A Job Dispatcher for Large and Heterogeneous HPC Systems Running Modern Applications

Cristian Galleguillos
Pontificia Universidad Católica de Valparaíso, Chile
University of Bologna, Italy

Zeynep Kiziltan
University of Bologna, Italy

Ricardo Soto
Pontificia Universidad Católica de Valparaíso, Chile

Abstract

High-performance Computing (HPC) systems have become essential instruments in our modern society. As they get closer to exascale performance, HPC systems become larger in size and more heterogeneous in their computing resources. With recent advances in AI, HPC systems are also increasingly being used for applications that employ many short jobs with strict timing requirements. HPC job dispatchers need to therefore adopt techniques to go beyond the capabilities of those developed for small or homogeneous systems, or for traditional compute-intensive applications. In this paper, we present a job dispatcher suitable for today's large and heterogeneous systems running modern applications. Unlike its predecessors, our dispatcher solves the entire dispatching problem using Constraint Programming (CP) with a model size independent of the system size. Experimental results based on a simulation study show that our approach can bring about significant performance gains over the existing CP-based dispatchers in a large or heterogeneous system.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases Constraint programming, HPC systems, heterogeneous systems, large systems, on-line job dispatching, resource allocation

Digital Object Identifier 10.4230/LIPIcs.CP.2021.26

Related Version *Previous Version*: `https://arxiv.org/abs/2009.10348`

Supplementary Material *Software (Source Code)*: `https://git.io/fjia1`
archived at `swh:1:dir:763b27390dd764a27ae8e7dff5dc0d724cbabe88`

Funding *Cristian Galleguillos*: Funded by Postgraduate Grant INF-PUCV 2020.

Acknowledgements We thank A. Bartolini, L. Benini, M. Milano, M. Lombardi and the SCAI group at Cineca for providing the Eurora data, and A. Borghesi and T. Bridi for sharing the implementations of the original CP-based dispatchers. We also thank the School of Computer Engineering of PUCV in Chile for providing access to computing resources.

1 Introduction

Motivations

High Performance Computing (HPC) is the application of supercomputers to solve complex computational problems, which has become indispensable for scientific progress, industrial competitiveness, economic growth and quality of life in our modern society [18, 22]. An HPC system is a network of computing nodes, each containing several powerful CPUs and a large pool of memory. The world's fastest systems today can reach hundreds of petaFLOPs (10^{15} floating-point operations per second) and they are expected to reach soon the exaFLOP level (10^{18} FLOPs) [16]. Indeed, today's most powerful system Fugaku has recently increased

© Cristian Galleguillos, Zeynep Kiziltan, and Ricardo Soto;
licensed under Creative Commons License CC-BY 4.0

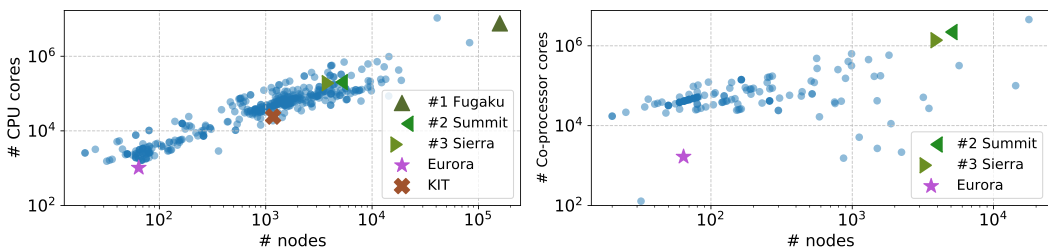


Figure 1 Eurora, KIT ForHLR II and top 500 HPC systems of June 2021.

its performance on a mixed-precision HPC-AI benchmark to 2 exaFLOPs.[1] In their march towards this elevated performance, HPC systems are getting larger in size and becoming more heterogeneous in their computing resources in an effort to keep the power consumption at bay. Figure 1 shows in blue dots the size of today's top 500 systems[1] and their number of CPU cores and co-processor cores (with the green triangles referring to the top 3 systems). The majority of these systems have thousands of nodes with tens and hundreds of thousands of CPU cores in total. Around 30% of them employ energy-efficient accelerators such as GPUs and Many Integrated Cores (MICs), in addition to the traditional CPUs and memory. The number of co-processor cores are above ten thousand in most of such systems.

Central to the efficiency and the Quality-of-Service (QoS) of an HPC system is the *job dispatcher* which decides the jobs to run next among those waiting in the queue (*scheduling*) and on which resources to run them (*allocation*). This is an on-line decision making problem because the process is repeated periodically as new jobs arrive to the system while some previously dispatched jobs are still running. Traditionally, HPC job dispatchers have been designed for compute-intensive jobs requiring days to complete. There is an increasing trend where HPC systems are being used for modern applications that employ many short jobs (< 1 h), such as data analytics as data is being streamed from a monitored system [23]. In such application scenarios, response times are critical for acceptable user experience, hence job dispatchers need to rapidly process a large number of short jobs in making on-line decisions. Though optimal dispatching is a critical requirement in HPC systems, the on-line job dispatching is an NP-hard optimization problem [5].

In this paper, we propose an on-line job dispatcher suitable for today's large and heterogeneous HPC systems running modern applications. Differently from the existing techniques based on heuristic algorithms [11, 27], we exploit the power of Constraint Programming (CP), which has a long track record of success in job scheduling and resource allocation problems [2]. While past work had already used CP in this context, the focus was on small or homogeneous systems where the nodes have only CPUs and memory.

Related work

The first CP-based HPC dispatcher was introduced in [3] and shown to obtain better solutions compared to a Priority Rule-Based (PRB) dispatcher [9, 15], which is widely adopted in commercial HPC workload management systems such as Altair PBS Professional [1] and SLURM Workload Manager [25]. The dispatcher was later embedded as a plug-in within the software framework of PBS professional [8]. Another CP-based dispatcher with the additional feature of limiting system power consumption was presented in [7, 6] and proved to outperform a PRB dispatcher on the instances with tight power capping values.

[1] https://www.top500.org

Subsequently, [13] presented two CP-based on-line job dispatchers for HPC systems, which we here refer to as PCP'19 and HCP'19. They were built on the previous CP-based dispatchers [3, 7] and redesigned for satisfying the challenges of systems running modern applications that employ many short jobs and that have strict timing requirements. A simulation study [13] based on a workload trace collected from an heterogeneous system Eurora [10] reveals that PCP'19 and HCP'19 yield substantial improvements over the original dispatchers [3, 7] and provide a better QoS compared to Eurora's dispatcher [17], which is a part of PBS Professional.

PCP'19 and HCP'19 are, however, not designed for today's large and heterogeneous systems. In PCP'19, the number of decision variables in the CP model increases proportionally to the number of nodes and the possible allocations of jobs in each node. Figure 1 shows where the Eurora system stands compared to today's top 500 systems. As we will show in our experimental results, PCP'19 cannot be used in a larger system like KIT ForHLR II[2], whose size is comparable to that of the majority of the top systems.

In HCP'19 instead, the problem is decoupled into scheduling and allocation problems. Only the scheduling problem is addressed using CP, and this is done without representing the nodes in the model by treating the resources of the same type across all nodes as a pool of resources. The allocation problem is then solved with a heuristic algorithm using the best-fit strategy [24], while fixing any inconsistencies introduced during scheduling due to the absence of the nodes in the model. The decoupled approach drops the number of decision variables dramatically compared to PCP'19, enabling HCP'19 to scale to larger systems. However, it mainly suits to homogeneous systems where all the nodes have only CPUs and memory, and thus the actual node of an allocated resource is not relevant. In an heterogeneous system, on the contrary, some nodes contain scarce resource types, such as GPUs and MICs, and allocating their CPUs carelessly (i.e., to jobs that do not need any of GPUs and MICs) may cause resource fragmentation [20]. The decoupled approach therefore may result in several iterations between scheduling and allocation in an heterogeneous system, decreasing the dispatching performance, as we will show in our experimental results with Eurora. The advantages of tackling the entire problem using CP, as was done in PCP'19, are that scheduling and allocations decisions are made jointly and that with the presence of nodes in the model, allocation strategies dedicated for heterogeneous systems [20] can be encoded as constraints.

Contributions

We exploit the strengths of PCP'19 and HCP'19 to overcome their limitations. We tackle the entire dispatching problem using CP, and to do that we present a new allocation model where the number of variables is system size independent. We combine this model with the scheduling model common to PCP'19 and HCP'19, and showcase the practical value of our approach. Our contributions are (i) a new HPC application domain emerging from today's large and heterogeneous systems to push the limits of complete methods for optimization, (ii) a novel CP-based online job dispatcher (PCP'21) suitable for such systems (iii) experimental evidence of the benefits of PCP'21 over PCP'19 and HCP'19 supported by a simulation study based on workload traces collected from the Eurora and KIT ForHLR II systems.

[2] https://www.scc.kit.edu/dienste/forhlr2.php

Organization

The rest of the paper is organized as follows. In Section 2, we introduce the on-line job dispatching problem in HPC systems, and describe briefly the CP scheduling and allocation models of PCP'19 as we will later use the same scheduling model in PCP'21 and contrast the allocation model with ours. In Section 3, we present our new CP allocation model and search algorithm. In Sections 4 and 5, we detail our simulation study and present our results. We conclude and describe the future work in Section 6.

2 Formal Background

2.1 On-line job dispatching problem in HPC systems

A *job* is a user request in an HPC system and consists of the execution of a computational application over the system resources. A set of jobs is a *workload*. A job has a name, required resource types (cores, memory, etc) to run the corresponding application, and an *expected duration* which is the maximum time it is allowed to execute on the system. An HPC system typically receives multiple jobs simultaneously from different users and places them in a *queue* together with the other waiting jobs (if there are any). The *waiting time* of a job is the time interval during which the job remains in the queue until its execution time.

An HPC system has N nodes, with each node $n \in N$ having a capacity $cap_{n,r}$ for each resource type $r \in R$. Each job i in the queue Q has an arrival time q_i, maximum number of requested nodes rn_i and a demand $req_{i,r}$ giving the amount of resources required from r during its expected duration d_i. The resource request of i is distributed among rn_i identical job units, preserving for each one $req_{i,r}/rn_i$ amount of resources from r, thus allowing to execute the rn_i job units in parallel. Job units can be tasks that are spanned across multiple nodes and that communicate between them during their entire execution (for instance an MPI job). A specific resource can be used by one job unit only. We have $rn_i = 1$ for serial jobs and $rn_i > 1$ for parallel jobs. The units of a job can be allocated on the same or different nodes, depending on the system availability. On-line job dispatching takes place at a specific time t for (a subset of) the queued jobs Q. The *on-line job dispatching problem* at a time t consists in *scheduling* each job i by assigning it a start time $s_i \geq t$, and *allocating* i to the requested resources during its expected duration d_i, such that the capacity constraints are satisfied: at any time in the schedule, the capacity $cap_{n,r}$ of a resource r is not exceeded by the total demand $req_{i,r}$ of the jobs i allocated on it, taking into account the presence of jobs already in execution. The objective is to dispatch in the best possible way according a measure of QoS, such as with minimum waiting times $s_i - q_i$ or the slowdown ($\frac{s_i - q_i + d_i}{d_i}$) for the jobs, which is directly perceived by the HPC users. A solution to the problem is a *dispatching decision*. Once the problem is solved, only the jobs with $s_i = t$ are dispatched. The remaining jobs with $s_i > t$ are queued again with their original q_i. During execution, a job exceeding its expected duration is killed. It is the workload management system that decides the dispatching time t and the subsequent dispatching times.

2.2 PCP'19 dispatcher

Scheduling model

The scheduling problem is modeled using Conditional Interval Variables (CIVs) [19]. A CIV $\tau_i \in \tau$ represents a job i and defines the time interval during which i runs. At a dispatching time t, there may already be jobs in execution which were previously scheduled and allocated.

We refer to such jobs as running jobs. The scheduling model considers in the τ variables both the running jobs and a subset $\bar{Q} \subseteq Q$ of the queued jobs that can start execution as of time t. The properties $s(\tau_i)$ and $d(\tau_i)$ correspond respectively to the start time and the duration of the job i. All job units of a job i start at the same time, therefore they share the same τ_i. Since the actual runtime (real) duration d_i^r of a running or queued job i is unknown at the modeling time, PCP'19 uses an *expected duration* d_i for $d(\tau_i)$, which is supplied by a job duration *prediction method*. For the queued jobs, we have $d(\tau_i) = d_i$. For the running jobs instead, $d(\tau_i) = max(1, s(\tau_i) + d_i - t)$ taking into account the possibility that $d_i < d_i^r$ due to underestimation. While the start time of the running jobs have already been decided, the queued jobs have $s(\tau_i) \in [t, eoh]$, where eoh is the end of the worst-case makespan calculated as $t + \sum_{\tau_i} d(\tau_i)$.

The capacity constraints are enforced via `cumulative`$([s(\tau_i)], [d(\tau_i)], [req_{i,r}], Tcap_r)$, for all $n \in N$ and for all $r \in R$, with $Tcap_r = \sum_n^N cap_{n,r}$. The objective function minimizes the total job slowdown $\sum_{\tau_i} \frac{s(\tau_i)-q_i+d(\tau_i)}{d(\tau_i)}$. The search for solutions focuses on the jobs with highest priority where the *priority* of a job i is its slowdown $\frac{t-q_i+d(\tau_i)}{d(\tau_i)}$ at the dispatching time t. We note that HCP'19 uses the same scheduling model and search.

Allocation model

The allocation model replicates each τ_i variable $p_{i,n}$ times for each $n \in N$, where $p_{i,n} = min(rn_i, \min_{r \in R} \lfloor \frac{cap_{n,r}}{req_{i,r}/rn_i} \rfloor)$ giving the minimum times a job unit can fit on n. The variable $u_{i,n,j}$ represents a possible allocation of a job unit j of i on node n and has $s(u_{i,n,j}) = s(\tau_i)$ and $d(u_{i,n,j}) = d(\tau_i)$. To define the allocation, the model relies on the execution state property (x) of CIVs. We have $x(u_{i,n,j}) \in [0, 1]$, meaning that it can be present or not in the solution. Instead for the scheduling variables we have $x(\tau_i) = 1$ because all of them need to be scheduled and thus be present in the solution. The model uses the `alternative` constraint [19] to restrict the number of variables in $\cup_{n \in N}[x(u_{i,n,j})]$ present in the solution to be the maximum number of requested nodes rn_i, that is $\sum_{n \in N} \sum_j x(u_{i,n,j}) = rn_i$ with $s(\tau_i) = s(u_{i,n,j})$ iff $x(u_{i,n,j}) = 1$. Additionally, the capacity constraints are enforced for each $n \in N$ and for each $r \in R$ as `cumulative`$([s(u_{i,n,j})], [d(u_{i,n,j})], [req_{i,r}/rn], cap_{n,r})$.

A drawback of this model is its number of variables. While the scheduling model has $|\bar{Q}|$ variables, the allocation model has $\sum_{i \in \bar{Q}} \sum_{n \in N} p_{i,n}$ variables, which increases proportionally to N (i.e., system size). A minimum of $1 + |N|$ variables are needed to model a serial job. Parallel jobs will require even more variables which may create difficulty in large systems with many parallel jobs.

3 PCP'21: a New CP-based Job Dispatcher

Our dispatcher PCP'21 imports the scheduling model, the objective function and the job priorities of PCP'19 and contains a new allocation model with $|\bar{Q}| + \sum_{i \in \bar{Q}} rn_i * |R|$ variables, which is system size independent. The number of variables thus depends on the number of resource types (which is a small value) multiplied by the sum of the requested nodes (which is usually much smaller than the system size) of all jobs in $\bar{Q}$ (which is a fixed value). Next, we present the allocation model and describe how we search on the scheduling and the allocation variables.

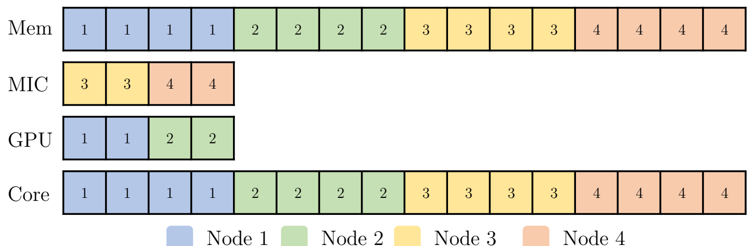


Figure 2 Representation of an example system.

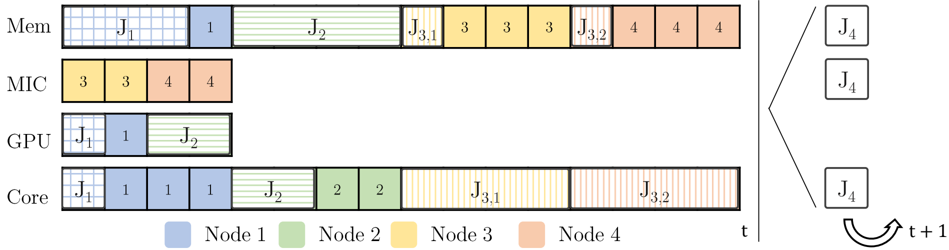


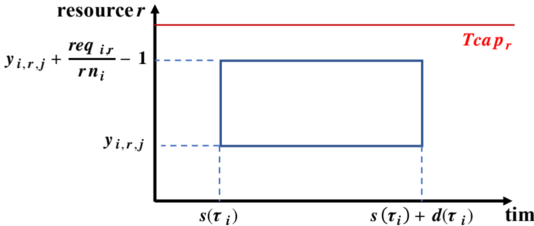
Figure 3 Allocation of some jobs on the example system at a dispatching time t.

Allocation model

In this new model, we represent the system in a way to emphasize the resources instead of the nodes as in the previous model. We consider all the resources of a certain type r in an ordered array by following the sequence of the nodes. This is exemplified in Figure 2 which represents a system with 4 nodes. Each node has 4 cores and 4 units of memory. The first two nodes have 2 GPUs, and the next two has 2 MICs. The array labelled as GPU, for instance, lists all the GPU resources available in the system. There are in total $2 * 2$ GPUs, the first two in the array are from the first node, the third and the forth from the second node. An array position refers to a specific resource of type r in a node n, which is highlighted with a colour and a number in Figure 2.

Let us assume that, at a dispatching time t, a job J_1 is still running, and three more jobs J_2, J_3, J_4 are extracted from the queue. As for their resource requests, let us assume that J_1 requires 3 memory units, 1 GPU and 1 core; J_2 4 memory units, 2 GPUs and 2 cores; J_3 2 memory units and 8 cores via two job units $J_{3,1}$ and $J_{3,2}$; and J_4 1 memory unit, 1 MIC and 1 core. Figure 3 shows a possible allocation of these jobs in the system after a dispatcher call at time t. The running job J_1 is allocated to the minimum positions available in the arrays corresponding to the requested resource types, hence it is allocated in node 1 (the first available node), occupying the first 3 memory, the first GPU and the first core positions in the corresponding arrays. Since J_2 requires two GPUs, it is allocated in the second node, occupying all the memory and GPU and the first two core positions in the green parts of the resp. arrays. The job units of J_3 do not fit in the same node, so they are equally distributed to the next two nodes, each occupying the first memory and all the core positions of the yellow and orange parts of the resp. arrays. As for J_4, it can be allocated only in the last two nodes, because it needs an MIC. As the cores of these nodes are all occupied, J_4 cannot be allocated and is postponed to the next dispatching time $t + 1$.

Following this representation, we model the positions of a job unit j of a queued job i on a resource type r via the variables $y_{i,r,j}$. As we also need to represent the time during which the allocation is valid, we use a two-dimensional box to model an allocation, as depicted in



$y_{i,r,j}$	Mem	MIC	GPU	Core
J_1	1	-	1	1
J_2	5	-	3	5
$J_{3,1}$	9	-	-	9
$J_{3,2}$	13	-	-	13
J_4	-	-	-	-

Figure 4 Modelling the allocation of a job unit j of a job i on a resource type r (left), and the assignment of the $y_{i,r,j}$ variables at time t in the example allocation (right).

Figure 4 (left). The y-axis gives the available positions and the x-axis gives the time interval during which the resource is consumed. The vertices of the box are defined by the variables in the origin: $s(\tau_i)$ which is the starting time of the job i and $y_{i,r,j}$ which is the starting position of the allocation. The box spans from the origin by the expected duration $d(\tau_i)$ in the x-axis, instead in the y-axis by $req_{i,r}/rn_i$ which is the required resource amount. As for the domains, we have $D(y_{i,r,j}) = [1, Tcap_r]$, where $Tcap_r = \sum_{n \in N} cap_{n,r}$. The domain of the starting time is the same as in the scheduling model, that is $D(s(\tau_i)) = [t, eoh]$.

Figure 4 (right) shows the assignment of the $y_{i,r,j}$ variables at time t in our example allocation. Take for instance J_2 which has one job unit (itself) and requires 4 memory units, 2 GPU and 2 cores. As we saw in Figure 3, it occupies in the Memory array the 5th to the 9th positions, in the GPU array the 3rd to the 5th, and in the Core array the 5th to the 7th. Consequently, the corresponding $y_{i,r,j}$ variables are assigned to the starting positions 5, 3 and 5, and the relative boxes span in the y-axis to the last positions 9, 5, and 7, respectively.

We also need to model the running jobs. To a job unit j of a job i on a resource type r, which was previously assigned the resources of a certain node, we now assign the minimum available position among those that refer to the same node. We have already exemplified this with J_1 in Figure 3. These resources are allocated to i during its $d(\tau_i)$. If multiple job units are assigned to the same node, the resources are occupied consecutively, leaving the higher indices free.

To enforce that a resource is used by one job unit only, we forbid the boxes to overlap via the `diffn` constraint [4]. For each $r \in R$, we have `diffn`$([s(\tau_i)], [d(\tau_i)], [y_{i,r,j}], [req_{i,r}/rn_i])$. As the domain size of the $y_{i,r,j}$ variables depends on the system size and can be large, we add implied constraints to shrink them. They are the classical `cumulative` constraints used together with a `diffn` constraint in packing problems, as was also done in [4]: `cumulative`$([s(\tau_i)], [d(\tau_i)], [req_{i,r}/rn_i], Tcap_r)$ and `cumulative`$([y_{i,r,j}], [req_{i,r}/rn_i], [s(\tau_i)], eoh)$. Given that the jobs units of jobs with $rn_i > 1$ have identical resource requests, their allocations are symmetric. We post an ordering constraint on the positions of the jobs unit of a job i on a resource type r to break symmetry: $y_{i,r,j} < y_{i,r,j+1}$ It is a strict ordering as the position variables take different values.

Finally, we need additional constraints to guarantee that certain allocations are in the same node. For that, we utilize a mapping array map_r for each resource type r, which is based on the system representation introduced earlier. The positions of map_r correspond to the available resources, indexed by 1 to $Tcap_r = \sum_{n \in N} cap_{n,r}$, and each value in the array is a number corresponding to a node. To ensure that the allocated resources of a job unit are in the same node, we post an `element` constraint, which indexes an array with a variable, as `element`$(map_{r_1}, y_{i,r_1,j}) =$ `element`$(map_{r_2}, y_{i,r_2,j})$ $\forall r_1, r_2 \in \hat{R}$, where $\hat{R}$ is the set of the

requested resource types of the job unit j of job i. We use the `element` constraint also to enforce that the positions spanning from $y_{i,r,j}$ to $y_{i,r,j}+(req_{i,r}/rn_i)-1$ refer to the same node: $\texttt{element}(map_r, y_{i,r,j}) = \texttt{element}(map_r, y_{i,r,j} + (req_{i,r}/rn_i) - 1)\ \forall r \in \hat{R}$ iff $req_{i,r}/rn_i > 0$.

Search

Similarly to PCP'19 and HCP'19, we use a custom search algorithm derived from the schedule-or-postpone algorithm [21] to search on the scheduling variables $s(\tau_i)$. At each decision node, we select the job i whose priority is highest and that can start first, and assign to $s(\tau_i)$ its earliest start time $min(s(\tau_i))$. Note that the priorities are calculated once statically at the dispatching time t before search starts.

Differently from PCP'19 and HCP'19, we interleave the scheduling and the allocation assignments of a selected job i. After assigning a scheduling variable $s(\tau_i)$, we search on the allocation variables $[y_{i,r,j}]$ of i. We start with the resource r which has the lowest availability at time t. Then we search on $[y_{i,r,j}]$ in lexicographical order and assign them their minimum values $min(y_{i,r,j})$, which guarantees consistency with the symmetry breaking constraints on the allocation variables.

Even though we have designed PCP'21 for systems engaged with heterogeneous workloads, an heterogeneous system may as well receive a workload with homogeneous resource requests (i.e., only CPU and memory) which creates symmetry among the requested resources. We adapt the search algorithm to break symmetry in such a scenario as follows. After a resource allocation attempt for a job i, if the search fails or wants to find a better solution, it backtracks to the scheduling variable $s(\tau_i)$, as opposed to backtracking within the $[y_{i,r,j}]$ variables.

Following PCP'19 and HCP'19, search is bounded by a time limit δ due to the problem complexity. Thus, the best solution returned within the limit is the dispatching decision. If, however, no satisfiability answer is obtained within the limit, the time limit is extended to $2 * \delta$, as opposed to restarting search with the new time limit $2 * \delta$ as was done in PCP'19 and HCP'19. This procedure continues until the time limit reaches δ_{max}. We suspend the search if the solution quality did not change after k consecutive time limit extensions.

4 Experimental Study

To evaluate the significance of our approach, we conducted an experimental study by simulating on-line job submission to two HPC systems. We dispatched jobs using PCP'21, PCP'19, HCP'19, and sought answers to the following questions: (1) how do the dispatchers compare when they are engaged in a workload with heterogeneous resource requests? (2) can PCP'19 and PCP'21 scale to a large system? As we said earlier, an heterogeneous system may as well receive a workload with homogeneous resource requests. We thus sought an answer also to the following question: (3) how much do we lose by using PCP'21 for a workload with homogeneous resource requests compared to using HCP'19 which is more suitable for an homogeneous system? Before we present the answers in Section 5, we describe in this section the ingredients of our experimental study.

HPC systems and workload datasets

Our study is based on two different workload traces collected from two different HPC systems. The first system is the Eurora [10], which was in production at CINECA datacenter in Italy until 2015. With 64 nodes, the system size is small compared to the current trend (see Figure 1), but the architecture is heterogeneous with each node containing 2 octa-core CPUs,

16 GB memory, and two of GPU or MIC. To answer the first question, we use the workload dataset with which PCP'19 and HCP'19 were tested in [13]. It consists of logs over 400,000 jobs submitted during the time period March 2014–August 2015 and is dominated by short jobs, making up 93.14% of all the jobs. As for resource requests, 22.8% of the jobs require only CPU and memory while 77.2% need in addition one of GPU or MIC.

The second system is the KIT ForHLR II[2], located at Karlsruhe Institue of Technology in Germany. We use this system to answer the second question because it has 1,173 nodes, a size comparable to the current trend (see Figure 1). 1,152 of these nodes are thin, each equipped with 20 cores and 64 GB memory, and the remaining 21 are fat each containing 48 cores, 4 GPUs, and 1 TB memory. Even though a small fraction of the nodes contain GPUs, we use a workload with homogeneous resource requests to answer also the third question. The workload dataset is available on-line. [3] It contains logs for 114,355 jobs submitted during the time period June 2016–January 2018. All the jobs require only CPU and memory, and 66.26% of them are short (< 1h).

Job duration prediction

We derived the expected durations d_i of jobs via three prediction methods. The first is a data-driven heuristic first proposed in [14] and later was shown to work well with PCP'19 and HCP'19 when simulating the Eurora dataset [13]. The heuristic constructs job profiles from the workload. Prediction is based on the observation that jobs with similar profiles have the same duration for long periods of time. For each job, the heuristic searches for the last job with a similar profile, and uses its duration to predict the duration of the new one. Each user is analyzed separately. The similar profile is identified using a set of rules. If all rules fail, then the user-declared wall-time is taken as the predicted duration. In all cases, the prediction is capped by the wall-time.

Despite being a valid alternative, this method relies on job names, a type of data omitted in the KIT ForHLR II and some other public datasets. We thus employed a second heuristic method that uses the run times of the last two jobs to predict the duration of the next job [26]. In both methods, the predictions are calculated on-line during the simulation and the knowledge base is updated upon job termination. The last prediction method is an oracle which gives the actual runtime (real) durations d_i^r and provides a baseline during the simulation of both datasets.

Simulation

We used the AccaSim workload management system simulator [12] to simulate the HPC systems with their workload datasets. Each job submission is simulated by using its available data, for instance, the owner, the requested resources, and the real duration, the execution command or the name of the application executed. AccaSim uses the real duration to simulate the job execution during its entire duration. Therefore job duration prediction errors do not affect the running time of the jobs with respect to the real workload data. The dispatchers are implemented using the AccaSim directives to allow them to generate the dispatching decisions during the system simulation.

[3] `https://www.cse.huji.ac.il/labs/parallel/workload/logs.html`

Table 1 Times obtained from the Eurora system.

Dispatcher	Avg. disp. time [ms]	Total sim. time [s]
HCP'19-D	392	208,231
PCP'19-D	511	271,586
PCP'21-D	209	111,373
HCP'19-R	357	189,522
PCP'19-R	469	249,367
PCP'21-R	256	136,401

Experimental settings

As a CP modelling and solving toolkit, we customized Google OR-Tools[4] 7.3 by implementing the `alternative` constraint and the proposed search algorithm and by making visible some variables of the solver, and ported it to Python 3.6 to implement PCP'21 in AccaSim. As for PCP'19 and HCP'19, we used their publicly available implementations[5], and carried over their parameters $m = 100$, $\delta = 1s$, $\delta_{max} = 16s$, $k = 2$. For the simulation of the KIT ForHLR II workload, which has only homogeneous resource requests, we adapted the search algorithm of PCP'21 to break symmetry among the requested resources as described in Section 3. We refer to this version of PCP'21 as PCP'21_s in the experimental results. All experiments were performed on a CentOS machine equipped with Intel Xeon CPU E5-2640 Processor and 16GB of RAM. The source code is publicly available at `https://git.io/fjia1`.

5 Experimental Results

In this section, we show our experimental results. In each simulation, we compare the dispatchers' performance (in Tables 1 and 2) in terms of (i) the average CPU time spent in generating a dispatching decision over all dispatcher invocations, including the time for modeling the dispatching problem instance and searching for a solution, and (ii) the total simulation time from the first job submission until the last job completion. We also compare the dispatchers' QoS (in Figures 5 and 10) in terms of the average slowdown and waiting times of the jobs. To refer to a dispatcher using a certain job duration prediction method, we append -D, -L2 or -R to the name of the dispatcher for the data-driven heuristic, the last-two heuristic and the real duration, respectively.

5.1 Simulation of the Eurora workload

We remind that we simulate a system like Eurora to compare all the dispatchers when they are engaged in a workload with heterogeneous resource requests. All the dispatchers complete the simulation. Comparing their performance in Table 1, we can clearly see the benefits of using PCP'21. With the decoupled approach of HCP'19, the performance drops almost by half (around 47%) when using -D. We observe a further performance decrease (around 59%) with PCP'19, which could be attributed to its higher number of decisions variables. PCP'21 is the most efficient dispatcher also when using -R, with gains around 28% and 45% compared to HCP'19 and PCP'19, resp.

[4] `https://developers.google.com/optimization/`
[5] `https://git.io/fjia1`

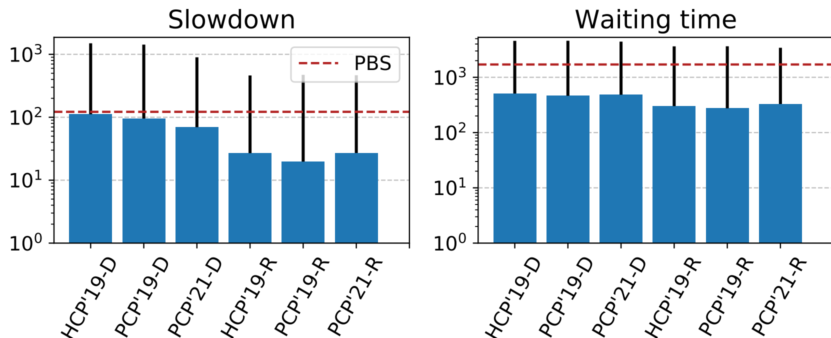


Figure 5 Average and error bars showing one std. deviation of slowdown and waiting times [s] obtained from the Eurora system.

As for the quality of decisions, we observe in Figure 5 that all dispatchers return better solutions than Eurora's PBS dispatcher. Among the CP-based dispatchers, PCP'21 results in the best average slowdown and a substantial decrease in the error when using -D. Otherwise, the dispatchers have similar (low) slowdown values when using -R and have similar waiting times when using either of job duration prediction methods.

As a side comment, we note in Table 1 that PCP'21-D performs better than PCP'21-R. Looking at Figure 5, however, we see that PCP'21-R finds better solutions than PCP'21-D within the time limit. In the instances of -D, jobs have similar durations (due to the way the data-driven prediction method works), instead in the instances of -R, jobs have a more diverse duration. The -R instances tend to be more difficult and take longer to solve, especially with PCP'21-R which uses the expected durations also in the allocation model.

Additional analysis is needed in order to quantify the reduction in the number of decision variables obtained by going from PCP'19 to PCP'21. During the simulation of an HPC system and its workload data, all dispatchers start with the same dispatching instance, but then they schedule and allocate jobs diversely. This in turn leads to different jobs running on different resources of the system as well as to different jobs waiting in the queue in the next dispatching time. We cannot therefore compare the dispatchers' model size on the distinct instances they entail throughout the simulation period. To analyze the dispatchers on the same instances, we saved the instances created during the simulation of the Eurora workload while using PCP'19-D and PCP'19-R as a dispatcher. Each instance is created when the simulator calls the corresponding dispatcher, and the instance is described by the queued jobs, the running jobs and their specific allocation on the system. We obtained in total 624,569 instances.

Figure 6 shows the ratio of the number of decision variables between PCP'21 and PCP'19 versus the percentage of the instances. For all instances, the ratio is below 0.1, proving the significance of the new allocation model. To confirm the impact on the search performance, we give in Figures 7 and 8, the ratio of the dispatching time and the number of fails. We note that while some instances are solved to optimality, some instances hit the time limit but even in that case PCP'19 and PCP'21 extend the time limit differently, as was described in Section 3. For almost all the instances, the ratio of the dispatching time is between 1 and 0.01, and the ratio of the number of fails is between 0.1 and 0, supporting the direct effect of model size on the dispatcher performance. In Figure 9, we show the ratio of the dispatching time versus the ratio of the number of fails for each individual instance. 93% and 88% of the instances fall into the region where both ratios are between 0 and 1 when using -D and -R, respectively.

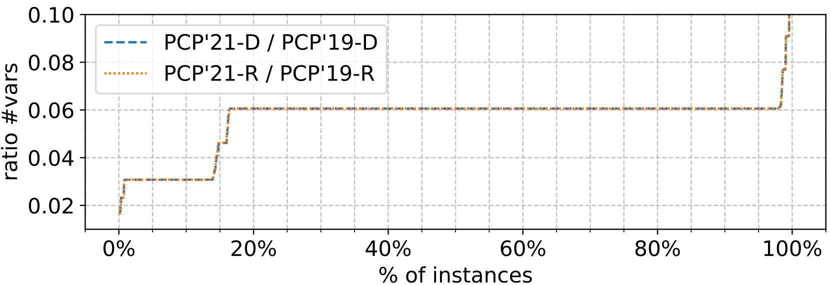


Figure 6 Ratio of the #decision variables between PCP'21 and PCP'19 on the individual Eurora instances.

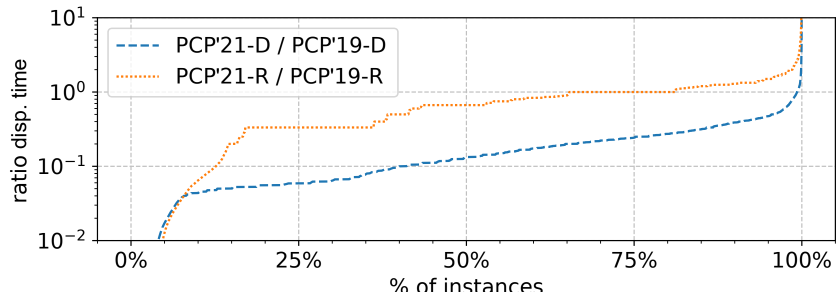


Figure 7 Ratio of the dispatching time between PCP'21 and PCP'19 on the individual Eurora instances.

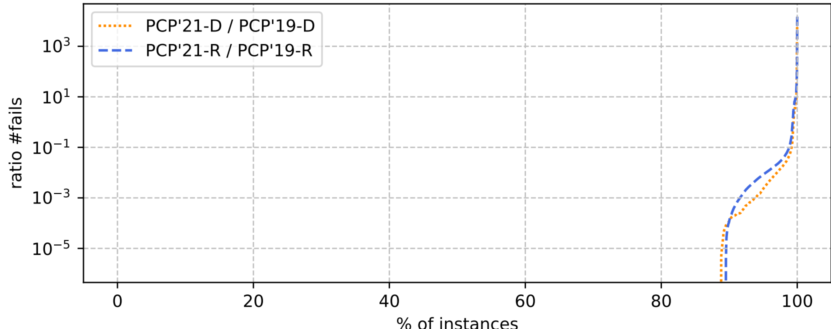


Figure 8 Ratio of the #fails between PCP'21 and PCP'19 on the individual Eurora instances.

We also analyzed the ratio of the quality of the dispatching decisions. The results (not shown here) are in line with those shown in Figure 5. The ratio is 1 for the vast majority of the instances.

5.2 Simulation of the KIT ForHLR II workload

We remind that we simulate a system like KIT ForHLR II to observe whether PCP'19 and PCP'21 can scale to a large system. We do it so by using a workload with homogeneous resource requests, because an heterogeneous system may as well receive an homogeneous workload and in that case we want to quantify any possible loss with respect to HCP'19 which is more suitable for an homogeneous system. We experiment with both PCP'21 and PCP'21$_s$ to see the importance of symmetry breaking with an homogeneous workload.

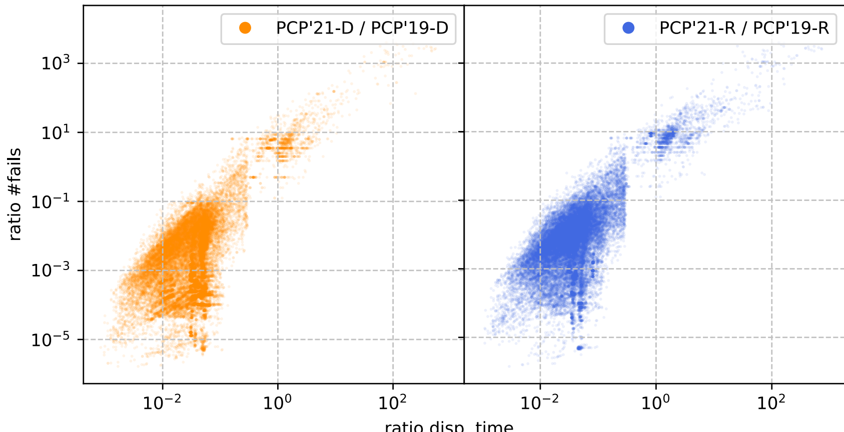


Figure 9 Ratio of the dispatching time (x-axis) vs ratio of the #fails (y-axis) between PCP'21 and PCP'19 on the individual Eurora instances.

Table 2 Times obtained from the KIT ForHLR II system.

Dispatcher	Avg. disp. time [ms]	Total sim. time [s]
HCP'19-L2	278	56,083
PCP'19-L2	∞	∞
PCP'21-L2	493	99,590
PCP'21$_s$-L2	334	67,471
HCP'19-R	269	54,289
PCP'19-R	∞	∞
PCP'21-R	476	96,030
PCP'21$_s$-R	342	69,094

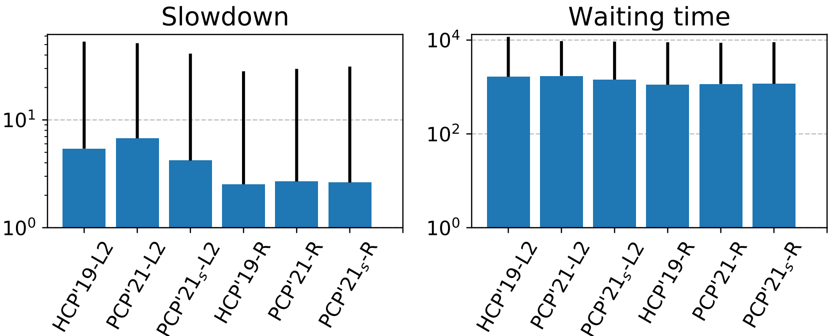


Figure 10 Average and error bars showing one std. deviation of slowdown and waiting times [s] obtained from the KIT ForHLR II system.

PCP'19 cannot complete the simulation for several days. At some point in time, it stops dispatching, even if new jobs are entering in the queue and the system is empty with all its resources available. This is because PCP'19 cannot handle certain dispatching instances within the available time limit and blocks the current and the next dispatching decisions. Instead PCP'21 and PCP'21$_s$ complete the simulation, as can be seen in Table 2, confirming its advantage to PCP'19 in a large system. We observe in the table that symmetry breaking

is crucial with an homogeneous workload. PCP'21$_s$ significantly reduces the total simulation time and average dispatching time compared to PCP'21. The performance of PCP'21$_s$ is not too far from that of HCP'19. The performance gap is around 27% and 20% when using -R and -D, resp. We can see in Figure 10 that in terms of the QoS, the dispatchers behave almost identically.

5.3 Discussion

We showed that when dispatching an Eurora workload dominated by short jobs with heterogeneous resource requests, PCP'21 is more efficient than the other dispatchers by 28% to 59%. While PCP'21 can scale to a large system like KIT ForHLR II, PCP'19 cannot. This is probably due to the high number of decision variables in the allocation model of PCP'19. Additional experiments on the individual Eurora instances generated by PCP'19 confirmed that PCP'21 substantially reduces the number of variables, the dispatching time, and the number of fails. We also argued that an heterogeneous system may as well receive an homogeneous workload. We showed that when dispatching a KIT ForHLR II workload dominated by short jobs with homogeneous resources requests, the use of an adapted version of the search algorithm that breaks symmetry among the identical resources is crucial. With this version of PCP'21(called PCP'21$_s$), the performance loss relative to HCP'19 is limited to 27%. Our results thus provide evidence for the significance of our approach in dispatcher performance in a large or heterogeneous system running modern applications.

While we have used real data representing the workload of modern systems and applications, our conclusions are based on a simulation study which is restricted by the capabilities of the simulator. For instance, AccaSim does not add the dispatching time to the waiting times of jobs. This seems to be the reason why we have not observed noteworthy gains with PCP'21 in the QoS. In a real system, jobs' waiting time (and slowdown) would increase during dispatching time, therefore dispatcher performance would directly affect the QoS.

6 Conclusions and Future Work

Constraint Programming (CP) has been been successfully applied to solve the on-line job dispatching problem in HPC systems [3, 7] including those running modern applications [13]. We argued that the limitations of the available CP-based job dispatchers may hinder their practical use in today's systems that are becoming larger in size and more heterogeneous in their computing resources. In an attempt to bring CP closer to a deployed application, we presented a new CP-based on-line job dispatcher for HPC systems (PCP'21). Unlike its predecessors, PCP'21 solves the entire problem using CP and its model size is independent of the system size. Experimental results based on a simulation study show that our approach can bring about significant performance gains over the existing dispatchers in a large or heterogeneous system.

In future work, we will devise and experiment with a meta-dispatcher that can switch between PCP'21 and HCP'19 depending on the workload type. Moreover, we will investigate the impact of performance in the QoS of a dispatcher by adapting the simulator to take into account the dispatching time in the calculation of the job waiting time. To improve the dispatcher performance further, we will study breaking the symmetry among the identical nodes (i.e. the nodes that have the same resource availability at a dispatching time t) and dominance breaking during search. We will also investigate whether large neighbourhood search can be beneficial. Towards our objective to deploy and evaluate a CP-based dispatcher in a real system, we plan to encode as constraints the allocation strategies proposed for heterogeneous systems [20].

References

1 Altair. Altair PBS professional (accessed may 27 2021), 2021. URL: https://www.altair.com/pbs-works/.

2 Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. Chapter 22 - constraint-based scheduling and planning. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 761–799. Elsevier, 2006.

3 Andrea Bartolini, Andrea Borghesi, Thomas Bridi, Michele Lombardi, and Michela Milano. Proactive workload dispatching on the EURORA supercomputer. In *Proceedings of Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014.*, volume 8656 of *Lecture Notes in Computer Science*, pages 765–780. Springer, 2014. doi:10.1007/978-3-319-10428-7_55.

4 Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994. doi:10.1016/0895-7177(94)90127-9.

5 Jacek Blazewicz, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource sonstraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983. doi:10.1016/0166-218X(83)90012-4.

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

7 Andrea Borghesi, Francesca Collina, Michele Lombardi, Michela Milano, and Luca Benini. Power capping in high performance computing systems. In *Proceedings of Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 524–540. Springer, 2015. doi:10.1007/978-3-319-23219-5_37.

8 Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE Trans. Parallel Distrib. Syst.*, 27(10):2781–2794, 2016.

9 Jirachai Buddhakulsomsiri and David S. Kim. Priority rule-based heuristic for multi-mode resource-constrained project scheduling problems with resource vacations and activity splitting. *European Journal of Operational Research*, 178(2):374–390, 2007. doi:10.1016/j.ejor.2006.02.010.

10 Carlo Cavazzoni. EURORA: a european architecture toward exascale. In *Proceedings of the Future HPC Systems - the Challenges of Power-Constrained Performance, FutureHPC@ICS 2012, Venezia, Italy, June 25, 2012*, pages 1:1–1:4. ACM, 2012. doi:10.1145/2322156.2322157.

11 Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - A status report. In *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004, Revised Selected Papers*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004. doi:10.1007/11407522_1.

12 Cristian Galleguillos, Zeynep Kiziltan, Alessio Netti, and Ricardo Soto. Accasim: a customizable workload management simulator for job dispatching research in HPC systems. *Cluster Computing*, 23(1):107–122, 2020. doi:10.1007/s10586-019-02905-5.

13 Cristian Galleguillos, Zeynep Kiziltan, Alina Sîrbu, and Özalp Babaoglu. Constraint programming-based job dispatching for modern HPC applications. In *Proceeding of Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019*, volume 11802 of *Lecture Notes in Computer Science*, pages 438–455. Springer, 2019. doi:10.1007/978-3-030-30048-7_26.

14 Cristian Galleguillos, Alina Sîrbu, Zeynep Kiziltan, Özalp Babaoglu, Andrea Borghesi, and Thomas Bridi. Data-driven job dispatching in HPC systems. In *Proceedings of Machine Learning, Optimization, and Big Data - Third International Conference, MOD 2017, Volterra, Italy, September 14-17, 2017, Revised Selected Papers*, volume 10710 of *Lecture Notes in Computer Science*, pages 449–461. Springer, 2017. doi:10.1007/978-3-319-72926-8_37.

15 R. Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, March 1989. doi:10.1007/BF01721162.

16 Stijn Heldens, Pieter Hijma, Ben van Werkhoven, Jason Maassen, Adam S. Z. Belloum, and Rob van Nieuwpoort. The landscape of exascale research: A data-driven literature analysis. *ACM Comput. Surv.*, 53(2):23:1–23:43, 2020. doi:10.1145/3372390.

17 Robert L. Henderson. Job scheduling under the portable batch system. In *Proceedings of Job Scheduling Strategies for Parallel Processing, IPPS'95 Workshop, Santa Barbara, CA, USA, April 25, 1995.*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 1995. doi:10.1007/3-540-60153-8_34.

18 ITIF. The vital importance of high-performance computing to u.s. competitiveness. information technology and innovation foundation. (accessed september 4, 2020), 2016. URL: http://www2.itif.org/2016-high-performance-computing.pdf.

19 Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008, Coconut Grove, Florida, USA*, pages 555–560. AAAI Press, 2008. URL: http://www.aaai.org/Library/FLAIRS/2008/flairs08-126.php.

20 Alessio Netti, Cristian Galleguillos, Zeynep Kiziltan, Alina Sîrbu, and Özalp Babaoglu. Heterogeneity-aware resource allocation in HPC systems. In *Proceedings of High Performance Computing - 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018*, volume 10876 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2018. doi:10.1007/978-3-319-92040-5_1.

21 C. Le Pape, P. Couronne, D. Vergamini, and V. Gosselin. Time-versus-capacity compromises in project scheduling. *AISB Quartetly*, pages 19–31, 1995.

22 PRACE. The scientific case for computing in europe 2018-2026. prace scientific steering committee. (accessed september 4, 2020), 2018. URL: https://prace-ri.eu/wp-content/uploads/2019/08/PRACEScientificCase.pdf.

23 Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and Jeremy Kepner. Scalable system scheduling for HPC and big data. *J. Parallel Distributed Comput.*, 111:76–92, 2018. doi:10.1016/j.jpdc.2017.06.009.

24 Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 9th Edition*. Wiley, 2014.

25 SLURM. SLURM workload manager, 2019. URL: http://slurm.schedmd.com.

26 Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):789–803, 2007. doi:10.1109/TPDS.2007.70606.

27 Rinki Tyagi and Santosh Kumar Gupta. A survey on scheduling algorithms for parallel and distributed systems. In *Silicon Photonics & High Performance Computing*, pages 51–64, Singapore, 2018. Springer Singapore.