ALMA MATER STUDIORUM - UNIVERSITÁ DI BOLOGNA

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

CICLO: XXX

SETTORE CONCORSUALE DI AFFERENZA 09/H1
SETTORE SCIENTIFICO DISCIPLINARE ING-INF/05

Scalable optimization-based Scheduling approaches for HPC facilities

Presentata da:
Thomas BRIDI

Coordinatore Dottorato:
Chiar.mo Prof.
Paolo CIACCIA

Supervisore:
Chiar.mo Prof.
Michela MILANO

Esame Finale 2018

Alma Mater Studiorum - Universitá di Bologna

Abstract

Scalable optimization-based Scheduling approaches for HPC facilities

by Thomas BRIDI

This Thesis deals with the problem of scheduling applications on High-Performance Computing (HPC) machines. The goal is to create a scheduler that can improve the solutions w.r.t. the state-of-the-art under different metrics. However, improving the solution quality is not enough: creating a scheduler for future HPC machines requires to take into account also overheads and scalability. In this thesis we present a comprehensive, scalable, scheduling approach that features both an off-line and an on-line component. The off-line component is based on Constraint Programming (CP), an optimization technique that is well-suited for scheduling problems and allows for great flexibility. We leverage this flexibility to present first a optimization method designed to optimize the job waiting times, which is then extended via heuristics and search strategies to deal with more complex objective functions. Unfortunately, such a complex objective function cannot be handled by a solver in an acceptable amount of time for online operation on a HPC machine in-production. We deal with this difficulty by making use of a second, distributed, on-line scheduler. This second scheduler is designed to dramatically decrease the computational overhead and achieve a scalability adequate to future ExaFlops HPC machines. The distributed scheduler is proactive, and it takes decisions so as to follow a desirable, pre-specified, utilization profile. This feature makes it possible to connect these two schedulers to create a hybrid system: the CP component computes the scheduling on a trace of forecasted jobs one day ahead, machine learning techniques extract from the solution a near-optimal and desirable utilization profile, and the online scheduler takes care of the actual scheduling decisions in a scalable fashion. The resulting architecture manages to improve the HPC machine profit by an average 8.6%, while decreasing the computational overhead and, under normal conditions, without any side effect.

List of Pubblications

Part of the work in this thesis have previously appeared in:

1. A. Bartolini, A. Borghesi, T. Bridi, M. Lombardi, and M. Milano. Proactive workload dispatching on the EURORA supercomputer. English. In *Principles and practice of constraint programming - 20th international conference, CP 2014, lyon, france, september 8-12, 2014. proceedings*. B. O'Sullivan, editor. Vol. 8656. In Lecture Notes in Computer Science. Springer. Springer International Publishing, 2014, pp. 765–780. ISBN: 978-3-319-10427-0. DOI: 10.1007/978-3-319-10428-7_55

2. T. Bridi, M. Lombardi, A. Bartolini, L. Benini, and M. Milano. A cp scheduler for high-performance computers. In. Vol. 1485, 2015, pp. 37–42. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85009223255&partnerID=40&md5=65ac2e65b77b8fbb06d15c101edd7bbd

3. T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE transactions on parallel and distributed systems*, 27(10):2781–2794, 2016. DOI: 10.1109/tpds.2016.2516997. URL: http://dx.doi.org/10.1109/TPDS.2016.2516997

4. T. Bridi, M. Lombardi, A. Bartolini, L. Benini, and M. Milano. DARDIS: Distributed And Randomized DIspatching and Scheduling. In *European conference on artificial intelligence (ECAI 2016)*. Vol. 285. Gal A. Kaminka et al., 2016, pp. 1598–1599

5. T. Bridi, M. Lombardi, A. Bartolini, L. Benini, and M. Milano. DARDIS: Distributed And Randomized DIspatching and Scheduling. In, *AI*IA 2016 advances in artificial intelligence*, pp. 493–507. Springer, 2016

6. T. Bridi, A. Bartolini, M. Lombardi, P. V. Hentenryck, M. Milano, and L. Benini. Profit-driven hpc scheduling optimization and pue analysis. *IEEE transactions on industrial informatics*, under review, 2017

7. T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. Hybrid offline-optimized and online-distributed profit-driven low-overhead scheduler for hpc with automatic node shut-down and turn-on. *IEEE transactions on parallel and distributed systems*, under review, 2017

8. C. Galleguillos, A. Sîrbu, Z. Kiziltan, O. Babaoglu, A. Borghesi, and T. Bridi. Data-driven job dispatching in hpc systems. In *International workshop on machine learning, optimization, and big data*. Springer, 2017, pp. 449–461

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
CP	**C**onstraint **P**rogramming
CSP	**C**onstraint **S**atisfaction **P**roblem
DARDIS	**D**istributed **A**nd **R**andomized **DI**spatcher and **S**cheduler
FLOPS	**FL**oating point **O**perations **P**er **S**econd
HPC	**H**igh **P**erformance **C**omputing
LNS	**L**arge **N**eighborhood **S**earch
MPI	**M**essage **P**assing **I**nterface
NFS	**N**etwork **F**ile **S**ystem
OpenMP	**Open** **M**ulti**P**rocessing
PBS	**P**ortable **B**atch **S**cheduler
PUE	**P**ower **U**sage **E**ffectiveness
RCP	**R**emote **C**o**P**y
RCPSP	**R**esource-**C**onstrained **P**roject **S**cheduling **P**roblem
SCP	**S**ecure **C**o**P**y
SSH	**S**ecure **SH**ell

Chapter 1

Introduction

This thesis studies the job scheduling and dispatching problem for High-Performance Computing (HPC) machines.

In these systems achieving a 1% improvement brings substantial advantages in terms of money, waiting time and power savings.

HPC machines peak performance, measured in FLOPs (floating point operation per second), has been continuously growing since their introduction in 1963 of TOP500 [9] ranking. From 1963 to nowadays the #1 performance for the Linpack benchmark [10] grown from 59.7GFlops to 93PFlops and it is expected that the Exascale (1EFlops) [11] will be reached around the year 2020.

Due to the enormous amount of investments done for these computing machines and the fast depreciation (3-5 years), the return of investment is one of the most important points to keep in mind in both hardware and software and also the scheduler can be set to be profit aware. However, also the user experience is important in these systems. For this reason, we have to consider also other metrics like job waiting time and number of late jobs.

With the new goal of the Exascale, a set of new opportunities comes to the scientific and industrial communities but this brings also a set of new challenges. The first one is the scalability. To create an ExaFlops computer two paths can be taken: the first is to create more computationally powerful nodes, the second is to create HPC with a higher number of nodes. For the first path, the problem is that the integrated circuit density limit is going to be reached an this will limit the computational power of a single chip [12]. For this reason, the most affordable way for the Exascale is to substantially increase the number of computational nodes into an HPC machine. This brings scalability problems both in the hardware and the software: starting from the nodes interconnections to the application scalability and even to the scheduler.

A second problem comes from a power limitation. Usually, energy providers fix the power limit for industrial buildings to 20MW. This limit is valid also for computing centers. The current #1 in the TOP500 list, the Sunway TaihuLight, can provide a computing power of 93PFlops with a power consumption of 15.37MW. This means that for the future Exascale machines, the engineers have to increase the computational power 10 times w.r.t. the current top HPC with an increment of the consumption of 0.3 times.

In this scenario the importance of the scheduler is pervasive. With an appositely designed scheduler, it is possible to not just to improve the system utilization and the user experience with highly scalable algorithms but also to improve the system efficiency in order to decrease the power consumption, anticipate the return of investments, increase the profit and create a greener computing system.

This results cannot be achieved just by a single approach like greedy and heuristic algorithms or distributed computing or optimization techniques but by a hybrid approach that exploits all of them.

The focus of the thesis is on the optimization of the scheduling results under several different metrics and the scalability to make this optimization usable in a real environment. HPC machines (also called supercomputers) are computer composed by a set of nodes connected by a network. Each node contains a set of resources (processors, accelerators, memory, etc.). The scheduling problem for HPC consist of a non-preemptable batch scheduling. Jobs are submitted to a high-level scheduler by the user, usually, through a login node of the machine. The scheduler reacts to the events like submissions and job terminations and tries to schedule and dispatch the waiting jobs. The scheduler selects the node(s) and the starting time for the job execution. It has to deal with resource requirements of the jobs and the available resources of the machine. This means that the scheduler reserves the resources to a job until its termination. This is an intentional behavior by the HPC administrator used to prevent preemption on the resources that would lead to performance worsening due to cache replacement and context switching. The objective of the scheduler is to optimize different metrics (depending on the needs of the computing center) as for example user's related metrics as waiting times, throughput, etc. From now on we refer to scheduling and dispatching as scheduling.

Job dispatching in supercomputers is a special case of the wider Scheduling and Allocation problem that arises in several different computing fields.

1.1 Content

This thesis deals with the job scheduling and dispatching problem in HPC systems which can be subdivided into two components:

1. the allocation problem: choosing the set of nodes to be assigned to each job
2. the scheduling problem: deciding the start time for each job avoiding resource overusage.

Our attention is directed to the scalability and the profitability of the scheduler. The idea is to create a highly scalable scheduler that can also improve the profit of a computing center. In fact, the content of this work can be subdivided into five different points:

A preliminary study on the application of Constraint Programming to the HPC scheduling problem We study the applicability of Constraint Programming to the HPC scheduling problem. We model the basic features of an HPC scheduler. The experimentations in this study demonstrated the feasibility of the approach and showed promising results w.r.t. the commercial scheduler PBS Professional 12. However, not all the features required for an in-production HPC have been taken into account.

A real-word CP scheduler We have developed a second approach, modeling the majority of the needed features for an in-production HPC such as arrays of jobs, "critical-priority" jobs, heterogeneous jobs, reservations, etc. Many techniques have been adopted to limit the computational overhead to an acceptable value. Finally, the scheduler has been embedded into a commercial scheduler: PBS Professional 12. Further experimentation shows that the approach is still feasible and can improve the results obtained by PBS in terms of makespan and waiting time but with scalability problems. Moreover, an experimentation on a real, in-production, HPC has been done with promising results in terms of system utilization.

A distributed and randomized scheduler (DARDIS) This new scheduling approach for HPC has been designed to drastically decrease the scheduling overhead and increase the scalability of the scheduler. The scheduler is designed to distribute the start time selection computation through the nodes while the dispatching is done by the job itself on the basis of the candidate starting times returned by the nodes. Results show that just in some cases it can improve the results obtained by commercial schedulers while the scheduling computation is 10 times faster.

A profit-aware offline scheduler This CP scheduler is an offline scheduling solution designed to optimize the profit obtained from the HPC considering not only the set of submitted jobs and the resources but also the cooling model and the external temperature forecast. The scheduler is designed to schedule synthetic jobs or the set of jobs submitted in the last 24h, obtain a sub-optimal utilization profile optimizing the profit. This profile is then sent to a profile-aware online scheduler. Moreover, this study also provides an analysis on the system efficiency to decide if to increase or decrease the workload of the next day to improve the profit. Experimentations in several different scenarios have been done considering: different external temperatures, cooling models, pricing schema, etc. It has been shown that the scheduler achieves profit improvements up to 6-7% w.r.t. standard and adhoc rule-based schedulers in the case of high-workload while no worsening have been found in case of low workload.

A hybrid offline-online scheduler This scheduler is designed to combine the near-optimal results obtained by optimization techniques such as Constraint Programming with a fast and scalable distributed scheduler. The scheduler is capable to plan the resources utilization in the future giving the possibility to turning on and off nodes of the system without any delay. The scheduler proved to improve the HPC machine profit by on average of a 8.6%. The scheduler also proved to be more scalable of both commercial heuristic schedulers and optimized CP schedulers. Moreover, no significant makespan improvement has been found in the case of low and medium workload requests. However, an increment of 10% in makespan has been found in case of high workload. This worsening is nature of the distributed scheduler.

1.2 Contribution

In this work, we use techniques as Constraint Programming, Distributed systems, and Heuristics. Our contribution consists of the creation of several different schedulers for HPC and analysis on the cooling efficiency of these systems and how to modulate the future system utilization to increase the profit. More in detail, our contribution can be summarized as follows:

- An online Constraint Programming scheduler [1, 3] that models all the different features needed for a commercial scheduler. This scheduler aims to minimize several different metrics such as the makespan, the number of late jobs and the waiting time but the best results have been obtained with the goal of minimizing the weighted queue time. The weighted queue time is a metric designed to take into account the waiting time of jobs in a fair way: each waiting is weighted on the expected waiting of the user.

- A distributed, randomized, and profile-aware scheduler [4, 5]. This scheduler is designed to dramatically improve the scalability of the state-of-the-art schedulers. This is the first distributed scheduler that enables the agent's agreement without a handshake or an agreement protocol. Moreover, this scheduler is shown to highly decrease the computational overhead for the scheduling and dispatching components.

- An offline and profit-aware Constraint programming scheduler [6]. This scheduler models both the HPC resources and the cooling of the system to optimize the scheduling to maximize the profit taking into account the weather forecast. This scheduler has proven to increase the profit of the system up to a 7% in case of high workload while no profit decrement has been found in case of low workload. Moreover, the results of this scheduler have been used to understand when it is more profitable for the system to increase or decrease the system utilization to improve the profit. This lead to a simple rule to forecast which is the best behavior of the scheduler for the next 24h to maximize the profit.

- A hybrid Online-Offline scheduling architecture [7]. This architecture is composed by an offline scheduler capable to optimize the profit of a chunk of jobs. This scheduler is triggered at fixed time intervals (24-48 hours) and, on the basis of a forecasted workload traces, it computes the optimal allocation. The scheduling solution obtained by the offline scheduler is then passed to a "desirable utilization profile generator" that generates a utilization profile that contains the schedule of jobs along with node shut down and start up. The desirable utilization profile is then fed to the Online scheduler (DARDIS). The online scheduler is a fast distributed scheduler designed for scalability. The online distributed scheduler is also designed to follow a desirable utilization profile variable in times planning the scheduling in the future.

1.3 Outline

The thesis is organized as follows.

Chapter 2 shows an introduction to HPC systems and how they work. It introduces the scheduling problem and shows the main algorithms used in real-life HPC machines. It shows the state of the art in scheduling algorithms, scheduling optimization, distributed scheduling. Finally, it introduces the Constraint Programming (CP) paradigm, how it works, the modeling, and the most used searches in this field.

Chapter 3 shows a preliminary study to the application of CP into the HPC scheduling and dispatching problem. The chapter explains the modeling of the scheduling problem on the EURORA HPC at CINECA, the first simulations.

Chapter 4 shows a first application of a CP model into a real and online scheduler. In this chapter, we modeled the most majority of the constraints to make our CP scheduler a working prototype. The simulations are made to show the scalability of the scheduler. Moreover, an evaluation on the HPC with a real and in-production workload has been made to show its usability.

Chapter 5 shows a distributed and randomized scheduling approach designed to solve the scalability problems of the CP scheduler. Distributed And Randomized DIspatcher and Scheduler (DARDIS) is a scheduler that offloads and parallelizes the scheduling problems to the nodes and the dispatching and commitment to the user. This approach has shown to be faster than heuristic algorithms but, in some cases, obtaining not good solutions.

Chapter 6 proposes an offline CP scheduler designed to interact with the DARDIS scheduler in order to improve its solution. The idea is to obtain near-optimal solutions with an offline scheduler (as the one proposed in this chapter) using traces obtained from workload forecast and then to guide the DARDIS decision exploiting the offline solution.

Chapter 7 proposes a hybrid offline CP and online distributed scheduling solution. The offline scheduler computes a sub-optimal scheduling, optimizing the profit, every 24 hours with the jobs submitted so far. From its best solution an optimal utilization profile is extracted and used for the next 24 hours into the online scheduler. The online and distributed scheduler (DARDIS) uses the utilization profiles generated from the offline scheduler to turn off and on the HPC resources to decreases the energy expenses.

Finally, Chapter 8 contains our final consideration and future works.

Chapter 2

Related work

This chapter gives a view of the state-of-the-art of the scheduling and dispatching problem for HPC. The chapter is subdivided into sections: Section 2.1 explains the HPC systems architecture. Section 2.2 explores the commercial and heuristic solutions used. Section 2.3 shows the state-of-the-art in the HPC scheduling dispatching problem, focusing on optimization and AI techniques. Section 2.4 explores the distributed scheduling approaches present in literature. Section 2.5 shows works not related to the scheduling and dispatching but still interesting for our work. Finally, section 2.6 shows the Constraint Programming paradigm, the CP problems modeling, and its searches.

2.1 HPC systems

High-Performance Computing is a technology widely used for research and industry when high computational power is needed. Usually, these machines are used for physics and chemistry simulations [13, 14, 15], fluid dynamics [16, 17, 18], material design [19, 20, 21], pharmaceutical [22, 23, 24] and so on.

HPC machines are massively parallel machines composed by a set of racks, each one with a set of nodes, each node usually contains multiple multi-cores CPUs, accelerators, and RAM (Figure 2.1). In these systems, looking to the TOP500 ranking [9], the core number lays the range from thousands [25] to tens of millions [26] core per HPC. This is translated in a number of nodes in the range from hundreds to millions. The computational power of HPC machines is measured in term of PetaFLOPS [27]. However, the emerging target is to point to the ExaFLOPS [28] for the year 2020. All these nodes are usually interconnected by both Gigabit Ethernet [29] and InfiniBand [30] interconnections. This multi-core, multi-processors, and multi-nodes infrastructure is exploited at its best using usually two computing framework Open Multiprocessing (OpenMP) [31] and Message Passing Interface (MPI) [32] in a hybrid fashion. Combining both MPI and OpenMP gives the possibility to instantiate multiple parallel tasks on different nodes (thanks to MPI) and in each node multiple threads (thanks to OpenMP). These resources are used without concurrency and preemption to maximize the performance. However, to select the right (free) resources and to execute the jobs, a scheduling and dispatching software is needed.

FIGURE 2.1: Marconi HPC, Eurora Rack, and Knights-Landing Node

These HPC machines are investment-intensive machines with short depreciation cycles. An average supercomputer reaches full depreciation in three to five years [33]. Hence their utilization has to be aggressively managed to produce an acceptable return on investment. Even relatively small improvements in utilization, throughput, and quality of service translate into significant financial gains.

2.2 HPC scheduling

The problem of batch scheduling is well-known and widely investigated [34, 35, 36, 37].

In the HPC jobs scheduling, we usually have a *login node* which is the only interface to the web. Through this node, the users can login and access to a remotely mounted home in which the user can develop and test its code for a limited amount of time. The user then can send its application to the HPC submitting its job to the scheduler. When submitting a job, the user can specify the number of nodes required for the execution (from now called **job units**), the amount of resources for each node (Cores, RAM, GPUs, etc.), the maximum execution time (from now called walltime), the queue (or partition) to submit. There are two different execution modes:

- Normal: in normal mode the user submit a script that executes its job, the script is then accessed into the selected nodes through protocols like NFS [38], scp [39], rcp [40], etc. And then it is executed without the possibility for the user to access to the standard output.
- Interactive: an ssh [41] connection to one of the selected nodes is open to the user. After that, the user can run commands to start a job directly on the node. In this way, the user can interact with the job as it is executed in its local terminal. If a user submits a job with multiple nodes, it can access the remaining nodes through ssh or it can execute a parallel job specifying the selected nodes through MPI.

After the submission, the scheduler takes care of the jobs selection following the algorithms and rules chosen by the administrator. Each node is then queried to find the set of nodes with enough free resources to execute all the job units of the job. If the resources are available, the job is executed and the next one is selected.

This scheduling problem can be classified as a variant of the Resource-Constrained Projects Scheduling Problem (RCPSP). This kind of problem is a well-studied problem and its complexity have been proven to be *NP-Hard* [42, 43, 44].

Some of the most widespread rule-based scheduling software in HPC facilities are PBS Professional [45], Torque [46], Slurm [47]. PBS Professional and Torque are branches of the original OpenPBS project (as described in [48] and [49]), the first is a commercial software distributed by Altair, the second is an open source version of the original PBS. Slurm is an open source scheduler: differently from PBS that uses queues, it uses resource partitioning to give a finer management and only one queue. In general, the large majority of commercial schedulers have a greedy component: the proposed heuristic does not explore the solutions space and generates a "good" solution. Neither local nor global optimality can be achieved.

The reader can refer to the works [50, 51, 52] for good surveys on scheduling algorithms used in HPC and computing clusters. Most of the algorithms described in these works can be implemented within commercial scheduling software by defining appropriate "scheduling rules" (e.g., the min-min algorithm can be implemented sorting jobs by increasing amount of required resources).

In a wider context, there is a large body of literature on scheduling and allocation for data-center workloads [53, 54, 55, 56] relying on the key assumption that partial or complete migration of parallel jobs is possible during their execution. Even though supercomputers will reasonably move toward more agile execution models [57, 58], the common practice today is that job migration is not allowed, to maximize performance and predictability [59].

2.2.1 Rule-based

Rule-based scheduling is one of the first types of algorithms proposed for the scheduling problem [60, 61, 62] but is still widely used in HPC [63].

This heuristic algorithm (see Algorithm 1) processes jobs and nodes in a given order which is specified via a customized rule (lines 1 and 2). When a job is processed, the scheduler considers each job unit (line 4) and starts querying the system nodes to find a sufficient amount of free resources (lines 5-10). When all job units have a candidate node, the job is started (lines 13-15). If a job cannot be immediately started, two alternative behaviors are possible:

- *Strict ordering*: This is the most priority conservative approach. If job i cannot be started, the scheduler stops and waits for the next termination event to restart the process from i (lines 17-19).
- *Non-strict ordering*: This is the most utilization aggressive approach. If job i cannot be executed, the scheduler skips it and tries to schedule job $i + 1$.

Algorithm 1 RB(J = list of jobs, N = list of nodes, Rule1 = jobs ordering rule, Rule2 = nodes ordering rule, Strict = true if strict ordering)

```
1:  $OJ$ = Order($J$,Rule1)
2:  $ON$ = Order($N$,Rule2)
3:  foreach job $j$ in $OJ$ do
4:      foreach $w$ job unit in $j$ do
5:          foreach node $n$ in $ON$ do
6:              foreach different resource $k$ in the node $n$ do
7:                  if the resource requirement of $j$ is lower or equal to the free resources of $k$ then
8:                      set the execution of the job unit $w$ of job $j$ to the node $n$
9:                  end if
10:             end for
11:         end for
12:     end for
13:     if all the job units of job $j$ have a node then
14:         run the job
15:     else
16:         clear the nodes of each job unit of job $j$
17:         if Strict then
18:             return
19:         end if
20:     end if
21: end for
```

Despite the simplicity and similarity of these two approaches it is important to note that choosing a strict or non-strict ordering can lead to two completely different behaviors. With the strict ordering, the jobs priority is always respected thus leading to a high underutilization of the resources. With the non-strict ordering, the priority is used as a soft constraint thus leading to a higher system utilization at the expenses of the user's fairness.

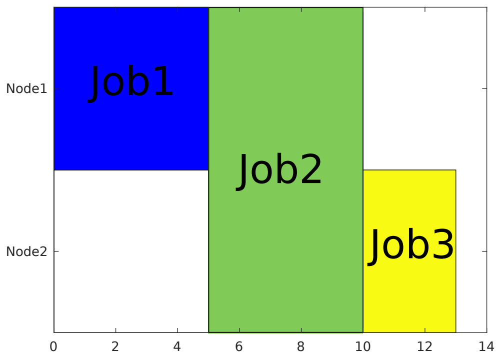
Many algorithms have been proposed to obtain a trade-off between these two. The majority of these algorithms start with a Strict-ordering-rule-based algorithm and after the first non-executed job applies a backfilling algorithm.

2.2.2 Backfilling

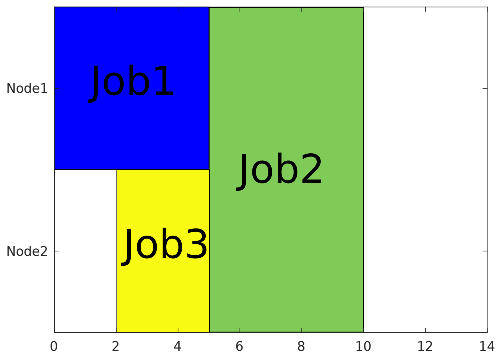
Backfilling algorithms are algorithms designed to fill resources left unused from the main scheduling algorithm (see figure 2.2).

Many variants of the backfilling algorithm have been proposed [64]. However, the most used are Conservative Backfilling and EASY-Backfilling.

The Conservative Backfilling algorithm is designed to increase the system utilization but paying high attention to the jobs fairness: the algorithm is designed to anticipate a job without delaying any higher priority job (see Algorithm 2). This requires a data structure to store the entire utilization profile of each resource of the system. This profile keeps trace of the utilization of



(A) Pre-backfilling



(B) Post-Backfilling

FIGURE 2.2: Scheduling example per and post Backfilling

Algorithm 2 ConservativeBackfilling(J = list of waiting jobs, N = list of nodes, Rule1 = jobs ordering rule, Rule2 = nodes ordering rule, Strict = true if strict ordering, ct = current time)

```
1:  for i in $1..|J|$ do
2:      foreach node $n$ in $N$ do
3:          Let $est_{i,n} = -1$
4:          while $est_{i,n} == -1$ do
5:              find the minimum possible start time $est_{i,n}$ for job $j$ on the node $n$
6:              if for the entire job execution, the utilization profile has not enough free resources then
7:                  $est_{i,n} = -1$
8:              end if
9:          end while
10:     end for
11:     if a set of $est_{i,*}$ with the same start time with cardinality equal to the number of job units have is found then
12:         update the utilization profile
13:         if $est_{i,n} == ct$ then
14:             run the job
15:         end if
16:     end if
17: end for
```

the running job and also of the waiting jobs after the computation of an estimated starting time. At each termination, the utilization profile is updated to free idle resources (due to the walltime overestimation problem). And then, the backfilling algorithm try to schedule waiting jobs in the idle resources exploiting the utilization profile to not create overutilization or delays.

The EASY-Backfilling algorithm is designed to increase the system utilization in a less fair way w.r.t. the conservative backfilling: The algorithm, unlikely the conservative, tries not to delay just the first waiting job. The

Algorithm 3 EASYBackfilling(J = list of jobs, N = list of nodes, Rule1 = jobs ordering rule, Rule2 = nodes ordering rule, Strict = true if strict ordering, ct = current time)

```
1: Let $OJ$ the list of running jobs from $J$ ordered by expected termination time
2: Let $w1$ the first waiting job from $J$
3: Let $W$ the list waiting job from $J$ but $w1$
4: **foreach** node $n$ in $N$ **do**
5:     Let $est_n = -1$
6:     **while** $est_n == -1$ **do**
7:         find the minimum possible start time $est_n$ for job $w1$ on the node $n$ iterating on $OJ$
8:     **end while**
9: **end for**
10: Let $fistWStart$ the minimum start time for the execution of all the job units in $est_n$
11: **foreach** job $w$ in $W$ **do**
12:     **if** $w$ can be executed on the free resources and terminates before $fistWStart$ **then**
13:         run the job
14:     **end if**
15: **end for**
```

algorithm (see Algorithm 3) computes the expected starting time of the first queued job and stores it with the assigned nodes. After that, it tries to schedule all the remaining jobs that can execute and terminate before the starting time of the first job.

In the works presented by Feitelson [65], Alem [66], a study on performances of these two different backfilling algorithms can be found: the study evaluates conservative backfilling versus EASY backfilling providing guidelines on their potential selection.

Many extensions to these algorithms have been proposed. The work presented by Yuan et al. [67] show a new version of the EASY backfilling algorithm to take into account fairness. As for the main scheduling algorithm for HPC, this is a greedy algorithm and does not explore solutions to get a local optimum. However, they propose an interesting concept of fairness that is achieved when a job start time is not delayed by a lower-priority job. This concept could lead to starvation. In our work, we propose a different concept of fairness where the job waiting times have to be distributed on the basis of the ratio between job priorities.

Another example of user-aware scheduling can be found in the work of Shmueli and Feitelson [68]. This work prioritizes jobs by the estimated response time and the seniority factor (minutes of waiting of the job). Then it applies the EASY backfilling algorithm.

However, all these greedy algorithms do not guarantee neither global nor local optimality of the solution.

2.3 HPC scheduling optimization

The problem studied in this work is a resource-constrained project scheduling problem (RCPSP) [44]. In the literature a plethora of works on this subject can be found [69, 70, 71].

Focusing on search-based schedulers, it is hard to find in the literature examples of optimization algorithms applied to a real in-production HPC scheduler. Sarood et al. [72] show an ILP model to constrain the power usage within the resource manager. This work is based on assumptions that do not hold in general for HPC workloads. For example, it proposes to improve the overall execution time by increasing/decreasing the number of nodes used by a job even during its execution. This is not possible in many HPC production environments where resources are locked to the job for its entire duration. In addition, the experiments in the work are made only by simulation on trace-log on a system that is smaller than current HPC standards.

In a wider context, there is a large body of literature on scheduling and allocation for data-center workloads [53, 54, 55, 56] relying on the key assumption that partial or complete migration of parallel jobs is possible during their execution. Even though supercomputers will reasonably move toward more agile execution models [57, 58], the common practice today is that job migration is not allowed, to maximize performance and predictability [59].

In the work by Soner et al. [73] we find another example of optimization in scheduling. The proposed solution always schedules jobs in arrival order and models job dispatching as an assignment problem. Differently from the approach described in the following sections, Soner et al. do not consider the very significant optimization opportunities that emerge when jobs can be extracted from queues in non-FIFO order.

An interesting approach can be found in the work of Kessaci et al. [74]. This is a meta-scheduler that uses multi-objective genetic algorithms to decide in which data center of a grid to send jobs, in order to optimize CO_2 emissions, energy consumption and profits providing a set of Pareto solutions. This work differs from the present one for the assumption behind the model: the authors consider the presence of *hard*-deadline for the jobs and one job can be dispatched to only one node using a FIFO policy. In our case study, *hard*-deadlines are not considered and each job can request more than one node.

In the works presented by Wang and Raicu [75] and in the work presented by Jones and Nitzberg [76] some interesting studies on schedulers performance and scalability are described: different infrastructure setups and greedy algorithms are compared to scale to larger scale HPC machines.

To the best of our knowledge, the only examples that apply optimization techniques to a scheduler in a production context are presented by Klusáček et al. [77] and Chlumsky et al. [78]. In these papers, the authors present an optimization technique applied to a scheduler. The second is developed as an extension of the open-source TORQUE scheduler. This extension replaces the scheduling core of the framework with a backfilling-like algorithm that inserts one job at a time into the schedule starting from a previous solution and then applies a Tabu Search to optimize the solution. Both these works use Tabu search to explore a number of local optimal solutions and consider a job as a set of resources. This assumption drastically decreases the flexibility of the scheduler by avoiding the possibility for a job to request more than one node. In our work, we consider jobs requiring a set of resources. In this way, we maintain the flexibility of commercial schedulers (like TORQUE and PBS Professional) but we deal with a more complex problem w.r.t. the work of Chlumsky.

The work presented by Shmueli and Feitelson [79] shows an interesting approach for the optimization of the backfilling algorithm. This approach exploits dynamic programming to improve results obtained by the classical backfilling algorithm to maximize the system utilization. However, the author considers only the case of one type resource, neither different kind of resources nor heterogeneous resources are considered, and a comparison with this work cannot be done.

The work presented by Tsafrir et al. [80] focuses on the execution-time prediction. The suggested technique uses the last two jobs execution from the same user to predict the job execution-time. A key point of the approach is that this prediction is used only for the scheduling and it does not substitute the job's walltime. This approach is shown to be lightweight and efficient, and differently from other approaches, it does not expose users to the risk of premature job killing. The authors state that this approach can be added to every classical backfilling scheduler, but this approach can profitably be added even to more complex scheduler like ours. However, the focus of our work is on the scheduling algorithm. For this reason, we will investigate the behavior of this technique applied to our CP scheduler in future works.

Several works such as [81] demonstrated that a job scheduler can be proactively used to constrain the power consumption at run-time by setting a desired power profile and schedule on the machine only the jobs which satisfy this constraint. This has the potential of reducing system overprovisioning. Moreover, the cooling power and cost required to cool down the heat generated by the system jobs depend on the overall power and environmental conditions [82]. Borghesi [81] shows that by dynamically modulating the power profile according to the environmental temperature, it is possible to improve the overall energy efficiency. As a matter of fact, this scenario requires to schedule a set of large number of jobs (jobs) in a large number of resources (nodes) while satisfying a variable/desired profile (power budget) which is variable in time.

Hurley et al. [83] tackled the energy-optimization problem at meta-scheduling level. The authors combine optimization and machine learning techniques, to minimize the energy expenses even in the case of variable and unknown a priori energy price.

2.4 Distributed scheduling

Distributed scheduling is an emerging approach designed to split and parallelize the computational overhead through different entities. The side effect of this approach is the difficulty in the entity synchronization: i.e. separated entities, in certain scenarios, have to agree on the scheduling result.

In many works, the scheduling agreement is not necessary (e.g. the Grid computing scheduling problem). The following are examples of this scenario.

In the work by Lu and Kumar [84], the authors present a distributed approach to scheduling. The problem consists of a set of centers in which activities are dispatched. Each activity has to pass through and execute in each center. Each center runs a heuristic algorithm to select from its activities queue, the activity to execute. In this work, the dispatching algorithm is a round-robin and the scheduling algorithm is a rule-based algorithm. While there is a similarity with distributed scheduling, this work has significant differences in the dispatching.

Ramamritham et al. [85] present a distributed scheduler. The proposed approach is based on bids for the dispatching. These bids can be random or based on estimations. This could lead to the condition in which a job has to migrate to avoid exceeding its deadline. In our work, we do not use estimations, and the dispatching phase considers all the system resources. For this reason, our work does not need the job migration, and if a job exceeds its deadline it is due to the high utilization of all the resources of the system.

The KDistr scheduler has presented in [86]. The system is composed of a hierarchy of meta schedulers with one root. All jobs are submitted to the root, then the root sends the job to K meta schedulers. The first scheduler that executes the job informs the other schedulers that the job is already in execution. Due to the fact that different schedulers can compute the scheduling of the same job at the same time, the authors use an atomic scheduling cycle.

Moreover, this is a meta-schedulers. This means that a job can execute only into one cluster/node. In our case, we deal with nodes instead of clusters and the main difference is that a job can have different job units executing in parallel on different nodes. For this reason, this scheduler does not work for the HPC scheduling problem.

A number of works using Particle Swarm Optimization for the scheduling can be found in literature [87, 88, 89]. These algorithms are optimization algorithms that explore a set of feasible solutions. The problem with these algorithms is the computational overhead. The best result obtained in this paper on a number of nodes and jobs halved w.r.t. our tests, show a computational overhead 6 times higher than ours. Distributed implementations of this approach have been studied [90] for different kinds of problems but never applied to scheduling.

The work presented by Montresor [91] shows the application of an ant colony algorithm to the problem of the scheduling in peer-to-peer systems. In this scheduler, the resources are nests, the ants have the duty to migrate jobs from highly loaded resources to low loaded resources. The starting assumption of this work is that a job can be migrated even during its execution. This assumption is not true in the majority of the domains studied by our work. Moreover, the authors consider only the load balancing objective. Finally, the ant colony approach does not consider the scheduling horizon for further optimization.

The work presented by Ortiz et al. [92] shows a distributed approach to the task management of activities in robotics systems. For what concerns the activity dispatching, each agent applies two possible rules: the first searches for the nearest goal, the second searches the further goal. This is one of the cases in which our scheduler could introduce further optimization considering not only the dispatching but also the activity scheduling.

An other scenario that considers a complete agreement between the entities is the consensus problem. Many works [93, 94, 95] studied this problem. However, the considered problem aim to a global agreement on the result. This constraint is too strict w.r.t. the scenario studied in this thesis: we need an agreement just between the nodes executing the same jobs. Moreover the complexity if this constraint leads to a time-to-solution not feasible in the HPC scheduling problem.

Optimization techniques have been applied to the problem of distributed scheduling [96, 97, 98]. However, as demonstrated in [3], centralized optimization approach cannot scale up to large-size systems. These distributed approaches add to the overhead of a centralized approach also an overhead due to communications between agents. For this reason these approaches are unfeasible in a real-time HPC scheduler.

2.5 On HPC energy, profit, and scheduling

Conficoni et al. [99] establish the relation between the cooling cost in HPC infrastructures, the IT power consumption, and the external ambient temperature. Indeed, accordingly to these parameters, the cooling circuitry operates at different set-point with different combinations of power consumption and external temperature. In air-cooled data centers, the variability of the power usage efficiency can range from 10 to 40% while, in case of hybrid cooling, the range is 9 to 13% depending on the external temperature fluctuations.

In the last years, many works have been proposed to improve the HPC scheduling problem. Some works target user's experience related metrics or the system utilization [**galleguillosdata**, 100, 3, 79, 80] (e.g. decreasing the users' waiting or improving the users' fairness). While other works focus on energy consumption [101, 102, 103, 104] (e.g. using power capping techniques). However, reducing costs does not always give the best incomes.

Some works started exploring the scheduling optimization problem aiming to improve the profit such as Zhao et al. [105]. Zhao et al. [105] propose a scheduler for cloud computing that takes into account the service level agreement to indirectly improve the profit. The difference between our work and this is that the complexity of our scheduling problem is greater, in fact, in cloud computing, the resource management is left to a resource manager while the scheduling is left to the meta-scheduler. Moreover, our work directly optimizes the profit.

Moghaddam et al. [106] propose a scheduler for distributed data-centers that optimizes the profit taking into account the incomes, the expenses and a penalty for the utilization of energy by "non-green" origins (e.g. carbon combustion). However, this work is suitable for distributed data-centers in which the origin of the electrical power is known a priori and fixed in time.

Faragardi et al. [107] study the problem of energy consumption and profit maximization in the geographically distributed computing centers. The goal of this work is to find a good trade-off between CO_2 emissions and profit improvement. However, this work does not consider the possibility of shut-down parts of the system as we do. Shutting down parts of the system enables the possibility of a win-win solution in which the energy consumption could be decreased and the profit increased at the same time.

Although resource shutdown and turn-on have been studied in fields like cloud computing, the only works that exploit the possibility of node shutdown and turn-on in HPC machines, at the best of our knowledge, are Hikita et al. [108] and Mammela et al. [109].

Hikita et al. [108] consider the case of the Kyoto University's HPC machine. The proposed scheduler is designed to act as a rule-based scheduler with the possibility to turn-off nodes after 30 minutes of idle and turn it back on after 30 minutes of high resources request. This approach, differently from ours, is a reactive approach. This is translated into 30 minutes of idle consumption of each node before to turn it off, 30 minutes of buffering before the restart, and 45 minutes of waiting before the restart completion. Moreover, this scheduler is highly prone to instability: under certain conditions (e.g. jobs request that creates high resource fragmentation or periodic submissions), this scheduler can continuously shut-down and turn-on nodes. Finally, the assumption of [108] is that the submissions distribution has a seasonality with long periods of low workload requests. This can be true for university workload but not for HPC designed for industry and research [110].

Also Mammela et al. [109] propose a rule-based scheduler capable of shutdown and turn-on nodes. As for the work by Hikita et al., this is a reactive scheduler that shutdown nodes on the basis of a threshold on the node idle time. In this case, the threshold is 50 seconds. Again, a reactive approach can lead to the condition in which the scheduler continuously shutdown and turn-on nodes. The behavior of the scheduler bring also security concerns: the behavior of the system can be exploited to create a denial of service. This seems an improvement w.r.t. the Hikita [108] work but in a real HPC system this is still an inapplicable solution.

2.6 Costraint Programming

Constraint Programming (CP) is an optimization technique belonging to the area of Artificial Intelligence (AI).

CP is a declarative programming paradigm in which the user can formulate a model, which is then fed to a solver that explores the space of possible solutions to find the best one (according to a given objective function).

This paradigm couple a Constraint Satisfaction Problem (CSP) to an objective function. Formally speaking, a CSP can be represented by a tuple $< X, D, C >$ in which X is a set of variables, D is the domain of the corresponding variable in X and C is a set of constraints. For each variable $X_i \in X$ we have a domain $D_i \in D$. This means that each value of D_i can be assigned to the variable X_i. With the constraint C_j we can further bound the domain of the variable in the scope $C_j.S$ of C_j. Constraints are designed to remove elements from the domain of the scope variable to remove unfeasible assignment. A solution is a variable assignment that holds for each constraint of the problem.

In the majority of the open problems in computer science, there are a high or infinite number of feasible solutions but just one or some of these are the best. For this reason CP is designed to solve CSP with the possibility to have an objective function that model the goodness of a solution 2.1.

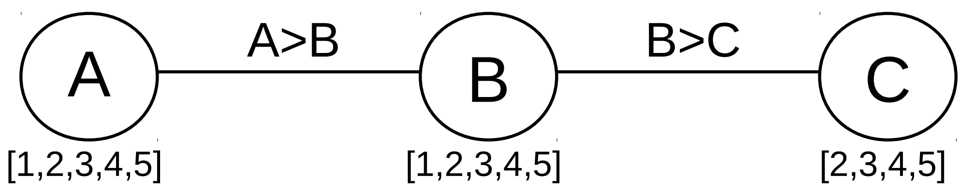
$$CP = CSP[+objectivefunction]^{*} \tag{2.1}$$

This process is similar to that of Mixed Integer Linear Programming (MILP). However, unlike in MILP, in CP a user is not forced to employ only linear constraints: instead, a model can be formulated using any constraint from a given (solver-dependent) library. These constraints have a semantic (i.e. they enforce certain properties on the solutions), and they are associated with one or more *filtering algorithms*. At search time, the solver interleaves branching decisions with invocations of the filtering algorithms, which examine the domains of the problem variables and remove values that are provably infeasible: by doing so, they enable (possibly dramatic) reductions of the search space. The CP research community has developed specific constraints (and filtering algorithms) for scheduling, which usually allow a CP solver to outperform a MILP one on this class of problems [111, 112].

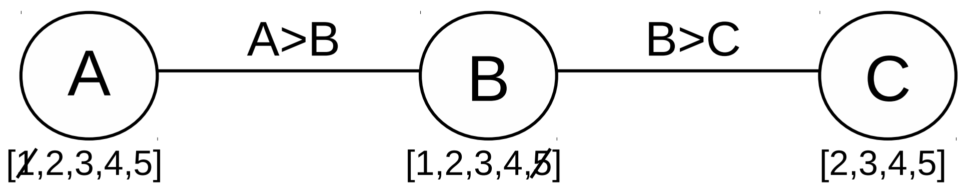
2.6.1 Constraint filtering and propagation

Each constraint is associated with its filtering algorithm. A filtering algorithm is an algorithm designed to remove provably infeasible elements from the domain of the variables in the domain scope. In Figure 2.3 Step 0 shows an example of a problem. In the problem, we have three variables A, B, and C each one with domain $[1, \ldots, 5]$ and two constraints $A > B$ and $B > C$. Figure 2.3 Step 1 shows the result of the filtering algorithm applied on the constraint $A > B$ that deletes 1 from A and 5 from B. After the filtering, the changes in the domain of a variable trigger the filtering on all the constraints containing the same variable in the scope. This step is called *constraint propagation*. Figure 2.3 Step 2 show the propagation triggered by the changes on the domain of B that calls the filtering algorithm on the constraint $B > C$. The filtering deletes 4 and 5 from C and 1 and 2 from B. This process is repeated until no more values can be deleted from the domain of the variables:

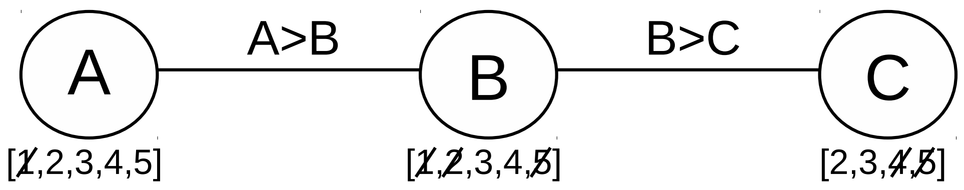
*The notation [..] in this case represents optionality



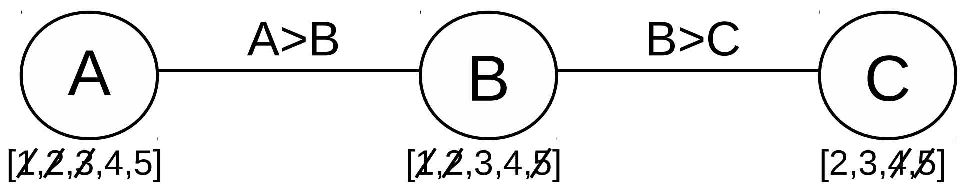
(A) Step 0



(B) Step 1



(C) Step 2



(D) Step 3

FIGURE 2.3: Filtering and propagation example

The further change on B calls the filtering algorithm of $A > B$ deleting 2 and 3 from A (Figure 2.3 Step 3).

2.6.2 Modeling

CP has powerful modeling syntax. With CP it is possible to model classical unary and binary constraint as in MILP (e.g. $A > 3$ and $A > B$) but also nonlinear constraints such as $A \neq B$. Moreover, CP introduces the concept of global constraints [113]. A global constraint is a constraint that involves a set of variables and it encapsulates a set of different constraints. An example of global constraint is the $AllDifferent(X)$. This constraint target a set of variables X and can be formally expressed as in equation 2.2.

$$\forall\, i, j \in |X|, with\ i \neq j : X_i \neq X_j \tag{2.2}$$

Suppose $X = \{X_1, X_2, X_3\}$ with $X_1 \in 1, 2, 3$ and $X_2, X_3 \in 1, 2$, the constraints generated by the global constraint $AllDifferent(X)$ will be:

$$X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3 \tag{2.3}$$

Modeling these constraints with binary constraints and enforcing Arc Consistency [114] no value can be excluded from the variable domain. However, it can be noticed that X_2 and X_3 can assume only values 1 and 2. Thus, these values can be deleted by the domain of X_1. This is achieved by by the $AllDifferent$ constraint due to the fact that global constraints have a wider knowledge of the problem while binary constraints only consider two variables at time.

The difference between CP and CSP relies on the possibility to add an objective function to the model. This function tells how a solution, among all the feasible solutions, is good. Considering the previous example, in which we have to assign a different value to a set of three variables, each one with a domain $[1, \ldots, 4]$, we can for example model a problem in which the best solution is the one with the highest possible values as result. This problem will be modeled as in equation 2.4.

$$\begin{aligned} & X = \{X_1, X_2, X_3\} \\ & x = [1, 2, 3, 4] \qquad \forall x \in X \\ & AllDifferent(X) \\ & maximize : \sum_{x \in X} x \end{aligned} \tag{2.4}$$

One of the optimal results obtainable from this model is $X_1 = 4, X_2 = 2, X_3 = 3$.

CP modeling for scheduling problems

In this section, we will discuss the CP modeling for scheduling problems. The problem we target here is the scheduling on non-preemptive jobs (i.e. each started job cannot be stopped until its termination), with fixed durations and fixed resources requirement, on a set of resources with fixed capacity (i.e. *cumulative* resources).

To explain this we introduce the concept of *interval variable* [115]. An interval variable is a variable that models an activity (or job). This kind of variable contains several different decisional variables. For sake of simplicity, we can say that an interval variables a is composed by a decisional variable for its starting time $a.st$, a decisional variable for its duration $a.d$, a decisional variable for its end time $a.et$ and a decisional variable for its presence $a.p$. If an interval is present, all the constraints on this variable propagate otherwise this variables do not propagate. The interval variable has also implicit constraints that model its consistency (e.g. the end time is equal to the start time

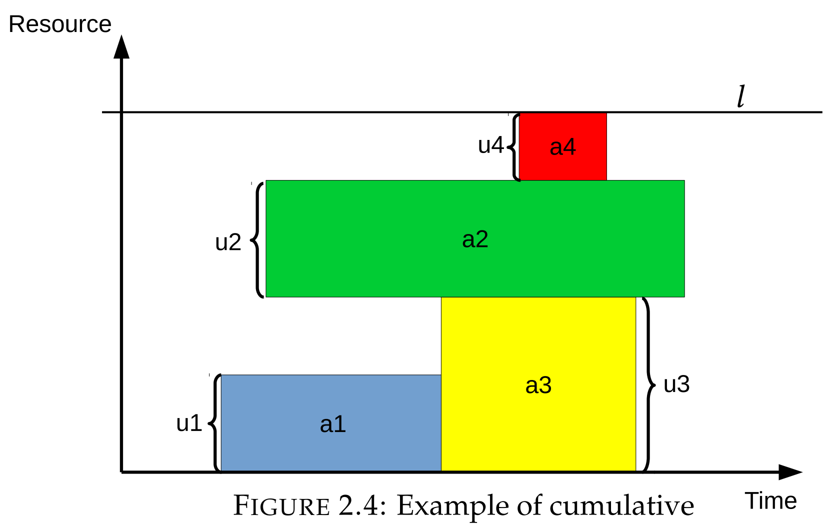


FIGURE 2.4: Example of cumulative

plus the duration). These constraints are formally explained in equation 2.5 where inf is the highest number representable in the used architecture.

$$
\begin{aligned}
&a.et = a.st = a.d = [-inf, \ldots, inf] \\
&a.p = [-1, \ldots, 1] \\
&(a.p == 1) \times (a.et = a.st + a.d)
\end{aligned}
\tag{2.5}
$$

Now we explain some of the most used constraints in scheduling and also in this thesis.

Synchronize The synchronize constraint $synchronize(a, b)$ is designed to synchronize two different interval variables. This means that the variable assumes the same start time, end time, duration and presence.

NoOverlap The no-overlap constraint $NoOverlap(a, b)$ is designed to set two activities to do not be active at the same time. Formally speaking, the interval variable a can start after the end of b or the variable a have to end before the start of b (Equation 2.6).

$$a.st \geq b.et \vee a.et \leq b.st \tag{2.6}$$

Cumulative The cumulative constraint $Cumulative(A, U, l)$ (Figure 2.4) is designed to schedule a set of activities, that requires a certain amount of resources, to never overpass the resource capacity. Formally speaking, having a set of interval variable $A = \{a_1, \ldots, a_n\}$ each one with a resource requirement $U = \{u_1, \ldots, u_n\}$ and a physical resource capacity l, the constraint hold iff the resource utilization of the resources never overpass the capacity in any time instant.

This constraint is used in HPC scheduling to limit the utilization within a single node resource.

Alternative The alternative constraint $Alternative(a, B, n)$ is designed to select n elements from the set of activities B to be synchronized with a. Formally speaking, a is a *present* alternative variable, B is a set with cardinality $\geq n$ of *optional* interval variables, and n is an integer. The constraint holds iff there are exactly n variables in B that can be set as *present* and synchronized with a (i.e. they assume the same start time, duration and end time). The remaining variables in B are set as *not present*.

Element The element constraint $c = Element(a, B)$ is designed to select an element c from the array B on the basis of the decisional variable a used as index of the array. This constraint simply select the a element from the array B but this selection is done at solving time, this means that a changes during the search phase of the solving engine and the variable a can be connected to other constraints. The element is then returned to c.

2.6.3 Search strategies

The constraint filtering and propagation usually terminates reducing the variable domains. However, to have a solution each variable has to be fixed. Even with a program-level complete propagation algorithm this can be achieved just in trivial problems. Thus, a search strategy have to be adopted to obtain a solution.

Constraint Programming gives the flexibility to choose from a wide selection of search strategies for the problem solution. Search strategies can be subdivided into three main different categories: Backtrack searches, Local searches, and Dynamic searches [116, 117].

A backtrack search for CP problems starts with a depth-first solution-tree traversal. At each step, the search strategy selects a variable to label (*labeling*) and then starts a propagation stage for each constraint targeting all the variables with a change in its domain. When a failure is reached, the strategy applies a backtrack (*branching*) on the previous decision point. Note that the propagation can apply not only to the model variables but also to equations, as for example the objective function. The representation of the objective function through a decisional variable let the solver propagate constraints e.g. when a new best solution is found that, hopefully, dramatically decrease the solution tree. Several heuristics can be selected for the branching decision. Different heuristics can lead to different results. A commonly adopted selection criterion is the "First-Fail Principle" [118]. This criterion is designed to delay failures. The criterion selects the variable with the smallest domain. In case of ties, it selects the variable involved in the highest number of constraints. Other techniques (such as [119]) learn the impact of each variable on the solution space reduction and select the branch using this information.

Improvements on the search can also be obtained by "no-goods" [120, 121, 122]. No-goods are designed to prevent the solver to repeat decisions that always lead to failures. This forward check learns from previous failures what caused it and then a constraint to avoid this situation is generated.

One of the most used and effective search strategies is the Large Neighborhood Search (LNS) [123]. This search combines the strength of the CP constraint propagation with the performance of local search. This search strategy starts from an initial solution of the problem. One a first solution has been obtained, LNS selects a set of variables to be fixed and a set of open variables. The fixed variables are assigned to the value of the last solution found while the local search explores the neighborhood of the previous solution reassigning different values to the open variables. After the local search termination, a new set of fixed and open variables is selected allowing the search to jump to an another neighborhood to explore. Usually, the termination condition of a neighborhood exploration is determined by the completion of the exploration or the reaching of a predetermined number of fails, depending on the size of the neighborhood. For what concern the neighborhood exploration, different search strategies can be used (such as the one in section 6.7). However, the most used is the schedule-and-postpone. The schedule and postpone strategy selects the first unfixed variable and set it to the minimum value in its domain. If this value is not feasible (it leads to a failure), mark this variable to be left aside during the search. After that, it continues with the next unfixed variable. When the domain of a variable left aside changes, the strategy removes the mark and from that moment can be selected to be assigned.

Chapter 3

Preliminary study on the application of CP in HPC scheduling: modeling and simulations

3.1 Introduction

Computing centers play a key role in modern ICT architectures: they run our internet services, keep track of our savings, make our research possible. They are also well known to be power hungry: in Italy, data centers make for $\sim$2% of the national energy consumption, for a total of 6.6 TWh (roughly that of the Calabria region, according to data by Fondazione Politecnico di Milano, 2010).

The mainstream solution to reduce such a gigantic consumption is to employ efficient hardware or efficient design. By doing so, it is possible to obtain remarkable reductions of the PUE index (Power Usage Effectiveness), i.e. the ratio between the power consumption of the whole data center and the power consumption of the IT equipment alone. Recently, a joint effort by the CINECA inter-university consortium [124] in Italy and the Eurotech group [125] has led to the design of the EURORA system. Thanks to an innovative liquid based cooling system and carefully chosen hardware components, this new machine has a PUE of just 1.05 and managed to reach the top of the Green 500 ranking in the first half of 2013, effectively becoming for a time the most efficient supercomputer on earth. As a comparison, PUE values of around 3 were still common in 2009.

However, reducing the PUE is just a half of the problem. Data by McKinsey [126] for US data centers reveals that on average only 6-12% of the power is employed for actual computation. The reason for this dramatically low value lies in *how efficiently the existing IT resources are used*. In particular, redundant resources are usually employed to maintain the quality of service under workload peaks. More redundant resources are also needed to compensate for the fragmentation resulting from suboptimal dispatching choices. As a consequence, a typical data center ends up packing a lot of idle muscles. Unfortunately, idle resources still consume energy: for a 1MW center with a 1.5 PUE, a 30% utilization means a 1M€ annual cost and 3,500 tons of CO_2.

In this context, optimization techniques can enable dramatic improvements in the resource management, leading to lower costs, better response times, and fewer emissions.

In this work, we tackle the problem of performing workload dispatching over the EURORA supercomputer, operating at the CINECA computing center in Bologna. The machine is employed for High Performance Computing (HPC) applications and has a job submission system currently managed by a PBS Dispatcher (Portable Batch System [127]). The dispatcher relies on a number of heuristic techniques to tentatively maintain a high machine utilization and keep the waiting times as small as possible. The CINECA staff has hints that the current system operation could be improved, but finding a more effective PBS configuration is a cumbersome and error-prone task: hence there is interest in alternative approaches. We propose to tackle workload dispatching via proactive scheduling using Constraint Programming. We adopt a rolling horizon approach, where our scheduler is awakened at certain events. At each of such activations, we build a full schedule and resource assignment for all the waiting jobs, but then we dispatch only those jobs that are scheduled for immediate execution. By taking into account forthcoming jobs, we avoid making dispatching decisions with undesirable consequences; by starting only the ones scheduled for immediate execution, the system can manage uncertain execution times.

Our long-term goal is the development of a state of the art workload dispatching approach to replace the current PBS logic. However, at this stage, our main objective is just is to assess the degree of improvement (in terms of waiting times and reduced idleness) that can be obtained by acting on the dispatching decisions. Since our focus is on investigating the solution quality, we do not enforce tight restrictions on the approach run-time (over-exploiting a bit the fact that HPC jobs tend to have large durations). We evaluated our approach by simulating its behavior on real workload traces from the EURORA machine. We compare the results of our approach with those of the currently operating PBS system, demonstrating that substantial improvements are indeed possible.

3.2 System Description and Motivations for Using CP

This section contains a brief presentation of the architecture of the EURORA supercomputer, a discussion about the current dispatching system, and a review to the motivations behind our choice of CP for building an alternative dispatcher.

The EURORA Supercomputer: As described in [100] EURORA has a modular architecture based on nodes (blades). In its the current state, the system counts 64 nodes, each one comprising 2 octa-core CPUs and 2 expansion cards configured to host an accelerator module: currently, 32 nodes host 2 powerful NVidia GPUs, while the remaining ones are equipped with 2 Intel MIC accelerators. Each node has 16GB of installed RAM memory. EURORA is interfaced with the outside world through a few dedicated computing nodes, physically positioned outside the rack: in particular, a designated login node connects EURORA to the users and runs the job dispatcher (PBS). One of the main boosting factors for the energy efficiency of the supercomputer is the adoption of a hot liquid cooling technology, i.e. the water inside the system can reach up to 50°C. This strongly reduces the energy required for operating the system, since no power is used for actively cooling down the water, and the waste-heat can be recovered as an energy source for other applications.

The PBS Dispatcher: The tool currently used to manage the workload on EURORA system is PBS (Portable Batch System), a proprietary job scheduler by Altair PBS Works with the primary duty of allocating computational tasks, i.e. batch jobs, among available computing resources. The main components of PBS are a server (which manages the jobs) and several daemons running on the execution hosts (i.e. the 64 nodes of EURORA), which track the resource usage and answer to polling request about the host state issued by the server component.

Jobs are submitted by the users into one of multiple queues, each one characterized by different access requirements and by a different approximate waiting time. Users submit their jobs by specifying 1) the number of required nodes; 2) the number of required cores per node; 3) the number of required GPUs and MICs per node (never both of them at the same time); 4) the amount of required memory per node; 5) the maximum execution time. All processes that exceed their maximum execution time are killed. The main available queues on the EURORA system are called *debug*, *parallel*, and *longpar*, and are described in Table 3.1 - for each of those queues we report the maximum number resources that a job could ask if it desires to belong to that queue, i.e. maximum number of nodes, maximum number of cores and GPUs (second column), maximum execution time, and also the approximate time it might wait before starting its execution.

Cyclically, PBS selects a job for execution by polling the state of one or more nodes, trying to find enough available resources to actually start the job execution. If the attempt is unsuccessful, the job is sent back to its queue and PBS proceeds to consider the following candidate. The choices are guided by priority values and hard-coded constraints defined by the EURORA administrators with the aim to have a good machine utilization and small waiting times. For example, the administrators decided to reserve some nodes to the debug queue and to force jobs in the *longpar* queue to start at night.

Why CP? In its current state, the PBS system works mostly as an on-line heuristic, incurring the risk to make poor resource assignments due to the lack of an overall plan. Also the hard-coded mapping constraints, designed as a way to ensure low waiting times for specific job classes (e.g. the *debug* queue), may easily cause resource under-utilization, and long waiting times

Queue	*Max Nodes*	*Max Cores/GPUs*	*Max Time*	*Approx. Wait*
debug	2	32/4	00:30:00	seconds
parallel	32	512/64	06:00:00	minutes
longpar	16	256/32	24:00:00	hours

TABLE 3.1: Access requirements and waiting times for the PBS queues in EURORA

for the remaining jobs (e.g. those in the *longpar* queue). A proactive dispatching approach should intuitively be able to improve the resource utilization and reduce the waiting times without the need of devising such hard-coded restrictions. The task of obtaining a proactive dispatching plan on EURORA can be naturally framed as a resource allocation and scheduling problem, for which CP as a long track of success stories.

W.r.t. other optimization techniques, CP is better suited due to the fact that most of its constraint are specifically designed for scheduling problem. Moreover, CP proved to overcome the performance of other optimization techniques, such as MIP, in scheduling problems [111, 112].

3.3 Design of a CP Approach

We adopt a rolling horizon approach, in which our scheduler is awakened whenever a job 1) enters the system or 2) ends its execution. At each iteration, we build a full schedule and mapping for all the jobs in the input queues, taking into account resource capacity limitations. We consider different performance metrics, which we treat either as objective functions or as soft-constraint. Then we dispatch only those jobs that are scheduled for immediate execution.

The schedule is computed based on the worst-case durations (as provided by the users), but the dispatcher reactivation is triggered by the job *actual* terminations (besides of course by their arrivals). Whenever this occurs, the jobs currently in execution cannot be migrated, but all the waiting ones can be re-scheduled to take advantage of the released resources.

3.3.1 Formal Problem Definition

We can now provide a precise definition of the scheduling problem solved at each activation of the dispatcher. Each job i enters the system at a certain arrival time q_i, by being submitted to a specific queue (depending on the user choices and on the job characteristics). By analyzing existing execution traces coming from PBS, we have determined an estimated waiting time for each queue, which applies to each job it contains: we refer to this value as ewt_i.

When submitting the job, the user has to specify several pieces of information, including the maximum allowed execution time D_i, the maximum number of nodes to be used rn_i, and the required resources (cores, memory, GPUs, MICs). By convention, the PBS systems consider each job as if it was divided into a set of exactly rn_i identical "job units", to be mapped each on a single node. It is therefore convenient to specify the resource requirements on a job-unit basis. Formally, let R be a set of indexes corresponding to the resource types (cores, memory, GPUs, MICs), and let the capacity of a node k for resource $r \in R$ be denoted as $cap_{k,r}$. We recall that the system has $m = 64$ nodes, each with 16 cores and 16 GB of RAM memory; 32 nodes have 2 GPUs each (and 0 MICs), and the remaining 32 nodes have 2 MICs each (and 0 GPUs). Finally, let $rq_{i,r}$ be the requirement of a unit of job i for resource r. The dispatching problem at time t consists in assigning a start time $s_i \geq t$ to each waiting job i and a node to each of its units. All the resource capacity limits should be respected, taking into account the presence of jobs already in execution. Once the problem is solved, only the jobs having $s_i = t$ are actually dispatched.

Informally speaking, in the big picture, the goal is to increase the resource utilization and reduce the waiting times, but those metrics can be meaningfully evaluated only once the actual job durations become known. Hence we formulate the problem in terms of several objective functions that are intuitively correlated with the metrics we are interested in. After extensive preliminary experimentations, we settled for the following possible problem objectives:

$$\max_{i=0..n-1} (s_i + D_i) \qquad \text{(makespan)} \qquad (3.1)$$

$$\sum_{i=0..n-1} \max\left(0, \frac{s_i - q_i - ewt_i}{ewt_i}\right) \qquad \text{(weighted tardiness)} \qquad (3.2)$$

$$\sum_{i=0..n-1} [[s_i - q_i > ewt_i]] \qquad \text{(num of late jobs)} \qquad (3.3)$$

where n is the number of jobs and the notation $[[-]]$ stands for the reification of the constraint between brackets. The makespan has been chosen because compressing the schedule length tends to increase the resource utilization. For the tardiness and the number of late jobs, we consider a job to be late if it stays queued for a time larger than ewt_i. The tardiness is weighted, because we assume that users that are already expecting to wait more (i.e. jobs with higher ewt_i) should adjust better to prolonged queue times. Both the tardiness based objectives are chosen to improve the perceived response time, in one case by avoiding (proportionally) long waiting times, in the second by reducing the number of jobs in the queues.

3.3.2 CP Model

Employed CP Techniques: We defined for the described scheduling problem a CP model that is based on Conditional Interval Variables (CVI, see [115]). A CVI τ represents an interval of time: the start of the interval is referred to as $s(\tau)$ and its end as $e(\tau)$; the duration is $d(\tau)$. The interval may or may not be present, depending on the value of its existence expression $x(\tau)$.

In particular, if $x(\tau) = 0$ the interval is not present and does not affect the model: for this situation we also use the notation $\tau = \perp$.

CVIs can be subject to a number of constraints, including the classical *cumulative* [112] to model finite capacity resources, and the more specific *alternative* constraint [115]. This last global constraint has the following signature:

$$alternative(\tau_0, [\tau_1, .., \tau_{n_\tau}], m_\tau) \tag{3.4}$$

The constraint forces all the interval variables $\tau_1, \tau_2, \ldots$ to have the same start and end time as τ_0. Moreover, exactly m_τ of $\tau_1, \tau_2, \ldots$ will be actually present if τ_0 is present. Formally, the constraint enforces:

$$s(\tau_0) = s(\tau_i),\ e(\tau_0) = e(\tau_i) \quad \forall i = 1..n_\tau \qquad \sum_{i=1}^{n_t} x(\tau_i) = m_\tau\, x(\tau_0) \tag{3.5}$$

Modeling Decisions and Constraints: In our model, we use a CVIs to model the scheduling decisions. In particular, we introduce an interval variable τ_i with duration D_i for each job waiting in the input queues or already in execution. Then, we fix the start of all τ_i corresponding to running jobs to their real value (which is known at this point). For the waiting jobs we have $s(\tau_i) \in t..eoh$, where t is the time instant for which the model is built and eoh can be given for example by t plus the sum of the maximum duration of all jobs*. All the τ_i variables are mandatory, i.e. $x(\tau_i) = 1$.

Mapping decisions should be taken at the level of single job-units. The modeling style we adopt for them is best explained by temporarily introducing a simplifying assumption, namely that no two units of the same job can be mapped on a single node. With this assumption, the mapping decisions can be modeled by introducing a second set of *optional* interval variables $\upsilon_{i,k}$ such that $x(\upsilon_{i,k}) = 1$ iff a unit of job i is mapped to node k.

However, mapping multiple units of the same job on the same node is possible and can be beneficial. To account for this possibility, we have to introduce for each job i multiple sets of υ variables. Specifically, we add one more index and we maintain the semantic, so that we have variables $\upsilon_{i,j,k}$ such that $x(\upsilon_{i,j,k}) = 1$ iff a unit of job i is mapped to node k. The j index is only used to control the number of job units that can be mapped to the same node. Finding a suitable range for the index is a critical step: on the one hand, allowing j to range on $0..rn_i - 1$ (i.e. one set of υ variables for each requested node) is a safe choice. On the other hand, it is impossible to map multiple units of the same job on the same node if doing so would exceed the availability of some resource. Hence, a valid upper bound on the number of υ variable sets for a single job i is given by:

$$p_i = \min\left(rn_i, \min_{r \in R} \left\lfloor \frac{cap_{k,r}}{r_{i,r}} \right\rfloor\right) \tag{3.6}$$

*Note that it is possible to shift all the domains by subtracting the smallest s_i to all values, so that at least one $s(\tau_i)$ has a minimum of 0

and for each job i, the index j can range in $0..p_i - 1$. Then we have to specify that exactly rn_i job-units should be mapped, i.e. that exactly such number of $\upsilon_{i,j,k}$ intervals should be present. This can be done by using an *alternative* constraint:

$$alternative(\tau_i, [\upsilon_{i,j,k}], rn_i) \qquad \forall i = 0..n-1 \tag{3.7}$$

Additionally, the alternative constraint forces all the job-units to start at the same time instant as τ_i. Now, the resource capacity restrictions can be modeled via a set of cumulative constraints:

$$cumulative([\upsilon_{i,j,k}], [D_i^{(p_i)}], [r_{i,r}^{(p_i)}], cap_{i,r}) \qquad \forall k = 0..m-1, \forall r \in R \tag{3.8}$$

where m is the number of nodes and the notation $D_i^{(p_i)}$ stands for a vector containing D_0 repeated p_0 times, then D_1 repeated p_1 times, and so on. As mentioned in Section 3.2 we disregard all the hard-coded constraints introduced by the PBS administrator and we trust the decision making capabilities of our optimization system with providing waiting times as low as possible.

Handling the Objective Function: We consider several variants of our dispatching problem, differing one from each other for the considered objective and for the possible presence of soft constraints. First, we have three "pure" models, obtained by adding on top of the presented formulation one of the problem objectives that we have discussed in Section 3.3.1:

$$\min \max_{i=0..n-1} e(\tau_i) \qquad \text{(makespan)} \tag{3.9}$$

$$\min \sum_{i=0..n-1} \max\left(0, \frac{s(\tau_i) - q_i - ewt_i}{ewt_i}\right) \qquad \text{(weighted tardiness)} \tag{3.10}$$

$$\min \sum_{i=0..n-1} [[s(\tau_i) - q_i - ewt_i > 0]] \qquad \text{(num. of late jobs)} \tag{3.11}$$

Then we consider three "composite" formulations obtained by choosing as a main cost function one of Equations (3.9)-(3.11), and then by posting a constraint on the value of the remaining ones. For example, assuming the makespan is the main objective, we get:

$$\min \max_{i=0..n-1} e(\tau_i) \tag{3.12}$$

$$\text{s.t.} \sum_{i=0..n-1} \max\left(0, \frac{s(\tau_i) - q_i - ewt_i}{ewt_i}\right) \leq \delta_0\, \theta_0 \tag{3.13}$$

$$\sum_{i=0..n-1} [[s(\tau_i) - q_i - ewt_i > 0]] \leq \delta_1\, \theta_1 \tag{3.14}$$

The values θ_0 and θ_1 are obtained by solving the pure models corresponding to the constrained functions. The parameters δ_0, δ_1 allow to tune the tightness of the constraints. The three new composite formulations are loosely inspired

i	rn_i	$rq_{i,core}$	$rq_{i,gpu}$	$rq_{i,mic}$	$rq_{i,mem}$	D_i
000	32	4	1	0	1000000	14000
001	1	14	1	0	400000	600
002	2	4	1	0	200000	14400
003	32	16	0	0	400000	800
004	32	3	0	2	800000	400

TABLE 3.2: An example of problem instance

i	$s(\tau_i)$	$v_{i,0,0}$	$v_{i,0,1}$	$v_{i,0,2..31}$	$v_{i,0,32..63}$	$v_{i,1,0}$	$v_{i,1,1}$	$v_{i,1,2..31}$	$v_{i,1,32..63}$
000	0	$\perp$	0	0	$\perp$	$\perp$	0	$\perp$	$\perp$
001	0	1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
002	600	600	$\perp$	$\perp$	$\perp$	600	$\perp$	$\perp$	$\perp$
003	0	$\perp$	$\perp$	$\perp$	0	$\perp$	$\perp$	$\perp$	$\perp$
004	800	$\perp$	$\perp$	$\perp$	800	$\perp$	$\perp$	$\perp$	$\perp$

TABLE 3.3: A feasible solution for the instance from Table 3.2

by multi-objective optimization approaches and aim at obtaining good solutions according to one global metric (say, resource utilization), while keeping acceptable levels for the other (say, waiting times).

Example of a solution: Let us suppose we have the set of waiting jobs described in Table 3.2, then a feasible solution to this instance is described in Table 3.3. As reported in the table, jobs 000, 001 and 002 can execute only on the nodes equipped with GPUs (i.e. node 0 to 31), job 004 can execute only in nodes with MICs (i.e. node 32 to 63). Tow units of job 000 are allocated on node 1, the other 30 units of job 000 are allocated in nodes from 2 to 31; node 0 is completely free and can run job 001 while job 000 is executing; job 003 can execute on nodes from 32 to 63; after the termination of job 001, job 002 can start its execution with two units on node 0 and after the termination of job 003, job 004 can start in nodes from 32 to 63.

3.4 Added Value of CP

The scheduler we realized is currently a prototype: it will eventually be deployed on the EURORA supercomputer, but this requires still considerable development and research effort. At this stage we (and the CINECA consortium) are interested in investigating the kind of improvements that could be obtained by changing the dispatcher behavior. On this purpose, we have compared the results we obtained with our dispatcher and the ones achieved by PBS as it is currently configured on EURORA.

We performed the comparison on real PBS execution traces, which contain all the information that is usually available at the job arrival times (i.e. the chosen queue, the resource requirements, the maximum execution time). Additionally, the traces report for each job two important pieces of information, namely the *actual* duration (which we use together with the arrival time

	Average Queue Time			
Model	All	Debug	Parallel	Longpar
MKS	187.14	4.77	161.81	0.01
MKS WT/NL	165.98	0.10	160.04	0.01
NL	722.04	2.30	316.92	369.14
NL MKS/WT	201.32	0.31	145.59	18.99
WT	662.18	2.16	203.50	446.34
WT MKS/NL	861.81	0.76	278.60	572.29
PBS	6840.81	17.34	2825.05	3600.40

TABLE 3.4: Models comparison, queue times

	Average Resource Utilization				
Model	*cores*	*GPUs*	*MICs*	*cores (%)*	*Avg jobs*
MKS	678.81	45.21	3.99	66%	121.68
MKS WT/NL	701.92	45.61	3.99	68%	121.92
NL	614.75	45.89	3.99	60%	116.58
NL MKS/WT	670.75	45.00	3.99	65%	121.21
WT	671.41	47.67	3.98	66%	120.50
WT MKS/NL	620.45	41.72	3.99	61%	119.07
PBS	447.98	29.16	0.33	46%	63.04

TABLE 3.5: Models comparison, system load

to simulate the scheduler activation events) and the start time assigned by PBS.

Our approach was implemented using IBM ILOG CP Optimizer [128] using its default search strategy, which is based on Self-adapting Large Neighborhood Search [129] guided by an Linear Programming relaxation. At each scheduler activation we use the best solution found within a time limit to decide the jobs that should start. To allow a fair comparison, all traces were pre-processed to reset the waiting time of all jobs that are in queue at the beginning of the trace, so that this is not taken into account. Additionally, we have subtracted from the PBS waiting times the overhead required for implementing the dispatching decision. This was experimentally identified by analyzing the traces themselves.

3.4.1 Evaluation of Our Models

We performed an evaluation of all our models on a PBS execution trace containing data for a batch of jobs that was considered for dispatching in a 2-hour long interval. The main performance metrics considered are (1) the time spent by the jobs in the queues while waiting their execution to begin (ideally as low as possible), and (2) the overall utilization of the system (ideally as large as possible). Waiting times are measure of the perceived quality of services, while a high utilization directly translates to a low number of idle (but still power consuming) resources.

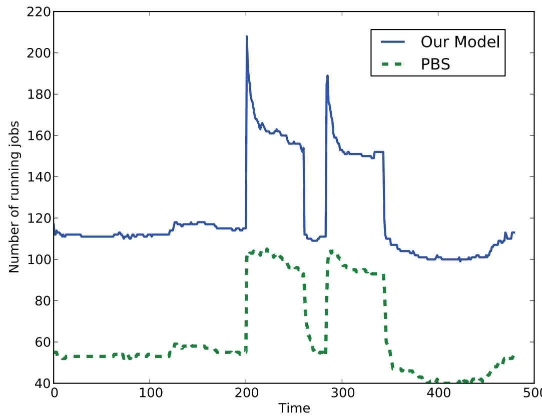
The results for the first batch (BATCH1) are presented in Table 3.4 and Table 3.5; the models evaluated are the three "pure" ones (Makespan [MKS],

	BATCH1	*BATCH2*	*BATCH3*
#jobs	437	434	619
#jobs DEBUG	237	133	127
#jobs PAR	130	240	415
#jobs LONGPAR	62	25	12
#jobs req. GPUs	85	203	224
#jobs req. MICs	3	1	1
#jobs req. 1 core	298	197	258
#jobs req. 2 cores	2	73	38
#jobs req. 4 cores	1	4	7
#jobs req. 5 cores	1	1	0
#jobs req. 6 cores	6	2	3
#jobs req. 8 cores	59	56	187
#jobs req. 8+ cores	70	101	126

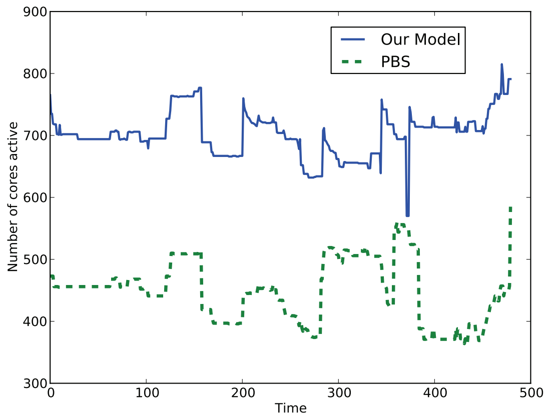
TABLE 3.6: Job traces composition

Weighted Tardiness [WT] and Num. of late jobs [NL]) plus the three composite ones (i.e. with Makespan as main objective and constraints on Weighted tardiness and Num. of late jobs [MKS WT/NL], and similarly for the others). In the table we can see the average waiting time per job (both total and per-queue). There is a remarkable improvement w.r.t PBS for all the models, and those using the Makespan as main objective (MKS and MKS WT/NL). All the composite models perform better than their pure counterparts when dealing with the jobs from *debug* queue (short and with relatively low requirements). The models with Makespan as primary objective do their best when dealing with the long jobs from the *longpar* queue.

The corresponding resource utilization statistics are reported in Table 3.5, showing for each model and PBS the average number of used cores, GPUs and MICs over time. Again, we can see a significant improvement in comparison to PBS performance, but in this case the differences between our models are less clear. In particular, the average numbers of used GPUs and MICs is very similar – probably because not every job needs an accelerator –, but we can notice that MKS WT/NL is the model which performs a bit better in terms of the average number of active cores. In the fifth column of the table we see the average number of jobs that are in execution at each time instant: more running jobs usually correspond to a higher utilization and a smaller time to complete the execution of the batch. Finally in the last column we report the average percentage of active cores on EURORA, which is a good index for the utilization of the whole system. As one can see, our best results (coming from the MKS WT/NL) are around 20% better than those of PBS. No approach was able to reach a 100% utilization: to a large extent, this appears to be due to the presence of bottleneck resources (e.g. GPUs) and to their allocation.

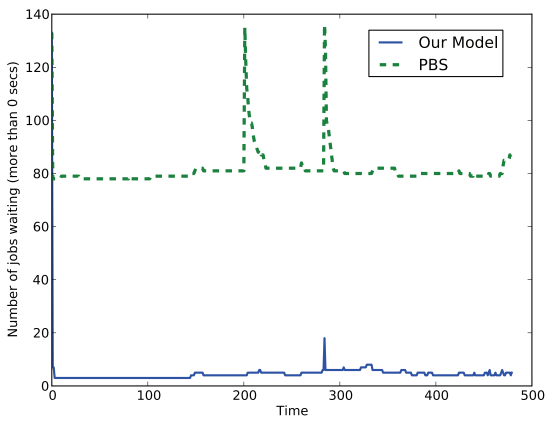


(A) Running Jobs

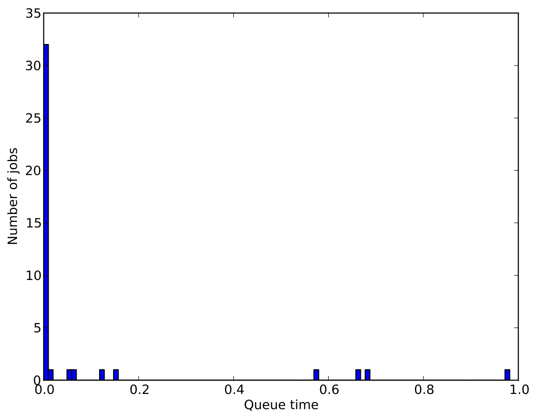


(B) Active cores

FIGURE 3.1: EURORA utilization on the first trace (BATCH1)



(A) Jobs in Queue



(B) Times in Queue

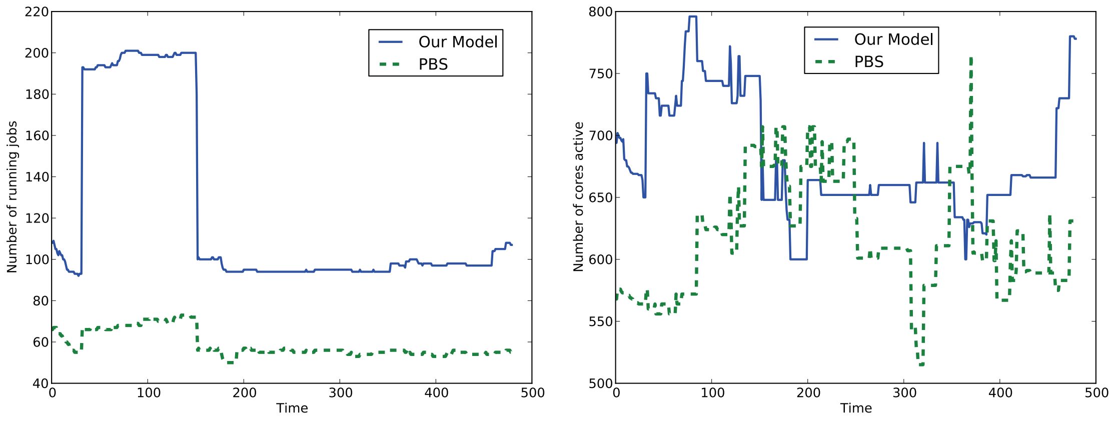
FIGURE 3.2: Waiting jobs and queue time for BATCH1

3.4.2 Comparison with PBS

The previous results show that our best model is a composite one, namely MKS WT/NL, thus such mode was chosen for a more detailed comparison with PBS on three PBS execution traces, each one corresponding once again to a two-hour time frame of the EURORA activity. The features of the job batches considered in each trace (i.e. BATCH1, BATCH2, BATCH3) are summarized in Table 3.6, which reports the total number of jobs, the number of jobs in each queue[†], the number of jobs requiring at least one GPU or MIC and the number of jobs requiring a certain number of cores.

We start by presenting the results for BATCH1, which is the same we used for evaluating the model. The jobs considered in this trace belong to a wide range of classes, with different resource requirements and different

[†]The sum of those values may be lower than the total, because we do not report detailed statistics for some minor queues.



(A) Running Jobs (B) Active cores

FIGURE 3.3: EURORA utilization on the second trace (BATCH2)

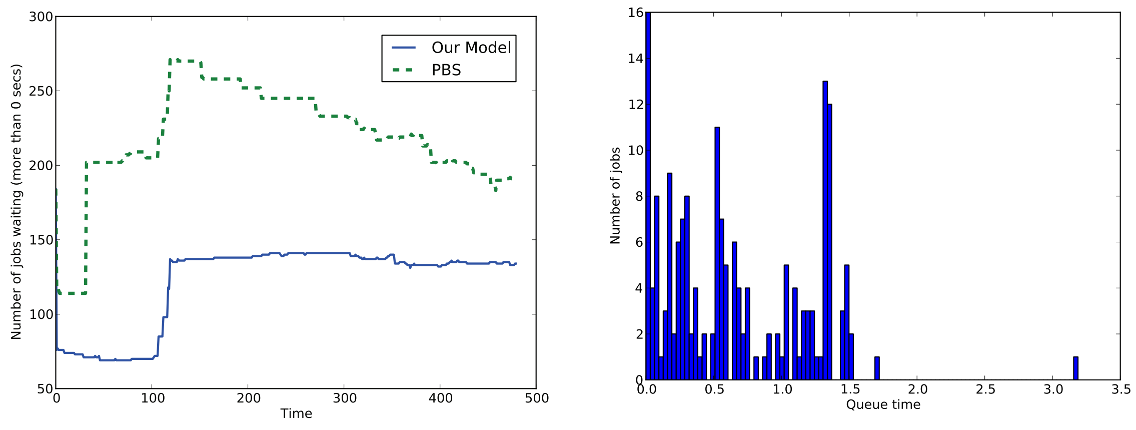
execution times. In Fig. 3.1a we can observe the number of active jobs in the considered time frame, for both our approach (solid line) and PBS (dashed line). Fig. 3.1b reports instead the number of active cores. Our approach significantly outperforms PBS, being able to execute more jobs concurrently and to use a larger fraction of the available cores. Neither approach managed to reach the optimal system usage: this could be due to (a combination of) the presence of bottleneck resources, to suboptimal allocation choices, or simply to the lack of more workload to be dispatched. Fig. 3.2a shows the number of waiting jobs at each time step for our approach and PBS. From the data in the figure, we can deduce that our approach managed to dispatch most of the incoming jobs immediately, suggesting that the machine underutilization is at least in part to blame on the lack of more jobs. Still, suboptimal choices and resource bottlenecks cause some jobs to wait (a relatively high number of them, in the case of PBS).

Fig. 3.2b contains a histogram with the waiting times for our model, weighted by the (inverse of) the Estimated Waiting Time of the queue they belong to. The histogram shows how many jobs (y-axis) wait for a certain amount of times their ewt_i (y-axis). The majority of the waiting jobs with our approach stay in their queue for a very short time, unlike in the case of PBS, where especially the jobs in the *longpar* queue tend to be considerably delayed. We recall that currently these jobs (which are characterized by longer durations than the remaining ones) are forced to execute only at night, for fear or delaying jobs in the $debug$ or $parallel$ queue. The evidence we provide here leads us to believe that such a strong constraint is in fact not needed when using a proactive approach, and its removal could provide benefits in terms of both queue time and average utilization of the supercomputer resources.

Fig. 3.3 and Fig. 3.4 refer instead to our second trace, i.e. to the jobs in BATCH2. This is another mixed group of jobs in terms of computational and resource requirements, but in this case we have many more GPU requests, putting a great strain on the dispatcher since GPUs in EURORA are a much fewer than cores. The consequences of this situation can be observed in Fig. 3.3a and Fig. 3.3a, respectively showing the number of running jobs and active cores over time. For both PBS and our model we notice that the number of jobs in execution, after an initial spike, reaches a cap in the middle section of the trace, although the percentage of actives cores is not even close to 100%. This cap occurs because in many cases, basically all waiting jobs are requiring a GPU and hence, even if there are have available cores, they cannot be used. Despite that, we still manage to achieve a largely improved schedule than the one of PBS in term of number of running jobs. In particular, the average number of active GPUs with our dispatcher is higher than 63: given that the whole supercomputer counts only 64 GPUs, this means that the performance obtained by our approach for the GPU-requiring jobs is very close to the theoretical limit.

We owe this result to the proactive nature of our scheduler, which allow us to more efficiently use constrained resources. For example, suppose we have node A and B, where A has n cores and 1 GPU while B has only n cores, and suppose that A and B are fully occupied by a previous job. We also have *job1* and *job2* waiting to start their execution: *job1* needs n cores and a GPU, whereas *job2* requires only the cores and has higher priority (for PBS). When the job currently occupying nodes A and B terminates, PBS selects *job2*, then it checks if on A there are enough cores to satisfy the requirements. Since this is true in our example, PBS dispatches *job2* on node A, using up all node cores and leaving the GPU idle. In this scenario, *job2* cannot start executing until the other job has terminated. Conversely our dispatcher would have made a smarter - and in this particular case obvious - decision, that is putting *job1* on B, since it only needs cores, and *job2* on A, without further delay. In Figure 3.4 we can see our performance in terms of queue times for BATCH2. We outperform PBS again but at the same time we notice how the number of jobs in queue (Fig 3.4a) follow a similar pattern in both systems, with a distinctive spike after a relatively low initial value: this happens because of the congestion on the GPUs resources we mentioned earlier – after all, optimization can provide improvement only as long as spare resources are available.

Finally, we can eventually consider BATCH3 and the results are displayed in Fig. 3.5 and Fig. 3.6. The jobs considered in this trace require, on average, a higher number of cores than all other traces and, for a large part, were submitted to the $parallel$ queue. They require proportionally fewer GPUs than the jobs in BATCH2, but still more than BATCH1. We manage again to obtain a better usage of computational resources on EURORA, as revealed in Fig. 3.5b and from the average percentage of actives cores (85% in our model versus 55% with PBS). One more time, these results are due to a smarter management of the different types of resources, although the limitations imposed by the relatively low number of available GPUs still has an impact on the number of running jobs (Fig. 3.5a). In Figure 3.6a we can see our model is able not to force to wait as many jobs as PBS, but only during the first half of the trace, while after that point the number of jobs in queue is comparable between the two dispatchers. One possible explanation for this is again the

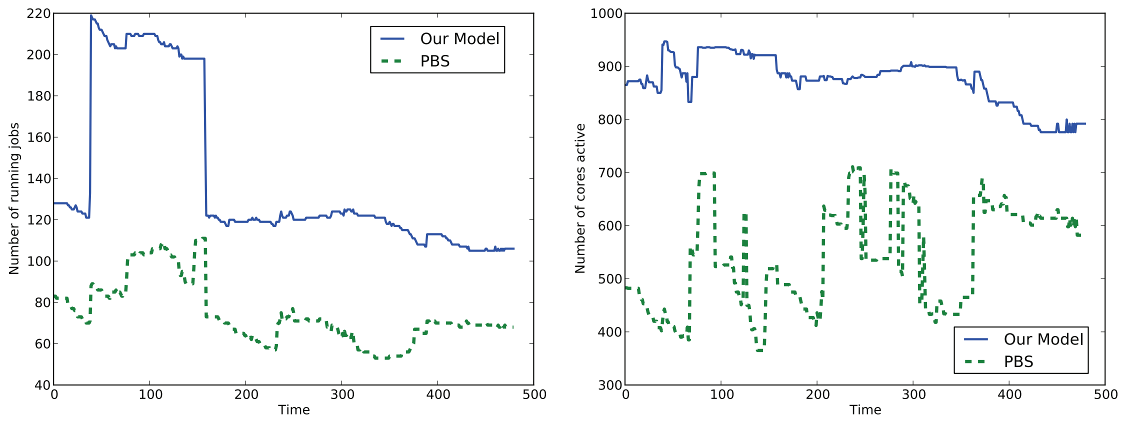


(A) Jobs in Queue

(B) Times in Queue

FIGURE 3.4: Waiting jobs and queue time for BATCH2

limit imposed by the GPUs availability, given that not all the cores are occupied, which forces more jobs to wait when a certain threshold for the number of GPUs required is reached.



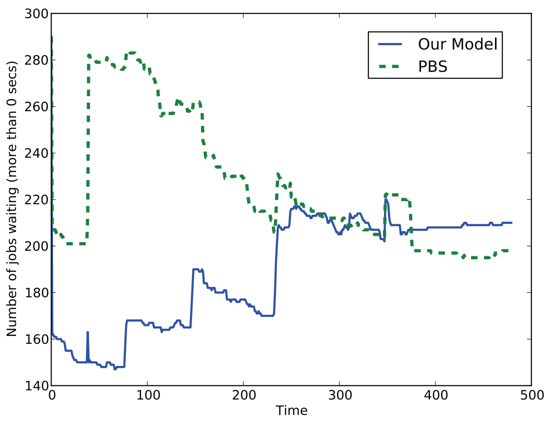
(A) Running Jobs

(B) Active cores

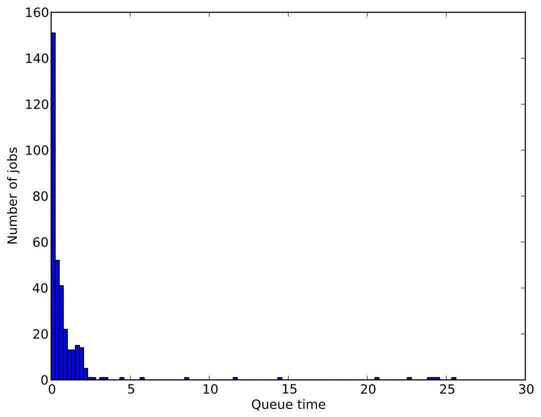
FIGURE 3.5: EURORA utilization on the third trace (BATCH3)

3.5 Conclusions

In this work have presented a CP based proactive workload dispatcher for the HPC EURORA supercomputer and compared its performance with those of the system currently in use (PBS). Our goal is to manage the computational resources on the platform so as to achieve a twofold result: increase the machine utilization and then reduce the job waiting times. A higher machine utilization translates into a lower consumption from idle resources and a large number of accepted jobs, with benefits for the supercomputer owner



(A) Jobs in Queue



(B) Times in Queue

FIGURE 3.6: Waiting jobs and queue time for BATCH3

and on the environmental side. Short waiting times correspond to a higher quality of service for the system users.

The problem we tackled was not an easy one, owing to the need to manage multiple objectives and to the limited availability of multiple, heterogeneous, resources. In both the considered metrics (machine utilization and waiting times) we considerably outperformed the current scheduler, showing that there are great margins for improvement when a proactive approach is used. The current, fundamentally reactive approach currently in use proved to have particular difficulties with the simultaneous management of different classes of resources (e.g. cores and GPUs). As a future long-term goal, we plan to further develop our model to replace (or at least complement) the scheduler currently in use on EURORA, with focus on improving its energetic behavior. To achieve this result, we will need to research and develop techniques to allow our approach to operate quickly enough to match the frequency of job arrivals. Moreover, we will need to make some adjustments to take into account the complex policies which regulate exactly the services provided by the supercomputer to its users.

Chapter 4

CP in HPC scheduling: a first application and evaluation on the EURORA HPC

High-performance computing centers are investment-intensive facilities with short depreciation cycles. An average supercomputer reaches full depreciation in three to five years [33]. Hence their utilization has to be aggressively managed to produce an acceptable return on investment. Even relatively small improvements in utilization, throughput, and quality of service translate in significant financial gains. A key role in this challenge is played by scheduling software that decides where and when a job has to execute. Users submit jobs to supercomputing machines specifying the amount of required resources (CPUs, GPUs, memory) and the maximum expected execution time (wall-time). In general, different "job queues" are available in HPC machines managing, for example, jobs featuring different priorities, execution time and user-requirements.

Commercial scheduling software (like PBS Professional [127], Torque [46], and Slurm [47]) can be configured via a set of rules managing the priorities of waiting jobs. These priority-rule-based algorithms are simple and reasonably fast, but the resource allocation and schedules found can be considerably improved in terms of job waiting time and QoS.

On the other hand, search-based approaches are much slower then priority based algorithms, but can obtain significantly better solutions. Constraint Programming (CP) and Integer Linear Programming (ILP) are two well known paradigms to solve NP-hard problems by efficiently exploring the solution space for optimizing one or more objective functions. These techniques, however, have seldom been used in HPC facilities as they are computational expensive and thus incompatible with the intrinsic on-line nature of HPC job schedulers.

In this work, we contradict this claim as we notice that HPC jobs exhibit a longer duration and lower arrival rate than that of e.g. enterprise servers and data-centers workloads. This opens significant opportunities for optimization-based scheduling.

We propose a complete and efficient CP approach for HPC machines that computes optimal schedules that minimize the job time-in-queue, keeping in mind the concept of fairness. Fairness is accounted by considering the expected average waiting time in queues declared by the supercomputing center. For this reason we designed an objective function that minimizes the job time-in-queue weighted on the expected average waiting time.

In parallel, we evaluate the impact of this optimization goal on other performance metrics such as late jobs, user QoS and scheduling overhead. The model extends the one in the work of Bartolini et al. [1], to account for multiple classes of jobs and their temporal dependencies. In addition, the solution space exploration strategies have been optimized for on-line use, taking into account the impact of the schedule computation time on machine utilization.

The CP solver has been embedded, as a plug-in, in the software framework of PBS Professional [127], a well-known commercial HPC scheduler, by replacing its rule-based scheduling engine. By linking our solver with a state-of-the-art HPC scheduling tool, we have been able to validate our approach on a real-life HPC machine, Eurora, from "Consorzio INtEruniversitario per il Calcolo Automatico" (CINECA). Eurora is a fully operational prototype of direct-liquid cooled HPC machine for future Tier 0 and energy aware HPC. It achieved the top GREEN500 ranking in June 2013 and has been used for production runs since 2013.

Experiments on Eurora over several weeks of operation under production workloads show that the new scheduler achieves significant improvements in job waiting time with respect to PBS Professional, while at the same time maintaining high machine utilization. In addition, an experimental evaluation on a wide range of synthetic workloads shows that the approach is flexible, robust and well-suited for integration in a portfolio of scheduling strategies to cover different levels of machine utilizations.

Simulated tests on Eurora-sized instances obtain average improvements of 21% on the waiting time of jobs and 22% on late jobs, although we introduce an overhead for computing higher quality solutions with respect to PBS that is 20 times higher. However, the overhead has a negligible impact on the job execution time: in our tests the worst case maximum-overhead over average-walltime ratio registered is only 5.26%. Experiments in a real production environment achieved an average improvement on job waiting times of 29% while maintaining the same average machine utilization.

While being suitable for real workloads, the CP-based approach suffers from scalability issues limiting its use in substantially larger workloads. For this reason, we have identified alternative approaches for an algorithm portfolio and conditions for their automatic selection.

The chapter is organized as follows: we start formally defining the HPC scheduling problem considered in this work in section 4.1. Section 4.2 provides some insights on Constraint Programming, the declarative programming paradigm used to model and solve the problem. Section 4.3 describes the optimization model and all the features implemented to make it usable on a real HPC center. Section 4.4 gives an overview of PBS Professional and the embedding of our scheduler in its framework. Overhead reduction techniques are also discussed here. In section 4.5 we show results on synthetic and real settings and we make statistics on the computational overhead.

4.1 The HPC Scheduling problem

Allocating and scheduling jobs on HPC machines can be defined as follows.

We consider a set of jobs $J = \{j_1, \ldots, j_n\}$. Each job is characterized by its maximal expected duration d_i (referred to as wall-time) and the number of jobs units u_i which is equivalent the number of virtual nodes required. Each job unit starts and ends with the job, and requires a certain amount of resources.

Every job $j_i \in J$ is submitted to a specific queue $q_h \in Q$ where $Q = \{q_1, \ldots, q_m\}$, to obtain the queue q_h in which the job j_i is submitted we can use the fucntion $queue(j_i)$. The job i enters in queue at time stq_i. Each queue is characterized by its expected waiting time ewt_h, which provides a rough indication of the queue priority. Waiting times larger than the ewt_h do not result in penalties for the computing center manager, but they may be an indication of poor QoS.

HPC machines are organized in a set of nodes $Nodes = \{node_1, \ldots, node_{Nn}\}$ and a set of resources $Res = \{res_1, \ldots, res_{Nr}\}$, like for example cores, memory, GPUs and MICs. Each node $node_j \in Nodes$ of the system has a capacity cap_{jr} for each resource $r \in Res$. Note that in case a resource is not present on a node its capacity is zero.

Each job unit k of job i requires an amount of resource req_{ikr} for each $r \in Res$.

The HPC allocation and scheduling problem accounts for finding for each job i a start time s_i, and for each job unit k of job i the node n_j where it has to be executed. Resources on all nodes cannot be exceeded at any point in time.

There are a number of other features required for an in-production HPC machine that the scheduler has to support

- Arrays of jobs: a user can submit a set of independent jobs with the same characteristics (resources, wall-time, queue of submission, etc…).
- Heterogeneous jobs: these jobs are synchronized (i.e., they start at the same time) but can ask multiple heterogeneous nodes (for example a job can ask one node with GPUs and another node without).
- Reservations: a reservation locks a set of resources for a given time window. Each reservation has an associated queue where jobs are submitted. Note that these jobs implicitly have a deadline. Jobs that do not fit the reservation are simply not scheduled.
- Standing reservations: standing reservations are periodic repetition of the same reservation.
- Stopped queues: a queue can be stopped at a certain point in time, meaning that every job in that queue cannot start until the queue is restarted.

- Prime-time and non-prime-time jobs: a job is a prime-time (resp. non-prime-time) job (and is submitted to a prime-time, resp. non-prime-time queue) if it should execute in a specific interval of time. If a job is neither prime-time nor non-prime-time it is an "anytime" job.

4.2 Constraint Programming

Constraint Programming is a declarative programming paradigm [130] particularly suitable for solving constraint satisfaction and optimization problems. A constraint program is defined on a set of decision variables, each ranging on a discrete domain of values that the variable can assume, and a set of constraints limiting the combination of variable-value assignments. For example, decision variable x ranging on the domain $[1..10]$, written as $x :: [1..10]$, means that variable x can be assigned to one (integer) value between 1 and 10.

After the creation of the model, the solver interleaves two main steps:

1. Constraint propagation: constraints are propagated by removing provably inconsistent values from variables domains. The constraint $x > y$ where both x and y range on $[1..10]$ removes value 1 for x and 10 for y.
2. Search: the search strategy explores alternative assignments of variable-values until either a solution is found or a failure is detected.

In case of optimization problems, when a solution is found its optimality is not guaranteed. Therefore the solver searches for better solutions if they exist, otherwise it proves optimality.

Constraint Programming is particularly suited for solving scheduling problems providing decision variables that correspond to activities. Each activity variable a is characterized by three features: $s(a)$ representing its start time, $d(a)$ its duration and $x(a)$ representing its execution state: if $x(a) = 0$ the activity is not considered in the model.

For scheduling problems, a number of global constraints have been developed the most important being the cumulative constraints for managing resource usage. $cumulative([a],[r],L)$: the constraint holds if and only if all the activities in $[a]$ whose resource requirement is in $[r]$ never exceed the resource capacity L at any point in time. A number of propagation algorithms are embedded in the cumulative constraints for removing provably inconsistent assignments of activity start time variables.

The algorithm adopted by the solver used in this work is the "Self-Adapting Large Neighborhood Search". The complexity of this algorithm is exponential within the number of decisional variables. In our case the number of decisional variables is $n + Nn * \sum_{i=1}^{n} u_i$. Note that this algorithm can be considered as an anytime algorithm providing the best solution obtained in a given amount of time. Clearly if the time is enough then the solver can find the optimal solution and prove the optimality. Much information on CP and how to translate a model into a program can be found in literature [131, 132]. Also information on the "Large Neighborhood Search" algorithm can be found in literature [133, 134, 129].

4.2.1 Motivational example

Rule-based scheduling algorithm reach the optimal solution only in a few cases. This is the reason why we chose to use CP because, e.g. let us suppose we have a system with four nodes and the resources specified in table 4.1.

Node id	Cores	GPUs	MICs	Memory
node1	16	2	0	16GB
node2	16	2	0	16GB
node3	16	0	2	16GB
node4	16	0	2	16GB

TABLE 4.1: Node test setup

Suppose we have two different setups for the scheduling priority

1. PBS by GPUs: with this setup jobs are ordered by decreasing number of requested GPUs.
2. PBS by Walltime: in this setup jobs are ordered by increasing expected execution time (as declared by the user).

Suppose we have also an optimization model that optimizes the total time in queue of jobs. The first example has four jobs with duration and resource requirements as specified in table 4.2.

Id	Dur.	Nodes	Cores	GPUs	MICs	Mem.
Job1	600	2	32	0	4	2GB
Job2	60	1	1	2	0	2GB
Job3	720	2	32	4	0	2GB
Job4	600	2	32	2	0	2GB

TABLE 4.2: Jobs set for test 1

Job1 and Job2 are submitted at time 0, Job3 and Job4 are submitted at time 5. In table 4.3 the solution obtained by PBS ordered by GPU is described. In table 4.4 the solution obtained by both PBS ordered by walltime and by the optimization model is presented.

Id	Start	node1	node2	node3	node4
Job1	0	0	0	1	1
Job2	0	1	0	0	0
Job3	60	1	1	0	0
Job4	780	1	1	0	0

TABLE 4.3: PBS by GPUs solution to test 1

Id	Start	node1	node2	node3	node4
Job1	0	0	0	1	1
Job2	0	1	0	0	0
Job3	660	1	1	0	0
Job4	60	1	1	0	0

TABLE 4.4: PBS by Walltime and optimization model solution to test 1

As we can see in table 4.5, in this test PBS by walltime and the optimization model reach the optimal solution, PBS by GPUs instead obtains a worse solution.

	PBS GPU	PBS Wallt.	Model
Total execution	1380	1320	1320
$\sum$ waiting	840	720	720

TABLE 4.5: Test 1 statistics

In the second test, we maintain the system behavior (nodes, PBS setups and submission times) but we use a different set of jobs (Table 4.6).

Id	Dur.	Nodes	Cores	GPUs	MICs	Mem.
Job1	70	2	32	0	4	2GB
Job2	60	1	1	2	0	2GB
Job3	480	2	32	0	0	2GB
Job4	600	2	32	2	0	2GB

TABLE 4.6: Jobs set for test 2

In this case the optimization model obtains the same solution of PBS ordered by GPUs (Table 4.7) and PBS by walltime obtains the solution described table 4.8.

Id	Start	node1	node2	node3	node4
Job1	0	0	0	1	1
Job2	0	1	0	0	0
Job3	70	1	1	0	0
Job4	60	1	1	0	0

TABLE 4.7: PBS by GPUs and optimization model solution to test 2

In table 4.9 we can observe that this time the optimal solution is obtained only by PBS by GPUs and the optimization model.

As we can see from these two very simple examples, rules-based scheduling can obtain the optimal solution only when the rule used by the administrator perfectly fits the problem instance. This is observable even with trivial problems like the one we showed. If we increase the size of the problem, it is

Id	Start	node1	node2	node3	node4
Job1	0	0	0	1	1
Job2	0	1	0	0	0
Job3	60	1	1	0	0
Job4	540	1	1	0	0

TABLE 4.8: PBS by Walltime solution to test 2

	PBS GPU	PBS Wallt.	Model
Total execution	660	1140	660
$\sum$ waiting	130	600	130

TABLE 4.9: Test 2 statistics

unlikely that we find a rule that gets the optimal solution. This is why it is so important to use complete optimization in this kind of problem.

4.3 CP Model

The problem considered is an on-line allocation and scheduling problem which is triggered by specific events: job submission, termination, modification of wall-time and job queue change. At any activation at time t, we have to consider two sets of jobs: (1) A is the set of jobs waiting on a queue and (2) B is the set of running jobs at current time t. The starting time of running job j_i can be obtained through the function $getStart(j_i)$. Running jobs cannot be migrated and therefore they should be considered as fixed. The resources they use are allocated and reserved for them. The decisions we have to take are on the waiting jobs in queues.

4.3.1 General model

We now present the CP model built at each activation of the scheduler at time t.

We model every job j_i as an activity variable a_i with start time $s(a_i)$ duration $d(a_i) = d_i$ and $x(a_i) = 1$.

The start time of each job $s(a_i)$ is a decision variable whose domain is $[t, Eoh]$ where t is the current time and Eoh is the end of the time horizon of the scheduler. Eoh can be computed in a conservative way as $\min_i(s(a_i)) + \sum_i d(a_i)\ \forall i \in A \bigcup B$ (we consider both the set of waiting jobs A and the set of running jobs B).

To model the allocation of job units to nodes, we create an activity variable a_{ikj} for each unit k of job i and for each possible assignment of node j. The start time and the duration of these activities are constrained to be equal to the start time and duration of the job i: $s(a_{ikj}) = s(a_i)$ and $d(a_{ikj}) = d(a_i)$. On activation variables $x(a_{ikj})$ we impose a constraint that forces only one allocation to be feasible, namely

$\sum_{j=1}^{N_n} x(a_{ikj}) = 1 \ \forall i, k$

On top of these decision variables we built a model described in equations 4.1. The first set of unary constraints limit the possible starts of waiting jobs to be greater than t. The second set of constraints assign the start time of running jobs to the real (already decided) start time. The third set of unary constraints limits allocation variables to be 1 if the job unit is assigned to node j, 0 otherwise. The fourth set of constraints limits a job unit to be assigned to only one node. Finally we have a cumulative constraint for each resource type for each node and limit the resource usage to stay below resource capacity at any point in time.

$$\begin{aligned}
&s(a_i) :: [t..Eoh] \ \forall i \in A \\
&s(a_i) = getStart(j_i) \ \forall i \in B \\
&x(a_{ikj}) :: [0,1] \ \forall i \in A \\
&\sum_{j=1}^{N_n} x(a_{ikj}) = 1 \ \forall k, \forall i \in A \\
&cumulative(a_{ikj}, req_{ikr}, cap_{jr}) \ \forall j \in N_n \forall r \in R
\end{aligned} \tag{4.1}$$

Note that the quantifiers on the right-hand side define how many replicas of the constraints appear in the model. For the indexes of the constraint variables not appearing among the quantifiers, we assume that they take all the available values. This is just a compact notation to identify sub-vectors (or sub-matrices) within data structures having a lot of indexes.

As far as the objective function is concerned, we consider the minimization of job waiting-times weighted by the expected waiting time of the queue where the job is submitted ewt_h. As often queues represent job priorities, the waiting coefficients are proportional to these priorities.

$$\min z = \sum_{i=1}^{n} \frac{s(a_i) - stq_i}{ewt_{queue(j_i)}} \tag{4.2}$$

This basic model should be enriched with a number of features needed to run the scheduler on a real HPC machine, as explained in section 4.1.

Array of jobs and heterogeneous jobs As far as array of jobs and heterogeneous jobs are concerned, they are very easily handled by the CP model: in the first case jobs simply share the same resource requirements, while in the second case they share the same starting time.

Reservations and Standing Reservations When a reservation is submitted, it is associated to a set of resources and at a specific time window. Therefore, at modeling level, we consider reservations as specific jobs, called reservation jobs, using reservation resources for the time window associated to the reservation. Standing reservations can be modeled as arrays of reservation jobs.

Stopped Queue When a queue h is stopped, all jobs waiting on it cannot be scheduled. Therefore their execution state variable should be zero.

$$x(a_i) = 0 \; \forall i \in q_h \tag{4.3}$$

Prime-time and non prime-time jobs Another important feature is the prime-time and non-prime-time jobs handling. This feature is easily handled by constraint programming models as we simply remove from the domain of start time variables of prime-time jobs forbidden (non-prime-time) intervals. Conversely we act for non-prime-time jobs.

4.3.2 Allocation of jobs within a reservation

In the above model, we have considered reservations as jobs using resources required by the reservation for the time span of the reservation itself. However, on real machines reservations can be seen as private queues where only eligible users can submit jobs. The scheduling and dispatching of jobs in the reservation queue have to be treated as a separate problem handled by a separate model (Equations 4.4). The motivation is that the execution time for a job submitted to a reservation queue is the time span of the reservation and the resources available for the job are limited to the reservation resources. In addition, jobs submitted to the reservation queue have a deadline. Each reservation has a fixed start time, a fixed duration and for each node a set of reserved resources. These data can be extracted by proper functions, namely $getStart(resv)$, $getEnd(resv)$ and $getResource(resv, j, r)$ where j is the node and r the resource type.

The resulting model considers only jobs in the reservation queue JR that, as before, are divided into waiting jobs A_{JR} and running jobs B_{JR}.

$$\begin{aligned}
& s(a_i) :: [max(t, getStart(resv))..getEnd(resv) - d(a_i)] \\
& \qquad \forall i \in A_{JR} \\
& s(a_i) = getStart(j_i) \; \forall i \in B_{JR} \\
& x(a_i) :: [0,1] \; \forall i \in A_{JR} \\
& x(a_{ikj}) :: [0,1] \; \forall i \in A_{JR} \\
& \sum_{j=1}^{N_n} x(a_{ikj}) = x(a_i) \; \forall k, \forall i \in A_{JR} \\
& cumulative(a_{ikj}, req_{ikr}, getResource(resv, j, r)) \\
& \qquad \forall j \in N_n \forall r \in R
\end{aligned} \tag{4.4}$$

The first set of unary constraints defines the domain of the start time of activity variables that are waiting on the reservation queue. This domain is lower bounded by the maximum between the current time and the start of the reservation, and it is upper bounded by the end of the reservation minus the job duration. The second sets of constraints simply fixes already started activities. Differently from the previous model, jobs waiting on the reservation queue, and consequently all their units, can be either executed or not. The cumulative constraint in this case is limited to jobs belonging to the reservation queues and to resources of the reservation.

4.3.3 Feasibility check

One of the most important component of real HPC schedulers is the feasibility check. Intuitively the scheduling problem instance cannot be infeasible. Otherwise, the whole machine would stop. The infeasibility could be due to errors both in job and reservation submissions. A small example of wrong job submission occurs when (1) we have two resources: one node with 2 GPUs and another node with 2 MICs, and (2) we have a job submission with one unit requiring one GPU and one MIC. In this case the instance is simply infeasible as such a resource (i.e., one node with both a GPU and a MIC) is not available on the machine.

An example of wrong reservation submission instead is due to lack of needed resources. We recall that a reservation is submitted with a fixed starting time, a fixed duration and a number of required resources. If these resources are not available for the time required the reservation is simply infeasible.

For both these problems we have a phase of the feasibility check (Figure 4.1). The first is the reservation feasibility check that checks if there is enough room for executing the reservation. Then we have a feasibility check for each job separately, ensuring that the job requires resources that are available in the machine. This is made by solving the model 4.5.

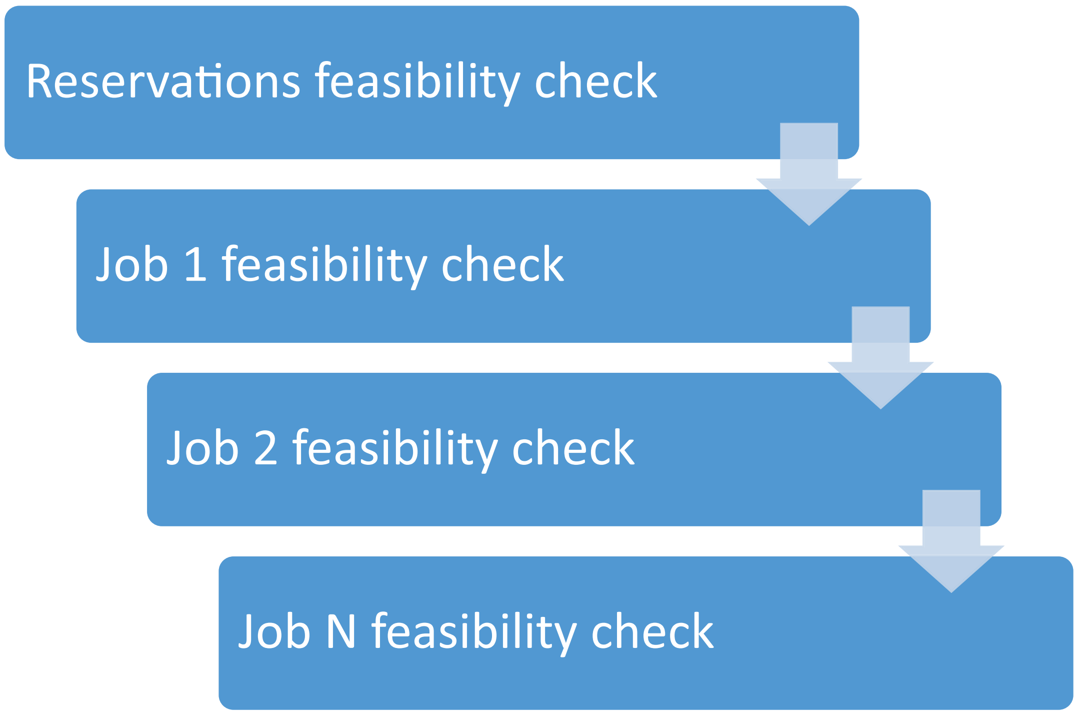


FIGURE 4.1: Feasibility check subdivision

$$\begin{aligned}
&job_i = s(i)\ \forall i \in A \\
&job_i = \perp\ \forall i \in B \\
&job_i = s(i)\ \forall i \in S \\
&job_i = s(i)\ or\ \perp\ \forall i \in F \\
&alternative(job_i, UN_{ijk}, u_i)\ \forall i \in A \bigcup B \bigcup S \bigcup F \\
&cumulative(UN_{ijk}, d_i^{P_{ij}}, r_{ijkl}^{P_{ij}}, rl_{jl})\ \forall k = 1..M, l \in R
\end{aligned} \tag{4.5}$$

Where S is the set of started reservations and F is the set of reservations with start time in the future. The second part is the job feasibility check: it checks if the m-th job can execute in the system, resulting n model 4.6.

$$\begin{aligned}
&job_m \geq t\ or\ \perp \\
&alternative(job_m, UN_{mjk}, u_m) \\
&cumulative(UN_{1jk}, d_m^{P_{1j}}, r_{1jkl}^{P_{1j}}, rl_{jl})\ \forall k = 1..M, l \in R
\end{aligned} \tag{4.6}$$

4.4 Framework architecture

Our scheduler has been embedded in the framework of PBS Professional. PBS Professional is composed by the following macro-components and services:

- *PBS_server* is a server that handles all the events and stores all the jobs, queues and settings, logs and information.
- *PBS_mom(s)* is a process running on each node of the HPC machine managing its resources.
- *PBS_scheduler* implements the scheduling algorithm of PBS Professional.
- PBS binaries (i.e. *qsub*, *qmove*, *qstat*, *PBS_rsub*, etc. . .) provide the interface between the users and the PBS internal components (i.e. *PBS_-server*, *PBS_mom*).
- *Hooks*, PBS gives the possibility to handle events with hooks. Hooks are scripts triggered by events. They can be used to get notifications of a new job submission, of a reservation submission, etc...

The original scheduler *PBS_sched* can be disabled and replaced with an ad-hoc scheduling algorithm. We take advantage of this functionality to embed our scheduler in PBS in a plug-and-play fashion (Figure 4.2). In this way, we leverage all the functionality of the PBS infrastructure such as tracking the system status and implementing scheduling decisions.

The framework receives events from the *PBS_server* using Hooks. The framework interacts with the *PBS_mom* component by asking the node state

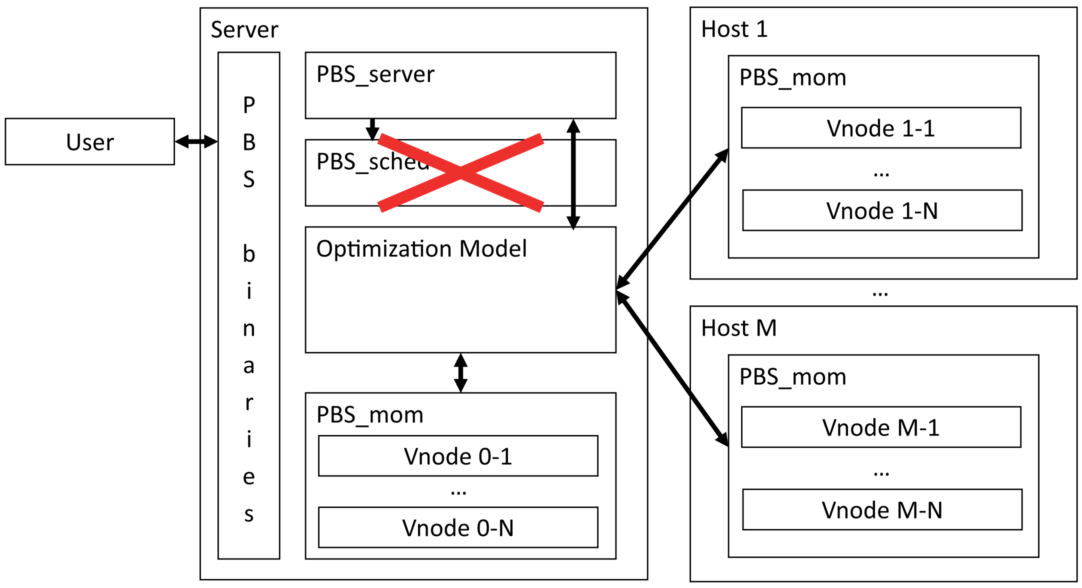


FIGURE 4.2: Framework macro architecture

through PBS binaries. Then our scheduler is run and its decisions are sent to the *PBS_mom* component.

Figure 4.3 show the workflow of our framework. All the self-generated events, "Job update" hook, and "New reservation" hook start a scheduling cycle. Differently, the "New job" and "Job terminated" hooks will trigger scheduling cycle only if the state of the system has been changed (i.e. awaken nodes, job deleted or new job submitted) since the last scheduling cycle. This avoids unnecessary overheads. Each scheduling cycle starts by (1) checking if a job currently in execution exceeds its walltime request (Overrun check).

If a running job exceeds this duration we flag the resources, in which the job is executing, as "used" to avoid other jobs to execute on the same resources. We let *PBS_server* take the corrective action (i.e. killing the job).

Subsequently (2) the scheduling cycle update an internal image of the nodes status which is used as input by the algorithm. In this phase the algorithm checks if the nodes have been crashed/switched-off/activated.

This information is used to verify which reservations and jobs can be executed (in figure as "Check reservations feasibility" and "Check jobs feasibility"). This check excludes either the jobs that does not satisfy the machine requirements (i.e. jobs asking more accelerator per node than the maximum available) and either the jobs which cannot execute due to the current system state (i.e. reduced number of active nodes).

All the jobs and reservations which passed the above mentioned feasibility check are eligible to execute. Before calling our CP model for these jobs, the scheduling cycle uses them to check if the machine starts from a feasible job allocation. If not, the algorithm waits until the machine states converge to a feasible starting point (i.e. it waits jobs to end in case of overutilization of some nodes).

If all the necessary conditions are satisfied the algorithm solves the model and checks the solution obtained does not generate overutilization.

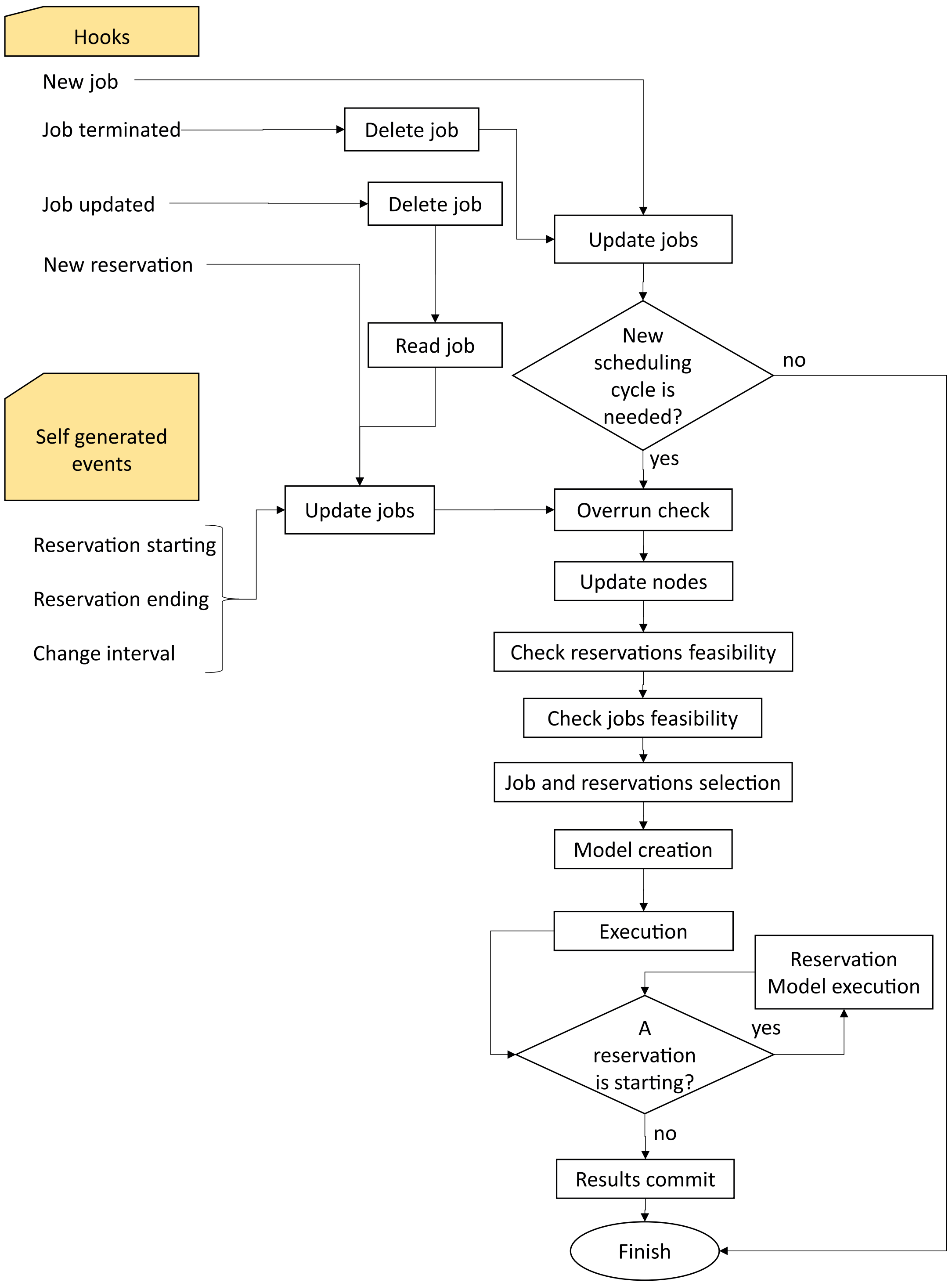


FIGURE 4.3: Workflow

Then for each starting reservation the algorithm solves a sub-scheduling problem (Section 4.3.2) considering only the jobs submitted in the reservation queue.

Finally, the model generates for each job the start time and the nodes in which execute, the job with starting time equal to the current time are then executed.

One of the key goals of our scheduler is reactivity. The framework has to use meta-heuristic algorithms to explore a large set of solutions of an NP-hard problem and to give a good solution in a reasonable amount of time. For this purpose we introduce several new overhead-reducing techniques. These techniques limit the execution time of the optimization model and, in case of too large instances (bigger than 1600 jobs and 65 nodes), the size of the instance too.

The first technique limits the CP model solution time to δ seconds where δ is computed as follow.

- Initially δ is set to K_1.
- The CP model is executed with δ as timeout.
- If the CP model does not generate any solution in the δ the model is re-executed up to M times with $\delta = K_2 * \delta$. If no solutions are found at the M-th iteration the scheduling cycle returns with no feasible schedule and waits for the next event to restart.
- If instead the CP model returns with one or more than one valid solution the algorithm uses the last found schedule (which minimizes the objective function). The algorithm sets δ equal to the time taken to find the first valid solution plus K_1. This ensures always to adapt the time-out δ to the CP model complexity.

In addition, to limit the maximum solution time we implemented several protection mechanisms. (1) We saturate δ at 300 seconds; (2) We stop after the first solution if $\delta > 60s$ or if the search of an improved solution takes more than 10s (3). All these values are empirically defined and guarantee an average overhead of ~ 6 seconds for the model solution in case of high instances (65 nodes and 1600 jobs). At the same time, this values allows us to find an almost optimal solution in large problems instances and gives enough time to solve medium problems.

Under these circumstances, a second technique limits the problem instance when in the previous scheduling cycle no solutions were found. Indeed if after step 2 no solution where found the number of queued job considered in the scheduling is halved keeping the first queued jobs and excluding the subsequent.

Three additional features have been supported: the Standing reservation, the Job array and heterogeneous jobs.

A standing reservation is a reservation repeated over time with user defined pattern. The user can specify, in addition to the default reservation information, the frequency (weekly, daily, hourly), the number of reservations to create and the days in which these reservations have to execute. The jobs array represent a chain of jobs. We implemented it as a sequence of different reservations with no binding constraints. Finally, heterogeneous jobs are jobs with different resource requirement for different jobs unit. For this job it is necessary to introduce a synchronization constraint (Equation 4.7) to execute every job unit simultaneously.

$$synchronize(job_i, sjob_{it}) \ \forall i = 1..N, t = 1..T \tag{4.7}$$

4.5 Experimental Results

We have evaluated the performance of our scheduler in two distinct experimental setups, namely (1) in a simulated environment; and (2) on the actual Eurora HPC machine.

The simulation-based tests are designed to compare the behavior of our scheduling system (referred to as CP) and PBS Professional in a controlled environment, where we can submit the same sequence of jobs to each scheduler and compare their performance in a fair fashion. Testing our system on Eurora instead enables the assessment of its effectiveness in a fully operational production environment. Therefore, our experimentation consists of:

- A direct comparison of the CP scheduler and two different PBS setups. These experiments are executed on a set of Virtual Machines (VM). Every VM runs a script that generates in a predictable fashion a sequence of jobs (each composed of a single *sleep* command).
- A statistical evaluation on the Eurora HPC with true jobs submitted by real users over five weeks[*].

The PBS software can be configured in different modes to suit the purpose of the system administrator. The following experiments consider two different PBS setups:

1. The CINECA PBS configuration (referred to as PBSFifo): this setup uses a FIFO job ordering, no preemption, and backfilling limited to the first 10 jobs in the queue.
2. A PBS configuration (referred to as PBSWalltime) designed to get the best trade-off between waiting time and computational overhead: this setup employs a strict job ordering (by increasing wall-time), no preemption and backfilling limited to the first 400 jobs. Ordering jobs by wall-time and using a high backfilling depth allows to reduce the job waiting times but incurs a larger overhead: this is mitigated by introducing the strict job ordering.

The quality of the schedules was measured according to a number of metrics. Specifically, we have defined:

Metrics on job waiting times:

[*]The time needed for the scheduling team in this computing center to evaluate a scheduling policy is of 1 week.

- *Average time in queue (AQ)*: total waiting time divided by the number of jobs.
- *Weighted queue time (WQT)*: sum of job waiting-times, each divided (for fairness) by the maximum wait-time of the job queue.

Metrics on tardiness:

- *Number of late jobs (NL)*: the number of jobs exceeding the maximum wait-time of their queue.
- *Tardiness (TR)*: sum of job delays, where the delay of a job is the amount of time by which the maximum wait-time of its queue is exceeded.
- *Weighted tardiness (WT)*: sum of job delays, each divided (for fairness) by the maximum wait-time of the job queue.

Metrics on computational overhead:

- *Average overhead (AO)*: average computation time of the scheduler.
- *Maximum overhead (MO)*: maximum computation time of the scheduler.
- *Overhead percentage on test time (%O)*: percentage of time spent in computation during the entire test.

4.5.1 Evaluation setup

Simulation-based tests

We have designed the simulation so as to evaluate the performance of our CP scheduler w.r.t. PBS. The experiments differ under a wide range of conditions with respect to number of jobs, job units, resources heterogeneity and platform nodes. The goal is to assess the scalability of both approaches and their ability to deal with workloads having different resource requirements and processing times.

Overall, the evaluation tends to be biased toward pessimistic configurations, in part because of the limited computational power of the Virtual Machines. The typical workload for the Eurora supercomputer turned up to be somewhere in the mid-range of hardness considered in the simulated tests, and definitely manageable by our approach. Clearly all the real Eurora traces have been considered in the simulated tests, but we have also scaled them down and up to cover a wide range of working conditions for the scheduler.

In these experiments, different HPC environments were built on top of virtual machines. We used a single VM for each environment and exploited virtual nodes (supported by the PBS framework) to simulate the supercomputer units.

We have performed tests on small environments with 4 nodes as well as on a Eurora-scale environments with 65 nodes. In each experiment, the same sequence of jobs is generated and submitted to each scheduling system.

Each VM was allowed to employ up to two cores and 5GB of RAM, on a physical machine with two CPUs with six-cores and hyper-threading, and 96GB of RAM. The two-cores limit was due to the chosen virtualization environment (Oracle VirtualBox). PBS logs are the source of all information about the performance of the compared approaches.

Evaluation on the HPC

The second set of experiments is run on the Eurora HPC system. Eurora [100] is a heterogeneous HPC machine of CINECA. It is a fully operational prototype for future green HPC. Eurora is composed by 65 nodes, one login node with 12 cores, 32 nodes with 16 cores at 3.1GHz and 2 GPU Kepler K20 each and 32 nodes with 16 cores at 2.1GHz and 2 Intel Xeon phi (MIC) each.

Users of this HPC machine submit jobs specifying the amount of resources, nodes and wall-time to a queue. Each queue has a name, a priority and a list of nodes where its jobs can be executed, after the submission. The scheduling and dispatching software currently used in Eurora is PBS Professional 12 from Altair. Eurora users can choose among three main queues

- debug: for small jobs with low wall-time. CINECA declares the maximum waiting time of this queue of 1 hour.
- parallel: for large jobs with medium wall-time. CINECA declares the maximum waiting time of this queue of 5 hours.
- np_longpar: for large jobs with high wall-time. This is a non-prime-time queue. This means that jobs from this queue can execute only in a non-prime-time interval (from 18:00 to 08:00). CINECA declares the maximum waiting time of this queue of 24 hours.

4.5.2 Test generation

We have designed a software component (see Figure 4.4) to generate and submit a repeatable sequence of dummy jobs (i.e. sleep commands). The generation process has been calibrated based on real data (12,000 jobs submitted to Eurora in December 2014). For calibrating the arrival rates, we relied instead on statistics collected over the whole year 2013 from the Fermi HPC machine [135] at CINECA; the Fermi was chosen in this case due to its longer history of utilization.

In detail, for each test we generate n jobs (where n is an input parameter) to be submitted over a 24 hours period of real-world time. A certain percentage of jobs is submitted during daytime (8 AM to 6 PM), and the rest is submitted during the night (6 PM to 8 AM). Job arrival times are uniformly spread within each interval. In all our experiments, 89% of the jobs arrive at daytime and 11% at nighttime. All numbers mentioned above have been extracted from the CINECA statistics on the Fermi HPC machine. The following statistics are extrapolated from the Eurora execution traces. A fixed ratio of the generated jobs is then assigned to each system queue. In particular, 27% of the jobs are submitted to the debug queue, 72% to parallel, and 1%

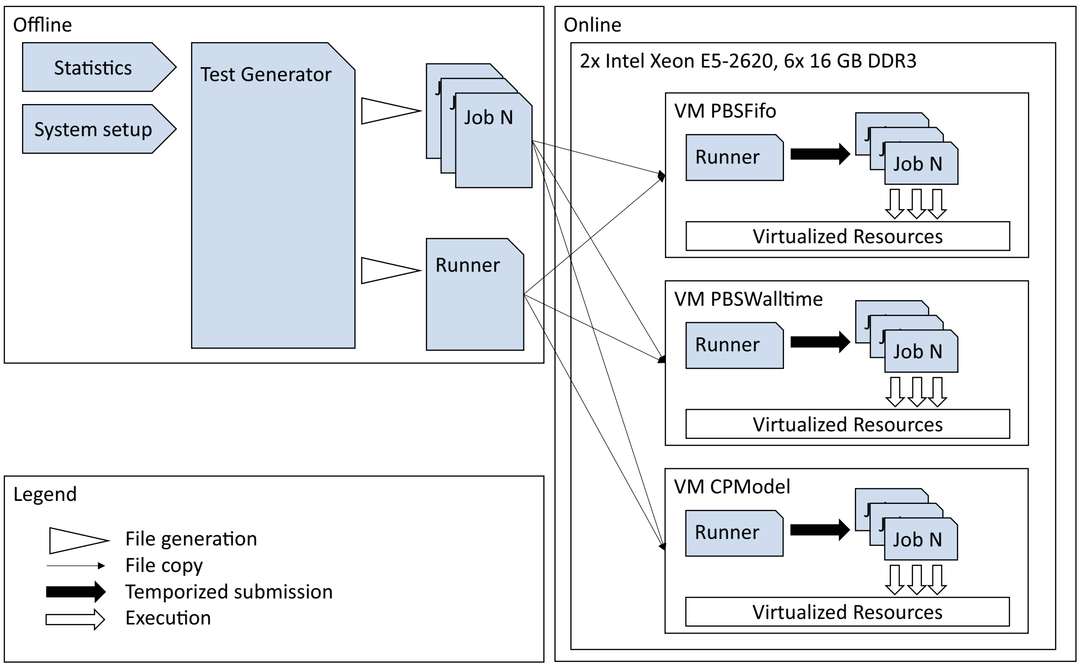


FIGURE 4.4: Test Generation

to np_longpar. The number of required nodes, cores, and the wall-time values are randomly generated for each job so as to match the Average Volume of Utilization (AVU) of its queue. In detail, we start by generating for each job i the number of requested nodes RN_i and the number of requested cores per node RC_i. In particular, let $nmin_q$ and $nmax_q$ be the minimum and maximum number of nodes for the queue q. Then, the node and core requests are obtained as:

$$RN_i = UD[1 \ldots mmax_q] \text{ and } RN_i = UD[1 \ldots 16]$$

where UD means a uniform distribution over the interval and $nmax_q = 2$ for the debug queue and $nmax_q = 32$ for parallel and np_longpar. Then for each job we compute a wall-time value W_i as:

$$W_i = \frac{AVU_q}{RN_i * RC_i} \tag{4.8}$$

where AVU_q is the Average Volume of Utilization for q. This value is obtained using the formula:

$$AVU_q = \overline{NR}_{i(q)} * \overline{CR}_{i(q)} * \overline{Wall}_{i(q)} \tag{4.9}$$

where $\overline{NR}_{i(q)}$ is the average number of nodes requested by jobs in q, $\overline{CR}_{i(q)}$ is the average number of requested cores (per node) and $\overline{Wall}_{i(q)}$ is the average wall-time of the jobs in the q. These statistics are obtained from Eurora data. Our AVU_q values are reported in Table 4.10.

The GPU, MIC, and memory requirements are generated so as to match

Queue	AVU
debug	6465
parallel	147145
np_longpar	111372

TABLE 4.10: Eurora jobs utilization

Queue	Amount of resource	% for GPUs	% for MICs
debug	0	96%	99%
	1	3%	0%
	2	1%	1%
parallel	0	31%	99%
	1	4%	1%
	2	65%	0%
np_longpar	0	19%	100%
	1	0%	0%
	2	81%	0%

TABLE 4.11: GPUs & MICs per node request distribution on Eurora

the average requirements observed on Eurora. In particular, jobs are partitioned into groups that are then assigned to a specific requirement value. The requirement values and the size of the partitions are reported in Table 4.11 for GPUs and MICs and in Table 4.12 for the memory.

Finally, the *execution* time of each job is generated via a two-step process. First, the jobs are partitioned in two sets according to a fixed proportion: the sizes are respectively 20% and 80% in our experiments, based on statistics from Fermi. Then, for the jobs in the first set the execution time is identical to the wall-time, while for the jobs in the second set the execution time is chosen uniformly at random between $0.20 * W_i$ and W_i (excluded).

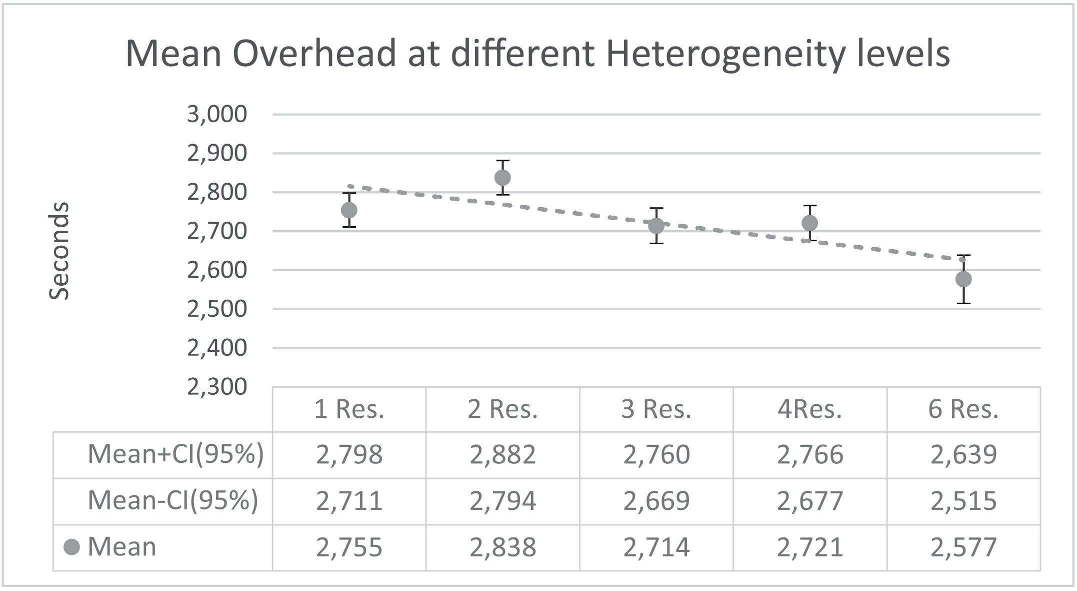
Test 0: Behavior at different heterogeneity levels

This test is designed to give an overview on how this scheduler would behave within different numbers of heterogeneous resources. More heterogeneity would increase the number of problem constraints, reduce the number of feasible job assignments, and as a consequence it would generally make the platform more complex to manage. Heuristic scheduling methods such as those employed by PBS would primarily be affected by this increase in complexity. In CP, however, adding more constraints leads to an interesting trade-off. On the one hand, the scheduling problem may indeed become more complex. On the other hand, however, CP has the ability to actively exploit the problem constraints to reduce the size of the search space. Hence, more constraints may actually improve the performance of a CP based approach.

In figure 4.5 we have reported an experiment that show how the results change, changing the number of resources (adding more heterogeneity). The

Queue	Mem in GB	% of jobs
debug	1GB	5%
	4GB	77%
	8GB	3%
	14GB	15%
parallel	1GB	22%
	4GB	17%
	8GB	55%
	14GB	6%
np_longpar	1GB	88%
	4GB	0%
	8GB	4%
	14GB	8%

TABLE 4.12: Memory per node request distribution on Eurora



	1 Res.	2 Res.	3 Res.	4Res.	6 Res.
Mean+CI(95%)	2,798	2,882	2,760	2,766	2,639
Mean-CI(95%)	2,711	2,794	2,669	2,677	2,515
● Mean	2,755	2,838	2,714	2,721	2,577

FIGURE 4.5: Mean overhead at different heterogeneity levels

test consider 4 nodes and 100 jobs, we can show the computational overhead for instance with 1, 2, 3, 4 and 6 different kind of heterogeneous resources. The test differs only by the resources requested by the jobs. However, being different jobs is not possible compare directly the result of metrics like ”time in queue“, etc... The only comparable metric is the overhead.

In summary: increasing the heterogeneity is likely to decrease the performance of PBS, but may have a beneficial effect on the performance of our approach as the reviewers can see looking at the trend of the overhead into the image.

Test 1: 4 nodes 99 jobs

First we tested a system with a low workload. The test simulates 4 Eurora nodes (2 with MICs and 2 with GPUs): the results are reported in Table 4.13.

	PBSFifo	PBSWalltime	CPModel
WQT	347.583	170.686	168.032
AQ	31011.7	35637.6	30025.7
NL	55	52	45
TR	1954690	2349450	1832660
WT	283.219	110.034	105.985
AO	0.22	1.21	3.27
MO	1	4	5
%O	0.1	0.3	0.4

TABLE 4.13: Simulated test with 4 nodes and 99 jobs

	PBSFifo	PBSWalltime	CPModel
WQT	152.94	137.74	119.77
AQ	10216.7	9465.42	8053.09
NL	65	60	46
TR	1298810	1223690	1003970
WT	60.03	55.97	46.40
AO	0.47	3.14	11.45
MO	3	10	19
%O	0.33	1.61	6.78

TABLE 4.14: Simulated test with 65 nodes and 330 jobs

PBSFifo has an advantage w.r.t. PBSWalltime and our model thanks to its lower overhead. However, both PBSWalltime and CP behave much better than PBSFifo w.r.t. all the performance metrics, in particular, those that take into account the priority of each queue (i.e. WQT and WT). The performance of our model is particularly good in this setting. In fact, we obtain substantial improvements w.r.t both PBSFifo and PBSWalltime on all the metrics on waitings and tardiness. This improvement is achieved at the cost of a larger overhead that, however, represents only the 0.4% of the makespan of the application.

Test 2: 65 nodes 330 jobs

Secondly we tested a system with a medium workload. In this test we simulate all the 65 Eurora nodes (32 with GPUs, 32 with MICs, and one log-in node): the results are in Table 4.14. Our model manages to considerably outperform PBSFifo and PBSWalltime in terms of all the metrics related to waiting time and delay. Also in this case, all the metrics on waitings and tardiness improve w.r.t. both PBSFifo and PBSWalltime even if the cost in terms of overhead grows to 6.78% which still justifies the gain obtained in all other metrics.

	PBSFifo	PBSWalltime	CPModel
WQT	1034.2	853.681	2441.3
AQ	32066.1	27046.2	34816.3
NL	234	200	376
TR	16798300	13693000	16774800
WT	776.405	623.287	2013.18
AO	1.02	15.47	34.82
MO	11	57	120
%O	0.69	8.8	18.95

TABLE 4.15: Simulated test with 65 nodes and 700 jobs

Test 3: 65 nodes 700 jobs

Finally we simulate a system with a high workload. We tested a 65 node configuration with a larger number of jobs (namely 700): the results are reported in Table 4.15.

Due to the large number of jobs and (more importantly) job units, in this case, our framework was forced to employ the overhead reduction techniques from Section 4.4. Such techniques are indeed effective in limiting the overhead, but they also have an adverse effect on the quality of the model solutions. As it can be seen in the table, our model yields a little improvement in tardiness w.r.t. PBSFifo, a small increase in the total time in queue, and a considerable increase of the number of late jobs, the WQT, and the weighted tardiness.

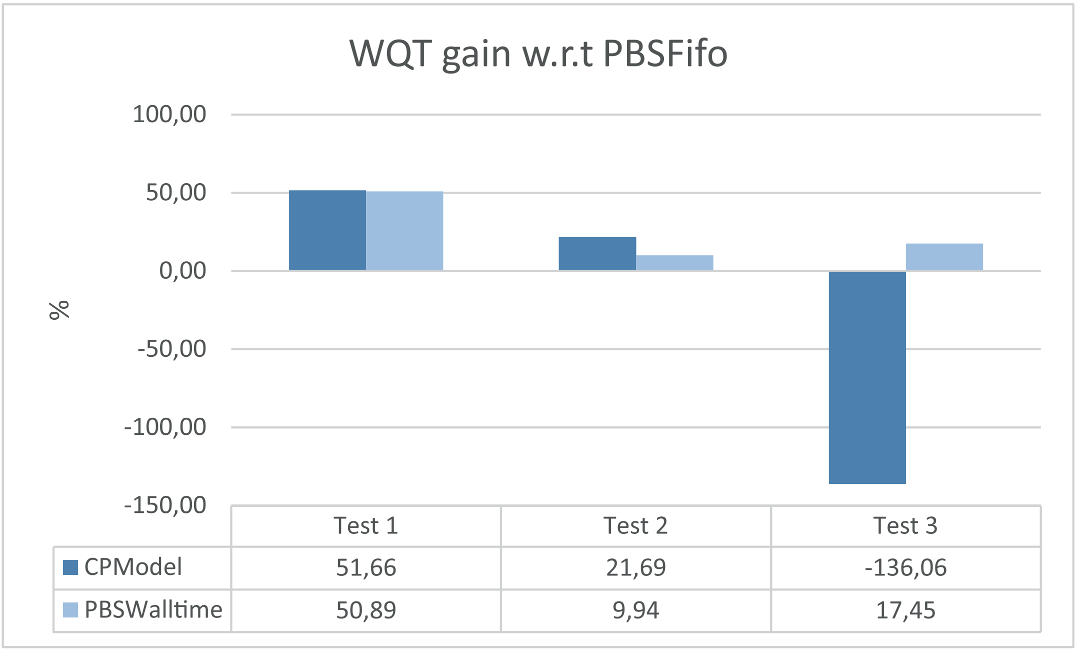
Introducing more effective overhead reduction techniques seems to be critical to improve the performance of our CP system. This is subject of current research activity.

In the following chapter we can have a better view of results, seeing how the behavior of the optimization model changes by increasing the hardness of the problem. Indeed the test seen are reported for increasing problem hardness.

Results comparison

We now provide a thorough comparison of the results obtained on the three tests. We will analyze each performance metric separately and investigate how the number of jobs and nodes affects the results. In each comparison, the performance of PBSFifo is used as a baseline and positive values denote improvements.

Figure 4.6 reports the relative improvement of CP and PBSWalltime over PBSFifo in terms of WQT. The two approaches behave similarly on Test 1 (i.e. the easiest one), both obtaining a $\sim 50\%$ improvement over PBSFifo. As the test becomes more complex, the performance of our model gets better, beating PBSFifo by a factor 22%, against the $\sim 10\%$ obtained by PBSWalltime. In Test 3 (the largest), the overhead reduction techniques are active and this leads to a degradation of our results while PBSWalltime improves over PBFifo by a factor $\sim 17\%$.



	Test 1	Test 2	Test 3
■ CPModel	51,66	21,69	-136,06
■ PBSWalltime	50,89	9,94	17,45

FIGURE 4.6: Weighted queue time gain w.r.t. PBSFifo

In terms of average queue time (see Figure 4.7), PBSWalltime tends to improve over PBSFifo as the test size increases. Our approach (due to the overhead reduction techniques) follows the opposite trend. This is behavior is similar to the one observed for the WQT metric.

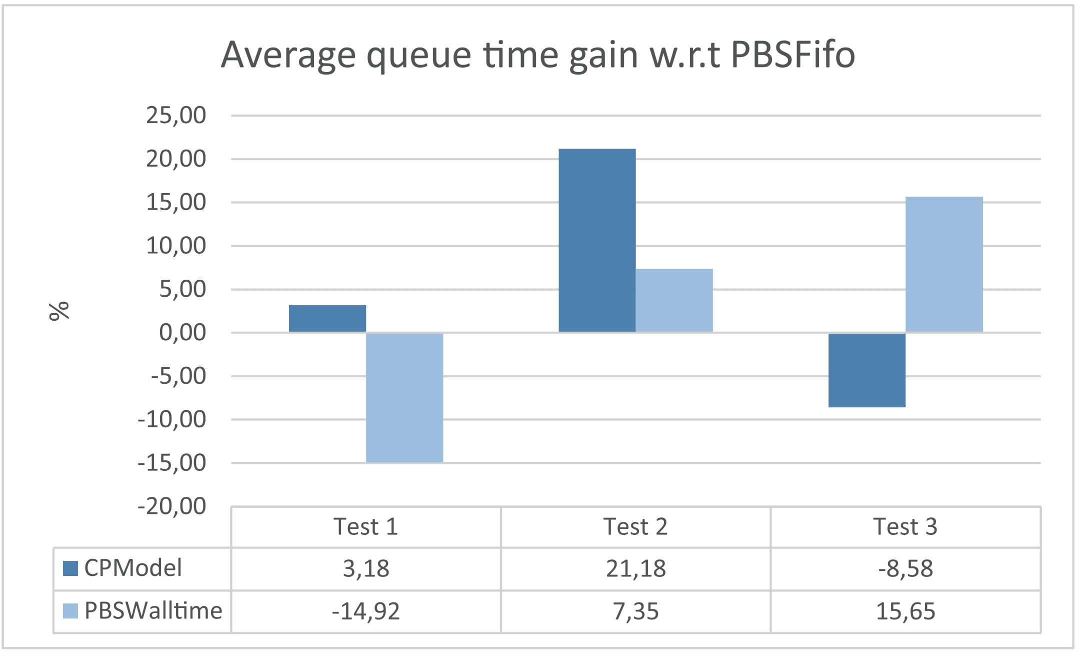
The results of the last three metrics (number of late jobs, tardiness and weighted tardiness (see Figures 4.8, 4.9, and4.10) follow the same trend as the WQT metric: CP works better than PBSFifo and PBSWalltime until the overhead reduction techniques are triggered (i.e. Test 3). In terms of total tardiness, the performance of our approach remains on par with that of PBSFifo even on Test 3, despite the large number of jobs.

Finally, Figure 4.12 compares the overhead to test-execution-time ratio for the three approaches. PBSFifo has the lowest overhead, followed by PBSWalltime, and then by our approach. The overhead of PBSWalltime is, however, the fastest growing one from Test 2 to Test 3, i.e. when the system scalability is more stressed. The overhead of our CP model grows more slowly thanks to the overhead reduction techniques.

Guidelines for algorithm portfolio selection

From the tests reported above, we can observe that the instance scale heavily affects the performance of the scheduler. In fact, being our approach based on search, the overhead introduced by running our scheduler grows with the instance size. On the contrary, we can notice that for realistic-sized instances, our approach is computationally feasible and provides significantly better results in terms of quality.

The purpose of this section is to identify a set of methods that can be used in a portfolio to solve HPC scheduling problems with increasing scale from lightweight to heavy ones, (see Figure 4.13).



	Test 1	Test 2	Test 3
■ CPModel	3,18	21,18	-8,58
■ PBSWalltime	-14,92	7,35	15,65

FIGURE 4.7: Average queue time gain w.r.t. PBSFifo

We have collected statistics on the execution of the Eurora HPC with the aim to characterize its workload. We have generated lower and higher workloads by reducing respectively increasing, the number of job units to test the scalability of the system.

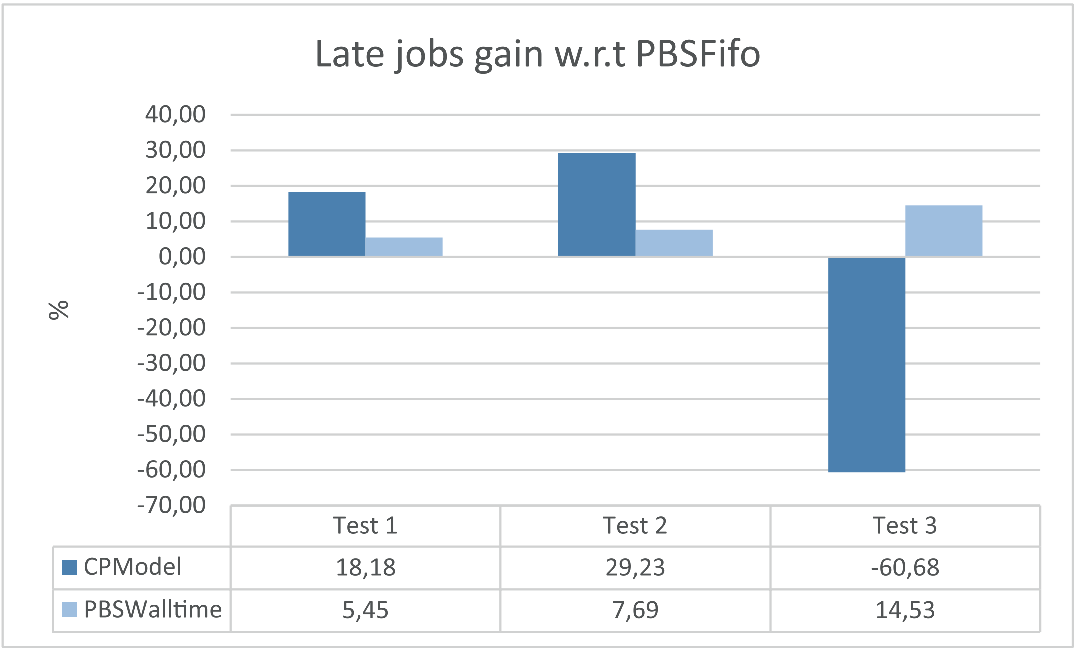
1. On one end of the spectrum we have lightweight workloads, featuring a small number of jobs, each requiring only few nodes. In this situation finding a good schedule is trivial, since the machine is under-loaded, and using powerful optimization techniques provides little benefit.

2. The second class includes mid-range realistic workloads, typically they are characterized by less than 4'100 job units for a 65 nodes HPC machine. This is the range where making good dispatching decisions is not trivial, but the problem size is still manageable. In this situation, the CP system tends to provide the best results.

3. Finally, workloads with a very large number of jobs requiring many computation nodes (namely more than 270'000 job units * nodes submitted in 24h), call for the use of overhead-reduction techniques (Section 4.4). This allows to find solutions in a reasonable amount of time, but with adverse effects on the solution quality. Therefore in this range PBS heuristic approaches become the techniques of choice.

4.5.3 Execution on Eurora

Thanks to our modeling and design efforts from Section 4.3 and 4.4, we have managed to obtain a scheduling system that is mature enough to be deployed and evaluated on the actual Eurora HPC machine.



	Test 1	Test 2	Test 3
■ CPModel	18,18	29,23	-60,68
■ PBSWalltime	5,45	7,69	14,53

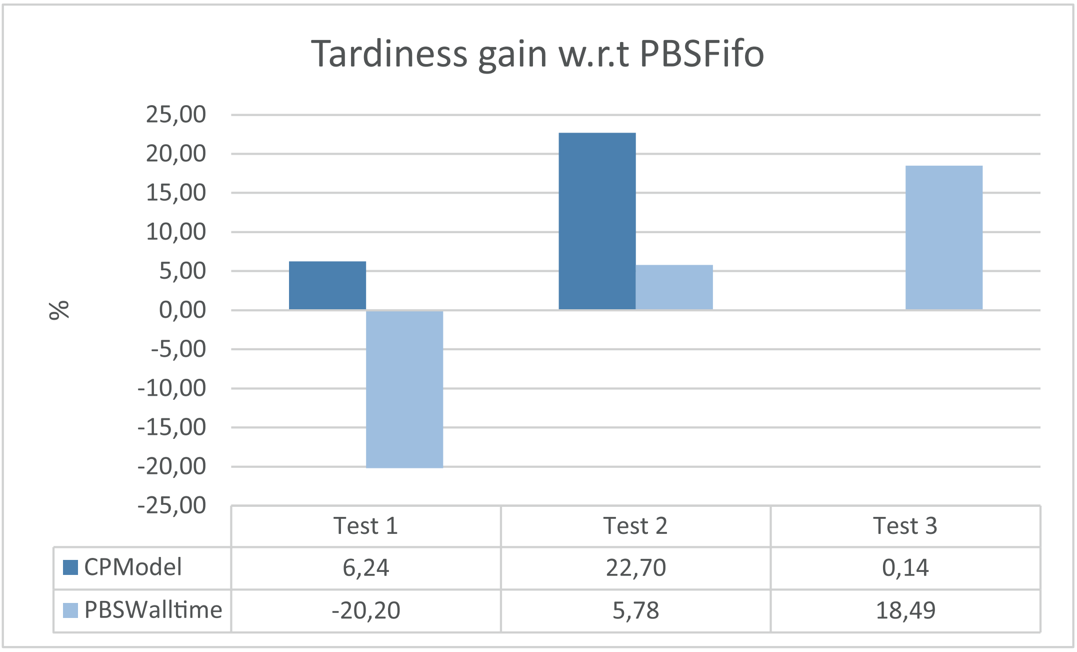
FIGURE 4.8: Number of jobs in late gain w.r.t. PBSFifo

In detail, we have compared the performance of our approach and the PBSFifo configuration (currently employed by CINECA) over five weeks of regular utilization of the HPC machine: the PBS scheduler was employed for the first three weeks and the CP system during the last two weeks. During such period, statistics were collected by relying on the PBS logs. The HPC users were unaware of the change of the scheduling system.

Since the comparison was performed in a production environment, it is impossible to guarantee that the two approaches process the same sequence of jobs. For this reason, the performance metrics that we employed in Section 4.5.1 are not meaningful in this setting and new metrics must be employed. This is due to the big variation between the number of jobs submitted in different days. For this, after some experimentation, we chose to compare the CP approach and PBSFifo in terms of: (1) the average WQT per job, and (2) the average number of used cores over time (i.e. the average core utilization).

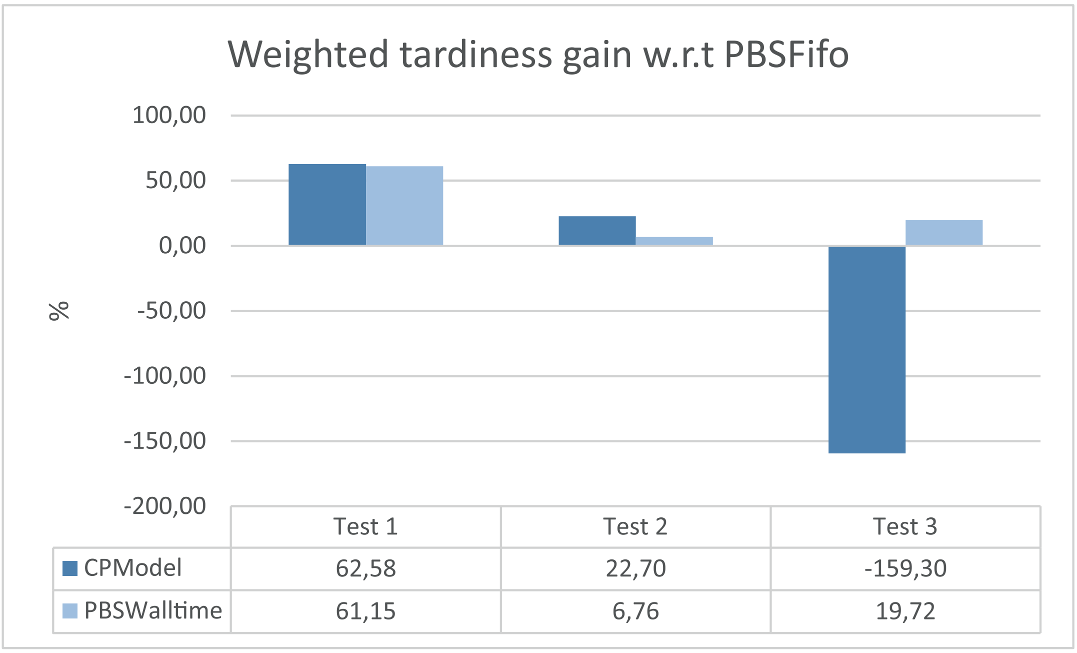
Figure 4.14 compares the two approaches in terms of the first metric. Our CP system performed consistently better with an average WQT per job of $\sim 2.50 * 10^{-6}$, against the $\sim 3.93 * 10^{-6}$ of PBSFifo. The standard deviation for the two approaches is very similar. The average core utilization obtained by both approaches during each week is instead reported in Figure 4.15: the two approach have similar performance in terms of this second metric, which ranges between 520 and 599 for PBSFifo and between 510 and 573 for CP.

We recall that, since it is not possible to ensure that the two scheduling approaches process exactly the same jobs, these results are in part workload-dependent. The metrics we chose are designed to allow a fair comparison, but better (e.g. more robust) metrics may definitely exist: their identification is left as a topic for future research.



	Test 1	Test 2	Test 3
■ CPModel	6,24	22,70	0,14
■ PBSWalltime	-20,20	5,78	18,49

FIGURE 4.9: Tardiness gain w.r.t. PBSFifo

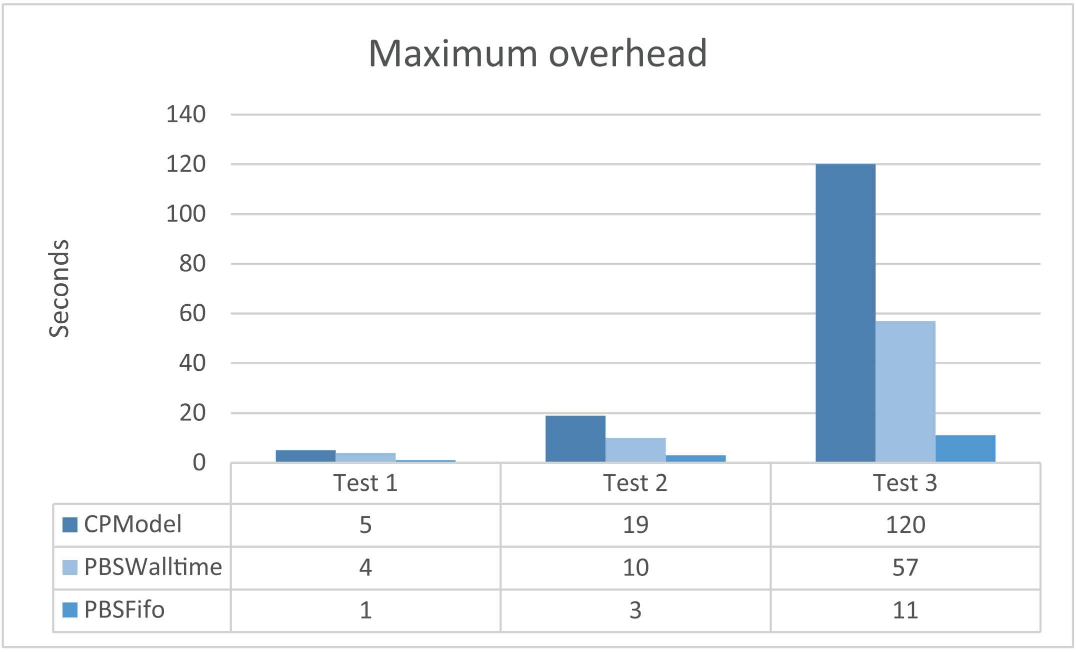


	Test 1	Test 2	Test 3
■ CPModel	62,58	22,70	-159,30
■ PBSWalltime	61,15	6,76	19,72

FIGURE 4.10: Weighted tardiness gain w.r.t. PBSFifo

4.5.4 Overhead distribution

Table 4.16 reports the average time for each execution phase of our system (i.e. the steps in the flow chart from Figure 4.3). From the table, it is clear that moving from the simulated platform to real HPC leads to a considerable *decrease* of the total overhead. The distribution of the total overhead in the simulated tests and on the real system is instead depicted in Figure 4.16 and 4.17: in the simulated tests, the model resolution makes for most of the total overhead; on the real HPC the distribution is more balanced, and some



	Test 1	Test 2	Test 3
CPModel	5	19	120
PBSWalltime	4	10	57
PBSFifo	1	3	11

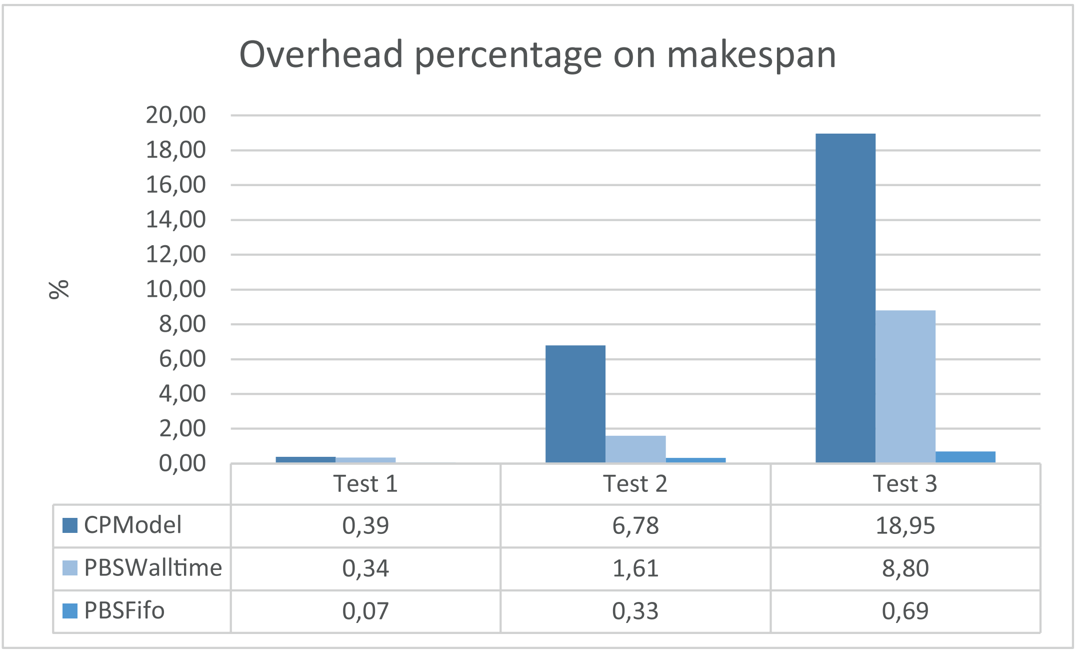
FIGURE 4.11: Maximum overhead gain w.r.t. PBSFifo

	Simulated test	Eurora
Update jobs	4.11	3.96
Update queues	0	0.02
Update Nodes	3.26	1.45
Check reservations feasibility	0	1.21
Check jobs feasibility	0.05	1.02
Jobs and reservations selection	0.07	0.02
Model creation	2.62	0.93
Model execution	21.91	2.31
Reservation check	0	0
Reservations model execution	0	0
Result commit	0.33	2.53
Total	32.35	13.45

TABLE 4.16: Optimization model average overheads (seconds)

phases (reservation/job feasibility check, and result commit) are proportionally much heavier than in the simulated platform.

The differences are likely due to multiple reasons. For sure, the model solution time was heavily affected by the performance gap between our VMs and the Eurora node where the scheduling system was deployed. It is, therefore, likely that the CP approach would in practice be more scalable (i.e. applicable successfully to even larger machines and workloads than Eurora) than what we observed in the simulated experiments.



	Test 1	Test 2	Test 3
CPModel	0,39	6,78	18,95
PBSWalltime	0,34	1,61	8,80
PBSFifo	0,07	0,33	0,69

FIGURE 4.12: Overhead percentage on execution time gain w.r.t. PBSFifo

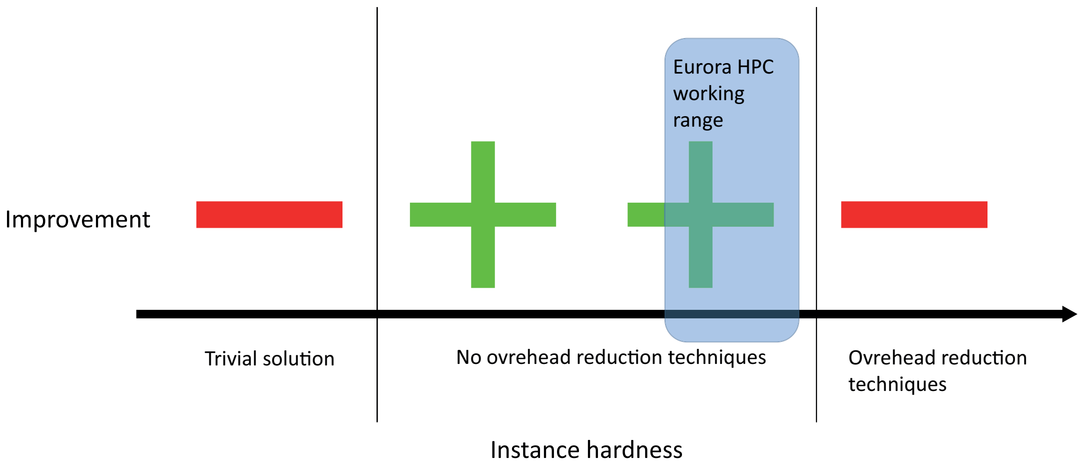
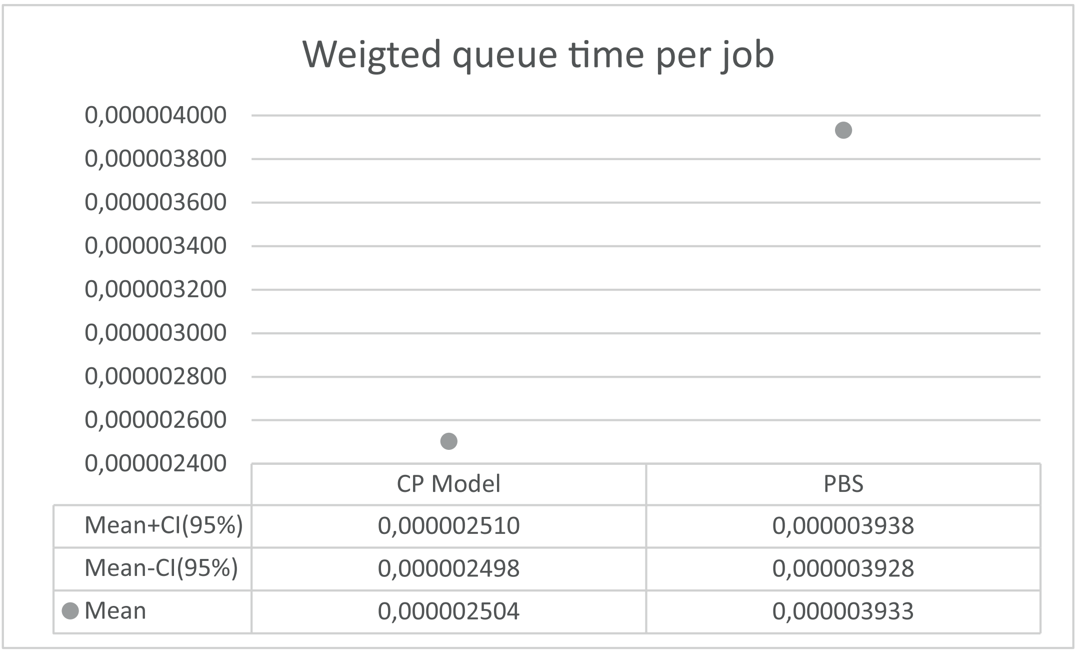
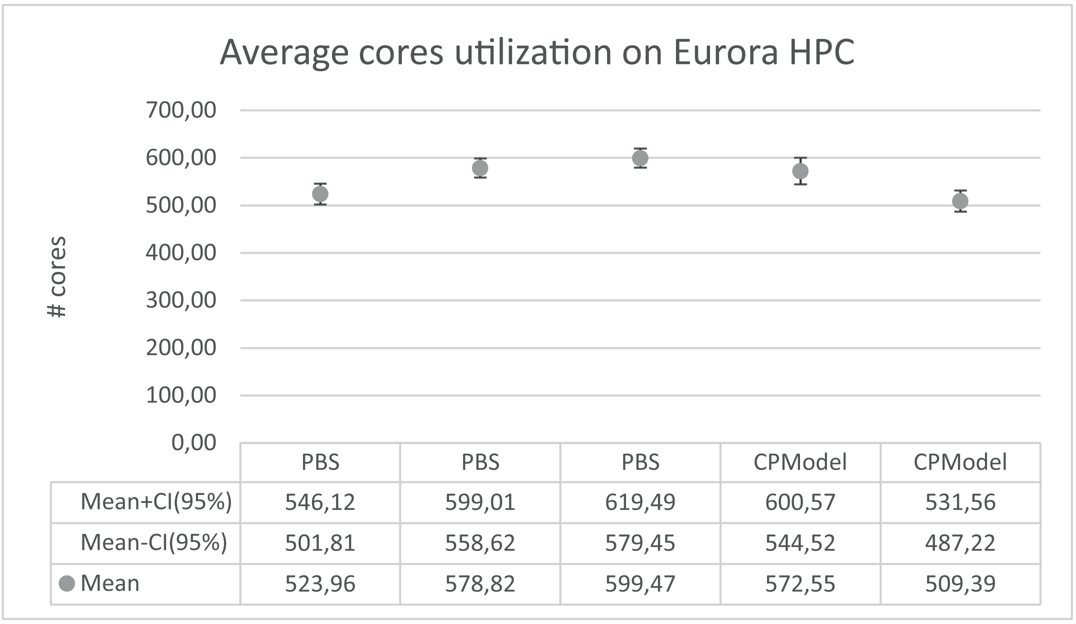


FIGURE 4.13: Working ranges



	CP Model	PBS
Mean+CI(95%)	0,000002510	0,000003938
Mean-CI(95%)	0,000002498	0,000003928
● Mean	0,000002504	0,000003933

FIGURE 4.14: Weighted queue time extrapolated from Eurora



	PBS	PBS	PBS	CPModel	CPModel
Mean+CI(95%)	546,12	599,01	619,49	600,57	531,56
Mean-CI(95%)	501,81	558,62	579,45	544,52	487,22
● Mean	523,96	578,82	599,47	572,55	509,39

FIGURE 4.15: Core utilization on Eurora

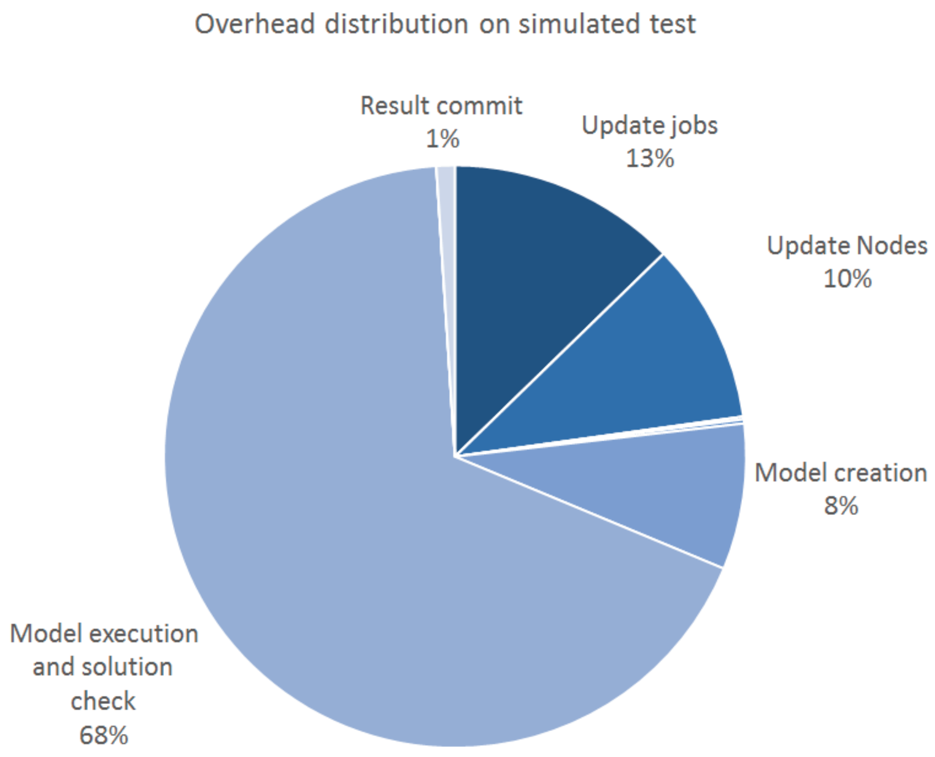


FIGURE 4.16: Overhead distribution for the simulated test

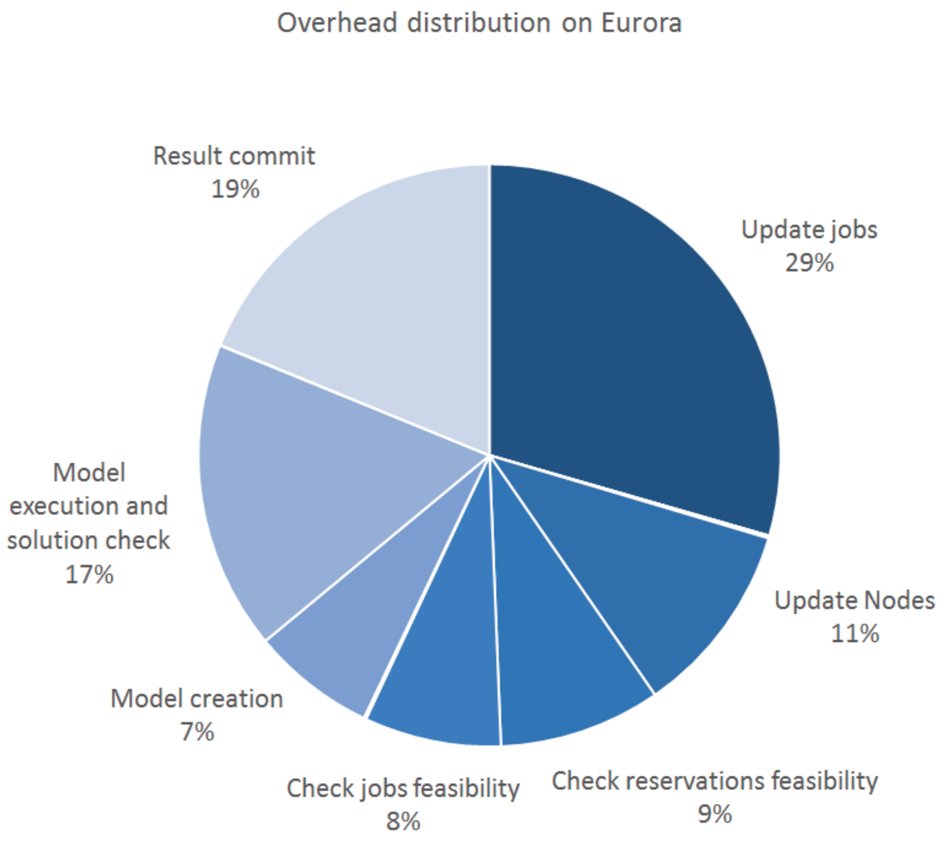


FIGURE 4.17: Overhead distribution for Eurora

4.6 Conclusion

In this work we presented a scheduler, based on Constraint Programming techniques, that can improve the results obtained from commercial schedulers highly tuned for a production environment. We implemented all the features to make it usable on a real-life HPC setting. The scheduler has been tested both in a simulated environment and on a real HPC machine with promising results. We have seen that in a simulated environment with a limited computational power the model has three working ranges (delimited by the hardness of the instance of the problem). The proposed solution can be suitably inserted in a portfolio of scheduling algorithms and dominates commercial approaches in the following conditions: The statistics on the Eurora HPC system show an improvement on the weighted queue time while maintaining similar levels of utilization.

Despite the system has been deployed on a real HPC machine, a number of improvements are still pending: First, the uncertainty on the execution time of jobs, can be considered in the scheduling algorithm and can be characterized through learning techniques as done in the work by Tsafrir et al. [80]. Considering the job execution time uncertainty heavily impacts the scheduler model thus affecting solution algorithms: techniques such as robust optimization and stochastic constraint programming have to be considered. A second improvement can be obtained by providing hot-starts to the optimization engine: they can be either be computed as the solution of the previous run or via sophisticated heuristics algorithms enriched with backfilling techniques. Finally a deeper integration of the optimization engine into the scheduling management framework can be obtained by a changing its source code, this would need longer development time but possibly reduce the overhead introduced by the interaction.

Chapter 5

Improving the HPC scheduling scalability with Distributed And Randomized DIspatcher and Scheduler (DARDIS)

Scheduling and dispatching are critical enabling technologies in High-Performance Computing (HPC). In this context, scalability is an issue: we have to allocate and schedule up to millions of tasks on hundreds of thousands of nodes.

If we take as an example the number of computational nodes a scheduler has to manage for high performance computers like the top HPC in 2015 [136] or the future HPC machine planned in the USA [137], these machines feature a number of nodes estimated between 50'000 and 1'000'000 [11, 138]. The scale of these machines is out of reach for complete and centralized scheduling approaches. This is a clear problem for today's commercial schedulers that are centralized and rule-based. It is quite clear that centralized approaches will hardly scale up to future large-scale (exascale) HPC systems [138]. Hence, scalable, distributed schedulers are needed to handle thousands of nodes while at the same time optimizing efficiency metrics (e.g. reducing operating cost, maximizing utilization).

In addition, the intrinsic computational power of future HPC installations is bounded by their total consumed power, which has a practical limit of 20 MWatt due to constraints in the energy provisioning system [139]. Until today, the de-facto solution to this problem has been to statically design a system which respects this constraint in terms of peak power. However, the peak power is reached only for a few applications, such as the HPL linpack[10], while during normal operation the machine consumes significantly less power. This leads to a reduced operating capacity and reduced return over investment. To avoid this issue, strict strategies try to preserve this bound at run-time. Several works take advantage of the node HW capabilities [140, 141, 142] to cap their power computation while other works tackle this problem at the job scheduler level [81]. When the power budget is decided based on exogenous inputs such as the electricity price, environmental parameters (i.e. ambient temperature) and administrative targets, this budget changes at run-time leading to a variable profile.

Classical job schedulers are rule-based [50]. These are heuristic schedulers that use rules to prioritize jobs. In these scheduling systems a job requests a set of resources on which the job will execute different job units. All the job units of the same job execute in parallel. The scheduler checks for each job unit of a given job if it can execute in a node while respecting the capacity of the target resources. If all the job units can use the requested amount of resources, the job is executed. To enforce fairness, supercomputing centers use different queues or partitions, targeting jobs with different resource request and different expected waiting time. This expected waiting time is viewed by the final user as a sort of soft-deadline and quantifies a user-satisfactory metrics.

In the literature there are several works on distributed scheduling [85, 86]. However, all of these approaches are suitable for the case of grid scheduling (meta-scheduling). The main limitation to apply these approaches to the HPC scheduling is that they do not support job workloads with multiple job units. Taking multiple job units into consideration would introduce, for the above mentioned schedulers, a stage of synchronization between nodes that would increase dramatically the number of exchanged messages between nodes and the overhead spent in the scheduling.

In Randomized Load Control [143], the algorithm schedules appliances activities using a load probability distribution based on a desired total energy consumption profile. The profile can be variable and can be designed to exploit the pricing model of the energy supplier to minimize the costs for the users. It is shown that the approach obtains results that match well the desired solution. Moreover, the author demonstrates that a randomized approach to the schedule of these activities can obtain results as good as an optimization model.

In this work, we substantially extend the work in [143] and apply it to the HPC domain, introducing the possibility to schedule tasks on more than one resource, we introduced the concept of node, that is a collection of resources. We, also, extended the work introducing a set of nodes and the possibility for a job to ask a number of job units for a parallel execution.

Our contribution is a Distributed And Randomized DIspatching and Scheduling for HPC (DARDIS) approach that is:

- **Distributed** to scale to very large systems. The scheduler and dispatcher basically leaves the dispatching choice to the job and then each node schedules its own jobs. Moreover, this approach can match the jobs starting times on different nodes for the jobs' parallel execution without communication between nodes.

- **With support for variable resources' utilization profile**. Each resource, can exhibit a variable/desired utilization profile. This feature is essential to reduce the costly overprovision in traditional power and cooling design strategies.

- **Randomized**. The scheduler uses different probability distributions for selecting the jobs starting times and dispatching policies to optimize different objective functions.

- **Customizable**. We will show three different setups of DARDIS each one designed to optimize a different goal: throughput, balancing, costs minimization.

- **Aware of user deadlines**. Each job can specify a time window in which it should start. This can be used by facility administrators to create jobs with different priorities (the smaller the window the higher the priority). Note that a window with length of one correspond to a reservation. The administrator could also decide to apply a pricing model inversely proportional size of the user-specified window.

DARDIS has been implemented using the MPI library. Tests have been executed on a server with 2xIntel Xeon DP 12 Core E5-2670v3 and 128GB of RAM and compared with state of the art for commercial approaches. We chose to compare DARDIS with rule-based approaches for two reasons: (1) in literature, to the best of our knowledge, is not present a distributed approach for the scheduling suited for HPC, i.e. all the distributed approaches studied do not consider the parallel execution of a job on different nodes, (2) two of the key features of DARDIS are responsiveness and scalability and the best approach that we can compare with under this metrics is rule-based scheduling.

In addition, to make allow a fair and meaningful comparison, we have compared our approach to a rule-based scheduler we custom tailored to enforce a variable profile. At each event, this scheduler tries to allocate a job considering not only the unused resources between the resource capacity and the current resource utilization, but also considering the unused resources between the variable profile and its current utilization.

Results show that this approach obtains benefits derived from the reduction of the overhead and the introduction of a desirable utilization profile, but also we show that DARDIS can improve deadline exceeding, queue waiting and utilization w.r.t. rule-based schedulers from $\sim 31\%$ to $\sim 57\%$ with a much lower computational cost (200x lower). Moreover, we demonstrate that the overutilization over the variable/desired profile obtained by DARDIS is created only to minimize metrics with a higher specified importance.

In section 5.1 we discuss the importance of variable constraints as power and thermal bounds for current and future HPC machines. In section 5.2 we describe a generic rule-based variable profile aware scheduling approach. In section 5.3 we show the DARDIS approach to the scheduling, all the setups designed and the objective they optimize. In section 5.5 we show experimental results and the comparison w.r.t rule-based schedulers. Finally, section 5.6 summarizes our conclusions.

5.1 Motivational example

Until today, the peak performance of supercomputers has been limited only by area, investment cost and available technology at procurement time. At operation time all the computing nodes are usually configured to run always at their maximum performance level disregarding their actual efficiency. In the recent years with the push toward the Exascale and more sustainable computing, supercomputing centers, as well as system manufacturers, have started facing the physical limits of the computing devices evaluating the possibility of constraining/capping at run-time different figures of merit (i.e. total power, used resources, node performance) to gain efficiency and cost effectiveness. In this section we give some motivations of these trends as a support of the proposed methodology.

Power Capping Several authors have presented strategies for constraining at run-time the power of an entire supercomputing machine allowing to use more computational nodes during average operations which consume lower power than the peak one while ensuring the safe limits during the execution of jobs which consumes peak power which happens rarely. To improve the efficiency of these solutions, authors in [144] consider variable in time and per-job power budget of each node to speed up critical sections. In addition, Inadomi et al. [145] show that nominally similar supercomputing nodes have different power vs performance trade-offs and that the power budget should be reallocated proportionally to the intrinsic performance of the node to produce a virtually homogeneous cluster.

Energy Capping Authors in [146] show that, as consequence of advanced cooling methodology (i.e. hybrid air/liquid cooling and free-cooling), the cooling efforts and the PUE depends on the instantaneous IT power (load of the machine). In this domain a time variable power capping can be used as a knobs to enforce a target PUE under different ambient temperature and condition.

Thermal Capping Authors of [147] show that the air flow distribution, in a supercomputer racks and rooms, strongly depends on the floorplan and on the node position in the rack, creating heterogeneous air flows which leads to difference in the cooling efficiency of different nodes. Effective strategies to avoid cooling overprovisioning while preserving safe working temperatures for all the processing elements would be to apply heterogeneous power budget to different nodes accordingly to their position and cooling efficiency.

As just described, supercomputers would gain benefit by operating under time-variable and per-resource power and resource usage constraints.

5.2 Profile aware scheduling

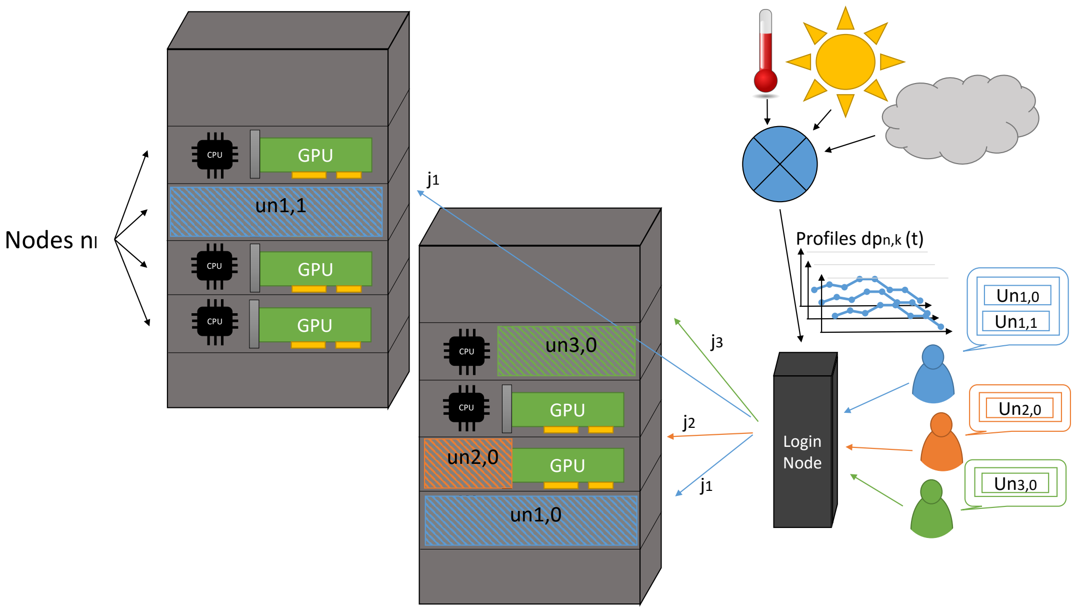


FIGURE 5.1: Example of a profile aware scheduling architecture

Algorithm 4 Profile aware rule-based scheduling

```
 1: procedure SCHEDULINGCYCLE(PRIORITYRULE, WAITINGJOBS, TIMESTAMP)
 2:     orderJobsBy(priorityRule, waitingJobs)
 3:     foreach job $i \in$ waitingJobs do
 4:         foreach job unit $w \in ju_i$ do
 5:             foreach node $n \in L$ do
 6:                 foreach resource kind $k \in K$ do
 7:                     if $up_{n,k}(timeStamp) + req_{i,k,w} < min(dp_{n,k}(timeStamp), c_{n,k})$ then
 8:                         set the execution of the job unit $w$ of job $i$ to the node $n$
 9:                     end if
10:                 end for
11:             end for
12:         end for
13:         if all the job units of job $i$ have a node then
14:         else
15:             clear the nodes of each job unit of job $i$
16:             call the backfilling algorithm
17:             return
18:         end if
19:     end for
20: end procedure
```

Figure 5.1 show the architecture of our variable profile aware rule-based scheduler. The scheduling under variable profiles problem can be modeled by a set of different kinds of resources rk_k (e.g. cores, GPU, mem, power), with $k \in K$, a set of nodes n_l, with $l \in L$ and a set of jobs a_i with $i \in A$ each composed by a set of job units $un_{i,w}$, with $w \in W_i$. Each node contains a resource for each kind $res_{n,k}$.Each resource has a capacity $c_{n,k}$, a variable/desired profile $dp_{n,k}(t)$ and a utilization profile $up_{n,k}(t)$, with $t \in [0,..,Eoh]$. The variable/desired profile is a profile described by the administrator that show how the resource should be used (in term of number of amount of resource used by jobs) in time. The utilization profile is the amount of resource used/reserved to already scheduled jobs.

Each job is submitted to the system at a time instant q_i. At the submission the job specifies its earliest start time est_i, the latest start time lst_i, its walltime wt_i, the number of job units ju_i and the amount of resource required $req_{i,k,w}$ for each kind of resource and for each job unit.

The scheduling problem consists of selecting start time st_i for each job j_i and a node to each job unit $un_{i,w}$ of the job such that:

$$\begin{aligned}
& st_i :: [est_i, .., lst_i] \; \forall i \in A \\
& un_{i,w} \in n_l \; \forall i \in A, \forall w \in W_i, \forall l \in L \\
& \sum_i req_{i,k,w} \leq c_{n,k} \; \forall t \in [0,..,Eoh], \forall k \in K, \\
& \qquad \forall i \in A | st_i \leq t \wedge st_i + wt_i > t \wedge un_{i,w} = n \\
& \sum_i req_{i,k,w} \leq dp_{n,k}(t) \; \forall t \in [0,..,Eoh], \forall k \in K, \\
& \qquad \forall i \in A | st_i \leq t \wedge st_i + wt_i > t \wedge un_{i,w} = n
\end{aligned} \tag{5.1}$$

5.2.1 Rule-based schedulers for variable profiles

To enforce a variable profile on the resources classical rule-based scheduler can be extended to include in the scheduler the profile knowledge and at each scheduling cycle to check if the resource constraint is fulfilled.

The Algorithm 4 describes the profile aware scheduling algorithm used in our experimentation. The only difference from a normal rule-based scheduling algorithm [50] is in line 7: the algorithm checks not only to not exceed the resource capacity but also the variable/desired profile for the resource.

5.3 DARDIS approach

In this work we propose an innovative approach for variable profile scheduling on massively parallel resources called DARDIS. In DARDIS we designed a different approach which has a (i) distributed nature to tackle the problem of the exponential growth of the supercomputer capacity and computing resources and (ii) enforces the satisfaction of a variable/desired profile not only at the current time stamp, as the rule-based scheduler does, but also by looking at future intervals. The main idea of the proposed scheduler is to partition the decision process in two main phases performed by two separate software entities: the "task manager" and the "node manager". The *task manager* is responsible for the job submission and the dispatching. This software component executes into the user-space (e.g. into the user workstation/PC). The *node manager* is responsible for the scheduling. This software component executes into the computing nodes.

Figure 5.2 shows the different phases of our approach. This phases are subdivided in:

Job Submission (1) - Our approach starts with a task manager submitting a job to all the node managers of the system. At submission time, it specifies the job ID, the number of job units, the amount of required resources, the walltime, the earliest start time and the latest start time for its execution. After the submission to all the node managers, the task manager waits for the responses.

Start-time probability generation (2) - Each node manager, after receiving the submitted job request, calls the start time probability generation phase in which the node manager generates a start time for the job according to an internal rule (Section 5.3.1).

Start-time response (3) - After the start time generation is completed, the node manager running in each node sends the generated start times to the task manager which run on the user pc.

Resource selection (4) - The task manager, after receiving the responses from all the nodes, applies a policy to select the nodes for the job units' execution. The policy depends on the goal of the scheduler, all the goals and policies are discussed in sections 5.3.3, 5.3.4 and 5.3.5. In case of error (like communication fault or node crash) a timer on the task manager let the protocol to continue with the responses obtained until that time. A number of missing responses from the node managers could increase the probability to do not find a feasible schedule for the job before its deadline. This probability depends on the state of the used resources. However, if an allocation is not found this case is ascribable to the case of a job that cannot be scheduled within its deadline (see Section 5.3.6).

Resource confirmation (5) - The task manager sends the result to all the node managers involved in the submission, namely, the one selected and those not selected by the dispatching policy.

Resource reservation (6) - The node managers in which the job has to execute, reserves the proper capacity for the execution, by modifying the utilization profile.

A submission sent during the start time probability generation of a previous job, waits for the end of the resource reservation of the job. This behavior was chosen for two reasons: (i) usually in the domains selected, the computation of the scheduling cycle takes negligible time w.r.t. the execution-time of jobs; (ii) with this approach we do not have to manage concurrency on the same resources (overutilization): the utilization profile cannot change during the protocol.

It must be noted that the number of exchanged messages for scheduling a single job depends linearly on the number of the nodes of the system. In particular, for each job we exchange $3 * L$ messages where L is the number of nodes involved in the submission.

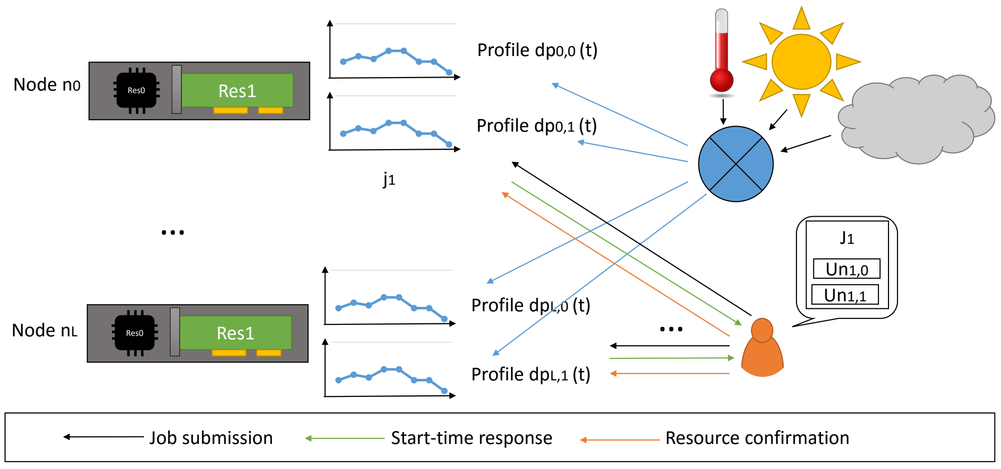


FIGURE 5.2: DARDIS architecture

5.3.1 Scheduling

As described above, the software component responsible for the start time generation is the node manager of the supercomputer which, computes the list of feasible start times for a given job on a node. This component manages the resources and the jobs present on the same node in which is running. We call this phases start time generation.

The start time generation process for the node n, starts by computing a fitting index for the submitted job i. This index indicates how many parallel runs of the same job unit of the job could be executed at a given start time s while satisfying the desired utilization profile for all the resources hosted in the node. Since the desired profile is variable, the node manager has to check for each time instant $t \in \{s, .., s + wt_i\}$ how many times the job's resource requirement req_i can fit the space left between the utilization profile and the variable/desired profile. The utilization profile represent the amount of resources reserved to scheduled jobs in each time instant t. This is repeated for each kind of resource K present in the node and the minimum index is held (equation 5.2).

$$I'(s) = min_k(min_t(\frac{dp_{n,k}(t) - up_{n,k}(t)}{req_{i,k,w}})) \\ \forall t \in \{s, .., s + wt_i\}, \forall k \in K \tag{5.2}$$

Note that $I'(s) = 1$ means that the job unit perfectly fits into the resources without exceeding none of the variable/desired profiles. $I'(s) > 1$ means that the job unit fits the variable/desired profiles and leaves some resource for other jobs. If $I'(s) < 1$, it means that the job unit exceeds at least one of the variable/desired profiles of the node's resources. The capacity instead cannot be exceeded by definition. To handle this case, we use equation 5.3. Where $\setminus$ represents integer division.

$$\begin{aligned} I(s) = min(I'(s), min_t((c_{n,k} - up_{n,k}(t)) \setminus req_{i,k,w})) \\ \forall t \in \{s, .., s + wt_i\}, \forall k \in K \end{aligned} \tag{5.3}$$

The index distribution I is calculated for each possible start time between the earliest start time est_i and the latest start time lst_i of the job: $I = \{I(est_i), .., I(lst_i)\}$. In this way we obtain the fitting profile for the job.

There are several different ways to obtain start times from the fitting profile, each way is designed to optimize a different goal. We will discuss these in sections 5.3.3, 5.3.4 and 5.3.5. Independently from the chosen start times generator, we can assume that this process does not generate start times in a deterministic way from the point of view of the job. This is due to the facts that: the generation is based on the utilization profile of each node, each node executes different jobs and the task manager is not aware about the job in execution on each node.

5.3.2 Dispatching

The dispatching problem, for a given job j_i with ju_i job units, consist in selecting a set of equals start times of cardinality equals to the number of job unit ju_i, the job will be executed on the nodes involved in the set of start times. Since the start times are generated by the nodes independently, the job does not know which values it will receive. However, it needs at least ju_i different resources that reply with the same start time in order to find a feasible allocation. To solve this issue, we model the dispatching problem as the birthday problem [148]. In its work, Von Mises shows that given a set of people of cardinality NP, the probability to find two persons with the same birthday date increases faster than the probability to find only one person with a birthday in a specific day of the year while increasing the cardinality of the set of people (NP). Unfortunately, in the HPC domain we have to guarantee that a job waits to execute only because there are no resources available and not depending on the scheduler architecture and policy. To have this guarantee behavior we can exploit the pigeon principle [149]. This principle states that having n items to be placed in m containers, to be sure that at least one container receives more than one item, n has to be greater or equal to $m + 1$. We used a generalization of this principle to find a feasible dispatching. We start from the hypothesis that each start time selected by the scheduling algorithm has a fitting index ≥ 1. If we have a job asking ju_i job units and a starting-times-window $window_i = lst_i - est_i$ on a HPC cluster composed by L nodes. We can guarantee a probability of dispatching the job on the cluster within $window_i$ equal to 1 simply by forcing each start-time generator running in each node to return a number of different start times equal to $responses_i = \lfloor \frac{(ju_i - 1) * window_i}{L} \rfloor + 1$. With this number of start times per node we have the guarantee that at least one start time in the interval $[est_i, .., lst_i]$ is valid for at leasts ju_i nodes.

As for the start time generation, even the dispatching can be customized to optimize different goals.

In the following sections we will discuss a set of different policies for customizing the start time generation process to target different goals, namely throughput, profile fidelity, balanced and deadline satisfaction.

5.3.3 Throughput driven DARDIS

This setup is designed to maximize jobs throughput while maintaining the variable constraint on the resources. This can be done by making the scheduler always selecting the earliest feasible start time for the job. This setup considers deadlines more important than the variable/desired profile limit. This means that if a job cannot be scheduled within its deadline while respecting the variable/desired profile, the scheduler tries to schedule the job always respecting its deadline disregarding the constraint on the profile.

The start time generator (node manager) selects from the window $window_i$ a set of cardinality $responses_i$ containing the minimum time instants with fitting index ≥ 1.

After that the task manager, in the dispatching algorithm, clusterizes the sets of responses obtained by each node by start time and selects the minimum start time with a set of nodes of cardinality ju_i.

Backfilling in DARDIS

Being the walltime specified by the job an estimation which is usually overestimated. The majority of the HPC schedulers apply a backfilling algorithm. The objective of this algorithm is to avoid resources underutilization created by jobs terminating before their walltime. This is usually done by anticipating the jobs that fits the unused resource but which does not introduce an additional delay to the jobs with higher priority than the anticipated ones [65]. Throughput driven DARDIS setup is the only designed to compute a backfilling algorithm. The algorithm is designed to compute the backfilling on a certain number of jobs ($depth$) for each node involved in the execution of the terminating job. At each job termination, each node involved in the job execution notifies the firsts $depth$ task managers scheduled in the future on that node. For this test we selected a depth equal to the number of cores for each node (32 in our experimental setup). After the job termination, the task manager restarts the algorithm requesting a new schedule only for the nodes in which the job has been dispatched previously. The only difference from the first schedule is that the window $[est_i, .., lst_i]$ of the job is modified in $[currentTimeStamp, .., st_i]$. If the node managers return a feasible set of start times in that interval, the start time and the utilization profiles are updated. Otherwise, the start time generated previously is hold.

5.3.4 Profile driven DARDIS

This setup is designed to obtain a utilization profile proportional to the input variable/desired resource profile which pays off when the profile is not only a constraint but is also an indicator of cost/penalty: i.e. the objective is to minimize the squared distance from the variable/desired profile. In this case, is preferred to create more fragmentation in the system to minimize the cost produced by the resource utilization. It must be noted that with this approach the number of jobs exceeding their latest start time will be higher with respect to the other policies. However, this setup always generates start time in the job interval $[est_i, .., lst_i]$. The higher number of jobs exceeding the deadline is due to the resource fragmentation obtained indirectly from the schedule. This setup considers deadlines more important than the variable/desired profile limit. This means that if a job cannot be scheduled within its deadline while respecting the variable/desired profile, the scheduler tries to schedule the job always respecting its deadline disregarding the constraint on the profile.

In this setup, the node manager generates start times with a probabilistic selection that chooses a random number rnd in the range $[0, .., \sum_{s=est_i}^{lst_i} I(s)]$. The start time st_i is then obtained by imposing the conditions $\sum_{t=0}^{st_i} I(t) \geq rnd$ and $\sum_{t=0}^{st_i-1} I(t) < rnd$. If the selected start time has a fitting index $I(t) < 1$, t is increased until the condition $I(t) \geq 1$ is verified. If a start time is not found in this range, the search is repeated starting from $I(est_i)$ and the first $I(t) \geq 1$ is chosen. If a start time cannot be found in the entire range, the allocation is infeasible on the specific resource and the generator fails returning a null value.

After that in the task manager the dispatching algorithm clusterizes the sets of responses obtained by each node by start time and selects randomly a feasible start time and a random subset of ju_i nodes.

5.3.5 Balance driven DARDIS

This setup is designed to improve balance not only in the allocation but also on the start time generation. Meaning that, this generator achieves a trade-off between throughput and profile chase on the start time generation and then selects the start times and the nodes to obtain a balance workload in each node. This setup considers deadlines more important than the variable/desired profile limit. This means that if a job cannot be scheduled within its deadline while respecting the variable/desired profile, the scheduler tries to schedule the job always respecting its deadline disregarding the constraint on the profile.

The start time generator is a probabilistic generator that chooses a random number rnd following the distribution $(lst_i - est_i)e^{-(lst_i - est_i)x}$. Then it computes the start time by imposing the conditions $\sum_{t=0}^{st_i} I(t) \geq rnd$ and $\sum_{t=0}^{st_i-1} I(t) < rnd$. If the selected start time has a fitting index $I(t) < 1$, it increases t until the condition $I(t) \geq 1$ is verified. If a start time is not found in this range, the search is repeated from $I(est_i)$ and the first $I(t) \geq 1$ is chosen.

If a start time cannot be found in the entire range, the allocation is infeasible on the specific resource and the generator fails returning a null value.

Then the dispatching algorithm in the task manager clusterizes the sets of responses obtained by each node by start time and applies a policy that selects the start time that $maximise : \sum_l I_l(st)$, with $l \in L$ and L the set of nodes. After the start time selection, the subset of ju_i nodes is selected with the same policy: the ju_i nodes with the highest $I_l(st)$.

5.3.6 Deadline exceeding

Independently from the used setup. If a job cannot be allocated it the interval $[est_i, .., lst_i]$, the task manager restarts the protocol with a new window $[est'_i, .., lst'_i]$. Where $est'_i = lst_i$ and $lst'_i = (lst_i - est_i) * 2$. This approach can be repeated a number of times. From experimentation we found that a good compromise is 3 attempt. If, again, a schedule cannot be found the task manager sets the allocation window after the last termination: $est'_i = max(st_i + wt_i)$ and $lst'_i = max(st_i + wt_i) + lst_i - est_i$. This last attempt gives us the certainty that if a job cannot be scheduled, this is due only by an error on the resource request or on the system state (e.g. the job is requiring a quantity of resources per job unit not present in any node, the whole set of resources required from a job is higher than the resources in the system, too many nodes of the system are turned off or crashed, etc...).

5.4 Complexity study

DARDIS is composed by two main components: the node manager and the task manager. The complexity of the node manager resides into the start time generation algorithm. The algorithm calculates a fitting profile for a submitted job. This fitting profile is composed by $window_i = lst_i - est_i$ fitting indexes for each kind of resource (K). Finally, a fitting index is derived by checking the variable/desired profile and the resource required by the job for each time instant of its execution wt_i. The complexity is given by $O(K * window_i * wt_i)$. Being K fixed and $window_i$ and wt_i dependent from the size of the input variables, this algorithm is pseudo-polynomial.

The complexity of the task manager resides into the node selection algorithm. The algorithm searches through the responses from all node managers L for a set of equals start times of cardinality ju_i. Being the response composed by $responses_i = \lfloor \frac{(ju_i - 1) * window_i}{L} \rfloor + 1$ different values, the complexity is $O(L * responses_i) = O(ju_i * window_i)$. For this reason, we can say that the complexity of the task manager is pseudo-polynomial. However, being usually $ju_i \ll wt_i$ the node selection algorithm is dominated by the start time generation algorithm in term of execution time.

The complexity of rule-based approach depends both on the number of nodes and the number of job units: $O(K * ju_i * L)$. The power of this approach comes from the fact that all of the terms in the DARDIS complexity are bounded in most of the batch scheduling environments while the number of nodes L will constantly increase. For this reason we can claim that our approach is more scalable w.r.t. a heuristic rule-based approach.

5.5 Experimental results

In this section we evaluate DARDIS against a set of rule-based schedulers. We first describe the rules used in the rule-based schedulers we compare with. We then describe the experiment setup and we define the performance metrics. Finally, we show two sets of results: a performance and an overhead comparison.

All the three different setups of DARDIS have been tested and compared against three different setups for the rule-based scheduler. The rule-based scheduling setups are:

- RB-FCFS: the jobs are ordered by increasing earliest start time. The algorithm checks to not exceed the variable/desired profile at scheduling time.
- RB-DF: the jobs are ordered by increasing latest start time. The algorithm checks to not exceed the variable/desired profile at scheduling time.
- RB-WT: the jobs are ordered by increasing walltime. The algorithm checks to not exceed the variable/desired profile at scheduling time.

The test is based on the parallel workload archive of the CEA Curie system [150]. This system originally was composed by 360 nodes with four 8-core processors and 128 GB of RAM. The scheduler in use is Slurm, and the system is subdivided in 33 partitions. The schedulers have been tested on 300 out of 360 of the fat nodes of the system for a total of 9600 cores and 38 TB of RAM. A total of 35538 jobs submitted in 22 days of regular workload have been extracted from the trace log and used for the benchmark. This trace log does not consider explicit deadlines. However, there are implicit soft deadlines in the setup which define the user satisfaction. For this reason, we use the arrival time as est_i. After that, we extrapolated the average waiting time for each partition $aqt(partition)$ and used it to compute the $lst_i = est_i + aqt(partition)$.

The metrics used for the comparison are different and with different targets. Some metrics are of interest for the user (user criteria), others for the system administrator (administrator criteria). We used the following metrics for the comparison:

- Makespan: it measures the completion time of the set of jobs. This is an administrator criteria: a scheduler that obtains a lower makespan within the same set of job means that it will produce a higher system utilization on average.

- Total waiting time of jobs: computed as the sum of all the jobs wait. This is a user criteria: a scheduler that obtains a lower total wait means that users will have to wait for lower time in general.

- Tardiness: computed as the sums of the delay w.r.t. the latest start time of the job. This is a user criteria: if the tardiness is low, it means that a few jobs have exceeded the deadline. However, it could also be an administrator criteria if the computing center has strict service level agreements and penalty on the job deadlines exceeding.

- Overutilization: computed as volume of resources utilization in time, that exceeds the variable/desired profile. This is an administrator criteria: a variable/desired profile exceed could be translated in too high expenses for the computing center or event to penalty from the energy provider.

- Number of late jobs: computed as number of jobs exceeding their deadline. As the tardiness this could be both a user or administrator criteria for the same reasons.

- Dissimilarity: this metric quantifies the dissimilarity of the final utilization profile and the desirable utilization profile. It is designed to indicate if two functions have a different shape without considering the difference of volumes. The metric uses as input two functions: the variable/desired profile $g(t)$ and the profile to measure $f(t)$. These two profiles are considered from the time instant 0 to the maximum deadline. First, the function $f(t)$ is multiplied by factor ψ, computed as $\psi = min_t(\frac{g(t)}{f(t)})$. This multiplication of the utilization profile with ψ is used to compare the Dissimilarity of two different utilization profiles with the same variable/desired profile without the bias introduced by high differences in the length of the two utilization profile. Than we compute the discrete auto-correlation for $g(t)$ R_{gg} and the cross-correlation between $g(t)$ and $f(t)$ R_{gf}. The metric is obtained as: $Dissimilarity = \frac{R_{gg}(0)-R_{gf}(0)}{R_{gg}(0)}$. This metric is not affected by the make-span of the profile: other metrics (as for example the squared difference between the desirable and the utilization profiles), do not give the possibility to compare utilization profiles with different durations. Obviously, this metric is meaningful only with the presence of jobs deadlines. This is an administrator criteria: in some cases, the cost of a resource like e.g. power, could be proportional (or almost proportional) to the variable/desired profile, in these cases a scheduler with a low Dissimilarity tends to minimize costs. However, this could produce high waiting for the users but the computing center could apply a pricing model that grants benefits to these.

Table 5.1 shows the first set of results in absolute values while figure 5.3 shows the results normalized by the maximum value achieved by the best DARDIS setup for each metric. Each DARDIS setup takes its name by the goal it optimizes.

For the makespan, total waiting and tardiness metrics the best results are obtained by the Throughput DARDIS. Throughput DARDIS outperforms all the rule-based schedulers of the 41-42% in Makespan, and 31-50% in total waiting and tardiness. The other two versions of DARDIS obtain poorer results for all these metrics. This is due to the fact that their optimization goals are in contraposition to these metrics.

For the number of late jobs metric, the best result is obtained by the Balanced version of DARDIS. Balanced DARDIS outperforms all the rule-based schedulers of the 54-57% in this metric. But also the others two versions of DARDIS obtain good result under this metric: Throughput DARDIS outperforms the rule-based schedulers of the 47-51% while the Profile DARDIS outperforms the rule-based schedulers of the 53-56%.

From the table we can notice that rule-based schedulers obtain better results in overutilization. It is important to note the motivation why both these scheduling approaches create overutilization. In these three setups, DARDIS is configured to use the variable/desired profile as a soft constraint. Moreover, this constraint by configuration this soft-constraint has lower priority than the deadline soft-constraint. Under this consideration we can motivate the high overutilization as result of the strict deadline. For the rule-based schedulers, the variable profile is a hard constraint which is checked only at submission time. This means that the overutilization obtained by DARDIS has been caused by a decrease in the number in job in late while the overutilization obtained by the rule-based scheduler derives from the architecture of the scheduler itself.

For the dissimilarity, Profile DARDIS outperforms the rule-based schedulers by the 0,2-2,4% while Throughput and the Balanced DARDIS behave similarly to the rule-based approaches.

	MKS (day)	Wait. (years)	Tard. (years)	Late jobs	Overutil.	Diss.
Thro. DARDIS	11.69	79.85	79.58	14722	781886	0.990
Bal. DARDIS	178.05	661.75	611.38	12725	1227779	0.981
Prof. DARDIS	200.44	759.96	759.46	13141	785639	0.974
RB-FCFS	19.70	158.33	158.70	29102	257856	0.976
RB-DF	19.97	129.44	128.47	30098	368234	0.980
RB-WT	19.93	116.50	115.34	27908	395657	0.998

TABLE 5.1: Results obtained by DARDIS and rule-based schedulers on 300 nodes and 35538 jobs scheduling

The radar chart in figure 5.3 gives a synthetic view of the comparison among the DARDIS setups on the quality metrics. In this chart all the metrics were normalized and better results are those nearest to the zero. The chart shows that Throughput DARDIS obtains the best results in makespan, waiting, tardiness and overutilization while the worst in late jobs and dissimilarity. The Profile DARDIS obtains the best in dissimilarity, an average result in late jobs and overutilization and the worse in makespan, waiting and tardiness. Instead, the Balance DARDIS obtains the best in late jobs, the worst result in overutilization but the average in all the remaining metrics.

Table 5.2 shows the overhead for the computation of an entire job scheduling. The overhead of DARDIS is subdivided in scheduling and dispatching

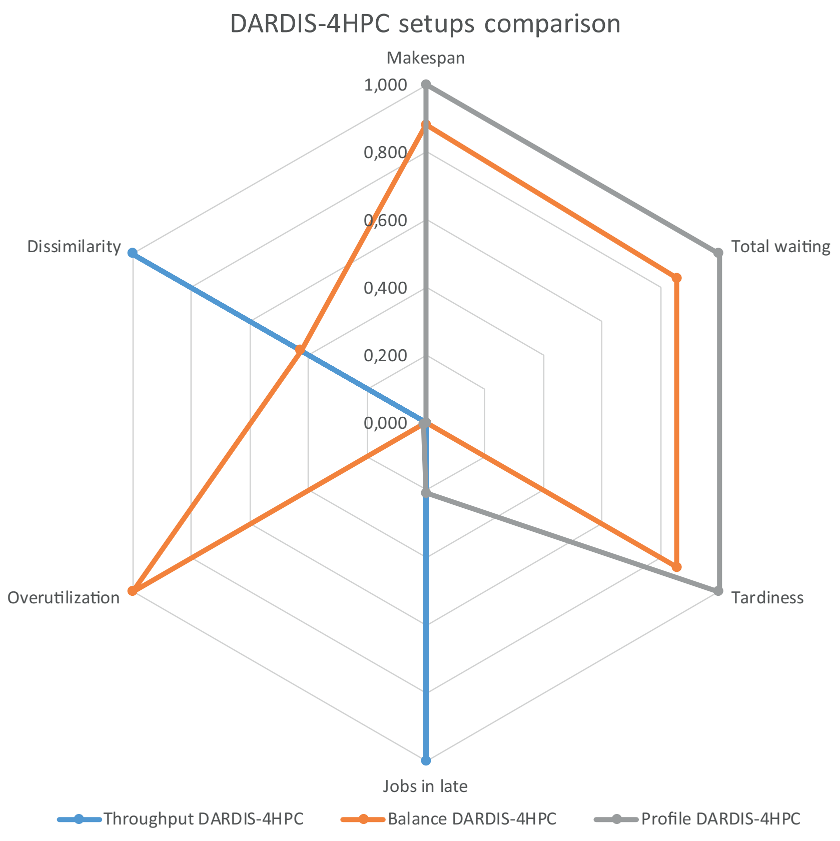


FIGURE 5.3: DARDIS results comparison

	Mean	Std. dev.
DARDIS Scheduling	0,018	0,129
DARDIS Dispatching	0,006	0,025
Rule-based total	5,356	5,077

TABLE 5.2: Overhead comparison of DARDIS and rule-based scheduler in seconds

while for the rule-based we have only total overhead of a scheduling cycle. From the table we can see that the most of the overhead of DARDIS is for the scheduling. Comparing the sum of scheduling and dispatching overhead of our approach to the rule-based scheduler we can evince that our approach is 214 times faster.

5.6 Conclusion

In conclusion, we presented a new scheduling approach for large scale HPC machines where the number of nodes and the number of jobs make a centralized approach infeasible. This approach is highly customizable in order to cover several behaviors for several domains. The approach is highly scalable due to its distributed nature. We evaluate three different setups of the approach. We have shown that the approach could obtain better result w.r.t. three different ad-hoc version of rule-based schedulers thanks to its distributed and probabilistic nature. Moreover, the possibility to specify a variable profile of desirable utilization increases the possibility of customization.

The test shows impressive improvements in Makespan, Total waiting, Number of late jobs and Tardiness metrics and good result in Dissimilarity. For the Overutilization metrics the rule-based scheduler obtains better results but at the expenses of number of late jobs. Moreover, these result have been obtained with a substantially lower computational overhead.

Future work will explore several directions. We will introduce new techniques to reduce the overhead, as for example a time-out for the job dispatching. Finally, we will introduce the possibility for the jobs to specify a variable profile of resource requirement. On the other side we will evaluate techniques for the creation of optimal variable/desired profiles to minimize computing center expenses in cooling and energy consumption. Other future work will evaluate the behavior of the scheduler within the introduction of running times prediction techniques.

Chapter 6

Optimal Profit-driven offline scheduling with cooling optimization

scheduling problems are common in a variety of fields, like manufacturing [151], fashion industry [152], wireless sensors networks [153], smart grids [154], etc. Scheduling optimization is a field studied since decades [155]. However, due to the continuous evolution of the application fields, the emerging of new problems, and new applications, it is still a very active research field [156, 157, 158]. Recent studies also have investigated scheduling optimization in HPC (e.g. [78]): in particular, a lot of work has been done in HPC scheduling optimization with Constraint Programming (CP) [1, 159, 81, 3]. The strength of CP for scheduling problems has been widely demonstrated [111, 112]. The benefits of CP are not only restricted to solution quality but also to the modeling flexibility that is often a desirable property in modeling complex scheduling problems.

Despite these benefits, the main limitation of CP for HPC scheduling is scalability, as indicated in [3], due to the NP-hardness of the problem [42, 43, 44]. CP approaches require significantly more computation time compared to rule-based schedulers, which typically produce lower-quality solutions. This work aims at addressing this trade-off by coupling these two approaches for a HPC scheduling problem with a complex objective function that takes into account (1) the profit obtained from the job executions; (2) the cost for running the workload; and (3) the cooling cost. To increase performance, we propose a heuristic to obtain a good starting solution and three different search strategies to improve the initial solutions.

To further increase scalability, our approach is designed for an off-line use. In particular, we produce schedules of job batches (usually the last 24 hours) with the goal to compute an optimized utilization profile. The obtained profile can then be used as a time-varying power cap, or in conjunction with profile aware and distributed job schedulers (e.g. [4]).

We evaluated our optimal scheduler under a wide variety of conditions, by different external temperatures, cooling models, workloads, and different pricing models. The proposed search strategies are compared with commercial rule-based schedulers and a commercial CP solver: ILOG CP Optimizer [132]. Moreover, we present an analysis of the efficiency and the utilization of the system in all the different scenarios. Finally, since many HPC centers are constrained to use commercial software for scheduling, we provide a simple decision-tree to aid system administrators to choose the most profitable scheduling approach (to apply to a commercial scheduler) on a day-by-day basis of the average system utilization of the previous day (i.e. a 24-hours rolling horizon strategy).

Results show that our approaches produce an improvement on the profit in the range 7.5-8% w.r.t. rule-based scheduling in the case of high workload and comparable results in case of low workload. In addition, our approach provides shorter schedules and therefore higher system utilization. A in-depth analysis of the PUE shows that our approach tends to increase the system utilization when needed and decrease it (to optimize the cooling) when the workload decreases.

The chapter is organized as follows. In Section 6.1 we show how this offline scheduler will be used into the scheduling workflow to obtain an online scheduler. Section 6.2 formally explains the job scheduling and dispatching problem. In Section 6.3 we recap the basics of classical commercial schedulers and present the setup used as a baseline for the experimental evaluation. Section 6.4 discuss the default ILOG CP Optimizer default scheduling search. Section 6.5 briefly introduces the CP paradigm and shows our optimization model. In Section 6.6 we explain the search heuristic that we use to obtain an initial solution. In Section 6.7 we present the multiple search heuristics that we employ to reduce the cooling cost (in terms of consumed energy) of the initial solution. In Section 6.8 we present our experimental results. Finally, Section 6.9 contains some concluding remarks.

6.1 Workflow

Figure 6.1 proposes an overview of the workflow of our scheduling approach. The users submit jobs to the DARDIS scheduler [4]. This scheduler distributes starting times computation through the nodes. Each node randomly selects a set of candidate starting times based on a desirable utilization profile. After that, the job selects the nodes for the execution and the starting time based on the candidate starting times. The job traces are stored and, after a fixed amount of time (e.g. each day), sent to our Profit-driven CP scheduler. This scheduler computes an offline, near-optimal, solution optimizing the schedule of the entire previous day. The solution is then processed to extract a desirable utilization profile, to be used by the DARDIS scheduler for optimizing the profit for the next day.

6.2 Scheduling problem

The HPC job scheduling and dispatching problem can be modeled as nonpreemptive scheduling problem with cumulative resources. Informally speaking, a solution assigns a set of nodes and a start time to each submitted job and no resource capacity is exceeded.

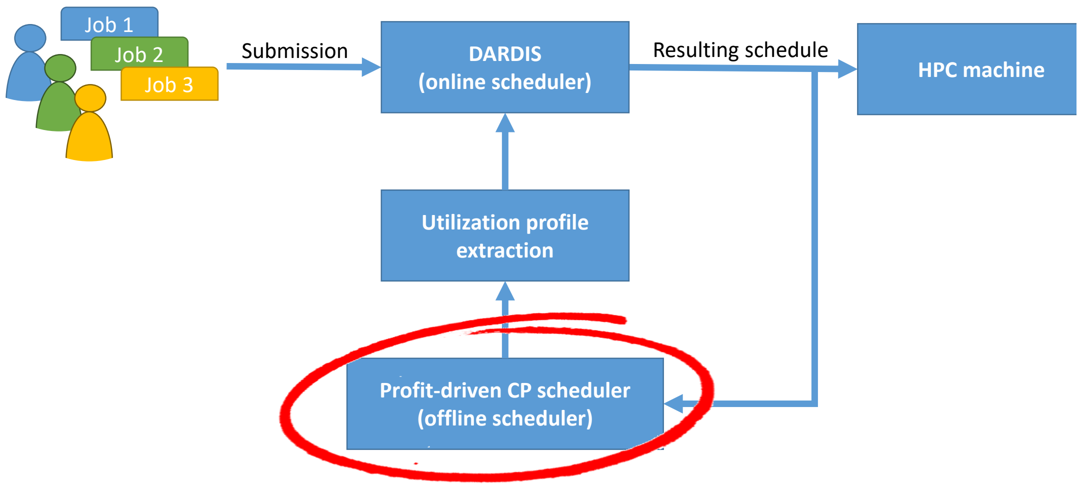

Job 1
Job 2
Job 3
Submission
DARDIS (online scheduler)
Resulting schedule
HPC machine
Utilization profile extraction
Profit-driven CP scheduler (offline scheduler)

FIGURE 6.1: Workflow of the interaction between the proposed scheduler and the DARDIS scheduler

More formally, the problem can be defined as follows. We are given a set RK of resource types (e.g., cores, GPU, memory, power budget), a set N of nodes, and a set A of jobs. Each node $n \in N$ has a capacity $c_{n,k}$ for each type of resource $k \in RK$. Each job $i \in A$ is composed of a set of job units UN_i. Each job unit of a given job has the same start time and duration (i.e., the job units must be synchronized). Each job i is submitted to the system at a time instant q_i, together with a specification of its walltime wt_i and the amount of resource it requires in *each job unit* $req_{i,k}$ for each type k of resource.

The job scheduling problem consists of selecting a start time st_i for each job $i \in A$ and a node $sn_{i,w}$ for each unit $w \in UN_i$ of the job such that:

$$\begin{aligned}
&st_i \in [q_i, .., H] \ (i \in A) \\
&sn_{i,w} \in N \ (i \in A, w \in UN_i) \\
&\sum_{i \in R(t,n)} req_{i,k} \leq c_{n,k} \ (t \in [0, .., H], n \in N, k \in K)
\end{aligned} \tag{6.1}$$

where H is the scheduling horizon and

$$R(t, n) = \{i \in A | st_i \leq t \wedge st_i + wt_i > t \wedge sn_{i,w} = n\}$$

is the set of jobs executing at time t on node n.

6.3 Rule-based scheduling

Rule-based scheduling is a widely used approach in HPC job scheduling. A rule-based scheduler processes jobs and nodes in a given order which is specified via a customized rule. When a job is processed, the scheduler considers each job unit and starts querying the system nodes to find a sufficient amount of free resources. When all job units have a candidate node, the job is started. If a job cannot be immediately started, two alternative behaviors are possible:

- *Strict ordering*: This is the most priority conservative approach. If job i cannot be started, the scheduler stops and waits for the next termination event to restart the process from i.
- *Non-strict ordering*: This is the most utilization aggressive approach. If job i cannot be executed, the scheduler skips it and tries to schedule the next job.

In the experimental results section (Section 6.8) we will compare the proposed approach against both a strict-ordering and a non-strict-ordering scheduler, using four different rules to order jobs. The following rules will be considered:

- *EST*: This rule tries to minimize the resource fragmentation. The set of queued jobs is ordered by increasing Earliest Start Time.
- *WT*: This rules is designed to increase the job throuput. The set of queued jobs is ordered by increasing Walltime.
- *Profit*: This rule has as goal the same metric used in our optimization model: the profit. The set of queued jobs is ordered by profit gained from the job execution divided by the cost of the job in decreasing order.

6.4 ILOG CP Optmizier default search

The default ILOG search uses a variation of a Large Neighborhood Search (LNS) [160] called Self-Adaptive LNS [129]. This approach is very similar to the approaches proposed in section 6.7. However, the default search into the neighborhood is a classical "Schedule or postpone" [161]. This search, when finds a job with a not feasible minimum start time, the job is no more considered until a propagation changes its minimum start time. This approach differs from our (Section 6.6) due to the fact that we preserve the job scheduling order.

6.5 Profit-driven CP scheduler

For modeling and solving the job scheduling problem, we use the Constraint Programming paradigm. CP is a declarative programming paradigm in which the user can formulate a model, which is then fed to a solver that explores the space of possible solutions to find the best one (according to a given objective function).

This two-step process is similar to that of Mixed Integer Linear Programming (MILP). However, unlike in MILP, in CP a user is not forced to employ only linear constraints: instead, a model can be formulated using any constraint from a given (solver-dependent) library. These constraints have a semantic (i.e. they enforce certain properties on the solutions), and they are associated to one or more *filtering algorithms*. At search time, the solver interleaves branching decisions with invocations of the filtering algorithms, which examine the domains of the problem variables and remove values that are provably infeasible: by doing so, they enable (possibly dramatic) reductions of the search space. The CP research community has developed specific constraints (and filtering algorithms) for scheduling, which usually allow a CP solver to outperform a MILP one on this class of problems [111, 112].

This section presents the model, which is implemented in CP Optimizers.

6.5.1 The Model Variables

This model consists of a set of interval variables a_i, each representing job i. Each such interval variable encapsulates four integer decision variables: $a_i.st$ denotes the job start time, $a_i.et$ the end time, $a_i.d$ the job duration, and $a_i.p$ whether the variable is "present", optional or "absent". To keep track of the job units assigned to each node, we use a set of interval variables $ju_{i,w,n}$, which represents the execution of the w-th unit of job i on node n. With $w \in [1, .., min(MU_{i,n}, |UN_i|)]$, where $MU_{i,n}$ is the maximum number of different job units of the i-th job that can execute simultaneously on the node n having the node completely free. The a_i variables are set to "present", the $ju_{i,w,n}$ are "optional" and capture the possibility of the unit executing on a node.

6.5.2 The Constraints

The constraints of the CP model are given in Equation 6.2:

$$\begin{aligned}
&a_i.setDuration(d_i)\ \forall i \in A\\
&a_i.setStartMin(q_i)\ \forall i \in A\\
&a_i.setStartMax(H)\ \forall i \in A\\
&\text{Cumulative}(ju_{(:,:,n)}, req_{(:,k)}, c_{n,k})\ \forall n \in N, \forall k \in RK\\
&\text{Alternative}(a_i, ju_{(i,:,:)}, |UN_i|)\ \forall i \in A
\end{aligned} \tag{6.2}$$

Since this is an offline scheduler, each activity a_i has a fixed duration d_i that corresponds to the duration obtained from logs. The notation $ju_{(i,:,:)}$ represents the set of all (optional) interval variables associated with job i. Similarly, $ju_{(:,:,n)}$ represents the set of (optional) intervals associated with node n. Finally, $req_{(:,k)}$ represents the set of the requirements of all job units for the resource type k.

The ALTERNATIVE constraints ensure that exactly $|UN_i|$ intervals associated with job i are present. In other words, just the $|UN_i|$ units of the job must be assigned to a node. Moreover, the ALTERNATIVE constraints synchronize a job and its units: They have the same starting and ending times. The CUMULATIVE constraints ensure that the job units assigned to node n do not exceed the node capacity for each resources type k at any given time instant.

6.5.3 The Objective Function

The definition of the objective function of our model relies on a set S of *time segments*. A time segment s is a portion of the day with an approximately uniform temperature (i.e., within an interval of 5 degrees Celsius). A segment s is defined by a start time $s.st$, an end time $s.et$, and a temperature $s.t$. We chosen the 5 degree tollerance as it is the discretization step in the cooling model [99] we use for estimating the cooling cost for a given operating condition. Equation 6.3 specifies the objective function, which maximizes the total profit during each time segment. For simplicity we define s_{first} as the first segment and s_{last} the last segment in S.

$$Maximize\Big(G * K_1 - W * K_2 - J * K_2\Big)$$

$$\begin{aligned}
G = &\sum_{i \in A} \big(overlapLength(a_i, s_{first}.st, s_{last}.et) * \\
&resourceCost(ju_i, res_{i,:})\big) \\
W = &\sum_{i \in A} overlapLength(a_i, s_{first}.st, s_{last}.et) * power_i \\
W_s = &\sum_{i \in A} overlapLength(a_i, s.st, s.et) * power_i \\
J = &\sum_{s \in S} W_s * LUT(W_s, s)
\end{aligned} \tag{6.3}$$

We now explain all the terms of this objective function. G is the profit obtained by all the jobs executed during the time segments in S. The expression $overlapLength(a_i, o, p)$ returns the amount of time the job a_i has spent executing during the interval $[o, .., p]$, with $o < p$.

$resourceCost(ju_{i,w,n}, res_{i,:})$ is an expression that returns the computing-center gain of the resources used by each job unit ju_i. This accounting approach is used by most HPC centers (e.g. [162]) to compute the amount of money spent by a user. The term $power_i$ represents the average power consumption measured by the system for job i. As indicated in [163] approximating the variable power consumption profile to the average power consumption when the whole machine is taken into account, leads to accurate results. Based on this value, we can compute W as the expenses of energy for the workload (i.e. to execute the jobs) during segment s. W_s contains the expenses for the workload during the segment s. The computation of W have been done without the use of W_s for better performance in the constraint propagation, however W_s is necessary to compute the cooling expenses J.

The expression $LUT(W_s, s)$ (Equation 6.4) returns the cooling efficiency for the workload of segment s, given the workload energy W_s of the segment, and the external temperature $s.t$. This expression is based on the PUE table from [99] and an example can be found in Table 6.1. This table contains the piecewise linear function of the PUE varying the IT power consumption and external temperature. With $PUE[s.t]$ we select the row that models the PUE

Temp \ Workload	55000 Watt	55500 Watt
10	1.1247272727	1.1247567568
15	1.4153090909	1.4153153153

TABLE 6.1: Example of PUE table

at the environment temperature $s.t$. The ELEMENT constraint ensures that when P'_s takes the value i the value of the expression is equal to $PUE[s.t][i]$. We use the ELEMENT constraint to relate the workload power P'_s, the external temperature $s.t$, and the efficiency of the cooling system. With $LUT(W_s, s)$, we compute J_s as the efficiency of the cooling system times the amount of energy for the workload during the time segment s. The equation then becomes:

$$\begin{aligned} P'_s &= \frac{W_s}{s.et - s.st} \\ LUT(W_s, s) &= Element(P'_s, PUE[s.t]) - 1 \end{aligned} \tag{6.4}$$

Given constants K_1 (amount of money obtained per volume of job utilization) and K_2 (cost of the energy) are known, Equation 6.3 captures that the CP model maximizes the profit as $G * K_1 - W * K_2 - J * K_2$. Note that K_1 and K_2 are fixed as happens in most computing-centers [164, 165, 166].

6.6 Heuristic for the first solution

The overall approach is implemented using *ILOG CP optimizer*, modeling jobs and job units as interval variables. As described above, an interval variable corresponds to a set of different decision variables (e.g., a start time, duration, presence, etc.). Each decision variable has a domain, defined by a lower bound lb and an upper bound[*] ub.

The proposed heuristic, iterates through jobs to find a minimum starting time. It differs from the rule-based schedulers which iterates through time trying to schedule each job. The pseudo-code is shown in Algorithm 5. The method is presented as iterative but actually it is implemented using the ILOGOAL construct provided by CP Optimizer. The algorithm selects the first job that has yet to be fixed (line 3). It then checks whether the minimum start time of the job is feasible (line 4). If it is a feasible start time, it is assigned (line 5). Otherwise, the heuristic increases the lower bound by one instant (line 7). Whenever a variable is assigned, the solver performs the constraint propagation step (line 9).

[*]We integrate the notation used above to access the bounds of each variable in a interval (e.g., the lower bound of the start time for the interval a will be $a.st.lb$)

Algorithm 5 Heuristic(J = ordered list of jobs)

```
1: while Some jobs have yet to be assigned a fixed start time do
2:     select the first job $i \in J$ with non-fixed $a_i.st$
3:     while $a_i.st.lb \neq a_i.st.ub$ do
4:         if $|UN_i|$ job units of $i$ can be started at $a_i.st.lb$ then
5:             $a_i.st.ub = a_i.st.lb$ (fix $a_i.st.lb$ as a start time)
6:         else
7:             $a_i.st.lb = a_i.st.lb + 1$
8:         end if
9:         (Constraint propagation, handled by the solver)
10:    end while
11: end while
```

Algorithm 6 The Multi-Search Approach

```
1: foreach ordering $O$ in $Orderings$ do
2:     $J$ = jobs ordered by $O$
3:     Heuristic($J$)
4: end for
```

6.7 Searches

We now present three local search procedures to improve the solution of the job-based heuristic.

6.7.1 Multi-Search

The first search strategy executes the job-based heuristics for different job ordering and it returns the best obtained solution. The considered ordering criteria are: (1) Earliest start time, (2) Job duration, (3) Latest start time, (4) Number of job units, (5) Number of required resources per node, (6) Total number of required resources, (7) Total number of required resources multiplied by the job duration, (8) Average job power, (9) Average job power/job profit, (10) Average job power/job profit as main ordering, increasing duration in case of ties, (11) Average job power/job profit as main ordering, decreasing duration in case of ties. Each ordering criteria is considered both in increasing and decreasing order.

6.7.2 Relaxation-Based Search

The next strategy is called Relaxation-Search and its goal is to decrease the search space for scalability purposes. This is obtained by deciding the dispatching for the most difficult jobs first, i.e., those with several job units, which require stronger synchronization. This Relaxation-Search strategy starts with the Multi-Search and adds two kind of constraints stated in Equation 6.5). The first constraint forces the next solution to improve the current best solution. The second set of constraints forces the jobs that require more

Algorithm 7 Delay-based Search

```
1:  Multi-Search()
2:  selects the first jobToDelay
3:  while termination condition not reached do
4:      jobToDelay.setStartMin(jobToDelay.getStartMin() + delay)
5:      if jobToDelay reached H then
6:          alreadyDelayed.add(jobToDelay)
7:          selects new jobToDelay
8:      end if
9:      foreach job $ad$ in alreadyDelayed do
10:         ad.setStart(bestSolution.ad.st)
11:     end for
12:     Heuristic(J)
13:     update bestSolution and bestSolutionValue
14: end while
```

than a job unit to the dispatching fixed by the Multi-Search solution. After these two constraints are imposed, the default search of ILOG CP Optimizer is employed improving the solution exploring: (1) different dispatching for the jobs with just one job unit and (2) different schedules for every job.

$$
\begin{aligned}
& G * K_1 - W * K_2 - J * K_2 \geq bestSolValue \\
& ju_{(i,j,n)}.p = bestSol.ju_{(i,j,n)}.p \\
& \qquad \forall i \in B, \forall j \in [1, \ldots, |UN_i|], \forall n \in N \\
& \qquad \text{where: } B = \{i \in A | \; |UN_i| > 1\}
\end{aligned}
\tag{6.5}
$$

6.7.3 The Delay Search

This last search strategy is called Delay-Search and is presented in Algorithm 7. Its key intuition is to delay jobs scheduled by the Multi-Search to explore different parts of the search space. The procedure selects a job to delay (line 2) and imposes that the job cannot be scheduled before at least $delay$ time units from its previously assigned start time (line 4). If the job is delayed after the time horizon H, it is stored in the list of already delayed jobs and a new job is selected to be delayed (lines 5-8). The $alreadyDelayed$ jobs are fixed to the start time selected in the best solution found so far (lines 9-11). Then, the heuristic is called (line 12) and the best solution is updated (line 13). This approach is repeated until a termination condition is reached (line 3). In our case the termination condition is a time limit or the fact that all the jobs have been delayed to the time horizon H [†]. After the first solution-tree descent, the search considers to delay all the jobs starting from the last fixed job.

[†]Once again, for simplicity, we presented this search procedure as an iterative method. It is however implemented via the CP Optimizer *IloGoal*

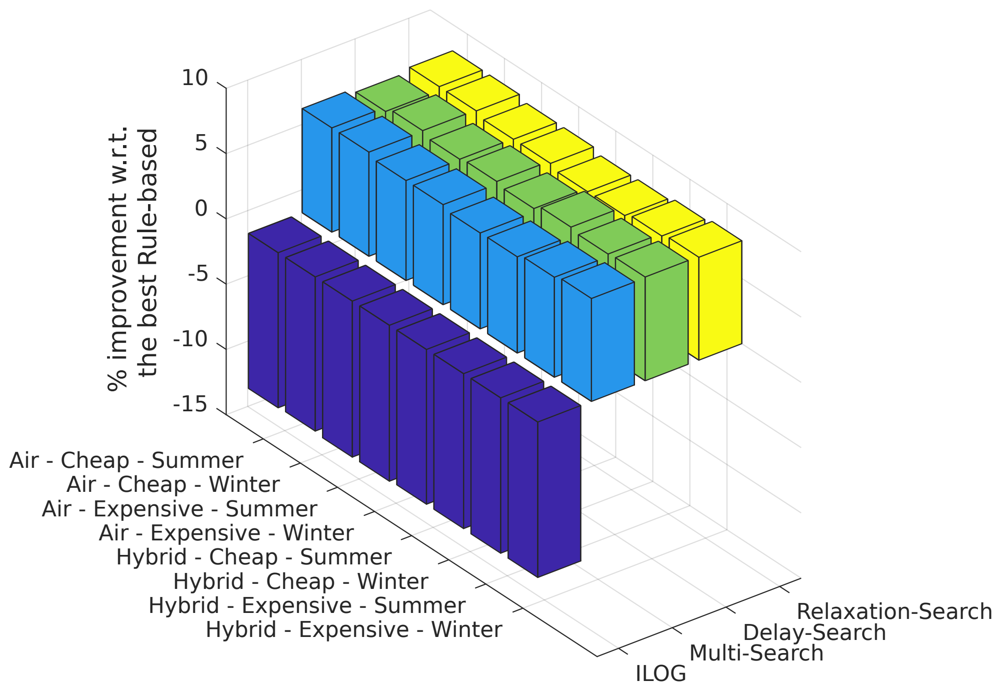


FIGURE 6.2: Percentage of improvement at submission end w.r.t. the best rule based scheduler for the High workload scenario

6.8 Results

6.8.1 The Test Case

The purpose of this evaluation is to investigate whether it is possible to improve the profit over commercial rule-based schedulers for large instances. The experiments have been performed on a number of different scenarios on a simulated HPC machine. The traces used for the job submission and resource request are based on the parallel workload archive of the CEA Curie system [150]. Two traces have been extracted:

1. **Low workload**: 8125 jobs submitted in 30 days;
2. **High workload**: 33583 jobs submitted in 23 days.

For each job a random power consumption in the range 7.8 to 11.11 W*core [167] has been generated to cover this missing information. The simulated HPC machine is composed by 300 nodes with 32 cores each and 128GB of RAM [167].

The tests have been executed in different scenarios to explore the behavior of the proposed solutions with different cooling systems:

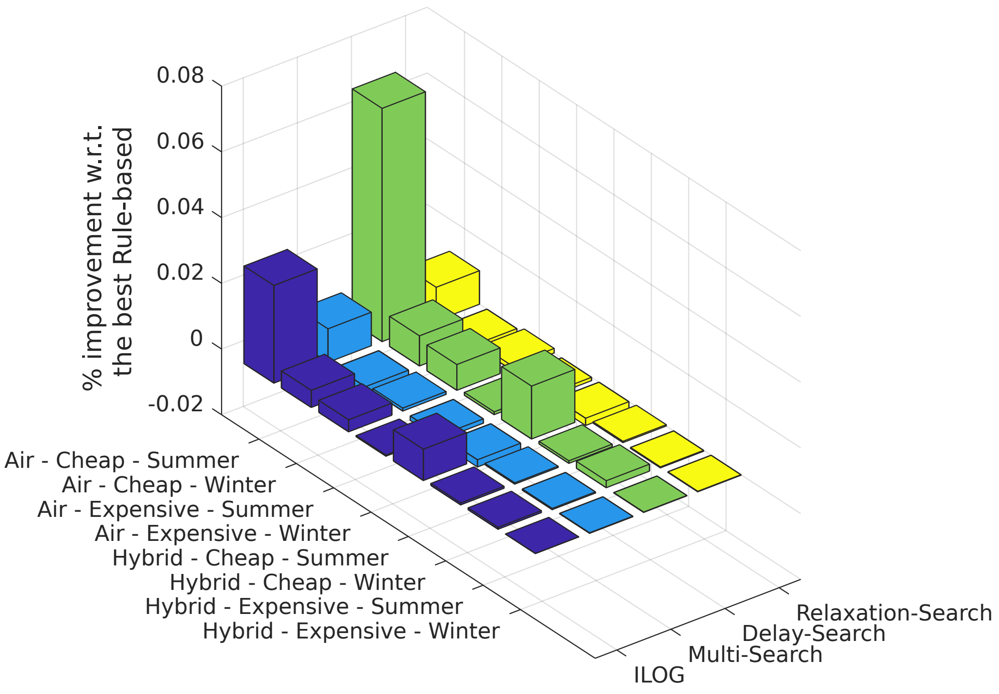


FIGURE 6.3: Percentage of improvement at submission end w.r.t. the best rule based scheduler for the Low workload scenario

TABLE 6.2: PUE efficiency in each scenario for ILOG, the proposed strategies, the best Rule-based scheduler, and Slope change

				ILOG	Multi-S.	Delay-S.	Relax.-S.	Best RB	Slope change
Air cool.	Summer	High	Cheap	1.43419	1.43407	1.43355	1.43406	1.43451	1.43497
			Expensive	1.43417	1.43401	1.43377	1.43402	1.43451	1.43497
		Low	Cheap	1.41417	1.41603	1.41022	1.41603	1.41701	1.43497
			Expensive	1.41397	1.41626	1.41049	1.41586	1.41701	1.43497
	Winter	High	Cheap	1.10795	1.11268	1.11254	1.11280	1.11295	1.11726
			Expensive	1.10784	1.11261	1.11253	1.11259	1.11295	1.11726
		Low	Cheap	1.09427	1.09530	1.09388	1.09514	1.09478	1.11726
			Expensive	1.09453	1.09590	1.09413	1.09560	1.09478	1.11726
Hyb. cool.	Summer	High	Cheap	1.08863	1.09222	1.09222	1.09221	1.09207	1.08727
			Expensive	1.08839	1.09218	1.09242	1.09218	1.09207	1.08727
		Low	Cheap	1.07419	1.07535	1.07353	1.07535	1.07513	1.08727
			Expensive	1.07464	1.07544	1.07345	1.07536	1.07513	1.08727
	Winter	High	Cheap	1.00306	1.00323	1.00319	1.00323	1.00317	1.00302
			Expensive	1.00306	1.00327	1.00326	1.00327	1.00317	1.00302
		Low	Cheap	1.00279	1.00280	1.00277	1.00280	1.00284	1.00302
			Expensive	1.00278	1.00290	1.00284	1.00289	1.00286	1.00302

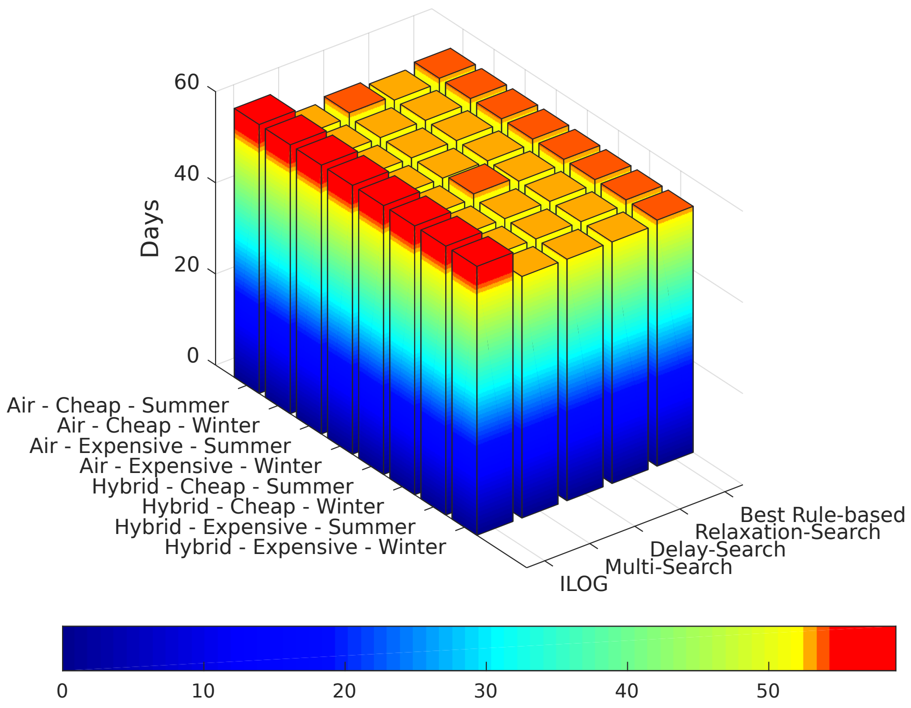


FIGURE 6.4: Tests makespan for the High workload scenario

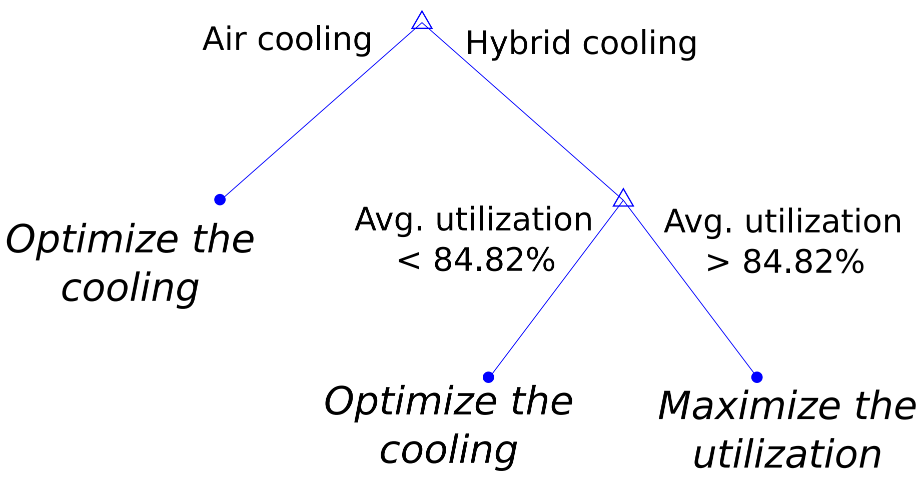


FIGURE 6.5: Classification tree for the efficiency

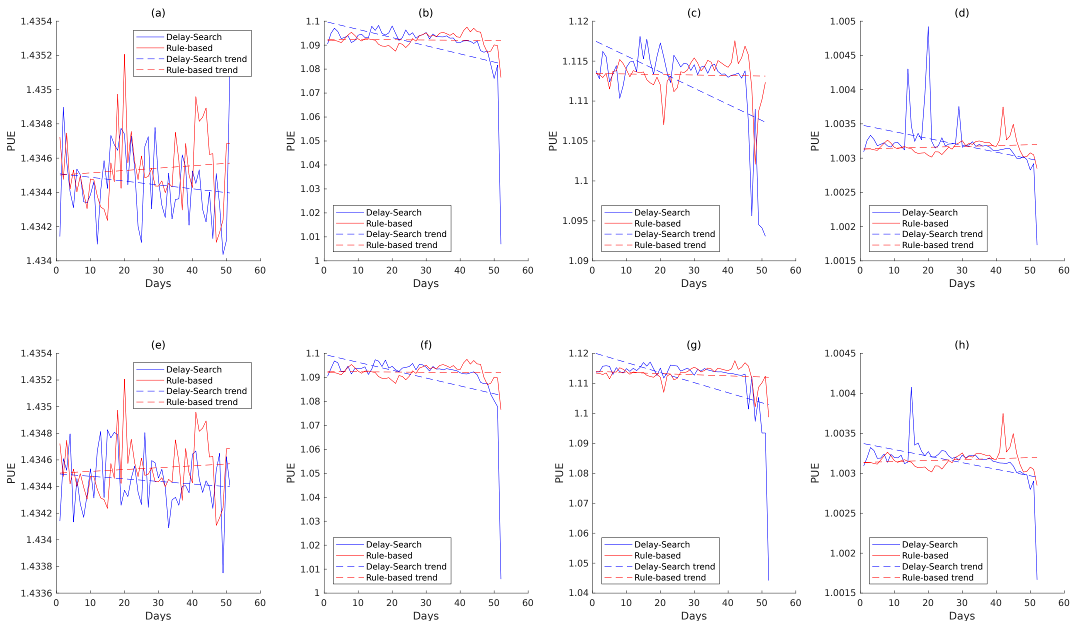


FIGURE 6.6: Daily PUE and trend for the Delay-Search and the best Rule-based in the High workload case for the scenarios: (a) Air - Expensive - Summer, (b) Hybrid - Expensive - Summer, (c) Air - Expensive - Winter, (d) Hybrid - Expensive - Winter, (e) Air - Cheap - Summer, (f) Hybrid - Cheap - Summer, (g) Air - Cheap - Winter, and (h) Hybrid - Cheap - Winter

1. An **Air cooling** with a Power Usage Effectiveness [‡] (PUE [168]) of ~ 1.4 (depending on the external temperature and the workload energy);
2. an **Hybrid cooling** system with a PUE of ~ 1.1;

different environment temperatures:

1. **Summer** temperatures;
2. **Winter** temperatures;

and different pricing level (K_1) resembling different hardware compositions:

1. **Cheap** pricing: 0.013 € * core * hours;
2. **Expensive** pricing: 0.10 € * core * hours.

The Energy pricing (K_2) is set to 0.13 €/KWh.

6.8.2 The Implementation

The implementation has been done in C++ using the *IBM ILOG CP Optimizer 12.7.0* solver. Tests have been executed on a 2x*Intel Xeon Processor E5-2670 v3* server with 128GB of RAM. Each model instance has been executed with a single worker (thread) with a time limit of 4000 seconds.

[‡]The Power Usage Effectiveness is a measure for datacenters efficiency. The calculation is $PUE = \frac{P_w + P_c}{P_w}$. Where P_c is the power spent in cooling and P_w is the power spent in workload

6.8.3 Profit comparison

This section analyzes the improvement on the profit in each scenario w.r.t. the rule-based scheduler. For space limitation we refer to "Rule-based scheduler" as the rule-based scheduler that in a given scenario obtained the **best** profit. Due to the high number of jobs submission in HPC machines, in general and in the selected profile, we analyze the profit improvement at the moment of the last submission. This is because an HPC has no jobs submission only in exceptional cases. Figure 6.2 shows the percentage of improvement w.r.t. the best rule-based scheduler, for ILOG and all the proposed search strategies, in the **high workload** case. While, Figure 6.3 shows the **low workload** case. As shown in these figures no significant improvement is obtained in the **low workload** scenario. Differently, in the **high workload** scenario, the improvement obtained by our search strategies is in the range 7.6%-8.1%. In particular, the Delay-based strategy has an average improvement 0.15% points higher w.r.t. the Multi-Search and the Relaxation-Search. The default search implemented in ILOG does not improve the results obtained by the rule-based scheduler and obtains, in fact, a worsening in the range 11.8-11.9%.

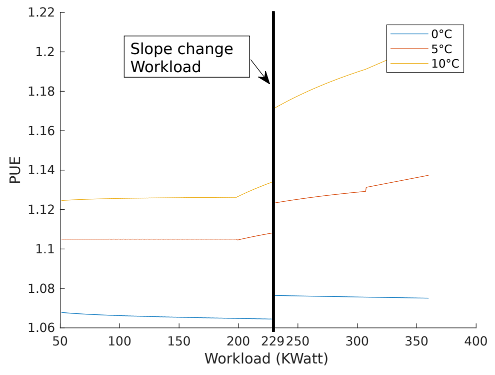
A further analysis on the profit shows that this increment is due to three factors: (1) the different set of jobs scheduled in the interval until the last submission, (2) an increment on the utilization on the days in which a higher number of jobs were run in the system, and (3) an improvement on the cooling in the days with a low system utilization.

Indeed, Figure 6.4 shows that, in general, the proposed search strategies have a lower makespan w.r.t. both the best rule-based scheduler and ILOG in the **high workload** scenario. The indirect relation between the makespan and the utilization proves that our strategies create a higher system utilization w.r.t. the classical rule-based schedulers. For the **low workload** scenario, no differences on the makespan are present.

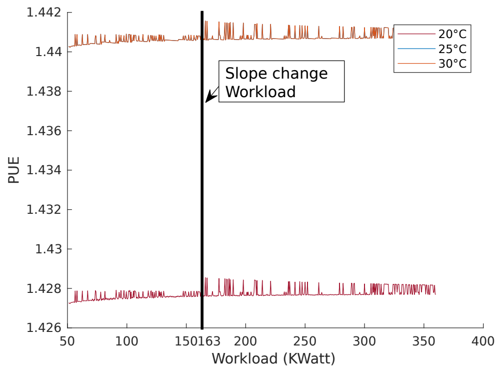
Table 6.2 shows the total PUE. Comparing the Delay-Search w.r.t. the best Rule-based we can see that our strategy obtains a higher PUE only in the **Hybrid cooling** scenario just in the case of **High workload** leading to, in general, a more efficient and greener system.

Since the Delay-Search is the best of our proposed search strategy, Figure 6.6 shows the PUE and the PUE trend line for the Delay-Search and the best rule based in each scenario of the **High workload** case. The trend line show the linear regressions of respectively the Delay-Search and Rule-based. From the trend line we can note that the Delay-based strategy has a higher PUE in the days in which the number of jobs in the system is higher and then it decreases within the decrement of the jobs in the system. This is not true for the Rule-based scheduler that has a uniform or, in some cases, increasing trend. This suggests that our strategy can decrease the system efficiency when is unneeded and not profitable. Note that in the **High workload** tests just the days from 1 to 23 have jobs submissions. The remaining days just the queued jobs are processed.

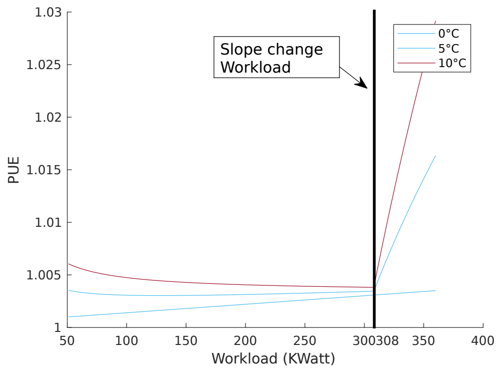
Given the previous intuition on the behavior of our strategies, we propose a second contribution with the analysis of the near-optimal solutions



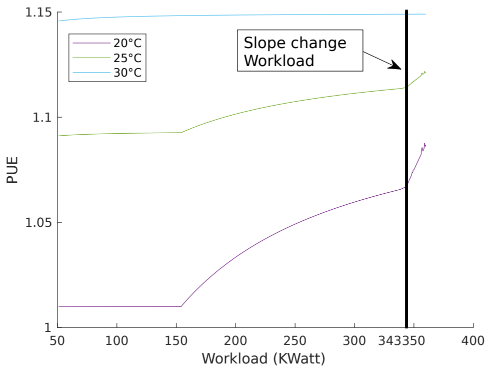
(A)



(B)



(C)



(D)

FIGURE 6.7: PUE at different external temperatures and Slope change workload in: (a) winter - air cooling, (b) summer - air cooling, (c) winter - hybrid cooling, and (d) summer - hybrid cooling from the results by [99]

obtained by our scheduler. The scope of this analysis is to provide a simple guide to decide which scheduling policy (e.g. between a policy that anticipate heavy jobs and decreases the system utilization or a policy that increases the utilization) is more profitable for the next day, given the average system utilization of the previous day. Indeed, in some cases, in HPC it is not possible to use non-commercial schedulers as the one proposed in this work.

This analysis discriminates the efficiency on the basis of the "Slope change" column in table 6.2. The column contains the efficiency point after which the PUE increases according to the data in [99]. This happens when the chillers need to be activated as effect of an higher IT power load or an increase in the ambient temperature. Note that the Slope change depends on the cooling type and external temperature. To obtain the Slope change point, we started from the data in [99], as shown in figure 6.7 we found the workload in which the PUE has the higher increment in each scenario. Given the Slope change workload and the daily external temperature traces we compute the slope change point as the average PUE weighted in time. This point is calculated daily because is unlikely for a computing center to change the scheduler setting more than one time per day. Analysis have been realized through a decision-tree set to classify, starting from the cooling type and the average system utilization of the previous day, which efficiency zone is more profitable for the next day. The efficiency zones are two:

1. "Optimize the cooling": In this zone is more profitable to slightly decrease the system utilization and save cooling costs;

2. "Maximize the utilization": In this zone the best result is obtained increasing the utilization as high as possible.

The dataset is realized using the near-optimal solutions obtained by the DelaySearche for a total of 538 items. As shown in figure 6.5 for an Air cooling based HPC system it is never convenient to increase the system utilization as high as to overpass the Slope change point. For a Hybrid cooling based HPC system, instead, it is more profitable to maximize the system utilization only when the average system utilization of the previous day was higher than 84.82%.

Figure 6.8 shows the prediction results, red circles are exact prediction for the "Maximize the utilization" class; blue circles are exact prediction for the "Optimize the cooling" class. Red "x" are wrong prediction for the class "Optimize the cooling", while blue "x" are wrong prediction for the "Maximize the utilization". On the x-axis the average percentage system utilization of the previous day while on the y-axis the distance from the Slope change point. From the figure it is possible to note that: (1) the number of wrong prediction is small (less than 4%), and (2) the errors are limited in a small neighborhood of the Slope change point. Note that this figure contains the prediction for both air cooling and hybrid cooling. For this reason at high system utilization we have both red and blue circles.

The accuracy of the system with this simple classification tree is 96.28%. However, figure 6.9 shows the same information as the previous figure but just with the prediction errors. From the plot it is possible to note that just six errors are in the range [0.001 - 0.01] while all the other errors are really near to the slope change point (range [-0.00002 - 0.001]). This suggests that errors appear just when optimizing the cooling or maximizing the utilization lead to similar results.

6.9 Conclusion

In conclusion we designed a CP model for the problem of jobs scheduling and dispatching in HPC machines that maximizes the profit. The model takes into account the gain obtained by the jobs, the money spent for the workload and the money spent for the cooling. The cooling model has been integrated in our objective function thanks to the results obtained by Conficoni et al. [99]. We have designed several different search strategies and we showed the results obtained by the basic search and the best two search strategies. We tested the model using real workload traces obtained by the parallel workload archive and simulating a real HPC with two different cooling systems, different environmental temperatures, and job pricing.

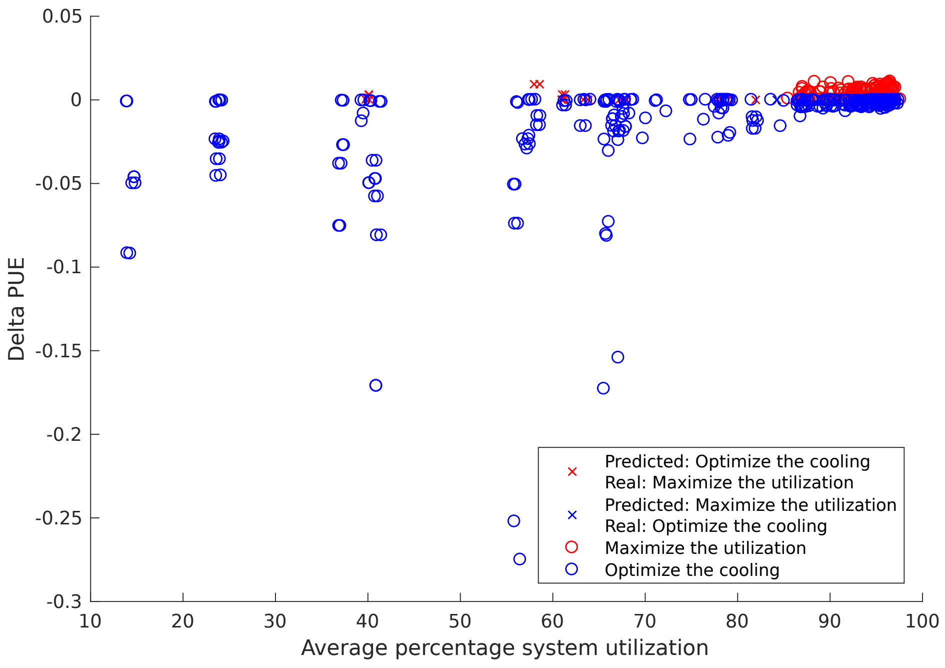


FIGURE 6.8: Distance from Slope change points at different system utilization percentage with efficiency prediction

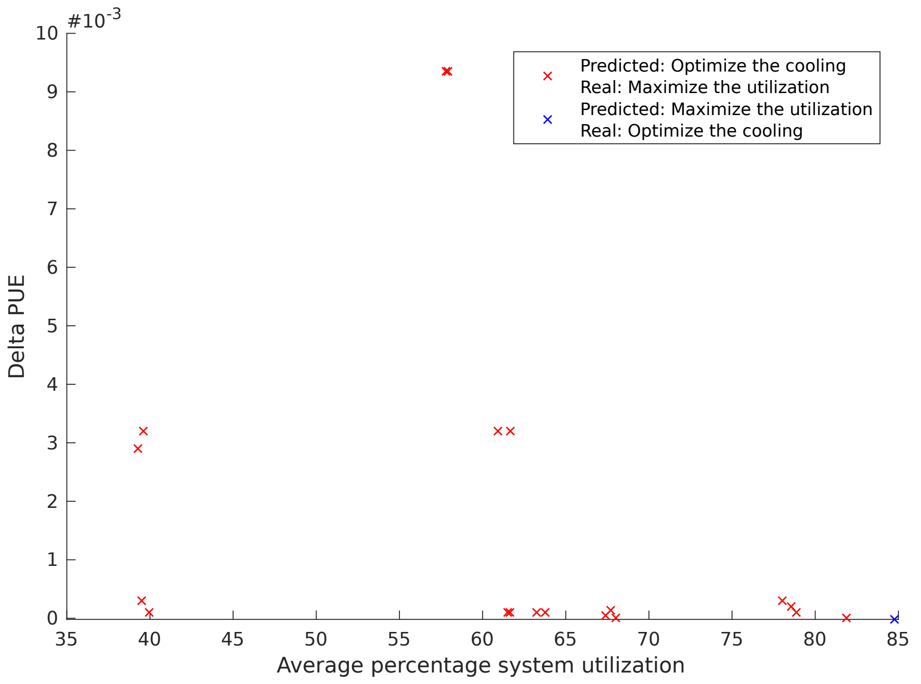


FIGURE 6.9: Efficiency prediction errors

The implemented search strategies showed to outperform the solutions obtained by the ILOG CP Optimizer solver on makespan and profit metrics in the majority of the cases, while no worsening is present in the remaining cases. Results show that our approach can improve the profit by 7-8% in the high workload scenario w.r.t. rule-based scheduler but also can decrease the makespan. The makespan decreasing can lead to a further 2% of profit improvement. Moreover, in the case of non optimal cooling systems, the total efficiency of the system can be improved. The analysis on the results suggest that this scheduler increases the system utilization when is needed while decreases the utilization to improve the cooling when is more profitable.

Finally, through the analysis of the PUE we provide a simple decision tree for system administrator to improve the profit just changing the scheduling policy of commercial schedulers.

However, this scheduler has been designed for offline scenarios. Future work will study the cooperation of this scheduler within an online and reactive approach with the aim to obtain better solutions w.r.t. a pure reactive approach, with lower response time w.r.t. an optimization approach and the introduction of job traces predictors instead of past traces.

Chapter 7

Hybrid Offline-Optimized and Online-Distributed Profit-driven low-overhead scheduler for HPC with automatic node shut-down and turn-on

High-performance computing (HPC) machines are highly expensive systems [169] with fast depreciation [33]. HPC facilities have to deal with high fixed costs but also with high variable costs (e.g. the energy consumption). Many works in the last years targeted the power consumption limitation of these systems [101, 102, 103]. However, to the best of our knowledge, there are no studies that relate the power consumption increasing to the profit increment. In fact, limiting the resources or the computational power of a supercomputing system decreases the expenses but, usually, also jeopardizes the incomes [107]. On the other hand, increasing too much the system utilization brings the system to an more inefficient work condition. The best result in terms of profit is to maximize the system utilization (and consequently the power consumption) when is needed and when the external factors are in favor and then limit the power consumption when the whole power is unnecessary.

The job scheduler seems the right component in which these power management mechanisms could be introduced. In fact, the scheduler knows the workload submitted to the system. However, in future supercomputers, the number of resources that the scheduler has to manage will increase while users want the same responsiveness of the system. Thus, scalability and low overhead are a major issue in these systems. However, the scheduling architecture used so far has no possibility to obtain scheduling solution that increase the profit considering the cooling system, the workload fluctuations, and the external temperatures while increasing the scalability of the scheduler and decreasing the overhead thus the scheduling architecture have to be rethought. In this work, we propose a new scheduling architecture. This architecture is composed by different solutions to improve the scheduling in terms of HPC machine profit, cooling expenses, scalability, and overhead. This complex task is realized within different approaches and modules:

- An Offline and optimized scheduler: This scheduler uses an optimization technique called Constraint Programming (CP) to schedule 24 hours of submitted jobs to optimize the profit taking into account the HPC model, the submitted jobs, *the cooling model*, and *the weather forecast*.

- A profile extractor: This module exploits machine learning techniques to extract a sub-optimal "desirable utilization profile" for the HPC resources to fed to the online scheduler.

- A Online, Distributed, and Profile Aware scheduler: This distributed scheduler is designed to minimize the overhead and improve the scalability w.r.t. the commercial scheduling approaches. The scheduler is also designed to follow a "desirable utilization profile" planning the scheduling of the jobs in the future.

- A Job statistics and forecast module: this module generates a forcasted synthetic jobs trace for the next 24 hours to fed to the offline scheduler.

We compare the proposed solution against commercial rule-based schedulers in a number of different scenarios. The studied scenario differs by cooling model and external temperatures. The workload traces are real trace from the Parallel workload archive [150]. In particular, three traces have been extracted to characterize different workload conditions: low (3207 jobs), medium (8125 jobs), and high (10892 jobs). Finally, the simulated HPC is a real HPC composed of 9600 cores.

The proposed scheduler is shown to improve the profit by an 8.6% in average in a realistic scenario. The results showed also the better scalability of our approach. Finally, tests on the makespan show that no significant worsening on the makespan has been found under low and average workload conditions. However, a worsening of 10% is found under high workload conditions.

This work is organized as follows. In Section 7.1 we formally describe the HPC scheduling problem. Section 7.2 shows the workflow of our proposed architecture. In section 7.3 we describe the offline and optimized scheduling approach used to optimize the profit of the HPC. Section 7.4 shows the profile extraction algorithm implemented. In section 7.5 we describe the online and distributed scheduling approach used in our architecture. In section 7.6 we show the results on profit, makespan and computational overhead. Finally, section 7.7 shows our conclusions.

7.1 Scheduling problem

Job Scheduling-and-dispatching in HPC is a non-preemptive (batch) scheduling that consists of assigning starting time and node to each job chunk of a job submitted by the user. The main constraint is to never overutilize the resources e.g. a single core can not be assigned to more than a job at the same time.

More formally, the problem can be defined as follows. Given a set RK of resource types (e.g., cores, GPU, memory, power budget), a set N of nodes, and a set A of jobs. Each node $n \in N$ has a capacity $c_{n,k}$ for each type of resource $k \in RK$. Each job $i \in A$ is composed of a set of job units UN_i. Each job unit of a given job has the same start time and duration (i.e., the job units must be synchronized). Each job i is submitted to the system at a time instant q_i, together with a specification of its walltime wt_i and the amount of resources it requires in *each job unit* $req_{i,k}$ for each type k of resource.

The job scheduling problem consists of selecting a start time st_i for each job $i \in A$ and a node $sn_{i,w}$ for each unit $w \in UN_i$ of the job such that:

$$\begin{aligned} & st_i \in [q_i, .., H] \ (i \in A) \\ & sn_{i,w} \in N \ (i \in A, w \in UN_i) \\ & \sum_{i \in R(t,n)} req_{i,k} \leq c_{n,k} \ (t \in [0, .., H], n \in N, k \in K) \end{aligned} \tag{7.1}$$

where H is the scheduling horizon and

$$R(t,n) = \{i \in A | st_i \leq t \wedge st_i + wt_i > t \wedge sn_{i,w} = n\}$$

is the set of jobs executing at time t on node n.

7.2 Workflow

Our proposed scheduler is built upon three main components:

1. A jobs statistics and forecast module: This module collects data from the jobs submissions and the obtained scheduling solutions. From this data, it extracts statistics and implements machine-learning algorithms to forecast the future jobs submissions (submission time, resource required, walltime, duration, power consumption etc.)

2. A Profit-driven CP scheduler [6]: This offline scheduler is triggered at fixed time intervals (24 or 48 hours in our tests) and it is designed to model the entire HPC machine, the jobs submitted to the system, the HPC machine power consumption, the cooling model, the cooling power consumption, the external temperature, and the profit incoming from the machine. The scheduler sets the execution of the jobs to a near-optimal solution that maximizes the profit by moving jobs in cooler hours of the day or maximizing the system utilization on the basis of the saturation of the systems and the submission requests.

3. A profile extractor: based on the analysis on the results obtained by the Offline CP scheduler we have designed an algorithm to produce a forcasted "desirable utilization profile" that limit the system resource utilization to decrease the expenses when it is more profitable.

4. A Distributed Reactive scheduler (DARDIS [4]): this online scheduler designed for high scalability and low overhead takes as input the jobs

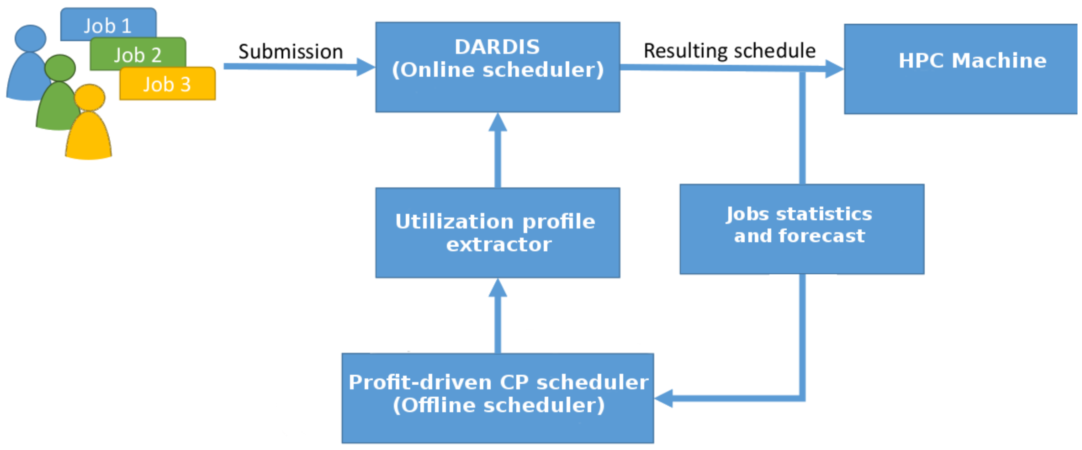

Job 1
Job 2
Job 3
Submission
DARDIS (Online scheduler)
Resulting schedule
HPC Machine
Utilization profile extractor
Jobs statistics and forecast
Profit-driven CP scheduler (Offline scheduler)

FIGURE 7.1: Workflow of the interaction between the Offline CP scheduler, Profile Extractor, and the online DARDIS scheduler

submission and the desirable utilization profile. The scheduler gives the possibility to turn-off single resources to decrease expenses. Moreover, due to the scheduler future planning, it is also possible to program the shut-down and the turn-on of entire nodes without creating any delay in the system.

These components are connected as in Figure 7.1. The workflow requires that the jobs are submitted to an online scheduler. After the scheduling decisions, the *Jobs statistics and forecast module* collects the jobs data and generates a forcasted job trace with the waiting jobs and the jobs forecast to be submitted in the near future (e.g. 24 or 48 hours). This trace is fed to the *Profit-driven CP scheduler*. The *Profit-driven CP scheduler* optimizes the jobs dispatching and scheduling in the near future aiming to the result with the best profit. This is achieved by maximizing the system utilization when the system is under a high workload request or moving job to the coolest hours of the day to decrease the cooling expenses when the jobs requests are lower. After that, the result of the offline CP scheduler is sent to the *profile extractor*. This module, exploits heuristic and regression tree algorithms to generate a profile designed to decrease the energy expenses of the system. In fact, the Offline scheduler suggests when it is more profitable not to use the entire system's resources while the *profile extractor* checks if it is convenient to shut-down and turn-on entire nodes. Finally, the generated desirable profile is then provided to the online scheduler. The scheduler plans the scheduling in the future, in this way, having a planned schedule and the desirable utilization profile it can shut-down and turn-on entire nodes without any delay caused by a late restart.

7.3 Offline CP scheduling

CP is a programming paradigm designed for optimization. This paradigm allows to model problems with complex constraint and it is specially suited for scheduling problems. This means that it provides many constraints designed for this kind of problems (e.g. the cumulative constraint [161]) and widely proved to overcome the performance of other optimization techniques, such as MIP, in scheduling problems [111, 112]. The optimization of CP relies on the concept of decisional variable. This variable has a domain of possible values initially feasible for a solution. A CP solver then, starting from the modeled problem, starts a search. During the search, the solver (1) selects the variable to assign and assigns a value from the feasible in its domain, (2) starts a constraint propagation to purge the domains of the remaining variables. When a solution of a fail is reached the algorithm backtracks to continue the exploration for better solutions.

Although the power of CP in scheduling problems, this approach presents scalability problems showed in [3]. The work shows that this approach is feasible for no more than an average submission day on a mid-tier HPC machine. For this reason many techniques has been adopted to improve the scalability and minimize the computational overhead of this approach. However, (as [6] shows) the scalability problem is the main reason why to apply an hybrid online-offline approach and do not use CP as online scheduler.

To easily model scheduling problem, the adopted CP solver (IBM ILOG CP Optimizer) provides a special variable called interval variable. An interval variable is a variable that models an activity (or job). This kind of variable contains several different decisional variables. For sake of simplicity, we can say that an interval variables a is composed of a decisional variable for its starting time $a.st$, a decisional variable for its duration $a.d$, a decisional variable for its end time $a.et$ and a decisional variable for its presence $a.p$. If an interval is present, all the constraints on this variable propagate otherwise this variable does not propagate.

Equation 7.2 explains the complete model [6]. The objective function is to maximize the profit. The profit is calculated by G the gain obtained by the jobs execution minus the energy expenses. The energy expenses are calculates as the energy spent for the HPC to execute the workload W plus the cooling energy J times the per-Joule rate of the energy provider. Each job a_i is constrained to have a duration $a_i.d$ equal to di the duration forcasted by the forecast module. The start time $a_i.st$ of the job has to be in the range $q_i - eoh$, respectively the forcasted submission time and the end of the scheduling horizon.

For each job we have a set of interval variables $ju_{(i,:,:)}$ of cardinality equal to the number of nodes of the system N times the number of job units $|UN_i|$ of the i-th job. This set of interval variables models the possibility of each job unit of the job to execute in each node of the system (also two different job units in the same node). However, to do that, we have to constraint the number of *present* job units to be equal to $|UN_i|$. This is done by the

	Summer		Winter	
	Air cooling	Hybrid cooling	Air cooling	Hybrid cooling
Low workload	29.50%	17.84%	7.37%	15.00%
Medium workload	10.46%	18.23%	13.19%	-7.50%
High workload	0.36%	0.20%	0.23%	0.01%

TABLE 7.1: DARDIS70% Vs. RB percentile profit improvement comparison

$Alternative$ constraint. This constraint sets the exactly $|UN_i|$ interval variables from $ju_{(i,:,:)}$ to be synchronized with a_i, all the remaining variables in the set are set to *absent*.

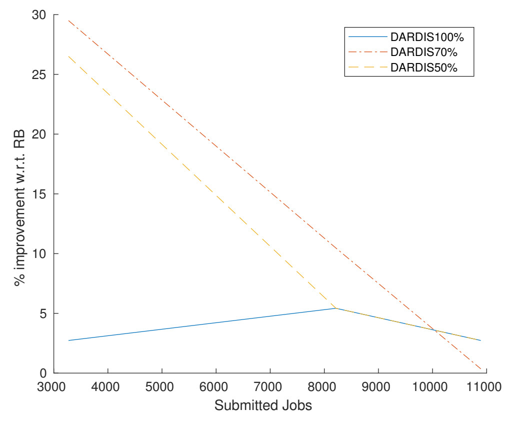
Finally, having a jobs resource requirement of $req_{(i,k)}$ for each job unit and a resource physical capacity of $c_{n,k}$ for the k resource type on the node n, the $Cumulative$ constraints the solver to set the starting times of each job to never overpass the resource limit utilization $c_{n,k}$ in any instant of time.

$$\begin{aligned} & Maximize\Big(G - (W + J) * K\Big) \\ \text{subject to :} & \\ & a_i.d == d_i \ \forall i \in A \\ & a_i.st := [q_i, \dots, eoh]) \ \forall i \in A \\ & \text{Alternative}(a_i, ju_{(i,:,:)}, |UN_i|) \ \forall i \in A \\ & \text{Cumulative}(ju_{(:,:,n)}, req_{(:,k)}, c_{n,k}) \\ & \qquad \forall n \in N, \forall k \in RK \end{aligned} \tag{7.2}$$

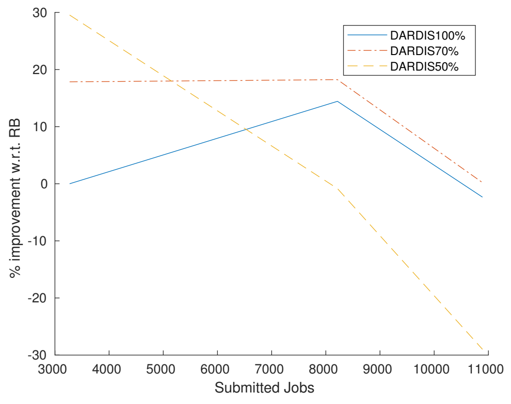
The optimization relies of the J part of the objective function. In fact, given a finite set of jobs to schedule, without considering the idle power of the resources, and with a fixed energy per Joule cost, the only part of the objective function that can change within different scheduling solution is the cooling expenses J. In fact, J depends on the cooling system efficiency model [99] and the external temperature, different start times for each job can move the efficiency point of the system by changing the total consumed power but also could move energy consumption in cooler hours of the day in which the efficiency increases.

7.4 Profile Extraction

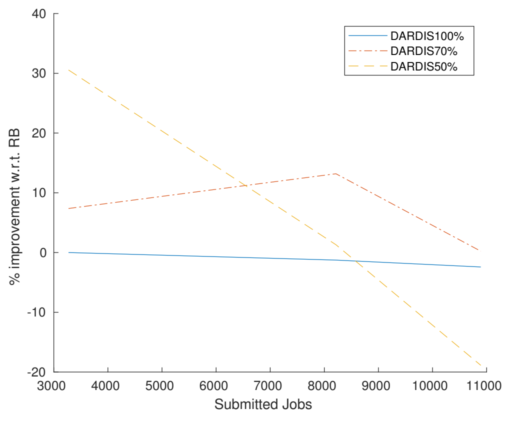
The profile extraction is based on the result analysis done in [6]. The work shows that if the average system utilization is higher than the 84%, it is more profitable for the next day to increase the system utilization rather than to optimize the cooling in case of low-efficient cooling systems. This work does not consider the possibility of shut-down and turn-on nodes to decrease expenses. The claim of this work is that under the possibility to shut-down and turn-on nodes, also with high-efficient cooling systems (such as hybrid cooling systems) it is profitable to limit the utilization with the same policy.



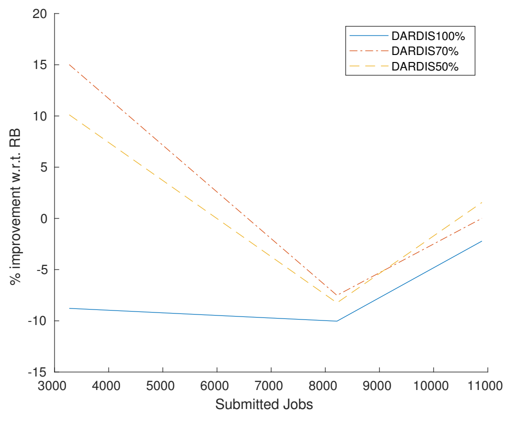
(A)



(B)



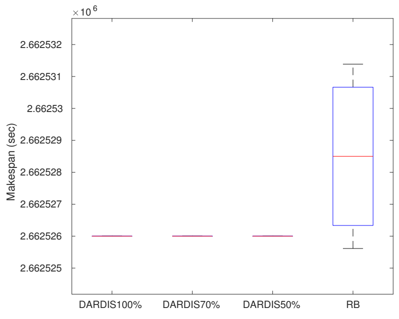
(C)



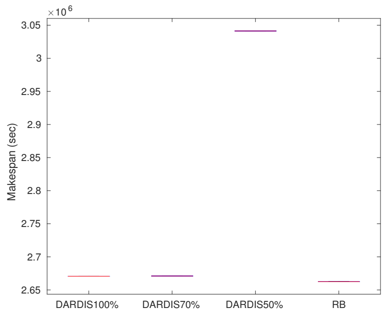
(D)

FIGURE 7.2: Profit improvement by DARDIS100%, DARDIS70%, and DARDIS50% w.r.t the best Rule-based scheduler in the scenario. (a) Summer temperature, Air cooling; (b) Summer temperature, Hybrid cooling; (c) Winter temperature, Air cooling; (d) Winter temperature, Hybrid cooling.

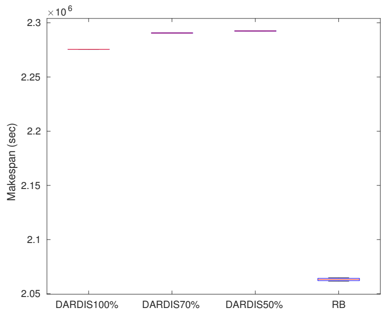
The algorithm (Algorithm 8) checks if in the last 24 hours the average system utilization is higher than the 84% of the resources of the system (lines 1-2). If it is higher, it sets the desirable system utilization for the whole next day to the maximum (the resource physical limit) (line 3). Otherwise, it generates the desirable profile hour by hour with the following policy. If in the last 24 hours the average system utilization is higher than 84%, it sets the next hour to the maximum (lines 7-8). Otherwise, it sets the desirable utilization profile to the average of the last hours plus a percentage $offsetPerc$ (line 10).



(A)



(B)



(C)

FIGURE 7.3: Average makespan obtained by DARDIS100%, DARDIS70%, DARDIS50%, and the best Rule-based with standard deviation. (a) Low workload trace; (b) Medium workload trace; (c) High workload trace.

Algorithm 8 ProfileExtraction($offsetPerc$)

```
1: set $prevDayAVG$ the average percentage resource utilization of the previous day
2: if $prevDayAVG > 84\%$ then
3:     set the desirable profile for the next 24h to 100%
4: else
5:     foreach hours $h$ in $[0, \dots, 24]$ do
6:         set $prev24hAVG$ the average percentage resource utilization of the previous 24hours
7:         if $prev24hAVG > 84\%$ then
8:             set the desirable profile for the next hour $h$ to 100%
9:         else
10:            set the desirable profile for the next hour $h$ to $prev24hAVG + (100 - prev24hAVG) * offsetPerc$ %
11:        end if
12:    end for
13: end if
14: $ON$ = Order($N$,Rule2)
```

7.5 Distributed Online scheduling

The online scheduling has to be managed by a fast and reactive scheduler that can plan the scheduling results in order to fulfill a desirable utilization profile. For this reason, we used DARDIS [4, 5]. The DARDIS scheduler is a highly tunable distributed and profile-aware scheduler.

The scalability of this scheduler is granted by the fact that the complexity of its scheduling algorithm is not based on the number of nodes but on the dimension of the scheduling horizon. To do that, DARDIS distributes the scheduling computation through the nodes. This means that each node, based on its utilization profile, decides a number of candidates starting time

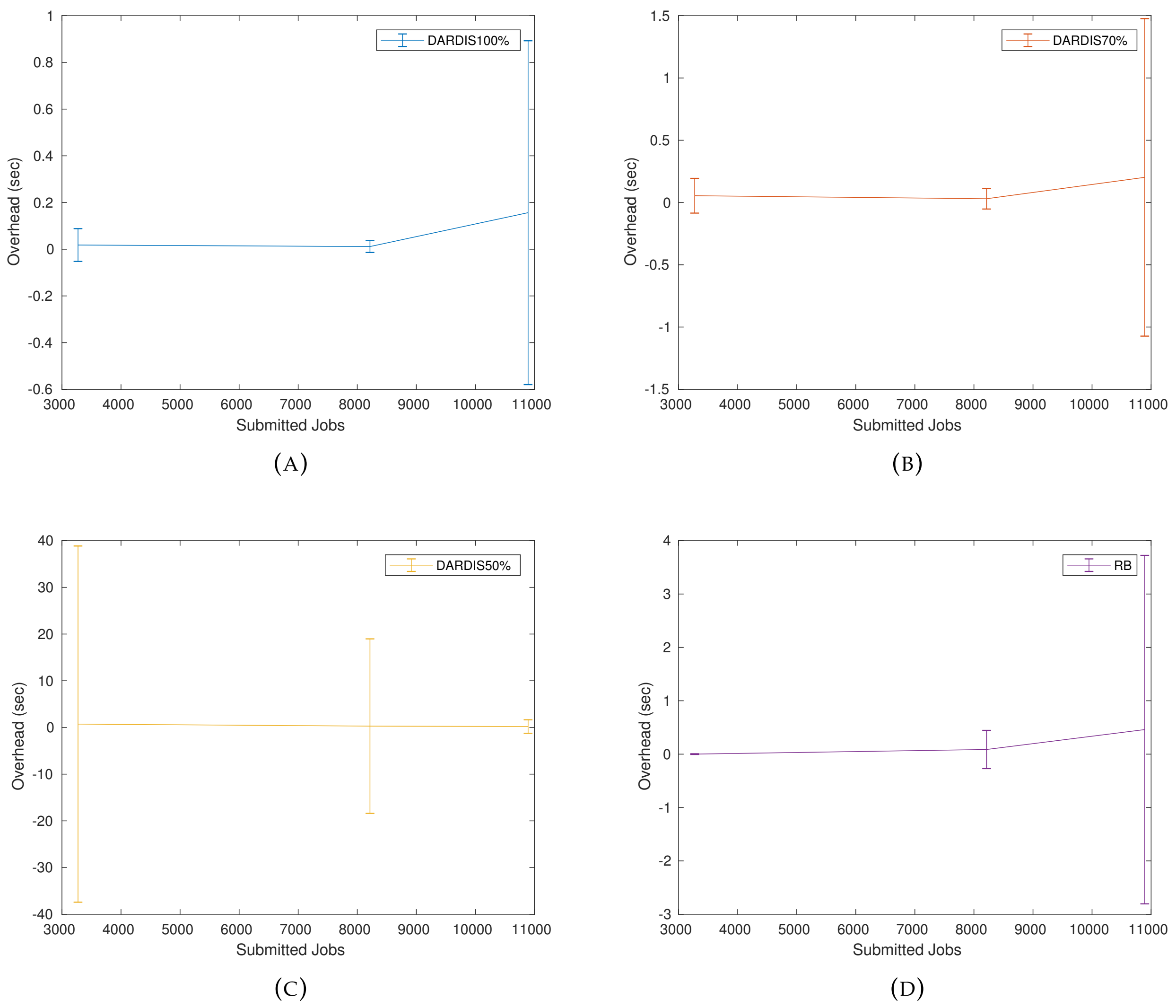


FIGURE 7.4: Average overhead with standard deviation, in seconds, at different workload for (a) DARDIS100%, (b) DARDIS70%, (c) DARDIS50%, and (d) the best Rule-based.

for a submitted job. The starting times are then sent to the job itself. The job selects the actual starting time between the candidates and thus the nodes for the execution.

DARDIS have been designed to guarantee that, for a given deadline for the job execution, the amount of resources is sufficient to execute the job, than the job is executed. Moreover, this is done without nodes synchronization and with the minimum amount of candidate starting times for each node (i.e. minimum message size).

DARDIS implements different policies for the candidate start times generation and the nodes selection. For the candidate start time generation we can choose:

- First: this policy is designed for scenarios in which it is important to have a high jobs throughput. The policy selects the sets of minimum start times in which the resource capacity constraint is satisfied.

- Uniform: this policy is designed to introduce randomization in the results. In fact, works such as [143] demonstrate that the introduction of randomization into the scheduling, in some scenarios, can introduce benefits on the results. As for example, in this scenario, the uniform policy has as results an utilization profile the better follows the desirable utilization profile. This policy uses to selects the start times the ratio $I(t) = min_{k \in RK}(\frac{D(t)-U(t)}{req_{(i,k)}})$, where t is a general candidate start time, $D(t)$ is the desirable utilization profile at time t, and $U(t)$ is the resource utilization (by previously scheduled jobs) at time t. After the computing of $I(t)$ in each possible time instant in which the job can be scheduled, the candidates start times are randomly selected using the distribution I.

- Exponential: this policy is designed to obtain a trade-off between the First and the Uniform policies. In fact, this policy introduces some randomization but the distribution selected tends to select candidates start times near to the minimum. To compute the probability distribution $I'(t)$ for the candidates start times, the policy obtains the $I(t)$ distribution that subsequently is weighted to an exponential distribution:

$$\begin{aligned} I'(t) &= I(t) * \lambda e^{-\lambda t} \\ \lambda &= \frac{1}{|I(t)|} \end{aligned} \tag{7.3}$$

After the candidates start times generation, each node sent the generated starting times to the job with the corresponding $I(t)$. The job, after that, to select the actual start time, applies one of the following policies:

- Min start: this policy is designed to maximize throughput. The policy orders the candidate start times from the minimum to the maximum. Then it checks the candidates and selects the first that allows the execution of $|UN_i|$ job units: $\sum_{n \in N} I_n(t) >= |UN_i|$, with $I_n(t)$ we mean $I(t)$ of the node n.

- Max probability: this policy is designed for scenarios in which high utilization peaks have to be discouraged. The policy, in fact, selects the starting time, within the candidates, in which the distance between the system utilization and the desirable profile is higher. This is done by ordering the starting time by decreasing $\sum_{n \in N} I_n(t)$ and selecting the first element iff $\sum_{n \in N} I_n(t) >= |UN_i|$.

- Random: this policy is designed to completely randomize the start time selection. This is done by ordering the starting time by decreasing $\sum_{n \in N} I_n(t)$ and deleting the candidates for which does not hold the constraint $\sum_{n \in N} I_n(t) >= |UN_i|$. After that, the actual start time is selected with a random uniform distribution.

7.6 Results

Tests have been made using thee different traces extracted from the parallel workload archive of the CEA Curie system [150]:

1. **Low workload**: 3207 jobs submitted in 30 days;
2. **Medium workload**: 8125 jobs submitted in 30 days;
3. **High workload**: 10892 jobs submitted in 23 days.

The simulated HPC machine is composed of 300 nodes with 32 cores each and 128GB of RAM [167].

The tests have been executed in different scenarios to explore the behavior of the proposed solutions with different cooling systems:

1. An **Air cooling** with a Power Usage Effectiveness [*] (PUE [168]) of ~ 1.4 (depending on the external temperature and the workload energy);
2. a **Hybrid cooling** system with a PUE of ~ 1.1;

and different environment temperatures:

1. **Summer** temperatures;
2. **Winter** temperatures.

The core hour pricing for the execution of the jobs is taken from the Amazon pricing model [170] while the energy cost is taken from the Italian energy provider [171].

The comparison has been done w.r.t. the six setups of the Rule-based scheduler. In each scenario, the setup with the best profit has been selected. For seek of simplicity, from now on, we will call the best Rule-based scheduler in each scenario “RB”.

In the following results we will show three different version of DARDIS:

- DARDIS100%: This version is used as baseline for the results comparison. This is the default DARDIS version set to use the “First” (see section 7.5) start time generator and the “Min start” dispatching policy. In this version, the assumption is that the scheduler can not turn-off the nodes of the HPC.
- DARDIS70%: This version is designed to turn-off nodes following the results of the profile generator in a conservative way. Given the forcasted desirable utilization profile, this scheduler shut down just the 30% of the unused resources while the remaining 70% is used to compensate system fragmentation. As for DARDIS100%, the selected start time generator is “First” and the dispatching policy is “Min start”.

[*]The Power Usage Effectiveness is a measure for data-centers efficiency. The calculation is $PUE = \frac{P_w + P_c}{P_w}$. Where P_c is the power spent in cooling and P_w is the power spent in workload

- DARDIS50%: This version is designed to turn-off nodes following the results of the profile generator aggressively. Given the forcasted desirable utilization profile, this scheduler shut down the 50% of the unused resources while just the remaining 50% is used to compensate system fragmentation. As for DARDIS100%, the selected start time generator is "First" and the dispatching policy is "Min start".

The implementation has been done in *C++* using the *IBM ILOG CP Optimizer 12.7.0* solver for the offline scheduler, *C++* using *MPI* for DARDIS and *python* for the profile extractor. Tests have been executed on a 2x*Intel Xeon Processor E5-2670 v3* server with 128GB of RAM.

7.6.1 Profit improvement

Figure 7.2 shows the percentile profit improvement w.r.t. the RB at the last job termination. Each plot shows the results obtained in the different workload traces (x-axis) by the three different version of DARDIS: DARDIS100%, DARDIS70%, and DARDIS50%. The figure shows four different plots representing four different scenarios: summer temperatures with air cooling system, summer temperatures with a hybrid cooling system, winter temperatures with air cooling system, and winter temperatures with a hybrid cooling system. As the figure shows, DARDIS100% is, in the majority of the cases, outperformed by DARDIS70% while, usually, it outperforms DARDIS50%. This suggests that turning off nodes can improve the profit obtained by an HPC but exceeding in the number of turned off nodes can result in a worsening.

In general, DARDIS50% obtains better results w.r.t. DARDSI70% just in the low workload trace. This confirms the hypothesis that the node turn-off limit of DARDIS50% is too aggressive in systems with this kind of fragmentation and the more conservative limit of DARDIS70% is, in general, better.

For what concerns the comparison w.r.t. the RB (table 7.1), DARDIS70% has an average improvement of 17% in the low workload trace, 8.6% in the medium workload trace and 0.2% in the high workload trace. Moreover, just one worsening w.r.t the RB scheduler have been found in the winter temperature, hybrid cooling, medium workload scenario.

7.6.2 Makespan

However, shutting down resources could provide an increment of the makespan. Figure 7.3 shows the average makespan and with the standard deviation obtained by DARDIS100%, DARDIS70%, DARDIS50%, and the RB schedulers in the three different traces: (a) low workload, (b) medium workload, and (c) high workload. In the low workload trace, we obtained not only a profit increment but also a makespan decrement. In the medium trace DARDIS100% and DARDIS70% obtained a negligible increment of the makespan of 0.3%. Finally, in the high workload trace, we have a worsening in the makespan of 10%. However, looking the makespan obtained by DARDIS70% and DARDIS100% we can claim that this desirable generation algorithm in conjunction with the DARDIS70% policy does not substantially impact the makespan. The majority of the makespan increment comes from the DARDIS algorithm itself.

7.6.3 Overhead

Figure 7.4 shows the overhead of the schedulers in each experiment. The four plots show respectively DARDIS100%, DARDIS70%, DARDIS50%, and the best rule-based scheduler. The plots show the average overhead to compute the scheduling of a job (considering also different scheduling cycles in which the job appears) for all the scenarios with the standard deviation at different workload dimensions.

As the plots show, the average overhead obtained by DARDIS100% and DARDIS70% is in the range 0.01-0.2 while the RB scheduler is in the range 0.0008-0.5. The behaviors of the schedulers show that, as expected, the overhead of the RB is lower in the low workload scenarios. In fact, the DARDIS100% and DARDIS70% shows a better scalability, having a lower average overhead and a consistently lower standard deviation in both the medium and high workload scenarios.

The behavior of DARDIS50% is completely different from the other two versions of DARDIS. In fact, it shows a decreasing overhead and standard deviation at increasing workload dimension. This is due to the fact that the complexity of this scheduler depends on the scheduling horizon and limiting the resource in such aggressive way as DARDIS50% does, highly increases the scheduling horizon. This is particularly visible by the standard deviation obtained by the scheduler that affects the jobs with the highest resource requirements (thus the most difficult to schedule).

7.7 Conclusion

In conclusion, we designed a new scheduling architecture for more profitable and greener HPC machines. This architecture is capable of considering the scheduling system, the cooling system, and the external temperature to generate a quasi-optimal, desirable utilization profile by an offline scheduler. While the online scheduler can schedule jobs with low overhead, profiling the scheduling and planning the shut-down and start-up of the nodes in the future. Three different versions of the scheduler have been tested w.r.t. six different setups of the rule-based scheduler in four different scenarios with three different workload traces. Our DARDIS70% scheduler shows to improve in average the HPC profit by an 8.6%. This improvement has been obtained within a lower average computational overhead and lower standard deviation. No significant worsening on the makespan has been found the low and medium workload. However, an increment of 10% on the makespan have been found in the high workload trace but this is imputable mainly to the DARDIS architecture and not to the node shut-down approach by DADIS70%.

A lot of work has to be done to improve the solution by DARDIS in term of makespan. Moreover, future work plans to introduce a synthetic workload forecaster capable of precisely generate an expected trace submission with jobs power consumption, in order to replace the current traces based on the previous 24 hours of submissions.

Chapter 8

Conclusion

In this thesis, we studied the HPC scheduling problem applying an optimization technique called Constraint Programming. Our target was to create a highly scalable and optimized scheduler. Many objective functions were taken into account.

We evaluated the applicability of CP and embedded a CP scheduler into a commercial scheduler trying different combinations of objective functions. The considered objective functions are the makespan, number of late jobs, tardiness, and weighted tardiness. Tests were made on different scenarios with increasing problem hardness and the scheduler have been compared to PBS Professional 12. The scheduler showed to improve up to 20% the waitings and 22% the number of late jobs. Even if this scheduler demonstrated to be well suited for an in-production mid-tier HPC, this approach showed to suffer from scalability problems.

For this reason, we designed a Distributed And Randomized DIspatcher and Scheduler (DARDIS). This scheduler is designed to exploit two different resources utilization profile: the first is the physical limit (capacity) of the resources, the second is a desirable utilization. The resource capacity is a *hard constraint*. This means that the capacity can never be exceeded. The desirable utilization profile, differently, it's just a recommended utilization profile and can be exceeded (*soft constraint*) if needed. The scheduler uses different rules to optimize the chasing of the desirable profile: random uniform start-time generation, random exponential start-time generation, and deterministic minimum start-time selection. Moreover, different rules can be adopted to select the nodes for the execution of the jobs: the nodes with the minimum start time, the nodes with the lower utilization-rate on the desirable utilization profile, and random uniform selection. The scheduler is designed to be distributed to improve the scalability. This is achieved by delegating the scheduling to the nodes: each node returns to the job a set of candidate start-times for the execution. Then, the dispatching is left to the job itself that, on the basis of the selected rule, chooses the set of nodes (and in fact also the start time between the candidates) for the execution. Tests were made on a critically high workload trace and DARDIS has been compared w.r.t. different rule-based schedulers. Results show that with different setups it is possible to achieve different results such as good profile chasing or job's throughput optimization etc. However, the tests showed that results are biased by the goodness of the desirable utilization profile. This means that the profile has to be finely tuned for the set of jobs we are expecting to be submitted. However, the key feature of this scheduler is the computational overhead. Results show that w.r.t. a rule-based scheduler the overhead is more than 200 times faster.

This brought us to study a better way to compute a desirable utilization profile. For this reason, we designed an offline CP scheduler that optimizes the HPC profit by modeling both the HPC and the cooling model and using as input the thermal forecast of the next 24 hours and the set of jobs submitted in the previous 24h. A heuristic and different searches have been designed to increase the scalability of this scheduler w.r.t. the default search implemented in ILOG CP Optimizer. And an extensive test in several different scenarios has been done to compare the results of our CP model and searches w.r.t. rule-based schedulers and the default search in ILOG CP Optimizer. The results show that in case of low workload no changes has been found w.r.t. heuristic schedulers while in case of high workload, the profit has been increased by 6-7% w.r.t. the best rule-based scheduler in that scenario. The default ILOG search showed to obtain a worsening of 10% w.r.t. the best rule-based scheduler in every condition.

Finally, we connected the two different approach we designed. The offline CP scheduler computes a near-optimal solution to improve the profit. From the result obtained, a profile extraction module extracts a utilization profile designed to shutdown nodes when the full HPC power is unnecessary and turn back on the nodes when a high computational power is required. After that, the utilization profile is fed to the online distributed scheduler that plans the scheduling in the future taking into account the node shutdown and turn-on. This approach showed to improve in average the HPC machine profit by an 8.6% and showed a better scalability w.r.t. the most used commercial schedulers. No worsening on the makespan have been found under low and medium workload requests, however, a 10% of worsening in the makespan has been found in case of high workload requests.

Future work will explore machine learning techniques for the traces of the offline CP scheduler. This will give us the possibility to compute the desirable future utilization profile on the basis of forcasted instead of past traces. This will involve in the creation of several different neural networks for the forecast of all the jobs information: submission time, resource requirement, job duration, and so on.

Bibliography

[1] A. Bartolini, A. Borghesi, T. Bridi, M. Lombardi, and M. Milano. Proactive workload dispatching on the EURORA supercomputer. English. In *Principles and practice of constraint programming - 20th international conference, CP 2014, lyon, france, september 8-12, 2014. proceedings*. B. O'Sullivan, editor. Vol. 8656. In Lecture Notes in Computer Science. Springer. Springer International Publishing, 2014, pp. 765–780. ISBN: 978-3-319-10427-0. DOI: 10.1007/978-3-319-10428-7_55.

[2] T. Bridi, M. Lombardi, A. Bartolini, L. Benini, and M. Milano. A cp scheduler for high-performance computers. In. Vol. 1485, 2015, pp. 37–42. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85009223255&partnerID=40&md5=65ac2e65b77b8fbb06d15c101edd7bbd.

[3] T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE transactions on parallel and distributed systems*, 27(10):2781–2794, 2016. DOI: 10.1109/tpds.2016.2516997. URL: http://dx.doi.org/10.1109/TPDS.2016.2516997.

[4] T. Bridi, M. Lombardi, A. Bartolini, L. Benini, and M. Milano. DARDIS: Distributed And Randomized DIspatching and Scheduling. In *European conference on artificial intelligence (ECAI 2016)*. Vol. 285. Gal A. Kaminka et al., 2016, pp. 1598–1599.

[5] T. Bridi, M. Lombardi, A. Bartolini, L. Benini, and M. Milano. DARDIS: Distributed And Randomized DIspatching and Scheduling. In, *AI*IA 2016 advances in artificial intelligence*, pp. 493–507. Springer, 2016.

[6] T. Bridi, A. Bartolini, M. Lombardi, P. V. Hentenryck, M. Milano, and L. Benini. Profit-driven hpc scheduling optimization and pue analysis. *IEEE transactions on industrial informatics*, under review, 2017.

[7] T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. Hybrid offline-optimized and online-distributed profit-driven low-overhead scheduler for hpc with automatic node shut-down and turn-on. *IEEE transactions on parallel and distributed systems*, under review, 2017.

[8] C. Galleguillos, A. Sîrbu, Z. Kiziltan, O. Babaoglu, A. Borghesi, and T. Bridi. Data-driven job dispatching in hpc systems. In *International workshop on machine learning, optimization, and big data*. Springer, 2017, pp. 449–461.

[9] Top500. Top500. http://www.top500.org.

[10] Top500. The linpack benchmark. http://www.top500.org/project/linpack/.

[11] N. Attig, P. Gibbon, and T. Lippert. Trends in supercomputing: the european path to exascale. *Computer physics communications*, 182(9):2041–2046, 2011.

[12] M. M. Waldrop. More than moore. *Nature*, 530(7589):144, 2016.

[13] R. A. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. A. Nichols, J. Nieplocha, et al. High performance computational chemistry: an overview of nwchem a distributed parallel application. *Computer physics communications*, 128(1-2):260–283, 2000.

[14] A. Dubey, S. Brandt, R. Brower, M. Giles, P. Hovland, D. Q. Lamb, F Loffler, B. Norris, B. OShea, C. Rebbi, et al. Software abstractions and methodologies for hpc simulation codes on future architectures. *Arxiv preprint arxiv:1309.1780*, 2013.

[15] H. C. Greenwell, W. Jones, P. V. Coveney, and S. Stackhouse. On the application of computer simulation techniques to anionic and cationic clays: a materials chemistry perspective. *Journal of materials chemistry*, 16(8):708–723, 2006.

[16] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. Walberla: hpc software design for computational engineering simulations. *Journal of computational science*, 2(2):105–112, 2011.

[17] A. A. Johnson and T. E. Tezduyar. 3d simulation of fluid-particle interactions with the number of particles reaching 100. *Computer methods in applied mechanics and engineering*, 145(3-4):301–321, 1997.

[18] T Tezduyar, S Aliabadi, M Behr, A Johnson, V Kalro, and M Litke. Flow simulation and high performance computing. *Computational mechanics*, 18(6):397–412, 1996.

[19] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, et al. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of physics: condensed matter*, 21(39):395502, 2009.

[20] K. Takahashi and Y. Tanaka. Material synthesis and design from first principle calculations and machine learning. *Computational materials science*, 112:364–367, 2016.

[21] C. R. Welch, W. F. Marcuson III, and I. Adiguzel. Will supermolecules and supercomputers lead to super construction materials? *Civil engineering magazine archive*, 78(11):42–53, 2008.

[22] G. Hager and G. Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[23] S. Wan and P. V. Coveney. Rapid and accurate ranking of binding affinities of epidermal growth factor receptor sequences with selected lung cancer drugs. *Journal of the royal society interface*:rsif20100609, 2011.

[24] S. Wan and P. V. Coveney. Molecular dynamics simulation reveals structural and thermodynamic features of kinase activation by cancer mutations within the epidermal growth factor receptor. *Journal of computational chemistry*, 32(13):2843–2852, 2011.

[25] Top500. Hōkūle'a hpc. https://www.top500.org/system/179103.

[26] Top500. Sunway taihulight hpc. https://www.top500.org/system/178764.

[27] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 2008 acm/ieee conference on supercomputing*. IEEE Press, 2008, p. 1.

[28] T. Geller. Supercomputing's exaflop target. *Communications of the acm*, 54(8):16–18, 2011.

[29] H. Frazier. The 802.3 z gigabit ethernet standard. *Ieee network*, 12(3):6–7, 1998.

[30] G. F. Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel i/o*, 42:617–632, 2001.

[31] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Ieee computational science and engineering*, 5(1):46–55, 1998.

[32] M. Snir. *Mpi–the complete reference: the mpi core*. Vol. 1. MIT press, 1998.

[33] M. Feldman. With roadrunner's retirement, petascale enters middle age. http://www.top500.org/blog/with-roadrunners-retirement-petascale-enters-middle-age/. 2013.

[34] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of parallel and distributed computing*, 61(6):810–837, 2001.

[35] M. Maheswaran, T. D. Braun, and H. J. Siegel. Heterogeneous distributed computing. *Wiley encyclopedia of electrical and electronics engineering*, 1999.

[36] T. Braun, H. Siegel, and A. Maciejewski. Heterogeneous computing: goals, methods, and open problems. *High performance computing—hipc 2001*:307–318, 2001.

[37] H. J. Siegel, H. G. Dietz, and J. K. Antonio. Software support for heterogeneous computing. *Acm computing surveys (csur)*, 28(1):237–239, 1996.

[38] S. Shepler, M. Eisler, D. Robinson, B. Callaghan, R. Thurlow, D. Noveck, and C. Beame. Network file system (nfs) version 4 protocol. *Network*, 2003.

[39] R. J. Billings. Secure copy method and device for stored programs. US Patent 4,550,350. 1985.

[40] B. Kantor. Bsd rlogin, 1991.

[41] T. Ylonen and C. Lonvick. The secure shell (ssh) protocol architecture, 2006.

[42] B. Chen, C. N. Potts, and G. J. Woeginger. A review of machine scheduling: complexity, algorithms and approximability. In, *Handbook of combinatorial optimization*, pp. 1493–1641. Springer, 1998.

[43] J. K. Lenstra, A. R. Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of discrete mathematics*, 1:343–362, 1977.

[44] J. Blazewicz, J. K. Lenstra, and A. R. Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete applied mathematics*, 5(1):11–24, 1983.

[45] P. Works. Pbs professional 12.2, administrator's guide, november 2013. 2013.

[46] A. Computing and G. Computing. Torque resource manager. *Online] http://www. adaptivecomputing. com*, 2012.

[47] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: simple linux utility for resource management. In *Job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[48] K. Qureshi, S. M. H. Shah, and P. Manuel. Empirical performance evaluation of schedulers for cluster of workstations. *Cluster computing*, 14(2):101–113, 2011.

[49] R. L. Henderson. Job scheduling under the portable batch system. In *Job scheduling strategies for parallel processing*. Springer, 1995, pp. 279–294.

[50] P. Salot. A survey of various scheduling algorithm in cloud computing environment. *International journal of research and engineering technology (ijret), issn*:2319–1163, 2013.

[51] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Workshop on job scheduling strategies for parallel processing*. Springer, 1997, pp. 1–34.

[52] R. Haupt. A survey of priority rule-based scheduling. *Operations-research-spektrum*, 11(1):3–16, 1989.

[53] I. A. Moschakis and H. D. Karatza. Evaluation of gang scheduling performance and cost in a cloud computing system. *The journal of supercomputing*, 59(2):975–992, 2012.

[54] C. Du, X.-H. Sun, and M. Wu. Dynamic scheduling with process migration. In *Cluster computing and the grid, 2007. ccgrid 2007. seventh ieee international symposium on*. IEEE, 2007, pp. 92–99.

[55] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. *Eecs department, university of california, berkeley, tech. rep. ucb/eecs-2009-55*, 2009.

[56] D. Jackson. New issues and new capabilities in hpc scheduling with the maui scheduler.

[57] S. Agarwal, R. Garg, and N. K. Vishnoi. The impact of noise on the scaling of collectives: a theoretical approach. In, *High performance computing–hipc 2005*, pp. 280–289. Springer, 2005.

[58] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: empirical approximation and impact on schedulability. *Proceedings of ospert*:33–44, 2010.

[59] R. Gioiosa, S. A. McKee, and M. Valero. Designing os for hpc applications: scheduling. In *Cluster computing (cluster), 2010 ieee international conference on*. IEEE, 2010, pp. 78–87.

[60] R. Haupt. A survey of priority rule-based scheduling. *Operations-research-spektrum*, 11(1):3–16, 1989.

[61] G. Bruno, A. Elia, and P. Laface. A rule-based system to schedule production. *Computer*, (7):32–40, 1986.

[62] S. S. Panwalkar and W. Iskander. A survey of scheduling rules. *Operations research*, 25(1):45–61, 1977.

[63] R. Wang, D. Tiwari, and J. Wang. Low power job scheduler for supercomputers: a rule-based power-aware scheduler. In *Data science and data intensive systems (dsdis), 2015 ieee international conference on*. IEEE, 2015, pp. 732–733.

[64] S. Srinivasan, R. Kettimuthu, V. Subramani, and P Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Parallel processing workshops, 2002. proceedings. international conference on*. IEEE, 2002, pp. 514–519.

[65] D. G. Feitelson. Experimental analysis of the root causes of performance evaluation results: a backfilling case study. *Parallel and distributed systems, ieee transactions on*, 16(2):175–182, 2005.

[66] A. W. M. Alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and distributed systems, ieee transactions on*, 12(6):529–543, 2001.

[67] Y. Yuan, Y. Wu, W. Zheng, and K. Li. Guarantee strict fairness and utilizeprediction better in parallel job scheduling. *Parallel and distributed systems, ieee transactions on*, 25(4):971–981, 2014.

[68] E. Shmueli and D. G. Feitelson. On simulation and design of parallel-systems schedulers: are we doing the right thing? *Parallel and distributed systems, ieee transactions on*, 20(7):983–996, 2009.

[69] S. Hartmann. A self-adapting genetic algorithm for project scheduling under resource constraints. *Nrl*, 49(5):433–448, 2002.

[70] J. Damay, A. Quilliot, and E. Sanlaville. Linear programming based algorithms for preemptive and non-preemptive rcpsp. *European journal of operational research*, 182(3):1012–1022, 2007.

[71] T. Bhaskar, M. N. Pal, and A. K. Pal. A heuristic method for rcpsp with fuzzy activity times. *European journal of operational research*, 208(1):57–66, 2011.

[72] O. Sarood, A. Langer, A. Gupta, and L. Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. IEEE Press, 2014, pp. 807–818.

[73] S. Soner and C. Özturan. Integer programming based heterogeneous cpu–gpu cluster schedulers for slurm resource manager. *Journal of computer and system sciences*, 81(1):38–56, 2015.

[74] Y. Kessaci, N. Melab, and E Talbi. A pareto-based ga for scheduling hpc applications on distributed cloud infrastructures. In *High performance computing and simulation (hpcs), 2011 international conference on*. IEEE, 2011, pp. 456–462.

[75] K. Wang. Towards next generation resource management at extreme-scales. *Iit, phd proposal*, 2014.

[76] J. P. Jones and B. Nitzberg. Scheduling for parallel supercomputing: a historical perspective of achievable utilization. In *Job scheduling strategies for parallel processing*. Springer, 1999, pp. 1–16.

[77] D. Klusácek, H. Rudová, R. Baraglia, M. Pasquali, and G. Capannini. Comparison of multi-criteria scheduling techniques. 2008.

[78] V. Chlumsky, D. Klusácek, and M. Ruda. The extension of torque scheduler allowing the use of planning and optimization in grids. *Computer science*, 13:5–19, 2012.

[79] E. Shmueli and D. G. Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *Journal of parallel and distributed computing*, 65(9):1090–1107, 2005.

[80] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *Parallel and distributed systems, ieee transactions on*, 18(6):789–803, 2007.

[81] A. Borghesi, C. Conficoni, M. Lombardi, and A. Bartolini. Ms3: a mediterranean-stile job scheduler for supercomputers-do less when it's too hot! In *High performance computing & simulation (hpcs), 2015 international conference on*. IEEE, 2015, pp. 88–95.

[82] X. Feng, R. Ge, and K. W. Cameron. Power and energy profiling of scientific applications on distributed systems. In *Parallel and distributed processing symposium, 2005. proceedings. 19th ieee international*. IEEE, 2005, pp. 34–34.

[83] B. Hurley, B. O'Sullivan, and H. Simonis. Icon loop energy show case. In, *Data mining and constraint programming*, pp. 334–347. Springer, 2016.

[84] S. H. Lu and P. Kumar. Distributed scheduling based on due dates and buffer priorities. *Automatic control, ieee transactions on*, 36(12):1406–1416, 1991.

[85] K. Ramamritham, J. Stankovic, W. Zhao, et al. Distributed scheduling of tasks with deadlines and resource requirements. *Computers, ieee transactions on*, 38(8):1110–1123, 1989.

[86] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *High performance distributed computing, 2002. hpdc-11 2002. proceedings. 11th ieee international symposium on*. IEEE, 2002, pp. 359–366.

[87] H. Izakian, B. T. Ladani, K. Zamanifar, and A. Abraham. A novel particle swarm optimization approach for grid job scheduling. In, *Information systems, technology and management*, pp. 100–109. Springer, 2009.

[88] S. Zhan and H. Huo. Improved pso-based task scheduling algorithm in cloud computing. *Journal of information & computational science*, 9(13):3821–3829, 2012.

[89] H. Izakian, B. T. Ladani, A. Abraham, and V. Snasel. A discrete particle swarm optimization approach for grid job scheduling. *International journal of innovative computing, information and control*, 6(9):4219–4233, 2010.

[90] L. Vanneschi, D. Codecasa, and G. Mauri. A comparative study of four parallel and distributed pso methods. *New generation computing*, 29(2):129–161, 2011.

[91] A. Montresor, H. Meling, and Ö. Babaoğlu. Messor: load-balancing through a swarm of autonomous agents. In, *Agents and peer-to-peer computing*, pp. 125–137. Springer, 2003.

[92] C. L. Ortiz, R. Vincent, and B. Morisset. Task inference and distributed task management in the centibots robotic system. In *Proceedings of the fourth international joint conference on autonomous agents and multiagent systems*. ACM, 2005, pp. 860–867.

[93] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory*. Springer, 1983, pp. 127–140.

[94] A. Olshevsky and J. N. Tsitsiklis. Convergence speed in distributed consensus and averaging. *Siam journal on control and optimization*, 48(1):33–55, 2009.

[95] J. Cortés. Distributed algorithms for reaching consensus on general functions. *Automatica*, 44(3):726–737, 2008.

[96] F. Benhamou. Principles and practice of constraint programming-cp 2006. In *12th international conference, cp*. Springer, 2006, pp. 25–29.

[97] C. P. Gomes, W. J. v. Hoeve, and B. Selman. Constraint programming for distributed planning and scheduling. In *Aaai spring symposium: distributed plan and schedule management*. Vol. 1, 2006, pp. 157–158.

[98] C. C. Rolf and K. Kuchcinski. *Distributed constraint programming with agents*. Springer, 2011.

[99] C. Conficoni, A. Bartolini, A. Tilli, C. Cavazzoni, and L. Benini. Integrated energy-aware management of supercomputer hybrid cooling systems. *Ieee transactions on industrial informatics*, 12(4):1299–1311, 2016.

[100] A. Bartolini, M. Cacciari, C. Cavazzoni, G. Tecchiolli, and L. Benini. Unveiling eurora - thermal and power characterization of the most energy-efficient supercomputer in the world. In *Design, automation test in europe conference exhibition (date), 2014*, 2014.

[101] N. Kaur, S. Bansal, and R. K. Bansal. Towards energy efficient scheduling with dvfs for precedence constrained tasks on heterogeneous cluster system. In *Recent advances in engineering & computational sciences (raecs), 2015 2nd international conference on*. IEEE, 2015, pp. 1–6.

[102] D. Cheng, X. Zhou, P. Lama, M. Ji, and C. Jiang. Energy efficiency aware task assignment with dvfs in heterogeneous hadoop clusters. *Ieee transactions on parallel and distributed systems*, 2017.

[103] K. Li, X. Tang, and K. Li. Energy-efficient stochastic task scheduling on heterogeneous computing systems. *Ieee transactions on parallel and distributed systems*, 25(11):2867–2876, 2014.

[104] J. Luo, L. Rao, and X. Liu. Temporal load balancing with service delay guarantees for data center energy cost optimization. *Ieee transactions on parallel and distributed systems*, 25(3):775–784, 2014.

[105] Y. Zhao, R. N. Calheiros, G. Gange, K. Ramamohanarao, and R. Buyya. Sla-based resource scheduling for big data analytics as a service in cloud computing environments. In *Parallel processing (icpp), 2015 44th international conference on*. IEEE, 2015, pp. 510–519.

[106] F. F. Moghaddam, R. F. Moghaddam, M. Cheriet, and Y. Lemieux. Defining green profit in distributed datacenters, 2015.

[107] H. R. Faragardi, A. Rajabi, T. Nolte, and A. H. Heidarizadeh. A profit-aware allocation of high performance computing applications on distributed cloud data centers with environmental considerations. *Csi journal on computer science and engineering jcse*, 2(1):28–38, 2014.

[108] J. Hikita, A. Hirano, and H. Nakashima. Saving 200kw and $200 k/year by power-aware job/machine scheduling. In *2008 ieee international symposium on parallel and distributed processing*, 2008, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536218.

[109] O. Mämmelä, M. Majanen, R. Basmadjian, H. De Meer, A. Giesler, and W. Homberg. Energy-aware job scheduler for high-performance computing. *Computer science - research and development*, 27(4):265–275, 2012. ISSN: 1865-2042. DOI: 10.1007/s00450-011-0189-6. URL: https://doi.org/10.1007/s00450-011-0189-6.

[110] D. Feitelson. Workload characterization and modeling book. http://www.cs.huji.ac.il/~feit/wlmod/wlmod.pdf. 2015.

[111] S Heipcke. Comparing constraint programming and mathematical programming approaches to discrete optimisation - the change problem. *Journal of the operational research society*, 50(6):581–595, 1999.

[112] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten. Constraint-based scheduling and planning. *Foundations of artificial intelligence*, 2:761–799, 2006. ISSN: 1574-6526. URL: http://linkinghub.elsevier.com/retrieve/pii/S157465260680026X.

[113] C. Bessiere and P. Van Hentenryck. To be or not to be... a global constraint. *Cp*, 2003:789–794, 2003.

[114] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial intelligence*, 57(2-3):291–321, 1992.

[115] P. Laborie and J. Rogerie. Reasoning with conditional time-intervals. In *Proc. of flairs*, 2008, pp. 555–560. URL: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:Reasoning+with+Conditional+Time-intervals\#0.

[116] P. Van Beek. Backtracking search algorithms. *Foundations of artificial intelligence*, 2:85–134, 2006.

[117] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[118] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[119] P. Refalo. Impact-based search strategies for constraint programming. *Cp*, 3258:557–571, 2004.

[120] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, 9(2):135–196, 1977.

[121] G. Katsirelos and F. Bacchus. Generalized nogoods in csps. In *Aaai*. Vol. 5, 2005, pp. 390–396.

[122] F. Bacchus. Extending forward checking. *Principles and practice of constraint programming–cp 2000*:35–51, 2000.

[123] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*. Springer, 1998, pp. 417–431.

[124] CINECA. Cineca inter-university consortium web site. http://www.cineca.it//en. Accessed: 2014-04-14.

[125] Eurotech. Eurotech group web site. http://www.eurotech.com/en/. Accessed: 2014-04-14.

[126] Ny times article about a survey by mc kinsey & co. http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html. Accessed: 2014-04-14.

[127] A. P. Works. Pbs professional®12.2 administrator's guide. http://resources.altair.com/pbs/documentation/support/PBSProAdminGuide12.2.pdf. 2013.

[128] P. Laborie. IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In *Proc. of cpaior*, 2009, pp. 148–162. URL: http://www.springerlink.com/index/y7r30xh22h1gp026.pdf.

[129] P. Laborie and D. Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. In *Proc. of mista*, 2007, pp. 276–284. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.4415\&amp;rep=rep1\&amp;type=pdf.

[130] T. Frühwirth and S. Abdennadher. *Essentials of constraint programming*. Springer Science & Business Media, 2003.

[131] M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, 1996.

[132] IBM. Constraint programming with cp optimizer. https://www.ibm.com/support/knowledgecenter/SSSA5P_12.3.0/ilog.odms.cpo.help/Content/Optimization/Documentation/Optimization_Studio/_pubskel/gscpoptimizer2528.html. 2011.

[133] D. Pisinger and S. Ropke. Large neighborhood search. In, *Handbook of metaheuristics*, pp. 399–419. Springer, 2010.

[134] D. Godard, P. Laborie, and W. Nuijten. Randomized large neighborhood search for cumulative scheduling. In *Icaps*. Vol. 5, 2005, pp. 81–89.

[135] F Falciano and E Rossi. Fermi: the most powerful computational resource for italian scientists. *Embnet. journal*, 18(A):p–62, 2012.

[136] TOP500. Tianhe-2 (milkyway-2) - th-ivb-fep cluster, intel xeon e5-2692 12c 2.200ghz, th express-2, intel xeon phi 31s1p. http://www.top500.org/system/177999. 2015.

[137] BBC. Supercomputers: obama orders world's fastest computer. http://www.bbc.com/news/technology-33718311. 2015.

[138] J. Lavignon et al. Etp4hpc strategic research agenda achieving hpc leadership in europe. http://www.etp4hpc.eu/wp-content/uploads/2013/06/ETP4HPC_book_singlePage.pdf. 2013.

[139] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, et al. Exascale computing study: technology challenges in achieving exascale systems. *Defense advanced research projects agency information processing techniques office (darpa ipto), tech. rep*, 15, 2008.

[140] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Parallel job scheduling for power constrained hpc systems. *Parallel computing*, 38(12):615–630, 2012.

[141] C. Lefurgy, X. Wang, and M. Ware. Power capping: a prelude to power shifting. *Cluster computing*, 11(2):183–195, 2008.

[142] B. Rountree, D. H. Ahn, B. R. De Supinski, D. K. Lowenthal, and M. Schulz. Beyond dvfs: a first look at performance under a hardware-enforced power bound. In *Parallel and distributed processing symposium workshops & phd forum (ipdpsw), 2012 ieee 26th international*. IEEE, 2012, pp. 947–953.

[143] M. Van Den Briel, P. Scott, and S. Thiébaux. Randomized load control: a simple distributed approach for scheduling smart appliances. In *Proceedings of the 23th international joint conference on artificial intelligence*. AAAI Press, 2013, pp. 2915–2922.

[144] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz. Dynamic power sharing for higher job throughput. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. ACM, 2015, p. 80.

[145] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, et al. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. ACM, 2015, p. 78.

[146] C. Conficoni, A. Bartolini, A. Tilli, G. Tecchiolli, and L. Benini. Energy-aware cooling for hot-water cooled supercomputers. In *Proceedings of the 2015 design, automation & test in europe conference & exhibition*. EDA Consortium, 2015, pp. 1353–1358.

[147] Y. Joshi and P. Kumar. *Energy efficient thermal management of data centers*. Springer Science & Business Media, 2012.

[148] R. Von Mises. *Mathematical theory of probability and statistics*. Academic Press, 1964.

[149] I. Herstein. Topics in algebra, blaisdell pub. *Co., mass*, 1964.

[150] P. W. Archive. http://www.cs.huji.ac.il/labs/parallel/workload/l_cea_curie/index.html. 2012.

[151] I. Sindicic, S. Bogdan, and T. Petrovic. Resource allocation in free-choice multiple reentrant manufacturing systems based on machine-job incidence matrix. *Ieee transactions on industrial informatics*, 7(1):105–114, 2010.

[152] W. Du, Y. Tang, S. Leung, L. Tong, A. V. Vasilakos, and F. Qian. Robust order scheduling in the fashion industry: a multi-objective optimization approach. *Ieee transactions on industrial informatics*, 2017.

[153] S. Girs, A. Willig, E. Uhlemann, and M. Björkman. Scheduling for source relaying with packet aggregation in industrial wireless networks. *Ieee transactions on industrial informatics*, 12(5):1855–1864, 2016.

[154] Y. Wu, X. Tan, L. Qian, D. H. Tsang, W.-Z. Song, and L. Yu. Optimal pricing and energy scheduling for hybrid energy trading market in future smart grid. *Ieee transactions on industrial informatics*, 11(6):1585–1596, 2015.

[155] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.

[156] S. Venugopalan and O. Sinnen. Ilp formulations for optimal task scheduling with communication delays on parallel systems. *Ieee transactions on parallel and distributed systems*, 26(1):142–151, 2015.

[157] Y. Hou, N. Wu, M. Zhou, and Z. Li. Pareto-optimization for scheduling of crude oil operations in refinery via genetic algorithm. *Ieee transactions on systems, man, and cybernetics: systems*, 2015.

[158] A. Zimmermann. An mip-based heuristic for scheduling projects with work-content constraints. In *Industrial engineering and engineering management (ieem), 2016 ieee international conference on*. IEEE, 2016, pp. 1195–1199.

[159] A. Borghesi, F. Collina, M. Lombardi, M. Milano, and L. Benini. Power capping in high performance computing systems. In *Principles and practice of constraint programming*. Springer, 2015, pp. 524–540.

[160] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete applied mathematics*, 123(1):75–102, 2002.

[161] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*. Vol. 39. Springer Science & Business Media, 2012.

[162] CINECA. Cineca new accounting policy. https://wiki.u-gov.it/confluence/pages/viewpage.action?pageId=64201371. 2017.

[163] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. Predictive modeling for job power consumption in hpc systems. In *International conference on high performance computing*. Springer, 2016, pp. 181–199.

[164] HudsonPower. `https://hudsonenergy.co.uk/corporate-businesses`. 2017.

[165] S. f. Research. `https://research.csc.fi/pricing-of-computing-services`. 2017.

[166] U. o. Maine. `https://acg.umaine.edu/pricing/fee-structure/`. 2017.

[167] TOP500. `https://www.top500.org/system/177003`. 2017.

[168] M. Uddin, R. Alsaqour, A. Shah, and T. Saba. Power usage effectiveness metrics to measure efficiency and performance of data centers. *Applied mathematics & information sciences*, 8(5):2207, 2014.

[169] IDC. Analysis of the characteristics and development trends of the next-generation of supercomputers in foreign countries. `http://www.aics.riken.jp/aicssite/wp-content/uploads/2017/05/Analysis-of-the-Characteristics-and-Development-Trends.pdf`. 2016.

[170] Amazon. Ec2 pricing. `https://aws.amazon.com/it/ec2/pricing/on-demand/`.

[171] Enel. `https://www.enelenergia.it/mercato/libero/it-IT/imprese/grandi-aziende/offerte-prezzo-fisso`. 2017.