UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

PhD program in Computer Architecture

Scheduling and resource management solutions for the scalable and efficient design of today's and tomorrow's HPC machines

Doctoral thesis by:

Marco D'Amico

Thesis advisors:

Julita Corbalan, Ana Jokanovic

Department of Computer Architecture (DAC)
Barcelona, June, 2021

Abstract

In recent years, high-performance computing research became essential in pushing the boundaries of what men can know, predict, achieve, and understand in the experimented reality. HPC Workloads grow in size and complexity hand to hand with the machines that support them, accommodating big data, data analytic, and machine learning applications at the side of classical compute-intensive workloads. Simultaneously, power demand is hugely increasing, becoming a constraint in the design of these machines.

The increasing diversification of processors and accelerators, new special-purpose devices, and new memory layers allow better management of these workloads. At the same time, libraries and tools are being developed to support and to make the most out of the hardware while offering standardized and straightforward interfaces to users and developers. Different scheduling and resource management layers are fundamental in organizing the work and the access to resources.

This thesis focuses on the job scheduling and resource management layer. We claim that this layer needs research in the following three directions: awareness, dynamicity, and automatization. First, awareness of the hardware and applications characteristics would improve the configuration, scheduling, and placement of tasks. As a second point, dynamic systems are more responsive. They react fast to changes in the hardware, e.g.,in failure cases, and they adapt to application's requirements changes. Finally, automatization is the last direction. In our opinion future systems need to act autonomously. A system that keeps relying on user guidance is prone to errors and requires unnecessary

user expertise.

This thesis presents three main contributions to fill those gaps. First, we developed DROM, a transparent library that allows parallel applications to shrink and expand dynamically in the computing nodes. DROM enables efficiently utilization of the available resources, with no effort for developers and users. We enabled vertical malleability, i.e., internal malleability in computing nodes. DROM enables mixing workflow stages to get on-line data while the workflow is running, and it allows to run urgent jobs without stopping others.We measured a negligible overhead and we integrating DROM with OpenMPI, OmpSs, and MPI.

As a second contribution, we developed a system-wide malleable scheduling and resource management policy that uses slowdown predictions to optimize the scheduling of malleable jobs. We called this policy Slowdown-Driven policy (SD-policy). SD-policy uses malleability and node sharing to run new jobs by shrinking running jobs with a lower slowdown, only if the new job has reduced predicted end time compared to the static scheduling. We obtained a very promising reduction in the slowdown, makespan, and consumed energy with workloads combining compute-bounded and memory-bounded jobs.

Ultimately, we used and extended an energy and runtime model to predict the job's runtime and dissipated energy for multiple hardware architectures. We implemented an energy-aware multi-cluster policy, EAMC-policy, that uses predictions to select optimal core frequencies and to filter and prioritize job submissions into the most efficient hardware in case of heterogeneity. This is done automatically, reducing user's intervention and necessary knowledge. Simulations based on real-world hardware and workloads show high energy savings and reduced response time are achieved compared to non-energy-aware and non-heterogeneous aware scheduling.

Dedicated to the finding of the self fading into the infinite Order.

... e alla mamma.

Acknowledgments

Prima di tutto devo ringraziare i miei genitori e i miei fratelli. Loro sono la base di tutto, la mia terra, le mie radici senza le quali non sarei mai potuto crescere fino a produrre i miei frutti. Massa e gravità, mi hanno sempre permesso la spinta necessaria per spiccare un salto e l'appoggio per atterrarci. Massa e gravitá, mi hanno sempre tenuto attaccato a questo sistema solare, anche quando me ne vado nei dintorni di Plutone.

Poi è il turno di ringraziare tutti compagni della mia vita, los compañeros de este viaje, my travel companions, ogni persona che ho incrociato mi ha isegnato qualcosa dandomi un pezzo di sé, come pietre che deviano lo scorrere di un fiume creando innumerevoli vortici. Pietricco sul fondale, pietre preziose, blocchi enormi che solo sfiorarli ti fanno sentire la loro imponenza. Ogni pietra è una deviazione di un cammino altrimenti troppo noioso.

A la meva família catalana, que em va obrir portes, i em va donar la benvinguda, gràcies. A través de vosaltres vaig poder sentir aquesta terra com a la meva terra, i plantar arrels que van créixer fins al meu cor. Barcelona és una ciutat magnífica en una terra magnífica, que et dóna l'oportunitat de descobrir tot el que ets.

Estoy muy agradecido a BSC y su gente. Gracias a la oportunidad que me dieron pude descubrir un mundo nuevo, de ideas, investigación, personas, crecimiento, oportunidades. Cada discusión, reunión, viaje, conferencia me ha hecho confrontar conmigo mismo y los demás, creando formas en el inmenso caos de la mente. A mis colegas, que aguantaron mi humor de cada día, gracias.

A mis directoras, fueron la ayuda fundamental para la realización de este proyecto, y no solo, también me respetaron, protegieron, y me enriquecieron en método, disciplina, y humanidad. Gracias infinidas.

Gracias a todos los maestros que encontré en mi camino, cuando ves uno el impacto es inmediato, solo hace falta una mirada.

Grazie, merci, gracias, thank you, arigatou.

Contents

List of figures

List of Tables

Journal papers

[1] Marco D'Amico and Julita Corbalan. Energy hardware and workload aware job scheduling towards interconnected HPC environments. In *IEEE Transactions on Parallel and Distributed Systems TPDS*. IEEE, 2021. (under review).

Conference papers

[2] Marco D'Amico, Ana Jokanovic, and Julita Corbalan. Holistic Slowdown Driven Scheduling and Resource Management for Malleable Jobs. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

Workshop papers

[3] Marco D'Amico, Marta Garcia-Gasulla, Víctor López, Ana Jokanovic, Raül Sirvent, and Julita Corbalan. DROM: Enabling Efficient and Effortless Malleability for Resource Managers. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, pages 41:1–41:10, New York, NY, USA, 2018. ACM.

[4] A. Jokanovic, M. D'Amico, and J. Corbalan. Evaluating SLURM Simulator with Real-Machine SLURM and Vice Versa. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 72–82, Nov 2018.

[5] Victor Lopez, Ana Jokanovic, Marco D'Amico, Marta Garcia, Raul Sirvent, and Julita Corbalan. DJSB: Dynamic Job Scheduling Benchmark. In *Job Scheduling Strategies for Parallel Processing*, pages 174–188, Cham, 2018. Springer International Publishing.

”The quickest way to learn about a new place is to know what it dreams of.”

Stephen King

1 Introduction

Scheduling, in its broader meaning, consists of distributing a set of tasks in a euclidean space. While linked to the inherited uncertainty of the observed world and human beings’ capacity to make decisions and planning, scheduling started to be largely studied with machines creation.

First, machines were designed to solve a unique scope, e.g., sewing a dress. The scheduling process was limited to establishing the necessary components and their order in a pipeline.

The scheduling concept evolved together with machines, and today it plays a vital role in performing tasks, such as the simulation of entire human brain areas, objective of the European Human Brain Project [51]. At a high level of abstraction, we used the following classification to describe the modern

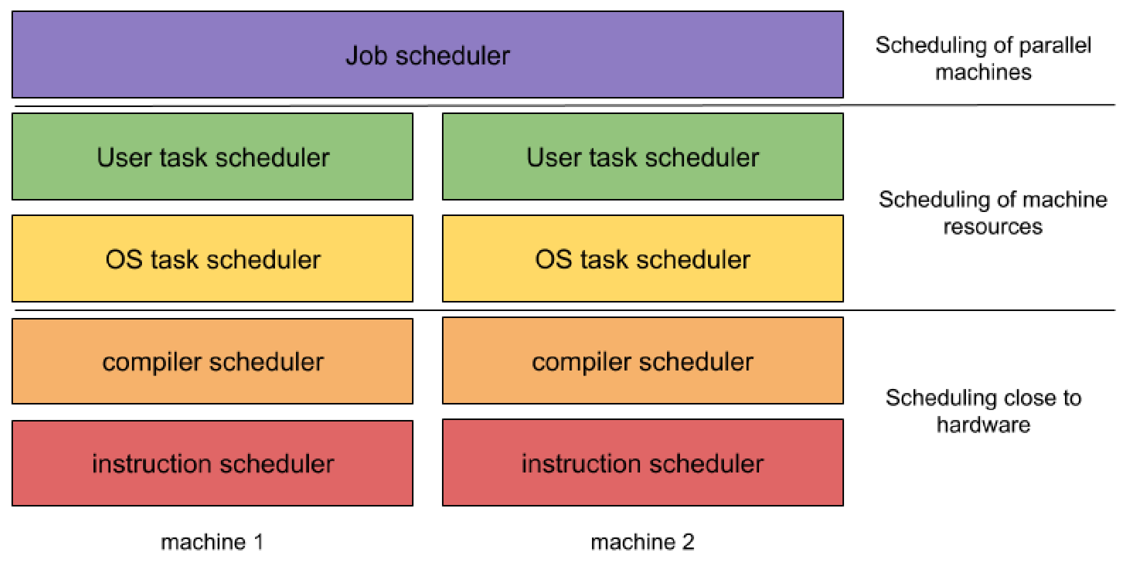


Figure 1.1: Scheduling layers classification.

machines scheduling layers, as shown in Figure 1.1: scheduling close to the hardware, scheduling of single machine's resources, scheduling of parallel machines. Each of these layers' objective is to maintain managed entities' coherence and optimize their flow to maximize their throughput.

The first layer includes hardware and software schedulers to optimize the flow of instructions in computing and memory devices. The hardware scheduler implements scheduling logic in the hardware. Some examples are the reordering of instructions for out-of-order processors such as the reorder buffer or the main memory scheduler, grouping reads and writes together to hide memory latencies. Software schedulers, included in the compiler, reorder the flow of instructions to maximize pipelines throughput without breaking the code logic. In this layer, the scheduled entity is called *instruction*, and we call the scheduler *instruction scheduler*.

The second layer is in charge of organizing one or multiple codes running on one or multiple computing resources within a single system, e.g., a mobile phone, a laptop, or a parallel machine's computing node, managed by a single piece of software, the Operating System (OS). The scheduler is a runtime component, implementing low complexity policies that run at a fine granularity. The smallest entity to the scheduler is a *thread (or task)*, and multiple tasks can be part of the same *process*, where they share the address space. The *task scheduler* guarantees access to hardware resources in an exclusive way by usually implementing circular time-partitioning policies based on fairness among tasks, which grants the concurrent execution of multiple tasks using context-switching techniques.

Besides, in the case of multi-core resources or multiple computing devices, the OS includes API to pin tasks to specific resources. This API can be used to provide tools for users and upper scheduling layers. Users can manage the pinning of their own processes to available resources. Higher-level schedulers, including programming models or third layer scheduling components, can use the same API to manage and reserve resources for users and processes.

Finally, in the third layer, we have the scheduling of clusters of computing nodes. These machines result from putting in communication multiple computing nodes, i.e., hardware running the above two scheduling layers. Each of the computing nodes comprises some hardware and runs their independent OS, making necessary a further layer of scheduling and resource management for the overall system resulting from their interconnection. This layer uses a higher granularity unit of computation, called *job*, and the scheduler is called *job scheduler*. Jobs are macro units of work that can request few computing cores to thousands of nodes and last up to weeks of computing time. They can be a single application as a collection of processes running on one or multiple computing nodes, multiple applications working together, or a series of applications with dependencies, called workflow. A series of job descriptions, with information such as their arrival time, requested resources, requested runtime, assigned resources, runtime, is called *workload*.

This thesis focuses on High-Performance Computing (HPC) machines, nowadays implemented as massive parallel machines. In these machines, the large number of available resources motivates the introduction of a resource manager, with management and monitoring capabilities, usually coupled with their job scheduler in a unique piece of software, called distributed resource management system (DRMS) or distributed resource manager (DRM).

This software allows higher-level management of machine resources. It gives an interface to users to be able to reserve a specific number of resources, their characteristics, the requested time, and the jobs or workflow description that need to run. Job requests are usually organized in queues with priorities, and when the resources requested by the user are available, the job will be launched. Given the high cost of maintaining and managing HPC machines, the job scheduler's main tasks are: reduce the wait time for users, avoid starvation of jobs, and increase the load maximizing the number of allocated resources.

Job schedulers also give system administrators tools to control user requests, permissions, accounting, organizing users' requests, and used resources in groups, entities, and projects to achieve high-level HPC machines' management.

The job scheduler will be the focus of this thesis. The need for research on this topic is motivated by the growing complexity of HPC machines, in size and distribution in space, their power supply, the variety of hardware architectures, and workloads running on them.

Today's needs in the scientific research community shape the future of HPC. To the best of our knowledge, we resume them in the following three points:

Increasing data storage demand leading to increasing research in new memory solutions and technologies

Since the introduction of big data workloads in HPC, memory became the main bottleneck, known as the memory wall. Researchers have been trying to break this wall by introducing new memory technologies and coordinating them in layers, to achieve both speed and vast storage capacity.

New trends include developing high bandwidth data transfer interfaces like High Bandwidth Memory (HBM) and High Bandwidth Cache (HBC), and non-volatile memory
(eNVM) as an intermediate level for applications manipulating big chunks of data, i.e., a secondary DRAM.

Another trend is the development of fast network-attached storage (NAS), acting as network storage or cache for exchanging data between attached computing nodes.

Finally, New memory-centric chip technologies are emerging to solve the bandwidth problems. Near-memory computing combines logic and memory in the same integrated circuit, while in-data computing brings computation directly inside the memory.

Increasing computation power demand leads to research in high parallel computing units and energy saving

Hardware miniaturization plays an important role in increasing computing power and reducing power consumption for processing units. Today it arrived at a scale that is close to the physical limit, so new alternatives to silicon have been studying.

At the same time, classical CPUs are now coupled with parallel processing units, e.g., GPUs and many-core processors. These units can achieve higher parallelism and better efficiency compared to multi-core CPUs. Moreover, vector extension for CPUs allows applications to benefit Single Instruction Multiple Data (SIMD) architectures.

In the future, a crucial limit for the scalability of HPC machines is their power consumption. In HPC machines' design, the amount of available power is limited, so researchers are trying to reduce their energy and power consumption by using Dynamic Voltage and Frequency Scaling (DVFS) tuning, power-capping techniques, and thermal control.

Heterogeneous workloads with different characteristics lead to workload-specific and highly heterogeneous hardware

The previously described trends cannot satisfy the increasing size and complexity of HPC workloads.

Special-purpose architectures are increasing to solve specific workloads like machine learning, deep learning, computer vision, high-performance data analytics (HPDA).

Simultaneously, exascale competition and open hardware initiatives started a race in which private and public sectors develop their hardware. EPI [37] initiative in Europe is trying to design its chips and accelerators. In the same way, the USA, China, and Japan are working on in-house hardware architectures.

Computing nodes can mix those different hardware technologies leading to highly heterogeneous clusters. Multiple clusters are starting to be coupled under the same system, generating what we call *multi-cluster environments*. An example is the prototype machine developed in DEEP-EST European project [34], proposing several heterogeneous clusters collaborating under the same machine and managed by a unique resource manager. Each cluster has general-purpose CPUs and accelerators, presenting different performances and energy behaviors.

From a broader perspective, with the advance of memories and interconnections, multiple machines will likely be connected and exchanging workloads, resulting in machines with huge potential but high hardware diversity and very complex management of resources.

Regarding job scheduling and resource management, we identified three main gaps in the research for the job scheduling and resource management layer in HPC. Those three gaps are connected to each other in a pyramid, like Figure 1.2 shows.

1. *Awareness*. The described HPC layers are implemented as semi-independent layers, with each layer having little to null awareness of the others. Instruction schedulers give no feedback to the task scheduler, and task schedulers give no information to the job scheduler. We claim that the more each component is aware of the other components, the better they can integrate each other to work in the most optimized way. At the same

time, the schedulers are not aware of the characteristics of the running workloads, e.g., their performance and energy profiles, and which hardware components (CPU, memory, network) stress more. Being aware of the workload's characteristics can lead to better decisions of the job scheduler in the process of allocating resources that maximize the job's and system's efficiency.

2. *Dynamicity*. In contrast to the static, an evolving system can faster and better adapt to changes in the hardware and the workload. Dynamicity needs awareness, such as information related to hardware and software collected at runtime, during job execution. While this information is typically stored and analyzed offline, we claim that future schedulers must make meaning of them at runtime and react fast to optimize workload configurations and placement. In a dynamic system, the resource manager can detect the resources' underutilization and reassign these resources to the jobs that can better exploit them. Additionally, the jobs can be dynamic themselves and require more or fewer resources during the runtime, depending on the characteristics of the job phase.

3. *Automatization*. Current job scheduling frameworks are rather user-guided than
   system-guided. In future machines, users and developers should not have an in-depth knowledge of optimal architectures and optimal parameters to run their jobs. However, they should instead be concentrated on their jobs' functional parts, letting the system find the optimal configuration. Nowadays, automatization can be guided by data. Data, typically used only for offline analysis, is now unlocking increased

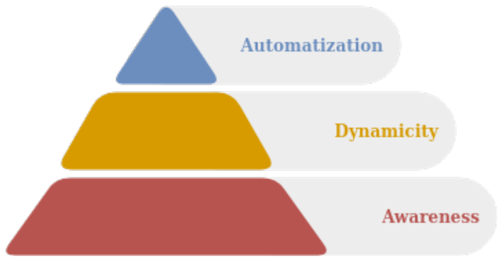


Figure 1.2: The pyramid of the gaps in the research.

awareness and dynamicity. Heterogeneous hardware increases the need for automatization to ease the user's effort to run workloads in increasingly complex systems.

More in the specific, we identified the following scenarios based on the identified scientific community needs and gaps in the research that we will address in this thesis:

1. Hybrid workloads: memory-bound and compute-bound workloads mixes make it difficult to make the most out of available hardware architectures. In both cases, some resources, either memory bandwidth or computing power, will be poorly utilized. Besides, emerging workloads, made up of simulations and data analytics, are optimized by starting the data analysis with in-between results: this can reduce computing time and wait time. It also serves as an early validation. For example, if early data does not pass the early validation, the simulation can be canceled, saving computing power and time. Finally, real-time workloads, differently from long batch jobs, need to run as soon as possible, even giving up some of the computing requirements.

2. Job scheduler complexity for hybrid workloads: the classical schedul-

ing approaches poorly perform and underutilize resources. For instance, static scheduling approaches do not mix hybrid jobs efficiently, as oversubscription often performs poorly. Also, static approaches have limits in the case of hardware failure and dealing with real-time workloads.

3. Hardware complexity and energy concerns: systems are growing in size and becoming more heterogeneous to accommodate the needs of hybrid workloads. New system architectures are emerging, like modular systems made up of multiple heterogeneous clusters that need specifically designed scheduling policies. Energy consumption became a concern, so new job schedulers need to tackle this issue by controlling and optimizing the system's amount of consumed energy.

For the above-mentioned scenarios, we propose three research questions:

1. Can we address new job needs by enabling malleable workloads with negligible overhead?

2. Can we use malleability at job scheduling level to efficiently accommodate and reduce the average system response time for new workloads mixes?

3. Can workload and hardware aware scheduling contribute to automatization and reduction of energy and response time in the growing heterogeneous systems?

This thesis answers by brings three main contributions, with the overall objective of improving the job scheduling efficiency, hardware efficiency, system utilization, and energy savings. Our work was integrated with well-known software and libraries used in production systems with a little additional effort needed. In particular we integrated our contributions with standard programming models like OpenMP[84], MPI[76], and research programming models like OmpSs[35]. We also implemented job scheduling policies in Slurm[9] workload manager, vastly used in HPC systems. To summarize, we present the main contributions, followed by an in-depth discussion.

- Implementation of Dynamic Resource Ownership Management (DROM) library for dynamic resource management of node's computing resources, allowing the job scheduler to dynamically change allocated resources at the job's runtime.
- Implementation of a malleable job scheduling Slowdown-Driven policy (SD-policy), automatically and dynamically redistributing allocated resources based co-scheduling and awareness of the job's slowdown.
- Implementation of a job scheduler energy-aware and workload-aware policy EAMC-policy, able to automatically adjust job's energy settings, and prioritize the most efficient hardware resources for jobs.

We first applied the concept of dynamicity to workloads. Jobs able to adapt to changes in resources dynamically are known as malleable. Malleability is proven to bring significant advantages to the system in terms of efficiency and response time [52]. However, it is challenging to implement due to a lack of awareness between HPC layers, standard malleable libraries for applications, and malleable job scheduling policies. We implemented the Dynamic Resource Ownership Management (DROM) library, enabling threads malleability and interaction with the job scheduler, making the job scheduler malleability-aware. While most solutions implement malleability between computing nodes, implying the movement of a high amount of data, we propose a library that implements fast malleability inside nodes by dynamically binding tasks to resources. The library is entirely transparent and integrated with modern programming models, making its use effortless for programmers. We analyzed the performance impact of using DROM with synthetic applications and applications from the neuroscience field. We observed a negligible overhead when changing the number of resources for the application at a reasonable frequency. We further evaluated different use cases of applications sharing nodes using DROM to partition available resources among processes and threads, and we observed lower completion time and better resource utilization compared to oversubscription.

Once we have malleable applications, we need a malleable-aware policy that takes advantage of this feature. This is our second contribution. First, we integrated the DROM library in the resource manager to take into account allocated resources dynamically. Then we design a Slowdown-Driven policy, SD-policy, that applies malleability automatically by shrinking jobs with a low slowdown to make room for new jobs, only if they will improve the predicted system slowdown. The policy shrinks jobs and uses a combination of node sharing and malleability to run new jobs.

We generated workloads that imitate modern users' behavior and jobs, mixing memory-bounded and CPU-bounded jobs. The policy works particularly well with those workloads since it allows better use of some poorly used computing resources assigned to memory-bounded jobs to run CPU-bounded jobs. In these scenarios, we observed a reduction in makespan, response time, and energy consumption compared to static scheduling.

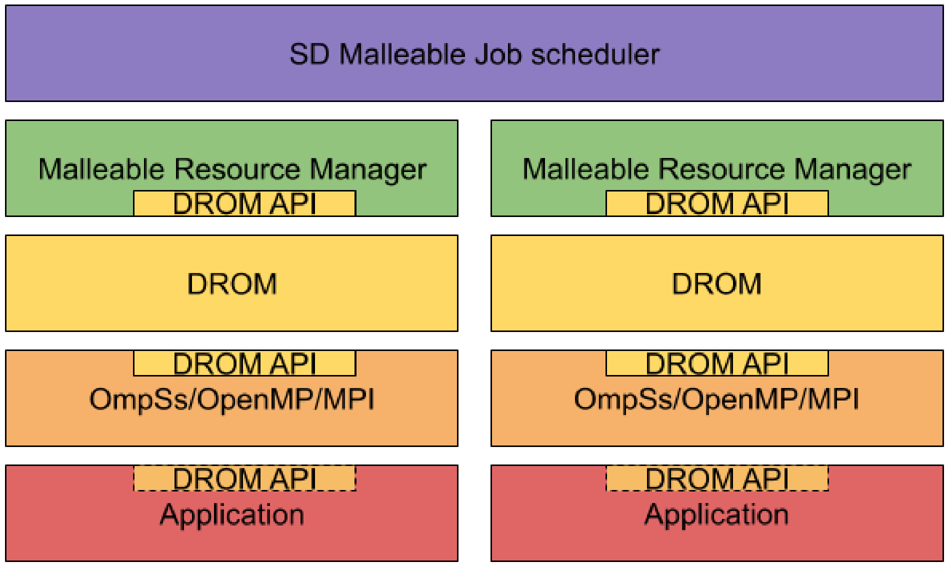


Figure 1.3: DROM plus Slowdown-Driven malleable scheduling policy software stack.

Figure 1.3 represents the first two contributions.

Finally, we tackle the lack of automatization in heterogeneous environments in hardware selection and configuration by developing an Energy and Workload Aware Job Scheduling Policy for Multi-Cluster Environments, EAMC-policy. This policy can automatically select the best frequency per job based on the previous job's energy and performance data and prioritize best performing hardware at scheduling time.

We integrated and extended different layers of the software stack to generate a holistic solution, as shown in Figure 1.4. We used energy collection libraries integrated with an application database, plus an interface for the job scheduler to gather predictions using different energy models. We extended the used energy models to produce precise predictions about the job's runtime, power, and energy consumption for different hardware. The solution is

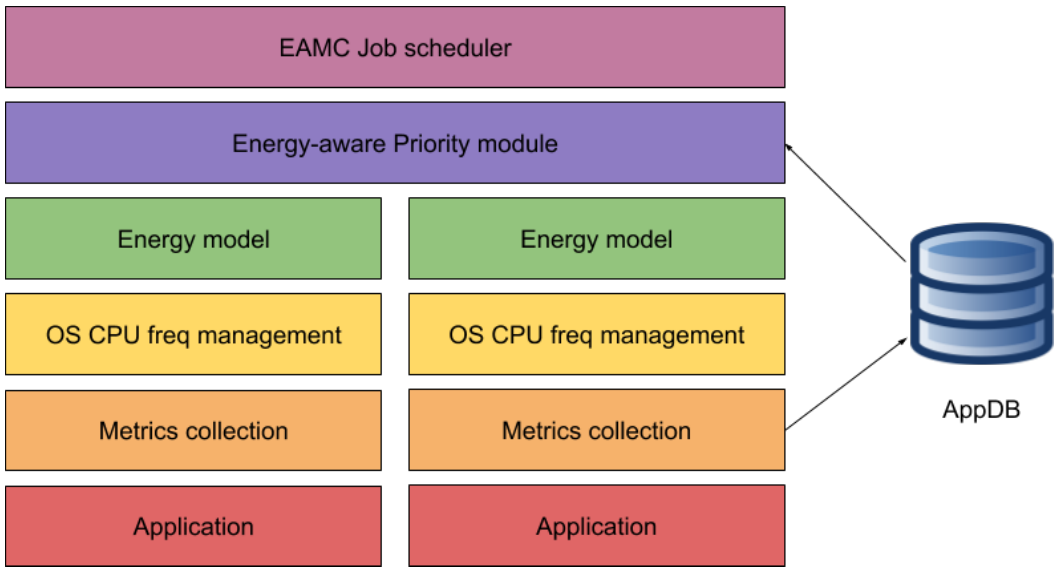


Figure 1.4: Energy Aware Multi-Cluster scheduling policy software stack.

transparent for the user, as we avoid requesting any hardware or application-specific information. We used the prediction to select the best frequencies and prioritize the more efficient hardware for each job.

Running EAMC-policy in a simulated heterogeneous environment with job data coming from real hardware produces a significant reduction of energy consumption and response time.

This thesis was evaluated using different methodologies, from application's analysis to analysis of workloads at the cluster level. Regarding application analysis, we used real-world applications coming from the physics and neuroscience fields.

For the evaluation of job scheduling policies, we used Slurm and Slurm Simulator. We created an environment to run Slurm as a job, without affecting system scheduler's configuration and need for administration privileges.

Slurm Simulator is a tool developed and previously used by the Slurm community. We put our effort on significant improvement of the precision and performance of this simulator. We also validated it and evaluated its precision. Besides, we also developed different testing environments and benchmark, scripts for the generation of synthetic and real workloads, and their analysis.

All this effort was published in articles and the tools are openly available, representing an important contribution for the research community, and even if not directly related with the specific contributions of this thesis, they are the base of all the work.

1.1 Thesis organization

Chapter 2 presents the background and related work. Chapter 3 presents our methodology for evaluating the proposed solutions, introducing novel methodologies and contributions on the available community tools. Chapter 4 and 5 are dedicated to the topic of malleability. In the former, we present DROM, a library that enables malleability for applications, while in the latter, we describe the Slowdown-Driven malleable policy. Chapter 6 presents our contribution to the energy and workload aware job scheduling, EAMC-policy. Chapter 7 concludes this thesis with insights on future work. Chapter A is an appendix containing information on how to download the code and reproduce this work partially or entirely. The authors wish you a good read.

"Each problem that I solved became a rule, which served afterwards to solve other problems."

René Descartes

2 Background and Related Work

We will start introducing the state of the art of job scheduling, the main job scheduling algorithms and the main research job schedulers. Then we introduce the concept of malleability, at applications and job scheduling level. Finally, we will introduce the hardware heterogeneity and the energy-aware job scheduling.

2.1 General Job scheduling

The job scheduling problem has been studied for half a century. Finding an optimal schedule is an optimization problem, with NP-complete complexity. The first book on the topic dates back to 1967[28], exploring mathematical models on the job shop scheduling problem (JSSP). Job shop scheduling is the class of optimization problems that minimize makespan for n jobs running on n nodes. In this class of problems, each job runs on a single node for a specific time. JSSP is different from the generic job scheduling problem because, in parallel machines, each job can request a variable number of nodes. This makes our problem a two-dimensional scheduling problem, scheduling in both time and space.

An important classification of job scheduling problems is: *online* and *offline* scheduling. In the offline scheduling problems, all the jobs and their characteristics are known in advance. In the online problem, jobs arrive over time, and the scheduler needs to iterate to find a new schedule when new jobs arrive. In this thesis, we study online scheduling problems since users' jobs can arrive at any time and are not known in advance. Some of the online algorithms are inspired by the offline job scheduling problems, and some others are specific to this kind of problem. Next, we present the main online scheduling algorithms.

Online job scheduling can be approached in multiple ways, using time-slicing (or time-sharing) approaches, space-sharing (or space-slicing approaches), or a combination of both.

Time-sharing

In the time-sharing or time-slicing approach, jobs are scheduled to run by using resources in specified time slots. If the time slot is not long enough to allow the job to finish, it is necessary to do a context switching and move the job's data in/out of the assigned processing unit to free it for another job.

Space-sharing

In the space-sharing or space-slicing approach, different processing units can be assigned to different jobs simultaneously, so resources are not assigned only to a single job.

Gang scheduling

In the Gang-scheduling approach, a group of tasks coming from the same job is scheduled simultaneously. This technique allows minimizing latencies of communications between tasks, thus reducing the response time of jobs and their makespan. If this technique is combined with a time-sharing approach, it will need coordinated context switching for the whole gang.

2.1.1 Job scheduling policies

Job scheduling policies can combine the described time and space approaches to solve the job scheduling problem efficiently.

Online job scheduling problems manage jobs in a simple or sorted queue and tries to schedule the jobs one by one. The main approaches used in the state of the art for these problems are:

- Longest Jobs First (LJF): it favors large jobs by dividing the arrived jobs into two categories and favoring the ones classified as large jobs.
- Shortest Jobs First (SJF): it is the same as LJF, but in this case, the small jobs are favored.
- First-In-First-Out (FIFO): it accumulates jobs in a queue in the arrival order, and the scheduler will try to schedule jobs in the same order.

- Priority Queues: It is the generalized case of FIFO, where a different priority is calculated for each job based not only on the arrival time but, e.g., on the requested time, number of nodes, or type of resources.

- Backfill: it tries to schedule jobs in an out-of-order fashion to achieve additional efficiency. This is the primary approach used nowadays in HPC machines. Backfill was firstly proposed by D. Lifka[63]. After the first proposal, different optimizations of the problem were researched, making backfill a family of policies. The basic idea of backfill is to increase the utilization of resources by starting jobs in an out-of-order fashion only if they do not delay jobs with higher priority in the job queue.

Online scheduling problems can be further divided into the following categories: *preemptive* job scheduling and *malleable* job scheduling. Preemptive job scheduling allows stopping and resuming jobs. This is particularly useful when implementing Quality of Service (QoS), e.g., prioritizing jobs that need low latency.

Malleable job scheduling is the scheduling problem in which jobs are not statically allocated, i.e., resources assigned to the job can dynamically change at runtime. This class of problems has recently become popular for the *dynamicity* property and its high potential in solving some of the HPC machines' issues.

2.2 Malleable job scheduling

Malleability has been studied since many years ago in the context of scheduling, with the definition and study of Malleable Parallel Task Scheduling (MPTS) problem. The theoretical research shows its potential benefits [67] [101] [78].

These works mainly pick the number of resources that best improves the performance of the parallel task based on a model of its performance given at schedule time. Once this number is set, the MPTS problem can be solved with a *non malleable* approximation algorithm. However, it oversimplifies the MPTS problem as the number of allocated resources do not change at runtime, since it is set only once by the performance model.

More recently, online scheduling approaches executed in simulated environments [52] showed the potential benefits of malleability on job's response time, by using fair process distribution and shrink and expand operation on jobs that expand the number of used processors.

Several proposals try to exploit malleability in non-simulated environments. Non simulated environments require not only malleable scheduling policies but a whole software stack infrastructure that supports them. In fact, to avoid unnecessary user knowledge and overhead for the developer, malleability needs to be transparent by being implemented in libraries and programming models. Regarding job schedulers, some of the production DRMS [9] [6] implement API for supporting flexible jobs allocations, i.e., adapt job allocations if the user asks for changes in the required resources. While this is the first step towards malleability, there is no scheduler implementing malleable job scheduling policies. The critical difference between those two concepts is that the malleable scheduler decides whether to increase or shrink jobs' allocations based on an internal decision not demanded by the user.

In the recent years the research community proposed some practical implementations of full-stack malleable scheduling solutions.

Utrera et al. [104, 103] use folding techniques to shrink and expand jobs plus FIFO-malleable and backfill-malleable policies that uses co-scheduling

	When	
Who decides	At Job submission	At runtime
User	Rigid	Evolving
System	Moldable	Malleable

Table 2.1: Jobs classification

and oversubscription to start MPI jobs when no resources are available.

Kale et al. [56] implement malleable and evolving jobs on the top of Charm++ and defined a scheduling policy based on equipartition. When a new job arrives, the scheduler recalculates the number of processors allocated to each running job. Prabhakaran et al. [89], using Charm++ malleability, implemented shrink/expand operations in a production scheduler, improving the scheduling strategy over Kale's work based on equipartition. Evaluation is performed on workloads combining rigid, evolving, and malleable jobs.

Chadha et al. [22] use an MPI extension and SLURM to build a malleable job scheduling policy guided by a ratio between the number of processes and the compute time. They use a two queue system to separate rigid and evolving jobs and use malleability if a new job cannot start.

2.3 Malleable applications and libraries

To introduce the concept of malleability, we use a classification of jobs based on their possibility to dynamically expand and reduce the allocation of their resources, done by Feitelson in [38, 42]:

- Rigid: jobs that require a certain predefined encoded number of processors.

- Moldable: jobs that allow the number of CPUs to be set at launch time, but not thereafter.

- Evolving: jobs with changing requirements, e.g., a sequence of serial and parallel phases. In this case, the application itself demands the change.
- Malleable: jobs that can adjust to changing allocations at runtime. This allows the most flexibility to the system.

Malleable applications have the highest degree of flexibility, making them more efficient for a number of scenarios, including:

- Hardware failure: malleability can be used in combination with resiliency solutions to avoid data loss in the case of hardware failure. While fault-tolerant interfaces recover the application in the case of partial or total failure, malleability can ease the application's adaptation to keep running on non-failed resources.
- Underutilization: it can happen when some resources of the system are left unallocated to jobs or scarcely unused by applications, thus unexploited. In the former, malleable job allocations allow re-adapting running components to use unused resources, expanding their job allocation, or shrinking them to make space for new jobs. In the latter a similar approach can be used with jobs not fully exploiting resources. Malleability gives the job scheduler much more possibilities in choosing and re-adapting job allocations to increase utilization.
- Responsiveness and Quality of Service (QoS): if a system is full and an urgent or high-priority job arrives, the system has no other choice but to either wait for some resources to be available or use preemption to run

the high-priority one by stopping or killing a running job and then continuing the execution or restarting it. With malleability, the job scheduler can shrink some running jobs to run the high priority job instead of completely stopping them.

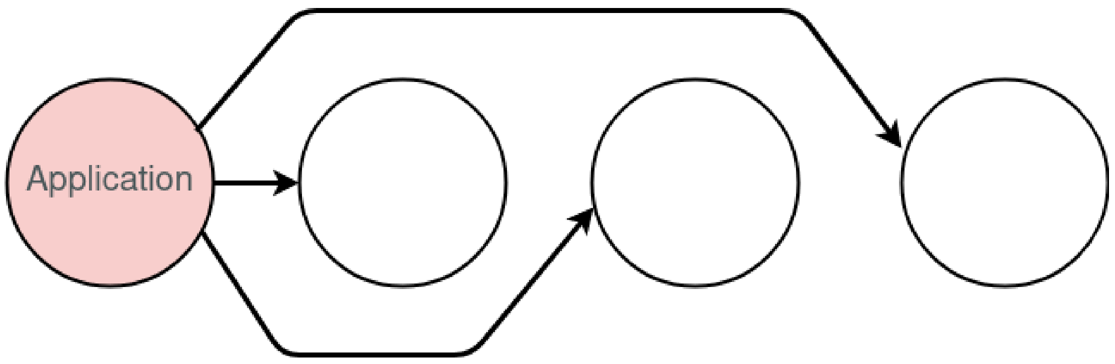
As already mentioned, having flexible applications has been shown to benefit system performance in reducing job response time and increase resource utilization [52]. While from a theoretical point of view, this statement is widely demonstrated and accepted, from a practical point of view, the constraints due to hardware and software limitations make adoption complicated.

We identified two major problems that the research community needs to face to ease the adoption of malleability in HPC systems:

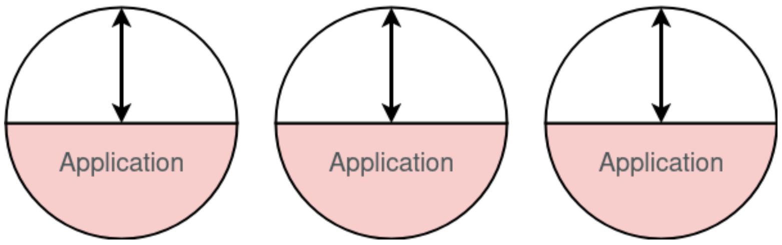
1. data movement: while it is relatively easy to move the code, it is much more expensive to move data, especially when it comes to big chunks, and data movement between slow interfaces, e.g. between computing nodes. Moving computation is much more affordable at the current state.

2. implementation complexity: it is a key feature for the adoption of malleability the avoidance of any overhead for users, hence malleability needs to be transparently implemented in programming models and system libraries. Unfortunately those were not created with this purpose in mind, and lack this feature in their design.

Based on the HPC machines physical organization, we further divided malleability into two categories, as shown in Figure 2.1:



(a) Horizontal malleability (inter-node).



(b) Vertical malleability (intra-node).

Figure 2.1: Horizontal and vertical malleability. The first involves changing the number of computing nodes, the second involves changing the used resources inside the computing nodes.

- Inter-node or horizontal malleability applies malleability to the number of used nodes by the application, involving new process creation and migration, plus data movement through the network interface.
- Intra-node or vertical malleability can change the number of allocated resources to the job inside the already used nodes. This case can involve creating new processes or threads, eventually invalidating part of the cached data.

2.3.1 Horizontal malleability: MPI

As precedently said, horizontal malleability needs data movement between nodes. MPI [76], the de-facto standard for multi-process communication, is used to spawn an application on multiple computing nodes. It gives API for process-to-process communication and group communication. MPI was designed as a static model, therefore the initial design did not include any API for malleability. Starting from version 2 of the standard, MPI introduced a set of API to spawn new processes through *mpi_comm_spawn()* directive. While the API allows spawning new processes at runtime, in general, the data redistribution due to using new computing nodes is delegated to the application.

Several proposals try to solve this issue in different ways. We can group them into online malleability and offline malleability approach. In the offline approach, the application is momentarily stopped from its execution, its state is saved, then data and code are moved to new nodes, and finally, the application restarts the execution from the saved state. Proposal using this solution has a simple implementation, with few changes in the MPI standard needed, but the approach is generally less performant than the online approach.

In the online approach, the application uses MPI APIs to spawn new processes, and then it connects to them, sending necessary data and continuing the execution without the necessity to stop and restore the application.

Given the high latency of data transfers between computing nodes, we decided not to further investigate horizontal malleability. At the moment of writing, new memory technologies are being developed that can reduce the overhead of data movement. Those technologies, e.g., Non-Volatile Memories (NVM), Network Attached Storage (NAS), Network Attached Memory

(NAM), can speedup data movement, giving a push to horizontal malleability.

2.3.2 Vertical malleability: OpenMP and OmpSs

By vertical malleability, we mean the capability of applications to mutate inside a computing node. In HPC, multi-core management is obtained through multi-process and multi-threading. Multi-threading is eased by programming models like OpenMP [84] and
OmpSs [35], besides already mentioned MPI.

While, for simplicity, using a single programming model like MPI allows managing multi-node applications and at the same time cores inside nodes using MPI processes, threads are lighter and faster than processes. Utrera [102] et al. use moldability plus folding techniques to adapt the job to available resources using oversubscription. Cera et al. [21] use a similar approach, dynamically changing the operating system CPUSETs for MPI processes. In this case, the number of processes is fixed, while the resource mapped to the process changes at runtime. This approach is easy to implement but can lead to unnecessary performance loss due to the context switching overhead.

OpenMP and OmpSs are programming models based on multi-thread libraries and allow light and fast thread management. OpenMP and OmpSs are good malleability candidates since they allow selecting the number of necessary threads at launch time and runtime.

This feature is just a small piece of the puzzle for obtaining malleable applications and malleable jobs, as there is no integrated way to control in which physical resources the threads will run, and there is no way for schedulers to ask the applications to change used resources. Luckily, the OS gives instruments to solve the first issue, with API that allows setting and dynamically changing the mapping of threads or processes to CPU cores. Regarding the second issue, filling this gap is part of this thesis, and it will be the topic of Chapter 4.

2.3.3 Others

Charm++ [55] is an object-oriented programming language. It divides the application transparently in fine granularity blocks, called charms. Charms can be dynamically created, supporting horizontal and vertical malleability. Charm++ is a different programming model, and developers need to rewrite their applications using this new paradigm, which makes this solution unacceptable for the thesis. Our thesis studies transparent solutions for malleability with no efforts for users and developers.

AMPI [50] is a Charm++ interface that enables MPI applications to run over Charm infrastructure. While this solution solves the rewriting applications' issue, it only partially supports MPI standard, and it is not compatible with other parallel programming models like OpenMP and OmpSs.

2.4 Energy-aware scheduling

HPC machines are extremely power demanding, such that the United States' Department of Energy set a goal of 20 MW power consumption for an exascale machine. In recent years research and industry put a high effort investigating how to stay under this constraint. Simultaneously, there is a significant shift from classical performance-oriented to an efficiency-oriented research mentality, with increasing awareness of green and eco-friendly concepts. Top500 [99] list now is accompanied by the Green500 [98], classifying most energy efficient machines. New metrics are proposed to draw up the list, i.e., TGI metric [94], as an alternative to GFLOPS/W, it evaluates non-computing devices, e.g., memories and disks.

Research in energy consumption moved in both hardware and software directions.

Regarding hardware techniques, miniaturization plays a vital role in reducing power consumption, but this process is getting more and more to the silicon physical limit, and alternative chemical elements are being studied. Meanwhile, Dynamic Frequency and Voltage Scaling (DVFS) became a popular solution to reduce consumption by dynamically modifying the processor's voltage and frequency. This technique is beneficial when trying to control the maximum power consumption or when dealing with memory-bounded applications. In the latter case, higher frequency does not help to reduce the runtime, bounded by the memory wall.

Regarding software techniques, we can classify them in the following categories, from a low-level to a high-level perspective of the system:

- Low level and OS interfaces for DVFS, collection of power metrics.
- Energy modeling and estimation for hardware and applications.
- Thermal, power, and energy-aware task scheduling in multi-core processors and GPUs, node-level powercapping.
- Thermal, power, and energy-aware job scheduling and resource management at cluster level: two main techniques are used to limit power consumption:

- overprovisioning: it is a strategy in which more hardware than the one that can be powered is bought, and part of it is selectively shut down or powered on based on the necessity.
- powercapping at the node, job, and system level.

Regarding energy-aware job scheduling solutions, we identified that most of the
research [69] mainly focuses on power scheduling and controlling, with few considerations that most of the times, minimizing power does not mean minimizing the energy.

Furthermore, as suggested by recent research [30], this whole variety of solutions miss automatic ways of setting energy configuration for the systems and user's activity. At the current state of the art, it is the user's responsibility to specify the type of resource needed, the number of resources, and the best energy settings for the submitted job. This complexity is not acceptable in heterogeneous multi-cluster environments, where a number of computing nodes can be chosen, each one with different characteristics, resulting in confusion and unnecessary knowledge needed for users.

This suggests that the system should autonomously make decisions, relying on users as little as possible. Job schedulers need to estimate information about the jobs, such as their runtime, performance, energy consumption for each hardware resource that can run it, and use this information to improve the scheduling.

Regarding energy-aware job scheduling, different proposals are contributing in a variety of ways combining described techniques. Some policies are based on modeled manufacturing and assembling variability [24], or temperature in different points of the system [77].

Sarood et al. [92], Barry et al. [61] explore overprovisioning, or powercapping [13] [90].

Auweter et al. implemented two policies: energy-to-solution and best-runtime, to select the most suitable frequency for jobs [8].

2.5 Distributed Resource Management Systems (DRMS)

It is commonly used in production systems to name Distributed Resource Management Systems (DRMS), which usually includes the job scheduler, tools for resource management and monitoring, job and user control, and monitoring. We will now present the main DRMSs, and in particular Slurm, vastly used in this thesis.

2.5.1 Slurm

Slurm[9] is a resource and workload manager software, free and Open Source, explicitly designed to satisfy the demanding needs of high-performance computing.

Slurm architecture

Slurm implements a master-slave cluster management architecture, as shown in Figure 2.2. At the top is a redundant pair of cluster controllers. These controllers serve as the managers of the compute cluster and implement a management daemon called slurmctld. The slurmctld daemon provides monitoring of computing resources, but most importantly, it maps incoming jobs to underlying compute resources. Each compute node runs a daemon called slurmd (node manager). The slurm daemon manages the node on which it

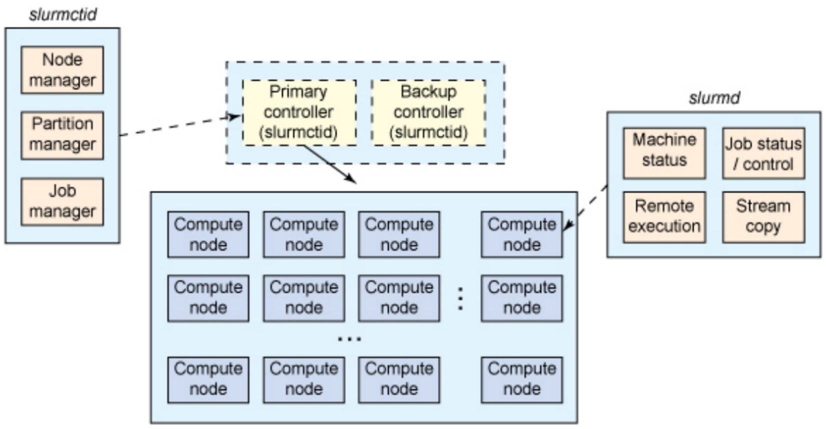


Figure 2.2: Slurm architecture[9]

Slurm kernel					
Authentication Plugin	**MPI plugin**	**Scheduling plugin**	**Select plugin**	**Task plugin**	**Priority plugin**
munge	pmix	backfill	cons_res	affinity	multifactor

Table 2.2: Slurm code structure is organized in plugins, to ease the code extension. Main used plugins are shown.

runs. It monitors the tasks running on the node, accepts work from the controller, and maps tasks on the cores within the node.

The use of optional plugins provides the features needed to satisfy the demands of HPC centers. Some of them are shown in Table 2.2, where on the top, we can see the plugin's generic name, and on the bottom, the used implementation in this work.

- Select plugin: it selects resources for each job. It can run at different granularity, thread, core, socket, and node level.
- Task plugin: it manages nodes' resources and their binding to processes and threads.
- Scheduling plugin: it implements backfill as a separate scheduler that runs controlled by a timer.

- Priority plugin: it assigns a priority to each job based on a number of factors, e.g., the arrival time, the number of nodes, the requested time, and partition.

Slurm is vastly used in production machines and by the research community. This motivated us to adopt this DRMS as a working environment. We opted for a full DRMS and not an experimental environment because our objective is to give a close-to-production implementation of our research. This choice presents different positive and negative repercussions.

On one side, Slurm allows saving the time to implement a prototype environment for the implementation and testing of the thesis's work. Also, by implementing the research in production software, the research can be brought to production faster. Finally, using Slurm or a production DRMS in the evaluation of scheduling policies has the advantage of making the evaluation closer to the real behavior in a system. A prototype job scheduler does not have the necessary complexity and scalability features to run on large scale systems.

On the other side, implementing the research on Slurm can be challenging work. Its code is vast and complex, sometimes redundant, challenging to understand, edit and test. Moreover, many features were added as an extension of a code whose design was not contemplating them, and even a small change in the Slurm logic can become a pain, requiring a high amount of effort to study, edit, and bug-fix.

2.5.2 Other DRMS

Flux [6] is a hierarchical DRMS designed to solve current exascale issues in job scheduling and resource management. Flux instances can run inside Flux or other DRMS instances, creating multiple scheduling layers to address scalability, offering at the same time API to manage instances and jobs running inside them. Flux is made up of a core and a scheduler that can dynamically load different scheduling policies.

OpenPBS [97] is fast, scalable, secure, and resilient DRMS for HPC environments. It can be extended by using its plugin structure; it is Open Source and part of the OpenStack HPC software infrastructure.

2.6 Job scheduler simulators

A scheduling simulator is a software that reproduces job scheduling behavior by simulating workload execution without actually running the jobs. This piece of software is handy when it comes to analyzing the performance of job schedulers implementation and its comparison to other implementations. We divide the job scheduling simulators in the state of the art in two categories: *general job scheduler simulators* and *implementation-specific job scheduling simulators*.

2.6.1 General job scheduler simulator

General job scheduling simulators are simulators that are not based on a specific DRMS, but they only simulate the job scheduling and placement. Being simpler allow them to run faster, but they do not simulate all that comes together with the DRMS.

There are plenties of general job scheduler simulators [36] [57] [18] [19]. These simulators are suitable for a theoretical evaluation of scheduling algorithms, and they usually include configurations for modeling the different platforms, partitions, hardware, energy, and networks. General job scheduler simulators lack enough details, characterization of parameters included in production software, and the software architecture that a system administrator might want to optimize to get the most out of a particular machine. Batsim presents some accuracy tests, done after developing an adaptor between Batsim and OAR [82] to allow using OAR schedulers into the simulator, and a submission system that reads and sends 800-jobs requests to OAR operating on 161 nodes. This methodology is hard to reproduce since few workload managers permit decoupling the scheduler code from the rest, and simulation is limited to the scheduler itself, not the whole software infrastructure, including other code parts and implementation-related parameters. Simbatch presents accuracy tests by implementing models for simulating batch schedulers and running 100 tasks on five node tests with OAR. In this case, it is not clear up to which detail the models are representing real job schedulers. In conclusion, standard job scheduling simulators are a good start for evaluating a scheduling algorithm, but they give only little hints about how they will perform on a real machine, not representing an extensive set of tools for system administrators.

2.6.2 Implementation-specific job scheduling simulators

Implementation-specific simulators keep all the details of a specific job scheduler, maintaining their architecture, reusing their source code, and giving the possibility to system administrators to try different configurations and algorithms with the objective of tuning system performance. We found four simulators in this category; the first is Qsim[96], an event-based simulator for Cobalt[7], specific job scheduler for Blue Gene systems. The second is Moab[27] scheduler that implements a simulator mode, in which the user can interact and control simulated time, but it is proprietary software. Flux [6] is a Resource Management framework that includes a simulator in its code, but there is no information about it included in the publications and documentation. There is no published evaluation of the consistency and the accuracy of any of these simulators. The last one is the Slurm Simulator, initially developed at Barcelona Supercomputing Center by A. Lucero [66], and based on Slurm version 2. In its second version, Trofinoff and Benini [100] from CSCS updated it to Slurm version 14 and brought a series of improvements. Finally, G. Rodrigo [91] improved the synchronization and the simulator speedup, together with a set of tools for workload generation, scheduler configuration, and output analysis.

Finally, Simakov et al. [93] attempted to validate their Slurm simulator version. Simakov simplified the simulator structure, serializing the code on a single process, the Slurm controller, that can be compiled as a simulator. While he significantly reduced the amount of executed code and complexity, he lost some of the features that Slurm can offer, e.g., multithreading, some of the plugins used inside Slurm node daemons, and the Slurm original architecture.

3 Methodology

Our methodology is the combination of single-application or workloads real-machine runs, and workloads trace-driven simulations. Depending on which layer of the software stack we are evaluating, we use one or another approach.

3.1 Applications and workloads real-machine runs methodology

Real-machine run is an accurate methodology that permits evaluating applications' behavior in detail. Whenever a job scheduling policy or a library affects applications' performance, it is recommended to evaluate and model the impact with this methodology. On the other side, real-machine runs are appropriate to analyze from one to a few hundred jobs for a total workload of up to two days on less than a hundred nodes. We further divide this approach into two categories: application analysis and workload analysis.

3.1.1 Application's analysis

When analyzing one or a few applications' performances, we instrument and collect a trace of the code behavior, e.g., programming models, libraries, and function calls, plus the hardware counters. We used the following tools for application analysis: Extrae [64] to samplea and extract traces, and Paraver [88] to visualize collected traces, presenting metrics such as:

- Runtime
- Instructions per cycle (IPC)
- Cycles per microsecond: processor cycles per microsecond dedicated to a specific thread
- Cycles per instructions (CPI)
- Memory bandwidth (GB/s)

3.1.2 Workload's analysis: Slurm Environment as a Job

To evaluate job scheduler performance on workloads running on real machines, this methodology allows installing and testing a job scheduling policy locally, without affecting the system level sofware. We install the DRMS in the user space and we run it as a job inside the system DRMS.

In this case, we focus on metrics that represent the total workload performance, rather than single application's, such as:

- Makespan: the time between the first job's start time and the last job's end time.

- Average wait time: the average of job's start time minus submission time.
- Average response time: the average of job's end time minus submission time.
- Average slowdown: the average of job's response time divided by its runtime.
- Total consumed energy: calculated as the sum of job's consumed energy or the whole system's consumed energy.

For the analysis of workloads, the main analyzed metric is the average response time and the slowdown, which measures how fast the system capable of completing user's requests. The main difference between the two metrics is that the slowdown, normalized by the job's runtime, gives the same importance to short and long jobs, while the response time doesn't.

We consider the makespan a secondary metric, since this metric could be easily affected by the simulation of the last jobs of the workload. If one long job running close to the end of the simulation extends the simulation, the makespan would be affected but the increase is not relevant for the evaluation. On the other side, we keep track of this metric to ensure it maintains acceptable values.

We implemented this methodology by creating an environment that launches a Slurm instance inside a Slurm job allocation. The minimum amount of nodes required for this job is two: one node is reserved for the Slurm controller, the rest of the nodes are computing nodes that can run jobs. The maximum amount of usable resources and time depends on the system DRMS configuration for the job's size and duration.

The environment is made up of different components, resumed in Figure 3.1:

- Job scheduler: a local installation of the job scheduler implementing a scheduling policy to test, in our case Slurm.
- Job scheduler configuration template: a template configuration file for the job scheduler. The template contains keywords that can be replaced with specific configurations, e.g., the number of nodes, their specifications, the nodes lists, the partitions, the controller hostname, the user name, the backfill parameters, and for the other plugins and scheduling features.
- Startup and finalize scripts: those scripts start the environment by setting necessary environment variables and starting controller and daemon nodes for the DRMS. i.e., Slurmctld and Slurmd daemons.
- Workload to jobs converter: this script maps workload traces files to batch jobs submissions using a time model that approximates the job's runtime in the trace by selecting parameters for applications. Applications are picked with specific distributions from a set of modeled applications. Used applications are described in Chapter 4 and 5.
- Batch job submission script: This is a template for a batch job. It accepts the binary to run and its parameters, together with batch script parameters, i.e., the number of nodes, the number of processes per node, and the number of threads per process.

The developed environment is Open Source and available at the BSC-RM github repository [32].

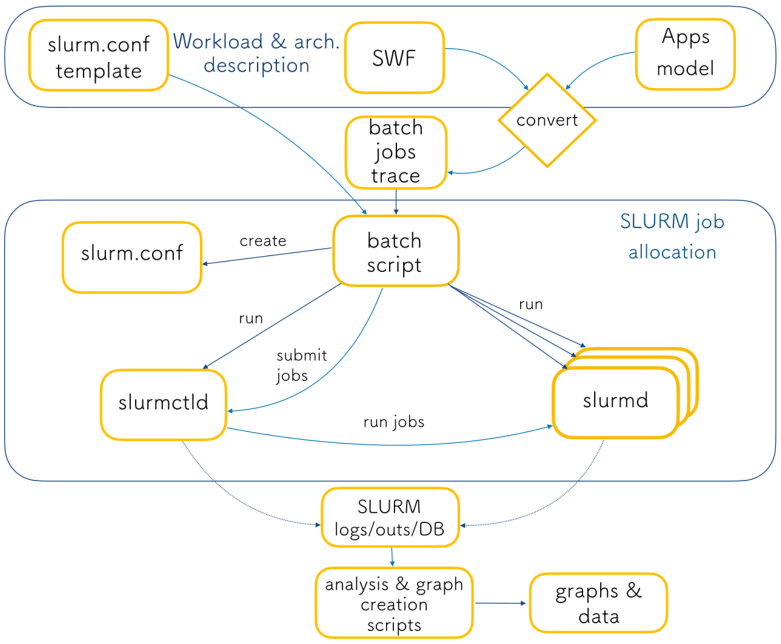


Figure 3.1: Slurm Enviroment as a Job.

3.2 Workload simulation methodology

This methodology allows evaluating job scheduling performance overcoming the limit of running applications on a real machine, while maintaining an acceptable precision. Based on job scheduling simulators, it simulates historical workload traces or synthetically generated workloads, up to months or years and hundreds of thousands of jobs, therefore multiple times faster than real-machine runs. Extracted metrics are generated over the simulated execution and are already described in the real-machine runs approach.

Since we use Slurm to implement and evaluate most of our work, we decided to use Slurm specific job scheduling simulator. Slurm has no official simulator, but the community developed it starting from some of its debugging features. Several years ago, the first version of the Slurm Simulator was created by a Slurm system administrator from Barcelona Supercomputing Center, A. Lucero [66]. Trofinoff and Benini [100], from CSCS, updated the simulator to Slurm version 14 and brought a series of improvements over it. G. Rodrigo [91] improved the synchronization and the simulator speedup, together with a set of tools for workload generation, scheduler configuration, and output analysis.

As none of these simulators satisfied our needs regarding accuracy and consistency, we took Rodrigo's simulator, and we improved it. A paper was published[4] describing the process of fixing the faulty code, optimizing simulation's speed, and evaluating its accuracy. In the same document, we present different scenarios where the Slurm Simulator can help evaluating a Slurm installation, e.g., by comparing different scheduler configurations.

As shown in Figure 3.2, the Slurm Simulator receives as an input the standard Slurm configuration file, *slurm.conf*, that allows for the specification of the system architecture, as well as job scheduler details such as scheduling and selection policies, and a trace file in a binary simulator's format. We have provided a converter from Standard Workload Format (SWF) [23, 40] to simulator's trace format to enable the simulator to use a vast amount of existing real-machine and modeled logs in the online repositories such as Feitelson's [39]. The Slurm simulator generates standard Slurm outputs, such as Slurm controller daemon's and Slurm daemon's logs, job completion log, and Slurm database files. We developed a set of scripts that converts the output in Comma-separated values (CSV) format, extracts the presented metrics, and shows them in different formats with charts.

Figure 3.2 shows three main components, i.e., three processes of the Slurm simulator, Slurm simulator's manager, Slurm controller daemon, and Slurm daemon.

- *sim_mgr* is in charge of controlling simulated time, i.e., it increments it by one second in each simulator's iteration. Also, it reads the input trace and submits the jobs to the Slurm controller when its arrival time is reached. The job submission is made using Slurm's API *sbatch*.
- *slurmctld* is a standard Slurm controller daemon, and all the core functions of the controller, such as job scheduling and selection policies, are the original Slurm code.
- *slurmd* is a simplified Slurm daemon since the jobs' execution is not simulated. It receives the job duration from the Slurm controller, sets the job's end time as a future event, and notifies the controller when the job's end time is reached. One *slurmd* process is in charge of all the nodes.

A wrapper for time functions returns simulated time instead of real-time.

Our work on the Slurm Simulator represents an important contribution of the thesis, and it includes the following:

- We removed the random variation from the simulator and made our simulator deterministic across multiple runs for the same input and set-up.
- We improved the accuracy, i.e., lowered the deviation of the simulator's system metrics values from the real-machine ones from the previous 12% to at most 1.7%.

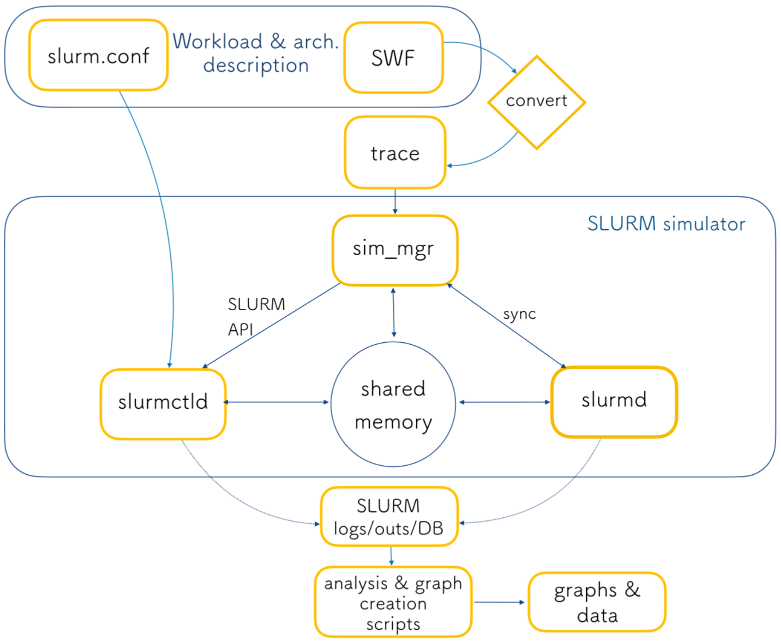


Figure 3.2: Slurm simulator's structure.

- We improved the simulator's performance by 2.6 times.
- We presented our methodology for the evaluation of the Slurm simulator on the real machine.
- We ported the simulator to the latest Slurm version at that time, 19.
- We implemented converters between Standard Workload Format (SWF) [23, 40] and the simulator's input trace and between Slurm's completion log and SWF.

Slurm Simulator is Open Source and published at BSC-RM GitHub repository [70].

"The ability to adapt is what survival is all about."

4
Enabling malleability with DROM library

In the design of future HPC systems, research in resource management shows an increasing interest in more dynamic control of the available resources. It has been proven that enabling the jobs to change the number of computing resources at run time, i.e., their malleability, can significantly improve HPC system performance. However, job schedulers and applications typically do not support malleability due to the common belief that it introduces additional programming complexity and performance impact. This paper presents DROM, an interface that provides efficient malleability with no effort for developers. The running application is enabled to adapt the number of threads to the number of assigned computing resources in a completely transparent way to the user by integrating DROM with standard programming models, such as OpenMP/OmpSs, and MPI. We designed the APIs to be easily used by any programming model, application, job scheduler, or resource manager. Our experimental results from two realistic use-cases analysis, based on malleability by reducing the number of cores a job uses per node and jobs co-allocation, show the potential of DROM for improving HPC systems' performance. In particular, the workload of two MPI+OpenMP neuro-simulators is tested, reporting improvement in system metrics, such as total run time and average response time, up to 8% and 48%, respectively.

4.1 Introduction

As early described, the HPC software stack consists of different layers, from parallel runtime to the job scheduler, each one responsible for a specific task. From a developer perspective, it is common to use different programming models to ease the programming on multi-core and multi-node machines. Those models can hide the low-level architectural details, and at the same time, extract the maximum performance from the systems.

From a system perspective, the job scheduler's objective is to maximize computing resources' efficient utilization. However, improving the system efficiency is, typically, not well accepted by users and application developers since their only objective is to speed up their application even if some of the resources are left underutilized. We claim that these two objectives must coexist and that cooperation between the different stack layers is the way to reach this goal.

From a workload perspective, new hybrid workloads combine memory-bound and compute-bound requirements. Each piece usually exploits only part of the available resources in the computing nodes, making the scheduling of jobs to exclusive resources inefficient. This is the example of the Human Brain Project, where neurosimultors, compute intensive applications are couple with small real-time analytics that analyze partial data obtained by the simulations. We claim that efficient node sharing could bring improvements in the utilization of hardware resources.

We propose to provide resource managers with more tools that will give them a dynamic control of resources allocated to the application and a particular feedback about the utilization of these resources transparently to users and developers. In this chapter, we extend the DLB [44] library with a new API designed to be used by the resource managers. This new API is presented as a transversal layer in the HPC software stack to coordinate the resource manager and the parallel runtime. We call this API *Dynamic Resource Ownership Management (DROM)*. DROM has been integrated with well know programming models, i.e. MPI [76], OpenMP [84] and OmpSs[35] and with the Slurm [9] node manager.

By integrating DROM with the above programming models, the API will work transparently to the application and developers. By integrating the API with Slurm, we enable efficient co-scheduling and co-allocation of jobs. This means that jobs are scheduled to share computing nodes by dynamically partitioning the available resources, improving hardware utilization and job response time.

This chapter presents the following contributions:

- Definition of DROM, an API that allows cooperation between any job manager and any programming model.

- Integration of DROM with Slurm node manager for effective resources distribution in the case of co-allocation.

- Integration of DROM with MPI, OpenMP and OmpSs programming models.
- Evaluation of DROM with real use cases and applications motivated based on needs in the Human Brain Project (HBP) [51].

The rest of the chapter is organized as follows: Section 4.2 presents the related work, Section 4.3 describes the DROM API, Sections 4.4 and 4.5 present the DROM integration with programming models and Slurm. Section 4.6 shows the experiments done to validate the integration and demonstrate the potential of this proposal, and finally Section 4.7 presents the conclusions and future work.

4.2 Related Work

Recent research shows an increasing interest in the development of interfaces and programming models that can enable malleability for applications.

Several studies propose malleability based on MPI [76], that allows, in different ways, to spawn new MPI processes at run time or use moldability and folding techniques [102]. These approaches are limited by the inherent program data partition between processes. Data partition and redistribution are application dependent, so they need to be done by application developers. Furthermore, data transfer among nodes has a high impact on performance, making malleability very costly, especially when using checkpoint and restart techniques. To support automatic data redistribution, MPI is usually constrained, e.g., malleability is only available for iterative applications using split/merge of MPI processes [68], or master/slave applications [26]. Martin et al. [74] try to automatize data redistribution, but only for vectors and matrices.

Recent work includes an effort on Charm++ [49] programming model to support malleability. Charm++ allows malleability for applications by implementing fine-grained threads encapsulated into Charm++ objects. This solution is not transparent to developers, i.e., they need to rewrite their applications using this programming model. Adaptive MPI [50] tries to solve this issue by virtualizing MPI processes into Charm++ objects, partially supporting MPI standard. Charm++ lacks a set of API that would allow communicating with the job scheduler because malleability features were studied for load balancing purpose. There was an effort to implement a Charm++ to Torque [89] communication protocol to enable malleability, but they are not comparable with DROM because DROM gives generalized APIs that can serve to communicate with any job scheduler or programming model.

Castain et al. in [20] presented an extensive set of APIs, part of PMIx project, including the job's expanding and shrinking features. It attempts to create standardized APIs that applications can use to request more resources from the job scheduler. Their approach is based on MPI, and it can be integrated with our approach based on shared memory programming models.

Despite this tendency in the research, users still do not have simple and efficient tools, neither the support from job schedulers in production HPC machines that would allow them to exploit malleability. We integrated DROM APIs with OpenMP [84] and OmpSs [35] programming models and Slurm [9] job scheduler. However, DROM is independent of them, and it can be integrated with any other programming models or job schedulers.

DROM manages computing resources by using CPUSETs, lightweight structures used at the operating system level, easy and fast to use and manipulate. A similar approach was presented by [21], based on dynamically changing the operating system CPUSETs for MPI processes, but in this case, there was no integration with the programming model. This approach is equivalent to oversubscription of resources, i.e., more than one process running in the same core, which generally harms the applications' performance, as demonstrated in [5]. In our integration, OpenMP and OmpSs adapt the number of threads to changes in the number of computing resources, avoiding oversubscription. Moreover, OpenMP and OmpSs use threads instead of processes, easier to create and destroy, more efficient, lighter than MPI processes. At the same time, we support hybrid MPI+OpenMP/OmpSs applications, which allow the expansion of DROM capabilities to multi-node environments.

4.3 DROM: Dynamic Resource Ownership Management

DROM interface provides a communication channel between an administrator process and other processes to adjust the number of threads accordingly.

In this section, we present the proposed DROM API, and we will detail how we have integrated it with Slurm.

4.3.1 DLB-DROM Framework

DROM is part of DLB[44], a framework transversal to the different layers of the HPC software stack, from the job scheduler to the operating system. The interaction with the different layers is always done through standard mechanisms such as PMPI [85], or OMPT [83] explained in more detail in Section 4.4. Thus, as a general rule, applications do not need to be modified or recompiled to be run with DROM as long as they use a supported programming model (MPI + OpenMP/OmpSs). By pre-loading the library, these standard mechanisms can be used to intercept the calls to the programming models and modify the number of required resources as needed.

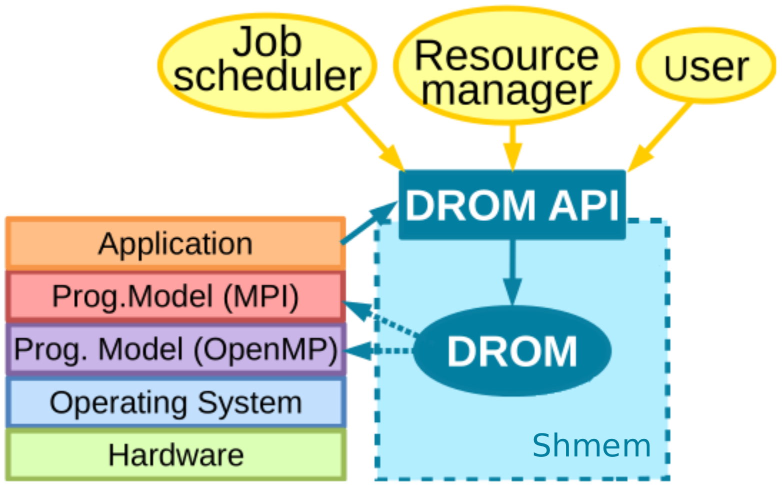


Figure 4.1: DROM Framework

In Figure 4.1 we can see the DROM module. DROM provides an API for external entities, such as a job scheduler, a resource manager, or a user, to re-assign the resources used by any application attached to it. Then, the DROM module running on each process will react and modify the computing resources allocated for the application. This procedure depends on the programming model, but in essence, it implies two steps. First, the application will modify the number of active threads running within the shared memory programming model (OpenMP, OmpSs). Lastly, each active thread will be pinned to a specific CPU core to avoid oversubscription during the coexistence of the many processes in the node.

DROM uses the node's shared memory to communicate the different processes, implemented as a common, lock protected address space where all processes attached to DROM can read and write. While the communication can be asynchronous for the *sender*, the *receiver*, by default, will use a polling mechanism based on the interception interfaces. This mechanism produces a negligible overhead but relies exclusively on the frequency of the programming model invocation. Alternatively, DROM also implements an asynchronous mode for the receiver using a helper thread and a callback system.

This design of the framework allows users or developers to add DROM support to an application with minimal effort. However, there are some considerations to weigh before running an application with DROM support, i.e., how the application reacts to an unintended change of the number of running threads and whether other hardware resources, apart from CPUs, can perform when other applications are co-allocated. The former only depends on the application implementation, and the latter may depend on several factors such as total memory consumed, and the I/O bandwidth:

- *Application's inherent non-malleability.* DROM may change the number of active threads, or *max_threads* in OpenMP nomenclature, at any time during the execution. For this reason, the application should be malleable. We consider an application completely malleable when its design allows changing the number of threads at any time, and its completion is still valid and successful. This condition requires a thread-based programming model with some level of malleability, although its effect does not need to be immediate. For example, OpenMP cannot modify the number of threads until the next parallel construct, but we consider it acceptable.

An application is not malleable when it obtains the number of threads at any arbitrary point of the execution and assumes it will not change in the future. For instance, a common practice in OpenMP applications is to allocate some auxiliary memory based on the current number of threads. Later, inside a parallel region, this memory will be indexed by the current thread identification number and may cause different errors depending on whether the team size is smaller or larger than assumed by the application. The suggested alternative is to exploit the features that the programming model already provides. For instance, a private array where its scope is limited to the parallel construct or a reduction clause where the programming model manages the auxiliary memory are two solutions that solve this issue and keep the application malleable.

- *Hardware is finite.* During the job co-allocation, DROM may reduce the number of active threads of other processes and may rearrange each thread's pinning to a new core, but it will not reduce the amount of allocated memory of any application. Therefore, the total memory capacity and bandwidth will be shared among applications.

4.3.2 DROM API for managing the co-allocation of applications

Processes attached to DROM can be managed from another process, referred as administrator process from now on. We consider Slurm the candidate for the administrator process, but the implementation of the interface presented in this section allows users to program their administrator process. In this case, the administrator process always runs as the same user making the submission, and the co-allocations are always limited to other applications of the same user.

The administrator process can manage other processes by communicating with DROM, mainly consisting of a single shared memory per node. Therefore, if the submission allocates more than one node, one administrator process must be created for each node that requires management, and eventual synchronization needs to be implemented within those processes.

The proposed DROM interface is presented below, and its code is Open Source [14]:

```
int DROM_Attach(void)
```

Attach current process to the system as DROM administrator. Once attached, the process is able to query or modify the process mask of other processes running with DROM support.

```
int DROM_Detach(void)
```

Detach current process from DROM system. If previously attached, a process must call this function to correctly close file descriptors and clean data.

```
int DROM_GetPidList(int *pidlist, int *nelems, int max_len)
```

Obtain the list of running processes registered in the DROM system.

```
int DROM_GetProcessMask(int pid, dlb_cpu_set_t mask,
   dlb_drom_flags_t flags)
int DROM_SetProcessMask(int pid, const_dlb_cpu_set_t mask,
   dlb_drom_flags_t flags)
```

Getter and Setter of the process mask for a given PID.

```
int DROM_PreInit(int pid, const_dlb_cpu_set_t mask,
   dlb_drom_flags_t flags, char ***next_environ)
```

Preinitialize a starting process into the DROM system, reserving some CPUs or making room in the node by shrinking other running processes according to *mask*. The usual workflow for this function is to register the current PID, then fork and exec into the new process keeping *next_environ* variable that permits the child process to register using the parent's process ID.

```
int DROM_PostFinalize(int pid, dlb_drom_flags_t flags)
```

Finalize a previously preinitialized process. This function should be called after a pre-initialized child process has finished its execution. The child process may have cleaned the shared memory if it runs a supported programming model, but this is not known from the job scheduler perspective. It is always recommended to call this function to clean the data.

Non-standard C types used in this interface are:

- `dlb_cpu_set_t` is actually a *void pointer* provided as an opaque type and it is casted back internally to `cpu_set_t`. This data set is a bitset where each bit represents a CPU. It is defined in the GNU C library [47].

- `dlb_drom_flags_t` is a custom bitset. This argument adds some flexibility to the interface by allowing some options like: whether the function call is synchronous or asynchronous, whether to steal the CPUs from other processes.

4.4 Integration of DROM with Programming Models

Currently supported thread-based programming models are OpenMP and OmpSs. DROM also supports MPI interception to add more synchronization points between the application and DROM, as well as to gather more information about the application structure and improve the resource scheduling policies.

4.4.1 Integration with OpenMP

Any OpenMP application can use the DROM library without having to be recompiled as long as the OpenMP runtime used supports OMPT. OMPT is a new interface introduced in the OpenMP Technical Report 4 [83], and included in the OpenMP 5.0 specification. The interface allows external tools to monitor the execution of an OpenMP program. Several OpenMP runtimes already include it in their latests versions, such as Intel's proprietary branch (2018.2.046) and their open-source branch based on LLVM's runtime [54].

If the OpenMP runtime implements this interface, DROM can register itself as a monitoring tool when the library is loaded. Then, it can set callbacks that will be automatically invoked for each parallel construct and implicit task creation, allowing to modify the number of resources accordingly.

4.4.2 Integration with OmpSs

OmpSs is a task-based programming model also developed at BSC. Its runtime includes

DROM support, and if enabled, any compiled application can enable the

DROM features provided by the runtime by setting the appropriate option.

4.4.3 Integration with MPI

HPC applications often request several computing nodes, thus, shared-memory programming models are not enough. A message passing interface is required for the communication among the different application processes, and the MPI standard is probably the most used for that purpose. Being aware of its importance, DROM implements an interception mechanism by using the MPI standard profiling interface, PMPI. PMPI allows any profiler, in this case DROM, to intercept any standard MPI call, and run custom code before and after the real MPI call.

DROM supports MPI interception and acts as an application profiler, but it does not implement malleability at the process level, i.e., MPI processes are never decreased or increased, nor any program data is ever moved between processes. For our purposes, MPI interception is only used to poll DROM and check if there are pending actions to take. If the program runs with a new version of OpenMP implementing OMPT or with OmpSs, the MPI layer is optional.

4.4.4 Integration with applications with no supported programming model

DROM has been designed to be easily used even for applications that do not run a supported programming model. The DROM library includes an interface for applications to become DROM-responsive and react to the reallocations performed by the manager process. Using the our interface in the application implies that it has to be recompiled, but it also offers more flexibility to only call DROM on *safe points* where the application can change the

```
#include "dlb.h"
int main(int argc, char **argv) {
    /* initialization */
    DLB_Init();
    ...
    /* main loop */
    for(i=0; i<end; i++) {
        if (DLB_PollDROM(&ncpus, &mask)
                == DLB_SUCESS) {
            modify_num_resources(ncpus, &mask);
        }
        #pragma omp parallel
        ...
    }
    /* Finalization */
    DLB_Finalize();
    ...
    return 0;
}
```

Listing 4.1: Iterative application manually invoking DROM

number of threads if it is not entirely malleable.

Listing 4.1 shows an example of an iterative application manually modified to support DROM. The effort for developers is minimal. First, the application needs to initialize and finalize DROM correctly when appropriate. Then, just before entering the malleable parallel code, it should poll the DROM module to check if the resources need to be readjusted and, if needed, perform the necessary actions. This adjustment needs to be done by the application. In the case of an OpenMP application, it may include a call to `omp_set_num_threads` and, optionally, a rebind of threads if the runtime is configured to bind them to CPUs.

DROM APIs were integrated into Slurm, to automatize the placement of jobs' tasks inside computing nodes whenever one or more malleable jobs are scheduled inside the same nodes.

The following implementation only affects job placement inside nodes, i.e., selecting for each node on which CPUs job will run, not affecting the scheduling process of jobs. *Slurmctld*, the cluster controller implementing the job scheduler, is unchanged.

A first use case for the presented integration is when the user asks to run multiple jobs in the same subset of resources, asking DROM to manage the available cores efficiently and automatically. A second use case is when the scheduler allows space sharing of resources, scheduling multiple jobs in the same nodes. It this case DROM can be an alternative to oversubscription, which oftern performs poorly.

The implementation is enclosed in the Slurm *task/affinity* plugin, in charge of distributing the resources assigned by *slurmctld* to the job's tasks.

Task/affinity is dynamically loaded by *slurmd* and *slurmstepd*, dividing the code flow in two parts. The first is done inside *slurmd*, in charge of managing single computing node resources, and thanks to the plugin, calculating and distributing CPU masks to tasks of the scheduled job. The second part is called by *slurmstepd*, a daemon that controls correct task launch and execution. At launch point, the plugin picks the mask assigned by *slurmd* and sets it.

In Figure 4.2 we give an example that clarifies the actions of DROM within Slurm. It illustrates the steps performed within DROM-enabled *slurmd* and *slurmstepd*. We present a scenario of two jobs starting to share a computing node. The *job 1*, to simplify the figure, is a one-task job already running in the *node 1*, while the *job 2* is a two-task job about to start on both *node 1* and *node 2*. Initially, *job 1* uses all the resources of *node 1*, then *job 2* takes part of them using DROM.

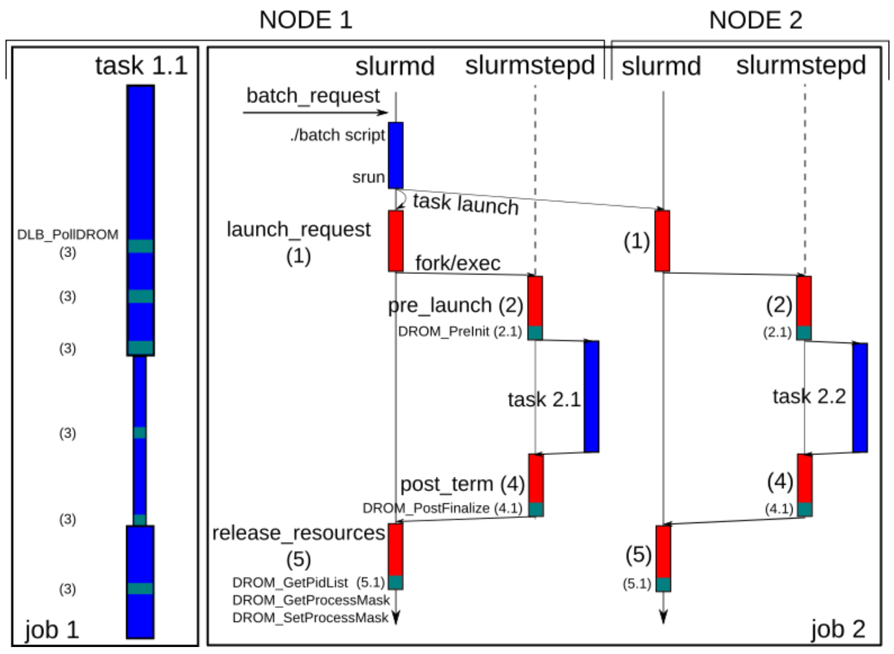


Figure 4.2: Slurm job launch procedure for DROM malleable applications in two computational nodes.

On the left we have *job 1* running *task 1.1* into *node 1*, on the right the start procedure for *job 2*. The vertical axis represents the time for each involved component. Red boxes are modified Slurm parts, blue boxes are unmodified parts, green boxes are DROM calls.

Starting from the top, *node 1*'s *slurmd* executes the submitted *batch script* for the *job 2*, that uses *srun* to launch a parallel malleable application. *Srun* sends requests of launching the tasks to the two nodes involved in *job 2* allocation. Both *slurmds* call *launch_request* (1) function, that calculates the

CPU mask for the starting task.

Since job 1 is already running, our implementation calculates a new mask for both new and running jobs. In this case, CPUs distribution is done to maintain processes balanced in the number of cores for each task, assuming that imbalance in hybrid MPI+OpenMP/OmpSs applications degrade performance. The algorithm also distributes cores keeping applications in separate sockets in order to improve data locality. In this scenario, for fairness, computational resources are equally partitioned among running jobs.

Following on, *slurmd* forks and executes *slurmstepd*. *Slurmstepd* calls a *pre_launch* (2) function, that sets calculated mask to the controlled task, and updates task 1.1 mask. This is done using *DROM_PreInit* (2.1) function. At the next malleability point, task 1.1 runs *DLB_PollDROM* (3), it gets a new CPU mask, and it applies it, reducing the number of assigned CPUs per task. Figure 4.2 shows the reduction in CPUs as shrinkage of the blue line. If job 1 runs on node 2, coordination is implicit in *slurmd*'s CPUs distribution, which gives the same placement for both nodes.

When a task ends *post_term* (4) is invoked, that involves a call to *DROM_PostFinalize* (4.1). This function can return CPUs to the initial owner of the CPUs, i.e., *job 1*. Of course, this is only possible if this job is still running and keep calling *DLB_PollDROM* (3).

When a job completes, *slurmd* calls *release_resources* (5), that redistributes free CPUs to still running tasks. In the case the job owner of the CPUs, in this case *task 1.1*, completes before the job 2, CPUs will be acquired by the job 2, that will expand its mask to increase node utilization. This is done by using *DROM_GetPidList*, *DROM_GetProcessMask* and *DROM_SetProcessMask* (5.1) APIs.

We first evaluate the introduced overhead of the DROM logic. We use an MPI + OmpSs application, and we compare their runtime when running with and without DROM.

Then, to evaluate the potential and utility of the DROM API, we perform two types of experiments that follow two realistic use case scenarios, supported by HBP:

1. *In-Situ Analytics*. The workload consists of two jobs: 1) a big and long job that we will refer to as *simulation* and 2) small and short job that we will refer to as *analytics*. This scenario corresponds to a use case of HBP in HPC machines, where a neuro-simulation is running, and a visualizer or a data analytics program can periodically check partial simulation results instead of waiting for the simulation to complete. In a standard system, to run the analytics, the user would launch a second job asking for resources and wait until they are available. Using DROM, the analytics would use part of the resources allocated to the simulation by temporarily shrinking its number of used resources. This permits running analytics in the same node, avoiding reading and writing data to disk as the analytics can exchange data with the simulation in-memory, or data transfer in case the analytics runs on a local machine.

2. *High-priority job*. In the second HBP use case, we consider the scenario of two jobs: 1) a long-running simulation and 2) a new high-priority long-running job, e.g., an interactive job or a high-priorityf simulation, arriving in the queue. In the absence of available resources, the high-priority job needs to wait in the job queue, or the already running job

needs to be preempted or oversubscribed, which would degrade the performance. With other malleability implementations, the simulation would need to shrink in the number of nodes, creating overhead due to data movement and checkpoint/restart operations. In the DROM case, the application can keep executing on the same number of nodes but on a reduced number of resources per node, while the high-priority job is scheduled to run in the same job allocation.

We use a set of real applications - two neuro-simulation applications and two synthetic benchmarks:

- NEST [60] is a simulator for spiking neural network models. It is parallelized with MPI and OpenMP. We have modified the code of NEST, based on version 2.12.0, to make it malleable[46]. Additionally, we have added calls to `poll_DROM` in the safe points where the number of threads can be changed.

- CoreNeuron is a simulator for modeling neurons and networks [59]. It is parallelized with MPI and OpenMP. We have modified the code to add calls to `poll_DROM` in safe points for malleability[58].

- Pils[45] is a synthetic benchmark, doing computation-intensive operations. It is parallelized with MPI + OmpSs. It can be configured to run with different numbers of MPI processes and OpenMP/OmpSs threads. In the evaluation, we use it to simulate compute-bound parallel data analytics.

- STREAM is a benchmark intended to measure sustainable memory bandwidth[75]. We configured it to run multiple iterations with an 8GB dataset. The application is parallelized with MPI + OpenMP. We used this benchmark to simulate memory-bound analytics software.

Pils and STREAM benchmarks are used to reproduce the behavior of in-situ visualizers and analytics used in HBP.

All the experiments are real-machine workload runs. For that purpose, we used MareNostrum III (MN3) supercomputer [15], based on Intel Sandy-Bridge processors, with each node containing two sockets with eight cores per socket and 128 GB of DDR3 memory. The operating system is a SLES distribution, with Platform LSF [53] resource manager. NEST and CoreNeuron were compiled using Intel 2017.1 compilers, and OpenMPI libraries version 1.10. Pils and STREAM were compiled with Mercurium 2.0.0 and Nanos 0.13a. We run the experiments using the original Slurm based on version 15.08.11 and the modified version that uses DROM to exploit malleability as described. To run the modified Slurm version we used the Slurm as a job methodology, and we instrumented applications to extract performance data.

All the reported results are an average of at least three runs performed in two MN3 nodes. We observed a maximum coefficient of variation of 3.4% in run time measurements. We analyzed the use cases from a system and application perspective by measuring:

- total run time
- single jobs response time
- average response time
- Instructions per Cycle (IPC)
- Cycles per microsecond

Each of the use cases is evaluated for several different configurations regarding the number of MPI processes and OpenMP threads per MPI process, as summarized in Table 4.1. All applications ask for 2 nodes and distribute MPI processes among them. We selected those configurations to con-

Application	Conf. 1: MPI x OpenMP	Conf. 2: MPI x OpenMP	Conf. 3: MPI x OpenMP
NEST	2 x 16	4 x 8	-
CoreNeuron	2 x 16	4 x 8	-
Pils	2 x 16	2 x 1	2 x 4
STREAM	2 x 2	-	-

Table 4.1: Use cases applications configurations.

sider in the evaluation eventual impacts on the performance of the number of MPI processes per node, e.g. having one MPI process per socket performs slightly better than having one per node. Also by using different configurations the placement and the lent cores distribution changes. As an e example, for NEST and CoreNeuron we observed increasing IPC switching from *Conf. 1* to *Conf. 2*. This is due to a different data access pattern and better data locality. Regarding Pils, in *Conf. 2* and *Conf. 3* it does request and run only on part of the node resources, even if the node is free. Even though it is supposed to be a small application, we run Pils in *Conf. 1* to have a reference case in which nodes are fully utilized for the whole execution time. Concerning STREAM, we do not need to change the configuration for it as the application is memory bound and over two CPUs per node performance keeps constant.

4.6.1 DROM overhead

In this evaluation we try to estimate the overhead of DROM on the MPI + OmpSs Pils application. Pils presents a small computation per loop cycle, so DROM continuously checks if there was a change in the DROM CPU mask for each cycle. To reduce the overhead DROM only checks if the specific boolean flag is set by the scheduler each time is called. We run Pils with two configurations: with 1000 and 10000 loops.

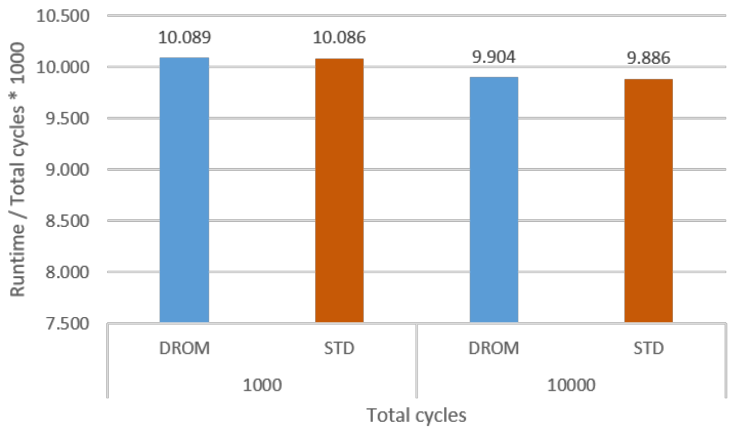


Figure 4.3: Runtime divided by the number of cycles multiplied by 1000 for Pils linked with DROM, and for the non DROM version, STD.

Figure 4.3 shows the runtime divided by the total number of cycles and multiplied by 1000. The first configuration shows an average overhead of 0.018% when DROM is used, while the second configuration shows an overhead of 0.2%. We consider this overhead negligible compared to other approaches of the state of the art, and we consider this evaluation a worst-case scenario, given the high frequency at which DROM is called.

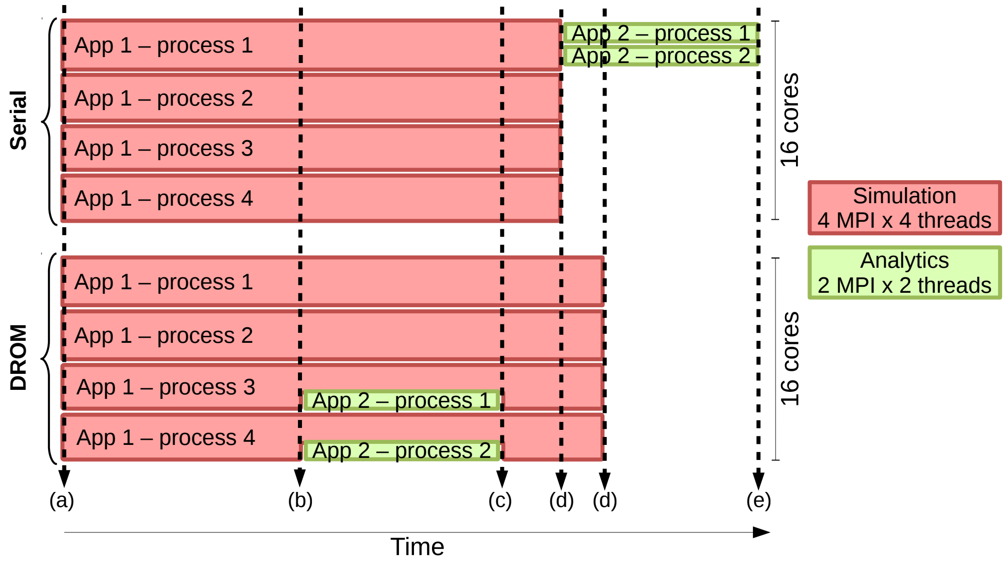


Figure 4.4: In situ analytics example.

4.6.2 Use Case 1: In Situ Analytics

In Figure 4.4 we can see a graphical representation of the first use case. The horizontal axis represents time while the vertical axis computational resources, i.e., number of cores. At time *(a)* the simulation is launched and started in the available resources. After 600 seconds, at time *(b)* the analytics is submitted.

We compare two scenarios, the *Serial* one considers that the analytics must wait for the simulation to finish before it can start at point *(d)* because no resources are available. Thus the simulation runs using all the available cores, and when it finishes, the analytic is executed. The second scenario is using *DROM* to start analytics immediately at time *(b)*, reducing the number of resources assigned to the simulation. Once the analytics finishes at point *(c)* the simulation gets its resources back.

We evaluated the following two-applications workloads. We will use the notation *simulation application+analytics application* when naming the workloads, i.e. NEST + Pils, NEST + STREAM, CoreNeuron + Pils, CoreNeuron + STREAM. For this use case, we evaluate and analyze the total run time, average response time, individual application's response time, followed by a discussion on the differences between Serial and DROM scenarios and the various configurations.

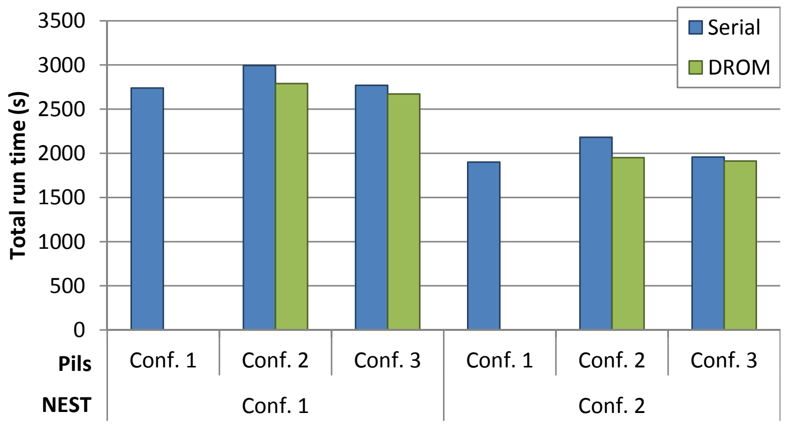


Figure 4.5: Run time of NEST + Pils workload. Y-axis represents total run time in seconds, X-axis shows the different configurations of the applications.

Figure 4.5 shows the difference in the workload's total run time when running the two configurations of NEST and Pils. Run time for DROM case is in average 5.9% better than Serial case for *Pils Conf 2* and *Conf 3*, and comparable to the reference case *Conf. 1*, which runs Pils on all the available resources. The average overhead of DROM scenario over *Pils Conf. 1* is 0.6%, varying with the analyzer's configuration. We observed that this is because of NEST implementation. Since its data is statically partitioned according to the maximum number of computational resources during initialization when applying malleability to shrink NEST, the tasks not computed by the removed threads are computed by some of the remaining resources, creating imbalance, as shown in Figure 4.6. This is a limitation of the application and not an overhead introduced by DROM. In fact, by increasing the number of stolen computing resources, like in the case of *Pils Conf. 3*, the number of excess tasks increases, and they are better distributed among the remaining resources. In this situation, we improve total run time up to 2.5% with respect to *Pils Conf. 1*, while for *Conf. 2* it reaches -2.6%. A fully malleable NEST version that does not partition data according to the initial number of threads would improve this result.

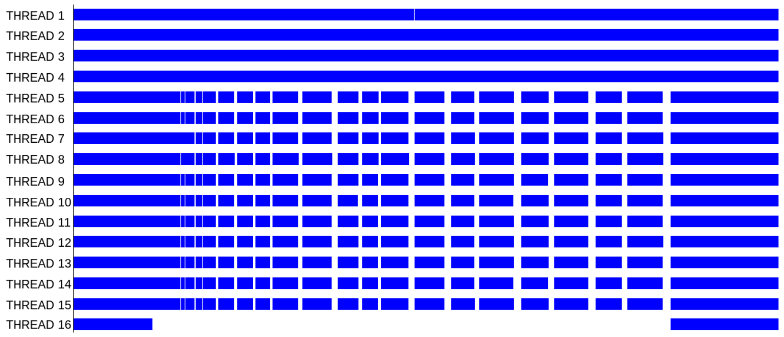


Figure 4.6: Trace showing simulator's threads on Y-axis. When thread 16 is removed, its data is computed by first 4 threads, while the others report lower utilization (white idle spaces).

Figure 4.7 shows single application's response time. Pils's response time, painted in lines pattern, decreases up to 96% due to DROM malleability reducing its wait time to zero, while its run time is approximately the same compared to the Serial case. This happens at the cost of NEST's response time, varying from 0% to 4.2%. A 0% increase in runtime is due to increasing IPC when running on a reduced number of threads.

In Figure 4.8 we can see the workload's total run time and each application's response time for NEST and STREAM use case. In this case, we remove 2 cores from the simulation to run a memory-intensive application, gaining on average 1.84% (up to 3.5%) in terms of total run time. STREAM's response time decreases up to 92% while NEST's increases up to 6.7% in the

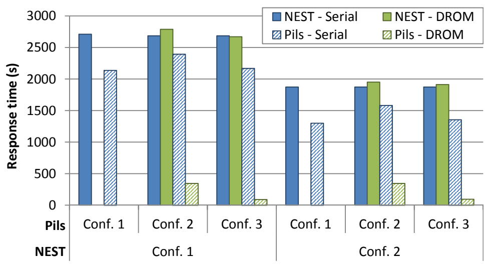


Figure 4.7: Individual response time of NEST and Pils in the NEST + Pils workload.

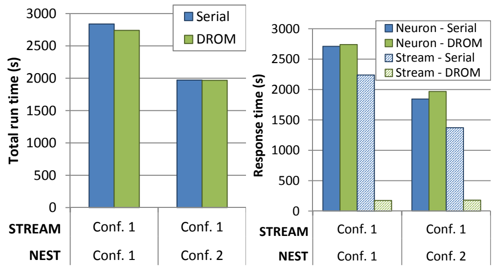


Figure 4.8: Run time (Left) and Response time (Right) of NEST + STREAM workload varying NEST configuration.

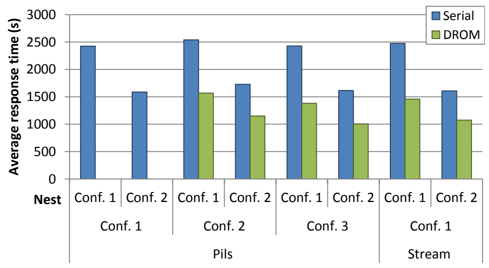


Figure 4.9: Average response time of NEST workloads.

worst case. Total run time is always better because of the benefits of memory-bound and compute-bound applications sharing the nodes.

Figure 4.9 shows average response time. Gain in DROM case is up to 48% and never less than 37% with respect to the Serial case.

Figures 4.10, 4.11, 4.12 and 4.13 show the same set of experiments but with CoreNeuron neuro-simulator. Results are very similar to NEST workloads, as also CoreNeuron presents the same data partition problem. Figure 4.10 shows improved run time when comparing with *Pils Conf. 2* and *Conf. 3*, and a maximum overhead of 5% compared to *Pils Conf. 1*. Compared to NEST, CoreNeuron shows slightly worse results when sharing with compute-intensive analytics like Pils, even if less affected by the number of requested resources, showing 2% of variation versus 5% of NEST. In Figure 4.12 the total run time is always better then the Serial cases for STREAM work-

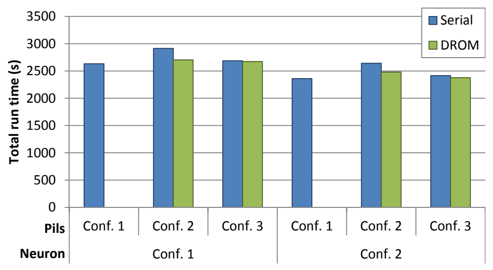


Figure 4.10: Execution time of CoreNeuron + Pils workload.

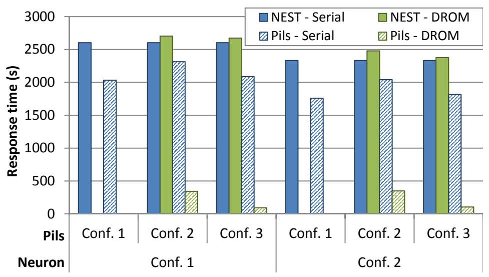


Figure 4.11: Individual response time of CoreNeuron and Pils in the CoreNeuron + Pils workload.

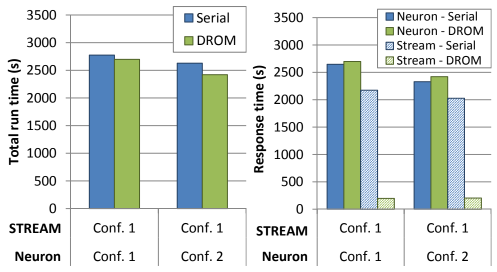


Figure 4.12: Execution time (Left) and Response time (Right) of CoreNeuron + STREAM workload varying CoreNeuron configuration.

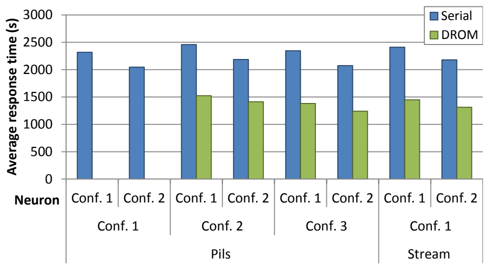


Figure 4.13: Average response time of CoreNeuron workloads.

loads (up to 8%), response time decreases up to 91% while CoreNeuron's increase is 4% in the worst case. Compared to NEST, it slightly performs better when sharing the node with memory-intensive applications like STREAM, with an average run time gain of 5.3% vs. 1.84% for NEST. Average response time in Figure 4.13 shows an average gain of 46.5% for the DROM scenario with respect to the Serial.

4.6.3 Use Case 2: High-priority job

In the second use case we analyze a single workload made up of two jobs, a long NEST and a long CoreNeuron simulation running on 2 MN3 nodes. Both jobs request *Conf. 1* presented in Table 4.1.

Again, we compare a *Serial* scenario in which the high-priority job can only start after the running job ends, and *DROM* scenario where the same job starts immediately by freeing some resources using the DROM interface.

Figure 4.14 presents traces for both scenarios. X-axes represent time, with same scale to compare total run time, while Y-axes show application's threads. At time *a)* NEST is submitted and runs on the entire two nodes allocation. After 1200 seconds, at time *b)* CoreNeuron is submitted. In the top trace, representing the Serial scenario, CoreNeuron needs to wait for all the resources to be freed to start, starting at time *c)*. The bottom trace represents the DROM scenario, in which CoreNeuron starts at submission time, sharing nodes with NEST. At time *d)* NEST ends, freeing half of the available resources, and CoreNeuron expands its allocation to keep maximum nodes utilization. In the DROM scenario, as both applications ask for two entire nodes, Slurm will apply the implemented automatic resource partition by reducing both new and running jobs used resources. Equipartition is applied, giving 16 CPUs per application on a total of 32.

We present total run time and response time to discuss system benefits of malleability for this use case, and application-related performance counters, like IPC and cycles per µs, to demonstrate applications are not interfering with each other when DROM is used for malleability.

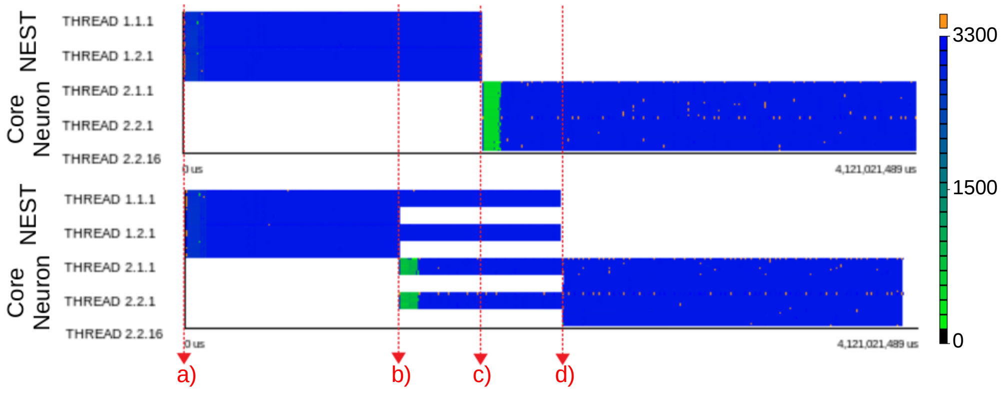


Figure 4.14: Traces showing cycles per µs of use case 2. Serial scenario is presented on the top, DROM on the bottom.

Looking at the total workload duration in Figure 4.14, in the case of DROM, better resource utilization leads to a total run time improvement of the 2.5%. The same figure shows the cycles per µs using colors. Showing the same color for both scenarios means there is no difference between Serial and DROM scenarios, and constant color during run time shows no variation in this metric when applying malleability to expand and shrink applications. The green color at the beginning of the CoreNeuron simulator shows lower cycles in the memory-intensive initialization phase. In the Serial scenario, during initialization, all computational resources are underutilized, while in DROM case, NEST keeps running, increasing utilization and reducing total workload run time.

Figure 4.15 shows the number of instructions per cycle for both configura-

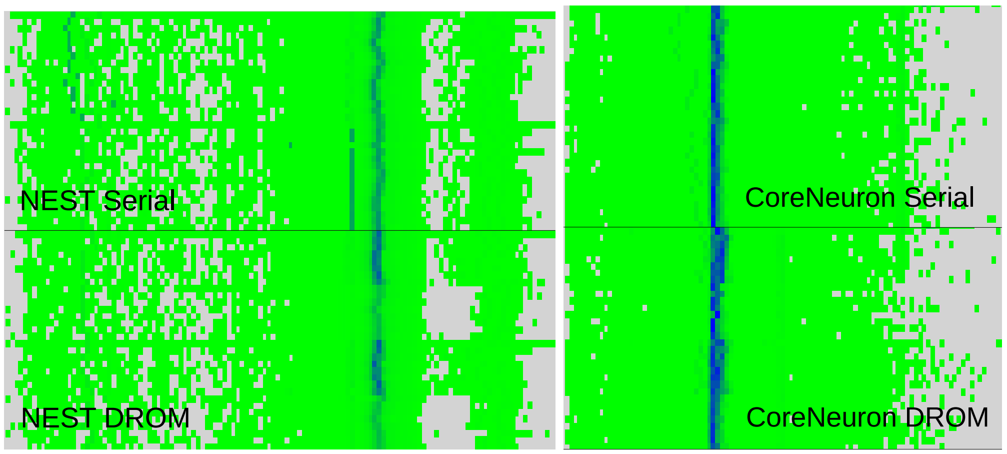


Figure 4.15: Histogram of instruction per cycle for CoreNeuron and NEST runing in serial and with DROM.

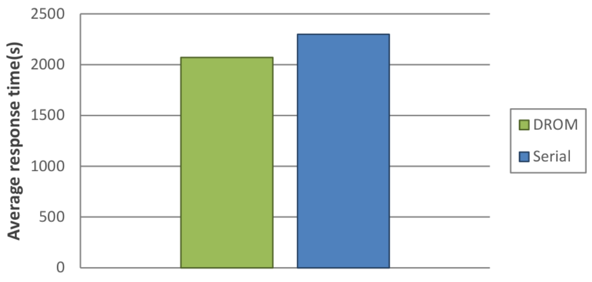


Figure 4.16: Average response time for use case 2 workload. DROM scenario improves response time by 10% with respect to the Serial scenario.

tions of the use case 2. Figures are grouped by application to be easily comparable. X-axes represent IPC in increasing order, Y-axes application's threads, blue dots show more frequent IPC and represent the histograms' main information. They demonstrate that the Serial and DROM scenario are comparable in terms of IPC. Regarding NEST, we distinguish some noise in the Serial scenario, and two-color variants for the most frequent IPC for DROM. This is because threads corresponding to the lighter color are removed to accommodate CoreNeuron at time (b) in Figure 4.14, distributing more computation on the darker part of the graph. For CoreNeuron, IPC in the Serial scenario is constant, but for DROM, we can distinguish two blue zones in correspondence to the threads in which the application starts, reporting slightly higher IPC. This is due to higher parallel efficiency when running on fewer OpenMP threads per MPI rank, improving total run time.

Finally, Figure 4.16 presents average response time for this use case. Response time improves by 10% compared to the Serial scenario due to gain in run time and because the high-priority job can start earlier, improving at the same time user experience when the job is interactive or giving earlier partial results when a simulation can start earlier.

4.7 Conclusions and Future Work

In this chapter, we presented DROM, an interface that enables malleability by creating communication between applications and job scheduler. The presented API permits to change computational resources allocated to a running application efficiently, without any overhead.

We designed the interface to be easily integrated into any programming model or directly into the application. We integrated with MPI, OpenMP, and OmpSs. Additionally, we presented an integration of the API with the Slurm node manager to achieve automatic distribution and placement of co-scheduled jobs inside nodes. We presented two use cases as a proof of concept, and we analyzed results from the workload point of view and application point of view. The evaluation of two use cases shows up to 48% improvement in average response time and up to 8% in total run time, by comparing to the serial case.

With this study, we open future work in two directions. On one side, we want to expand the potential of DROM, with new functionalities, like collecting useful data from applications at run time. The collected information can be consulted by an external to get info about applications performance and send them to the job scheduler to improve scheduling decisions. On the other side, we want to tight the communication between the different layers of the HPC software stack, i.e., by developing DROM-aware scheduling and resource management policies. The simplicity of DROM APIs gives more freedom to the scheduler, that can implement malleable scheduling techniques, for instance, by choosing one or multiple specific jobs to share computational nodes, or at the resource management level, by choosing as ”victim” nodes the ones with lower utilization. Combined with a job scheduler/resource manager, DROM can be used in many different ways, including implementing new scheduling policies based on malleability, e.g., policies based on co-scheduling, or as an alternative to jobs preemption. Job scheduling policy is the taken direction for this thesis, presented in the following chapter.

"The supermind knows unquestionably what is "right", and problems result only when the conscious mind stubbornly refuses to recognize this superior knowledge."

Robert A. Monroe

5 Slowdown driven scheduling and resource management for malleable jobs

In the last years, malleability in job scheduling is becoming more critical because of the increasing complexity of hardware and workloads. In this setting, using nodes in an exclusive mode is not always the most efficient solution as in traditional HPC jobs, where applications were highly tuned for static allocations but offering zero flexibility to dynamic executions. This chapter proposes a new holistic, dynamic job scheduling policy, Slowdown Driven (SD-Policy), which exploits applications' malleability as the key technology to reduce the average slowdown and response time of jobs. SD-Policy is based on backfill and node sharing. It applies malleability to running jobs to make room for jobs that will run with a reduced set of resources only when the estimated slowdown improves over the static approach. We implemented SD-Policy in Slurm and evaluated it in a real production environment, and with a simulator using workloads of up to 198K jobs. Results show better resource utilization by reducing makespan, response time, slowdown, and energy consumption, up to respectively 7%, 50%, 70%, and 6%, for the evaluated workloads.

5.1 Introduction

In static HPC environments, it is a widely extended practice to allocate full nodes for exclusive utilization. This solution minimizes interference between jobs, and applications can be tuned to exploit all the node resources. However, emerging execution environments, architectures, applications, and workflows are more and more increasing in complexity. To support the new workloads, computing nodes increased their complexity, with the multicore and manycore technology delivering high computing power, combined with different memory layers for better I/O management. It is difficult for every single piece of the workload to exploit all the available resources, so dynamic resource management approaches need to be evaluated.

In a context where jobs can share nodes to take advantage of all the available resources, the execution of those workflows could improve machine's efficiency.

Resource sharing as a strategy to improve resource utilization has been considered before with time-sharing approaches such as Gang-scheduling [48] or space-sharing, such as oversubscription for co-scheduling [86], both being not a specific policy but a family of policies in the same way backfill includes many variants. Resource sharing often poorly perform because of the contention created by multiple applications that, tuned to run in exclusive job allocations, share the same resources, and by the overhead of context switching.

Our proposal overcome this problem by exploiting malleability [41] as a way for efficiently managing resources inside computing nodes. Programming models such as
OpenMP [84], OmpSs [35], and MPI [76] give basic support to changing the number of threads or tasks and their binding to resources, but the rest of the HPC layers lack communication between them, and this feature remains isolated. We started to address this problem by developing DROM, an API [33] that enables the scheduler to communicate with applications that can adapt at run time to changes in the computing resources. We took advantage of shared memory programming models to hide programming complexity and to allow dynamic cores allocation efficiently. We also showed that malleability outperforms static resource sharing solutions and reduces response time.

This chapter presents SD-Policy, a Slowdown Driven, dynamic job scheduling policy, which exploits malleability offered by DROM as the key technology to reduce the average slowdown and response time of jobs. SD-Policy, based on backfill, applies malleability to running jobs to make room for jobs that will run with a reduced set of resources, only when the estimated slowdown improves over the static scheduling. SD-Policy supports mixed workloads with malleable, moldable, and static applications, ideal for transition to a malleable environment. Our approach is holistic, as we put effort into connecting all the HPC software stack: applications, programming models, node resource management, and system-level scheduling need to work in coordination to achieve the best performance. We implemented SD-Policy into the Slurm [9] by extending its plug-ins and evaluated it with real-machine runs using Slurm as Job methodology in Marenostrum4 (MN4) [16], using up to 49 computing nodes, and with Slurm simulator [4], using standard workloads [39] [40], up to 198K jobs on 5040 nodes. With the two methodologies, we demonstrate the feasibility of the SD-Policy in a real production system and that significant performance benefits can be obtained with dynamic scheduling policies when executing long, meaningful workloads. The main contributions of this chapter are:

- A dynamic job scheduling, Slowdown-Driven, policy for malleable jobs based on backfill and co-scheduling.
- An efficient resource selection algorithm based on system metrics feedback, i.e., average slowdown.
- The integration of our proposal in the full software stack: Slurm workflow manager, standard programming models like OpenMP, OmpSs, and MPI, with a negligible effort from developers.

This chapter is organized as following: Section 5.2 present the state of the art of the subject of scheduling in the context of malleable jobs. Section 5.3 describes the developed job scheduling policy and its implementation in a well know workload manager, and Section 5.4 presents both simulations and real runs evaluations on the presented policy and its parameters, together with an in-depth analysis of their effects on system performance. Finally, Section 5.5 resumes and concludes the chapter, with insight on future work.

5.2 Related work

Several studies have been done to reduce programming complexity when developing malleable applications. Utrera [104, 103] uses folding techniques plus FCFS-malleable and backfill-malleable policies that use co-scheduling and oversubscription to start MPI jobs when not enough resources are available. Kale [56] implement malleable and evolving jobs on the top of Charm++ and defined a scheduling policy based on equipartition. Prabhakaran [89], using Charm++ malleability, implemented shrink/expand operations in a production scheduler, together with a scheduling strategy based on equipartition and combining rigid, evolving, and malleable jobs. Chadha [22] uses an MPI extension and SLURM to build a malleable job scheduling policy guided by a ratio between the number of processes and the compute time. Martin [73] introduces FLEX-MPI library for dynamic reconfiguration of MPI applications based on checkpoint and restart, while Comprés at al. [26] implement MPI malleability with online data redistribution, plus shrink and expand operations in Slurm. In general, data redistribution in malleable MPI implies overhead of data movement and effort for developers, not in line with our research, being effortless and efficient. Cera [21] implements malleability based on dynamic CPUSETs using MPI and a production resource manager. This approach is similar to how we use malleability, but in our case, we do not oversubscribe MPI processes because we demonstrated it could degrade the application's performance [5] and we integrate with shared memory programming models for better performance. While supporting MPI for multi-node applications, our approach uses DROM interface [33], which allows malleability in computational nodes by changing the number of threads of OpenMP [84] or OmpSs [35] applications. This approach enables effortless, dynamic, and zero overhead moldability and malleability in the number of cores and allows the scheduler to make decisions in real-time with almost instantaneous adaptation from applications.

Many malleability approaches were studied in the context of the grid, taking advantage of the application's feedback, like [95] [105] [12]. Those approaches use the application's performance models and run time performance measurements. Our algorithm differs because it uses feedback from the scheduler itself, scheduling at a higher level of abstraction, based on system metrics, e.g., average system slowdown rather than application feedback.

5.3 Slowdown driven algorithm

This section presents Slowdown Driven policy and its implementation [31] in Slurm Workload Manager.

SD-Policy is a variant of backfill, co-scheduling malleable jobs in non exclusively allocated nodes, where they can efficiently partition the available resources using malleability. It is based on the simple idea that if an arriving job is malleable and no enough resources are available to run it by static backfill algorithm, SD-Policy selects some of the already running jobs, called mates, shrinks them to run the new job, only if predicted system slowdown is improved. Mates, selected by minimizing their slowdown increase, will be expanded back when the new scheduled job terminates.

We divide the SD-Policy in three parts: *scheduling level*, *resource selection level*, and *node level*. We will start describing the scheduling policy in the controller from a system-level point of view. We will then present the resource selection and placement algorithm based on the impact on job mates and system

Listing 5.1: SD-Policy scheduling algorithm

```
schedule(new_job)
    j = new_job
    if(!(nodes = select_nodes(j,free_nodes ,null))
        if(!malleable(j))
            return
    else run_job(j, nodes)
    static_end = get_wait_time(j) + j.req_time
    mall_end = j.req_time + runtime_increase(j)
    if (static_end > mall_end)
        s_mates = select_nodes(j,free_nodes ,nodes)
        if(s_mates)
            update_stats(j,s_mates)
            s_nodes = get_nodelist(s_mates)
            run_job(j,s_nodes)
```

metrics feedback. Finally, we will explain how malleable applications interact with DROM and the node manager and the implemented logic for shrinking and expanding operations, cores selection, and distribution. At the end of the chapter, we present the used runtime models for the scheduling algorithm and the simulator.

5.3.1 Scheduling algorithm

The scheduling algorithm is a variant of backfill that considers malleability. It first tries static placement of each job, and if not enough free resources are available, it attempts the flexible scheduling approach for the same job. Malleable backfill runs for each job right after the static trial and not after the static backfill completed for all jobs. This strategy favors the scheduling of jobs in order of priority. The scheduling algorithm is detailed in Listing 5.1.

The scheduler uses an end time prediction, *static_end* and *mall_end*, to estimate if applying malleability would improve the new job slowdown. In the affirmative case, it calls the malleable resource selection algorithm. End times are calculated using an estimation of the wait time in the static case by creating a map of job reservations based on arrival order to find out when the new job will start. In the malleable case, the job will begin immediately, while the increase in runtime is proportional to the decrease in resources, using models explained later in detail.

The scheduler collects the list of selected mates by calling *select_nodes*, using the algorithm described in Section 5.3.2, and for each mate and the new job, it updates the requested time and the predicted end time. Then it communicates with the node managers, starting the procedure described in Section 5.3.3.

We implemented Listing 5.1 in the Slurm backfill scheduler by editing *sched/backfill* plug-in.

5.3.2 The resource selection algorithm

The resource selection problem, when co-scheduling multiple jobs with malleability, is reduced to selecting best job mates that will shrink their allocation to make room for the new jobs. Selecting mates is an NP-complete problem. The objective function tries to find the best set of mates with minimum penalty p, i.e., the jobs that receive minimum performance impact when malleability is applied. We describe it with Equations 5.1-5.3. Given:

- $x_i \in \{0, 1\}$: mate i is selected
- n number of mates
- p_i penalty estimated for mate
- P maximum penalty for a single mate

- w_i weight of the mate i
- W weight of the scheduled job

we define the *Performance Impact (PI)* with the objective function 5.1:

$$PI = min \sum_{i=1}^{n} x_i * p_i \tag{5.1}$$

With the constraints 5.2 and 5.3:

$$p_i < P, \forall i \in \mathbb{R} \tag{5.2}$$

$$\sum_{i=1}^{n} x_i * w_i = W \tag{5.3}$$

Following on, we define the Performance Impact PI, penalties p and P, weights w and W, and the heuristic used to solve this problem.

Performance Impact

PI is defined as the sum of slowdown increase for each mate when malleability is applied. As Feitelson [43] reports, there is no best metric when we talk about job scheduling evaluation, but it seems slowdown metric helps faster convergence in the preemptive scheduling, similar to our policy. Slowdown, normalizing response time by the runtime, does not give precedence to long jobs like response time, increasing fairness for users submitting short to medium jobs.

Penalties

We calculate the penalty *p* assigned to each mate based on estimation of the slowdown increase as:

$$p_i = (wait\_time + increase + req\_time)/req\_time \tag{5.4}$$

Equation 5.4 considers *wait time*, increase in total run time based on Equation 5.6, *increase*, and the user *requested time* for the job. Equation 5.4 is an estimation, as the job duration is usually not equal to the requested time.

The penalty will give precedence to jobs that waited less in the queue and jobs requiring a larger amount of time, so the impact in slowdown will be minimal. We define *P* as *MAX_SLOWDOWN*, a cut-off for *p*. A cut-off is needed for two main reasons: reducing the eligible mates to reduce the computation and avoiding penalizing jobs with a high slowdown. We implemented this parameter in two ways:

1. *A static value* chosen by system administrators. The value can be chosen empirically or by analyzing the history of a system. In this case, the slowdown must be calculated using user time estimation, not the real runtime, because this metric is the only one the system can use to predict the slowdown of running and waiting jobs.

2. *A dynamic value*: the scheduler automatically sets the cut-off based on the average system slowdown considering running jobs, and it is updated every time the controller is not busy in scheduling jobs. The basic idea is to spread the slowdown in a similar way among running jobs, so jobs exceeding the average slowdown are not considered for malleabil-

ity. Like median and 70 percentile, other metrics were analyzed, but they did not report improvement overall.

Weights

Following constraint 5.3, we define weight w as the number of allocated computing nodes for the mate, and W as the number of computing nodes requested by the scheduled job. Constraint 5.3 helps to keep balance in the system in terms of the number of cores per node jobs use in the allocated nodes, assuming jobs are statically load-balanced, which is the typical case in the HPC environment.

Heuristic

The proposed heuristic tries all the different combinations of mates, by iterating recursively on the list of mates for the power of m times, where m is a configurable parameter representing the maximum number of mates. We did not see improvements in system metrics increasing m over two from our evaluation with standard workloads, so we kept it as an optimal value. For $m = 2$ the complexity is quadratic, which is acceptable for our purpose. Some optimizations could further reduce this complexity. For each combination that satisfies constraints 5.2-5.3, we calculate the Performance Impact and update the best solution if improving it. Mates list is sorted based on the mates' penalty p. If the list is too big, it can be reduced by considering only the first nm mates, with lower p. The algorithm supports contiguous allocations, node filtering by name, architecture, memory, and network constraints. Options such as including free nodes, and having more than two mates per node are supported.

Listing 5.2: SD-Policy node selection heuristic

```
select_nodes(new_job, nodes, free_nodes)
    if (free_nodes.count > new_job.requested_nodes)
        return s_nodes = pick_nodes(new_job, free_nodes)
    if(nodes && malleable(newjob))
        mates = get_list_of_mates()
        mates = filter_and_sort(mates, MAX_SLOWDOWN)
        s_mates = pick_mates(newjob, mates, nodes)
        return s_mates
```

We set the following constraint: new jobs must finish inside mates' allocation to avoid, in the case of multiple mates, that one of them finishes earlier. We avoid this case because the new job would expand to occupy the full nodes of just a part of its allocated space, creating unbalance in the case the application cannot balance its load dynamically. The constraint also avoids creating an additional delay for jobs scheduled to run afterward. Listing 5.2 synthesizes the presented algorithm.

We implemented the malleable resource selection algorithm in Slurm *select/linear* plug-in. This plug-in is in charge of selecting entire nodes for jobs to be scheduled, respecting all job's constraints, and optimizing the nodes' placement using different criteria. We opted for the linear plug-in because malleable jobs can expand and shrink in the node, so there is no need to select individual cores at this point of the scheduling flow. In this way the node manager has more freedom in binding specific cores to jobs, having a better view of the node usage since it can directly communicate and gather information from applications.

5.3.3 Node management algorithm

Node management is the bottom layer of resource management, and it directly interacts with jobs. In the SD-Policy, node managers can select the right amount of computing cores to give to each job, assign them at launch time, and control their number at runtime.

To achieve described malleability, we integrated DROM API into the node manager to automate the placement of jobs' tasks inside computing nodes whenever one or more malleable jobs are co-scheduled in overlapping job allocations. Node managers calculate tasks to cores distribution among jobs and automatically, keeping jobs balanced and isolated. Exceptional cases with nodes running different configurations or jobs asking a different distribution are supported, e.g., using a manual organization to optimize resource usage for master-slave application architectures or memory-intensive jobs that do not need a high number of computing cores but instead more memory bandwidth. In the second case, particularly common in the described HPC workloads, a distribution with few computing cores per socket will leave more resources to mates jobs while fully taking advantage of the memory bandwidth.

We defined the *SharingFactor*, a limit on computational resources that can be taken from a running job in a computational node when shrunk, to implement fairness for mates, and to study performance when changing the number of assigned resources. We evaluated static values for this parameter in Marenostum IV (MN4) [16] and different cores distributions algorithms for the automatic distribution case. Results show that the best overall performance is obtained when the applications run isolated in separate sockets. For MN4, the number of sockets is two, so the sharing factor is set to 0.5.

Listing 5.3: SD-Policy node management algorithm

```
while(1)
    if(new_job = get_next_job())
        if(malleable(newjob))
            runningjobs.add(new_job)
            distribute_cpu(runningjobs)
            for job in running_jobs
                shrink_job(job)
            DROM_run(newjob)
        run(newjob)
    if(end_job = get_finished_job())
        if(malleable(end_job))
            if(end_job == mate)
                distribute_cpu(runningjobs)
            else
                owner = get_owner(end_job, runningjobs)
                expand_job(owner)
            DROM_clean(end_job)
```

In a dynamic approach, online performance analysis of running jobs would feed a tuning algorithm for selecting optimal values of SharingFactor, further increasing nodes efficiency.

The number of cores assigned to the malleable scheduled job depends on the SharingFactor and the minimum amount of cores to which the running jobs can shrink. This last value is equal to the static number of MPI ranks the application is running, to which we assign a minimum of one computing resource per rank.

The implementation in Slurm, described from the DROM point of view in Chapter 4 was extended, and it is presented in Listing 5.3:

1. At job start or end, the node manager (slurmd) interacts with DROM to get information about running tasks. In the case of a starting job, our algorithm recalculates affinities for all the node tasks. Cores distribution intelligently maintains running and new processes balanced in the num-

ber of cores per task, assuming, without any other information, that the imbalance degrades performance. Cores distribution keeps jobs in separate sockets to improve data locality and reduce interference between jobs. At the job's end, the algorithm returns the cores of the ended job to the owner. If any of the already running owners terminates its execution before the new job, its cores will be distributed to the remaining running tasks to increase node utilization.

2. Once calculated cores distribution, the job manager (slurmstepd) checks if the dependencies created by redistribution of cores are satisfied. In other words, it checks if the jobs where the new job will take cores from are already running or about to start to assure that DROM assigns cores correctly. Afterward, the job manager launches tasks using the DROM API.

3. At the job's launch time, DROM launches and sets the affinity for each new task and attaches them to the DROM space. At the end time, it cleans information of tasks from the DROM space. DROM also sets the new affinities for running tasks, so when the tasks reach the next malleability point, they can adapt to the change. At the end time, the API can be set to return cores to the original owners.

5.3.4 Runtime model for malleable jobs

SD-Policy is based on time estimations, so we implemented a model shaping how applications' duration is affected when malleability is applied. We already showed that the performance impact of DROM applications when changing the number of resources is negligible, so we assume the duration of jobs is proportional to the number of assigned computing resources and the ability of jobs to adapt to load unbalance.

To estimate the increase in the runtime of a shrunk job, we partitioned the job duration while sharing nodes in T time slots t, each time slot is a different job's resource configuration. We proportioned the runtime to the number of used resource for each t by putting in relation the static duration with the number of resources. We developed two models: the *ideal* and the *worst case*.

In the *ideal model*, applications do not suffer from the imbalance in the number of resources used by each process. With this model, if one job's task uses one computing node and the other task just half, the performance will be linear with the number of resources. In this case, we compute the increase in the runtime with Equation 5.5.

$$increase = \sum_{t=1}^{T} (req\_cpus / used\_cpus_t * time_t) \tag{5.5}$$

This model represent applications that can dynamically adapt their load to the resources at run time. *tot_cpus* is the original number of cores used in the log, *used_cpus* the assigned number at each time slot.

In the *worst case model* the same scenario would bring to lower utilization since performance is limited by the less used node n over all nodes N. We compute the increase in the runtime with Equation 5.6.

$$\begin{aligned} increase = \sum_{t=1}^{T} (req\_cpus / \min(cpus\_per\_node_{nt}) * time_t) \\ \forall n \in \{1, N\} \end{aligned} \tag{5.6}$$

The second model represents statically balanced applications. In this case, unbalanced changes in the used resources among nodes generate imbalance. The characterization of applications would lead to a more detailed model that would improve simulated experiments precision, but in its absence, we can give an upper and a lower bound by considering the two models.

We used the described models for time estimations in the SD-Policy and calculating jobs' runtime in the Slurm Simulator. In the SD-Policy case, we use the *worst case model*, to be able to grant correct jobs execution and completion. In the simulator, we try both models and compare results in Section 5.4.3. Running real runs experiments, in Section 5.4.4 will be the third source of information on how typical HPC applications perform when adapted to be malleable.

5.4 Evaluation

The evaluation is divided into two parts, depending on the used methodology: workloads real-run and workloads simulations. We use the first approach to give proof of the effective implementation of the policy running in a production system, and its integration with well-known schedulers and programming models, as well as a performance evaluation with benchmarks, neural network spiking simulators coming from Human Brain Project [51], and a computational multi-physics solver. We evaluate this workload on MN4, on up of 49 computing nodes with two sockets equipped with Intel Xeon Platinum 8160 processors, 48 cores, 96GB of main memory per node, for a total of 2352 cores. Each workload runs for about two days.

Based on whole systems simulations, the second methodology scales the evaluation to workloads up to 80640 cores for eight months, allowing the analysis of the SD-Policy performance running on entire HPC systems.

Table 5.1 presents information about workloads used for the evaluation. All workloads and models come from Feitelson database web page [39]. We generated workloads 1, 2, and 5 with the model developed by Cirne [25], based on the characterization of four different logs. We configured it to use the ANL arrival pattern, and we scaled the model to the considered system size. Since we were interested in our algorithm's performance when the time-based predictions are precise, we generated *Workload 2*, equal in distribution to *workload 1* but with the job requested time same to the real duration. *Workload 3* is part of the RICC installation trace from 2010, a realistic workload characterized by a high number of small jobs requesting few nodes, ranging from short to long runtime, up to four days. *Workload 4* is the cleaned version of the CEA-Curie log from 2011, only considering the primary partition.

Workload 5 was created from the Cirne model, then converted to real applications submissions. We modeled applications behavior and scalability presented in Table 5.2, then we calculated parameters to adapt the applications to each entry of the workload, regarding the number of requested nodes and duration of the job. We chose a set of applications with different behaviors in the utilization of CPUs and main memory. In the lack of statistics about HPC workloads characterization in the literature, we organized the workload in three main types of applications, equally distributed: compute-bounded jobs with low memory utilization (PILS), memory-bounded jobs with lower CPU utilization (STREAM), large simulations, memory and compute-intensive (CoreNeuron, NEST, Alya).

The following section presents different evaluations:

ID	Log/model	# jobs	System (nodes/cores)	max job (nodes/cores)	Avg resp. time (s)	Avg slowdown	Makespan (s)
1	Cirne	5000	1024/49152	128/6144	122152	3339,5	899888
2	Cirne_ideal	5000	1024/49152	128/6144	126486	3501	896024
3	RICC-sept	10000	1024/8192	72/576	43537	1341	407043
4	CEA-Curie	198509	5040/80640	4988/79808	29858,5	3666,5	21615111
5	Cirne_real_run	2000	49/2352	16/768	56482	4783,1	159313

Table 5.1: Description of workloads

Application	% workload	ReqNodes	ReqTime	CPU utilization	Memory utilization
PILS [45]	30.5%	small to high	small/med	high	low
STREAM [75]	30.8%	small to high	small/med	low	high
CoreNeuron [59]	35,5%	small to high	small to high	high	med
NEST [60]	2.6%	small to high	small to high	high	med
Alya [106]	0.6%	small	high	high	med

Table 5.2: Workload characterization for real runs evaluation

- Evaluation of MAX_SLOWDOWN
- Analysis of simulated workload 4
- Evaluation of runtime models
- Analysis of Workload 5 in a real environment

We used the following metrics for the evaluation, presented as single values calculated over the whole workload or per-job-category, dividing the workload in categories based on the size and the length of the jobs.

- Makespan
- Average response time
- Average slowdown
- Energy consumption
- Number of jobs scheduled with malleability

5.4.1 Evaluation of MAX_SLOWDOWN

Different values of MAX_SLOWDOWN can have a high impact on the performance of SD-Policy. Low values will limit the number of times malleability can be applied, and high values could degrade jobs and system performance. We simulated workloads 1, 2, 3, and 4 using SharingFactor of 0.5 and the ideal runtime model. We tried different values for MAX_SLOWDOWN represented by the following labels: MAXSD 5, MAXSD 10, MAXSD 50, MAXSD infinite, and the dynamic cut-off based on feedback from running jobs slowdown, DynAVGSD.

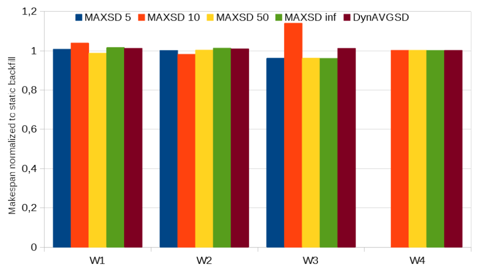


Figure 5.1: Makespan for workload 1 to 4 changing the MAX_SLOWDOWN parameter, normalized to static backfill simulation.

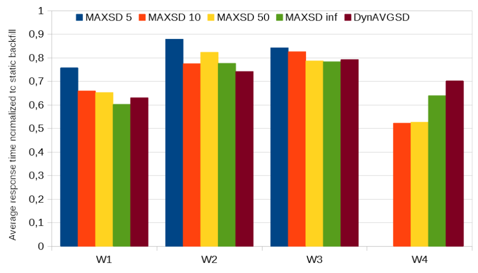


Figure 5.2: Average response time for workload 1 to 4 changing the MAX_SLOWDOWN parameter, normalized to static backfill simulation.

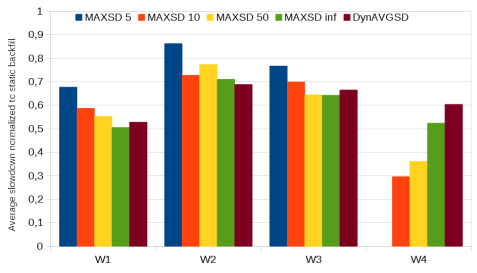


Figure 5.3: Average slowdown for workload 1 to 4 changing the MAX_SLOWDOWN parameter, normalized to static backfill simulation.

Figures 5.1, 5.2, and 5.3 show the makespan, average response time and average slowdown for the workloads, normalized to static backfill. In the first three workloads, increasing the slowdown limit improves the system's slowdown, showing that not filtering mates still improves the system performance overall. We observe a reduction in slowdown up to 49.5%, 31%, 25.7%, and 70.4%, respectively, for Workloads 1, 2, 3, and 4.

On one side, even with a high limit for MAX_SLOWDOWN, our algorithm tries to avoid applying malleability when it would not bring improvements, avoiding performance loss also when increasing the parameter to an infinite value. On the other side, a system administrator could still consider using a relatively low limit to avoid some jobs being penalized too much, or implement different queues with different QoS policies using different MAXSD configurations.

In Workload 2, the slowdown does not monotonically decrease with the increase of the limit, showing an increase at value 50 but still outperforming static backfill. SD-Policy DynAVGSD brings further improvements in the same workload, where the SD-Policy works with real job durations and not the user requested time. The explanation of the observed behaviors resides in the fact that variance in the real average slowdown is much higher than in the predicted average slowdown when using user-requested time, so a dynamic value of MAX_SLOWDOWN benefits this evaluation. Also, using real predicted metrics allows the SD-Policy to be more precise. This observation suggests that using a predictive method for job's runtime, i.e., based on machine learning, rather than asking the user, will improve our policy's performance. Comparing workload 1 and 2, the static backfill for the first workload outperforms the static backfill for the second by 8.6%, showing the interesting result that precision of job's duration does not always produce better average system slowdown. On the other side, having an exact job duration allows jobs not to be delayed in their start time, making backfill correct. Backfill behavior, also base algorithm of SD-Policy, influences our approach with Workload 1 performing 29.6% better than Workload 2.

In Workload 4, the best value is obtained in the MAXSD 10 case, while higher values and the dynamic version do not improve results. For this significant workload, we have the maximum observed reduction of the response time and the slowdown up to respectively 50% and 70%, showing the high benefits a system can have over a static backfill approach.

5.4.2 Analysis of a big workload

We simulated workload 4, significant in length and number of nodes, and analyzed details by partitioning the jobs into categories depending on the requested resources and runtime. We compared the static backfill version with the MAXSD 10 version. The SD-Policy improved slowdown by 70.4% while keeping makespan constant. We compared average slowdown, average runtime and average wait time for static backfill and the SD-Policy, presenting the ratios between the two versions in Figures 5.4, 5.5 and 5.6. The malleable version highly improves the slowdown of jobs consuming up to 4 hours and asking up to 512 nodes, up to 569%. Relatively small and short jobs have a very high slowdown, compared to larger and longer jobs. Those are the primary jobs that benefit from the SD-Policy.

The rest of the slowdown heatmap, in Figure 5.4 keeps having better values even for bigger and larger jobs, except for three categories. Two categories contain few jobs to make some conclusions, but the 121 jobs asking 512 to 1024 nodes with duration in 12 hours to 1-day range show an increase of 15% in the average slowdown. Even if it seems this job category is penalized, the average slowdown for those jobs was lower than other neighbors in the heatmap, so the result shows that the SD-Policy generates a more fair distribution of the slowdown compared to static backfill. The rest of the jobs, even if affected in their runtime (Figure 5.5), because of the malleability, improve their wait time (Figure 5.6), showing fairness is not an issue for some particular category of jobs.

Figure 5.7 shows the trend of the average slowdown per day, comparing the static backfill and the SD-Policy per day, together with the number of jobs scheduled with malleability. It is visible that the peaks in slowdown are highly reduced all over the simulated time, and, apart from an initial spike of jobs that are selected as mates, the average slowdown for our policy never increases over the static. The reduction of peaks in the slowdown is usually associated with a high number of jobs scheduled with malleability. The total number of malleable scheduled jobs is 20476, the number of mates is 17102, corresponding to 10.3% and 8.6% of the workload.

5.4.3 Evaluation of runtime models

When one of the mates completes before the user requested time, it could lead the other job running in the same node to take the freed cores, but only for part of its job allocation, creating unbalance in its load. We run simulations with the two runtime models presented in Section 5.3.4 for workloads 1 to 4, using SD-Policy DynAVGSD, to estimate lower and upper bounds for this overhead.

Figure 5.8 shows the impact in makespan, average response time, and average slowdown for the two models. The worst-case model increases response time, up to 11% for workload 1 with respect to the ideal model, negligible for

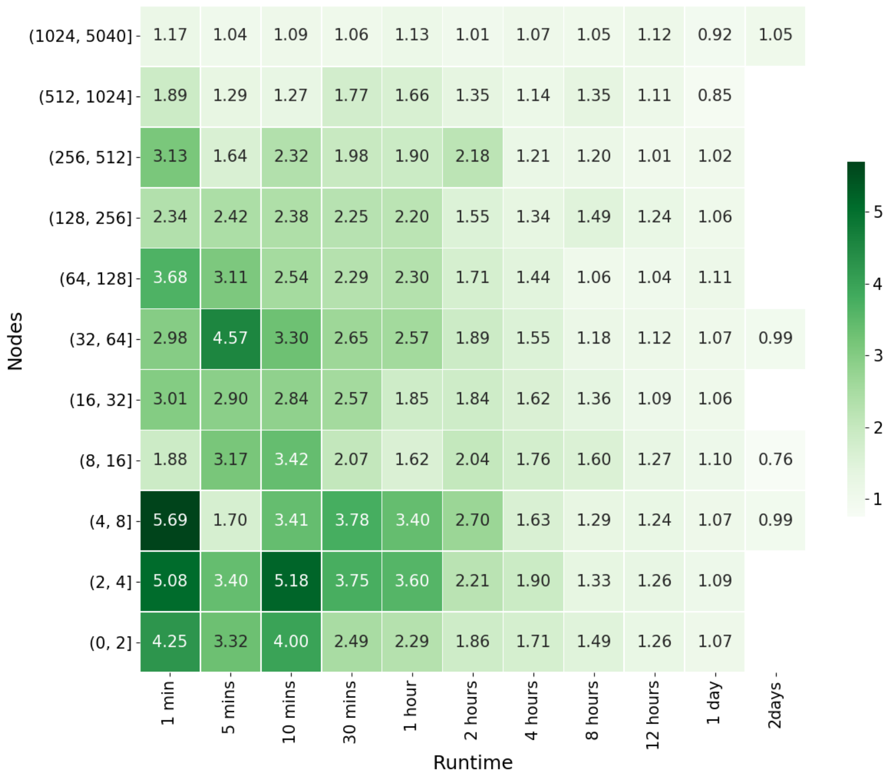


Figure 5.4: Heatmap showing the ratio between slowdown of static backfill and SD-Policy MAXSD 10 using workload 4, for different job categories.

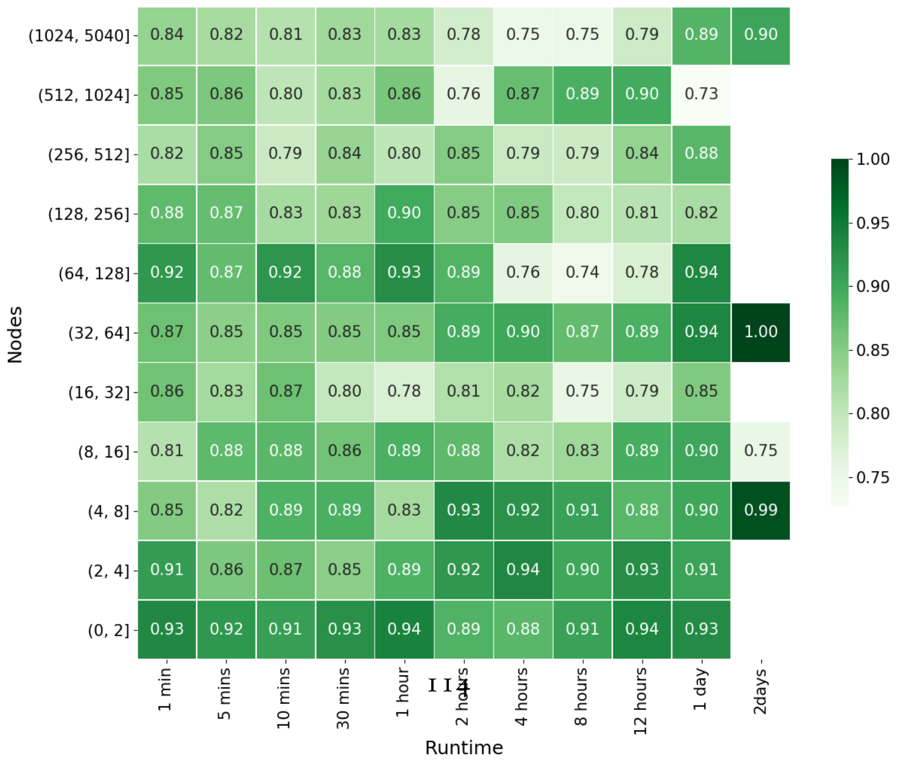


Figure 5.5: Heatmap showing the ratio between run time of static backfill and SD-Policy MAXSD 10 using workload 4, for different job categories.

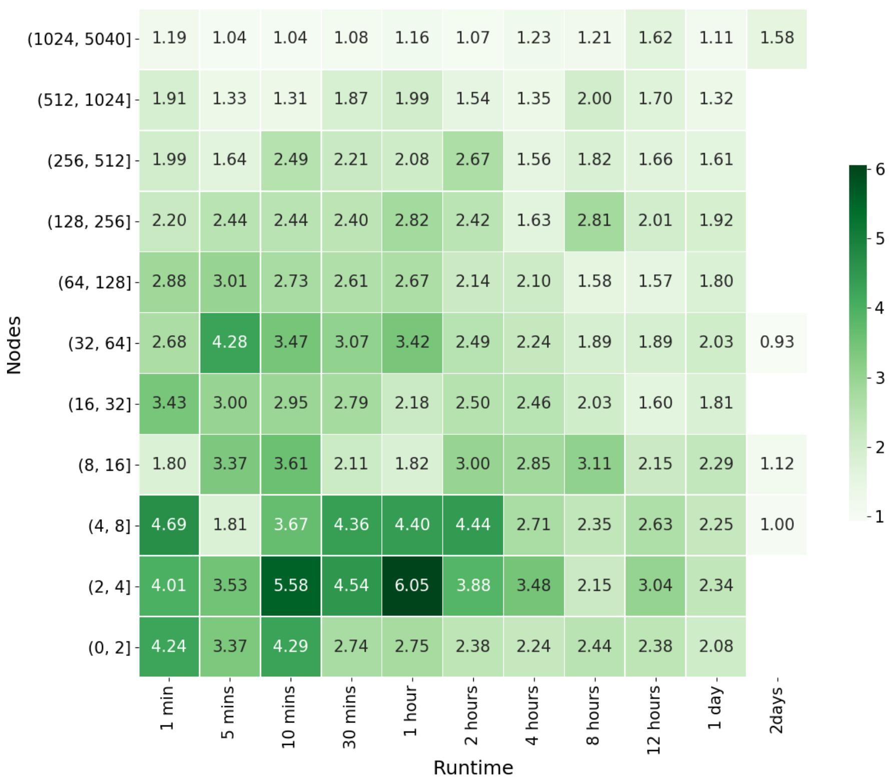


Figure 5.6: Heatmap showing the ratio between wait time of static backfill and SD-Policy MAXSD 10 using workload 4, for different job categories.

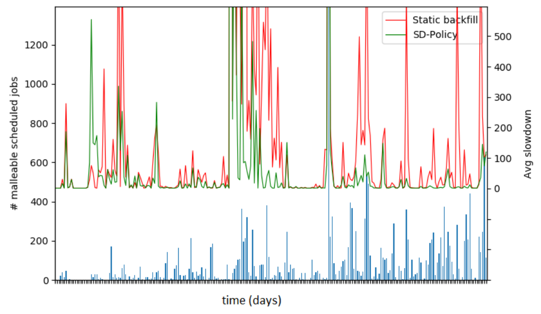


Figure 5.7: Columns represent the number of jobs scheduled with malleability per day, their y axis is on the left. The two lines represent slowdown per day of static backfill and SD-Policy MAXSD 10.

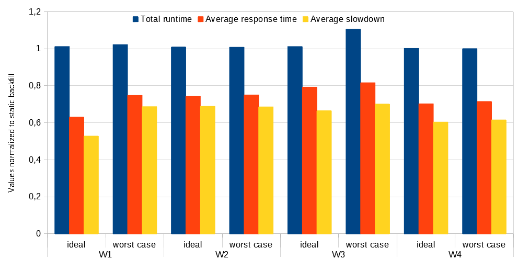


Figure 5.8: Makespan, average response time and slowdown for workloads 1-4, running SD-Policy with feedback, using ideal and worst case model. x axis represent the workloads, metrics are normalized to static backfill simulations.

workload 3 and 4, less than 1.5%. Average slowdown, similarly to average response time, increase by 16% in Workload 1, only 3.5% and 1% in Workloads 3 and 4, while still outperforming static backfill. Makespan increases by 9% in Workload 3, while less than 1% in the other cases. Workload 2 is not affected by using the worst-case model, as the jobs requested times are exact, and it allows the SD-Policy to avoid creating unbalances.

Using a predictive model that lowers the gap between requested time and real runtime would reduce the evaluated overhead to 0, as the time predictions would be more precise.

5.4.4 Analysis of a real run

Having a proposed policy implemented with evaluations in a real system is a complex task. Many publications skip this part, only basing the evaluation on simulation results. This is particularly true in the case of malleability, where all the layers of the HPC software stack need to be aware of the directly connected layers.

On the one hand, simulations are essentials, as they allow evaluating a scheduling algorithm in a large system with extensive workloads, but they do not have the same precision and complexity as a real system. It is not easy, for instance, to evaluate the energy consumption of the SD-Policy and compare it with a real system, as well as modeling a real runtime model for different malleable jobs. We put the effort in developing a working Slurm version, and we made its code available [31], and we used it to evaluate Workload 5, based on the Cirne model using ANL arrival pattern, with 2000 jobs. Jobs ask a maximum of 16 nodes, 768 cores per job, on a system of 49 nodes, 2352 cores. We converted Cirne's log to a real-job submission list for Slurm by using a set of malleable applications. We selected parameters to respect runtime and requested resources, and we generated, using a script, a list of submissions, respecting arrival time. Table 5.2 shows the list of used applications, the type, and the percentage of jobs running them.

We run the workload in the MN4 supercomputer by spawning a Slurm instance inside a job of the Slurm installation in MN4 to have exclusive access to a subset of 50 nodes of the machine to run the workload. The number of nodes and the makespan of this evaluation are constrained by system queue limits, 50 nodes for 48 hours, so we adapted the Cirne model. The controller uses the first node, while the other nodes are the computing nodes.

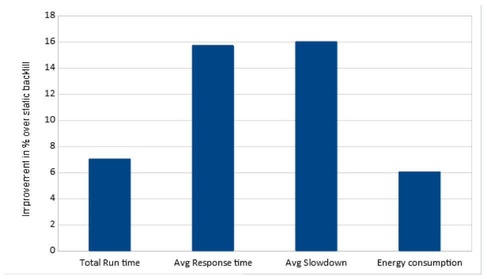


Figure 5.9: Improvement in percentage of the SD-Policy over the static backfill for makespan, average response time, average slowdown and energy consumption.

Results in Figure 5.9 show that makespan decreases by 7%, average response time, and average slowdown by about 16%, compared to the static backfill run. Results show less gain over the simulations because the considered number of nodes is lower in this evaluation. Shorter makespan is obtained by better utilization of nodes' resources, as 449 jobs out of 539 scheduled with malleability have a better runtime than the static execution if we proportionate it to the number of used resources. This behavior is due to two main reasons, already found in Chapter 4. The first reason is the better use of resources by memory bounded jobs, or jobs with memory bounded phases, i.e., during application's initialization and finalization: in this situation, running computing-bounded jobs takes advantage of cores underutilized. The second reason is related to scalability problems of applications that cannot perfectly scale to the high number of cores in the nodes: in this case, using malleability to partition available cores improves performance. As a consequence of better resource utilization, we save 6% of energy over the static approach, which is an important result considering the increasing energy consumption of HPC systems.

5.5 Conclusions and future work

This chapter presented a novel scheduling policy based on malleability and slowdown minimization, the SD-Policy. We described the algorithm and its implementation into Slurm and integrating it into the whole HPC software stack. We presented different parametric evaluations, together with two more complete cases of study, a big simulation, and a real run into a supercomputer. We showed that SD-Policy reduces the response time and the slowdown up to 50% and 70% for the CEA Curie log, while we saw an improvement of nodes utilization that brought 7% makespan and 6% energy reduction for a real run.

Future work will focus on three main points:

1. Integration of online performance metrics collection: by having information about applications, we will perform a better placement of malleable jobs and dynamic adapting SharingFactor.

2. Improved scheduler: feedback mechanism, heuristic, and runtime estimation, based on machine learning approaches.

3. Malleability at MPI level, unlocking new possibilities for the scheduler that can shrink or expand jobs allocations in the number of nodes.

6 Energy driven priority scheduling in heterogeneous multi-cluster environments

With the aim of optimizing performance and energy consumption in environments made up of multiple heterogeneous clusters, this chapter proposes an Energy-Aware-Multi-Cluster (EAMC) job scheduling policy. EAMC-policy is able, by predicting performance and energy consumption of arriving jobs for different hardware architectures and processor frequencies, to optimize the scheduling and placement of jobs, reducing workload's energy consumption, makespan, and response time. The policy assigns a different priority to each job-resource combination so that the most efficient ones are favored, while less efficient are still considered on a variable degree, reducing response time and increasing cluster utilization.

We implemented EAMC-policy in Slurm, and we evaluated a scenario in which two CPU clusters collaborate in the same machine. Simulations of workloads running applications modeled from real-world show a reduction of response time and makespan by up to 25% and 6%, while saving up to 20% of total energy consumed when compared to policies minimizing runtime, and by 49%, 26%, and 6% compared to policies minimizing energy.

6.1 Introduction

Over the last years we have witnessed an explosion of hardware heterogeneity. Those architectures collaborate in computing nodes, which in turn are grouped in clusters. In some cases, different clusters cooperate in the same machine, we call those environments *heterogeneous multi-cluster environments*. A practical example is an old machine still active, like in the case of Marenostrum III[15] and Minotauro[17], or heterogeneous clusters like Power9, running aside Marenostrum IV[16].

While these machines have their own resource manager, the author hypothesis is that interconnecting the clusters under one resource manager can highly improve overall utilization.

Another example is the prototype machine developed in DEEP-EST project [34], proposing a number of heterogeneous clusters collaborating under the same machine, and managed by a unique resource manager. Each cluster has general-purpose CPUs and accelerators, presenting different performances and energy behaviors.

From a broader perspective, with the advance of memories and interconnections, multiple machines will likely be connected and exchanging workloads, resulting in machines with huge potential but also high hardware diversity and more complex management of resources.

On the other side, HPC machines are extremely power demanding, such that the US. Department of Energy set a goal of 20 MW power consumption for an exascale machine. In recent years research and industry put a high effort investigating how to stay under this constraint. At the same time, there is a significant shift from classical performance-oriented to an efficiency-oriented research mentality, with increasing awareness of green computing.

We identified that most of the research [69] is mainly focusing on power scheduling and controlling, with few considerations on the fact that minimizing power does not mean minimizing the energy. While power-awareness is essential, the authors' opinion is that power-saving solutions are related to a constraint in HPC machines' design. On the other side, an energy-aware scheduler needs to optimize energy-performance trade-off continually. For this reason, power-awareness and energy-awareness techniques are not exclusive, and they can be efficiently combined.

Furthermore, as suggested by recent research [30], this whole variety of solutions miss automatic ways of configuring the systems and user's activity. In particular, the job scheduling and resource management layer manages all the resources and gives an interface to users to be able to use them. At the current state of the art, we identified that it is the user's responsibility to specify the type of resource needed, the number of resources, and the best energy settings for the submitted job. This complexity is not acceptable in heterogeneous multi-cluster environments, where a number of computing nodes can be chosen, each one with different characteristics, resulting in confusion and unnecessary knowledge needed for users.

Filling those gaps, we propose the Energy-Aware Multi-Cluster scheduling policy, EAMC-policy. Enabling per-job energy-performance characterization and comparison on heterogeneous environments at job scheduling level, EAMC automatizes jobs' placement and selection of optimal clock frequencies, respecting a trade-off between performance, energy consumption, and response time. For this purpose, we extended an energy model from the related work[11] to characterize applications' runtime and energy on different hardware resources. Based on the extended model, we implemented a multi-objective energy and performance classification. We used it to assign a different priority to each job-resource combination, favoring the optimal ones but keeping the non-optimal to balance the load and reduce response time. By automatizing the process of selecting frequency and optimal hardware, EAMC-policy reduces user's overhead and required expertise, error-prone information, and eventual malicious behavior.

We integrated it into Slurm[9] and the Slurm Simulator[4]. Integrating the prediction model in a job scheduling simulator, we made the Slurm Simulator workload and energy-aware, capable of calculating energy consumption based on the type of application, and not only the hardware like most of the job scheduling simulators. We modeled various applications and benchmarks behavior in terms of energy and performance for multiple architectures, and we distributed them in a workload generated with Cirne's model[25].

We studied the case of a heterogeneous two-cluster environment, each one equipped with processors with different characteristics, i.e., number of cores, frequencies, and cache size. We evaluated performance by changing the distribution of jobs that favor the first and the second cluster. By running job scheduling simulations, we observed improvements in energy consumption by up to 20% while reducing response time by 25% and makespan by 6%.

This chapter is structured as follows: Section 6.2 resumes the state of the art of the energy-aware job scheduling topic, Section 6.3 describes the preliminary work on job modeling, energy tools, and interfaces. Section 6.4 explains the EAMC-policy in detail, while Section 6.5 evaluates it and compare variants of the policy and its parameters with the standard version of the DRMS. Finally, Section 6.6 resume and conclude the chapter, with insights on the future work.

6.2 Related Work

Several surveys exist and resume the work done in the energy field for HPC. Czarnul et al. [30] give an overview of leading energy-aware HPC technologies, diversifying computing environments, device types, metrics, benchmarks, energy-saving methodology, and energy simulators.

Regarding energy-aware job scheduling, Maiterth [69] resumes some of the leading HPC centers' energy strategies and their future directions. Most centers work on powercap solutions, few are working on overprovisioning, and two investigate energy-aware solutions. We found this is a general tendency in the research, with powercap adopted as the primary strategy. In our vision, optimizing energy consumption is a crucial factor, while power limits are a constraint due to machines' powering.

We identified the scarcity of research on energy-aware job scheduling solutions for heterogeneous multi-cluster environments. The authors consider this is an important field to investigate, given the potential and complexity of those environments.

Netti et al. [81] explore the effect of different ways of prioritizing critical resources, the scarcest and most demanded. While this work brings improvement for heterogeneous clusters, it is not energy and workload aware. Extending this work to include energy-aware prioritization could lead to interesting results.

Some policies are based on a model that is aware of manufacturing and assembling variability. Chasapis et al. [24] propose different scheduling policies considering processor manufacturing variability under a power constraint. Moore et al. [77] propose a temperature aware workload placement by detecting cooling inefficiencies that may come from places relatively distant from the temperature sensor, and it tries to minimize heat re-circulations. EAMC used energy model platform is able to use a per-node energy model, so that manufacturing variability can be modeled.

Sarood et al. [92] combine malleability and DVFS to create a scheduling policy that adapts the workload to a strict power budget in over-provisioned systems. Similarly, in Chapter 5, we used malleability and node sharing techniques to reduce response time, makespan, and energy consumption.

Barry et al. [61] explore overprovisioning by powering down nodes in a controlled way using online simulation and controlling system slowdown to keep it to acceptable values. This method works well on non highly loaded systems, but it is difficult to exploit in the opposite case.

Borghesi et al. [13] implement hybrid scheduling techniques for systems under a power cap. They perform a power estimation by using machine learning techniques on historical data instead of a per-job characterization. While this interesting approach does not require the user to specify energy tags, as required by our model, it isn't easy to obtain the same per-job precision as our approach.

In the research of Rajagopal et al. [90], based on this work [87], researchers integrate into Slurm [9] a power-aware scheduler that allows setting powercaps and collects power metrics at runtime to adjust them. It incorporates an interface to choose a frequency based on user hints, and powercap constrains. The solution works with a basic power model using hardware information and user guidance, with estimations representing an upper limit and not the real power consumption. Our research differs since it optimizes energy, not power. Moreover, no energy input from users is required to set the best frequency, but it is predicted and set automatically using online metrics collection and historical data. In their work, DVFS is used when reaching the powercap, while in our work, we use DVFS to select the optimal performance-efficiency trade-off for each job. In future work, the two policies could be combined, i.e., by choosing optimal frequency for performance and efficiency while assuring a powercap.

Auweter et al. implemented two policies: energy-to-solution and best-runtime, to select the most suitable frequency for jobs [8]. Users are required to specify tags for similar jobs to be able to use the proper job characterization. This work is the base of our research, as we implemented a similar description of runtime, energy, and power for jobs. On our side, our instrumentation allows us to verify the correct use of energy tags and recalculate optimal frequencies at runtime. We implemented it in Slurm. We extended the energy model to characterize jobs on heterogeneous clusters. We propose the EAMC-policy to prioritize the most efficient architectures following an energy-performance trade-off strategy instead of energy-to-solution or best-runtime.

6.3 Preliminary work

EAMC-policy selects optimal energy configuration and places jobs based on energy and performance estimations. EAMC requires estimating the job's runtime and power consumption at scheduling time before the job start. In this section, we describe the designed framework that allows EAMC-policy operation.

6.3.1 Job power and runtime modeling

To estimate runtime, power, and energy consumption for different processors, frequencies, and applications, we revised available energy models from the state of the art. Various energy models and tools for energy prediction are proposed in the literature, with distinct complexity. Some of them are general, based on hardware information only, whereas the chosen [11] is able to model different applications' behavior for different architectures. The model needs two inputs:

1. application data: a collection of metrics that characterize applications. The model uses runtime, average consumed power, average number of memory transactions per instructions (*TPI*), and average cycles per instruction (*CPI*) at the reference frequency f_{ref}.

2. hardware coefficients: a set of parameters obtained by running a learning phase based on a set of benchmarks. The chosen model uses six coefficients learned using linear regression: *A, B, C, D, E, F*.

Equations 6.1, 6.2, and 6.3 describe the mathematical model that estimates power consumption P and runtime T at the frequency f, starting from a default frequency f_{ref} at which application data is given.

$$P(f) = A * P(f_{ref}) + B * TPI(f_{ref}) + C \tag{6.1}$$

$$CPI(f) = D * CPI(f_{ref}) + E * TPI(f_{ref}) + F \tag{6.2}$$

$$T(f) = T(f_{ref}) * \frac{CPI(f)}{CPI(f_{ref})} * \frac{f_{ref}}{f} \tag{6.3}$$

To implement and use the energy model in a real-world environment, we used Energy-Aware Runtime (EAR) [29].
EAR is a framework that provides an energy-efficient solution for HPC clusters. It includes monitoring, accounting of the applications' performance, and it integrates energy optimization via DVFS at node and cluster level. It is deployed on SuperMUC-NG [65] and implements the previously described energy model.

EAR was used to learn hardware coefficients *A, B, C, D, E, and F* to model two computing nodes equipped with general-purpose processors with a different number of cores and frequencies, as described in Section 6.5.

In a non-simulated environment, EAR collects application data dynamically at runtime and store them in the application database, to be used by the energy model. In our test simulated environment, we use static data collected by running several applications with diversified computing and memory behaviors in a previous phase. Used applications are described in Table 6.1.

To evaluate the proposed policy in our test system, we used and extended an interface developed in the context of the DEEP-EST project [80].

The interface is configurable with different machine configurations and energy models. DRMS uses it to retrieve energy, time, and power estimations. It can be used in a real environment as a standard interface that abstracts underlying energy models or to simulate tools like EAR in simulated environments.

To run our simulated environment, we implemented the chosen prediction model by extending the interface. Each hardware configuration is identified by a model id, a series of hardware coefficients, a node configuration, and a range of frequencies. Different hardware configurations can exist for the same node, e.g., a node using or not using the accelerator or a processor using or not using the vector extension.

6.3.2 Application Database

The Application Database *AppDB* stores application metrics collected at runtime for each hardware configuration. For each tuple, appDB contains fields as the appID, the modelID, the necessary application metrics.

In a real system, application database entries for jobs are created on the fly. At first run, there is no energy information associated with the job. Still, metrics are collected by EAR, which creates an entry in the DB associated to the user specified appID. For the first run, optimal frequency is not estimated previously, but it is set at runtime as soon as data is available.

In the case of multiple clusters but with application data related to only one of the clusters, performance and energy consumption can be estimated from data in the appDB in combination with the hardware model, or they can be evaluated when the application runs for the first time on the hardware.

In our simulated environment, for the sake of simplicity, the appDB is static and available at the beginning of simulations. To obtain the application's data, we run applications in different architectures, we collect necessary metrics using the EAR accounting and store them in the database.

6.3.3 Model extension for multi-cluster environments

When collecting metrics, while hardware counters and average power are automatically collected by EAR, the application's runtime is potentially variable in each run.

From the DRMS point of view, it is not straightforward to estimate the runtime parameter with precision, so commonly, users are asked to give a requested time for the job. These values are usually far from the real job's runtime, and it can also be a default value in the case the user doesn't specify it.

Having requested time as the only time-related information for the job, the energy model assumes this value relates to the primary partition. To calculate a requested time for the other partitions, we extended the energy model to learn a seventh parameter, *time_coefficient*. time_coefficient represents the ratio between the main and another partition runtime. This parameter is estimated by running the same application on both hardware architectures. Future work will still consider more complex models that do not need this phase, e.g., based on already collected hardware metrics.

6.3.4 User interface for job submission

When submitting a job, users can avoid specifying which hardware architecture they want to use if they accept that the job will run on nodes automatically selected by the scheduler.

We evaluated EAMC on processors from the same family, so we automatically consider the app can run on every available hardware architecture, thanks to binary compatibility. Considering similar architectures is not far from the case of more heterogeneous ones, given the high effort of research and industry in supporting interoperability, hardware-independent programming languages, and libraries. In the case there is no binary compatibility, users can specify different binaries for the same appID. This information can be stored in the appDB or contained in the job script in a format readable by a job script parser.

Users are still in charge of specifying the number of requested nodes and processes. The number of threads or processes per node can be configured by using Slurm plugins or DROM in a more intelligent fashion. If a memory requirement is specified, and some nodes cannot satisfy this requirement, a new number of nodes will be calculated.

Using more heterogeneous architectures, on the other side, increases the complexity of parameters for the job. As an example, the number of requested nodes needs to increase to maintain a reasonable runtime when switching from a fast and power-consuming resource to a slow and less demanding one. Future work will try to take into account those details by increasing the complexity of the hardware models.

It is also the user's responsibility to specify an *appID* in the job script to recognize jobs with the same behavior. It is important for users to specify different appIDs for a same application using different input, whenever the input modifies the job's behavior. On the other side, EAR can verify appID related history corresponds once the job starts, giving the DRMS instruments to detect and penalize fraudulent users' behavior. Suppose runtime collected metrics do not conform to stored information in the appDB for the specified appID. In that case, the application keeps running, and a new frequency is calculated based on runtime data. If the user repeatedly uses wrong appIDs, the DRMS can detect it by analyzing his history of jobs, return a warning message, or take additional actions.

6.4 Energy-aware job scheduling in multi-cluster environments

EAMC-policy is based on priority backfill, with energy predictions influencing the priority of arriving jobs. Its code is publicly available [71]. This section describes the implementation of the policy.

In the policy presentation, for simplicity, we use the term *partitions* as a logical representation of resources inside the DRMS. While ideated with the introduced concept of multi-clusters, the policy is generic, independent of the physical and logical representation of the hardware, either physically or non physically separated, grouped, or non grouped in partitions.

6.4.1 Problem definition

We model the proposed policy as an online job scheduling problem. While this problem usually has as objective function the minimization of the makespan, in our case, we describe it as a multi-objective optimization problem with objectives of the reduction of makespan and energy consumption.

The most efficient solver for job scheduling problems in the literature is the backfill algorithm. We used a backfill version with priorities, where each job gets assigned a priority based on several factors, e.g., arrival time, requested number of nodes.

To make backfill energy aware, we included a runtime and energy classification of jobs as a further factor for priorities, thus influencing the scheduling order. More precisely, we calculate a priority for each *job-partition* combination *jp*, defined as a job *j* running on a partition *p* in two phases using Equation 6.4 and 6.5. The first equation is used to estimate the optimal processor frequency f_{opt}, while the second is used to classify and assign a different priority to each p for j.

Firstly, in Equation 6.4, given a *jp*, with the objective of calculating the optimal frequency f_{opt}, we define the metric $dist_{jp}$ as the euclidean distance from the origin in the Euclidean space representing time and energy normalized to the minimum value of the respective metrics (E^{min}_{jp}, t^{min}_{jp}) in the prediction space of *jp*. *t_weight* parameter increases or decreases runtime estimation weight over energy.

$$dist_{jp} = min\sqrt{\frac{E_{jp}(f)}{E_{jp}^{min}}^2 + t\_weight * \frac{t_{jp}(f)}{t_{jp}^{min}}^2}, \quad \forall f \in [f_{min}, f_{max}] \tag{6.4}$$

As a second point, in Equation 6.5, given multiple job-partition entries for a job, to classify partitions, we used the Euclidean distance again to calculate *part_eff*, in this case normalizing energy and time among all partitions of the job *j* (E^{min}_{j}, t^{min}_{j}) at frequency f_{opt} calculated for each partition with Equation 6.4. As a result, *part_eff* can be used as an indicator of performance and efficiency, and it can be used to influence the *jp* priority. Again, *t_weight* affects the weight of time over energy.

$$part\_eff = \sqrt{\frac{E_p(f_{opt})}{E_j^{min}}^2 + t\_weight * \frac{t_p(f_{opt})}{t_j^{min}}^2}, \quad \forall p \in partitions \tag{6.5}$$

6.4.2 Proposed policy

Figure 6.1 and 6.2 represent two variants of the EAMC-policy. First, a job, represented on the top left, is submitted. At submission time, the policy assigns the priority to arriving jobs and, in the case of multiple partitions, to each job-partition combination. We extended this part of the DRMS with the Energy-prediction Priority Module (EPM). Once a different priority for each job-resource is assigned, an EAMC-Scheduler (EAMCS) schedules jobs, from a unique priority queue, in order of priorities. Following on, we describe EPM and EAMCS components.

EAMC-Scheduler (EAMCS) and Energy-prediction Priority Module (EPM)

EAMCS, based on EASY-backfill [79], tries to schedule one job-partition per time in order of priority. Using backfill allows scheduling jobs considering a trade-off between system energy, performance, and average wait time. As an example, if the higher priority job cannot run due to lack of resources or a big enough time window for backfilling it, the lower priority job-partition can start if enough resources are available. While not being the best solution in terms of energy and runtime, it is optimal when considering the wait time.

The right equilibrium of jobs running on the favored and non-favored partition can lead to optimal performance in the time-energy dimension. Extremes behaviors, where jobs poorly performing are discarded, are managed in one of the policies alternatives described in this section, with details on EAMCS.

Listing 6.1 describe EPM implementation in detail. In the first inner loop, EPM gets runtime and energy predictions using the energy interface and energy models (line 5). The parameter *time_coefficient* is used as a multiplier to predict runtime and energy consumption for different partitions. Then, using Equation 6.4, it individuates the best frequency for each job-partition, setting it as the default value (line 6). The same loop calculates ${E^{min}}_j$ and ${t^{min}}_j$.

Once frequencies are picked, the second loop evaluates and sort partitions according to Equation 6.5 (lines 11-13). In the last loop, *set_priority()* function (line 16) is in charge of assigning a priority to each partition according to the selected strategy. In the end, the job-partition couple enters the job priority queue.

We developed three strategies that implement different ways of assigning priorities to job-partitions, depending on how aggressive is the policy in favoring optimal partitions over arrival order of jobs:

- **EAMC-Reorder**: it changes the order of partitions for *the same job*, prioritizing job-partition with better performance. As a consequence, EAMCS maintains the arrival order of jobs while reordering partitions within the job. Jobs-partition entries are queued similarly to job2 in Figure 6.1.

- **EAMC-PriorityInc**: it assigns a different priority to each job-partition based on the position in the sorted partition list. As shown in Figure 6.2, the two job-partition are inserted in different points of the job queue,

Listing 6.1: Energy Prediction Module (EPM) implementation

```
for each j in submitted_jobs:
    if (!j->partitions)
        j->partitions = get_partitions(
            j->app_id);
    for each p in j->partitions:
        EA-API_get_predictions(p,j->app_id);
        p->best_f_val = get_optimal_freq(p,
            p->energy, p->time);
        if (p->energy < emin)
            emin = p->energy;
        if (p->time < tmin)
            tmin = p->time;
    for each p in j->partitions:
        p->part_eff = get_part_efficiency(
            p->energy,p->time,emin,tmin);
    sort(j->partitions,cmp_part_eff());

    for each p in j->partitions:
        p->priority = set_priority(p,policy_type);
        enqueue(job_queue,j);
```

creating two separate queues, one with preferred partitions, the others for non-optimal partitions. In this case, EAMCS favors optimal job-partitions over non-optimal ones, giving less weight to arrival time order.

- **EAMC-PriorityInc-V**: similarly to PriorityInc, based on the same figure, it assigns a different priority to each job-partition, based on the sorted partition list, with the difference that the priority is lowered if the evaluated metric is below a certain threshold. We set the threshold to 35% from the optimal part_eff metric. As an example, if *j1p1* exceeds the threshold and *j2p2* does not, *j1p1* would go at the bottom of the queue. EAMCS first checks all optimal job-partition entries, then the second choices, and finally all the job-partitions below the threshold.

6.5 Evaluation

The evaluation is based on simulations using the BSC Slurm jobs scheduler Simulator [4]. We modeled a workload of 5000 jobs, with a makespan between 10 and 15 days, depending on applications distribution and used frequency, using Cirne model [25] configured with ANL arrival pattern. Jobs' average requested nodes are 18.29, and jobs' average duration is 3.65 hours at the highest frequency. We run the workload in the following simulated clusters:

1. p1: 512 nodes equipped with: 2x Platinum 8168 CPU [2.70GHz-1.20GHz] 24C, TDP 205W and 12 x 16GB DDR4 SDRAM

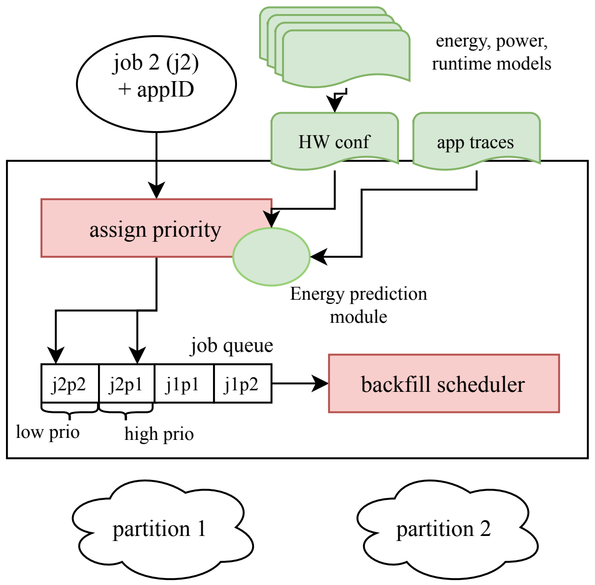


Figure 6.1: Job submission and scheduling phases for EAMC-Reorder policy. The big-box represents the DRMS. Red boxes are modified parts, and green boxes are energy prediction components. With this policy variant, jobs-partition entries are reordered keeping the same priority in the queue.

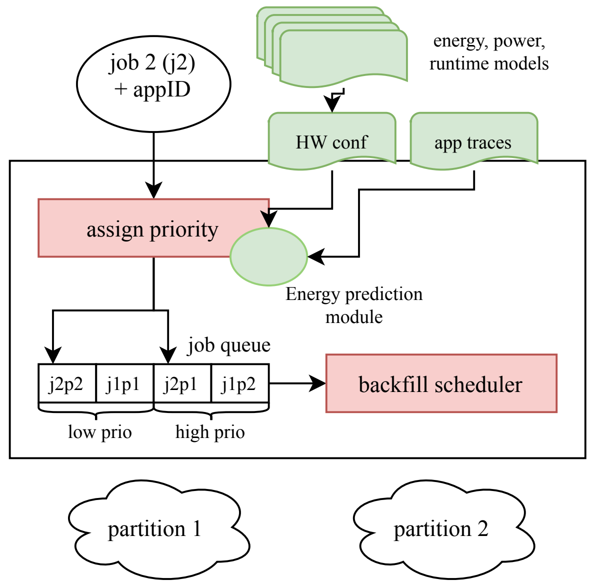


Figure 6.2: Job submission and scheduling phases for EAMC-PriorityInc policies. The big-box represents the DRMS. Red boxes are modified parts, and green boxes are energy prediction components. With these policy variants, job queue is separated in a higher priority and lower priority queues depending on energy and performance evaluations.

2. p2: 512 nodes equipped with: 2x Gold 6254 CPU [3.10GHz-1.20GHz] 18C, TDP 200W and 12 x 32GB DDR4 SDRAM

3. p3: 512 nodes equipped with: 2x Gold 6148 [2.4GHz-1GHz] 20C, TDP 150W and 12 x 16GB DDR4 SDRAM

The number of modeled hardware architectures was limited by the available architectures and permissions needed to collect necessary data. We sized the number of simulated jobs and system size according to the time limits imposed by the testing environment.

We run the EAR learning phase to get coefficients for the simulated nodes on Lenox[62] cluster. We used data from eight applications, running them on one node using all the available cores. [*]. Table 6.1 describes the set of applications, presenting different memory and compute profiles, reported through average cycles per instruction (CPI), memory bandwidth (GB/s), the ratio of the runtime and power between partitions, and the ratio between the delta between the maximum and minimum runtime and the delta of frequency Δr/Δf. The higher the value of Δr/Δf, the more the runtime scales with the frequency, e.g., ep.D runtime is highly affected by changes in the processor's frequency, while STREAM is not.

We simulated two different systems:

1. A system made up of p1 and p2.

2. A system made up of p1 and p3.

Comparing p1 and p2, while in terms of performance seven out of eight applications favor the first partition, regarding power, the first partition showed

[*]Applications data is available [72]

slightly higher power consumption for the CPU component, more than nominal 5 Watts, while the second partition showed up to double DRAM consumption compared to the first. From an energy perspective, applying Equations 6.4 and 6.5, for the tested t_weight values, the first seven apps favor the first partition, while app 8 favors the second. For this comparison app8, STREAM, is a weak scaling benchmark, where an amount of memory is allocated per-process, i.e., per-core. The second partition, having a reduced number of cores, has fewer data to manage, explaining the difference in runtime while showing similar hardware metrics. This is the typical behavior of some memory-bounded applications.

For the same comparison, we distributed the applications among the workload by randomly drawing samples from three distributions:

1. 13% benefits from partition 2: while the distribution is uniform among all applications, 87% of the workload favors the first partition.
2. 33% benefits from partition 2: In this 33% of jobs run app8, which favor the second partition, the remaining 67%, uniformly distributed among remaining apps, prefer the first.
3. 50% benefits from partition 2: finally, this case evaluates an even load among the two partitions in terms of the number of jobs per favored resource. 50% of jobs run app8.

Comparing p1 and p3, at base frequency five out of eight applications run optimally on partition 1, while at the optimal frequency all the applications run optimally on the same partition. In this case, we used strong scaling version of STREAM, that uses a fixed amount of memory.

AppID	Name	CPI	GB/s	Runtime (s)	Power (W)	$\Delta r / \Delta f$ (s^2)
1	lu.C	0.67 / 0.66 / 0.57	70.95 / 63.73 / 59.7	52.49 / 57.45 / 62.78	386 / 367 / 331.25	2.38 / 2.87 / 4.18
2	ep.D	0.60 / 0.60 / 0.60	0.03 / 0.04 / 0.03	64.20 / 74.20 / 86.59	349 / 337 / 290.32	5.67 / 6.37 / 8.97
3	bt-mz.C	0.38 / 0.39 / 0.38	23.33 / 19.96 / 17.7	43.95 / 51.32 / 57.70	417 / 411 / 342.73	3.52 / 4.05 / 5.65
4	sp-mz.C	0.44 / 0.50 / 0.42	66.09 / 52.24 / 51.17	46.05 / 59.76 / 58.58	460 / 427 / 375.47	3.07 / 3.34 / 4.64
5	lu-mz.C	0.62 / 0.62 / 0.63	19.87 / 21.14 / 18.89	59.21 / 61.41 / 65.98	355 / 360 / 308.81	4.75 / 4.70 / 6.33
6	ua.C	0.99 / 0.95 / 0.87	51.29 / 49.05 / 45.5	46.96 / 53.11 / 54.51	368 / 360 / 324.6	1.74 / 1.84 / 2.64
7	DGEMM	0.40 / 0.40 / 0.38	66.43 / 62.76 / 50.87	47.40 / 52.60 / 61.97	491 / 497 / 366.25	3.29 / 3.08 / 4.09
8	STREAM	4.46 / 3.67 / 3.55	138.87 / 137.47 / 136.86	107.55 / 81.43 / 110.34	381 / 365 / 330.52	0.16 / 0.06 / 0.13

Table 6.1: Set of applications and their characteristics. Metrics specified in the order of partitions, in the format p1 / p2 / p3

The evaluation follows in this section, where we analyze the performance of the developed policies compared to *Base Slurm*, and policies inspired by Auweter [8].We call the two polices *Min_energy* and *Min_runtime*.

Base Slurm is configured to run jobs at default frequency, i.e., the maximum frequency, and each job is submitted only to the optimal partition.

Min_runtime runs jobs at the frequency that minimize the runtime, while Min_energy runs jobs at the frequency minimizing the job's consumed energy. For those policies, jobs are submitted to both partitions, and each job-partition gets the same priority. The scheduler tries to run the job in the default order, first in the first specified partition, and immediately after in the second, not being able to establish the favored one.

Evaluated metrics:

- Makespan
- Average response time
- Sum of the jobs' consumed energy: sum of individual jobs' consumed energy, not including idle nodes.
- Sum of applications runtime: Sum of individual jobs' runtime, used to understand the impact of EAMC-policies on the jobs' performance.
- Average frequency: Average of selected frequencies on total workload, or per partition.
- Percentage of applications in the optimal partition: The percentage of jobs scheduled in the favorite partition.

We first analyze the system p1/p2. We compare the EAMC policies variants to Base Slurm, Min_runtime, and Min_energy. We then compare Min_runtime and Min_energy to our policy when changing the applications' distribution in the workload. Finally, for system p1/p3, we analyze the 13% case. For both systems, p1/p2 and p1/p3, we analyze the performance when changing the t_weight parameter, giving more importance to performance over energy efficiency.

6.5.1 Comparing to Base Slurm

This section compares EAMC-policy variants' performance with Base Slurm, where jobs are submitted only to the optimal partition, and with Min_runtime and Min_energy, where jobs are submitted to multiple partitions with a frequency that minimizes runtime or energy. In Base Slurm, jobs run at the default frequency, i.e., the maximum frequency, system p1/p2 is considered.

Figure 6.3 shows the improvement in percentage over the Base Slurm for the analyzed metrics. Applications are equally distributed among the workload (13% case), and EAMC is configured with *t_weight*=1.

Analyzing makespan and response time, asking all the available partitions has a considerable impact on all the evaluated scenarios, independently from the job-partition priority, and up to 39%, and 64% for Min_runtime. EAMC performs similarly while achieving up to 9% energy saving compared to Min_runtime. This indicates that the more the systems are interconnected, the better the utilization of resources, response time, and makespan. Besides, workload and hardware aware scheduling can save energy without sacrificing system performance.

Base Slurm is able to schedule all the jobs in the optimal partition, obtain-

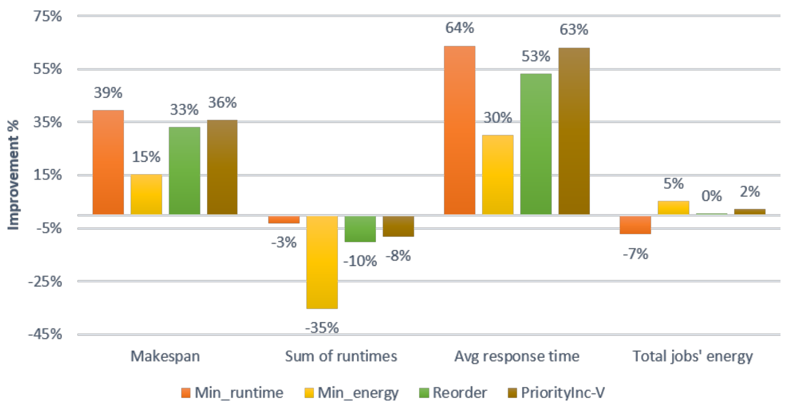


Figure 6.3: Savings in terms of makespan, average response time, sum of runtimes and sum of jobs' energy over Base Slurm for EAMC policies, Min_runtime and Min_energy.

ing the best runtime for each job, but at the cost of the time needed to wait for the favoured architecture. This is costly in the case of unbalanced load among the partitions, as in this evaluation. In the general case, partitions will hardly be constantly balanced, making this policy too static and not suitable for real systems. Min_runtime achieve very good results for time metrics, at the cost of consumed energy, up 12% compared to Min_energy. Min_energy obtain 5% energy saving while not scheduling jobs in the optimal performance, but it gives up half of response time compared to other policies.

Regarding EAMC policies, we can observe that Reorder underperforms slightly compared to other variants because favoring the job's arrival time order leads to fewer possibilities for the scheduler to favor optimal job-partitions. PriorityInc and PriorityInc-V performs similarly to Min_runtime in terms of makespan and response time while increasing energy savings by 9%. Increased jobs' runtime, given the lower processors' frequencies, is compensated by the workload aware scheduling of jobs.

Min_energy and Min_runtime only schedule jobs to the first available partition, achieving a low number of running jobs in the optimal partition, as we can observe in Figure 6.5. EAMC policies, particularly PriorityInc versions, can schedule more jobs in the favored partition, especially in p2, running almost 100% of STREAM, achieving shorter runtime and higher energy savings.

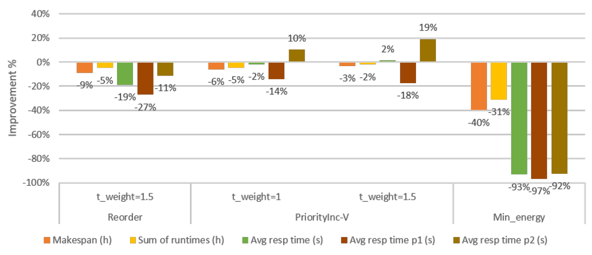
6.5.2 Changing t_weight parameter for three apps distributions

As previously commented, seven out of eight applications in our set favor the first partition. In this evaluation, we test the different application distributions: 13% (uniform distribution), 33%, and 50% of jobs benefiting p2. The objective is to evaluate performance for different load levels per optimal partition by changing the number of jobs running app with id 8, STREAM. We run all the EAMC policies, and we tested two values for t_weight parameter for Equation 6.4 and 6.5: 1 and 1.5. The latter increases the weight of performance over energy in the EPM.

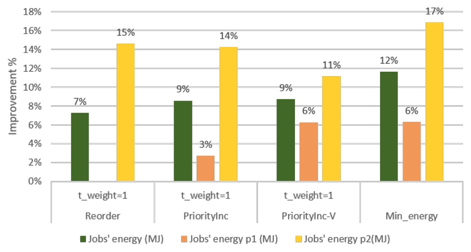
This evaluation focuses on the trend between the three workload distributions, presenting more insights on the 33 and 50 scenarios, since we already analyzed workload 13. Figure 6.4 reports a summary of time and energy metrics for Workload 13, 33, and 50, normalized to
Min_runtime, and expressed as increase in percentage over it.

At first glance, moving from distribution 13 to 33 and 50, we notice increasing performance for EAMC when comparing to the other two policies. In those cases, the scheduler can schedule jobs in the optimal partition without sacrificing the response time, as Base Slurm does.

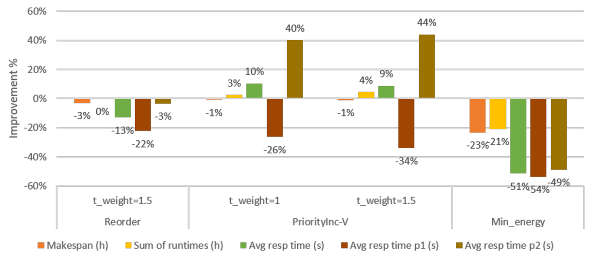
We observe a reduction of response time, by up to 14% and 25% respectively for workloads 33 and 50, when using PriorityInc-1.5. Looking at sin-



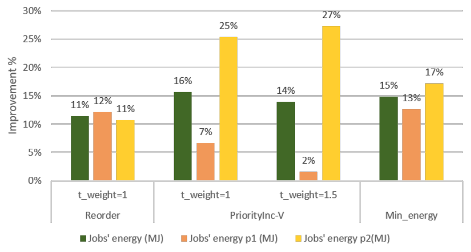
(a) Time metrics for Workload 13%



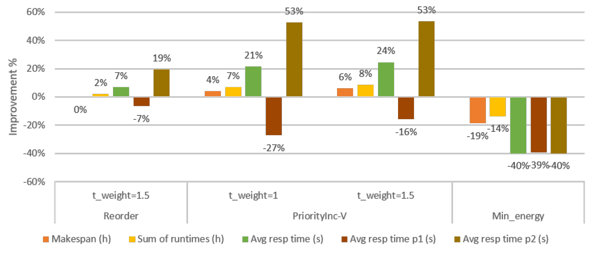
(b) Energy metrics for Workload 13%



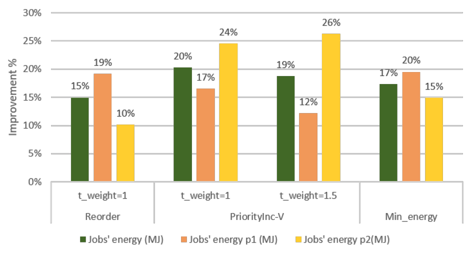
(c) Time metrics for Workload 33%



(d) Energy metrics for Workload 33%



(e) Time metrics for Workload 50%



(f) Energy metrics for Workload 50%

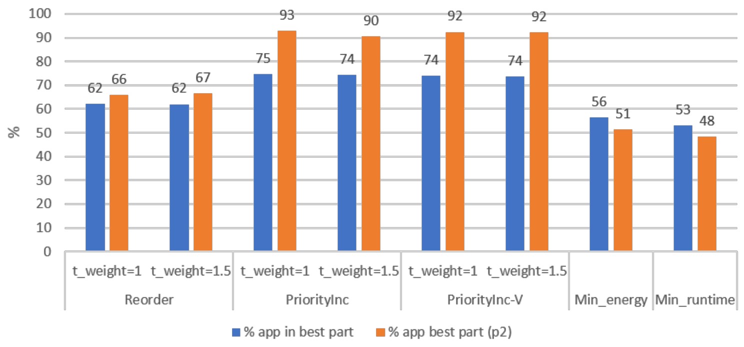
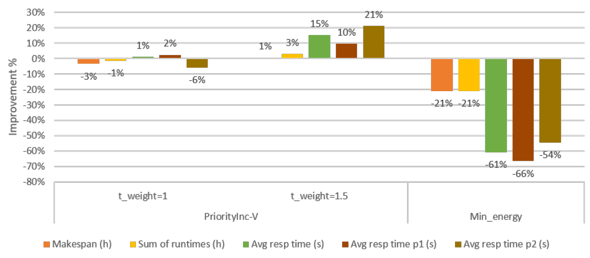
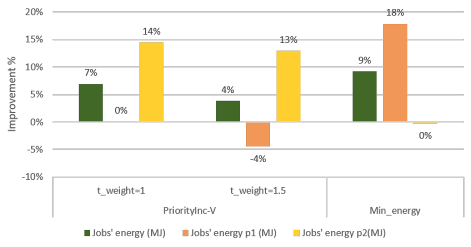


Figure 6.5: Average percentage of applications scheduled in the optimal partition for workloads 13%, 33%, and 50% for the system p1/p2. Presented the average of the two partition, and the partition p2.



(a) Time metrics for Workload 13%



(b) Energy metrics for Workload 13%

Figure 6.6: Main time (left) and energy (right) metrics, normalized to Min_runtime policy and reported as improvement in percentage over it for the second system configuration (p1 and p3). Reported PriorityInc-V, and Min_energy.

gle partitions, we notice that p2 shows the greatest improvement in response time, up to 51% decrease for the same policy, and up to 53% in PriorityInc-V-1.5. This behavior is compensated by an increase of response time in p1, given the lower processor frequency, up to -14% and -16%. When using t_weight=1, this value gets to -36%, but still increasing overall response time thanks to better usage of p2. While the Reorder policy doesn't reach other EAMC variants in total average response time, it shows more balance in response time between partitions. Gains in response time are accompanied by good results for makespan, ranging from -5% to 6% compared to Min_runtime.

Min_energy does not perform well in time-related metrics. It increases response time and makespan by up to 51% and 23% compared to Min_runtime, and by up to 65% and 25% compared to PriorityInc, for workload 33 and 50.

Min_runtime energy consumption increases by 17% and 21% for workloads 33 and 50 with respect to Min_energy. EAMC policies show up to 4% of improvement over Min_energy and up to 20% over Min_runtime. Energy savings are higher in the second partition, where app8, with almost null frequency scaling, can benefit from a reduced runtime and energy when running over it. PriorityInc-V only slightly improves energy savings than PriorityInc, showing a low number of jobs that perform particularly bad on a single partition. We identified that app2, ep.D, was affected by the policy variant's threshold out of the eight applications. Due to EAMC scheduling, the number of jobs running in the optimal partition reaches up to 82%, compared to about 55% in the compared policies. This improves the jobs' runtime and energy consumption, unlocking the described results. The average frequency for EAMC policies is close to Min_runtime values concerning p1 and close to Min_energy in p2. With the increasing number of app8 in workloads 33 and 50, the average frequency in p2 decreases, as this app's optimal frequency is the lowest.

As a final remark, in Workload 50 Base Slurm response time and energy consumption improve over Min_runtime by 19% and 10% at the cost of 7% of makespan. This is a specific situation where the load is overall balanced among partitions. We observed that in general, Base Slurm could not adapt to changes in the load. Besides, while the load is balanced overall, it is not balanced continuously, so EAMC can take advantage of temporary unbalance, further reducing time and energy metrics.

To conclude, PriorityInc-V-1.5 achieves better time-related metrics, with little to none impact on energy consumption, being our favorite policy configuration. Thanks to the ability to reduce the priority of jobs that have a high impact on performance when running on the secondary partition, it further improves the scheduling. This policy would be our pick for systems on which the performance has primary importance. PriorityInc-1 and PriorityInc-V-1 achieve higher energy saving at a small cost of response time. This pick will be the favorite if energy savings or powering costs have an essential importance.

6.5.3 Changing t_weight parameter for system p1/p3

As a final evaluation, Min_energy and Min_runtime to EAMC for the second system configuration. For this evaluation, at maximum frequency, 5 out of 8 applications perform optimally on p1, 3 on p3, while at optimal frequency, all applications perform optimally on p1. Results for Workload 13% are presented in Figures 6.6a and Figures 6.6b.

For this evaluation, in the three scenarios the applications are scheduled first on the optimal partition, so EAMC Reorderning is not as effective as in the precedent evaluation.

PriorityInc-V is still able to improve performance in time and energy by reducing the priority of less performant partition. PriorityInc-V with t_weight=1 improves energy by 7% in average, while time metrics are comparable. On the other side, PriorityInc-V with t_weight=1.5 shows 15% improvements in response time, and 4% of improvement of energy consumption. While Min_en achieve 2% improvement over PriorityInc-V in terms of total energy, it increases the response time be 61%, a very high overhead.

6.6 Conclusions and Future Work

In this chapter, we presented a new scheduling policy that can estimate the energy and runtime of jobs and use this information to optimize their placement in heterogeneous multi-clusters environments. We used Energy-Aware Runtime and Energy-Aware APIs to connect and monitor applications' behavior. We extended EAR energy model to predict runtime and energy when applying DVFS for multiple hardware architectures. We used those tools in EAMC-policy, made up of an Energy Prediction Module that predicts energy and runtime for arriving jobs and assigns a different priority to each job-resource combination in heterogeneous environments. Our EAMC Scheduler, based on priority backfill, schedules jobs-resource entities in the same priority assigned by the EPM. We developed three variants of the policy, and we compared them with the Slurm and policies based on the state of the art. EAMC-Reorder is able to reduce response time and energy consumption favoring job's arrival order. EAMC-PriorityInc variants are able to optimize the scheduling to further improve makespan, response time and energy savings, up to 6%, 25% and 20% compared to policies minimizing runtime, and up to 26%, 49%, and 6% compared to policies minimizing energy. As a future work, this policy will be deployed and tested on a production machine, the DEEP-EST prototype, with experimental workloads, giving the opportunity to generate, collect, and analyze metrics generated by the system as a feedback for the policy. As a second path, EAMC-policy will be expanded and tested with more heterogeneous systems. To achieve it, a more complex job and hardware modeling need to be performed, not only modeling energy and runtime at different frequencies, but improving the modeling of the behavior of applications for each architecture in terms of performance and amount of resources needed. Finally, it is interesting to explore how the presented policy behave when integrated with other power-saving techniques, e.g. overprovisioning and powercapping.

"At last I will devote myself sincerely and without reservation to the general demolition of my opinions."

Rene Descartes

7 Conclusion

In this thesis, we individuated and analyzed three main trends in the development of future HPC machines: research in memory-centric workloads, increasing power demand, increasing hardware heterogeneity.

Based on these trends, we further individuated some key features essential for future machines' success. Our claim is that three concepts need research effort: awareness, dynamicity, automatization. We investigated the adoption of those ideas with a primary focus on the job scheduling layer.

We proposed three novel contributions:

- DROM, an interface that unlocks vertical malleability for applications and programming models, and dynamic control for resource managers.

Results show negligible overhead in the analyzed use cases, where one or multiple applications use DROM to share node's resources. We learnt that system could benefit when jobs share the same nodes, especially when combining memory-bound with compute-bound applications, or when they are not able to scale to all the node's resources.

- SD-policy, a malleable job scheduling policy that uses jobs' slowdown as feedback for deciding job's sizes dynamically. The strategy uses DROM to make room for new jobs to run by shrinking the ones with a low slowdown, only if the new jobs can finish earlier when scheduled with malleability. SD-policy can significantly benefit the system response time, especially with low-latency jobs, or combining computing-bound and memory-bound workloads. As a side effect, it redistributes the slowdown more evenly among the workload. We learnt that user requested time is the main source of uncertainty in using a system-guided feedback to configure the scheduler, but that at the same time exact requested time values do not grant the optimal scheduling flow.

- EAMC-policy, an energy and workload aware job scheduling policy for heterogeneous clusters. The strategy models jobs in terms of performance and runtime by using mathematical models. It automatically selects optimal frequency and prioritizes the most efficient hardware resources by reordering the job's scheduler queue. Results show improvements in the system response time and energy saving when implementing EAMC-policy over other energy-saving policies with no heterogeneous hardware awareness. We learnt that workload-awareness is a complex task that needs more effort by the research community, but that has a high impact in making HPC systems more efficient.

This work uses different methodologies based on real runs and simulated experiments. These methodologies were investigated and improved during this thesis, representing themselves as secondary contributions.

To conclude, we propose a number of research topics that can be considered as a continuation of this work, and new research directions emerged during the study of the state of the art.

- *Awareness*. Workload and hardware modeling should be extended. We consider it is necessary to classify workloads based on their behavior, i.e., compute-bounded or memory-bounded. This classification can be done at the job level, or more in detail, by classifying the different job phases. DROM could be extended for collecting runtime data related to hardware and job and make this information available for analysis by other components. A model could be developed to classify the jobs. The first difficulty consists of learning the job phases' granularity, e.g., a job phase can last milliseconds to hours. This depends on the type of application and the used data-set. The second difficulty is the classification algorithm based on collected job data. Machine learning approaches can be used. The third difficulty is the dependency between jobs, their input, and the used hardware. While hardware characteristics are known, it is not always the same with the job's input data. Some jobs read input from input files or folders. Others accept parameters or models and generate data structure at runtime based on them. Some applications can switch from compute-bounded to memory-bounded depending on the input data. All these possibilities should be taken into account o build a general classifier. A per-job classification could not be enough, and multiple classifications per job might be necessary.

As a second point, job schedulers could use job classification to improve scheduling and placement. SD-policy would benefit by coupling memory-bounded with compute-bounded jobs to enhance the sharing of nodes' resources. EAMC-policy could also benefit from better job classification. At the actual state of the art, energy and performance models need the job to run at least once on every architecture with the same or similar input. A model that generates energy and runtime profiles for all available architectures based on a single run, hardware profiles, and other similar jobs would be a promising research direction.

Finally, there is the need for more integration between the software stack layer. Participation to initiatives such as OpenStack, and collaborative projects are essential for this integration.

- *Dynamicity*. emerging memory technologies might be the key to more efficient horizontal malleability. Network-attached memories could act as a shared memory between nodes and as a way to exchange data between nodes faster. DROM implement vertical malleability, and it could be integrated with horizontal malleability approaches. Job schedulers could take advantage of both technologies to optimize the scheduling in different scenarios.

  As a second point, it could be interesting to test DROM in specific use cases, e.g., increase the response time in small systems, as an alternative to oversubscription. It would be interesting to integrate DROM in systems with QoS policies and services that need different computing power depending on the time of the day, such as internet services or HPC services such as databases management systems.

As a final point, it would be interesting to research dynamic approaches in the field of energy. By having an awareness of the job's requirements and phases, it would be possible to adjust the processor frequency at runtime accordingly. Besides, EAMC-policy would greatly benefit from integrating with SD-policy and study how malleability could affect energy consumption, especially in memory-bounded workloads.

- *Automatization*. The less user intervention is needed, the more a system can act autonomously. User intervention can be reduced by increasing awareness. Concerning job scheduling, for the same reasons explained for job modeling, it is not straightforward to automatize the job's space and time requirements. While malleability relaxes the former, the latter remains a field to research. Scheduling approaches based on non-defined time slots might be worth investigating, together with new interfaces for job submissions. As a final comment, we think that soon the single HPC site's computational power could not satisfy the HPC community's needs. Researchers can achieve further scaling by tightening the collaboration and the organization between different HPC sites. What a single machine cannot achieve in terms of scalability, multiple machines can achieve working together. Further research in all the hardware and software stack and a new layer of scheduling would be needed, coordinating the distributed clusters. DRMS can achieve this by evolving its architecture from a master-slave to a hierarchical or graph-based approach. Finally, humbleness, love, and compassion would absolutely help to this purpose.

A
Appendix

This appendix gives some information about the availability of the code related to this thesis. Five repositories are available, the first two are related to our contributions in the tools and methodologies:

1. Slurm Simulator [70]: the code has been improved, maintained and ported to Slurm version 19
2. Slurm as a job [32]: the code was cleaned, different versions are available, for homogeneous and for multi-cluster systems.

The remaining three repositories are related to the three main contributions:

3. DROM [14]: DROM is distributed as part of the DLB package, main-

tained by BSC. The folder: doc//examples contains different subfolders:

- drom: it includes an example application that runs with DROM, commands or API can be used to dynamically change the used cores.
- mpi+omp_ompt: it contains Pils application, MPI+OpenMP version that uses OMPT to make DROM calls transparent.
- mpi+ompss: it contains Pils application, MPI+OmpSs version, plus scripts to run it with DROM.

DLB documentation offers some more information on how to use DROM [10].

4. SD-policy [31] is available over Slurm version 16. When compiled, DLB_HOME environment variable needs to be set to the installation path of DLB.
5. EAMC-policy [71] is available as part of Slurm Simulator v19. When compiled, LRZ_MODEL environment variable needs to be set to the installation path of LRZ energy interface [80].

Bibliography

[6] Ahn, D. H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., & Schulz, M. (2014). Flux: A next-generation resource management framework for large hpc centers. In *2014 43rd International Conference on Parallel Processing Workshops* (pp. 9–17).

[7] Argonne National Laboratory (2020). Cobalt website. https://www.anl.gov/mcs/cobalt-componentbased-lightweight-toolkit, Accessed: 2020-07-30.

[8] Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., & Wilde, T. (2014). A case study of energy aware scheduling on supermuc. In J. M. Kunkel, T. Ludwig, & H. W. Meuer (Eds.), *Supercomputing* (pp. 394–409). Cham: Springer International Publishing.

[9] B., Y. A., A., J. M., & M., G. (2003). Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing* (pp. 44–60).

[10] Barcelona Supercomputing Center (BSC) (2020). DLB-DROM documentation. https://pm.bsc.es/ftp/dlb/doc/user-guide/how_to_run.html#drom-example, Accessed: 2020-05-7.

[11] Bell, R. H., Brochard, L., DeSota, D. R., Panda, R. D., Thomas, F., et al. (2013). Energy-aware job scheduling for cluster environments. US Patent 8,612,984.

[12] Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., & Zagorodnov, D. (2003). Adaptive computing on the grid using apples. *IEEE Transactions on Parallel and Distributed Systems*, 14(4), 369–382.

[13] Borghesi, A., Bartolini, A., Lombardi, M., Milano, M., & Benini, L. (2018). Scheduling-based power capping in high performance computing systems. *Sustainable Computing: Informatics and Systems*, 19, 1 – 13.

[14] BSC (2018). DLB-DROM source code. https://github.com/bsc-pm/dlb/, Accessed: 2020-07-30.

[15] BSC (2020a). Marenostrum III website. https://www.bsc.es/marenostrum/marenostrum/mn3, Accessed: 2020-02-12.

[16] BSC (2020b). Marenostrum IV website. https://www.bsc.es/es/marenostrum/marenostrum, Accessed: 2020-02-12.

[17] BSC (2020c). Mintauro website. https://www.bsc.es/es/marenostrum/minotauro, Accessed: 2020-02-12.

[18] Caniou, Y. & Gay, J. S. (2009). Simbatch: An api for simulating and predicting the performance of parallel resources managed by batch systems. In E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, & S. Jha (Eds.), *Euro-Par 2008 Workshops - Parallel Processing* (pp. 223–234). Berlin, Heidelberg: Springer Berlin Heidelberg.

[19] Casanova, H., Giersch, A., Legrand, A., Quinson, M., & Suter, F. (2014). Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10), 2899–2917.

[20] Castain, R. H., Solt, D., Hursey, J., & Bouteiller, A. (2017). Pmix: Process management for exascale environments. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17 New York, NY, USA: ACM.

[21] Cera, M. C., Georgiou, Y., Richard, O., Maillard, N., & Navaux, P. O. A. (2010). Supporting malleability in parallel architectures with dynamic cpusets mapping and dynamic mpi. In K. Kant, S. V. Pemmaraju, K. M. Sivalingam, & J. Wu (Eds.), *Distributed Computing and Networking* (pp. 242–257). Berlin, Heidelberg: Springer Berlin Heidelberg.

[22] Chadha, M., John, J., & Gerndt, M. (2020). Extending slurm for dynamic resource-aware adaptive batch scheduling.

[23] Chapin, S. J., Cirne, W., Feitelson, D. G., Jones, J. P., Leutenegger, S. T., Schwiegelshohn, U., Smith, W., & Talby, D. (1999). Benchmarks and standards for the evaluation of parallel job schedulers. In *Proceedings of the 13th International Workshop Job Scheduling Strategies for Parallel Processing (JSSPP)* (pp. 66–89).: Springer-Verlag.

[24] Chasapis, D., Moretó, M., Schulz, M., Rountree, B., Valero, M., & Casas, M. (2019). Power efficient job scheduling by predicting the impact of processor manufacturing variability. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19 (pp. 296–307). New York, NY, USA: Association for Computing Machinery.

[25] Cirne, W. & Berman, F. (2001). A comprehensive model of the supercomputer workload. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*.

[26] Comprés, I., Mo-Hellenbrand, A., Gerndt, M., & Bungartz, H. (2016). Infrastructure and api extensions for elastic execution of mpi applications. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016 (pp. 82–97).

[27] Computing, A. (2020). Maui simulator. http://docs.adaptivecomputing.com/maui/16.0simulations.php, Accessed: 2020-07-30.

[28] Conway, R. W., Miller, L. W., & Maxwell, W. L. (2003). *Theory of scheduling*. Dover.

[29] Corbalan, J. & Brochard, L. (2020). EAR: technical documentation. https://www.bsc.es/sites/default/files/public/bscw2/content/software-app/technical-documentation/ear.pdf, Accessed: 2020-02-12.

[30] Czarnul, P., Proficz, J., & Krzywaniak, A. (2019). Energy-aware high-performance computing: Survey of state-of-the-art tools, techniques, and environments. *Scientific Programming*, 2019, 8348791:1–8348791:19.

[31] D'Amico", M. (2018a). "sd-policy source code: https://github.com/marcodamico/slurm_sd-policy".

[32] D'Amico", M. (2018b). "slurm as a slurm job: https://github.com/bsc-rm/slurm_as_a_job".

[33] D'Amico, M., Garcia-Gasulla, M., López, V., Jokanovic, A., Sirvent, R., & Corbalan, J. (2018). Drom: Enabling efficient and effortless malleability for resource managers. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18 (pp. 41:1–41:10). New York, NY, USA: ACM.

[34] DEEP Project (2020). DEEP Project website. https://www.deep-projects.eu/, Accessed: 2020-02-12.

[35] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., & Planas, J. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21.

[36] Dutot, P.-F., Mercier, M., Poquet, M., & Richard, O. (2016). Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator. In *20th Workshop on Job Scheduling Strategies for Parallel Processing* Chicago, United States.

[37] EPI (11/11/2020). European Process Initiative (EPI), https://www.european-processor-initiative.eu/.

[38] Feitelson, D. G. (1997). Job scheduling in multiprogrammed parallel systems.

[39] Feitelson, D. G. (Dec 8, 2005). *Logs of Real Parallel Workloads from Production Systems*. http://www.cs.huji.ac.il/labs/parallel/workload/.

[40] Feitelson, D. G. (Feb 21, 2006). *The Standard Workload Format*. http://www.cs.huji.ac.il/labs/parallel/workload/swf.html.

[41] Feitelson, D. G. & Rudolph, L. (1996a). Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing* (pp. 1–26). Berlin, Heidelberg.

[42] Feitelson, D. G. & Rudolph, L. (1996b). Towards convergence in job schedulers for parallel supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96 (pp. 1–26). London, UK, UK: Springer-Verlag.

[43] Feitelson, D. G. & Rudolph, L. (1998). Metrics and benchmarking for parallel job scheduling. In D. G. Feitelson & L. Rudolph (Eds.), *Job Scheduling Strategies for Parallel Processing* (pp. 1–24). Berlin, Heidelberg: Springer Berlin Heidelberg.

[44] Garcia, M., Corbalan, J., & Labarta, J. (2009). Lewi: A runtime balancing algorithm for nested parallelism. In *International Conference on Parallel Processing*.

[45] Garcia, M., Labarta, J., & Corbalan, J. (2014). Hints to improve automatic load balancing with lewi for hybrid applications. *Journal of Parallel and Distributed Computing*.

[46] Garcia-Gasulla, M. (2017). Malleable nest source code. https://github.com/mggasulla/nest-simulator/tree/malleability, Accessed: 2020-07-30.

[47] GNU (2018). The GNU C library: CPU Affinity. https://www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html.

[48] Gorda, B. & Brooks, E. I. (1991). Gang scheduling a parallel machine.

[49] Gupta, A., Acun, B., Sarood, O., & Kale, L. V. (2014). Towards Realizing the Potential of Malleable Parallel Jobs. In *Proceedings of the IEEE International Conference on High Performance Computing*, HiPC '14 Goa, India.

[50] Huang, C., Lawlor, O., & Kalé, L. V. (2003). Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*.

[51] Human Brain Project (2017). https://www.humanbrainproject.eu/en/.

[52] Hungershofer, J. (2004). On the combined scheduling of malleable and rigid jobs. In *16th Symposium on Computer Architecture and High Performance Computing*.

[53] IBM (2014). Platform LSF. www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.2/lsf_welcome.html.

[54] Intel (2016). Intel©openmp runtime library. https://www.openmprtl.org/, Accessed: 2020-07-30.

[55] Kalé, L. & Krishnan, S. (1993). CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93* (pp. 91–108).: ACM Press.

[56] Kale, L. V., Kumar, S., & DeSouza, J. (2002). A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*.

[57] Klusáček, D. & Rudová, H. (2010). Alea 2: Job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10 (pp. 61:1–61:10). ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[58] Kumbhar, P. (2017). Malleable coreneuron source code. https://github.com/BlueBrain/CoreNeuron/tree/hbp_dlb.

[59] Kumbhar, P. & Hines, M. (2016). Coreneuron neuronal network simulator optimization opportunities and early experience. In *GPU Technology Conference*.

[60] Kunkel, S. et. al (2017). Nest 2.12.0. https://doi.org/10.5281/zenodo.259534.

[61] Lawson, B. & Smirni, E. (2005). Power-aware resource allocation in high-end systems via online simulation. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05 (pp. 229–238). New York, NY, USA: Association for Computing Machinery.

[62] Lenovo (2015). Lenox Cluster. https://www.top500.org/system/178548, Accessed: 2020-02-12.

[63] Lifka, D. A. (1995). The anl/ibm sp scheduling system. In D. G. Feitelson & L. Rudolph (Eds.), *Job Scheduling Strategies for Parallel Processing* (pp. 295–303). Berlin, Heidelberg: Springer Berlin Heidelberg.

[64] Llort, G., Servat, H., González, J., Giménez, J., & Labarta, J. (2013). On the usefulness of object tracking techniques in performance analysis. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[65] LRZ (2020). SuperMuc-NG Site. https://doku.lrz.de/display/PUBLIC/SuperMUC-NG, Accessed: 2020-02-12.

[66] Lucero, A. (2011). Simulation of batch scheduling using real production-ready software tools. In *Proceedings of the 5th IBERGRID*.

[67] Ludwig, W. & Tiwari, P. (1994). Scheduling malleable and nonmalleable parallel tasks. In *SODA*, volume 94 (pp. 167–176).

[68] Maghraoui, K. E., Desell, T. J., Szymanski, B. K., & Varela, C. A. (2007). Dynamic malleability in iterative mpi applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)* (pp. 591–598).

[69] Maiterth, M., Koenig, G., Pedretti, K., Jana, S., Bates, N., Borghesi, A., Montoya, D., Bartolini, A., & Puzovic, M. (2018). Energy and power aware job scheduling and resource management: Global survey — initial analysis. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 685–693).

[70] Marco D'Amico (2020a). BSC Slurm Simulator Github code. https://github.com/BSC-RM/slurm_simulator_tools, Accessed: 2020-05-7.

[71] Marco D'Amico (2020b). EAMC-policy Github code. https://github.com/marcodamico/slurm_simulator_eamc-policy, Accessed: 2020-05-7.

[72] Marco D'Amico; Julita Corbalan (2020). Energy models and application data for three Intel CPUs.

[73] Martín, G., Marinescu, M.-C., Singh, D. E., & Carretero, J. (2013). Flex-mpi: An mpi extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In *Euro-Par 2013 Parallel Processing* (pp. 138–149). Berlin, Heidelberg: Springer Berlin Heidelberg.

[74] Martín, G., Singh, D. E., Marinescu, M.-C., & Carretero, J. (2015). Enhancing the performance of malleable mpi applications by using performance-aware dynamic reconfiguration. *Parallel Comput.*, 46.

[75] McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. *Technical Committee on Computer Architecture*.

[76] Message Passing Interface Forum (2015). MPI Specifications 3.1, http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[77] Moore, J., Chase, J., Ranganathan, P., & Sharma, R. (2005). Making scheduling "cool": Temperature-aware workload placement in data centers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05 (pp.5). USA: USENIX Association.

[78] Mounie, G., Rapine, C., & Trystram, D. (1999). Efficient approximation algorithms for scheduling malleable tasks. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '99 (pp. 23–32).

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

[80] Navarrete, C. (2020). Library implementation for the DEEP-EST energy aware scheduling. https://www.deep-projects.eu/images/materials/library.pdf, Accessed: 2020-05-12.

[81] Netti, A., Galleguillos, C., Kiziltan, Z., Sîrbu, A., & Babaoglu, O. (2018). Heterogeneity-aware resource allocation in hpc systems. In R. Yokota, M. Weiland, D. Keyes, & C. Trinitis (Eds.), *High Performance Computing* Cham: Springer International Publishing.

[82] OAR (2020). OAR resource manager. http://oar.imag.fr/documentation.

[83] OpenMP (2016). Tr4: Openmp version 5.0 preview 1.

[84] OpenMP (27/4/2018). OpenMP 4.5 Specifications, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[85] OpenMPI (2016). Pmpi profiling interface. https://www.open-mpi.org/faq/?category=perftools.

[86] Ousterhout, J. (1982). Scheduling techniques for concurrent systems. In *Proc. 3rd International Conference on Distributed Computing Systems*.

[87] Patki, T., Lowenthal, D. K., Sasidharan, A., Maiterth, M., Rountree, B. L., Schulz, M., & de Supinski, B. R. (2015). Practical resource management in power-constrained, high performance computing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15 (pp. 121–132). New York, NY, USA: Association for Computing Machinery.

[88] Pillet, V., Labarta, J., Cortes, T., & Girona, S. (1995). Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-*

18: transputer and occam developments, volume 44 (pp. 17–31).: IOS Press.

[89] Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., & Kale, L. V. (2015). A batch system with efficient adaptive scheduling for malleable and evolving applications. In *2015 IEEE International Parallel and Distributed Processing Symposium* (pp. 429–438).

[90] Rajagopal, D., Tafani, D., Georgiou, Y., Glesser, D., & Ott, M. (2017). A novel approach for job scheduling optimizations under power cap for arm and intel hpc systems. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)* (pp. 142–151).

[91] Rodrigo, G. P., Elmroth, E., Ostberg, P.-O., & Ramakrishnan, L. (2017). Scsf: A scheduling simulation framework. In *Proceedings of the 21st International Workshop Job Scheduling Strategies for Parallel Processing (JSSPP)*.

[92] Sarood, O., Langer, A., Gupta, A., & Kale, L. (2014). Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 807–818).

[93] Simakov, N. A., Innus, M. D., Jones, M. D., DeLeon, R. L., White, J. P., Gallo, S. M., Patra, A. K., & Furlani, T. R. (2018). A slurm simulator: Implementation and parametric analysis. In S. Jarvis, S. Wright, & S. Hammond (Eds.), *High Performance Computing Systems. Per-*

formance Modeling, Benchmarking, and Simulation (pp. 197–217). Cham: Springer International Publishing.

[94] Subramaniam, B. & Feng, W. (2012). The green index: A metric for evaluating system-wide energy efficiency in hpc systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum* (pp. 1007–1013).

[95] Sun, H., Cao, Y., & Hsu, W. (2011). Fair and efficient online adaptive scheduling for multiple sets of parallel applications. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*.

[96] Tang, W., Lan, Z., Desai, N., & Buettner, D. (2009). Fault-aware, utility-based job scheduling on blue, gene/p systems. In *2009 IEEE International Conference on Cluster Computing and Workshops* (pp. 1–10).

[97] Team, O. (2020). A batching queuing system. *Software Project, Altair Grid Technologies, LLC, www.openpbs.org*.

[98] Top500 (2020a). Green500 Supercomputer Sites. https://www.top500.org/green500/, Accessed: 2020-02-12.

[99] Top500 (2020b). Top500 Supercomputer Sites. https://www.top500.org/, Accessed: 2020-02-12.

[100] Trofinoff, S. & Benini, M. (2015). *Using and Modifying the BSC Slurm Workload Simulator*. Slurm User Group Meeting 2015.

[101] Turek, J., Schwiegelshohn, U., Wolf, J. L., & Yu, P. S. (1994). Scheduling parallel tasks to minimize average response time. In *Proceedings*

of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '94 (pp. 112–121). USA: Society for Industrial and Applied Mathematics.

[102] Utrera, G., Corbalan, J., & Labarta, J. (2004). Implementing malleability on mpi jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*.

[103] Utrera, G., Corbalan, J., & Labarta, J. (2014). Scheduling parallel jobs on multicore clusters using cpu oversubscription. *The Journal of Supercomputing*, 68(3), 1113–1140.

[104] Utrera, G., Tabik, S., Corbalan, J., & Labarta, J. (2012). A job scheduling approach for multi-core clusters based on virtual malleability. In *Euro-Par 2012 Parallel Processing* (pp. 191–203).

[105] Vadhiyar, S. S. & Dongarra, J. J. (2005). Self adaptivity in grid computing: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4), 235–257.

[106] Vázquez, M., Houzeaux, G., Koric, S., Artigues, A., Aguado-Sierra, J., Arís, R., Mira, D., Calmet, H., Cucchietti, F., Owen, H., Taha, A., Burness, E. D., Cela, J. M., & Valero, M. (2016). Alya: Multiphysics engineering simulation toward exascale. *Journal of Computational Science*, 14, 15–27. The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills.