first step. Its input is a DAG (our workflow G) and a number of blocks to partition it into. Computing an initial partition into k blocks with DAGP returns an array of block numbers. For instance, with $n = 5$ tasks and $k = 2$ processors, we may obtain a result [1,1,1,2,2], meaning that tasks 1,2 and 3 are in the same block V_1, and tasks 4 and 5 are in another block V_2. Note that we tentatively partition the DAG into k' blocks, with $1 \leq k' \leq k$, and compute the makespan returned by the heuristic for all values of k'. The best result is kept.

Step 2: Assignment. We start with the partitioning function obtained from Step 1, and we insert the resulting blocks into a priority queue Q – with the required block memory size as priority. The processors, in turn, are inserted into a queue M in descending order w. r. t. their available memory size (see Algorithm 1). Then, we assign the largest block to the free processor with the largest memory as long as there are remaining blocks and processors (while loop starting in Line 6). During this assignment (algorithm FITBLOCK), it may happen that the block does not fit on the processor, *i. e.*, its memory requirement exceeds the memory available on the processor. In this case the block is further partitioned, and the resulting sub-blocks are inserted back into the queue Q. Hence, the size of Q can become bigger than that of M. If this happens, the unassigned blocks are further partitioned (loop of Line 15) to the size of the memory of the smallest processor in the computing system – by another call to FITBLOCK that will not perform any assignment.

The subroutine FITBLOCK (Algorithm 2) works as follows. Its input is the block V_i to be fitted, the priority queue of blocks, the processor where the block should be scheduled on, and a flag `doMap` that indicates that we really want to place the block on this processor – rather than just partitioning it down to the processor's memory size. If a block V_i fits on the desired processor and if the flag is true, the block is mapped on its target processor and the algorithm returns the block that was placed. Otherwise, we further partition this block using the same partitioner as in Step 1.

Although we strive for dividing the block into two new blocks, sometimes DAGP cannot do that due to balancing constraints. Hence, PARTITION may generate more than two blocks. We then insert the resulting blocks back into the queue Q and a NULL result is returned, indicating that no block has been placed on the processor. Because Q is a priority queue, it maintains the correct order of the blocks in terms of their memory requirement. Otherwise, a relatively small

Algorithm 1 – Step 2: BIGGESTASSIGN

```
1:  procedure BiggestAssign(F, S)
2:                ▷ Input: initial partition F, computing system S.
3:      Init PQ Q with F;      ▷ max-priority queue of the blocks
4:      Init queue M with S;
5:                ▷ a queue of processors sorted by memory size
6:      while not Q.empty() and not M.empty() do
7:          V_m ← Q.extractMax(); p_m ← M.head();
8:          V_r ←FitBlock(V_m, Q, p_m, true);
9:                  ▷ Fit the block while actually trying to map it
10:         if V_r ≠ NULL then M.remove(p_m);
11:                     ▷ Remove the processor that is now busy
12:         end if
13:     end while
14:     p_min ← arg min M(S); ▷ processor with smallest memory
15:     while not Q.empty() do        ▷ No more processors, but
    remaining partition blocks
16:         V_m ← Q.extractMax();
17:         FitBlock(V_m, Q, p_min, false);
18:          ▷ Fit the block without actually putting it on the proc.
19:     end while
20: end procedure
```

Algorithm 2 – Step 2 subroutine: FITBLOCK

```
1:  procedure FitBlock(V_i, Q, p_j, doMap) ▷ Input: a block V_i that
    needs to be fitted, priority queue Q of other blocks, available
    proc. p_j, a flag doMap that indicates that we want to map the
    resulting subtree on the processor
2:      if r_{V_i} ≤ M_j then
3:          if doMap then
4:              proc(V_i) ← p_j;
5:              return V_i;
6:          end if
7:      else
8:          (V_{m_1}, V_{m_2}, ...) ← Partition(V_m, 2);
9:          for V_k ∈ {V_{i_1}, V_{i_2}, ...} do
10:             Q.add(V_k);              ▷ Reinsert resulting blocks
11:         end for
12:     end if
13:     return NULL;
14: end procedure
```

block obtained during repartitioning could block a bigger waiting block from being assigned.

Step 2 returns at least a valid partial assignment of blocks to processors, but not necessarily a complete one: all assigned blocks have to fit, but some blocks may not be assigned yet.

Step 3: Merging. The focus of this step lies in the makespan, affected by heterogeneous processor speeds. We operate on the quotient graph Γ that is built from the original graph G using the blocks from Step 2 and their (partial) assignment to processors. According to the definition of the quotient graph, each block V_i of the original graph becomes a vertex in Γ. Since the assignment in Step 2 may have been incomplete, some of these vertices are

assigned to processors, while some may not. We try to improve the makespan by merging unassigned vertices in Γ to the assigned ones in a way that impairs the makespan the least. Because there are still unassigned vertices, we operate with the *estimated makespan*.

We first observe that merging two unassigned vertices in Γ will not help our cause: such a merge results in a bigger, still unassigned vertex that will be more difficult to place on a processor. Therefore, we merge an unassigned vertex with an assigned one.

The next observation is that the makespan is constrained by the critical path in Γ. Indeed, the critical path is the one that dictates the makespan when computing the bottom weights (see Eq. (1)). Merging more vertices to one that lies on the critical path will increase the makespan, while merging outside of the critical path will only affect the makespan if the merge changes the critical path. Hence, we prefer merging an unassigned vertex outside of the critical path to a vertex that is not on the critical path, but we still allow merges on the critical path if no other solution is possible. In particular, if a node on the critical path is not assigned and considered for merging, it can be interesting to merge it to its parent or children, thus saving communication on the critical path.

First, let us consider an algorithm that finds the best possible merge for a single quotient vertex v (Algorithm 3). It takes the vertex v in question, a list of candidates that can be used for merging A, and the graph Γ as the input. We can merge v to one of either its parents or its children, but only those that are in A (Line 4). For each such vertex v', we tentatively merge it with v, an action that results in the creation of a new vertex v_m, and changes Γ to Γ' (Line 5). In the bad case, Γ' contains cycles after this action, as in Fig. 2 if a and b are merged. In the general case, introducing a cycle into the graph is a reason to discard the tentative merge. However, there is one easily solvable case, illustrated by Fig. 2. In this case, the merge creates a cycle of length 2. This situation can (often) be resolved by merging the third vertex into the set.

This is exactly what we do if we encounter a cycle of length 2 (Line 7): we tentatively merge the already merged vertex v_m with the second vertex in the cycle (Line 8). If the graph still has cycles after that, then we discard this merge: we restore v_m to its original state and continue with another merge candidate for v. Otherwise, we store the third vertex as v_o, an optional third vertex, and operate on the graph Γ'' that contains this triple-merged vertex.

If Γ' remains cycle-free or if a single 2-vertex cycle has been eliminated, we assess the quality of this merge (Line 15). The memory requirement of v_m must not exceed the memory available on the processor of v', the merge candidate. We compute the makespan of the modified Γ' (Line 16). If this is the best makespan we have encountered yet, we store it and the vertex that allowed it (Line 17) and proceed to try out the next merge. We return the best makespan found, the vertex that yielded it, and the optional third vertex.

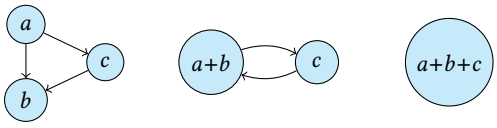

Figure 2: Merging two vertices can create a cycle of length 2. Merging all three vertices may solve the problem.

Algorithm 3 – Step 3 subroutine: FindMSOptMerge

```
1:  procedure FindMSOptMerge(v, A, Γ)      ▷ A block v (vertex in Γ) that needs to be merged, a set A of vertices that can be used for the merge, the graph Γ.
2:      μ_min ← +∞; v_min ← NULL; v_o ← NULL;
3:                          ▷ v_o is an optional third vertex in the merge
4:      for each v' ∈ (Π_v ∪ C_v) ∩ A do
5:          {v_m, Γ'} ← Merge(v,v');
6:          if isCyclic(Γ') then
7:              if Cycle(Γ').size=2 then
8:                  {v_m, Γ''} ← Merge(v_m,Cycle(Γ')[1]);
9:                  if isCyclic(Γ'') then Unmerge(v_m); break;
10:                 else  v_o ← Cycle(Γ)[1]; Γ' ← Γ'';
11:                 end if
12:             else  Unmerge(v_m); break;
13:             end if
14:         end if
15:         if r_{v_m} ≤ M_{proc(v')} then
16:             μ ← makespan(Γ');      ▷ Assess the impact of this merge on the makespan of the entire graph
17:             if μ ≤ μ_min then
18:                 μ_min ← μ; v_min ← v';
19:                 Γ ← Γ';
20:             end if
21:         end if
22:         Unmerge(v_m);
23:     end for
24:     return (μ_min, v_min, v_o);
25: end procedure
```

With Algorithm 3 described, we can now formulate the merging algorithm used in Step 3 (Algorithm 4). We first construct a quotient DAG Γ (Line 2) and compute its critical path $\mathcal{P}$. Each vertex in Γ additionally has a counter z (number of times it was seen) that is set to 0 at the beginning. Then, we build the set A of all assigned tasks (Line 4). We iterate over all unassigned blocks (Line 6). When doing so, we first try to merge the unassigned block to an assigned one that is not on the critical path (Line 8). If a feasible merge has been found this way, we execute it in Line 10, then assign the resulting vertex to the processor, and recompute the critical path. Otherwise, we repeat the same attempt, but now we try to merge to any assigned task – no matter if it is on the critical path or not (Line 15). If this merge succeeds, we execute it in the same fashion. If no merge can be found even on the critical path, then we consider the possibility of trying to merge this vertex at a later point in time. If a vertex has unassigned parents or children, it may become mergeable later, when these parents or children have themselves been assigned (Line 22). In this case, we reinsert the vertex into the set of unassigned vertices to be checked later. However, to avoid a situation where two vertices are being constantly reinserted after each other, we use the counter $v.c$. We increase it on each reinsert and stop reinserting after 2 times. Finally, we return the quotient graph (with the assigned tasks).

Step 4: Swaps. We perform local search via block swaps to further improve the makespan. We first identify all feasible swaps (pairs of vertices that fit into the respective memory of the processor

Algorithm 4 – Step 3: MergeUnassignedToAssigned

procedure MergeUnassignedToAssigned($\{V_1, \ldots, V_k\}$, P)
▷ Input: Blocks $V_1, \ldots, V_k$, processors P.
  $\Gamma \leftarrow$ QuotientDAG($G, \{V_1, \ldots, V_k\}$); ▷ Quotient DAG from the blocks of G
  $\mathcal{P} \leftarrow$ CriticalPath(Γ);
  $A \leftarrow \{v \in \mathcal{V} : proc(v) \neq NULL\}$; ▷ Assigned vertices
  $U \leftarrow \{v \in V(\Gamma) : proc(v) = NULL\}$; ▷ Unassigned vertices
  for $v \in U$ **do**
    $U \leftarrow U \setminus v$; $\mu_{min} \leftarrow \infty$; $v_{min} \leftarrow NULL$;
    $\mu_{min}, v_{min}, v_o \leftarrow$ FindMSOptMerge(v,$A \setminus \mathcal{P}$,Γ);
    if $v_{min} \neq NULL$ **then** ▷ Found a feasible merge
      $\{v_m, \Gamma'\} \leftarrow$ Merge(v,v_{min},v_o);
      $\Gamma \leftarrow \Gamma'$; A.remove(v_{min}); A.remove(v_o);
      proc(v_m) $\leftarrow$ proc(v_{min});
      $\mathcal{P} \leftarrow$ CriticalPath(Γ); ▷ Critical path could have changed after the merge.
    else ▷ No feasible merge in $A \setminus \mathcal{P}$
      $\mu_{min}, v_{min}, v_o \leftarrow$ FindMSOptMerge(v,A,Γ); ▷ Look for a feasible merge anywhere in Γ
      if $v_{min} \neq NULL$ **then** ▷ Found a feasible merge
        $\{v_m, \Gamma'\} \leftarrow$ Merge(v,v_{min},v_o);
        $\Gamma \leftarrow \Gamma'$; A.remove(v_{min}); A.remove(v_o);
        proc(v_m) $\leftarrow$ proc(v_{min});
        $\mathcal{P} \leftarrow$ CriticalPath(Γ);
      else
        if ($C_v \cap U \neq \emptyset$ **or** $\Pi_v \cap U \neq \emptyset$) **and** $v.c \leq 1$ **then** $U \leftarrow U \cup \{v\}$; $v.c$++; ▷ Unassigned parents or children
        end if
      end if
      return false; ▷ No solution could be found
    end if
  end for
  return Γ;
end procedure

Algorithm 5 – Step 4: Swap (swap best pair until no feasible improving swap exists)

function Swap(Γ, $\mathcal{S}$) ▷ Quotient DAG Γ whose tasks $\mathcal{V}$ have been assigned to processors, computing system $\mathcal{S}$
  $best \leftarrow$ pair(Γ, makespan(Γ));
  while true **do**
    $curr \leftarrow best$;
    for all pairs (v, v'') of $\mathcal{V} \times \mathcal{V}$ **do**
      if isSwapFeasible($best.first$,v,v') **then**
        $\Gamma' \leftarrow$ swap($best.first$, (v, v'));
        $next \leftarrow$ pair(Γ', makespan(Γ'));
        if $next.second < curr.second$ **then**
          $curr \leftarrow next$; ▷ better solution is stored
        end if
      end if
    end for
    if $curr.second < best.second$ **then**
      $best \leftarrow curr$; $\Gamma \leftarrow best.first$;
▷ execute improving swap and work further with the resulting quotient graph
    else return $best$;
      ▷ stop because no improving swap exists
    end if
  end while
end function

of the other vertex). The best swap is stored and executed. We stop when no more makespan-improving swaps can be made (see Algorithm 5).

Furthermore, it may happen, in particular with small workflows, that the partitioner does not create enough blocks and some of the processors remain idle. In that case, some of these idle processors may be faster than those the blocks were assigned to. For this reason, we add one final step that is only activated if there are idle processors after swapping. In this step, we go through the critical path of Γ and try to move each task to a faster idle processor that can hold it memory-wise. If such a processor exists, we move the task there and recompute the critical path. We do this as long as there are tasks in the critical path that have not been moved.

5 EXPERIMENTAL EVALUATION

5.1 Setup

All algorithms are implemented in C++ and compiled with g++ (v.11.2.0). DagHetMem, proposed in Section 4.1 as a variant of the existing memDag, functions as baseline algorithm – because there is no previous work in our scenario that respects the memory constraints. Exceeding the memory would yield invalid and thus incomparable mappings. The experiments are executed on workstations with 192 GB RAM and 2x 12-Core Intel Xeon 6126 @3.2 GHz and CentOS 8 as OS. Code, input data, and experiment scripts are available for review at https://zenodo.org/records/10992671.

We first describe the set of workflows used in the evaluation and then the clusters on which the workflows are mapped.

5.1.1 Workflow instances. The input data for the experiments consists of two sets of workflows: real-world workflows obtained from [10] and workflows obtained by simulating real-world workflows with the WFGen generator [9]. First, we discuss how the graph topology is generated, then we focus on the weights associated to tasks and edges.

Workflow graphs. For real-world workflows, their nextflow definition (see [10]) was downloaded from the respective repository and transformed into .dot format using the nextflow option "-with-dag". The resulting DAG contains many pseudo-tasks that are only internal representations in nextflow (and not actual tasks); that is why we removed them.

For the simulated workflows, the graph is produced by the WFGen generator, based on a *model workflow* and the desired number of tasks. The model workflows are described on the WFGen website, and we used the following ones: 1000Genome, BLAST, BWA, Epigenomics, Montage, Seismology, and SoyKB. Other models could not be generated without errors. As number of tasks, we use: 200, 1 000, 2 000, 4 000, 8 000, 10 000, 15 000, 18 000, 20 000, 25 000, 30 000. We divide the workflows into three groups by size: small ones with up to 8 000 tasks, middle ones with 10 000 to 18 000 tasks, and big ones

with 20 000 to 30 000 tasks. Note that for some workflow models, such as SoyKB or Montage, only a subset of the workflow sizes could be generated, because the workflow generator either took a disproportionally long time or yielded errors.

Overall, this yields four workflow types, denoted by real, small, mid (middle-sized), and big.

Generation of task and edge weights. For the real-world workflows, we use historical data files provided by Bader *et al.* [3]. The columns in these files are measured Linux PS stats, acquired during an execution of a nextflow workflow. Each row corresponds to an execution of one task on one cluster node. Since the operating system cannot distinguish between (a) the RAM the task uses for itself and (b) the RAM it uses to store files that were sent or received from other tasks, the values in the historical data are task memory requirements (input/output files plus memory consumption of the computation). In a similar manner, the historical data provided by [3] do not store the actual weights of edges between tasks, but only the overall size of files that the task sends to all its children.

For each task, historical data can contain multiple values, obtained from the runs with different input sizes. To avoid underestimation, we take the maximum among all values from different runs on the same cluster node. Not all tasks have historical runtime data stored in the tables. In fact, for two workflows, Bader *et al.* do not provide data for more than 50% of the tasks. For two more, around 40% of tasks have no historical runtime data stored. Hence, in the absence of historical data about a task, we give it a weight of 1. Because the historical data contains absolute measured values and the cluster node information is a relative value, we normalize all values extracted from the historical data by the smallest one. This way, task memory weights fit into the memory of the cluster nodes. Additionally, this way, the tasks without historical data receive less insignificant values compared to tasks with historical data.

For the simulated workflows, we generate random execution times and memory weights for each task as well as edge weights for each precedence constraint. We generate uniformly distributed values between 1 and 10 for edge weights, 1 and 1000 for the workloads, and 1 and 192 for memory weights. When doing so, we try to mimic the weights observed in the historical data, hence *e. g.* the low lower bounds for the workloads.

5.1.2 Target computing systems. To fully benefit from the historical data, the *default* experimental environment that we consider is a cluster based on the same six kinds of real-world machines that were used in the experimental evaluation in [3]. We set the number of each kind of node to six, thus having 36 processors in total.

Each machine has a (normalized) CPU speed and a memory size (in GB), and we list them as (name, speed, memory): (*local*, 4, 16) – very slow machines; ($A1$, 32, 32), ($A2$, 6, 64), ($N1$, 12, 16) – average machines; ($N2$, 8, 8) – machine with very small memory; and ($C2$, 32, 192) – *luxury* machine with high speed and large memory (see Table 2).

Note that, by nature, memory sizes are normalized values, relative to each other, too. Therefore, we additionally normalize memory weights of real-world workflows to the maximum size of 192 to make sure they fit. For simulated workflows, we increase memory sizes proportionally until the task with the biggest memory requirement still has a processor it could be executed on.

Processor name	CPU speed (GHz)	Memory size (GB)
local	4	16
A1	32	32
A2	6	64
N1	12	16
N2	8	8
C2	32	192

Table 2: Cluster configuration (default).

To test various settings, we also vary the cluster configuration and consider variants of the default cluster presented above:

- *Small* and *large* clusters. While the default cluster consists of 36 nodes (6 of each kind), we consider a small cluster with three processors of each kind (18 processors in total), and a large cluster with ten processors of each kind (60 processors in total).
- More and less heterogeneous clusters (*MoreHet*, *LessHet*, and *NoHet*). In addition to the default cluster, we also consider more and less heterogeneous clusters (Table 3). For a cluster with more heterogeneity, the smaller half of memories is made twice smaller, whereas the bigger half is made twice bigger. The same holds for processor speeds. For the less heterogeneous cluster, we reverse the procedure: smaller values are increased two-fold, while bigger values are reduced. An exception is made for the biggest memory size: we leave it at 192 to make sure that the largest memory requirements of tasks can still be met. In the homogeneous cluster *NoHet*, each processor has to be able to hold even the most memory-demanding task. Therefore, in this case, all processors have to be $C2$, the processor with the biggest memory capacity.
- Finally, it is interesting to examine the impact of the communication-to-computation ratio (*CCR*) by modifying the bandwidth β (also used in the makespan computation). To explore different scenarios, we vary β between 0.1 and 5.

MoreHet	Speed	Memory	*LessHet*	Speed	Memory
local*	2	8	local'	8	64
A1*	64	64	A1'	16	64
A2*	3	128	A2'	12	128
N1*	24	8	N1'	12	64
N2*	4	4	N2'	16	32
C2*	64	384	C2'	16	192

Table 3: Clusters with more or less heterogeneity. Memory size is in GB, speed is in GHz.

5.2 Results

We first present results on the default cluster for all types of workflows. We then study the impact of other cluster configurations, report running times, and summarize the results.

5.2.1 Default cluster. Fig. 3 (left) shows the relative makespan of DagHetPart (ratio of makespans by DagHetPart and DagHetMem, in %), by workflow types (geometric mean over the ratios of

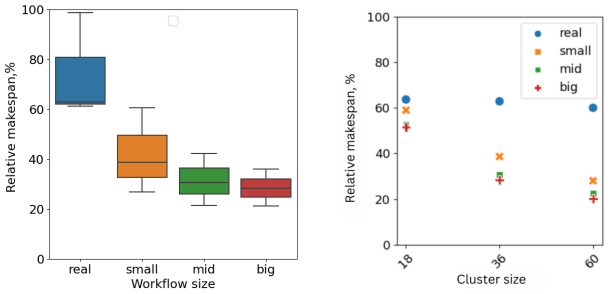

Figure 3: Left: Relative makespan (in %) of DagHetPart compared to DagHetMem on default cluster. Right: Relative makespan (in %) on different cluster sizes (x-axis: number of CPUs), by workflow size. Smaller is better.

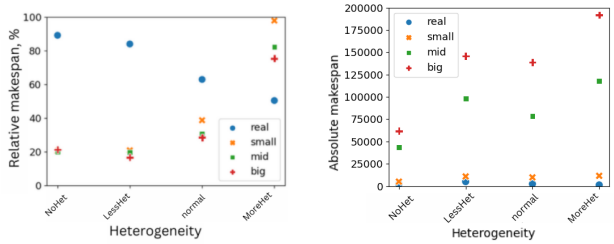

Figure 4: Relative (left, baseline: DagHetMem) and absolute (right) makespan of DagHetPart for different levels of heterogeneity. Smaller is better.

each workflow); hence the lower, the better. When averaging over all workflow types, the relative makespan is 41%; hence DagHetPart yields mappings that are on average 2.44× better in makespan than the baseline. The trend is that DagHetPart can achieve the best improvement on workflows of the categories big (3.3× better over all workflows, 4.82× better on the most fanned-out families), and middle (3.23× better over all, 4.84× on fanned-out). Yet, even on real-world workflows (which are small, the smallest one has only 11 tasks), DagHetPart is still 1.59× better.

On the default cluster, DagHetPart can schedule 13 workflows out of 14 big ones and 31 out of 32 small ones. The baseline was also unable to schedule the small workflow, but scheduled the big one. All middle-sized and all real-world workflows can be scheduled successfully by both DagHetPart and DagHetMem.

It is more challenging with real-world workflows to improve on DagHetMem due to several reasons. The first one is their small size, ranging from only 11 to 58 tasks. As a consequence, the partitioner is unable to decompose these workflows into the desired number of blocks (36, as the number of nodes in the cluster); instead, the workflows are split into 11 to 14 blocks in the first step of DagHetPart. This already severely limits the algorithm's ability to exploit both the parallelism and the heterogeneity. On top of that, more than half of the tasks in real-world workflows have no historical data. This leads to a situation where relatively few tasks have actual values and a long "tail" of tasks has values of 1. It makes sense to schedule these tiny tasks together on one machine, which also limits the ability to use different machines in the cluster.

5.2.2 Impact of parallelism – small and large clusters. Exploiting parallelism in the workflows is the main reason for DagHetPart's success. Being able to utilize even the slower/smaller (memory-wise) processors yields higher improvements on clusters with more nodes: DagHetPart yields on large workflows makespans that are 4.96× better. Fig. 3 (right) illustrates this trend. The trend is less obvious for real-world workflows, because only one real-world workflow is big enough to benefit from the large cluster. The remaining four do not occupy even all nodes in the default cluster, so that adding new ones has little effect on the makespan.

Moreover, on the large cluster, we are able to schedule all workflows from all types, while no solutions were found for two workflows on the default cluster. On the small cluster with 18 nodes, we are able to schedule 13 out of 14 big workflows (DagHetMem: 13), 13 out of 17 middle-sized workflows (DagHetMem: 15), and 30 out of 32 small workflows (DagHetMem: 31). The algorithms did not succeed on some configurations where there are not enough resources for the workflow. It is non-trivial to figure out whether a solution would be achievable, other than doing an exhaustive search. We believe that the user should rather consider using a larger platform in these cases.

5.2.3 Impact of heterogeneity – NoHet, MoreHet, and LessHet clusters. Fig. 4 (left) illustrates the impact of the heterogeneity level on the relative makespan. Somewhat unintuitively, with more heterogeneity, the relative makespans grow, i.e., DagHetPart wins against the baseline DagHetMem by a smaller margin, except for real-world workflows. The absolute makespan values of DagHetPart have a similar tendency to grow with more heterogeneity (Fig. 4, right). Note that this is not because DagHetPart works better in a less heterogeneous environment. The reason lies in the cluster configuration: clusters with less heterogeneity still have to be able to fit even the most memory-demanding task. Hence, they still have the *luxury* processors $C2$, as well as others that are modified to be more similar to them. In case of no heterogeneity, all processors are $C2$. All the parts that were earlier processed by slow processors are now being processed by $C2$ processors or similar ones. In case of more heterogeneity, on the other hand, the rest of the cluster is weaker in comparison. The parts that are assigned to weaker processors take longer to execute.

On top of that, the baseline receives a benefit in our more heterogeneous clusters when choosing the processors, since it starts with the processor with the largest memory. In our cluster with more heterogeneity, this processor has not only more memory in relation to the others (compared to the normal scenario), but also the highest speed. The strategy of DagHetMem to map big parts on these first processors pays off more than in the default cluster, so that it becomes harder for DagHetPart to improve this baseline. Still, regardless of the level of heterogeneity and the type of workflow, DagHetPart improves over the baseline in all cases.

As already mentioned, real-world workflows show a different trend from the generated ones. The relative makespan further improves with an increased level of heterogeneity. Here, the small size of the workflows plays a positive role: the workflows are mapped exclusively on the fastest nodes, so when these nodes are stronger, then DagHetPart, a heuristic that is able to better utilize parallelism, wins. If the size of the workflow requires the use of weaker nodes, then the benefit provided by very fast nodes becomes less pronounced for DagHetPart.

5.2.4 Impact of computational demands – Workflows with four times bigger w_u. To explore the impact of higher computational demands, we evaluate workflows that deviate only in the workloads of their tasks w_u. Thus, for the generated workflows, we assign four times larger workloads; for the real-world ones, we multiply the historical values by four. Relative makespans in both cases are virtually identical. For example, for real-world workflows, DagHetPart produces makespans that are 62.8% of those produced by DagHetMem (1.59× better). On real-world workflows with increased computational demands, DagHetPart produces an average makespan of 61.73% (1.62× better). Small, middle-sized and big workflows yield average makespans of 36.4%, 30.9%, and 30.3% on high-demand workflows, respectively, while normal-demand workflows yield 38.6%, 30.63%, and 28.4%. This means that a symmetrical increase in computational demands of all tasks has little impact on DagHetPart's quality.

5.2.5 Behavior of workflow types when scaling workflow size. Previously, we presented geometric means of makespans among workflows in a certain size category (*e. g.* big or small). Yet, each category consists of workflows of different kinds, depending on the model workflow that was used for their generation. For example, the "small" category contains the "Epigenomics" workflow with 2 000 tasks, while "big" contains the scaled one with 30 000 tasks. Hence, in this section we present unaggregated results from DagHetPart on each workflow family, varied by workflow size.

Fig. 5 shows, for each workflow type, the relative makespans in relation to the size. The solution quality of DagHetPart varies here across different families of workflows. There are workflows that are consistently easy for DagHetPart to map, *e. g.* Seismology, BWA and BLAST. These are workflows with the highest degrees and fanout. There are workflow families where DagHetPart performs better with increasing size of the workflow, like Genome and Soykb (note that for families like Soykb or Montage, not all workflow sizes are present). Especially on small Soykb workflows, the improvement achieved by DagHetPart is smaller than usual (less than 20%). Soykb starts with a chain of tasks and ends with a fork-join segment. With growing size, however, there is more parallelism to be utilized, and hence DagHetPart is able to improve more over DagHetMem.

Absolute makespans per workflow family are presented in Fig. 6. For all families except two, the makespans seem to grow in a roughly linear manner. For the two difficult families soykb and epigenomics, the makespans seem to grow faster. Yet, considering that the relative makespans for these families fall slightly with the increasing workflow size, the superlinear growth of absolute makespans is apparently a feature of the workflow, not of DagHetPart's quality.

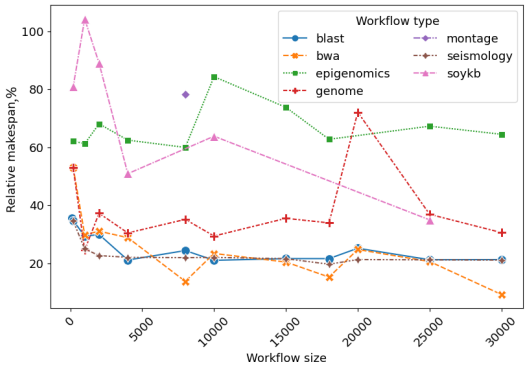

Figure 5: Makespan of DagHetPart relative to DagHetMem for different workflow families. Dotted lines are meant to improve readability.

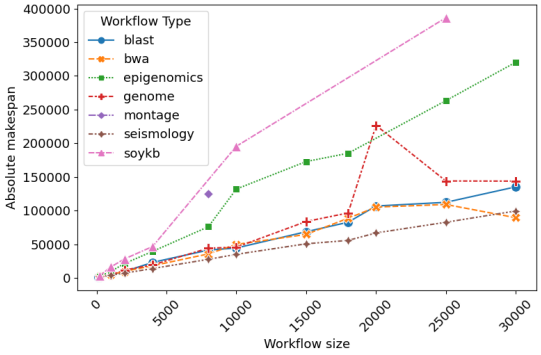

Figure 6: Absolute makespan of DagHetPart for different families of workflows. Dotted lines are meant to improve readability.

5.2.6 Impact of communication-to-computation ratio (CCR). Fig. 7 shows the effect of different cluster bandwidths on the relative performance of DagHetPart compared to the baseline. The general trend shows that higher bandwidths allow DagHetPart to better exploit heterogeneity and produce better makespans. In particular, on small workflows, the relative makespan decreases by 13 percentage points from the smallest bandwidth to the largest one. For the other two groups, the effect is smaller: for big workflows, the difference between the worst and the best relative makespan is only 5.3 percentage points. Real-world workflows prove resistant to changes in bandwidth, because they occupy fewer nodes and the communication costs matter less.

Blocks are based on the partition provided by the dagP algorithm, which optimizes for edge cut. Even though we merge and repartition the blocks further, this effect of low communication costs seems to persist. This explains the relatively small reaction of DagHetPart makespan on varying bandwidths.

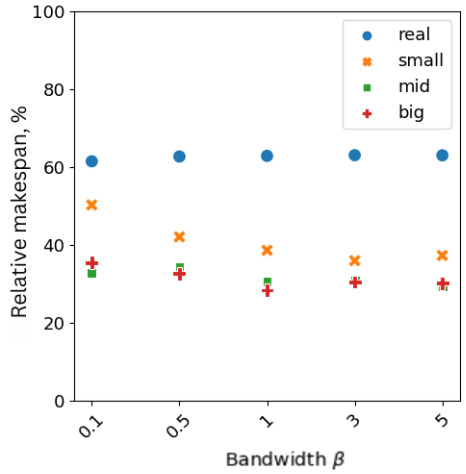

Figure 7: Relative makespan of DagHetPart compared to DagHetMem on default cluster, in percentage, as a function of bandwidth. Smaller is better.

However, different families of workflows react differently to changes in the bandwidth. On the two most fanned-out families (BWA and Blast), DagHetPart provides 3.14 times smaller makespans on small workflows in a cluster with the biggest bandwidth in comparison to the smallest bandwidth. On the two least fanned-out families (Soykb and Epigenomics), the makespans in the same conditions are only 1.27 times smaller than those on the smallest bandwidths. For the middle-sized workflows, the result for the most fanned-out families 3.31× better, while the least fanned-out ones yield 1.22× smaller makespans on the biggest bandwidths. On big workflows with the biggest bandwidth, the two most fanned-out families lead to makespans that are 3.3× better compared to the results for the smallest bandwidth. The two least fanned-out ones, in turn, only lead to makespans that are 1.4× better on average in this comparison.

5.2.7 Running times. Both algorithms need on real-world workflows less than one second on average. That is why it is of no concern that DagHetMem is much faster in this category. Also, the small workflows can be handled in a few seconds by both heuristics, and DagHetPart is only 63% slower than DagHetMem. On middle-sized workflows, both algorithms are nearly equally fast and require less than three minutes on average. DagHetPart is even faster than DagHetMem on big workflows; on average, it can map them in less than 11 minutes and thus scales even to big workflows in very reasonable time.

The running time of a mapping/scheduling algorithm has to be seen in the context of the running time improvement of the resulting schedule. For real-world workflows, the geometric mean of the makespan improvement corresponds to 5.3 minutes, while the average absolute scheduling running time is around 0.5 seconds. The makespan improvement can correspond to up to 97.8 hours for large workflows (methylseq). For synthetic workflows (small/middle/big), scheduling time is on average 2.68/166/647 *seconds*; the resulting makespan improvements correspond to 10.9/41.8/84.3 *hours*. On all categories of workflows, this is a significant gain and a good reason to use an advanced mapping/scheduling method.

In fact, the running time of DagHetMem is dominated by the effort to compute the optimal memory traversal over the entire workflow. With increasing workflow size, this procedure becomes more and more time-consuming. DagHetPart computes optimal memory traversals only on the blocks that are smaller in size. Therefore, for bigger workflows, its performance is better than that of DagHetMem. Table 4 shows relative and absolute running times of DagHetPart, where the relative ones are normalized by those of DagHetMem. Fig. 8 shows the relative running time of DagHetPart for each workflow, again relative to DagHetMem. Finally, Fig. 9 reports absolute running times, confirming the trend. The logarithmic y-axis lowers the perceived variance for the big workflows, but allows the reader to grasp the variance of the smallest workflows.

Workflow set	avg. relative runtime	avg. absolute runtime (sec)
real-world	406	0.5
small	1.63	2.83
middle	1.02	166.39
big	0.85	647.13

Table 4: Relative (compared to DagHetMem) and absolute running times of DagHetPart.

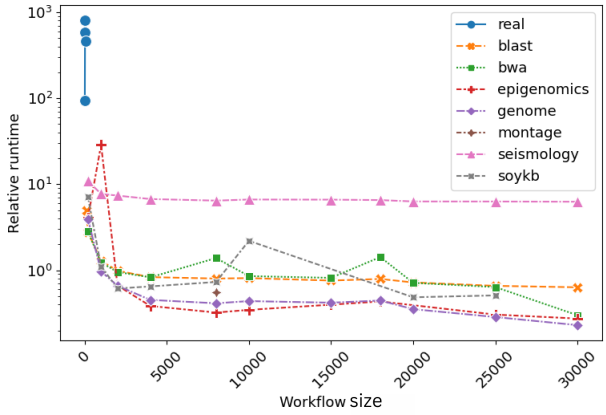

Figure 8: Rel. running time of DagHetPart to DagHetMem. Dotted lines are meant to improve readability.

5.2.8 Summary. Overall, DagHetPart is improving makespans upon the baseline for all workflow sizes and on all cluster configurations, even though the gain is affected by the cluster configuration. In particular, with a large cluster, there is a significant gain, which grows to an improvement of 4× to 5× compared to the baseline (except for real-world workflows that are smaller in size). While the relative makespan increases when the heterogeneity is increased,

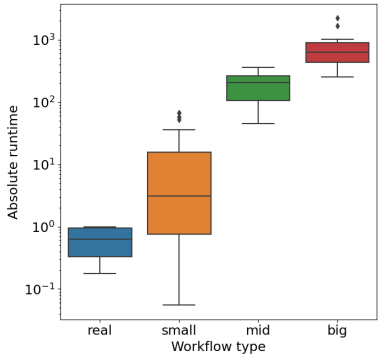

Figure 9: Absolute running time of DagHetPart (by workflow types). Note that the y-axis has a logarithmic scale.

we always observe a consistent improvement over the baseline, even in a fully homogeneous setting. Also, increasing the computational demand leads to similar relative values of makespan. Finally, increasing the communication bandwidth in the cluster has a positive impact on the improvement of the makespan of DagHetPart, which better exploits the workflow parallelism compared to DagHetMem. DagHetPart employs more processors and requires more communication. When the bandwidth grows, it benefits more from a high throughput of the execution environment.

6 CONCLUSIONS

We have tackled the fundamental but very complex problem of mapping large-scale memory-constrained workflows, modeled as a directed acyclic graph, onto a heterogeneous platform where each processor may have a different memory capacity and a different processing speed. In this setting, the objective is to minimize the total execution time (makespan). One of the main challenges is that some processors may not have enough memory to execute the workflow part assigned to them. While several mapping and scheduling algorithms have been proposed in various models, the problem of makespan minimization while accounting for memory constraints on DAGs has never been addressed, to the best of our knowledge.

To palliate the lack of heuristics accounting for memory constraints, we first proposed a baseline heuristic, DagHetMem, building upon existing work, that produces a valid mapping. Next, we designed a sophisticated new four-step heuristic, DagHetPart, which gradually improves a mapping solution that respects all constraints. Extensive simulation experiments on diverse workflows demonstrate that it pays off significantly to account for the platform heterogeneity and to exploit the parallelism available. On average, the makespan achieved by DagHetPart is a factor of 2.44 smaller than the baseline's, and it may become up to 5 times smaller for big workflows and large clusters. This significant improvement as well as its good scalability to big workflows are strong indications that the heuristic should work well in practical scenarios. As future work, we plan to perform more real-world experiments to confirm this trend, and to add one more level of heterogeneity by considering different communication bandwidths.

REFERENCES

[1] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. 2019. A survey on scheduling strategies for workflows in cloud environment and emerging trends. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–36.

[2] Shaikhah AlEbrahim and Imtiaz Ahmad. 2017. Task scheduling for heterogeneous computing systems. *The Journal of Supercomputing* 73 (2017), 2313–2338.

[3] J Bader, F Lehmann, L Thamsen, J Will, U Leser, and O Kao. 2022. Lotaru: Locally Estimating Runtimes of Scientific Workflow Tasks in Heterogeneous Clusters. In *Proc. 34th Intl. Conf. on Scientific and Statistical Database Mgmt. (SSDBM)*. ACM.

[4] Gabriel Bathie, Loris Marchal, Yves Robert, and Samuel Thibault. 2021. Dynamic DAG Scheduling Under Memory Constraints for Shared-Memory Platforms. *International Journal of Networking and Computing* 11, 1 (2021), 27–49.

[5] Anne Benoit, Mourad Hakem, and Yves Robert. 2009. Contention awareness and fault tolerant scheduling for precedence constrained tasks in heterogeneous systems. *Parallel Comput.* 23 (2009), 171–187.

[6] Luiz F Bittencourt, Alfredo Goldman, Edmundo RM Madeira, Nelson LS da Fonseca, and Rizos Sakellariou. 2018. Scheduling in distributed systems: A cloud computing perspective. *Computer science review* 30 (2018), 31–54.

[7] Peter Brucker. 2004. *Scheduling Algorithms*. Springer-Verlag.

[8] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Faraj, et al. 2022. More Recent Advances in (Hyper)Graph Partitioning. *ACM Comput. Surv.* 55, 12 (Nov 2022).

[9] Tainã Coleman, Henri Casanova, Loïc Pottier, Manav Kaushik, Ewa Deelman, and Rafael Ferreira da Silva. 2022. WfCommons: A framework for enabling scientific workflow research and development. *Future Generation Computer Systems* 128 (2022), 16–27.

[10] Philip A Ewels, Alexander Peltzer, Sven Fillinger, Harshil Patel, Johannes Alneberg, Andreas Wilm, Maxime Ulysse Garcia, Paolo Di Tommaso, and Sven Nahnsen. 2020. The nf-core framework for community-curated bioinformatics pipelines. *Nature biotechnology* 38, 3 (2020), 276–278.

[11] Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. 2015. Parallel scheduling of task trees with limited memory. *ACM Trans. on Par. Computing* 2, 2 (2015), 13.

[12] Chengbin Fan, Hui Deng, Feng Wang, Shoulin Wei, Wei Dai, and Bo Liang. 2015. A survey on task scheduling method in heterogeneous computing system. In *8th Intl. Conf. on Intelligent Networks and Intelligent Systems (ICINIS)*. IEEE, 90–93.

[13] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*.

[14] Changjiang Gou, Anne Benoit, and Loris Marchal. 2020. Partitioning tree-shaped task graphs for distributed platforms with limited memory. *IEEE Trans. on Parallel and Distributed Systems* 31, 7 (2020), 1533–1544.

[15] Suna He, Jigang Wu, Bing Wei, and Jiaxin Wu. 2021. 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*. 571–577.

[16] Julien Herrmann, M. Yusuf Özkaya, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. 2019. Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs. *SIAM J. on Scientific Computing (SISC)* 41, 4 (2019), A2117–A2145.

[17] Mathias Jacquelin, Loris Marchal, Yves Robert, and Bora Uçar. 2011. On optimal tree traversals for sparse matrix factorization. In *2011 IEEE International Parallel & Distributed Processing Symposium*. 556–567.

[18] Enver Kayaaslan, Thomas Lambert, Loris Marchal, and Bora Uçar. 2018. Scheduling series-parallel task graphs to minimize peak memory. *Theoretical Computer Science* 707 (2018), 1–23. https://doi.org/10.1016/j.tcs.2017.09.037

[19] Svetlana Kulagina, Henning Meyerhenke, and Anne Benoit. 2023. Mapping tree-shaped workflows on systems with different memory sizes and processor speeds. *Concurrency and Computation: Practice and Experience* (2023).

[20] Young Choon Lee, Hyuck Han, Albert Y Zomaya, and Mazin Yousif. 2015. Resource-efficient workflow scheduling in clouds. *Knowledge-Based Systems* 80 (2015), 153–162.

[21] Ulf Leser, Marcus Hilbrich, Claudia Draxl, et al. 2021. The Collaborative Research Center FONDA. *Datenbank-Spektrum* 21, 3 (2021), 255–260.

[22] Junwen Liu, Shiyong Lu, and Dunren Che. 2020. A survey of modern scientific workflow scheduling algorithms and systems in the era of big data. In *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 132–141.

[23] Ji Liu, Esther Pacitti, and Patrick Valduriez. 2018. A Survey of Scheduling Frameworks in Big Data Systems. *Intl. J. of Cloud Computing* 7 (01 2018).

[24] Orlando Moreira, Merten Popp, and Christian Schulz. 2017. Graph Partitioning with Acyclicity Constraints. In *16th Intl. Symp. on Experim. Algorithms, SEA 2017 (LIPIcs, Vol. 75)*. Schloss Dagstuhl - Leibniz-Zentrum f. Informatik, 30:1–30:15.

[25] M Yusuf Özkaya, Anne Benoit, Bora Uçar, Julien Herrmann, and Ümit V Çatalyürek. 2019. A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning. In *2019 IEEE Intl. Parallel and Distrib. Processing Symp. (IPDPS)*. IEEE, 155–165.

[26] Oliver Sinnen. 2007. *Task scheduling for parallel systems*. Vol. 60. John Wiley & Sons.

[27] Sara Stoudt, Váleri N. Vásquez, and Ciera C. Martinez. 2021. Principles for data analysis workflows. *PLOS Computational Biology* 17, 3 (03 2021), 1–26.