A Constraint Programming Scheduler for Heterogeneous High-Performance Computing Machines

A Constraint Programming Scheduler for Heterogeneous High-Performance Computing Machines

Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, Luca Benini

Abstract—Scheduling and dispatching tools for High-Performance Computing (HPC) machines have the key role of mapping jobs to the available resources, trying to maximize performance and Quality-of-Service (QoS). Allocation and Scheduling in the general case are well-known NP-hard problems, forcing commercial schedulers to adopt greedy approaches to improve performance and QoS. Search-based approaches featuring the exploration of the solution space have seldom been employed in this setting, but mostly applied in off-line scenarios. In this paper, we present the first search-based approach to job allocation and scheduling for HPC machines, working in a production environment.

The scheduler is based on Constraint Programming, an effective programming technique for optimization problems. The resulting scheduler is flexible, as it can be easily customized for dealing with heterogeneous resources, user-defined constraints and different metrics.

We evaluate our solution both on virtual machines using synthetic workloads, and on the Eurora HPC with production workloads. Tests on a wide range of operating conditions show significant improvements in waitings and QoS in mid-tier HPC machines w.r.t state-of-the-art commercial rule-based dispatchers.

Furthermore, we analyze the conditions under which our approach outperforms commercial approaches, to create a portfolio of scheduling algorithms that ensures robustness, flexibility and scalability.

Index Terms—Constraint Programming, Optimization, HPC, Scheduling, Resource allocation, Supercomputer.

1 Introduction

High-performance computing centers are investment-intensive facilities with short depreciation cycles. An average supercomputer reaches full depreciation in three to five years [1]. Hence their utilization has to be aggressively managed to produce an acceptable return on investment. Even relatively small improvements in utilization, throughput, and quality of service translate in significant financial gains.

A key role in this challenge is played by scheduling software that decides where and when a job has to execute. Users submit jobs to supercomputing machines specifying the amount of required resources (CPUs, GPUs, memory) and the maximum expected execution time (wall-time). In general, different "job queues" are available in HPC machines managing, for example, jobs featuring different priorities, execution time and user-requirements.

Commercial scheduling software (like PBS Professional [2], Torque [3], and Slurm [4]) can be configured via a set of rules managing the priorities of waiting jobs. These priority-rule-based algorithms are simple and reasonably fast, but the resource allocation and schedules found can be considerably improved in terms of job waiting time and QoS.

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

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

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

In parallel, we evaluate the impact of this optimization goal on other performance metrics such as late jobs, user

- *T. Bridi, M. Lombardi and M. Milano are with DISI, University of Bologna. Viale Risorgimento 2, 40123, Bologna, Italy. E-mail: {thomas.bridi, michele.lombardi2, michela.milano}@unibo.it.*
- *A. Bartolini and L. Benini are with DEI, University of Bologna. Viale Risorgimento 2, 40123, Bologna, Italy. E-mail: {a.bartolini, luca.benini}@unibo.it.*
- *A. Bartolini and L. Benini are with the Integrated Systems Laboratory at ETH Zurich, Switzerland. E-mail: {barandre, luca.benini}@iis.ee.ethz.ch.*

QoS and scheduling overhead. The model extends the one in the work of Bartolini et al. [5], to account for multiple classes of jobs and their temporal dependencies. In addition, the solution space exploration strategies have been optimized for on-line use, taking into account the impact of the schedule computation time on machine utilization.

The CP solver has been embedded, as a plug-in, in the software framework of PBS Professional [2], a well-known commercial HPC scheduler, by replacing its rule-based scheduling engine. By linking our solver with a state-of-the-art HPC scheduling tool, we have been able to validate our approach on a real-life HPC machine, Eurora, from "Consorzio INtEruniversitario per il Calcolo Automatico" (CINECA). Eurora is a fully operational prototype of direct-liquid cooled HPC machine for future Tier 0 and energy aware HPC. It achieved the top GREEN500 ranking in June 2013 and has been used for production runs since 2013.

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

Simulated tests on Eurora-sized instances obtain average improvements of 21% on the waiting time of jobs and 22% on late jobs, although we introduce an overhead for computing higher quality solutions with respect to PBS that is 20 times higher. However, the overhead has a negligible impact on the job execution time: in our tests the worst case maximum-overhead over average-walltime ratio registered is only 5,26%. Experiments in a real production environment achieved an average improvement on job waiting times of 29% while maintaining the same average machine utilization.

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

The paper is organized as follows: we start discussing related work in section 2 then we formally define the HPC scheduling problem considered in this paper in section 3. Section 4 provides some insights on Constraint Programming, the declarative programming paradigm used to model and solve the problem. Section 5 describes the optimization model and all the features implemented to make it usable on a real HPC center. Section 6 gives an overview of PBS Professional and the embedding of our scheduler in its framework. Overhead reduction techniques are also discussed here. In section 7 we show results on synthetic and real settings and we make statistics on the computational overhead.

2 Related work

The problem of batch scheduling is well-known and widely investigated. The interested reader can refer to the work of Salot [6] for a good survey on scheduling algorithms used in HPC and computing clusters. Most of the algorithms described in this work can be implemented within commercial scheduling software by defining appropriate "scheduling rules" (e.g., the min-min algorithm can be implemented sorting jobs by increasing amount of required resources). In the works presented by Feitelson [7] and Alem and Feitelson [8], a study on performances of two different backfilling algorithms can be found: the study evaluates conservative backfilling versus EASY backfilling providing guidelines on their potential selection.

In general, the large majority of existing approaches have a greedy component: the proposed heuristic does not explore the solutions space and generates a "good" solution. Neither local nor global optimality can be achieved. Focusing on search-based schedulers, it is hard to find in the literature examples of optimization algorithms applied to a real in-production HPC scheduler. Sarood et al. [9] show an ILP model to constrain the power usage within the resource manager. This work is based on assumptions that do not hold in general for HPC workloads. For example, it proposes to improve the overall execution time by increasing/decreasing the number of nodes used by a job even during its execution. This is not possible in many HPC production environments where resources are locked to the job for its entire duration. In addition, the experiments in the work are made only by simulation on trace-log on a system that is smaller than current HPC standards.

In a wider context, there is a large body of literature on scheduling and allocation for data-center workloads [10] [11] [12] [13] relying on the key assumption that partial or complete migration of parallel jobs is possible during their execution. Even though supercomputers will reasonably move toward more agile execution models [14] [15], the common practice today is that job migration is not allowed, to maximize performance and predictability [16].

In the work by Soner et al. [17] we find another example of optimization in scheduling. The proposed solution always schedules jobs in arrival order and models job dispatching as an assignment problem. Differently from the approach described in the following sections, Soner et al. do not consider the very significant optimization opportunities that emerge when jobs can be extracted from queues in non-FIFO order.

An interesting approach can be found in the work of Kessaci et al. [18]. This is a meta-scheduler that uses multi-objective genetic algorithms to decide in which data center of a grid to send jobs, in order to optimize CO_2 emissions, energy consumption and profits providing a set of Pareto solutions. This work differs from the present one for the assumption behind the model: the authors consider the presence of *hard*-deadline for the jobs and one job can be dispatched to only one node using a FIFO policy. In our case study *hard*-deadlines are not considered and each job can request more than one node.

In the works presented by Wang and Raicu [19] and in the work presented by Jones and Nitzberg [20] some interesting studies on schedulers performance and scalability are described: different infrastructure setups and greedy algorithms are compared to scale to larger scale HPC machines.

To the best of our knowledge, the only examples that apply optimization techniques to a scheduler in a pro-

duction context are presented by Klusáček et al. [21] and Chlumsky et al. [22]. In these papers, the authors present an optimization technique applied to a scheduler. The second is developed as an extension of the open-source TORQUE scheduler. This extension replaces the scheduling core of the framework with a backfilling-like algorithm that inserts one job at a time into the schedule starting from a previous solution and then applies a Tabu Search to optimize the solution. Both these works use Tabu search to explore a number of local optimal solutions and consider a job as a set of resources. This assumption drastically decreases the flexibility of the scheduler by avoiding the possibility for a job to request more than one node. In our work we consider jobs requiring a set of resources. In this way we maintain the flexibility of commercial schedulers (like TORQUE and PBS Professional) but we deal with a more complex problem w.r.t. the work of Chlumsky.

The work presented by Yuan et al. [23] show a new version of the EASY backfilling algorithm to take into account fairness. As for the main scheduling algorithm for HPC, this is a greedy algorithm and does not explore solutions to get a local optimum. However, they propose an interesting concept of fairness that is achieved when a job start time is not delayed by a lower-priority job. This concept could lead to starvation. In our work, we propose a different concept of fairness where the job waiting times have to be distributed on the basis of the ratio between job priorities.

Another example of user-aware scheduling can be found in the work of Shmueli and Feitelson [24]. This work prioritizes jobs by the estimated response time and the seniority factor (minutes of waiting of the job). Then it applies the EASY backfilling algorithm. However, this greedy algorithm does not guarantee optimality of the solution obtained as it does not consider the resource required from jobs and their wall-times.

The work presented by Shmueli and Feitelson [25] show an interesting approach for the optimization of the backfilling algorithm. This approach exploits dynamic programming to improve results obtained by the classical backfilling algorithm to maximize the system utilization. However, the author considers only the case of one type resource, neither different kind of resources nor heterogeneous resources are considered, and a comparison with this work cannot be done.

The work presented by Tsafrir et al. [26] focus on the execution-time prediction. The suggested technique uses the last two jobs execution from the same user to predict the job execution-time. A key point of the approach is that this prediction is used only for the scheduling and it does not substitute the job's walltime. This approach is shown to be lightweight and efficient, and differently from other approach does not expose users to the risk of premature job killing. The authors state that this approach can be added to every classical backfilling scheduler, but this approach can profitably be added even to more complex scheduler like ours. However, the focus of our work is on the scheduling algorithm. For this reason, we will investigate the behavior of this techniques applied to our CP scheduler in future works.

Our contribution is a complete optimization model which can be applied to a real HPC system. Differently from other optimization approaches, we evaluate both the dispatching and the scheduling performance. In addition, our approach enables a controlled trade-off between schedule computation time and optimality.

3 The HPC Scheduling problem

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

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

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

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

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

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

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

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

job is neither prime-time nor non-prime-time it is an "anytime" job.

4 Constraint Programming

Constraint Programming is a declarative programming paradigm [27] particularly suitable for solving constraint satisfaction and optimization problems. A constraint program is defined on a set of decision variables, each ranging on a discrete domain of values that the variable can assume, and a set of constraints limiting the combination of variable-value assignments. For example, decision variable x ranging on the domain $[1..10]$, written as $x :: [1..10]$, means that variable x can be assigned to one (integer) value between 1 and 10.

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

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

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

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

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

The algorithm adopted by the solver used in this work is the "Self-Adapting Large Neighborhood Search". The complexity of this algorithm is exponential within the number of decisional variables. In our case the number of decisional variables is $n + Nn * \sum_{i=1}^{n} u_i$. Note that this algorithm can be considered as an anytime algorithm providing the best solution obtained in a given amount of time. Clearly if the time is enough then the solver can find the optimal solution and prove the optimality. Much information on CP and how to translate a model into a program can be found in literature [28] [29]. Also information on the "Large Neighborhood Search" algorithm can be found in literature [30] [31] [32].

5 CP Model

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

5.1 General model

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

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

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

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

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

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

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

Note that the quantifiers on the right-hand side define how many replicas of the constraints appear in the model. For the indexes of the constraint variables not appearing among the quantifiers, we assume that they take all the available values. This is just a compact notation to identify

sub-vectors (or sub-matrices) within data structures having a lot of indexes.

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

$$\min z = \sum_{i=1}^{n} \frac{s(a_i) - stq_i}{ewt_{queue(j_i)}} \tag{2}$$

This basic model should be enriched with a number of features needed to run the scheduler on a real HPC machine, as explained in section 3.

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

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

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

$$x(a_i) = 0 \; \forall i \in q_h \tag{3}$$

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

5.2 Allocation of jobs within a reservation

In the above model, we have considered reservations as jobs using resources required by the reservation for the time span of the reservation itself. However, on real machines reservations can be seen as private queues where only eligible users can submit jobs. The scheduling and dispatching of jobs in the reservation queue have to be treated as a separate problem handled by a separate model (Equations 4). The motivation is that the execution time for a job submitted to a reservation queue is the time span of the reservation and the resources available for the job are limited to the reservation resources. In addition, jobs submitted to the reservation queue have a deadline. Each reservation has a fixed start time, a fixed duration and for each node a set of reserved resources. These data can be extracted by proper functions, namely $getStart(resv)$, $getEnd(resv)$ and $getResource(resv, j, r)$ where j is the node and r the resource type.

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

$$\begin{aligned}
&s(a_i) :: [max(t, getStart(resv))..getEnd(resv) - d(a_i)] \\
&\qquad \forall i \in A_{JR} \\
&s(a_i) = getStart(j_i) \; \forall i \in B_{JR} \\
&x(a_i) :: [0,1] \; \forall i \in A_{JR} \\
&x(a_{ikj}) :: [0,1] \; \forall i \in A_{JR} \\
&\sum_{j=1}^{N_n} x(a_{ikj}) = x(a_i) \; \forall k, \forall i \in A_{JR} \\
&cumulative(a_{ikj}, req_{ikr}, getResource(resv, j, r)) \\
&\qquad \forall j \in N_n \forall r \in R
\end{aligned} \tag{4}$$

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

5.3 Feasibility check

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

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

For both these problems we have a phase of the feasibility check. The first is the reservation feasibility check that checks if there is enough room for executing the reservation. Then we have a feasibility check for each job separately, ensuring that the job requires resources that are available in the machine.

6 Framework Architecture

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

- *PBS_server* is a server that handles all the events and stores all the jobs, queues and settings, logs and information.

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

The original scheduler *PBS_sched* can be disabled and replaced with an ad-hoc scheduling algorithm. We take advantage of this functionality to embed our scheduler in PBS in a plug-and-play fashion (Figure 1). In this way, we leverage all the functionality of the PBS infrastructure such as tracking the system status and implementing scheduling decisions.

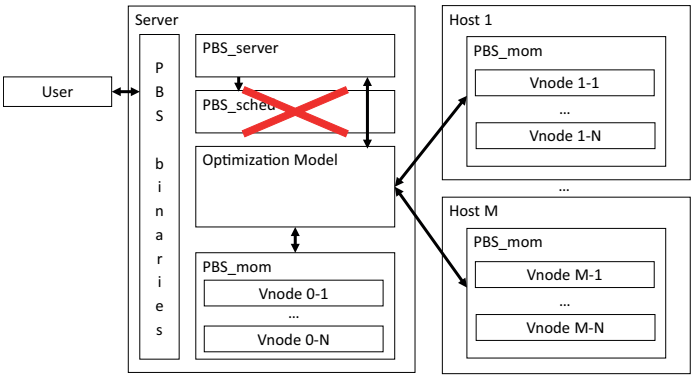

Fig. 1: Framework macro architecture

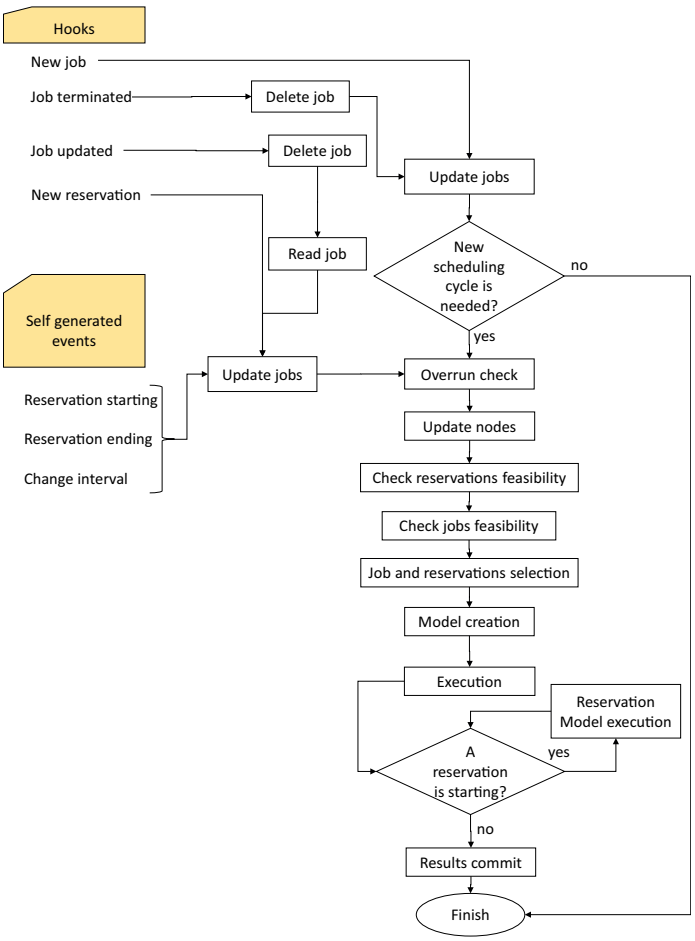

Fig. 2: Workflow

The framework receives events from the *PBS_server* using Hooks. The framework interacts with the *PBS_mom* component by asking the node state through PBS binaries. Then our scheduler is run and its decisions are sent to the *PBS_mom* component.

Figure 2 shows the workflow of our framework. The hooks "Job update", and "New reservation" trigger a scheduling cycle. Instead hooks "New job" and "Job terminated" trigger a scheduling cycle only if the state of the system has changed (i.e. awaken nodes, job deleted or new job submitted) since the last scheduling cycle. This avoids unnecessary overheads. Each scheduling cycle starts by (1) checking if a running job exceeds its wall-time request (Overrun check) in which case it is killed by the *PBS_server* component.

After this step, (2) the scheduling cycle updates an internal image of the node status which is used as input for the algorithm. In this phase the algorithm checks the node status (crashed/switched-off/activated). This information is used for the feasibility check that decides which reservations and jobs can be executed (in figure as "Check reservations feasibility" and "Check jobs feasibility"). Unfeasible jobs and reservations are rejected and excluded from the scheduling cycle.

If all the necessary conditions are satisfied the algorithm solves the model and decides the allocation of job units and their starting time.

Those job whose starting time is equal to the current time are then executed.

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

The first technique imposes and adjusts a timeout (referred to as δ) to the CP solver. δ is computed as follows:

- Initially δ is set to a predefined time K_1.
- The CP model is then executed for δ time.
- If the CP model does not generate any solution, the model is re-started a given number of times (say M), with increasing values of $\delta = K_2 * \delta$. We choose an exponential increment due to the NP-hardness of the problem. If after M iterations a solution has not been found, the scheduler simply waits for the next event.
- If instead the CP model returns one or more solutions the algorithm sets δ equal to the time taken to find the first solution plus K_1. This ensures to adapt the timeout δ to the instance hardness.

In addition, to limit the maximum solution time we

used several thresholds. (1) The maximum δ can be 300 seconds; (2) If the first solution takes more than $60s$ the search is stopped. (3) If the search of an improved solution takes more than 10s, again the search is stopped. All these thresholds have been empirically defined and guarantee an average overhead of ~ 6 seconds for solution process for hard instances (65 nodes and 1600 jobs).

Another limitation to the solution time can be obtained by reducing the size of the instance: after the second iteration with increased timeout, if no solution is found, the number of queued job considered in the scheduling is halved keeping the first queued jobs and excluding the other.

7 Experimental Results

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

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

- A direct comparison of the CP scheduler and two different PBS setups. These experiments are executed on a set of Virtual Machines (VM). Every VM runs a script that generates in a predictable fashion a sequence of jobs (each composed of a single *sleep* command).
- A statistical evaluation on the Eurora HPC with true jobs submitted by real users over five weeks[1].

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

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

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

Metrics on job waiting times:

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

Metrics on tardiness:

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

Metrics on computational overhead:

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

7.1 Evaluation setup

7.1.1 Simulation-based tests

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

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

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

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

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

7.1.2 Evaluation on the HPC

The second set of experiments is run on the Eurora HPC system. Eurora [33] is a heterogeneous HPC machine of CINECA. It is a fully operational prototype for future green

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

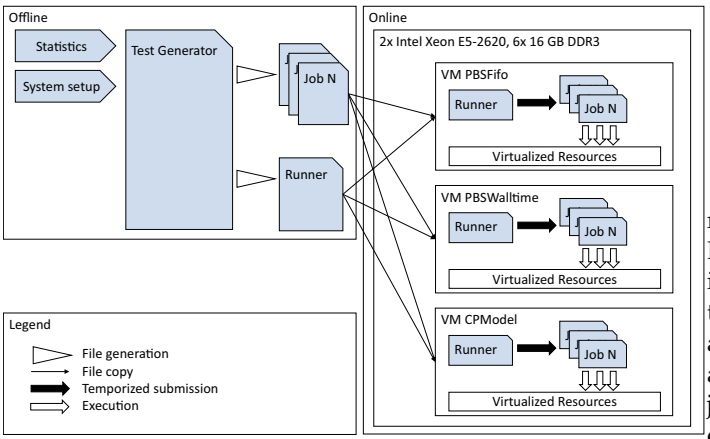

Fig. 3: Test Generation

HPC. Eurora is composed by 65 nodes, one login node with 12 cores, 32 nodes with 16 cores at 3.1GHz and 2 GPU Kepler K20 each and 32 nodes with 16 cores at 2.1GHz and 2 Intel Xeon phi (MIC) each.

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

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

7.2 Test generation

We have designed a software component (see Figure 3) to generate and submit a repeatable sequence of dummy jobs (i.e. sleep commands). The generation process has been calibrated based on real data (12,000 jobs submitted to Eurora in December 2014). For calibrating the arrival rates, we relied instead on statistics collected over the whole year 2013 from the Fermi HPC machine [34] at CINECA; the Fermi was chosen in this case due to its longer history of utilization.

In detail, for each test we generate n jobs (where n is an input parameter) to be submitted over a 24 hours period of real-world time. A certain percentage of jobs is submitted during daytime (8 AM to 6 PM), and the rest is submitted during the night (6 PM to 8 AM). Job arrival times are uniformly spread within each interval. In all our experiments, 89% of the jobs arrive at daytime and 11% at nighttime. All numbers mentioned above have been extracted from the CINECA statistics on the Fermi HPC machine. The following statistics are extrapolated from the Eurora execution traces. A fixed ratio of the generated jobs is then assigned to each system queue. In particular, 27% of the jobs are submitted to the debug queue, 72% to parallel, and 1% to np_longpar. The number of required nodes, cores, and the wall-time values are randomly generated for each job so as to match the Average Volume of Utilization (AVU) of its queue. In detail, we start by generating for each job i the number of requested nodes RN_i and the number of requested cores per node RC_i. In particular, let $nmin_q$ and $nmax_q$ be the minimum and maximum number of nodes for the queue q. Then, the node and core requests are obtained as:

Queue	AVU
debug	6465
parallel	147145
np_longpar	111372

TABLE 1: Eurora jobs utilization

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

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

$$W_i = \frac{AVU_q}{RN_i * RC_i} \tag{5}$$

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

$$AVU_q = \overline{NR}_{i(q)} * \overline{CR}_{i(q)} * \overline{Wall}_{i(q)} \tag{6}$$

where $\overline{NR}_{i(q)}$ is the average number of nodes requested by jobs in q, $\overline{CR}_{i(q)}$ is the average number of requested cores (per node) and $\overline{Wall}_{i(q)}$ is the average wall-time of the jobs in the q. These statistics are obtained from Eurora data. Our AVU_q values are reported in Table 1.

The GPU, MIC, and memory requirements are generated so as to match the average requirements observed on Eurora. In particular, jobs are partitioned into groups that are then assigned to a specific requirement value. The requirement values and the size of the partitions are reported in Table 2 for GPUs and MICs and in Table 3 for the memory.

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

7.2.1 Test 0: Behavior at different heterogeneity levels

This test is designed to give an overview on how this scheduler would behave within different numbers of heterogeneous resources. More heterogeneity would increase the number of problem constraints, reduce the number of feasible job assignments, and as a consequence it would generally make the platform more complex to manage. Heuristic scheduling methods such as those employed by

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

TABLE 2: GPUs & MICs per node request distribution on Eurora

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

TABLE 3: Memory per node request distribution on Eurora

PBS would primarily be affected by this increase in complexity. In CP, however, adding more constraints leads to an interesting trade-off. On the one hand, the scheduling problem may indeed become more complex. On the other hand, however, CP has the ability to actively exploit the problem constraints to reduce the size of the search space. Hence, more constraints may actually improve the performance of a CP based approach.

In figure 4 we have reported an experiment that show how the results change, changing the number of resources (adding more heterogeneity). The test consider 4 nodes and 100 jobs, we can show the computational overhead for instance with 1,2,3,4 and 6 different kind of heterogeneous resources. The test differs only by the resources requested by the jobs. However, being different jobs is not possible compare directly the result of metrics like "time in queue", etc... The only comparable metric is the overhead.

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

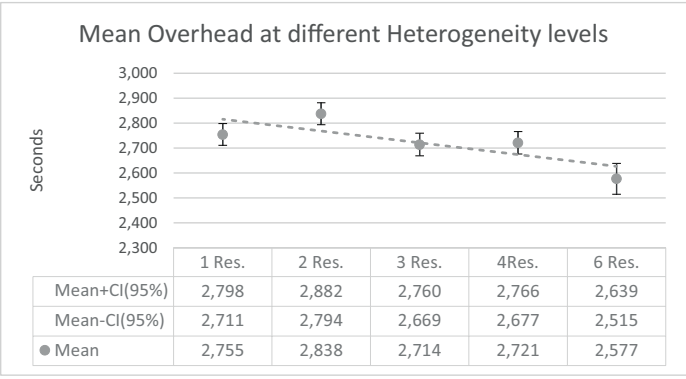

Fig. 4: Mean overhead at different heterogeneity levels

	PBSFifo	PBSWalltime	CPModel
WQT	347,583	170,686	168,032
AQ	31011,7	35637,6	30025,7
NL	55	52	45
TR	1954690	2349450	1832660
WT	283,219	110,034	105,985
AO	0,22	1,21	3,27
MO	1	4	5
%O	0,1	0,3	0,4

TABLE 4: Simulated test with 4 nodes and 99 jobs

	PBSFifo	PBSWalltime	CPModel
WQT	152,94	137,74	119,77
AQ	10216,7	9465,42	8053,09
NL	65	60	46
TR	1298810	1223690	1003970
WT	60,03	55,97	46,40
AO	0,47	3,14	11,45
MO	3	10	19
%O	0,33	1,61	6,78

TABLE 5: Simulated test with 65 nodes and 330 jobs

7.2.2 Test 1: 4 nodes 99 jobs

First we tested a system with a low workload. The test simulates 4 Eurora nodes (2 with MICs and 2 with GPUs): the results are reported in Table 4. PBSFifo has an advantage w.r.t. PBSWalltime and our model thanks to its lower overhead. However, both PBSWalltime and CP behave much better than PBSFifo w.r.t. all the performance metrics, in particular, those that take into account the priority of each queue (i.e. WQT and WT). The performance of our model is particularly good in this setting. In fact, we obtain substantial improvements w.r.t both PBSFifo and PBSWalltime on all the metrics on waitings and tardiness. This improvement is achieved at the cost of a larger overhead that, however, represents only the 0,4% of the makespan of the application.

7.2.3 Test 2: 65 nodes 330 jobs

Secondly we tested a system with a medium workload. In this test we simulate all the 65 Eurora nodes (32 with GPUs, 32 with MICs, and one log-in node): the results are in Table 5. Our model manages to considerably outperform PBSFifo and PBSWalltime in terms of all the metrics related to waiting time and delay. Also in this case, all the metrics on waitings and tardiness improve w.r.t. both PBSFifo and PBSWalltime even if the cost in terms of overhead grows to 6,78% which still justifies the gain obtained in all other metrics.

7.2.4 Test 3: 65 nodes 700 jobs

Finally we simulate a system with a high workload. We tested a 65 node configuration with a larger number of jobs (namely 700): the results are reported in Table 6.

Due to the large number of jobs and (more importantly) job units, in this case, our framework was forced to employ the overhead reduction techniques from Section 6. Such techniques are indeed effective in limiting the overhead, but they also have an adverse effect on the quality of the

	PBSFifo	PBSWalltime	CPModel
WQT	1034,2	853,681	2441,3
AQ	32066,1	27046,2	34816,3
NL	234	200	376
TR	16798300	13693000	16774800
WT	776,405	623,287	2013,18
AO	1,02	15,47	34,82
MO	11	57	120
%O	0,69	8,8	18,95

TABLE 6: Simulated test with 65 nodes and 700 jobs

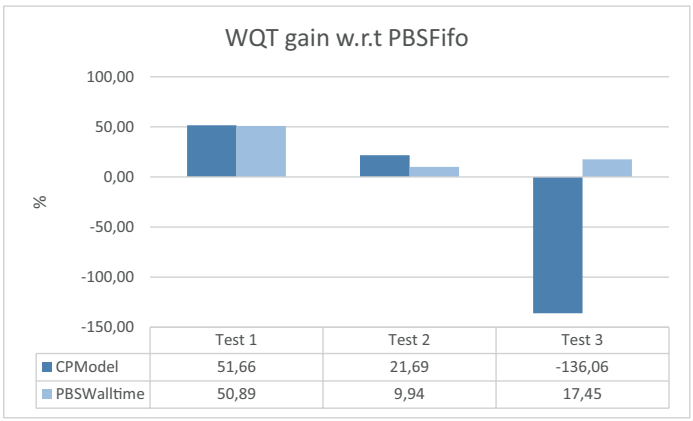

Fig. 5: Weighted queue time gain w.r.t. PBSFifo

model solutions. As it can be seen in the table, our model yields a little improvement in tardiness w.r.t. PBSFifo, a small increase in the total time in queue, and a considerable increase of the number of late jobs, the WQT, and the weighted tardiness.

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

7.2.5 Results comparison

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

Figure 5 reports the relative improvement of CP and PBSWalltime over PBSFifo in terms of WQT. The two approaches behave similarly on Test 1 (i.e. the easiest one), both obtaining a $\sim 50\%$ improvement over PBSFifo. As the test becomes more complex, the performance of our model gets better, beating PBSFifo by a factor 22%, against the $\sim 10\%$ obtained by PBSWalltime. In Test 3 (the largest), the overhead reduction techniques are active and this leads to a degradation of our results while PBSWalltime improves over PBFifo by a factor $\sim 17\%$.

In terms of average queue time (see Figure 7), PBSWalltime tends to improve over PBSFifo as the test size increases. Our approach (due to the overhead reduction techniques) follows the opposite trend. This is behavior is similar to the one observed for the WQT metric.

The results of the last three metrics (number of late jobs, tardiness and weighted tardiness (see Figures 8, 9, and10) follow the same trend as the WQT metric: CP works better than PBSFifo and PBSWalltime until the overhead reduction techniques are triggered (i.e. Test 3). In terms of total tardiness, the performance of our approach remains on par with that of PBSFifo even on Test 3, despite the large number of jobs.

Finally, Figure 12 compares the overhead to test-

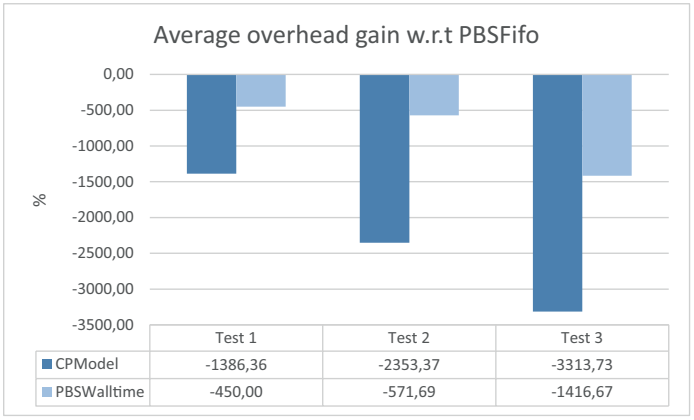

Fig. 6: Average overhead gain w.r.t. PBSFifo

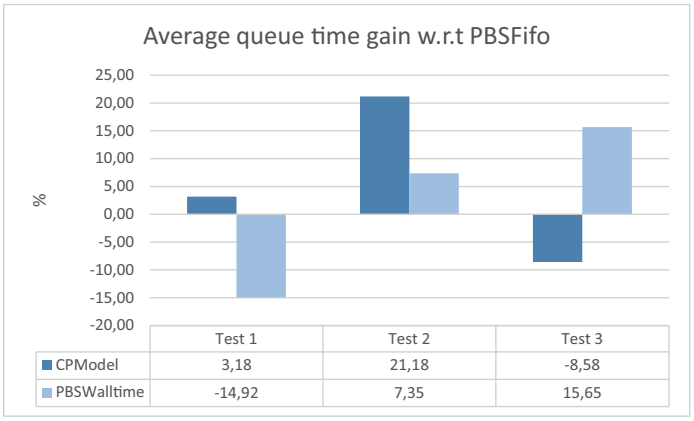

Fig. 7: Average queue time gain w.r.t. PBSFifo

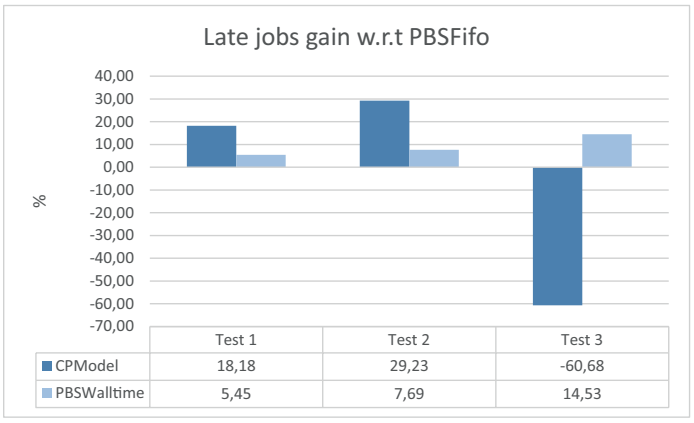

Fig. 8: Number of jobs in late gain w.r.t. PBSFifo

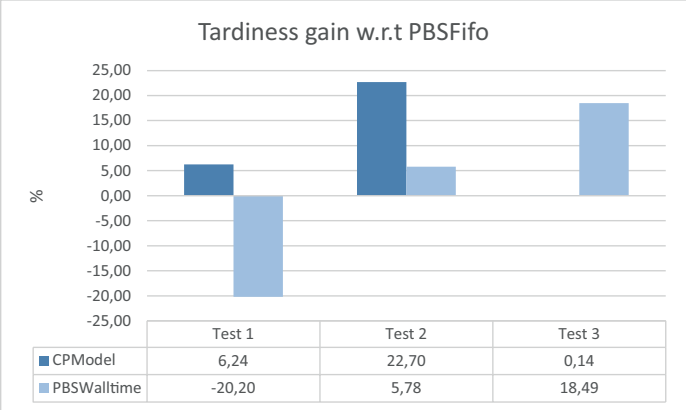

Fig. 9: Tardiness gain w.r.t. PBSFifo

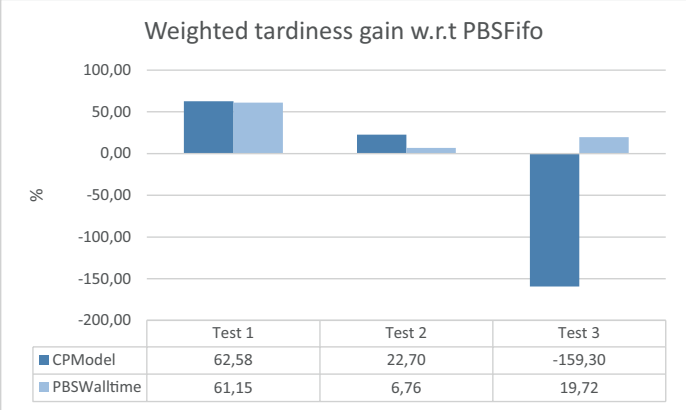

Fig. 10: Weighted tardiness gain w.r.t. PBSFifo

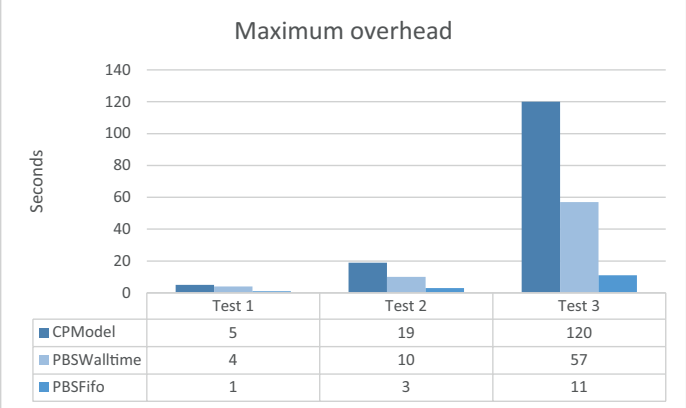

Fig. 11: Maximum overhead gain w.r.t. PBSFifo

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

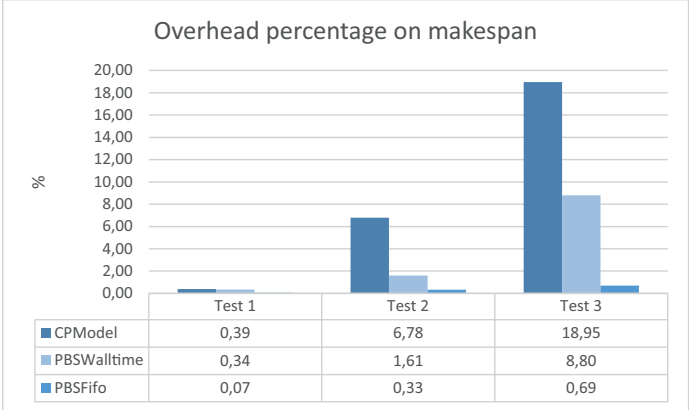

Fig. 12: Overhead percentage on execution time gain w.r.t. PBSFifo

7.2.6 *Guidelines for algorithm portfolio selection*

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

The purpose of this section is to identify a set of methods that can be used in a portfolio to solve HPC scheduling problems with increasing scale from lightweight to heavy ones, (see Figure 13).

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

1) On one end of the spectrum we have lightweight workloads, featuring a small number of jobs, each requiring only few nodes. In this situation finding a good schedule is trivial, since the machine is under-loaded, and using powerful optimization techniques provides little benefit.
2) The second class includes mid-range realistic workloads, typically they are characterized by less than 4'100 job units for a 65 nodes HPC machine. This is the range where making good dispatching decisions is not trivial, but the problem size is still manageable. In this situation, the CP system tends to provide the best results.
3) Finally, workloads with a very large number of jobs requiring many computation nodes (namely more than 270'000 job units * nodes submitted in 24h), call for the use of overhead-reduction techniques (Section 6). This allows to find solutions in a reasonable amount of time, but with adverse effects on the solution quality. Therefore in this range PBS heuristic approaches become the techniques of choice.

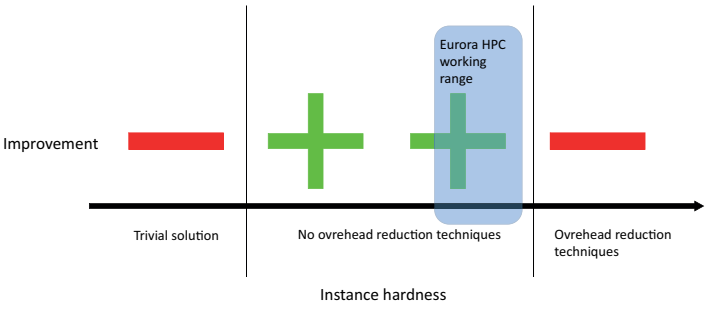

Fig. 13: Working ranges

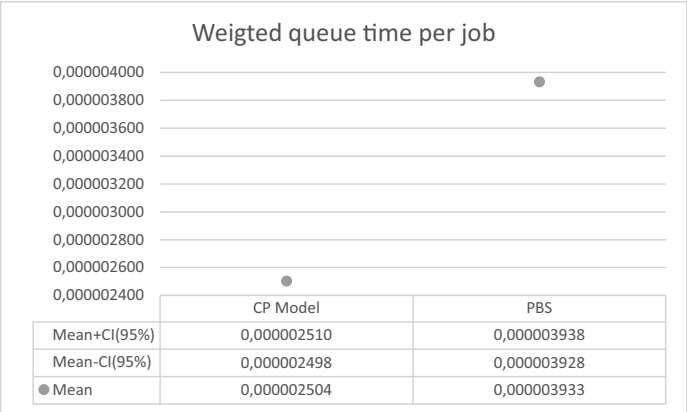

Fig. 14: Weighted queue time extrapolated from Eurora

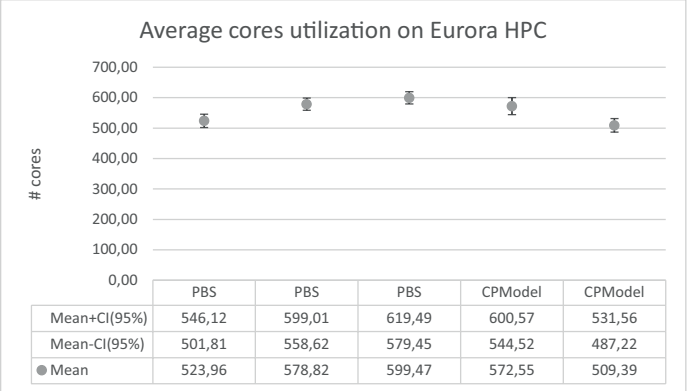

Fig. 15: Core utilization on Eurora

7.3 Execution on Eurora

Thanks to our modeling and design efforts from Section 5 and 6, we have managed to obtain a scheduling system that is mature enough to be deployed and evaluated on the actual Eurora HPC machine.

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

Since the comparison was performed in a production environment, it is impossible to guarantee that the two approaches process the same sequence of jobs. For this reason, the performance metrics that we employed in Section 7.1.1 are not meaningful in this setting and new metrics must be employed. This is due to the big variation between the number of jobs submitted in different days. For this, after some experimentation, we chose to compare the CP approach and PBSFifo in terms of: (1) the average WQT per job, and (2) the average number of used cores over time (i.e. the average core utilization).

Figure 14 compares the two approaches in terms of the first metric. Our CP system performed consistently better with an average WQT per job of $\sim 2.50 * 10^{-6}$, against the $\sim 3.93 * 10^{-6}$ of PBSFifo. The standard deviation for the two approaches is very similar. The average core utilization obtained by both approaches during each week is instead reported in Figure 15: the two approach have similar performance in terms of this second metric, which ranges between 520 and 599 for PBSFifo and between 510 and 573 for CP.

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

7.4 Overhead distribution

Table 7 reports the average time for each execution phase of our system (i.e. the steps in the flow chart from Figure 2). From the table, it is clear that moving from the simulated platform to real HPC leads to a considerable *decrease* of the total overhead. The distribution of the total overhead in the simulated tests and on the real system is instead depicted in Figure 16 and 17: in the simulated tests, the model resolution makes for most of the total overhead; on the real HPC the distribution is more balanced, and some phases (reservation/job feasibility check, and result commit) are proportionally much heavier than in the simulated platform.

	Simulated test	Eurora
Update jobs	4,11	3,96
Update queues	0	0,02
Update Nodes	3,26	1,45
Check reservations feasibility	0	1,21
Check jobs feasibility	0,05	1,02
Jobs and reservations selection	0,07	0,02
Model creation	2,62	0,93
Model execution	21,91	2,31
Reservation check	0	0
Reservations model execution	0	0
Result commit	0,33	2,53
Total	32,35	13,45

TABLE 7: Optimization model average overheads (seconds)

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

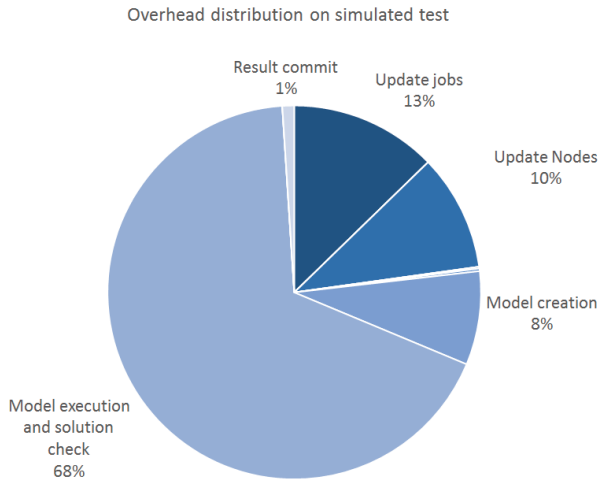

Fig. 16: Overhead distribution for the simulated test

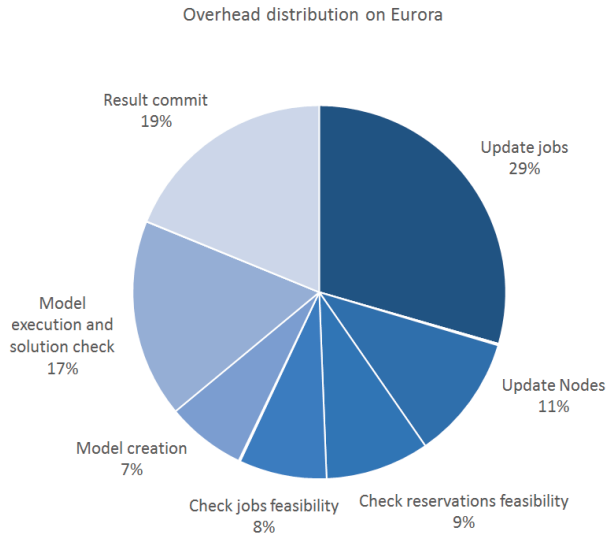

Fig. 17: Overhead distribution for Eurora

8 Conclusion

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

Despite the system has been deployed on a real HPC machine, a number of improvements are still pending: First, the uncertainty on the execution time of jobs, can be considered in the scheduling algorithm and can be characterized through learning techniques as done in the work by Tsafrir et al. [26]. Considering the job execution time uncertainty heavily impacts the scheduler model thus affecting solution algorithms: techniques such as robust optimization and stochastic constraint programming have to be considered. A second improvement can be obtained by providing hot-starts to the optimization engine: they can be either be computed as the solution of the previous run or via sophisticated heuristics algorithms enriched with back-filling techniques. Finally a deeper integration of the optimization engine into the scheduling management framework can be obtained by a changing its source code, this would need longer development time but possibly reduce the overhead introduced by the interaction.

Acknowledgments

This work was partially supported by the FP7 ERC Advance project MULTITHERMAN (g.a. 291125), by the YINS RTD project (no. 20NA21 150939), evaluated by the Swiss NSF and funded by Nano-Tera.ch with Swiss Confederation financing and by CINECA. The authors would like to thank CINECA in particular C. Cavazzoni, I. Baccarelli and A. Federico for granting us the access to their systems and Altair, in particular P. Masera and D. Dorella, for providing us access to undocumented APIs.

References

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

[2] P. Works, "Pbs professional 12.2, administrators guide, november 2013," http://citi.clemson.edu/palmetto/files/PBSProAdminGuide12-2.pdf, 2012.

[3] A. Computing and G. Computing, "Torque resource manager," http://docs.adaptivecomputing.com/torque/6-0-0/torqueAdminGuide-6.0.0.pdf, 2015.

[4] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[5] A. Bartolini, A. Borghesi, T. Bridi, M. Lombardi, and M. Milano, "Proactive workload dispatching on the eurora supercomputer," in *Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, B. OSullivan, Ed. Springer International Publishing, 2014, vol. 8656, pp. 765–780.

[6] P. Salot, "A survey of various scheduling algorithm in cloud computing environment," *International Journal of research and engineering Technology (IJRET), ISSN*, pp. 2319–1163, 2013.

[7] D. G. Feitelson, "Experimental analysis of the root causes of performance evaluation results: a backfilling case study," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 2, pp. 175–182, 2005.

[8] A. W. M. Alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 6, pp. 529–543, 2001.

[9] O. Sarood, A. Langer, A. Gupta, and L. Kale, "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 807–818.

[10] I. A. Moschakis and H. D. Karatza, "Evaluation of gang scheduling performance and cost in a cloud computing system," *The Journal of Supercomputing*, vol. 59, no. 2, pp. 975–992, 2012.

[11] C. Du, X.-H. Sun, and M. Wu, "Dynamic scheduling with process migration," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*. IEEE, 2007, pp. 92–99.

[12] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[13] D. Jackson, "New issues and new capabilities in hpc scheduling with the maui scheduler," http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.5070&rep=rep1&type=pdf.

[14] S. Agarwal, R. Garg, and N. K. Vishnoi, "The impact of noise on the scaling of collectives: A theoretical approach," in *High Performance Computing–HiPC 2005*. Springer, 2005, pp. 280–289.

[15] A. Bastoni, B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," *Proceedings of OSPERT*, pp. 33–44, 2010.

[16] R. Gioiosa, S. A. McKee, and M. Valero, "Designing os for hpc applications: Scheduling," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. IEEE, 2010, pp. 78–87.

[17] S. Soner and C. Özturan, "Integer programming based heterogeneous cpu–gpu cluster schedulers for slurm resource manager," *Journal of Computer and System Sciences*, vol. 81, no. 1, pp. 38–56, 2015.

[18] Y. Kessaci, N. Melab, and E. Talbi, "A pareto-based ga for scheduling hpc applications on distributed cloud infrastructures," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 2011, pp. 456–462.

[19] K. Wang and I. Raicu, "Towards next generation resource management at extreme-scales," http://datasys.cs.iit.edu/publications/2014_IIT_PhD-proposal_Ke-Wang.pdf, 2014.

[20] J. P. Jones and B. Nitzberg, "Scheduling for parallel supercomputing: A historical perspective of achievable utilization," in *Job Scheduling Strategies for Parallel Processing*. Springer, 1999, pp. 1–16.

[21] D. Klusáček, H. Rudová, R. Baraglia, M. Pasquali, and G. Capannini, "Comparison of multi-criteria scheduling techniques," in *Grid Computing*. Springer, 2008, pp. 173–184.

[22] V. Chlumsky, D. Klusácek, and M. Ruda, "The extension of torque scheduler allowing the use of planning and optimization in grids," *Computer Science*, vol. 13, pp. 5–19, 2012.

[23] Y. Yuan, Y. Wu, W. Zheng, and K. Li, "Guarantee strict fairness and utilizeprediction better in parallel job scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 4, pp. 971–981, 2014.

[24] E. Shmueli and D. G. Feitelson, "On simulation and design of parallel-systems schedulers: are we doing the right thing?" *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 7, pp. 983–996, 2009.

[25] ——, "Backfilling with lookahead to optimize the packing of parallel jobs," *J. Parallel Distrib. Comput.*, vol. 65, no. 9, pp. 1090–1107, 2005.

[26] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 6, pp. 789–803, 2007.

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

[28] M. Wallace, "Practical applications of constraint programming," *Constraints*, vol. 1, no. 1-2, pp. 139–168, 1996.

[29] IBM, "Modeling with ibm ilog cp optimizer - practical scheduling examples," http://public.dhe.ibm.com/common/ssi/ecm/ws/en/wsw14076usen/WSW14076USEN.PDF.

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

[31] D. Godard, P. Laborie, and W. Nuijten, "Randomized large neighborhood search for cumulative scheduling." in *ICAPS*, vol. 5, 2005, pp. 81–89.

[32] P. Laborie and D. Godard, "Self-adapting large neighborhood search: Application to single-mode scheduling problems," *Proceedings MISTA-07, Paris*, pp. 276–284, 2007.

[33] 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.

[34] F. Falciano and E. Rossi, "Fermi: the most powerful computational resource for italian scientists," *EMBnet. journal*, vol. 18, no. A, pp. p–62, 2012.

Thomas Bridi received the degree in Computer Science Engineering from the University of Bologna, Italy, in 2014. He is a PhD student at Department of Computer Science and Engineering (DISI), University of Bologna, Italy. His research interests include Scheduling Optimization and Constraint Programming.

Andrea Bartolini received a Ph.D. degree in Electrical Engineering from the University of Bologna, Italy, in 2011. He is currently a postdoc researcher in the Department of Electrical, Electronic and Information Engineering Guglielmo Marconi (DEI) at the University of Bologna. He also holds a post-doc position in the Integrated Systems Laboratory at ETH Zurich. His research interests concern dynamic resource management ranging from embedded to large scale HPC systems with special emphasis on software-level thermal and power-aware techniques. His research interest also includes ultra-low power design strategies for biosensors nodes operating in near-threshold.

Michele Lombardi is a an assistant professor (no tenure track) at University of Bologna. He is working on the integration of heterogeneous techniques for Combinatorial Optimization, and on hybrid off-line/on-line optimization. His expertise is on Constraint Programming, Integer Linear Programming and Machine Learning, with main applications on resource allocation and scheduling problems for embedded systems.

Michela Milano is associate professor at the department of Computer Science and Engineering at the University of Bologna. Her research interest cover theoretical and practical aspects of constraint reasoning and optimization. Application areas include planning and scheduling, computational sustainability, embedded system design, energy systems and smart cities. Michela Milano is author of more than 130 papers in International conferences and journals. She is Editor in Chief of the Constraints Journal and Area Editor for INFORMS Journal on Computing and Constraint Programming Letters. She was program chair of CPAIOR 2005, CPAIOR 2010 and CP2012. Michela Milano is the coordinator of the EU-funded ePolicy project and participating to two other EU project on smart cities.

Luca Benini is currently a Full Professor with the University of Bologna and the Chair of Digital Circuits and Systems at ETHZ. His research interests are in energy efficient system design and MultiCore SoC design. He is also active in the area of energy efficient smart sensors and sensor networks for biomedical and ambient intelligence applications. He has authored over 700 papers in peer reviewed international journals and conferences, four books and several book chapters. He is a member of the Academia Europaea