Memory-aware Adaptive Scheduling of Scientific Workflows On Heterogeneous Architectures

Abstract—**The analysis of massive scientific data often happens in the form of workflows with interdependent tasks. When such a scientific workflow needs to be scheduled on a parallel or distributed system, one usually represents the workflow as a directed acyclic graph (DAG). The vertices of the DAG represent the tasks, while its edges model the dependencies between the tasks (usually data to be communicated to successor tasks). When executed, each task requires a certain amount of memory and if that exceeds the available memory, the execution fails. The typical goal is to execute the workflow without failures (i.e., satisfying the memory constraints) and with the shortest possible execution time (i.e., to minimize its makespan).**

To address this problem, we investigate the memory-aware scheduling of DAG-shaped workflows on heterogeneous platforms, where each processor can have a different speed and a different memory size. We propose a variant of HEFT (Heterogeneous Earliest Finish Time) that (in contrast to the original) accounts for memory and includes eviction strategies for cases when it might be beneficial to remove some data from memory in order to have enough memory to execute other tasks. Furthermore, while HEFT assumes perfect knowledge of the execution time and memory usage of each task, the actual values might differ upon execution. Thus, we propose an adaptive scheduling strategy, where a schedule is recomputed when there has been a significant variation in terms of execution time or memory. The scheduler has been closely integrated with a runtime system, allowing us to perform a thorough experimental evaluation on real-world workflows. The runtime system warns the scheduler when the task parameters have changed, and a schedule can be recomputed on the fly. The memory-aware strategy allows us to schedule task graphs that would run out of memory with a state-of-the-art scheduler, and the adaptive setting allows us to significantly reduce the makespan.

I. Introduction

The analysis of massive datasets, originating from fields such as genomics, remote sensing, or biomedical imaging – to name just a few – has become ubiquitous in science; this often takes the form of workflows, i.e., separate software components chained together in some kind of complex pipeline [21]. These workflows are usually represented as directed acyclic graphs (DAGs). The DAG vertices represent the software components (or, more generally, the workflow *tasks*), while the edges model I/O dependencies between the tasks [1], [23]. Large workflows with resource-intensive tasks can easily exceed the capabilities of a single computer and are therefore executed on a parallel or distributed platform. An efficient execution of the workflows on such platforms requires the mapping of tasks to specific processors; to increase utilization by reusing finished processors, one also needs a task schedule (i.e., a valid execution order that respects the dependencies) and possibly also starting times for the tasks.

Modern parallel and in particular distributed computing platforms are often heterogeneous, meaning they feature varying CPU speeds and memory sizes. In general, having different memory sizes per CPUs makes it more challenging for an algorithm to compute a schedule that respects all memory constraints – meaning that no task is executed on a processor with less memory than needed for the task. Violating a memory constraint is, however, very important to avoid (possibly expensive) runtime failures and to provide a satisfactory user experience. Hence, building on previous related work [15], [16], [20], we consider a scheduling problem formulation that takes memory sizes as explicit constraints into account. Its objective is the very common *makespan* [23], which acts as proxy for the total execution time of a workflow. However, to the best of our knowledge, the only memory-aware heuristics that would account for memory constraints partition the DAG and do not reuse processors once they have processed a part of the graph. This approach leads to high values of makespan compared to a more fine-grained solution with processor reuse.

While previous work with memory constraints has focused on partitioning the DAG and not on reusing processors during execution, a seminal list scheduling heuristic for workflows on heterogeneous platforms, without accounting for the memory constraint, is HEFT (heterogeneous earliest finish time) [30]. It has two phases: (i) each task is assigned a priority and (ii) the tasks in a priority-ordered list are assigned to processors, where the "ready" task with the highest priority is scheduled next on the processor where it would complete its execution first. HEFT has been extended (e.g., by Shi and Dongarra [29]) and adjusted for a variety of different scheduling problem formulations. Yet, none of them adhere to memory constraints as addressed in this paper – see the discussion of related work in Section II. Another limitation of HEFT (and many other scheduling strategies) in practice is their assumption that the task running times provided to them are accurate. In practice, this is not the case and deviations from user estimates or historical measurements are very common [18]. As a consequence, it is advisable to adapt the schedule when *major* deviations occur. However, the original list-based schedulers, such as HEFT, are designed for a static setting with accurate task parameters.

The main contributions of this paper are both algorithmic and experimental:

- We formalize the problem with memory constraints, where communication buffers are used to evict data from memory if it will be later used by another processor.
- We design three HEFT-based heuristics that adhere to

memory size constraints: HEFTM-BL, HEFTM-BLC, and HEFTM-MM (M behind HEFT for memory, BL for bottom level, BLC for bottom level with communication, and MM for minimum memory traversal). The difference between the new heuristics is the way they prioritize tasks (for processor assignment).

- We implement a runtime system able to provide some feedback to the scheduler when task requirements (in terms of execution time and/or memory) differ from the initial predictions, and we recompute a schedule, based on the reported deviations.
- We perform extensive simulations, first in the static case by comparing the schedules produced by these heuristics with the classical HEFT as baseline (the latter does not take memory sizes into account); while HEFT returns invalid schedules that exceed the processor memories and cannot execute correctly, the new heuristics are able to successfully schedule large workflows and do so with reasonable makespans.
- In the dynamic setting, we use a runtime system that allows us to simulate workflow executions. The runtime system introduces deviations in running times and task memory requirements and communicates them to the scheduler; the scheduler can then recompute a schedule. Without these recomputations, most schedules become invalid after deviations, since the memory constraint is exceeded for most workflows, which demonstrates the necessity of a dynamic adjustment of the schedule.

We first review related work in Section II. Then, we formalize the model in Section III and present our algorithms in Section IV. The adaptation of the heuristics in a dynamic setting is discussed in Section V, and the results of our experiments are presented in Section VI. Finally, we conclude and provide future working directions in Section VII.

II. Related work

HEFT-based algorithms. Introduced in 2002, HEFT [30] is a list-based heuristic, consisting of two phases: task prioritization/ordering and task assignment. In the first phase, the algorithms compute bottom levels of the tasks based on some priorities (create the list), and then schedule tasks in the order of these priorities. The modifications of HEFT revolve around the way the priorities of the tasks are computed and the logic of the processor assignment. All such algorithms assume a heterogeneous execution environment.

Some variants have been designed with various ways of ordering tasks, for instance based on an optimistic cost table in PEFT (Predict earliest finish time) [4], or by combining the standard deviation with the communication cost weight on the tasks in HSIP (Heterogeneous Scheduling with Improved task Priorities) [31].

The TSHCS (Task Scheduling for Heterogeneous Computing Systems) algorithm [2] improves on HEFT by adding randomized decisions to the second phase. The SDC algorithm [29] considers the percentage of feasible processors in addition to a task's average execution cost in its weight. HEFT can also be adapted in cloud-oriented environments [27] and even combined with reinforcement learning techniques [33], but none of these variants of HEFT consider memory constraints, to the best of our knowledge.

Memory-aware scheduling algorithms. The way processor memories are represented in the model has a decisive impact on the way the constraint is formulated and addressed in the algorithm. Different models of memory available on processors and memory requirements of tasks have been presented.

Marchal et al. [25] assume a memory model where each processor has an individual memory available. Workflow tasks have no memory requirements, but they have input and output files that need to be stored in the memory. A polynomial-time algorithm for computing the peak memory needed for a parallel execution of such a workflow DAG is provided, as well as an integer linear programming (ILP) solution to the scheduling problem. The memory model requires the deletion of all input files when starting the task.

Other models consider for instance a dual-memory system [17] where a processor can have access to two different kinds of memory, and each task can be executed on only one sort of memory. The authors then present an ILP-based solution for this problem formulation. Yao et al. [34] consider that each processor has its own internal memory, and all processors share a common external one. The internal (local) memory is used to store the task files. The external memory is used to store evicted files to make room for the execution of a task on a processor. In [12], the authors consider connected processors with individual limited memories forming a global memory, with different access times to memory, and no weights on edges. An ILP is proposed in this setting. There are also some cloud-oriented models that include costs associated with memory usage [22].

Overall, there are a variety of memory models, but, to the best of our knowledge, the only study on a multiprocessor platform that is fully heterogeneous, with individual memories, is the one from [20]. Yet, they propose only a partitioning-based mapping of the workflow, without processor reuse. Hence, in [20], there is no need for communication buffers to store data that should be communicated between two processors when tasks are ready to execute.

Dynamic/adaptive algorithms. We finally review related work in a dynamic setting. With no variation in task parameters, DVR HEFT [28] rather considers that new tasks arrive in the system. They use an almost unchanged HEFT algorithm in the static step, executing three slightly varying variants of task weighting and choosing the variant that gives the best overall makespan. The dynamic critical path (DCP) algorithm for grids maps tasks to machines by calculating the critical path in the graph dynamically at every step [26]. The authors schedule the first task on the critical path to the best suitable processor and recompute the critical path. The heuristic also uses the same processor to schedule parent and children tasks, as to avoid data transfer between processors.

Garg et al. [14] propose a dynamic scheduling algorithm for heterogeneous grids based on rescheduling. The procedure involves building a first (static) schedule with HEFT, periodic resource monitoring, and rescheduling the remaining tasks. Upon rescheduling, a new mapping is calculated from scratch, and this mapping is accepted if the resulting makespan is smaller than the previous one.

De Olivera et al. [11] propose a tri-criteria (makespan, reliability, cost) adaptive scheduling algorithm for clouds. They solve a set of linear equations that represent the cost of an execution based on the criteria. The authors test four scenarios – one preferring each criterion and a balanced one. The algorithm chooses the best virtual machine for each next task based on the cost given by the model. The authors used workflows with less than ten tasks, but repeated them so that the execution had up to 200 tasks.

Daniels et al. [10] formalize the concept of robust scheduling with variable processing times on a single machine. The changes in runtimes of tasks are not due to changing machine properties, but are rather task-related (and thus unrelated to each other). The authors search for an optimal schedule in relation to a performance measure. Then they proceed to formulate the Absolute Deviation Robust Scheduling Problem as a set of linear constraints.

While several related works consider building a new schedule once some variation has been observed, we are not aware of work implementing a real runtime system that interacts with the scheduler and has been tested on workflows with thousands of tasks, as we propose in this paper. Furthermore, we are not aware of any previous work discussing dynamic algorithms combined with memory constraints.

III. Model

The applications we target, large scientific workflows for which we do not have exact a priori knowledge, are described in Section III-A. Then, the type of heterogeneous system on which the applications are to be executed is presented in Section III-B. The optimization problem is defined in Section III-C, and the key notation is summarized in Table I.

A. Applications: Large scientific workflows

Following common practice, we represent a workflow by a DAG (Directed Acyclic Graph) $G = (V, E)$. The set of vertices V corresponds to tasks, while edges express the precedence constraints. Hence, a directed edge $e = (u, v) \in E$ means that task $u \in V$ must be executed before task $v \in V$. A cost $c_{u,v}$ is associated with each edge, representing the size of the output of task u, to be used by task v. Furthermore, w_u is the number of operations performed by task $u \in V$, and m_u is the amount of memory required by task u to be executed. We denote by $parent_u$ the tasks preceding task $u \in V$, which must be completed before u can be started: $parent_u = \{v \in V : (v, u) \in E\}$. A *source* task is a task without parents. Similarly, the children of task $u \in V$ are $child_u = \{v \in V : (u, v) \in E\}$, and a *target* task has no children. Each task may have multiple parents and children.

Symbol	Meaning
$G = (V, E)$	Application DAG (tasks and edges)
m_u	Amount of memory required by task u
w_u	Number of operations of task u
$c_{u,v}$	Size of output file for edge $(u, v) \in E$
$\mathcal{S}$, k	Computing platform, total number of processors
M_j, MC_j, s_j	Memory size, comm. buffer size, and speed of proc. p_j
β	Bandwidth in the compute system
blu	Bottom level of task u

TABLE I
Key Notation

Note that m_u is the total memory usage of a task during its execution, including input and output files currently being read and written, and hence the total memory requirement for executing task u is the maximum of the following: (i) the total size of the files to be received from the parents, (ii) the total size of the files to be sent to the children, and (iii) the total memory size m_u (which often reaches the maximum):

$$r_u = \max\left\{ m_u, \sum_{v:(v,u)\in E} c_{v,u}, \sum_{v:(u,v)\in E} c_{u,v} \right\}.$$

Furthermore, we operate in a context where we do not have perfect knowledge of the task parameters (w_u and m_u) before the tasks start their execution, but only estimates [14], [26]. Hence, scheduling decisions are made on these estimated parameters, and may be reconsidered at runtime when a task starts its execution and we know its exact parameters.

B. Heterogeneous system

The target platform is a heterogeneous system $\mathcal{S}$ with k processors $p_1, \ldots, p_k$. For $1 \le j \le k$, each processor p_j has an individual memory of size M_j, a communication buffer of size MC_j and a speed s_j. We can decide to evict some data from the main memory when we are sending the data to another processor; it then stays in the communication buffer until it has been sent. The execution time of a single task $u \in V$ on a processor p_j is $\frac{w_u}{s_j}$, and all processors are connected with an identical bandwidth β.

We keep track of the current ready time of each processor and each communication channel, rt_j and $rt_{j,j'}$, for each processor j and all pairs (j, j'). Initially, all the ready times are set to 0. We also keep track of the currently available memory, $availM_j$ and $availC_j$, on the processor memory and communication buffer, respectively. Furthermore, PD_j is a priority queue with the *pending data* that are in the memory of size M_j but may be evicted to be communicated if more memory is needed on p_j. They are ordered by non-decreasing size and correspond to some $c_{u,v}$.

In oder to compute the memory requirement of a DAG, we use memDag [19], an algorithm that transforms the workflow into a series-parallel graph and then finds the traversal that leads to the minimum memory consumption.

C. Optimization problem

The goal is to find a schedule of the DAG G for the k processors, so that the makespan (total execution time) is minimized while respecting memory constraints. If a processor runs out of memory to execute a task mapped on it, the schedule is said to be *invalid*.

Since tasks are subject to variability, we aim at minimizing the actual makespan achieved at the end of the execution, while decisions may be taken w. r. t. the estimated task parameters.

Note that the problem is already NP-hard even in the homogeneous case and without memory constraints, because of the DAG structure of the application. Hence, we focus on the design of efficient scheduling heuristics.

IV. Scheduling heuristics

We design variants of HEFT that account for memory usage and aim at minimizing the makespan. First, we present in Section IV-A the baseline HEFT heuristic; as it does not account for the memory, it may return invalid schedules that will not be able to run successfully on the platform (due to running out of memory). Then, Section IV-B focuses on the presentation of the novel heuristics, including eviction strategies to move some data in communication buffers in case there is not enough memory available on some processors.

A. Baseline: original HEFT without memories

Original HEFT does not account for memory sizes. Its schedules can be invalid if tasks are assigned to processors without enough memory. These solutions can be viewed, however, as a "lower bound" for a valid solution that respects memory constraints.

HEFT works in two stages. In the first stage, it computes the ranks of tasks by computing their non-increasing bottom levels. The bottom level of a task is defined as

$$bl(u) = w_u + \max_{(u,v)\in E}\{c_{u,v} + bl(v)\}$$

(with the max yielding 0 if there is no outgoing edge). The tasks are sorted by non-decreasing ranks.

In the second stage, the algorithm iterates over the ranks and tries to assign the task to the processor where it has the earliest finish time. We tentatively assign each task v to each processor j. The task's starting time st_v is dictated by the maximum between the ready time of the processor rt_j and all communications that must be orchestrated from predecessor tasks $u \notin T(p_j)$. The starting time is then:

$$ST(v,p_j) = \max\{rt_j, \max_{u\in\Pi(v)}\{FT(u) + c_{u,v}/\beta, rt_{proc(u),p_j} + c_{u,v}/\beta\}\}$$

Finally, its finish time on p_j is $FT(v,p_j) = st_v + \frac{w_v}{s_j}$.

Once we have computed all finish times for task v, we keep the minimum $FT(v,p_j)$ and assign task v to processor p_j.

Assignment to processor. When assigning the task, we set the ready time rt_j of processor j to be the finish time of the task. For every predecessor of v that has been assigned to another processor, we adjust ready times on communication buffers $rt_{j',j}$ for every predecessor u's processor j': we increase them by the communication time $c(u,v)/\beta$.

B. Memory-aware heuristics

Like the original HEFT, the memory-aware versions of HEFT consist of two stages: first, they compute the task ranks, and second, they assign tasks to processors in the order defined in the first stage. We consider three variants of HEFT accounting for memory usage (HEFTM), which only differ in the order they consider tasks to be scheduled in the first stage.

Compute task ranks.

Our three variants of memory-aware HEFT work as follows:

- HEFTM-BL orders tasks by non-increasing bottom levels, where the bottom level is defined as
$$bl(u) = w_u + \max_{(u,v)\in E}\{c_{u,v} + bl(v)\}$$
(max yields 0 if there is no outgoing edge).
- HEFTM-BLC gives more priority to tasks with potential large incoming communications, hence aiming at clearing the memory used by files as soon as possible, to have more free memory for remaining tasks to be executed on the processor. Therefore, for each task, we compute a modified bottom level accounting for communications:
$$blc(u) = w_u + \max_{(u,w)\in E}\{c_{u,w} + blc(w)\} + \max_{(v,u)\in E} c_{v,u}.$$
- Finally, HEFTM-MM orders tasks in the order returned by the MEMDAG algorithm [19], which corresponds to a traversal of the graph that minimizes peak memory usage.

Task assignment.

Then, the idea is to pick the next free task in the given order, and greedily assign it to a processor, by trying all possible options and keeping the most promising one. We first detail how a task is tentatively assigned to a processor, by carefully accounting for the memory usage. Next, we explain the steps to be taken to effectively assign a task to a given processor.

Tentative assignment of task v on p_j.

Step 1. First, we need to check that for all predecessors u of v that are mapped on p_j, the data $c_{u,v}$ is still in the memory of p_j, i.e., $c_{u,v} \in PD_j$. Otherwise, the finish time is set to $+\infty$ (invalid choice).

Step 2. Next, we check the memory constraint on p_j, by computing

$$Res = availM_j - m_v - \sum_{u\in\Pi(v), u\notin T(p_j)}\{c_{u,v}\} - \sum_{w\in Succ(v)}\{c_{v,w}\}.$$

$T(p_j)$ is the set of tasks already scheduled on p_j; by Step 1, their files are already in the memory of p_j. However, the files from the other predecessor tasks must be loaded into memory before executing task v, as well as the task data of size m_v, and the data generated for all successor tasks. Res then checks whether there is enough memory; if it is negative, we have exceeded the memory of p_j with this tentative assignment. In this case ($Res < 0$), we try to evict some data from memory so that there is enough memory to execute task v. Clearly, we need to evict at least Res data. To this end, we propose a greedy approach, which evicts the largest files of

PD_j until data of size Res have been evicted. A variant where the smallest files are evicted first has been tested; it led to comparable results. While tentatively evicting files, we remove them from the list of pending memories and move them into a list of memories pending in the communication buffer. We keep track of the available buffer size, too – every time a file is moved into the pending buffer, the available buffer size is reduced by the file size.

If, after tentatively evicting all files from PD_j, we still do not have enough memory, or if we exceed the size of the available buffer during this process, we set the finish time to $+\infty$ (indicating an invalid choice).

Step 3. We tentatively assign task v on p_j. Its starting time st_v is dictated by the maximum between rt_j and all communications that must be orchestrated from predecessor tasks $u \notin T(p_j)$. The starting time is therefore:

$$ST(v,p_j) = \max\{rt_j, \max_{u \in \Pi(v), u \notin T(p_j)} \{FT(u), rt_{proc(u),p_j}\} + c_{u,v}/\beta\}.$$

Finally, its finish time on p_j is $FT(v,p_j) = ST(v,p_j) + \frac{w_v}{s_j}$.

Assignment of task v.
Once we have computed all finish times for task v, we keep the minimum $FT(v,p_j)$ and assign v to processor p_j. In detail:

- We evict the files corresponding to edge weights that need to be evicted to free the memory. We remove these files from pending memories PD_j, add them to pending data in the communication buffer, and reduce the available buffer size accordingly.
- We calculate the new $availM_j$ on the processor. Next, we subtract the weights of all incoming files from predecessors assigned to the same processor and add the weights of outgoing files generated by the currently assigned task.
- For every predecessor of v that has been assigned to another processor, we adjust ready times on communication buffers $rt_{j',j}$ for the processor j' that the predecessor u has been assigned to: we increase them by the communication time $c(u,v)/\beta$. We also remove the incoming files from either the pending memories or pending data in buffers of these other processors, and increase the available memory or buffer sizes on these processors.
- We compute the correct amount of available memory for p_j (for when the task is done). Then, for each predecessor that is mapped to the same processor, we remove the pending memory corresponding to the weight of the incoming edge, also freeing the same amount of available memory (increasing $availM_j$). For each successor, we rather add the edge weights to pending memories and reduce $availM_j$ by the corresponding amount.

V. Dynamic scenario

In a workflow execution environment, the scheduling method interacts with the runtime environment, which provides information such as resource estimates. This information may include memory usage, running time, graph structures, or the status of the underlying infrastructure. In order to ensure that the information is up to date, a monitoring system observes the workflow execution and collects metrics for tasks and the underlying infrastructure. By incorporating dynamic monitoring values, e.g., the resources a task consumed, the runtime environment can incorporate the data into the prediction model to provide more accurate resource predictions. Also the underlying infrastructure can change during the workflow execution. Examples are processor failures, node recoveries, or acquisition of new nodes. Even if the hardware infrastructure does not change, the set of nodes provided as a scheduling target might change due to release or occupation in shared cluster infrastructures. As infrastructure information and resource predictions are dynamically updated and provided to the scheduler during workflow runtime, the previous schedule may become invalid, so that a new one must be calculated.

For state-of-the-art memory prediction methods, a cold-start median prediction error for heterogeneous infrastructures of approximately 15% has been observed [24]. Online prediction methods were able to significantly reduce the error during runtime, with the reduction reaching up to one third of the cold-start error [5], [32]. Such a dynamic execution environment requires a dynamic scheduling method where the schedule can be recomputed during workflow execution.

Retracing the effects of change on an existing schedule

After the monitoring system has reported changes, we need to assess their impact on the existing schedule. These changes can invalidate the schedule (*e. g.* if there is not enough memory for some tasks to execute anymore), they can lead to a later finishing time (*e. g.* if some tasks are longer and they delay other tasks), or they can have no effect (*e. g.* if new processors joined the cluster, but the old schedule did not account for them). To assess the impact, we need to retrace the schedule.

First, we find out if at least one processor that had assigned tasks has terminated operation – this instantly invalidates the entire schedule. We then iterate over all tasks of the workflow in a topological order – any of the orderings given by rankings BL, BLC or MM is a topological ordering. We then repeat steps similar to those we did during tentative assignment in the heuristics, except that we do not choose a processor anymore, but rather we check whether the current processor assigned to the task still fits.

For each task v, we first assess its current memory constraint Res using Step 2 from the heuristic. The factors that affect Res are possible changes in m_v, in $c_{u,v}$ from predecessors u or $c_{v,w}$ from successors w, available memory $availM_j$ on the processor (due to either changed M_j or changed memory requirements from other tasks). If Res was originally positive (no files evicted from memory into the communication buffer), then it has to stay this way – otherwise, evicted files can invalidate subsequent tasks. If Res was originally negative, then we need to make sure that evicted files still fit into the communication buffer. If either Res is newly negative or the communication buffer is not large enough, then this invalidates the schedule. We update the $availM_j$ and $availMC_j$ values according to the new memory constraints.

Then, we can re-calculate the finish time of the task on its processor like in Step 3. The factors that affect it are changes in own execution time w_v of the tasks, changed ready time of the processor (after delayed previous tasks), and changed communication buffer availability.

Finally, after having updated the processor's values, we move on to the next task.

VI. Experimental evaluation

We first describe the experimental setup in Section VI-A. Then, we report results on static experiments to assess the performance of the memory-aware heuristics in Section VI-B, before discussing the heuristics' behavior in a dynamic setting in Section VI-C. Finally, we report running times of the heuristics in Section VI-D.

A. Experimental setup

All algorithms are implemented in C++ and compiled with g++ (v.8.5.0). The experiments are managed by simexpal [3] and 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 to allow reproducibility of the results at https://zenodo.org/records/13919214 and https://zenodo.org/records/13919302.

Before presenting the results, we first describe the set of workflows we used for the evaluation and then the clusters on which the workflows were scheduled. We also explain the way the runtime system works and interacts with our scheduler.

1) Workflow instances: We run experiments on a set of several real-world workflows from Ref. [6] (atacseq, bacass, chipseq, eager, and methylseq). We use the WFGen generator [7] to create larger variants of the workflows above and add the instances created this way to the set.

a) Workflow graphs: For the five real-world workflows, their nextflow definition (see [13]) 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 size-increased workflows, the graph is produced by the WFGen generator, based on a *model workflow* and the desired number of tasks. We used the real-world workflows as models, except for bacass since it leads to errors in the generator. 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, and 30 000. We divide the workflows into four groups by size: tiny ones with up to 200 tasks, small ones with 1 000 to 8 000 tasks, middle ones with 10 000 to 18 000 tasks, and big ones with 20 000 to 30 000 tasks.

b) Task and edge weights: For the real-world workflows, we use historical data files provided by Bader et al. [6]. 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 total memory requirements (input/output files plus memory consumption of the computation). In a similar manner, the historical data provided by Ref. [6] 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. The same workflow can require different memory capacities and take different times to execute depending on the size of its input. We simulate these various runs by obtaining values corresponding to each input size. For each of the four families, there are five input sizes; we thus run each workflow in five variants corresponding to these inputs.

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 the tasks have no historical runtime data stored. Hence, in the absence of historical data about a task, we give it fixed weights: an execution time of 1, a memory requirement of $50MB$, and files written and received of $1KB$. These values align with the findings of Ref. [6] about small tasks.

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 Ref. [6]. We set the number of each kind of node to 12, yielding 72 processors in total. Each machine has a (normalized) CPU speed and a memory size (in GB), and we list them as tuples (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 II).

We also consider a more constrained setting by varying the cluster configuration. The *memory-constrained cluster* consists of 72 nodes (12 of each kind) as the default cluster, but each node has 10 times less memory. Hence, the *luxury* machine $C2$ has 19.2 GB memory in this setting instead of 192 GB, $A2$ has 6.4 GB instead, of 64 GB etc. The processor speeds and their relations stay unchanged (see Table II).

Note that in both clusters, we set the size of the communication buffer to be equal to ten times the memory size.

3) Runtime system: To simulate the execution of a workflow, we implemented a runtime system. It reads the historical

Processor name	CPU speed (GHz)	Memory size (GB)	
		default	*mem-constrained*
local	4	16	1.6
A1	32	32	3.2
A2	6	64	6.4
N1	12	16	1.6
N2	8	8	0.8
C2	32	192	19.2

TABLE II
Cluster configurations.

data and builds weights for tasks as explained above. In the static case, these values are being sent to the scheduler, which builds a schedule according to these weights. In the dynamic setting, in turn, the runtime system applies a deviation function to the values. This function computes a normally distributed random deviation value, where the initial value is the mean and the deviation is 10%. This scenario corresponds to the real-life scenarios identified in Ref. [6] and other works dedicated to predicting task running times [8], [9].

Thus, the scheduler receives deviation values and makes decision based on them. This leads to several types of possible issues:

- A processor is blocked by another task. If the scheduler underestimated the execution time of a task, it will block another one from starting.
- A predecessor has not finished yet. The scheduler may request a task to start on its processor, while some of the predecessors of the task have in fact not yet completed their execution – the task is therefore not yet ready.
- Not enough memory. If the scheduler underestimated the amount of memory a task requires, this task might not be able to execute on a chosen processor.
- A task took less time than expected. We only consider this case if a task took more than 10% less time than expected. In this case, we want to exploit the newly acquired free time by possibly starting other tasks earlier.

B. *Results in a static setting*

We first study the heuristics' behavior when the weights do not change during runtime, hence the scheduler has perfect knowledge of the task memory requirements and execution times. We have compared two eviction strategies, starting with large files first or small files first, and did not observe any significant changes in terms of schedule validity or makespan. Thus, we only present results with the eviction of largest files.

1) Scheduling on the default cluster: On the default cluster, the three memory-aware heuristics are able to schedule all workflows (see Figure 1), while the baseline HEFT has a success rate of only 24.2% (75.7% failure rate). Indeed, HEFT is only able to schedule small workflows; it cannot schedule any workflow with more than 4000 tasks correctly as some tasks run out of memory. As soon as we are not in a setting with abundant processing resources for small workflows, it is hence necessary to adopt a memory-aware strategy in order to produce valid schedules.

We also report in Figure 2 the relative makespan found by the memory-aware heuristics, normalized to the makespan achieved by HEFT, often with an invalid over-optimistic schedule that exceeds the bound on memory. On average, the makespans found by HEFTM-BL are 7.8% worse than those found by the baseline, the makespans of HEFTM-BLC are 8.0% worse, and those found by HEFTM-MM are 82.6% worse. These are still very encouraging results, in particular for HEFTM-BL and HEFTM-BLC, since the makespans of HEFT correspond to invalid schedules.

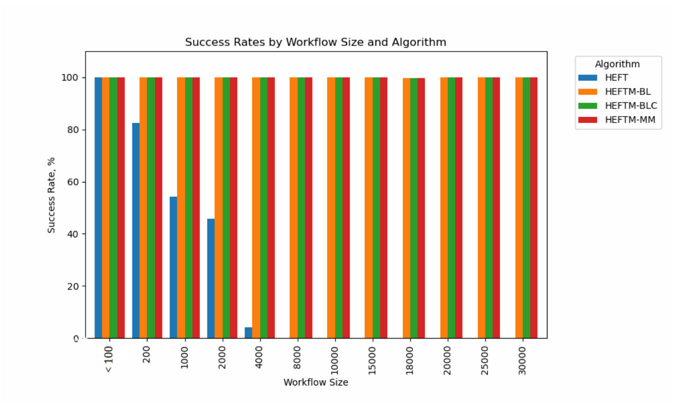

Fig. 1. Success rates on the default cluster. Higher is better.

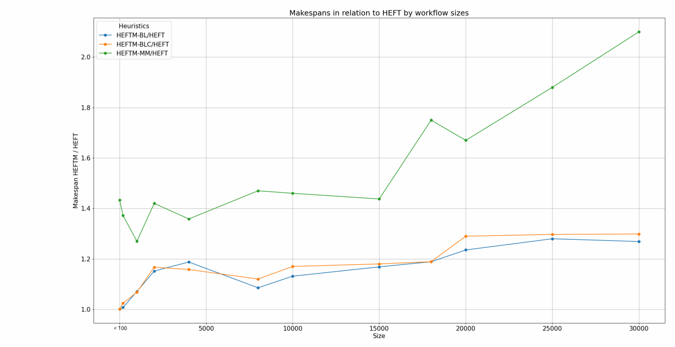

Fig. 2. Relative makespans of heuristics normalized by HEFT makespan, by workflow size, on default cluster. Smaller is better.

Finally, we study the percentage of memory occupied by the schedule, which is another good indicator of the memory usage of the heuristics and their ability to produce valid schedules. Figures 3 and 4 show the percentage of memory occupied on average by the schedule produced by the different heuristics for different workflow sizes; first on all schedules (including invalid HEFT schedules) and then only on valid schedules (hence, no results for HEFT on large workflow sizes).

HEFTM-MM continuously outperforms the other heuristics in terms of memory usage, using from 46% less memory on the smallest workflows to 4 times less memory on the largest ones with $30\,000$ tasks. If we consider the invalid HEFT schedules, too, we see that they would require more and more memory on average, which explains why these schedules rapidly become invalid. This is because some assignments require more than 100% of the memory (which makes them invalid). We can assess the degree of invalidity by comparing HEFT memory usage with the memory usage of HEFTM-BL. HEFTM-BL differs from the baseline only in the sense that it respects the available memory on the processors. For the largest workflows, HEFT schedules require almost twice as much memory as HEFTM-BL.

2) Scheduling on the memory-constrained cluster: On the memory-constrained cluster, HEFT produces valid assign-

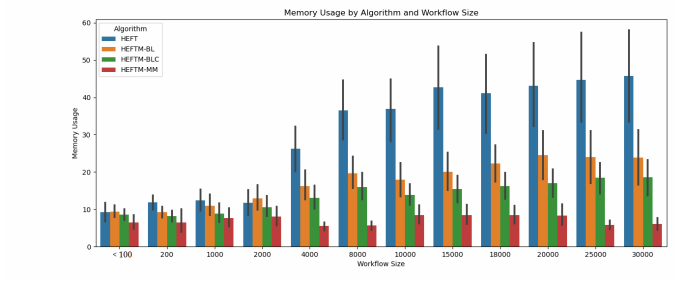

Fig. 3. Memory usage on default cluster, including invalid HEFT schedules.

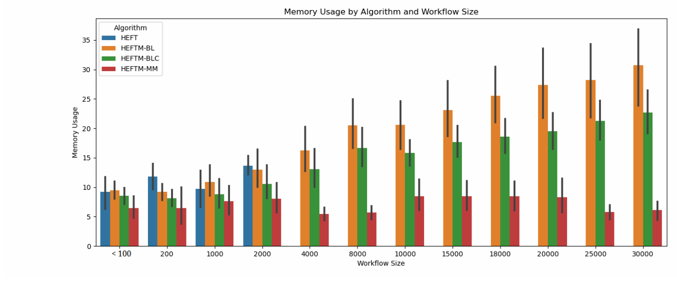

Fig. 4. Memory usage on default cluster, considering only valid HEFT schedules.

ments in only 14 out of 290 experiments 290 (4.8% success rate). The successful schedules are achieved exclusively on the tiny workflows (with only two 200-task size-increased workflow among them). HEFTM-BL successfully schedules 38% of the workflows, HEFTM-BLC is able to schedule 49% of them, while HEFTM-MM still schedules all of them, including even the largest ones, see Figure 5. As also observed on the default cluster, HEFTM-MM seems to be less affected by the workflow size than the other heuristics.

Similarly to the default cluster, we observe that the makespan of HEFTM-MM is usually higher than with HEFT (see Fig. 6), but the HEFT schedules are almost all invalid. It is thus worthwhile to resort to HEFTM-MM for large workflows in a constrained cluster, since tasks are processed in an order that minimizes the memory usage of schedule.

Memory usages on the constrained cluster are depicted in Fig. 7, and we observe that the memory footprint of HEFTM-MM remains constant with workflow size.

C. Dynamic experiments on the memory-constrained cluster

The makespan in case of no recomputation becomes invalid as soon as at least one task finds itself in an invalid memory size situation – that is, if the scheduler assumed the task's memory to be smaller than actually needed and assigned the task to a processor with not enough memory capacity. Due to an extremely constrained memory in this cluster, only 134 experiments out of 1160 for all algorithm variants end with a valid makespan without recomputation. In case of HEFT, 14 valid initial makespans were computed. Out of them, 13 remain valid until the end, after all the update requests. The same 13 experiments end with a valid

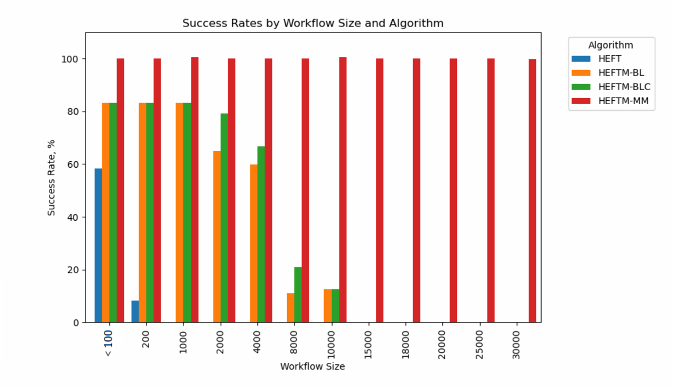

Fig. 5. Success rates on the memory-constrained cluster. Higher is better.

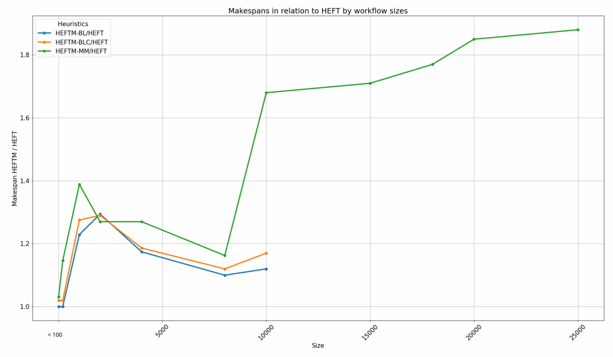

Fig. 6. Relative makespans on the memory-constrained cluster. Smaller is better.

makespan in case of no recomputation. So, these workflows require so few resources that there is no point in re-scheduling them. In case of HEFTM-MM, all 290 workflows can be scheduled initially and all the schedules remain valid until the end. 16 experiments end with a successful schedule without recomputation, a rate of 5.5%. HEFTM-BLC produces 142 valid initial schedules and keeps 141 of them valid until the end. 50 experiments are successful without recomputation, a rate of 35% out of all successful final schedules. HEFTM-BL keeps 105 schedules valid until the end out of 110 initial valid schedules. 55 remain valid until the end even without recomputation, roughly 50% of all successes.

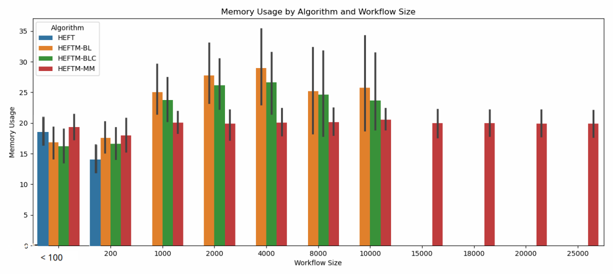

Fig. 7. Memory usage on the memory-constrained cluster.

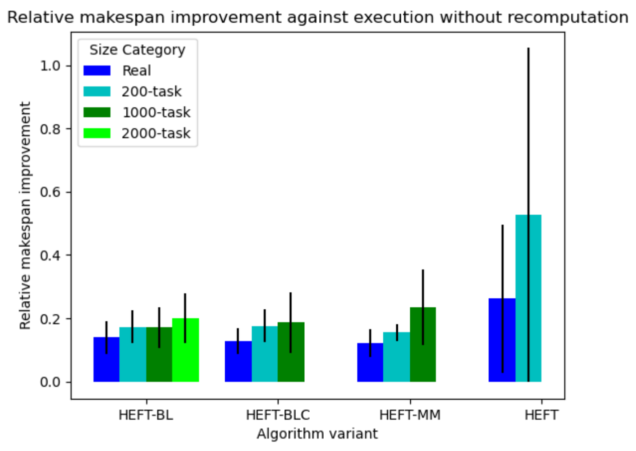

Fig. 8. Relative (excess) makespan of HEFTM-BL, HEFTM-BLC and HEFTM-MM. Smaller is better.

HEFTM-BL and HEFTM-BLC are successful on smaller workflows and fail on larger ones, the success of the strategy without recomputation is limited to even the smallest of these smaller workflows. Thus, the strategy without recomputation delivers valid makespans (independently of the algorithm) on 56 original workflows with < 100 tasks, 47 200-task ones, 25 1000-task ones, and 6 2000-task ones.

Figure 8 shows the increase in makespan in case of no recomputation for these experiments. With growing size of the workflow, the excess makespan of not recomputing grows – from 13.9% to 20% for HEFTM-BL, from 12.7% to 18.7% on HEFTM-BLC (but there is no data for the 2000-task workflows in this case), 12.1% to 23.5% for HEFTM-MM. The larger variations for HEFT can be explained by the small amount of data – for instance, there are only 2 200-task workflows for this case.

D. Running times of the heuristics

In order to answer the runtime system without holding it up for too long, the scheduler needs to compute a schedule reasonably quickly. The bottom-level-based heuristics HEFTM-BL and HEFTM-BLC yield smaller running times than HEFTM-MM, and also scale better with growing workflow sizes (see Fig. 9). Their running times are similar and grow from tens of milliseconds for the smallest workflows to 25-27 seconds for the largest workflows. HEFTM-MM, in turn, computes a memory-optimal traversal of the entire workflow to compute the task ranks. It also takes only tens of milliseconds for the smallest inputs, but requires thousands of seconds for the large(st) workflows (1172.7s for 20 000-task workflows and 2994.9s for 30 000-task ones). This increased running time is, however, offset by the unique 100% success rate this algorithm obtained in our experiments when scheduling large workflows in difficult (memory-constrained) setups.

VII. Conclusion

We have formalized a scheduling problem in memory-constrained environments where tasks may exceed the memory available on a processor and resort to communication buffers to store and communicate data between processors. In this context, we have designed three memory-aware HEFT-based heuristics which account for memory constraints when scheduling workflow tasks.

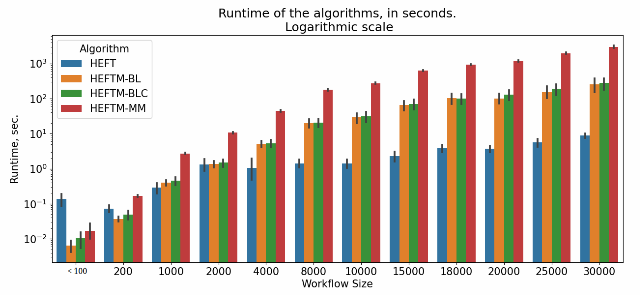

Fig. 9. Average running times (in seconds) of the heuristics w. r. t. workflow size. The y-axis is logarithmic.

Two of these heuristics rely on a task ordering w. r. t. the bottom level of tasks, with the objective of minimizing the makespan and hence scheduling critical tasks first. The third one, HEFTM-MM, handles tasks in an order dictated by an efficient traversal of the workflow in terms of memory requirements, hence reducing the memory used by the schedule. Experimental results on a large set of workflows from real-world applications demonstrate that the memory-aware heuristics produce valid schedules successfully. In the most memory-constrained setting, HEFTM-MM succeeds to schedule even the largest workflows, while the other heuristics return invalid schedules that exceed the memory capacity. This advantage of HEFTM-MM comes at the price of worse makespans than those of HEFTM-BL and HEFTM-BLC. As expected, the baseline HEFT, which is not memory-aware, returns invalid schedules in almost all cases, except for very small workflows.

Another key contribution is that we have adapted these heuristics for a dynamic setting, where exact task parameters (execution times and memory requirements) are not known in advance. We have implemented a runtime system that interacts with the scheduler, providing exact parameter values once a task arrives in the system, while only estimates are known for future tasks. Some preliminary experiments have been conducted in this setting and demonstrated that it is necessary to adapt the schedule on the fly in order to avoid an execution failure because of a memory shortage. To the best of our knowledge, this is the first study of adaptive algorithms accounting for memory constraints.

This work can be extended in several directions. First, the model could be refined to include heterogeneous bandwidths. More importantly, it would be interesting to consider other types of variability, *e. g.*, if new tasks (dis)appear in the workflow graph, or if there is variability in the platform, with processors arriving and departing. We believe that we could adapt the current approach based on recomputing schedules on the fly, and we plan to perform a new set of experiments to further assess the impact of dynamic scheduling techniques.

Acknowledgements: Funding acknowledgement suppressed to preserve anonymity.

References

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

[2] S. AlEbrahim and I. Ahmad. Task scheduling for heterogeneous computing systems. *The Journal of Supercomputing*, 73:2313–2338, 2017.

[3] E. Angriman, A. van der Grinten, M. von Looz, H. Meyerhenke, M. Nöllenburg, M. Predari, and C. Tzovas. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms*, 12(7):127, 2019.

[4] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694, 2014.

[5] J. Bader, N. Diedrich, L. Thamsen, and O. Kao. Predicting dynamic memory requirements for scientific workflow tasks. In *2023 IEEE International Conference on Big Data (BigData)*, pages 182–189, 2023.

[6] J. Bader, F. Lehmann, L. Thamsen, J. Will, U. Leser, and O. Kao. Lotaru: Locally estimating runtimes of scientific workflow tasks in heterogeneous clusters. *Proceedings of the 34th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2022.

[7] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, and R. Ferreira da Silva. Wfcommons: A framework for enabling scientific workflow research and development. *Future Generation Computer Systems*, 128:16–27, 2022.

[8] R. F. Da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, and M. Livny. Toward fine-grained online task characteristics estimation in scientific workflows. In *Proceedings of the 8th workshop on workflows in support of large-scale science*, pages 58–67, 2013.

[9] R. F. Da Silva, G. Juve, M. Rynge, E. Deelman, and M. Livny. Online task resource consumption prediction for scientific workflows. *Parallel Processing Letters*, 25(03):1541003, 2015.

[10] R. L. Daniels and P. Kouvelis. Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management science*, 41(2):363–376, 1995.

[11] D. de Oliveira, K. A. Ocaña, F. Baião, and M. Mattoso. A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds. *Journal of grid Computing*, 10:521–552, 2012.

[12] J. Ding, L. Song, S. Li, C. Wu, R. He, Z. Su, and Z. Lü. A heuristic method for data allocation and task scheduling on heterogeneous multiprocessor systems under memory constraints. In Z. Tari, K. Li, and H. Wu, editors, *Algorithms and Architectures for Parallel Processing*, pages 360–380, Singapore, 2024. Springer Nature Singapore.

[13] P. A. Ewels, A. Peltzer, S. Fillinger, H. Patel, J. Alneberg, A. Wilm, M. U. Garcia, P. Di Tommaso, and S. Nahnsen. The nf-core framework for community-curated bioinformatics pipelines. *Nature biotechnology*, 38(3):276–278, 2020.

[14] R. Garg and A. K. Singh. Adaptive workflow scheduling in grid computing based on dynamic resource availability. *Engineering Science and Technology, an International Journal*, 18(2):256–269, 2015.

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

[16] S. He, J. Wu, B. Wei, and J. Wu. 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)*, pages 571–577, 2021.

[17] J. Herrmann, L. Marchal, and Y. Robert. Memory-aware list scheduling for hybrid platforms. In *2014 IEEE international parallel & distributed processing symposium workshops*, pages 689–698. IEEE, 2014.

[18] A. Hirales-Carbajal, A. Tchernykh, R. Yahyapour, J. L. González-García, T. Röblitz, and J. M. Ramírez-Alcaraz. Multiple workflow scheduling strategies with user run time estimates on a grid. *Journal of Grid Computing*, 10:325–346, 2012.

[19] E. Kayaaslan, T. Lambert, L. Marchal, and B. Uçar. Scheduling series-parallel task graphs to minimize peak memory. *Theoretical Computer Science*, 707:1–23, 2018.

[20] S. Kulagina, H. Meyerhenke, and A. Benoit. Mapping large memory-constrained workflows onto heterogeneous platforms. In *Proceedings of the 53rd International Conference on Parallel Processing, ICPP 2024, Gotland, Sweden, August 12-15, 2024*, pages 305–316. ACM, 2024.

[21] U. Leser, M. Hilbrich, C. Draxl, P. Eisert, L. Grunske, P. Hostert, D. Kainmüller, O. Kao, B. Kehr, T. Kehrer, C. Koch, V. Markl, H. Meyerhenke, T. Rabl, A. Reinefeld, K. Reinert, K. Ritter, B. Scheuermann, F. Schintke, N. Schweikardt, and M. Weidlich. The collaborative research center FONDA. *Datenbank-Spektrum*, 21(3):255–260, 2021.

[22] B. Liang, X. Dong, Y. Wang, and X. Zhang. Memory-aware resource management algorithm for low-energy cloud data centers. *Future Generation Computer Systems*, 113:329–342, 2020.

[23] J. Liu, E. Pacitti, and P. Valduriez. A survey of scheduling frameworks in big data systems. *International Journal of Cloud Computing*, 7(2):103–128, 2018.

[24] M. J. Malik, T. Fahringer, and R. Prodan. Execution time prediction for grid infrastructures based on runtime provenance data. In *Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science*, pages 48–57, 2013.

[25] L. Marchal, H. Nagy, B. Simon, and F. Vivien. Parallel scheduling of dags under memory constraints. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 204–213. IEEE, 2018.

[26] M. Rahman, R. Hassan, R. Ranjan, and R. Buyya. Adaptive workflow scheduling for dynamic grid and cloud computing environment. *Concurrency and Computation: Practice and Experience*, 25(13):1816–1842, 2013.

[27] Y. Samadi, M. Zbakh, and C. Tadonki. E-HEFT: Enhancement Heterogeneous Earliest Finish Time algorithm for Task Scheduling based on Load Balancing in Cloud Computing. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 601–609, 2018.

[28] S. Sandokji and F. Eassa. Dynamic Variant Rank HEFT Task Scheduling Algorithm Toward Exascale Computing. *Procedia Computer Science*, 163:482–493, 2019. 16th Learning and Technology Conference 2019Artificial Intelligence and Machine Learning: Embedding the Intelligence.

[29] Z. Shi and J. J. Dongarra. Scheduling workflow applications on processors with different capabilities. *Future Generation Computer Systems*, 22(6):665–675, 2006.

[30] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[31] G. Wang, Y. Wang, H. Liu, and H. Guo. HSIP: A Novel Task Scheduling Algorithm for Heterogeneous Computing. *Scientific Programming*, 2016(1):3676149, 2016.

[32] C. Witt, J. van Santen, and U. Leser. Learning low-wastage memory allocations for scientific workflows at icecube. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 233–240. IEEE, 2019.

[33] A. Yano and T. Azumi. CQGA-HEFT: Q-learning-based DAG scheduling algorithm using genetic algorithm in clustered many-core platform. *Journal of Information Processing*, 30:659–668, 2022.

[34] Y. Yao, Y. Song, Y. Huang, W. Ni, and D. Zhang. A memory-constraint-aware list scheduling algorithm for memory-constraint heterogeneous multi-processor system. *IEEE Transactions on Parallel and Distributed Systems*, 34(4):1082–1099, 2022.