Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Computer Science and Engineering

Ciclo XXIX

Settore Concorsuale di afferenza: 09/H1

Settore Scientifico disciplinare: ING-INF/05

POWER-AWARE JOB DISPATCHING IN HIGH PERFORMANCE COMPUTING SYSTEMS

Presentata da: **Andrea Borghesi**

Coordinatore Dottorato

Prof. Paolo Ciaccia

Relatore

Prof.ssa Michela Milano

Correlatore

Prof. Luca Benini

Esame finale anno 2017

Abstract

This works deals with the power-aware job dispatching problem in supercomputers; broadly speaking the dispatching consists of assigning finite capacity resources to a set of activities, with a special concern toward power and energy efficient solutions. We tackle the problem from different angles and introduce novel optimization approaches to address its multiple aspects.

The proposed techniques have a broad application range but are particularly aimed at applications in the field of High Performance Computing (HPC) systems. In recent years there has been a remarkable increase in the worldwide installed capacity and the peak computational performance of HPC machines. Together with the computational increase the HPC community observed a growing power demand that would lead supercomputers to unacceptable power consumptions. For this reason, lately the focus has been steadily shifting from maximum peak computational performance towards a more balanced and power-aware approach.

Devising a power-aware HPC job dispatcher is a complex and multi-faceted problem, where diverse, partially contrasting goals must be satisfied. An additional layer of difficulty is represented by the online nature of the problem: generally speaking, supercomputers do not run the same set of application over their entire life but rather have to react to new and unexpected job requests. Hence, a solution to the problem must be computed in real time and therefore stringent time limits must be respected in order not to disrupt the system behaviour. This aspect discourages the usage of exact methods and instead suggests the adoption of heuristic techniques. Historically, this have always been the case in the HPC practice and literature, at the cost of settling for extremely suboptimal solutions. The application of optimization techniques to the dispatching task is still an unexplored area of research and can drastically improve the performance of HPC systems.

In this work we first tackle the job dispatching problem on a real HPC machine, the Eurora supercomputer hosted at the Cineca research center, Bologna. We propose a Constraint Programming (CP) model that outperforms the dispatching software currently in use. An essential element to take power-aware decisions during the job dispatching phase is the possibility to estimate jobs power consumptions before their execution. To this end, we applied Machine Learning techniques to create a prediction model that was trained and tested on the Euora supercomputer, showing a great prediction accuracy. Then we finally develop a power-aware solution, considering the same target machine, and we devise different approaches to solve the dispatching problem while curtailing the power consumption of the whole system under a given threshold. We propose a heuristic technique and a hybrid method combining CP with a heuristic algorithm. Both our methods are able to solve practical size instances and outperform most of the current state-of-the-art techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work addresses the workload dispatching problem in High Performance Computing (HPC) systems – also referred to as supercomputers. Given a workload composed by a set of activities (or jobs) the dispatching problem consists in assigning a pool of finite capacity resources and a start time to each activity to produce a feasible schedule, i.e. a schedule respecting the resource capacity constraints at any time. Job dispatching in supercomputers is a branch of the wider Scheduling and Allocation problem (S & A) that arises in several different computing fields. Workload dispatchers in real HPC systems must satisfy a number of different requirements, ranging from respecting the users' needs to meeting system owners' productivity goals. We are especially concerned with the issue of power consumption, a matter which is acquiring increasing relevancy along with the growth of worldwide HPC installed capacity.

Supercomputers peak performance, measured in FLOPs (floating point operation per second), has been growing steadily since their introduction in the 1960s and it is expected that the Exascale (10^{18}) will be reached around 2023, even though a handful of more optimistic estimates forecast the Exascale as early as 2020. Nonetheless, a very concerning obstacle towards reaching this goal is posed by the power consumption of nowadays HPC machines. Prediction trends indicate that simply scaling up current architectures would lead to unsustainable power demands (hundreds of MWatts). Several diverse research avenues are currently being explored in order to improve the energy efficiency of supercomputers by at least one order of magnitude, ranging from novel hardware solutions to improved system management.

HPC systems are large machines composed by numerous physical parts that need to be carefully orchestrated. An essential role of any software infrastructure overseeing a supercomputer is workload dispatching, whose policies and actions can have deep repercussions on the overall system performance. Job dispatching is a notoriously complex task, due in part to the sheer scale of the problem (supercomputers easily have hundreds or thousands of computing nodes where tens of jobs need to run simultaneously) and in part to its online nature, because optimization must be performed respecting real-time constraints (theoretically no delay to workload execution should be introduced by the dispatching system). This happens since the applications which are going to be submitted to the supercomputer are not known in advance (except a few exceptions) and therefore it is not possible to compute an optimal schedule offline, but a dispatcher must

rather be able to respond to unforeseen jobs arrivals.

The online requirement of HPC job dispatching coupled with the intrinsic difficulty of S & A problems (which have been known to be NP-hard since long time) has led to overwhelming preponderance of heuristic solutions, with an almost complete disregard of combinatorial optimization techniques. This is partially due to the fact that many combinatorial methods aim at finding exact, optimal solutions but typically require excessively long times to explore the space of possible solutions and have been therefore confined within domains where the optimization can be performed offline (i.e. embedded systems).

Nevertheless, if the exactness constraint is relaxed, i.e. settling for good solutions rather than optimal ones, combinatorial techniques have been proved to be very effective in finding solutions outperforming many heuristic approaches while respecting the time limit. For example, Constraint Programming (CP) has a long history of success when dealing with Scheduling problems. The important point is to find the right trade-off between solution quality and effort spent in searching for solutions. Another promising approach relies in decomposing the job dispatching problem in simpler sub-problems, reducing its complexity, and mixing methodologies from combinatorial optimization and heuristic techniques, exploiting their respective strong suits.

1.1 Content

We deal with the job dispatching problem in HPC systems which can be decomposed in two components: 1) the Allocation problem, i.e. choosing the set of finite capacity resources to be assigned to each activity and 2) the Scheduling problem, i.e. deciding the start time for each job in order to produce a feasible schedule that respect all the involved constraints. We are particularly interested in devising a power-aware solution capable to bound the power consumption of the HPC system. The main content of this work can be divided in three parts.

Scheduling & Allocation in HPC Systems: We begin with addressing the job dispatching problem on a real HPC system (namely the Eurora supercomputer hosted at CINECA in Bologna). We aim at finding solutions that maximize the Quality-of-Service perceived by users (measured in terms of waiting time) and at the same time maximize also the system utilization; these two goals are not mutually exclusive but finding the right trade-off is a non-trivial problem. Keeping the system utilization at a high level serves a two-fold purpose: 1) satisfying more users requests per period of time (with a positive effect for the system owners thanks to the higher revenues) and 2) reducing the number of idle computational units, which would otherwise consume significant amounts of unused power. We focus on devising an online job dispatcher and therefore the time needed to find a solution must respect strict limits.

While the vast majority of current job dispatchers present a reactive nature and their scheduling and allocation policies can usually be described by simple set of rules (i.e. First-In-First-Out), we define a novel, proactive job dispatcher that explores a much wider space of possible decisions. At the core of our approach there is a Constraint Programming model. Due to the real-time requirements of online dispatching, the proposed approach does not strive to find optimal solutions but it specializes in finding the best among the feasible ones encountered within the time limit. We show that the proposed model leads

to remarkable improvements in terms of both machine utilization and waiting times for queued jobs with respect to the currently used dispatcher, i.e. Portable Batch System (PBS).

Predictive Model for the Power Consumption of HPC Jobs: The fundamental advantage of proactive dispatching is long-term reasoning over a group of activities rather then taking immediate, short-term rewarding decisions that can lead to performance degradation over longer periods. An effective reasoning process relies on the possibility of making informed choices, i.e. the dispatcher needs to evaluate the consequences of its scheduling and allocation decisions. A good schedule can be produced only if we know at decision time (within a degree of confidence) how a workload will evolve; for example, job dispatchers tend to assume that the amount of resource requested by each activity will be constant throughout its whole duration.

Having this in mind it is clear that, in order to preserve the benefits provided by a proactive dispatcher while adding power awareness, the ability to predict the power consumption associated to any workload before its execution assumes a critical importance. We therefore consider the problem of estimating the power consumption of HPC applications using predictive models created with Machine Learning techniques. For this scope, we implemented an information gathering infrastructure that assembled measurements from a variety of physical sensors (such as power consumption, temperature, etc.) on the Eurora supercomputer.

We then take advantage of this infrastructure and the collected information to create a data set that we use to train a collection of predictive models to estimate the power consumption of any job. These models take as inputs only information available at dispatch time. We then tested the quality of our predictions against historical traces of past workloads executed on the supercomputer. The results of our analysis show that the power consumption of each job, including jobs never encountered before, can be predicted with a very high level of accuracy.

Power-Aware Job Dispatching: Finally, we take into account the power-related issues and we propose two methods to create a job dispatcher able to contain the power consumption of a real HPC system within an administrator-decided power budget. To reach this end we devise two approaches: 1) a heuristic algorithm and a 2) hybrid approach that decomposes the problem in the scheduling stage and the allocation stage, solving the former via Constraint Programming and the latter with a heuristic technique. The introduction of the power constraint adds a new layer of complexity and consequently we refrain again from obtaining optimal solutions due to real-time constraints. The goal of the dispatcher is to minimize the performance degradation and QoS decrease that are inherently associated with the imposition of a bound on the power consumption.

The approach we propose is orthogonal to the most commonly used techniques in today's supercomputers, which generally rely on power monitoring infrastructure and dynamic power management through an intervention on the performance of computational resources, i.e. reducing the operating frequency of a CPU leads to a decreased power consumption. In our method we want to act on the Scheduling & Allocation decisions alone, for example with a careful planning of the execution order. Since Scheduling & Allocation decisions take place during the dispatching phase the capability of estimating the power con-

sumption of any HPC job during this phase assumes a paramount importance. For this reason we integrate into the dispatcher the ML models to predict jobs power consumptions described previously.

The experimental analysis reveals that our approach is able to outperform the current State-of-the-Art methods used in power capped supercomputers. Moreover, we demonstrate that it is possible to combine our approach with other techniques such as those acting on the computing node performance; mixing non-mutually exclusive methods allows to exploit their respective strengths, resulting in the best final outcome. We apply our dispatcher to a real supercomputer (again the Eurora HPC system), taking into consideration its cooling infrastructure, and we show how the novel power capping method enables to maximize the supercomputer efficiency. Lastly, we consider the case of variable power budgets focusing on the case of sudden power cap decrease, i.e. critical situation where the power consumption of the current workload must be drastically lowered due to the malfunctioning of other components of the HPC machine. We therefore add a module to the power capped dispatcher that is able to cope with these situations through a reduction of the computing nodes performance.

1.2 Contribution

This work is positioned at the intersection of High Performance Computing, combinatorial optimization and Machine Learning. The main contributions of this work regard the development of novel approaches to solve Scheduling & Allocation problems in the context of High Performance Computing, with a special emphasis on power-aware solutions. In particular we both applied state-of-the-art and devised original methods belonging to the area of combinatorial optimization and Machine Learning in order to tackle S & A in supercomputers. The contributions of this work are the following:

- A novel, proactive, Constraint Programming based job dispatcher for supercomputers. The basic idea of the CP model is to consider multiple jobs (i.e. all jobs in the waiting queue) whenever a new schedule is computed and explore a broad space of possible solutions, instead of using the reactive, rule-based policy adopted by the vast majority of current supercomputers. The use of Constraint Programming allowed us to beat the system currently adopted on the HPC machine used as a case study and at the same time the high flexibility of the CP paradigm simplifies the implementation of the new model to other real world systems. This work was published in [BBB+14].

- A new methodology to develop predictive models to estimate the power consumption of HPC applications employing existing Machine Learning techniques. The most important element we provide is the development and description of the entire process leading from the data collection infrastructure to the creation and use of the predictive models. We deal with several challenges that arise in real supercomputer, ranging from establishing the minimal amount of data needed to guarantee a good prediction accuracy and how to create a good training set to feed the Machine Learning models starting from the raw measurement data. We devise a

method that was tested on the historical power consumption traces of a real system; a key strength of our approach is the possibility to be applied to different systems. This work was partially presented in [BBL+16a]

- Two novel approaches to deal with the problem of job dispatching in a HPC system with power cap. We propose a heuristic algorithm and an innovative hybrid technique combining CP and a heuristic technique. The hybrid methods is loosely inspired by the decomposition techniques found in the combinatorial optimization literature but solves the problem in a new way. The comparison with the state-of-the-art power capped job dispatchers in HPC reveals that our methods perform extremely well. Moreover, our approach is extremely flexible and can be easily integrated with different techniques from the state-of-the-art in order to achieve even better performance or address sudden changes in the available power budget. Parts of this work have been published in [BCLB15, BCL+15] and another part (the comparison with the state-of-the-art) should appear on the journal *Transactions on Parallel and Distributed Systems*.

1.3 Outline

The structure of the thesis is the following. Chapter 2 gives a panoramic overview of the Scheduling & Allocation problem in HPC machines, with a particular focus on power related issues. The chapter also provides a brief introduction and state-of-the-art discussion on the enabling methodologies used to tackle the main problem, namely the Constraint Programming paradigm and Machine Learning. Chapter 3 then formally introduces the job dispatching problem and presents a model to solve it. Chapter 4 addresses the issue of predicting the power consumption of HPC applications, describing the predictive models created for that purpose and the data collecting process employed to gather the information needed to effectively use Machine Learning techniques. Chapter 5 introduces the power dimension to the job dispatching problem. The chapter describes a novel job dispatcher able to bound a supercomputer power consumption and compares its performance with other methods from the current state-of-the-art.

Chapter 2

Related Work

This chapter gives a panoramic view on the existing research related to High Performance Computing systems, with a focus on optimization of workload management. The chapter is split in three main subsections. First, Section 2.1 introduces the High Performance Computing problem, focusing on the main issues considered in this work, i.e. workload management optimization and power-awareness. Afterwards, the following sections discuss the enabling technologies that will form the grounds for the approaches presented in the rest of the work. Section 2.2 considers the Constraint-Based Scheduling approach, describing the Constraint Programming paradigm and its application to Scheduling and Allocation problems. Section 2.3 discusses Machine Learning, its techniques and its relevance in High Performance Computing setting.

2.1 HPC systems

The term *High Performance Computing (HPC) systems* – also called *supercomputers* – refers to machines with very large computational capacities, which have reached nowadays tens of PetaFLOPS[1], i.e. the number of floating point operations per second. Trends in the last twenty years show an exponential increase of the peak performance; following these trends the ExaFLOPs (10^{18}) scale is expected to be reached in 2020-2023 [KR13, Kha11]. Supercomputers are used in a wide range of computationally intensive applications, such as weather forecasting, physical modeling, aerodynamic research, probabilistic analysis, molecular dynamics simulation and many others [Rus78, AAA01, Jos97, KAB+00, Gid86, CW03, Mis90, EHB+13, CKL+13, ABD+14, MKM15, VFHB14, HBBD16, KHO+16]. HPC systems are also part of those computing centers which play a key role in modern ICT architectures, running our internet services, managing several infrastructures, making our research possible.

Whereas earlier HPC systems used innovative designs and parallelism to achieve superior computational performance [CLHH09] and usually only a few processors, since the end of the 20th century machines with thousands of processors began to appear and today supercomputers are usually composed by a very large number (tens or hundreds of thousands) of "off the shelf", general

[1]FLOPS is an acronym for FLoating-point Operations Per Second, 1 PetaFLOPS corresponds to 10^15 FLOPS

(a) Eurora (b) Fermi

Figure 2.1: Two supercomputers hosted by CINECA [CIN] in Bologna

purpose processors – CPUs, graphical units, hardware accelerators – linked by fast connections [HJS00, HT89]. As the number of components increases, supercomputers managers have to address and solve numerous challenges in order to guarantee good performance for the customers and maintain reasonably low functioning costs for the owners. Another aspect adding an additional layer of complexity is the heterogeneous nature of the HPC systems, since they may include various types of resources (CPU, HW accelerators, Random Access Memory, hard disks,..), as well as various different kinds of computing units, each of them with specific characteristic and ability to fulfill different requirements.

Typically, supercomputers have several users which submit their own computationally intensive applications (usually called "jobs" or "tasks" in the literature). These jobs need to be assigned to a portion of the system resources – according to their requirements – and then be executed. From the users point of view the notion of good performance entails that the applications are correctly run and complete their execution, possibly with the smallest delay possible between the submission of the task and its start. Conversely, HPC systems owners must satisfy all the users requests but at the same time have their own peculiar set of concerns: keep as low as possible the operational costs of the machine (power consumption – see Section 2.1.3, maintenance costs..) while maximizing profits[2], i.e. servicing the greatest possible number of customers without degrading users' performance.

2.1.1 HPC Workloads

The behaviour of a supercomputer strongly depends on the characteristics of the applications running on it, i.e. its workload [CS93, CCF+99, Stu, YZ13b]. Typically, users submit their jobs into the system specifying a set of resource requirements. For example each job may specify the number of execution nodes,

[2] They also have to take into account the finite lifespan of HPC machines, usually no more than 3-5 years [Fel13]

the number of needed CPUs, the number of hardware accelerators (such as GPUs), the amount of volatile memory (RAM), etc. In most of HPC systems users must also declare an estimated maximum duration for each submitted job, the *wall time*, and if the job lasts longer than this expected duration the workload manager is authorized to kill the job. For this reason, users tend to over-estimate the wall time of their applications, in order not to risk their task being killed.

In the literature there are several works dedicated to model HPC workloads [LF03,DF99,Dow98,MHCD10,AF01], which are useful to understand how systems are used and what could be done to improve their performance (and this is especially important in large scale system). Usually, workload models are composed by the set of statistical distributions describing the parameters characterizing HPC tasks, i.e. the runtime, the amount of requested resources and resource actually used, network communications, etc. The data used to create these models can be generally found in the accounting logs generated by HPC systems.

Cirne et al. [CB01b] investigate the relation between requested time and the real execution time and then study the jobs arrival pattern of several different systems. As it turns out, the arrival pattern of HPC applications is strongly dependent on the considered system (it is not trivial to generalize the observations made on a particular supercomputer) and there is a clear correlation between requested and execution time.

2.1.1.1 HPC Jobs Models

The vast majority of applications executed on supercomputers belong to the class of *parallel jobs*, that is they are composed by independent communicating activities. From the dispatcher point of view parallel jobs can be broadly divided in four categories. Such categories depends on how and when the amount of requested resources is specified; such an amount can be specified by the user (either within the program itself or through the job submission specific format) or it may be decided by the dispatcher. Moreover, the resource used (number of nodes, cores, etc.) can be fixed at the start of the execution or it can vary dynamically at run-time [Str03, HJS+04].

The categories are the following (summarized also in Table 2.1):

- *Rigid jobs* require a fixed number of resources, no more nor less than the amount specified by the user at submission time; they are fine-tuned for specific resources configurations and they cannot cope with situations such as a reduction of the number of available cores. These tasks are commonly written in the message passing paradigm (MPI [mpi16, GLDS96]), where all communication between processors is designed to have low latency.

- *Evolving jobs* may change the number of used resources at run-time; it is important to notice that the change is initiated by the application itself. The system must satisfy the request or the job will not be able to continue its execution. The scheduler is only aware of the amount of resources needed by the application at any time.

- *Moldable jobs* have a fixed amount of resources which is determined by the scheduler when the job starts; the decision of the scheduler depends on

who decides	*when the decision is taken*	
	at submittal	*at run-time*
User	Rigid	Evolving
System	Moldable	Malleable

Table 2.1: HPC Applications Taxonomy [FR96]

the job properties and on the state of the system. Moldable jobs usually work with a wide range of resources, but also have a minimum number of resources to use; additional resources might improve the performance, possibly up to a saturation point.

- *Malleable jobs* can run on a variable set of resources and the configuration change is initiated by the system. The scheduler decides if the job has to release resources or if additional ones are assigned; the job has no influence on this decision.

Nowadays, the vast majority of HPC applications running in supercomputers belong to the rigid category. There are two main reasons behind this preference: 1) managing rigid jobs is much simpler than the dealing with the other job models, thanks to their fixed amount of requested resources; 2) rigid jobs require a fine-tuned set of resources, often carefully optimized by the user, and that implies that their behaviour can be reasonably predicted – conversely, using different configurations might have unexpected repercussions (both positive and negative) on an application performance. However it is important to notice that rigid jobs have indeed their disadvantages, in particular their lack of flexibility can easily lead to an increased resource fragmentation and/or not optimal performance.

Research in the moldable jobs area is very active and has lead techniques that are currently experiencing a rise in their adoption [CB00,SKS03,SSK+02,CB02, SLS06, HSC09, HHTC14]. Many modern applications written in MPI [mpi16] exhibit moldable features, allowing them to choose to exploit different parallelisms for execution just before their start. Downey [Dow97] studies the runtime speedup of moldable jobs with different numbers of allocated processors and builds a model for them. Using this model as basis, Wu et al. [WTJ+15] investigate how resource allocation affects the average turnaround time of moldable jobs in clusters.

Cirne et al. [CB01a] propose a workload model for moldable jobs, which is based on a users' survey and analytical models. Their moldable job model takes as input a rigid job with only one known shape (partition size and execution time) and provides as output a realistic set of shapes for the job. Some existing supercomputer workload management systems already support submission of moldable jobs. For example, Load Sharing Facility (LSF) [lsf16], owned by IBM, lets users specify a range of processors requirements at submission time. However, its scheduling mechanism for moldable jobs is quite simple, basically adopting a greedy method to allocate as many processors as possible within the range specified by a moldable job.

The benefits granted by evolving and malleable jobs have been studied extensively in the literature [NVZ96, GTU91, MVZ93, CV96, SML+07, UCL04], al-

though very few works went beyond the proof of concept phase. Typically, these analysis compare the cost of dynamic reconfiguration to the performance improvement but fail to give a full picture. Dynamically changing the number of resources in most cases requires complex interactions between the operating system and the application run-time system [ABLL92] and this complexity often makes the implementation of malleable (or evolving) jobs too hard for real machines. A few works have tried nevertheless to exploit dynamic adaptability within data centers and HPC systems [Kal93, KAJ+12, AGJ+14].

2.1.2 HPC Systems Dispatching Problem

In order to optimally manage a HPC system a wide array of issues, which may arise in different phases during the life of these systems, needs to be addressed; many of these issues can be framed as optimization problems (finding the best solution from all feasible solutions). An essential element to successfully deploy a HPC job is deciding which resources will be allocated and providing an efficient run-time schedule, balancing the system users and owner different needs. This task is often referred to as *Scheduling & Allocation* or *Dispatching* problem and it is a widely studied in the literature [Gre75, ET96, ABSM01, MBS01, BSM01, SDA96, HL05]. In addition to *allocation*, the task of assigning a set of resource to an application is also referred to as *mapping*; in the rest of this work we will use both equivalent expressions. To summarize, the fundamental aspects of the HPC dispatching problem (as we are going to see in Chapter 3) consists in 1) optimally allocating the resources required by each task in its description, and 2) provide an optimal job execution order (explicitly assigning a start time to each task).

The problem of mapping and scheduling a set of tasks to different machines has been shown to be NP-complete in the general case [GJ79, BLK83, LKB77, FB89, IK77] as well as some restricted cases [Ull75], such as scheduling tasks with one or two time units to two processors and scheduling unit-time tasks to an arbitrary number of processors. The problem complexity is the reason why the majority of the approaches adopted in real-world systems rely on heuristic techniques and are more committed to finding feasible solutions rather obtaining optimal ones.

We can distinguish between *offline* and *online* approaches to the dispatching process. In the former the allocation and the schedule are computed before the actual execution; this method requires many details regarding the workload to be known a priori. For example, this approach is useful in the case of embedded systems because these devices usually run the same set of applications during their entire lifetime. In the HPC setting the required knowledge about the jobs which are going to run on the supercomputer is a major drawback, since it is not always possible to know in advance which tasks are going to be submitted by the users (except some particular cases, i.e. weather forecast applications need to run periodically). In the second approach, online methods, mapping and scheduling decisions are taken at *run-time*, upon the arrival of new tasks in the system. On one hand this removes the requirement of oracle-like knowledge of workloads, but on the other hand the mapping/scheduling algorithm must be extremely fast – preferrably with constant or low-degree polynomial time. Since, as mentioned before, multiprocessor allocation and scheduling on bounded resources is NP-hard, online approaches, in general, cannot guarantee optimality,

but they concentrate on satisfying "safe acceptance" criteria, i.e. schedulable tasks may be rejected but non-schedulable ones will never be accepted.

The problem of parallel job scheduling on HPC systems and large data centers has been extensively studied for many years [FR95, FR97, Fei97, CCS+06, MCF+98, TTDB13, CPS+96]. Existing mapping and scheduling approaches typically assume to have a precise and formal description of input applications (i.e. the jobs to be scheduled), able to explicitly expose task parallelism. For example, in the HPC context usual applications are composed by independent tasks executing without interfering with others, each having its own subset of assigned and non-shared resources. Usually an abstract representation of the system resources is supposed to be available. In [FRS+, FR98] Feitelson et al. carry out a thorough review of theoretical and practical results in the field, providing an insightful classification of different scheduling methods.

2.1.2.1 Existing Heuristics Techniques

As said previously, online job dispatching presents strict time constraints and therefore a common way to cope with this problem in real time is to use *heuristic* approaches, which do not guarantee optimal solutions but conversely are extremely fast. For example, the forefather of all scheduling algorithms might be the *First-Come-First-Served* (FCFS) scheduling algorithm, which is also referred to as *First-In-First-Out*, FIFO, policy. FCFS is a simple and static job scheduling policy, where a job is served on arrival basis; it is a classic example of a very simple but quick algorithm to decide which job to schedule – the first job to enter the system is also the first to be executed. While FIFO and its basic variants are extremely fast, their excessive simplicity is not well suited to deal with the complexity of job dispatching in a HPC system; for instance, a job requiring a large amount of unavailable resources can prevent the execution of other later jobs that would fit on the system. In many circumstances it could be more useful, from the system level perspective, to delay the execution of certain jobs disregarding their arrival time.

Currently, the vast majority of existing approaches, both in the literature and in real systems do not explore in depth the solution space. Most dispatchers settle with "good" solutions, neither globally nor locally optimal. Most commercial solutions, like PBS [Wor15] or Torque [Sta06] follow this approach. In many systems the burden of producing good schedules is still a task strictly reserved to system administrators. While their expertise in the field helps them reaching good practical solutions, the implicit complexity, with its multiple resource constraints and conflicting objectives, naturally makes the HPC job dispatching problem well suited to be addressed exploiting more advanced and automated optimization techniques. In the next paragraph we are going to discuss three commonly used heuristic techniques categories.

Backfilling Algorithm An important component of many commercial dispatchers is the *backfilling* algorithm [FRS05, SF09, SF03], which extends the core FIFO algorithm. The main idea of this algorithm is to fit smaller jobs on resources not used by the system due to fragmentation. Therefore, the jobs in the queue are executed (partially) disregarding the submission or priority order. In practice smaller jobs are moved ahead in order to fill gaps in the schedule, if they do not delay the execution of other jobs waiting in the job queue. This widely

spread technique must be used carefully because it can lead to fairness problems (newly arrived jobs can be executed before those waiting from a long period) and, in more dangerous cases, to starvation (the new jobs that keep on arriving prevent an old, low priority job to begin its execution indefinitely) [YWZL14].

The most commonly used version of the backfilling algorithm is the so called *EASY* backfilling algorithm, introduced by Mu'alem and Feitelson in [MF01]. This algorithm is a more aggressive version of typical backfilling methods because it moves ahead small jobs to fill holes in the schedule provided that they not delay only the *first* job in the job queue – in opposition to the more standard approach which requires all or multiple waiting jobs not to be delayed by the backfilling action. The performance of EASY backfilling depends heavily on the workload but in the majority of cases it performs better than more conservative backfilling algorithms. The same paper presents also a very interesting result: backfilling actually works better when users overestimate jobs run-times by a substantial factor.

Another backfilling technique is the *K-reserved based* policy [ABM11], where each job in the waiting queue has a counter containing K number of times that it has been overtaken by subsequent requests. K-reserved based policy works by defining a window of waiting jobs of size k, ordered in FIFO fashion. If a new job with high priority enters the system, the oldest job waiting for allocation can be bypassed provided the subsequent job is within a window of K consecutive jobs that start with the oldest waiting job. When all jobs within the window have been started, the backfilling mechanism is suspended until a large-enough set of resources becomes available for allocation to the oldest job, at which point the window is moved forward to the next oldest waiting job. This strategy tries to combine the benefit of backfilling (out-of-order execution) without disrupting the fairness of FIFO policies.

The behaviour of many heuristic scheduling techniques depends on the estimate of job durations, i.e. the preference is often given to shorter jobs. Usually an estimate is provided by the user at submission time, but this value might be inaccurate. In fact users typically over-estimate the duration of a job because they do not want their application to be abruptly terminated because the allotted time slot has expired while the job is still running. To deal with this issue, Tsafrir et al. [TEF07] propose a backfilling algorithm (EASY++) which does not rely on the user-provided duration estimate but instead employs a system-generated duration prediction.

Lawson et al. present [LSP02] a self-adapting, multiple-queue backfilling algorithm for parallel systems that directs incoming jobs to different queues according to the user-estimated job execution time. The key idea is to separate short and long jobs (using two distinct queues) thus decreasing the likelihood that a short job is overly delayed in the queue behind a very long job. Each queue is assigned to a non-overlapping partition of system resources on which jobs from the queue can execute; these partitions dynamically adapt in response to the workload evolution and system utilization.

Gang Scheduling Many scheduling techniques employed in real time systems adopt the *preemption* mechanism [ZRS87, SAÅ+04, KSS+05, Ves07, CV01, Sch96]: jobs can be stopped and re-allocated (possibly in different resource partitions). This means jobs do not have to run to completion once they have

started and the scheduler can interrupt the execution of a job at any time for starting a different job; the total processing time is not affected by preemption. Despite the proven fairness of preemption and time slicing techniques [SY00], these mechanisms are not widely used in real HPC systems due to the requirements in terms of hardware (high interconnection bandwidth is needed) and software (due to limitations on the message-passing architecture).

An extension of the base preemption is the ability to preempt all the subtasks of a parallel job at the same time, as well as restarting all the members of another job. This mechanism is called *gang scheduling* [Ous82, FR92, Fei96, FJ97]. In this case preemption is used to improve performance in face of unknown run-times. Gang scheduling prevents short jobs from being stuck in the queue waiting for long ones, and improves fairness. Gang scheduling works with every programming model, but has some limitations: the overall system performance is not always optimal, though it benefits individual jobs. There could also be a high overhead due to processors fragmentation and context switching.

Priority Rules Based Scheduling Many heuristic algorithms used to quickly produce good quality schedules fall in the *Priority Rules Based (PRB)* scheduling category [Hau89, THS02, KA96, SL93, ERL90]. The main idea underlying these methods is to sort the set of tasks which need to be scheduled, constructing the ordered list by assigning a priority to each task. Tasks are selected according to the priorities order and each selected task is mapped on the requested resources and started (if there are enough resources available), trying to minimizes a cost function. The algorithms in this category provide schedules of good quality and their performance is comparable to other approaches, usually with lower scheduling times [KA99].

The rules used by PRB algorithms can be fairly simple. For instance, *Smallest Job First* (SJF) [Lam14] and *Largest Job First* (LJF) [LC91] update the job queue – the jobs waiting to be executed – in, respectively, increasing and decreasing order in terms of job size[3]. The jobs are then selected for execution drawing them from the sorted queue. Other examples of job queue ordering are *Minimum estimated Execution Time* (MinET) and *Maximum estimated Execution Time* (MaxET), that sort the jobs depending on their expected durations – in an ascending and descending order, respectively.

PRB algorithms are often coupled with simple allocation strategies, such as *First-Fit* (FF), a strategy that, as the name says, tries to map a job in the first partition of system resources that can satisfy the job requirements. Alternatively, also the order in which the set of available resources is considered can be influenced by priority rules. For example, a power-aware job dispatcher operating in a heterogeneous supercomputer can prefer computational nodes with lower power/energy consumption (given that more than one node can satisfy the resources request).

We distinguish two approaches which are part of the PRB scheduling class. The first method is known as *serial* or *list scheduling* [KW59]. In algorithms belonging to this typology, at every stage the set of already scheduled activities and a decision set – containing activities whose predecessors have all been scheduled and thus can in turn start themselves – are identified. The priority

[3] Job size can be defined in different ways, for example job duration, job width (number of requested resources) or a combination of the two

rules are used to select an activity from the decision set and then this task is scheduled at the earliest possible time, guaranteeing to respect precedence and resource feasibility constraints. The other approach is generally called *parallel scheduling* [BB99]. In this method, at each stage the set of activities which can be scheduled, the decision set, contains only those tasks whose precedences and resource requirements can be satisfied in that time instant; among those activities a single one is selected using the priority rules. The main difference compared to the serial method is that resources and their finite capacities are considered when building the decision set, rather then when the start times are assigned.

An example of priority rule-based heuristic applied to a resource constrained scheduling problem can be found in [BK07]. The exact problem tackled by the authors consists of the scheduling of a set of activities which can be split if necessary. All resources considered are renewable and each resource unit may not be available at all times due to resource vacations, which are known in advance. Splitting an activity has a direct cost and impact on the total completion time and this is taken into account in a priority rule-based heuristic which allows to control activity splitting (i.e. minimize the number of times it happens). Klein et al. [Kle00] present several heuristics and meta-heuristics aimed at dealing with real-world resource constrained project scheduling problems. Traditionally, such heuristics construct a schedule by planning in a forward direction starting from the project's beginning; this work introduces new backward and bidirectional planning strategies, integrating them into priority rule-based algorithms. Kolisch et al. [Kol96] provide an analysis of several classical priority rules used for resource constrained project scheduling, showing their limitations and drawbacks, and then propose two new rules which lead to a better management of the finite-capability resources.

Chandio et al. [CXT+13] investigate the performance of several job scheduling strategies in large-scale parallel computational systems. They conducted their experimental evaluation on a real system with several heterogeneous workloads, taking into account different metrics to establish the best scheduling policy. Their analysis revealed that a single policy is not sufficient for resource management in parallel computing environments. This happens because different operational conditions and workloads are best served by different scheduling strategies; to overcome this limit, they suggest the adoption of dynamic and adaptive policies. Aida [Aid00] investigates the impact of job sizes on the scheduling policy performance. The paper evaluates the performance of job scheduling algorithms under various workload models, composed by different typologies of jobs (with different average sizes). The results indicate that, broadly speaking, algorithms classified in the first-fit category are less affected by jobs sizes than priority rules-based techniques.

2.1.3 Power/Thermal Considerations

The large number of processing units employed in current HPC machines requires a huge amount of electrical power, creating many power related issues (i.e. thermal design power, CPU power dissipation, power consumption) which generally have greater relevance than the case of more traditional computing systems due to the sheer number of involved resources.

Basically since their introduction decades ago, supercomputers' development

has been guided by the "performance at any cost" mentality, which aimed at increasing the computational capability with an almost complete disregard of corollary issues. This attitude is perfectly exemplified by the Top500 list [DMS94], which ranks the best supercomputers in the world depending on their computational peak power (GFLOPS)[4]. This mentality has to be abandoned because it is no longer sustainable due to the power aware considerations limiting the further growth of HPC systems [Fen03]. New solutions and more energy efficient approaches need to be developed and applied to the next generation of supercomputers if we want to reach the Exascale [FFG08].

The growing concern about power consumption and dissipation issues in supercomputers led to the birth of Green500, an organization that ranks the Top500 supercomputers based on their energetic efficiency [FC07]. The Green500 list takes into consideration an energy efficiency metric, GFLOPS per Watt (GOPS/W), for "big" enough supercomputers, i.e. able to pass the 96 TFLOPS threshold. Nowadays, the most powerful system is Sunway TaihuLight which reaches 93 PetaFLOPS with 15.371 MWatts of power dissipation [FLYeA16]. Thanks to its innovative design, it is also ranked 3^{rd} in the Green500, with 6.05GigaFLOPS/Watt. The second fastest machine (and record holder from 2013 until 2016) is Tianhe-2 and can be found at position 33^{rd} of the Green500, with 1.9GigaFLOPS/Watt.

Exascale supercomputers built with today's technology would led to an unsustainable power demand (hundreds of MWatts of power) while according to [BBCea08] an acceptable range for an Exascale supercomputer is 20MWatts. For this reason, current HPC systems must significantly increase their energy efficiency, with a goal of 50GFLOPS/W. Today's "greenest" supercomputers achieve around 9 GFLOPS/W, thus a wide gap still needs to be closed in order to satisfy Exascale requirements. Koomey et al. [KBSW11] examined the relationship between computing power and electricity required to operate a computing unit. The conclusion is that computation per kilowatt-hour has doubled every 1.57[5] year. While this is a promising finding, the implications for the HPC context still have to be clarified [SSSF13].

A detailed survey of the research on power management techniques for high performance systems can be found in [LZ10, CGF05]. As clearly pointed out by Hsu et al. [HFA05], a change of the current perspective is required: the focus must shift from performance-based metrics (such as the performance-power ratio) to new ones which take into account different aspects of the problem, i.e. integrating the notions of total cost of ownership, productivity and reliability. Wilde et al. [WAS14] complain that most of the research in the field of power-aware systems tends to focus on specific areas and does not try to combine its findings with the results obtained in different (but related) sectors. To overcome this situation they propose a comprehensive *Framework for Energy Efficient HPC Data Centers* that should be used by infrastructure managers to evaluate their systems from a holistic point of view. They identify four "pillars" establishing the basis of the evaluation: 1) Building Infrastructure; 2) HPC Hardware; 3) HPC System Software; and 4) HPC Applications. While most HPC centers already perform partial optimizations within each of the pillars,

[4]The rankings are based on the score obtained while running HPL [PWDC], a portable implementation of the high-performance LINPACK benchmark written in Fortran for distributed-memory computers

[5]This relationship is often called *Koomey's Law*

optimization efforts crossing the pillars' boundaries are still rare.

The power consumed by HPC systems is converted into heat, therefore, beside the IT power strictly needed for the computation, the additional power consumption of the cooling infrastructure must be taken into account. The extra infrastructure needed for cooling down the HPC systems has been proved to be a decisively limiting factor for the energy performance [Bel07]; a common approach taken to address this problem is the shift from air cooling to the more efficient liquid cooling. To further reduce the cooling cost HPC systems uses hot water recycling and free-cooling solutions [KRA12a]. Indeed when environmental conditions allow it, it is possible to decrease the computing units' temperatures through direct exchange of heat with the ambient. The amount of heat removed with this approach is proportional to the temperature gradient between the supercomputer outlet temperature (water or air) and the inlet ambient temperature. During cold days the gradient increases, enabling a larger heat portion to be removed without switching on the chillers. At the same time a hot internal temperature (i.e. hot-water cooling) increases the heat exchanged with the ambient.

A broadly used metric for power efficiency is 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. Equation 2.1 illustrates how PUE is computed, given that P_T is the total amount of power consumed by the whole HPC facilities and P_IT is the power consumption of the IT equipments.

$$PUE = \frac{P_T}{P_{IT}} \tag{2.1}$$

Nowadays PUE is by far the most widespread metric to evaluate the power efficiency of HPC systems, but it is not exempt from flaws. For example Yuventi et al. [YM13] argue that PUE is an instantaneous representation of electrical energy consumption that encourages operators to report the minimum observed values of PUE. Hence, PUE only conveys an understatement of the minimum possible energy use. As a fix they propose the use of different energy-based metrics or average PUE over a significant time period (e.g., a year).

One mainstream solution to reduce the gigantic power consumption is to employ efficient hardware or efficient design. By doing so, it is possible to obtain remarkable reductions of the PUE index. However, reducing the PUE is just a half of the problem. Data by McKinsey [McK] 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.

The rest of this section is devoted to describe some of the most commonly used approaches to create power-aware HPC systems.

2.1.3.1 Over-provisioning

Supercomputers' components such as CPUs and memory possess a vendor-specified Thermal Design Power (TDP) corresponding to the maximal power required by the subsystem. Generally, the maximum power consumption of a HPC system is determined by the sum of the TDP of its subsystems to take into account worst-case scenario where all components work at their TDP level. This design approach is called *worst-case provisioning* and can be optimal in contexts where power is plentiful and nodes are scarce resources; since power is now one of the limiting factors in HPC systems, worst-case provisioning became a drag on performance. An alternative solution comes from the observation that the system components rarely operate at their TDP limit. Recent advances in processor and memory hardware designs have made it possible to control the power consumption of the CPU and memory through software. The ability to constrain the maximum power consumption of the subsystems below the vendor-specified TDP value allows us to add more machines while ensuring that the total power consumption of the data center does not exceed its power budget, i.e. *over-provisioning* [SLGK14, PLS+15]. Over-provisioning means that not all the processing units in a supercomputer can run at their peak performance and power consumption. Hence, either only a subset of available computing units can execute at its full capability or all the computing nodes are active but at lower power levels.

Patki et al. [PLR+13] study how a policy of over-provisioning combined with hardware-enforced power bounds leads to better performance across a range of standard benchmarks. In particular, leveraging over-provisioning requires that applications use efficient configurations; the best configuration depends on application scalability and memory contention. The paper makes the (strong) assumption that the jobs are moldable; while this is true in the supercomputer used as a case study (and moldable jobs keep on getting more widespread), this cannot be taken for granted in all HPC systems, yet.

In [SLGK14], Sarood et al. describe an ILP model to enforce power capping in a HPC cluster through over-provisioning. Their approach combines over-provisioning with a power-aware scheduler. However, this work focuses on data centers and is based on assumptions that do not hold in typical HPC workloads. For example, the proposed method requires to change the number of nodes used by a job during its execution – malleable jobs. While this is very common in data centers, where virtualization and migration allow to move around and reshape the workload, in the majority of today's HPC environments this is not possible yet, since resources are locked to a job for its entire duration.

2.1.3.2 Energy Proportionality

Research in power and energy aware HPC systems has been a very active area in recent years and many different topics have been studied. Following the trend born in the servers field, current data center and HPC machines are moving towards the development of *energy proportional* systems [SF13, AMW+10, VG10, LCG+14]. The concept of energy proportionality implies that the power consumption should be proportional to the current workload: if the workload is low (not computationally intensive) the machine should consume little energy, and when the workload intensifies the power consump-

tion subsequently rises too. Most of the current supercomputers instead operate always at the maximal operational frequency, therefore at the maximum level of power consumption. Energy-proportional machines would present a wide dynamic power range and they would enable great savings in terms of energy and power [BH07a, Cam10].

Although energy proportionality was proposed as a measure orthogonal to over-provisioning, it is expected to reduce the energy-saving benefits of over-provisioning. Energy proportionality has the goal to reduce the energy waste caused by the partially utilized severs; energy-proportional computing systems strive to spend only as much energy as required by the given load. Theoretically, in an ideal energy proportional system the idle servers would consume no power. However, this clashes with the reality where idle resources keep on consuming, if they are not completely shut down. Since in an over-provisioned system we cannot turn off the nodes[6], the idle power consumption is an unresolved issue that prevent the realization of a complete energy proportional machine [VAG10].

2.1.3.3 Voltage & Frequency Scaling

In recent years a growing popularity has been associated to Dynamic Voltage and Frequency Scaling (DVFS) [HF05a, ECLV12b], a technique that trades processor performance for lower power consumption. With DVFS a processor can run at one of the supported frequency/voltage pairs lower than the nominal one. Lower frequency and voltage lead to significantly lower power consumption, allowing more jobs to run simultaneously without changing the power target. Several runtime systems that apply DVFS to reduce application energy consumption have been proposed [LFL06, KFL05, FL05]. These runtime dispatchers are often designed to take advantage of certain application characteristics, such as load imbalance or different computational phases, and therefore they may save power consumption only when applied to specific tasks. Usual DVFS approaches [LWW07, WC08] select the optimal power state (specified in terms of voltage or frequency) for each application, given the power budget available and the current power consumption, with the goal of decreasing the QoS for the users as little as possible

Previous studies [SMB+02, BM01] have shown clear gains in terms of power and energy savings when employing approaches based on varying frequencies and voltages. One of the main practical problems limiting the adoption in DVFS is the trade-off between reduced power consumption and increased execution time, which could be unacceptable in several contexts. Consequently, many research works try to exploit frequency reduction without impacting, as far as possible, the application duration. Rountree et al. [RLdS+09] developed a run-time system, called *Adagio*, to make DVFS practical for real-world scientific applications, with the aim of minimizing execution delays while obtaining significant energy savings. Their algorithm works at the application level: the key point is to consider the tasks composing the whole HPC job (analyzing internal MPI calls between the sub-tasks) and identify which tasks can be slowed down through frequency scaling without impacting the final job duration.

In other cases, in spite of the job runtime increase due to frequency scaling, overall performance (often measured with several, diverging metrics) may

[6] At least not without degrading the system performance due to the mandatory delay required to reboot a resource

improve with DVFS application thanks to shorter job wait times [ECLV12a, ECLV10a]. If we focus on the thermal power problem currently there exist a few insightful case-studies about the feasibility of employing DVFS to reduce the thermal power envelope of HPC machine nodes, thus improving reliability and diminishing the power spent for cooling while minimizing the impact on performance [FGC05, MGSG12]. A variant of frequency scaling is introduced by Gandhi et al. [GHBD+09]. Their technique works by alternating between high-performance state and low-power idle states on the execution nodes. This methods switches between extreme states, whereas more usual DVFS approaches utilize a wider range of possible frequencies. This approach has yet to be properly tested on a real supercomputer.

The problem of when and how to change frequencies in order to optimize energy savings is NP-complete and this has led to many heuristic energy-saving algorithms. Rountree et al. [RLF+07] present a method to estimate an optimal energy savings bound for a given MPI application using DVFS, using a Linear Programming model. The experiments on real HPC benchmarks show that existing techniques exploiting voltage and frequency scaling can work well for some programs, but for others little energy savings is possible.

Raghu et al. [RSB13] propose an ILP model to solve the scheduling & allocation problem with a focus on energy reduction and load balancing. Their model uses DFVS and relies on the knowledge of the optimal frequency/voltage combination for each application to be scheduled. This knowledge is obtained executing several offline runs of the considered HPC jobs, each run using different frequency and voltage values; in this way the approach builds a knowledge base. This knowledge base is then used at run time by the ILP to dispatch jobs in an optimal way (aiming at minimal makespan, load balancing and energy minimization). An obvious drawback of this technique is the requirement of a knowledge base that needs to be constructed offline and the issue of dealing with HPC jobs never seen before (and therefore not in the knowledge base).

The main issue with DVFS-based approaches is the trade-off between power savings and decrease in performance: reducing the operating clock clearly increases the duration of the applications which run on the slowed-down resources. To overcome this issue, several methods try to apply DVFS only in periods of low system activity or in particular phases of a job execution. For example, in [FLP+07], Freeh et al. study the energy-time trade-off of high performance cluster nodes with several power states available. They conclude that applying DVFS to applications with memory or communication bottlenecks does not imply large time penalties. A clear weakness of this strategy is that it strongly relies on the nature of the running applications, which must be known and modeled in advance, before their actual execution. Their analysis requires fine-grained monitoring of the execution of HPC jobs (i.e. run-time monitoring of the MPI calls), which is not available in most supercomputers and for all workloads. In [HF05b] Hsu et al. propose to solve this problem through a power-aware *adaptive* algorithm which does not employ any application-specific information a priori, but implicitly gathers such information. Unfortunately, this requires specific monitoring instruments to collect the needed data and these tools are not currently available in the majority of supercomputers.

A method to extend the EASY-backfilling algorithm with power budgeting capability through frequency scaling is discussed by Etinski et al. [ECLV10a]. The results show clear improvements in terms of energy savings and also a better

utilization of the system and reduced waiting time for the users, thanks to the possibility to execute more jobs concurrently if their frequency (thus power) is reduced. In [ECLV10b], Etinski et al. propose an approach which tries to minimize the performance loss by reducing the frequency only when the system is in a low level of utilization. They combine the EASY backfilling algorithm as scheduling policy with a frequency assignment module. When a new job is dispatched the system assigns an operational frequency to its execution node (such frequency will be kept for the whole job lifetime); this frequency may be lower than the maximal one depending on the current levelsystem utilization and power consumption. Their results show that it is possible to obtain good energy savings without degrading the overall performance of the system. The same authors present also a different approach in [ECLV12a]: in the latter work they propose a scheduling policy based on integer linear programming (ILP). Instead of relying on the backfilling algorithm the new scheduler decides which jobs need to be executed and the optimal frequency solving a optimization problem. This method offers better performance in terms of average job wait time over various power budgets, but it has the disadvantage of having been tested only on a relatively simple case-study and the corresponding ILP model cannot be directly extended to different kinds of problems.

2.1.3.4 RAPL-based techniques

An alternative to direct frequency scaling is Intels *Running Average Power Limit* RAPL [Cor09, DGH+10, RAdS+12], which provides a software configurable and hardware enforced power cap per CPU socket. While DVFS works by selecting a frequency or voltage among the range of possible ones, RAPL is a management interface that only requires the user to define a power threshold. The internal hardware then performs automatic frequency scaling and power throttling in order to keep the power consumption within the user-specified limit. RAPL employs an internal model of energy consumption to compute the average power consumption over a time frame and tries to enforce the power cap as precisely as possible. The main advantages of RAPL w.r.t. DVFS are the integration of power monitoring and control inside the chip and a finer granularity (compared to the finite set of frequencies of DVFS).

Methods using this mechanism usually employ simple scheduling and allocation policies (such as EASY-BF and First-Fit); at run time the power consumption is measured and kept under control. For example, Bodas et al. propose a power-aware scheduler [BSRH14] that decides the power available to each node of the system depending on the current power consumption and power budget. The efficacy of this method is limited by the simplicity and rigidity of the rules that assign the power to the nodes. Ellsworth et al. [EMRS15] present a more complex scheme to decide the power allocated to each node – which they call *Dynamic Power Sharing*. Initially the overall available power budget is uniformly divided among all nodes; periodically the algorithm adjusts the allocated power depending on actual consumptions, i.e. if a node consumes less power than the allocated one the excess capacity can be transferred to a different node which needs it. RAPL is used to enforce the node power limit at run time. The main drawback of these techniques is the same that troubles DVFS mechanisms, namely the indiscriminate power reduction implies an increase in job duration (performance loss).

2.1.3.5 Configuration Selection

Several research works try to exploit the possibility offered by moldable jobs. In particular, many works aim at minimizing power or energy consumption through *configuration* selection, that is deciding the best possible configuration of a job in terms of number of nodes and number of threads (and additionally also operating frequency). Such configuration is generally decided before the job execution and it is kept fixed at run time [BLR+14, SWAB15].

For example, Patki et al. [PLS+15] propose a job dispatcher operating on top of an EASY-backfilling scheduler and deciding the best configuration of the application to be run. The choice is based on the predicted power consumption and duration for each combination of application-configuration. While these techniques proved to be effective, their dependence on the moldable job model limits their adoption. Furthermore, deciding the best configuration requires to perform multiple offline runs of each application in all possible configurations, to understand the configurations power-performance trade-offs.

Marathe et al. [MBL+15] introduce *Conductor*, a run-time system that combines configuration selection with adaptive power balancing, with the goal of curtailing power consumption with minimal performance degradation. The configuration selection is dynamically performed at run-time and selects the optimal thread concurrency level[7] and frequency; an hardware power bound is then applied (RAPL). The power-balancing mechanism dynamically distributes the available power among all running jobs, assigning more power to critical paths. Critical paths are sets of tasks that should not be slowed down because doing that would increase the duration of the job composed by them. In order to dynamically detect critical paths the run-time system needs to be coupled with a fine-grained monitoring infrastructure that analyze the low-level MPI calls among sub-tasks.

Although over-provisioning aims at maximizing power efficiency for a given power budget, this method also has the disadvantage of excessive energy consumption, if the resource manager is not carefully tuned. Langer et al. [LDKeA15] examine how the careful selection of configurations can lead to significant savings in energy consumption with little impact on the execution time in over-provisioned data centers.

2.1.3.6 Idle Power Consumption & Workload Consolidation

Another issue in many supercomputers is the fact that computing nodes consume similar amounts of energy whether they are running an application or not – the idle power consumption is smaller than the active one but still much greater than zero (because inactive nodes are not completely powered off). In phases of low workloads some nodes (or their components) could be turned off in order to save power without incurring in severe performance losses. In [HHN08a] Hikita et al. implement an energy-aware scheduler with a simple strategy: when a node is not used for more than half an hour, the node is turned off. Despite the long time required by rebooting powered off nodes when needed (around 45 minutes), the scheduler manages to obtain great energy savings. Obviously, this solution is clearly strongly correlated to the considered system: other HPC

[7] Jobs can be seen as partially malleable: the number of used resources are fixed, but the number of threads can change during the execution.

centers would probably not allow their nodes to be idle for such long times. A more dynamic approach is described in [PBCH01]: load concentration is used to group the workload in few nodes in order to turn off the remaining ones. Again, even though this method obtains considerable power savings, it may not be directly applied to different supercomputers.

Although idle power consumption is a problem, the power consumed by active resources is vastly greater. Therefore using more nodes than needed (for example due to workload fragmentation) leads to non-optimal energy consumption. Hikita et al. [HHN08b] propose a method to tackle this issue. Their approach relies on workload consolidation: the scheduler tries to put as many jobs as possible in the same nodes (consolidation) and to turn off inactive nodes (or at least force them in idle state). Their analysis demonstrates that energy savings are possible, but there is a trade-off with QoS. In particular wait times tend to increase because the time required to re-activate a turned off node can be quite long.

Mammela et al. [MMB+12] present an energy-aware scheduler that can be applied to HPC systems without any changes to their hardware components. The idea is again to turn off idle nodes whenever possible, that is every time the scheduler detects that no activity can be scheduled for a sufficiently long time on a certain node. The main drawback of this approach is the fact that it strongly depends on the possibility to turn off idle nodes, which cannot be taken for granted in every HPC systems and usually has non negligible associated costs (i.e. if we turn off a node a long time may be required to boot it again).

2.1.3.7 Cooling Infrastructure

Common approaches for the design of the facility as well as for the PUE computation assume average or worst-case ambient parameters (temperature and humidity) as well as peak power consumption. However peak power consumption is a rare event, which may not happen for the entire lifetime of the supercomputer. Moreover this static design approach becomes suboptimal when dealing with free-cooling. Indeed, as described previously, in this circumstance the amount of IT power which can be removed without activating the chillers depends on the external temperature and humidity level, and thus varies across daily and night hours and seasons. The power consumption of the cooling infrastructure has a great influence on the overall system consumption, therefore many methods have been proposed to improve the cooling system energy efficiency. A lot of research effort have gone in the direction of improving the technology adopted by cooling infrastructure. Here in this work we are not going to discuss in detail innovative technical solutions or new hardware components because that would be out of our scope. We are instead more interested in understanding which strategies have been proposed for a better utilization of the supercomputer cooling infrastructure in conjunction with the workload management.

Current supercomputers cooling infrastructures are designed to withstand peak performance power consumption. However, everyday activity does not stress machines so hard and the typical workload is below the 100% resource utilization and also the jobs submitted by different users specify different computational requirements [YZ13a]. Hence cooling infrastructures are often over-designed. To reduce overheads induced by this cooling over-provisioning several

works suggest to optimize job dispatching (mapping plus scheduling) exploiting non-uniformity in thermal and power evolutions [BCTB13, KRA12b, KRAL13, KQC13] – yet most of these works are based on simulations and model assumptions and are not mature enough to be implemented on a HPC system in production.

Banerjee et al. [BMVG11] integrate a model of the cooling system in a power and energy aware job scheduler. The dispatcher is aware of the dynamic behaviour of the Computer Room Air Conditioner (CRAC) units and places jobs in order to reduce cooling demands from the CRACs. For example, it tries to evenly distribute the jobs among different servers not to overload any cooling unit. The proposed approach also manages at run-time the CRAC thermostat set point to reduce cooling energy consumption. The same authors also perform an analysis [BMVG10] of the performance of different allocation policies in terms of energy efficiency – after having modeled the cooling infrastructure. An alternative algorithm is then proposed (HTS), which places the jobs in the system nodes taking into account heat recirculation and the temperature requirements (maximal temperature allowed) of the computing nodes.

2.1.3.8 Resource Heterogeneity

Hardware heterogeneity as well as dynamic power management have started to be investigated to reduce the energy consumption. The idea is to limit the power consumed by a supercomputer exploiting the power variation which can be found across the computing resources of homogeneous [SWAB15] or heterogeneous [FBC+14] large-scale systems, in order to create energy and power aware schedulers.

Some authors tried to combine the possibility of adopting different configuration thanks to moldable jobs with the exploitation of power and performance variability among nodes and components within the same system. Shoukourian et al. [SWAB15] devised a power-aware configuration adviser that tries to dispatch jobs on a supercomputer minimizing the total energy consumption. The core aspect of the proposed approach is that different resource partitions have different performance and different power consumptions; thanks to previous offline experiments the adviser knows the best configurations for each application and for different set of nodes. At real time the configuration adviser integrates the knowledge about the optimal configuration with the information on the available resource partition and selects the best configuration for the specific situation. The main limitation of such methods is their reliance on system/component-specific characteristics, which prevents their use on different machines.

Wilde et al. [WASB15] discuss existing node power variations in two HPC systems. They introduce three energy-saving techniques: node power aware scheduling, node power aware system partitioning, and node ranking based on power variation, which takes advantage of the node variability. The results of their theoretical analysis show that using node power aware system partitioning and node ranking based power variation leads to energy savings with minimal negative impact on the system. The main limit of their study is the theoretical-only nature of the evaluation: even though the suggested techniques are interesting, no real implementations are tested.

2.1.3.9 Power Aware Workload Management

Power and energy consumption can be minimized also working on a careful planning of the activities to be executed. Changing the jobs execution order can have large impact on the power consumption of a system. For example, the cooling infrastructure requires more power when the external temperature is higher: the workload can be accordingly decreased during hot weather. In a similar direction, Yang et al. [YZW+13] propose an energy price aware scheduler that adapts the workload of the supercomputer depending on the current price of the electricity. The main idea is that during the day the cost of electrical energy is higher while it decreases at night; therefore a sensible policy is to move high power consumption jobs in off-peak periods (at night-time). The scheduling decisions are taken by a 0-1 Knapsack model which tries to assign low power consuming jobs during the day and high power ones during the night; a fair-share mechanism is also implemented to avoid incurring in job starvation. The main limit of this method is that it has been tested only on a simulator and the paper does not answer a fundamental question, namely how to compute (or estimate) the power consumption of a HPC application before its execution.

Zhou et al. [ZLTD13] extend the work previously done in [YZW+13] and devise a more sophisticated Knapsack model to schedule and allocate jobs in an energy-aware fashion. The rationale is the same as before: try to execute more power consuming jobs in off-peak periods of the day. In this case the objective is dual, minimize the energy consumption and maximize system utilization (try not too have many idle nodes) to cope with depreciation costs. The model is solved through Dynamic Programming.

Gómez-Martín et al. [GMVRGS15] discuss different energy aware resource allocation methods, with a focus on heterogeneous systems. The aim of the paper is to present a new simulator that considers performance and energy consumption and use this tool to evaluate currently used resource selection and scheduling policies. Finally, a new method based on a non-dominated sorting genetic algorithm (NSGA-II [DPAM02]) is proposed. The main advantage of the new algorithm compared to traditional techniques is the ability to deal with complex, multi-objective goal functions.

Job dispatching can be used to curtail the power consumption of a system – power capping – acting on the job execution order alone, without requiring any HW modification nor any change in the operational frequencies of the computing nodes [WYV+16]. In this case the dispatcher needs to have information about the power consumption of the tasks to schedule in order to make the correct schedule decisions. Since these power information are needed at schedule time, before the execution of the job we cannot count on direct power measurements but we should rely on power consumption estimates – methods to obtain such estimations will be discussed later. Since we are dealing with estimates, bad predictions must be taken into account. Hence, the software-based power capping approach may not exclude the hardware one, because the latter can still be required in case of misprediction. In general hardware control can prevent power limit violations but usually at a significant performance cost [POF+15]. Therefore, the best usage of the software techniques is to create the best possible schedules, given the current information, such as to avoid as much as possible the triggering of hardware capping countermeasures (i.e. avoiding the "emergency brake"). The combination of hardware and software technique, in

conjunction with real time power monitoring to reduce the number of violations of the power constraint, has been examined in preliminary studies [BSRH14]

Probably, the simplest way to obtain a power-constrained scheduler acting only on jobs execution order is to modify the common EASY-backfilling algorithm (EASY-BF) [MF01]. The base algorithm tries to fit as many jobs as possible on the system (selecting them from a FIFO queue); a job fits in the system if enough resources are available. In order to add power awareness it is sufficient to consider the power as an additional resource – a job fits in the system only if its power consumption will not cause the overall power to exceed a given budget.

2.1.3.10 Compilation-based Methods

Another different approach to control the power consumption of HPC systems is based on the possibility offered by modern compilers. The application binaries produced by compilers from the source code can be optimized in order to consume the smallest possible amount of energy [BTW14, FKM+11, ST03]. Whereas in the past compilers have focused mainly (very often exclusively) on the performance, i.e. running as fast as possible, there is a growing interest in considering the energy impact on the final executable generated by modern compilers. As is often the case there is again a trade-off between optimizing for maximal performance or minimal energy consumption [KVIY01].

2.1.3.11 Power Capping

A standard approach to limit the amount of power consumed by HPC systems is *power capping* [LWW08, FWB07, RCC12, KZA+12], which means bounding a supercomputer not to exceed a certain value of consumed power (respecting a power "cap"). This approach somehow encompasses and can exploit all the methodologies illustrated so far. Power capping can be implemented through several strategies. Today's most employed power budgeting mechanisms rely on the hardware components capacity to operate at different frequencies and therefore with different power consumptions. The main idea is to limit the computing nodes performance when the total power consumption gets closer to the critical threshold [CHCR11, VAN08]. The power-curtailing action is generally initiated by the resource manager in combination with the job dispatcher, which dynamically collects information on the current power consumption and decides accordingly the amount of power to allocate to each computing node [Ben12].

A problem to consider when implementing power capping methods is the variability in terms of power and performance which can be observed among nominally equivalent processing units (and other hardware components). Modern processors suffer from increasingly large power variations due to the chips manufacturing process [MHJI07, Nas04]. Only recently these power variations and their implications for HPC systems with a power budget have been recognized and studied. For example, Inadomi et al. [IPI+15] analyze the manufacturing variability on four supercomputers with different micro-architectures and its impact on their HPC applications; they then propose a variation-aware power capping strategy with the goal to maximize applications performance. The key point is to determine optimal module-level power allocations to ensure performance homogeneity; frequency scaling proved to be the most effective

mechanism to implement the proposed policy.

A very interesting analysis of different power capping strategies can be found in [PMW+15]. Petoumenos et al. compare different approaches (ranging from DVFS to forced idleness, passing through compiler-based optimization) on the same system and try to establish which one can provide the best performance-power saving trade-off. The results indicate that DVFS is generally the most effective technique despite the weakness of a limited granularity w.r.t. RAPL (DVFS is constrained by the finite and small number of different operational frequencies that can be chosen). However, their study does not conclude that one method is always better than the others. Keeping that in mind, the authors encourage mixed approaches which could benefit from the best characteristics of each method; for example, combining frequency scaling with forced idleness provided the best overall results on the tested system.

2.2 Constraint Based Job Dispatching

In the previous section we discussed the High Performance Computing context and in particular we described the job dispatching problem in HPC systems. In this section we are going to define this problem in a more detailed way and we are going to show how it can be included in the broader category of the so called Scheduling & Allocation problem, a staple of optimization research and practice. A great part of the section is devoted to introduce the Constraint Programming paradigm which is a fundamental enabling technology for the solution approaches presented in the rest of this work (see in particular Chapters 3 and 5).

A Scheduling & Allocation problem consists of allocating finite resources to activities over time [BT13]. Hence, finding a solution means assigning a start time and a set of resources to each activity, while respecting a set of constraints (i.e. to model the finite resources) and trying to optimize an objective. This problem has been the subject of a huge number of studies from different research fields and it is referred to also as Resource Constrained Project Scheduling Problem (RCPSP). The problem of dispatching jobs in a HPC system can be naturally framed as a scheduling & allocation problem: a supercomputer offers a set of finite resources (shared or exclusive) and the dispatching problem consists in assigning a starting time and deciding a resource partition for each activity to be executed.

Constraint Programming (CP) is a programming archetype with a long history of success with scheduling & allocation problem and thus is an ideal candidate to solve the job dispatching problem in the High Performance Computing context. The general scheduling & allocation problem is also referred to as *Constraint-Based Scheduling* with the nomenclature adopted by the CP community [BPN01a]. We are now going to introduce the Constraint Programming methodology and then discuss its application to the resource constrained scheduling problem.

2.2.1 Constraint Programming

Constraint Programming is a programming paradigm belonging to the Artificial Intelligence (AI) area. It deals with Constraint Satisfaction Problems (CSP),

usually defined by a triple $\langle X, D, C \rangle$, where X is a set of variables, D a set of domains and C a set of constraints. Each variable X_i may take a value $v \in D_i$ from its associated domain; each constraint C_j defines (and limits) the values that can be assumed by a set of variables, the *scope* of the constraint – denoted as $S(C_j)$. The scope of a constraint specifies the set of variables it can be applied to; the constraint is said to be posted on $S(C_j)$. A solution to the CSP is an assignment of the variables compatible with every constraint.

A CP *model* thus is defined by variables, domains and constraints; a *filtering algorithm* is associated to every constraint C_j and its purpose is to remove certainly infeasible values from the domain of the variables in its scope. The filtering process can be seen as a way to infer additional constraints from the existing ones. When a domain D_i is modified by a constraint, such change may lead other constraints involving the same variable to filter out more values – this mechanism is called *constraint propagation*. Many algorithms were proposed to perform this propagation in the most efficient way [Bes94]. Usually this process proceeds until a fixed point is reached, i.e. it is impossible to filter out further values from each involved domain.

Typically CSP problems are solved through tree-search; each branching decision activates the filtering algorithm and constraint propagation, actively reducing the search space (often performing much faster than complete enumeration). However, the filtering and propagation mechanisms alone are not sufficient to find a solution in all the situations. First, if a problem possesses multiple solutions filtering and propagation cannot choose between them (by definition filtering and propagation do not arbitrarily remove feasible values). Furthermore, eliminating all infeasible values in a CSP is, in general, as complex as solving the problem itself; in practice one needs be satisfied with some kind of local consistency [Bar05]. Several kinds of local consistencies have been proposed; the most widely adopted are the *arc consistency* (AC), *generalized arc consistency* (GAC), *Bound Consistency* (BC) and *K-Consistency*. Local consistencies define properties that the constraint problem must satisfy after constraint propagation. An extremely well written discussion on constraint propagation and local consistencies can be found in [Bes06]. For the reasons exposed above, a crucial aspect for all CP methods is the definition of a *search strategy*, which is used to find the actual solutions.

Summarizing, in order to solve a CSP using a CP approach it is necessary to define two key components: a model and a search strategy. Thus:

$$CP = model + search$$

Modeling and search can be seen as "independent", i.e. a model could be solved exploiting many search strategies and vice-versa a single search strategy could be used with different models.

2.2.1.1 Modeling in CP

Constraint Programming provides a very expressive language which allow the creation of compact yet expressive models. The high level of versatility offered by CP is one of its most useful aspects. There are almost no restrictions on variables and constraints. For example, the variables range from classical types like integer and real to less common ones, like set-variables [Ger94] and interval variables [LR08, Lab09b] (as we will see, these are extremely useful

in scheduling problems for modeling activities). Each class of variables allows several kinds of constraints, from the simple unary linear constraints to more complex ones. For example, we can find linear mathematical constraints as $X_i + X_j \geq X_k - 2$ or non-linear ones, $X_i \neq X_j, X_i = \min\{X_j, X_k\}$, etc. We can also have constraints specifically tailored for certain variable types, i.e. the $non\_overlapping(X_i, X_j)$ constraint for interval variables which imposes that two activities X_i and X_j must execute in different time periods, with no overlap between them. Another possible constraint type is represented by meta-constraints (or *reified* constraints) used within mathematical or logical expressions, i.e. $(X_i = 0) \geq (X_j \neq 1)$.

2.2.1.2 Global Constraint

A very interesting (and widely used) group of constraint goes by the name of *global constraints* [BCR05, Rég04, Sim96]. A global constraint is posted on a set of variables and encapsulates several homogeneous constraints. For example the well known `aldiff` is posted on a set of variables $X_0, X_1..X_n$ and imposes inequalities among all of them.

$$\forall X_i, X_j \quad \text{with} \quad i \neq j : \quad X_i \neq X_j$$

Very powerful filtering algorithms have been devised for every global constraint. One of the strongest motivation to use global constraint is that the filtering algorithm associated with them is stronger than the conjunction of the independent filtering algorithms of the local constraints corresponding to the global one. This is due to the fact that filtering is a local mechanism, because it is associated to each constraint independently. If a constraint is decomposed in a set of non-global constraints, the filtering algorithm for each sub-constraint has fewer knowledge about the overall problem and therefore has a less effective pruning capability. For example, suppose we have this simple CSP on finite domains:

$$\begin{aligned} &X_0 \neq X_1, X_1 \neq X_2, X_0 \neq X_2 \\ &X_0 \in \{1, 2, 3\} \\ &X_1, X_2 \in \{1, 2\} \end{aligned}$$

If we enforce Arc Consistency no value can be excluded from the variables domains, because for each variable X_i all values v have a support in the domain of X_j, for every single constraint $X_i \neq X_j$. However, if we look at all the constraints at the same time, we can notice that, in order to obtain a feasible solution, values 1 and 2 must be assigned to X_1 and X_2. Thus, the domain of X_0 can be reduced to $D_0 = \{3\}$. This kind of global-level domain pruning can be extremely effective, especially with large size problems.

In addition, global constraint are very expressive and allow more compact models. Global constraints (and their powerful filtering algorithms) are one of the key strengths of Constraint Programming. Many and diverse kinds of techniques have been used to implement filtering algorithms for with global constraints [vHK06]. For a complete survey of the (many) existing global constraints we refer to [RÍ11].

2.2.1.3 Search in CP

Constraint Programming allows to use a wide selection of search strategies; these techniques may differ greatly and have a strong impact on the resolution of CSP problems. However, the majority of search methods falls in one of three main classes, Backtrack Search, Local Search and Dynamic Search. For a thorough survey of search strategies in CP see [VB06, RVBW06].

In this section we focus on the most commonly used search strategy in CP problems, *Backtrack Search* (BS). A backtracking search to find a solution to a CSP can be seen as performing a depth-first traversal of a search tree. Backtrack search works by posting constraints on variables, activating propagation at each decision step (a node in the search tree) and then backtracking on failures. Constraints are used to check whether a node may possibly lead to a solution of the CSP and to prune subtrees containing no solutions. The branches are often ordered using a heuristic, with the left-most branch being the most promising. To ensure completeness, the constraints posted on all the branches from a node must be mutually exclusive and exhaustive.

The term *branching strategy* indicates the choice of the kind of constraint to post at each search node. With an *enumeration* branching strategy at each decision point we select a variable and generate a branch for each possible value, posting a set of unary constraints $x_i = v, \quad \forall v \in D_i$. If *binary choice point* is adopted one value from its domain is assigned to the selected variable, posting $x_i = v$ on a branch and $x_i \neq v$ on the other branch. Finally in *domain splitting* the variable is not necessarily instantiated, but rather the choices for the variable are reduced in each subproblem, using two branches with constraints as $x_i \geq \gamma$ and $x_i < \gamma$.

In all branching strategies a criterion must be specified to decide which variable must be selected at each search node and which values must be assigned. A commonly employed criterion is the so called "First-Fail Principle" [HE80] which recommends to select the variable more likely to generate a failure. A common implementation of this heuristic is to select at each decision step the variable with the smaller domain [GB65]; in case of ties in the selection, the choice could be based on the degree of the variable, i.e. the number of constraints in which it is involved [Bré79]. Another general strategy is presented in [Ref04] and it is called Impact Based Search: the key idea is to associate to each variable its impact, i.e. the relative reduction of the search space it can provide, if a certain value is assigned. The impact are learned from the observation of domain reduction during search.

Another way to exploit information gathered during the search process are *no-goods* [SS77, KB05, Bac00], sets of assignments and branching constraints that are not consistent with any solution. The goal is to learn from previous failures, deducing their cause and obtaining an explanation. If we are using a backtracking search, each dead end corresponds to a no-good. Thus no-goods are the cause of all futile search effort. Once a no-good for a dead end is discovered, it can be ruled out by adding a constraint. Several works studied how to effectively find no-goods, both using the static structure of the CSP [Bru81, Dec86, Dec90] and dynamically during the search [Gin93, Pro93, SV94].

It has been observed that backtracking algorithms can be unstable on some instances, since seemingly small changes to a variable or value ordering heuristic can lead to great differences in running time. An explanation for this phe-

nomenon is that heuristics make mistakes. Depending on the number of mistakes and how early in the search the mistakes are made (and therefore how costly they may be to correct), there can be a large variability in performance between different heuristics. To address this problem, two usually adopted techniques are *restarts* and *randomization*. Randomization consists in introducing some random element either when selecting the variable and/or when choosing which value to assign; a useful introduction to the topic can be found in [Gom04]. A randomized search method is usually restarted after some time – to prevent bad random choice to ruin the overall performance. Different restarting strategies have been proposed for this purpose [LSZ93, Wal99].

Finally, alternatives to depth-first search have been proposed. For example, Limited Discrepancy Search [HG95], which can be seen as a sort of best-first search strategy, since the first nodes to be explored are those deemed "better" by a certain heuristic. When the search does not follow the values ordering heuristic and does not take the left-most branch out of a node we have a discrepancy. Least Discrepancy Search considers earlier the nodes with fewer discrepancies. A generalized approach of this technique is called Decomposition Based Search [vHM04]: using a heuristic the values of each variable are classified as "good" or "bad". At first only good values are considered and during the search the variable are allowed to assume an increasing number of bad values.

2.2.2 Modeling a scheduling problem with CP

In this section we discuss a Constraint Programming model for scheduling problems. We deal with a simplified version of the general scheduling problem for the sake of clarity. We consider only non-preemptive activities with a fixed durations. "Non-preemptive" means that once an activity started it cannot be interrupted. The resources available are finite (also called "cumulative" in the literature) and additive, i.e. the total resource requirements of multiple activities executing concurrently is the sum of the every single task requirement.

Scheduling problem over a set of activities $A = \{a_0, a_1, ..\}$ and resources $R = \{r_0, r_1, ..\}$ is usually modeled in CP with the introduction of three integer variables for each task (see [BPN01a]):

1. ST_i is the activity start time (when the activity begins its execution)
2. ET_i is the activity end time (when the activity ceases its execution)
3. D_i is the activity duration (the number of time instants required to complete an activity)

These three variables must always satisfy the constraint $ST_i + D_i = ET_i$. Here we are dealing with fixed durations hence $D_i = d_i$ (where d_i are fixed values). Conventional names are used to define bounds for start and end variables:

- $\min(ST_i) = EST(a_i)$ - Earliest Start Time
- $\max(ST_i) = LST(a_i)$ - Latest Start Time
- $\min(ET_i) = EET(a_i)$ - Earliest End Time
- $\max(ET_i) = LET(a_i)$ - Latest End Time

EST and LET may be forced by users to specify release time and deadline on activities. To represent precedence constraints we use linear constraint between start and end times of different activities $ST_i \geq ET_j$. The limitations derived from the finite resources are usually expressed through a global `cumulative` constraint [AB93] posted on each resource $r_k \in R$:

$$\texttt{cumulative}([ST_i], [D_i], [rq_{ik}], C_k)$$

where $[ST_i]$ and $[D_i]$ are the vectors of start times and durations variables, $[rq_{ik}]$ are the resource requirements and C_k is the capacity of resource k. The cumulative constraint enforces that at each point in time, the cumulated requirements of the set of tasks that overlap at that point, does not exceed a given limit. Formally, if we assume all ST_i variables range between 0 and eoh (End Of Horizon), the `cumulative` constraint enforces:

$$\forall i \in A, \forall k \in R, \forall \tau = 0, .., eoh : \sum_{ST_i \leq \tau < ST_i + D_i} rq_{ik} \leq C_k \qquad (2.2)$$

In every time point τ the sum of requirements $[rq_{ik}]$ of running tasks ($ST_i \leq \tau < ST_i + D_i$) on resource k must not exceed its capacity C_k. In Figure 2.2 we can see the resource consumption profile of a set of five activities allocated on a resource of capacity of nine units. In this example the cumulative constraint holds since at each point in time we do not have a cumulated resource consumption strictly greater than the upper limit.

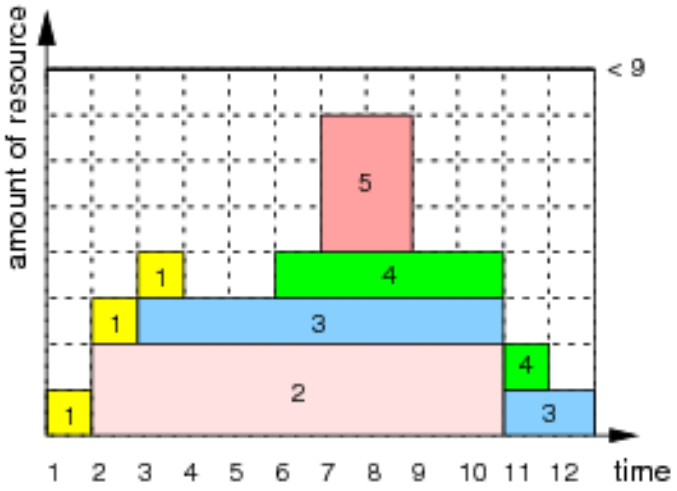


Figure 2.2: Example of resource consumption profile. Source Global Constraint Catalog [glo15]

A solution for the scheduling problem (a *schedule*) is a feasible assignment for every ST_i and ET_i variable. To summarize, 2.3 shows a quite used simple CP model used for scheduling problems. The objective function $F(A)$ is not specified since there are many different objectives to consider, some of them mentioned in the Section 2.2.3.

$$
\begin{array}{rll}
\min: & F(A) & \\
\text{s.t.}: & ST_i + D_i = ET_i & \forall a_i \in A \\
 & ET_i \leq ST_j & \forall a_i \prec a_j \\
 & \texttt{cumulative}([ST_i], [D_i], [rq_{ik}], C_k) & \forall r_k \in R \\
\text{with}: & ST_i, ET_i \in \{0, ..., eoh\} & \\
 & D_i = d_i &
\end{array}
\tag{2.3}
$$

2.2.3 Objective Functions

This section describes some objective functions commonly used in scheduling problems. Probably the most frequently used objective is the *makespan*, the total *completion time* which is defined as the worst completion time among all the activities to be scheduled:

$$mks = \max_{a_i \in A} ET_i \tag{2.4}$$

The makespan can be also considered as the total length of the schedule and the general goal is to minimize it. Generally, in HPC settings there is a correlation between the utilization of the systems, i.e. the percentage of resources that are not in an idle state, and the makespan, since when the former increases the latter decreases – a system with higher utilization rate is executing more jobs in parallel and will probably complete earlier the workload execution.

If activities are subjected to deadlines dl_i they may incur in penalties if they fail to finish within a certain time. We therefore define the *tardiness* of an activity as $Tr(a_i) = \max(0, dl_i - ET_i)$. It is also possible to associate a *weight* w_i to each activity which reflects the fact that it could be worse to exceed the deadlines of some jobs rather than others, depending on their priorities. This situation is very common in HPC contexts, since different jobs and different users may have different needs in terms of when a task must be completed – i.e. it could be worse exceeding the deadline for a short and urgent task rather than long ones. The goal in this case could be the minimization of the maximum weighted tardiness [Jac55]:

$$max_{WT} = \max_{a_i \in A} Tr(a_i) \tag{2.5}$$

or to minimize the total weighted tardiness:

$$tot_{WT} = \sum_{a_i \in A} Tr(a_i) \tag{2.6}$$

Alternatively activities may receive penalties if they start before a certain date (*earliness* cost); the objective function definition is analogous to the tardiness

case. It is obviously possible to combine both tardiness and earliness in the objective proceeding in a similar fashion [KH06].

Finally, there may be interest in minimizing the number of jobs exceeding their deadlines (*number of late jobs* [Moo68]); this can be modeled considering that a job is late if $ET_i > dl_i$:

$$nlj = \sum_{a_i \in A} [[ET_i - dl_i > 0]] \tag{2.7}$$

These objective functions are all *regular*, i.e. there is no benefit in delaying an activity towards the end of the project[8]. Generally, CP is better suited to deal with objective functions defined as the maximum over a set of expressions, as in the case of makespan or maximum tardiness. This is due to the fact that in this case any constraint on the objective value is effectively back-propagated on all the domains of the involved activities. Conversely CP tends to performs worse with functions involving sums due to poor propagation.

2.2.4 Filtering for Cumulative Constraints

As mentioned when introducing Constraint Programming, in order to decrease the time needed for the search it is very important to devise effective techniques to reduce the search space. This is done in CP by reducing variable domains through filtering algorithms. Every type of constraint has its own filtering method and cumulative constraints are no exception. In fact, many filtering algorithms for resource constraints have been studied and implemented. We are going to see three of the mainly used filtering techniques; for a complete survey of propagation methods on resource constraints see [BCP08, BP07, Lab14, LP$^+$05]. In the remaining part of this section we assume again activities durations to be fixed ($D_i = d_i$); note that without this assumption the proposed algorithms would still work by assuming $d_i = \min(D_i)$.

Time-Table filtering Time-table (TTB) propagation techniques [LP94] compute and maintain during the search the aggregated demands profile (the sum of every activity requirement) at each time interval τ. This information allows to restrict the domains of the start and end times of activities by removing the dates that would necessarily lead to an over-consumption of the resource.

More in detail, whenever an activity a_i shows the condition $LST(a_i) \leq EET(a_i)$ we can be certain that such activity will run (and consume required resources rq_{ik}) during any time interval $\tau \in [LST(a_i), EET(a_i)]$ – this is called an *obligatory region*. This information forces an update in the time-table global data structure and the aggregated demand for the resource is increased by rq_{ik} at time τ. We denote the aggregated resource consumption for a resource k at time τ as $RQ_k(\tau)$. Since the time-table structure changes only on modifications of LST and EET the access to $RQ_k(\tau)$ with binary search requires $O(log|A|)$ (A is the set of activities). Iterating over the entire structure takes $O(|A|)$. The cumulative demand is defined as:

$$RQ_k(\tau) = \sum_{LST(a_i) \leq \tau < EET(a_i)} rq_{ik} \qquad \forall k \in R \tag{2.8}$$

[8] More precisely: when two resource feasible schedules have been constructed such that each activity under the first schedule starts no later than the corresponding starting time in the second schedule, then the first schedule is at least as good as the second schedule

Obviously, if a time τ exists such that $RQ_k(\tau) > C_k$ (C_k resource capacity), the current schedule cannot lead to any solution and search must backtrack. In addition, if there exists an activity a_j requiring rq_{jk} amount of resource k at time τ_0 such as:

1. $EET(a_j) \leq \tau_0 < LET(a_j)$
2. $\forall \tau \in [\tau_0, LET(a_j)) : RQ_k(\tau) + rq_{jk} > C_k$

then activity a_j cannot end after date τ_0, since otherwise there would be an over-usage of resource k. Thus τ_0 is a new valid upper bound for ET_j.

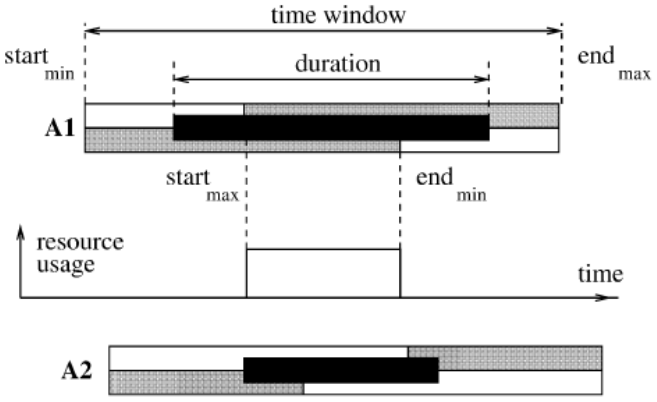


Figure 2.3: TTB propagation example

In Figure 2.3 we can see an example of time-table filtering: activity $A1$ has an obligatory region ($LST(A1) \leq EET(A1)$) and the resource usage is not equal to zero. Conversely activity $A2$ has no obligatory parts, thus it does not enable any propagation. The main advantage of this technique is its relative simplicity and its low algorithmic complexity. It is the main technique used today for scheduling discrete resources. Unfortunately, time-table filtering propagate nothing until the activities' time windows become so small that some time instants are necessarily covered by some activity. Furthermore this method does not exploit the existence of precedence constraints between activities.

Edge Finder filtering Edge-Finding methods [Pin91] reason about the order in which activities execute on a given resource. A complete discussion of these methods is out of the scope of this work hence we refer to [BP96] for a thorough review of existing edge-finding approaches, while here we simply hint the main idea. On unary resources edge-finding algorithms detect situations when an activity A must execute before (or after) a set of activities Γ. We can then draw two types of conclusions: new ordering relations ("edges" in the graph representing the possible orderings of activities) and new bounds for start and end time variables. If Γ is a subset of activities on a unary resource and $A \notin \Gamma$ another activity on the same unary resource, $EST(X)$, $LET(X)$, $Dur_{min}(X)$ are the minimal start time, maximal end time and minimal duration over all activities in a set X, the typical edge-finding technique relies on the kind of implication described by the following expressions.

Let (1) be:

$$LET(\Gamma \cup A) - EST(\Gamma) < Dur_{min}(\Gamma \cup A)$$

and (2):

$$ET(A) \leq \min_{\Gamma^S \subseteq \Gamma} (LET(\Gamma^S) - Dur_{min}(\Gamma^S))$$

then the edge-finding rule is (1) $\Rightarrow$ (2)

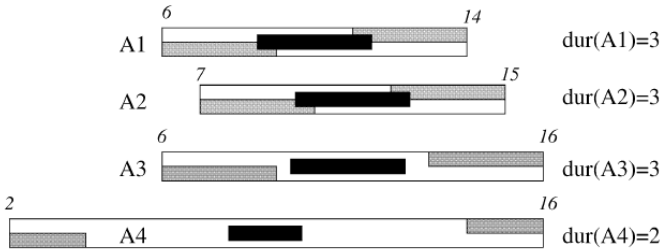


Figure 2.4: Edge-finding propagation example

In Figure 2.4 we can see an example of edge-finding propagation. If we consider $A = A_4$ and $\Gamma = \{A_1, A_2, A_3\}$ the condition in (1) are satisfied ($LET(\Gamma \cup A) = 16$, $EST(\Gamma) = 6$ and $Dur_{min}(\Gamma \cup A) = 11$). The algorithm would propagate and compute a new upper bound for A_4: $ET(A) \leq (16-9) = 7$ (taking $\Gamma^S = \{A_1, A_2, A_3\}$)

The cumulative case is more complicated due to the many activities that may overlap on a cumulative resource. In this case edge-finding methods use "energy based reasoning" to detect relations between activities, where the energy is defined as the product of duration times amount of required resource. Edge-finding algorithms are very powerful but their non negligible computational complexity represents a severe drawback [BPN01b].

Energetic Reasoning Energetic reasoning [ELT91, LEE92] consists in comparing the amount of resource energy required over a time interval to the total amount of energy available over the same interval. Similarly to the edge-finding technique energetic reasoning works on the time bounds of the activities, removing invalid values when possible.

Again let us consider an unary resource (but the method could be extended to the cumulative case). Energetic reasoning key point is to find pairs of activities (A, B) such that if A precedes B this would lead to a dead end, because the resource could not provide enough energy between $EST(A)$ and $LET(B)$ to execute A, B and all the other activities that necessarily execute on the same time interval $[t_1, t_2)$. More precisely let C be an activity and $[t_1, t_2)$ a time window; the energy required by C in such time window is:

$$E_C^{(t_1,t_2)} = \max(0, \min(EET(C) - t_1, t_2 - LST(C), Dur(C), t_2 - t_1))$$

Then if this condition holds:

$$LET(B) - EST(A) < Dur(A) + Dur(B) + \sum_{C \notin \{A,B\}} W_c^{LST(A),LET(B)}$$

we know that A cannot be scheduled before B, hence it must go after it. This propagation allows to modify the time bounds for both $LET(B)$ and $EST(A)$.

Energetic reasoning (like edge-finding) is very effective when dealing with pure scheduling problems but still presents the same drawbacks of time-tabling. Their weakness is to consider only the absolute position of activities over time (their time-bounds) rather than their relative positions (the precedence relations between them). Hence, the propagation starts only when time windows are sufficiently small. Furthermore, propagation may be very limited when the current schedule contains many precedence constraints.

2.2.5 Search Strategies in Scheduling Problems

This section introduces two of the most adopted and well-known searching strategies tailored for scheduling problem.

Schedule or Postpone Pape et al. [PCVG94] proposes *schedule-or-postpone* tree-search strategy which tends to produce *active* schedules, i.e. schedules where no activity can be left shifted. The schedule-or-postpone strategy is complete if the objective function is regular (non decreasing in the end time of the activities) because in this case there would always be an optimal active schedule [SKD95][9]. At each node of the search-tree this method selects an activity a_i and open a choice point: on the left branch a_i has its start time set at $EST(a_i)$, on the right branch a_i is marked as non-selectable until its earliest start time is modified by propagation – i.e. *postponed*. Generally, the criterion used to select an activity among all the available ones at each decision point is to prefer the activity with the smaller earlier start time, using latest end times as tie-breaker.

Precedence Constraint Posting *Precedence constraint posting* is a search strategy (also called PCP) that aims at resolving conflicts which may arise when many activities try to access the same finite resources by adding additional precedence constraints. This idea has been proved to be very effective when applied to CP context [LG95,Lab05,PCOS07]. In particular Laborie proposes a technique [Lab05] that relies on the identification and consequent resolution of Minimal Conflict Sets (MCS). A MCS for a resource r_k is a subset of activities such that:

$$\sum_{a_i \in MCS} rq_{ik} > C_k \tag{2.9}$$

$$\forall a_i \in MCS : \sum_{a_j \in MCS \setminus \{t_i\}} rq_{jk} \leq C_k \tag{2.10}$$

$$\forall a_i, a_j \in MCS \quad \text{with} \quad i < j : ST_i < ET_j \vee ST_j \leq ET_i \quad \text{is consistent with the current state} \tag{2.11}$$

where a_i are activities and rq_{ik} their requirements on resource k of capacity C_k; ST_i and ET_i are, respectively, start times and end times of the activities. Eq. 2.9 requires the set to be a conflict (resource over-usage), Eq. 2.10 is the

[9] Unless particular precedence constraints are used

minimality condition (if any activity in the set were to be removed, the conflict would not exist) and Eq. 2.11 says that the activities may overlap. The MCS resolution requires to impose a single precedence relation between two activities in the set; the complete search consists in selecting a MCS at each decision node and branching on all its possible resolvers (the different precedence relations) in the children nodes until there are no more MCSs.

2.2.6 Decomposition Techniques

A lot of research over the last decades produced CP algorithms very good at solving small or medium sized instances; the main strengths of CP are strong inference methods and powerful search heuristics. The former is very apt at quickly detecting any infeasibility while the heuristics are used to guide the search towards areas in the search space which will likely contain solutions. Constraint Programming deals with optimization problems using branch and bound techniques, with the cost represented by an objective variable – new solutions are required to have lower costs than the previous ones (through constraint placed on the objective variable). However these methods often risk to be ineffective when tackling larger problem instances. For instance, in the case of allocation & scheduling under constrained resources in HPC systems we may have to deal with problems formed by thousands (or even more) of variables, depending on the number of resources hosted in the machine (as told before we can have supercomputers with tens of thousands processing units) and on the number of tasks submitted by users (up to thousands per day in larger systems).

The sheer complexity of many problems renders even the most advanced CP methods ineffective unless an alternative strategy is applied. In order to face this issue, in the last few years many works have studied approaches that effectively decompose and solve large scale problem, i.e. the original problem is divided in sub-problems whose individual solutions can be found with relatively lower computational effort. Decomposition techniques have been proved to be useful also in smaller scale (but hard) problems too, often because while there are no efficient way to cope with the original problem its components can be solved with specialized and effective algorithms. For example, Benini et al. [BBGM05] describe a method to deal with allocation and scheduling in Multi-Processor Systems on Chip. The key idea is to decompose the problem in the scheduling component, solved through Constraint Programming, and the allocation component solved with Integer Linear Programming.

An important component in many decomposed approach is the so called *Local Search*. This term represents a large class of meta-heuristics commonly used in computationally hard optimization problems [GH06, BPW$^+$12, MA04, AEOP02, LMS03]. The typical approach used in Local Search starts from an initial feasible solution and then proceeds to apply small changes to the solution in order to produce a new one; if the new solution is better than the older one it becomes the new current solution. After that, new local moves are iteratively applied to further explore the solution space until a fixed point or a time limit is reached. The main drawback of this approach is given by the local nature of the moves: if the current solution is a local minimum (given that we are dealing with a minimization problem), moves to neighbours solutions may never be able to escape such local minimum and the algorithm could not find possible global minima. To overcome this limitation many extension have been proposed, such

as introducing randomization [NW07, Len97] or preventing to go back to the previous solution even if it was better than the new one (Tabu Search [GL97]). Delving in the depths of local search is outside the scope of this work but the interested reader can find an interesting starting point in this book by Aarts et al. [AL97].

2.2.6.1 Benders Decomposition

A very common decomposition techniques well suited for scheduling & allocation problems is the *Benders Decomposition*. The classical Benders Decomposition [Geo72, BM91] method decomposes a problem into two loosely connected subproblems. It enumerates values for the connecting variables. For each set of values enumerated, it solves the subproblem that results from fixing the connecting variables to these values. The solution of the subproblem generates a Benders cut that the connecting variables must satisfy in all subsequent solutions. The process continues until the master problem and subproblem converge providing the same value. The idea is to "learn by mistake" and the use of Benders cuts accomplish the goal of eliminating superfluous solutions. The classical Benders cut is a linear inequality based on Lagrange multipliers obtained from a solution of the subproblem dual. The typical Benders approach, however, requires that the subproblem is a continuous linear or nonlinear programming problem.

Logic-Based Benders Decomposition LBBD [HO03] is an extension of the traditional scheme that enables generic solvers to be used as subproblem solvers. LBBD can be applied to any class of optimization problem but a method to generate Benders cuts must be identified for each different class of problems – and this is usually not a trivial task. LBBD has been applied to numerous application, in particular it had a great success with planning and scheduling problems [Hoo07], and it has been also used in conjunction with Constraint Programming [EW01]. LBBD is used in [FZB09] to solve a location-allocation problem, i.e. deciding where to locate a set of facilities and allocate clients to them, and the results show that it outperforms the traditional ILP approach. LBBD has also been applied to scheduling problems [Hoo05a, Hoo05b, TB12]; for example Canto uses LBBD [Can08] to solve the problem of scheduling preventive maintenance activities in a power plant.

A great advantage of LBBD is the possibility to use heterogeneous techniques to solve the master and the subproblems and this can lead to very efficient solutions. The main limitation of LBBD-based approach lies in the difficulty to generate effective cuts, which in turns leads to a inefficient exploration of the search space and a slow convergence towards the optimal solution, especially with large-scale problems. Another disadvantage of LBBD is the risk of loosing valuable information (e.g. if master and problem variables are connected by tight constraints) when we decouple master and subproblems.

2.2.6.2 Large Neighborhood Search

A method extensively studied in the literature and very appreciated for its practical effectiveness is *Large Neighborhood Search* or *LNS* [Sha98], a framework that combines the search power of CP with the scaling performance of local search. As in local search, we start from an initial solution of the problem

(which can be found through standard CP search or faster heuristics) and then we modify it. However, instead of making small changes to the solution, as is typical with local search move operators, a subset of variables from the problem are selected. These variables are then unassigned while the remaining ones are locked to their values in the current solution, then the search for an improving solution happens through reassigning only the unassigned variables. The search strategy used for finding a new assignment for the relaxed variables can be chosen in order to best fit the problem taken in consideration. There are three crucial aspects affecting the quality of LNS: 1) the fragment selection procedure (which are the variables that will be relaxed), 2) the fragment size and 3) the search limit – the stopping criterion applied to the search of the new solution with the relaxed variables. The best methods to address these points have not been decided yet and they are subject of a lot of research efforts.

LNS has proved to be a very effective tool for solving complex optimization problems, however applying LNS to real world problems still requires a great deal of problem domain knowledge since heuristics to select effective neighborhoods must be discovered for each problem class. Carchrae et al. [CB09] show how to reduce the required expertise using adaptive techniques to create algorithms that adjust their behaviour to suit the problem instance being solved. With a similar purpose, Laborie et al. [LG] present an approach called Self-Adapting Large Neighborhood Search, which combines Large Neighborhood Search with a portfolio of neighborhoods selection and completion strategies together with Machine Learning techniques to converge on the most efficient way to solve the target problem. The re-enforcement learning scheme, although quite simple, ensures a quick convergence on the most effective neighborhoods, search strategies and their associated parameter values and is a key factor in the robustness of the approach.

Godard et al. [GLNI] use a LNS technique to solve cumulative constrained scheduling problems, where resources may execute several activities in parallel, provided the resource capacity is not exceeded. It relies on a general approach based on calculating *partial-order schedule* from a fixed start time schedule. A partial-order schedule, POS, for a problem P is a graph where the nodes are the activities of P and the edges represent temporal constraints between pairs of activities, such that any possible temporal solution is also a consistent assignment [PSCO04]. In the context of LNS, POSs provide a very powerful way to inject flexibility into the schedule while keeping interesting features from one solution to the other. Danna et al. [DP03] consider the job-shop scheduling problem with earliness and tardiness costs, i.e. jobs should be scheduled exactly at certain times in order not to incur in "penalties". The paper compares two approaches for dealing with this problem, one being a LNS based method tailored for the problem, the other being a form of LNS called Relaxation Induced Neighborhood Search which is a generic and *unstructured* algorithm, relying only on a continuous relaxation of the Mixed-Integer Programming model of the problem to define its neighborhood.

Palpant et al. [PAM04] propose a technique to overcome scale issues in resource scheduling problem. They introduce a method which combines local search with subproblem exact resolution (LSSPER). The method can be seen as a hybrid scheme: each step fixes a subpart of the current solution while the other part defines a subproblem solved by a heuristic or exact solution approach. The key factor of the method is the choice of the subproblem to be optimized

and the paper analyzes several different strategies for this task.

2.2.6.3 Adaptive Randomized Decomposition

A different approach targeted at large scale problems is Adaptive Randomized Decomposition (ARD), proposed by Bent and Van Hentenryck in [BH07b,BH10] to tackle large scale vehicle routing problems. Its goal is to find a set of decouplings, i.e. subproblems which can be independently optimized and whose solutions can be merged back in an existing solution to produce a better one; the choice of algorithms for optimizing the sub-problems is independent from the ARD scheme. Pacino et al. [PH11] study the effectiveness of LNS and ARD methods when dealing with a constraint-based scheduling problem, in particular the flexible job-shop, a generalization of the traditional job-shop scheduling where activities have a choice of machines. In this work LNS uses random, temporal and machine neighborhoods, while ARD takes advantage of 1) temporal (consider the activities running within a time window) and 2) machine (select the activities executing on a subset of the machine) decouplings to generate subproblems.

Simon et al. [SCVH12] employed ARD to tackle the problem of the restoration of the electrical power system after significant disruptions caused by natural disasters. The problem is very challenging since it combines routing and power restoration components. Previously, a 3-stage decomposition was used but the routing sub-problem proved to be a bottleneck, therefore in the paper a new Vehichle ARD is proposed and it is shown to provide good results while addressing problems of considerable size (several thousands of components).

2.3 Machine Learning

The field of Machine Learning (ML) is a branch of Artificial Intelligence with the scope of building computer programs that automatically improve with experience [Mit97]. Machine Learning draws on concepts and findings from many different areas, for example statistics, philosophy, information theory, biology, cognitive science, computational complexity, and control theory. ML methods have proven to have great practical value in many different application domains, ranging from data mining in large databases that might contain valuable patterns not easy to see by a human eye to improve the performance of machines and automated systems. They have proven to be very effective in a large number of contexts, including computer vision, speech recognition, document classification, automated driving, computational science, and decision support.

ML algorithms use sets of data in order to extract information or generate knowledge bases. Datasets are represented using the same set of features, that can be continuous, categorical or binary. If the instances forming a data set have corresponding known labels (the corresponding correct output) the learning is called *supervised* (Section 2.3.1). Conversely, *unsupervised* learning deals with unlabeled instances (Section 2.3.2.1); the application of unsupervised algorithm can lead to the discovery of unknown properties or classes among the unlabeled data. Another type of learning is *reinforcement* learning (Section 2.3.2.2): the training input provided to the learning model by an external trainer (the environment) has the aspect of a reinforcement signal, a measure of how well the

N.	*Temperature* (°C)	*Forecast*	*Windy*	*# Friends*	*Class*
1	24	Sunny	True	10	P
2	13	Cloudy	False	8	P
3	34	Sunny	False	4	N
4	26	Sunny	False	3	N
5	0	Rain	True	12	N
6	28	Cloudy	False	9	P

Table 2.2: Training Set Example

system is operating. The learner is not commanded to take certain actions but rather it must discover the actions generating the best reward, with a trial-and-error mechanism.

Many books provide thorough and comprehensive introductions to ML and its application, among them [MCM13, RNC+03]. The rest of this Section discusses several techniques employed to solve ML problems.

2.3.1 Supervised Learning

Supervised learning is the machine learning task of inferring a function from labeled training data consisting of a set of training examples. The key hypothesis of supervised methods is the *inductive learning hypothesis* [Mit97]:

> Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

In this case the learning scheme involves acquiring general concepts from training examples. The process of concept learning can be seen as a search through a large space of hypotheses; the goal of the learning is to find the hypothesis that best fit the training examples. A detailed survey of supervised techniques and algorithms can be found in [KZP07].

Training data, forming the *training set*, comprises instances of training examples. Each example is a pair composed by an input object (typically a set of features) and a desired output. Supervised learning algorithms analyze the training data and infer a function, i.e. the relationship between the set of input features and the output, which can be used to determine the labels of new data. In order to do so, the algorithms must generalize from the training data to unseen situations. An example of training data set can be seen in Table 2.2. Let assume we want to learn if a particular Thursday night is apt for playing football. The data set is composed by instances of Thursday nights characterized by a set of attributes: the temperature (a numerical value), if the night is windy or not (values $True$ or $False$), the weather forecast (*sunny*, *clouds* or *rain*), number of friends in town (a numerical value). The concept to learn is the fact that the Thursday night is suited for playing football (class P) or not (class N).

Numerous ML applications can be dealt with the supervised learning scheme. For example, a very common task for supervised learner is *classification*, i.e. identify the class – or type – of an unclassified object, depending on the learned

classification function. A high-level approach for solving the classification problems consists in finding a "simple" classifying rule with good performance on the training data [HK16].

2.3.1.1 Decision Trees

A very commonly used technique to approximate discrete-valued target functions are *decision trees* [Qui86, RM05, SL90]. Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node of the tree performs a test on some attribute of the instance and each descending branch from the node corresponds to a possible value of the attribute. Decision trees can be applied to instances that can be represented as attribute-value pairs. The data set can contain errors and the missing data. Figure 2.5 displays an example of decision tree (related to the data set of Table 2.2).

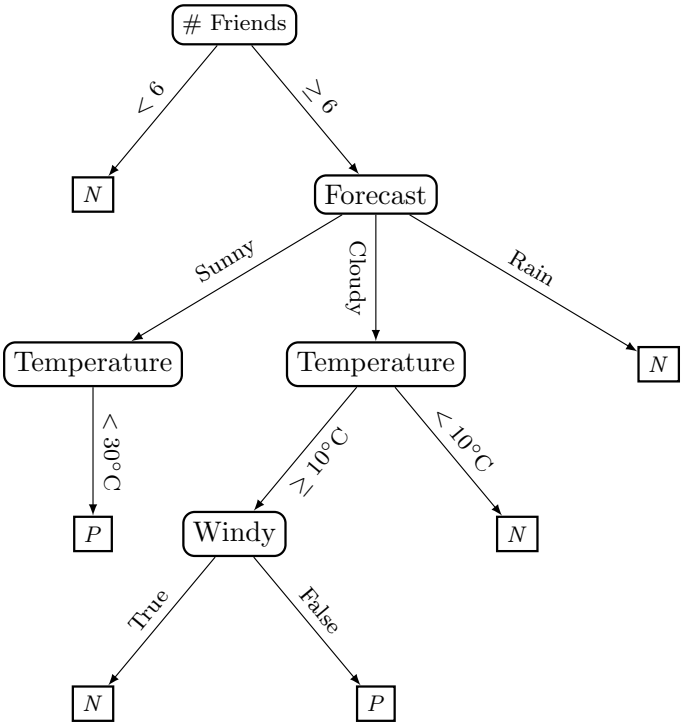


Figure 2.5: Decision Tree Example

The most common type of decision trees requires the target function to assume only discrete values (distinct classes), thus performing classification. There are extensions of decision tree learning that are capable to deal with real values [BFSO84, WW96, OW03, XWVA05] – *regression* trees. Broadly speaking, most of the regression tree techniques differ from decision trees only in having values rather than classes at the leaves. Nevertheless, decision trees tend to perform better when dealing with categorical features and outputs.

The canonical algorithm in the literature for building decision trees is the C4.5 [Qui14] proposed by Quinlan as an extended version of an earlier algorithm ID3 [Qui79] from the same author. Quinlan also proposed an extended version of C4.5 in order to tackle the issue of continuous output, the algorithm M5 [Q+92]. While decision trees have values at their leaves, M5 generate "model" trees that represent multivariate linear models. Model trees tend to be smaller than regression trees. Many other approaches rely on the core algorithm defined by ID3 and C4.5; the core consists of a top-down, greedy search through the decision trees space. Starting from the tree root an attribute is selected and a statistical test is performed to evaluate the impact of the attribute on the classification. For each possible value of the selected attribute a descendant node is created and the training set instances are directed in the appropriate node (down the branch corresponding to the instance value of the attribute). The process is repeated using the example in the training set and descending the tree, each time selecting the best attribute at the current tree level. At each node the attribute is selected through a heuristic and the decision cannot be backtracked.

A crucial component of the algorithm is to detect which are the most "informative" attributes, the attributes with higher relevancy in order to classify an instance and therefore those that should be tested first – ideally at the root of the decision tree. A statistical property called *information gain* has been introduced to measure how well a given attribute can separate the data set instances according to their target classification. To understand the most widespread metric for information gain the concept of *entropy* must be introduced. Entropy is a concept derived from information theory and it measures the average amount of information necessary to identify the class of an example in a data set [Jay57, CT12, And08]. Given a set S with negative and positive examples and the class C_j the entropy is measured as:

$$Entropy(S) = -\sum_{j=1}^{K} \frac{freq(c_j, S)}{|S|} \times log_2 \left(\frac{freq(c_j, S)}{|S|} \right) \tag{2.12}$$

where the target attribute can assume K different values (classes); $freq(c_j, S)$ is the frequency of examples of class j in the set (i.e. the number of examples of the class divided by the total number of examples). Entropy can also be seen as a measure of the unevenness of collection of examples: higher entropy corresponds to a more variegated set. In the case of two classes, given the proportions of negative and positive example p^- and p^+ ($p^- = 1 - p^+$) the equation can be rewritten as:

$$Entropy(S) = -p^+ \times log_2 p^+ - p^- \times log_2 p^- \tag{2.13}$$

Having introduce the entropy, the information gain produced by a test on an attribute A is the expected reduction in entropy caused by partitioning the examples according to this attribute:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Vals(A)} \frac{S_v}{S} Entropy(S_v) \tag{2.14}$$

where $Vals(A)$ is the set of possible values for attribute A and S_v is the subset of S composed by those examples whose attribute A has value v. Note that the

second term is the expected value of the entropy after S is partitioned using attribute A. Information gain is precisely the metric used by ID3 and C4.5 to select the best attribute, giving preference to attributes with higher information gains.

A decision tree, or any learned hypothesis h, is said to *overfit* the training data if another hypothesis h' exists that has a larger error than h when tested on the training data, but a smaller error than h when tested on the entire dataset. For example, an hypothesis listing only positive examples of the training set is equivalent to a rule that memorize the training sample, thus having a very small (null) error on the training set. The drawback is that said rule could predict the class of an example if and only if the example appeared already in the training set. Consequently the error on the entire data set would be much greater. More generally, overfitting is a concern because algorithms will typically be optimizing over the training sample. The two most common approaches to tackle overfitting are: 1) stopping the training algorithm before the point when the learned model perfectly fits the data; 2) pruning the induced decision tree [BKK+98]. Several analysis have been made to identify the best pruning methods [BA97, Bru00, Elo99].

A commonly appreciated aspect of decision trees is their high human readability; it is possible to look at a decision tree scheme and understands why the learning model classifies a certain instance as belonging to a certain class. Another great advantage of decision trees is their ability to deal with incomplete information, i.e. instances with missing feature values. One simple strategy for dealing with a missing attribute value is to assign it the value that is most common among training examples at the tree node [Min89]. A more sophisticated strategy consists in assigning a probability to each of the possible values that the attribute can assume. These probabilities are computed using the observed frequencies of the various values among the training set instances. This is the strategy adopted by C4.5.

2.3.1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a robust and generalized approach to approximate real-valued, discrete-valued and vector-valued target functions [Hop82]. ANNs proved to be extremely effective to deal with a large variety of problems, such as pattern recognition [Bis95, Fuk88, CG88, Rip07], handwritten character recognition [LBD+89, RMS89, SBB+92], face recognition [LKL97, LGTB97], stock market prediction [KAYT90, GKD11], image compression [DH95] and many others.

The study of ANNs has been partially inspired by the observation of how the neurons organize their structure in tightly connected networks in biological systems such as the human brain. Artificial neural networks are composed by densely interconnected simple units that take a number of real-valued inputs (possibly coming from other units) and produce a single real-valued output. There exist different type of fundamental units that serve as building blocks in ANNs; one common kind is the so called *perceptron* [Ros62], depicted in Figure 2.6. The perceptron takes as input a vector of N real values X, calculates a linear combination of the input and then generates an output $O(X)$ that can be 0 or 1, depending on the linear combination being smaller or higher than a threshold.

$$O(X) = \begin{cases} 1 & \text{if } \sum_i^N x_i w_i \geq w_0 \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

where w_i is a real value called *weight* determining the contribution of the input x_i; w_0 represents the threshold.

Single perceptrons can be used as linear classifiers. The linear combination of weighted inputs is taken as input by the activation function which is triggered only when its input exceeds a given threshold. Commonly used activation functions are non-linear functions such as sigmoid $\sigma(x) = 1/(1 + e^x)$, tanh $tanh(x) = 2\sigma(2x) - 1$, rectified linear unit (RelU) $ReLU(x) = max(0, x)$ and many others. The single perceptrons can represent the primitive boolean functions like AND, OR, NAND and NORA. Since *every* boolean function can be represented through combinations of these primitives all boolean functions can be expressed using a two levels deep network of perceptrons, where the second stage collects the output of multiple first-stage units.

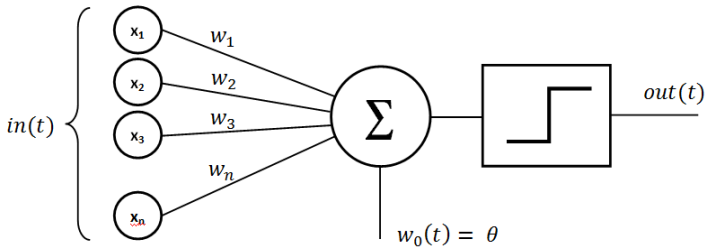


Figure 2.6: Example of ANN. Source: `https://github.com/cdipaolo/goml/tree/master/perceptron`

Perceptrons can only classify linearly separable groups of instances. Linearly separable means that drawing a straight line or plane on all input instances is sufficient to distinguish those belonging to the target class. If the instances cannot be separated in this fashion, a classifier based on single perceptron will never be able to classify them. In order to overcome the limitations of single perceptrons, artificial neural networks are modeled as collections of neurons that are connected in an acyclic graph – the outputs of some neurons can become inputs to other neurons [RHW85]. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Often, ANN are organized in separate layers of neurons; typical ANN topologies have an initial layer (*input* layer), a final layer (*output*) and one or multiple intermediate layers (*hidden* layers). An example of basic ANN can be seen in Figure 2.7. Multi-layered ANNs where the information flows from the input layer to the output layer with no cycle allowed are also called *feedforward* networks. Feedforward networks containing three layers of units are able to approximate any function to arbitrary accuracy, given a sufficient (potentially very large) number of units in each layer [Cyb88, Cyb89].

The first step of creating a ANN consists of training the model in order to determine the input-output mapping. In the learning process the weights of the connections between the neurons are updated until they reach the correct value;

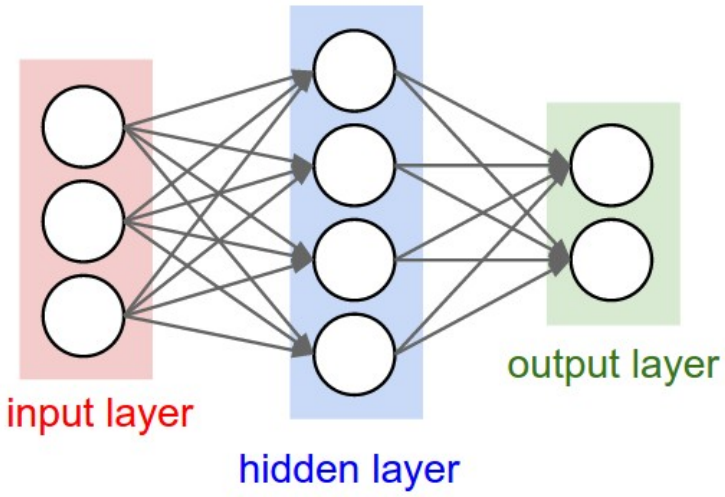


Figure 2.7: Example of ANN. Source: `http://cs231n.github.io/neural-networks-1/`

after the training stage the weights are fixed. Afterwards, the network can be used to generate an output given a vector of real values as an input, for example performing classification tasks.

The most commonly used algorithm to train ANNs is the *backpropagation* algorithm [RHW88, CR95]. The core steps of the backtracking algorithm are the following. 1) Give a training sample $< X, T >$ to the ANN as input and compare the generated output with the expected result; compute the error in each output neuron. For each output unit k the error δ_k is computed with the formula: $\delta_k = o_k(1 - o_k)(t_k - o_k)$, with o_k the output of the neuron and t_k the target outcome. 2) Propagate the error "backwards" to hidden layers. The error formula for hidden neurons h is $\delta_h = o_h(1 - o_k) \sum_k w_{kh}\delta_k$; the error assigned to hidden neurons depends on the errors received by the output neurons, weighted by the weights of the connections between them. 3) Update each network weight $w_{ji} = w_{ji} + \Delta_{ji}$ where $\Delta_{ji} = \eta\delta_j x_{ji}$. x_{ji} is the input value to which the weight is applied and η is the learning rate. The weight-update loop in backpropagation may be iterated a huge number of times, therefore several different termination conditions can be used to halt the procedure. One may choose to stop after a fixed number of iterations, or once the error on the training examples falls below some threshold, or after the error measure has not improved after a certain number of iterations or once the error on a separate validation set of examples meets some criterion – to keep overfitting in check.

Over the years a lot of research effort has been put in order to improve the efficiency of this fundamental component of ANN-based learning [LBOM98,LK90, MVA99, RB93, VON92]. For example Wang et al. [WTT+04] propose a novel backpropagation algorithm aimed at avoiding the local minima problem caused by neuron saturation in the hidden layer. The main point of the new method is to adapt the activation functions in order to prevent saturation in hidden layer neurons. Some approaches address the weaknesses of backpropagation, such as the risk of overfitting [TLL95, LG00]. Schittenkopf et al. [SDB97] describe a

strategy to avoid overfitting in two-layered networks. They use two additional linear layers and principal component analysis to reduce the dimension of both inputs and internal representation; in this way less significant neurons and better generalization are obtained. Giles et al. [Gil01] show that increasing the number of hidden units and applying backpropagation with early stopping leads to ANN able to generalize well. This is due to the excess capacity of hidden layer that allows better fit for regions of high non-linearity. Early stop guarantees that the increased net will not be over-trained and therefore overfitted.

Artificial Neural Networks are powerful instruments capable of approximating universal functions. One of the main drawback of ANN is their lack of "transparency": ANNs behave as black boxes and once a network has been trained is not always trivial (or even possible) to understand the criteria it uses to produce a certain output given a set of inputs. Another problem with ANN is choosing the correct number of hidden layers: underestimating the number of neurons can lead to poor approximations while too many neuron nodes can result in overfitting and overall complicates the training phase. A very good analysis on the right number of hidden layers and neurons can be found in [CY01,KP00]. Even networks of practical size can represent a large number of nonlinear functions, making ANNs a powerful instrument for learning discrete and continuous functions whose general form is unknown in advance.

2.3.1.3 Statistical Learning

Statistical approaches for Machine Learning rely on an underlying probabilistic model providing the probability that an instance belongs to each class, rather than a simple classification. The most common types of statistical ML techniques are based on *bayesian inference* [Jen96, Ric97, Gha01, Nie08] founded on the computation of posterior probabilities as described by the Bayes Theorem. Bayesian methods can be used to determine the most probable class of an instance. A Bayes classifier combines the predictions of all classes, weighted by their posterior probabilities, to obtain the most probable classification of each new instance. Bayesian methods are often called *bayesian networks*.

In ML we often want to know the best hypothesis (or class) given a certain set of observed data D. The Bayesian inference allows us to compute the most probable hypothesis h given the observed data and any initial knowledge about the prior probabilities of all hypotheses in the hypotheses space H. Bayes theorem provides a way to calculate the probability of a hypothesis following the probabilities of observing various data given the hypothesis. The Bayes theorem states:

$$P(h|D) = \frac{P(D|h)P(H)}{P(D)} \tag{2.16}$$

$P(h)$ is the prior probability of the hypothesis, i.e. the probability the hypothesis holds before having observed any data; $P(D)$ indicates the probability that training data D will be observed (given no knowledge about which hypothesis holds); $P(D|h)$ represents the probability of observing data D if hypothesis h is true. $P(h|D)$ is called the *posterior* probability and in ML defines the probability that the hypothesis holds given the observed data. In typical learning scenarios the learner desires to know which hypothesis (from a set of candidates) is the most probable one, given the observed data. This hypothesis is defined as the *maximum a posteriori* (MAP) hypothesis. Bayes theorem can be used to

compute the posterior probability of each hypothesis and the MAP hypothesis h^{MAP} is going to be the output of the learning model – i.e. a class.

$$h^{MAP} = \arg\max_{h \in H} P(h|D) \tag{2.17}$$

$$= \arg\max_{h \in H} \frac{P(D|h)P(h)}{P(D)} \tag{2.18}$$

$$= \arg\max_{h \in H} P(D|h)P(h) \tag{2.19}$$

$P(D)$ can be dropped because is independent from h.

Naive Bayesian Classifiers (NBC) are a widely adopted Bayesian learning method. NBCs have been found useful in many practical applications [Mur06, Ris01, Leu07, MN$^+$98, DXYY07, TCWX09]. A bayesian classifier applies to learning tasks where each instance is defined by a tuple of attributes $a_1, a_2, ...a_n$ and one target value v_j that can take only one value from a set V. The adjective "naive" refers to the simplifying assumption that the attribute values are conditionally independent given the target value [Goo50, Nil65]. This means that given the target value v_j of an instance, the probability to observe the conjunction of attribute values $a_1, a_2, ...a_n$ is the product of the probabilities of the individual attributes: $P(a_1, a_2, ...a_n|v_j) = \prod_i P(a_i|v_j)$. With this assumption the problem of finding the most probable hypothesis h^{MAP} (Eq. 2.19) can be reformulated as the problem of finding the most probable value v^{MAP}:

$$v^{MAP} = \arg\max_{v_j \in V} P(v_j|a_1, a_2, ..a_n) \tag{2.20}$$

$$= \arg\max_{v_j \in V} \frac{P(a_1, a_2, ..a_n|v_j)P(v_j)}{P(a_1, a_2, ..a_n)} \tag{2.21}$$

$$= \arg\max_{v_j \in V} P(a_1, a_2, ..a_n|v_j)P(v_j) \tag{2.22}$$

$$= \arg\max_{v_j \in V} P(v_j) \prod_i P(a_i|v_j) \tag{2.23}$$

The greatest advantage of the naive Bayes classifier is the short time required for training. Conversely, the major flaw of the NBC is the crucial assumption on the independence of the attributes $a_1, a_2, ...a_n$ given the target value v_j. This assumption drastically limits the complexity of the learning function. *Bayesian Network* (BN), also called *Bayesian Belief Networks*, are an extension of the simpler NBC born with the goal of overcoming this limitation [N$^+$04, Hec98, Cha91, WMOSI12, HMW95]. A bayesian network describes the probability distribution of a set of variables. In contrast to NBCs which assume that all the variables are conditionally independent given the value of the target variable, BNs apply conditional independence only among *subsets* of the variables.

Bayesian networks can described with acyclic directed graphs where the arcs represent causal influences among the input features while the lack of possible arcs encodes conditional independences. The standard learning process of a BN is composed by two subtasks: first, the graph structure must be learned and secondly its parameters are computed. Probabilistic parameters (representing the relationships between the attributes) are typically expressed in a tabular format, one table for each variable. Thanks to the independence granted by the network computing the joint distribution is just a matter of multiplying the

tables. When training a BN there could be two scenarios: the network can be already known, for example provided by an expert, or the network structure must be discovered as well. The computational cost of the second scenario is very high and therefore many approximate methods have been proposed, ranging from greedy algorithms [HMC06, Chi02] to local search approaches [AdC03]. Madden [Mad03] performed a very interesting and thorough analysis of several different methods to train bayesian networks.

The most power features of BNs (compared to decision trees or ANNs) is the possibility to consider prior information about a problem, in terms of relationships among its features. The biggest issue of BN is related to the computational complexity of creating and learning a previously unknown network. The process of network discovery is a NP-hard problem [CH92], which could be too costly to perform or impossible if the number of variable and combinations grows too large. Another weak point is the inherent dependency on the quality of the prior information: a BN is useful only if the prior knowledge is reliable. Wrong prior probabilities would distort the entire network and invalidate the results.

2.3.1.4 Instance-based Learning

So far we have seen learning methods that aim at building a general and explicit description of the target function (the relation between the input features and the target output). Instead, *instance-based* learning methods simply gather training examples and the generalization from these examples is postponed until a new instance needs to be classified [Aha97, DMA98]. For these reasons these methods are also referred to as "lazy-learning" algorithms [Mit97] because the learning process is delayed until needed. Instance-based methods compute "local" approximations of the target functions (applied only in the neighbourhood of the new instance to be classified) and never derive a target function capable of working over the entire instance space. This is an advantage when the target function is extremely complex but can be still described by a collection of simpler local functions.

One of the most common and most used instance-based algorithms is the *k-nearest neighbour* algorithm (*k-NN*). The cornerstone of the *k-NN* algorithm is the observation (assumption) that instances within a dataset are in close proximity with other instances with similar properties [CH67]. If the instances in the train set are labeled, the *k-NN* classifier assigns the class to new instances depending on the most common label in the neighbourhood. The k parameter determines the number of neihgbour instances to be included in the computation. A key point is the way used to establish if two instances are neighbours; in order to do that a metric that measures the distance between two instances must be introduced. Generally, instances can be seen as points in a n-dimensional space where each one of the n-dimensions corresponds to one of the n-features used to describe an instance. The distance $D(x, y)$ between two points/instances x and y can be then computed with the following equation:

$$D(x, y) = \sqrt{\sum_{i=1}^{m} |x_i - y_i|^2} \tag{2.24}$$

where m is the number of instance features. This distance metrics is also referred to as *Euclidean distance*. Many different measurement types are possible and

adopted on different *k-NN* implementations. For more precise results, many algorithms employ weighted schemes: for example the contribution of each k neighbour is weighted by its relative distance from the instance to be classified. Wettschereck et al. [WAM97] survey many weighting policies adopted in instance-based learning methods.

k-NN algorithms have some weaknesses. First, they are very sensitive to the choice of the metric used to measure the distance between two instances [SF81, BGRS99]. Secondly, there is no error-proof criterion to choose the right k (the size of the neighbourhood), other than cross-validation or other computationally-expensive techniques [GWB$^+$03, WNC06, WS09]. The majority of instance-based local classifiers such as *k-NN* are not completely lazy because, for example, the neighbourhood size k (or other parameters) is usually chosen by cross-validation on the training set, which can require significant preprocessing and risks overfitting. Garcia et al. [GFGS10] propose instead a completely lazy approach that requires no preprocessing: instead of having a fixed neighbourhood of size k they demonstrate that using the average local probabilities over a set of neighbourhoods performs similarly to cross-validation. They also show that this process can also be seen as a Bayesian way to estimate the neighbourhood size.

A key advantage of instance-based approaches is that instead of learning a single target function for the whole instance space these techniques estimate the function at a local level, differently for each new instance to be classified. Lazy-learning methods require less computational time during the training phase (the training basically consists of storing example instances) but more computation time during the classification phase. This can obviously be a drawback and therefore techniques for efficiently indexing training examples are a practical solution to reduce classification time. A possible method to decrease the computation time to classify new instances is to reduce the number of features to consider, using only the strictly necessary ones [KJ99, YL04]. Other approaches aim instead at reducing the number of stored instance via instance-filtering algorithms [KCJ01, WM00, WM97]. Jahromi et al. [JPJ09] try to make *k-NN* more efficient addressing the problem from two different directions. On one hand, they propose a weighting policy for the data set instances in conjunction with a learning algorithm that attempts to maximize the classification accuracy by adjusting the weights of the training instances. On the other hand, the proposed weighting policy serves also to reduce the number of considered instances because those whose weight is lower than a given threshold are deleted from the training set.

2.3.1.5 Support Vector Machines

Another approach to deal with supervised machine learning is offered by *Support Vector Machines* (*SVM*) [Vap13, Bur98, CST00, SS04]. Let assume we have labeled training data $\{x_i, y_i\}, y_i \in \{-1, 1\}, x_i \in \mathbf{R}^d$. The basis of SVMs lays on the notion of "separating hyperplane", a hyperplane in the space $\mathbf{R}^d$ that separates the positive from the negative examples. Let d_+ and d_- be the shortest distance from the separating hyperplane to the closest positive and negative examples. The difference $d_+ - d_-$ is called the "margin" of the separating hyperplane. If the data set examples are linearly separable the support vector algorithm searches the separating hyperplane with the largest margin (see

Figure 2.8). Creating the largest possible distance between the separating hyperplane and the instances on either side of it has been proven to reduce an upper bound on the expected generalization error [VK82].

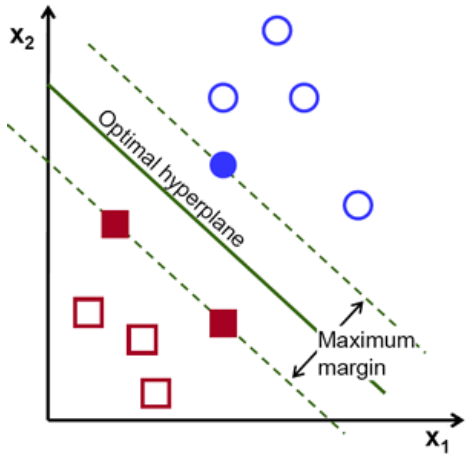


Figure 2.8: Example of separating hyperplane. Source: `http://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html`

The problem of finding the hyperplane with the maximal margin can be formulated in this way:

$$x_i \cdot \boldsymbol{w} + b \geq 1 \qquad \forall y_i = +1 \tag{2.25}$$

$$x_i \cdot \boldsymbol{w} + b \leq -1 \qquad \forall y_i = -1 \tag{2.26}$$

These equations can recombined in one set of inequalities:

$$y_i(x_i \cdot \boldsymbol{w} + b) - 1 \geq 0 \qquad \forall i \tag{2.27}$$

The solution in the two-dimensional case has the form depicted in Figure 2.8. The training points for which the equality of Eq. 2.27 holds are called the support vectors (solid shapes in Fig. 2.8); removing them would cause the solution to change. The problem can be reformulated with a Lagrangian reformulation (see [Bur98] for a detailed explanation); we therefore introduce a set of Lagrangian multipliers $\alpha_i, i = 1, .., l$ one for each inequality constraints in Eq. 2.27. We finally obtain the Lagrangian:

$$L_p \equiv \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{l} \alpha_i y_i (x_i \cdot \boldsymbol{w} + b) + \sum_{i=1}^{l} \alpha_i \tag{2.28}$$

This is a quadratic convex problem and can be solved through standard algorithms, for example using the dual representation of the problem [Fle87]. In the end the solution will be a linear combination of the support vectors.

The mechanism described above cannot be directly applied to the case of non-separable data, introduced, for example, by misclassified instances. The problem can be addressed by using a soft margin that accepts some misclassifications of the training instances [VCC+99]. To overcome this limitation we

can introduce a set of positive slack variables $\xi_i, i = 1, .., l$ [CV95]. The problem then becomes:

$$x_i \cdot \boldsymbol{w} + b \geq 1 - \xi_i \qquad \forall y_i = +1 \tag{2.29}$$

$$x_i \cdot \boldsymbol{w} + b \leq -1 - \xi_i \qquad \forall y_i = -1 \tag{2.30}$$

$$\xi_i \geq 0 \qquad \forall i \tag{2.31}$$

Consequently, the Lagrangian primal is:

$$L_p \equiv \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{l}\xi_i - \sum_{i=1}^{l}\alpha_i\{y_i(x_i \cdot \boldsymbol{w} + b) - 1 + \xi_i\} - \sum_{i=1}^{l}\mu_i\xi_i \tag{2.32}$$

where μ_i are the Lagrange multipliers introduced to enforce positivity of the xi_i. C is a parameter chosen by the user, with larger values corresponding to assigning a higher penalty to errors. The Lagrangian primal problem is then solved as in the previous separable case.

Unfortunately many problems in the real world involve non-linear relationship between the input features and the output value. One solution is to map the data in a higher dimensional space and define a separating hyperplane there [BGV92, ABR64]. This high-dimensional space is called *transformed feature space*, in opposition to the *input space* defined by the training set. The input data need to be mapped to some other Euclidean space $\mathcal{H}$ using a mapping function Φ:

$$\Phi : \mathbf{R}^d \mapsto \mathcal{H} \tag{2.33}$$

The SVM training algorithm is basically composed by dot products $x_i \cdot x_j$ and thus the remapping would take the form $\Phi(x_i) \cdot \Phi(x_j)$. The next step is to introduce a "kernel" function K such that $K(x_i, x_j) = \Phi(x_i) \cdot \Phi(x_j)$; this function would be sufficient for the training algorithm (avoiding the need of explicitly define Φ). Therefore kernels are special function that allow the inner products to be computed in the feature space, without passing through the mapping [SB99]. Once the hyperplane is obtained, the kernel function classifies new points within the feature space. The classification function has this expression:

$$f(x) = \sum_{i=1}^{N_s}\alpha_i y_i \Phi(s_i) \cdot \Phi(x) + b = \sum_{i=1}^{N_s}\alpha_i y_i K(s_i, x) + b \tag{2.34}$$

where s_i are the support vectors and N_s is their number. The selection of the right kernel function is extremely important for the accuracy and efficiency of the SVM training. Different classes of problems require different kinds of kernels. Genton [Gen01] surveys several types of kernel functions but does not investigate which are the optimal kernels for given a problem. It is common practice to test a range of potential kernels and use cross-validation over the training set to find the best one. This has the negative side effect of increasing the time needed for training the SVM. Selecting a kernel (and its setting) is analogous to choosing the number of hidden layer in a neural network.

An interesting aspect of SVMs is that the model complexity of a SVM does not depend on the number of features present in the training data, therefore SVMs are very apt at dealing with learning tasks where the number of features is large with respect to the number of training instances.

2.3.1.6 Ensemble Methods

A different approach with respect to those presented so far, whose general aim was to obtain the best prediction model (given a certain problem and setting), is represented by *ensemble methods* [Die02, ZM12]. Ensemble methods are learning algorithms that construct a set of prediction models and then classify new data points by taking a (weighted) vote of their predictions; the predictors are individually trained. It has been discovered that in many case ensembles are more accurate than the individual classifiers that compose them [HS90]. The construction of good ensemble methods to combine single classifier predictions has been a very active research area in ML [Die00b, TG03, ZWT02, SK01].

Dietterich wrote a very good survey of ensemble methods [Die00a], reviewing the performance of several algorithms and explaining why ensembles outperform single classifiers in many contexts. In order for an ensemble to be more accurate than its components the necessary and sufficient condition is for the individual classifiers to be accurate and diverse. An accurate classifier must be able to classify a new instance x with an error rate smaller than random guessing. Two classifiers are diverse if they make different errors on different examples. From a practical point of view there are three reasons that allow ensemble classifiers to outperform individual ones. 1) Learning algorithms try to find the best hypothesis fitting the training set searching in the (possibly huge) hypotheses space. If the data set is too small compared to the size of the hypotheses space, individual classifiers may find different hypotheses that fit the small training set; combining different predictions means that an ensemble can "average" their votes and decrease the risk of choosing the wrong classifier. 2) Many learning techniques perform some kind of local search in the hypotheses space. Even if there are sufficient data it is difficult to find the best hypothesis due to computational limitations; for example both neural networks and decision trees training are NP-hard [BR88, HR76]. An ensemble built running local search from many different point can lead to better estimate of the unknown function between input data and target. 3) In most ML applications the unknown function cannot be represented by any of the hypotheses; this stems from the fact that with a finite training set even the most flexible and expressive ML algorithm can only explore a finite set of hypotheses. Using weighted sums of hypotheses ensemble methods may provide a way to increase the space of representable functions.

Two of the most commonly used ensemble methods are *Boosting* and *Bagging*. Bagging [Bre96b, Bre96a] is an ensemble method that trains the individual components on a random redistribution of the original data set; each classifier training set is generated by randomly drawing, with replacement, N examples (N is the size of the original data sets). Some examples may be repeated in the resulting training set while other may be left out. Boosting [SFBL97, Fre90, Sch90] is a family of methods that create series of classifiers; the training set chosen for each classifier depends on the performance of the previous classifier. With Boosting technique the previously misclassified examples are chosen more often than the correctly predicted ones; the idea is that new classifiers in the series will be able to improve the prediction accuracy of examples that are currently not well handled. Opitz et al. [OM99] evaluate the performance of Bagging and Boosting methods on different data sets, using neural networks and decision trees as classification algorithms. They observe that Bagging is almost always more precise than individual classifiers but some-

times less precise than Boosting; conversely Boosting can lead to classifier less accurate than individual ones (especially with neural networks). This happens because Boosting methods performance depends heavily on the data set, in particular Boosting ensembles tend to incur in overfitting when dealing with noisy data sets.

An example of widespread ensemble method is *Random Forest* [Bre01,LW02, GBS06]. As the name suggest Random Forest classifiers are composed by a set of decision trees. Random Forest creates multiple trees; for each tree only a random subset of the input features is used at every splitting node and no pruning is performed. Since only a portion of the input features is used and no pruning, the computational load of Random Forest is relatively light. The classification of new instances is made through a majority vote. Random forest are often used because they are much less prone to overfitting compared to models using a single decision tree.

2.3.2 Unsupervised and Reinforcement Learning

In this work we were mainly (almost exclusively) focused on supervised Machine Learning techniques. As we are going to see in Section 2.3.3 most ML methods commonly used in HPC area deal with prediction problems involving labeled data and therefore supervised algorithm are typically employed. Nevertheless, for the sake of completeness we are going to briefly present two different paradigms of ML approaches, namely *unsupervised* Machine Learning and *reinforcement* Machine Learning.

2.3.2.1 Unsupervised Learning

Unsupervised Machine Learning main difference compared to the supervised approach is the lack of labels from the data [HS99,Bar89]. Unsupervised learning studies how systems can learn to represent particular input patterns in a way that reflects the statistical structure of the overall collection of input patterns [Day99]. There are no explicit target outputs or external evaluation or reward. The only elements that can be used by unsupervised learning methods are the input patterns x_i, often assumed to be independent samples from an underlying unknown probability distribution $P(x)$ and some explicit or implicit a priori information to determine what is relevant. Ghahramani provides a great high-level introduction on unsupervised learning [Gha04], from the perspective of statistical learning. A very interesting survey on unsupervised algorithms for automation, classification and maintenance tasks can be found in [KMI+15].

The key concept underlying unsupervised learning is the possibility to learn a probabilistic model coherent with the input data. The model should be capable of estimating the probability distribution of a new input x_n given a series of previous inputs $x_1, .., x_{n-1}$; the model learns $P(x_n|x_1, .., x_{n-1})$. At the heart of most of probabilistic models there is again the Bayes theorem and its corollaries – as already introduced and discussed in Section 2.3.1.3. These models can then be used for outlier detection or monitoring – i.e. detecting unexpected outcomes. Other application of such a learned model are represented by classification task or communication and data compression [Mac03].

In many real world contexts there might be a mixture of labeled and unlabeled data. This could happen very easily for example in domains where

collecting data is cheap (i.e. the internet) but the labeling phase is much more expensive or time consuming. The research area of *semi-supervised learning* focuses precisely on this kind of issues. The main aspect to be considered in semi-supervised learning is how the data distribution of the unlabeled data influences the supervised learning problem [See00]. Delving into the details of semi-supervised learning is outside of the scope of this work but we can hint that many of the proposed approaches try to infer a manifold, graph structure or tree structure from unlabeled data and employ this information to determine how labels should be generalized to new unlabeled points [Kru64,JS02,ZGL+03].

2.3.2.2 Reinforcement Learning

Reinforcement-based learning systems perform actions and receive punishments, negative reinforcement, or prizes, positive reinforcement [SB98, Sut92, TL00]. Consequently, the learner will try to take those actions guaranteeing the best possible result. Reinforcement-based learning systems can be seen also as *agents* operating in an environment and receiving positive, negative or neutral rewards according to their actions; the rewards are usually given by a trainer. The task of the agent is to learn from this indirect reward and choose the sequence of actions that guarantee the best overall outcome. An important aspect in reinforcement learning is the need to balance exploration and exploitation [SB98]. In order to optimally execute a task a learning agent needs to exploit the "good" actions that it learned through the rewards mechanism but at the same time the learning process requires that multiple different actions must be taken – exploring the possible actions space by taking new actions. Neither exploration nor exploitation can be pursued exclusively without failing at the task. The exploration/exploitation dilemma has been intensively studied by mathematicians for many decades [Bel56, BT95].

The most important components present in the vast majority of reinforcement learning system are the following: 1) a policy, 2) a reward function and 3) a value function. A policy defines the behaviour of an agent, the action it will take in a given state (state perceived from the environment). In general, policies may be stochastic. The reward function specifies the goal: it maps every possible state (or state-action couple) to a single value representing the desirability of that state. The only goal of a reinforcement learning agent is to maximize the reward function in the long run. While the reward function indicates the immediate attractiveness of a certain state, the value function presents a long-term point of view. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Therefore the value function takes into account the reward functions of the states that may follow. Rewards are given directly by the environment (response obtained after reaching a certain state) whereas values are predictions of rewards; actions and decisions are made based on value judgments.

A very good survey on reinforcement learning (even though a bit dated) can be found in [KLM96]. Gosavi [Gos09] considers more recent developments in reinforced ML with a particular regard to its application in control theory and to agents performing decision-making. Both these aspects can be modeled as Markov decision problems [Put14], which were traditionally solved through Dynamic Programming. The author argues that reinforcement learning is a powerful tool to deal with these kind of problems thanks to its ability to solve

near-optimally, complex and large-scale Markov decision processes.

2.3.3 Applying Machine Learning to HPC Systems

As we have seen in the previous sections a very important phase in supercomputer management is the scheduling & allocation phase (see Section 2.1). The scheduling order and the selection of execution nodes can have a great effect on the overall performance of a system, in terms of QoS for the end user and utilization for the owners. Dispatching decisions can also heavily impact the power and energy consumption or the thermal behaviour of a system. The possibility of taking power-aware (or focused on any other metric) decisions is strongly correlated on the information regarding the workload available at schedule-time. For example, job dispatchers decide where and when execute a job depending on the amount of resources requested and on the current resource availability. Similarly, in order to take "good" decisions in terms of power a job dispatcher needs to have some information about how the execution of a certain workload will impact the system power consumption.

More precisely job dispatchers need to know the power consumption (or at least an estimate) of each application before deciding a schedule. For example if we consider again the case of power capped supercomputers, the goal is to guarantee *a priori* that the power constraint will not be violated in any moment (with a certain level of confidence). For this reason the capability of predicting the power consumptions of the jobs which need to be run is extremely important for the optimal implementation of a power capping method, as underlined by several works [PSLeA16, SSPeA14, BPV10].

Research in modeling and predicting data centers' power efficiency and performance is a very active area. Energy consumption models are critical in designing and optimizing energy-efficient HPC management system. A very interesting and complete survey of the state-of-the-art techniques used for energy consumption modeling and prediction for data centers and their components can be found in [DWF16]. Dayarathna et al. identify some open issues that still need to be addressed in order to improve the current state of power modeling in data centers: few approaches target power modeling at system-level; many of the techniques found in literature focus on a limited number of CPU metrics; the effectiveness and accuracy of the proposed methods is somehow lacking. Gao [GJ14] has recently proposed a model to estimate the PUE of a data center using Artificial Neural Networks. The model takes as input workload, cooling, power, together with other external information such as outside temperature, wind speed, etc. This allowed for testing various data center scenarios and improving PUE for the system under analysis. The training and testing of the proposed model was performed using large amount of data available at Google's data centers.

A greater prediction accuracy is related to a better performance of a power capped dispatcher (in terms of higher machine utilization and greater energy savings) [CGUeA08]. Intuitively, if we could exactly know the power consumed by each application we could generate optimal schedules and be sure that these schedules will never exceed the power budget; conversely, we may obtain suboptimal solutions when we deal with imperfect estimates. We may want to be robust and never violate the power constraint (for example, employing a tighter power budget), or we can accept to exceed the power limit from time to time,

relying on the hardware to fix these violations.

A common way to estimate an application energy or power consumption exploits hardware performance counters which monitors the system's components usage during the workload execution [CLPeA14,CM05,SBM09]. Some approaches use application signatures to predict power consumption across different architectures [ORS+10]; others employ job performance counters [WOPW13, NMN+10]. Despite the good accuracy obtained with these models the need to know the performance counters, which should be measured at runtime, clashes with the idea of having power consumption predictions available during the dispatching phase. Similarly, other methods rely on real-time load measurement [Dar15, TDM11, KJCP14, MDZD09]. These methods do not allow for prediction in real life scenarios, since load counters cannot be known in advance, unless they can be predicted through other methods.

A model to predict energy and power consumptions is presented by Shoukourian et al. in [SWAB14]. The authors propose an approach which does not require any application code instrumentation and allows for ahead of time power and energy consumption prediction. The main limit of the described method is that it considers only jobs which occupied entire computational nodes (this is due to the characteristics of the considered supercomputer). This on one hand simplifies the power consumption prediction but on the other hand cannot be directly generalized to different systems where multiple applications can possibly concurrently run on the same node.

A very interesting model to predict power consumption in a HPC system is described by Sirbu et al. [SB16]. The model takes as input workload information (amount of resources used at any time by the application) and produces as output the resulting system-level power consumption. The workload information can be forecast at dispatch time knowing the scheduler policies and subsequently the prediction model can be used to estimate the power consumption of an application. The model can be also used to evaluate the impact of different workload scheduling policies w.r.t. power consumption and therefore adapt the dispatcher behaviour.

Interest in power predictions is not limited to power capping. For example, Auweter at al [ABB+14] developed an energy aware scheduler able to reduce energy consumption of supercomputers. For this purpose they introduced a prediction model to forecast power and performance application in case of different execution frequencies. The model relies heavily on precise information about the application executables and requires the user to provide a tag identifying similar jobs. While this is an interesting direction, currently users-provided information cannot be taken for granted.

Both in HPC systems and data-centers machine learning techniques have been adopted to accurately predict several parameters associated with a given workload, such as power consumption, CPU usage, actual task duration, etc. The accuracy of the estimate relies on the huge amount of data which is generated by large scale systems and that collect each job's features. These predictions may then be used to develop power aware schedulers [BGT11, BGN+10]. Other approaches employ statistical methods to model the power process of HPC applications [SSPeA14].

The impact that uncertainties in the energy consumption can have on energy-aware schedulers has been studied by Iturriaga et al [IGN14]. They performed an experimental evaluation over realistic workloads and scenarios, and validated

by measurements of power consumption on a real data center. Both offline and online scheduling algorithms were implemented on the target cluster. The results show that errors in real-world scenarios can have a significant impact on the accuracy of the scheduling algorithms. Online strategies generally outperformed offline ones, due to the major capability to deal with imperfect estimates.

Chapter 3

Job Dispatching in HPC systems

High Performance Computing systems and data centers play in the world ICT infrastructure a role as big as (sadly) their power consumption. In many cases, a surprising amount of such consumption is due to idle resources, either introduced to face workload peaks or leftovers of workload fragmentation. 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 ∼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 [CIN] in Italy and the Eurotech group [Eur] 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 [McK] 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.

Contributions In the complex context of HPC dispatching optimization techniques can enable dramatic improvements in the resource management, leading to lower costs, better response times, and fewer emissions. In particular, a critical stage of a supercomputer management process is the job dispatching phase. As it is known from the literature since many years, Constraint Programming is a great approach for dealing with scheduling and allocation problems due the flexibility and expressiveness of the language coupled with extremely efficient resolution techniques. The application of CP based methods to the supercomputing world is a novel research direction that allows to reap off the multiple benefits generated by improvements in the management of HPC machines. In order to gain these benefits we devise a CP model to cope with the complex, multi-dimensional and multi-objective HPC dispatching problem, using a real supercomputer as a case study. We then compare the proposed CP model with the dispatching system currently in use on the HPC system.

Outline The chapter is organized as follows: Section 3.1 describes in a more detailed way the problem tackled; Section 3.2 introduces the supercomputer we used as a case study in order to deal with a realistic problem. Section 3.3 defines the challenges that need to be overcome to develop a dispatcher usable on a real system. Section 3.4 represents the main contribution of this chapter and describe the CP approach to perform job dispatching in HPC systems. Then Section 3.5 shows the experimental results and compares the CP model with the dispatcher currently in use on the supercomputer used as case study.

Publications Part of the work at the base of this chapter has been published to international conferences in [BBB+14].

3.1 Problem Statement

In this chapter, we tackle the problem of workload dispatching in a High Performance Computing machine. This task is performed by a software module that decides *where* (assigning a set of resources) and *when* (choosing a start time) a job has to execute. Users submit jobs to supercomputing machines specifying the amount of required resources (CPUs, GPUs, memory, etc.) 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. As an example of HPC machine we use the Eurora supercomputer [BCC+14], developed by Eurotech and Cineca [CIN]. Eurora system will be the case study employed in the majority of this work; nevertheless all proposed approaches can be generalized in order to be implemented in different system.

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 [Wor15]). 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.

3.2 Eurora System

The Eurora supercomputer prototype has ranked first in the Green500 list in July 2013, achieving 3.2 GFlops/W on the Linpack Benchmark with a peak power consumption of 30.7 KW. Eurora has been supported by PRACE 2IP project [PRA] and it serves as testbed for next generation Tier-0 system. Its outstanding energy efficiency is achieved by adopting a direct liquid cooling solution and a heterogeneous architecture with general purpose HW components (Intel Xeon E5, Intel Xeon Phi and NVIDIA Kepler K20). Eurora cooling solution is highly efficient and enables hot water cooling, that is suitable for hot water recycling and free-cooling solutions [KRA12c, 9.911]. For its characteristics Eurora is a perfect vehicle for testing and characterizing next-generation "greener" supercomputers.

3.2.1 System Description

As described in [BCC+14] the architecture of Eurora consists of 8 stacked chassis (half-rack), each of them hosting 8 node cards and 16 expansion cards (Fig. 3.1). The node card is the basic element of the system and comprises 2 Intel Xeon E5 Series (SandyBridge) processors and 2 expansion cards configured to host an accelerator module. One half of the nodes use E5-2658 processors including 8 cores with 2.0 GHz clock speed while the other half uses E5-2687W processors including 8 cores with 3.1 GHz clock speed; 58 nodes have 16 GB RAM and the remaining 6 (with processors at 3 GHz clock rate) have 32 GB RAM. The accelerator modules can be Nvidia Tesla (Kepler) or, alternatively, Intel MIC KNC (Xeon phi).

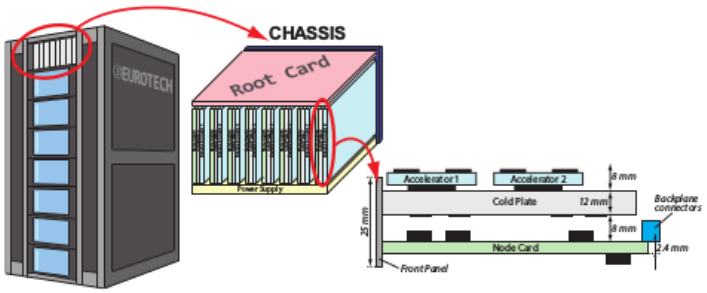


Figure 3.1: EURORA Architecture

Each node of Eurora currently executes a SMP CentOS Linux distribution version 6.3. Eurora is interfaced with the outside world through two dedicated

computing nodes, physically positioned outside the Eurora rack - the login node, linking Eurora to the users, executes the batch job dispatcher (PBS) and connects to the same shared file system, and the master node, connected to all the root cards and visible only to system administrators. Moreover, Eurora adopts a hot liquid cooling technology, i.e. the water inside the system can reach up to 50°C. This strongly reduces the cooling 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 energy source for other applications.

Eurora features an integrated and low-overhead monitoring system composed by a set of software daemons and parsing scripts. The SW daemons run periodically (every 5 second) on each node to collect traces of the processing elements (CPUs, GPUs, Xeon Phy) activity by mean of HW performance counters. For each core it gathers values from the Performance Monitoring Unit 2 as well as the core temperature sensors, and the time-step counter. In addition, for each CPU it gathers the monitoring counters (power unit, core energy, dram energy, package energy) present in the Intel Running Average Power Limit (RAPL) interface. The parsing scripts process offline the raw log of the performance counters to generate performance metrics (CPI, Load, Temperature, Power, etc.) and relate them with the job running on the node.

3.2.2 Current Dispatcher

The tool currently used to manage the workload on Eurora system is PBS (Portable Batch System) [Wor15], 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 requests 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 of 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.

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

3.3 Online Dispatching

A key challenge for our approach is the "online" requirement, the need to devise a job dispatcher able to answer to the arrival of new, unexpected jobs and to perform in *real-time*. That requires to have fast time-to-solution each time a new schedule is computed; an online dispatcher has to take scheduling and allocation decision for every job whenever it enters the system. This issue is partially mitigated by the long duration of many HPC applications (a duration of a few hours is not uncommon at all) and therefore jobs tend to arrive in a supercomputer at a slower pace than in more dynamic data centers, which in turn allows for longer times to compute a schedule. As a downside, typical supercomputer have thousands of nodes and the dispatching problem complexity exponentially increases with both the number of resources and number of jobs.

Due to the real-time issue, typical online dispatching softwares have a "reactive" nature, they take the scheduling and allocation decisions considering only the newly arrived jobs, without reflecting on the jobs coming from other queues or the jobs still waiting due to previous resources unavailability. In our work we want to create a *proactive* dispatcher, that is a dispatcher that takes in consideration not only the last job whose arrival triggered the scheduling event, but all jobs present on the system. Therefore we will have to focus on the scalability of our solutions. In order to cope with scalability and real-time issues the most commonly adopted strategy is not to care about reaching optimal solutions. Virtually all dispatching tools employed on real world supercomputers use simple, heuristic methods to guarantee good but probably sub-optimal solutions in real time. We follow the same approach and therefore we do not aim at optimal solution but rather we want to reach the best possible one given a tight time limit to explore the solutions space. Our goal will be to provide better solutions than the current standard, PBS, which is a non-exact technique as well.

3.4 Job Dispatching in HPC: a CP Approach

In its current state, the PBS system works mostly as an online 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 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 a supercomputer can be naturally framed as a resource allocation and scheduling problem, for which CP as a long

track of success stories. However, to our knowledge this is the first attempt to frame the HPC dispatching problem within the CP paradigm.

3.4.1 Rolling Horizon

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.

The scheduler has to react to runtime events, like new job submissions and job terminations, and obtain a new scheduling solution from the supercomputer's current state. To solve this problem, since we are in a non-preemptive system, we must obtain all the jobs running on the supercomputers and set their actual start time and the used nodes, as well as insert the jobs in queues.

3.4.2 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 eqt_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. Job-units belonging to the same job can be mapped on different nodes (but not necessarily) and they must have the same start time (it is also assumed that they share the same duration).

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 $st_i \geq t$ 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 $st_i = t$ 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:

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

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

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

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.

3.4.3 Model Definition

We defined for the described dispatching problem a CP model that is based on Conditional Interval Variables (CVI, see [LR08]). 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* [BLLN06] to model finite capacity resources, and the more specific *alternative* constraint [LR08]. 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:

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

3.4.3.1 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[1]. 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 $x(\upsilon_{i,k}) = 1$ if 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 $x(\upsilon_{i,j,k}) = 1$ if 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:

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

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. 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.

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

3.4.3.2 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 3.4.2:

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

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

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

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:

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

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

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

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 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).

3.4.3.3 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). Two units of job 000 are allocated on node 1, the other 30 units of job 000 are allocated in nodes 2 to 31; node 0 is completely free and can run job 001 while job 000 is executing; job 003 can execute on nodes 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 32 to 63.

i	rn_i	$rq_{i,core}$	$rq_{i,gpu}$	$rq_{i,mic}$	$rq_{i,mem}$ *(KB)*	D_i (seconds)
000	32	4	1	0	1000	14000
001	1	14	1	0	400	600
002	2	4	1	0	400	14400
003	32	16	0	0	400	800
004	32	3	0	2	800	400

Table 3.2: An example of problem instance

i	$s(\tau_i)$	$\upsilon_{i,0,0}$	$\upsilon_{i,0,1}$	$\upsilon_{i,0,2..31}$	$\upsilon_{i,0,32..63}$	$\upsilon_{i,1,0}$	$\upsilon_{i,1,1}$	$\upsilon_{i,1,2..31}$	$\upsilon_{i,1,32..63}$
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

3.5 Experimental Results

The results we are going to discuss now focus on the prototype version of the proposed scheduler. At this stage we are interested in investigating the kind of improvements that could be obtained by changing the dispatcher behaviour. For 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 to simulate the scheduler activation events) and the start time assigned by PBS.

Our approach was implemented using IBM ILOG CP Optimizer [Lab09a] using its default search strategy, which is based on Self-adapting Large Neigh-

Model	*Average Queue Time* (seconds)			
	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

Model	*Average Resource Utilization*				*Avg. # jobs*
	cores	*GPUs*	*MICs*	*cores (%)*	
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

borhood Search [LG07] 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.

In the first run we set a time limit of 60 seconds and if the model returns no solution within the allowed timespan (and did not explore all the solutions), we recreate the model setting a new time limit - multiplying by 5 the previous one. However, if the time limit reaches a threshold of 2 hours, this cycle stops and an empty schedule is returned. In practice this means that for that particular time point, the system state does not change and the dispatcher waits until the next time instant (when probably some jobs are completed) to compute a new scheduling. In our tests, this condition has never been reached: we were always able to find a solution in the first run.

3.5.1 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 a measure of the perceived quality of services, while a high utilization directly translates to a low number of idle (but still power consuming) resources and to a more efficient use of the machine 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], 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 (one order of magnitude), and especially for those using the Makespan as main objective

	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. 3 cores	0	0	0
#jobs req. 4 cores	1	4	7
#jobs req. 5 cores	1	1	0
#jobs req. 6 cores	6	2	3
#jobs req. 7 cores	0	0	0
#jobs req. 8 cores	59	56	187
#jobs req. 8+ cores	70	101	126

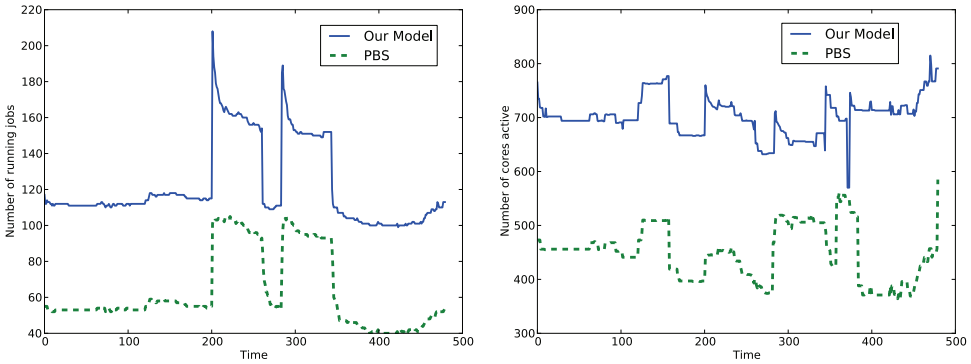
Table 3.6: Job traces composition

(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 performs slightly better in terms of the average number of active cores. In the last column of the table (*Avg. # jobs*) 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 fifth 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 (obtained by 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.

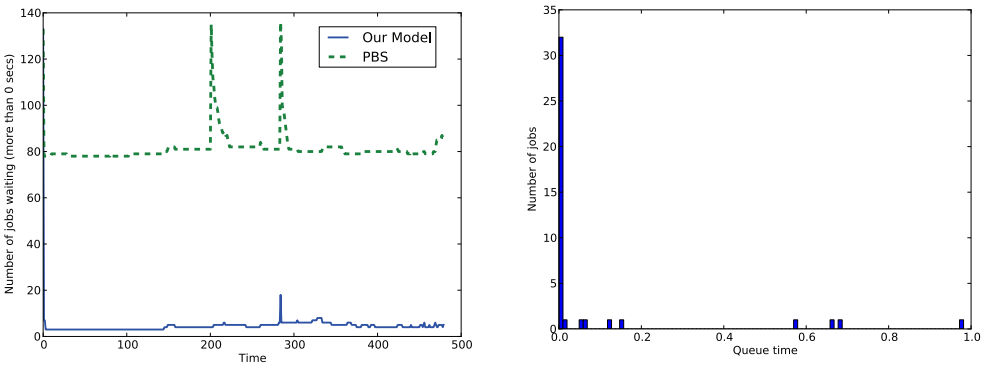
3.5.2 Comparison with PBS

The previous results show that our best model is a composite one, namely MKS WT/NL, thus such model was chosen for a more detailed comparison with PBS on three 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



(a) Running Jobs (b) Active cores

Figure 3.2: Eurora utilization on the first trace (BATCH1)



(a) Jobs in Queue (b) Times in Queue

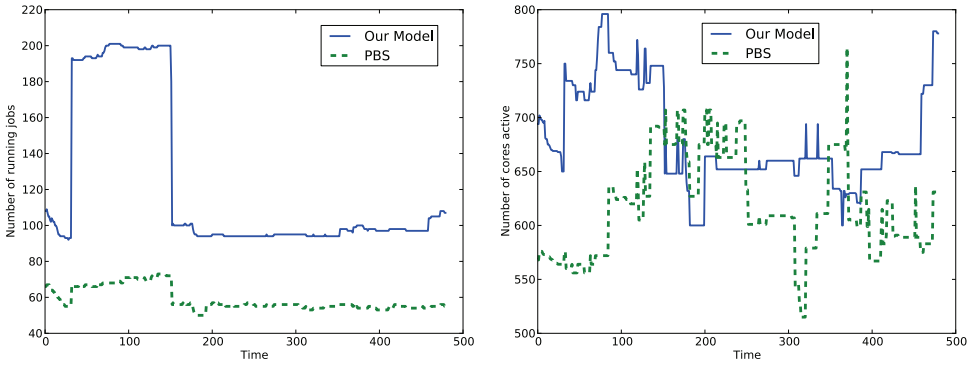
Figure 3.3: Waiting jobs and queue time for BATCH1

reports the total number of jobs, the number of jobs in each queue[2], 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 composing this trace belong to a wide range of classes, with different resource requirements and different execution times. In Fig. 3.2a we can observe the number of active jobs in the considered time frame, for both our approach (solid line) and PBS (dashed line). Fig. 3.2b 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 reached the optimal system usage: this could be due to (a combination of) the presence of bottleneck resources, to suboptimal

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



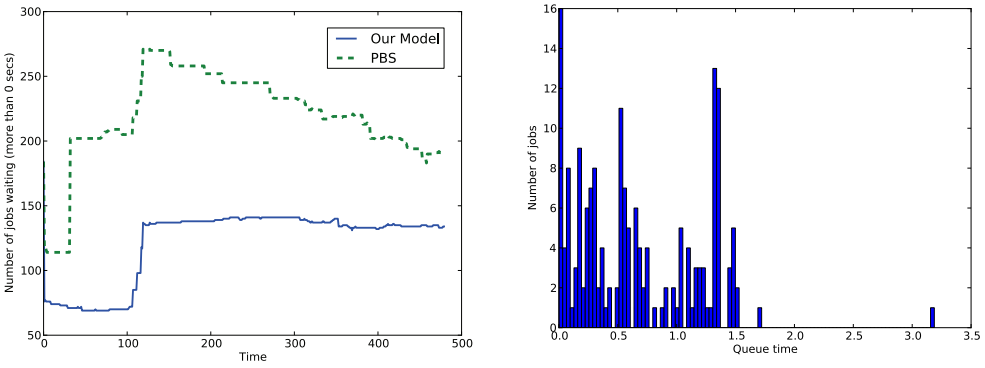
(a) Running Jobs (b) Active cores

Figure 3.4: Eurora utilization on the second trace (BATCH2)

allocation choices, or simply to the lack of more workload to be dispatched. Fig. 3.3a 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. 3.3b 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 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. 3.4 and Fig. 3.5 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 much fewer than cores. The consequences of this situation can be observed in Fig. 3.4a and Fig. 3.4b, 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 available cores they cannot be used. Despite that, we still manage to achieve a largely improved schedule than the one of PBS in terms of number of running jobs. In particular, the average number of active GPUs with



(a) Jobs in Queue

(b) Times in Queue

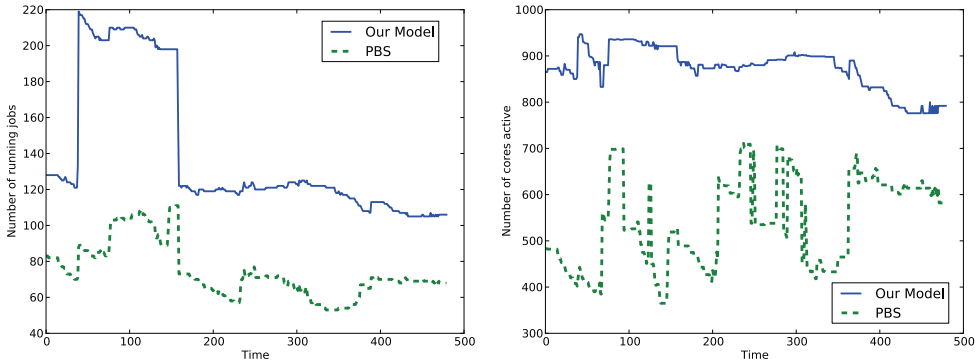
Figure 3.5: Waiting jobs and queue time for BATCH2

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.

In Figure 3.5 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 3.5a) follows 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.

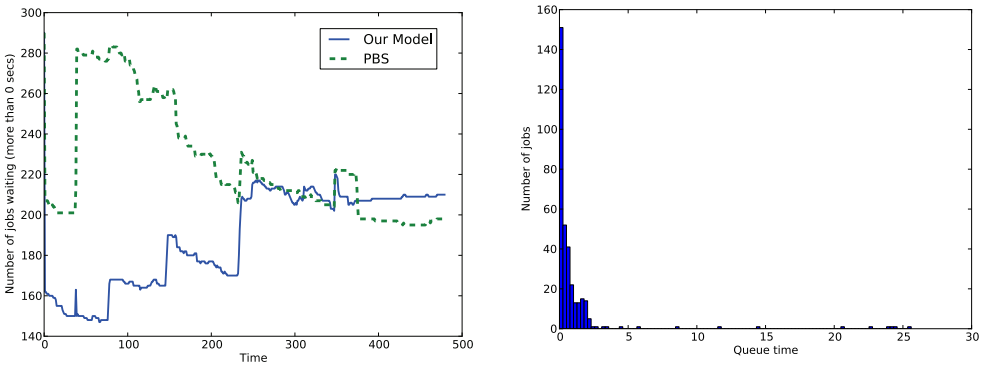
Finally, we can eventually consider BATCH3 and the results are displayed in Fig. 3.6 and Fig. 3.7. The jobs considered in this trace require, on average, a higher number of cores than all other traces and, for a large part, were submitted to the *parallel* queue. They require proportionally fewer GPUs 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. 3.6b 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. 3.6a).

In Figure 3.7a 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 waiting jobs is similar between the two dispatchers. One possible explanation for this is again the 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 required GPUs is reached.



(a) Running Jobs (b) Active cores

Figure 3.6: Eurora utilization on the third trace (BATCH3)

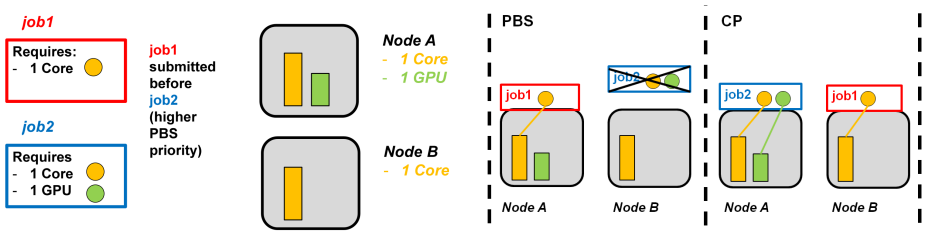


(a) Jobs in Queue (b) Times in Queue

Figure 3.7: Waiting jobs and queue time for BATCH3

Example of CP and PBS different scheduling decisions In Figure 3.8 we can see an example of a very simple dispatching situation that gives an insight on why the CP proactive model takes better decisions compared to the simpler PBS heuristic policy. Suppose we have node A and B, where A has 1 core and 1 GPU while B has only 1 core, and suppose that A and B are fully occupied by previous jobs. There are two job that need to be scheduled: *job1*, requiring one core, and *job2*, requiring one core and one GPU. We suppose that *job1* was submitted first and therefore it has a higher priority from PBS point of view. Both the jobs are in the waiting queue because the requested resources are unavailable. Figure 3.8a depicts this situation.

Let suppose that all jobs currently occupying nodes A and B terminate at the same moment, triggering a scheduling event. In Figure 3.8b we see the different decisions taken by the PBS-like dispatcher (on the left) and by the CP proactive dispatcher (on the right). PBS starts looking at the queued jobs



(a) Jobs Submission (b) Jobs Dispatching

Figure 3.8: Proactive dispatching VS PBS: an example

following the priority order and selects *jobs1*; subsequently, it assigns *jobs1* to the first available node, in this case *A* (the nodes are statically ordered without considering their usages). Then PBS considers the second element in the waiting queue, *jobs2*, and tries to see if it can fit given the remaining available resources: unfortunately the only node with an available core is *B* that lacks the required GPU; therefore *jobs2* is forced to wait, resulting in a sub-optimal machine utilization.

If we instead look at the dispatching decisions taken by a CP approach we notice a different situation. Due to its proactive nature, it consider *all* jobs that are waiting in the queue and need to be executed. The CP dispatcher can then explore different situations; in this simple case it just tries the few possible placements of the two jobs on the two nodes. It will then realize that putting *jobs2* on *Node A* and *jobs1* on *Node B* is the best solution, in terms of higher machine utilization and lower waiting times for the jobs involved.

3.6 Chapter Summary

In this chapter we have discussed a Constraint Programming based proactive workload dispatcher for the HPC systems, using the Eurora supercomputer as a case study. We saw that the tackled problem is not an easy one, owing to the need to manage multiple objectives and to the limited availability of multiple, heterogeneous resources. The goodness of our dispatcher was evaluated considering two important aspects: 1) increasing machine utilization and 2) reducing jobs 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 and on the environmental side. Short waiting times correspond to a higher quality of service for the system users.

The higher machine utilization is especially important from the point of view of the machine owner because supercomputers are investment-intensive facilities with short depreciation periods. An average supercomputer reaches full depreciation in three to five years [Fel13], thus their utilization must be kept as high as possible, in order to produce an acceptable return on investment. Every small (even relatively) improvement in utilization and throughput might lead to great financial gains.

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 fundamentally reactive approach currently in use proved to have particular difficulties with the simultaneous management of different classes of resource (e.g. cores and GPUs).

The dispatcher we discussed here was just a prototype but with further work Bridi et al. managed to develop a wholly functioning, real-time job dispatcher based on the model here presented. Their effort led to the implementation of the proposed method on the Eurora supercomputer; the implementation details are beyond the scope of this work but the results can be found in [BBL+16b].

Chapter 4

Predicting Power Consumptions in HPC Sytems

The quest towards the much sought-after Exascale in High Performance Computing passes necessarily through a significant decrease of the power consumption of next-generation machines. In order to achieve this goal several different approaches have been suggested, ranging from more power-efficient HW solutions to more sophisticated power and energy aware management systems. The software-oriented methodologies generally rely on a combination of dynamic, reactive actions and planning strategies. An example of reactive action is the reduction of the operational clocks of some computational resources in a system if the dynamically measured temperature in the facility reaches a critical point. Conversely, given a a set of applications to be scheduled a planning strategy would decide the execution order in advance (before the actual execution) in order to optimize it w.r.t. some objective function.

In order to apply such a planning-based optimization strategies the ability to forecast the impact of the workload execution on the system has paramount importance. For example, the power that will be consumed by a HPC application during its execution is an essential requirement to devise an optimized power-aware strategy. This is especially true for tools such as proactive job dispatchers (see Chapter 3) that need to know *a priori* the power consumption (or at least an estimate) of each application before deciding a schedule. In turn, this knowledge leads to foreseeable system behaviours (within a certain level of confidence).

For this reason the capability of predicting the power consumptions of the jobs which need to be run is extremely important for the optimal implementation of power-aware approaches, as underlined by several works [PSLeA16, SSPeA14, BPV10]. In addition a greater prediction accuracy implies better performance (in terms of higher machine utilization and greater energy savings) [CGUeA08]. Let's consider the goal of having a power capped HPC system, i.e. a system whose power consumption will never exceed a given budget. Intuitively, if we could exactly know the power consumed by each application we could generate optimal schedules and be sure that these schedules will always respect the power

constraint. Conversely, we may obtain sub-optimal solutions when we deal with imperfect estimates – in this case we may want to be robust and never violate the power cap (for example, employing a tighter power budget), or we can accept to exceed the power limit from time to time.

A typical approach to obtain the power or energy consumption of an application consists in measuring the physical sensors which monitors the system's components during the workload execution [CLPeA14, CM05]. Although the measurements collected in this way are extremely precise their run-time only availability clearly collides with the requirement of a job dispatcher that demands these values at schedule-time. There is therefore a clear need for a different approach, although online monitoring systems will be certainly helpful in a second phase, after the jobs have started.

As we have seen in Chapter 2 (in particular Section 2.3.3) many research works [SB16, ABB+14, SWAB14] lately explored the possibility to forecast the power or energy consumption of supercomputer workload *before* the execution. In this chapter we are going to discuss the problem of predicting the power consumption of HPC applications using only the information available at submission time, namely without using real time measurements or internal program counters.

Contributions The possibility to forecast the power consumed by applications is an essential prerequisite in order to develop job dispatchers capable of containing power and energy consumption of High Performance Computing systems. The predictions must be available at schedule-time, i.e. when the dispatcher takes its decision, in order to be of any use. This implies that the prediction must involve only information available at schedule-time, for example the resources requested and the maximum duration. Machine Learning methodologies are widely adopted in many different areas to create many diverse prediction models. The first step needed to create a ML model is to have a large amount of data to use for the training phase - the accuracy of the prediction is strongly related to the size of the data set. For this purpose in this chapter we are going to describe the information collecting infrastructure that was implemented on the Eurora supercomputer. The large database containing information about historical workloads and the related power measurements gathered on the system constitutes the starting point for the following creation of a prediction model. Data sets need also to be "prepared", extracting and aggregating the significant information from the mass of raw data. Once the data set has been processed we can use Machine Learning algorithms to learn prediction models. The common procedure to test the quality of the forecast is splitting the data set in two components, train and test set, using the first during the learning phase and the latter as a way to compare estimates and real values. We test the goodness of the proposed prediction models against historical data from Eurora; the results reveal that the accuracy of our estimates is very good, ranging from 92% to 95%.

Outline This chapter is organized as follows. Section 4.1 describes the information collected on the Eurora supercomputer; we give the description of the data gathering infrastructure and a few example of the analysis that can be performed on such data. Section 4.2 studies how to extract the jobs power con-

sumptions from the raw data. Once a data set suitable for Machine Learning algorithms has been obtained, in Section 4.3 we present the prediction models used to estimate power consumptions. Finally we test the accuracy of our models comparing estimates and real data in Section 4.4.

Publications Part of the work at the core of this chapter has been published to international conferences in [BBL+16a].

4.1 Eurora Data

On the Eurora system we implemented an open access database where we gathered the information collected on the supercomputer. Our database contains data generated by the monitoring framework, ranging from the measurements of physical sensors installed on the machine to the jobs run on the system; the collecting infrastructure and the set of scripts loading the data on the database perform their tasks without interfering with Eurora workload. The database stores several months (precisely 18) of traces of Eurora activities, providing a large data set for studying the behaviour of a green supercomputer.

The large amount of data collected (more or less 500GB of memory storage) is the key element that allows the use of Machine Learning algorithms to forecast the jobs power consumptions at schedule time. This information has also been used to perform a preliminary analysis of machine's behaviour; we were interested in understanding the features characterizing Eurora typical workload. In the rest of this section we are going to discuss the monitoring & collecting infrastructure (Section 4.1.1) and after we present some examples of data analysis (Section 4.1.2).

4.1.1 Collecting Infrastructure

As said in [BCC+14] the data gathering framework is composed by a set of software daemons and scripts which collect at run-time the hardware sensor values for each component on the various nodes (see Fig. 4.1). The monitoring framework employs *online* and *offline* software components. The *online* components are responsible for readings the physical sensors' measurements and save them as log traces in the shared file system repository. These software modules are time triggered, and monitor, concurrently for each node, the power, the temperature and the level of activity of the two CPUs (CPU stats), two GPUs (GPU stats), two Xeon Phi cards (MIC stats) and the two board temperature sensors. We also developed a set of *offline* scripts that parse and then post-process the collected data traces; these processed traces are then used to extract the information which is going to fill the database, and that is going to be employed to enable statistical evaluation and modeling and any kind of relevant analysis. In addition to the data concerning the HW sensors, we also developed a set of daemons that retrieve the information relative to the jobs executed on the systems. This is a key point because with this information we are able to relate workloads and operating conditions, thus increasing the number and type of analysis that we can perform.

The open-access database stores in a single access point all information obtained through tracking the workload, from its submission till its execution and

completion, considering both micro (core power, temperature, instruction density, etc.) and macro effects (liquid cooling temperature and pump flow). We use a MySql database [WA02], hosted on a machine kindly provided to us by Cineca. We chose to use a standard and non-proprietary technology in order to improve the ease of access to any user. The raw data generated by the measurement framework (HW data) are parsed by a set of scripts written in Python and Unix Bash languages; the jobs information are obtained through queries submitted to the Eurora job dispatcher PBS, with its own scripting language. After that, all the data are loaded into the databases, with the whole process performed every two hours and it takes between 10 and 20 minutes - this is not a problem since this task takes place on the login node of Eurora (where the raw data are stored before being loaded to the database) and it has almost no computational impact on that node.

4.1.1.1 HW Sensors

In the following subsection we describe the database structure regarding the HW sensors data and the jobs information.

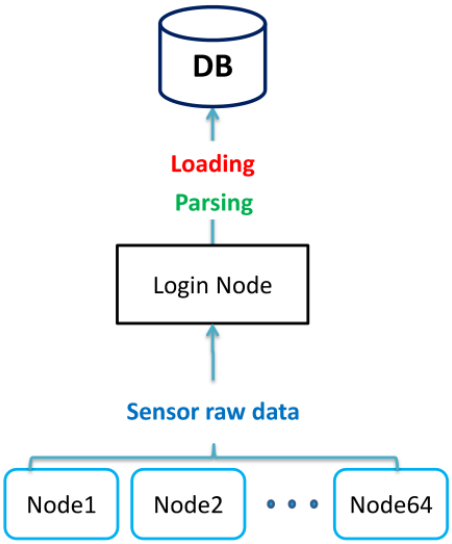


Figure 4.1: HW data flow

We collect data from several physical sensors for each node. More in particular we store data regarding the CPUs (two for each node), the cores (16 per node), the GPUs (two or zero per node), the MICs (two or zero per node) and the boards; we also collect data regarding the power supply and cooling system of the machine. A graphic representation of the HW data and relative tables in the database can be seen in Figure 4.2 and in the following sections we describe these tables in more detail.

In order to be able to retrieve the data stored on the DB without having to wait for too long[1], we added a few typical database accessory structures

[1]The database access time increases dramatically when the database grows larger. In our

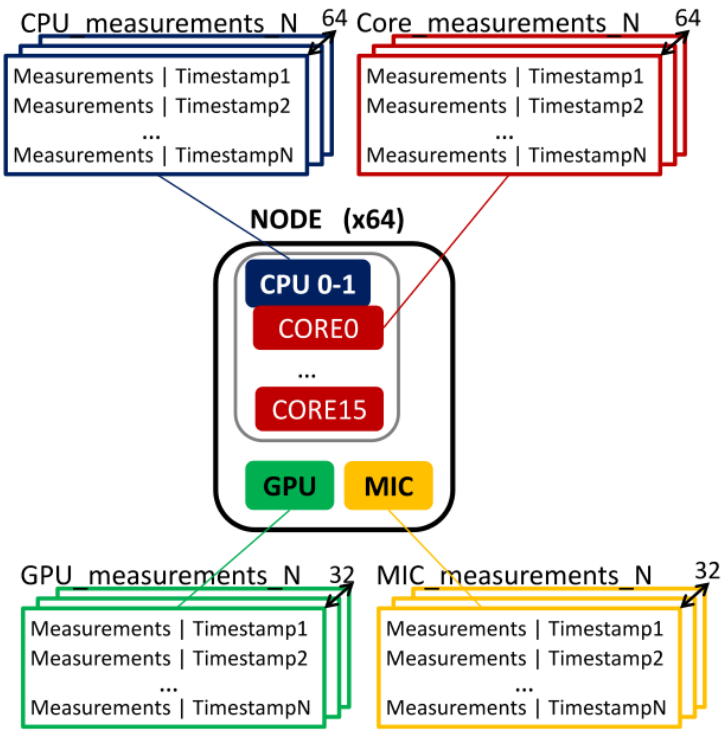


Figure 4.2: HW Data and Tables

called *indexes* which allow a faster access to the data. We had to find the right balance between the number of indexes and the time required to load the data onto the database – more indexes grant faster access times but unfortunately have the effect of increasing write times (since the DB has to update both the raw data plus the additional index structure). After an empirical evaluation, we eventually settled for a pair of indexes for each table.

We are now going to see in more detail the information stored in the database.

CPUs For every node we have a database table storing the CPUs information. There are 64 of these tables, namely *cpu_measurements_001, cpu_measurements_002, .. , cpu_measurements_064*. For each CPU we collect the data listed in Table 4.1 (the *cpu_id* distinguish between the two CPUs of the node). Data types are MySql specific.

Cores Table 4.2 shows the data regarding the cores. Again we have a table for each node called *core_measurements_001, core_measurements_002, .. , core_measurements_064*; remember that each node has 16 cores so in this case the field *core_id* ranges between 0 and 15.

cases we are dealing with a database with a dimension of several hundreds of Gigabytes and the access times can vary between a few seconds and a few minutes, depending on the type of MySql query used

Field Name	*Measure Unit*	*Meaning*
cpu_id	Integer	Identifies the CPU
pow_cpu	Watt	Power consumed by the CPU
pow_dram	Watt	Power consumed by the memory
pow_package	Watt	Power consumed by the whole package
dT_cpu	ms	Sampling interval in ms
timestamp	datetime	Time stamp of the measure

Table 4.1: CPUs tables entry fields

Field Name	*Measure Unit*	*Meaning*
core_id	Integer	Identifies the Core
cpi	Float	Clock Per Instruction
load_core	%	Load
mfreq	MHz	Maximum Frequency
rfreq	MHz	Real Frequency
dT_core	ms	Sampling interval in ms
temp	°C	Core Temperature
ips	GOPs	Instructions Per Second
timestamp	datetime	Time stamp of the measure

Table 4.2: Cores tables entry fields

GPUs We can see the fields of the entries in the GPUs tables in Table 4.3. We have one table for each node with a GPU, i.e. nodes from 33 to 64 (thus *gpu_measurements_033, gpu_measurements_034, .. , gpu_measurements_064*) and in each of these nodes there are 2 GPUs.

MICs Table 4.4 displays the fields of the entries in the MICs tables. We have one table for each node with a MICs, i.e. nodes from 1 to 32 (thus *mic_measurements_033, mic_measurements_034, .. , mic_measurements_064*) and in each of these nodes there are 2 MICs.

Field Name	*Measure Unit*	*Meaning*
gpu_id	Integer	Identifies the GPU
gpu_load	%	GPU Load
mem_load	%	Load of the GPU-dedicated Memory
gpu_freq	MHz	GPU Frequency
smem_freq	MHz	Static Memory Frequency
mem_freq	MHz	Memory Frequency
pow	Watt	Consumed Power
temp	°C	GPU Temperature
timestamp	datetime	Time stamp of the measure

Table 4.3: GPUs tables entry fields

Field Name	*Measure Unit*	*Meaning*
mic_id	Integer	Identifies the MIC
cpu_temp	°C	The MIC CPU temperature
mem_temp	°C	The MIC memory temperature
fan_in_temp	°C	Fan-in temperature
fan_out_temp	°C	Fan-out temperature
core_rail_temp	°C	Core rail temperature
uncore_rail_temp	°C	Uncore rail CPU temperature
mem_rail_temp	°C	Memory rail temperature
core_freq	MHz	MIC core frequency
total_power	Watt	Total MIC power
2x3_power	Watt	Low Power Limit
2x4_power	Watt	High Power Limit
phys_power	Watt	Physical Power Limit
free_mem	MB	MIC Free Memory
tot_mem	MB	MIC Total Memory
mem_usage	MB	MIC Used Memory
timestamp	datetime	Time stamp of the measure

Table 4.4: MICs table entry fields

Field Name	*Measure Unit*	*Meaning*
Sensor1	°C	Temperature of board
Sensor2	°C	Temperature of board
timestamp	datetime	Time stamp of the measure

Table 4.5: Boards table entry fields

Boards Table 4.5 presents the fields of the entries in the boards tables. There is one table for each node (one board for each node) therefore we have *board_measurements_001, board_measurements_002, .. , board_measurements_064*. For each board we measure its temperature in two different position.

Power & Cooling Systems The data from the power and cooling subsystems has a sampling interval of 1 minute. We measure information regarding the power supply such as current, voltage, frequency, absorption, total power consumed by the machine. For the cooling system we store temperature of the water in the 4 racks, temperature of the water flowing in and out, flow rate, position of the valve regulating the water flow.

4.1.1.2 Workload Information

Eurora uses PBS from Altair as job dispatcher which provides APIs to trace running and finished jobs; we use these API to obtain jobs related information, sampling time of five minutes. We then load the jobs statistics in the database every two hours. We use two tables:

1. table *jobs* keeps the information characterizing each job (job id, name, owner, start time, etc.);

2. table *jobs_to_nodes* tracks on which nodes a particular job has executed; to maintain this relation we employ a *foreign key* that relates each entry of this table with an entry of *job* table[2].

Figure 4.3 shows a scheme representing the relation between a job and the related entries in the database; a job is characterized by information such as its owner, its name, start time, etc.. (stored in table *jobs*), and by the nodes where it run (table *jobs_to_nodes*). Combining these information we can also retrieve the sensor measurements which were taken during the execution of a particular job.

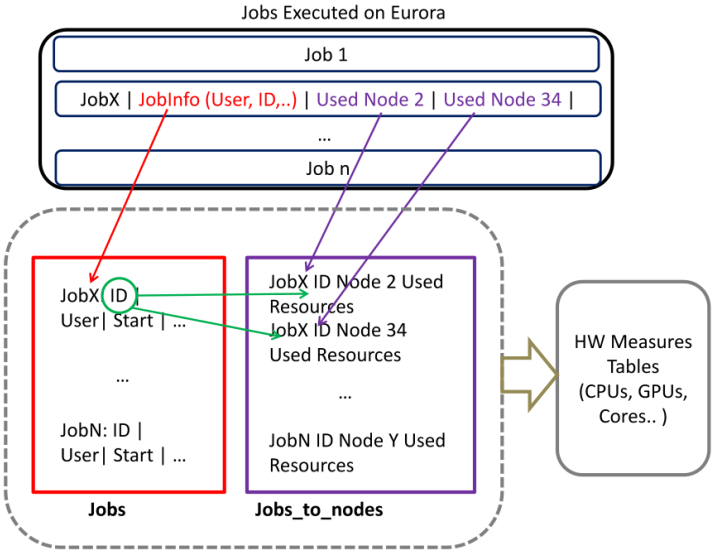


Figure 4.3: Jobs Info and Tables

The fields of the entries in *jobs* are summarized in Table 4.6; since we are only keeping track of completed jobs we always know when a job terminated (*end_time*) and its termination status (*exit_status*). The field *queue* indicates the queue where the job were assigned (due to its requirements). Since a job can be submitted but forced to wait by the dispatcher (machine too busy, low priority, etc.) we memorize both when a job is submitted, *start_time*, and when its execution really begins, *run_start_time*. When a user of the Eurora system submits a job (the *owner*) its requirements must be specified: the number of nodes required (*node_req*), the number of cores (*cpu_req*) and the amount of memory[3]. The user must also provide an estimated execution time, *time_req*, that represents an upper bound to the duration of the job - if a job has not finished within the expected time, it will be canceled by PBS. If, for any reason, a job can not terminate its execution successfully and is deleted we save this information in the *deleted* field.

Table 4.7 displays the structure of the table *jobs_to_nodes* containing the relation between job and its execution nodes. Any row of the table represents

[2]Every entry in *jobs_to_nodes* requires a field called *job_id* (the foreign key) which points to a specific entry in the table *jobs*

[3]The number of used GPUs and MICs is stored in a different table

Field Name	*Measure Unit*	*Meaning*
job_id	int	Identifies the job
job_id_string	varchar	String representation of the Id
job_name	varchar	The name chosen by the owner
queue	varchar	The queue to which the job belongs
start_time	datetime	When the jobs was submitted to PBS
run_start_time	datetime	When the job begins its execution
end_time	datetime	When the jobs has terminated
owner	varchar	The owner of the job
node_req	smallint	The number of nodes required
cpu_req	smallint	The number of cores required
mem_req	int	The amount of memory required
time_req	varchar	The time requested
deleted	varchar	If a job was deleted before its completion

Table 4.6: Jobs table entry fields

Field Name	*Measure Unit*	*Meaning*
job2node_id	int	Identifies a node-job pair
node_id	smallint	Identifies the node
job_id_string	varchar	Identifies the Job
ncpus	smallint	Number of cores used
ngpus	smallint	Number of GPUs used
nmics	smallint	Number of MICs used
mem_requested	int	Amount of memory used

Table 4.7: Jobs_to_nodes table entry fields

a job (*job_id*) run on a node (*node_id*); if a job required more than one node, there will be multiple entries with the same *job_id* but different *node_id*. The field *ncpus* tells the numbers of cores used by the job on the node; similarly the fields *ngpus*, *nmics* and *mem_requested* define, respectively, the number of GPUs, MICs and amount of memory.

4.1.2 Example of Collected Data

Figure 4.4 shows the percentage of resources requested by the jobs grouped by submission queue. The four queues active on Eurora (*debug*, *longpar*, *parallel* and *reservation*) are displayed in the x-axis and in the y-axis there are the five resources considered (in red the nodes, green for the cores, blue for the GPUs, yellow for the MICs and the memory in cyan). In the z-axis each bar represents the percentage of resources that can be attributed to the jobs which belong to a particular queue. It is easy to see that the jobs in *parallel* use the larger portion of resources in almost every case, with the exception of MICs which are primarily used by jobs in *longpar*; the explanation is fairly simple, i.e. jobs in *parallel* and in *longpar* are the most computationally intensive ones and require more resources, whereas *debug* contains only light and relatively short lived jobs and *reservation* comprises only special jobs, much fewer than the other queues.

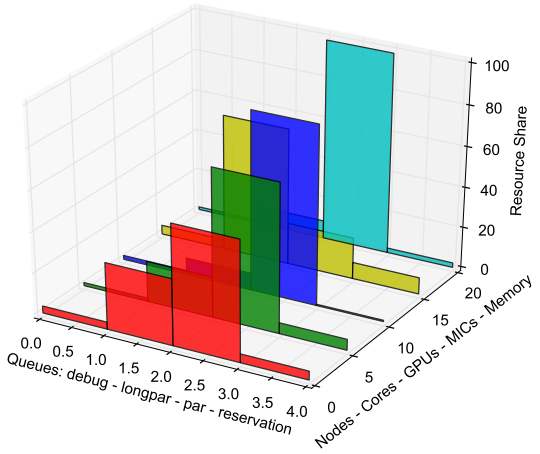


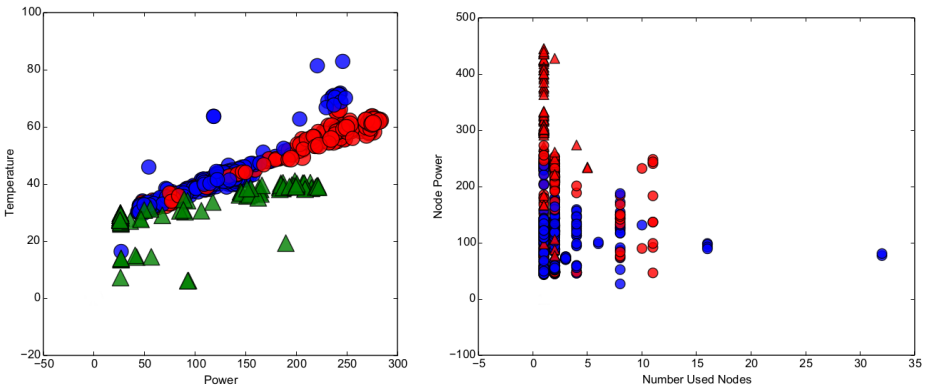
Figure 4.4: Percentage of requested resources, grouped by queue

Figure 4.5a plots for each jobs the average core temperature and CPU and GPU power; in the y-axis there is the average core temperature in °C and in the x-axis there is the average consumed power in Watt. The figure gives this information for both CPUs and GPUs, whose values are respectively identified by circles and triangles. The red circles are the jobs which run on the 3.1GHz nodes while the blue ones are the job in the 2.1GHz nodes; the green triangles represent jobs which required at least a GPU - and necessarily executed on a 3.1GHz node[4]. There is a clear linear relation between the temperature and power for the CPUs, with the higher frequency nodes almost always consuming and heating more than their lower frequency counterparts (except for a handful of very power consuming jobs on 2.1GHz nodes - top right corner). The same is true for the GPUs where the linear relation is less steep, suggesting a lower package thermal resistance.

Figure 4.5b portrays the relation between consumed power and number of nodes used for each job. In y-axis we see the power in Watt, computed as the integral of the CPUs and GPUs power consumption for each node contributing to a given job. The x-axis reports the number of nodes used by the job. Again red circles for jobs using 3.1GHz nodes - but with no GPU - and blue ones for the 2.1GHz nodes; the read triangles stand for the jobs that used also a GPU. The majority of the jobs used only few nodes (between 1 and 5) and for the same number of used nodes the jobs with the higher power consumption are those running on, in this order, 3.1GHz with GPU, 3.1GHz without GPU and 2.1GHz nodes. It must also be noted that there is a clear trend showing a reduction of the node average power as the number of node used rises. This can be explained by the increase in the communication-to-computation ratio in larger applications.

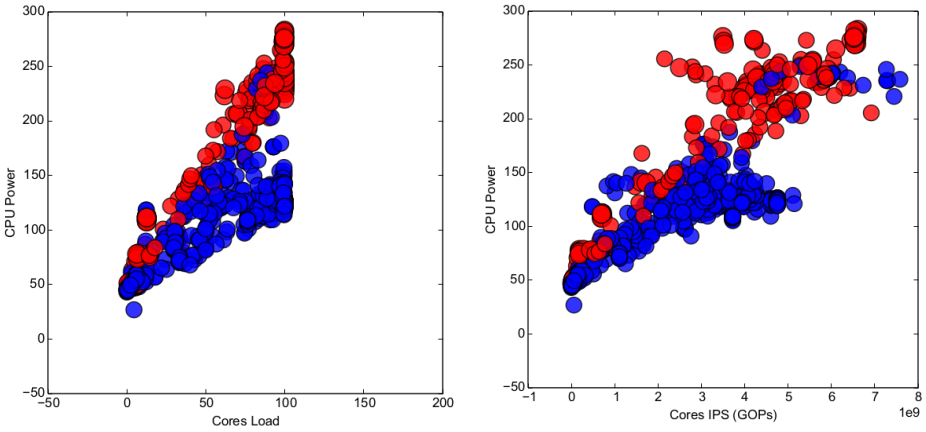
Figure 4.6a plots the CPU power (Watt) in the y-axis and the cores average

[4]GPUs are mounted only on nodes with 3.1GHz frequency



(a) Temperature (C) VS Power(W) (b) Power VS # Nodes

Figure 4.5: (a) Temperature and power for CPU and GPU; (b) Power Consumed and number of nodes used by job.



(a) Power (W) VS Load (b) Power (W) VS IPS (GOPs)

Figure 4.6: (a) Power and Load; (b) Power and Instructions Per Seconds.

load for the duration of the job is in the x-axis; the load is computed as the mean of all the loads of the cores used by a jobs (usually more than one) and it ranges between 0 (idle core) and 100 (core at maximum capacity). There is clearly a strong positive correlation between the core load and the power consumption, markedly more for jobs executed on higher frequency nodes - we maintained the same coloration seen in the previous graphs (blue and red circles) but here we do not differentiate between jobs that use a GPU or not.

Figure 4.6b shows the relation between the power consumed (Watt) by a job and its mean Istruction Per Second (IPS). In the y-axis we show the CPU power and on the x-axis the IPS measure; we again used the red and blue circles (no distinction for the GPU usage). In this case the plot is quite scattered but we can nevertheless observe the relationship between these measures, with the

power increasing as the IPS increases.

The proposed database can be used also to understand the relation between a user and the properties of its submitted job. This aspect can have a crucial significance for system owners and users, as it allows to devise different treatments and accounting mechanisms for different types of users. In Figure 4.7 we plotted the power consumed by a job (in Watt) on the y-axis and the duration of the job on the x-axis and we assigned a different color for each user - i.e. all the blue circles are jobs submitted by the same user. In the plot we reported only five users chosen randomly to keep the plot readable. Jobs with higher energy consumption fall in the top right corner of the plot. From this figure we can recognize that some users (such as the one identified by light-red circles) submit jobs without any discernible pattern - at least for the metrics considered in this plot - whereas others (like the yellow circles) always submitted quite similar jobs. This information can be used by user-aware dispatching to achieve machine power balancing or to avoid abrupt power changes in the delivery network.

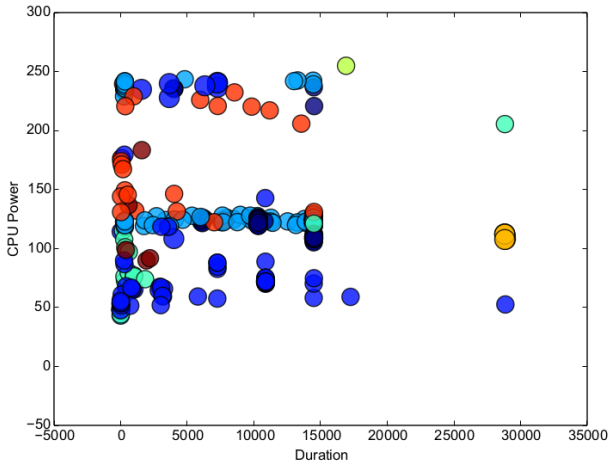


Figure 4.7: Power and duration of a job, grouped by user

4.2 Job Power Profiling

After having described the data collecting system we are now going to focus on the most important task of such a system (at least for the purpose of this work): how to predict the power consumption of HPC applications. In this section we begin with discussing and empirically validating two extremely important key concepts: 1) the power of a job can be approximated with its average value keeping a good accuracy; 2) a method to compute the power consumed by jobs which required only portion of nodes. In Section 4.3 we are going to explicitly introduce the prediction models.

As we observed previously, multiple jobs can run concurrently on the same node: while we can directly measure only the power consumed by the CPUs

the jobs can be allocate to single cores. Moreover, we know on which node a job run but we cannot distinguish the exact CPU it used (two CPUs per node). Therefore we cannot directly measure the power consumption of applications which do not occupy entire nodes. The number of jobs using only node portions is not negligible at all, since that is actually the majority of jobs which run on Eurora supercomputer[5]. This kind of problem may emerge in many different HPC systems due to the various power measurements methodologies found in current supercomputers [SSWeA14]. In the following sections we explain how we deal with such a problem.

While the power consumption of a job depends on all the HW resources used (CPUs, GPUs, etc.) in the rest of this section we are going to focus on the power consumption of the CPUs, thus disregarding HW accelerators. CPUs power consumptions are the most difficult to deal with, due to the fact multiple jobs can run on the same CPUs. Currently, HW accelerators cannot be shared by multiple jobs. Therefore from now on with "power" we are referring to the CPU power consumption.

As already mentioned we must make a distinction between two kinds of problem: jobs requiring entire nodes (one or more) and jobs requiring only a portion of a node. In the first case we can simply use the power measures we collected; in the latter case we need a method to compute the power consumptions with the data at our disposal. This method is described in Section 4.2.1.

A HPC application power consumption may vary during its lifetime due to the nature of the application itself and of the different phases which compose it, but the impact of such variability has not been extensively studied. Conversely, if the power consumption was constant - i.e. if we can associate to each job a precise, single value - the task of a job dispatcher would be greatly simplified[6]. Our idea is to use the *mean* power to represent the power consumption of a job, in the hope that the power variability is relatively low and the power consumption relatively constant or that when we add all the job power traces the variability of the power of each job is compensated by the others. This mean value is calculated as the average of all power measurements collected during the job lifetime.

Now we are going to consider jobs which executed occupying entire nodes in order to illustrate why we do not lose too much information with our approximation. We chose to use these jobs for the sake of simplicity but the same considerations can be applied to the case of applications running on node portions. In Figure 4.8 we can see the power consumption profiles of three different jobs, A (Fig. 4.8a), B (Fig. 4.8b) and C (Fig. 4.8c). The first two jobs present a similar profile: their power consumption is quite constant and with a relatively small variability. In particular the power consumption of job B shows very little variability while job A powers have a higher variance - but still quite small compared to the mean value. As job C reveals, this is not always the case: there are also jobs whose power consumption changes more drastically during their lifetime (see the sharp increase in the power consumed by job C). Nevertheless, we can still estimate a job power consumption through its mean (average) value

[5]It is probably due to the fact that Eurora was originally a prototype and only later entered production phase

[6]Proactive job dispatchers prefer to know in advance the job power consumption as a single value; they could theoretically manage the power as a more complex object (i.e. a curve instead of a single value) but we risk to incur in significant performance losses

because the jobs with significant power variability are only the minority of all the jobs. The overwhelming majority of jobs show a low standard deviation in their power consumptions.

In Fig. 4.8d finally we see the histogram of the "normalized" standard deviations distribution. For each job we first compute the mean and the standard deviation of all the power measurements then we divide the standard deviation by the mean value to obtain a normalized standard deviation (to let us compare standard deviations of different jobs). We can easily see that for the vast majority of the jobs the standard deviation of the power consumptions is less than 10% of the mean value - and in many cases even less than 5%. This means that the standard deviation is, on average, very small and consequently the power variability is not too big. This is probably a characteristic of HPC jobs, which are generally carefully tuned to avoid big workload changes during the key computation phases.

This observation allows us to estimate the power consumption with the mean value without losing too much accuracy. It is clear that when associating a single power to a job we are going to lose some information about the real power consumption and therefore commit a certain (small) error. A job dispatcher with power capping is interested in the total power consumption of the system: when we sum all the jobs the errors tend to compensate each other thus the average error we commit is low.

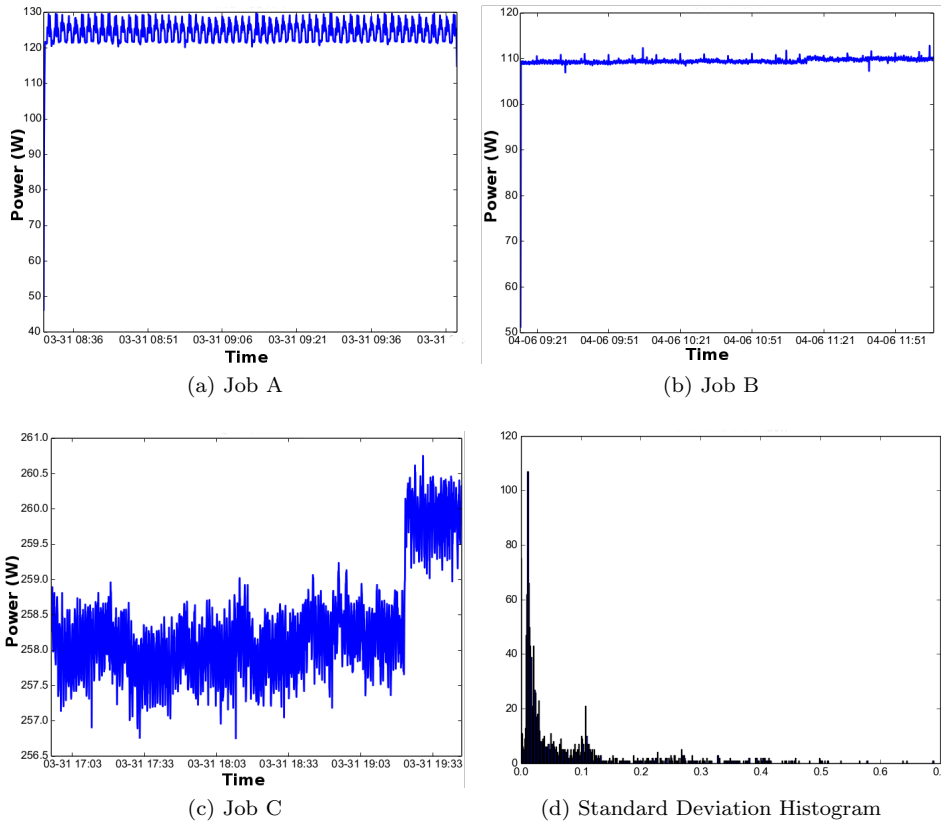
4.2.1 Shared Resource Power Consumptions

We describe now the technique to estimate the power of jobs running on portions of nodes. As discussed before we do not have a direct measurement of the power consumed by jobs using only a portion of a node. In this section we present a method to compute a power for these jobs. The main idea behind our approach is that each job consumes an amount of power proportional to its requirements (more specifically the number of requested cores). We are thus making the assumption that the resource requirements declared by the users are truthful and constant (at least up to a certain degree).

An important requirement of our approach is the knowledge of the number of running jobs and "active" cores in each node at every time. Number of active cores means the number of cores that should be used in a node given the number of cores requested by all jobs running on such node. We therefore created these node profiles using the historical job traces. The information regarding the number of cores active on a node at any given time is fundamental for our approach since it allows us to understand the amount of power associated with a job: if a job j runs alone on node i the number of active cores is equal to the number of cores requested by job j and all the power consumption can be attributed to that job. Conversely if more jobs are running concurrently, each job will contribute to only a portion of the whole power (i.e. with 2 jobs sharing a node, using all the node cores, each job can be associated with half the measured power). This reasoning motivates the power we associate to each job as expressed in Equation 4.1

$$P_j = \overline{P_{ij}} \frac{cr_j}{nca_{ij}} \tag{4.1}$$

where P_j is the power associated to job j, cr_j is the number of cores required by



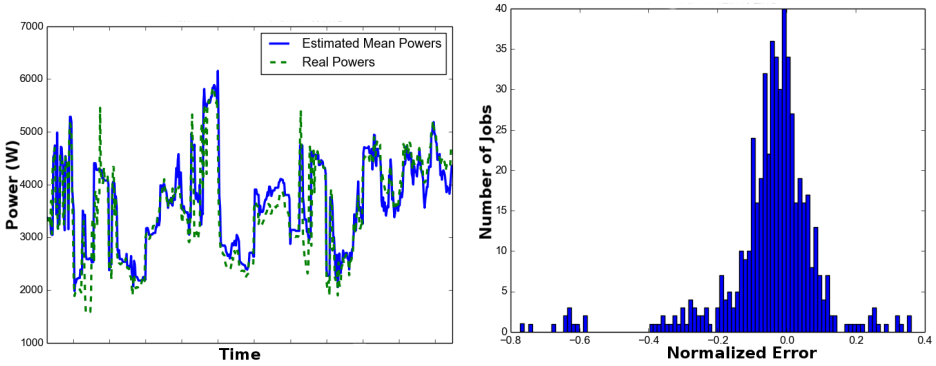
(a) Job A (b) Job B

(c) Job C (d) Standard Deviation Histogram

Figure 4.8: Real power consumptions of 3 jobs (A, B and C) and standard deviation distribution histogram

job j, $\overline{P_{ij}}$ is the average power of the node i on which the job j has run during the job lifespan and nca_{ij} is the weighted average number of cores which were active on node i during the duration of job j. In practice this equation says that the power associated to a job depends on the amount of workload related to the job - i.e. how many of the active cores on the nodes were used by the job.

While $\overline{P_{ij}}$ and cr_j are information stored in our database and ready to be used, nca_{ij} needs to be computed. This value tells the average number of cores (related to the number of concurrent jobs) which were active during the lifetime of the job; this number can be computed using the node profiles described previously, which can tell us the number of active cores in any moment of the job duration. More precisely we divide the job lifespan in sub-intervals where the number of active cores is constant and then we compute the average number of cores, weighted by the sub-interval duration. To formalize, given a node i we suppose to have job j, with duration Dur_j, which starts at ST_j and terminates at ET_j. We then have a set of intervals $s \in j$, each one with duration Dur_s; in a sub-interval the number of active cores is nc_s. The weighted number of active



(a) Computed VS Real Power (b) Error Histogram

Figure 4.9: Comparison between the real aggregated power and the computed aggregated power. Mean Error: 0.011

cores can be computed as follows:

$$nca_{ij} = \sum_{s \in j} nc_s \frac{Dur_s}{Dur_j} \tag{4.2}$$

We designed an experiment to test the accuracy of the approximation method for jobs using node portions. For a given time interval we compare the total real power of the system, computed as the sum of all the node power measurements at each given time (we use time sub-intervals of 5 minutes). Then, for every time considered we also compute the total estimated power, i.e. the sum of the powers of the jobs running at that time; the power of each job is calculated with the method discussed above. A clear limitation in this approach is the fact that we are comparing two aggregate metrics (total real power and total estimated power) and we do not calculate the single job error.

In Figure 4.9 some results are shown - the trace corresponds to a one-day period. As we can see the results are remarkably good, for example in the period considered we can see that our power estimate has an accuracy around 99%. The results can nevertheless be worse in general, but the accuracy for our whole data set is always between 95% and 96%. Part of the error we observe is not due to the method itself, but actually is caused by the imperfect data in our possession. For example, our algorithm strongly depends on the traces of the jobs which run on Eurora and particularly on the resource requirements declared by the users. This can be a source of error because users declare the peak requirement of their application but the average usage may be lower (i.e. asking for 4 cores but actually using just one on average). Nevertheless, there is a discrepancy between the resource requirements declared by the users and the resource actually used and such discrepancy is going to be a problem for the prediction model. We discuss this issue in Section 4.3.3.

To summarize, the proposed method has a clear drawback in the fact that we compute the job power making reasonable but not verifiable (and actually not always true) assumptions: 1) we consider the number of cores requested to be an accurate estimate of the job activity for its whole duration (whereas jobs could

require a certain number of cores and actually use them in a variable pattern), 2) we assume that a job will always use a core at its fullest capacity (while actual load measurements from the cores reveals that the load is not always zero or 100%), 3) we consider the average power during the lifetime of a job, when in reality the power during the job duration may vary. All these assumptions force that our computation may not produce exact values of job powers but rather an "approximation". Another big limit in this method is the issue of incomplete information. For example we could not consider the missing jobs in our DB: we know for certain (comparing the real measured load and the job traces) that our job collecting system is faulty and some jobs were not correctly stored on the DB. Hence, in some periods we fail to consider the correct number of jobs and active cores in our computation. This has a non negligible impact on the error introduced by our computation: preliminary experiments have shown that the accuracy of our method clearly increases in the time intervals where the jobs were correctly stored in the DB.

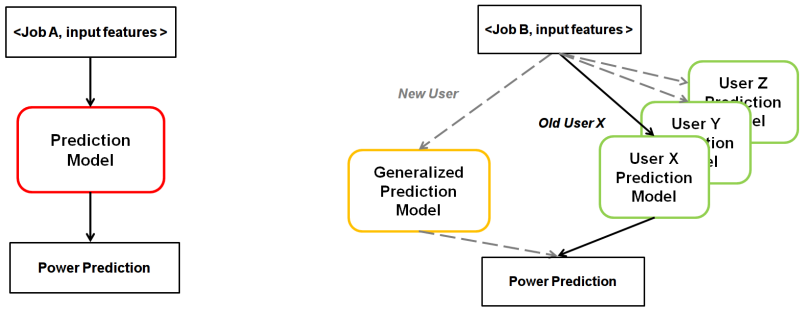
4.3 Powers Prediction Model

In the previous sections we have seen that it is possible to describe a job power consumption with a single value - the mean value - and we are able to do that for every job running on the system, whether it occupies an entire node or not. With this information we can now create a prediction model to estimate jobs power consumptions. For this purpose we employed a Machine Learning approach (ML) which relies on the large amount of historical data in our possession: using the knowledge we have on the applications that run in the past we can learn a model able to predict the consumption of future jobs. We implemented our machine learning models with a Python module called *scikit-learn* [PVGeA11].

The basic idea of a machine learning predictor is to learn a model which correlates a set of *input features*, or independent variables, with a *target*, or dependent variable. In our particular context we want to correlate the job characteristics (duration, requirements, etc), i.e. the features, with the job power consumption. This correlation can be learned by the model thanks to the large number of example which constitutes the training set, i.e. the data regarding past jobs, with their characteristics and power consumptions. After the learning phase the model can estimate the powers for new jobs (not seen during the training). A critical element for the success of ML techniques is the availability of large data sets for the training phase; in the Eurora's case we have a data set comprising tens of thousands of jobs (more or less 100k), which is more than enough to obtain good quality predictions.

We developed two different approaches, one to be used with jobs requiring entire nodes (Sec. 4.3.1) and one to be used with jobs occupying only portions of nodes (Sec. 4.3.2). In this second case, we created multiple prediction models: one for each user with already enough collected data plus a generic one for new users.

The reason behind this diversification is based on the different type of data at our disposal: we can be sure of the power associated to jobs which run on entire nodes (see previous sections) but the power associated to jobs using node portions is the result of an approximation that already introduces some inaccuracy. In order to assure a fair comparison between the estimates and



(a) Jobs on Exclusive Resources

(b) Jobs on Shared Resources

Figure 4.10: Predictors Scheme

the real power data, thus excluding sources of error not directly referred to the prediction itself, we preferred to keep two separate prediction models. We start describing the prediction model of the jobs on entire nodes and we later continue with the jobs using node portions.

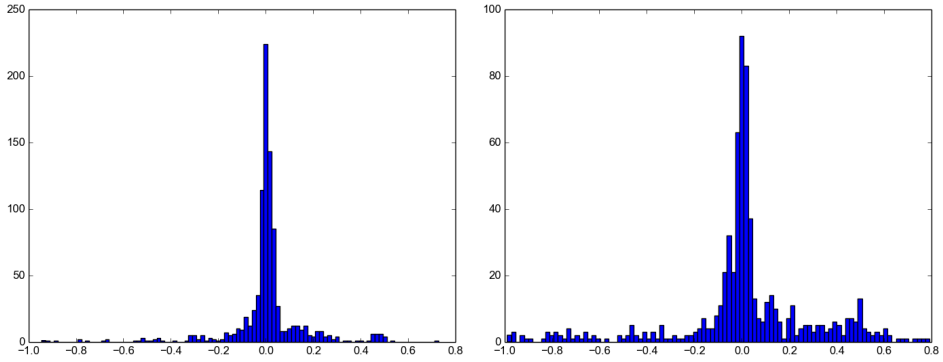
In Figure 4.10 we can see the schemes of both predictors, on the left the predictor for jobs running on entire (or multiple) nodes and on the right the predictor(s) for jobs running on shared nodes. We are going to discuss them more in detail in the following sections.

4.3.1 Exclusive Resources

In the case of jobs on entire nodes we created a single model which takes into account the following features: user, queue, requested duration, number of requested nodes, number of requested cores, number of requested GPUs, number of requested Xeon Phi[7], amount of requested memory. The ML method we used for this model is called Random Forest Regression [Bre01], which is an evolution of the classical and widely used Decision Tree [Qui86]. Other than Random Forest we tried other supervised regression techniques, using the default implementations provided by sci-kit. We used Generalized Linear Model, Support Vector Machines, Decision Trees and ensemble methods (which combine the predictions of several base estimators) such as Bagging, AdaBoost and Gradient Tree Boosting. We then chose the approach with the best accuracy. The time required to train the model is very low, less than 3 seconds, and the time required to make a prediction is negligible, much less than a second. The training should be performed once with the available historical data and then the model could be updated regularly with the new jobs' information collected during the normal execution.

A common way to compute the quality of a prediction model is to split the data set into two subsets: the training set and the test set. With the training set we can learn the prediction model, which is then used to estimate the power

[7]The number of requested HW accelerators is important because GPUs and Xeon Phi are mounted on computing nodes with different power consumptions, i.e. a job requiring a GPU will necessary run on a CPU consuming more power than those with a Xeon Phi



(a) Test Set A: 943 Jobs (b) Test Set B: 714 Jobs

Figure 4.11: Prediction errors histograms for two test sets

consumptions of the jobs in the test set. These estimates are then compared with the real power consumptions of the jobs in the test set, computing the "prediction error" for each job in the test set[8]. The average of these prediction errors then measures the quality of the prediction, with lower values indicating higher quality. The average prediction error computed in this way is around 4%-5%, which guarantees a very accurate prediction. Figure 4.11a and Figure 4.11b show the histograms of the prediction errors for two different test sets of jobs on entire nodes. These results were obtained with a training set of around 10k jobs and the test sets are, respectively, of 943 and 714 jobs (test and training sets randomly extracted from the whole data set).

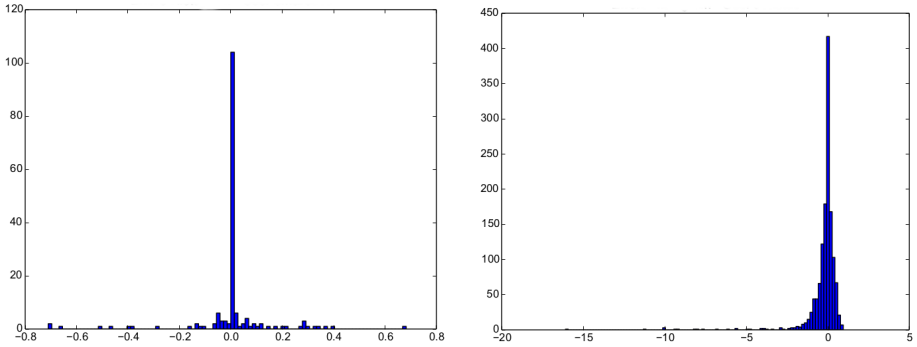
Figure 4.10a portrays the scheme of the predictor for job on exclusive nodes. As can be easily seen, it is fairly simple: when a job is submitted its features (defined by the job request) are fed to the single prediction models, which in turns produces the power consumption forecast as an output.

4.3.2 Shared Resources

In the case of jobs using only node portions the prediction turned out to be more difficult than the entire node case and that forced us to try with a different approach. In particular, we chose to create a prediction model for each user, since it is probable that a certain user will submit jobs with similar patterns. Also in this case the best ML technique has been selected after a preliminary study and the best performance was obtained with Decision Tree Regression [BFSO84] - also a Decision Tree based method. The problem with having one predictor per user is that we split our training (and test) set and therefore the dimension of these sets decreases. If the test set dimension decreases too much the learning loses its effectiveness and consequently we cannot make good predictions. This problem happens particularly if there are users who submitted few jobs and thus we cannot actually learn a prediction model for them[9]. To

[8] We actually used a normalized prediction error: $(real\_power - predicted\_power)/real\_power$

[9] This is also a problem for new users to whom we cannot build any prediction model until a sufficient number of jobs are submitted



(a) User A (b) User B

Figure 4.12: Prediction error histograms of two different users

solve this issue together with the single user predictors we also have a "general" predictor, devised without splitting the test set and including all users. This generalized predictor is slightly less accurate than the specific ones, but it is an essential component in our prediction mechanism.

The input features of the generalized predictor are the same used for the entire nodes models plus the following: the job name, the number of jobs running on the system at the job start time, the number of cores used on the system at the job start time and the ratio between active cores and total number of cores on the system at job start time. These additional information mainly serve to characterize the supercomputer state and we also added the job name (the application executable) since jobs with similar names - by the same users - usually represent different runs of the same application (and probably are similar in terms of power consumption). The specific (per user) predictors have all the input features of the generalized predictor minus the user name: obviously, since it would be the same for all the jobs in the test set. The training time is very fast also in this case, usually less than 2 seconds for each user predictor.

The accuracy of the prediction has been computed as before, i.e. using the average prediction error. Now we have different accuracies for the different user predictors and to obtain an aggregate measure we calculate the mean of all the average prediction errors (one average prediction error per user, plus the generalized one). The quality of the prediction is lower than in the case of jobs on entire nodes, with an average error around 15% - for some user the error could be smaller than 3% while for other users it could be up to 35%. In Figure 4.12 we can see the histograms of the prediction errors for two different users, User A and User B. In the case of User A (Fig. 4.12a) we see how the accuracy is very good since the errors are very small and centered on zero. Conversely Fig. 4.12b reveals that for User B the prediction is definitely less precise; it's easy to see that while the majority of jobs are well predicted (small error around zero), a few of them are extremely bad predicted (queue of errors much smaller than -1, on the left of the graph). In the following section we are going to describe the reason of this behaviour.

Figure 4.10b depicts the prediction scheme in the case of jobs running on shared nodes; this case is more complex than the entire nodes one. When a

job is submitted it can belong to a previously seen user (*Old User*, right hand side of the image) or to a user never seen before (*New User*, left hand side). In the former case, the job features are given as input to the corresponding user prediction model; if the user is new the job features enter the generalized prediction model. In any case, the chosen prediction model will produce a power estimate as output. The predictor models devised for the jobs running on node portions could be also used without modifications also to predict the power of jobs running on entire nodes (jobs on entire nodes are a subset of jobs on node portions). However, the accuracy of the prediction decreases w.r.t. to the dedicated predictor.

4.3.3 Outliers Management

After collecting the prediction results described previously we investigated them to understand why some predictions were so wrong. The first thing we noticed is that these bad estimated jobs have small powers (compared to the rest of the jobs). As we see in the next section, these "bad" jobs do not have a great impact on the aggregate prediction precision (that is, when we compare the total real system power consumption and the total predicted power) and could actually be disregarded. Furthermore we claim that these jobs are *outliers*, i.e. jobs not representing a typical Eurora workload. This is due to two factors. First, these jobs show "strange" behaviours: they usually have a very short duration (under 5 minutes and very often even less)[10]. This frequently means that these jobs did not have a correct execution and terminated abruptly, possibly without actually using the requested resources.

The second issue is related to the discrepancy between the declared resource requirements and the resources actually used - due to both incorrect estimates by users and varying levels of resource utilization during the job execution. The jobs whose estimates are extremely wrong present a significantly higher level of discrepancy between the declared resource requirements and the resources actually used. If we discard these outliers from the average prediction error computation, the accuracy increases sharply: the mean error becomes smaller than 4%. This consideration leads us to formulate two simple guidelines to obtain better predictions: 1) it is extremely important that the users declare realistic requirements and should be fostered to do that; 2) the job monitoring framework has to keep track of those jobs which did not terminate their execution correctly and must be able to distinguish them from the "successful" ones.

4.4 Experimental Results

In this section we present the conclusive results of the methods we introduced in the rest of the chapter: estimate of a job power consumption with the mean value of multiple powers measures and prediction of such mean powers with a ML model. We want to know the final error we obtain after having applied all the stages of our method, each of them introducing some inaccuracy. For this purpose we tested our predictions against the real system power consumptions

[10]We cannot just delete all jobs with short durations from the train set since in this way we could discard legitimate jobs

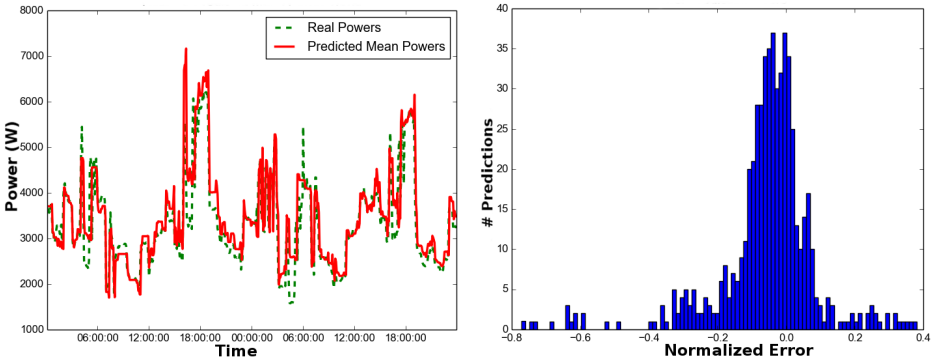
(again, we consider only CPU powers). The experiment set up is the same employed to compare mean power consumptions and real ones, that is we compare the total real power in the system at time t with the sum of all power predictions of jobs running at time t. The power predictions are obtained with the previously mentioned model and for the sake of simplicity we did not employ the dedicated predictor for the jobs running on entire nodes.

In our accuracy calculations we decided to disregard the outliers, as described in Section 4.3.3. In Figure 4.13 we can see some results; the figure corresponds to a two-days period. On the left the predicted trend is compared to the real power trend and on the right we can see the histograms of the prediction errors. As we can see the results are very good, with a mean error smaller than 6%. This is true also if we consider more extended periods; the average error for the whole test period (a month) is around 8% and 9%.

Although the average error is good, we must consider the nature of our mispredictions in relation to the power capping. More specifically we can observe *under-prediction* and *over-prediction*. The first one could lead to emergencies (the real power actually exceeds the cap planned in the dispatcher) and the second one leads to machine under-utilization and it is undesirable as well. The concerns about underestimates is that they may lead to power and thermal overshoots. More in detail, when the predicted power is below actual power for a time significantly longer than the thermal time constants of the HPC machine, and the prediction error is above 10%, then hardware power throttling would kick in at run time to prevent temperature overshoots, leading to undesirable performance losses and possible violations of service level agreements. We call these events "critical underestimates". Our error analysis reveals that critical underestimates would happen for less than 2% of the operating time of the machine. For example, in the 2-days period of Figure 4.13 we registered 37 periods when the predicted power was lower than the real one. While this could seem a high number, the great majority of these under-prediction periods were very short: 90% of the times the under-predictions lasted for less than 2 minutes and 80% of the under-predictions lasted less than 1 minute. The longest under-prediction period lasted for 8 minutes. These values are very short w.r.t. a typical HPC application duration (i.e. a few hours).

Our prediction model tends to underestimate the power consumption - the under-predictions are $\sim$70% of all errors. This is not fault of the prediction model itself but it is due to the previous phase (power estimation with mean values and for job in portions of node). The main reasons for this are the use of mean power instead of real ones and the discrepancy between the requested resources and those actually occupied. When we integrate the results of our prediction models in a job dispatcher with power capping we must take into account the under-predictions. Even though these results are very good we can nevertheless consider under-predictions and there a few ways to cope with them. On one hand, we can back up our dispatcher with a HW power cap mechanism to ensure never to exceed the power budget. Another solution could be to require the dispatcher to respect a power constraint tighter than the real one, thus guaranteeing never to surpass the desired power budget.

Data set size importance An extremely important factor for the quality of the prediction is the availability of information regarding previous jobs. The



(a) Predicted VS Real Power (b) Error Histogram

Figure 4.13: Comparison between the real total power and the predicted total power. Mean Error: 0.056

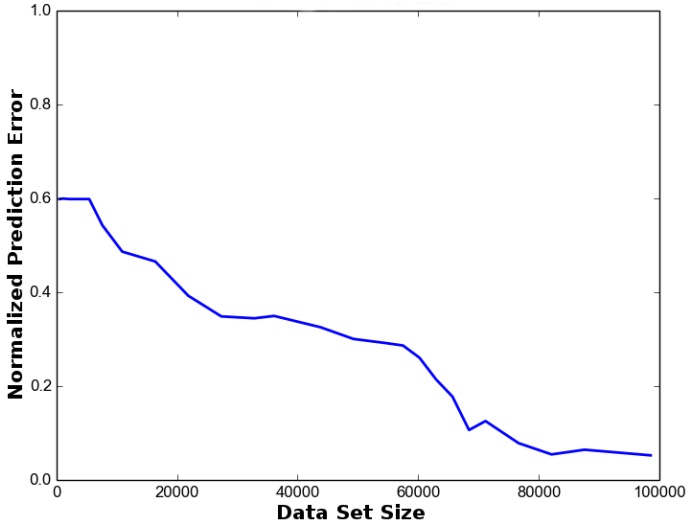


Figure 4.14: Data Set size and prediction error

size of the data set used to train our learning machine models greatly influences the results accuracy. To give an idea of the relevance of this point we can look at Figure 4.14. We have on the x-axis the increasing data set sizes (obtained through random sampling of our original data set) and on the y-axis the corresponding mean normalized error of the prediction. The mean error reported is the aggregated, final one. We can easily see that the prediction accuracy decreases (the error increases) when the data set size shrinks. A good dimension for the data set is around 80k entries (previously observed jobs)[11].

[11] In Eurora's case this corresponds to less than 3 months of observation

4.5 Chapter Summary

In this chapter we discussed a method to estimate the power consumptions of applications running on HPC systems. Such estimates represent a fundamental tool to assist the task of online job dispatchers: the capability of forecast power consumptions allows job schedulers to optimize the system management. In order to be of any real use the power predictions must be available at schedule-time, i.e. when scheduling & allocation decisions are taken, and therefore they must be based on information present at that time.

We have considered the Eurora supercomputer as a case study thanks to the data gathering infrastructure that collected valuable information (relative to the jobs which run in the past and to the measurements of physical sensors) for several months of the operating life of the machine. The data collected gave us the possibility to employ Machine Learning techniques and create a set of prediction models for the purpose of predicting power consumptions. The models created were calibrated on Eurora-specific data but the methodologies used can be easily generalized and applied to different supercomputers; the only requirement is to have a data collecting infrastructure in order to train new Machine Learning model. The prediction quality strongly depends on the availability of large amount of data.

Finally we can give two additional insights on the issue of power consumptions estimation. First, we can approximate the power consumption of a job with a single value, the average of all the power measurements taken during its lifetime, with only a small loss of precision. Dealing with a single values instead of multiple ones is a great help for job dispatchers. Then we tackled the problem of co-executing jobs, i.e. applications which run on the same node and using only a portion of the node resources. The problem arose from the mismatch between the minimal allocation unit in the system and the granularity of the power measures collected. We propose an algorithm to estimate the mean power consumptions of such jobs and proved its efficacy.

Chapter 5

HPC Job Dispatching under Power Cap Constraints

High Performance Computing systems are envisioned to reach the Exascale in 2023 but multiple challenges still have to be overcome [TC12, Sim12, BBHU10]. One of the most complex issues is the power consumption because if the energy efficiency does not improve dramatically the power consumption of next-generation systems will be unacceptable. With current technologies an Exascale supercomputer would consume around hundreds of MWatts while a commonly accepted upper bound for the power consumption of a supercomputer is around 20MW [BBCea08]. Such a huge challenge must be tackled from several angles: new, more energy efficient hardware components should be developed, novel power-aware workload and resource management software needs to be devised, HPC applications have to become more flexible, users should shift from a performance-only focus to more balanced perspectives, etc.

In the past years many research works have dealt with the HPC power management issue and several strategies have been proposed. *Power Capping* is a methodology widespread both in the research literature and in real world systems and it consists in keeping the power consumption of a HPC center within a power budget (the *power cap*). This purpose serves as a catalyst to accelerate the development of various techniques that actually implement such a constraint. Many mechanisms have been studied to reach the goal of a power capped supercomputer, ranging from a better management of the system resources to exchanging computational performance for reduced power consumption.

Contributions In this chapter we are going to introduce a novel power capping strategy for HPC systems, a job dispatcher which is able to keep the power consumption within a certain budget acting only on the jobs execution order. Many currently used mechanisms require a trade-off between reduced power consumption and application performance. For example if we force the power of a computational unit not to exceed a given budget, the jobs running on that

node would possibly be slowed down[1]. The performance decrease can be acceptable or it might lead to problems in terms of Quality-Of-Service for the users.

Our approach relies entirely on the benefits of optimized and proactive workload dispatching. We reckon that it is possible to achieve a power capped system through a careful planning of the workload execution, without incurring in a significant performance degradation. Nevertheless, a strong point of the proposed method is the possibility to combine our "smart" scheduling & allocation method with techniques that dynamically adjust the power of the computational resources, since these approaches are orthogonal and not mutually exclusive. The integration of the proactive dispatching policy and reactive mechanisms allows to better cope with the mutating operating conditions of the HPC facilities.

Outline The chapter content is organized as it follows. In Section 5.1 we briefly cover the background and the problem context. Then in Section 5.2 we present a job dispatching model with power cap for HPC system; we discuss two different approaches, one based on a heuristic algorithm and one based on a hybrid approach that combines a Constraint Programming model and a heuristic technique. Our methods are compared against the state-of-the-art in Section 5.3. Afterwards, in Section 5.4 we illustrate how to combine the job dispatcher with the cooling system of a real supercomputer, studying how this integration can lead to energy savings and better power management. Finally, in Section 5.5 we consider the problem of variable power budget and the way the job dispatcher can cope with this issue; in particular we present an additional module to reduce the power consumption of an already running HPC workload using a CP model plus Dynamic Voltage and Frequency Scaling (DVFS).

Publications Part of the work forming the core of this chapter has been published to international conferences in [BCLB15, BCL+15]. An extended version should appear on the journal *Transactions on Parallel and Distributed Systems*.

5.1 Context

As we saw in Chapter 2 (especially in Section 2.1.3) the issue of reducing the power consumption of supercomputers is strongly felt within the HPC community and many research avenues have been explored to address this problem. A common denominator among several proposed techniques is the goal of bounding the power consumption of a system under a certain budget – power capping. Nowadays power capping approaches can be grouped in four different (and quite broadly defined) areas: 1) techniques which exploit some system related characteristics (i.e. node variability) or make assumptions on the nature of the workload; 2) techniques employing some form of frequency scaling; 3) techniques employing Intels *Running Average Power Limit* (RAPL); 4) techniques based on heuristic or proactive algorithms and that work on the job execution order.

[1] The slowdown strongly depends on the nature of the workload

The methods based on specific workload features, such as the possibility of using moldable or malleable jobs, are limited in their adoption by their implicit assumptions, since most of today's supercomputers only allow rigid jobs (whose number of used cores and execution threads are fixed at dispatch time and cannot change dynamically). Conversely, the main drawback of methods employing some form of frequency scaling (DVFS) or physical power bound (RAPL) is almost unavoidable degradation[2] of the application performance – generally speaking, reduced power implies increased job duration.

A radically different and poorly explored direction is represented by *proactive* dispatching, i.e. planning in advance the execution of all the activities to be run (taking dispatching decisions for every job in the waiting queue). The dispatcher can then solve an optimization problem to find the best scheduling and allocation while respecting the power constraint. In this approach the dispatcher needs to have information about the power consumption of the tasks to schedule, in order to take the correct decisions. Since these power-related information are needed at schedule time, before the execution of the job, power consumption estimates are required. In the rest of the chapter we are going to discuss a job dispatcher of this kind. We combine here the insights that were gathered in the previous chapters. In particular we are going to consider again the HPC scheduling & allocation problem as in Chapter 3, but this time we want to obtain a *power-aware* dispatcher. A critical component for this dispatcher is the capability to estimate job power consumptions hence the prediction models discussed in Chapter 4 are going to be exploited.

5.2 Job Dispatcher with Power Cap

In this section we describe two job dispatchers with power capping. We consider two different approaches: 1) a heuristic algorithm, 2) a hybrid method which decomposes the problem and uses both Constraint Programming and a heuristic technique. The first method is faster while the second method manages to find the best solutions. A key point of our approach is that it requires the power consumption estimate for each job to be scheduled; this knowledge must be known *before* the actual job execution. Hence we assume that such a power consumption forecast is available for each application. Such estimate can be obtained through machine learning techniques (as discussed in Chapter 4).

5.2.1 Problem Definition

We give now a more detailed definition of the problem, i.e. job dispatching in a supercomputer subjected to a power cap. The problem is very similar to the one described in Chapter 3 (particularly in Section 3.4.2); in this chapter we consider the extended version where the power consumption must be constrained within a certain budget.

We have a set of jobs $J = \{j_1, ..j_{NJ}\}$. Every job $j_i \in J$ enters the system at a certain arrival time eqt_i, by being submitted to a specific queue (depending on

[2]Unless careful optimization is performed at task-level. The required level of detail and access to the inner components of a job is often not allowed in nowadays HPC systems, with the notable exception of a few research centers (i.e. Lawrence Livermore National Laboratory, US) with access to the source codes of their workload.

the user choices and on the job characteristics), $q_h \in Q$ where $Q = \{q_1, .., q_m\}$. Each queue is characterized by its expected waiting time ewt_h, which provides a rough indication of the queue priority. Each job specifies a maximal expected duration (its wall time) d_i. Each job is composed by a set of sub-units; the number of job units of job i is u_i. Each job unit starts and ends with the job, and requires a certain amount of resources.

HPC machines are composed by sets of nodes $N = \{n_1, .., n_{NN}\}$ and sets of resources $R = \{r_1, .., r_N R\}$, i.e. cores, GPUs and MICs. Each node $n_j \in N$ has a capacity cap_{jr} for every resource $r \in R$. If a resource is absent from a node the corresponding capacity is zero. Each job unit k of job i requires an amount of resource $req_{ikr}, \forall r \in R$. Each job unit has a (estimated) power consumption p_i; we assume that the overall job consumption is evenly distributed to all job units, $p_{ik} = p_i/u_i$. For the sake of uniformity the power can be seen as an additional resource, forming the new set $R' = R \cup \{power\}$; therefore $req_{ikr} = p_{ik} \quad if \quad r = power$.

The dispatching problem at time τ requires to assign a start time $st_i \geq \tau$ to each waiting job i and a node to each of its units. All the resource and power capacity limits should be respected, taking into account the presence of jobs already in execution. Once the problem is solved, only the jobs having $st_i = \tau$ are actually dispatched. The single activities have no deadline or release time (i.e. they do not have to end within or start after a certain date), nor the global makespan is constrained by any upper bound. The goal is to reduce the waiting times, as a measure of the Quality-of-Service guaranteed to the supercomputer users.

5.2.2 Heuristic Approach

The first approach belongs to a class of scheduling techniques known in the literature as *Priority Rules Based* scheduling (PRB) [Hau89]. The main idea is to order the set of tasks to be scheduled, constructing the ordered list by assigning priority for each task. Tasks are selected in the order of their priorities and each selected task is assigned to a node; even the resources are ordered and the ones with higher priority are preferred - if available. This is a heuristic technique and it is not able to guarantee an optimal solution but has the great advantage of being extremely fast.

The jobs are ordered w.r.t to their expected wait times, with the "job demand" (job requirements multiplied by the job estimated duration) used to break ties. Therefore, jobs which are expected to wait less have higher priority, subsequently jobs with smaller requirements and shorter durations are preferred over heavier and longer ones. The mapper selects one job at time and maps it on a available node with sufficient resources. The nodes are ordered using two criteria: 1) at first, more energy efficient nodes are preferred (i.e. cores that operate at higher frequencies also consume more power) 2) in case of ties, we favour nodes based on their current load (nodes with fewer free resources are preferred[3]).

The PRB algorithm proceeds by iteratively trying to dispatch all the activities that need to be run and terminates only when there are no more jobs to

[3]This criterion should decrease the fragmentation of the system, trying to fit as many job as possible on the same node

dispatch. We suppose that at time $t = 0$ all the resources are fully available, therefore the PRB algorithm starts by simply trying to fit as many activities as possible on the machine, respecting all resource constraints and considering both jobs and nodes in the order defined by the priority rules. Jobs that cannot start at time 0 are scheduled at the first available time slot. At each time-event the algorithm will try to allocate and start as many waiting jobs as possible and it will keep postponing those whose requirements cannot be met yet.

Those jobs which could not be started at time $t = 0$ will be considered at the next time-event, defined as the earliest time point when some resources may become available, i.e. when a job currently running terminates. Then at the following time-event t' the algorithm will try to allocate and start as many waiting job as possible and it will keep postponing those whose requirements cannot be met yet.

The algorithm considers and enforces constraints on all the resources of the system, including power. This means that power is seen as an additional resource: a job can be scheduled in the machine if there are enough available physical resources (such as cores or GPUs) and if adding its predicted power to the current system consumption would not cause a violation of the power budget.

Algorithm 1 shows the pseudo code of the PRB algorithm. Lines 1-6 initialize the algorithm; J is the set of activities to be scheduled and N is the set of nodes (ordered with the $orderByRules()$ function which encapsulates the priority rules). At every iteration - until all jobs have been scheduled (line 7) - the algorithm tries to schedule all the units belonging to a job (line 9). The function $checkFit(u_i, n)$ (line 12) determines if there are enough available resources to run job unit u_i on node n - power is considered as well. If the result is positive, the unit is mapped and the system state is updated ($updateUsages(u_i, N)$, line 13), vice versa we register that at least a job unit could not be mapped (line 16). Only if all its units have been mapped a job can actually start (lines 17-21). If the job cannot start the changes to the system must be undone (line 24). After all jobs that can start at time t have been scheduled, the algorithm proceeds to the next time point, that is the closest moment when some resources will become free (i.e. the earliest end time among the running activities, line 26).

We note that since in this problem we have no deadline on the single activities, the PRB algorithm will always find a feasible solution, for example delaying the least important jobs until enough resources become available due to the completion of previously started tasks. To prevent job starvation the job priority grows with the time spent in the waiting queue (line 25).

5.2.3 Hybrid Approach

The key idea of the hybrid method is to decompose the allocation & scheduling problem in two stages: 1) obtain a schedule using a relaxed CP model of the problem; 2) find a feasible mapping using a heuristic technique (see Figure 5.1).

Since we use a relaxed model in the first stage, the schedule obtained may contain some inconsistencies; these are fixed during the mapping phase, thus we eventually obtain a feasible solution, i.e. a feasible allocation and schedule for all the jobs. This two stages are repeated n times, where n has been empirically chosen after an exploratory analysis, keeping in mind the trade-off between the quality of the solution and the computational time required to find one. To make

Algorithm 1: PRB algorithm

```
1  time ← 0
2  startTimes ⟵ ∅
3  endTimes ⟵ ∅
4  runningJobs ⟵ ∅
5  orderByRules(R)
6  orderByRules(J)
7  while J ≠ ∅ do
8      for j ∈ J do
9          for u_i ∈ j do
10             for n ∈ N do
11                 canBeMapped ← true
12                 if checkFit(u_i, n) then
13                     updateUsages(u_i, N)
14                     break
15                 else
16                     canBeMapped ← false
17         if canBeMapped = true then
18             J ⟵ J − {j}
19             runningJobs = runningJobs ∪ {j}
20             startTimes(j) ← time
21             endTimes(j) ← time + d(j)
22         else
23             undoUpdates(j, N)
24     orderByRules(R)
25     orderByRules(J)
26     time ← min(endTimes)
```

this interaction effective, we devised a feedback mechanism between the second and the first stage, i.e. from the infeasibilities found during the allocation phase we learn new constraints that will guide the search of new scheduling solutions at following iterations.

We implement the power capping requirement as an additional constraint: on top of the finite resources available in the system such as CPUs or memory, we treat the power as an additional resource with its own capacity (i.e. the user-specified power cap), which we cannot "over-consume" at any moment [Col14]. In this way, the power used in the active nodes (i.e. those on which a job is running) summed to the power consumed by the idle nodes will never exceed the given threshold.

The next sections will describe in more detail the two stages of the decomposed approach.

5.2.3.1 Scheduling Problem

The scheduling problem consists in deciding the start times of a set of activities $i \in I$ satisfying the finite resource constraints and the power capping constraint.

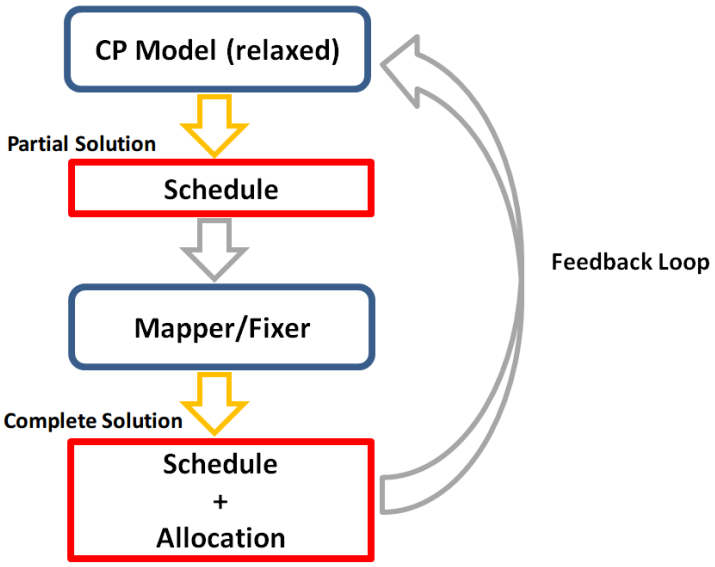


Figure 5.1: Decomposition Scheme

Since all the job-units belonging to the same jobs must start at the same time, during the scheduling phase we can overlook the different units since we need only the start time for each job. Whereas in the actual problem the resources are split among several nodes, the relaxed version used in the two-stages approach considers all the resources of the same type (cores, memory, GPUs, MICs) as a pool of resources with a capacity Cap_r^T equal to the sum of all the single resources capacities, $Cap_r^T = \sum_{k \in K} cap_{k,r} \quad \forall r \in R$. As mentioned before the power is considered as an additional resource type of the system, so we have an extended set of indexes $R^{'}$ corresponding to the resource types (cores, memory, GPUs, MICs plus the power); the overall capacity Cap_{power}^T is equal to the user-defined power cap.

We define the scheduling model using Conditional Intervals Variables (CVI) [LR08]. 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 $x(\tau)$ (if not present it does not affect the model). CVIs can be subject to several different constraints, among them the `cumulative` constraint [BLLN06] used to model finite capacity resources.

$$\forall r \in R^{'} \quad cumulative(\tau, req_r, Cap_r^T) \tag{5.1}$$

where τ is the vector with all the interval variables, where req_r are the job requirements for resource r; as a remainder, R' is the set of resources including the power. Since in the scheduling phase we are not distinguishing different jobs units, the power associated to a job is the overall predicted consumption (p_i). The cumulative constraints in 5.1 enforces that, at any given time, the sum of all job requirements will not exceed the available capacity (for every resource type).

With this model it would be easy to define several different goals, depending

on the metric we want to optimize. In order to improve users satisfaction while maintaining a high job turnaround, we decided to use as objective function the *weighted queue time*, i.e. we want to minimize the sum of the waiting times of all the jobs, weighted on estimated waiting time for each job (greater weights to job which should not wait long, for example those in queues with higher priority):

$$\min \sum_{i \in I} \frac{\max ewt_i}{ewt_i}(s(\tau_i) - q_i) \tag{5.2}$$

Search Strategies To solve the scheduling model we implemented a custom search strategy inspired by the Schedule Or Postpone strategy [PCVG94]. The classical Schedule Or Postpone tree-search algorithm selects a job and opens a choice point: on the left branch the earliest start time is assigned; the right branch used in backtracking the activity is postponed, i.e. marked as non selectable until propagation modifies its earliest start time (thus the variable is postponed). Basically, this strategy either manages to start a job at the current time or postpones it when this is not possible, due to the propagation generated by the involved constraint. Usually the order in which the jobs are considered follows the earliest start time ordering, from lowest to highest.

We extended this search strategy developing a custom strategy we called *Weighted-Random Set Times Forward*. The search algorithm is the same but the variable selection criterion changes. In our approach each job is chosen with probability p, drawn from a weighted random distribution, i.e. variables with higher weight have a higher chance of being selected. The job weights mimic the priority rules used in the heuristic algorithm, thus giving precedence to jobs that can start first and whose resource demand is lower. When a variable is selected we create again a split in the search tree, assigning the earliest start time on the left branch and postponing in the right branch.

This strategy proved to be very effective and able to rapidly find good solutions w.r.t. the objective function we are considering in this problem. With different goals we should possibly change the search strategy accordingly (as with the priority rules).

5.2.3.2 Allocation Problem

The allocation problem consists in mapping each job unit on a node. Furthermore, in our approach the allocation stage is also in charge of fixing the infeasibilities possibly generated in the previous stage. In order to solve this problem we developed an algorithm which falls in the PRB category. Typical PRB schedulers decide both mapping and start times, whereas in our hybrid approach we need only to allocate the jobs.

The behaviour of this algorithm (also referred as *mapper*) is close to the heuristics PRB algorithm described in Section 5.2.2 and in particular the rules used to order jobs and resources are identical. The key difference is that now we already know the start and end times of the activities (at least the possible ones, they may change if any infeasibility is detected). This algorithm proceeds by time-steps: at each time event t it considers only the jobs that should start at time t according to the relaxed CP model described previously – the difference with the previously described PRB algorithm is that the former considers at each time-event all the activities that still need to be scheduled.

During this phase the power is also taken into account, again seen as a finite resource with capacity defined by the power cap; here we consider both active and idle nodes powers. If the job can be mapped somewhere in the system the start time t from the previous stage is kept, otherwise - if there are not enough resources available to satisfy the requirements - the previously computed start time is discarded and the job will become eligible to be scheduled at the next time-step $t^{'}$. At the next time event $t^{'}$ all the jobs that should start are considered, plus the jobs that possibly have been postponed due to scarce resources at the previous time-step. Through this postponing we are fixing the infeasibilities inherited from the relaxed CP model.

Again, since in this problem we have no constraints on the total duration of a schedule, it is always possible to delay a job until the system has enough available resources to run it, thus this method is guaranteed to find a feasible solution.

5.2.3.3 Subproblems Interaction

We designed a basic interaction mechanism between the two stages with the goal to find better quality solutions. The main idea is to exploit the information regarding the infeasibilities found during the allocation phase to lead the search of new solutions for the relaxed scheduling problem. In particular whenever we detect a resource over-usage at time τ which requires a job to be postponed during the second stage of our model we know that the set of job running at time τ forms a *Conflict Set* (not minimal), i.e. not all activities in the set can run concurrently. A possible solution for a conflict set is for example to introduce precedence relationships among activities (thus forcing an ordering among the jobs in the conflict set) until the infeasibility is resolved.

In our approach we use the conflict set detected in the mapping phase to generate a new set of constraints which impose that not all jobs in the set will run at the same time. We then introduce in the CP model a fake cumulative constraint for each conflict set. The jobs included in this cumulative constraint are those belonging to the conflict set, each of them with a "resource" demand of one; the capacity not to be exceeded is given by the conflict set size minus one. These cumulative constraints enforce that the conflicting jobs will not run at the same time. This mechanism does not guarantee yet that the new solution found by the CP scheduler will be feasible since the detected conflict sets are not minimal, nevertheless it provides a guide for the CP model to produce solutions which will require less "fixing" by the mapper.

In conjunction with the additional cumulative constraint at each iteration we also cast a further constraint on the objective variable in order to force the new solution to improve in comparison to the previous one.

5.2.3.4 Difference with classical LBBD

The hybrid method we have described was loosely inspired by the Logic-Based Benders Decomposition (LBBD) technique [HO03]. LBBD is a widely used decomposition methods that have been proved to be very effective at dealing with Scheduling & Allocation problems [CHHSW16, CLCS10, CH10, TB12, FZB09]. LBBD main idea consists in decomposing the original problem in two components, a master problem and one or multiple subproblems.

This method divides the problem variables in two groups x and y, assigns values to x by solving the master problem (containing only variables in x) to optimality, so as to define a subproblem containing only the variables belonging to y. It then tries to solve the subproblem: if the trial values from the master problem are not acceptable (i.e. it is impossible to find a solution for the subproblem) a *no-good* (or *cut*) is generated and new trial values are generated solving again the master problem with the additional constraint. The success of the decomposition depends on both the degree to which decomposition can effectively exploit underlying structures and the quality of the cuts inferred. One of the main disadvantage of the LBBD is the risk of having a slow convergence rate, if the problem is not well suited to be decomposed in the required fashion.

Even though our decomposed approach formally speaking is not a Benders decomposition, we employed a similar partitioning of the original problem plus a feedback loop that shifts no-goods from the subproblem to the master problem. Standard LBBD decomposition for the Scheduling & Allocation problem usually considers the allocation stage as master problem and the scheduling phase as subproblem. In our hybrid method we swap these two phases: our master problem is the scheduling and allocation is the subproblem. Moreover, in our master problem we employ a relaxation of the original scheduling problem.

5.2.4 Preliminary Results

As a starting point we want to investigate the kind of impact that introducing a power capping feature may have on the dispatcher behaviour and performance. The dispatcher we realized can work in two different modes: *offline*, i.e. the resource allocation and the schedule are computed for each job before the actual execution, and *online*, i.e. allocation and scheduling decisions are taken at run-time, upon the arrival of new tasks in the system. Clearly, the actual implementation on a supercomputer would require the online strategy since the workload is submitted by users and not statically known a priori. Nevertheless, we decided to use the offline strategy to perform a set of preliminary experiments since we wanted to test our approaches with reproducible conditions. We also needed our techniques to be stable in order to implement them on a production system and the offline strategy allows us to better verify that.

The proposed methods were implemented using or-tools [Goo], Google's software suite for combinatorial optimization. We performed an evaluation of all our approaches on PBS execution traces collected from Eurora in a timespan of several months. From the whole set of traces we extracted different batches of jobs submitted at various times and we used them to generate several instances of different size, i.e. the number of jobs per instance (a couple of hundreds of instances for each size). Since in this experimental evaluation we were concerned only in the offline approach the real enter queue times were disregarded and in our instances we assume that all jobs enter the system at the same time. The main performance metric considered is the time spent by the jobs in the queues while waiting their execution to begin (ideally as low as possible).

5.2.4.1 Evaluation of Our Models

We decided to make our experiments using two artificial versions of the Eurora machine: A) a machine composed by 8 nodes and B) a machine with 32 nodes.

In addition to the real traces (set *base*) we generated two additional sets of instances: one composed by especially computationally intensive jobs, in terms of resource requested (*highLoad*), and one formed by jobs having an unusually high number of job-units (*manyUnits*). These additional groups were generated using selected subsets of the original traces. From these three sets we randomly extracted subsets of smaller instances with dimension of 25, 40, 50, 60, 80, 100, 150 and 200 jobs; for each size we used 50 different instances in our tests. The instances of dimension 25 and 50 run on the smaller machine A while the remaining instances executed on machine B.

For each instance we ran the PRB algorithm (*PRB*), the hybrid approach with no feedback iteration (*DEC_noFeedBack*) and the hybrid approach with feedback (*DEC_feedBack*). We tested the hybrid approach both with and without the interaction between the two layers because the method without feedback is much faster than the one with the interaction, therefore better suited to a real-time application as a HPC dispatcher. We then wanted to understand the trade-off between time required to reach a solution and its quality.

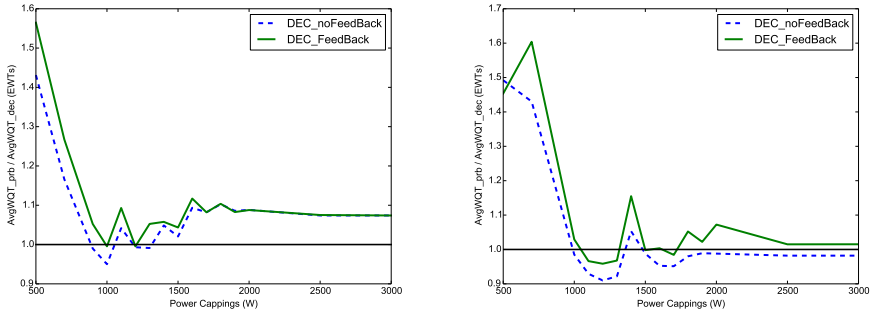
The CP component of the decomposed method has a timeout which forces the termination of the search phase. The timeout is set to 5 seconds; if the solver finds no solution within the time limit, the search is restarted with an increased timeout (multiplying by 2 the previous timeout), until we reach a maximum value of 60 seconds - which is actually never reached in our experiments. The *PRB* is extremely fast and it manages to find a solution in fractions of second even with the larger instances. *DEC_noFeedBack* requires up to 3-4 seconds with the larger instances (but usually a lower quality solution is found in less than a second) and the *DEC_FeedBack* requires significant larger times to compute a solution due to the multiple iterations, in particular up to 15-20 seconds with the instances of 200 jobs.

Each experiment was repeated with different values of power capping to explore how the bound on the power influences the behaviour of the dispatcher. At every run we measured the average weighted queue time of the solution, that is the average time spent waiting by the jobs, weighted with the expected wait time (given by the queue of the job). As measuring unit we use the *Expected Wait Time*, (EWT), i.e. the ratio between the real wait time and the expected one. An EWT value of 1 tells us that a job has waited exactly as long as expected; values smaller than 1 indicate that the job started before the expected start time and values larger than 1 that the job started later than expected.

To evaluate the performance of the different approaches we then compute the ratio between the average weight queue time obtained by *PRB* and by the two hybrid methods; finally we plot these ratios in the figures presented in the rest of the section. Since the best average queue times are the lowest ones, it is clear that if the value of the ratio goes below one the *PRB* approach is performing better, while when the value is above one then the hybrid approaches are obtaining better results. The following figures show the ratios in the y-axis while the x-axis specify the power capping level; we only show significant power capping levels, i.e. the one large enough to allow a solution to the problem, hence the x-scale range may vary.

Machine with 8 nodes Figures 5.2, 5.3 and 5.4 show the results of the experiments with the 8-nodes machine; each figure corresponds respectively to

the *base* workload set of instances (both 25 and 50 jobs sizes), the *highLoad* case and finally the *manyUnits* case. The solid lines represent the ratios between the average queue times obtained by *PRB* and those obtained by *DEC_feedBack*; conversely, the dashed lines are the ratios between *PRB* and *DEC_noFeedBack*. As we can see in Figure 5.2 with an average workload the hybrid approaches usually outperform the heuristic algorithm, markedly in the 25 jobs case and especially with tighter power budgets. With looser power cap levels usually the hybrid approaches and *PRB* offer quite similar performance; this is reasonable since when the power constraints are more relaxed the allocation and scheduling decisions are more straightforward and the more advanced reasoning offered by CP is less necessary.



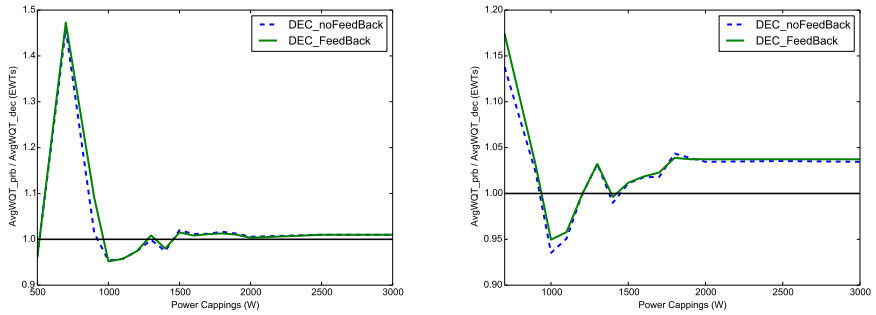
(a) 25 Jobs (b) 50 Jobs

Figure 5.2: 8 Node - *Base* Set

It is easy also to see that the hybrid method with the feedback mechanism always outperforms the feedback-less as we expected: the feedback-less solution has always inferior quality w.r.t. to those generated by the method with the interaction - the first solution produced by *DEC_feedBack* is the same produced by *DEC_noFeedBack*. Our focus should be on the extent of the improvement guaranteed by the feedback mechanism in relation to the longer time required to reach a solution. For example, Fig.5.2 (corresponding to the original Eurora workloads) shows that the feedback method offers clear advantages, in particular with tighter power constraints (around 10%-15% gain over the feedback-less one), a fact that could justify its use despite the longer times required to reach a solution. Conversely, Figure 5.3 reveals that the two hybrid approaches offer very similar performance if the workload is very intensive, leading us to prefer the faster *DEC_noFeedBack* in these circumstances.

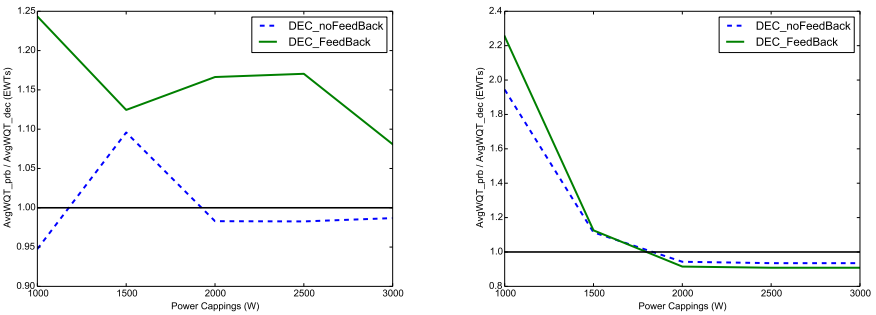
If we consider the workload characterized by an unusually high amount of job-units per job we see slightly different results. As displayed in Figure 5.4, *DEC_feedBack* definitely outperforms the other two methods with tight power cap values, especially in the case of instances of 25 jobs. When the power constraints get more relaxed the three approaches offer almost the same results.

Machine with 32 nodes In Figures 5.5 and 5.6 we can see the results of some of the experiments done on the machine with 32 nodes (in particular there are the cases of size 40 and 100).



(a) 25 Jobs (b) 50 Jobs

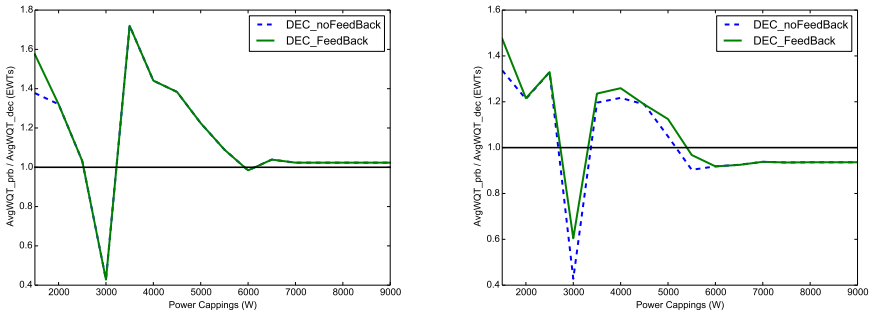
Figure 5.3: 8 Node - *HighLoad* Set



(a) 25 Jobs (b) 50 Jobs

Figure 5.4: 8 Node - *ManyUnits* Set

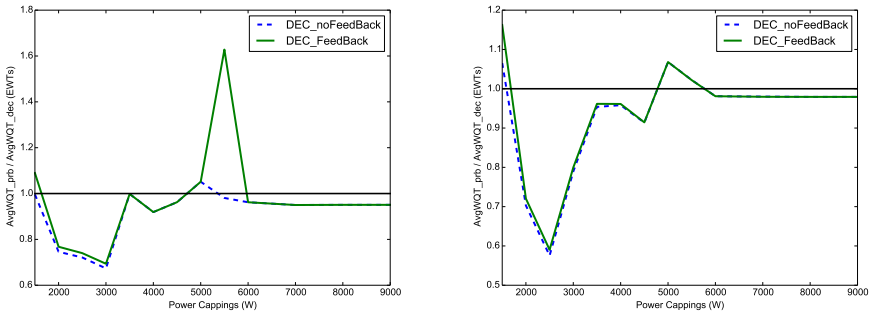
Figure 5.5 shows again the comparison between the two hybrid approaches and the heuristic technique in the case of average workload. The pattern here is slightly different from the 8-nodes case: after an initial phase where the hybrid methods perform better, *PRB* offers better results for intermediate levels of power capping (around 3000W); after that we can see a new gain offered by the hybrid techniques until we reach a power cap level around 6000W (the power constraint relaxes), where the three approaches provide more or less the same results. It is evident again that the method with feedback outperforms the feedback-less one, especially with larger instances.



(a) 40 Jobs (b) 100 Jobs

Figure 5.5: 32 Node - *Base* Set

In Figure 5.6 we can see the results obtained with the computationally more intensive workload. In this case *PRB* performs generally better at lower power cap levels until the power constraint become less tight and the three methods produce similar outcomes again.



(a) 40 Jobs (b) 100 Jobs

Figure 5.6: 32 Node - *Highload* Set

To summarize, with the 32-nodes machine with average workloads the hybrid approaches perform definitely better than the heuristic technique for most power capping levels except a small range of intermediate values (up to a 60% reduction of average queue times). On the contrary, if we force very intensive workloads we obtain better outcomes with *PRB*, even though with usually smaller, but still significant, differences (around 10%). We can also see that with these intensive workloads the difference between the two hybrid approaches is minimal and this suggest that the basic feedback mechanism needs to be refined.

After this first batch of experiments conducted on a reduced version of the machine and with the simplifying assumption of the offline strategy, we can conclude that our novel approaches can find good quality solutions for the HPC job dispatching problem with power cap. An important point that needs to be emphasized is that both proposed approaches can find a solution respecting

tight real-time constraint (i.e. could be implemented and employed on a real supercomputer). In this section we limited the evaluation to our techniques and therefore the natural following step is to ask ourselves how our method performs compared to other power capping policies found in the HPC literature and state-of-the-art. The answer to this question is the aim of next section (Sec. 5.3).

5.3 Comparison with State-of-Art

In this section we are going to compare the performance of the the job dispatcher with power capping proposed in Section 5.2 with other state-of-the-art methods. In order to make such a comparison we created a simulation framework to implement all dispatchers. The simulator takes as an input an instance composed by a set of job requests (user name, job id, resource requested, etc.) and then the chosen dispatcher takes the scheduling and allocation decisions. Applications fall in one of three possible categories: CPU-bound, memory-bound and mixed (average workload slightly skewed toward CPU usage). To increase the realisms of the simulation we drop the two simplifying assumptions made in the previous section: 1) we consider the full version of Eurora with its 64 nodes; 2) we use the online mode.

The job instances used are both historical job traces which ran on the Eurora system and synthetic benchmarks created following the said traces. In the case of the synthetic benchmarks, the features of each job (its duration, requested resources, etc.) are drawn from random distributions defined by parameters learned from past workloads. Each job in an instance has an arrival time, i.e. the moment when it enters the system; the arrival times are distributed within a time window. The arrival times distribution can follow three different models: 1) uniform distribution; 2) left-skewed distribution (a larger fraction of the jobs starts close to the window origin); 3) grouped distribution (also referred as *burst*), where jobs arrive in groups. For example a job instance might be composed by 400 jobs that enter in the system in 15 minutes.

5.3.1 Scalability-oriented Modifications

In order to obtain a more realistic comparison of the different methods for power capping we wanted to increase the size of the experiments with respect to the preliminary study described in Section 5.2.4. In particular we wanted to perform tests on the whole Eurora machine (up to 64 nodes) and we wanted to increase the dimension of the job instance size (up to 1000 jobs). We then made some changes to our methods to improve their scalability.

The first change regards the heuristics approach (the PRB algorithm) – and consequently it applies also to the heuristic algorithm used during the allocation phase in the hybrid method. The complexity of the algorithm (see Algorithm 1) is strongly related to the number of jobs and nodes; in lines 24 and 25 the algorithm reorders both nodes and jobs according to the rules previously defined. These two sorting actions can be discarded. First, we assume that the nodes remain constant or at least they will not change during the external loop that iterates among all jobs to schedule; in case the nodes actually change the best way to reorder them would be an asynchronous function external to the dis-

patching loop. Secondly, the jobs order depends on their characteristics and on the time they spent in the waiting queue - the latter factor is needed to avoid starvation (jobs that wait longer get higher priority). The time spent in queue can only change during different invocations of the dispatcher and therefore also jobs reordering can be discarded (at least inside the loop that starts at line 7).

We also modified the scheduling problem. Empirical observations made us understand that reasoning on the whole set of jobs to be scheduled was not really useful due to the finite number of nodes. In our dispatcher a feasible solution found at time t assigns a start time to each job but only those that have a start time equal to t are actually put in execution. Therefore a lot of time is spent searching for start times of jobs that will not really execute at the current scheduling event; this is especially true if the number of jobs is much larger than the number of available nodes and resources. The number of postponed jobs increases proportionally to the number of jobs that need to be scheduled. Our solution is to bound the number of jobs to be scheduled: instead of considering the whole waiting queue we limit the number of variables in the CP model for the scheduling phase. This is a trade-off between faster solution times and less optimal solutions – we are losing part of the proactive strength of our approach. In our case, preliminary experiments showed that limiting the number of jobs to be considered at 100 significantly decreases the search time without impacting the quality of the solution.

We also changed the custom search strategy to find a solution for the scheduling problem. Since we have very stringent time limits (due to the real-time requirements) larger instances could prevent the solver to find any solution in the allowed time; this is extremely bad because a real dispatcher must be able to guarantee a solution in any case – a poor quality solution is still better than no solution. We therefore need a way to find at least one solution in any case, a fail-proof mechanism. Hence, the new search strategy starts from the solution of the heuristic approach and then tries to improve it during the remaining search time. After the first solution is found, new solutions are searched through Schedule or Postpone strategy. The allowed search time is initially one second; if no solution is found within the time limit we start doubling it and continue until a solution is found. The maximal time limit that we allow in this way is one minute; however, in all our experiments (even with the largest instances) the time required to find a solution for the scheduling phase is always smaller than 4 seconds.

5.3.2 Experimental Setup

We selected a sub-set of the methods from the State-of-Art for the comparison with our approach (see Section 2.1.3 for the related literature). First we excluded all methods based on moldable or malleable jobs since we consider only rigid ones[4]. We also disregarded methods relying on system-specific features, i.e. node variability. Finally, our dispatcher deals with a non-trivial problem (multi-resource system, jobs composed by sub-units, etc) and therefore we did not considered approaches not applicable to such problem. For example we implemented [ECLV10a] instead of a more recent technique from the same authors [ECLV12a] because the latter would have required to develop an entirely

[4]Most current HPC systems employ rigid jobs

different ILP model than the one presented in their paper.

The following are the chosen methods. Our two approaches: I) the heuristic algorithm *LS* (Section 5.2.2) and II) the hybrid approach *DEC* (Section 5.2.3). III) The power-aware EASY-backfilling extension described in Section 2.1.3.9 – referred to as *BF*. IV) A technique employing frequency scaling based on [ECLV10a] – referred to as *DVFS*. Two methods relying on RAPL: V) *Simple* power-aware scheduler presented in [BSRH14] and VI) a dynamic power-sharing methods discussed in [EMRS15] - referred to as *DynShare*.

5.3.2.1 Impact of the power reduction/frequency scaling

A key aspect of some implemented models is the possibility to reduce the power consumption in a node via RAPL or frequency scaling (*DVFS*, *Simple*, *DynShare*). Since we are not considering application-level optimization, such as reducing the frequency only for memory-intensive task or tasks reordering, it is safe to assume that, in general, the duration of a "slowed down" job will increase. The amount of the change is a non-trivial issue: it depends on the hardware implementation (RAPL is a proprietary solution), the nature of the application, etc. In our work we tested different power-duration relation scenarios.

Let assume that we decrease the power on a certain node by a certain amount, for example we go from P to $P' = M^P P, 0 < M^P \leq 1$. The ideal case is that the duration is not affected: $D' = D$ - this is a limit case and can be used to compare a method against an idealized situation that favours to the extreme techniques that reduce power by changing operating frequency. The opposite case defines the duration increase as directly proportional to the power decrease: $D' = M^D D, M^D = 1/M^P, M^D \geq 1$. Between the best-case and this worst-case scenario (from the point of view of RAPL based solutions) we can have intermediate cases, where the duration increase is modulated by a factor F: $D' = FM^D D, M^D = 1/M^P, M^D \geq 1, 0 < F \leq 1$. This general formula can also include the first two extreme cases. In our experiments we tried the following factors: $0.25, 0.5, 0.75$.

In the scenarios discussed so far we applied indiscriminately the same power-decrease/duration-increase model to all jobs. We also implemented a *mixed* model where each job has its own factor (given that we apply the more general formula). The factor of each job is drawn from one among three random distributions, one for each application type - memory-bound jobs will obtain lower factors (smaller duration increase) than the CPU intensive ones.

5.3.2.2 Evaluation Metric

To compare the different techniques we chose the so-called *Bounded Slowdown* (BSLD), a metric commonly used in the literature [HFA05,ECLV10b,ECLV10a, ECLV12a] and defined by the ratio between the time spent waiting in the system and the job runtime.

$$BSLD = max(\frac{wait\_time + run\_time'}{max(\theta, run\_time)}, 1) \tag{5.3}$$

where $wait\_time$ is the time spent waiting for execution, run_time is the "original" duration (i.e. the duration specified when the job is submitted), run_time'

is the final duration (possibly changed due to power reduction or frequency scaling), θ is a threshold used to avoid the bias of very short jobs on the average value. In all our experiments we set a threshold of five minutes. This metric takes into account both the slowdown introduced by the dispatchers that enforce power capping by postponing jobs and the slowdown caused by the performance penalty due to frequency scaling. The BSLD assumes values ≥ 1 (with 1 being the optimal value); lower values indicate better performance.

5.3.3 Results

For every job instance we performed several experiments: first we run each dispatcher without power cap, to establish the uncapped maximal power consumption. Then we run again the dispatchers on the same instance imposing a power cap with decreasing power budget values (expressed as percentage of the uncapped power consumption). We run experiments on instances of varying size and arrivals window width, ranging from 50 jobs in five minutes to 1000 jobs in half an hour; different initial conditions were also tested, from an initially empty system to a machine where 70% of the nodes are already fully occupied. For every combination of number of jobs/time window/starting condition we ran experiments on at least 20 different job instances and gathered the corresponding evaluation metric for each run. The results displayed in the following plots are the average values of the collected statistics.

The first thing we noticed is the very poor performance of the *Simple* dispatcher; in any possible condition (low or high power budget) and any instance size this method provides schedules that are an order of magnitude worse (in terms of average BSLD) compared to the other ones. This is due to the lack of a power sharing mechanism among nodes and this leads to extremely unbalanced situations that penalize jobs that did not enter in the system as first. We decided to exclude this method from the following plots because it would have not added any significant information.

A second point we are not going to discuss further concerns the performance of our methods *LS* and *DEC*. As it was expected due to their implementation *DEC* always provides better - or equal - results than *LS*, due to the fact that they share the first solution and only *DEC* runs an additional search to improve it. When the problem grows in size *DEC* might not be able to find improving solutions (due to the tight time limit we impose) and the distance between our methods narrows.

Figure 5.7 portrays the result obtained with instances of 50 jobs arriving in 5 minutes, uniform distribution of arrival times. In Figure 5.7a the machine was empty at the initial state, hot start (HS) = 0%. The power budget ranges from the maximum (100%) till a value of 10% of the maximum - we want to stress out that values lower than 40% of the unconstrained power are extremely small and quite rare in real systems. Each column corresponds to the average BSLD obtained by a dispatching method. We show our methods (*LS* & *DEC*), backfilling with power cap *BF*, frequency scaling *DVFS* and several scenarios for the RAPL methods *DynShare*; the different scenarios correspond to the different power/duration models (see Sec. 5.3.2.1). The number at the end of the name indicates the factor: 0 and 1 for the extreme cases (no duration increase and proportional increase); 0.25, 0.5 and 0.75 are the intermediate values; *mixed* denotes the scenario where each application has its own factor.

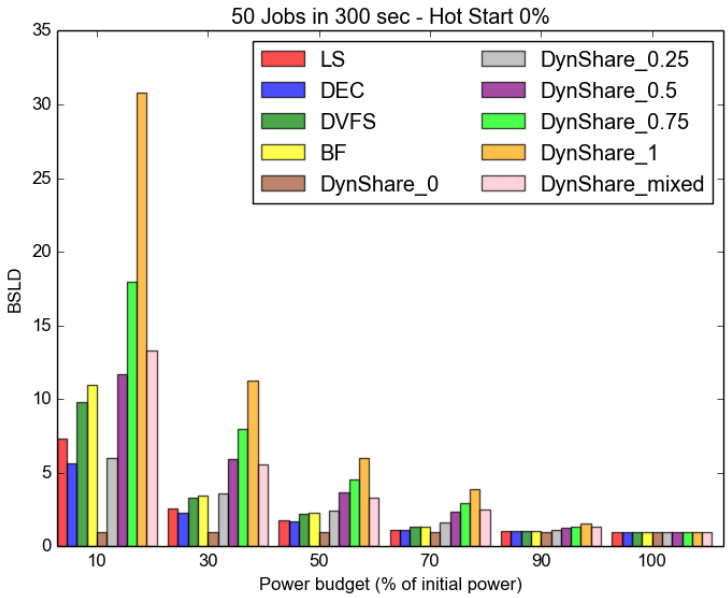
With such small instances the machine resources are almost never fully occupied (50 jobs for 64 nodes) and therefore the only obstacle preventing jobs to run in the system is the power constraint; this can be clearly seen noticing the small BSLD values at higher power budget - only slightly larger than one. It is clear that our methods offer very good performance: both *LS & DEC* perform better than all remaining methods except the *DynShare_0* at power budgets between 30% and 70%. As mentioned before *DynShare_0* represents an optimal, unrealistic lower bound (power decrease does not affect job duration and BSLD metric) and obviously performs better than all other approaches - especially when the power is the main issue. The average BSLD of the other *DynShare* techniques worsens rapidly with the increase of the power-duration factor (from 0.25 up to 1); *DynShare_mixed* has a performance close to a factor of about 0.5-0.6. As expected *DVFS* offers better results than *BF* (both better than RAPL with such small instances and empty machine). In our tests the improvement of *DVFS* w.r.t. *BF* is smaller than the one presented in [ECLV10a] and this happens due to the more complex problem tackled (job and job-units versus jobs only, multi-resource machine versus single resource, nodes of different types).

5.3.3.1 Initial State Impact

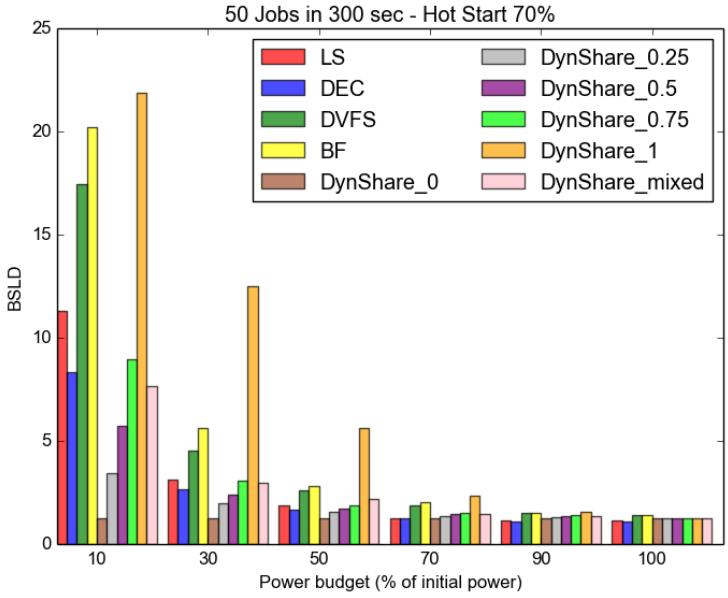
The situation changes when we consider a non empty machine; in Fig. 5.7b we can observe the results if the 70% of the nodes are occupied before the jobs arrival. Now the methods using RAPL perform much better (w.r.t. the other ones), only the case with duration increase proportional to the power decrease maintains its bad performance; *DVFS* and *BF* provide much worse results. *LS & DEC* manage to remain on par with all RAPL methods only as long as the power budget does not get too low (smaller than 30%); at a (quite unrealistic) power budget of 10% only *DEC* obtains an average BSLD close to *DynShare_mixed* and *DynShare_0.75*. If we look at the numbers we see that the differences in relative performance are due to the better results obtained by *DynShare*, while the other methods performance is worse than the empty machine case. We would expect to observe a generalized performance deterioration: in an occupied machine jobs are forced to wait until more resources become available.

To understand why this is happening we have to distinguish between two different kinds of dispatchers. RAPL-based methods are *dynamic* dispatchers, they let all jobs enter the system and then dynamically adjust their power consumptions - no power-check is performed before admitting jobs. The other methods can be seen as *static*: they decide before dispatching which jobs can be executed and their strength is based on the quality of the generated schedule. When the machine is already occupied static approaches suffer from the lack of optimization possibilities, thus reducing their effectiveness[5]. Conversely, one of the weakest point of reactive methods is the lack of a power-based admission control that may lead to too many jobs in the system and therefore widespread slowdown generated by mandatory power reductions. With a partially occupied machine fewer jobs can enter the system due to the fewer available resources (cores, GPUs, etc.), large power reductions happen less frequently and as a result the average BSLD improves.

[5] *LS & DEC* can also be seen as *proactive* dispatchers: they take into account *all* jobs which need to be executed when creating an optimal schedule



(a) Hot Start = 0%



(b) Hot Start = 70%

Figure 5.7: Average BSLD; 50 jobs 300s; HS=0%; uniform distribution

5.3.3.2 Instance Size Impact

In Figure 5.8 we can observe the behaviour of the different dispatchers when the instance size increases, in particular up to 200 jobs in 300 seconds - in this case the jobs arrive in groups (burst arrival). We restrict the analysis to more

realistic power budget (namely we discard the 10% case). The first thing worth to be noted is the fact that the different dispatchers obtain different results even at maximum power budget. The reason is that when the instance size increases the problem becomes harder in terms of resource availability, therefore an approach able to generate better schedules leads to better performance. In other words, even with no power constraint (power budget equal to 100%) some jobs must wait for free resource and our more sophisticated methods (*LS* & *DEC*) manage to produce lower queue times than the simple EASY-BF - hence lower BSLD. This is an extremely important point because it allows our method to outperform even *DynShare_0*, because this method (like all RAPL-based ones) focuses only in minimizing the runtime-increase penalty (see Equation 5.3).

With higher power budget (75%-100%) both *LS* and *DEC* outperform all remaining methods; when the power constraint gets tighter the RAPL begins to be effective and the gap with our methods is reduced - but at 50% it is still the best approach. When the power budget gets even lower (30%) the quality of the generated schedule becomes relatively less important and RAPL-based approaches take the lead - though *DEC* performance is still roughly equal to *DynShare_075* and better than *DynShare_mixed*. Both *DVFS* and *BF* generally provide worse results than the other methods.

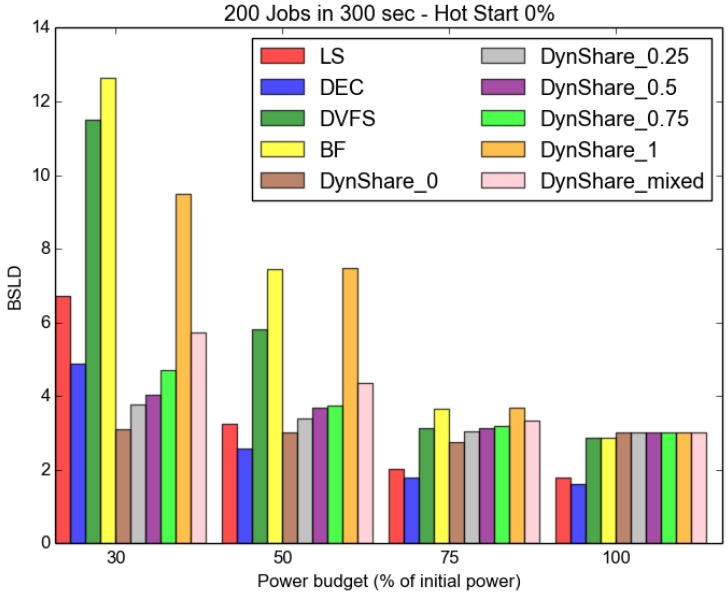


Figure 5.8: Average BSLD; 200 jobs in 300s; HS=0%; burst arrival

When we increase again the size of the instance, with a number of job ≥ 400, we notice a clear limit of the RAPL-based methods in their simpler implementations. We can see this in Figure 5.9. Figure 5.9a displays the average BLSD for instances of 400 jobs in 300 seconds, burst arrival mode, empty initial state of the system. We see that for smaller power budgets ($\leq 30\%$) the RAPL-based methods are not able to find solutions: it is impossible to satisfy the power constraint just by applying the RAPL-mechanism. This happens for two reasons: 1) there is no power-aware job admission control - the basic EASY-BF

scheduler always let jobs enter the system; 2) we assumed that there is a lower bound on the minimal power consumption of a job - we cannot arbitrarily decrease the job power below a certain threshold (1 Watt in our experiments). These two conditions (all jobs in an instance can enter the systems - provided enough resources are available - and a non-zero minimal power consumption) prevent pure reactive techniques such as the *DynShare* methods from keeping the system power consumption within the desired budget. This is not a problem for the remaining approaches due to the capability to limit *a priori* the number of jobs entering the system.

As we identified, the problem lies in the basic implementation of the RAPL-based methods, lacking a power-aware admission control mechanism. Even though this is not discussed in the literature, we assume that real supercomputers employ some form of admission control and therefore we modified the *DynShare* methods to add such a mechanism. We therefore substituted the EASY-BF scheduler of the basic implementation with the power-aware version; in practice we combined a first scheduling stage that employs *BF* and a second stage where the power is dynamically managed with RAPL. Directly assigning the desired power budget to the *BF* algorithm makes no sense because the power cap would be already enforced and the RAPL-mechanism would never be triggered. Therefore the power budget considered in the first stage is three times the target power budget[6]; the remaining power slack is covered by RAPL action.

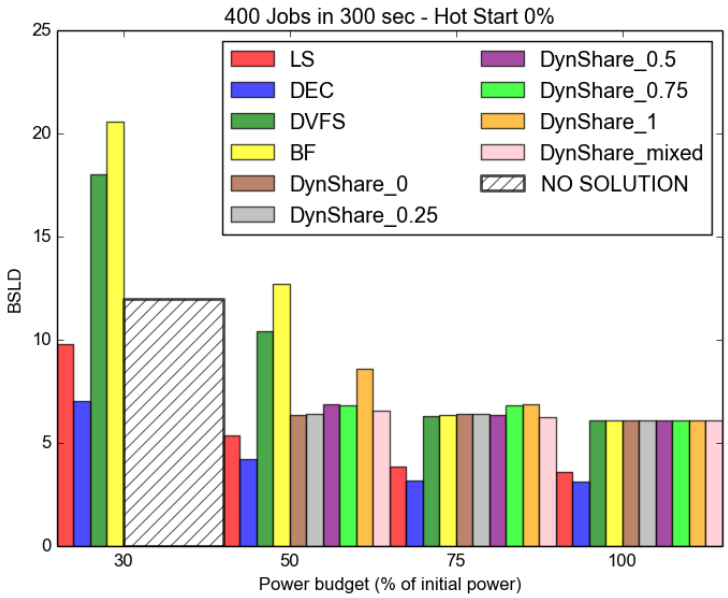
In Fig. 5.9b we can observe the results obtained after the change. Now even RAPL-based approaches provide solutions at lower power budgets. As we saw for instances composed by 200 jobs, *LS* & *DEC* clearly outperform the remaining methods when the problem is less power constrained and good scheduling decisions have a greater impact. With the tightening of the power constraint (power budget $\leq 30\%$), the possibility to decrease power consumption at run times starts to reap its benefits; nevertheless *DEC* still trails very closely *DynShare_0* and provides lower average BSLD than all other methods.

The results obtained with larger instances (up to 1000 jobs, window size ranging between 15 and 30 minutes) show a identical behaviour. The RAPL-based methods cannot find solutions without a previous power-guided admission control. When we add the admission control, *LS* & *DEC* perform equally or slightly worse (higher BSLD) than *DynShare* with lower power-performance factor (0 and 0.25). This happens only for lower budgets; again, with budgets ranging from 50% to 100% our methods clearly outperform all remaining ones (decisive advantage when the problem is less power constrained).

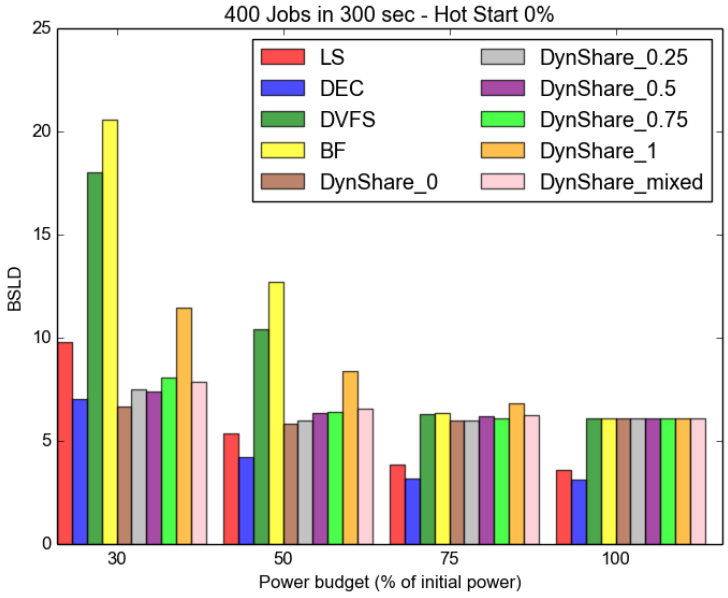
5.3.3.3 Job Arrivals Mode Impact

In Figures 5.10, 5.11 and 5.12 we compare the behaviour of the different approaches when the job arrival mode changes; in all cases we consider instances of 100 jobs in 300 seconds. In Fig. 5.10 jobs arrivals are uniformly distributed in the time window, in Fig. 5.11 the jobs are heavily concentrated towards the start of the time frame (70% of them arrive within the first 30 seconds) and Fig. 5.12 presents the *burst* case, where jobs enter the system in groups (groups sizes range between 5 and 20, groups arrival times are uniformly distributed).

[6] "Three times" is an empirically computed value



(a) No admission control



(b) Power-based admission control

Figure 5.9: Average BSLD; 400 jobs in 300s; HS=0%; burst arrival. Testing the importance of a power-aware job admission control for *DynShare* methods

We restricted the plot to power budgets between 50% and 100%. The results reveal that all dispatchers are quite indifferent to the arrival times mode, with similar average BSLD in all cases. *LS* & *DEC* manage to obtain slightly lower BSLD values if the arrival times are compressed at the start of the time inter-

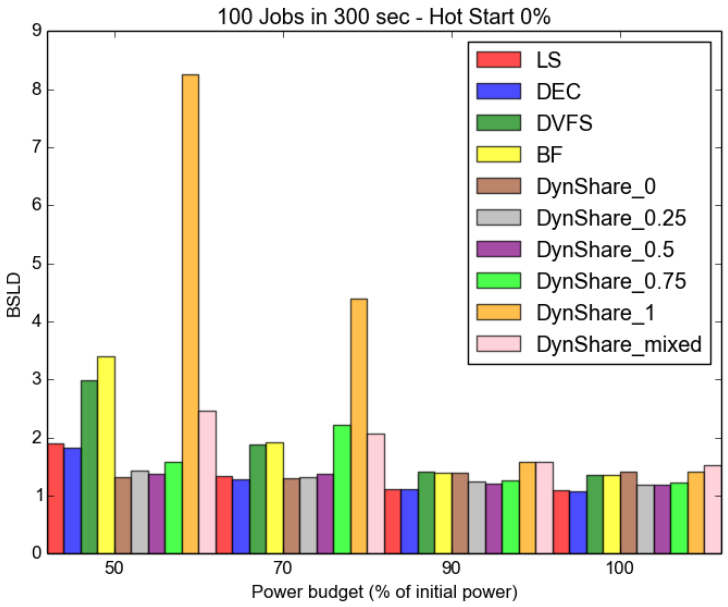


Figure 5.10: Uniform distribution

val (Fig. 5.11), because these proactive methods perform better when reasoning with larger number of jobs.

5.3.3.4 Historical Traces

We also performed experiments using real historical traces of jobs that run on Eurora. The experiment setup is the same we used for the synthetic benchmark, with the only difference that in this case the arrival window and mode are defined by the trace itself.

On the historical traces we tested all previous dispatchers plus an additional type of RAPL-based approach; in particular we added a new scenario for the relationship between power decrease and duration change (see Section 5.3.2.1 for the other scenarios). In this case we link the duration increase to the nature of the application, defined by its Clock-Per-Instructions (CPI) value. Low values of CPI (≤ 1) indicate CPU-bound applications while higher values (≥ 5) are related to memory-bound applications; intermediate values suggest less unbalanced applications. The CPI values of the jobs in the historical traces have been measured by Eurora monitoring infrastructure.

Given a value of clock-per-instructions CPI the new duration D' change is computed by this formula:

$$D' = D(1 - \frac{1}{CPI} + \frac{1}{CPI * M^P}) \tag{5.4}$$

where M^P is the power multiplier (the ratio between the new desired value P' and the original power P), D is the original duration.

The new approach for historical traces (called *DynShare_CPI*) shares the same algorithm with the other RAPL-based methods (*DynShare*) and uses the

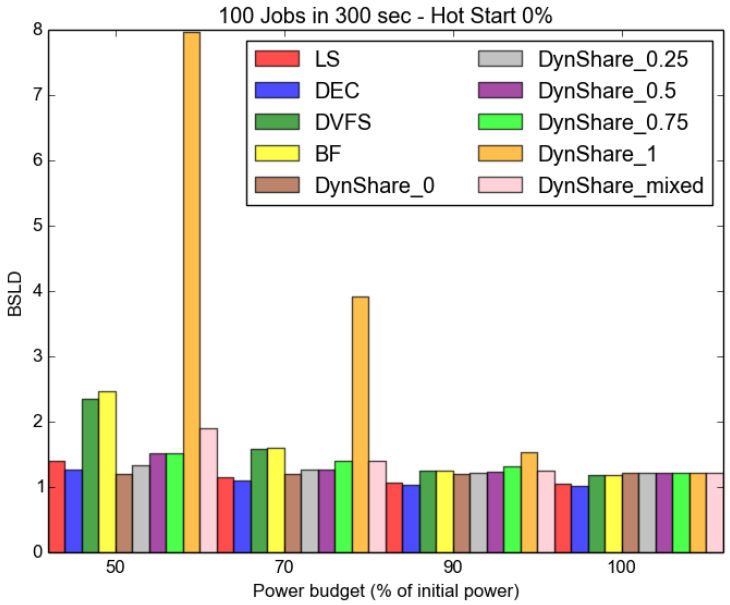


Figure 5.11: Left-skewed distribution

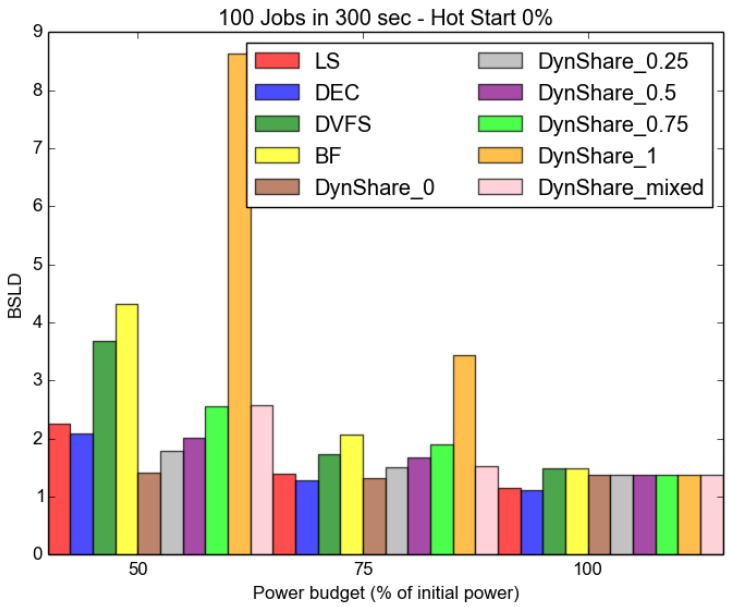


Figure 5.12: Burst arrival

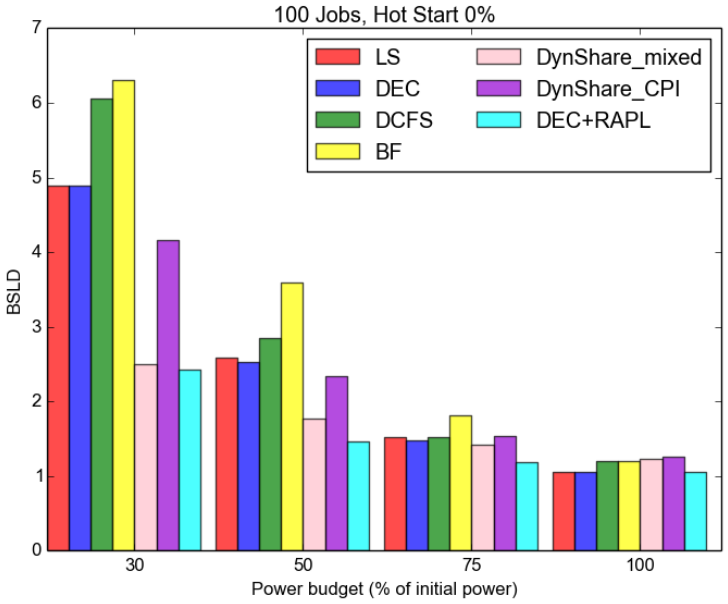


Figure 5.13: Average BSLD; 100 jobs from historical traces

factor defined in Eq. 5.4 to compute the job duration change given a required power modification. With historical traces we run experiments using the following approaches: *LS*, *DEC*, *BF*, *DVFS*, *DynShare_mixed* and *DynShare_CPI*. Figure 5.13 portrays the average BSLD computed on 80 historical traces with a job size equal to 100; Figure 5.14 considers the case of 200 jobs and Figure 5.15 400 jobs. First, we notice that using the CPI-based criterion to compute the duration increase (*DynShare_CPI*) produces worst average BSLD compared to the *DynShare_mixed* case. This is due to the fact that the historical traces are mostly composed by CPU-intensive jobs with low CPI values and therefore the impact of power reduction is heavier. Then, compared to the synthetic benchmarks case we observe a relative decrease of the performance of *LS* & *DEC* w.r.t. *DynShare_mixed* (and other methods able to modulate the power).

The discrepancy of results between historical and synthetic traces needs to be explained. The key difference between the two types of benchmarks is the job arrival frequency, which is much higher in the synthetic benchmarks. For example if we look at the case of instances of 100 jobs, while time windows of synthetic benchmarks range from 5 minutes to 2 hours the average time window of historical traces is more or less 6 hours. These characteristics of the historical workload are probably due to the fact that Eurora has been used as a prototype and it was never heavily loaded in the measurement period, while a machine in production would be much more loaded on average.

Another key difference is that many jobs in the historical traces have extremely short durations - around 15%-20% of jobs last less than 1 minute; in the synthetic benchmarks there are no such short jobs. Drastically decreasing the job arrival frequency changes the difficulty of the dispatching problem because fewer jobs are in the system, fewer jobs are forced to wait due to unavailable resources and using a “smart” scheduling policy loses its benefits. Moreover,

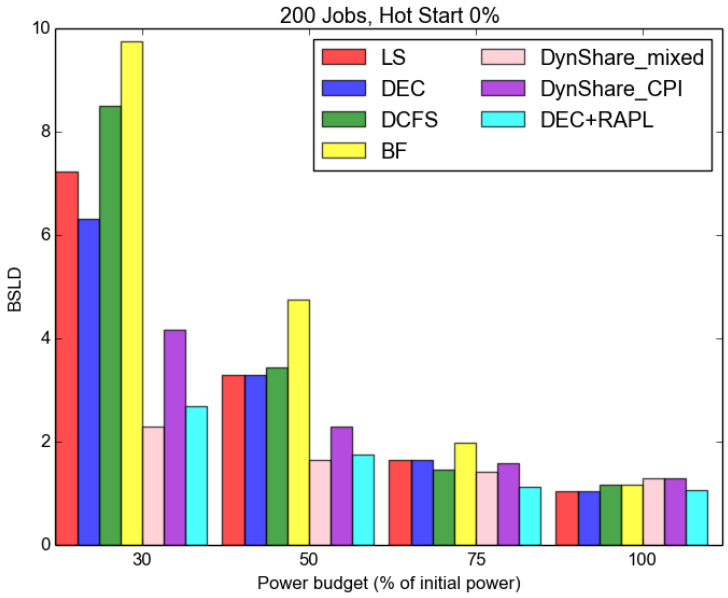


Figure 5.14: Average BSLD; 200 jobs from historical traces

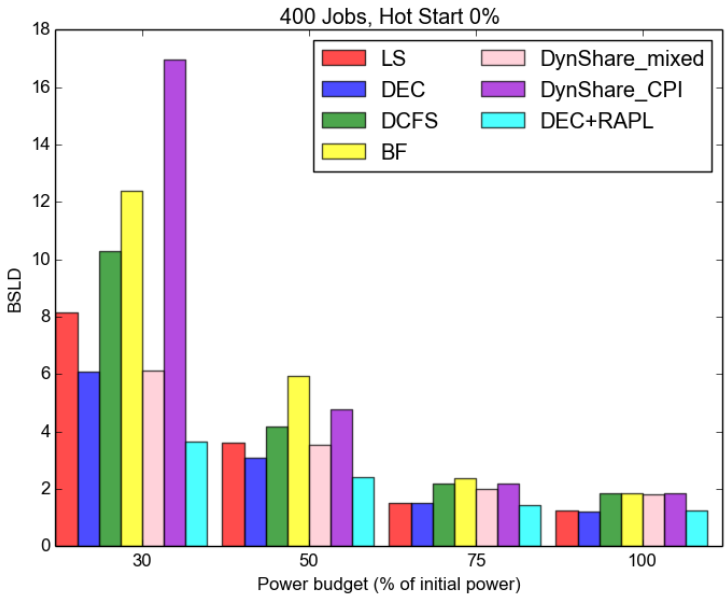


Figure 5.15: Average BSLD; 400 jobs from historical traces

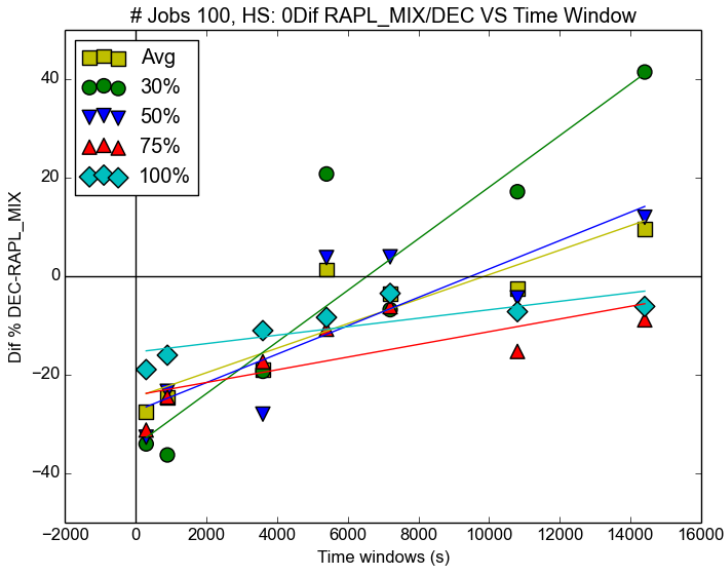


Figure 5.16: Average BSLD; 100 jobs from historical traces

the power aspect acquires greater relevance and the methods able to modulate the power consumption become more favorable.

The correlation between the performance difference of the proactive methods (*LS* & *DEC*) w.r.t. methods using RAPL and the job arrival time window can be easily seen in Figure 5.16, which illustrates the performance difference between *DEC* and *DynShare_mixed* with varying time windows. On the x-axis the job arrival time window is displayed and the y-axis shows the BLSD percentage difference, for each power cap[7], between the considered methods; negative values correspond to the cases when *DEC* outperforms *DynShare_mixed* and vice-versa. The figure presents the results for 4 different power cap values and for the average results computed on all power cap (yellow square markers). The arrival times follow a random uniform distribution. It can be easily observed that if the window size increases (thus the job arrival frequency decreases) *DynShare_mixed* begins to outperform *DEC* if the power cap is smaller than 50%; even with larger power budget the relative performance of *DEC* degrades with the increase in time window size - experiments performed with even larger time windows, not shown in this graph, confirm the trend observed here.

This correlation gave us an insight on how combining the benefits of proactive and power-modulating methods in order to obtain a dispatcher well suited to cope with any type of workload. The main idea is that if the job arrival frequency is high *LS* & *DEC* provide better results when job arrival frequency decreases under a threshold the best method is some variant of *DynShare*. We therefore implemented the new approach *DEC+RAPL* that uses both *DEC* and *DynShare_mixed*, depending on the job arrival frequency; the correct threshold was obtained through empirical evaluation and it is equal to one job every two minutes. The experiments prove that combining the best of both world yields

[7] $\%DIF^{pcap} = 100 \times \frac{BSLD^{pcap}_{DEC} - BSLD^{pcap}_{RAPL\_MIX}}{BSLD^{pcap}_{RAPL\_MIX}}$

the best results. Looking again at Figures 5.13, 5.14 and 5.15 we focus now on the last column on the right which represents *DEC+RAPL*. The new dispatcher obtains better or equal results than the other methods in almost all situation (except the case of 30% power budget and 200 jobs). Even though not shown here, the results of the experiments conducted on the synthetic benchmarks are analogous. For example, with an arrival window of 5 minutes (such as the one of previous figures) *DEC+RAPL* has the same performance of *DEC* because with such a high arrivals frequency the RAPL component is never activated.

5.3.3.5 Mispredictions Impact

As noted in Chapter 4 the power consumption predictions can be inaccurate. This is a problem especially in the case of under-prediction, i.e. the forecast power consumption is smaller than the real one, because *LS* & *DEC* might produce schedules violating the power budget (they enforce a constraint based on the predictions). A possible solution is to impose them a tighter power cap than the target one, so that if under-predictions happen the desired budget would still be respected; however, decreasing the power cap has the downside of performance degradation. We conducted a set of additional experiments to quantify the performance decrease due to handling the under-predictions for our proactive dispatchers, *LS* & *DEC*.

In the new batch of experiments we use the previous jobs instances but we impose that 10% of the job in an instance are under-predicted; for these jobs the predicted power is 40% smaller than the real one. We then run again *LS* & *DEC* (RAPL-based methods use the real power thus do not require additional experiments) using a smaller power budget, i.e. $power\_cap = 0.95 \cdot power\_target$. After a preliminary analysis we discovered that an extremely small power cap decrease is sufficient to ensure that the target power cap is respected; for example, using a power cap 1% smaller than the target cause both *LS* & *DEC* not to violate the budget.

In Figure 5.17 we see the results of the experiments with under-predictions in the case of 100 jobs (synthetic benchmark). We see the main dispatchers shown before plus *LS* & *DEC* using a power cap value 1% smaller than the target, represented by the hatched columns *LS_vU99* & *DEC_vU99*. The target power budget is respected while the BSLD slightly rises. If we take into account all the instances, given a 1% power cap reduction w.r.t. target, the average BSLD increases by 4.2% for *LS* and by 4.4% for *DEC*.

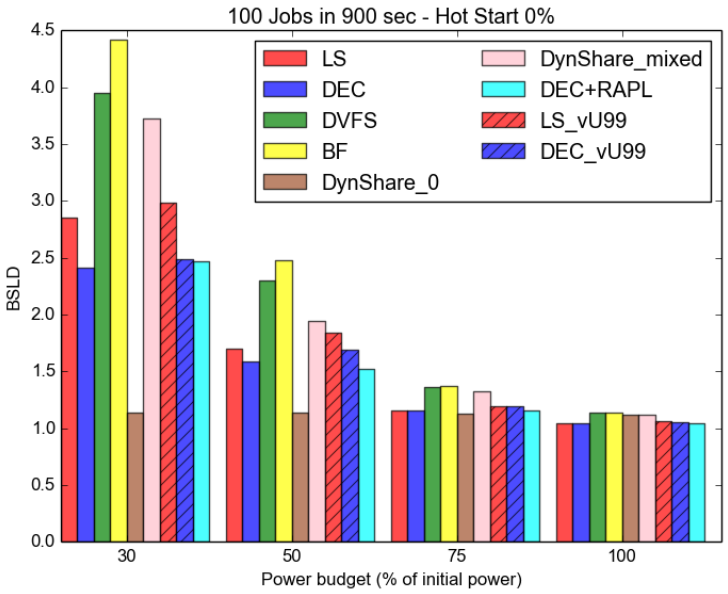


Figure 5.17: Average BSLD; 100 jobs in 900s; HS=0%; uniform distribution

5.4 Case Study: Integration with Cooling System

After having discussed the job dispatcher with power capping and having compared its performance with other techniques from the state-of-the-art, we are going now to see an example of its integration with another component of the supercomputer facilities. We want to see how the proposed method would interact with the cooling system. For example, the cooling infrastructure requires different amount of power in different environmental condition – summarizing, when the external temperature is higher more energy and power are required to cool down the machine. Since the whole power budget available for a power capped supercomputer always includes the power consumed by the cooling infrastructure, changes in the external conditions may have an impact on the power that can be used for the computation. In turn, this can influence the performance granted by the job dispatcher.

To evaluate the benefit of our proposed strategy we model the impact of ambient temperature and power consumption of the Eurora supercomputer and we generate a compact model which returns the needed cooling power given as inputs the IT power and the ambient temperature. Thanks to this model we can calculate the energetic efficiency (PUE) that can be obtained with our dispatcher. We can also use this model in a reverse fashion: given a target PUE and an ambient temperature we can compute the power budget available for the IT infrastructure and therefore deduce a power cap to be given as input to our job dispatcher.

5.4.1 Eurora cooling system

The cooling architecture of the direct liquid-cooled HPC system considered in this work is reported in Fig. 5.18. The coolant (water in our case) is refrigerated by a chiller with free cooling capability, i.e. it can let the liquid to be passively cooled by the outside environment, with no activation of the thermodynamical cycle commonly performed for this purpose. A variable speed pump is then used to push the liquid into the cooling pipes at a given flow rate. In addition, a three-way valve can be regulated to bypass part of the return coolant, mixing it to the chiller outlet flow, before re-entering the supercomputer.

Since the cooling devices are responsible for significant power consumption w.r.t. the overall facility, energy-aware scheduling approaches could clearly benefit by exploiting information about such parts. To this aim, a model of the overall cooling system, predicting its power requirement is needed. Beside the HPC workload, the cooling power consumption depends also on the environmental conditions and the supercomputer thermal state. Roughly speaking, if the ambient temperature is "cold enough", heat generated by heavy workloads can be efficiently removed going in free-cooling mode. The same reasoning applies if the supercomputer IT component temperatures are far from the critical values when the heavy computation begins.

Accurate characterization of all such dependencies is a rather complex task; time-consuming CFD-based techniques, typically adopted at design stage, are clearly unfeasible for integration with online schedulers. Hence, we need some low-complexity, approximated model, still accurate enough to describe the system main thermal and power features but able to quickly provide results to be used during the scheduling phase. In this respect, we use an analytical, lumped parameters thermal model, derived from thermodynamics first principles [YMO07] [Mas05]. The main idea is to describe the HPC machine, the chiller, and the cooling circuit thermal behaviour as the interaction between two main heat exchanger, one (HE_1) between the machine and the cooling circuit, the other at the evaporator side of the chiller (HE_2). Bearing in mind these considerations, the following system thermal dynamics are obtained:

$$\begin{aligned}
\dot{T}_{HPC} &= \tfrac{1}{C_{HPC}}\left(P_{th} - R_{HE1}^{-1}\left(T_{HPC} - \frac{T_{out} + \alpha T_{outch} + (1-\alpha)T_{out}}{2}\right)\right) \\
\dot{T}_{out} &= \tfrac{1}{C_{HE1}}\left(R_{HE1}^{-1}\left(T_{HPC} - \frac{T_{out} + \alpha T_{outch} + (1-\alpha)T_{out}}{2}\right) - \right. \\
&\qquad \left. qc_v\rho(T_{out} - T_{in})\right) \\
\dot{T}_{outch} &= \tfrac{1}{C_{HE2}}\left(qc_v\rho(T_{out} - T_{in}) - R_{HE2}^{-1}\left(\tfrac{T_{out}+T_{outch}}{2} - T_0\right)\right).
\end{aligned} \tag{5.5}$$

where T_{HPC} is the supercomputer IT devices aggregate temperature, T_{out} is the coolant outlet temperature and T_{outch} is the temperature of the liquid exiting the chiller. Parameters C_{HPC}, C_{HE1} and C_{HE2} model the HPC and heat exchange points thermal capacitance, respectively. Similarly, R_{HE1} and R_{HE2} are the heat exchangers thermal resistances. Finally, P_{th} is the thermal power generated by the HPC computation, ρ and c_v are the coolant density and specific heat (at constant volume) respectively, while q is the coolant flow rate and $\alpha \in [0, 1]$ denotes the position of the three-way valve.

In order to complete the model, we need to express the cooling elements power consumption as a function of the variables in (5.5) and the environmental conditions. For what concerns the pump, its power can be expressed as

a super-linear function of the imposed flow rate [SSM+13]; here a quadratic function $P_{Pump} = k_{pump}q^2$ is selected, with k a component specific coefficient. Regarding the chiller, its coefficient of performance (COP), defined as the ratio between the chiller cooling load and its power consumption, is approximated as the *Carnot efficiency* of the refrigerating cycle, i.e. $COP_{ch} = \frac{T_{0Ch}}{T_{amb} - T_{0Ch}}$. Beside neglecting non idealities, the previous expression does not depends on the internal chiller variables, which are typically not available; the dependence on the outside temperature T_{amb} is captured, thus it can be reasonably adopted for representing the efficiency of chilling devices. Finally, the overall cooling power consumption can be expressed as:

$$P_{cool} = k_{pump}q^2 + COP_{ch}^{-1}P_{th}. \tag{5.6}$$

Knowing that the cooling system control knobs q, α, and T_0 (i.e. the chiller working point) are selected to satisfy the hard temperature constraints on both the HPC IT device and the coolant inlet and outlet flows, model 5.5 can be integrated to predict the system thermal evolution, and use the corresponding variables in (5.6) to update the cooling power consumption and then the PUE[8] estimations.

Clearly, if the cooling control variables α, q and T_0 are known by the scheduler, eq. (5.6) can be directly used with no integration of (5.5).

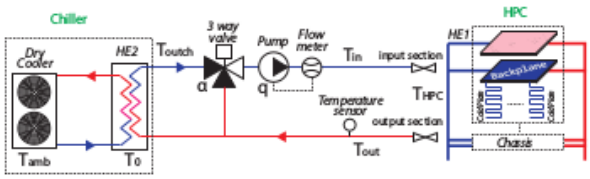


Figure 5.18: HPC cooling system scheme.

5.4.2 Free-cooling Modeling

We used the model described in 5.4.1 to evaluate the impact of the environmental conditions, the workload and the chiller free-cooling capability on the cooling power consumptions for Eurora supercomputer, whose thermal parameters and constraints are reported in Table 5.1. A rather exhaustive set of operating conditions have been tested; the thermal power P_{th} ranged from the idle condition $P_{th} = 5.5$kW to the HPC maximum power $Q_{max} \approx 40kW$; several intermediate ambient temperatures ($T_{amb} = \{0, 5, 10, 15, 20, 25, 30, 35, 40\}$°C) were also taken into account, covering the seasonal average temperature variations. In addition, the cooling management strategy actually implemented in Eurora has been assumed: the liquid flow rate, the valve position and the chiller operation are regulated to keep the inlet water temperature lower than 25°C (but higher than 18° C to avoid condensation), and a bounded temperature gradient $T_{out} - T_{in} < 5$°C.

Figures 5.19a and 5.19b show the obtained cooling power consumption and the corresponding PUE, respectively. It can be clearly noticed the effect of

[8] Defined as $\frac{(P_{cool}+P_{th})}{P_{th}}$

Parameter	Value
R_{HE1}	0.7 [mK/W]
R_{HE2}	0.6 [mK/W]
C_{HE1}	225 [kJ/K]
C_{HE2}	450 [kJ/K]
C_{HPC}	150 [kJ/K]
T_{inMIN}	18 [°C]
T_{inMAX}	25 [°C]
T_{outMIN}	18 [°C]
T_{HPCmax}	85 [°C]
q_{max}	12 [m^3/h]
c_v	4186 [J/(Kg× K)]
ρ	1000 [Kg/m^3]

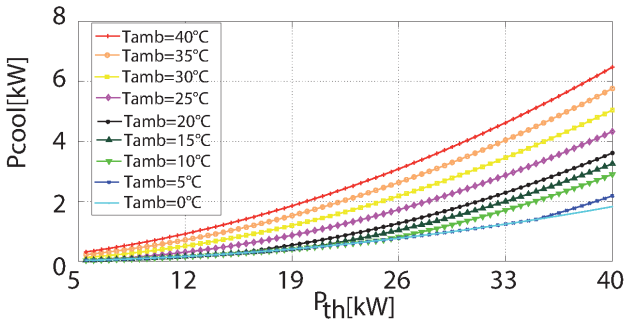
Table 5.1: Eurora Cooling System Parameters

both the environmental conditions and the workload (Fig. 5.19c); the kinks in the plots for T_{amb} $0 - 20$°C denote the point where the free cooling condition is no longer feasible and the chiller thermodynamic cycle is activated – before that points only the pump power is accounted. At $T_{amb} = 0°$ the system is able to operate in free-cooling for all the P_{th} range, while for $T_{amb} > 25°C$ free-cooling is unfeasible due to constraint on $T_{inMAX} < 25°$, thus an almost linearly increasing PUE is obtained as the workload rises. We can further notice that the PUE increases with the workload also for ambient temperatures allowing free-cooling, denoting an increased pump power consumption due to flow rate increase. This is a system-specific feature stemming from Eurora cooling system sizing and the corresponding control strategy.

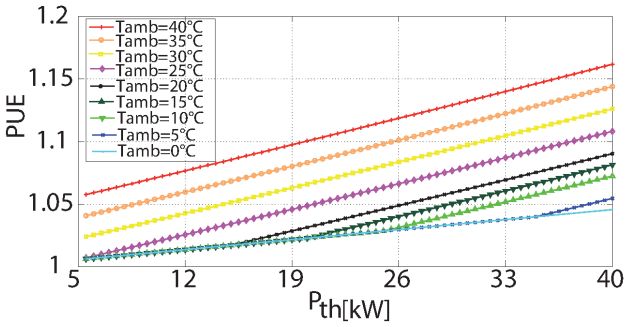
These information can be the input for the power-capped dispatcher discussed in previous sections. The online scheduler could therefore use a varying power budget, based on the current environmental and workload request scenario, with the goal of optimizing the whole supercomputer power consumption, i.e. cooling infrastructure included. In other words, given an "efficiency threshold", namely the desired PUE level not be exceeded, the corresponding P_{th} maximum admissible value can be determined. Thus, the HPC computational power can be scheduled/shaped so that the system will operate within these limits, as much as possible, according to other performance constraints. Fig. 5.19c shows the maximum thermal power (directly related to the power budget) which can be removed by the Eurora cooling mechanism, for different required PUE upper bounds PUE_{lim}, and environmental conditions.

In order to obtain better resolution, the model data have been interpolated over the 2014 annual temperatures in Zola Predosa, Bologna, the closest weather station to Cineca, where Eurora is hosted. As expected, the maximum power budget is indirectly proportional to the ambient temperature (due to the chiller COP) and directly proportional to the PUE limit. However, corners can be noted again. Beside the obvious transition from full power ($40kW$) to linear decreasing to keep the desired PUE, other jumps occur mostly in the range 20-25°C. Again, the free-cooling is responsible for this behaviour. For lower temperatures high efficiency is obtained thanks to such option, then the chiller

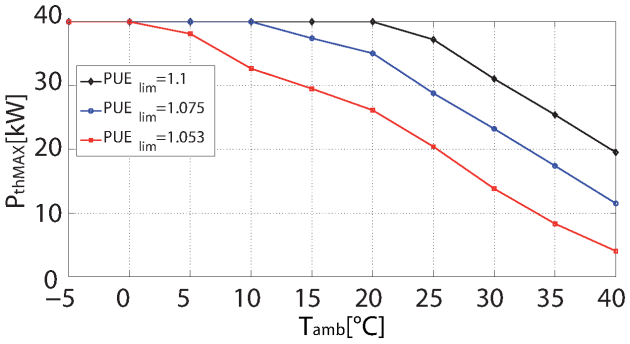
has to be electrically operated to meet thermal constraints, augmenting the cooling consumption and requiring a power shaping if a constant PUE has to be guaranteed.



(a) Power consumption



(b) PUE



(c) Maximum power budget

Figure 5.19: Overall cooling system power consumption (a) and corresponding PUE (b) for different T_{amb} and HPC workloads producing thermal power P_{th}. Maximum power budget as a function of T_a and PUE upper bounds (c).

5.4.3 Experimental Results

In this section we are going to explore the integration of the power capped job dispatcher with the cooling system model presented in Section 5.4.1 and

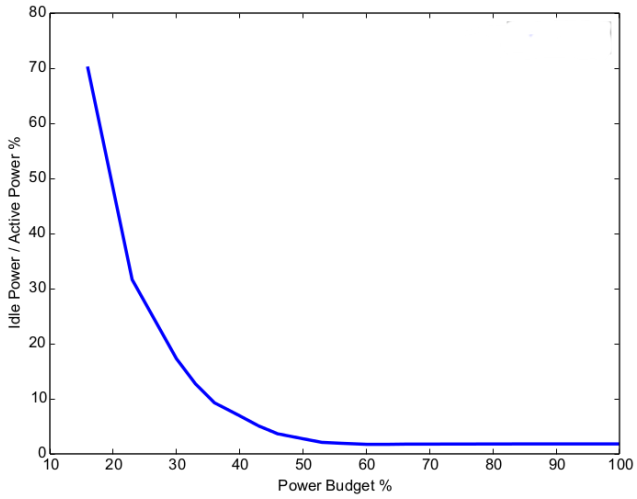


Figure 5.20: Idle Power and Active Power Ratio VS Power Budget (%)

5.4.2. More in particular, we conducted experiments using the hybrid approach (see Section 5.2.3). We are going to see that applying power capping to a supercomputer featuring free-cooling according to the ambient temperature can lead to substantial energy savings. We will also notice that power capping has also the side effect of increasing idle energy cost against active power. Idle energy or idle power is the amount of energy/power consumed by computing resources when no application is using them; sadly, idle power consumption in modern computing units is still greater than zero. Thus, future green supercomputers need to develop integrated power capping and power management mechanisms in order to switch off unused resources when not required.

We evaluate the performance obtained by the proposed dispatcher in case of varying power cap levels. As we have seen in previous sections (Sec. 5.2.4 and Sec. 5.3.3) if we decrease the power budget the overall performance in terms of average queue times decreases. This happens because if we impose tighter power constraints fewer tasks can be executed concurrently, therefore some jobs must be postponed and hence forced to wait longer.

We also have tested how the introduction of a power cap influences the power efficiency of the supercomputer. We used the same experimental setup described in Sec. 5.2.4. Results are shown in Figure 5.20. In the x-axis we see the power capping budget imposed, expressed as a percentage. A power budget of 100% means that the power constraint is more relaxed, while when we decrease the power budget (here down to the 20% of the maximal value) the power capping gets tighter. The maximal power budget was computed as the sum of all machine components maximum power consumptions (Thermal Design Power, TDP). In the y-axis we plot the ratio between the idle and active power consumed by the machine when executing all the scheduled jobs. The figure shows that if we reduce the power budget too much (down to the 50% of the maximal value) the percentage of power spent by idle components of the machine become a very relevant part of the total energy - if we reduce the power

Pue	Budget[KW]	Idle/active	EffectivePue+Idle	WQTloss
1.1	39.14	2.00%	1.100	0.0%
1.075	33.29	2.04%	1.075	0.7%
1.05	29.39	2.62%	1.057	8.6%

Table 5.2: Impact of the proposed power budgeting ambient temperature-aware on one year supercomputer center usage scenario.

budget to 20%, the idle power amounts to the 70% of the total power, i.e. a very inefficient power consumption. This is due to the fact that with smaller power budgets fewer jobs can execute in parallel, then more systems components are not used; since the unused components still consume some energy the overall idle energy increases.

These results are mainly due to the fact that in our experiments we did not consider the opportunity to switch off unused nodes. This is an interesting result as it clearly states that power capping solutions need to be integrated with idle resources shutdown and power management schemes in order to deliver the expected power saving. If this is not done the power capping risks to increase the idle power percentage as it works reducing the number of resources active at the same time. Constraint programming can be used to consider also the set-up and shut-down time for the idle resources; nevertheless such a problem was not part of the research discussed in this work.

We finally want to give an insight about a practical usage of the methodology proposed in this section, i.e. the integration between a power capped job dispatcher and the cooling system model. We collected the average hourly ambient temperature for the entire year 2014 from the ARPA ambient station of Zola Predosa[9], close by to the Cineca supercomputing center. We then forced three PUE targets (1.1, 1.075, 1.05) and we computed with our cooling model the maximum power budget which ensures the target PUE for each hourly ambient temperature during the year. According to the hourly power budget we computed the hourly Quality-of-Service loss, measured as normalized average queue time (WQTloss), and idle over active power percentage. We finally combined this value with the target PUE to compute an effective PUE embedding the energy efficiency loss due to the increased idle power percentage. In Table-5.2 we show the average results for the 2014 hourly ambient temperatures.

From the table we can see that imposing a PUE of 1.1 does not require significant power budgeting as the ambient temperatures sustains the 40KWatt budget for most of the time. Average power budget imposed by our approach is of 39KWatt. If we impose instead a PUE of 1.075 we can see that now the average power budget over the year decreases to 33KWatt, which leads only to the 0.7% of weighted queue time loss (i.e. user QoS loss) w.r.t. the 1.1 PUE case; at the same time, the idle power does not increase significantly, leading to an effective PUE equal to the target one. Finally, if we impose a constant PUE equal to 1.05 for the entire year, we can notice that our approach will lead to an average power budget of 29KWatt over the entire year, which in turn produces a weighted queue time loss of almost 9% and an effective PUE of 1.057. The actual PUE becomes slightly higher than the target one due to the increase of

[9] ARPA weather data Zola Predosa station

the idle power percentage in the supercomputer. These results show that our proposed power capped dispatcher can obtain significant power savings (PUE reduction) with negligible QoS loss, even when taking into account the cooling infrastructure and its susceptibility to environmental condition. Moreover, the idle power increase (the negative side effect) still remains below the PUE reduction induced by the power capping.

5.5 Variable Power Budget

In this section we are going to consider another problem related to power capping in HPC systems. So far, we discussed a job dispatcher that takes a fixed power budget as input - the power constraint is enforced at the same level for the whole duration of the experiments. Although in our experiments we presented the results obtained with different power budgets, each experiment required a constant power cap value. This is a clear limitation in real world contexts because the power budget available for the IT infrastructure of a supercomputer can change following mutating conditions and/or requirements. For example in Section 5.4 we saw how the external temperature can affect the cooling system efficiency and in turn this impacts the system power cap (less IT power if the ambient temperature is higher).

Therefore we added to our dispatcher(s) the possibility to modify the power constraint at run time. We extended the simulation framework in order to accept changing power cap levels as input; currently the different power caps are given in input as a list of couples $< t, p >$, specifying a power budget p at time t. The following question we must ask regards the behaviour of the system when the power cap changes. If the power budget increases at the next scheduling event more waiting jobs can start their execution (if the other required resources are available). In order to maximize system utilization the power constraint change triggers a new scheduling event, thus the dispatcher does not have to wait for the end of a running application to compute a new schedule.

When the power available is reduced the power constraint can be violated, depending on the power of the applications already running in the system. If the current workload power consumption does not exceed the reduced power cap, at the next schedule event the dispatcher will respect the new constraint by simply postponing the execution of new applications. Conversely, if the current power consumption is larger than the new power budget we have to take more drastic actions. An example of such action is reducing the power consumption of active jobs by decreasing the operational frequency of the computational resources they are using (Dynamic Voltage and Frequency Scaling) or setting a power cap to the involved nodes (RAPL).

Different methods can also be employed to reduce the consumption of jobs already in execution, for example techniques that modify at run time the characteristics of an application. Jobs that can dynamically change the number of resource used and can be migrated in other node partitions are defined as *malleable* in the literature [GASK14, Str03, DM07]. Other researches point in the direction of having jobs that can be stopped (reducing the power consumption to the node idle state level) and later restarted or having jobs that can be "slowed down" acting on the software level [WMES09, WMES10, CFG+05]. These methods are currently under study [CGR+10, MDSV07, UCL04, SAB+16]

but their adoption is not widespread yet.

Even though these methods are promising they are not implemented in nowadays supercomputers therefore we decided to focus on more widespread techniques such as frequency scaling. In the rest of this section we study the possibility to employ DVFS and act on the frequencies of the jobs currently running in the system, in order to slow them down and reduce the overall power consumption.

5.5.1 Frequency Reassignment Problem

In this section we are not going to deal with the problem of dispatching jobs in a supercomputer but only with the frequency assignment problem. More in particular, we are tackling the problem of how to optimally reassign the frequencies of a set of jobs running on the supercomputer and consuming a certain amount of power if the overall power constraint of the system changes. Therefore for the rest of the paper we will not consider the dispatching of the tasks in the HPC machine but we will assume that schedule and allocation have already been decided – and cannot be changed. We propose three different methods to deal with the frequency reassignment problem on a real supercomputer: 1) a greedy algorithm (Section 5.5.2), 2) a CP model with a dedicated search strategy (Section 5.5.3) and 3) a MIP model (Section 5.5.4). We use again the Eurora supercomputer as a testbed for our methods.

Frequency Variability Impact The key idea behind DVFS is that if we change the frequency of a task running on a HPC system (actually on every IT system) its power consumption changes accordingly, along with its duration. For example if we lower the frequency the power consumption decreases while the duration increases. The amount of this change strictly depends on the hardware characteristics of the considered system. A previous work [FBC+14] studied the impact of the frequency variability on jobs power consumption and duration in Eurora. The results of that paper are the base of our frequency assignment strategy; our decision whether to change the frequency of an activity or not relies on the power and duration variations related to such frequency change.

There are two main factors which determine the impact of changing a job frequency: 1) the application type and 2) the execution node. The cited paper studies three kinds of jobs: a real HPC application (Quantum EXPRESSO [GBBeA09]), a synthetic CPU-bound benchmark (i.e. a task which particularly stresses the CPU) and a synthetic MEM-bound benchmark (i.e. memory intensive application). In our work we follow this distinction. Clearly, slowing down a CPU-intensive application would cause a larger increase in duration w.r.t. to the same slow down for a memory-bound job, since CPU-bound jobs depend more heavily on the CPU speed and frequency. The second factor is the node on which the application is running: the frequency variations have different impact depending on the node type. In Eurora we have *high frequency* nodes, with frequency ranging from 1.2GHz up to 3.4GHz, and *low frequency* nodes, with frequency from 1.2GHz up to 2.1GHz. The current system always runs the applications at the maximum speed allowed by the execution nodes.

5.5.1.1 Problem Definition

We assume a set of jobs is currently running in the system consuming a total power P_{sys} at the current time ct and then we have to decide the best way to reassign the job frequencies given that the available power budget has decreased (i.e. the new power budget $P^{'}_{sys} = 0.7 * P_{sys}$). Hence, each job i is already running, with a start time st_i and an expected end time et_i, with $et_i = st_i + d_i$ where d_i is the expected duration at the current running frequency. In our current implementation the expected duration is the maximum allowed execution time declared by the user at submission time[10]. $d_i^{elapsed} = ct - st_i$ represents the elapsed duration of a job. Each job belongs to a specific queue (depending on the user choice 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.

We assume that each job is running at frequency f_i; the default dispatcher always assigns the maximum frequency possible when scheduling but the dispatcher can deal with different frequencies as input. Every job is also characterized by the type of the node it is running on nt_i and the application type at_i. The node type could assume two different values: 0, corresponding to a low frequency node, and 1, corresponding to a high frequency node. The application type can assume three values: 0 for average applications, 1 for CPU-bound applications and 2 for memory-bound applications. Both node type and application type have a strong impact on the variation of power and duration given a frequency change. Each job has a related power consumption p_i, which is the power consumed by the job running at the frequency decided by the dispatcher.

The goal of our problem is to assign a frequency to each job in order to respect the new power constraint. The main objective is not to disrupt the performance in terms of Quality-of-Service for the users while at the same time saving energy; both these goals can be reached by minimizing the job durations increase due to the jobs slow down. The slow down must be careful: if we slow down a job "too much" we could lose the energy benefit. Since $energy = power * time$ increasing the duration without sufficiently decreasing the power may lead to a rise in total energy.

As we have seen in previous chapters and sections, on Eurora we have four main queues, *debug*, *parallel*, *longpar* and *reservation*. Jobs belonging to the last queue need to be run within certain time frames, due to agreements between the computing center and its customers.

5.5.1.2 Problem Extensions

The base problem is quite simple and not particularly constrained and we wanted to also address more complex extensions. Therefore we modified the original problem with additional constraints that, while not currently used in the Eurora supercomputer, are common requirements in HPC settings. We first added job deadlines, i.e. each job i has a deadline dl_i and it must finish within that deadline (thus constraining the allowed duration increase). This will be referred as *Extension D*. Then we implemented a second extension (*Extension*

[10]Since this user-estimated duration usually differs from the actual job duration, future works may instead use a duration predicted via Machine Learning

R), namely we set dependencies among jobs (in particular end-to-end relationships). Each job i may have a set of "related" jobs Rel_i whose ends need to come after job i end[11]. Finally, we considered the case with both the deadlines and the job relations constraints, *Extension D-R*.

5.5.2 Greedy Algorithm

We first introduce a greedy algorithm: we decrease the frequency of few jobs as much as possible until we reach the desired power saving. The main advantage of this simple algorithm is its efficiency: even on the larger instances (up to 2000 jobs) the time required to produce a solution is negligible. This is a key aspect we cannot overlook since our purpose is to be able to implement our techniques on real systems with tight real-time requirements. We are therefore not interested in optimal solutions but in the best solution obtainable within a strict time limit.

The pseudo-code for the greedy algorithm is presented in Alg. 2. The set of currently running jobs is called *running_jobs*, *current_power* is the sum of the jobs powers, *desired_power* is an input of the algorithm and defines the new desired power budget; lines 1-3 initialize the algorithm. Line 4 sorts the jobs according to a combination of different factors: I) the job power consumption (we try to slow down "big" applications first in order to slow fewer jobs); II) the remaining duration of the job compared to the overall duration (we prefer not to slow down jobs closer to completion); III) the job priority, computed w.r.t. the expected waiting time (activities in queue with higher priorities should face smaller slowdowns). The actual equation which determines the weight w_i of a job i is:

$$w_i = p_i * (\alpha w_i^D + (1 - \alpha) w_i^Q) \tag{5.7}$$

where p_i is the power consumption, $w_i^D = d_i / d_i^{elapsed}$ is the ratio between total and elapsed duration and $w_i^Q = ewt_i / \max_{i \in J}(ewt_i)$ is the queue priority factor. α is the parameter which tells us which factor has more impact (in our experiments $\alpha = 0.6$).

Once we have sorted the jobs we proceed to loop until the desired power saving is reached (Line 5), in particular we extract the first job in the sorted list (Line 8) and we compute the minimal frequency at which he can run (Line 9) and the related new power and duration (Line 10 and 11). These functions are based on the values derived from the mentioned paper [FBC+14] and depend on the job characteristic (i.e. application type).

The algorithm then applies the changes to the selected job (notifying the change in power consumption and the new end time et_i', Line 12); finally in Line 13-14 we update the cumulative power gain obtained and remove the job from the list of those which can be selected (a slowed job cannot be slowed furthermore: its frequency has been already set to the minimum). If we set all jobs frequencies to their minimum value and we do not reach the desired power decrease then the algorithm simply fails (Line 6-7). This means that it is impossible to keep the power consumption under the required constraint without interrupting some of the running jobs[12].

[11] In a real application we would have to deal also with start-to-end relationships but in this case we suppose all our jobs have already started hence these possible relations must already

Algorithm 2: Greedy Algorithm

```
1  J ← running_jobs
2  power_goal ← current_power − desired_power
3  power_saved ← 0
4  Sort(J)
5  while power_saved < power_goal do
6      if J = ∅ then
7          return 0
8      j ← J[0]
9      new_freq ← GetMinFreq(j)
10     power_gain ← FindPowerGain(j, new_freq)
11     new_duration ← FindNewDur(j, new_freq)
12     Update(running_jobs, j, new_freq)
13     J ← J − {j}
14     power_saved ← power_saved + power_gain
15 return 1
```

We tried to modify the greedy algorithm in order to cope with the problem extensions. For example, in the case of the deadlines, setting the frequency of a selected job to its minimum only if the deadline will not be violated. We also tried to set the frequency at the middle value and check the deadline constraints. These variations strongly degrade the efficacy of the greedy algorithm as it is not able to solve the vast majority of instances with extensions. The main problem is that adding new constraints to the problem renders the greedy algorithm ineffective: with tighter problems a solution can be found only "spreading" the slow down among several jobs, which is opposite to the drastic behaviour of the greedy method (maximum slowing down of as few jobs as possible).

5.5.3 CP Approach

The greedy algorithm is very fast but it clearly lacks reasoning power. We then devised a Constraint Programming model to optimally reassign the job frequencies. Given our set J of running jobs, the most important variables of the model are a set of integer variables which represent the job frequencies $F_i \quad \forall i \in J$. The allowed frequency domain of each variable depends on the node on which the job is running: for low power nodes the allowed values are [1200, 1300, 1400, .., 2100]GHz and for high power nodes the range is [1200, 1400, 1600, .., 2800, 3100, 3400]GHz. Consequently, the domain of the F_i could either be [0, ..,9] if the job runs on a 2.1GHz node or [0, .., 10] if the job runs on a 3.1GHz node.

The relationship between the different frequencies and the changes in power (and duration) is encoded in a set of vectors. Given a job i we can identify the correct vector for the durations DM and the powers PM given the job application type ai_i and node type nt_i (the frequency change has a different

hold

[12] In the current system running applications cannot be interrupted and restarted and for this reason we do not consider this possibility

impact with different type of task or nodes). Each vector contains as many elements as the possible frequencies; each element specifies the duration/power change w.r.t. to a base frequency (as base frequency we chose the maximum). For example if a job has a duration of d_i at maximum frequency, the duration at frequency F_i' becomes $d_i' = DM[F_i'] * d_i$. The method for computing the new power is analogous.

The duration of job i can be seen as $d_i = d_i^{elapsed} + d_i^R$, the duration elapsed so far plus the remaining duration. These multipliers are encoded in two auxiliary variables for each job, P_i^{Mul} for power and D_i^{Mul} for duration. These variables are related to the frequency by the equations $D_i^{Mul} = DM[F_i]$ and $P_i^{Mul} = PM[F_i]$. These relations are expressed in the CP model via *element* constraints [HC88]:

$$element(F_i, DM, D_i^{Mul}) \quad \forall i \in J \tag{5.8}$$

$$element(F_i, PM, P_i^{Mul}) \quad \forall i \in J \tag{5.9}$$

We use two additional auxiliary variables to represent the new power NP_i and duration ND_i for each job:

$$NP_i = P_i^{Mul} * p_i \qquad \forall i \in J \tag{5.10}$$

$$ND_i = D_i^{Mul} * d_i^R + d_i^{elapsed} \qquad \forall i \in J \tag{5.11}$$

The duration increase DI_i of a job can be expressed as:

$$DI_i = (ND_i - d_i) = (D_i^{Mul} - 1) d_i^R \quad \forall i \in J \tag{5.12}$$

Now we need to impose the constraint on the new powers, i.e. their sum must not be greater than the new power cap p_{cap}:

$$\sum_{\forall i \in J} NP_i \leq p_{cap} \tag{5.13}$$

We also want to set special constraints for the duration increase of jobs in the reservation queue (JR), namely we want the maximum and the average duration increase of those jobs to be less or equal of, respectively, di_{res}^{max} and di_{res}^{avg}:

$$\frac{1}{|JR|} \sum_{\forall i \in JR} DI_i \leq di_{res}^{avg} \tag{5.14}$$

$$max_{\forall i \in JR} DI_i \leq di_{res}^{max} \tag{5.15}$$

The frequency reassignment problem has a dual goal: on one hand we want to reduce the energy consumed by the HPC system, on the other hand we want to maintain a good performance for the end users - i.e. we want to keep the duration increases as low as possible. We have experimented with several objective functions in order to reflect the different aspects we may be interested to optimize.

For example we tried to maximize the energy savings introducing a new set of variables to describe the difference between the energy consumed by a job with the new frequency and the energy consumed with the old frequency. The energy difference δE_i is given by the following equation:

$$\delta E_i = d_i^{elapsed} * p_i + (ND_i - d_i^{elapsed}) * NP_i - d_i * p_i \quad \forall i \in J \tag{5.16}$$

where $d_i * p_i$ is the old duration (old power times old duration) and the first two terms of the equations represent, respectively, the energy consumed so far by the job (duration so far $d_i^{elapsed}$ times old power) and the energy consumed till the end time (remaining duration times new power). The objective function then tries to minimize the sum of these energy differences: $\min \sum_{\forall i \in J} \delta E_i$.

For the duration increase we instead used this equation:

$$\min \sum_{\forall i \in J} DI_i \tag{5.17}$$

After having performed initial experiments we found out that minimizing the duration increases always guarantees an improvement in consumed energy, i.e. in no cases slowing down jobs caused the system to consume more energy. For this reason - combined with the fact that users strongly prefer their jobs not be slowed down - we decided to use the minimization of the duration increases (Eq. 5.17) as objective function in our experiments.

5.5.3.1 Search Strategy

We now discuss the search strategy. The foremost thing to remember is that we do not compute an optimal solution, we do not need a complete strategy: our goal is to obtain good solutions as fast as we can. Therefore we explored a selection of search strategies imposing a realistic time limit of 5 seconds - i.e. we want to know which is the best method within this time constraint.

The first search strategy we used is a basic version of the typical strategy used for integer variables in CP. Namely, the variable selection proceeds among unbound variables and chooses the variable with the smallest domain (the one with the smallest number of possible values). In case of tie, the selected variables is the one with the lowest min value. For the value selection, we first try to assign the maximum allowed value for the selected variable. We are going to call this strategy *CP Standard*.

Heuristic-based Search The CP standard search can produce good results but it requires a much longer time than the greedy algorithm - during the 5 seconds time limit the greedy algorithm was often able to produce better solutions, especially on instances of non-trivial size. We then devised a new search strategy able to produce solutions at least as good as the greedy ones and in a much shorter time than the standard strategy. To do that we used the insight provided by the quality of the greedy solutions to create a heuristic strategy search which combines both the benefit of the greedy algorithm and the capacity of exploring a larger search space typical of CP search strategies. Namely, our strategy starts from the solution generated by the greedy algorithm - hence we can quickly obtain a first, feasible solution - then tries to improve the current solution changing the value assignment of some variables.

The search proceeds in typical CP fashion: at each decision point a variable is selected and a new value is assigned. The variable selection procedure exploits the heuristic too: the variables are ordered with the same method used for the greedy algorithm and then the first variable in the ranking is chosen[13]. The

[13]There is actually a small change in the ranking method: if the objective function includes the energy, the weight of Eq. 5.7 is multiplied by the energy of the job (computed with the duration and power before the frequency change)

key idea is to consider first the variables that will have a greater impact on the objective function. Once a variable is selected we set its value to the maximum allowed. We will refer to this strategy as *CP + Heuristic*.

Large Neighborhood Search We also implemented a Large Neighborhood Search strategy [CB09], a metaheuristic which has been shown to be effective in solving several CP problems. Similarly to Local Search, with LNS we modify an existing solution to the problem. However, instead of making small modifications to a solution, changing one single variable, a subset of variables (called *fragment*) from the problem are selected and relaxed. A complete search - bounded by a search limit - is then performed on these relaxed variables. To perform this complete search step we used the heuristic-based search described in the previous section.

The three main aspects impacting the LNS efficacy are 1) the fragment selection procedure, 2) the fragment size and 3) the search limit. Despite the success of LNS, no generic principle has emerged yet on how to choose the parameters. They are currently set either with domain dependent heuristics or chosen randomly. Lacking a general scheme or set of guidelines, we investigated different possibilities, exploiting the knowledge of the problem domain.We tried with several parameters before finding the most performing configuration for our situation.

First, we need to decide the fragment selection procedure: select the subset of variables that should be relaxed. The simplest strategy is to randomly select variables until the fragment size is reached. Another possibility consists in selecting contiguous variables; this strategy is generally useful if used in problems with an underlying structure where contiguous are somehow related. In our case the variables are independent and therefore this selection method does not provide additional benefit compared to a random selection; conversely, random selection is able to explore the search space more effectively, avoiding local minima. Having knowledge of our problem, we can also use a selection mechanism that favours variables with a higher dynamic impact; the variable impact is defined by Mairy et al. in [MSD+10]. The dynamic impact of a variable tries to capture the effect that the variable would have on the objective if it is relaxed. Let S be a solution to the problem. The impact of a value v for a variable x_i is defined as the difference that is induced on the objective by setting x_i to v rather than to its value in S. We also tried with giving precedence to variable with higher weights (the weight is defined as in the heuristic-based search). After a preliminary exploration, the best method turned out to be combining complete random selection and weighted selection[14]. This strategy is able to provide good results thanks to trade-off between the diversification induced by the random component and the focus on promising variables offered by the weights-guided strategy.

After having decided the variables selection procedure we must choose the fragment size and search limit. A common policy used in practice is to have fixed values for both these parameters and as a first step we experimented as well with different fixed sizes and limits. Then, inspired by the work of Mairy et al. [MDVH11], we tried a different approach based on reinforcement learning. As

[14] A variable i is relaxed with probability $P = \psi \frac{w_i}{\sum_{\forall i \in J} w_i} + (1 - \psi)\frac{1}{|J|}$ where w_i is the weight and $\psi \in [0, 1]$ is a real number

we saw in Section 2.3.2.2, the core of reinforcement learning cab be summarized as choosing actions and observing the results. This idea can be applied to learning the right parameters for LNS: after an initial setup, the parameters values are updated according to the quality of the solutions found by previous LNS iterations. We thus explore the parameters space guided by the previous results.

5.5.3.2 Extensions

We must now model the extensions described in Sec. 5.5.1.2. The deadlines and the relations constraints are respectively expressed by the following equations:

$$ET_i \leq dl_i \qquad \forall i \in J \tag{5.18}$$

$$ET_i \leq ET_j \qquad \forall j \in Rel_i \qquad \forall i \in J \tag{5.19}$$

where $ET_i = st_i + d_i^{elapsed} + D_i^{MUL} d_i^R$ is the variable representing a job end time and Rel_i is the set of jobs related to i.

We update the search strategy as well. We still use the LNS framework but now the method to find the solution of the relaxed fragment changes. After a preliminary study, we discovered that if jobs relations are involved (Extensions R and D-R) the best search strategy to solve the relaxed problem at each LNS iteration is the CP standard search strategy. Conversely, to deal with the deadline we modified the heuristic described in Sec. 5.5.3.1. Now we obtain the first solution with a variant of the greedy algorithm: instead of indiscriminately setting the frequencies to their minimum levels, the modified version does that only if it will not force the slowed job to go beyond its deadline. Note that this first solution may be infeasible, i.e. could violate the power constraint. After the first solution is found the second phase is performed as before.

5.5.4 MIP Approach

We describe now a third approach to solve the frequency assignment problem, namely a Mixed Integer Program (MIP) model. This kind of method is generally well suited to deal with assignment problem (without scheduling constraints). In the MIP model we have a set of binary variables X where X_{if} assumes value 1 if job i has frequency f, 0 otherwise; index i has the whole set of job J as a range and f can vary in the job frequency range F, defined as before based on node type. Each job has exactly one frequency:

$$\sum_{f \in F} X_{if} = 1 \qquad \forall i \in J \tag{5.20}$$

For each job i we have a set of power and duration multipliers, one for each allowed frequency, respectively d_{if}^{Mul} and p_{if}^{Mul}. These multipliers combined with the current job power p_i and the binary variable allow us to express the power constraint (the power must be inferior to p_{cap}):

$$\sum_{i \in J} \sum_{f \in F} p_{if}^{Mul} p_i X_{if} \leq p_{cap} \tag{5.21}$$

We also have to impose the limits on the reservation jobs' duration increase: the maximum increase for these jobs must be less or equal than di_{res}^{max} and the

average increase less or equal than di_{res}^{avg}. We use another set of parameters $r_i, i \in J$ where $r_i = 1$ if job i is in the reservation queue, 0 otherwise. The reservation constraints are then the following (average and maximal):

$$\sum_{i \in J} \sum_{f \in F} r_i (d_{if}^{Mul} - 1) d_i^R X_{if} \leq di_{res}^{avg} \sum_{i \in J} r_i \tag{5.22}$$

$$\sum_{f \in F} r_i (d_{if}^{Mul} - 1) d_i^R X_{if} \leq di_{res}^{max} \qquad \forall i \in J \tag{5.23}$$

where $(d_{if}^{Mul} - 1) d_i^R X_{if}$ represents the duration increase of job i. Finally we specify the objective function. For the MIP model we considered only the minimization of the duration increase.

$$\min \sum_{i \in J} \sum_{f \in F} (d_{if}^{Mul} - 1) d_i^R X_{if} \tag{5.24}$$

In the MIP model the constraints introduced by the extensions are the same as in the CP model, namely Equations 5.18 and 5.19. The end time is expressed as $ET_i = st_i + d_i^{elapsed} + \sum_{f \in F} d_{if}^{Mul} d_i^R X_{if}$.

The method to solve the MIP problem is embedded in the MILP solver and it is the same for every type of problem, base and extended versions. As in most ILP problem the solving method falls in the branch-and-bound or cutting planes category, refined by years of research in the Operation Research community. The discussion of these methods is entirely outside the scope of this work.

5.5.5 Methods Comparison

We are going now to see the results of the experiments we performed in order to compare the three proposed methods. As a reminder, we exclusively study the frequency reassignment problem: we assume to have a set of running jobs (already mapped and scheduled) and due to the power budget variation we need to change these jobs frequencies. We decided to use the average duration increase of the jobs as our performance metric. This is the main concern for HPC system users and, as a side effect, we were able to guarantee energy savings even without focusing directly on the energy.

The proposed methods were implemented using *or-tools* [Goo]. The MIP solver used on top of or-tools is the open-source *Cbc* (Coin-or branch and cut) MILP solver [CO]. We evaluated all the approaches on instances which represent realistic workloads, derived from traces collected on Eurora in a timespan of several months. We have instances of several sizes, from smaller ones composed of 50 jobs up to the bigger ones with 2000 jobs. As told before, we have strict time constraints to produce a solution due to the real-time nature of our application. Therefore we set a time limit of 5 seconds to the solvers in all the experiments. The time limit is not a problem for the greedy algorithm on any of our instances; the time required by this method is always around a few hundreds of milliseconds.

5.5.5.1 Models Evaluation

In this section we compare the performance of the different approaches when dealing with the base version of the problem (without extension). In particular

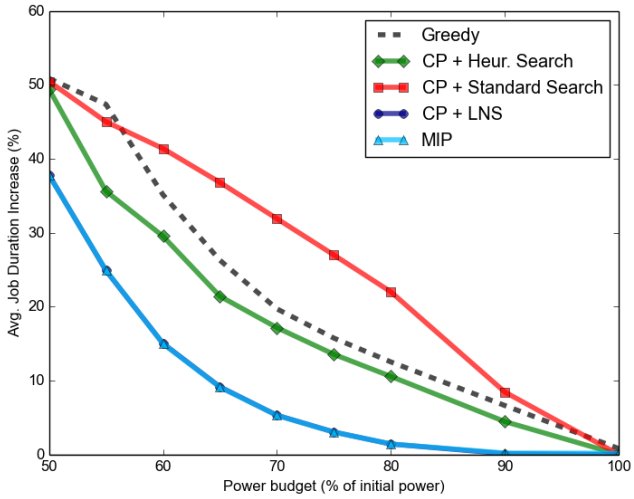
we show the results obtained by: 1) the greedy algorithm, 2) the CP model using standard search, 3) the CP model using heuristic-based search, 4) the CP model using LNS and 5) the MIP model. For the LNS case we used the technique with the best performance, namely the combination of weighted and random fragment selection (fragment size and time limit decided via reinforced learning) and the heuristic-based search to solve the relaxed fragments. For every model we ran experiments on 30 instances. For each instance we first compute the initial power (the power consumed by all jobs in the instance) then we try to reduce the power available and assign new frequencies; as power cap levels we tested decreasing percentages of the initial power (the power levels are identified by the markers in the plots). In the following graphs we report the duration increases and the energy differences; these values are the normalized averages for all instances.

To briefly summarize our experiments, when the MIP solver manages to find a solution within the time limit it also proves that the solution is optimal, so the remaining models can at best try to get as close as possible. In particular, in the case of the base problem the MIP approach can solve all the instances to optimality and clearly outperforms the other models, especially on larger instances. If we start to consider the problem extensions the MIP solver struggles more to find a solution for larger instances (obviously always within the time limit) w.r.t. to the CP solvers. Nevertheless, when it does find a solution such solution is optimal (or very close to the optimal one) and still outperforms the remaining models on the corresponding instance.

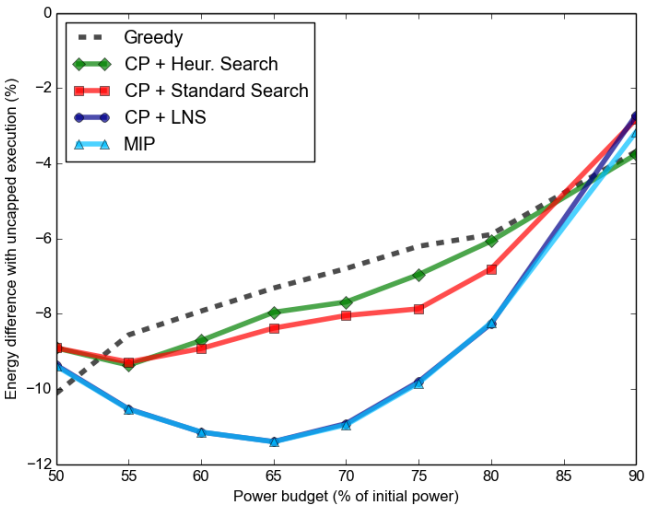
Figure 5.21 presents two graphs. On the top we can see the average duration increase (as a percentage) obtained when decreasing the power budget available (from the right, maximum power, to the left). The bottom graph depicts instead the energy difference (in percentage) w.r.t. to the maximal power budget. Although the objective function considers only the job durations, we report also the energy differences to prove that we obtain significant energy savings, even without explicitly focusing on it.

All the results we obtained clearly reveal that when we reduce the power budget the average duration of the jobs increases, as a large fraction of jobs need to be slowed down. As we can see in Figure 5.21a when we consider small instances (from 50 to 200 jobs) the MIP solver and CP plus LNS both provide optimal solutions - their lines completely overlap. It is very easy to observe that already with smaller instances there is a great gap between CP plus LNS and CP methods without LNS: in particular CP with the standard search provides the worst results, even worse than the greedy algorithm - except when the power becomes very tight (50%-55%). CP with the heuristic search is able to find solutions equal or better than the greedy algorithm ones - as we expected since it starts from the greedy solution and then improves it. In Figure 5.21b we can see that while both CP plus LNS and MIP find solutions very close to the optimal, these solutions are not the same: their related energy savings slightly differ.

In Figure 5.22 we increase the instance size up to 600 jobs and again we see both the duration increase (Figure 5.22a) and the energy difference (Figure 5.22b). When the instances become larger (400 to 800 jobs) we start to see the gap separating the solutions found by the MIP solver from those provided by the CP methods: Figure 5.22a reveals that even CP plus LNS is not able to find the optimal solution provided by MIP. However, the gap is minimal (at least for



(a) Avg. Duration Increase

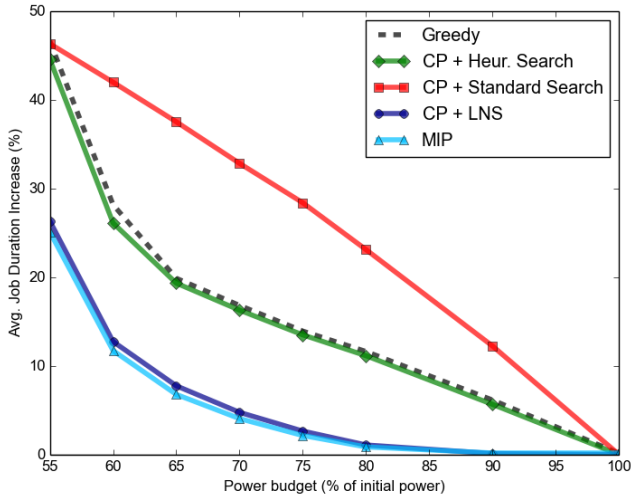


(b) Energy Difference

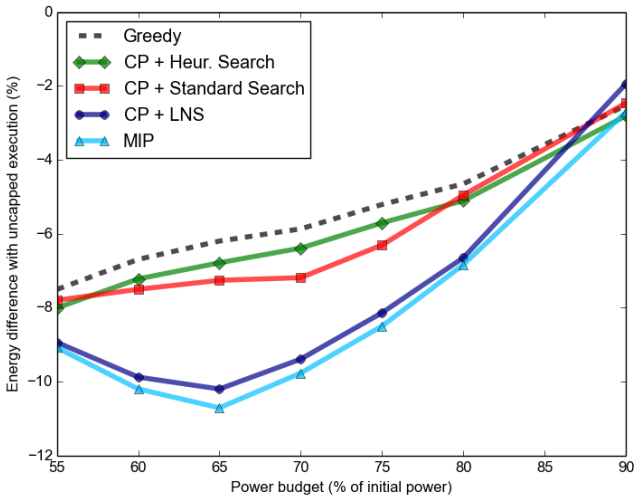
Figure 5.21: Base Problem - 100 Jobs (Duration increase and energy difference)

the 600 jobs case) and LNS still performs much better than the remaining CP and greedy methods.

In Figure 5.23 we consider an even bigger instance size, namely 1000 jobs. We can observe that the situation worsens for CP and greedy approaches when the instances grow larger (more than 1000 jobs): in Fig 5.23a we can see how the MIP now definitely outperforms all other methods, CP and LNS included. We can also see that with larger instances it is very difficult for the heuristic-based search to improve the first solution obtained by the greedy method. LNS manages to explore the search space more effectively but not enough to reach an optimal solution. With even bigger instances the optimality gap grows even



(a) Avg. Duration Increase



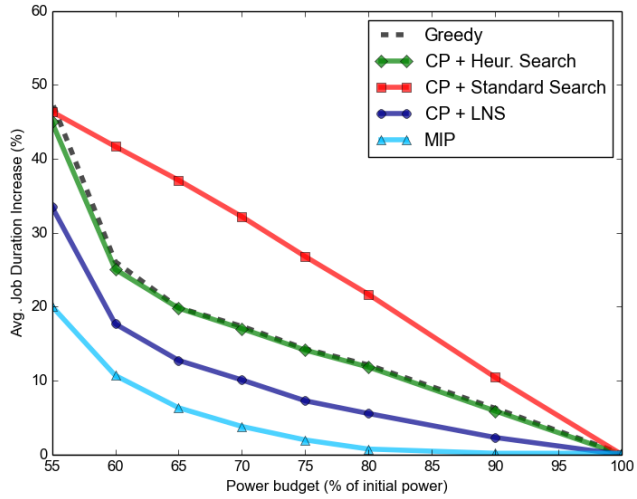
(b) Energy Difference

Figure 5.22: Base Problem - 600 jobs)

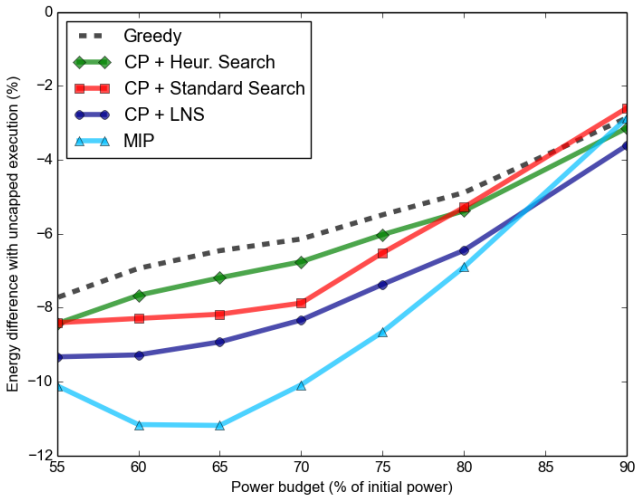
further. The main reason is the fact that the base problem is rather loosely constrained and, as it is known from the literature, MIP techniques are very effective when dealing with pure assignment problems.

5.5.5.2 Problem Extensions

The extensions we introduced raise the difficulty of the problem. As mentioned before we did not consider the greedy algorithm to deal with the extensions. Our experiments with the extended model reveal a behaviour comparable to the base



(a) Avg. Duration Increase



(b) Energy Difference

Figure 5.23: Base Problem - 1000 Jobs)

problem: the MIP approach is again the one able to find the best solutions, CP plus LNS approaches the quality of MIP with smaller instances and the gap gradually increases with larger instances. The remaining CP methods perform far worse then the previous models. The plots for all the extended models are extremely similar to those previously seen, hence we are not going to show them.

What we want to focus on is a key difference with the base problem, at least for the extensions which consider relationships among jobs. While all the CP models manage to find a solution for almost all the instances within the

Solver	*600 Jobs*			*1500 Jobs*			*2000 Jobs*		
	D	R	D-R	D	R	D-R	D	R	D-R
MIP	100	100	95	100	50	70	100	35	45
$CP + LNS$	100	100	95	100	100	90	100	95	95
$CP + Std$	100	100	100	100	100	100	100	100	100

Table 5.3: Extensions Experiments Summary. Each value represents the percentage (%) of solved instances.

time limit, the MIP solver fails to obtain feasible solutions for many of the larger instances. This aspect can be seen in Table 5.3. The table reports the percentages of solved instances for the MIP solver, CP plus LNS and CP with standard search (CP plus heuristic solves the same number of instances solved by CP plus LNS). Instances composed by 600, 1500 and 2000 jobs are reported (smaller instances are always solved by every model).

As a summary of the experimental evaluation, when we consider the base problem the MIP method outperforms the other ones when the dimension of the instances increases; on smaller instances the gap between the CP and MIP approaches is drastically reduced. The greedy algorithm proved to be the fastest method but often unable to find good quality solutions. If we take into account extensions increasing the difficulty of the initial problem the first weakness of the MIP approach is revealed, especially on larger instances. In particular, whereas the CP model almost always reaches a solution within the time limit, the MIP solver struggles with the job relations extension and finds a solution only in about half of the test instances (although it still outperforms the other approaches when it does find a solution).

5.6 Chapter Summary

In this chapter we addressed the issue of reducing the power consumption of HPC systems. We focused on a strategy known in literature as *power capping* whose goal is to contain the power consumption of a system within a power budget. We explored different facets of this issue. First, we introduced a power capped job dispatcher for HPC systems whose strength relies on the possibility to limit the power consumption acting on the job execution order alone. This method significantly differs from the vast majority of todays' techniques that aim at reducing power consumption through limiting the performance of computing nodes.

We proposed two different approaches to implement our strategy: 1) a heuristic algorithm and 2) a hybrid approach that decomposes the problem and combines a CP model (used to solve the scheduling subproblem) with a heuristic technique (allocation subproblem). The former is able to quickly find solutions - always a great skill in a setting with stringent real time requirements - while the latter can provide better results thanks to a deeper exploration of the solutions space. Both approaches do not aim at finding optimal solutions; in HPC dispatching guaranteeing a schedule is more important than reaching the theoretical optimum. A fundamental aspect of our approach is the requirement to

know the power consumption of a job before its real execution. This estimate is used by our dispatcher to schedule all jobs in such a way to guarantee that the power budget will never be violated during actual execution. Therefore we used the Machine Learning predictive model discussed in Chapter 2.3.3, capable of estimating the power consumption of a job with high accuracy, using only the information available at dispatching-time.

Thanks to a simulation framework based on the Eurora supercomputer we were able to test our methods and to compare them with several other techniques selected from the state-of-the-art. The results are extremely promising. Our approach is able to outperform the majority of the other methods (at least until we do not make unrealistic optimistic assumptions that favour excessively the methods based on hardware mechanisms). Our approach has also a very good scalability and it proves that proactive online job dispatching leads to better scheduling decisions compared to more reactive and less "smart" strategies. We also demonstrated that our approach is orthogonal to techniques that exploit hardware mechanisms to reduce power consumptions. This led us to combine the best of both worlds (integrating an hardware-based reactive component into our proactive scheduling-based approach) and obtain an even better HPC job dispatcher.

As case study, we tried to evaluate the benefits, in terms of energy savings, that can be obtained if the power capped job dispatcher is integrated with the other components of a supercomputer, such as the cooling infrastructure. The cooling system is one of the most energy-consuming part of HPC system facilities and its power consumption directly impacts the power available for the IT part. For example, the cooling system efficiency is strictly related to the outside temperature: in hotter days more power is required by the cooling infrastructure and therefore less power is available for the computing nodes. We then presented a free-cooling model describing the Eurora cooling system and we studied its behaviour in conjunction with the job dispatcher, revealing good performance in terms of machine utilization and energy savings.

Finally we considered the aspect of variable power budget. In many real-world situations the power budget available might vary during the course of days or weeks (again, in hotter days or periods less power is available for the IT infrastructure) and we are interested to see how that would affect our power capping method. We then extended the job dispatcher adding the possibility of having a variable power constraint. Subsequently a new problem arose: if the power budget decreases we risk that the current running jobs would already violate the changed power constraint. We developed a new module to tackle this issue: a frequency reassignment component that decreases the operational frequency of the nodes currently in use, causing a decrease in power consumption while augmenting the duration of the involved jobs. We presented and evaluated different techniques to solve this problem: 1) a greedy algorithm, 2) a CP model and 3) a MIP model.

There are clearly many very interesting areas of research still unexplored. A first extremely important step would be the implementation of our approach in a real HPC system, facing the unexpected real world challenges that would surely be encountered. Another very interesting aspect that needs to be studied is the accounting system. For example we can try to understand if users would be willing to accept slowed applications in exchange for discount rates. This is a complex problem which has not been studied in a satisfying way, yet, and

we reckon that it would be a great area to prove once again that applying optimization techniques to HPC issues can lead to great benefits.

Chapter 6

Conclusion

In this work we addressed the problem of online job dispatching in High Performance Computing systems and proposed several approaches to solve the problem. We explored a number of different techniques, ranging from heuristic algorithms to combinatorial optimization, looking for the right balance between solution quality and search effort. Our goal was to find feasible but not necessarily optimal solutions under the strict time limit imposed by the real time requirements of the task. In particular, we investigated how to apply Constraint Programming to the job dispatching problem and showed that a proactive strategy can perform better than the most widely adopted policies in supercomputers nowadays.

A central part of our study was devoted to the creation of a power-aware job dispatcher for HPC systems. We presented two novel approaches, one based on a heuristic technique and one based on the decomposition of the dispatching problem into its sub-components, scheduling (solved via Constraint Programming) and allocation (solved via a heuristic algorithm). Both our approaches have their foundations in the idea that the power consumption of a supercomputer can be bound within a power budget acting only on the management of the workload, i.e. changing the execution order of the activities. This approach is orthogonal to most of the methods for power-aware dispatching found in the literature and current state-of-the-art, that typically employ mechanisms trading performance for reduced power consumption. We demonstrated that the proposed approaches are scalable and perform better than most of the alternative techniques; moreover they can be extended with the integration of state-of-the-art mechanisms to further improve performance.

An important prerequisite for having a job dispatcher capable of enforcing a power cap is that some knowledge about the estimated power consumption of HPC applications must be available at schedule time, in order to affect the dispatching decisions. Thus, we devised a methodology to collect relevant data on a supercomputer and create a set of predictive models based on Machine Learning techniques and aimed at predicting HPC jobs power consumptions; the models were then seamlessly integrated into the dispatcher.

Future works are described in Chapters 3, 4 and 5. Additionally, we plan to refine the power-aware job dispatcher in order to implement it in a real HPC machine.

Bibliography

[9.911] ASHRAE Technical Committee (TC) 9.9. 2011 thermal guidelines for data processing environments expanded data center classes and usage guidance. Technical report, American Society of Heating, Refrigerating and Air-Conditioning Engineers, 2011.

[AAA01] F. Allen, G. Almasi, and et al. Andreoni. Blue Gene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2):310–327, 2001.

[AB93] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57 – 73, 1993.

[ABB+14] A. Auweter, A. Bode, M. Brehm, L. Brochard, N. Hammer, H. Huber, R. Panda, F. Thomas, and T. Wilde. A case study of energy aware scheduling on supermuc. In JulianMartin Kunkel, Thomas Ludwig, and HansWerner Meuer, editors, *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 394–409. Springer International Publishing, 2014.

[ABD+14] Matthew Anderson, Maciej Brodowicz, Luke Dalessandro, Jackson Debuhr, and Thomas Sterling. A dynamic execution model applied to distributed collision detection. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, pages 470–477, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[ABLL92] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[ABM11] Ismail Ababneh and Saad Bani-Mohammad. A new window-based job scheduling scheme for 2d mesh multicomputers. *Simulation Modelling Practice and Theory*, 19(1):482 – 493, 2011. Modeling and Performance Analysis of Networking and Collaborative Systems.

[ABR64] A. Aizerman, E. M. Braverman, and L. I. Rozoner. Theoretical foundations of the potential function method in pattern recog-

nition learning. *Automation and Remote Control*, 25:821–837, 1964.

[ABSM01] S Ali, T.D Braun, H.J Siegel, and A.A Maciejewski. *Heterogeneous Computing*. J Urbana, Encyclopedia of Distributed Computing, Kluwer Academic, Norwell, 2001.

[AdC03] Silvia Acid and Luis M de Campos. Searching for bayesian network structures in the space of restricted acyclic partially directed graphs. *Journal of Artificial Intelligence Research*, 18:445–490, 2003.

[AEOP02] Ravindra K Ahuja, Özlem Ergun, James B Orlin, and Abraham P Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[AF01] Ahuva W Mu Alem and Dror 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.

[AGJ+14] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 647–658, Piscataway, NJ, USA, 2014. IEEE Press.

[Aha97] David W. Aha, editor. *Lazy Learning*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[Aid00] Kento Aida. Effect of job size characteristics on job scheduling performance. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–17. Springer, 2000.

[AL97] Emile Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[AMW+10] Dennis Abts, Michael R Marty, Philip M Wells, Peter Klausler, and Hong Liu. Energy proportional datacenter networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 338–347. ACM, 2010.

[And08] David R Anderson. *Information Theory and Entropy*. Springer, 2008.

[BA97] Leonard A Breslow and David W Aha. Simplifying decision trees: A survey. *The Knowledge Engineering Review*, 12(01):1–40, 1997.

[Bac00] Fahiem Bacchus. Extending forward checking. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 35–51, London, UK, UK, 2000. Springer-Verlag.

[Bar89] Horace B Barlow. Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.

[Bar05] Roman Barták. R. dechter, constraint processing, morgan kaufmann (2003). *Artif. Intell.*, 169(2):142–145, 2005.

[BB99] David D. Bedworth and James E. Bailey. *Integrated Production Control Systems: Management, Analysis, Design.* John Wiley and Sons, Inc., New York, NY, USA, 2nd edition, 1999.

[BBB+14] A. Bartolini, A. Borghesi, T. Bridi, M. Lombardi, and M. Milano. Proactive workload dispatching on the EURORA supercomputer. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 765–780. Springer, 2014.

[BBCea08] K. Bergman, S. Borkar, D. Campbell, and et al. Exascale computing study: Technology challenges in achieving exascale systems, September 2008.

[BBGM05] Luca Benini, Davide Bertozzi, Alessio Guerri, and Michela Milano. Allocation and scheduling for mpsocs via decomposition and no-good generation. In *Principles and Practice of Constraint Programming-CP 2005*, pages 107–121. Springer, 2005.

[BBHU10] Ron Brightwell, Brian W Barrett, K Scott Hemmert, and Keith D Underwood. Challenges for high-performance networking for exascale computing. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6. IEEE, 2010.

[BBL+16a] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. *Predictive Modeling for Job Power Consumption in HPC Systems*, pages 181–199. Springer International Publishing, Cham, 2016.

[BBL+16b] 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, Oct 2016.

[BCC+14] 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*, March 2014.

[BCL+15] 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 - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, pages 524–540, 2015.

[BCLB15] 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 *2015 International Conference on High Performance Computing & Simulation, HPCS 2015, Amsterdam, Netherlands, July 20-24, 2015*, pages 88–95, 2015.

[BCP08] Nicolas Beldiceanu, Mats Carlsson, and Emmanuel Poder. New filtering for the cumulative constraint in the context of non-overlapping rectangles. In Laurent Perron and MichaelA. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5015 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin Heidelberg, 2008.

[BCR05] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. 2005.

[BCTB13] Andrea Bartolini, Matteo Cacciari, Andrea Tilli, and Luca Benini. Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller. *IEEE Trans. Parallel Distrib. Syst.*, 24(1):170–183, 2013.

[Bel56] Richard Bellman. A problem in the sequential design of experiments. *Sankhyā: The Indian Journal of Statistics (1933-1960)*, 16(3/4):221–229, 1956.

[Bel07] Christian L Belady. In the data center, power and cooling costs more than the it equipment it supports. *Electronics cooling*, 13(1):24, 2007.

[Ben12] Shajulin Benedict. Energy-aware performance analysis methodologies for {HPC} architecturesan exploratory study. *Journal of Network and Computer Applications*, 35(6):1709 – 1719, 2012.

[Bes94] Christian Bessiere. Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1):179–190, 1994.

[Bes06] Christian Bessiere. Constraint propagation. *Foundations of Artificial Intelligence*, 2:29–83, 2006.

[BFSO84] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984.

[BGN+10] Josep Ll Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on energy-Efficient Computing and Networking*, pages 215–224. ACM, 2010.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is nearest neighbor meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.

[BGT11] Josep Ll Berral, Ricard Gavalda, and Jordi Torres. Adaptive scheduling on power-aware managed data-centers using machine learning. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 66–73. IEEE Computer Society, 2011.

[BGV92] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[BH07a] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40, 2007.

[BH07b] Russell Bent and Pascal Van Hentenryck. Randomized adaptive spatial decoupling for large-scale vehicle routing with time windows. *AAAI*, 2007.

[BH10] Russell Bent and Pascal Van Hentenryck. Spatial, temporal, and hybrid decompositions for large-scale vehicle routing with time windows. *... and Practice of Constraint ProgrammingCP ...*, 2010.

[Bis95] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[BK07] Jirachai Buddhakulsomsiri and David S. Kim. Priority rule-based heuristic for multi-mode resource-constrained project scheduling problems with resource vacations and activity splitting. *European Journal of Operational Research*, 178(2):374 – 390, 2007.

[BKK+98] Jeffrey P Bradford, Clayton Kunz, Ron Kohavi, Cliff Brunk, and Carla E Brodley. Pruning decision trees with misclassification costs. In *European Conference on Machine Learning*, pages 131–136. Springer, 1998.

[BLK83] J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11 – 24, 1983.

[BLLN06] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten. Constraint-based scheduling and planning. *Foundations of Artificial Intelligence*, 2:761–799, 2006.

[BLR+14] Peter E. Bailey, David K. Lowenthal, Vignesh Ravi, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Adaptive configuration selection for power-constrained heterogeneous systems. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, BRACIS '14, pages 371–380, Washington, DC, USA, 2014. IEEE Computer Society.

[BM91] MJ Bagajewicz and V Manousiouthakis. On the generalized benders decomposition. *Computers & chemical engineering*, 15(10):691–700, 1991.

[BM01] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 171–182. IEEE, 2001.

[BMVG10] Ayan Banerjee, Tridib Mukherjee, Georgios Varsamopoulos, and Sandeep K. S. Gupta. Cooling-aware and thermal-aware workload placement for green hpc data centers. In *Proceedings of the International Conference on Green Computing*, GREENCOMP '10, pages 245–256, Washington, DC, USA, 2010. IEEE Computer Society.

[BMVG11] Ayan Banerjee, Tridib Mukherjee, Georgios Varsamopoulos, and Sandeep KS Gupta. Integrating cooling awareness with thermal aware workload placement for hpc data centers. *Sustainable Computing: Informatics and Systems*, 1(2):134–150, 2011.

[BP96] Philippe Baptiste and Claude Le Pape. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Scheduling, Proceedings 15 th Workshop of the U.K. Planning Special Interest Group*, 1996.

[BP07] Nicolas Beldiceanu and Emmanuel Poder. A continuous multi-resources cumulative constraint with positive-negative resource consumption-production. In Pascal Van Hentenryck and Laurence Wolsey, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 4510 of *Lecture Notes in Computer Science*, pages 214–228. Springer Berlin Heidelberg, 2007.

[BPN01a] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling*. Kluwer Academic Publishers, 2001.

[BPN01b] P. Baptiste, C.L. Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research & Management Science. Springer US, 2001.

[BPV10] L. Brochard, R. Panda, and S. Vemuganti. Optimizing performance and energy of hpc applications onpower7. *Computer Science - Research and Development*, 25(3-4):135–140, 2010.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[BR88] Avrim Blum and Ronald L. Rivest. Training a 3-node neural network is np-complete. In *Proceedings of the First Annual Workshop on Computational Learning Theory*, COLT '88, pages 9–18, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[Bré79] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979.

[Bre96a] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[Bre96b] Leo Breiman. Stacked regressions. *Machine learning*, 24(1):49–64, 1996.

[Bre01] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[Bru81] Maurice Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information processing letters*, 12(1):36–39, 1981.

[Bru00] Ivan Bruha. From machine learning to knowledge discovery: Survey of preprocessing and postprocessing. *Intelligent Data Analysis*, 4(3, 4):363–374, 2000.

[BSM01] TracyD. Braun, HowardJay Siegel, and AnthonyA. Maciejewski. Heterogeneous computing: Goals, methods, and open problems. In Burkhard Monien, ViktorK. Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing HiPC 2001*, volume 2228 of *Lecture Notes in Computer Science*, pages 307–318. Springer Berlin Heidelberg, 2001.

[BSRH14] Deva Bodas, Justin Song, Murali Rajappa, and Andy Hoffman. Simple power-aware scheduler to limit power consumption by hpc system within a budget. In *Proceedings of the 2Nd International Workshop on Energy Efficient Supercomputing*, E2SC '14, pages 21–30, Piscataway, NJ, USA, 2014. IEEE Press.

[BT95] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[BT13] Kenneth R Baker and Dan Trietsch. *Principles of sequencing and scheduling*. John Wiley & Sons, 2013.

[BTW14] Prasanna Balaprakash, Ananta Tiwari, and Stefan M. Wild. *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation: 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers*, chapter Multi Objective Optimization of HPC Kernels for Performance, Power, and Energy, pages 239–260. Springer International Publishing, Cham, 2014.

[Bur98] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[Cam10] Kirk W Cameron. The challenges of energy-proportional computing. *Computer*, 43(5):0082–83, 2010.

[Can08] Salvador Perez Canto. Application of benders decomposition to power plant preventive maintenance scheduling. *European journal of operational research*, 184(2):759–777, 2008.

[CB00] Walfredo Cirne and Francine Berman. Adaptive selection of partition size for supercomputer requests. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPDPS '00/JSSPP '00, pages 187–208, London, UK, UK, 2000. Springer-Verlag.

[CB01a] W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 8 pp.–, Apr 2001.

[CB01b] Walfredo Cirne and Francine Berman. A comprehensive model of the supercomputer workload. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 140–148. IEEE, 2001.

[CB02] Walfredo Cirne and Francine Berman. Using moldability to improve the performance of supercomputer jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002.

[CB09] Tom Carchrae and J.Christopher Beck. Principles for the design of large neighborhood search. *Journal of Mathematical Modelling and Algorithms*, 8(3):245–270, 2009.

[CCF+99] Steve J Chapin, Walfredo Cirne, Dror G Feitelson, James Patton Jones, Scott T Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Job Scheduling Strategies for Parallel Processing*, pages 67–90. Springer, 1999.

[CCS+06] Jiannong Cao, AlvinT.S. Chan, Yudong Sun, SajalK. Das, and Minyi Guo. A taxonomy of application scheduling tools for high performance cluster computing. *Cluster Computing*, 9(3):355–371, 2006.

[CFG+05] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM.

[CG88] Gail A. Carpenter and Stephen Grossberg. The art of adaptive pattern recognition by a self-organizing neural network. *Computer*, 21(3):77–88, 1988.

[CGF05] Kirk W Cameron, Rong Ge, and Xizhou Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, (11):40–47, 2005.

[CGR+10] Márcia C. Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe O. A. Navaux. *Supporting Malleability in Parallel Architectures with Dynamic CPUSETsMapping and Dynamic MPI*, pages 242–257. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[CGUeA08] J. Choi, S. Govindan, B. Urgaonkar, and et Al. Profiling, prediction, and capping of power consumption in consolidated environments. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10. IEEE, 2008.

[CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[CH92] Gregory F Cooper and Edward Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine learning*, 9(4):309–347, 1992.

[CH10] Elvin Coban and John N Hooker. Single-facility scheduling over long time horizons by logic-based benders decomposition. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 87–91. Springer, 2010.

[Cha91] Eugene Charniak. Bayesian networks without tears. *AI magazine*, 12(4):50, 1991.

[CHCR11] R. Cochran, C. Hankendi, A. K Coskun, and S. Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 175–185. ACM, 2011.

[CHHSW16] Elvin Coban, Aliza Heching, J. N. Hooker, and Alan Scheller-Wolf. *Robust Scheduling with Logic-Based Benders Decomposition*, pages 99–105. Springer International Publishing, Cham, 2016.

[Chi02] David Maxwell Chickering. Optimal structure identification with greedy search. *Journal of machine learning research*, 3(Nov):507–554, 2002.

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

[CKL+13] Michele Carpené, Iraklis A Klampanos, Siew Hoon Leong, Emanuele Casarotti, Peter Danecek, Graziella Ferini, André Gemünd, Amrey Krause, Lion Krischer, Federica Magnoni, et al. Towards addressing cpu-intensive seismological applications in europe. In *International Supercomputing Conference*, pages 55–66. Springer, 2013.

[CLCS10] Jin Xin Cao, Der-Horng Lee, Jiang Hang Chen, and Qixin Shi. The integrated yard truck and yard crane scheduling problem: Benders decomposition-based methods. *Transportation Research Part E: Logistics and Transportation Review*, 46(3):344–353, 2010.

[CLHH09] Sao-Jie Chen, Guang-Huei Lin, Pao-Ann Hsiung, and Yu-Hen Hu. *Hardware Software Co-Design of a Multimedia SOC Platform*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[CLPeA14] GL.T. Chetsa, L. Lefevre, J. Pierson, and et Al. Exploiting performance counters to predict and improve energy performance of hpc systems. *Future Generation Computer Systems*, 36:287–298, 2014.

[CM05] G. Contreras and M. Martonosi. Power prediction for intel xscale®processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 221–226, New York, NY, USA, 2005. ACM.

[CO] COIN-OR. Cbc (coin-or branch and cut) milp solver. `https://projects.coin-or.org/Cbc`.

[Col14] Francesca Collina. Tecniche di workload dispatching sotto vincoli di potenza. Master's thesis, Alma Mater Studiorum Università di Bologna, 2014.

[Cor09] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developers Manual, December 2009. no. 253669-033US.

[CPS+96] Soumen Chakrabarti, Cynthia A Phillips, Andreas S Schulz, David B Shmoys, Cliff Stein, and Joel Wein. Improved scheduling algorithms for minsum criteria. In *Automata, Languages and Programming*, pages 646–657. Springer, 1996.

[CR95] Yves Chauvin and David E Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

[CS93] M. Calzarossa and G. Serazzi. Workload characterization: a survey. *Proceedings of the IEEE*, 81(8):1136–1150, Aug 1993.

[CST00] Nello Cristianini and John Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[CT12] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[CV96] Su-Hui Chiang and Mary K Vernon. Dynamic vs. static quantum-based parallel processor allocation. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 200–223. Springer, 1996.

[CV01] S-H Chiang and Mary K Vernon. Production job scheduling for parallel shared memory systems. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 10–pp. IEEE, 2001.

[CW03] J Candy and RE Waltz. Anomalous transport scaling in the diii-d tokamak matched by supercomputer simulation. *Physical review letters*, 91(4):045001, 2003.

[CXT+13] Aftab Ahmed Chandio, Cheng-Zhong Xu, Nikos Tziritas, Kashif Bilal, and Samee U Khan. A comparative study of job scheduling strategies in large-scale parallel computational systems. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 949–957. IEEE, 2013.

[CY01] L. S. Camargo and T. Yoneyama. Specification of training sets and the number of hidden neurons for multilayer perceptrons. *Neural Comput.*, 13(12):2673–2680, December 2001.

[Cyb88] G. Cybenko. Continuous valued neural networks with two hidden layers are sufficient. Technical report, 1988.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[Dar15] Waltenegus Dargie. A stochastic model for estimating the power consumption of a processor. *IEEE Transactions on Computers*, 64(5):1311–1322, 2015.

[Day99] Peter Dayan. Unsupervised learning. *The MIT encyclopedia of the cognitive sciences*, 1999.

[Dec86] Rina Dechter. *Learning while searching in constraint-satisfaction problems*. University of California, Computer Science Department, Cognitive Systems Laboratory, 1986.

[Dec90] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[DF99] Allen B. Downey and Dror G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Perform. Eval. Rev.*, 26(4):14–29, March 1999.

[DGH+10] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM.

[DH95] Robert D Dony and Simon Haykin. Neural network approaches to image compression. *Proceedings of the IEEE*, 83(2):288–303, 1995.

[Die00a] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[Die00b] Thomas G Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, 40(2):139–157, 2000.

[Die02] Thomas G Dietterich. Ensemble learning. *The handbook of brain theory and neural networks*, 2:110–125, 2002.

[DM07] Richard A. Dutton and Weizhen Mao. Online scheduling of malleable parallel jobs. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '07, pages 136–141, Anaheim, CA, USA, 2007. ACTA Press.

[DMA98] Ramon Lopez De Mantaras and Eva Armengol. Machine learning from examples: Inductive and lazy methods. *Data & Knowledge Engineering*, 25(1):99–123, 1998.

[DMS94] J. J. Dongarra, H. W. Meuer, and E. Strohmaier. 29th top500 Supercomputer Sites. Technical report, Top500.org, November 1994.

[Dow97] Allen B Downey. A model for speedup of parallel programs. Technical report, Berkeley, CA, USA, 1997.

[Dow98] Allen B Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, 1(1):133–145, 1998.

[DP03] Emilie Danna and Laurent Perron. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. *Principles and Practice of Constraint Programming ...*, pages 817–821, 2003.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[DWF16] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2016.

[DXYY07] Wenyuan Dai, Gui-Rong Xue, Qiang Yang, and Yong Yu. Transferring naive bayes classifiers for text classification. In *Proceedings of the national conference on artificial intelligence*, volume 22, page 540. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[ECLV10a] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Optimizing job performance under a given power constraint in hpc centers. In *Green Computing Conference, 2010 International*, pages 257–267, Aug 2010.

[ECLV10b] Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Utilization driven power-aware parallel job scheduling. *Computer Science - Research and Development*, 25(3):207–216, 2010.

[ECLV12a] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Parallel job scheduling for power constrained {HPC} systems. *Parallel Computing*, 38(12):615 – 630, 2012.

[ECLV12b] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Understanding the future of energy-performance trade-off via {DVFS} in {HPC} environments. *Journal of Parallel and Distributed Computing*, 72(4):579 – 590, 2012.

[EHB+13] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, et al. 591 tflops multi-trillion particles simulation on supermuc. In *International Supercomputing Conference*, pages 1–12. Springer, 2013.

[Elo99] Tapio Elomaa. The biases of decision tree pruning strategies. In *International Symposium on Intelligent Data Analysis*, pages 63–74. Springer, 1999.

[ELT91] Jacques Erschler, Pierre Lopez, and Catherine Thuriot. Raisonnement temporel sous contraintes de ressource et problèmes d'ordonnancement. *Revue d'intelligence artificielle*, 5(3):7–32, 1991.

[EMRS15] Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. Dynamic power sharing for higher job throughput. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 80:1–80:11, New York, NY, USA, 2015. ACM.

[ERL90] Hesham El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.*, 9(2):138–153, June 1990.

[ET96] M.M. Eshaghian and J.F. Traub. *Heterogeneous Computing.* Artech House, Norwood, MA, 1996.

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

[EW01] Andrew Eremin and Mark Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In *Principles and Practice of Constraint ProgrammingCP 2001*, pages 1–15. Springer, 2001.

[FB89] D. Fernandez-Baca. Allocating modules to processors in a distributed system. *Software Engineering, IEEE Transactions on*, 15(11):1427–1436, Nov 1989.

[FBC+14] Francesco Fraternali, Andrea Bartolini, Carlo Cavazzoni, Giampietro Tecchiolli, and Luca Benini. Quantifying the impact of variability on the energy efficiency for a next-generation ultra-green supercomputer. In *International Symposium on Low Power Electronics and Design, ISLPED'14, La Jolla, CA, USA - August 11 - 13, 2014*, pages 295–298, 2014.

[FC07] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *IEEE Computer*, 40(12), December 2007.

[Fei96] Dror G Feitelson. Packing schemes for gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110. Springer, 1996.

[Fei97] Dg Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). *IBM Research Report RC19790 (87657) 2nd Revision*, 16:104–113, 1997.

[Fel13] M. Feldman. With roadrunners retirement, petascale enters middle age. https://www.top500.org/news/with-roadrunners-retirement-petascale-enters-middle-age/, 2013.

[Fen03] Wu-chun Feng. Making a case for efficient supercomputing. *Queue*, 1(7):54–64, October 2003.

[FFG08] Wu-chun Feng, Xizhou Feng, and Rong Ge. Green supercomputing comes of age. *IT professional*, 10(1):17–23, 2008.

[FGC05] Xizhou Feng, Rong 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*, pages 34–34, April 2005.

[FJ97] Dror G Feitelson and Morris A Jettee. Improved utilization and responsiveness with gang scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 238–261. Springer, 1997.

[FKM+11] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[FL05] Vincent W. Freeh and David K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 164–173, New York, NY, USA, 2005. ACM.

[Fle87] R. Fletcher. *Practical Methods of Optimization; (2Nd Ed.)*. Wiley-Interscience, New York, NY, USA, 1987.

[FLP+07] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, Barry L. Rountree, and Mark E. Femal. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):835–848, June 2007.

[FLYeA16] Haohuan Fu, Junfeng Liao, Jinzhe Yang, and et Al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):1–16, 2016.

[FR92] Dror G Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[FR95] Dror G Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In *Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer, 1995.

[FR96] Dror G Feitelson and Larry Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer, 1996.

[FR97] D.G. Feitelson and L. Rudolph. *Job Scheduling Strategies for Parallel Processing: IPPS '97 Workshop, Geneva, Switzerland, April 5, 1997, Proceedings*. Number v. 3 in Lecture Notes in Artificial Intelligence. Springer, 1997.

[FR98] DrorG. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In DrorG. Feitelson and Larry

Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 1998.

[Fre90] Yoav Freund. Boosting a weak learning algorithm by majority. In *COLT*, volume 90, pages 202–216, 1990.

[FRS+] Dror G Feitelson, Larry Rudolph, Kenneth C Sevcik, Uwe Schwiegelshohn, and Parkson Wong. Theory and Practice in Parallel Job Scheduling 1 Introduction 2 Survey of Theoretical Results.

[FRS05] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004. Revised Selected Papers*, chapter Parallel Job Scheduling — A Status Report, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[Fuk88] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

[FWB07] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007.

[FZB09] Mohammad M Fazel-Zarandi and J Christopher Beck. Solving a location-allocation problem with logic-based benders decomposition. In *Principles and Practice of Constraint Programming-CP 2009*, pages 344–351. Springer, 2009.

[GASK14] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V Kalé. Towards realizing the potential of malleable jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[GB65] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, October 1965.

[GBBeA09] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, and et Al. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502 (19pp), 2009.

[GBS06] Pall Oskar Gislason, Jon Atli Benediktsson, and Johannes R Sveinsson. Random forests for land cover classification. *Pattern Recognition Letters*, 27(4):294–300, 2006.

[Gen01] Marc G Genton. Classes of kernels for machine learning: a statistics perspective. *Journal of machine learning research*, 2(Dec):299–312, 2001.

[Geo72] Arthur M Geoffrion. Generalized benders decomposition. *Journal of optimization theory and applications*, 10(4):237–260, 1972.

[Ger94] Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In *Logic Programming - Proceedings of the 1994 International Symposium, pages 339–358, Massachusetts Institute of Technology*, pages 339–358. The MIT Press, 1994.

[GFGS10] Eric K Garcia, Sergey Feldman, Maya R Gupta, and Santosh Srivastava. Completely lazy learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(9):1274–1285, 2010.

[GH06] Philippe Galinier and Alain Hertz. A survey of local search methods for graph coloring. *Computers & Operations Research*, 33(9):2547–2562, 2006.

[Gha01] Zoubin Ghahramani. An introduction to hidden markov models and bayesian networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 15(01):9–42, 2001.

[Gha04] Zoubin Ghahramani. Unsupervised learning. In *Advanced lectures on machine learning*, pages 72–112. Springer, 2004.

[GHBD+09] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, Jeffrey O Kephart, and Charles Lefurgy. Power capping via forced idleness. 2009.

[Gid86] Dimitri Gidaspow. Hydrodynamics of fiuidizatlon and heat transfer: Supercomputer modeling. *Applied Mechanics Reviews*, 39(1):1–23, 1986.

[Gil01] Rich Caruana Steve Lawrence Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, volume 13, page 402. MIT Press, 2001.

[Gin93] M. L. Ginsberg. Dynamic Backtracking. 1:25–46, 1993.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GJ14] Jim Gao and Ratnesh Jamidar. Machine learning applications for data center optimization. *Google White Paper*, 2014.

[GKD11] Erkam Guresen, Gulgun Kayakutlu, and Tugrul U Daim. Using artificial neural network models in stock market index prediction. *Expert Systems with Applications*, 38(8):10389–10397, 2011.

[GL97] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[GLNI] Daniel Godard, Philippe Laborie, Wim Nuijten, and S A Ilog. Randomized Large Neighborhood Search for Cumulative Scheduling.

[glo15] Global constraint catalog. `http://sofdem.github.io/gccat/`, 2015. Accessed: 2015-03-26.

[GMVRGS15] Csar Gómez-Martín, Miguel A. Vega-Rodrguez, and José-Luis González-Sánchez. Performance and energy aware scheduling simulator for hpc: evaluating different resource selection methods. *Concurrency and Computation: Practice and Experience*, 27(17):5436–5459, 2015. cpe.3607.

[Gom04] CarlaP. Gomes. Randomized backtrack search. In Michela Milano, editor, *Constraint and Integer Programming*, volume 27 of *Operations Research/Computer Science Interfaces Series*, pages 233–291. Springer US, 2004.

[Goo] Google. or-tools. `https://developers.google.com/optimization/`.

[Goo50] I.J. Good. *Probability and the Weighing of Evidence*. Charles Griffin, 1950.

[Gos09] Abhijit Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009.

[Gre75] SA Greibach. *Lecture Notes in Computer Science*. 1975.

[GTU91] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 19, pages 120–132. ACM, 1991.

[GWB+03] Gongde Guo, Hui Wang, David Bell, Yaxin Bi, and Kieran Greer. Knn model-based approach in classification. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 986–996. Springer, 2003.

[Hau89] R. Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, 1989.

[HBBD16] Alexander Heinecke, Alexander Breuer, Michael Bader, and Pradeep Dubey. *High Order Seismic Simulations on the Intel Xeon Phi Processor (Knights Landing)*, pages 343–362. Springer International Publishing, Cham, 2016.

[HC88] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proceedings of the 7th National Conference on Artificial Intelligence. St. Paul, MN, August 21-26, 1988.*, pages 660–664, 1988.

[HE80] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[Hec98] David Heckerman. A tutorial on learning with bayesian networks. In *Learning in graphical models*, pages 301–354. Springer, 1998.

[HF05a] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 1–, Washington, DC, USA, 2005. IEEE Computer Society.

[HF05b] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 1. IEEE Computer Society, 2005.

[HFA05] Chung-Hsing Hsu, Wu-chun Feng, and Jeremy S Archuleta. Towards efficient supercomputing: A quest for the right metric. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. pages 607–613. Morgan Kaufmann, 1995.

[HHN08a] Junichi Hikita, Akio Hirano, and Hiroshi Nakashima. Saving 200kw and $200 k/year by power-aware job/machine scheduling. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[HHN08b] Junichi Hikita, Akio Hirano, and Hiroshi Nakashima. Saving 200kw and $200 k/year by power-aware job/machine scheduling. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[HHTC14] Kuo-Chan Huang, Tse-Chi Huang, Mu-Jung Tsai, and Hsi-Ya Chang. *Moldable Job Scheduling for HPC as a Service*, pages 43–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[HJS00] M.D. Hill, N.P. Jouppi, and G. Sohi. *Readings in Computer Architecture*. The Morgan Kaufmann Series in Computer Architecture and Design Series. Elsevier Science & Technology Books, 2000.

[HJS+04] Ligang He, Stephen A Jarvis, Daniel P Spooner, Xinuo Chen, and Graham R Nudd. Hybrid performance-oriented scheduling of moldable jobs with qos demands in multiclusters and grids. In *Grid and Cooperative Computing-GCC 2004*, pages 217–224. Springer, 2004.

[HK16] John Hopcroft and Ravindran Kannan. Foundations of data science. Book Draft, 2016.

[HL05] Willy Herroelen and Roel Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of operational research*, 165(2):289–306, 2005.

[HMC06] David Heckerman, Christopher Meek, and Gregory Cooper. A bayesian approach to causal discovery. In *Innovations in Machine Learning*, pages 1–28. Springer, 2006.

[HMW95] David Heckerman, Abe Mamdani, and Michael P Wellman. Real-world applications of bayesian networks. *Communications of the ACM*, 38(3):24–26, 1995.

[HO03] John N Hooker and Greger Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

[Hoo05a] J. N. Hooker. A hybrid method for the planning and scheduling. *Constraints*, 10(4):385–401, 2005.

[Hoo05b] John N Hooker. Planning and scheduling to minimize tardiness. In *International Conference on Principles and Practice of Constraint Programming*, pages 314–327. Springer, 2005.

[Hoo07] John N Hooker. Planning and scheduling by logic-based benders decomposition. *Operations Research*, 55(3):588–602, 2007.

[Hop82] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[HR76] Laurent Hyafil and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[HS90] Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence*, 12:993–1001, 1990.

[HS99] Geoffrey E Hinton and Terrence Joseph Sejnowski. *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

[HSC09] Kuo-Chan Huang, Po-Chi Shih, and Yeh-Ching Chung. Adaptive processor allocation for moldable jobs in computational grid. *International Journal of Grid and High Performance Computing (IJGHPC)*, 1(1):10–21, 2009.

[HT89] A.R. Hoffman and J.F. Traub. *Supercomputers: directions in technology and applications*. National Academy Press, 1989.

[IGN14] Santiago Iturriaga, Sebastián García, and Sergio Nesmachnow. An empirical study of the robustness of energy-aware schedulers for high performance computing systems under uncertainty. In *Latin American High Performance Computing Conference*, pages 143–157. Springer, 2014.

[IK77] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, April 1977.

[IPI+15] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, Masaaki Kondo, and Ikuo Miyoshi. 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*, SC '15, pages 78:1–78:12, New York, NY, USA, 2015. ACM.

[Jac55] J.R. Jackson. *Scheduling a production line to minimize maximum tardiness*. Research report. Office of Technical Services, 1955.

[Jay57] Edwin T Jaynes. Information theory and statistical mechanics. *Physical review*, 106(4):620, 1957.

[Jen96] Finn V Jensen. *An introduction to Bayesian networks*, volume 210. UCL press London, 1996.

[Jos97] Rajani R Joshi. A new heuristic algorithm for probabilistic optimization. *Computers & operations research*, 24(7):687–697, 1997.

[JPJ09] Mansoor Zolghadri Jahromi, Elham Parvinnia, and Robert John. A method of learning weighted similarity function to improve the performance of nearest neighbor. *Information sciences*, 179(17):2964–2973, 2009.

[JS02] Martin Szummer Tommi Jaakkola and Martin Szummer. Partially labeled classification with markov random walks. *Advances in neural information processing systems (NIPS)*, 14:945–952, 2002.

[KA96] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 7(5):506–521, 1996.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381 – 422, 1999.

[KAB+00] Ricky A Kendall, Edoardo Aprà, David E Bernholdt, Eric J Bylaska, Michel Dupuis, George I Fann, Robert J Harrison, Jialin Ju, Jeffrey A Nichols, Jarek Nieplocha, et al. High performance computational chemistry: An overview of nwchem a distributed parallel application. *Computer Physics Communications*, 128(1):260–283, 2000.

[KAJ+12] Laxmikant V. Kale, Anshu Arya, Nikhil Jain, Akhil Langer, Jonathan Lifflander, Harshitha Menon, Xiang Ni, Yanhua Sun, Ehsan Totoni, Ramprasad Venkataraman, and Lukasz Wesolowski.

Charm++ migratable objects + active messages + adaptive runtime productivity + performance, 2012.

[Kal93] L. V. Kale. Parallel programming with charm: An overview. Technical report, 1993.

[KAYT90] Takashi Kimoto, Kazuo Asakawa, Morio Yoda, and Masakazu Takeoka. Stock market prediction system with modular neural networks. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 1–6. IEEE, 1990.

[KB05] George Katsirelos and F Bacchus. Generalized nogoods in CSPs. *Aaai*, pages 390–396, 2005.

[KBSW11] Jonathan G Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *Annals of the History of Computing, IEEE*, 33(3):46–54, 2011.

[KCJ01] Miroslav Kubat and Martin Cooperson Jr. A reduction technique for nearest-neighbor classification: Small groups of examples. *Intelligent Data Analysis*, 5(6):463–476, 2001.

[KFL05] N. Kappiah, Vincent W. Freeh, and D.K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 33–33, Nov 2005.

[KH06] Jan Kelbel and Z Hanzálek. A case study on earliness/tardiness scheduling by constraint programming. *... Conference on Principles and Practice of ...*, 2006.

[Kha11] Mohammad A Khaleel. Scientific grand challenges: Crosscutting technologies for computing at the exascale-february 2-4, 2010, washington, dc. Technical report, Pacific Northwest National Laboratory (PNNL), Richland, WA (US), 2011.

[KHO+16] Pramod Kumbhar, Michael Hines, Aleksandr Ovcharenko, Damian A. Mallon, James King, Florentino Sainz, Felix Schürmann, and Fabien Delalondre. *Leveraging a Cluster-Booster Architecture for Brain-Scale Simulations*, pages 363–380. Springer International Publishing, Cham, 2016.

[KJ99] Ludmila I Kuncheva and Lakhmi C Jain. Nearest neighbor classifier: simultaneous editing and feature selection. *Pattern recognition letters*, 20(11):1149–1156, 1999.

[KJCP14] Minjoong Kim, Yoondeok Ju, Jinseok Chae, and Moonju Park. A simple model for estimating power consumption of a multicore server system. *International Journal of Multimedia and Ubiquitous Engineering*, 9(2):153–160, 2014.

[Kle00] Robert Klein. Bidirectional planning: improving priority rule-based heuristics for scheduling resource-constrained projects. *European Journal of Operational Research*, 127(3):619 – 638, 2000.

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[KMI+15] Memoona Khanum, Tahira Mahboob, Warda Imtiaz, Humaraia Abdul Ghafoor, and Rabeea Sehar. A survey on unsupervised machine learning algorithms for automation, classification and maintenance. *International Journal of Computer Applications*, 119(13), 2015.

[Kol96] Rainer Kolisch. Efficient priority rules for the resource-constrained project scheduling problem. *Journal of Operations Management*, 14(3):179 – 192, 1996.

[KP00] Mark A Kon and Leszek Plaskota. Information complexity of neural networks. *Neural Networks*, 13(3):365–375, 2000.

[KQC13] Dhireesha Kudithipudi, Qinru Qu, and AyseK. Coskun. Thermal management in many core systems. In Samee Ullah Khan, Joanna Koodziej, Juan Li, and Albert Y. Zomaya, editors, *Evolutionary Based Solutions for Green Computing*, volume 432 of *Studies in Computational Intelligence*, pages 161–185. Springer Berlin Heidelberg, 2013.

[KR13] Peter Kogge and David R. Resnick. *Yearly update: exascale projections for 2013.* Oct 2013.

[KRA12a] Jungsoo Kim, M. Ruggiero, and D. Atienza. Free cooling-aware dynamic power management for green datacenters. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 140–146, July 2012.

[KRA12b] Jungsoo Kim, M. Ruggiero, and D. Atienza. Free cooling-aware dynamic power management for green datacenters. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 140–146, July 2012.

[KRA12c] Jungsoo Kim, M. Ruggiero, and D. Atienza. Free cooling-aware dynamic power management for green datacenters. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 140–146, July 2012.

[KRAL13] Jungsoo Kim, Martino Ruggiero, David Atienza, and Marcel Lederberger. Correlation-aware virtual machine allocation for energy-efficient datacenters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1345–1350, San Jose, CA, USA, 2013. EDA Consortium.

[Kru64] Joseph B Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[KSS+05] Rajkumar Kettimuthu, Vijay Subramani, Srividya Srinivasan, Thiagaraja Gopalsamy, Dhabaleswar K Panda, and P Sadayappan. Selective preemption strategies for parallel job scheduling. *International Journal of High Performance Computing and Networking*, 3(2-3):122–152, 2005.

[KVIY01] Mahmut Kandemir, N. Vijaykrishnan, Mary Jane Irwin, and Wu Ye. Influence of compiler optimizations on system power. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(6):801–804, December 2001.

[KW59] James E. Kelley, Jr and Morgan R. Walker. Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 160–173, New York, NY, USA, 1959. ACM.

[KZA+12] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M Tullsen, and T Simunic Rosing. Managing distributed ups energy for effective power capping in data centers. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pages 488–499. IEEE, 2012.

[KZP07] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques, 2007.

[Lab05] Philippe Laborie. Complete MCS-based search: Application to resource constrained project scheduling. *IJCAI*, 2005.

[Lab09a] P. Laborie. IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In *Proc. of CPAIOR*, pages 148–162, 2009.

[Lab09b] Philippe Laborie. Ibm ilog cp optimizer for detailed scheduling illustrated on three problems. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR '09, pages 148–162, Berlin, Heidelberg, 2009. Springer-Verlag.

[Lab14] Philippe Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. In *Sixth European Conference on Planning*, 2014.

[Lam14] Bernard Lampard. *Program Scheduling and Simulation in an Operating System Environment.* GRIN Verlag, USA, 2014.

[LBD+89] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[LBOM98] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[LC91] Keqin Li and K-H Cheng. Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):413–422, 1991.

[LCG+14] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. *SIGARCH Comput. Archit. News*, 42(3):301–312, June 2014.

[LDKeA15] A. Langer, H. Dokania, L.V. Kale, and et Al. Analyzing energy-time tradeoff in power overprovisioned hpc data centers. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 849–854, May 2015.

[LEE92] Pierre Lopez, Jacques Erschler, and Patrick Esquirol. Ordonnancement de tâches sous contraintes: une approche énergétique. *Automatique-productique informatique industrielle*, 26(5-6):453–481, 1992.

[Len97] Jan K Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 1997.

[Leu07] K Ming Leung. Naive bayesian classifier. *Polytechnic University Department of Computer Science/Finance and Risk Engineering*, 2007.

[LF03] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105 – 1122, 2003.

[LFL06] Min Yeol Lim, Vincent W Freeh, and David K Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *SC 2006 conference, proceedings of the ACM/IEEE*, pages 14–14. IEEE, 2006.

[LG] Philippe Laborie and Daniel Godard. Self-Adapting Large Neighborhood Search : Application to single-mode scheduling problems Self-Adapting Large Neighborhood Search.

[LG95] Philippe Laborie and Malik Ghallab. Planning with sharable resource constraints. In *IJCAI*, volume 95, page 1643, 1995.

[LG00] Steve Lawrence and C Lee Giles. Overfitting and neural networks: conjugate gradient and backpropagation. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 1, pages 114–119. IEEE, 2000.

[LG07] P. Laborie and D. Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. In *Proc. of MISTA*, 2007.

[LGTB97] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[LK90] James Leonard and MA Kramer. Improvement of the backpropagation algorithm for training neural networks. *Computers & Chemical Engineering*, 14(3):337–341, 1990.

[LKB77] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In B.H. Korte P.L. Hammer, E.L. Johnson and G.L. Nemhauser, editors, *Studies in Integer Programming*, volume 1 of *Annals of Discrete Mathematics*, pages 343 – 362. Elsevier, 1977.

[LKL97] Shang-Hung Lin, Sun-Yuan Kung, and Long-Ji Lin. Face recognition/detection by probabilistic decision-based neural network. *IEEE transactions on neural networks*, 8(1):114–132, 1997.

[LMS03] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. *Iterated local search*. Springer, 2003.

[LP94] C. Le Pape. Implementation of resource constraints in ilog schedule: a library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66, Summer 1994.

[LP+05] Claude Le Pape et al. Constraint-based scheduling: A tutorial. 2005.

[LR08] P. Laborie and J. Rogerie. Reasoning with conditional time-intervals. In *Proc. of FLAIRS*, pages 555–560, 2008.

[lsf16] Ibm spectrum lsf. `http://www-03.ibm.com/systems/spectrum-computing/products/lsf/`, 2016. Accessed: 2016-10-19.

[LSP02] Barry G Lawson, Evgenia Smirni, and Daniela Puiu. Self-adapting backfilling scheduling for parallel systems. In *Parallel Processing, 2002. Proceedings. International Conference on*, pages 583–592. IEEE, 2002.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173 – 180, 1993.

[LW02] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[LWW07] C. Lefurgy, X. Wang, and M. Ware. Server-level power control. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 4–4, June 2007.

[LWW08] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, 2008.

[LZ10] Yongpeng Liu and Hong Zhu. A survey of the research on power management techniques for high-performance systems. *Software - Practice and Experience*, 40(11):943–964, 2010.

[MA04] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[Mac03] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[Mad03] M Madden. The performance of bayesian network classifiers constructed using different techniques. In *Proceedings of European conference on machine learning, workshop on probabilistic graphical models for classification*, pages 59–70, 2003.

[Mas05] M. Massoud. *Engineering Thermofluids: Thermodynamics, Fluid Mechanics, and Heat Transfer*. Springer, 2005.

[MBL+15] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. *A Run-Time System for Power-Constrained HPC Applications*, pages 394–408. Springer International Publishing, Cham, 2015.

[MBS01] Muthucumaru Maheswaran, Tracy D. Braun, and Howard Jay Siegel. *Heterogeneous Distributed Computing*. John Wiley and Sons, Inc., 2001.

[MCF+98] Jose E Moreira, Waiman Chan, Liana L Fong, Hubertus Franke, and Morris A Jette. An infrastructure for efficient parallel job execution in terascale computing environments. In *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, pages 50–50. IEEE, 1998.

[McK] 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.

[MCM13] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[MDSV07] KE Maghraoui, Travis J Desell, Boleslaw K Szymanski, and Carlos A Varela. Dynamic malleability in iterative mpi applications. In *7th Int. Symposium on Cluster Computing and the Grid*, pages 591–598, 2007.

[MDVH11] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. Reinforced adaptive large neighborhood search. In *The Seventeenth International Conference on Principles and Practice of Constraint Programming (CP 2011)*, page 55, 2011.

[MDZD09] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[MF01] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, June 2001.

[MGSG12] Yan Ma, Bin Gong, Ryo Sugihara, and Rajesh Gupta. Energy-efficient deadline scheduling for heterogeneous systems. *Journal of Parallel and Distributed Computing*, 72(12):1725 – 1740, 2012.

[MHCD10] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[MHJI07] K. Meng, F. Huebbers, R. Joseph, and Y. Ismail. Modeling and characterizing power variability in multicore architectures. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*, pages 146–157, April 2007.

[Min89] John Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine learning*, 4(2):227–243, 1989.

[Mis90] Ignacy Misztal. Restricted maximum likelihood estimation of variance components in animal model using sparse matrix inversion and a supercomputer. *Journal of dairy science*, 73(1):163–172, 1990.

[Mit97] Thomas M Mitchell. Machine learning. *New York*, 1997.

[MKM15] Vladimir Mironov, Maria Khrenova, and Alexander Moskovsky. *On Quantum Chemistry Code Adaptation for RSC PetaStream Architecture*, pages 113–121. Springer International Publishing, Cham, 2015.

[MMB+12] Olli Mämmelä, Mikko Majanen, Robert Basmadjian, Hermann De Meer, André Giesler, and Willi Homberg. Energy-aware job scheduler for high-performance computing. *Computer Science-Research and Development*, 27(4):265–275, 2012.

[MN+98] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.

[Moo68] J. Michael Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15(1):102–109, 1968.

[mpi16] The message passing interface standard. `http://www.mcs.anl.gov/research/projects/mpi/`, 2016. Accessed: 2016-10-19.

[MSD+10] Jean-Baptiste Mairy, Pierre Schaus, Yves Deville, et al. Generic adaptive heuristics for large neighborhood search. In *7th workshop on local search techniques in constraint satisfaction (LSCS2010)*, volume 36, page 130, 2010.

[Mur06] Kevin P Murphy. Naive bayes classifiers. *University of British Columbia*, 2006.

[MVA99] George D. Magoulas, Michael N. Vrahatis, and George S Androulakis. Improving the convergence of the backpropagation algorithm using learning rate adaptation methods. *Neural Computation*, 11(7):1769–1796, 1999.

[MVZ93] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 11(2):146–178, 1993.

[N+04] Richard E Neapolitan et al. Learning bayesian networks. 2004.

[Nas04] Sani R Nassif. The impact of variability on power. In *International Symposium on Low Power Electronics and Design: Proceedings of the 2004 international symposium on Low power electronics and design*, volume 9, pages 350–350, 2004.

[Nie08] Daryle Niedermayer. An introduction to bayesian networks and their contemporary applications. In *Innovations in Bayesian Networks*, pages 117–130. Springer, 2008.

[Nil65] Nils J Nilsson. Learning machines. 1965.

[NMN+10] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115–122. IEEE, 2010.

[NVZ96] Thu D Nguyen, Raj Vaswani, and John Zahorjan. Parallel application characterization for multiprocessor scheduling policy design. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 175–199. Springer, 1996.

[NW07] Frank Neumann and Ingo Wegener. Randomized local search, evolutionary algorithms, and the minimum spanning tree problem. *Theoretical Computer Science*, 378(1):32–40, 2007.

[OM99] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.

[ORS+10] Catherine Mills Olschanowsky, Tajana Rosing, Allan Snavely, Laura Carrington, Mustafa M Tikir, and Michael Laurenzano. Fine-grained energy consumption characterization and modeling. In *High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2010 DoD*, pages 487–497. IEEE, 2010.

[Ous82] John K Ousterhout. Scheduling techniques for concurrebt systems. In *ICDCS*, volume 82, pages 22–30, 1982.

[OW03] Cristina Olaru and Louis Wehenkel. A complete fuzzy decision tree technique. *Fuzzy sets and systems*, 138(2):221–254, 2003.

[PAM04] Mireille Palpant, Christian Artigues, and Philippe Michelon. LSSPER : Solving the Resource-Constrained Project. (1997):237–257, 2004.

[PBCH01] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems, 2001.

[PCOS07] Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F Smith. From Precedence Constraint Posting to Partial Order Schedules. *Ai Communications*, 20:1–17, 2007.

[PCVG94] C Le Pape, P Couronné, D Vergamini, and V Gosselin. Time-versus-capacity compromises in project scheduling. *In Proc. of the 13th Workshop of the UK Planning Special Interest Group*, pages 1–13, 1994.

[PH11] Dario Pacino and Pascal Van Hentenryck. Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. *International Joint Conference on Artificial ...*, pages 1997–2002, 2011.

[Pin91] E. Pinson. A practical use of jackson's preemptive schedule for solving the job shop problem. *Ann. Oper. Res.*, 26(1-4):269–287, January 1991.

[PLR+13] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 173–182, New York, NY, USA, 2013. ACM.

[PLS+15] Tapasya Patki, David K. Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L. Rountree, Martin Schulz, and Bronis R. de Supinski. Practical resource management in power-constrained, high performance computing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 121–132, New York, NY, USA, 2015. ACM.

[PMW+15] P. Petoumenos, L. Mukhanov, Z. Wang, H. Leather, and D. S. Nikolopoulos. Power capping: What works, what does not. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pages 525–534, Dec 2015.

[POF+15] Kevin Pedretti, Stephen L. Olivier, Kurt B. Ferreira, Galen Shipman, and Wei Shu. Early experiences with node-level power capping on the cray xc40 platform. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*, E2SC '15, pages 1:1–1:10, New York, NY, USA, 2015. ACM.

[PRA] Prace. partnership for advanced computing in europe.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[PSCO04] Nicola Policella, Stephen F Smith, Amedeo Cesta, and Angelo Oddi. Generating robust schedules through temporal flexibility. In *Proceedings of the 14th International Conference on Automated Planning & Scheduling, ICAPS04*, pages 209–218, 2004.

[PSLeA16] S. Pakin, C. Storlie, M. Lang, and et Al. Power usage of production supercomputers and production workloads. *Concurr. Comput. : Pract. Exper.*, 28(2):274–290, February 2016.

[Put14] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[PVGeA11] F. Pedregosa, G. Varoquaux, A. Gramfort, and et Al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[PWDC] A. Petitet, R. C. Whaley, Jack Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.

[Q+92] John R Quinlan et al. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, volume 92, pages 343–348. Singapore, 1992.

[Qui79] J. R. Quinlan. Discovering rules by induction from large collections of examples. 1979.

[Qui86] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[Qui14] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[RÍ11] JC Régin. Global Constraints: a survey. *Hybrid optimization*, 2011.

[RAdS+12] 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*, pages 947–953, May 2012.

[RB93] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference On*, pages 586–591. IEEE, 1993.

[RCC12] Sherief Reda, Ryan Cochran, and Ayse K Coskun. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 32(5):0064–75, 2012.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *Principles and Practice of Constraint Programming CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer Berlin Heidelberg, 2004.

[Rég04] Jean-Charles Régin. Global constraints and filtering algorithms. In *Constraint and Integer Programming*, pages 89–135. Springer, 2004.

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[RHW88] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[Ric97] Thomas Richardson. An introduction to bayesian networks. *Journal of the American Statistical Association*, 92(439):1215–1217, 1997.

[Rip07] Brian D Ripley. *Pattern recognition and neural networks*. Cambridge university press, 2007.

[Ris01] Irina Rish. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46. IBM New York, 2001.

[RLdS+09] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM.

[RLF+07] Barry Rountree, David K Lowenthal, Shelby Funk, Vincent W Freeh, Bronis R De Supinski, and Martin Schulz. Bounding energy consumption in large-scale mpi programs. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 49. ACM, 2007.

[RM05] Lior Rokach and Oded Maimon. Top-down induction of decision trees classifiers-a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005.

[RMS89] A Rajavelu, Mohamad T Musavi, and Mukul Vassant Shirvaikar. A neural network approach to character recognition. *Neural Networks*, 2(5):387–393, 1989.

[RNC+03] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[Ros62] Frank. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, pages 245–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 1962.

[RSB13] H. V. Raghu, S. K. Saurav, and B. S. Bapu. Paas: Power aware algorithm for scheduling in high performance computing. In *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, pages 327–332, Dec 2013.

[Rus78] RM Russell. The CRAY-1 computer system. *Communications of the ACM*, 1978.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[SAÅ+04] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2-3):101–155, 2004.

[SAB+16] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R Beccari, Luca Benini, Radim Cmar, Carlo Cavazzoni, Jan Martinovi, Gianluca Palermo, Martin Palkovi, et al. Autotuning and adaptivity approach for energy efficient exascale hpc systems: the antarex approach. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 708–713. IEEE, 2016.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[SB99] Bernhard Schölkopf and Christopher JC Burges. *Advances in kernel methods: support vector learning*. MIT press, 1999.

[SB16] Alina Sîrbu and Ozalp Babaoglu. *Power Consumption Modeling and Prediction in a Hybrid CPU-GPU-MIC Supercomputer*, pages 117–130. Springer International Publishing, Cham, 2016.

[SBB+92] Eduard Säckinger, Bernhard E Boser, Jane M Bromley, Yann LeCun, and Larry D Jackel. Application of the anna neural network chip to high-speed character recognition. *IEEE Transactions on Neural Networks*, 3(3):498–505, 1992.

[SBM09] K. Singh, M. Bhadauria, and S.A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, July 2009.

[Sch90] Robert E Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[Sch96] Uwe Schwiegeishohn. Preemptive weighted completion time scheduling of parallel jobs. In *European Symposium on Algorithms*, pages 39–51. Springer, 1996.

[SCVH12] Ben Simon, Carleton Coffrin, and Pascal Van Hentenryck. *Randomized adaptive vehicle decomposition for large-scale power restoration*. Springer, 2012.

[SDA96] Howard Jay Siegel, Henry G Dietz, and John K Antonio. Software support for heterogeneous computing. *ACM Computing Surveys (CSUR)*, 28(1):237–239, 1996.

[SDB97] Christian Schittenkopf, Gustavo Deco, and Wilfried Brauer. Two strategies to avoid overfitting in feedforward networks. *Neural networks*, 10(3):505–516, 1997.

[See00] Matthias Seeger. Learning with labeled and unlabeled data. Technical report, 2000.

[SF81] R Short and Keinosuke Fukunaga. The optimal distance measure for nearest neighbor classification. *IEEE transactions on Information Theory*, 27(5):622–627, 1981.

[SF03] Edi Shmueli and Dror G. Feitelson. *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper*, chapter Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling, pages 228–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[SF09] Edi Shmueli and Dror G. Feitelson. On simulation and design of parallel-systems schedulers: Are we doing the right thing? *IEEE Trans. Parallel Distrib. Syst.*, 20(7):983–996, July 2009.

[SF13] Balaji Subramaniam and Wu-chun Feng. Towards Energy-Proportional Computing for Enterprise-Class Server Workloads. In *3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, Prague, Czech Republic, April 2013. Best Paper Award.

[SFBL97] Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods, 1997.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998.

[Sim96] H Simonis. A problem classification scheme for finite domain constraint solving. In *Proceeding of workshop on constraint applications, CP96, Boston*. Citeseer, 1996.

[Sim12] Horst D Simon. Barriers to exascale computing. In *International Conference on High Performance Computing for Computational Science*, pages 1–3. Springer, 2012.

[SK01] W Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 377–382. ACM, 2001.

[SKD95] Arno Sprecher, Rainer Kolisch, and Andreas Drexl. Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 80(1):94 – 102, 1995.

[SKS03] Sudha Srinivasan, Sriram Krishnamoorthy, and P Sadayappan. A robust scheduling technology for moldable scheduling of parallel jobs. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 92–99. IEEE, 2003.

[SL90] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. 1990.

[SL93] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):175–187, February 1993.

[SLGK14] 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*, SC '14, pages 807–818, Piscataway, NJ, USA, 2014. IEEE Press.

[SLS06] Gerald Sabin, Matthew Lang, and P Sadayappan. Moldable parallel job scheduling using job efficiency: an iterative approach. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 94–114. Springer, 2006.

[SMB+02] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H Albonesi, Sandhya Dwarkadas, and Michael L Scott.

Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 29–40. IEEE, 2002.

[SML+07] Ozan Sonmez, Hashim Mohamed, Wouter Lammers, Dick Epema, et al. Scheduling malleable applications in multicluster systems. In *2007 IEEE International Conference on Cluster Computing*, pages 372–381. IEEE, 2007.

[SS77] Richard M Stallman and Gerald J Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, 9(2):135–196, 1977.

[SS04] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.

[SSK+02] Srividya Srinivasan, Vijay Subramani, Rajkumar Kettimuthu, Praveen Holenarsipur, and P Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *International Conference on High-Performance Computing*, pages 174–183. Springer, 2002.

[SSM+13] M. M. Sabry, A. Sridhar, J. Meng, A. K. Coskun, and D. Atienza. Greencool:an energy-efficient liquid cooling design technique for 3d mpsocs via channel width modulation. *IEEE Trans. on TCAD*, pages 524–537, 2013.

[SSPeA14] C. Storlie, J. Sexton, S. Pakin, and et Al. Modeling and predicting power consumption of high performance computing jobs. *arXiv preprint arXiv:1412.5247*, 2014.

[SSSF13] Balaji Subramaniam, Winston Saunders, Tom Scogland, and Wu-chun Feng. Trends in Energy-Efficient Computing: A Perspective from the Green500. In *4th International Green Computing Conference*, Arlington, VA, June 2013.

[SSWeA14] T.R.W. Scogland, C.P. Steffen, T. Wilde, and et Al. A power-measurement methodology for large-scale, high-performance computing. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 149–159, New York, NY, USA, 2014. ACM.

[ST03] John S. Seng and Dean M. Tullsen. The effect of compiler optimizations on pentium 4 power consumption. In *Proceedings of the Seventh Workshop on Interaction Between Compilers and Computer Architectures*, INTERACT '03, pages 51–, Washington, DC, USA, 2003. IEEE Computer Society.

[Sta06] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[Str03] Achim Streit. *Self-tuning job scheduling strategies for the resource management of HPC systems and computational grids.* PhD thesis, University of Paderborn, 2003.

[Stu] R Scott Studham. Hpc workload characterization.

[Sut92] Richard S Sutton. Introduction: The challenge of reinforcement learning. In *Reinforcement Learning*, pages 1–3. Springer, 1992.

[SV94] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(02):187–207, 1994.

[SWAB14] H. Shoukourian, T. Wilde, A. Auweter, and A. Bode. Predicting the energy and power consumption of strong and weak scaling hpc applications. *Supercomputing frontiers and innovations*, 1(2):20–41, 2014.

[SWAB15] H. Shoukourian, T. Wilde, A. Auweter, and A. Bode. Power variation aware configuration adviser for scalable hpc schedulers. In *High Performance Computing Simulation (HPCS), 2015 International Conference on*, pages 71–79, July 2015.

[SY00] Uwe Schwiegelshohn and Ramin Yahyapour. Fairness in parallel job scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[TB12] Tony T Tran and J Christopher Beck. Logic-based benders decomposition for alternative resource scheduling with sequence dependent setups. In *ECAI*, pages 774–779, 2012.

[TC12] Matthew Tolentino and Kirk W Cameron. The optimist, the pessimist, and the global race to exascale in 20 megawatts. *Computer*, 45(1):0095–97, 2012.

[TCWX09] Songbo Tan, Xueqi Cheng, Yuefen Wang, and Hongbo Xu. Adapting naive bayes to domain adaptation for sentiment analysis. In *European Conference on Information Retrieval*, pages 337–349. Springer, 2009.

[TDM11] Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel. Accurate mutlicore processor power models for power-aware resource management. In *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, DASC '11, pages 419–426, Washington, DC, USA, 2011. IEEE Computer Society.

[TEF07] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):789–803, June 2007.

[TG03] Aik Choon Tan and David Gilbert. Ensemble machine learning on gene expression data for cancer classification. 2003.

[THS02] Haluk Topcuoglu, Salim Hariri, and Ieee Computer Society. Performance-Effective and Low-Complexity. 13(3):260–274, 2002.

[TL00] Sebastian Thrun and Michael L Littman. Reinforcement learning: an introduction. *AI Magazine*, 21(1):103–103, 2000.

[TLL95] Igor V Tetko, David J Livingstone, and Alexander I Luik. Neural network studies. 1. comparison of overfitting and overtraining. *Journal of chemical information and computer sciences*, 35(5):826–833, 1995.

[TTDB13] TT Tran, Daria Terekhov, DG Down, and JC Beck. Hybrid queueing theory and scheduling models for dynamic environments with sequence-dependent setup times. *... Scheduling (ICAPS 2013), To ...*, 2013.

[UCL04] Gladys Utrera, Julita Corbalan, and Jesus Labarta. Implementing malleability on mpi jobs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 215–224, Washington, DC, USA, 2004. IEEE Computer Society.

[Ull75] J.D. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[VAG10] Georgios Varsamopoulos, Zahra Abbasi, and Sandeep KS Gupta. Trends and effects of energy proportionality on server provisioning in data centers. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–11. IEEE, 2010.

[VAN08] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 175–184. ACM, 2008.

[Vap13] Vladimir Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 2013.

[VB06] Peter Van Beek. Backtracking search algorithms. *Handbook of constraint programming*, pages 85–134, 2006.

[VCC+99] Konstantinos Veropoulos, Colin Campbell, Nello Cristianini, et al. Controlling the sensitivity of support vector machines. In *Proceedings of the international joint conference on AI*, pages 55–60, 1999.

[Ves07] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239–243. IEEE, 2007.

[VFHB14] Erik Vermij, Leandro Fiorin, Christoph Hagleitner, and Koen Bertels. *Exascale Radio Astronomy: Can We Ride the Technology Wave?*, pages 35–52. Springer International Publishing, Cham, 2014.

[VG10] Georgios Varsamopoulos and Sandeep KS Gupta. Energy proportionality and the future: Metrics and directions. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 461–467. IEEE, 2010.

[vHK06] Willem-Jan van Hoeve and Irit Katriel. Global constraints. *Handbook of constraint programming*, pages 169–208, 2006.

[vHM04] Willem Jan van Hoeve and Michela Milano. Decomposition based search - A theoretical and experimental evaluation. *CoRR*, cs.AI/0407040, 2004.

[VK82] Vladimir Naumovich Vapnik and Samuel Kotz. *Estimation of dependences based on empirical data*, volume 40. Springer-Verlag New York, 1982.

[VON92] Arjen Van Ooyen and Bernard Nienhuis. Improving the convergence of the back-propagation algorithm. *Neural Networks*, 5(3):465–471, 1992.

[WA02] Michael Widenius and Davis Axmark. *Mysql Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[Wal99] Toby Walsh. Search in a small world. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, pages 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[WAM97] Dietrich Wettschereck, David W Aha, and Takao Mohri. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11(1-5):273–314, 1997.

[WAS14] Torsten Wilde, Axel Auweter, and Hayk Shoukourian. The 4 pillar framework for energy efficient hpc data centers. *Comput. Sci.*, 29(3-4):241–251, August 2014.

[WASB15] Torsten Wilde, Axel Auweter, Hayk Shoukourian, and Arndt Bode. *Taking Advantage of Node Power Variation in Homogenous HPC Systems to Save Energy*, pages 376–393. Springer International Publishing, Cham, 2015.

[WC08] X. Wang and M. Chen. Cluster-level feedback power control for performance optimization. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 101–110, Feb 2008.

[WM97] D Randall Wilson and Tony R Martinez. Instance pruning techniques. In *ICML*, volume 97, pages 403–411, 1997.

[WM00] D Randall Wilson and Tony R Martinez. Reduction techniques for instance-based learning algorithms. *Machine learning*, 38(3):257–286, 2000.

[WMES09] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Hybrid full/incremental checkpoint/restart for mpi jobs in hpc environments. In *Dept. of Computer Science, North Carolina State University*, 2009.

[WMES10] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. Hybrid checkpointing for mpi jobs in hpc environments. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 524–533. IEEE, 2010.

[WMOSI12] Philippe Weber, Gabriela Medina-Oliva, Christophe Simon, and Benoît Iung. Overview on bayesian networks applications for dependability, risk analysis and maintenance areas. *Engineering Applications of Artificial Intelligence*, 25(4):671–682, 2012.

[WNC06] Jigang Wang, Predrag Neskovic, and Leon N Cooper. Neighborhood size selection in the k-nearest-neighbor rule using statistical confidence. *Pattern Recognition*, 39(3):417–423, 2006.

[WOPW13] Michal Witkowski, Ariel Oleksiak, Tomasz Piontek, and J Weglarz. Practical power consumption estimation for real life hpc applications. *Future Generation Computer Systems*, 29(1):208–217, 2013.

[Wor15] Altair PBS Works. Pbs professional®13.1 administrator's guide. http://www.pbsworks.com/pdfs/PBSProAdminGuide13.1.pdf, 2015.

[WS09] Kilian Q Weinberger and Lawrence K Saul. Distance metric learning for large margin nearest neighbor classification. *Journal of Machine Learning Research*, 10(Feb):207–244, 2009.

[WTJ+15] Song Wu, Qiong Tuo, Hai Jin, Chuxiong Yan, and Qizheng Weng. Hrf: A resource allocation scheme for moldable jobs. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 17:1–17:8, New York, NY, USA, 2015. ACM.

[WTT+04] X.G. Wang, Z. Tang, H. Tamura, M. Ishii, and W.D. Sun. An improved backpropagation algorithm to avoid the local minima problem. *Neurocomputing*, 56:455 – 460, 2004.

[WW96] Yong Wang and Ian H Witten. Induction of model trees for predicting continuous classes. 1996.

[WYV+16] Sean Wallace, Xu Yang, Venkatram Vishwanath, William E. Allcock, Susan Coghlan, Michael E. Papka, and Zhiling Lan. A data driven scheduling approach for power management on hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 56:1–56:11, Piscataway, NJ, USA, 2016. IEEE Press.

[XWVA05] Min Xu, Pakorn Watanachaturaporn, Pramod K Varshney, and Manoj K Arora. Decision tree regression for soft classification of remote sensing data. *Remote Sensing of Environment*, 97(3):322–336, 2005.

[YL04] Lei Yu and Huan Liu. Efficient feature selection via analysis of relevance and redundancy. *J. Mach. Learn. Res.*, 5:1205–1224, December 2004.

[YM13] Jumie Yuventi and Roshan Mehdizadeh. A critical analysis of power usage effectiveness and its use in communicating data center energy consumption. *Energy and Buildings*, 64:90–94, 2013.

[YMO07] D.F. Young, B. R. Munson, and T. H. Okiishi. *A Brief Introduction to Fluid Mechanics*. John Wiley and Sons LTd, 2007.

[YWZL14] Yulai Yuan, Yongwei Wu, Weimin Zheng, and Keqin Li. Guarantee strict fairness and utilizeprediction better in parallel job scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 25(4):971–981, April 2014.

[YZ13a] Haihang You and Hao Zhang. Comprehensive workload analysis and modeling of a petascale supercomputer. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 7698 of *Lecture Notes in Computer Science*, pages 253–271. Springer Berlin Heidelberg, 2013.

[YZ13b] Haihang You and Hao Zhang. *Job Scheduling Strategies for Parallel Processing: 16th International Workshop, JSSPP 2012, Shanghai, China, May 25, 2012. Revised Selected Papers*, chapter Comprehensive Workload Analysis and Modeling of a Petascale Supercomputer, pages 253–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[YZW+13] Xu Yang, Zhou Zhou, Sean Wallace, Zhiling Lan, Wei Tang, Susan Coghlan, and Michael E. Papka. Integrating dynamic pricing of electricity into energy aware scheduling for hpc systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 60:1–60:11, New York, NY, USA, 2013. ACM.

[ZGL+03] Xiaojin Zhu, Zoubin Ghahramani, John Lafferty, et al. Semi-supervised learning using gaussian fields and harmonic functions. In *ICML*, volume 3, pages 912–919, 2003.

[ZLTD13] Zhou Zhou, Zhiling Lan, Wei Tang, and Narayan Desai. Reducing energy costs for ibm blue gene/p via power-aware job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 96–115. Springer, 2013.

[ZM12] Cha Zhang and Yunqian Ma. *Ensemble machine learning*. Springer, 2012.

[ZRS87] Wei Zhao, Krithi Ramamritham, and John A Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on computers*, 100(8):949–960, 1987.

[ZWT02] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: many could be better than all. *Artificial intelligence*, 137(1):239–263, 2002.