Optimizing Profit and Cooling Costs in HPC Cluster Scheduling

No Author Given

No Institute Given

Abstract. In the last few years, the importance of the job scheduling in High-Performance Computing (HPC) have been increasing recogized. The application of optimization models in this setting is challenging however due to the large-scale nature of problem that involves thousands of daily job submission. This paper considers a high-fidelity scheduling model arising in HPC clusters that maximizes profit over time taking into account job revenues, energy costs, and cooling costs. It proposes a constraint program for this task, a novel heuristic based constraint programming, and three search strategies. Our proposals are compared to state-of-the-art rule-based schedulers and a commercial CP solver. Experimental results on a real case study shows that our approach improves profits by more than 7% over rule-based scheduler and 15% over the default search of the commercial solver. These results indicate the potential of constraint programming in scheduling HPC clusters.

Keywords: Scheduling, Constraint Programming, Heuristic, Search, Profit, Green, HPC, Supercomputing

1 Introduction

The importance of the job scheduling in High-Performance Computing (HPC) have received increased attention in recent years. In particular, optimization technology has been shown to produce substantial improvements for various performance metrics [3–5]. For instance, Bridi et al [4] successfully applied a Constraint Programming (CP) model to improve job waiting times by up to 20% compared to rule-based schedulers, exploiting the strength of CP for scheduling problems [2]. Moreover, the benefits of CP are not only restricted to performance quality; CP also provides the modeling flexibility that is often welcome in modeling complex scheduling problems in HPC.

The main limitation of CP for HPC is scalability, as indicated in [4]. CP approaches require significantly more computation time compared to rule-based schedulers, which typically produce lower-quality solutions. This paper aims at addressing this tradeoff by coupling these two approaches for an HPC problem with a complex objective function that takes into account (1) the profit obtained from the job executions; (2) the expenses for the workload; and (3) the expenses for the cooling. To achieve a good tradeoff between efficiency and solution quality,

we propose a constraint-programming heuristic to obtain a good starting solution and three different search strategies.

Experimental results on a real case-study shows that our approach dominates the Randomized Large Neighborhood Search (RLNS) implemented in *IBM ILOG CP Optimizer*, as well as classical rule-based schedulers that are widely used in the field. Compared to rule-based schedulers, our approach improves the profit by more than 7% primarily through a reduction of the makespan. Our approach also produces improvements of more than 10% compared to the default search of CP Optimizer. Finally, it is worth mentioning that, in the case of a low-efficiency system, our approach also improves the cooling schedule, leading both a more efficient and greener system. These results are compelling given the sophistication of the model proposed here. They demonstrate that constraint programming has a significant role to play in the large-scale scheduling problems that arise in HPC clusters.

The paper is organized as follows. Section 2 discusses related work and Section 3 formalizes the job scheduling and dispatching problem for HPC clusters. Section 4 presents the classical rule-based schedulers used in our comparison. Section 5 describes our optimization model. Section 6 presents the heuristic to obtain an initial solution, while Section 7 presents the three search strategies we propose. Section 8 presents our experimental results and Section 9 concludes the paper. The paper focuses only on the offline optimization model and leaves the integration with a reactive approach for future work.

2 Related Work

Salot et al. [10] presents an overview of scheduling algorithms for HPC facilities. The majority of the commercial schedulers (e.g., PBS Professional [13], Torque [7], and Slurm [15]) make it possible to to implement many of these algorithms by specifying an appropriate jobs priority (rule-based scheduling). Our experimental results consider several such scheduling policies.

The use of optimization technology has been received increased interest in recent years but it was largely ignored before. We review the related work in optimization in the rest of this section.

Chlumsky et al. [5] propose a Tabú search to optimize job scheduling. This approach has been tested as a plugin for the Torque scheduler. Unfortunately, the proposed approach is feasible only for meta-scheduling: It does not model the execution of job chunks on different nodes in parallel.

The work by Bartolini et al. [3], which was later expanded by Bridi et al. [4], presents a CP model for optimizing the waiting time of the job. Experimental results indicated that CP can be successfully applied to scheduling HPC facilities but also showed consistent scalability problems preventing the application of the approach to real-size instances. The present paper overcomes this problem by proposing a heuristic solution which is then improved by three search algorithms.

Hurley et al. [9] tackled the energy-optimization problem at the meta-scheduler level. The authors show that, by combining optimization and machine learning

techniques, it is possible to minimize the energy expenses even in the case of variable and unknown a priori energy price.

Yang et al. [14] present a power-aware job scheduler designed to optimize the energy bill of the HPC cluster exploiting the variability of the energy cost. However, this work consider neither the cooling system nor external factors (such as the outside temperature).

Conficoni et al. [8] establish the relation between the cooling cost in HPC infrastructures, the IT power consumption, and the external ambient temperature. Indeed, accordingly to these parameters, the cooling circuitry operates at different set-point with different power consumption. In air-cooled data centers, the variability of the power usage efficiency can range from 10 to 40% while, in case of hybrid cooling, the range is 9-13% depending on the external temperature fluctuations.

3 HPC Job Scheduling and Dispatching

HPC job scheduling and dispatching can be modeled as non-preemptive scheduling problem with cumulative resources. Informally speaking, a solution assigns a node and a start time to each job unit for all submitted jobs. No resource capacity can be exceeded.

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

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

$$\begin{aligned} & st_i \in [q_i, .., H] \ (i \in A) \\ & sn_{i,w} \in N \ (i \in A, w \in UN_i) \\ & \sum_{i \in R(t,n)} req_{i,k,w} \leq c_{n,k} \ (t \in [0, .., H], n \in N, k \in K) \end{aligned} \tag{1}$$

where H is the scheduling horizon and

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

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

4 Rule-based scheduling

Rule-based scheduling is the prominent approach approach to solve HPC job scheduling. A rule-based scheduler processes jobs and nodes in a given order

which is specified via a customized rule. When a job is processed, the scheduler considers each job unit and starts querying the system nodes to find a sufficient amount of free resources. When all job units have a candidate node, the job is started. If a job cannot be immediately started, two alternative behaviors are possible:

- *Strict ordering*: if job i cannot be started, the scheduler stops and waits for the next termination event to restart the process from i.
- *Non-strict ordering* if job i cannot be executed, the scheduler skips it and tries to schedule job $i + 1$.

Our experimental results will compare our approach against both a strict-ordering and a non-strict-ordering scheduler, using four different rules to order jobs. The following rules will be considered:

- EST: The set of queued jobs is ordered by increasing Earliest Start Time.
- WT: The set of queued jobs is ordered by increasing Wallclock Time.
- Profit: The set of queued jobs is ordered by profit gained from the job execution divided by the cost of the job in decreasing order.

Since all nodes in our experimental setup are homogeneous, they are ranked in a lexicographic order.

5 The Constraint Programming Model

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

The Model Variables The model consists of a set of interval variables a_i, each representing job i. Each such interval variable encapsulates four integer decision variables: $a_i.st$ denotes the interval start time, $a_i.d$ the interval duration, and $a_i.et$ the end time, and $a_i.p$ whether the interval variable is present or absent in the solution. To keep track of the job units assigned to each node, the model uses a set of interval variables $ju_{i,w,n}$, which represents the execution of unit w of job i on node n. While the a_i variables are present, the $ju_{i,w,n}$ are optional and capture the possibility of the unit executing of various nodes.

The Constraints The constraints of the CP model are given in Equation 2:

$$\begin{aligned}
&a_i.setDuration(wt_i)\ \forall i \in A \\
&a_i.setStartMin(q_i)\ \forall i \in A \\
&a_i.setStartMax(H)\ \forall i \in A \\
&\text{Cumulative}(ju_{(_,_,n)}, req_{(_,k,_)}, c_{n,k})\ \forall n \in N, \forall k \in RK \\
&\text{Alternative}(a_i, ju_{(i,_,_)}, |UN_i|)\ \forall i \in A
\end{aligned} \tag{2}$$

Each activity a_i has a fixed duration that corresponds to the wallclock time of the job. The notation $ju_{(i,\_,\_)}$ represents the set of all (optional) interval variables

associated with job i. Similarly, $ju_{(\_,\_,n)}$ represents the set of (optional) activities associated with node n. Finally, $req_{(\_,k,\_)}$ represents the set of the requirements of all job units for the resource type k.

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

The Objective Function The definition of the objective function of our model relies on a set S of *time segments*. A time segment s is a portion of the day with an approximately uniform temperature (i.e., within an interval of 5 degrees Celsius). A segment s is defined by a start time $s.st$, an end time $s.et$, and a temperature $s.t$. The 5 degree tolerance is chosen to be compliant with the cooling model [8]. Equations 3 specifies the objective function, which maximizes the total profit during each time segment.

$$Maximize\left(\sum_{s\in S} G_s * K_1 - (W_s + J_s) * K_2\right)$$

$$\begin{aligned} G_s &= \sum_{i\in A} execTime(a_i, s) * resourceCost(ju_i, res_i) \\ W_s &= \sum_{i\in A} execTime(a_i, s) * power_i \\ J_s &= W_s * LUT(W_s, s) \end{aligned} \tag{3}$$

We now explain all the terms of this objective. G_s is the amount or resources used by all the jobs during the time segment s. The expression $execTime(a_i, s)$ returns the amount of time the job a_i has spent executing during segment s:

$$\begin{aligned} execTime(a_i, s) = {} & a_i.d * (a_i.st \geq s.st) * (a_i.et < s.et) + \\ & (s.et - a_i.st) * (a_i.st \geq s.st) * (a_i.st < s.et) * (a_i.et \geq s.et) + \\ & (s.et - s.st) * (a_i.st < s.st) * (a_i.et \geq s.et) + \\ & (a_i.et - s.st) * (a_i.st < s.st) * (a_i.et \geq s.st) * (a_i.et < s.et) \end{aligned} \tag{4}$$

$resourceCost(ju_{i,w,n}, res_i)$ is an expression that returns the cost of the resources used by each job unit ju_i. This accounting is used by most HPC centers (see for instance [6]) to quantify the used resources and to compute the amount of money spent by a user. The term $power_i$ represents the power consumption measured by the system for job i. Based on such value we can compute the amount W_s of energy spent for the workload (i.e. to execute the jobs) during segment s. The expression $LUT(W_s, s)$ (Equation 5) returns the cooling efficiency for the workload of segment s, given the workload energy W_s of the segment, and the

external temperature $s.t$. This expression is based on the PUE table from [8]. We use a TABLE constraint to relate the workload power P'_s, the temperature $s.t$, and the efficiency of the cooling system. With $LUT(W_s, s)$, we compute J_s as the amount of energy for the workload during the time segment s times the efficiency of the cooling system. The equation then becomes:

$$\begin{aligned} powerW_s &= \frac{W_s}{s.et - s.st} \\ P'_s &= round(powerW_s, 500) \\ LUT(W_s, s) &= Table(P'_s, s.t, PUE) \end{aligned} \tag{5}$$

Given constants K_1 (amount of money obtained per volume of job utilization) and K_2 (cost of the energy) are known, Equation 3 captures that the CP model maximizes the profit as $\sum_{s \in S} G_s * K_1 - (W_s + J_s) * K_2$.

For the sake of simplicity, this paper assumes that S is the set of segments of a given day. In practice, S should contain a segment for the jobs already in execution, and a segment for the jobs that cannot be scheduled today and have to be delayed to tomorrow. The overall approach is implemented using *ILOG CP optimizer*, modelling jobs as interval variables. As described above, an interval variable corresponds in fact to a set of different decision variables (e.g., a start time, duration, presence, etc.). Each decision variable has a finite domain, defined by a lower bound lb and an upper bound[1] ub.

6 A Job-Based Heuristic for Constraint Programming

This section presents a job-based heuristic for scheduling jobs in HPC clusters, exploiting the constraint-programming propagation engine. *The heuristic, which iterates over each job successively and determines its minimum starting time, is a significant departure from the rule-based schedulers which iterates through time trying to schedule each job.* The pseudocode is shown in Algorithm 1. The method is presented as a loop but is actually implemented using the ILOGOAL construct provided by CP Optimizer. The algorithm selects the first job that has yet to be fixed (line 3). It then checks whether the minimum start time of the job is a feasible start time (line 4). If it is feasible, the start time is assigned (line 5). Otherwise, the heuristic increases the lower bound (line 7). Whenever a variable is assigned, the solver performs constraint propagation (line 9).

7 Search Heuristics

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

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

Algorithm 1 Heuristic(J = ordered list of jobs)

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

Algorithm 2 The Multi-Search Approach

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

7.1 Multi-Search

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

7.2 Relaxation-Based Search

The next strategy is called RelaxationSearch and its goal is to decrease the search space for scalability purposes. This is achieved by fixing the dispatching for the most difficult jobs to dispatch, i.e., those with several units, which require strong synchronization. This RelaxationSearch strategy starts with the Multi-Search and adds two types of constraints depicted in Equation 6). The first constraint forces the next solution to improve the current one. The second set of constraints forces the jobs that require more than a job unit to the values fixed by the Multi-Search solution. After these two constraints are imposed, the default search of CP Optimizer is employed improving the solution exploring: (1) different dispatching for the jobs with just one job unit and (2) different schedules for every job.

Algorithm 3 Delay-based Search

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

$$\begin{aligned} &\sum_{s \in S} G_s * K_1 - (W_s + J_s) * K_2 \geq bestSolValue \\ &ju_{(i,j,n)}.p = bestSol.ju_{(i,j,n)}.p \; \forall i \in B, \forall j \in [1, \ldots, |UN_i|], \forall n \in N \\ &\quad \text{where: } B = \{i \in A |\ |UN_i| > 1\} \end{aligned} \tag{6}$$

7.3 The Delay Search

This last search strategy is called DelaySearch and is presented in Algorithm 3. It key intuition is to delay jobs that are normally scheduled greedily to explore different parts of the search space. This search starts with the Multi-Search (line 1). The procedure then selects a job to delay (line 2) and imposes that the job cannot be scheduled before at least *delay* time units (line 4). If the job is delayed after the time horizon H, it is stored in the list of already delayed jobs and a new job is selected to be delayed (lines 5-8). The *alreadyDelayed* jobs are fixed to the start time selected in the best solution found so far (lines 9-11). Then, the heuristic is called (line 12) and the best solution is updated (line 13). This approach is repeated until a termination condition is reached (line 3). In our case the termination condition is a time limit or the fact that all the jobs have been delayed to the time horizon H [2].

8 Experimental Results

The Test Cases: The purpose of this evaluation is to investigate whether it is possible to improve over rule-based schedulers for large instances. The experiments have been performed on a number of different scenarios on a simulated

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

HPC machine. The test cases are built by collecting the job data (including the time of the request submission and the resource data) from the parallel workload archive of the CEA Curie system [1]. The test cases span a period of 23 days with of 33583 submitted jobs. For each job, a random power consumption in the range of 7.8Watt-core to 11.11Watt-core [11] has been generated in order to cover this missing information. The simulated HPC machine is composed by 300 nodes with 32 cores each and 128GB of RAM. These test cases, with more than 30,000 jobs, are appropriate to test scalability.

The Scenarios: The test case has produced by combining the core data on job submissions with different scenarios to explore the behavior of the proposed algorithms with different cooling systems. They are defined in terms of the Power Usage Effectiveness (PUE), which measures the efficiency of a data center [12]. The PUE is given by $PUE = \frac{P_w+P_c}{P_w}$, where P_c is the power spent in cooling and P_w is the power spent in workload. The experimental results consider two main scenarios

1. An air cooling with a PUE of about 1.4, depending on the external temperature and the workload energy consumption;
2. an hybrid cooling system with a PUE of about 1.1;

that are combined with two different environment temperatures for the summer and the winter. These scenarios will enable us to explore how the schedules change for different levels of cooling efficiency.

The Experimental Setting: The test cases are executed by running the scheduling algorithms at the beginning of each day over the horizon. Each run was executed with a single worker with a time limit of 4000 seconds for the search procedures (after the initial solution by the MultiSearch), which is significantly lower than the available time for an off-line scheduler (86,400 seconds).

The Implementation: All the algorithms are implemented in *C++* and use the *IBM ILOG CP Optimizer 12.7.0* solver. The experimental results were performed on a 2x*Intel Xeon Processor E5-2670 v3* server with 128GB of RAM.

Profit Over Time: Figures 1, 2, 3, and 4 depict the sum of the profits over the time horizon for our best search strategy, the CP Optimizer RLNS, and the best rule-based scheduler for each scenario. All the schedulers reach more or less the same profits as the picture depicts the entire schedules for each approach. As a result, the figures only display the benefits of the cooling optimization. Note also the shape of the plots: The sum of the profits of our search strategies always dominate both the ILOG search and the best rule-based scheduler except for the first days. *The most interesting outcome however is the makespan.* Our search strategies produce smaller makespans compared to both rule-based schedulers and the default ILOG search. and the rule-based schedulers.

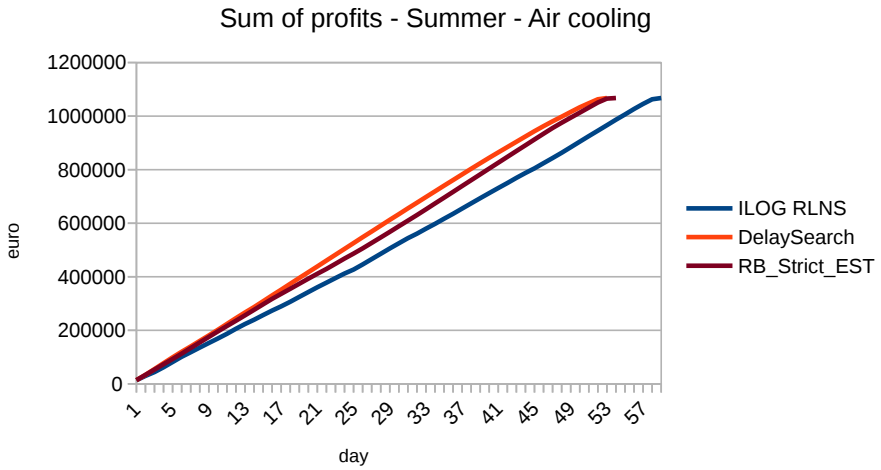

Fig. 1. The Sum of the Profits: Scenario with Summer Temperatures and Air Cooling.

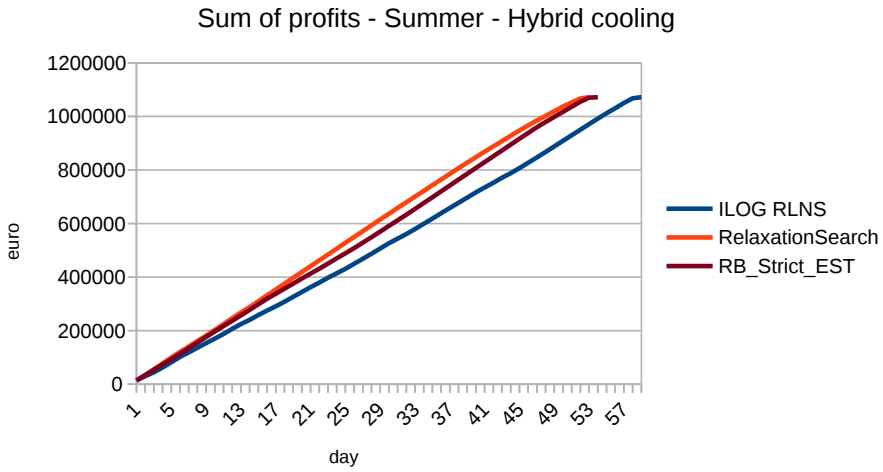

Fig. 2. Sum of the profits: Dummer Temperatures and Hybrid Cooling.

The Benefits of Optimization Now that the behavior of the algorithms is better understood, we are in a position to present the core contributions of this work. The key results are presented in Tables 1 and 2. The third column of Table 1 shows the improvement in the profit at the time of the last submission (day

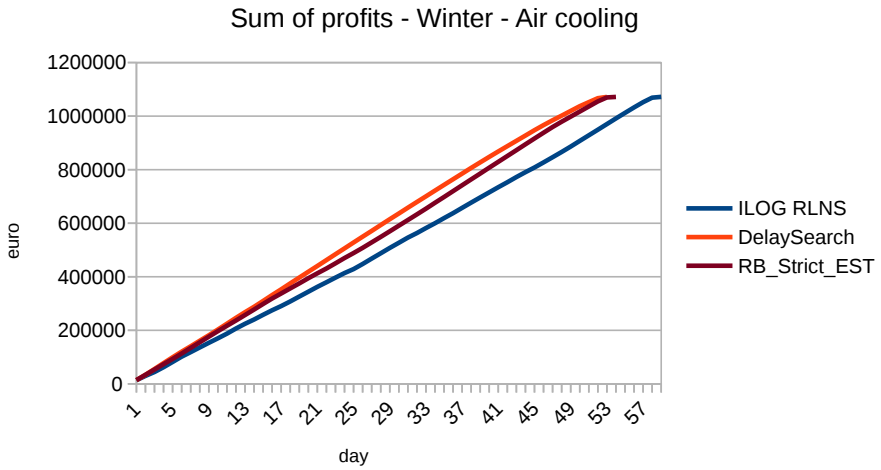

Fig. 3. Sum of the Profits: Winter Temperatures and Air Cooling.

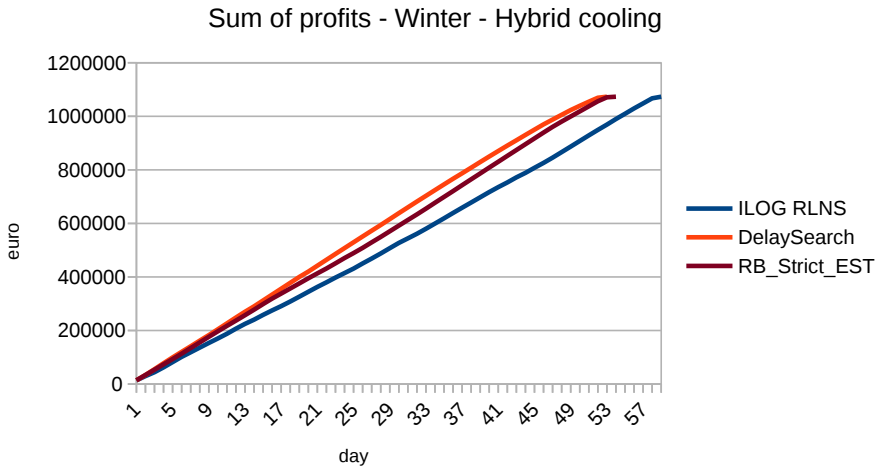

Fig. 4. Sum of the Profits: Scenario with Winter Temperatures and Hybrid Cooling.

23).[3] This measure is highly significant since HPC clusters are intensively used (i.e., only a few days each year has no submitted jobs): It captures the fact that our new scheduling algorithms can schedule more jobs. The experimental results indicate that the profit improvement is between 7.66% and the 7.95% compared

[3] Note that the jobs will run for much longer, due to their intense computational requirements.

Cooling	Season	Profit (%)	Makespan (%)	PUE Best Search	PUE Best RB
Air	Winter	7.71	1.85	1.11	1.11
	Summer	7.66	1.85	1.43	1.43
Hybrid	Winter	7.86	1.85	1.00	1.00
	Summer	7.95	1.85	1.09	1.09

Table 1. Profit Improvement and Makespan Improvement over the Best Rule-Based Scheduler. PUE of our Best Search and PUE of the Best Rule-Based Scheduler.

Cooling	Season	Average daily profit Increments
Air	Winter	20144.28€
	Summer	20222.91€
Hybrid	Winter	20227.90€
	Summer	20249.67€

Table 2. Average Daily Profit Increments of the New Algorithms.

to the best rule-based scheduler. The fourth column shows the improvement in makespan compared to the best rule-based scheduler. The new scheduling algorithms improves the makespan by 1 day over the 54 day horizon. Table 2 depicts the average daily profit increments due to the shorter makespan: The increments are more than 20100€ which is quite significant. Regarding efficiency, Columns 5 and 6 of Table 1 shows that there is no difference between the approaches.

Comparison of the Proposed Search. Figures 5, 6, 7, and 8 compares the proposed search strategies. These figures show the improvement in the total profit obtained compared to the ILOG default search (RLNS). Note that, because the algorithms are run each day on the submitted job, it is possible that the DelaySearch and the RelaxationSearch be worse than the MultiSearch, even if they are seeded with the MultiSearch solution. In fact, the results show that this happens on some of the scenarios. This is due to the fact that the algorithms greedily optimize each day.

The results show that, in most cases, the best result is obtained by the DelaySearch with respectively 10.51854%, 10.451758%, and 10.753464% in the summer-air-cooling, winter-air-cooling, and winter-hybrid-cooling scenarios. For the summer scenario with hybrid cooling, the best search is RelaxationSearch which improves the result obtained by the ILOG solver by 10.36506%. These results are particularly interesting, since they show the MultiSearch is a strong performer, despite its simplicity. In other words, the job-based heuristic based on constraint programming, when run with different orderings, already produces significant improvements over the state-of-the-art.

Computational Times: It is also worthwhile to mention a few computational results. The total time of the MultiSearch is 3378 seconds in average, which

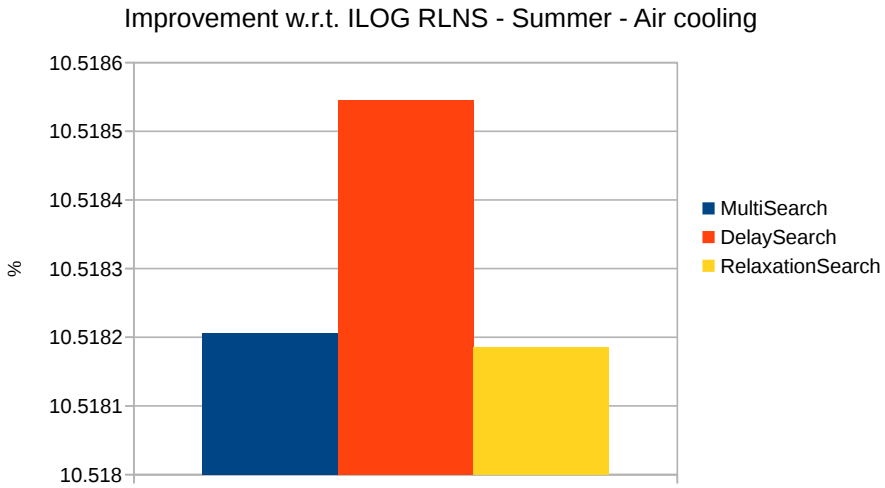

Fig. 5. Profit Improvement: Scenario: Summer Temperatures and Air Cooling.

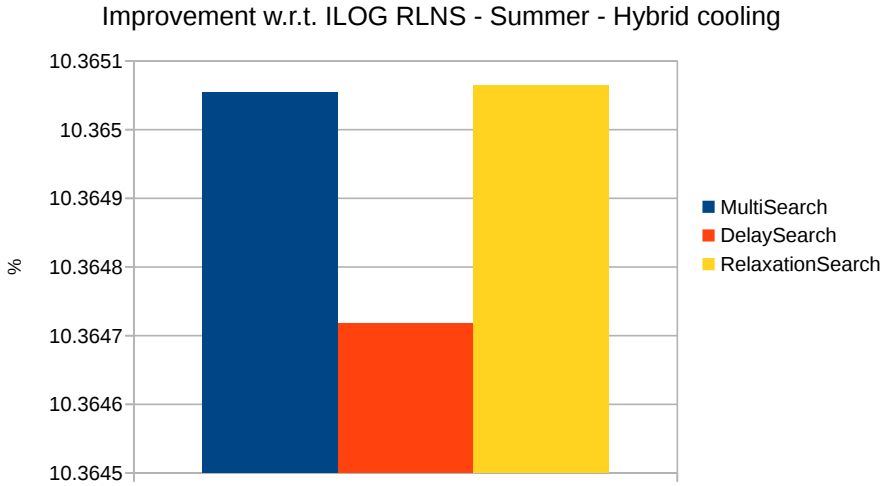

Fig. 6. Profit Improvement: Summer Temperatures and Hybrid Cooling.

would give a parallel time of about 300 seconds on 11 processors.. The average to the best solution of the DelaySearch is 601 seconds (not counting the MultiSearch time) and it improves the initial solution for 48% of the days. The average time to the best solution for the RelaxationSearch is about 75 seconds and it improves the initial solution in 8.5% of the days. These results show that opti-

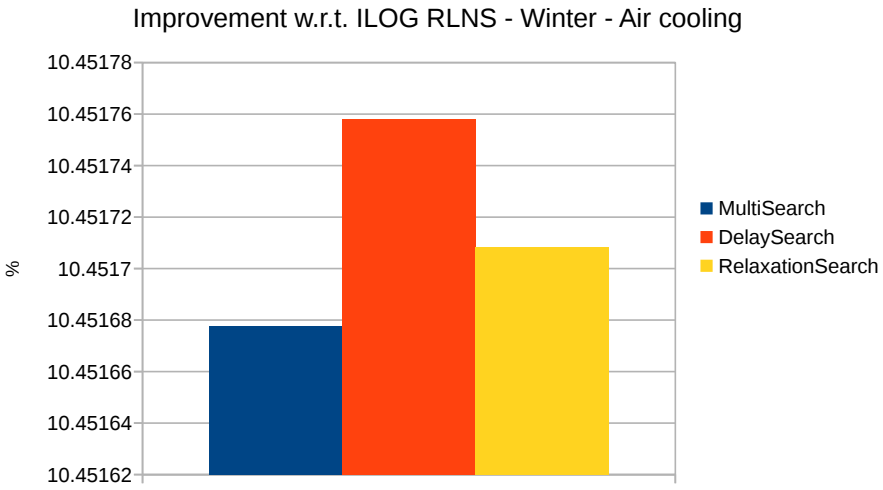

Fig. 7. Profit Improvement: Winter Temperatures and Air Cooling.

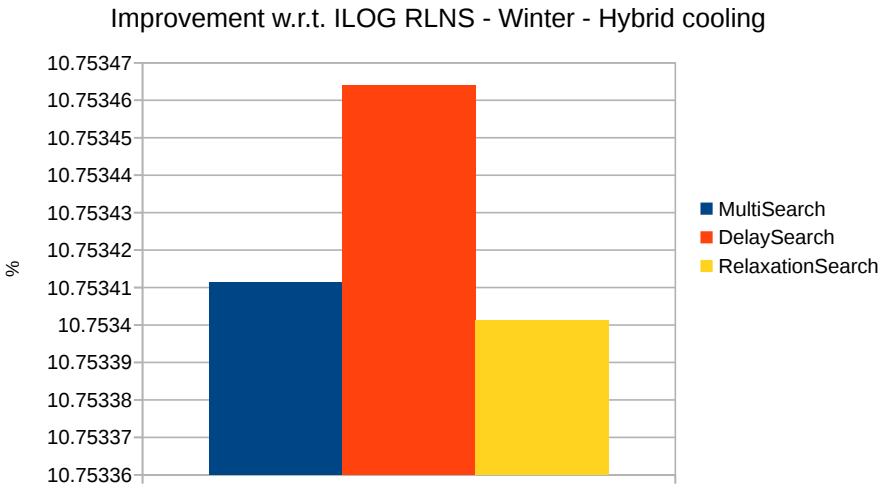

Fig. 8. Profit Improvement: Winter Temperatures and Hybrid Cooling.

mization technology has a significant opportunity to produce further significant improvements on rule-based schedulers.

Table 8 shows a comparison on the efficiency on the proposed search strategies. Each column show the percentage of improvement compared to the best rule-based scheduler divided by the time to solution. Results show that the MultiSearch gives the best improvement per seconds with a 0.0031 of average

Cooling	Season	Impr./seconds MultiSearch	Impr./seconds DelaySearch	Impr./seconds RelaxationSearch
Air	Winter	0.0030658	0.0000001	0.0000003
	Summer	0.0031243	0.0000005	-0.0000003
Hybrid	Winter	0.0031372	0.0000001	-0.0000001
	Summer	0.0031320	-0.0000009	0.0000002

Table 3. Search Efficiency: Percentage of Improvement over Solution Time.

improvement per second of computation. The DelaySearch (when it can improve the results by the MultiSearch) has an average efficiency of $2.3e^{-7}$ percent per seconds and the RelaxationSearch an average improvement of $3.5e^{-7}$ percent per seconds. However, The DelaySearch can improve the result obtained by the MultiSearch in many more scenarios.

9 Conclusion

Results in recent years have demonstrated the importance of the job scheduling in High-Performance Computing (HPC). However, the application of optimization models in HPC is challenging due to the large-scale nature of problem, This paper considers a high-fidelity scheduling model arising in HPC clusters that maximizes profit over time taking into account job revenues, energy costs, and cooling costs, building on results by Conficoni et al. [8]. The paper proposed a constraint program for this task, a novel heuristic based constraint programming, and three search strategies. Our proposals are compared to state-of-the-art rule-based schedulers and a commercial CP. Experimental results on a real case study shows that our approach improves profits by more than 7% over rule-based scheduler and 10% over the default search of the commercial solver. These results indicate the potential of constraint programming in scheduling HPC clusters.

This paper focused in the offline scheduler. Our future work will study the cooperation of our proposal within an online and reactive approach. Our goal is to obtain better solutions than a pure reactive approach within acceptable running times. Given the efficiency and solution quality of our MultiSearch algorithm, there is hope that constraint programming can play a role in an online and reactive setting as well.

References

1. Archive, P.W.: `http://www.cs.huji.ac.il/labs/parallel/workload/l_cea_curie/index.html` (2012)
2. Baptiste, P., Laborie, P., Le Pape, C., Nuijten, W.: Constraint-based scheduling and planning. Foundations of Artificial Intelligence 2, 761–799 (2006)

3. Bartolini, A., Borghesi, A., Bridi, T., Lombardi, M., Milano, M.: Proactive workload dispatching on the EURORA supercomputer. In: O'Sullivan, B. (ed.) Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8656, pp. 765–780. Springer, Springer International Publishing (2014)
4. Bridi, T., Bartolini, A., Lombardi, M., Milano, M., Benini, L.: A constraint programming scheduler for heterogeneous high-performance computing machines. IEEE Transactions on Parallel and Distributed Systems 27(10), 2781–2794 (oct 2016), http://dx.doi.org/10.1109/TPDS.2016.2516997
5. Chlumsky, V., Klusácek, D., Ruda, M.: The extension of torque scheduler allowing the use of planning and optimization in grids. Computer Science 13, 5–19 (2012)
6. CINECA: Cineca new accounting policy. https://wiki.u-gov.it/confluence/pages/viewpage.action?pageId=64201371 (2017)
7. Computing, A., Computing, G.: Torque resource manager. online] http://www.adaptivecomputing. com (2012)
8. Conficoni, C., Bartolini, A., Tilli, A., Cavazzoni, C., Benini, L.: Integrated energy-aware management of supercomputer hybrid cooling systems. IEEE Transactions on Industrial Informatics 12(4), 1299–1311 (2016)
9. Hurley, B., OSullivan, B., Simonis, H.: Icon loop energy show case. In: Data Mining and Constraint Programming, pp. 334–347. Springer (2016)
10. Salot, P.: A survey of various scheduling algorithm in cloud computing environment. International Journal of research and engineering Technology (IJRET), ISSN pp. 2319–1163 (2013)
11. TOP500: https://www.top500.org/system/177003 (2017)
12. Uddin, M., Alsaqour, R., Shah, A., Saba, T.: Power usage effectiveness metrics to measure efficiency and performance of data centers. Applied Mathematics & Information Sciences 8(5), 2207 (2014)
13. Works, P.: Pbs professional 12.2, administrators guide, november 2013 (2013)
14. Yang, X., Zhou, Z., Wallace, S., Lan, Z., Tang, W., Coghlan, S., Papka, M.E.: 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. p. 60. ACM (2013)
15. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Job Scheduling Strategies for Parallel Processing. pp. 44–60. Springer (2003)