MPI Malleability Validation under Replayed Real-World HPC Conditions

Sergio Iserte[a,*], Maël Madon[b], Georges Da Costa[b], Jean-Marc Pierson[b] and Antonio J. Peña[a]

[a]Barcelona Supercomputing Center (BSC), Barcelona, Spain
[b]University of Toulouse, CNRS, Toulouse INP, IRIT, Toulouse, France

ARTICLE INFO

Keywords:
Dynamic Resource Management
Replay with Feedback
Malleability
Workload Characterization
Cluster Computing

ABSTRACT

Dynamic Resource Management (DRM) techniques can be leveraged to maximize throughput and resource utilization in computational clusters. Although DRM has been extensively studied through analytical workloads and simulations, skepticism persists among end administrators and users regarding their feasibility under real-world conditions. To address this problem, we propose a novel methodology for validating DRM techniques, such as malleability, in realistic scenarios that reproduce actual cluster conditions of jobs and users by replaying workload logs on a High-performance Computing (HPC) infrastructure. Our methodology is capable of adapting the workload to the target cluster. We evaluate our methodology in a malleability-enabled 125-node partition of the Marenostrum 5 supercomputer. Our results validate the proposed method and assess the benefits of MPI malleability on a novel use case of a pioneer user of malleability (our "PhD Student"): parallel-efficiency-aware malleability reduced a malleable workload time by 27% without delaying the baseline workload and increasing global resource utilization by 8%.

1. Introduction

High-performance computing (HPC) facilities are critical for advancing scientific research, engineering, and data analysis across various domains, from genomic research [1, 2] to multi-physics simulations [3, 4, 5] through artificial intelligence [6, 7, 8]. These computational clusters rely on efficiently utilizing resources to maximize their productivity. Dynamic resource management (DRM) techniques have emerged as key strategies to improve the utilization of these systems [9, 10, 11]. These techniques enable the flexible allocation and reallocation of computational resources and reconfiguring jobs accordingly to adapt to the dynamic nature of HPC workloads. Among others, malleability based on the Message-Passing Interface (MPI) is one of the most extended DRM approaches to process layout reshape and data redistribution [12, 13, 14, 15].

Despite their potential, DRM techniques face several challenges that hinder their adoption in production environments [16]. Particularly, MPI malleability requires 1) a malleability-ready resource management system (RMS), which many existing facilities do not possess, and 2) a malleability framework to develop malleable applications compatible with the RMS and the available distributed parallel runtime systems in the cluster. These kinds of limitations increase the complexity of evaluating DRM techniques with realistic workloads and foster skepticism among end users and system administrators regarding DRM's practical benefits and feasibility. In this regard, previous studies evaluating DRM techniques have relied on simplistic simulations, biased benchmarks, or synthetic traces, often failing to accurately represent real-world systems' conditions. These analytic workloads do not capture the variability and unpredictability of user behaviors observed in production systems. As a result, researchers in this field have faced challenges in conducting impactful demonstrations that significantly influence the community.

To address the challenges of evaluating DRM techniques, we propose a novel methodology that leverages supercomputer logs to generate realistic workloads based on actual user submission behaviors, including temporal patterns across hours and days of the week. Our approach introduces a mechanism called the *User-Based Submitter*, designed to scale across different computing clusters. Rather than relying on synthetic simulations, this method replays real user interactions from production supercomputers, enabling a more accurate and comprehensive evaluation of DRM techniques.

We further incorporate malleable jobs into these replayed workloads to assess the impact of DRM on job execution and system performance. This allows us to observe the effects of job malleability and resource reallocation in a controlled yet realistic environment. Notably, the study includes the perspective of a pioneering user—submitting the first malleable jobs in a 125-node partition—demonstrating the potential of our methodology to validate previous findings and uncover new insights in real HPC settings. It is worth noting that the cluster size used in our experiments ranks among the largest evaluated in state-of-the-art malleability studies (see Section 2).

In summary, this paper presents 1) **a methodology to replay logs in any computational cluster keeping the logic behind user submissions**. Instead of a mathematical model, this methodology uses the techniques of user sampling from a recorded log and replay with feedback to reproduce the workload in a target system (see Section 3). This paper

*Corresponding author
sergio.iserte@bsc.es (S. Iserte)
ORCID(s): 0000-0003-3654-7924 (S. Iserte); 0000-0001-9476-4682 (M. Madon); 0000-0002-3365-7709 (G. Da Costa); 0000-0001-8948-0474 (J. Pierson); 0000-0002-3575-4617 (A.J. Peña)

particularly 2) **designs and analyzes a novel case study of the adoption of malleability in a production system**. For this purpose, the authors exemplify this event with a new user ("the student") who is the pioneer in submitting malleable jobs to the cluster. Among all the possible scenarios, the authors have decided to develop a case that paves the way to more complex malleable workloads (see Section 4). Similarly to hardware simulators that have to be validated in actual hardware to be published, this paper presents, for the first time in the literature, 3) **the validation of MPI malleability using a real workload in a malleability-enabled supercomputer**, without simulations or synthetic benchmarks (see Section 5).

The paper ends with a discussion of the applicability of the presented technique and the meaningful results obtained in the evaluation of the validation methodology (see Section 6). It concludes with a summary of our contributions and findings (see Section 7).

2. Related Work

This section reviews the previous efforts done in the fields of workload replay and dynamic resource management.

2.1. Workload Replay

A classical approach to evaluate resource management policies consists of replaying a historical workload from its log. However, modifying the target infrastructure or scheduling policies impacts the system's performance (e.g., response time, computing speed, resource availability), which has a crucial impact on the submission behavior of its users [17, 18]. As a result, many previous works in the field have generalization issues, since they do not account for these behaviors. A solution is to perform closed-loop simulations [19], where the submitted workload adapts to the simulated performance of the system. To do so, the authors in [20] extract relevant submission patterns from historical workloads to replay them in simulations. More recently, the authors in [21] provide an in-depth study of this technique, named "replay with feedback". In [22], the authors go one step further and suggest using *resampling* with feedback for performance evaluation where the historical workload is scaled to the target infrastructure through user sampling.

2.2. Dynamic Resource Management

The MPI paradigm is the *de facto* standard for implementing distributed parallel applications designed to run on computational clusters. MPI provides efficient and straightforward mechanisms for communication among different processes, known as ranks, each with its own memory address space. In a traditional MPI job, a fixed number of ranks are initiated at the start and remain active until the completion of the job. This static allocation of resources may lead to inefficiencies, especially in dynamic and heterogeneous computing environments.

MPI malleability addresses these inefficiencies through DRM. This technique allows the number of MPI ranks to be modified during the execution of a program, enabling the application to resize on the fly [23]. Malleable MPI applications may adapt to changing computational resources, improving overall system utilization and application performance. Although MPI malleability could be seen as a variant of checkpoint/restart, most modern solutions implement on-memory data redistribution, reducing overheads from accessing storage media.

Closely related to malleability is the concept of moldability. While malleability involves adjusting the number of MPI ranks during the execution of a job, moldability determines the number of ranks at the time of job submission before its initialization [24]. In this regard, jobs specify a range of sizes, allowing the job scheduler to choose the most suitable configuration based on current system availability and load. This approach provides an extra degree of flexibility by enabling better resource allocation at job scheduling time, which can lead to improved system efficiency and potentially shorter wait times for job initiation. However, unlike malleable applications, moldable applications do not adjust their resource usage dynamically during runtime.

Surveys in [25, 26] extensively review the state–of–the–art of malleability in HPC systems. Instead, we focus on how malleability frameworks and actual RMSs have been evaluated with workloads to demonstrate the usefulness of DRM in clusters.

ReSHAPE is a coupled solution for adaptive workloads, including its specific reconfiguration libraries, scheduler, and runtime system. This strong integration forces ReSHAPE users to develop applications that are compatible with this system. In [27], ReSHAPE is evaluated in a 50-node cluster with a 5-job workload composed of the benchmarks LU, MM, Master-worker, Jacobi, and FFT, submitted simultaneously. A subsequent publication added to the workload job instances of a malleable version of LAMMPS [28].

PARM (acronym of Power-Aware Resource Manager) relies on over-provisioning, power capping, and job malleability, based on CHARM++ and Slurm [29]. PARM is evaluated in a 38-node cluster with a 5-job workload comprising the benchmarks and applications Wave2D, Jacobi2D, LeanMD, Lulesh, and AMR. Larger experiments relied on the Slurm simulator with a log of 68,936 jobs submitted to a 40,960-node cluster. The authors assume that all the jobs in the workload are malleable and assign them random scalability from a set of samples.

The solution developed by Prabhakaran et al. combines AMPI (based on CHARM++) with the RMS Torque/Maui to tackle malleable jobs [30]. The authors evaluate in a 15-node cluster a workload based on a modified version of the ESP benchmark to contain various percentages of rigid, malleable, and evolving jobs composed of 230 instances. While the rigid jobs followed the benchmark directives, the flexible jobs were a malleable version of LeanMD and an evolving implementation of the synthetic Quadflow.

The Dynamic Management of Resources Library (DMRlib) (detailed in Section 3.3) was evaluated in a 129-node cluster with four benchmarks and applications (CG, Jacobi, N-Body, HPG-Aligner [31]) instantiated in 2,000-job workloads with various percentages of malleable jobs and turning on and off malleability in the applications of the workloads [32]. The authors leveraged the Feitelson synthetic workload generator [33] to evaluate job scheduling algorithms in HPC. This generator provides a way to simulate job arrival patterns, job sizes, and resource demands based on empirical data from real-world systems.

While prior research has explored the potential of DRM through benchmarks and simulations, these studies often rely on synthetic workloads with arbitrarily defined job sizes and submission patterns—such as steady or bursty arrivals—that fail to capture the complexity and variability of real-world HPC usage. These simplified models limit the generalizability and applicability of the findings to production systems.

In contrast, our work introduces a methodology that replays actual user behavior extracted from supercomputer logs, preserving the nuanced temporal and structural characteristics of real workloads. This approach enables the validation of past and future synthetic setups and their results, overcoming the limitations of synthetic modeling where user behavior is too complex to be accurately captured by mathematical functions. By grounding the evaluation in real-world conditions, we aim to bridge the gap between simulation-based studies and practical deployment in HPC infrastructures.

3. Methodology

In order to study DRM under real-world conditions, our method consists of reproducing the nominal activity of a cluster computing infrastructure. We use a workload submitter based on the logic of users (Section 3.1). The nominal activity is represented by submitting non-malleable jobs. These jobs, later called "baseline workload", are reproduced from historical workload logs and sized to our testbed through user sampling (Section 3.2). To study the effects of DRM on top of the baseline workload, the submitter includes an additional user leveraging moldability and malleability in her jobs, thanks to a malleable runtime (Section 3.3).

3.1. User-Based Workload Submitter

The core of the experiments presented in this paper relies on a piece of software running throughout the whole experiment duration: the User-Based Submitter (UBS)[1].UBS reproduces HPC users issuing job submissions to the Slurm resource manager at certain timestamps. There are two types of users as described below:

- *Traditional users* submit jobs whose execution time, number of parallel resources, and submission time are given in an input file, based on a historical workload. These are used to replay the baseline workload. Consequently, the submitted jobs replicate the traditional rigid workload, allocating the requested resources during the desired time.

- *Generative users* submit malleable jobs, using feedback on the status of previous submissions to decide on the following job submissions (see [21]). More precisely, a generative user takes as input a triplet $(t_0, \Delta t, N)$ with t_0 the timestamp of their first submission, Δt the think time between the end of the previous job and the submission of the new job and N the total number of submissions from this user.

In the UBS, traditional and generative users are managed in two dedicated threads. The thread managing the traditional users sleeps until the next submission time, then calls a shell script issuing the job submission, passing it all relevant parameters about the job (i.e.: type, duration, or number of resources). The thread of the generative users uses a socket to be informed of the completion of previous jobs. When a job owned by a generative user terminates, a new thread is started, sleeping for Δt seconds before calling the submission script.

3.2. User Sampling

To create a realistic baseline workload of the actual activity in a cluster platform, we leverage historical workload logs. The challenge is to adapt this workload to the size of our testbed *without losing the logic of submission of the original users*. Thus, instead of randomly sampling jobs from the original log, we rather perform a sampling of *users*, proceeding as described below (see Algorithm 1).

First, we choose a target level of activity for the platform, in terms of average or median load submitted per day. For example, if the testbed has 100 nodes and we target 85% activity, we want the baseline workload to have approximately 85×24 node-hours per day. To reach the level, we proceed iteratively by randomly adding users to a user pool. At each step, we calculate the load submitted daily by the users in the pool by summing the product of execution time and number of nodes for all jobs submitted by these users in the original log. If the activity reaches the desired level, the process is stopped; otherwise, we add a new user. The last added user may overshoot the target; in this case, that user is removed from the pool before adding another.

3.3. Dynamic Resource Management

This research enables DRM by leveraging moldability and malleability thanks to the Dynamic Management of Resources Library (DMRlib) [32]. DMRlib is a high-level API that facilitates the adoption of malleability in HPC codes. DMRlib implements a communication layer between the parallel distributed runtime (PDR) and the RMS, driving the management of processes and resources transparently to the user while providing flexibility to increase productivity and resource utilization of HPC facilities.

Figure 1 depicts the operation of dynamic resource management in DMRlib, which is described as follows:

[1]Code available at https://gitlab.bsc.es/siserte/ubs

Algorithm 1 User sampling from workload

Input: $\bar{m}$ = the target median/average load per day
Input: M = the number of nodes in the target platform
$U \leftarrow$ list of users in the workload
$pool \leftarrow \{\}$, $m \leftarrow 0$
while $m \notin [.9\bar{m}, 1.1\bar{m}]$ **do**
  $u \leftarrow$ draw a random user from U, without replacement
  if max(nodes required by u) $< M$ **then**
    $pool$.add(u)
    $m \leftarrow$ load per day from users in $pool$
    if $m > 1.1\bar{m}$ **then**
      $pool$.remove(u)
      $m \leftarrow$ load per day from users in $pool$
    end if
  end if
end while
if $m \notin [.9\bar{m}, 1.1\bar{m}]$ **then**
  print("SAMPLING FAILED!")
end if
Output: $pool$

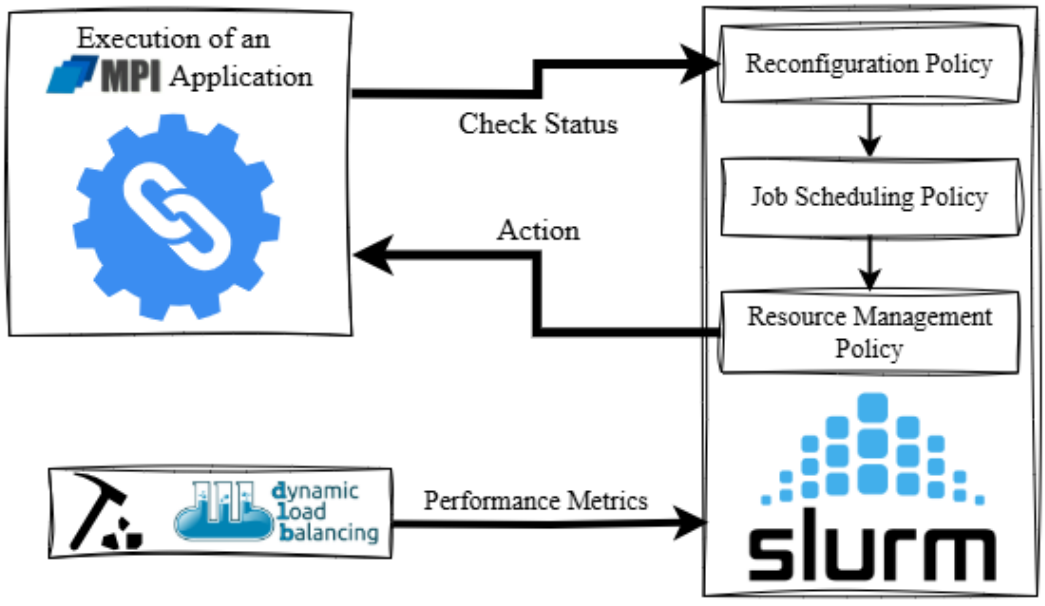


Figure 1: DMRlib Application–MPI–Slurm communication.

- During execution, jobs periodically expose their readiness for reconfiguration to the RMS.
- This communication occurs at a synchronization point specified in the code, where the reconfiguration process may commence. In iterative applications, the end of an iteration often serves as an ideal synchronization point.
- The RMS, with its cluster-wise information and the performance metrics provided by a monitor, determines reconfiguration actions following its defined policies, and informs the PDR.
- If this results in a change to the job size, the RMS reallocates resources and returns the new number of processes, which may involve either expanding or shrinking the job.

- Finally, the PDR redistributes the data among the processes according to the application's guidelines, allowing jobs to continue execution with the new process layout from the point where the reconfiguration was triggered.

In this work, DMRlib has been configured with MPICH[2], as the PDR, and Slurm[3] as the RMS [34]. DMRlib integrates a monitoring module to track performance metrics during job execution and react accordingly by requesting/suggesting expansion or shrinking actions to the RMS. Metrics are gathered by TALP [35], which collects "POP metrics"[4]. This set of metrics is organized hierarchically and is multiplicative; that is, the value of the parent metric is equal to the product of the child metrics. Among others, Parallel Efficiency (PE) reveals the inefficiency in splitting computation over processes and then communicating data among processes. PE is a compound metric whose components reflect two critical factors to attain good parallel performance by ensuring even distribution of computational work across processes (load balance) and minimizing time communicating data among processes (communication efficiency).

4. Experimental Setup

We implement the method described above to evaluate the impact on the job submissions of a user enabling DRM in a computing cluster. For the purpose of this narrative, we portray this user as a PhD student working under a tight deadline, submitting malleable instances of a scientific application. This section describes the testbed, workload, and specific parameters of the experimental campaign.

4.1. Cluster Computing Infrastructure

The evaluation is performed in the general-purpose partition of the Marenostrum 5 (MN5) supercomputer, a pre-exascale machine integrated into the EuroHPC-JU European supercomputing infrastructure[5]. The nodes of this partition are based on Intel Sapphire Rapids (4^{th} Generation Intel Xeon Scalable Processors), each equipped with two Intel Xeon Platinum 8480+ with 56 cores running at 2 GHz of base frequency, for a total of 112 cores and 256 GB of DDR5 memory. The nodes are interconnected through a 100 Gbit/s ConnectX-7 NDR200 InfiniBand network.

Regarding the software stack, the executions rely on the GCC-10.4 compiler, the MPICH-3.2 MPI implementation, and a customized version of Slurm to support malleability included in the dynamic resource management framework DMRlib. Slurm is deployed using a node running its controller daemon, while the remaining nodes act as compute nodes. Slurm is configured with `sched/backfill` and `priority/multifactor` policies.

To perform these experiments, we have enabled the Slurm job epilog mechanism to communicate job completions to the job submitter UBS described in Section 3.1. Furthermore, we have leveraged a Slurm resource selection plugin that supports malleability [32]. In this regard, every time a malleable job triggers a reconfiguration, Slurm will determine the action that the runtime must perform, which may be *expand*, *shrink*, or *none*. This Slurm plugin is based on `select/linear`, and its algorithm is explained in Section 4.4.

The experiments are scoped within 125 nodes for up to 48 hours, which are the limits of our available submission queue. Notice that a node from the allocation acts as the controller and the remaining 124 as actual compute nodes.

4.2. Traditional Users: The Baseline Workload

The baseline workload is adapted from the most recent available workload on the Parallel Workload Archive KIT-FH2-2016[6] log, recorded from the ForHLR II system located at the Karlsruhe Institute of Technology in Germany. Notice that this is the most recent workload publicly available and it is still a representative example of a general-purpose computational partition, unaffected by interference from deep learning jobs, better suited for accelerated partitions. To adapt the log to our testbed, we perform the following steps:

1. The original infrastructure has two queues: the default queue with 1,152 20-core compute nodes, and the visualization queue with 21 nodes comprising CPUs and GPUs. We focus on the first queue and exclude the GPU-enabled.

2. The record is 1.5 years long, and we only have two days of computation available in our testbed. Consequently, we decide to speed up the replay by **applying a time-scaling factor of 10**, i.e., execution times and inter-arrival times are divided by 10 in the replay[7]. In this regard, we can replay $2 \times 10 = 20$ days from the original workload.

3. By studying the distribution of job submissions over time, we select as representative the month of July 2017 because it features a homogeneous volume of submissions over days, well-distributed among users, and the presence of the characteristic day/night and weekday/weekend patterns. As a result, we select for the experiments **the 19-day period from July 3^{rd} to July 21^{st} (included), 2017**.

4. Finally, we **apply the user sampling method** described in Section 3.2, with a target in average node-hour per day of $.84M$, where $M = 124 \times 24$ is the maximum number of node-hour available per day in our testbed.

The resulting workload contains 23 users. The median number of node-hours submitted per day is 2,272, i.e., 76%

[2] https://www.mpich.org

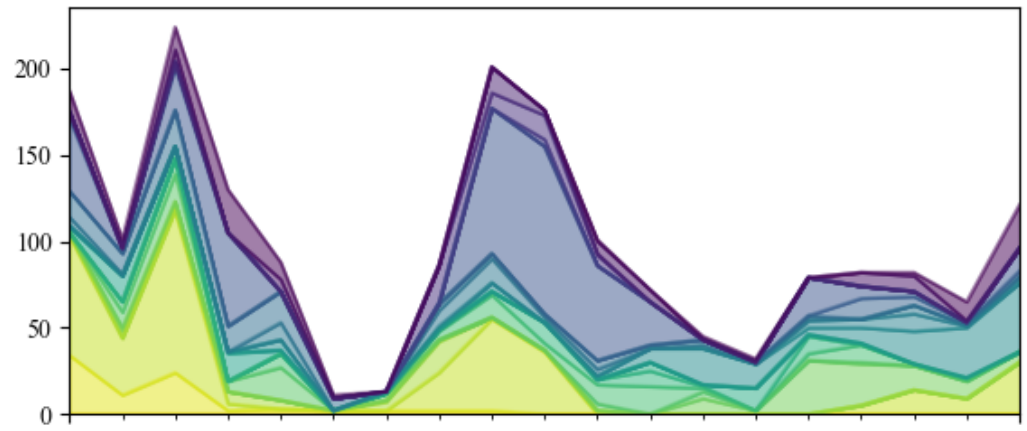
[3] https://slurm.schedmd.com

[4] https://pop-coe.eu/node/69

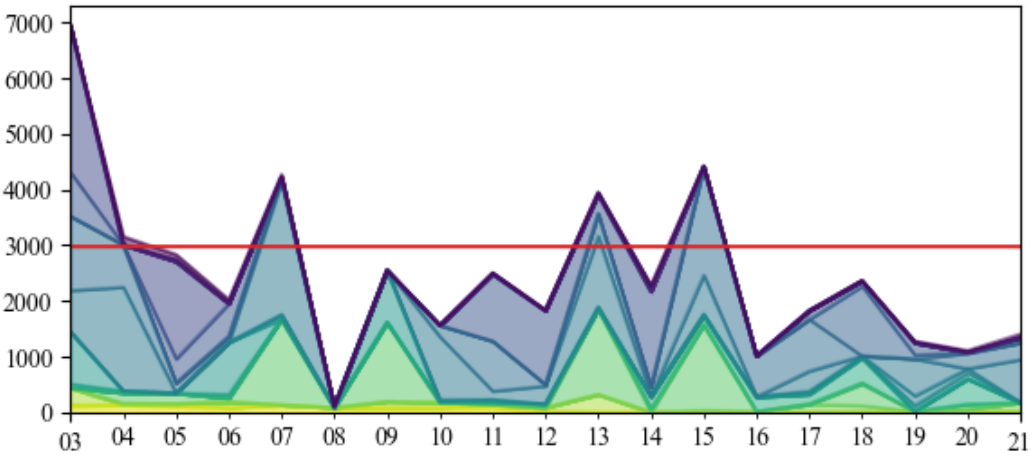
[5] https://www.bsc.es/supportkc/docs/MareNostrum5/overview

[6] https://www.cs.huji.ac.il/labs/parallel/workload/l_kit_fh2/index.html

[7] This factor was chosen to balance the tradeoff between replaying more days from the original workload and suffering from overheads of phenomena that we cannot speed up (like the time taken by scheduling decisions).



(a) Number of submissions per day.



(b) Node-hours submitted per day.

Figure 2: Distribution of job submissions and platform capacity over the days of July 2017 (x-axis).

of the maximum capacity of the testbed. The average number of node-hours submitted per day is 2,496, i.e., 84% of the maximum capacity of the testbed. Some characteristics of this workload are provided in the form of graphs. Figure 2 displays the number of daily submissions and node-hours submitted per day. Each color represents a particular user, and the stacked graphs show the submission volume per day. Figure 3 shows the cumulative proportion of jobs as a function of their duration. It shows that a vast majority lasts less than one hour (slightly more than 60%). Figure 4 shows the distribution of job execution times and job sizes. Jobs are mainly requesting 1, 3, or 4 nodes (see Figure 4a). But it also shows that large jobs (see 26, 30, and 32 nodes in the x-axis of Figure 4b), while rare, represent a significant part of the total executed mass. Since the jobs in the log have a fixed duration, the replayed jobs submitted to the cluster will allocate resources for the same period, either by using `sleep` commands or implementing active waits.

Warm-up Period. Looking at Figure 2b, we note that the first day of the workload, July 3rd, features high activity in node-hours submitted. This day, Monday, comes after a weekend of low activity on the platform. However, it is not until 11:00 AM that the platform runs in nominal conditions. In other words, at that time, resources are virtually fully allocated, and there are pending jobs waiting for resources. We thus define this period as a "warm-up" in the experiments, and malleable jobs start being submitted just after it (t_0 input of the *generative user* described in Section 3.1).

4.3. A Generative User: The PhD Student

To thoroughly evaluate our methodology, we count on an additional generative user to submit jobs on top of the baseline workload. "The student" is working with a simulation tool, particularly, a positive definite transport equation solver named MPDATA[8]. Its algorithm performs iterative time steps to simulate physical phenomena, requiring five input arrays and producing a single output array essential for subsequent time steps. The malleable version of MPDATA[9] redistributes the input arrays between source and target processes, respectively, the number of processes before and after a reconfiguration. MPDATA malleable demonstrated

[8]The MPDATA algorithm serves as the foundation of the EULAG multiscale fluid solver (Eulerian/semi-Lagrangian) [36], which is responsible for calculating the advection of a non-diffusive quantity in a flow field.

[9]Code available at `https://gitlab.bsc.es/siserte/mpdata-dmr`.

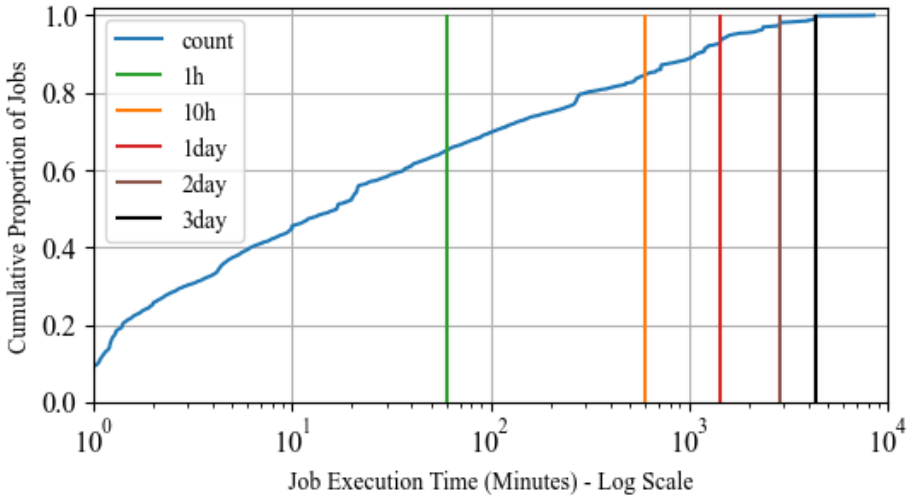
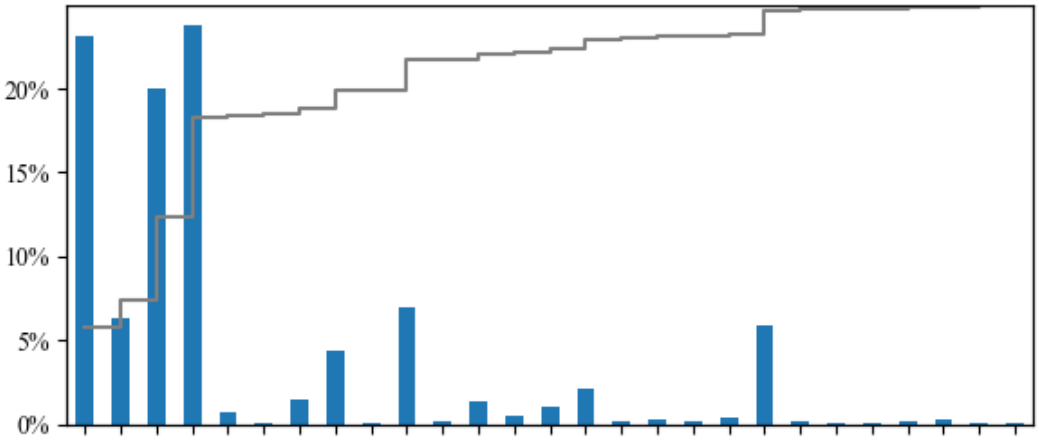
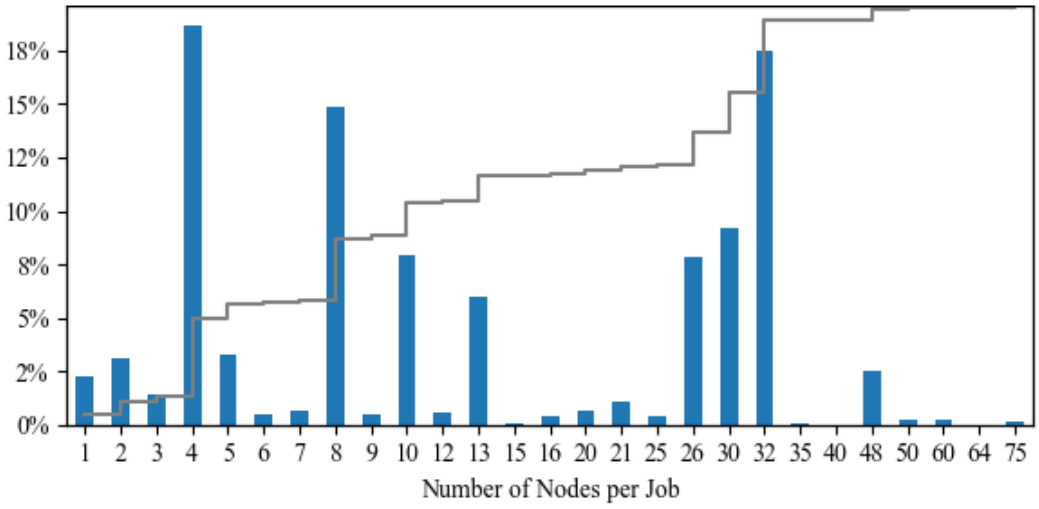


Figure 3: Distribution of job execution time in the baseline workload.



(a) Proportion of jobs.



(b) Proportion of node-hours.

Figure 4: Job sizes in the baseline workload. The distribution is shown by the number of jobs (top) and number of node-hours (bottom), and the cumulative distribution is represented in grey.

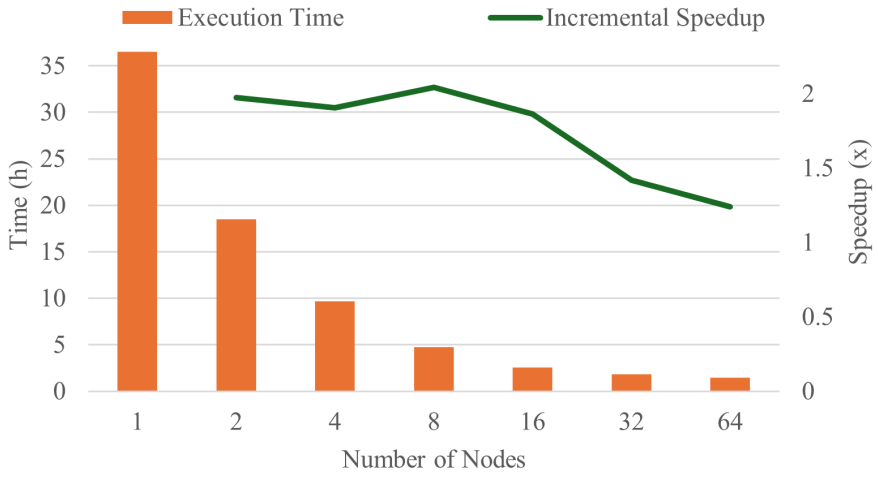


Figure 5: MPDATA scalability in MN5.

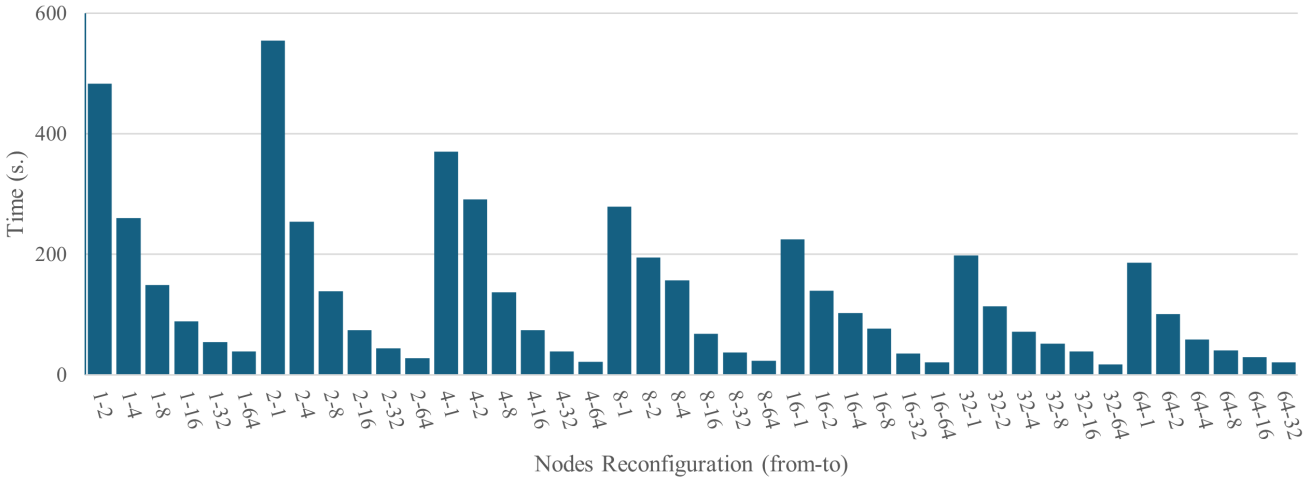


Figure 6: MPDATA reconfiguration times in MN5.

to increase the utilization of resources and reduce the power consumption in dynamic workloads [37].

For her study, the student must run ten instances (N) of MPDATA sequentially. She will start launching jobs after the warm-up period and wait two scaled hours (720 actual seconds) before submitting the next job. This think time (Δt) corresponds to a hypothetical analysis of the results and the preparation of the subsequent execution.

The student is requested to complete her workload in 14 days; in other words, she has a deadline of two weeks to present her results. Since she is in a hurry, an alarm is triggered on her phone when a job finishes, and she immediately starts processing data and setting up the next job.

MPDATA is configured with a computational domain of $8,192 \times 1,024 \times 128$ cells, iterating during 1,800 steps. This computational domain enables the user to run MPDATA from 1 to 64 nodes. Figure 5 showcases the scalability of MPDATA with the given configuration in MN5. The application scales up to 16 nodes linearly, from where it shows speedups lower than $2x$.

4.4. Experiments Definition

Five experiments have been designed to study and analyze the effect of job malleability and dynamic management of resources over an existing workload. Each experiment was launched on an independent 48-hour reservation of the 125-node testbed (see Section 4.1). With these experiments, we will illustrate scenarios that represent different user behaviors:

1. **Baseline**: this experiment corresponds to executing the baseline workload described in Section 4.2. This experiment sets the original activity of the workload execution in the cluster.

2. **Static32**: this experiment executes sequentially over the baseline workload, the 10-job workload of the PhD student launched with 32 nodes each. This is a realistic case, since with 32 nodes the execution time is $\approx$ 14 scaled hours

Algorithm 2 Reconfiguration Policy Algorithm in Slurm

```
1:  result ← NONE
2:  η ← EvaluateMyParallelEfficiency()
3:  if JobCanBeInitiatedWithPartOfMyResources() then
4:     if η < 0.85 then
5:        SetMaximumPriorityToTargetPendingJob()
6:        result ← SHRINK
7:     end if
8:  else
9:     R ← ThereAreAvailableResources()
10:    if R ≠ ∅ then
11:       if η > 0.10 then
12:          result ← EXPAND
13:       end if
14:    end if
15: end if
16: return result
```

(10x the actual time in Figure 5). It means that she could finish her executions in less than six scaled days in an ideal scenario where jobs were not delayed.

3. **Static16**: this is another non-malleable experiment similar to *Static32* but launching the jobs with 16 nodes, instead. In this case, the student works with the shortest configuration time, the one within the 2x speedup in Figure 5, which lasts $\approx$ 18.5 scaled hours. Theoretically, she could complete her executions in less than eight scaled days. We have discarded to configure a hypothetical *Static8* experiment, since the theoretical time required to run the 10-job workload with eight nodes under ideal conditions is 47.5 scaled hours, while the maximum wall time granted for the experiment is 48 hours.

4. **AlwaysGrow**: in this experiment, the PhD student instantiates malleable jobs over the baseline workload. Jobs are submitted moldable, requesting a 1–64 range of nodes. Since she is under a tight deadline, she decides to disable shrinkages; jobs may only be expanded up to 64 nodes upon the RMS decision. By default, the reconfiguration policy will plan an expansion when sufficient resources are available to upscale. Furthermore, the student has defined a reconfiguration inhibitor of `#current_nodes` iterations to avoid abuse of reconfiguration operations and reduce overhead generated by them (see Figure 6).

5. **ParEfficiency**: in this case, jobs are submitted within the 2–64 nodes range. Malleability limits are defined as one node at the minimum and 64 at the maximum. However, the student configures an additional inhibitor for reconfigurations longer than 50 scaled minutes. According to Figure 6, which depicts the measured reconfiguration time for all the possible node combinations of *from–to* reconfigurations, those greater than five minutes are avoided, specifically the reconfigurations *1 to 2*, *2 to 1*, and *4 to 1*. This experiment leverages the performance-aware support of the dynamic resource manager (see Section 3.3). Algorithm 2 depicts the reconfiguration policy. Every time a malleable job triggers a reconfiguration, Slurm will determine the action that the DMRlib runtime must perform. In this regard, Slurm's plugin will check if a pending job may be initiated with some of the resources that would be available if the malleable job relinquishes them after its shrinkage (line 3). If a target job may be initiated, the policy checks the current parallel efficiency of the malleable job. If the parallel efficiency does not reach a minimum threshold (line 4), the target job priority in the queue will be increased (line 5), and the malleable job will be shrunk (line 6). If no job may be initiated and there are available resources in the cluster (line 10), the policy checks the current parallel efficiency. If the value exceeds a determined threshold (line 11), indicating that the execution may still leverage additional resources, the malleable job will be expanded (line 12).

This policy aims to obtain as many resources as possible if no other job in the queue may use them. However, suppose any job in the queue may be initiated, and the malleable job is not reasonably using the resources (within a parallel efficiency threshold). In that case, the RMS reassigns the resources to a pending job in order to increase global cluster productivity and efficiency.

5. Results

In this section, the results obtained after running the experiments are analyzed. We use the following metrics for the validation of MPI malleability: resource allocation rate (Section 5.1), makespan (Section 5.2), and waiting time (Section 5.3)[10].

Note that the metrics have different interpretations depending on the type of workload replay, as well-explained by

[10] Since the execution time is fixed and determined by the log, and the completion time is the sum of waiting plus execution times, the completion time is entirely dependent on the waiting time.

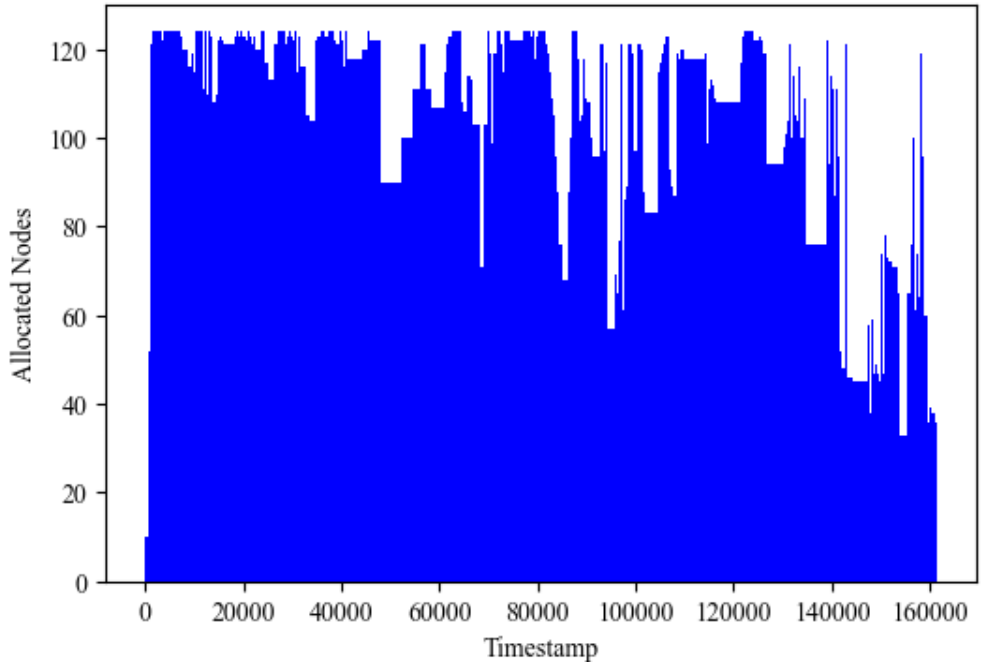


Figure 7: Resource allocation (Y-axis) in each second of the execution (X-axis) for the *Baseline* experiment.

Feitelson [22]. In the case of replay with feedback (like here with the PhD student workload), the primary performance metric is the makespan, i.e., the total time to execute the workload. On the contrary, the makespan in timestamp-based replay (like here with the baseline workload) is dictated by the original submission timestamps. In this case, the primary performance metric is the waiting times for each job.

5.1. Resource Allocation Rate

Figure 7 showcases the total allocated nodes for every timestamp in the *Baseline* experiment. Correspondingly, figures 8a, 8b, 9a, and 9b showcase stacked bar plots where each timestamp depicts the sum of the allocated resources by the baseline plus the PhD student workloads. The figures present different patterns: while the static experiments in figures 8a and 8b show how the student jobs have to wait for slots that satisfy the static requests, the malleable experiments in figures 9a and 9b leverage fragmentation where student jobs run.

Figure 10 gives the allocation rate for each experiment, which is the average node utilization over the makespan of the experiment, excluding the warm-up period. We observe that overlapping the PhD student workload to the baseline supposes an increase of around 7–10% of the resource allocation. The figure shows that malleable experiments present a similar allocation rate to the static scenarios. This means that MPI malleability can maintain resource utilization while reducing the makespan of the PhD workload, as we will see in the following analyses.

5.2. Makespan

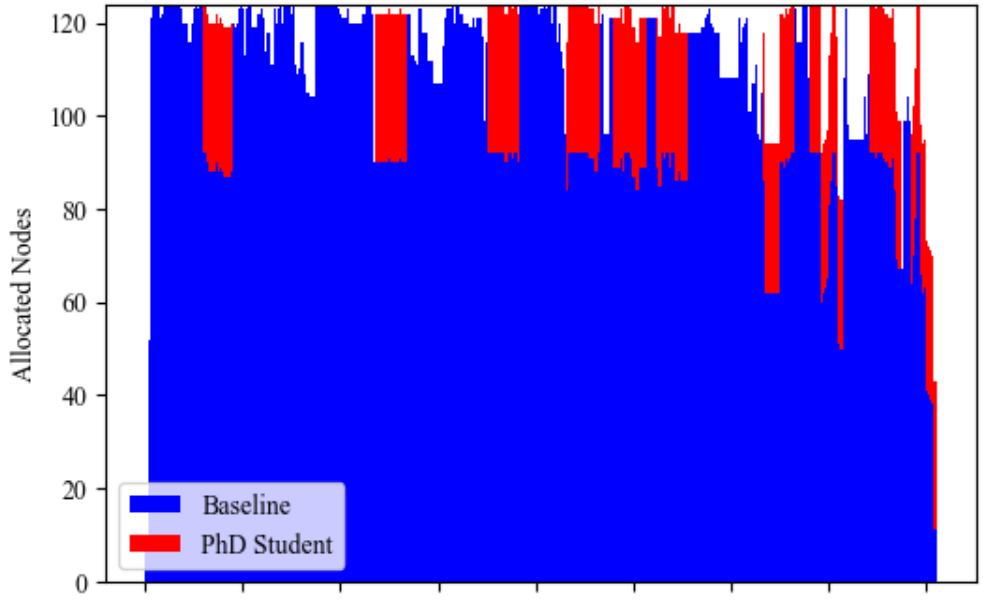
Figure 11 displays the makespan for the different experiments. The right column is the makespan for the PhD workload only, whereas the left column is for both PhD and baseline workload. As a result, the left column also corresponds to the total experiment duration.

As discussed in the introduction of this section, the left column is not meaningful, since makespan is dictated by the rigid submission timestamps replayed from the original log. This is why all the values in this column are similar to the *Baseline* scenario. The only exception is *StaticN32*, where the makespan is significantly increased compared to *Baseline*. *StaticN32*, as depicted in Figure 12, takes 6.27 scaled hours more time than the baseline to finish due to a PhD job still running at the end of the experiment.

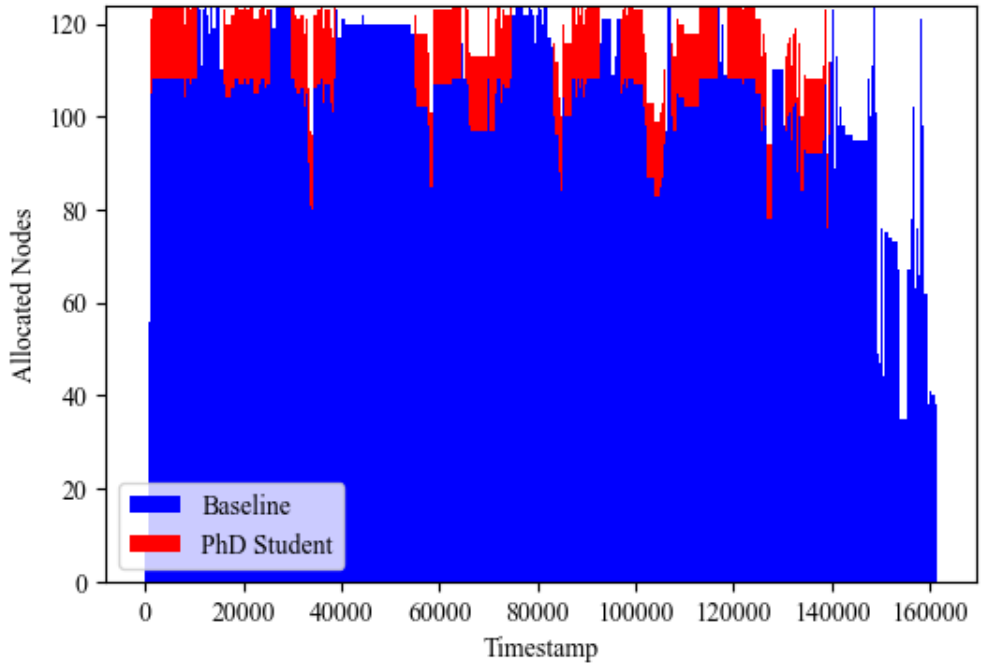
More interesting is the right column of Figure 11, representing the time the student needs to complete her workload. The student workload in *StaticN32* is the longest because the scheduler cannot easily find a spot to fit the job and delays those large jobs. Compared to the static, we appreciate a non-negligible time reduction from part of the malleable workloads. Notably, the "smartest" dynamic experiment, *ParEfficiency*, needs $\approx 73\%$ the time to complete the student jobs compared to *StaticN32*.

5.3. Waiting time

To analyze how baseline jobs are affected by the submission of extra jobs from the PhD student, we look at their mean waiting times in the different scenarios (Figure 12). It is patent that the original workload suffers longer delays on average when the PhD student jobs are submitted: the mean waiting time increases from 1,725s to 3,007s or 4,365s in the best and worst cases, respectively. Besides, the two dynamic



(a) *StaticN32*

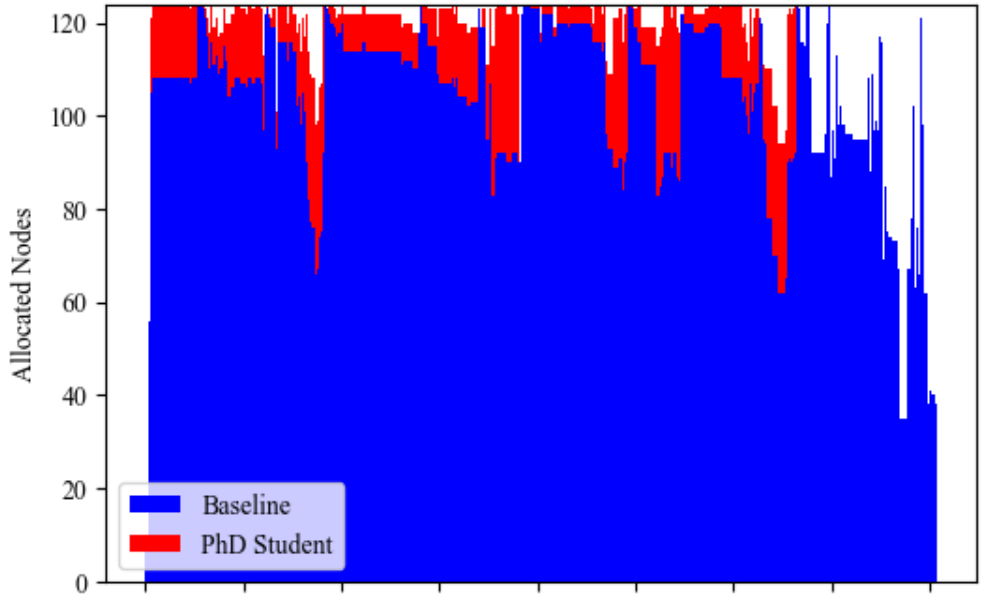


(b) *StaticN16*

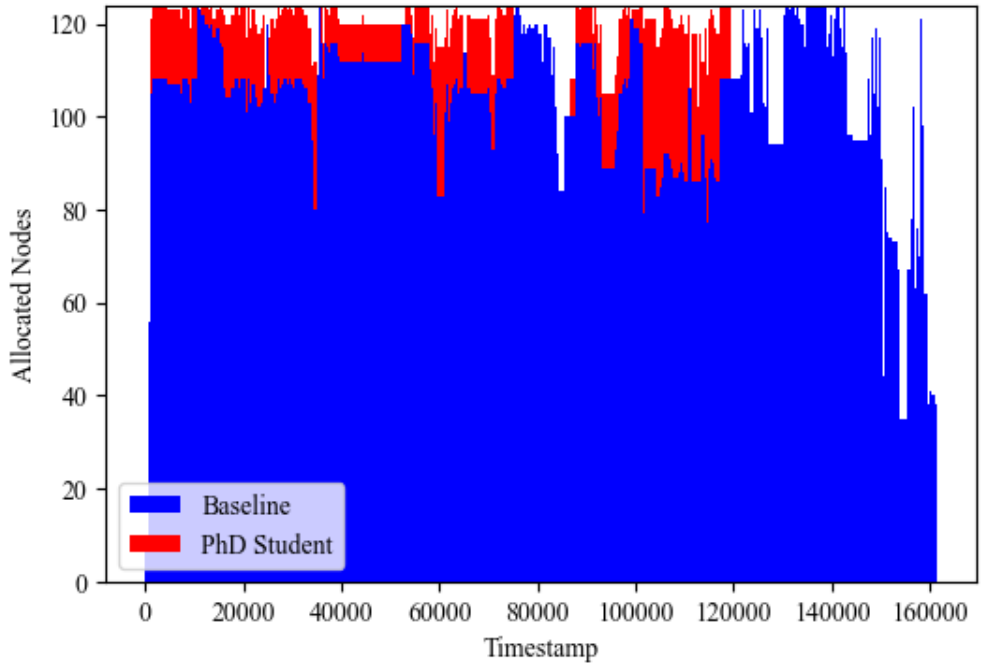
Figure 8: Resource allocation (Y-axis) over execution time (X-axis) for the *student* `static` experiments. 32-node allocations are harder to fit, increasing makespan.

experiments (*AlwaysGrow* and *ParEfficiency*) pose a larger impact on the baseline waiting time. This can be explained by the greedy nature of the malleability configurations, starting with fewer resources and expanding as resources are released in the platform, making the baseline jobs wait longer.

In a finer detailed analysis, we studied the accumulated waiting time throughout the execution (Figure 13). It shows how the waiting time increases as new jobs arrive in the queue, and since there are not enough available resources until around timestamp 77,000 (July 10^{th}), they cannot be quickly started. After this milestone, the waiting time progressively starts growing again. More specifically, Figure 14 showcases the waiting time difference of the various experiments compared to the Baseline for each job. One of the most valuable insights we extract is the behavior detected around job 750 (July 10^{th}), where waiting times drop. This is because there is a period around July 8^{th} and 9^{th}, as shown



(a) *AlwaysGrow*



(b) *ParEfficiency*

Figure 9: Resource allocation (Y-axis) over execution time (X-axis) for the *student* `dynamic` experiments. Dynamic policies, especially *ParEfficiency*, reduce makespan.

in Figure 2a, where the submissions drastically decrease, and it takes over a day to drain the queue. That is why jobs submitted after that event present a virtually null waiting time.

5.4. Focus on the PhD Student

Figure 15 illustrates the mean completion time of the student's jobs. The completion time is composed of the sum of the waiting time and the execution time. The malleable experiments present a higher execution time compared to their static counterpart. This is because malleable jobs are likely to be initiated with fewer resources. Then, dynamic jobs are progressively expanded following their respective reconfiguration policies. The more resources assigned to the jobs, the lower the execution time (see Figure 5). However, reconfigurations come with a cost (see Figure 6), and this cost is included in the execution time.

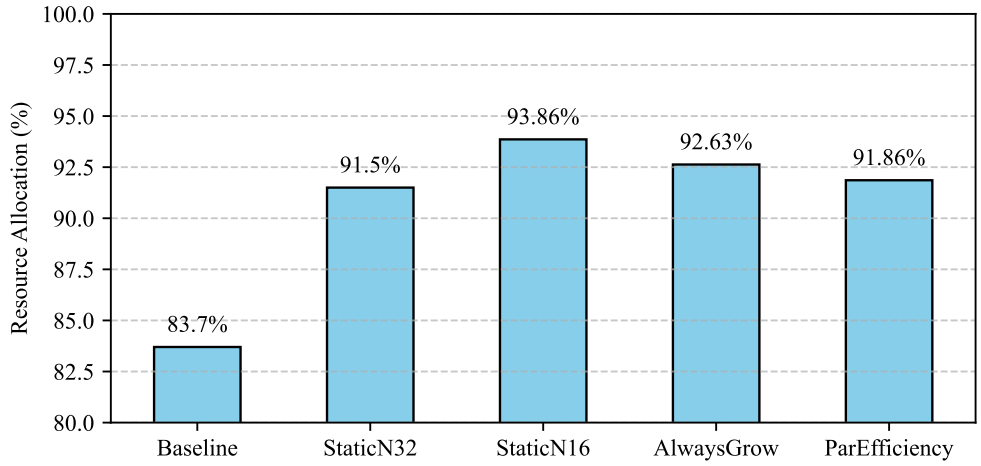


Figure 10: Average percentage of allocated resources (Y-axis) for the five experiments (X-axis). *ParEfficiency* has a more balanced resource allocation rate while reducing the makespan.

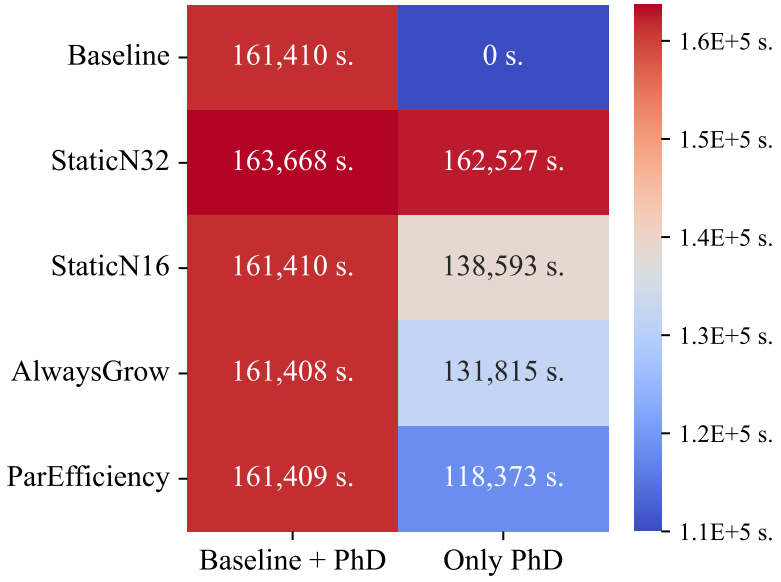


Figure 11: Heatmap representing the makespan (s.) of experiments (Y-axis) and workloads (X-axis). *ParEfficiency* completes the *student* workload in 73% of the *StaticN32* time.

The total cost of reconfigurations in our case is given in Table 1. In *ParEfficiency*, the overhead is reduced thanks to preventing the slowest reconfigurations. Furthermore, since the malleable jobs are adapted to the workload and the performance, there are seven *32 to 16*-node shrinkages to increase productivity by reducing the student workload time by 10% compared to *AlwaysGrow*.

If we pay attention to the node-hours consumed by the PhD Student jobs, Figure 16 shows that metric for each of those jobs in the experiments. Furthermore, a dashed line represents the PhD student jobs mean node-hour per experiment.

Static jobs present a virtually constant node-hour consumption (standard deviation < 1). As shown in Figure 5, the PhD jobs deliver lower speedup running on 32 than in 16 nodes, $1.41x$ and $1.86x$, respectively. For this reason, the consumption is 17 node-hours higher in the *StaticN32* experiment. Regarding MPI malleability, since those jobs can be resized, they show variable node hours within the experiments (standard deviation > 4). Remarkably, *AlwaysGrow* and *ParEfficiency* contain the two "cheapest" jobs because they run most, or the whole of their time, respectively, in 8 nodes. It is with eight nodes that the maximum performance is attained.

6. Discussion

Our methodology adapts historical workload traces to a target infrastructure, accounting for the fact that reproducing logs on the same system is often not feasible. The goal is to provide a realistic path for gradual DRM adoption in HPC centers.

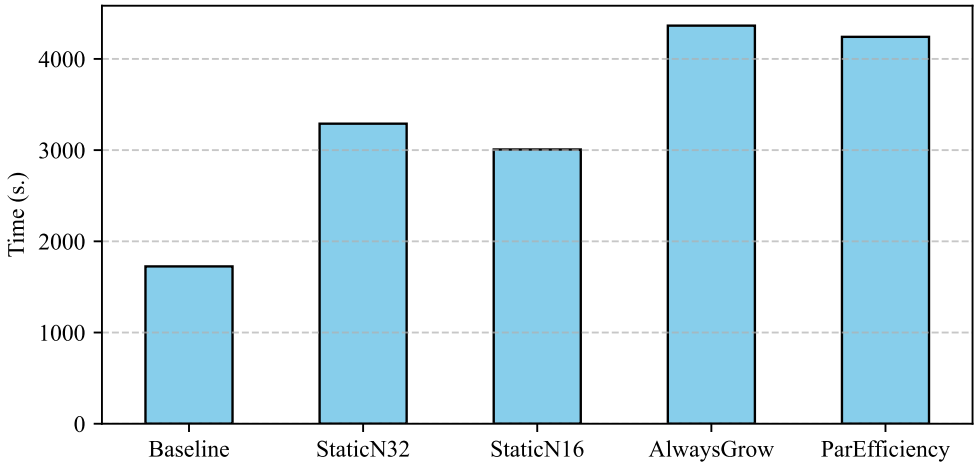


Figure 12: Average waiting time of the baseline jobs (Y-axis) for the five experiments (X-axis). `Dynamic` experiments need more time to fit queued jobs in favor of the makespan.

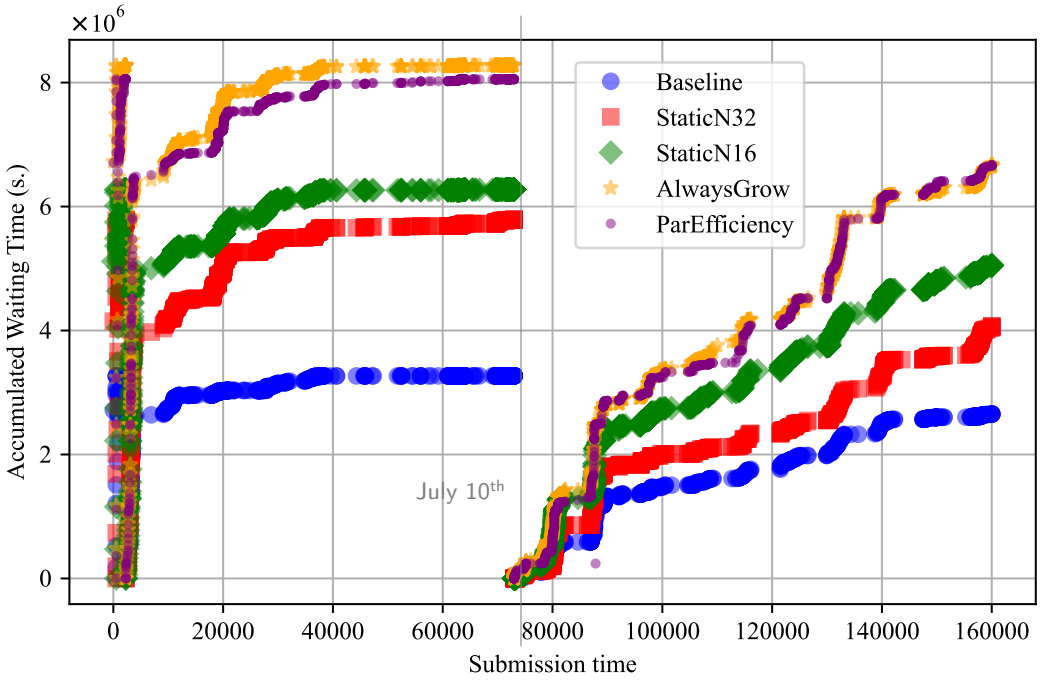


Figure 13: Accumulated waiting time throughout the workloads executions.

We do not expect full adoption of DRM techniques to happen immediately. Instead, early adopters (technically skilled users motivated to exploit malleability) would be the first to submit DRM-enabled jobs to production environments still dominated by fixed-size workloads. This hybrid workload composition enables us to assess the incremental benefits of malleability without needing full-scale user migration.

This setup also offers valuable insight into how malleable jobs can coexist with standard jobs and make use of fragmented resources that would otherwise go idle. Running our experiments on a real Slurm deployment allowed us to combine malleability with Slurm's native backfilling policy, improving resource utilization.

While the number of test scenarios is limited by the need for live system execution, our methodology produces high-fidelity results. Moreover, it can be reused by organizations with their own workload traces, enabling site-specific evaluation of malleability's practical benefits and deployment challenges.

We argue that our method is more robust than many alternatives commonly used in the literature (see Section 2). We (i) use a real workload trace rather than synthetic jobs, and (ii) right-size the workload by randomly sampling users, which preserves authentic submission and temporal patterns.

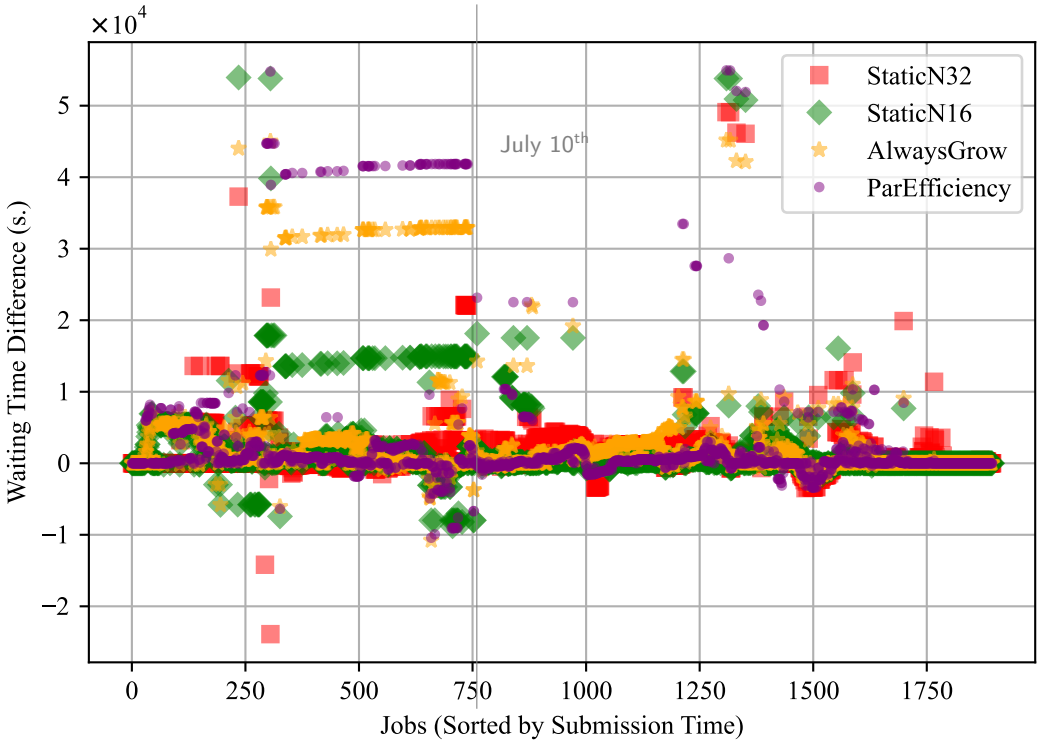


Figure 14: Difference in job waiting times compared to the baseline experiment.

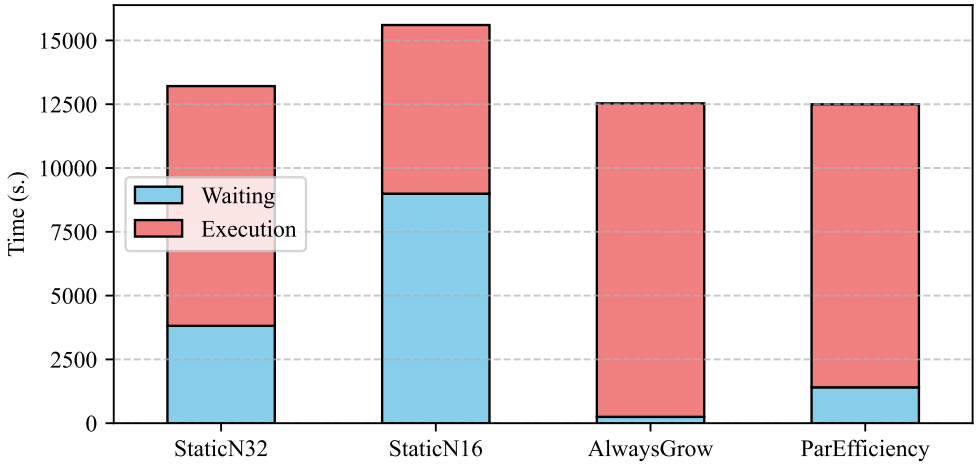


Figure 15: Mean student's job completion (waiting + execution) time (Y-axis) for the four experiments (X-axis). *Student* jobs barely wait in the `Dynamic` experiments.

The 84% baseline utilization target reflects highly loaded workloads in the Parallel Workload Archive, representative of real production environments. We further increased total utilization to 90% when adding the PhD workload, which remains realistic for many production clusters.

At the end of the day, the PhD student successfully meets the deadline in the *ParEfficiency* scenario. Counter-intuitively, large fixed resource requests, which users might assume to be beneficial, actually work against minimizing completion time. Additionally, overly aggressive malleable strategies can introduce excessive reconfiguration overhead, limiting their ability to efficiently leverage resource fragmentation.

Fragmented resource availability, a common characteristic of HPC infrastructures, represents the ideal use case where malleability becomes crucial. This fragmentation is what our student leverages. Table 2 focuses on the time period during which the student is actively running jobs (corresponding to the red areas in figures 8a, 8b, 9a, and 9b). The table presents the resource consumption in *nodes/day* for the workloads, along with their proportion of the total available area (124 nodes × time) shown in the first row.

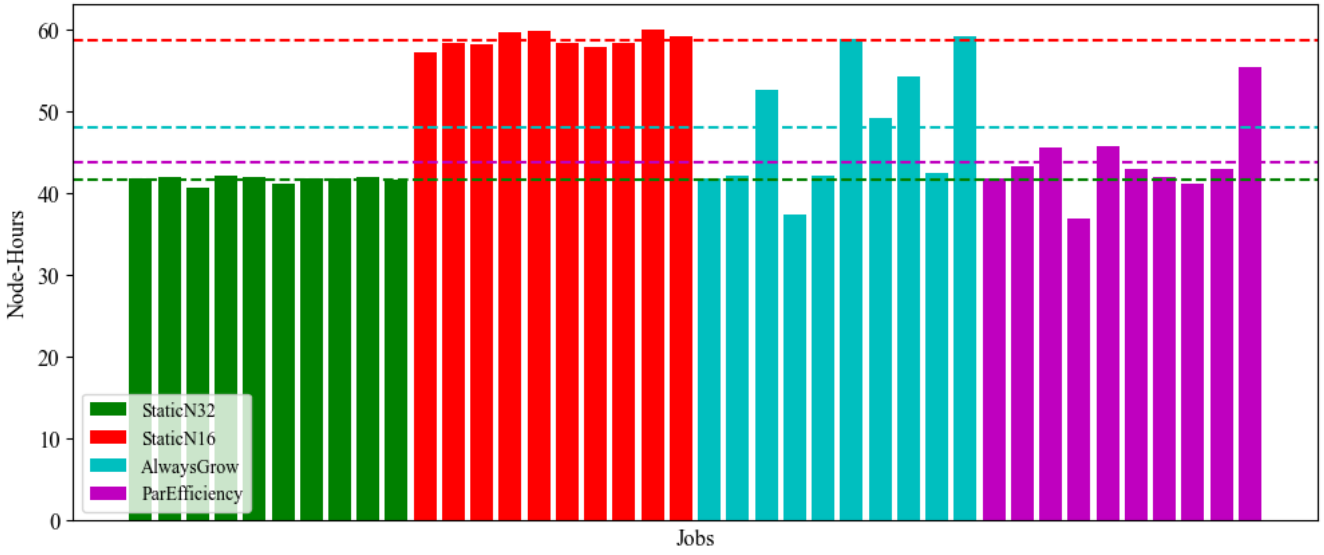


Figure 16: Individual student job's Node-hours (Y-axis) grouped by colors for the four experiments (X-axis). Dashed lines represent the mean per experiment (Y-axis).

	#	Mean Time	Overhead
AlwaysGrow	27+0	160 s.	4,332 s.
ParEfficiency	18+7	77 s.	1,929 s.

Table 1
Per experiment (first column) we find the number of reconfigurations in format *expansions + shrinkages* (second column), reconfiguration mean time (third column), and reconfiguration overhead (fourth column).

The data highlights an interesting trade-off. Although the most cost-effective strategy for the student is *StaticN16* (third row), it does not only allow meeting the deadline but also results in increased costs for the baseline workload (second row) and higher overall cluster resource consumption during that period (fourth row), compared to the optimal *ParEfficiency*.

Given these findings, users who contribute to higher infrastructure efficiency by running *parallel-efficiency-aware malleable jobs*—which increase resource utilization without extending job execution times—could be incentivized. One possible approach would be to award a reduction in their usage quota proportional to their efficiency gains, encouraging broader adoption of malleability in HPC environments.

7. Conclusion

In this work, we have introduced a novel methodology to validate dynamic resource management techniques by adapting supercomputer logs to DRM-enabled computing clusters. Unlike previous studies that relied on benchmarks or simulations with synthetic workloads, we employ a novel methodology called *User-Based Submitter* that combines

	StaticN32	StaticN16	AlwaysGrow	ParEfficiency
Total	232.83 n/day (100%)	200.54 n/day (100%)	189.09 n/day (100%)	171.52 n/day (100%)
Baseline	193.5 n/day (83.11%)	173.7 n/day (86.62%)	164.2 n/day (86.84%)	145.68 n/day (84.93%)
Student	23.95 n/day (10.29%)	17.4 n/day (8.68%)	20.01 n/day (10.58%)	18.25 n/day (10.64%)
Accumulated	217.45 n/day (93.39%)	191.1 n/day (95.29%)	184.21 n/day (97.42%)	163.93 n/day (95.57%)

Table 2
Resource consumption (nodes/day) and its total percentage during the student's activity for the different experiments.

user sampling and workload replay with feedback. By leveraging real-world data and testing on an MPI malleability-enabled supercomputer's partition, we provide a realistic and actionable framework for evaluating new resource management and job scheduling policies.

We have demonstrated its effectiveness through a novel use case in which an HPC user pioneered the use of MPI malleability to complete her workload on time. Our validation confirms the benefits of malleability in a supercomputer environment, showing that the malleable workload time was reduced by 27% without delaying the baseline workload while increasing overall resource utilization by 8%. We believe that our methodology can be broadly applied and represents a promising step toward promoting DRM mechanisms in next-generation HPC infrastructures.

CRediT authorship contribution statement

Sergio Iserte: Conceptualization, Methodology, Software, Investigation, Writing - Original Draft, Visualization, Project administration. **Maël Madon:** Methodology, Investigation, Software, Writing - Original Draft, Visualization. **Georges Da Costa:** Investigation, Writing - Review & Editing, Supervision. **Jean-Marc Pierson:** Investigation, Writing - Review & Editing, Supervision. **Antonio J. Peña:** Resources, Writing - Review & Editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.

Funding sources

The researchers from BSC are involved in the project The European PILOT, which has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreements No. 101034126 and No. PCI2021-122090-2A under the MCIN/AEI and the EU NextGenerationEU/PRTR. They are also grateful for the support from the Department of Research and Universities of the Government of Catalonia to the AccMem (Code: 2021 SGR 00807). Antonio J. Peña was partially supported by the Ramón y Cajal fellowship RYC2020-030054-I funded by MCIN/AEI/ 10.13039/501100011033 and by "ESF Investing in your future".

Data availability

The artifact source-code can be cloned from GitHub: https://github.com/siserte/dmr-poc.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to improve language and readability. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

References

[1] H. Martínez, J. Tárraga, I. Medina, S. Barrachina, M. Castillo, J. Dopazo, E. S. Quintana-Ortí, A dynamic pipeline for RNA sequencing on multicore processors, in: Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 235–240. doi:10.1145/2488551.2488581.

[2] H. Zhong, X. Pan, Z. He, H. Wang, D. Huang, Z. Chen, GPU acceleration for DNA sequence alignment algorithm and its application, CCF Transactions on High Performance Computing 7 (2) (2025) 169–177. doi:10.1007/s42514-024-00203-0.

[3] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation, Journal of Chemical Theory and Computation 4 (3) (2008) 435–447. doi:10.1021/ct700301q.

[4] M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, E. D. Burness, J. M. Cela, M. Valero, Alya: Multiphysics engineering simulation toward exascale, Journal of Computational Science 14 (2016) 15–27. doi:10.1016/j.jocs.2015.12.007.

[5] D. Caviedes-Voullième, M. Morales-Hernández, M. R. Norman, I. Özgen Xian, SERGHEI (SERGHEI-SWE) v1.0: a performance-portable high-performance parallel-computing shallow-water solver for hydrology and environmental hydraulics, Geoscientific Model Development 16 (3) (2023) 977–1008. doi:10.5194/gmd-16-977-2023.

[6] R. Martínez-Cuenca, J. Luis-Gómez, S. Iserte, S. Chiva, On the Use of Deep Learning and Computational Fluid Dynamics for the Estimation of Uniform Momentum Source Components of Propellers, iScience 26 (2023) 1–14. doi:10.1016/j.isci.2023.108297.

[7] P. Rosciszewski, A. Krzywaniak, S. Iserte, K. Rojek, P. Gepner, Optimizing Throughput of Seq2Seq Model Training on the IPU Platform for AI-accelerated CFD Simulations, Future Generation Computer Systems 143 (2023) 149–162. doi:10.1016/j.future.2023.05.004.

[8] W. F. Godoy, P. Valero-Lara, K. Teranishi, P. Balaprakash, J. S. Vetter, Large Language Model Evaluation for High-Performance Computing Software Development, Concurrency and Computation: Practice and Experience 36 (26) (2024) e8269. doi:10.1002/cpe.8269.

[9] H.-J. Bungartz, C. Riesinger, M. Schreiber, G. Snelting, A. Zwinkau, Invasive computing in HPC with X10, in: Proceedings of the third ACM SIGPLAN X10 Workshop, X10 '13, ACM, New York, NY, USA, 2013, pp. 12–19. doi:10.1145/2481268.2481274.

[10] M. Garcia, J. Labarta, J. Corbalan, Hints to improve automatic load balancing with LeWI for hybrid applications, Journal of Parallel and Distributed Computing 74 (9) (2014) 2781–2794. doi:10.1016/j.jpdc.2014.05.004.

[11] V. Lopez, J. Criado, R. Peñacoba, R. Ferrer, X. Teruel, M. Garcia-Gasulla, An OpenMP Free Agent Threads Implementation, in: OpenMP: Enabling Massive Node-Level Parallelism: 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14–16, 2021, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2021, pp. 211–225. doi:10.1007/978-3-030-85262-7_15.

[12] G. Martín, M.-C. Marinescu, D. E. Singh, J. Carretero, FLEX-MPI: an MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems, in: Euro-Par Parallel Processing, 2013, pp. 138–149.

[13] I. Martín-Álvarez, J. I. Aliaga, M. Castillo, S. Iserte, Proteo: a framework for the generation and evaluation of malleable MPI applications, The Journal of Supercomputing (Jul. 2024). doi:10.1007/s11227-024-06277-5.

[14] D. Huber, S. Iserte, M. Schreiber, A. J. Peña, M. Schulz, Bridging the Gap Between Genericity and Programmability of Dynamic Resources in HPC, in: ISC High Performance 2025 Research Paper Proceedings (40th International Conference), 2025, pp. 1–11.

[15] S. Iserte, I. Martín-Álvarez, K. Rojek, J. I. Aliaga, M. Castillo, W. Folwarska, A. J. Peña, Resource optimization with MPI process malleability for dynamic workloads in HPC clusters, Future Generation Computer Systems (2025) 107949doi:10.1016/j.future.2025.107949.

[16] S. Iserte, R. Mayo, E. S. Quintana-Ortí, V. Beltran, A. J. Peña, DMR API: Improving Cluster Productivity by Turning Applications into Malleable, Parallel Computing 78 (2018) 54–66. doi:10.1016/j.parco.2018.07.006.

[17] N. Zakay, D. G. Feitelson, Preserving User Behavior Characteristics in Trace-Based Simulation of Parallel Job Scheduling, in: 2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems, 2014, pp. 51–60. doi:10.1109/MASCOTS.2014.15.

[18] S. Schlagkamp, Influence of Dynamic Think Times on Parallel Job Scheduler Performances in Generative Simulations, in: 3rd conference on Networked Systems Design & Implementation, 2017. doi:10.1007/978-3-319-61756-5_7.

[19] B. Schroeder, A. Wierman, M. Harchol-Balter, Open versus closed: a cautionary tale, in: Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3, NSDI'06, USENIX Association, USA, 2006, p. 18.

[20] N. Zakay, D. G. Feitelson, On Identifying User Session Boundaries in Parallel Workload Logs, in: Workshop on Job Scheduling Strategies for Parallel Processing, 2013. doi:10.1007/978-3-642-35867-8_12.

[21] M. Madon, G. Da Costa, J.-M. Pierson, Replay with Feedback: : How does the performance of HPC system impact user submission behavior?, Future Generation Computer Systems 155 (C) (2024) 66–79. doi:10.1016/j.future.2024.01.024.

[22] D. G. Feitelson, Resampling with Feedback: A New Paradigm of Using Workload Data for Performance Evaluation, in: Workshop on Job Scheduling Strategies for Parallel Processing, 2021. doi:10.1007/978-3-030-88224-2_1.

[23] D. G. Feitelson, Packing Schemes for Gang Scheduling, in: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, IPPS '96, Springer-Verlag, Berlin, Heidelberg, 1996, pp. 89–110.

[24] U. Lublin, D. G. Feitelson, The workload on parallel supercomputers: modeling the characteristics of rigid jobs, Journal of Parallel and Distributed Computing 63 (11) (2003) 1105–1122. doi:10.1016/S0743-7315(03)00108-4.

[25] J. I. Aliaga, M. Castillo, S. Iserte, I. Martín-Álvarez, R. Mayo, A Survey on Malleability Solutions for High-Performance Distributed Computing, Applied Science 12 (2022) 1–32. doi:10.3390/app12105231.

[26] A. Tarraf, M. Schreiber, A. Cascajo, J.-B. Besnard, M.-A. Vef, D. Huber, S. Happ, A. Brinkmann, D. E. Singh, H.-C. Hoppe, A. Miranda, A. J. Peña, R. Machado, M. G. Gasulla, M. Schulz, P. Carpenter, S. Pickartz, T. Rotaru, S. Iserte, V. Lopez, J. Ejarque, H. Sirwani, F. Wolf, Malleability in Modern HPC Systems: Current Experiences, Challenges, and Future Opportunities, IEEE Transactions on Parallel and Distributed Systems (2024) 1–14doi:10.1109/TPDS.2024.3406764.

[27] R. Sudarsan, C. J. Ribbens, ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment, in: 2007 International Conference on Parallel Processing (ICPP 2007), 2007, pp. 44–44. doi:10.1109/ICPP.2007.73.

[28] R. Sudarsan, C. J. Ribbens, D. Farkas, Dynamic Resizing of Parallel Scientific Simulations: A Case Study Using LAMMPS, in: Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 175–184. doi:10.1007/978-3-642-01970-8_18.

[29] O. Sarood, A. Langer, A. Gupta, L. Kale, Maximizing throughput of overprovisioned HPC data centers under a strict power budget, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, IEEE Press, New Orleans, Louisana, 2014, pp. 807–818. doi:10.1109/SC.2014.71.

[30] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, L. V. Kale, A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications, in: Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS '15, IEEE Computer Society, USA, 2015, pp. 429–438. doi:10.1109/IPDPS.2015.34.

[31] S. Iserte, H. Martínez, S. Barrachina, M. Castillo, R. Mayo, A. J. Peña, Dynamic reconfiguration of noniterative scientific applications: A case study with HPG aligner, The International Journal of High Performance Computing Applications 33 (5) (2019) 804–816. doi:10.1177/1094342018802347.

[32] S. Iserte, R. Mayo, E. S. Quintana-Ortí, A. J. Peña, DMRlib: Easy-coding and Efficient Resource Management for Job Malleability, IEEE Transactions on Computers 70 (2020) 1443–1457. doi:10.1109/TC.2020.3022933.

[33] D. G. Feitelson, L. Rudolph, Towards Convergence in Job Schedulers for Parallel Supercomputers, in: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, IPPS '96, Springer-Verlag, Berlin, Heidelberg, 1996, pp. 1–26.

[34] S. Iserte, I. Martín-Álvarez, K. Rojek, J. I. Aliaga, M. Castillo, A. J. Peña, Towards the Democratization and Standardization of Dynamic Resources with MPI Spawning, in: R. Wyrzykowski, J. Dongarra, E. Deelman, K. Karczewski (Eds.), Parallel Processing and Applied Mathematics, Springer Nature Switzerland, Cham, 2025, pp. 287–300. doi:10.1007/978-3-031-85697-6_19.

[35] V. Lopez, G. Ramirez Miranda, M. Garcia-Gasulla, TALP: A Lightweight Tool to Unveil Parallel Efficiency of Large-scale Executions, in: Proceedings of the 2021 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy, PERMAVOST '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 3–10. doi:10.1145/3452412.3462753.

[36] K. Rojek, , R. Wyrzykowski, Parallelization of 3D MPDATA Algorithm Using Many Graphics Processors, in: Proceedings of the 13th International Conference on Parallel Computing Technologies - Volume 9251, Guide Proceedings, 2015, pp. 445–457. doi:10.1007/978-3-319-21909-7_43.

[37] S. Iserte, K. Rojek, A Study of the Effect of Process Malleability in the Energy Efficiency on GPU-based Clusters, Journal of Supercomputing 76 (2020) 255–274. doi:10.1007/s11227-019-03034-x.