H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D3.3.
Scheduling Algorithms and Policies

Version 1.0

Date: July 13, 2023

Type: Deliverable
WP number: WP3

Editor: Hamid Fard
Institution: TUDA

Project co-funded by the European Union Horizon 2020 JTI-EuroHPC research and innovation programme and Spain, Germany, France, Italy, Poland, and Sweden		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	20/02/23	Hamid Fard	TUDA	Proposed the structure.
2	08/03/23	Hamid Fard	TUDA	Wrote the introduction section.
3	09/03/23	Hamid Fard	TUDA	Wrote the background section.
4	14/03/23	Hamid Fard	TUDA	Extended the literature survey section.
5	17/03/23	Javier Garcia-Blas	UC3M	Added malleability strategies of Hercules in-memory file system.
6	17/03/23	Marc-André Vef	JGU	Added Section 3.1 on revising file systems to be malleable.
7	20/03/23	Alberto Cascajo	UC3M	Updated malleability strategies in in-memory-FS
8	20/04/23	Njoud O. Al-Maaitah	UC3M	Added section 4.1 Malleable EASY backfilling scheduler
9	10/04/23	Hamid Fard	TUDA	Revised the deliverable; inserted new material; rewrote and reorganized the existing material to make the whole writing more coherent.
10	22/05/23	Guillaume Pallez	Inria	Added node stealing section.
11	31/05/23	Taylan Özden	TUDA	Added the I/O-intensity–aware scheduler.
12	06/06/23	Taylan Özden	TUDA	Added mathematical background and algorithms of the I/O-intensity–aware scheduler.
13	13/06/23	Taylan Özden	TUDA	Added evaluation of the I/O-intensity–aware scheduler.

Executive Summary

The compromise between parallelism and I/O congestion must be addressed to efficiently execute data-intensive applications on exascale computing systems. In the ADMIRE project, we target balancing computation and I/O for running compute- and data-intensive applications, mainly focusing on resource provisioning and job scheduling. In this deliverable, as the primary outcome of task Task 3.2 of ADMIRE, we present a set of innovative malleability policies that facilitate malleability operations for malleable jobs, computation, and I/O resources. Furthermore, we introduce two scheduling algorithms designed to reduce job makespan and enhance overall system utilization. A central idea behind our scheduling algorithms is to benefit from the potential of malleability to dynamically balance computation and I/O load distributions while running a combination of rigid and malleable jobs. As part of our work in the context of the Malleability Manager in Work Package 3, the proposed schedulers in this deliverable will be integrated into the Intelligent Controller component in Work Package 6 to provide efficient solutions for executing rigid and malleable jobs submitted by the users.

Contents

Chapter 1

Introduction

High-performance computing (HPC) is a distributed computing paradigm that benefits from integrating powerful computers, high-speed networks, and large data storage to perform complex calculations and extensive data processing at high speeds. HPC systems are used widely in solving complicated problems in different areas. Considering the software and hardware technologies used in HPC, we can categorize current HPC systems in cluster, grid, or cloud infrastructures [24]. Although our focus in the ADMIRE project (and in this deliverable) is on exascale HPC clusters, our achievements could also be used in the other categories above.

Moving toward exascale computing, the traditional trend of maximizing parallelism in HPC applications must be revised. We may need to compromise between the parallelism degree and I/O congestion to run large data-intensive applications in HPC, as the bandwidth could present a significant bottleneck to such problems. To this end, in the ADMIRE project, we elaborate on balancing computation and I/O while running compute- and data-intensive jobs. Therefore resource provisioning and job scheduling are key foci of our research in this project to ensure the efficient execution of jobs in HPC.

To provide the resources, scheduling objectives in HPC can be classified as user- and system-centric objectives [17]. For instance, decreasing response time and execution time of jobs are user objectives, while increasing throughput and utilization are system-centric objectives. Scheduling objectives are sometimes conflicting from different points of view. For instance, considering dynamic voltage and frequency scaling (DVFS) of compute nodes, to run a job faster, the cores should operate at a higher frequency, but this could cause increased energy consumption which may not be what the cluster providers prefer [4].

In this deliverable, we propose some novel malleability policies that allow malleable operations for malleable jobs, computation, and I/O resources. In addition, we propose two scheduling algorithms for decreasing the makespan of jobs and increasing system utilization. The core idea behind our scheduling algorithms is to dynamically balance computation and I/O load distributions for running jobs. Firstly, we introduce an enhanced version of the EASY backfilling scheduler that leverages the potential utilization of malleability operations. Subsequently, we introduce an I/O intensity-aware scheduler that considers the I/O intensity of jobs to minimize the occurrence of congestion on the shared parallel file system (PFS).

This deliverable is organized as follows. The next chapter provides the required background to better understand our contribution. We present our proposed malleability policies in Chapter 3. Chapter 4 explains our scheduling approaches, accompanied by an evaluation and analysis of the results. Finally, in Chapter 5, we draw our conclusion.

Chapter 2

Background

This chapter presents the required background to understand the deliverable better. First, we explain the scheduling problem in HPC. Then we demonstrate the concept of job malleability in an HPC environment. and in continuation, we survey the literature.

2.1 Scheduling in HPC

Usually, job management systems in HPC clusters function as batch systems, such as Slurm, IBM LSF/CSM, and PBS. In the batch systems, users first specify their jobs in job scripts, which include the applications, required resources, walltimes, etc., and then using these job scripts, submit their jobs to a queue and wait to complete the jobs. The component in batch systems responsible for choosing the jobs from the job queue and assigning the resources to them is called the job scheduler.

Scheduling jobs in HPC environments is one of the fundamental topics in theory and practice that impacts different aspects of HPC systems, from system utilization to user experience. Scheduling can be defined as a function that maps the set of jobs the users submit onto the available resources provided for running these jobs—both in space and time. Scheduling in HPC determines which resources and when should be allocated for running the jobs in the batch system. Then the scheduler is in charge of a two-dimensional division of resources to run the jobs in both time and space [15].

To optimize makespan, scheduling jobs in HPC is known as an NP-hard problem. Considering the desire of different stakeholders, scheduling problems might be even more complex. For instance, the users are interested in decreasing the execution time of their jobs and experiencing lower response time, while the system providers may care more about other metrics, such as throughput, utilization, and energy consumption of resources. Therefore to solve this problem, we inevitably need to find trade-off solutions to achieve the specified objectives.

Matthias Hovestadt et al. argue that job scheduling in HPC may follow two different approaches: queue-based and plan-based [22]. Queue-based algorithms schedule jobs only when free resources are available and make no resource assignments for the future beyond filling those free resources. In contrast, plan-based algorithms provide resource planning for all jobs in the present and the future. Compared to queue-based approaches, plan-based scheduling can provide better schedule solutions. But this comes at the cost of precisely estimating the execution times of jobs and a higher time complexity of the algorithm. Both queue- and plan-based scheduling approaches are well-studied in the literature. Still, production environments tend to employ queue-based scheduling algorithms because of their simplicity and lower sensitivity to imprecise wall-time estimates.

These two classes of scheduling algorithms represent two ends of a spectrum, characterized particularly by the degree of how far they plan into the future and how thoroughly they explore the solution space. Scheduling algorithms used in production can be considered a mixture of both classes. For example, backfilling, a mechanism often combined with queue-based algorithms, must ensure the jobs running in the future would not be delayed because of filling gaps with lower-priority jobs. Furthermore, plan-based algorithms may still consider a queue and assign resources according to their criteria, respecting the submission order and possibly compromising efficiency.

		when it decided?	
		at submission	during execution
who decides?	user	rigid	evolving
	system	moldable	malleable

Table 2.1: Parallel jobs classification. [17]

2.1.1 Backfilling

Backfilling is a common job scheduling mechanism in many batch systems, such as Slurm. Using backfilling, the scheduler could start lower-priority jobs before higher-priority jobs, given that the resource allocation of lower-priority jobs does not cause delaying the start time of the higher-priority jobs. This precondition is satisfied by the availability of the walltime of jobs specified by the user in the job scripts. Backfilling helps improve utilization and throughput of the cluster of resources as some small gaps may be used by the jobs with lower resource demands waiting in the job queue [45]. We should notice that although backfilling needs an estimation of the execution time of jobs, this estimation does not need to be precisely accurate. Surprisingly, even inaccurate estimates may yield better performance than cases we provide precise estimations of job execution times [46].

Backfilling has been proposed in different flavors, but the most widely used mechanisms are the extensible Argonne scheduling system (EASY) [28] and conservative backfilling (CONS) [34]. The scheduler of EASY uses aggressive backfilling, moving some of the small jobs to the head of the queue for using available gaps in the schedule plan, provided that they do not delay the first job in the queue. Conservative backfilling follows a stricter discipline such that the small jobs can move ahead only if they do not delay any job in the queue.

2.1.2 Scheduling policies

Scheduling policies are sometimes considered imprecisely equal to scheduling algorithms, but actually, they differ. In this part, to better distinguish scheduling policies, we explain what we mean when we use the term *scheduling policy*.

A batch scheduler can be configured to follow a set of policies. These policies tell the batch scheduler how to behave under different conditions. In other words, it guides the scheduler in allocating compute resources between users or workloads. For instance, we may define a policy as specifying the maximum runtime of a job in the queue or the maximum number of concurrent jobs for each individual user. These respectively mean that jobs longer than the specified time are not allowed to run, and the submitted jobs of a user would be rejected if the user has already reached the maximum number of concurrently running jobs. HPC administrators can usually adjust the scheduling policies in the batch system using predefined configuration commands.

In Chapter 3, we propose several malleability policies. These policies can actually be interpreted as scheduling policies because we are enforcing some malleability operations on malleable jobs under different conditions that will impact the availability of resources and decisions of the scheduler.

2.2 Malleability

As shown in Table 2.1, we can classify parallel jobs in four different categories—depending on how flexible the resource requirements are [17]. Rigid jobs always require a fixed set of resources. The resources of moldable jobs can be changed after submission until the job is started. Evolving jobs can be reconfigured after job submission and even during execution time. The user specifies the reconfiguration request in the job script or application definition. Malleable jobs can be reconfigured during execution time upon request of the scheduler.

As shown in Figure 2.1, shrinking and expanding are two possible operations to reconfigure the resources of malleable jobs. Of course, evolving jobs have the same reconfiguration operations, but respecting the classes defined in Table 2.1, the user specifies when and how the reconfiguration should occur. These operations

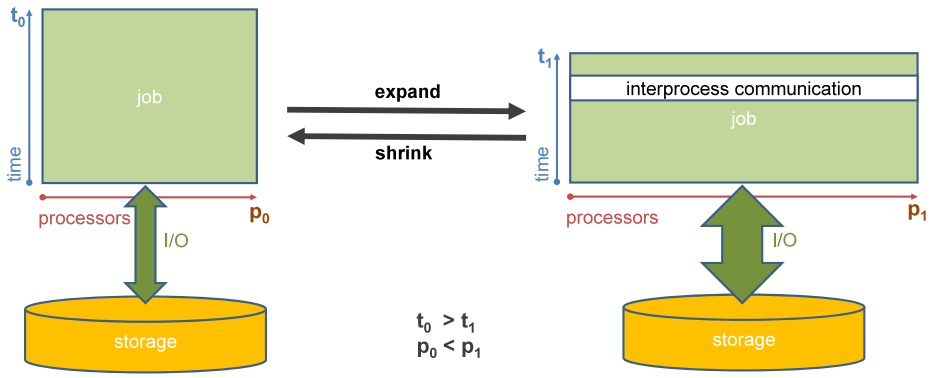


Figure 2.1: Malleation operations. Shrink and expand reconfigure the resources assigned for a malleable job.

provide opportunities for better use of resources and optimizing job executions. By shrinking, we may release part of the resources assigned for a malleable job that could allow us to start some new jobs waiting in the queue or reconfigure others that can benefit from these released resources. Expanding resources in malleable jobs could increase resource utilization and possibly improve job execution times. Shrink and expand operations are decided based on the trade-off between the reconfiguration overheads (e.g., data redistribution) and the gain obtained by applying the malleation operations. This trade-off is analyzed as part of the objectives defined in the ADMIRE proposal.

To realize malleability in HPC, we need to support it in the parallel runtime and batch system. We also need a communication mechanism between these two. Several runtime environments, such as Charm++ [25], could provide malleability in HPC. Malleability support has also been provided on the batch system level, such as in Torque/Maui [37]. In ADMIRE, we support malleability at different levels, from scheduling algorithms to MPI processes using FlexMPI [30] and ad-hoc file systems using GekkoFS [50] and Hercules [39]. We encourage the reader to consider the previously published deliverables D3.1 and D3.2 to understand better the malleability and the opportunities and challenges it brings.

2.3 Related work

The traditional scheduling approaches for rigid jobs in HPC consider neither the reconfiguration need of malleable jobs nor the efficient use of I/O resources for data-intensive jobs. In this section, we survey the related work considering these diversions. We study the scheduling approaches in three groups: scheduling multi-resource systems, data-intensive jobs, and malleable jobs.

2.3.1 Scheduling in multi-resource systems

Common scheduling approaches in HPC consider only compute nodes for scheduling decisions. But the ever-growing types of resources in HPC encourage researchers to consider the other involved resources such as storage tiers, GPUs, etc. Similarly, in the ADMIRE project, to balance I/O and compute performance, scheduler decisions do not only consider compute nodes but also take I/O resources such as the parallel file system (PFS) bandwidth into account. Below, we study some of the efforts in this area.

Seung-Hwan Lim et al. introduced a job model [29], when multiple shared resources may be involved in job processing. They analyzed job slowdowns due to contention for multiple resources in a system, referred to as the dilation factor. The authors observed that contention for multiple resources makes the dilation factors of jobs non-linear. This job model is characterized by the vector-valued loading statistics and the dilation factors of a job set, given by a quadratic function of their loading vectors.

A heuristic called Tetris [19] packs the jobs based on CPU, memory, disk, and network bandwidth resource requirements to maximize the task throughput. Their multi-resource scheduling mechanism improves both average job completion time and makespan. The core idea behind this heuristic is serving jobs with less remaining work by a higher priority. They implemented their approach in the YARN framework.

A job scheduling framework named BBSched [14], which schedules CPUs and burst buffers, formulates the problem as a multi-objective optimization problem and then solves it using a multi-objective genetic approach. The objective is to enhance the utilization of multiple resources by offering a Pareto set to facilitate decision-making and optimization. The authors presented that their approach is general and can be employed for any resource type except the local SSDs that they considered for their case study.

Hongyang Sun et al. theoretically and experimentally compared list scheduling and pack scheduling algorithms for a set of moldable tasks constrained by processor cores and high-bandwidth memory [44]. In list scheduling, the jobs are executed based on a priority list while pack scheduling divides the jobs into a set of packs such that the jobs in the lower-order priority packs cannot be started before any higher-order pack. They also observed that pack scheduling competes or outperforms list scheduling in practice, particularly in multi-resource scheduling scenarios. Inspired by their results, they designed a strategy to indirectly solve the scheduling problem via a transformation to a single-resource-type problem to decrease the makespan.

The problem of multi-resource scheduling is modeled as multi-capacity bin-packing by Mehdi Sheikhalishahi et al. [41]. They proposed multi-capacity-aware scheduling heuristic for every job that applies to a group of jobs to address the capacity imbalance. This approach demonstrates performance improvements for wait-time and slowdown metrics, resource utilization, and energy efficiency.

2.3.2 Scheduling of data-intensive jobs

Ever increasing volume of data in modern HPC applications drives scheduling algorithms dealing with the data intensity of applications. Moreover, emerging new storage technologies like burst buffers must be considered for the efficient execution of these applications. This section reviews some of the prominent works in this area.

Stephen Herbein et al. extended FCFS and EASY backfilling algorithms by integrating I/O awareness to reduce PFS contention [21]. Their approach benefits from the available burst buffers in the HPC system to absorb random I/O bursts and reduce I/O contention at the PFS level. They incorporate the knowledge of the I/O subsystems and I/O awareness into the Flux resource management system [1].

Mihaela-Andreea Vasile et al. [49] proposed a hybrid scheduling approach considering job and resource clustering for batch jobs and workflow applications. They first partition the available resources into separate clusters. Then assign the jobs to a proper resource cluster and finally employ some classical scheduling algorithms to schedule the jobs in each group. The shortest-job-first and earliest-deadline-first algorithms are used for independent jobs, and the earliest-time-first and modified-critical-path algorithms are considered for dependent jobs modeled as directed acyclic graphs (DAGs).

Francieli Boito et al. offered an I/O scheduling approach [55] to tackle the I/O congestion in concurrent I/O of different jobs on a shared bandwidth. They grouped the jobs in separate sets based on the mean time between two consecutive I/O phases of each job. The jobs in a single set cannot do I/O operations concurrently, and concurrent I/O cannot access the shared bandwidth exclusively. They controlled the congestion using a heuristic to prioritize the sets and then choose a set for doing I/O.

Shu-Mei Tseng et al. demonstrated the cause and consequence of the different sources of interference for asynchronous I/O operations in HPC systems [47]. The authors focused on studying contention for core access and main memory bandwidth. They studied the performance overhead of interference caused by different sources, including the application behavior, number of concurrent I/O threads, I/O strategies (including sendfile, read/write, and mmap/write), and CPU architecture.

To reduce the I/O contention in burst buffers, Weihao Liang et al. introduced a scheduling strategy for running concurrent data-intensive jobs in HPC systems equipped with burst buffers [27]. They provided some modeling and analysis for the I/O behavior of applications, burst buffers, and I/O contention in such resources. Their algorithm chooses the most under-utilized burst buffer nodes for each new job entering the system to minimize the I/O contention from other active jobs. They evaluated their algorithm using the average of job efficiency, waiting time, and system utilization and observed improvement in both the job performance and the system utilization.

Using machine learning models, Daniel Nichols et al. presented a model [35] to predict the variation of jobs in the scheduler queue. They used this model in their job scheduler, called RUSH, to reduce performance variability and improve overall system utilization. They implemented their scheduler by modifying the Flux [1]

framework.

Zhou Zhou et al. [56] translated I/O congestion into a classical scheduling problem by monitoring the system state and job activities. Once I/O congestion is detected, all concurrent I/O requests will be scheduled or coordinated like normal jobs on HPC systems. They showcased two approaches addressing the I/O-congestion problem, targeting different objectives: I/O-aware batch scheduling and bandwidth allocation on I/O servers.

To reduce cross-application I/O interference in HPC systems, Matthieu Dorier et al. proposed a framework [12] considering the overall machine-wide efficiency. They assumed known application size, I/O behavior, and interference time as the main factors impacting the I/O interference for concurrent applications. The authors studied three coordination strategies for cross-application communications: interfering, serializing, and interrupting. Their framework exploits a combination of these sub-optimal strategies to achieve a desirable dynamic selection, especially when applications present different I/O behavior and requirements. For instance, the framework may instruct an application to pause its I/O activity in favor of another application and resume afterward or still access the file system in contention with another application.

We should notice that integrating I/O scheduling into traditional job scheduling could better support running data-intensive applications in HPC. For instance, Ana Gainaru et al. analyzed Argonne's Mira system and observed that burst buffers could not absorb I/O bursts at all times [18]. To cover this problem, they introduced several I/O scheduling algorithms to mitigate congestion caused by I/O interference of concurrent applications accessing a shared parallel file system. Each proposed scheduler targets different objectives, such as fairness and system performance, which the administrator could choose. Their schedulers need a global view of the system and of the past behavior of all applications running at a given time. I/O scheduling is a rich research area, and since it will be investigated deeply in the deliverables provided by WP4, we ignore surveying the related work of that domain in the current deliverable.

2.3.3 Scheduling of malleable jobs

By adapting to load variations during runtime, malleable jobs could improve resource utilization and the overall performance of the HPC system. This section reviews some efforts in the domain of scheduling malleable jobs.

Gladys Utrera1 et al. [48] extended FCFS scheduling to support malleability. They assumed that virtual malleability could apply to all jobs limitless. This means that they assumed jobs could adapt to changes in the number of CPUs at runtime, preserving the original number of processes. Although this assumption may be naive, the results achieved compared to FCFS with EASY backfilling are impressive enough to observe the power of malleability to improve system performance. A similar work [23] analyzed the quality of FCFS schedule solutions considering having a combination of rigid, moldable, and malleable jobs in the system. They statistically and experimentally showed how utilization and average response time could be improved compared to having no flexibility in running jobs.

An API called DROM has been proposed by Marco D'Amico et al. [10] that enables the scheduler in Slurm to communicate with applications that can adapt at run time to changes in the computing resources. Based on this, they presented malleability-enabled backfilling scheduling [11]. The algorithm aims to reduce the average slowdown and response time of jobs. This algorithm shrinks the resources of running jobs to make room for jobs that will run with fewer resources only when the estimated slowdown beats the static approach.

Suraj Prabhakaran et al. introduced a job scheduling strategy [37] to schedule a hybrid combination of rigid, malleable, and evolving jobs in the HPC system. By combining dependency analysis and backfilling approach, the scheduler chooses the proper malleation operations to improve the system's performance. Then it applies an equipartitioning strategy for fairness with unused resources in the previous analysis. They included their algorithm in an extension of the Torque/Maui batch system to support malleability.

Chapter 3

Malleability policies

In this chapter, we discuss our malleability policies proposed in ADMIRE. Respecting the target of this deliverable which focuses on scheduling algorithms and policies, we believe that malleability policies can be interpreted as scheduling policies because malleation operations on both compute nodes and I/O resources affect scheduling decisions. To this end, in this chapter, we first discuss the policies proposed for the malleability of compute nodes, and then we discuss malleability policies in I/O. We need to remind the reader that malleability in the storage system will be discussed in more detail in our next deliverable (D3.4) in WP3.

3.1 Node stealing policy

When discussing malleability, one of the open questions is the availability of resources for malleation operations: to speed up the execution of a job by assigning extra nodes to the job, how should we provide these nodes? There are many solutions ranging from keeping a pool of available nodes (but reducing *de facto* utilization of the machine) or taking a node from another running job (*node stealing*).

As a use-case for the development of such node stealing policies, we have studied [13] a worst-case scenario, one where jobs are not malleable and where stealing a node interrupts the job from whom the node was stolen (we call this job the *victim*). This can be seen as a *failure* from the *victim* perspective since it is interrupted. In addition, we expect the theft to occur at a time that is hard to predict, for instance, when I/O congestion happens. Hence this could be modeled as a system failure. This idea could be extended to be also used for malleable jobs.

Hence in the first study, we have chosen to see it from a fault-tolerance angle, including various resilience mechanisms. There are several decisions to explore:

- **Which job to interrupt?** Clearly, small jobs (the jobs assigned to a few nodes) are good candidates because they are easier to re-schedule. But interrupting a small job whose waiting time is already high may not be fair to the owner of that job, so trade-offs between different optimization metrics must be achieved.
- **When to interrupt?** The simplest solution is immediately when the I/O congestion is observed or after the failure, but the interrupted job will lose the work done since its last checkpoint. Alternative solutions are to wait for a checkpoint before the interruption or immediately enforce a proactive checkpoint, depending upon what is possible.

In our work, we were able to make the following contributions [13]:

- A thorough description of the problem and how to measure its usefulness;
- The design of SFSJ (*Steal From Small Jobs*), a strategy which chooses the job to interrupt among those with the smallest number of nodes and, if ties, with the shortest execution time so far;
- An evaluation of SFSJ in a simulated framework based upon trace-based scenarios.

In the following we focus on highlighting and motivating the problem. We start by presenting a toy example.

3.1.1 Toy example and description

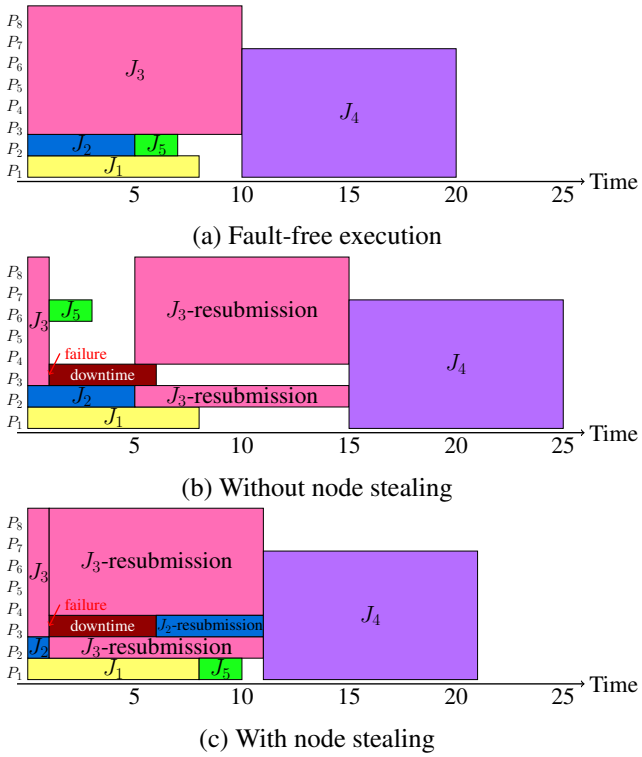


Figure 3.1: Toy example, job details in Table 3.1. Subfigures (b) and (c) assume that a failure occurred at $t = 1$ on P_3.

This section uses a toy example to detail the various impacts of node stealing. It provides insight into the decision made throughout this Section. Consider a platform with eight nodes. Five jobs are released at time $t = 0$: see Table 3.1 and Figure 3.1 for details on these jobs. Since all five jobs are released simultaneously at time $t = 0$, we can assume that the scheduler has broken ties so that the jobs are scheduled in the order J_1, J_2, ..., up to J_5.

At time $t = 0$, the scheduler starts J_1 on P_1, J_2 on P_2, and J_3 on P_3 to P_8. It reserves P_1 to P_6 for J_4 at $t = 10$. At time $t = 5$, it backfills J_5 on P_2 since it will not delay J_4. Figure 3.1(a) depicts the fault-free execution.

We consider now that the platform will experience failures. To simplify the example, jobs are not checkpointed and can resume immediately after a failure if there are available nodes, meaning that we neglect any recovery cost. Each node's downtime (rejuvenation time) is $D = 5$, meaning that a node struck by a failure at time t is up again at time $t + 5$. Suppose then that a failure strikes P_3 at $t = 1$. Figure 3.1(b) depicts the standard scenario. J_3 fails at $t = 1$ and is now the job with the highest priority for rescheduling. There are only five free nodes at $t = 1$, and this holds true until $t = 5$. Hence J_3 is scheduled for execution at $t = 5$ on nodes P_2 and P_4 to P_8 (since P_3 is unavailable until $t = 6$ due to downtime). Now J_3 completes at time 15 and J_4 completes at time 25. Using backfilling, J_5 is scheduled at $t = 1$ on one available node (P_6 in the figure). Interestingly, we see that the smallest job has finished earlier in the presence of a failure than without one, while the large jobs have suffered the most from the failure.

What happens instead if we steal a node when the failure strikes P_3 at $t = 1$? We represent this new scenario in Figure 3.1(c). At $t = 1$, we steal P_2 and thereby interrupt job J_2. Job J_3 can re-execute immediately on nodes P_2 (replacing P_3) and P_4 to P_8. J_3 now finishes at time 11. Then J_2 has the highest priority and can re-execute on P_3 when is up again at time 6. Now J_2 completes at $t = 11$. Then J_4 is scheduled at time 11 and completes at time 21. Using backfilling, J_5 executes on P_1 when it becomes available.

Table 3.1 reports some statistics about the flow (i.e. time between submission and execution) of the five jobs in the different scenarios without or with node stealing. We see that the flows of the large jobs J_3 and

Table 3.1: Job information for the toy example.

id	Release time	Job size	Job length	Flow without node stealing	Flow with node stealing
J_1	0	1	8	8	8
J_2	0	1	5	5	11
J_3	0	6	10	15	11
J_4	0	6	10	25	21
J_5	0	1	2	3	10

J_4 have decreased at the price of increasing the flow of the small jobs J_2 and J_5. The maximum value of the flow has decreased from 25 to 21. However, its mean value has increased from 11.2 to 12.2. This is interesting as it shows that the mean flow is highly influenced by small jobs, while these jobs are not the most critical jobs on HPC platforms. Another widely used metric is the weighted mean flow, where the mean is weighted by the number of nodes of each job. Here, the weighted mean flow without node stealing is $(1 \times 8 + 1 \times 5 + 6 \times 15 + 6 \times 25 + 1 \times 3)/(1 + 1 + 6 + 6 + 1) = 17$, while the one with node stealing is 14.733.

We also see that the total idle time of the eight nodes has decreased. Altogether, node stealing seems quite beneficial here. Beyond this toy example, a major contribution of this work is to assess the usefulness of node stealing in various realistic execution scenarios.

3.1.2 Node stealing design

This section provides a high-level description of the classic conservative backfilling strategy used by batch schedulers (Section 3.1.3) and details how to extend it to implement node stealing (Section 3.1.4).

3.1.3 Baseline strategy

First-Come First-Serve (FCFS) is a simple approach to submit jobs on parallel supercomputers. However, FCFSoften leads to a waste of resources: when there are not enough free nodes for the next job, these free nodes remain waiting until additional nodes become available. A widely-used solution is to use non-FCFS polices, i.e., to allow for a (limited) reordering of the jobs in the queue. *Backfilling* schedulers [33] have been proposed to allow small jobs further away in the queue of waiting for jobs to be processed whenever there are enough resources for them. Backfilling may lead to delay some previously allocated jobs, hence it must be controlled so as to guarantee that large jobs will get processed eventually. This is why, in the *conservative backfilling* algorithm, short jobs are moved ahead only if they do not delay any previous job already scheduled.

When a failure hits the system, the remaining part of the job that failed is put back into the scheduling queue, with the highest priority. Depending upon the absence or presence of a resilience mechanism, the remaining part of the job can represent either the whole job or the fraction of the job after the last checkpoint. The schedule is then recomputed with all jobs that have not started yet. If there are multiple jobs that have failed in the queue, they are sorted by non-decreasing arrival time. In the next section, BASELINE will denote this conservative backfilling scheduling strategy.

3.1.4 Node stealing protocol

Node stealing should be seen as a feature that can be added on top of any batch scheduling strategy. In the work we did [13], we add this feature on top of BASELINE scheduling. The core idea is the following: when a failure hits a job (say job J_1), and if there is no (free) node available at the time of a failure, then we select another job (say job J_2) which we *interrupt*. A node from job J_2 is allocated to job J_1, so that job J_1 can resume its execution immediately, either from its last checkpoint if any, or from scratch. Job J_2 is then marked as failed, and it is restarted, again from its last checkpoint if any, otherwise from scratch. The schedule is then

recomputed with the following priorities: (high) job J_1; (medium) job J_2; (low) other submitted jobs in the order of the underlying scheduling algorithm (here BASELINE).

We focused on a single node stealing strategy and select the job to interrupt (called *victim*) using the following procedure: *among all running jobs that use the fewest nodes, we select the one that has been submitted the latest*. In other words, the selection criteria are job size first, and job release time to break ties. If no victim job is found with fewer nodes than the failed job, node stealing is not activated. We call this strategy SFSJ (*Steal From Small Jobs*). Other node stealing strategies are discussed and evaluated in [13], along with the possibility to take a proactive action, i.e., checkpoint the job chosen to be interrupted before actually interrupting it.

We point out that BASELINE and SFSJ behave exactly the same when a free node is available when a job is struck by a failure. Both strategies have the failed job re-submitted with high priority, and therefore start re-execution immediately. However, when no free node is available when a job is struck by a failure, the strategies differ: BASELINE lets the failed job wait until enough resources become available, while SFSJ interrupts another job to be able to restart the failed job immediately.

The next step for this project will include malleability. In this situation, instead of interrupting a job, node stealing will solely increase its execution time.

3.2 Malleability policy in the Hercules ad-hoc on-memory file system

Ad-hoc file systems have been proposed as a feasible solution for bursting local storage resources and adapting the storage level to the needs of any particular applications [39,50]. They enable the creation of a temporary file system that adapts to the application deployment in the HPC environment based on profiling of the applications or user hints. However, even assuming that the deployment of the ad-hoc file system fits well with the initial application's I/O needs, the I/O behavior of an HPC application might change during the different execution phases. Thus, I/O requirements will change. Examples would be workflows, machine learning applications, or large-scale simulations where data output depends on the simulation steps [40].

Applying malleability techniques, ad-hoc file systems would be a possible solution to adapt the storage system to the I/O needs of the different phases of an application. Malleability would allow to extend or shrink the ad-hoc files system deployment at runtime following application I/O demands [9,42]. This section presents the design, implementation, and evaluation of malleability techniques in the Hercules distributed ad-hoc in-memory file system. Those techniques allow adapting (expanding or shrinking) the number of data nodes following a desired QoS metric. Our approach eliminates the necessity of migrating large amounts of data between data nodes. We are based on ruled-based distribution policies, which modify the placement mechanism with low impact. Rules are orchestrated in terms of predefined thresholds. In the developed prototype, malleability techniques are applied to each deployment of the Hercules systems; preliminary evaluation results show the feasibility of our solution.

We have included some malleability techniques in Hercules to facilitate dynamic modifications of the number of data nodes at runtime. Hercules can expand or shrink the number of data nodes to increase or reduce the I/O throughput of the application based on its I/O needs. In any ad-hoc deployment of Hercules, when a dataset is created, we store the list of servers storing data in the metadata servers. That way, we can always query where Hercules should write/read the blocks for each dataset by applying the data distribution policy defined for that ad-hoc deployment. In addition to the list of servers, we store the number of servers storing the dataset to avoid extra operations with the list of nodes. This reduces the quering time at block mapping.

3.2.1 Enabling malleability operations

In Hercules, malleability operations can be started by two alternative sources: external controller or internal heuristic. We provide an API to enable an *external controller* to pass as an argument the new number of servers and the list of new data nodes to be added/removed from a Hercules ad-hoc deployment. In this case, Hercules will expand/shrink that deployment following the controller commands. Additionally, Hercules provides an *internal heuristic* to determine whether a malleability operation should be carried out. In our system, an application can define a recommended I/O throughput (RIO) for the I/O system. As an application executes,

Hercules is tracking the actual throughput provided by the I/O system to the application (AIO), with three possible results: near the RIO (N_{io}), below the RIO (B_{io}), and above the RIO (A_{io}). The distance to the throughput is currently computed using a time series of the throughput obtained by the write/read throughput of the consecutive operations of the application on the different datasets. The distance for an I/O operation is computed as the difference between the RIO and the AIO:

$$d_i = AIO_i - RIO_i, \forall\, i\ 1..n \tag{3.1}$$

We use a sliding window with a size *k* to decide which is the behavior of the system based on the average distance for the sliding window values:

$$avg_d = average(d_j..d_{j+k}) \tag{3.2}$$

We also used a predefined throughput tolerance metric (T_{io}) that determines the next action to be carried out in terms of commissioning or decommissioning:

$$N_{io} = 0; B_{io} = 0; A_{io} = 0; \tag{3.3}$$
$$if\ avg_d \in [RIO - T_{io}, RIO + T_{io}] \rightarrow N_{io} = 1 \tag{3.4}$$
$$if\ avg_d < (RIO - T_{io}) \rightarrow B_{io} = 1 \tag{3.5}$$
$$if\ avg_d > (RIO + T_{io}) \rightarrow A_{io} = 1 \tag{3.6}$$

The sliding window size can be adjusted to fit the behavior of applications with changing I/O demands. Anyway, after executing the previous equations, Hercules will take the possible decisions:

- Keeping the same number of servers ($N_{io} = 1$),
- On-line commissioning for expanding the file system servers ($B_{io} = 1$), or
- Off-line decommissioning for shrinking the file system servers ($A_{io} = 1$).

Following the former definitions, a Hercules dataset can be in three stages: *commissioning, decommissioning, or ready*, which is also reflected in the metadata entry. The initial status is always ready.

3.2.2 Expanding the Hercules file system

To expand the deployment of Hercules, we have designed an online commissioning mechanism for increasing the file system servers that avoid blocking I/O operations while resources are being assigned. When B_{io} is activated, Hercules asks for new server nodes to the resource manager of the system (RMS) and keeps on working until the resources are available; the deployment status is set as *commissioning*.

Once the resources have been acquired, the new servers are connected to the ad-hoc deployment, and they will be used for new datasets that will include in the metadata the new number of servers and the extended list of servers; the deployment status is set as *ready* and new datasets will start using the extended deployment. Already existing datasets keep on using the servers previously allocated for the existing blocks and will start using the extended deployment for new blocks. An extended dataset is defined as a list of intervals ans ranges that determines the location of the blocks (mapping heuristic). It will usually be small, as we do not foresee having many malleability operations on an ad-hoc deployment.

One important question is to calculate the number of servers to be added to the current deployment. Again, if an external controller indicates the number and the list, Hercules will just add those new metadata / data servers to the current deployment. If our internal heuristic is used, the distance metric is applied to calculate the number of new servers to be requested from the RMS. More specifically, we use the average distance to compute an estimation of the throughput to increase the number of servers.

Figure 3.2 shows the ready and commissioning stages of the malleation process. As you can see, in the ready stage Hercules works with four file system servers (see $DN_{1..N}$), and by using the ROUND ROBIN data

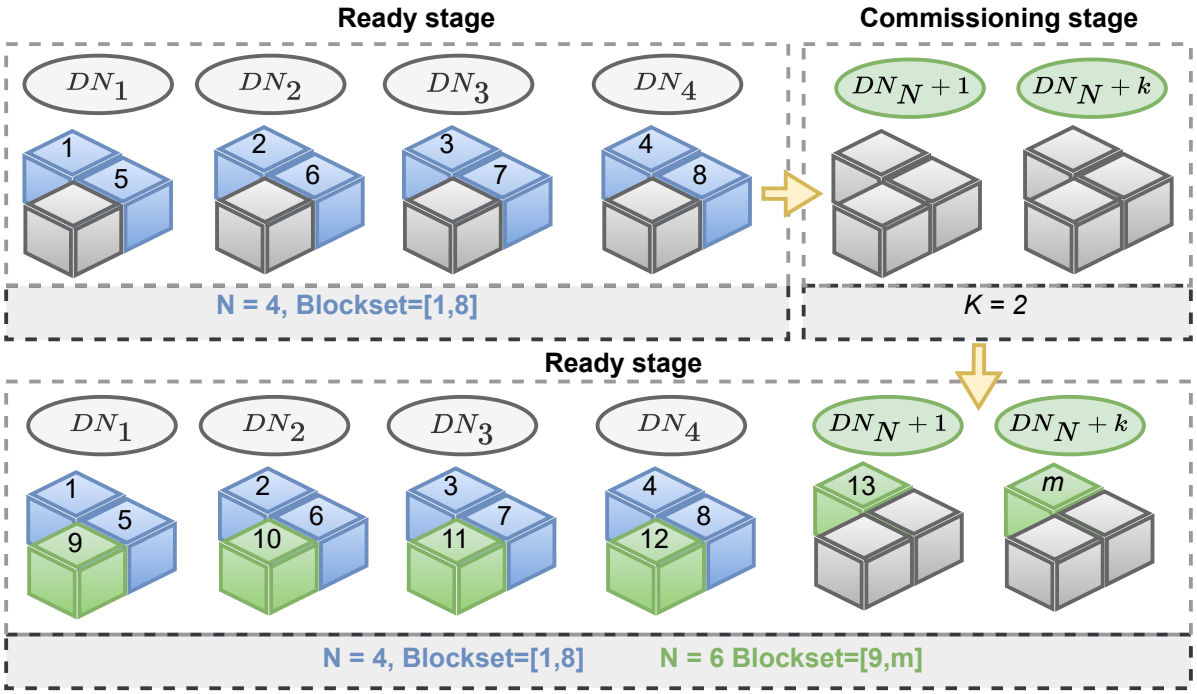


Figure 3.2: Extending a Hercules ad-hoc deployment with a ROUND ROBIN policy for blockset distribution.

distribution policy, the datasets 1 to 8 are distributed among them. Then, after the commissioning stage, two extra file system servers are added (see $DN_{N+1..N+k}$), which are used by Hercules following the same policy. This mechanism allows using the new servers without the overhead of migrating the data unless absolutely necessary. The only thing to do is to discriminate block location considering the interval associated with the servers commissioned when the block was created. We are already working to provide an offline migration mechanism to consolidate formerly existing datasets to use all nodes of an extended deployment. This mechanism will be executed if the application makespan is long enough to compensate for the migration overhead.

3.2.3 Shrinking the Hercules file system

Figure 3.3 depicts the execution workflow in the case of an off-line decommission. Once the target data node is selected (red box in the figure), datasets using this data node are marked as being in the decommissioning state, so updates are blocked until the decommission/data balancing has been completed. Data balancing is needed to speed up decommission operations and to stabilise the duration and performance of all re-scaling operations.

Malleability follows a bottom-up approach. Once decommissioning is started, metadata nodes notify clients that both the number and layout of data nodes have changed. Then, clients wait up to receive a final message informing them that decommission/data balancing is completed. During this procedure, data nodes migrate data blocks to the new corresponding data node. This data redistribution aims to reduce data staging costs and also maintain the data distribution policy. Hercules provides two alternatives for performing data balancing: 1) a total redistribution of data blocks that requires a costly data movement between all data nodes or 2) partial redistribution, which aims to minimise data movement in case of dealing with small files (as shown in Figure 3.3).

3.2.4 Experimental evaluation

In this section, we describe the experiments conducted to evaluate Hercules's performance, the evaluation environment setup, and the results obtained from the tests made. The hardware used to carry out the experiments consists of a 64-nodes cluster running Ubuntu 20.04.5 LTS. Each node is equipped with two processors of type

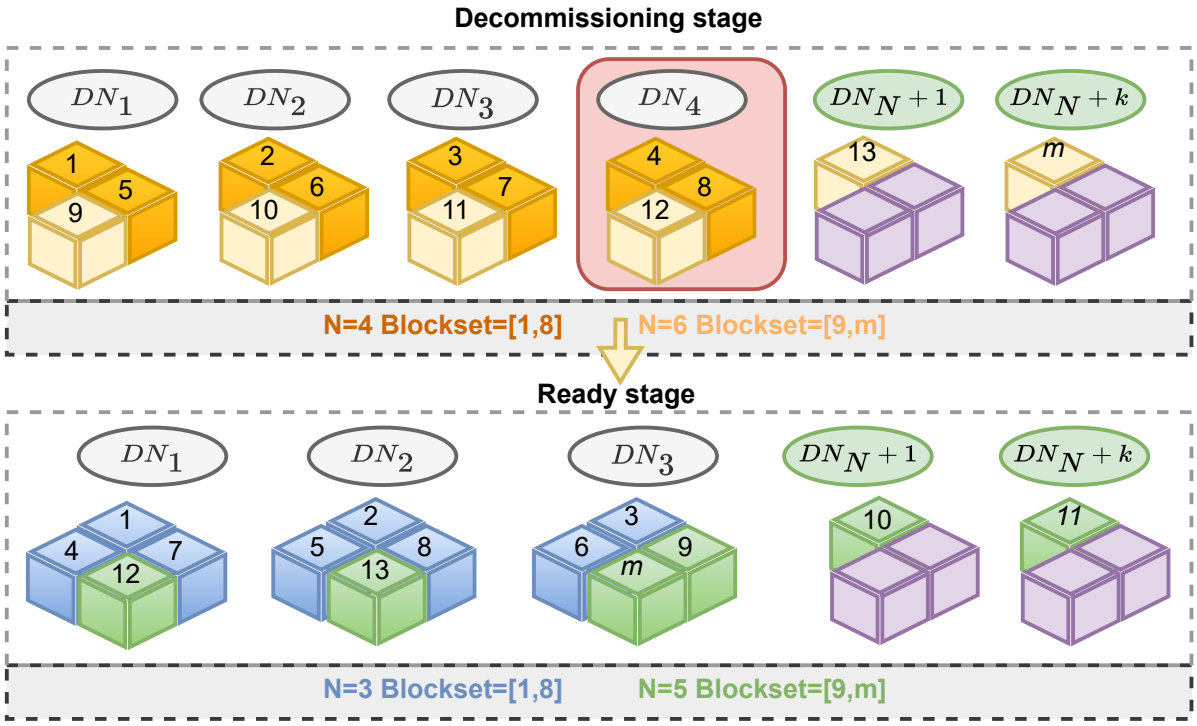


Figure 3.3: Shrinking a Hercules ad-hoc deployment. Blue and green blocks represent the previously extended dataset and purple blocks correspond to a different dataset.

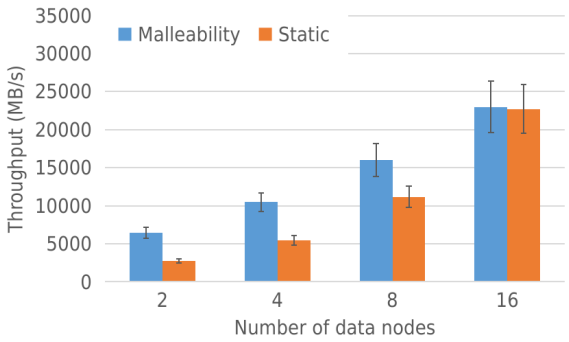
Intel Xeon CPU E5-2697 v4 16-Core with a total of 32 physical cores and a clock speed per core of 2.6 GHz. The network topology is created with three switches conforming to a fat-tree network with two levels. All the compute nodes are connected through the Intel Omni-path network reaching peak performance of 100 Gbps. The software employed is UCX 1.15, Open MPI 4.1 and *glib*. UCX exposed OPA network using *ibverbs* library, reaching a similar bandwidth compared with the native Open MPI installation.

Experimental results were obtained using the IOR benchmark, a widely-used solution for measuring I/O performance at scale, and IO500 [8], a benchmark suite bundled with execution rules targeting throughput and metadata performance. The evaluation metrics shown in this deliverable correspond with the average value of 10 consecutive executions. This subsection shows the evaluation results of comparing Hercules doing malleability operations (online commissioning) against a static deployment using 2 to 16 data nodes. We performed two sets of experiments classified as weak and strong scalability.

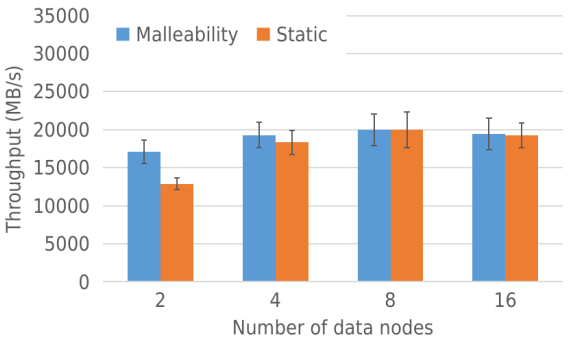
Weak-scaling evaluation. In those experiments, the resulting file size increases according to the number of clients deployed, since every client is writing 100 MB in a shared file.

We have evaluated online commissioning (see Section 3.2.2) and the malleability has been configured to use up to 16 data nodes. For example, in a configuration of 4 data nodes, a static configuration will perform all the operations with this value, but with the malleability configuration this value is taken as a lower bound (the initial data nodes) and then, during execution time, the system will allocate more nodes (with a maximum of 16) depending on I/O needs.

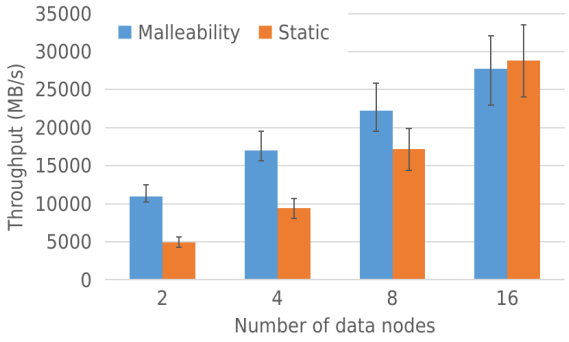
Figures 3.4a and 3.4b plot the throughput for both write and read operations using a block size of 256 kB. In general, for write operations, Hercules performs better when using *malleability* compared with a corresponding *static* configuration. When we set the lower bound to 2 data nodes, *malleability* configuration has a performance gain of 57.54% in the best case and 1.10% in the most limiting case (16/16). We can observe that when the number of data nodes increases, the difference between the throughput under malleability and throughput under static conditions decreases, as we are closer to the number of available data nodes. When we reach the



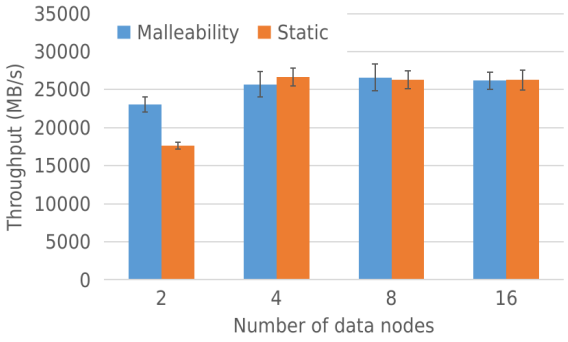
(a) Write. 256 kB block size.



(b) Read. 256 kB block size.



(c) Write. 1 MB block size.



(d) Read. 1 MB block size.

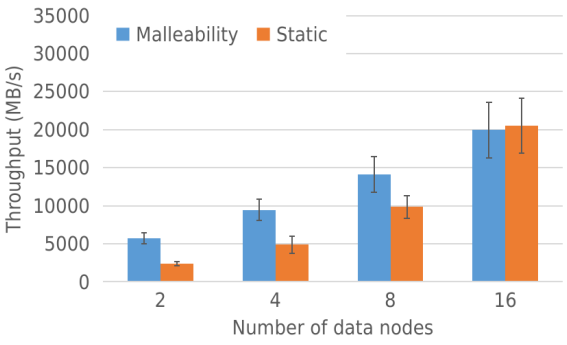
Figure 3.4: Weak scalability evaluation when fixing 16 compute nodes and 1 process per compute node.

maximum number of nodes, the system lacks resources to be expanded, thus malleability generates a small overhead. We have forced this situation when comparing a 16/16 deployment, appreciating an overhead of 1.10% due to malleability operations. For read operations, we can observe a different behaviour; the performance gain is 24.48% and 4.99% for 2 and 4 data nodes respectively, for 8 data nodes there is no difference (less than 0.0003%) and in the last one malleability has a gain of 1.12% for a 16 client/16 servers deployment. This is due to the data locality in the experiments and cannot be associated with malleability effects. Regarding read operations, we saturate the network with a small number of data nodes.

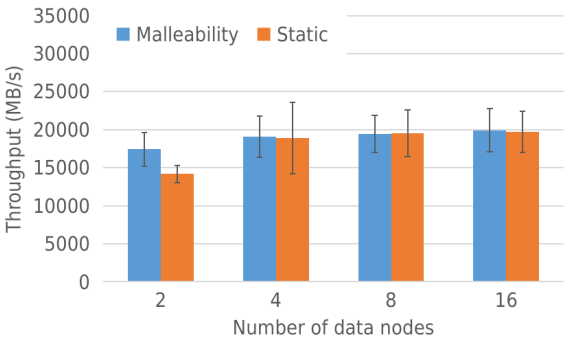
Figures 3.4c and 3.4d show the throughput for write and read operations using a block size of 1 MB. In both situations (malleability and static) we can observe an increment in the general throughput compared with a 256 kB block size. For write operations, Hercules has a performance gain from 54.72% to 22.96% with 2 to 8 data nodes and a loss of 3.87% with 16 data nodes, again caused by the overhead of implementing malleability operations. In the case of read operations with 2 data nodes, Hercules obtains a performance gain of 23.53%, getting the best performance with 8 data nodes with a performance gain of 1.26%.

Strong-scaling evaluation. For strong scalability tests, we set the size to 1 GB and divide it between the number of clients to get how many MB should be written per client.

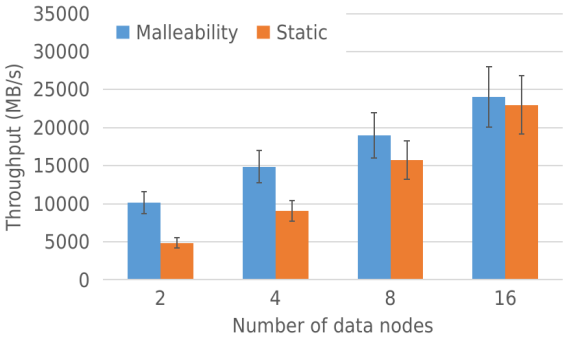
Figures 3.5a and 3.5b show the throughput for write and read operations using a block size of 256 kB. For write operations, we can observe a performance gain from 57.83% to 30.12% with 2 to 8 data nodes when using *malleability*. Additionally, we get an overhead of 3.02% when there are 16 data nodes compared with the *static* situation. Whereas, for read operations, we can see that with only 2 data nodes, Hercules implementing *malleability* experiences a notable performance gain of 18.64%, while for 4 and 16 data nodes the performance gain is 1.10% and 0.85%, respectively, and for 8 data nodes there is a loss of 0.38%. Figures 3.5c and 3.5d show the throughput for write and read operations using a block size of 1 MB. For write operations, we can depict the same behaviour compared with the 256 kB block size configuration, with the difference that when using a bigger block (1 MB) the overall throughput increases on average by a factor of $1.47x$. In this experiment, *malleability* has a performance gain from 51.98% to 4.42% for 2 to 16 data nodes, respectively. For read operations, we get an average speedup of $1.10x$ compared with the 256 kB block size's configuration, obtaining a performance



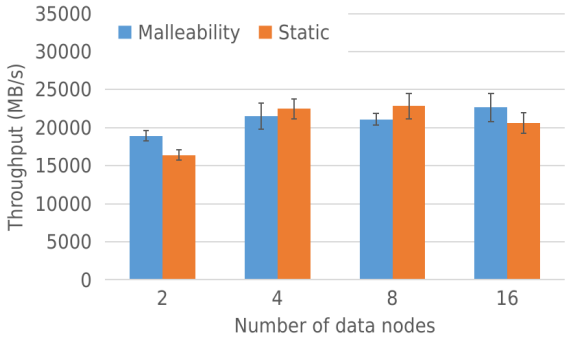
(a) Write. 256 kB block size.



(b) Read. 256 kB block size.



(c) Write. 1 MB block size.



(d) Read. 1 MB block size.

Figure 3.5: Strong scalability evaluation when fixing 16 compute nodes and 1 process per compute node. Shared file of 1 GB.

gain of 13.27% and 9.07% with 2 and 16 data nodes, respectively.

3.3 Revising ad-hoc user space file systems to be malleable

Besides investigating malleability policies in Hercules, other challenges arise from the fact that both file systems (GekkoFS and Hercules) are running entirely in user space. While user space file systems are much simpler and less time-consuming to develop, these challenges can hinder the scope of malleable options in such file systems. In the following, we discuss an extended file system architecture that is useful for all user space file systems that use an interception mechanism (as is the case with GekkoFS and Hercules, for example), offering further opportunities for I/O malleability.

3.3.1 Motivation

Ad-hoc file systems are often accessed by a single application and can be optimized to the application requirements. This can include data distribution patterns that match how an application accesses its data, changes to cache consistency guarantees, or complete modifications of I/O protocols that relax POSIX expectations. Recent studies suggest that not all I/O functions or strong consistency file system semantics are required by applications [26, 52, 53], making such mechanisms helpful for boosting performance. User space implementations are particularly effective for these use cases since they can incorporate these optimizations into the file system without kernel restrictions.

Nevertheless, these file systems and their configurations are still *static* and must be determined when the file system is launched. With more intelligent HPC systems that can react to available resources, file systems should become *malleable*. This means they can dynamically change their configurations and algorithms, such as adjusting consistency guarantees or the number of used I/O servers. However, transitioning from a static to a malleable file system has its challenges and may require significant architectural design changes.

Since user space file systems are not restricted by the kernel, they can offer configuration options to modify their behavior. For instance, GekkoFS offers options such as selecting a specific metadata backend, choosing the data distribution algorithm, enabling symlink support, and altering file system protocols, among others, as policies. Nonetheless, such configurations must be set before the file system is launched and cannot be changed afterward, remaining static for the application's lifetime. In the next subsection, we propose an architecture that considers this limitation and attempts to offer a malleable platform for user space file systems based on I/O interception mechanisms.

3.3.2 Revising GekkoFS's architecture

We examine the architecture of a real burst buffer file system (GekkoFS), and extend it to support malleability reconfigurations, discussing the final architecture's advantages and limitations. The primary goal of this redesign is to improve the compatibility of user-space file systems while establishing a framework that can support various malleability techniques. While the proposed architecture is GekkoFS-specific, the design is generic enough to be adopted by other user space file systems.

While examining GekkoFS's current design, we discovered that GekkoFS I/O servers are already equipped to support malleable mechanisms to a certain extent. Due to GekkoFS's decoupled architecture, adding mechanisms such as increasing or removing server nodes during runtime is relatively straightforward. GekkoFS's original design already supports this use case without any issues as long as the file system is empty. Changing the number of server nodes, however, would work inefficiently: since a file's data and metadata server node are computed by generating hash keys, these keys would change, and most of the data in the file system would need to be redistributed. As discussed above in the case of Hercules, there are techniques to address this. Other methods [32] are possible and are currently in the implementation process.

On the other hand, the GekkoFS client design was unsuitable for supporting a broad set of malleability techniques. Since GekkoFS clients are restricted to an application process's lifespan, implementing relaxed cache consistency protocols and aligning them more closely with NFSv4 semantics [20] on a node level, for example, is impossible. Temporarily relaxing cache consistency guarantees, such as during the write burst of bulk-synchronous applications [3], could considerably improve performance, particularly for small I/O latency-sensitive requests. Overall, a GekkoFS client design that supports these use cases would complement a minimal *syscall_intercept* or `libc` interposition library by outsourcing more complex client code to another client process that operates on node-granularity. Therefore, as a part of this overhaul, a new file system component is added – the *GekkoFS proxy*.

Figure 3.6 illustrates the GekkoFS architecture with the newly introduced GekkoFS proxy. The proxy acts as a gateway between the client and daemons, forwarding all communication between the two components. As a result, the GekkoFS client complexity is considerably reduced. Since all file system communication now passes through the proxy, other advanced file system features can be implemented. For instance, with caching on the proxy, relaxed consistency models can be implemented that define how long data remains at the proxy before being distributed to the daemons. One of the supported cache consistencies could be similar to NFSv4 [20], providing *close-to-open* semantics where data is distributed once a file is closed. Therefore, multiple client processes on the same node can operate on a distributed file at local speeds. Other use cases could involve batching, which bundles many small I/O requests at the proxy before sending a corresponding remote request instead of serving each I/O request individually [38], or encryption. The applied strictness of these features could be steered through an API accepting malleable requests at runtime exposed via an RPC interface of the GekkoFS daemon. As part of the ADMIRE API, we are currently working on exposing these interfaces to the ADMIRE software stack so that it can be steered by policies in the malleability manager.

Nevertheless, using the GekkoFS proxy adds one additional communication step in the form of *inter-process communication* (IPC) for each file system operation sent to the daemon. This increases the latency for each operation, which may affect both data and metadata throughput.

The details for this topic and the corresponding scalability evaluation were published at the *2nd International Workshop on Malleability Techniques Applications in High-Performance Computing* (HPCMALL23) with the title *"From Static to Malleable: Improving Flexibility and Compatibility in Burst Buffer File Systems"* [51]. Overall, in this paper, we have discussed the critical issues of `LD_PRELOAD` as an interception

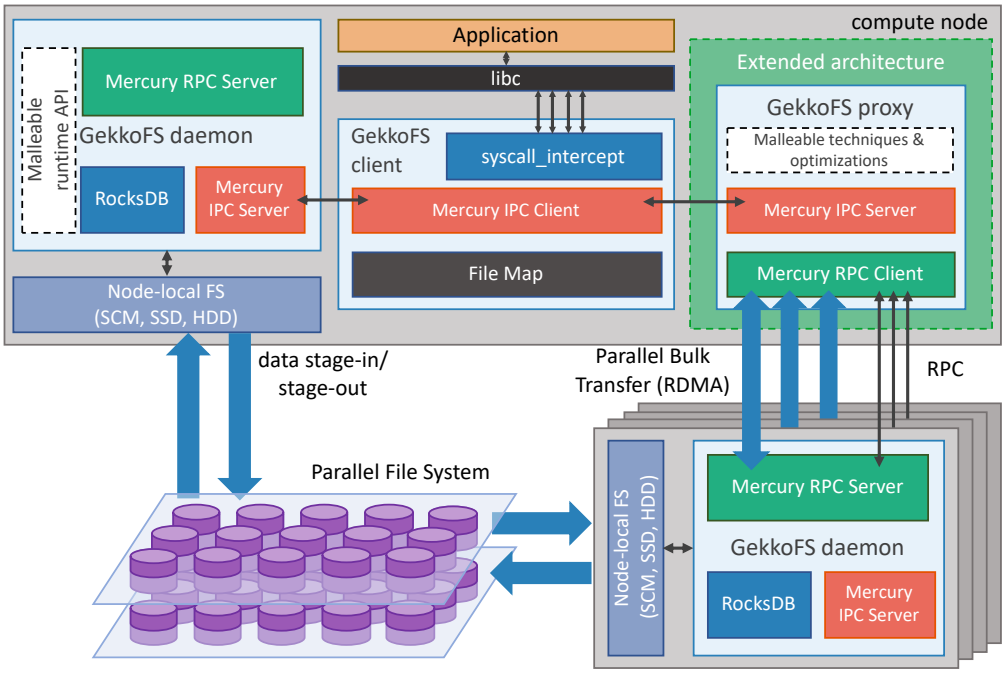


Figure 3.6: An extended GekkoFS architecture featuring the proxy.

mechanism and have proposed a new architecture for GekkoFS that overcomes these challenges, including presenting a path forward to broad malleability support in burst buffer file systems that can be deployed ad-hoc. We have evaluated the new architecture for latency- and throughput-sensitive operations and have concluded that the proxy only induces minor overheads, maintaining most of the file system's performance. Particularly for user space file systems that rely on an interception mechanism, as in the cases of GekkoFS and Hercules, this architecture offers new ways to support malleability and other optimisation techniques, which we will implement and evaluate in the remainder of the project.

3.3.3 Performance expectations of malleable file systems

We consider many different ways to realize I/O malleability in ad-hoc file systems. The most important one is file system extension and shrinking which GekkoFS partly supports (due to its ongoing development) and Hercules (see Section 3.2). While other techniques can be malleable, e.g., configurations that affect file system protocols, the performance implications are not always clear. For instance, due to the nature of ad-hoc file systems and that they usually provide almost linear scaling with the number of nodes [3], it is reasonable to expect at most twice the I/O performance if the number of nodes is doubled. Although the exact performance improvements depend on many variables, such as network bandwidth, latency, current usage, or the storage devices in use, it is, by and large, predictable. To this end, we are working on providing a performance tool that evaluates possible combinations to find the most suitable one for a given HPC environment.

On the other hand, malleable policies which directly depend on an application's I/O behavior are more challenging to evaluate. For instance, if an application only writes data without other processes reading this data, strong consistency semantics are no longer required and such an application could benefit from heavy write-back caching, significantly speeding up file system performance [38]. Understanding application I/O behavior and mapping the corresponding malleable policies is one of the key motivational points of the EuroHPC I/O tracing initiative. Further information on this initiative can be found in Deliverable 2.3.

Chapter 4

Scheduling algorithms

Scheduling jobs in HPC involves allocating computing resources and managing their execution. In the context of ADMIRE, we try to balance computation and I/O to avoid PFS congestion. In Section 2.3, we surveyed some of the related works in this area. This chapter first presents the novel scheduling algorithms for malleable jobs to address the imbalanced demand for computing and I/O resources. We present first an extension of EASY backfilling and then an I/O-intensity–aware scheduling algorithm. In continuation, we introduce a new simulator, called ElastiSim, that we developed to evaluate scheduling algorithms supporting malleable jobs.

4.1 Malleable EASY backfilling scheduler

Although techniques such as backfilling (see Section 2.1.1) and gang scheduling [16] attempt to shorten the time a job spends in the queue, it is still common for jobs to fall behind in the queue because they require only a few more processors than are currently available. Malleability, where the number of processors allocated to jobs can be shrunk or expanded at runtime [43], comes as a rescue for this problem.

We proposed an algorithm to schedule malleable parallel jobs using EASY backfilling (EBF) and taking the benefits of malleability if backfilling cannot find a suitable job to fill the resource gaps. In the malleable EASY backfilling (MEBF) algorithm, malleation decisions are applied to the jobs based on their priorities and lengths only when they are not harming the performance. The feasibility routine checks the malleability effect on the performance of the candidate's jobs. The experimental results show that our algorithm achieves significant performance benefit. Before describing MEBF, we consider the following fundamental assumptions:

- All jobs of the workload are parallel jobs.
- A malleable application can run on any number of nodes within a given interval.
- The cost of malleability (reconfiguration cost) has been considered.
- Only compute nodes are considered as resources.
- The HPC system is homogeneous, i.e., all nodes have the same computing power and connection to the network.

4.1.1 Scheduling algorithm

The workloads that can be run on contemporary HPC systems have become more complicated ,in regards to diversity and size, compared to traditional HPC applications [11]. Due to the exclusive allocation approach, the overall execution time of the workloads is significantly affected when large jobs are submitted and idle resources exist [54]. As mentioned before in Section 2.1.1, backfilling enables fair and efficient scheduling and has been used in most production systems. However, in most studies, EASY backfilling is used to schedule jobs that are allocated to a constant number of resources. Our contribution is to design a backfilling variant capable of scheduling malleable applications. Since backfilling decisions are highly dependent on the estimated execution time, and to avoid user under- or overestimation, our algorithm reads the job's runtime from recorded

executions on the dedicated system rather than relying solely on user input as a closer approximation. Algorithm (1) explains the proposed work.

MEBF first tries to serve each submitted job according to its priority. It gives the higher-priority jobs a chance to be executed before the lower-priority jobs. If an arriving job is malleable and not enough resources are available to execute it, backfilling is invoked to find another malleable job with a minimum number of nodes that can be allocated with the available resources. If backfilling does not find a suitable job, expansion of the longest-running job can be checked. If the state of the system indicates that there are no more free nodes at all, a list of candidate jobs based on their priority is created to make space for the waiting job. We used Equations (4.1) and (4.2) to check if a job can be shrunk.

$$granted\_nodes = job\_assigned\_nodes \times sharing\_factor \tag{4.1}$$

The granted nodes are the number of nodes that could be removed from the running malleable job(s) and taken as a fraction of the assigned nodes. This fraction is determined by *sharing_factor* which defines the shrinkage step (i.e., shrink amount)

$$shrink\_increase = \frac{granted\_nodes}{job\_assigned\_nodes} \times rem\_exe\_time + \alpha \tag{4.2}$$

where *rem_exe_time* is the estimated remaining time of a job until it is completed. We used the recorded execution times of the jobs on a dedicated system to calculate the estimated time. While *alpha* represents the cost of deallocation.

Shrinking is performed only if the ratio between the predicted execution time with shrinking and the static execution time does not exceed a certain threshold (γ), as defined in Equations (4.3) and (4.4). It is important to mention that nodes are only stolen from low-priority jobs.

$$new\_est\_exe = rem\_exe\_time + shrink\_increase \tag{4.3}$$

$$\frac{new\_est\_exe}{rem\_exe\_time} <= \gamma \tag{4.4}$$

The job expansion is carried out after checking whether the expansion is feasible. The predicted remaining execution time should be greater than a quarter of the recorded execution time of the job. Otherwise, the job is considered almost complete and the expansion will not provide the desired benefits. In addition, the allocation cost (α) of the new nodes is considered.

4.1.2 Evaluation

in this section, we evaluate our proposed scheduling algorithm using Elastisim 4.3. We first describe our experimental setup and then discuss our results.

4.1.2.1 System model

The crossbar represents a homogeneous cluster, where hosts are interconnected by a crossbar switch with as many ports as hosts so that each disjoint pair of hosts can communicate simultaneously at full speed. For simplicity, our experiments were conducted on a platform with a crossbar topology with 500 nodes. All nodes have the same computational power (100 GFLOPS/s) and are connected directly to the switch via private links with the same bandwidth (100 Gb/s). Table 4.1 summarizes the platform description.

4.1.2.2 Workload model

The malleable workload in our experiments is divided into two units: jobs and application models. While jobs define scheduling-related parameters such as the requested number of nodes and priority, application models represent the application executed on the simulated platform. Each application model contains multiple phases, while each phase includes multiple tasks. The set of simulated jobs and the corresponding application models

Algorithm 1: Malleable EASY Backfilling (MEBF)

```
   input : Jobs, nodes, time
 1 // workload lists updated each schedule
 2 workload_init(Jobs)
 3 // resources lists updated each schedule
 4 free_nodes = {n_0, ..., n_i : n.state == FREE}
 5 sorted_jobs ← sort(jobs, "priority")
 6 for each job, job ∈sorted_jobs do
 7     if job.state = PENDING then
 8         for i ∈ pref_nodes ... min_nodes; STEP : −1 do
 9             if i ≤ free_nodes then
10                 job.assign(i)
11                 break
12             end
13         end
14         if free_nodes = 0 & low_r ≠ 0 then
15             SHRINK(low_r)
16         else
17             if free_nodes ≠ 0 & rmjobs ≠ 0 then
18                 backfilled_job =BACKFILL(pending_jobs)
19                 if backfilled_job = "None" then
20                     EXPAND(high_r)
21                 else
22                     backfilled_job.assign(pref_nodes)
23                 end
24             else
25                 EXPAND(high_r)
26             end
27         end
28     end
29 end
```

Table 4.1: Platform specifications

Feature	Value
number of nodes	500
speed	100 GFLOPS
bandwidth	100 Gb/s
topology	crossbar

form the workload and define the simulation scenario. In our work, we tested three types of application models: compute-intensive, read-intensive, and write-intensive. We combined these application models with workloads of different sizes (100 jobs, 200 jobs, and 300 jobs). In each experiment, the workload is composed of either 100% compute-intensive, 100% read-intensive, or 100% write-intensive jobs. Finally, a mixed workload combines 50% compute-intensive, 25% read-intensive, 25% write-intensive jobs.

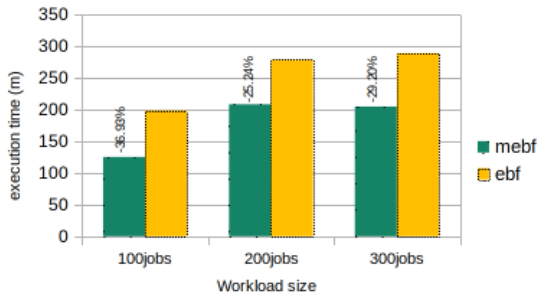
The phases within each modeled application consist mainly of reading data from the parallel file system

by a root node and scattering data by the root node to all other nodes. The main phase includes two tasks: an iterative computation task that collects data from all nodes and a parallel file system write task.

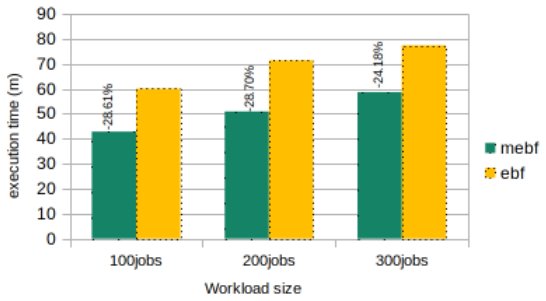
MEBF has been evaluated using the simulator ElastiSim (see Section. 4.3). A set of concurrent, malleable parallel jobs with different resource requirements are executed on a crossbar platform represented by a pool of 500 identical resources. Jobs are submitted offline, that is, all jobs are specified before the scheduling algorithm starts. We used the following metrics for the evaluation:

- Average execution time of jobs: the average time elapsed between start and completion.
- Average turnaround time: the average time elapsed between submission and completion.
- Average node allocations: the average of how many times the node has been allocated by a job.

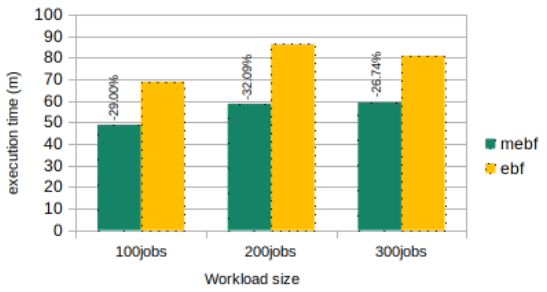
Our algorithm tries to apply malleability with guardedly to avoid the negative impact on the execution of the running jobs. We simulated the workload using a shrink factor of 0.20, while the feasibility cut-off value (γ) was 1.24, which represents feedback based on the ratio between the estimated execution time with malleability and the estimated execution time without malleability. Figure 4.1 shows the average execution time of the workloads. We observe a reduction of the average execution time of up to 30.4%, 27.1%,29.27%, and 28.9% for compute-intensive, write-intensive, read-intensive, and mixed workloads, respectively.



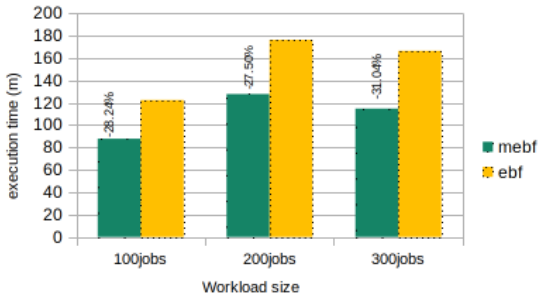
(a) Compute-intensive



(b) Write-intensive



(c) Read-intensive



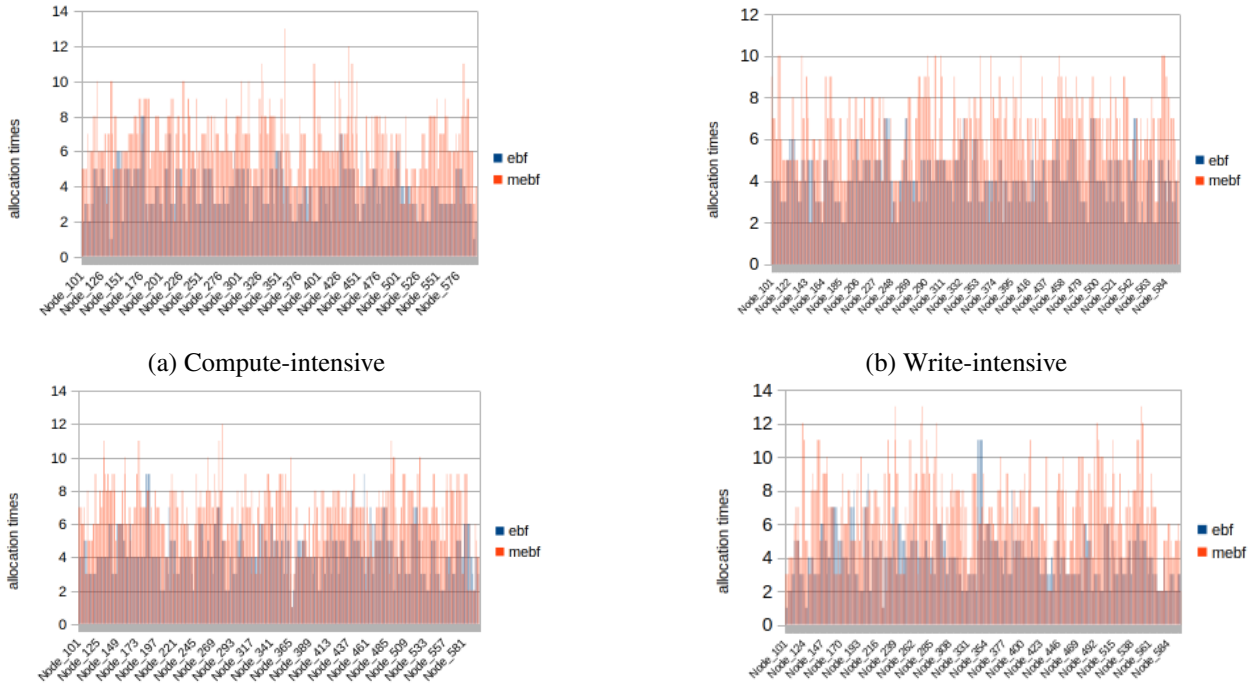
(d) Mixed

Figure 4.1: Execution time of different applications with varying workload sizes (100, 200, and 300 jobs)

Regarding node utilization, Figure 4.2 and Figure 4.3 show the superiority of the MEBF approach in utilizing platform nodes. Figure 4.3 shows that scheduling the four applications with MEBF enables each node to be allocated by more than 6.4 jobs on average, while the platform nodes are used only by 4.1 jobs on average when using standard EBF.

The CPU utilization over time is shown in Figure 4.4. It is evident that MEBF allows the same workload (100 jobs) of each application to terminate early. This behavior occurs in response to expansion events. Our selection criterion chooses the longest job from a pool of higher-priority jobs to perform the expansion. Once the job expands, the load of the job is distributed to more nodes so that the long job finishes earlier.

Figure 4.5 shows that the same workload of 100 jobs can be completed in less than 400 minutes with MEBF, while it takes more than 700 minutes with EBF.



(a) Compute-intensive

(b) Write-intensive

(c) Read-intensive

(d) Mixed

Figure 4.2: System utilization. Comparison of different applications with a workload size of 300 jobs.

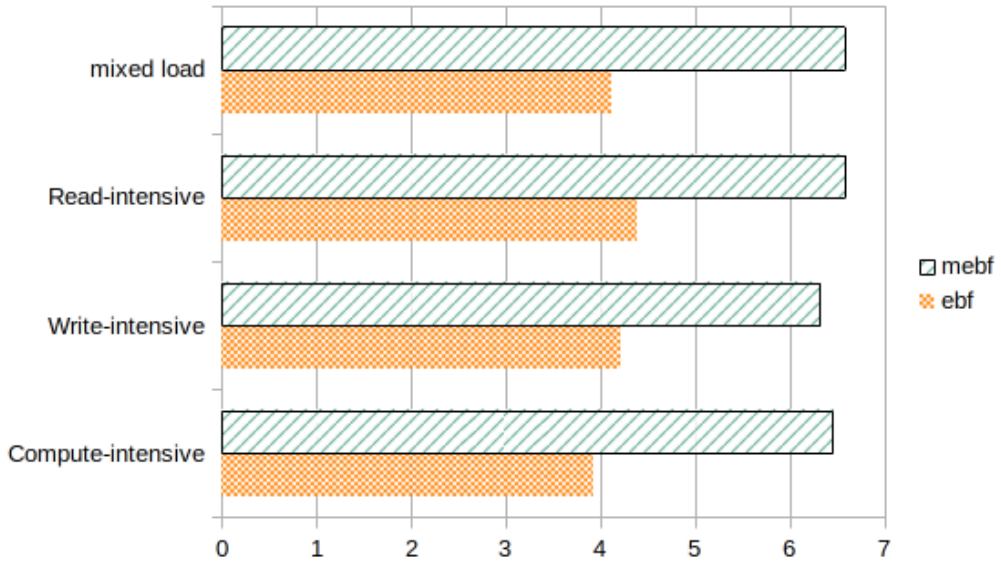
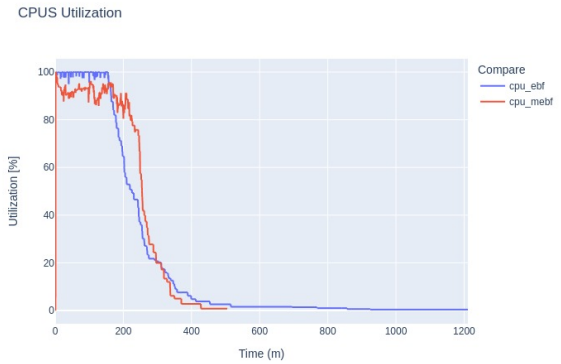
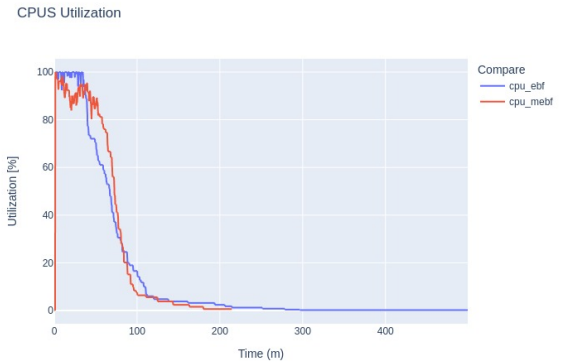


Figure 4.3: The average of node utilization with MEBF and EBF of a workload composed of 300 jobs from different applications

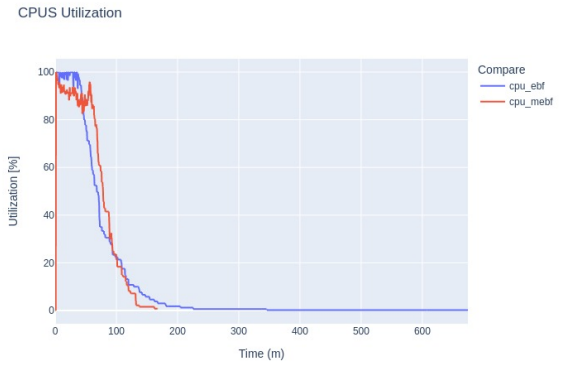
From this evaluation, we can conclude that malleable EASY backfill is able to schedule different malleable workloads (i.e. I/O , compute-intensive and mixed) allowing for the benefits of malleability such as better use of resources and reducing the execution time and response time. The reduction of the average execution time of up to 30.4% in the compute-intensive workload.



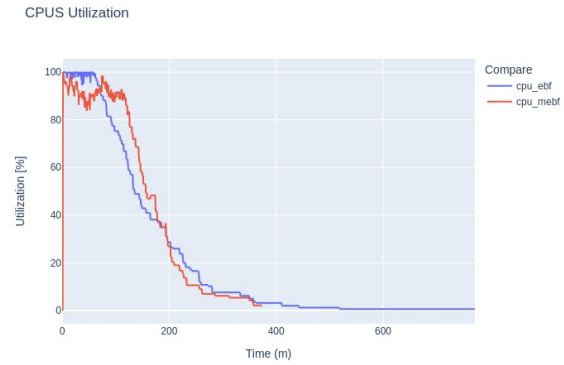
(a) Compute-intensive



(b) Write-intensive

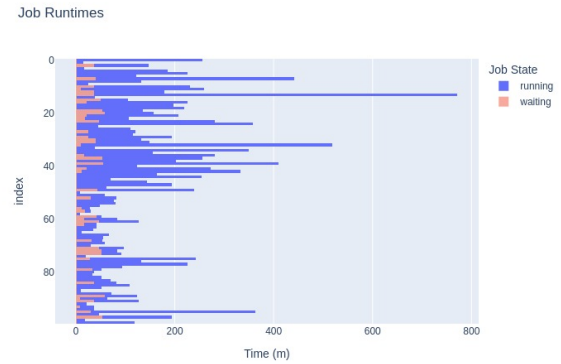


(c) Read-intensive

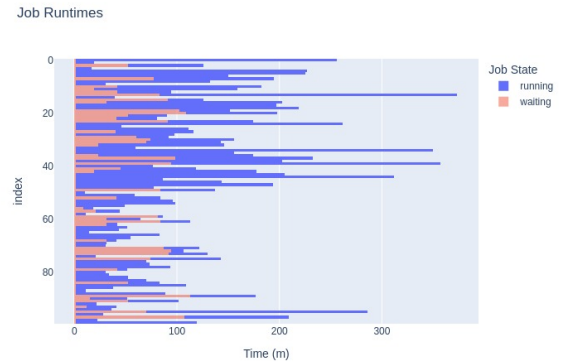


(d) Mixed

Figure 4.4: CPU utilization of different applications with a workload size of 100 jobs



(a) Mixed load composed of 100 jobs executed with standard EBF



(b) Mixed load composed of 100 jobs executed with MEBF

Figure 4.5: Gant chart of 100 jobs with mixed load.

4.2 I/O-intensity–aware scheduler

In HPC, the I/O intensity of a job refers to the volume and frequency of I/O operations compared to the amount of computation performed by the job. The I/O intensity of a job can be affected by several factors, including the number of I/O operations, the size of the data transfer (read and write), and the storage specification, such as speed and capacity. Traditional HPC systems are typically optimized for compute-intensive jobs rather than I/O-intensive jobs. Therefore the jobs with high I/O intensity can be particularly challenging to run efficiently on HPC systems.

Data-intensive applications spend a significant portion of their execution time on I/O operations such as reading input, writing output, or checkpointing intermediate results. This means that any speedups achieved by accelerating computation are limited by the fraction of time spent on I/O. As Amdahl's law states in its most general version, "the potential enhancement of overall system performance, gained by optimizing a single part of a system, is restricted by the fraction of time that the improved part is used." [2] Conversely, this implies that compared to compute-intensive applications, the performance of data-intensive applications depends to a higher degree on the available I/O bandwidth.

The following example illustrates this relationship (Figure 4.6). Consider four jobs, two identical data-intensive ones, and two identical compute-intensive ones, each one occupying the same number of nodes. We co-schedule them as two pairs in a sequence, the first two in parallel, followed by the remaining two in parallel. Jobs running in parallel share the total bandwidth of the PFS in equal proportions. If we co-schedule jobs with the same intensity characteristics (Figure 4.6a) instead of mixing data-intensive with compute-intensive ones (Figure 4.6b), the relative slowdown of the data-intensive jobs will be much higher. This loss will not be compensated by the gain of the compute-intensive ones that they derive from low file-system contention. This results in a much shorter makespan, observable in the mixed scenario (Figure 4.6b). Our scheduling algorithm exploits this optimization by trying to co-schedule data-intensive with compute-intensive jobs.

This logic even applies if the PFS bandwidth is the only bottleneck. On real systems, further bottlenecks beyond the overall bandwidth may exist, such as metadata servers and local network links and switches. If high I/O contention hits one of those bottlenecks, the PFS may even be underutilized, resulting in an additional penalty. Properly balancing I/O and computation may, therefore, also reduce the risk of such extra performance degradation, possibly increasing the speedup beyond what was demonstrated in the example.

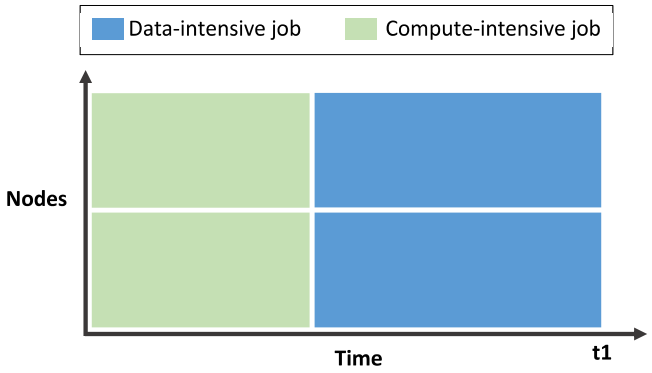
Below, we first define and model our problem formally. Then we explain our proposed scheduler which makes the decisions based on the available knowledge of the I/O intensity of submitted jobs to decrease the makespan. Finally, we evaluate the results and present how our algorithm can improve makespan in various scenarios.

4.2.1 Problem definition

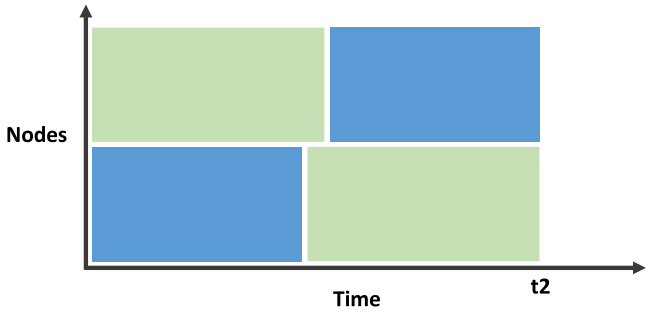
We target the problem of scheduling a combination of rigid and malleable jobs in an HPC infrastructure including a shared PFS. We assume that applications have exclusive access to compute resources, but the bandwidth to the storage system is shared between all compute nodes. The assumed applications comprise recurring—but not necessarily periodic—I/O phases, such as applications that collectively checkpoint their current state or transactional workloads. Furthermore, we assume that the applications can fully utilize potential bandwidth to the PFS. Future versions of our proposed algorithm will investigate a wider variety of applications.

During I/O-intensive phases, when a large number of nodes access the PFS simultaneously, applications compete for the limited I/O resources, potentially leading to congestion. How often this phenomenon occurs depends on how balanced the computational and I/O activities of applications on the system are distributed. We define the state of imbalance as the execution of a set of jobs that introduces a significant underutilization or permanent utilization of I/O resources at capacity. With the latter scenario, the PFS is forced to share its bandwidth with a higher number of nodes than optimal, consequently leading to a likely underutilization later on.

However, optimal utilization depends on the possible sets of jobs the batch system can schedule. A workload comprising only I/O-intensive applications will inevitably lead to PFS congestion, no matter the schedule of jobs. A naive scheduling approach not considering the I/O intensity of applications poses the eminent risk of



(a) Parallel execution of the jobs with similar intensities



(b) Mixed execution of the jobs

Figure 4.6: Two variants of schedule solutions. In the second scenario, we observe an improvement in the makespan by $t1 - t2$.

an imbalanced schedule, representing the problem our algorithm solves. As a subset of our workload comprises malleable jobs, we investigate optimizing the PFS utilization not only by finding optimal schedules prior to job execution but also by continuous observation and reconfiguration of malleable jobs.

Our proposed scheduling algorithm aims to minimize congestion by keeping the overall I/O intensity of running applications as close as possible to the average intensity defined by all jobs in the system (running and queued jobs). The main objectives of the algorithm are: (i) anticipating and minimizing PFS congestion, (ii) exploiting malleability to control the load on the PFS, and (iii) applying a fair scheduling approach, preventing job starvation.

4.2.2 Definition of I/O intensity

We hypothesize that, on average, the overall I/O intensity of running jobs can not perform better than the average intensity considering the entire workload, including queued jobs. Scheduling decisions leading to lower-than-average I/O intensities have the potential risk of higher I/O intensities during later periods, congesting the PFS. To measure the current and target the average I/O intensity, we define I/O intensities for jobs, systems, and workloads as follows.

I/O intensity of a job. We define the I/O intensity of a job as a relation among the jobs in the system rather than a representation of a system metric. This approach allows site administrators to modify our definition to reflect their workload based on the system details. Following our assumption that the application's I/O is recurring, bursty, and fully utilizes the network bandwidth, we assign a single number to the job representing the I/O intensity through the relative time spent doing I/O. Tools such as Score-P [31] can facilitate gathering I/O and total times, allowing Extra-P [5] to extrapolate and predict I/O intensity metrics for multiple job configurations. Although this metric introduces a dependency on the system, the application profiles we investigate will keep their relation among each other, even when the intensity metric will vary in absolute numbers. For each job j

and each configuration (i.e., number of assigned nodes) $nodes_j$, we define:

$$intensity_j(nodes_j) = \frac{io\_walltime_j}{total\_walltime_j}$$

such that $io\_walltime_j$ and $walltime_j$ denote respectively the total time spent for I/O and the total walltime of the job j.

I/O intensity of the system. The system I/O intensity represents the current I/O load on the system introduced by all running jobs. It is the only metric the scheduling algorithm directly influences by its decision-making. As not all jobs contribute equally to the system load because the number of assigned nodes differs for each job, we propose the following definition, considering a set RJ representing all running jobs and $nodes_j$ representing the actual configuration of a job (i.e., assigned number of nodes):

$$intensity(\mathbb{S}) = \frac{\sum_{j \in RJ}(intensity_j(nodes_j) \cdot nodes_j)}{total\ number\ of\ allocated\ nodes}$$

I/O intensity of the workload. The workload I/O intensity represents the I/O load of the entire workload, including queued jobs QJ. We calculate the workload I/O intensity analogously to the system I/O intensity but on a theoretical machine that could run all jobs simultaneously, applying the following formula:

$$intensity(\mathbb{W}) = \frac{\sum_{j \in RJ \cup QJ}(intensity_j(nodes_j) \cdot nodes_j)}{total\ number\ of\ cluster\ nodes}$$

4.2.3 Model

The reliability of the results and evaluation of our proposed scheduling algorithm depends on the underlying models on which we conduct the experiments. In this section, we introduce the system and workload model constituting the foundations of our experiments.

4.2.3.1 System model

Our scheduling algorithm targets homogeneous systems comprising a set of compute nodes and a central storage system (i.e., the PFS). All compute nodes provide a network connection/link with a maximum transfer rate of bw_{link}. We model the PFS as a shared storage tier accessible by all compute nodes with a maximum I/O transfer rate of bw_{pfs}. Conclusively, a job's maximum transfer rate to the PFS depends on the number of assigned nodes and is limited by $\min(nodes_j \cdot bw_{link},\ bw_{pfs})$. During I/O-intensive phases, when the aggregated transfer rate of all utilized network links is limited by bw_{pfs}, we apply a fair-sharing policy and equally distribute the transfer rate bw_{pfs} among all participating nodes. Figure 4.7 illustrates our system model.

4.2.3.2 Workload model

The workload is composed a set of jobs J and each job $j \in J$ is specified by the following attributes:

- Type: $type_j \in \{\text{RIGID}, \text{MALLEABLE}\}$
- Submission time (in seconds): $submit\_time_j \in \mathbb{R}_0^+$
- State: $state_j \in \{\text{PENDING}, \text{RUNNING}, \text{COMPLETED}\}$
- Minimum number of assignable nodes: $nodes\_min_j \in \mathbb{N}^+$
- Maximum number of assignable nodes: $nodes\_max_j \in \mathbb{N}^+$
- Possible configurations[1]:
  $Conf_j \subseteq \{n \in \mathbb{N}^+ : nodes\_min_j \leq n \leq nodes\_max_j\}$

[1] A rigid job j' has only one valid configuration as it implies $nodes\_min_{j'} = nodes\_max_{j'}$

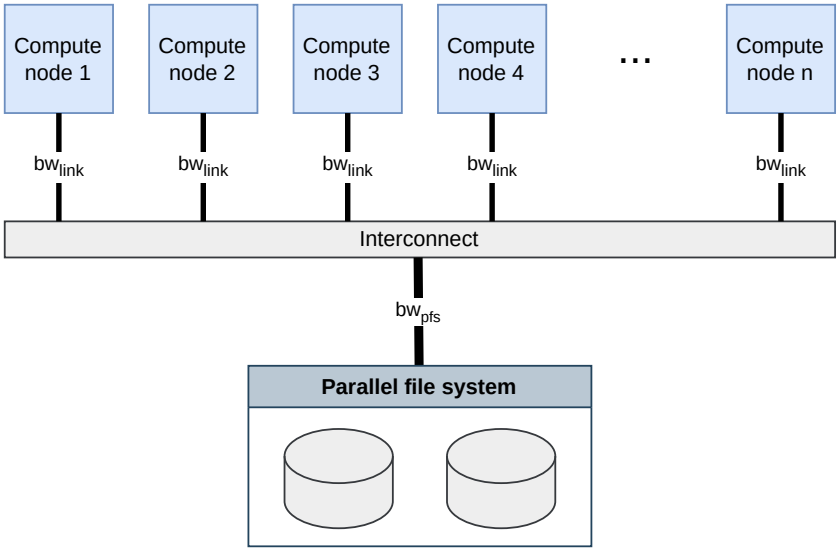


Figure 4.7: The system model comprising compute nodes and a shared parallel file system.

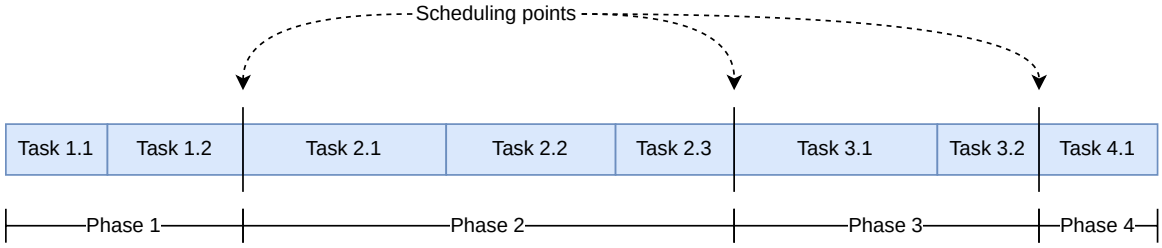


Figure 4.8: The application model comprising phases, tasks, and scheduling points. The number of phases and tasks might vary depending on the simulated application.

- Preferred configuration: $nodes_j^{pref} \in Conf_j$
- Applied configuration: $nodes_j \in Conf_j$
- I/O intensity for configuration $nodes_j$: $intensity_j(nodes_j) \in \mathbb{R}_0^+$

We further define an application model as illustrated in Figure 4.8, describing the system load a job can introduce. Each application model contains a set of phases representing the various stages in the execution of an application, and each phase contains a set of tasks describing the low-level system activity—either computational or I/O. After each phase, we introduce scheduling points for malleable jobs to allow reconfigurations during runtime.

4.2.4 Scheduling algorithm

Based on the definition of I/O intensity in Section 4.2.2, we propose the I/O-intensity—aware scheduler, targeting the minimization of congestion on HPC systems equipped with a shared PFS. Our scheduler aims to balance the I/O intensity of the executing workload. Since this balance depends on running and queued jobs, the scheduler balances the I/O load on the system by minimizing $|intensity(\mathbb{W}) - intensity(\mathbb{S})|$ when making any decision.

The system invokes the proposed scheduler at each job submission, job completion, and whenever an application reaches a scheduling point. Based on invocation triggers and subsequent decision-making, the scheduler updates the system and workload I/O intensities to reflect the current state and build the basis for upcoming scheduling decisions. Table 4.2 lists all triggers and which metrics they affect.

As making the schedule decisions based purely on I/O intensity metrics may eventually lead to starvation (some jobs may always be postponed in favor of other jobs), we propose a weighted priority for reordering the

Trigger	Affected I/O intensity metric
job submission	$intensity(\mathbb{W})$
job admission	$intensity(\mathbb{S})$
job completion	$intensity(\mathbb{S})$ and $intensity(\mathbb{W})$
job reconfiguration	$intensity(\mathbb{S})$ and $intensity(\mathbb{W})$

Table 4.2: Scheduling algorithm triggers and affected I/O intensity metrics.

jobs based on the I/O intensity and the order of jobs arrival. This weighted priority depends on the metric α, representing the site administrator's choice of how aggressively the scheduling algorithm can optimize for I/O intensity balance.

Let $\alpha \in [0, 1]$ be the job reordering intensity with $\alpha = 0$ representing a first-come-first-serve (FCFS) policy (i.e., absolute fairness) and $\alpha = 1$ lifting all restrictions that prevent starvation on the algorithm, maximally optimizing for I/O balance. As we consider possible reconfigurations of malleable jobs, we derive our proposed priority for queued and running jobs in the set of candidates $C = QJ \cup RJ$. Each candidate $c \in C$ holds an integer value pos_c, representing its relative position in the queue in the order of submission. Starting with the first pending job, the algorithm assigns integer values $[0..|C| - 1]$ and calculates the fairness priority value λ, normalized between 0 and 1, as in Equation 4.5.

$$
\begin{aligned}
min\_pos &= \min_{c \in C}(pos_c) \\
max\_pos &= \max_{c \in C}(pos_c) \\
\lambda_c &= \frac{pos_c - min\_pos}{max\_pos - min\_pos}
\end{aligned}
\tag{4.5}
$$

For each configuration of the candidate c, we calculate δ, denoting the priority value representing the I/O intensity. The algorithm first calculates the potential new system intensity according to Equation 4.6, then builds the absolute difference to the workload intensity as defined in Equation 4.7.

$$
intensity_{new} = \frac{\sum_{j \in RJ}(intensity_j(nodes_j) \cdot nodes_j) + intensity_c(nodes_c) \cdot nodes_c}{total\ number\ of\ allocated\ nodes + nodes_c} \tag{4.6}
$$

$$
\delta_c^{nodes_c} = |intensity(\mathbb{W}) - intensity_{new}| \tag{4.7}
$$

Furthermore, we introduce a tuple $(\lambda_c, \delta_c^{nodes_c}, c, nodes_c)$ for each configuration of the candidate c, constituting the set of tuples P representing all possible priority values and their respective configurations. After determining the minimum and maximum value of all δ values, the algorithm uses the normalized priority value δ (Equation 4.8) and calculates the weighted priority based on the previously chosen value α (Equation 4.9).

$$
\begin{aligned}
min\_delta &= \min\{(\lambda_c, \delta_c^{nodes_c}, c, nodes_c) \in P : (\lambda_c, \delta_c^{nodes_c}, c, nodes_c) \mapsto \delta_c^{nodes_c}\} \\
max\_delta &= \max\{(\lambda_c, \delta_c^{nodes_c}, c, nodes_c) \in P : (\lambda_c, \delta_c^{nodes_c}, c, nodes_c) \mapsto \delta_c^{nodes_c}\}
\end{aligned}
$$

$$
norm(\delta') := \frac{\delta' - min\_delta}{max\_delta - min\_delta} \tag{4.8}
$$

$$
wp(\alpha', \lambda', \delta') := (1 - \alpha) * \lambda + \alpha \cdot norm(\delta') \tag{4.9}
$$

We collect all weighted priorities in the set of tuples WP (Equation 4.10) and, in the final step, the scheduler determines the best possible candidate and its respective configuration by applying Equation 4.11 to determine the tuple $(wp, c, nodes_c) \in WP$, such that wp is minimal, representing the tuple containing the smallest weighted priority, the best candidate c^{best}, and the best configuration $nodes_c^{best}$.

Algorithm 2: I/O-intensity–aware scheduler

Input: List of jobs: J; list of nodes: N; invocation trigger: $trigger$; triggering job: j

```
1  if trigger ≠ SCHEDULING_POINT then
2      if trigger = JOB_SUBMISSION then
3          add_job_to_workload_io_intensity(j);
4      else if trigger = JOB_COMPLETION then
5          remove_job_from_system_io_intensity(j);
6          remove_job_from_workload_io_intensity(j);
7      end
8      find_and_schedule_job(J, N);
9  else
10     nodes_j^new ← get_best_configuration(j);
11     if nodes_j^new ≠ {} then
12         nodes_j^old ← nodes_j;
13         nodes_j ← nodes_j^new;
14         update_system_io_intensity(j, nodes_j^old);
15         if nodes_j^new < nodes_j^old then
16             find_and_schedule_job(J, N);
17         end
18     end
19 end
```

$$WP = \{(\lambda_c, \delta_c^{nodes_c}, c, nodes_c) \in P : (\lambda_c, \delta_c^{nodes_c}, c, nodes_c) \mapsto (p(\alpha, \lambda_c, \delta_c^{nodes_c}), c, nodes_c)\} \tag{4.10}$$

$$(c^{best}, nodes_c^{best}) = \min\{(wp, c, nodes_c) \in WP : (wp, c, nodes_c) \mapsto wp\} \mapsto (c, nodes_c), \tag{4.11}$$

If malleable applications reach a scheduling point, the scheduler optimizes solely for I/O intensity. For a malleable job j' reaching a scheduling point in its current configuration $nodes_{j'}^{cur}$, the scheduler calculates the new system intensity for all configurations $nodes_{j'}^{new} \in Conf_{j'}$, where $nodes_{j'}^{new} \neq nodes_{j'}^{cur}$ according to Equation 4.12. The scheduler reconfigures the malleable job by choosing the configuration such that the absolute difference to the workload intensity is minimal (Equation 4.13).

$$intensity_{total} = \sum_{j \in RJ} (intensity_j(nodes_j) \cdot nodes_j) - intensity_{j'}(nodes_{j'}^{cur}) \cdot nodes_{j'}^{cur}$$

$$intensity_{new} = \frac{intensity_{total} + intensity_{j'}(nodes_{j'}^{new}) \cdot nodes_{j'}^{new}}{total\ number\ of\ allocated\ nodes - nodes_{j'}^{cur} + nodes_{j'}^{new}} \tag{4.12}$$

$$nodes_{j'} = \min\{nodes_{j'} : nodes_{j'} \mapsto |intensity(\mathbb{W}) - intensity_{new}|\} \tag{4.13}$$

Algorithm 2 describes the main body of our scheduling approach. At each invocation, the batch system provides the scheduler with (1) the list of jobs J, (2) the list of nodes N with $state_n \in \{\text{FREE}, \text{ALLOCATED}\}$ describing the current state of the node n, (3) the invocation $trigger \in \{\text{JOB\_SUBMISSION}, \text{JOB\_COMPLETION}, \text{SCHEDULING\_POINT}\}$, and (4) the job j triggering the invocation.

Lines 2–8 describe the scheduler's actions at job submissions and completions. The scheduler recalculates the I/O intensities of the workload and the system, depending on the invocation trigger. While job submissions only affect the workload I/O intensity, job completions affect both I/O intensity metrics. As malleable jobs can have multiple configurations and, therefore, multiple I/O intensity metrics, the scheduler considers the I/O intensity of the preferred configuration of a malleable job in `add_job_to_workload_io_intensity()`. In line 8, `find_and_schedule_job()` determines the best possible candidate to schedule based on our introduced weighted priority metric, which we further describe in Algorithm 3. In lines 10–18, the scheduler

Algorithm 3: $find\_and\_schedule\_job(J, N)$

```
   Input: List of jobs: J; list of nodes: N
 1 pending_jobs ← {j ∈ J : state_j = PENDING};
 2 free_nodes ← {n ∈ N : state_n = FREE};
 3 if |pending_jobs| > 0 ∧ |free_nodes| > 0 then
 4     potential_jobs ← pending_jobs;
 5     retrieve_min_nodes(potential_jobs);
 6     while |potential_jobs| > 0 ∧ |free_nodes| > min_nodes do
 7         c, nodes_c^new ← get_best_candidate(potential_jobs, |free_nodes|);
 8         if c ≠ {} then
 9             nodes_c ← nodes_c^new;
10             add_job_to_system_io_intensity(j);
11             potential_jobs ← potential_jobs \ {c};
12             if |potential_jobs| > 0 then
13                 min_nodes ← retrieve_min_nodes(potential_jobs);
14             end
15         end
16     end
17 end
```

handles applications that reach a scheduling point. The scheduler first calls `get_best_configuration()` to retrieve the best possible configuration according to Equation 4.13, then checks if a better configuration could be determined (line 11). In that case, we save the current configuration, apply the new configuration, and update the system I/O intensity, based on the job—including its new configuration—and its previous configuration (lines 12–14). The function `get_best_configuration()` always issues an expand if it leads to an optimized I/O intensity but only issues a shrink if better options are available in the queue. Therefore, the scheduler calls `find_and_schedule_job()` again if the new configuration is smaller than the previous one to schedule the better option (lines 15–17).

In the function `find_and_schedule_job()`, described by Algorithm 3, the scheduler first identifies pending jobs and free nodes subject to scheduling (lines 1–2). Suppose both lists contain at least one element each. In that case, we consider all pending jobs as potential jobs to schedule and call `retrieve_min_nodes()` to determine the minimum number of nodes required for any potential job (lines 3–4). As long as there are potential jobs left and the number of free nodes suffice, we determine the best possible candidate job based on the minimal weighted priority described in Equation 4.11 (line 7). If there are enough free nodes to schedule for the candidate (i.e., `get_best_candidate()` returns a non-empty solution), we apply the new configuration, update the system I/O intensity, and remove the candidate from the list of potential jobs (lines 9–11). If any potential job is left, the scheduler updates the minimum number of nodes required to consider it for the next potential iteration (line 12–14).

4.2.5 Evaluation

In this section, we evaluate our proposed scheduling algorithm. We first describe our experimental setup and discuss our results afterward.

4.2.5.1 Experimental setup

We evaluated our proposed scheduling algorithm using ElastiSim (see Section 4.3). Following our system model described in Section 4.2.3.1, we simulated 500 compute nodes, each with a computing power of 100 GFLOP/s and a network link capacity (bw_{link}) of 100 Gbit/s. We attached a parallel file system shared by all compute nodes with a peak I/O performance (bw_{pfs}) of 48 GB/s, equally distributed among compute nodes during congestion.

Workload	Generation parameters
Number of jobs	4000
Number of I/O peaks	4
Jobs per I/O peak	200
I/O peak positions (job ID)	[447, 1283, 2414, 3355]
Computational load	$Beta(1, 1) \cdot 200$ TFLOP
Average I/O load	$Beta(2, 8) \cdot 256$ GiB
Peak I/O load	$Beta(1, 0.1) \cdot 256$ GiB

Table 4.3: Generation parameters of the synthetic workload.

Our workload comprises 4000 jobs, including 80 % rigid and 20 % malleable jobs. Each job repetitively runs a sequence of two tasks, starting with computational load, followed by writing a file to the PFS, simulating a checkpointing task involving all nodes. The number of repetitions ranges between 10 and 25 with a uniform distribution. Malleable jobs introduce a scheduling point after each checkpoint. While rigid jobs request a fixed number of compute nodes between 2 and 20, based on their combined computational and I/O load, the scheduler can schedule an arbitrary number of nodes for malleable jobs in the same range.

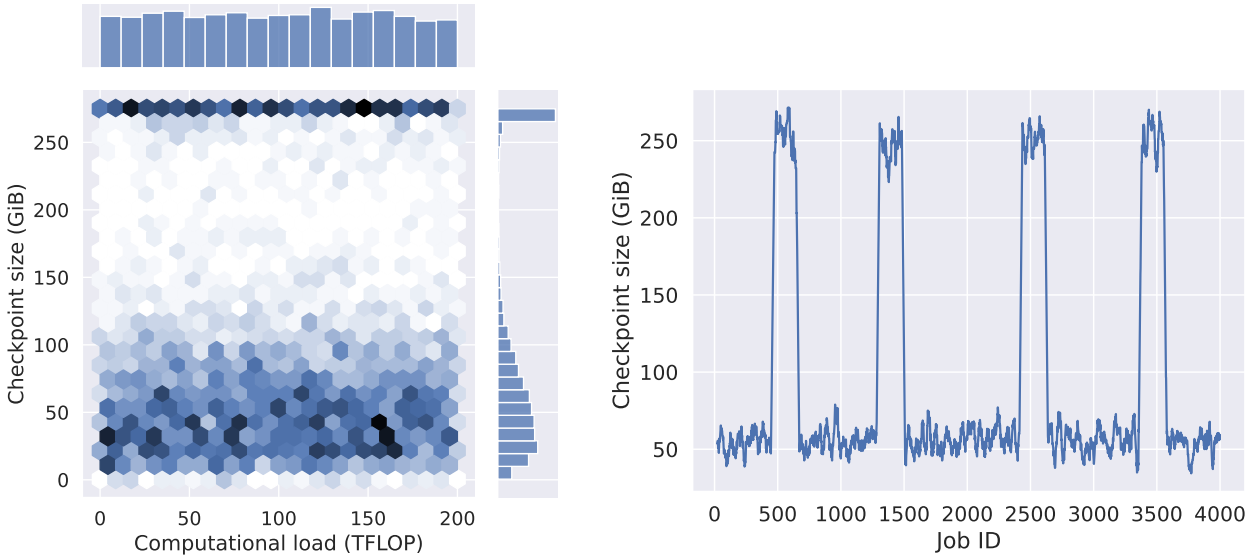
We generated a synthetic workload introducing a peak computational load of 200 TFLOP and a peak checkpoint size of 256 GiB. We introduced four I/O peaks in our workload, each making up 5 % of the workload (i.e., 200 jobs). Figure 4.9 shows the density heatmap illustrating the distribution of the computational loads and checkpoint sizes and the synthetically introduced I/O peaks, and Table 4.3 summarizes the generation parameters of the synthetic workload. Based on the system specification, we estimated the I/O and total walltime per job, calculated the I/O intensity following our definition in Equation 4, and attached it to each job as an attribute, representing the I/O intensity of the job.

To have a baseline for evaluation, we implemented a malleable FCFS algorithm (called FCFSm) that greedily expands jobs up to the maximum number of nodes when (1) a job reaches a scheduling point and (2) free nodes are available. This baseline algorithm differs from our novel scheduler with the reordering intensity of 0.0 to the extent that it lacks any indicator to issue a beneficial shrink operation and, therefore, only considers expand operations. As 20 % of the jobs were malleable, the expansion of jobs utilized free nodes and, thus, replaced the necessity for backfilling. We compared our proposed scheduling algorithm for all job reordering intensities $\alpha \in \{0.2, 0.3, 0.4, 0.5, 0.6\}$ with the baseline FCFSm algorithm.

4.2.5.2 Simulation results

The I/O-intensity–aware scheduler keeps constantly track of both I/O-intensity metrics. Figure 4.10 shows the observed intensity metrics for the FCFSm baseline and the five chosen values of the reordering intensity α. For both FCFSm and the I/O-intensity–aware scheduler, we observe that the workload I/O intensity adjusts over time with jobs arriving and leaving the system. With no optimization, the system I/O intensity reflects the previously introduced I/O peaks in the synthetic workload. Figures 4.10b–4.10f demonstrate that our proposed reordering intensity gradually improves the workload intensity approximation with increasing values for α—and that $\alpha = 0.6$ is sufficient for an approximation with negligible deviation, considering the I/O characteristics of our synthetic workload.

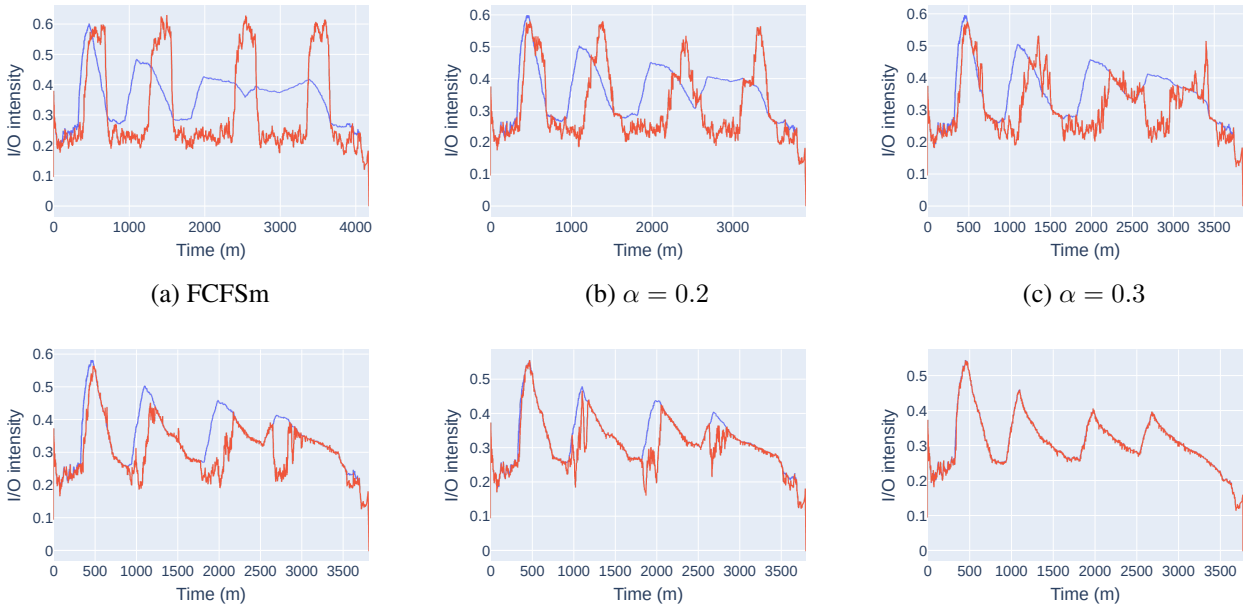
To evaluate our estimation of the I/O intensity metric and how accurately it reflects system parameters, we measured the CPU and PFS utilization. Figure 4.11 presents both utilization metrics and—in comparison with the previous Figure 4.10—indicates that our system I/O intensity metric correlates with the actual PFS utilization observed on our system model. During I/O peaks, the PFS utilization operates close to 100 %, indicating congestion. Compared to FCFSm, our proposed scheduler can effectively reduce congestion. Starting with a reordering intensity $\alpha = 0.2$, we can observe reduced PFS congestion with further improvements for



(a) Density heatmap representing the distribution of compute- and I/O intensive jobs.

(b) Varying checkpoint sizes of arriving job showing the introduced bursts of I/O-intensive jobs.

Figure 4.9: Compute- and I/O-intensities of the simulated scenario.



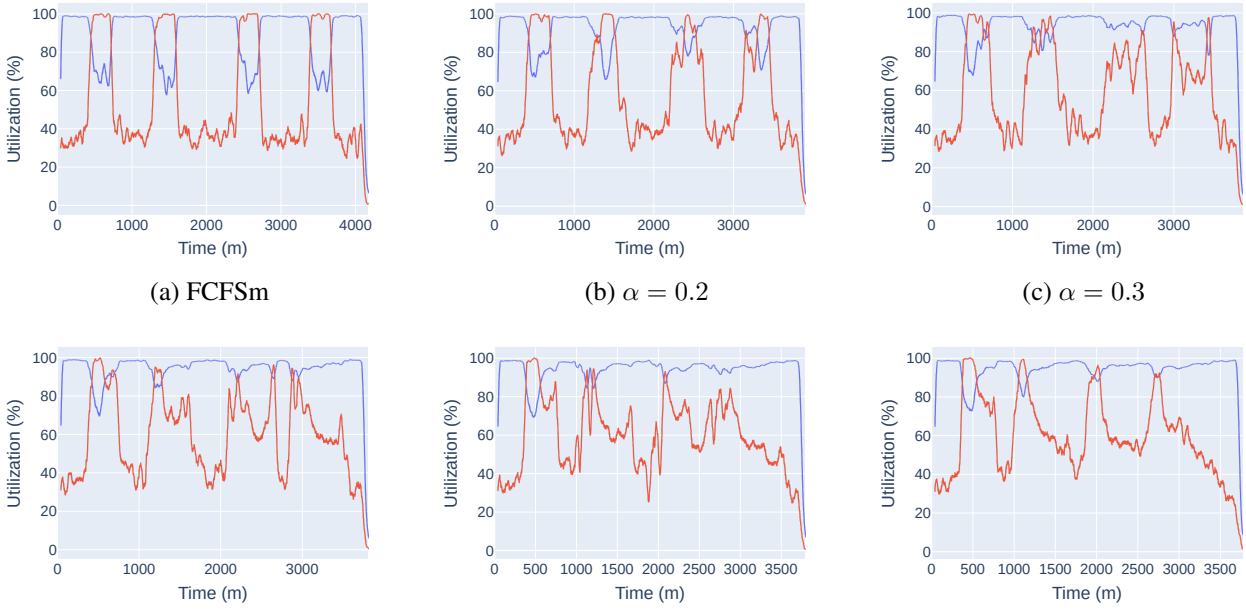
(a) FCFSm (b) $\alpha = 0.2$ (c) $\alpha = 0.3$

(d) $\alpha = 0.4$ (e) $\alpha = 0.5$ (f) $\alpha = 0.6$

Figure 4.10: Comparison of workload (blue) and system I/O intensities (red) for increasing job reordering intensities α. For $\alpha = 0.6$, the system I/O intensity approximates the workload I/O intensity with a negligible difference.

increasing values for α. Reduced PFS congestion is also observable when we evaluate CPU utilization. As our scheduler reduces the load on the PFS, the CPU utilization increases, allowing the system to improve efficiency by decreasing I/O times.

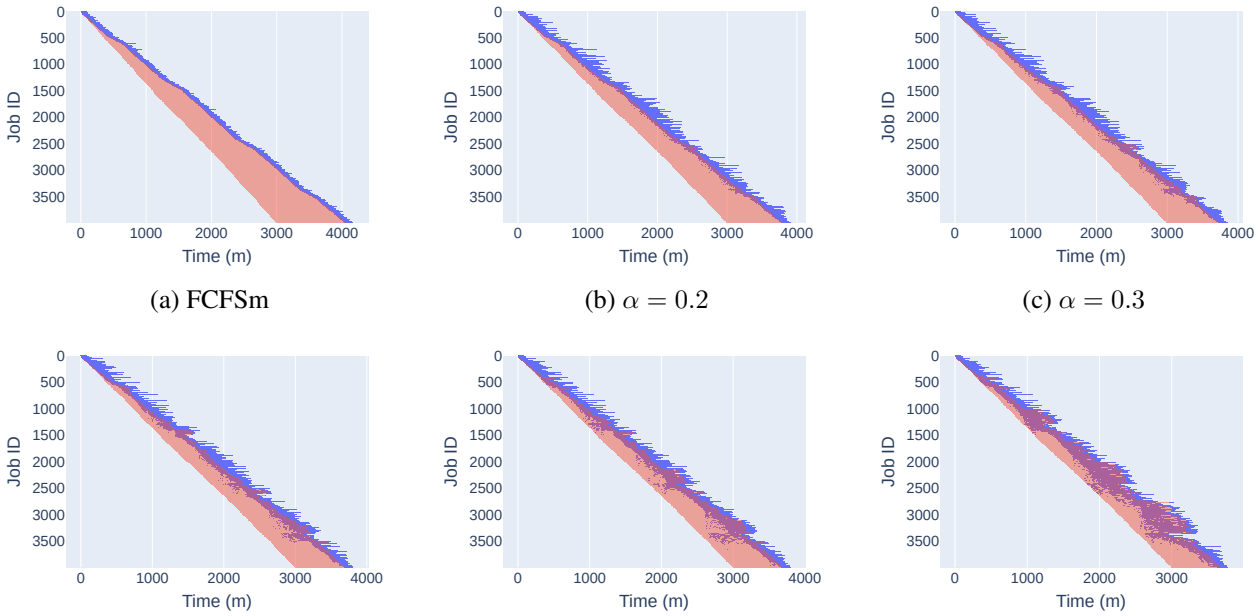
As our proposed priority metric considers fairness, we analyzed job arrival and admission times and visualized them using Gantt charts in Figure 4.12. The results demonstrate that our proposed scheduler considers fairness in its scheduling policy. When we apply the FCFSm algorithm, we, expectedly, see the scheduler admitting jobs to the system based on their arrival time. In comparison, our novel scheduler introduces a visible



(a) FCFSm (b) $\alpha = 0.2$ (c) $\alpha = 0.3$

(d) $\alpha = 0.4$ (e) $\alpha = 0.5$ (f) $\alpha = 0.6$

Figure 4.11: Relative CPU (blue) and PFS utilization (red). CPU utilization is the accumulated processing power of the entire system (i.e., all compute nodes).



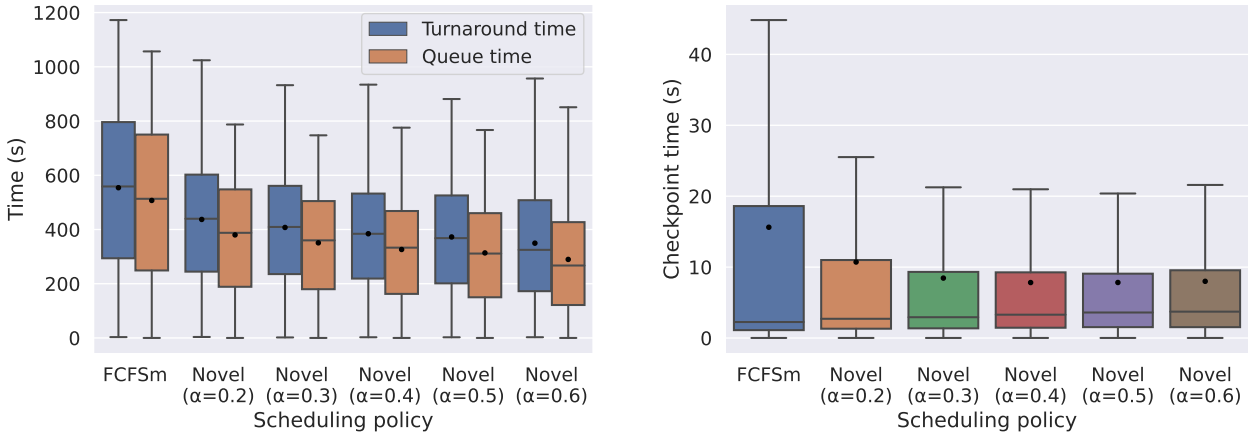
(a) FCFSm (b) $\alpha = 0.2$ (c) $\alpha = 0.3$

(d) $\alpha = 0.4$ (e) $\alpha = 0.5$ (f) $\alpha = 0.6$

Figure 4.12: Gantt charts visualizing running (blue) and waiting jobs (red).

tolerance in admission times for increasing values of α. In a figurative sense, the job reordering intensity α represents a window of possible admission times that gets narrower for smaller and wider for higher values.

We further measured turnaround, queue times, and the time spent during checkpoints for each job to analyze the improvement achieved by our scheduler, represented by box plots in Figure 4.13. The results demonstrate that our scheduler achieves improved turnaround and queue times for increasing values of the reordering intensity α . We can further observe the same trend in particular when analyzing checkpoint times. Our scheduler improves the average checkpointing time by 31.4 % for $\alpha = 0.2$, 45.9 % for $\alpha = 0.3$, and 49.9 % for $\alpha = 0.4$.



(a) Turnaround and queue times.

(b) Checkpoint times (outliers due to readability reasons).

Figure 4.13: Box plots visualizing turnaround, queue, and checkpoint times. Horizontal lines represent the median, and black dots the average values.

However, even though we could observe that $\alpha = 0.6$ yields an optimal system I/O intensity approximating the workload I/O intensity with a negligible difference, the improvement of checkpointing times begins to stagnate for $\alpha \geq 0.4$, indicating that our synthetic workload reached a saturation point and could not further benefit from I/O optimization and reduced PFS congestion.

4.2.6 Summary and future work

We presented the I/O-intensity–aware scheduler, an approach for the combined scheduling of rigid and malleable jobs that anticipates and minimizes congestion on the PFS. To tackle the increasing demand for I/O optimization, we defined customized I/O intensity metrics, introduced our scheduling approach based on our proposed job reordering intensity, and conducted large-scale experiments to assess our scheduler's capability to flatten I/O peaks.

Our results demonstrate that the I/O-intensity–aware scheduler can reduce I/O times by more than 45 %, achieved by minimizing PFS congestion. The adjustable reordering intensity further allows site administrators to modify our scheduler's optimization potential, reflecting the system and the processed workload. As we can observe an increasing interest in malleability, we extended our scheduler to handle reconfigurations of malleable jobs allowing us to further optimize PFS utilization and increase system throughput.

For future versions of the I/O-intensity–aware scheduler, we consider a workload model that includes various I/O patterns. We will extend our application model considering I/O tasks that do not occur recurrently and do not involve all nodes. Furthermore, we will extend our workload model to reflect how optimized an application issues I/O requests, as the achieved I/O bandwidth of an application depends not only on system parameters but also on how optimized I/O requests are. To generate our workload, we will further consider I/O characterizations of real systems to increase the applicability of our scheduling approach (e.g., continuous I/O characterization [6]). Finally, we will evaluate an adaptive approach that dynamically modifies the job reordering intensity during runtime rather than specifying it beforehand. We hypothesize that dynamic adaptation based on the workload I/O intensity can positively affect the fairness during lower I/O intensities and the scheduler's ability to flatten peaks during higher I/O intensities.

4.3 ElastiSim

As experiments on large-scale distributed systems are expensive and time-consuming, simulations are indispensable for evaluating scheduling algorithms for malleable jobs. Although simulators such as Batsim, Alea, or AccaSim exist, they do not support malleable jobs and the necessary scheduling protocols to support mal-

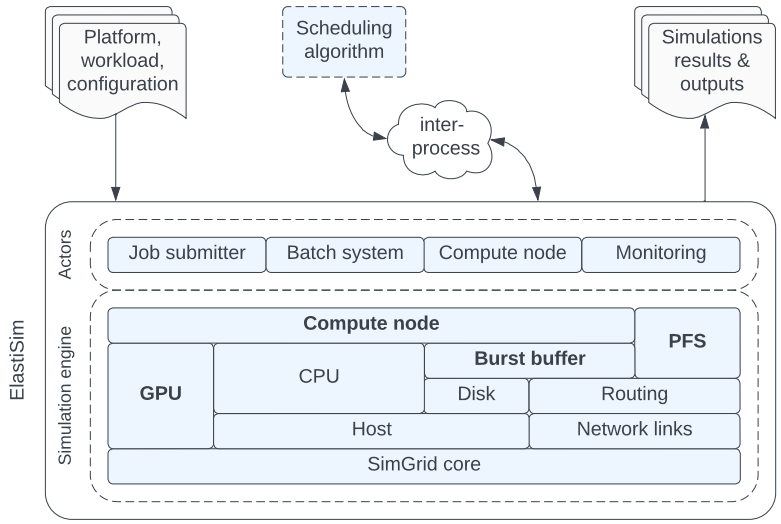


Figure 4.14: The architecture of ElastiSim. Extensions to SimGrid are highlighted in bold. Compute nodes act as part of the simulation engine *and* as system actors as they are considered a system resource but also communicate with the batch system on a continuous basis.

leability. To fill this gap of simulators supporting malleability, we developed ElastiSim [36] a batch-system simulator to evaluate algorithms for the combined scheduling of rigid, moldable, and malleable jobs.

ElastiSim is a discrete-event simulator written in C++ and based on SimGrid [7], a simulation framework for distributed systems such as HPCs, clouds, and grids, modeling network communication as flows within the simulated platform. We employ and extend SimGrid features to simulate all components relevant to scheduling malleable jobs on distributed systems. Through the interfaces we provide, users can integrate their scheduling algorithms and apply them to scenarios that users can describe in detail as simulation inputs. Figure 4.14 describes the architecture of ElastiSim, separating the concerns of platform simulations, actors within the simulated scenario, and user-provided inputs.

As the validity of simulation results relies significantly on the simulated workload, we provide a detailed workload modeling approach. Users describe their workload as a set of jobs defining attributes relevant to the scheduler (e.g., number of nodes) and an application model that describes the simulated application. We model applications as multiple phases, each containing various low-level activities (i.e., tasks). Malleable jobs introduce a scheduling point after each phase, allowing them to respond to reconfiguration request issued by the scheduler.

To simulate a scheduling scenario, ElastiSim requires users to provide the scheduling algorithm, the platform description, the workload, and a configuration file describing the conditions of the simulated scenario. The scheduling algorithm provided by the user is a standalone process, attaching itself to the simulator process that simulates the platform and applies the decisions of the scheduler process. Figure 4.15 illustrates the actors involved in simulating a scheduling scenario. After the simulated scenario, ElastiSim provides detailed results comprising job runtimes and system utilization.

We evaluated ElastiSim by establishing two experiments to (1) validate our proposed workload model and (2) assess ElastiSim's capabilities to simulate large-scale scenarios. As deep-learning applications are inherently malleable (i.e., reconfigurable after each epoch), we considered several deep-learning networks, modeled them in ElastiSim, and compared the runtimes per epoch of the real-world application and the simulated application. To apply Elastim in a large-scale scenario, we created a workload based on logs of the Microsoft DL cluster Philly, randomly assigned a deep-learning network to each job, and simulated six different scheduling algorithms, with four applicable to malleable jobs. The results have demonstrated that ElastiSim can model real-world applications in simulated scenarios with high accuracy and provides consistent and meaningful results in large-scale simulations.

To expedite the development of scheduling algorithms for malleable jobs, we published ElastiSim as an

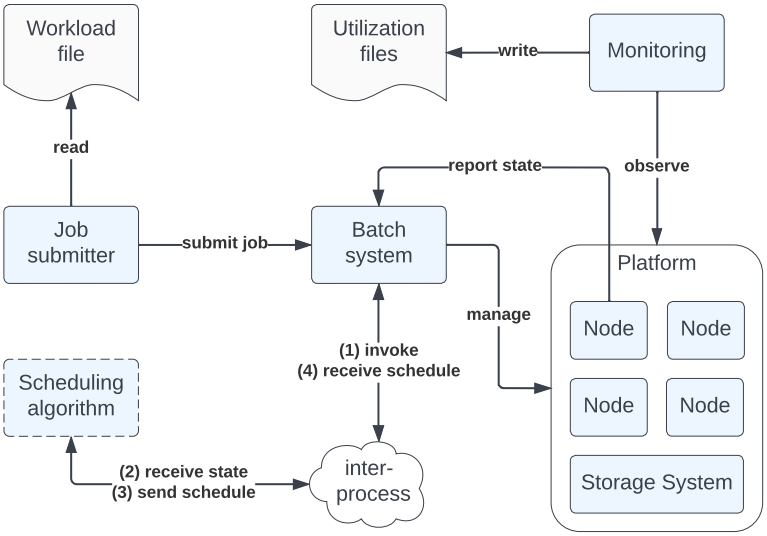


Figure 4.15: Actors and their communication during a simulated scenario. ElastiSim does not consider the scheduling algorithm as a separate actor although it runs as a stand-alone process outside the simulator process (dashed lines).

open-source project on GitHub[2][3]. We further provide a Python interface that facilitates the evaluation of novel algorithms. For future versions of ElastiSim, we plan to provide semantics for workflow support and extend our application model to initiate reconfiguration requests to support evolving jobs.

[2] https://github.com/elastisim
[3] https://elastisim.github.io

Chapter 5

Conclusion

In exascale computing, data-intensive applications deal with massive amounts of data, often characterized by large-scale data processing, analysis, and storage requirements. These applications typically require powerful computational capabilities and efficient data management techniques to handle the volume, velocity, and variety of data involved. Here, resource scheduling plays a crucial role due to the scale and complexity of the systems involved. We could indicate several key reasons that highlight the importance of resource scheduling in this context, including system throughput, performance optimization, efficient resource utilization, load balancing, data access, data movement, and fault resilience. Job malleability, as the ability to modify or adapt resources assigned to running jobs, provides a potential optimization opportunity in resource scheduling mechanisms.

To this end, in this deliverable, we focused on proposing resource scheduling policies and algorithms considering the presence of malleable jobs in the job queue of HPC systems. We introduced several malleability policies for computation and I/O. We also proposed two scheduling algorithms to deal with the problem. To evaluate the scheduling algorithms dealing with malleable jobs, we introduced ElastiSim.

In the policy proposed for the malleability of compute nodes, we followed a node-stealing approach. In the first study, we considered the problem from a fault-tolerance angle, including various resilience mechanisms to choose the proper job and proper time for the interruption. Then we explained the I/O malleability policies using two approaches: GekkoFS and Hercules. We illustrated the redesign of the GekkoFS and also the prototype proposed based on Hercules to include malleability for the I/O operations.

We presented first an extension of the EASY backfilling scheduler that benefits from the potential use of malleability operations to increase system utilization and decrease the makespan of jobs. We observed at least 30% improvement in both makespan and resource utilization, compared to pure EASY backfilling, which ignores malleability. Then we presented an I/O intensity-aware scheduler that uses the I/O intensity of jobs reduces the possibility of congestion occurring on the shared PFS. Our extensive evaluation showed an improvement of more than 49% for average I/O times and up to 9.6% reduction in makespan compared to the FCFS scheduler.

Appendix A

How to use ElastiSim

The easiest way to get started with ElastiSim is by cloning the example project available on GitHub[1]. This scenario simulates an FCFS (first come, first serve) scheduling algorithm applied on 32 rigid jobs with alternating compute and I/O phases running on a crossbar topology with 128 compute nodes. The following steps will create a Docker container including all the required libraries for ElastiSim and start the simulation.

A.1 Installation

To build the container required to run ElastiSim, install Docker and execute the following command:

```
docker build -t elastisim .
```

A.2 Simulation

To run the simulation, execute the following commands in two different sessions:

A.2.1 Linux:

```
docker run -v $PWD/data:/data -v $PWD/algorithm:/algorithm -u `id -u
   (cont.)$USER` --name elastisim -it --rm elastisim /data/input/
   (cont.)configuration.json --log=root.thresh:warning
docker exec -u `id -u $USER` -it elastisim python3 /algorithm/algorithm.
   (cont.)py
```

A.2.2 Mac OS:

```
docker run -v $PWD/data:/data -v $PWD/algorithm:/algorithm --name
   (cont.)elastisim -it --rm elastisim /data/input/configuration.json --
   (cont.)log=root.thresh:warning
docker exec -it elastisim python3 /algorithm/algorithm.py
```

A.2.3 Windows (PowerShell):

[1] https://github.com/elastisim/example-project

```
docker run -v ${PWD}\data:/data -v ${PWD}\algorithm:/algorithm --name
   (cont.)elastisim -it --rm elastisim /data/input/configuration.json --
   (cont.)log=root.thresh:warning
docker exec -it elastisim python3 /algorithm/algorithm.py
```

The first command runs the ElastiSim simulator process and accepts two inputs:

- the configuration file (JSON)
- the logging level

For a more detailed output, change `--log=root.thresh:warning` to `--log=root.thresh:info` (caution: verbose).

The second command runs the scheduling algorithm.

Appendix B

Terminology

- Ad-hoc storage system: ephemeral storage system that only exists in a determined period, i.e., during a job's execution.
- Amdahl's law: the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used
- Backfilling: allowing shorter jobs to be scheduled ahead of longer jobs if this does not increase the waiting time of jobs with higher priority
- Batch systems: a software component in the HPC systems responsible for resource management and job execution
- Burst buffer: fast intermediate storage layer positioned between the job and the back-end storage systems in the HPC system architecture
- ElastiSim: a simulator for the scheduling of malleable jobs on HPC clusters
- FCFS: first-come, first-served scheduling algorithm
- GekkoFS: a user-level distributed file system for HPC clusters
- Hercules: a distributed file system with scalable metadata servers cluster and scalable and fault-tolerant data servers cluster
- I/O intensity: the amount of I/O relative to the computational workload a program performs
- Malleable job: a class of parallel jobs that can be reconfigured during execution on behalf of the scheduler
- Malleation operations: shrinking and expanding resources for malleable jobs to improve the target objectives of the scheduling algorithm
- MEBF : a varient of EASY Backfilling scheduling to support malleable jobs.
- Parallel file system (PFS): a storage system to store data across multiple networked servers to facilitate high-performance access through simultaneous input/output operations
- Slurm: a widely used batch system in HPC

Bibliography

[1] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I. Ingólfsson, Joseph Koning, Tapasya Patki, Thomas R.W. Scogland, Becky Springmeyer, and Michela Taufer. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems*, 110:202–213, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19317169, doi:https://doi.org/10.1016/j.future.2020.04.006.

[2] Gene M. Amdahl. Oral history interview with Gene M. Amdahl, Charles Babbage Institute, University of Minnesota, hdl:11299/104341, 1989.

[3] André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip H. Carns, Toni Cortes, Scott Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, and Marc-Andre Vef. Ad hoc file systems for high-performance computing. *J. Comput. Sci. Technol.*, 35(1), 2020.

[4] Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano, and Raffaele Tripiccione. Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. *CoRR*, abs/1703.02788, 2017. URL: http://arxiv.org/abs/1703.02788, arXiv:1703.02788.

[5] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proc. of the ACM/IEEE Conference on Supercomputing*, SC13, pages 1–12. ACM, 11 2013.

[6] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2011. doi:10.1109/MSST.2011.5937212.

[7] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *JPDC*, 74(10):2899–2917, 2014.

[8] Konstantinos Chasapis, Jean-Yves Vet, and Jean-Thomas Acquaviva. Benchmarking Parallel File System Sensitiveness to I/O Patterns. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 427–428. IEEE, 2019.

[9] N. Cheriere, M. Dorier, G. Antoniu, S. M. Wild, S. Leyffer, and R. Ross. Pufferscale: Rescaling HPC Data Services for High Energy Physics Applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 182–191, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[10] Marco D'Amico, Marta Garcia-Gasulla, Víctor López, Ana Jokanovic, Raül Sirvent, and Julita Corbalan. Drom: Enabling efficient and effortless malleability for resource managers. In *Workshop Proceedings of the 47th International Conference on Parallel Processing*, ICPP Workshops '18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3229710.3229752.

[11] Marco D'Amico, Ana Jokanovic, and Julita Corbalan. Holistic slowdown driven scheduling and resource management for malleable jobs. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3337821.3337909.

[12] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in hpc systems through cross-application coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 155–164, 2014. doi:10.1109/IPDPS.2014.27.

[13] Yishu Du, Loris Marchal, Guillaume Pallez, and Yves Robert. Doing better for jobs that failed: node stealing from a batch scheduler's perspective. working paper or preprint, April 2022. URL: https://inria.hal.science/hal-03643403.

[14] Yuping Fan, Zhiling Lan, Paul Rich, William E. Allcock, Michael E. Papka, Brian Austin, and David Paul. Scheduling beyond CPUs for HPC. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, jun 2019. URL: https://doi.org/10.1145%2F3307681.3325401, doi:10.1145/3307681.3325401.

[15] D Feitelson. Job scheduling in multiprogrammed parallel systems ibm research report rc 19970. *Second Revision*, 1997.

[16] Dror G Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992. URL: https://www.sciencedirect.com/science/article/pii/074373159290014E, doi:https://doi.org/10.1016/0743-7315(92)90014-E.

[17] Dror G. Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–26, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[18] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of hpc applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022, 2015. doi:10.1109/IPDPS.2015.116.

[19] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2619239.2626334.

[20] Thomas Haynes. Network file system (NFS) version 4 minor version 2 protocol. *RFC*, 7862, 2016.

[21] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R.W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable I/O-aware job scheduling for burst buffer enabled hpc clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 69–80, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2907294.2907316.

[22] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in hpc resource management systems: Queuing vs. planning. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–20, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[23] J. Hungershofer. On the combined scheduling of malleable and rigid jobs. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 206–213, 2004. doi:10.1109/SBAC-PAD.2004.27.

[24] Hameed Hussain, Saif Ur Rehman Malik, Abdul Hameed, Samee Ullah Khan, Gage Bickler, Nasro Min-Allah, Muhammad Bilal Qureshi, Limin Zhang, Wang Yongji, Nasir Ghani, Joanna Kolodziej, Albert Y. Zomaya, Cheng-Zhong Xu, Pavan Balaji, Abhinav Vishnu, Fredric Pinel, Johnatan E. Pecero, Dzmitry Kliazovich, Pascal Bouvry, Hongxiang Li, Lizhe Wang, Dan Chen, and Ammar Rayes. A survey on resource allocation in high performance distributed computing systems. *Parallel Computing*,

39(11):709–736, 2013. URL: https://www.sciencedirect.com/science/article/pii/S016781911300121X, doi:https://doi.org/10.1016/j.parco.2013.09.009.

[25] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. Technical report, University of Illinois at Urbana-Champaign, USA, 1993.

[26] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *6th Annual International Systems and Storage Conference, SYSTOR '13*. ACM, 2013.

[27] Weihao Liang, Yong Chen, Jialin Liu, and Hong An. Cars: A contention-aware scheduler for efficient resource management of hpc storage systems. *Parallel Computing*, 87:25–34, 2019. URL: https://www.sciencedirect.com/science/article/pii/S016781911830382X, doi:https://doi.org/10.1016/j.parco.2019.04.010.

[28] David A. Lifka. The anl/ibm sp scheduling system. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '95, page 295–303, Berlin, Heidelberg, 1995. Springer-Verlag.

[29] Seung-Hwan Lim and Youngjae Kim. A quantitative model of application slow-down in multi-resource shared systems. *Performance Evaluation*, 108:32–47, 2017. URL: https://www.sciencedirect.com/science/article/pii/S0166531616302309, doi:https://doi.org/10.1016/j.peva.2016.10.004.

[30] Gonzalo Martín, Maria-Cristina Marinescu, David E. Singh, and Jesús Carretero. Flex-mpi: An mpi extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In *Euro-Par 2013 Parallel Processing*, pages 138–149, 2013.

[31] Dieter an Mey, Scott Biersdorf, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-p: A unified performance measurement system for petascale applications. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 85–97, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[32] Alberto Miranda, Sascha Effert, Yangwook Kang, Ethan L Miller, Andre Brinkmann, and Toni Cortes. Reliable and randomized data distribution strategies for large scale storage systems. In *18th International Conference on High Performance Computing*. IEEE, 2011.

[33] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE transactions on parallel and distributed systems*, 12(6):529–543, 2001.

[34] A.W. Mu'alem and D.G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001. doi:10.1109/71.932708.

[35] Daniel Nichols, Aniruddha Marathe, Kathleen Shoga, Todd Gamblin, and Abhinav Bhatele. Resource utilization aware job scheduling to mitigate performance variability. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 335–345, 2022. doi:10.1109/IPDPS53621.2022.00040.

[36] Taylan Özden, Tim Beringer, Arya Mazaheri, Hamid Mohammadi Fard, and Felix Wolf. Elastisim: A batch-system simulator for malleable workloads. In *Proc. of the 51st International Conference on Parallel Processing (ICPP), Bordeaux, France*. ACM, August 2022.

[37] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kale. A batch system with efficient adaptive scheduling for malleable and evolving applications. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 429–438, 2015. doi:10.1109/IPDPS.2015.34.

[38] Yingjin Qian, Wen Cheng, Lingfang Zeng, Marc-André Vef, Oleg Drokin, Andreas Dilger, Shuichi Ihara, Wusheng Zhang, Yang Wang, and André Brinkmann. Metawbc: Posix-compliant metadata write-back caching for distributed file systems. In *SC '2: The International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, Texas, USA, 2022*. IEEE, 2022.

[39] Francisco José Rodrigo Duro, Fabrizio Marozzo, Javier García Blas, Jesús Carretero Pérez, Domenico Talia, and Paolo Trunfio. Evaluating data caching techniques in DMCF workflows using Hercules. *e-archivo.uc3m.es*, 2015.

[40] Emilia Rosti, Giuseppe Serazzi, Evgenia Smirni, and Mark S Squillante. Models of parallel applications with large computation and i/o requirements. *IEEE Transactions on Software Engineering*, 28(3):286–307, 2002.

[41] Mehdi Sheikhalishahi, Richard M. Wallace, Lucio Grandinetti, José Luis Vazquez-Poletti, and Francesca Guerriero. A multi-dimensional job scheduling. *Future Generation Computer Systems*, 54:123–131, 2016. URL: https://www.sciencedirect.com/science/article/pii/S0167739X1500076X, doi:https://doi.org/10.1016/j.future.2015.03.014.

[42] David E. Singh and Jesus Carretero. Combining malleability and I/O control mechanisms to enhance the execution of multiple applications. *Journal of Systems and Software*, 148:21–36, 2019.

[43] Rajesh Sudarsan and Calvin J Ribbens. Scheduling resizable parallel applications. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[44] Hongyang Sun, Redouane Elghazi, Ana Gainaru, Guillaume Aupy, and Padma Raghavan. Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 194–203, 2018. doi:10.1109/IPDPS.2018.00029.

[45] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007. doi:10.1109/TPDS.2007.70606.

[46] Dan Tsafrir and Dror G. Feitelson. The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help. In *2006 IEEE International Symposium on Workload Characterization*, pages 131–141, 2006. doi:10.1109/IISWC.2006.302737.

[47] Shu-Mei Tseng, Bogdan Nicolae, Franck Cappello, and Aparna Chandramowlishwaran. Demystifying asynchronous i/o interference in hpc applications. *The International Journal of High Performance Computing Applications*, 35(4):391–412, 2021. arXiv:https://doi.org/10.1177/10943420211016511, doi:10.1177/10943420211016511.

[48] Gladys Utrera, Siham Tabik, Julita Corbalan, and Jesús Labarta. A job scheduling approach for multi-core clusters based on virtual malleability. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 191–203, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[49] Mihaela-Andreea Vasile, Florin Pop, Radu-Ioan Tutueanu, Valentin Cristea, and Joanna Kołodziej. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Generation Computer Systems*, 51:61–71, 2015. Special Section: A Note on New Trends in Data-Aware Scheduling and Resource Provisioning in Modern HPC Systems. URL: https://www.sciencedirect.

com/science/article/pii/S0167739X14002532, doi:https://doi.org/10.1016/j.future.2014.11.019.

[50] M. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann. GekkoFS - A Temporary Distributed File System for HPC Applications. In *2018 IEEE International Conference on Cluster Computing*, pages 319–324, 2018.

[51] Marc-André Vef, Alberto Miranda, Ramon Nou, and André Brinkmann. From static to malleable: Improving flexibility and compatibility in burst buffer file systems. In *High Performance Computing. ISC High Performance 2023 International Workshops - Hamburg, Germany*. Springer, 2023.

[52] Chen Wang. Detecting data races on relaxed systems using recorder, 2022.

[53] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of HPC applications. In *HPDC '21: The 30th International Symposium on High-Performance Parallel and Distributed Computing, Virtual Event*. ACM, 2021.

[54] Junweon Yoon, Taeyoung Hong, Chanyeol Park, Seo Young Noh, and Heonchang Yu. Log analysis-based resource and execution time improvement in HPC: A case study. *Applied Sciences (Switzerland)*, 10(7), 2020. doi:10.3390/app10072634.

[55] Francieli Zanon Boito, Guillaume Pallez, Luan Teylo, and Nicolas Vidal. IO-SETS: Simple and efficient approaches for I/O bandwidth management. working paper or preprint, April 2022. URL: https://hal.inria.fr/hal-03648225.

[56] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/o-aware batch scheduling for petascale computing systems. In *2015 IEEE International Conference on Cluster Computing*, pages 254–263, 2015. doi:10.1109/CLUSTER.2015.45.