Elasticity for HPC Clusters: A Simulation-based Analysis with Real Workloads

Bachelor Thesis

presented to
Department for Electric Engineering/Computer Science
Software Engineering Research Group
University of Kassel

Patrick Zojer
28204891

Examiners:
Dr. Jonas Posner
Prof. Dr. Oliver Hohlfeld

Kassel, April 29, 2025

Statutory Declaration

I declare on oath that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this, nor a similar work, has been published or presented to an examination committee.

Kassel, April 29, 2025
Patrick Zojer

Contents

List of Figures

List of Tables

1 Introduction

Modern *supercomputers*, often referred to as *High-Performance Computing (HPC)* clusters, continue to become more complex, powerful, and essential in today's high-technology world. They stand out due to their large number of processors, making them valuable for complex computing tasks. An HPC cluster consists of multiple *nodes* interconnected within a network. A *job* in an HPC system is a program submitted to a *job scheduler* (e.g., Slurm [1]), along with information such as the time the job will be finished (*timelimit*) or the required number of nodes to run the program (*job size*). A set of jobs is referred to as a *workload*.

The job scheduler assigns the required number of nodes to each job using an algorithm that determines which jobs run on which nodes and when. In traditional HPC systems, jobs are *rigid*, meaning they run on a fixed, user-defined number of nodes for their entire runtime.

A key objective in HPC job scheduling is to achieve a high node utilization from which both supercomputer operators and users benefit, as lower utilization means wasted resources. However, 100% node utilization is rarely achievable. This is because job sizes vary, and the number of available nodes at any given time is unlikely to match the exact requirements of the next job in the queue, or even any job in the queue. As a result, some nodes may remain free despite pending jobs in the queue, extending job wait times—a situation that is undesirable for both operators and users.

Real workload data of HPC clusters such as Theta [2], Cori [3], and Eagle [4] show an average node utilization range between 30% and 80%. These low values are not only caused by job scheduling inefficiencies but also by factors such as empty job queues or system maintenance. The workload data does not include explicit information on node availability at each moment in time, such as system failures, maintenance periods, or reserved resources.

One way to increase node utilization is to improve job scheduling flexibility through the concept of *resource elasticity*. Instead of specifying a fixed number of nodes, elastic jobs define a preferred number of nodes and a node range, allowing the number of assigned nodes to change dynamically during execution. There are multiple types of elastic jobs [5] such as *evolving*, where the job itself can request changes to the number of assigned nodes from the job scheduler. In contrast, for *malleable* jobs the job scheduler can change the number of assigned nodes. A suitable job scheduler for elastic workloads must be able to dynamically *expand* jobs (adding nodes to improve resource utilization) and *shrink* jobs (removing nodes to accommodate new jobs) between computation phases. While “resources” in HPC can refer to various elements such as storage (both local and network) or GPUs, this study focuses exclusively on CPU nodes when using the term.

However, introducing elasticity in HPC clusters is challenging. Besides requiring a scheduler that supports elastic jobs (which few currently do), the submitted applications themselves must be elastic. Not all applications can run efficiently across a range of nodes or dynamically adjust their resource usage. Therefore, adopting elasticity requires modifications on the operator and user sides, making the transition more complex. Also, the most common application interface for parallel programming MPI (Message Passing Interface) does barely support elasticity.

Given these practical hurdles, much elasticity research relies on simulations to explore its potential benefits [6, 7, 8]. This is a reasonable approach considering the complexity and cost of setting up real test environments. A lot of studies uses their own simulation software, but shortly a comprehensive open-source simulator was released—ElastiSim [9]. It is common to use synthetic workloads to study elasticity in simulations, which brings up the question of how significant the results are if the input is not as realistic as possible. Real workload data eliminates the problem entirely, while providing more realistic insights. If real data yields similar results, it would strengthen the conclusions and could enhance acceptance and applicability.

This study analyzes the impact of malleable jobs in HPC clusters regarding multiple metrics (job makespan, job wait time and node utilization). For that, we use the ElastiSim simulation software [10]. To accomplish simulations as close as possible to a real environment, workloads from real HPC clusters (Haswell, KNL, Theta, and Eagle) were analyzed, prepared, and integrated in the simulations. The contained jobs were also converted into malleable jobs to simulate elasticity. Each workload was simulated with malleable job proportions increasing from 0% to 100% in 20% increments. To account for variability in job selection, each workload was simulated using 10 different random seeds, which influence which jobs are converted into malleable jobs.

The EASY backfilling algorithm [11] calculates the estimated start of upcoming jobs with a time limit within the job should be completed and can start jobs with a lower time limit without delaying others. In addition to a rigid EASY backfilling job scheduler, four additional schedulers supporting dynamic node changes of malleable jobs were used. Three of them are already existing scheduling algorithms [6], which all follow a central strategy of shrinking jobs when necessary to start more waiting jobs. The fourth job scheduling algorithm is a contribution of this work: it integrates a new approach by keeping jobs at their preferred number of nodes as long as possible for optimal performance.

This work presents a comparative analysis of the impact of malleable jobs and different job scheduling algorithms. We examine multiple metrics including job wait time, job makespan, job turnaround time, node utilization, and the number of shrinks and expands performed. Our results provide insights into how malleability and different job scheduling algorithms impact the performance of both the overall supercomputer and individual jobs. The results show that malleability consistently reduces job turnaround time, thereby shortening the total makespan. However, the extent of improvement and the impact on other metrics varies depending on the workload and scheduling algorithm used. Depending on the workload characteristics, the decrease in average turnaround times ranges from 36.9% to 66.87% compared to 100% rigid simulations. The increase in average node utilization reaches from 16.23% to 51.94%.

The remaining thesis is structured as follows. In Chapter 2, the background, such as the general structure of HPC clusters and how executions on them work, are described. This is followed by an overview of the simulation environment, workload handling, and experimental design in Chapter 3. Chapter 4 presents the simulation results organized by workload, followed by a cross-workload summary. Chapter 5 presents related work and Chapter 6 is dedicated to further discussion, including limitations and the overall conclusion.

2 Background

This chapter provides an overview of job scheduling in HPC clusters in Section 2.1. Section 2.2 introduces the concept of elasticity and explores its impact on both supercomputer efficiency and workloads.

2.1 Job Scheduling in HPC Systems

HPC clusters consist of multiple independent computation nodes (computers), which may have different hardware configurations. While clusters are often built with homogeneous hardware, they are often gradually upgraded, leading to mixed hardware configurations. In this study, we consider homogeneous clusters because publicly available cluster information and datasets do not provide details on individual node hardware. Nodes can either be in a free state, where no job is currently running, or in an assigned state, where they are actively processing a job. Additional states, such as maintenance or shared usage, exist in real environments but are excluded from this study due to insufficient data and to simplify complexity.

The workload manager monitors the overall status of the HPC cluster. Often as part of the workload manager, the job scheduler places incoming jobs into a queue according to predefined strategies and fairness policies. This helps prevent excessive wait times for any single job. Job schedulers may also consider usage history or user quotas to prevent resource monopolization. Furthermore, the job scheduler tracks job execution to detect when jobs start, finish, or fail, and manages resource allocation accordingly—releasing nodes when jobs complete and assigning them to new jobs in the queue.

The job scheduler maintains a queue of submitted jobs, typically ordered by submission time. A scheduling algorithm then uses this queue to decide when and where each job should run, based on the queue order and current resource availability. Events such as job submission, completion, and cancellation typically trigger the scheduler. The algorithm evaluates the available resources and predefined job data (e.g., submit time, time limit, number of nodes, memory). Scheduling algorithms are essential for balancing system efficiency and fairness. While fairness prevents excessive wait times, it can sometimes conflict with optimal resource utilization, making a trade-off between both goals necessary.

First-come, first-served (FCFS) is a common and fair strategy for scheduling algorithms. However, FCFS can be inefficient when the next job in the queue cannot be started due to resource constraints, even though later jobs might fit. Backfilling [11] addresses this by allowing jobs further down the queue to start earlier, improving overall resource utilization. However, traditional backfilling does not consider fairness and may delay larger jobs significantly, as smaller jobs are more likely to fit into scheduling gaps.

This study uses the EASY backfilling approach [11] to address this issue. While sometimes expanded as *Earliest Available Start Time Yielding*, this backronym does not appear in the original publication and is used here for consistency with modern terminology. It preserves fairness by only allowing jobs to skip ahead if they do not delay the start time of the next job in the queue. Each job's user-defined time limit indicates when it should complete, which the scheduler uses to estimate resource availability and determine which jobs can start without causing delays.

To evaluate scheduling performance under different configurations, this study uses several commonly accepted metrics in HPC systems:

Wait Time: The amount of time a job spends in the scheduler's queue before it begins execution. High wait times can lead to user dissatisfaction and inefficient resource usage.

Makespan: In this study, the term refers to the total execution time of an individual job—from the moment it starts until it finishes.

Turnaround Time: The time from a job's submission to its completion ($WaitTime + Makespan$). This metric is important from the user perspective, as it indicates how long users wait to get their results.

Node Utilization: The percentage of available compute nodes actively used over the simulation period. Higher utilization implies better use of the system's resources.

2.2 Elasticity in HPC

Elasticity gives both the job scheduler and jobs more flexibility: It allows resources to be changed dynamically based on workload demands and changes in cluster conditions, such as varying node usage or hardware issues.

Elastic job scheduling refers to the job scheduler's ability to dynamically adjust resource allocation based on real-time factors, such as node utilization or resource availability. Backfilling is considered an early form of elastic job scheduling, as it improves resource utilization by allowing smaller jobs to run ahead of larger ones. For resource adjustments during execution, jobs must be elastic—a capability that is not yet standard.

Malleable jobs enable the job scheduler to adjust the number of allocated resources dynamically during execution. While HPC resources typically include elements such as storage (both local and network), in the context of this study, the term "resources" refers specifically to CPU nodes, as the simulations are based solely on CPU usage. A predefined minimum and maximum node value is needed in which the computation can be done. Therefore, the program must adapt to changing resources without losing progress or crashing, whereby the adjustment of resources makes the most sense between computation phases. Additionally, both the scheduler must support elastic jobs for effective implementation. In this work, malleable jobs represent the only form of elastic job type considered.

3 Methodology

This chapter describes in Section 3.1 the simulator used in this work and its setup. Section 3.2 deals with analysis, filtering, and conversion of the real workloads. Lastly, Section 3.3 leads to the final steps before execution.

3.1 Simulation Environment

This section is divided into two parts: Subsection 3.1.1 explains the structure of the simulator, while Subsection 3.1.2 describes the configuration used in this study.

3.1.1 ElastiSim

This study uses the simulation software ElastiSim [9] for all simulations. ElastiSim is a simulation framework for job scheduling that supports malleable jobs and various scheduling algorithms. It extends SimGrid [12], a framework designed for simulating distributed platforms and applications.

The cluster configuration (e.g., number of nodes, FLOPS, storage system) is defined in a SimGrid compatible `XML` file. The workload must be provided separately and must include information such as submit time, required nodes, job type, runtime (FLOPS), and application model. Malleable jobs use `num_nodes_min`, `num_nodes_max`, and `num_nodes_pref` to describe the number of nodes that can be assigned. Rigid jobs only need `num_nodes_pref`, which in this case means the exact number of required nodes.

The important variables are summarized in Table 3.1.

ElastiSim uses a hierarchical structure for application models, which allows simulating complex jobs with different phases and multiple iterations. Scheduling points

Variable	Description
`num_nodes`^{EJ}	The number of nodes assigned to a job
`num_nodes_min`^{EJ}	Maximum number of nodes for the job
`num_nodes_max`^{EJ}	Minimum number of nodes for the job
`num_nodes_pref`^{J}	Number of nodes the job prefers to run on or explicit needs for rigid job
`parallel_percentage`	Parallelized proportion of the program (e.g., 0.95 is 95%)
`max_node_efficiency_threshold`	The highest number of nodes where scaling factor(speedup divided by the number of nodes) is just below that value is set to `num_nodes_max`
`pref_node_efficiency_threshold`	Defines the `parallel_percentage` by choosing the closest possible value to reach that scaling factor
`dividation_split_time`	Defines how many seconds one computation phase take running at `num_nodes_pref`
`divide`^{J}	Number of iterations of a job calculated through `dividation_split_time`

^{E} ElastiSim variable
^{J} Job related

Table 3.1: Simulation environment: Important variables used

are automatically inserted by ElastiSim between phases or phase iterations, where reconfigurations may be applied. Since the application model is defined in advance, the number of scheduling points is also fixed. Therefore, the duration between scheduling points, further called *tick rate*, is a result of the application model and cannot be set directly. The performance model can be set individually and is also part of the application model.

ElastiSim provides a Python interface for developing custom scheduling algorithms. To control when the scheduling algorithm is invoked, suitable parameters (e.g., `schedule_on_job_submit`, `schedule_on_job_finalize`) can be set in the configuration file.

3.1.2 Simulator Configuration

All jobs consist of a single phase and use a `divide` value, which defines into how many parts the total computational effort (in FLOPS) is split. This value is passed to the application model's iteration parameter. In this study, the performance model uses Amdahl's law [13], see Equation 3.1, to calculate the computation time for each part depending on the actual assigned nodes.

$$\frac{1}{(1 - parallel\_percentage) + \frac{parallel\_percentage}{num\_nodes}} \tag{3.1}$$

For rigid jobs, the `divide` value is set to 1 because no reconfigurations are possible. For malleable jobs, the `divide` value is calculated by dividing the number of seconds required to complete the job (when running with `num_nodes_pref` assigned) by `dividation_split_time`. This results in a tick rate equal to `dividation_split_time` when the job runs on its preferred number of nodes. Otherwise, the tick rate for malleable jobs fluctuates depending on how many nodes are currently assigned.

This behavior is inherent to the design of ElastiSim's application model, which requires the number of iterations to be fixed at the beginning—a common case in parallel applications. The total load is then distributed across the currently assigned nodes using the calculated speedup. The relationship between the speedup at `num_nodes_pref` and `num_nodes` determines whether the tick rate shortens or extends, as described in Equation 3.2. As a result, the tick rate is capped at an upper limit of 5 × `dividation_split_time`, while the lower limit can drop below 0.01 seconds.

$$dividation\_split\_time \times \frac{(1 - parallel\_percentage) + \frac{parallel\_percentage}{num\_nodes}}{(1 - parallel\_percentage) + \frac{parallel\_percentage}{num\_nodes\_pref}} \tag{3.2}$$

The range of this effect can be influenced by adjusting the threshold value and the distribution from which `parallel_percentage` is drawn. However, this would also affect the workload behavior, leading to no other way than accepting it, and studies [14] show the total resource change time can be inside a similar range.

The job scheduler is invoked upon job submission, completion, and reconfiguration—moments at which resource adjustments are most meaningful.

In addition to the four previously existing scheduling algorithms [6], a new one was developed and included in this study. All five implemented scheduling algorithms use FCFS.

`easy_rigid_backfill`
: With the EASY backfilling feature, it provides performance above typical rigid schedulers and serves as the baseline for all simulation results.

`malleable_average`, `malleable_min` & `malleable_pref`
: These malleable algorithms attempt to start as many jobs as possible by shrinking running jobs to their average, minimum, or preferred node count, respectively. When a decision to shrink or expand must be made, the algorithm selects the job that is furthest from its respective target (minimum, preferred, or average node count) to shrink, and conversely, the job closest to its target is expanded if resources become available. In the case of `malleable_average`, the deviation from the preferred value is considered as a percentage, reaching 100% at the maximum. Expansions are only initiated when no more jobs can be started and nodes would otherwise remain free. All three also use the EASY backfilling feature.

`malleable_prefkeeper`
: A malleable scheduling algorithm developed as part of this study. It aims to maintain the preferred node count for each job, shrinking only when this value is exceeded and expanding only when idle nodes cannot start another job. Like the others, it uses EASY backfilling. The preferred node count is assumed to provide a good speedup efficiency, as users tend to optimize for turnaround time and cost (limited computing time) [15]. The actual benefit depends heavily on the performance model.

3.2 Workload Preparation

This study uses real workload data from the following HPC clusters:

Theta [16]

The Theta HPC cluster was operated by the Argonne Leadership Computing Facility (ALCF) from the installation in 2017 until its retirement end of 2023. All 4,392 nodes got an Intel Xeon Phi 7230 reaching 11.7 PetaFLOPS in total, placed rank 16 in the Top500 list (06/2017). There was also a GPU partition (ThetaGPU) with 24 nodes for AI training.

Eagle [17]

The Eagle HPC cluster was operated by the National Renewable Energy Laboratory Computational Science Center (NERSC) from the installation in 2018 until its decommission in June 2024. Every of the 2,618 nodes contained two Intel Xeon Gold Skylake 6154 processors, providing a peak performance of 8 PetaFLOPS, placed rank 35 in the Top500 list (11/2018). Therefore, 50 nodes are also equipped with two NVIDIA Tesla V100 GPU accelerators each.

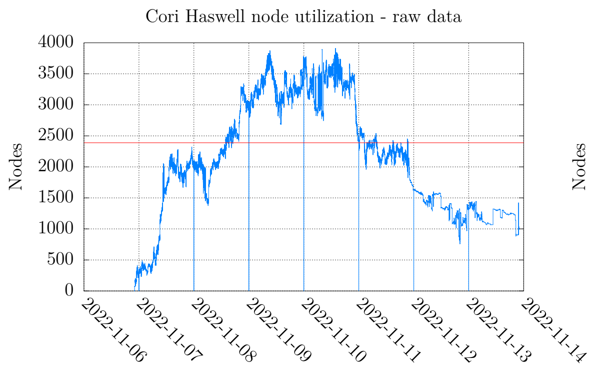
Cori [18]

Cori was operated by the National Energy Research Scientific Computing Center from the installation in 2015 until its retirement in May 2023, named in honor of the first American woman to win a Nobel Prize. The Haswell partition consisted of 2,388 nodes with Intel Xeon E5-2698 v3 processor with peak performance of 2.81 PetaFLOPS. The KNL partition consisted of 9,688 nodes with Intel Xeon Phi 7250 processors with a peak performance of 29.5 PetaFLOPS, placed rank 5 in the Top500 list (11/2016).

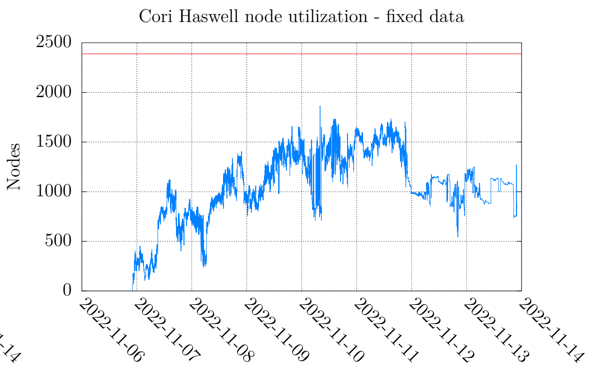
Workload datasets come from different external sources, meaning their structure and content vary. As this study only deals with CPU cluster simulations, GPU configurations need to be detected and carefully removed. Provided Eagle data also shows how many GPUs were requested, which makes it simple to remove those jobs by filtering all jobs with more than 0 requested GPUs. Technically, it's possible to use the GPU node partition without requiring GPUs. However, given that there are only 50 GPU nodes and 2,568 CPU-only nodes, it would not make sense to submit a job to the GPU partition, nor would it be welcomed by cluster operators. Those 50 nodes are also subtracted from the total node count, which completes the first step for the EAGLE data.

THETA also had a GPU partition, but the workloads for both partitions are provided separately, which makes additional steps to take account for GPU nodes unnecessary, while HASWELL and KNL do not have any GPU's installed.

For the next step we generated node utilization plots from all workload datasets, using the start/end time and number of nodes from jobs, to reveal anomalies. Those graphs are not expected to be exactly the same as the node utilization graphs from simulations. They are using the start time to reconstruct the node utilization while the simulations use the submission time to run under the same conditions.



(a) Raw data



(b) After cleaning and merging

Figure 3.1: HASWELL: The raw data facing some abnormalities (Figure 3.1a). HASWELL got 2,388 nodes and the utilization shows a peak of nearly 4,000 nodes, which reflects the insufficent data from shared jobs. Drops on every change of day occur because of the split jobs inside the daily workload datasets. After merging splitted jobs and removing mixed jobs the graph looks realistic (Figure 3.1b)

Figure 3.1 compares the raw and cleaned node utilization data for the HASWELL workload. For HASWELL, the raw data (Figure 3.1a) shows that the node utilization drops to 0% at the start of each new day. The peak reaches approximately 4,000 nodes, despite the HASWELL partition having only 2388 nodes [18]. Further, 192 nodes are for `genepool` and `genepool_shared` QOS reserved, from which are $\sim$ 34.54% of the total node utilization and $\sim 27.07\%$ alone from the shared ones, which shows clearly the artificial blowup in node utilization from those jobs. Data from CORI (HASWELL and KNL) is originally provided in daily segments. If a job starts but ends on a different day, it appears as separate entries for each day. Therefore, CORI also supports mixed jobs, which means multiple jobs can share one node, and it is not possible to merge them among other reasons, due to different start/end times. Even if ElastiSim supported that kind of jobs, a proper simulation with mixed jobs would not be an option without knowing how much of a node is used by a job. The best way is getting rid of them instead of simulation a total random-guessed behavior.

Those problems are only related to CORI workload data. To get clean data (Figure 3.1b), split jobs were identified by comparing their IDs and submit times, then merged using a Python script, while mixed jobs were filtered by QOS.

A summary of how many jobs where filtered out during those processes is shown by Table 3.2. Especially HASWELL lost a lot of jobs due to this filtering process, but considering that all of those jobs can share a node to run the real runtime of those jobs is lower than the raw calculated value. The THETA workload was not changed at all, as it did not contain mixed jobs or GPU jobs and has not shown any other anomalies.

HPC Cluster	Raw Jobs	Cleaned Jobs	Runtime loss
EAGLE	7,390,585	72,580	~165,000 h (~ 1.275%)
KNL	55,587	14,063	~41 h (~ 0.004%)*
HASWELL	119,781	91,200	~33,370 h (~13.680%)*

* Jobs with shared nodes

Table 3.2: EAGLE, KNL & HASWELL: Raw jobs and cleaned jobs

For the simulation the workload data needs to be converted into a suitable data structure, see Table 3.3. Rigid jobs are simply converted as all necessary information is already available. If `timelimit` is undefined, 125% of the original runtime will be set, which is the average limit of analyzed data. Malleable jobs need a minimum and maximum node value, which is calculated with the speedup formula and an efficiency threshold value and ensures reasonable speedups inside the resulting node range [19].

Data	Type
`job_id`	`int`
`submit_time`	`DATE_FORMAT`
`num_nodes`	`int`
`runtime_seconds`	`int`
`timelimit` (optional)	`int`

Table 3.3: Simulator workload: Required job information

THETA and EAGLE provide workload data over years of the operating cluster. Thus, we have to set a timeframe to be simulated. For this, further analysis is necessary to avoid exceptional workloads, with fewer/more jobs or bigger/smaller ones than usual. Therefore, the workload datasets were used for further analysis to generate and plot statistics. This enables the identification of possible maintenance (low utilization segments), huge jobs often at the end of the year, probably for testing purposes, and other abnormalities. Left with normal segments of different length its save to choose any of them expecting reliable results. The time frames ultimately chosen for all workloads are listed in Section 3.3. Further workload characteristics, such as node requirements and job runtimes, are analyzed in Sections 4.1-4.4.

3.3 Experimental Design

To evaluate the effects of malleability, the malleable job proportion is increased in 20% increments from 0% to 100%. To mitigate the impact of randomness—such as particularly favorable or unfavorable job distributions—each simulation is repeated using 10 different random seeds. These seeds determine which specific jobs are converted into malleable jobs at each proportion. All results reported in Chapter 4 represent the average across these 10 runs.

ElastiSim produces raw output data from which various performance metrics are computed. To capture variability across simulation runs, error bars in the figures show the interquartile range (IQR), which reflects the middle 50% of values. The IQR is preferred over the standard deviation, as the latter assumes normally distributed data—a condition that cannot be assumed for metrics derived from real workload behavior.

To ensure fair comparisons, the first 12 hours of each simulation are excluded from analysis. This warm-up phase allows the cluster to ramp up from an idle state and often shows atypical behavior. Prior testing confirmed that this period introduces significant variability in performance metrics. After this point, the system reaches a steady state, and the results more accurately reflect typical performance patterns (see Figure 3.2).

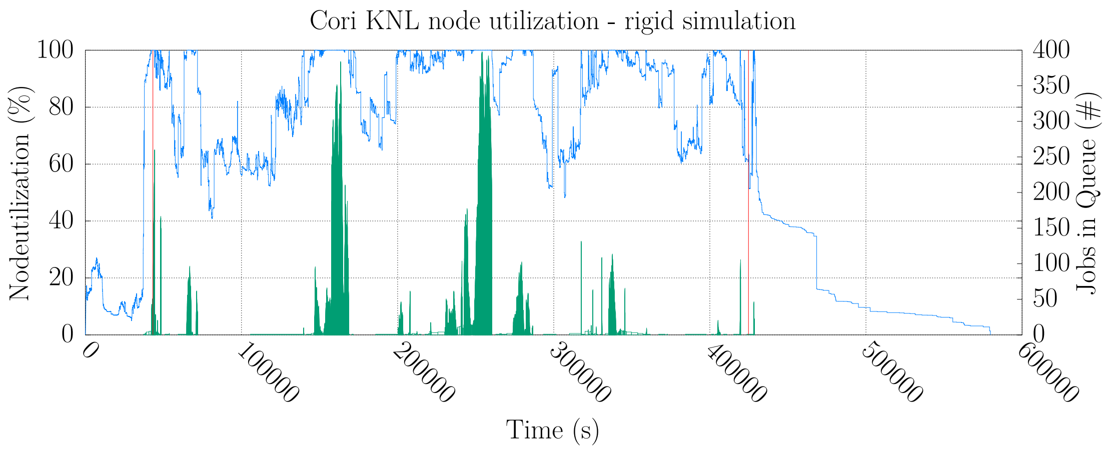


Figure 3.2: KNL: Node utilization graph with 100% rigid jobs. The red lines indicate the 12-hour warm-up period at the beginning and the time of the last job submission. Only the segment between these lines is used for metric calculations to avoid the influence of ramp-up and job depletion phases on the results.

Furthermore, any results after the last job submission are excluded from the analysis. This prevents incomplete job data from affecting the overall performance analysis, as jobs that continue running beyond the last submission can distort node utilization, expands/shrinks, and job completion times.

To reduce the number of simulations and thereby getting fast results, we have done technical optimizations. Since the seeds only determine whether a job will be malleable, only one seed per simulation is needed for 100% and 0% malleability.

More simulations can be cut down by running only the `rigid_easy_backfill` scheduler with a full rigid workload, because every algorithm is using FCFS with backfilling in this case. Vice versa, the `rigid_easy_backfill` scheduler is not performing better or worse with a malleable workload, also saving some simulations. Those missing simulations do not affect the results due to the mentioned reasons.

Dataset	**Start**	**Duration**	**Jobs**	**Tick rate**	**Nodes**
THETA [16]	2023-08-01 00:00:00	28 days	2.550	1 s	4,392
EAGLE [17]	2022-09-01 00:00:00	28 days	143,829	10 s	2,568
KNL [18]	2022-11-06 22:00:00	5 days 2 hours	41,524	10 s	9,688
HASWELL [18]	2022-11-07 00:00:00	5 days	28,259	1 s	2,388

Table 3.4: Simulation configurations

The tick rate significantly affects simulator runtime, and with large workloads, the runtime can easily exceed one week on our university cluster. Avoiding too long runtimes with all different workloads, some workloads need to be shortened or simulated with a higher tick rate. Especially CORI datasets contain workload data from a very short period of time, and simulating at a higher tick rate with more data is more reasonable, see Table 3.4. This change has just a very slight effect on the results and is inevitable for the affected workloads.

4 Results and Analysis

This chapter presents the results of the simulation experiments, analyzing how real-world workloads behave under varying levels of job malleability and across different scheduling algorithms. The analysis focuses on key performance metrics: job turnaround time, wait time, makespan, and node utilization.

Each workload is examined individually in Sections 4.1–4.4, highlighting workload-specific behavior and the influence of malleability. Section 4.5 concludes the chapter with a cross-workload summary that compares overall trends and reflects on the effects of different scheduling algorithms.

The following results reflect the average across simulations using 10 different random seeds per configuration.

4.1 Cori Haswell

The workload characteristics of Haswell are shown in Figure 4.1, which illustrates the distribution of job runtimes and required node counts.

This workload consists mostly of small jobs: Approximately 97.8% of all jobs request 32 or fewer nodes (see Figure 4.1a), and around 50% require only a single node. Regarding runtime (Figure 4.1b), 75% of jobs finish in under 1,000 seconds, which is reasonable given the total of 28,259 jobs submitted within a five-day period.

The node utilization plot in Figure 4.2 shows a significant job submission burst beginning after 300,000 seconds. This can lead to high peaks in wait time, depending on job size and duration, which in turn increases the average.

Figure 4.3 presents the simulation results across four key metrics. For each metric, interquartile ranges (IQR) are shown as error bars to highlight variability in the results.

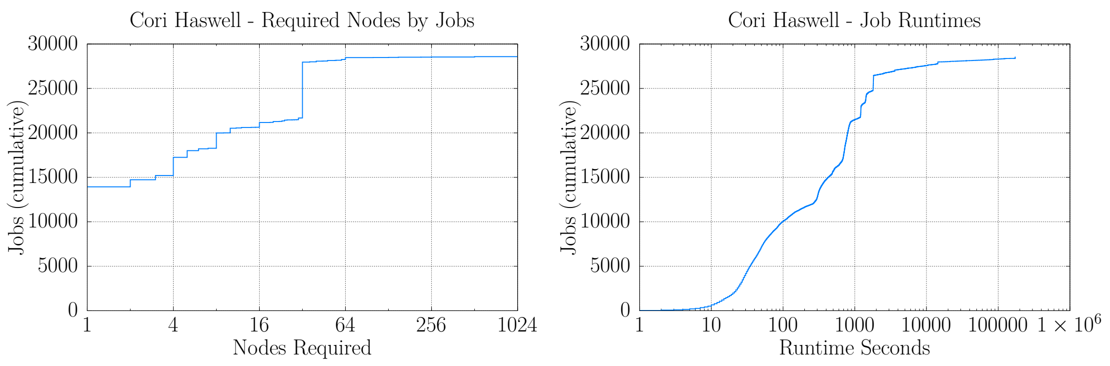


Figure 4.1: Distribution of job sizes and runtimes in the HASWELL workload. About 50% of jobs require only one node, approximately 22% request 32 nodes, and the rest are distributed up to 1,024 nodes. Most jobs complete in under 1,000 seconds.

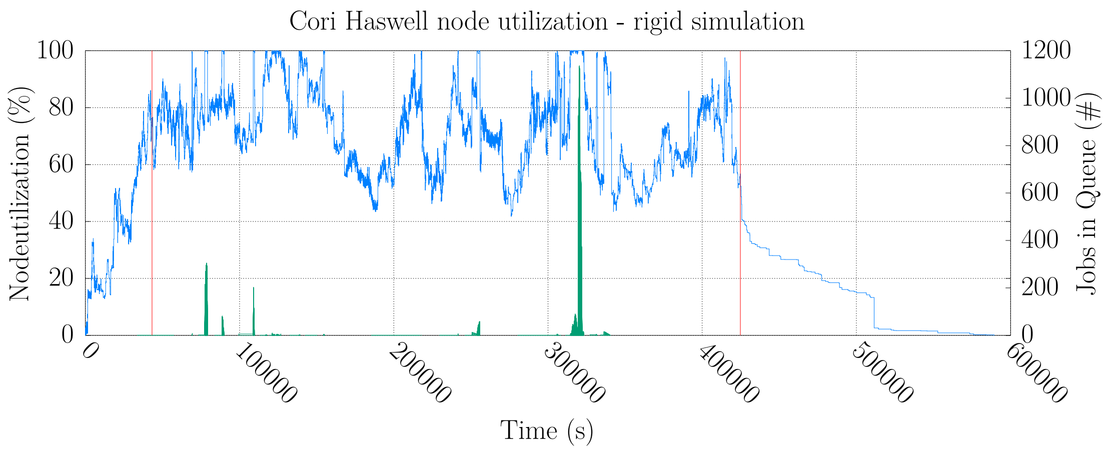
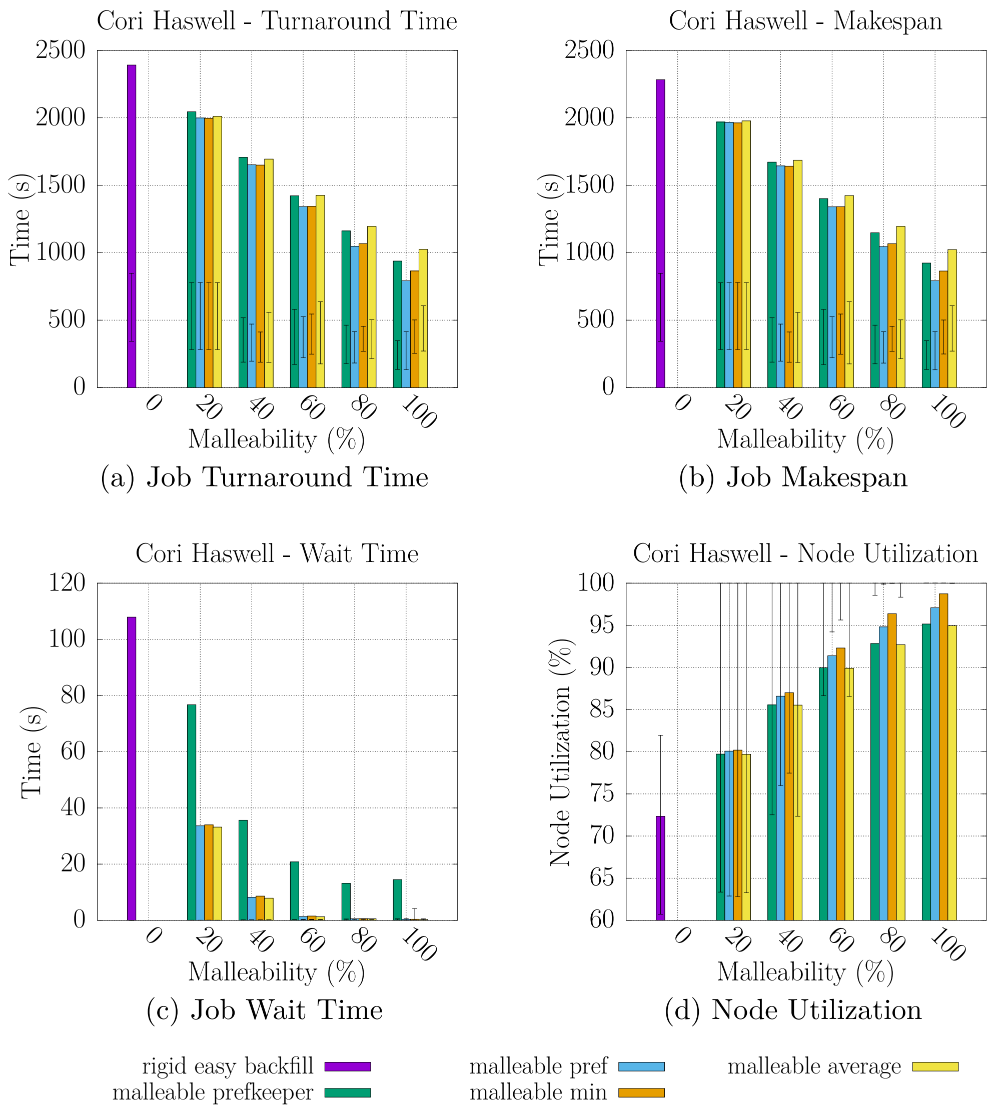


Figure 4.2: HASWELL: Node utilization graph with 100% rigid jobs and the `rigid_easy_backfill` scheduling algorithm. The red lines indicate the cut-offs: after 12 hour warmup (left) and at the last submitted job (right). After about 300,000 seconds, the green peak reveals a huge number of submitted jobs.

The average **job turnaround time** (Figure 4.3a) shows a steady decline from approximately 2391 seconds in the 100% rigid case, improving by 15–20% for each additional 20% of malleability. At 100% malleability, the turnaround time drops to



(a) Job Turnaround Time (b) Job Makespan

(c) Job Wait Time (d) Node Utilization

Figure 4.3: HASWELL: Average results regarding multiple metrics with IQR as error bars

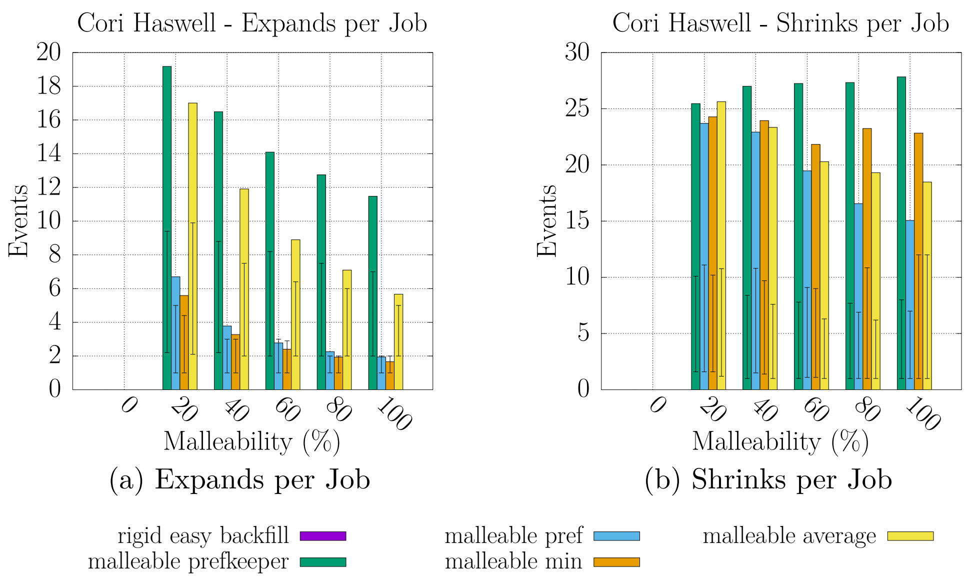
792 seconds.

This trend is mirrored in the **job makespan** (Figure 4.3b), which also decreases significantly with higher malleability proportions.

The **job wait time** (Figure 4.3c) falls below 10 seconds as early as 40% malleability.

Although `malleable_prefkeeper` exhibits noticeably higher average wait times, the IQR reveals that most wait times remain below one second even at 20% malleability. The elevated average is caused by the clustered job submission observed earlier, as supported by median wait times, which stay at or below 0.3 seconds across all proportions.

The average **node utilization** (Figure 4.3d) begins at 72% (with an upper IQR bound at 82%) for the rigid workload. It rises above 86% at 20–40% malleability, with a wider IQR spread that gradually tightens as malleability increases. At 100%, utilization peaks between 95–99%, depending on the scheduling algorithm.



(a) Expands per Job

(b) Shrinks per Job

Figure 4.4: HASWELL: Number of expands and shrinks per job

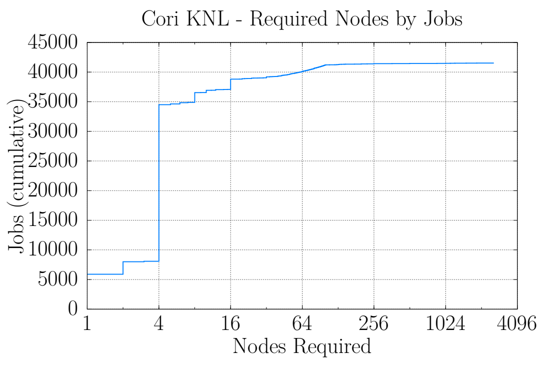
Regarding reconfiguration overhead, expand and shrink events per job (Figure 4.4) generally decrease with higher malleability. As shown in Figure 4.4a, `malleable_prefkeeper` generates the most average expand events, never falling below 11.4 per job, followed by `malleable_average`, which consistently remains above 5.6. Shrink behavior (Figure 4.4b) starts at around 25 events per job for all schedulers at 20% malleability. However, only `malleable_average` and `malleable_pref` exhibit a consistent downward trend as malleability increases.

In contrast, `malleable_prefkeeper` and `malleable_min` maintain higher levels of shrink activity throughout.

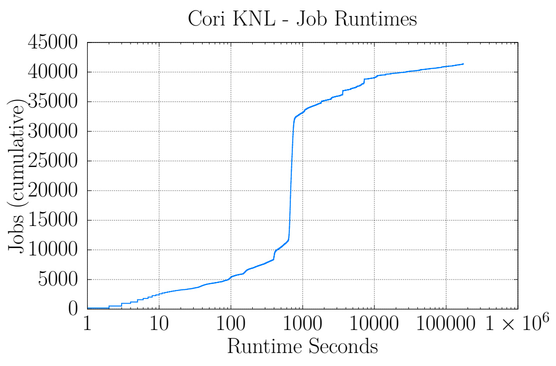
4.2 Cori KNL

The workload characteristics of KNL are shown in Figure 4.5, which illustrates the distribution of job runtimes and node requirements.

This workload, like Haswell, consists primarily of small jobs: Approximately 94.4% request 32 or fewer nodes (Figure 4.5a), and around 63% request exactly four nodes. In terms of runtime (Figure 4.5b), 80% of jobs complete within 1,000 seconds, with a significant peak in jobs with 600–800 seconds runtime. The total number of submitted jobs over five days is 41,524, a high number similar to Haswell.



(a) Number of nodes required by jobs



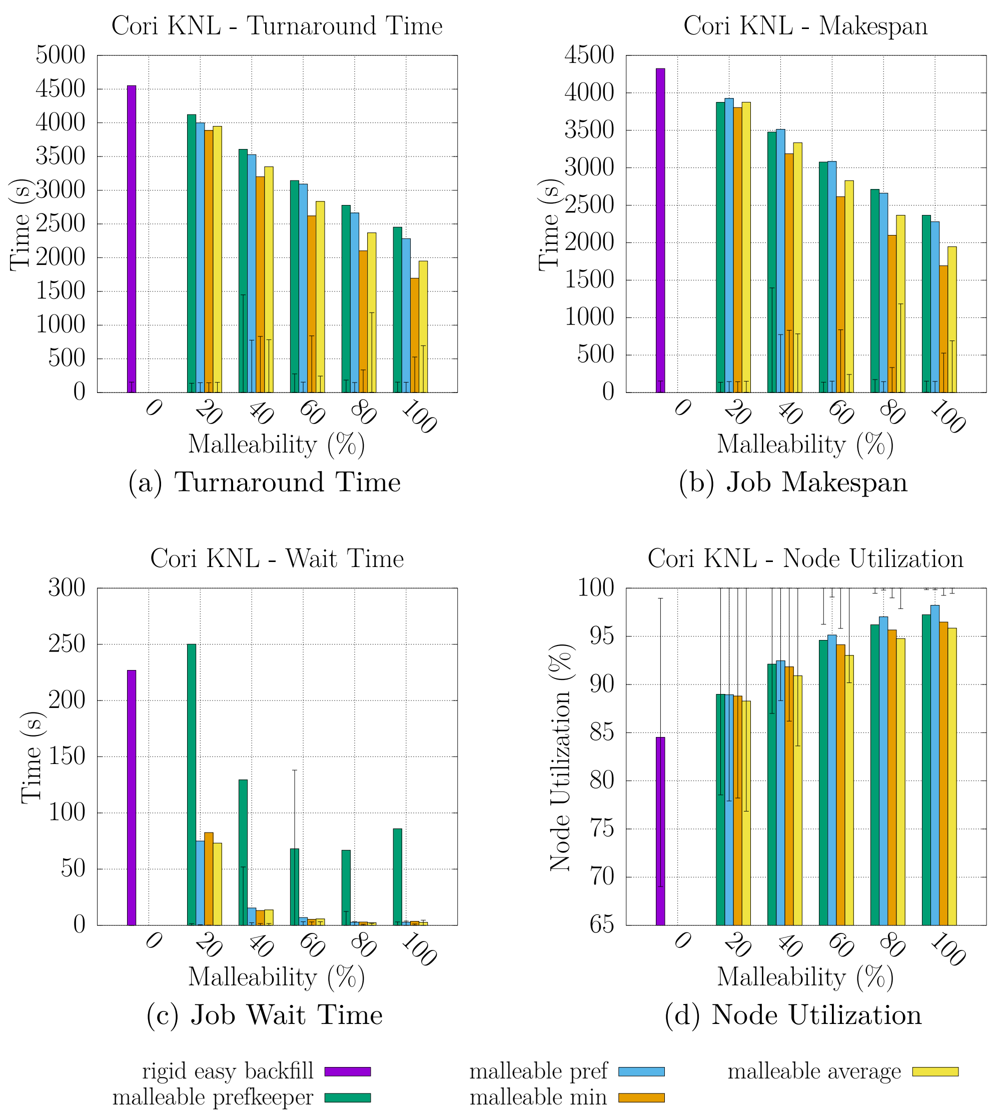
(b) Runtime of jobs

Figure 4.5: Distribution of job sizes and runtimes in the KNL workload. About 63% of jobs require four nodes and approximately 19% need less than four. Most jobs complete in under 1,000 seconds.

Figure 4.6 presents results across the four key metrics.

The average **job turnaround time** (Figure 4.6a) decreases consistently with rising malleability, falling from 4551 seconds to 1696 seconds at full malleability. Each 20% increment yields approximately 10–20% improvement.

The same pattern is observable in **job makespan** (Figure 4.6b), which closely tracks the turnaround time decline.



(a) Turnaround Time

(b) Job Makespan

(c) Job Wait Time

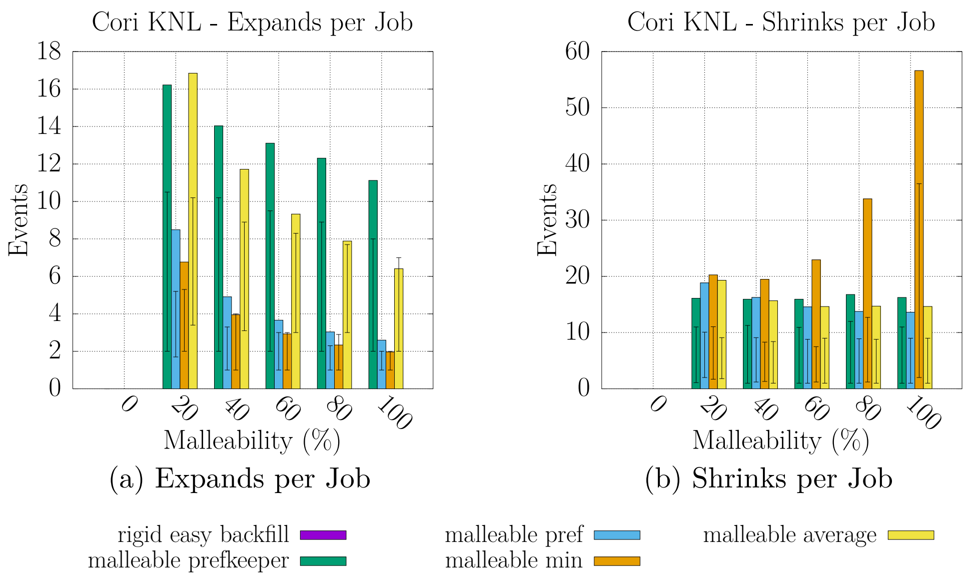
(d) Node Utilization

Figure 4.6: KNL: Average results regarding multiple metrics

For **job wait time** (Figure 4.6c), values fall below 20 seconds at 40% malleability. Again, `malleable_prefkeeper` stands out with higher averages and this time visibly increased IQRs at 40% and 60%. In Figure 3.2 showed earlier, the node utilization plot for KNL reveals broader peaks in job submission than HASWELL, which likely contributes to the wider IQR spread in this metric.

The **job wait time** (Figure 4.6c) falls below 20 seconds as early as 40% malleability. Again, `malleable_prefkeeper` is the only one that breaks the ranks, but this time also with increased upper IQR bounds, especially with 40 and 60% malleability proportions. The node utilization plot shown in Section 3.3 Figure 3.2 also showed segments of high submission. However, the submission peaks are more widely disturbed, what is reflected by an increased IQR.

The **node utilization** (Figure 4.6d) begins high at 85% and reaches around 95% at 60% malleability. Beyond that point, the gains stagnate, suggesting the system is nearing capacity limits. The IQR narrows with higher malleability, indicating more consistent resource use across seeds.



(a) Expands per Job (b) Shrinks per Job

Figure 4.7: KNL: Number of total expands and shrinks

Figure 4.7 shows shrink and expand operations per job.

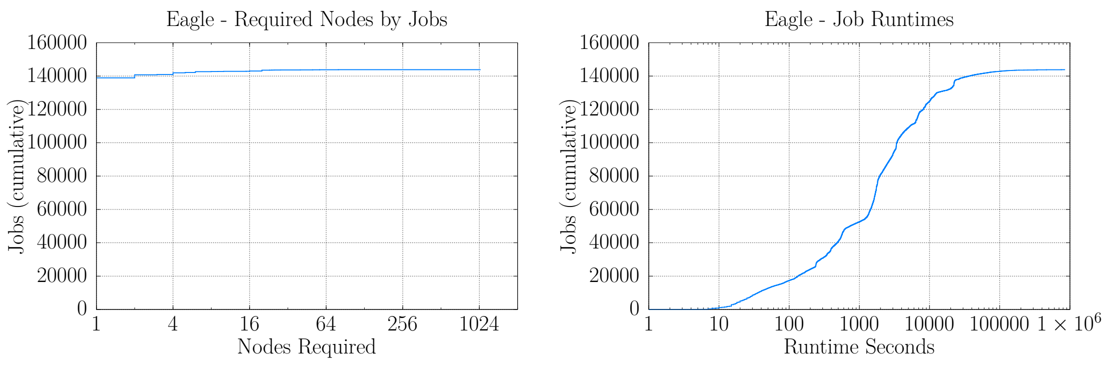
Expand events (Figure 4.7a) decrease steadily with higher malleability. `malleable_prefkeeper` and `malleable_average` consistently trigger the most expand operations, more than double the amount of the other algorithms.

In contrast, shrink events (Figure 4.7b) remain relatively stable. Most algorithms start at around 20 shrinks per job and show only slight decreases. `malleable_min` is the exception, showing an increase.

4.3 Eagle

The workload characteristics of Eagle are shown in Figure 4.8, which illustrates the distribution of job runtimes and required node counts.

This workload consists almost entirely of small jobs: Approximately 96.6% request only one node (Figure 4.8a). In terms of runtime (Figure 4.8b), 86.8% of jobs finish in under 10,000 seconds.



(a) Number of nodes required by jobs (b) Runtime of jobs

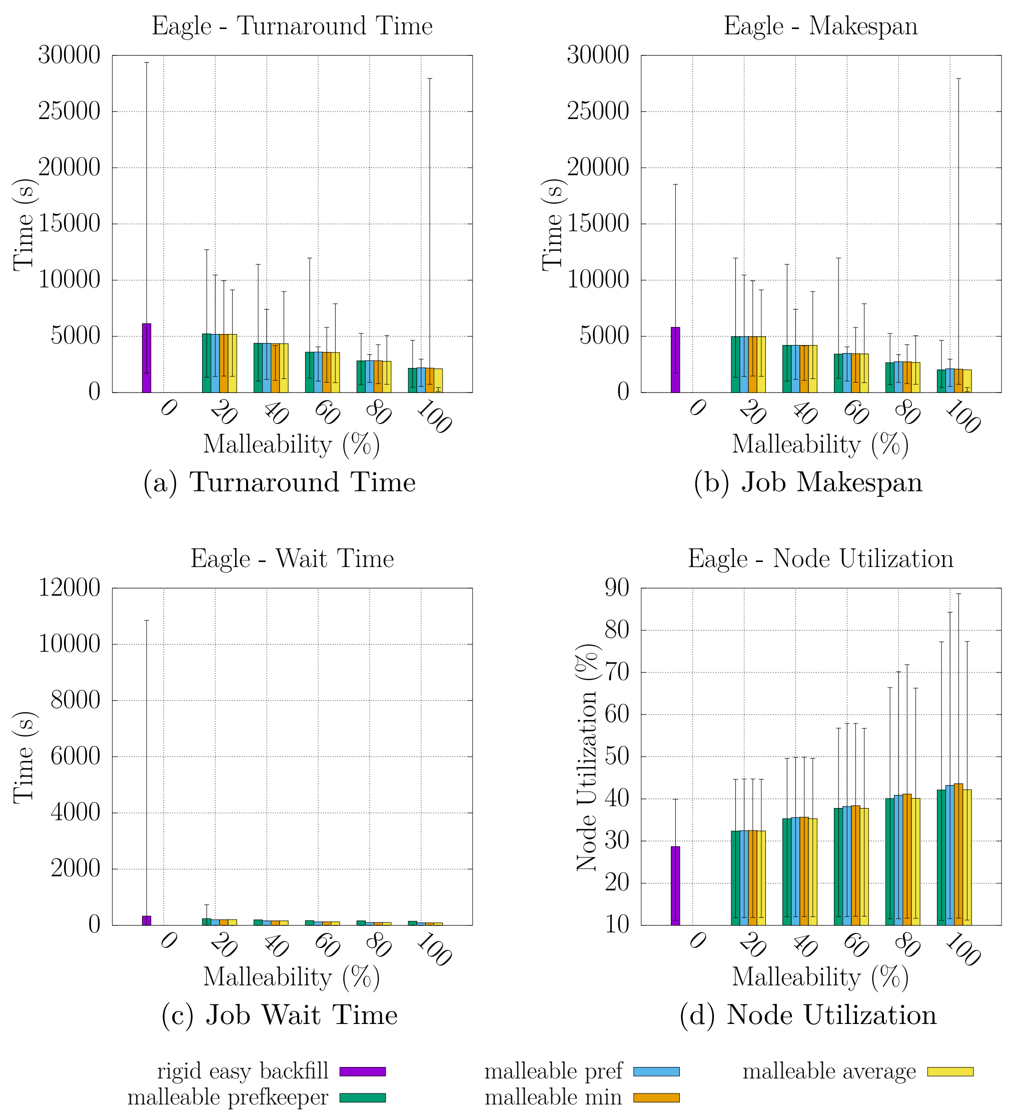
Figure 4.8: Distribution of job sizes and runtimes in the Eagle workload. About 96.6% of jobs require only one node. Most jobs complete in under 10,000 seconds.

The metrics in Figure 4.9 reveal a workload quite different from the Cori workloads.

The **job turnaround time** (Figure 4.9a) drops from a rigid baseline of 6,138 seconds to 2,114 seconds at full malleability—representing a 65% improvement. Although the differences between bars appear visually small due to scale, the upper IQR of 29,377 seconds for the rigid case confirms a wide spread in job completion times.

A similar pattern is observed in the **job makespan** (Figure 4.9b), which follows the same downward trend as malleability increases.

The **job wait time** (Figure 4.9c) improves by up to 73%, decreasing from 330 seconds to 89 seconds. Notably, the upper IQR bounds also drop considerably,



(a) Turnaround Time (b) Job Makespan (c) Job Wait Time (d) Node Utilization

Figure 4.9: EAGLE: Average results regarding multiple metrics

meaning that even the 75th percentile of jobs benefit from significantly shorter waiting times.

The **node utilization** (Figure 4.9d) starts at a low average of 28.7%, rising to 43.6% at 100% malleability. This low baseline suggests that underutilization is not caused by resource scarcity, but likely by characteristics of the job mix.

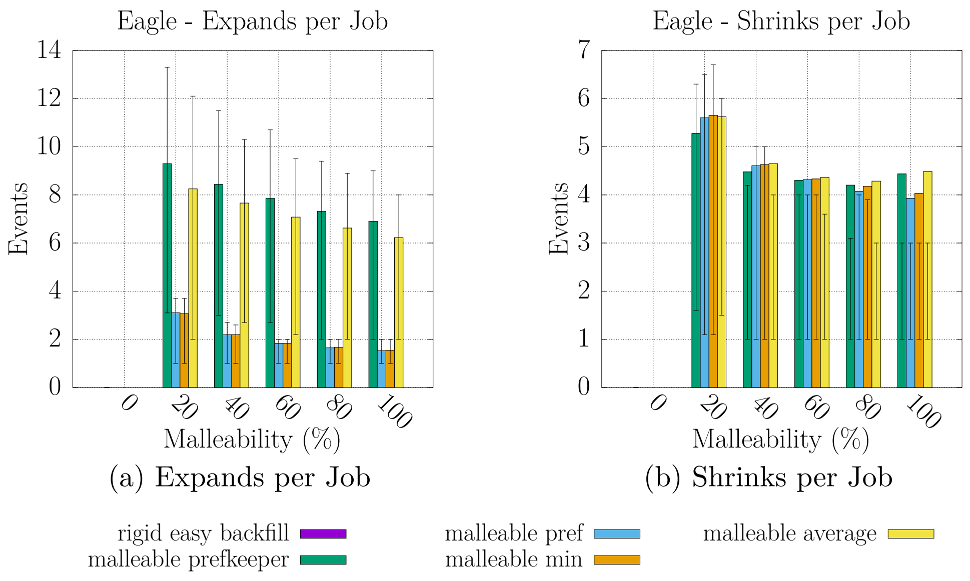


Figure 4.10: Eagle: Number of total expands and shrinks

In terms of reconfiguration events (Figure 4.10), the patterns are largely consistent across algorithms.

The number of expand events per job (Figure 4.10a) decreases slightly with increased malleability. `malleable_prefkeeper` and `malleable_average` start with 8–9 expands per job and decline to 6–7, while `malleable_pref` and `malleable_min` begin at around 3 and drop to 1–2.

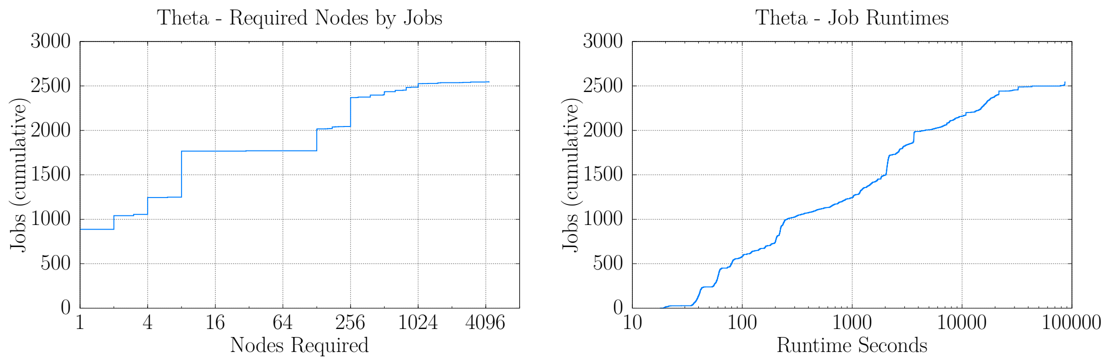
Shrink events (Figure 4.10b) are similar across all algorithms and exhibit only minor decreases with increasing malleability. The most noticeable reduction occurs between 20% and 40%.

4.4 Theta

The workload characteristics of Theta are shown in Figure 4.11, which illustrates the distribution of job runtimes and required node counts.

Among all four workloads, Theta has the most even distribution in node requirements (Figure 4.11a). The most common requests are for one node (approximately

34.8%), followed by 8 nodes (20.3%) and 256 nodes (12.6%). Job runtimes are also broadly distributed, with around 84.7% of jobs completing within 10,000 seconds (Figure 4.11b).



(a) Number of nodes required by jobs (b) Runtime of jobs

Figure 4.11: Distribution of job sizes and runtimes in the THETA workload. The majority of the jobs require one node (∼34.8%), followed by 8 nodes (∼20.3%) and 256 nodes (∼12.6%). Most jobs(∼84.7%) complete in under 10,000 seconds.

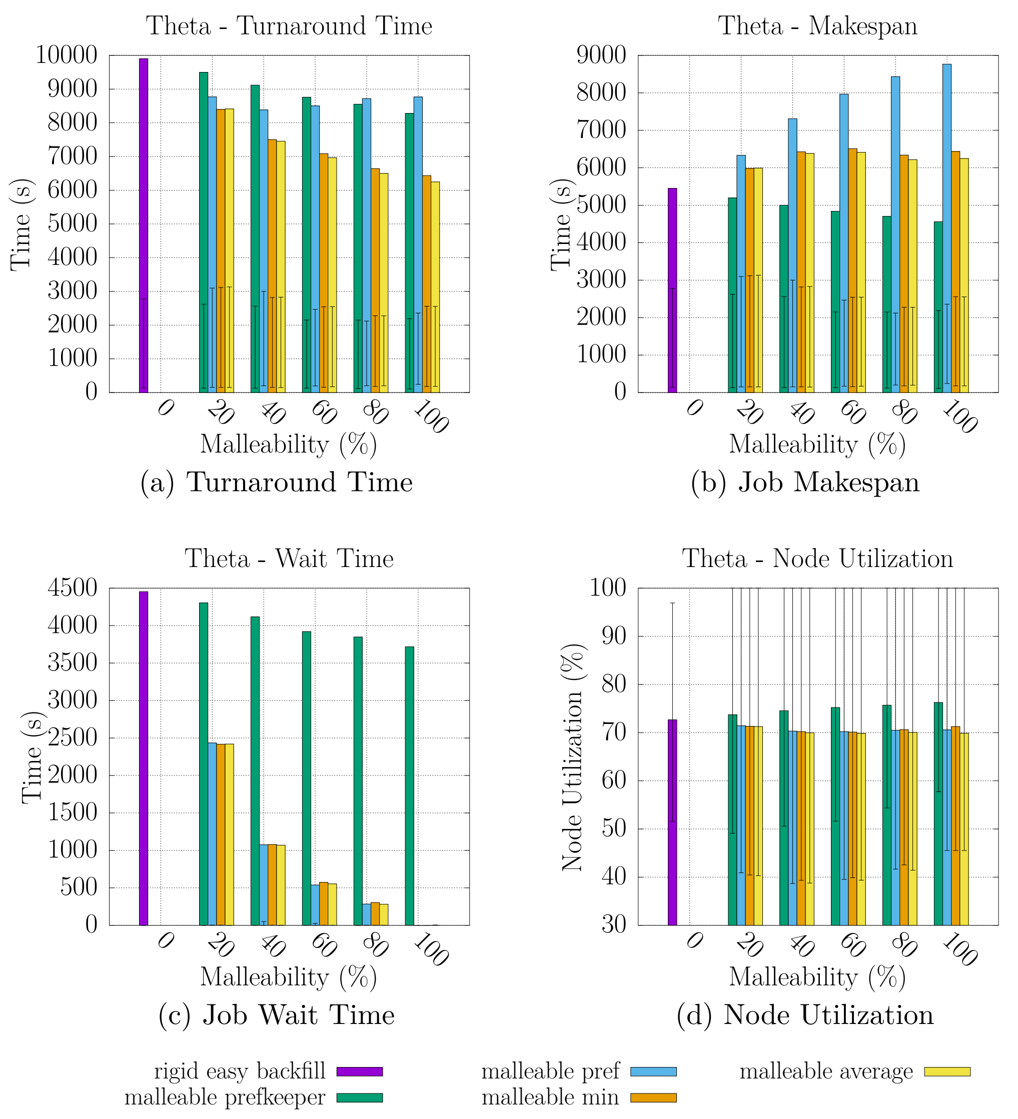
Figure 4.12 shows that THETA behaves differently than the other workloads, with greater variation between scheduling algorithms.

For **job turnaround time** (Figure 4.12a), three algorithms follow the familiar decreasing trend, achieving up to 36.9% improvement. However, `malleable_pref` shows a slight increase in turnaround time as malleability increases, deviating from the expected pattern.

The **job makespan** (Figure 4.12b) further highlights this anomaly. `malleable_pref` exhibits a clear increase in average makespan, while only `malleable_prefkeeper` shows consistent improvements—up to 16.3%.

In terms of **job wait time** (Figure 4.12c), `malleable_prefkeeper` remains relatively high despite a modest 16.5% improvement. By contrast, the other algorithms reduce wait times by more than 99%, marking a dramatic improvement.

The **node utilization** (Figure 4.12d) also deviates notably from previous workloads. `malleable_prefkeeper` achieves consistently slightly higher average utilization across malleability levels, while the other algorithms show little to no improvement compared to the rigid baseline, regardless of malleability proportion.

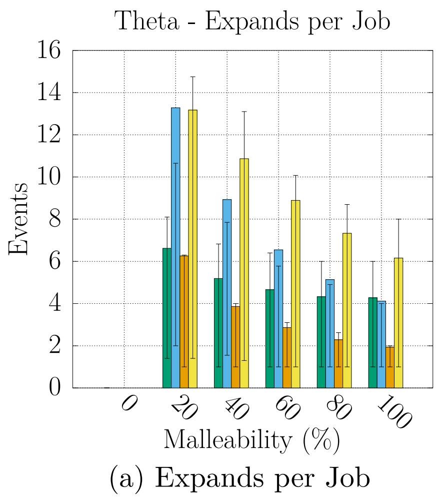


(a) Turnaround Time (b) Job Makespan (c) Job Wait Time (d) Node Utilization

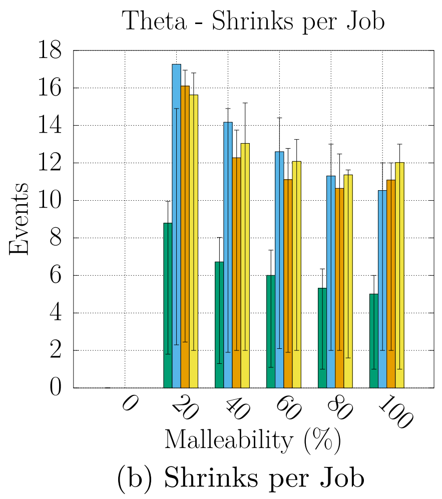
Figure 4.12: THETA: Average results regarding multiple metrics

The results in Figure 4.13 confirm the uniqueness of this workload's behavior in terms of reconfiguration frequency.

As shown in Figure 4.13a, `malleable_prefkeeper` begins with just over 6 expand events per job, similar to `malleable_min` at 20% malleability. At 100% malleability, both `malleable_prefkeeper` and `malleable_pref` converge to similar values, with



(a) Expands per Job



(b) Shrinks per Job

rigid easy backfill
malleable prefkeeper
malleable pref
malleable min
malleable average

Figure 4.13: Theta: Number of total expands and shrinks

`prefkeeper` remaining nearly constant.

Shrink events (Figure 4.13b) do not decrease significantly across malleability levels. However, `malleable_prefkeeper` consistently produces far fewer shrinks—about 50% fewer than the other algorithms across all tested configurations.

4.5 Cross-Workload Summary

As shown earlier in Table 3.4, the workloads from Theta, Eagle, KNL, and Haswell differ in total job count. However, since the timeframes vary, Table 4.1 presents the job submission rates in *jobs per hour* to enable fairer comparison.

Dataset	**Haswell**	**KNL**	**Eagle**	**Theta**
Job submission rate [jobs/hour]	235.49	340.36	214.03	3.79

Table 4.1: Job submission rates in *jobs per hour*

The extremely low submission rate of THETA is consistent with its longer job runtimes and higher node demands. In contrast, EAGLE features a massive number of small, single-node jobs, yet its submission rate is not especially high. This helps explain the low node utilization seen in that workload. The most comparable workloads are HASWELL and KNL, which also exhibit the most similar results across all metrics.

The primary focus of this study is not to develop new scheduling algorithms, but rather to investigate how malleability impacts real workloads. Still, comparing the tested scheduling strategies provides additional insight into performance dynamics.

The following algorithms were evaluated:

`easy_rigid_backfill`
: A rigid baseline scheduler using EASY backfilling. Since it lacks malleability support, it consistently performs worst in scenarios with malleable jobs. All algorithms behave identically on 100% rigid workloads, so only this one was simulated in that case. Likewise, it was not applied to workloads containing malleable jobs.

`malleable_average`, `malleable_min` & `malleable_pref`
: These schedulers shrink running jobs to allow waiting jobs to start. Their performance differs mainly in how they redistribute free resources: based on average, minimum, or preferred node targets. Differences arise when no more jobs can be started and the distribution strategy determines utilization or reconfiguration behavior.

`malleable_prefkeeper`
: This algorithm tries to preserve the preferred node count for each job, shrinking only when needed. It tends to yield higher wait times but can lower makespan. Its effectiveness varies depending on workload characteristics.

Except for THETA, the `easy_rigid_backfill` scheduler consistently showed the worst performance compared to malleable-capable strategies. THETA ’s uneven job submission patterns and higher node requirements may explain why the rigid strategy performed relatively better there.

Across the other workloads, malleability consistently improved turnaround time, makespan, and wait time while also increasing node utilization. Only `malleable_prefkeeper` resulted in higher average wait times, though even those remained below four minutes—a minor issue in typical HPC contexts.

Regarding reconfiguration behavior, `malleable_pref` triggered the fewest expands and shrinks across most workloads. An exception is THETA, where `malleable_prefkeeper` achieved the best overall performance.

5 Related Work

Numerous studies have explored the concept of elasticity and have demonstrated its benefits across various metrics (e.g. [6, 7, 8]. These studies show that even low proportions of malleable jobs can already lead to significant improvements in job wait times, job makespan and node utilization across different workloads and scheduling algorithms [7]. This observation aligns closely with the results of this study.

The study of *Eberius et al.* [20] is particularly relevant, as it also evaluates the impact of malleability using the Cori workload data. However, there are some differences between *Eberius et al.* [20] and our study, as described in the following.

In particular, the node utilization graphs with 100% rigid jobs differ significantly, especially in case of HASWELL. This discrepancy arises from differences in workload preparation—specifically, this study merges split jobs and removes mixed jobs, as detailed in Section 3.2; and *Eberius et al.* [20] did this not.

For example, the original HASWELL workload contains 119,781 jobs. After we merged jobs (a process that does not result in runtime loss), 8,726 jobs were removed. The referenced study reports only 118,818 jobs, suggesting some are excluded due to the chosen time window (from 11/7/22 to 11/12/22), even though some job submission times in the dataset start as early as 11/6/22 around 10 p.m. However, in this study, only 110,555 jobs remain after the merging step, even before removing mixed jobs. This suggests that the referenced study likely did not apply the same preparation steps, which explains why the node utilization graphs cannot be identical. In particular, their rigid simulation appears to have high often very near to 100% average node utilization. As the referenced study does not report average utilization metrics specific to the main execution period (excluding ramp-up and ramp-down), and since the figures do not allow for precise value extraction, a scientifically robust comparison cannot be made.

They use a one second tick rate for every simulation instead of one second for Haswell and ten seconds for KNL. Their scheduling algorithms also differ: while both studies employ backfilling, the referenced study does not specify the type of backfilling used beyond this general description, precluding a direct comparison of backfilling strategies. The malleable schedulers in their work use a "least recently modified" policy—either in a conservative form (one reconfiguration per tick) or an aggressive variant (no limit). Moreover, the reconfiguration behavior differs significantly: In their study, expansions double and shrinks halve the number of nodes, whereas this work employs a more fine-grained adjustment strategy. By avoiding coarse-grained resizing constraints, the fine-grained approach adopted here allows for increased flexibility and can improve elasticity, notably for workloads with a wide range of job sizes.

The KNL workload exhibits fewer differences due to preparation, and the trends across both studies remain closely aligned. Key patterns in node utilization and metric progression are clearly recognizable. While direct comparisons for Haswell are difficult due to the differing baselines, the node utilization improvements from malleability observed in KNL—about 35% with 100% malleable jobs—are consistent across both studies. The methodological differences and the closely aligned results between the studies provide additional support for the positive impact of malleability.

6 Discussion and Conclusion

This chapter begins by summarizing the key findings of the study in Section 6.1. Section 6.2 then addresses limitations and outlines directions for future work.

6.1 Summary of Key Findings

In this study, real workloads from Haswell, KNL, Eagle, and Theta supersede synthetic workload data to obtain more realistic results. These workloads were carefully prepared to ensure realistic behavior and were converted into rigid and malleable jobs for the ElastiSim simulator. To ensure reliability, ten different seeds were used to determine which jobs would become malleable. In addition to the `rigid_easy_backfill` scheduling algorithm, four other algorithms with malleability support were applied.

Results show that the scheduling algorithms themselves did not significantly impact malleability performance, as was initially expected. Only `malleable_prefkeeper` deviated slightly from the general trend. In three out of four workloads, it showed higher wait times and generally less pronounced benefits—though still performing well overall. Notably, for one workload where the other algorithms did not improve makespan and node utilization, `malleable_prefkeeper` achieved clear improvements in these metrics.

Malleability showed strong benefits across all metrics and workloads examined in this study. While the most significant improvements were observed with larger workloads, the gains remained significant across all scenarios and algorithms. On average, compared to purely rigid workloads, malleability achieved consistent improvements of 51.46% in job turnaround time, 83.36% in job wait time, 39.47% in job makespan, and 24.05% in node utilization across all workloads and scheduling algorithms. These improvements not only benefit cluster operators through better resource efficiency but also improve user experience by job reducing turnaround times. These effects were evident even with a low malleability ratio of just 20%.

6.2 Limitations and Future Directions

Attempting to model how malleability affects real-world HPC clusters is inherently challenging due to the complexity of these systems. Several aspects could be further improved or extended to address limitations in this study:

- Real datasets with greater detail and longer and continuous observation periods could help improve experiments like this.
- Adding simulated data traffic (I/O), potentially across multiple application phases, would be a valuable extension.
- As HPC clusters increasingly incorporate GPU nodes, it becomes necessary to adapt to datasets from GPU-enabled systems.
- A validated model or empirical study for the conversion from rigid to malleable jobs could improve accuracy.

Running simulations with more varied application models in the future—or even mixed-model workloads—would make the results more robust. Introducing multiple phases and varied performance models could help reveal the strengths and weaknesses of different scheduling algorithms more clearly.

With its exceptionally detailed cluster data, the M100 dataset [21] offers great potential for these kinds of enhancements and more.

Simulation results also suggest opportunities for developing scheduling algorithms tailored to specific operational goals. For example, scheduling with elasticity in mind could help maintain target node utilization levels—making it possible to shut down unused nodes for energy savings, maintenance, or other operational reasons. These strategies would reflect how HPC clusters are often planned with specific usage scenarios in mind.

As complexity increases, so does the effort required, and at some point, transitioning to real clusters with malleability support becomes the best way forward.

Bibliography

[1] Andy B. Yoo, Michael A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing. JSSPP 2003. Lecture Notes in Computer Science.* Ed. by Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Vol. 2862. Springer, 2003, pp. 44–60. DOI: 10.1007/10968987_3.

[2] Argonne Leadership Computing Facility (ALCF). *Theta Workload.* URL: https://reports.alcf.anl.gov/data/theta.html (visited on 03/25/2025).

[3] David Eberius. *elastic-sim.* Contains Cori workloads. URL: https://github.com/davideberius/elastic-sim (visited on 03/25/2025).

[4] National Renewable Energy Laboratory (NREL). *Eagle Workload.* URL: https://catalog.data.gov/dataset/nrel-eagle-supercomputer-jobs (visited on 03/25/2025).

[5] D. G. Feitelson and L. Rudolph. "Toward Convergence in Job Schedulers for Parallel Supercomputers". In: *Proceedings Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP).* Springer, 1996. DOI: 10.1007/BFb0022284.

[6] Jonas Posner, Fabian Hupfeld, and Patrick Finnerty. "Enhancing Supercomputer Performance with Malleable Job Scheduling Strategies". In: *Euro-Par 2023: Parallel Processing Workshops.* Ed. by Demetris Zeinalipour, Dora Blanco Heras, George Pallis, Herodotos Herodotou, Demetris Trihinas, Daniel Balouek, Patrick Diehl, Terry Cojean, Karl Fürlinger, Maja Hanne Kirkeby, Matteo Nardelli, and Pierangelo Di Sanzo. Cham: Springer Nature Switzerland, 2024, pp. 180–192. ISBN: 978-3-031-48803-0. DOI: 10.1007/978-3-031-48803-0_14.

[7] Debolina Halder Lina, Sheikh Ghafoor, and Thomas Hines. “Scheduling of Elastic Message Passing Applications on HPC Systems”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dalibor Klusáček, Corbalán Julita, and Gonzalo P. Rodrigo. Cham: Springer Nature Switzerland, 2023, pp. 172–191. ISBN: 978-3-031-22698-4. DOI: 10.1007/978-3-031-22698-4_9.

[8] Njoud O. Almaaitah, David E. Singh, Taylan Özden, and Jesus Carretero. “Performance-driven scheduling for malleable workloads”. In: *The Journal of Supercomputing* 80.8 (May 2024), pp. 11556–11584. ISSN: 1573-0484. DOI: 10.1007/s11227-023-05882-0.

[9] T. Özden, T. Beringer, A. Mazaheri, H. M. Fard, and F. Wolf. “ElastiSim: A Batch-System Simulator for Malleable Workloads”. In: *Proceedings of the 51st International Conference on Parallel Processing (ICPP ’22)*. Bordeaux, France: Association for Computing Machinery, 2023. DOI: 10.1145/3545008.3545046.

[10] Taylan Özden. *ElastiSim – A Simulator for Elastic HPC Scheduling*. URL: https://github.com/elastisim (visited on 06/15/2024).

[11] Dror G. Feitelson and Larry Rudolph. “Parallel job scheduling: Issues and approaches”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–18. ISBN: 978-3-540-49459-1. DOI: 10.1007/3-540-60153-8_20.

[12] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In: *Journal of Parallel and Distributed Computing* 74 (2014), pp. 2899–2917. DOI: 10.1016/j.jpdc.2014.06.008.

[13] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings Spring Joint Computer Conference (SJCC)*. ACM, 1967. DOI: 10.1145/1465482.1465560.

[14] Jonas Posner. “Dynamic Resource Management: Comparison of Asynchronous Many-Task (AMT) and Dynamic Processes with PSets (DPP)”. In: *Proceedings of the Workshop on Asynchronous Many-Task Systems and Applications*. To be published. 2025.

[15] Allen B. Downey. “A parallel workload model and its implications for processor allocation”. In: *Cluster Computing* 1.1 (1998), pp. 133–145. ISSN: 1573-7543. DOI: 10.1023/A:1019077214124.

[16] Argonne Leadership Computing Facility (ALCF). *Theta System*. URL: https://www.alcf.anl.gov/alcf-resources/theta (visited on 03/25/2025).

[17] National Renewable Energy Laboratory (NREL). *Eagle System*. URL: https://web.archive.org/web/20221128214549/https://www.nrel.gov/hpc/eagle-system-configuration.html (visited on 03/25/2025).

[18] National Energy Research Scientific Computing Center (NERSC). *Cori System*. URL: https://web.archive.org/web/20231001013828/https://docs.nersc.gov/systems/cori/ (visited on 03/25/2025).

[19] Jonas Posner and Claudia Fohry. “Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing”. In: *50th International Conference on Parallel Processing Workshop*. ICPP Workshops ’21. Lemont, IL, USA: Association for Computing Machinery, 2021. ISBN: 9781450384414. DOI: 10.1145/3458744.3473361.

[20] David Eberius, Md. Wasi-Ur Rahman, and David Ozog. “Evaluating the Potential of Elastic Jobs in HPC Systems”. In: *Proceedings of the SC ’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W ’23. Denver, CO, USA: Association for Computing Machinery, 2023, 1324–1333. ISBN: 9798400707858. DOI: 10.1145/3624062.3624199.

[21] Andrea Borghesi, Carmine Di Santi, Martin Molan, Mohsen Seyedkazemi Ardebili, Alessio Mauri, Massimiliano Guarrasi, Daniela Galetti, Mirko Cestari, Francesco Barchi, Luca Benini, Francesco Beneventi, and Andrea Bartolini. “M100 ExaData: a data collection campaign on the CINECA’s Marconi100 Tier-0 supercomputer”. In: *Scientific Data* 10.1 (May 2023), p. 288. ISSN: 2052-4463. DOI: 10.1038/s41597-023-02174-3.