Heterogeneity-aware resource allocation in HPC systems

Heterogeneity-aware Resource Allocation in HPC Systems

Alessio Netti[1], Cristian Galleguillos[1,2], Zeynep Kiziltan[1], Alina Sîrbu[3,4], and Ozalp Babaoglu[1]

[1] Department of Computer Science and Engineering, University of Bologna, Italy
{alessio.netti, zeynep.kiziltan, ozalp.babaoglu}@unibo.it
[2] Escuela de Ing. Informática, Pontificia Universidad Católica de Valparaíso, Chile
cristian.galleguillos.m@mail.pucv.cl
[3] Department of Computer Science, University of Pisa, Italy
[4] Science Division, New York University Abu Dhabi, United Arab Emirates
alina.sirbu@unipi.it

Abstract. In their march towards exascale performance, HPC systems are becoming increasingly more heterogeneous in an effort to keep power consumption at bay. Exploiting accelerators such as GPUs and MICs together with traditional processors to their fullest requires heterogeneous HPC systems to employ intelligent job dispatchers that go beyond the capabilities of those that have been developed for homogeneous systems. In this paper, we propose three new heterogeneity-aware resource allocation algorithms suitable for building job dispatchers for any HPC system. We use real workload traces extracted from the Eurora HPC system to analyze the performance of our allocators when they are coupled with different schedulers. Our experimental results show that significant improvements can be obtained in job response times and system throughput over solutions developed for homogeneous systems. Our study also helps to characterize the operating conditions in which heterogeneity-aware resource allocation becomes crucial for heterogeneous HPC systems.

1 Introduction

Motivation. Modern scientific discovery is increasingly being driven by computation and High-Performance Computing (HPC) systems have come to play a fundamental role as "instruments" not unlike the microscopes and telescopes of the previous century [20]. Despite the enormous progress that has been achieved in processor technologies, we are still far from considering many important problems "solvable" using a computational approach. These problems include turbulence of fluids in finite domains, combustion hydrodynamics, computational biology, natural language understanding and modeling of the human brain [1]. Future HPC systems will achieve the performance required to solve these problems through a combination of faster processors and massive parallelism. Yet, a homogeneous parallelism employing millions of processor cores will result in power requirements that are unsustainable. Thus, the parallelism has to be heterogeneous, employing specialized energy-efficient accelerator units such as GPUs and

MICs in addition to the traditional CPUs. In fact, among the top 100 HPCs of the latest Top500 List[5] (updated on 06-2017), almost 30% are based on GPUs and/or MICs.

Traditionally, HPC systems have been used to run compute-intensive jobs requiring days or even weeks to complete their massive computations. There is an increasing trend where HPC systems are being used to run "big data workloads" consisting of many shorter jobs performing data analytics as data is being streamed from a monitored system [16, 17]. The ability to build predictive models from streamed data opens up the possibility for acting on the predictions in real time [15]. Turning this scenario into an effective "on-line control" mechanism requires intelligent strategies to achieve elevated levels of system performance with high throughput and low response times so that the predictive models built from data analytics correspond to recent, rather than a distant, past states of the monitored system.

The potential to fully exploit the raw computing potential of an HPC system and deliver it to applications (jobs) is conditional on intelligent system software making informed decisions to efficiently manage system resources. Among these decisions, those made by a *dispatcher* regarding job executions are particularly important for ensuring high levels of system performance. In an HPC system, the *scheduler* component of a dispatcher selects which jobs to run next among those currently in the wait queue; whereas the *allocator* component decides which resources to allocate for running them. While the scheduling aspect of dispatching has received considerable attention in the literature [5], the allocation problem has been studied to a lesser extent. Intelligent allocation is particularly important in heterogeneous systems where poor decisions can lead to poor resource usage and consequently poor performance of critical applications [13].

Related Work. Resource allocation strategies used in many popular HPC workload management systems [14, 22] can be characterized as variations of well-known memory allocation heuristics such as *First-Fit* (FF) and *Best-Fit* (BF) [19]. In memory allocation, FF chooses the first block of memory that is large enough to satisfy the request. In an analogous manner, an FF resource allocator chooses the first resource among a list of available resources that satisfies a job's request. FF is primarily focused on satisfying a single job request without any regard for global considerations of resource usage. A BF allocator, on the other hand, chooses a resource, among a list of available resources, that is able to satisfy the job's request while leaving the smallest possible unused capacity.

These simple heuristics can be improved in several ways in an effort to utilize resources more intelligently so as to improve job response times and system throughput. In [3], a "lookahead" capability is added to BF, taking into account the needs of other jobs in the queue. Specifically, resources are allocated to the current job in a manner such that, if possible, enough resources remain available for the largest job (requiring the largest amount of resources) in the queue. A similar idea can be applied to scheduling where a *backfilling scheduler* [18] selects

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

short, low-resource jobs to fill the gaps in resource usage left over after scheduling larger jobs, even if the short jobs are not next in the queue. Large jobs that cannot be scheduled are blocked and resources are reserved for them. Instead of considering just one job at a time during the backfilling phase, multiple jobs can be considered together so that resource usage is improved [18]. In [12], resource allocation takes into account saturation of shared resources, such as memory bandwidth, that can cause some jobs to take longer to complete. To make a suitable allocation that does not reduce system throughput, penalties based on memory bandwidth saturation are included in the FF allocation heuristic.

The main shortcoming of the allocation strategies described above is that they were designed for a single resource type such as a CPU and do not consider characteristics inherent to heterogeneous systems, including different resource types or different configurations of the same resource type. This limitation can lead to unbalanced usage and fragmentation of heterogeneous resources, and cause undesirable delays. For instance, in [4], a dispatcher is presented for the heterogeneous HPC system *Eurora* [7] that has GPU co-processors in half of its nodes and MIC co-processors in the other half. For allocation, however, the dispatcher uses the simple BF heuristic, sorting the nodes by the total number of available computing resources, making no distinction between CPUs, GPUs or MICs. Consequently, with many jobs requesting just CPUs as processing resources, a simple allocation of computing resources will result in unnecessary delays for jobs that require GPUs or MICs in addition to CPUs.

In [23] multiple resources (CPU, bandwidth, memory) are considered and bottleneck resources are identified to obtain fair resource allocation to users. However, they do not consider systems with resources that are available only on a subset of all the nodes, such as the GPUs and MICs that characterize the systems we are analyzing. To the best of our knowledge, no previous work focusing on resource allocation for heterogeneous HPC systems exists.

Contributions. In this paper, we present several resource allocation algorithms for heterogeneous systems that adopt different strategies for minimizing wastage of critical resources, and consequently, minimizing job delays. The algorithms are based on simple heuristics that exhibit good performance with low computational overhead, and are general enough to be applied to any heterogeneous HPC system where critical resources need to be managed efficiently. We evaluate our allocators when combined with a suite of different schedulers using a workload trace collected from the Eurora HPC system. Our experimental results show that significant improvements can be obtained in job response times and system throughput compared to standard solutions like FF and BF. Our study also helps to characterize the operating conditions in which heterogeneity-aware resource allocation becomes crucial for heterogeneous HPC systems.

Organization. The rest of the paper is organized as follows. The next Section briefly describes the Eurora HPC system, its workload datasets, and the scheduling algorithms we used in conjunction with our allocators for dispatching

Job class	Share	Count	Average duration [hh:mm:ss]
All	100%	372320	00:16:08
CPU-based	22.8%	85046	00:47:36
MIC-based	0.7%	2500	00:56:28
GPU-based	76.4%	284774	00:06:23

Table 1. Frequency and average duration of all jobs and the three classes CPU-based, MIC-based and GPU-based in the Eurora workload.

purposes. In Section 3 we introduce our allocation algorithms, while Section 4 presents our experimental evaluation results. Section 5 concludes the paper.

2 HPC System, Data and Job Dispatching

2.1 Eurora and the workload dataset

We evaluate our allocation strategies using workload data collected from the Eurora HPC system [7]. Eurora is a hybrid installation hosted at Cineca[6], the largest datacenter in Italy, that uses a combination of CPUs, GPUs and MICs to achieve very high energy efficiency. The system consists of 64 nodes, each equipped with two octa-core CPUs (Intel Xeon E5) and two accelerators. Half of the nodes have two GPUs as accelerators (Nvidia Tesla Kepler), while the other half have two MICs (Intel Xeon Phi Knights Corner). The resulting system is highly heterogeneous, making allocation of resources to jobs nontrivial.

The HPC workload, consisting of almost 400,000 jobs that were run on Eurora during the time period April 2014 – August 2015, has been recorded as a trace and made available by Cineca. For our study, we classify the jobs in the workload based on their duration as *short* (under 1 hour), *medium* (between 1 and 5 hours) and *long* (over 5 hours). Of all the jobs, 93.14% are short, 6.10% are medium and 0.75% are long. Hence, the workload is quite varied from this point of view. We further divide jobs into three classes based on the computing resources that they require: *CPU-based* jobs use CPUs only, while *MIC-based* and *GPU-based* jobs use MIC or GPU accelerators, respectively, in addition to CPUs. Table 1 shows statistics for each job class in the workload. We observe that GPU-based jobs are the most numerous, followed by CPU-based jobs, while MIC-based jobs are relatively few. In terms of duration, we observe that CPU-based jobs are on average longer than GPU-based jobs, consuming significantly more resources. This heterogeneity of job classes increases the difficulty of allocation decisions. Since CPU-based jobs are longer, they may keep nodes that have accelerators busy for longer periods, during which their accelerators are not available for other jobs. Given that GPU-based jobs are the most frequent, this can cause bottlenecks to form in the system, motivating the development of heterogeneity-aware allocation algorithms to be described in the following sections.

[6] https://www.cineca.it/

2.2 Scheduling algorithms in job dispatching

In an HPC system, allocation goes hand in hand with scheduling in order to perform job dispatching. To test our allocation algorithms, we combined them with four state-of-the-art scheduling algorithms: *Shortest Job First* (SJF), *Easy Backfilling* (EBF), *Priority Rule-Based* (PRB), and *Constraint Programming Hybrid* (CPH). All these algorithms have been previously applied to Eurora workload data in [11], where it was shown that accurate prediction of job duration can improve scheduling performance. In the rest of this section we describe briefly the schedulers employed.

SJF. At scheduling time, the SJF scheduler selects the shortest job among all jobs in the queue to be scheduled first. The job is then passed to the allocator to be granted the required resources. A predicted job duration is used as the job length to establish an order.

EBF. This scheduling algorithm considers jobs in order of their arrival [21]. If there aren't enough available resources in the system for a given job that has been selected for scheduling, the job is blocked, and a reservation is made for it. A reservation consists of a starting time (computed using the predicted duration of running jobs) when enough resources are expected to become available to start the blocked job. A set of resources, as determined by the allocator, is also associated with the reservation and will be used for the blocked job at reservation time. While the blocked job waits for its reserved resources to become available, the algorithm will schedule shorter jobs that are expected to terminate before the starting time of the reservation (again based on predicted duration), using currently unused resources.

PRB. This algorithm sorts the set of jobs to be scheduled according to a priority rule, running those with higher priority first [6]. In our work, we use priority rules based on jobs' urgency in leaving the queue, as introduced by Borghesi et al. in [4]. To determine if a job could wait in the queue, the ratio between the current waiting time and the expected waiting time of the job is calculated. The expected waiting time is computed from data as the average waiting time over a certain queue. As a tie breaker, the "job demand" is used, which is the job's combined resource requirements multiplied by the predicted job duration.

CPH. One of the drawbacks of the aforementioned heuristic schedulers is the limited exploration of the solution space. Recent results show that methods based on *constraint programming* (CP) are able of outperforming traditional PRB methods [2]. To increase scalability, Borghesi et al. introduce a hybrid scheduler called *CPH* [4] combining CP and a heuristic algorithm, which we use in this paper. CPH is composed of two phases. In the first phase jobs are scheduled using CP, minimizing the total job waiting time. At this stage, each resource type is considered as a unique resource — CPU availability corresponds to the sum of the available CPUs of all the computing nodes, memory availability corresponds to the sum of the memory availability of all the computing nodes, and so on. Due to the problem's complexity, the search is bound by a time limit; the best solution found within the limit is the scheduling decision. The preliminary

schedule generated in the first stage may contain some inconsistencies because of considering the available resources as a whole. The second phase performs resource allocation according to a heuristic in which any inconsistencies are removed. The specific heuristic being used depends on the allocator component of the dispatcher. If a job can be mapped to a node then it will be dispatched, otherwise it will be postponed.

2.3 Job duration prediction in job dispatching

An issue in simulating job dispatching strategies regards what information contained in the workload can be used when making decisions. A specific case is that of job durations. Since the workload data contains exact job durations, it is tempting to use them in order to make dispatching decisions. However, in a real system, exact job durations are not known in advance, so dispatching decisions cannot be based on this knowledge. Here, we take this into account and use instead *predicted* job durations, based on a very simple heuristic that was proposed in [11] and exploits time locality of job durations for individual users that was observed in the Eurora workload dataset. Specifically, it has been observed that consecutive jobs by the same user tend to have similar durations, especially when they have the same profile (job name, resources requested, queue, etc). From time to time, a switch to a different duration is observed, which could happen, for example, when the user changes input datasets or the algorithm itself. Using this observation, the authors devise a set of rules to apply in order to predict job durations. They record job profiles for users, and their last durations. When a new job arrives in the system, they look for a job with the same or similar profile, and consider its duration to be also the duration of the new job. If no past profile is similar enough, the predicted duration is the default wall-time of the queue where the job is submitted. In case a match is found, the predicted duration is capped by the maximum wall-time of the queue. Both default and maximum wall-time values of the queues are part of the configuration of the dispatcher.

The mean absolute error (MAE) of this heuristic prediction with respect to the real job duration on the Eurora workload dataset is shown to be 40 mins [11]. In the absence of any prediction, users supply dispatchers their own job duration estimation, which is typically the maximum wall-time of the queue. In the absence of even this information, the dispatchers use the default wall-time of the queue. We will refer to this as the wall-time approach. The MAE of the wall-time approach on the Eurora workload is 225 mins [11], which is dramatically worse than that of the proposed prediction technique. The heuristic prediction therefore shows an improvement of 82% over the wall-time approach. We shall note that the time locality of job durations for individual users is not specific to the Eurora workload. It can also be observed in other workload datasets to which the same heuristic prediction can be applied. An example is the Gaia workload dataset[7] of the University of Luxemburg. We calculated that the MAE

[7] http://www.cs.huji.ac.il/labs/parallel/workload/l_unilu_gaia/index.html

of the wall-time approach is 2918 mins, while it is 220 mins with our heuristic prediction, showing an improvement of 93%. The notable improvement over the Eurora dataset can be explained by the fact that the maximum wall-time values in Gaia are higher than those of Eurora.

3 Allocation Algorithms

Here we describe the allocation algorithms that we designed and implemented for heterogeneous HPC systems. We assume that a job is composed of multiple job units (such as MPI processes), each having the same resource requirement. All the algorithms are based on the *all-requested-computers-available* principle [21]: when an allocator is invoked for job j, nodes in the system are searched sequentially according to a specific order, and the largest number of *job units* of job j are allocated while traversing the nodes. The allocation process ends when either all job units have been allocated, in which case the allocation succeeds, or the list of nodes is exhausted, in which case the allocation fails. The jobs are ordered as specified by the scheduler, while the ordering criteria for the nodes is specific to the allocator. Our algorithms provide custom criteria for node ordering, resulting in allocation strategies with different strengths. The algorithms are generic and provide configuration parameters, hence they do not rely on any specific system architecture and can be tuned suitably for any heterogenous system in consideration. In the following, we call a resource type *critical* if careless usage of the respective nodes may cause bottlenecks in the system.

Balanced heuristic. The main focus of this algorithm is to avoid the fragmentation of *user-defined critical resource types*, like accelerators, by *limiting* and *balancing* the use of the respective nodes. The limiting effect is achieved by pushing the nodes with critical resources towards the end of the list of available nodes. In this way, by selecting nodes from the beginning of the list, jobs that do not need critical resources will not block such resources. The *balancing* effect is achieved by *interleaving* nodes having different types of critical resources, thus not favoring any of them.

By default, the critical resource types for Eurora are MIC and GPU, but they can be modified by the user based on the system architecture. The algorithm works in two phases. First, all nodes in the system are collected in *bins*: there is a bin for each critical resource type, and nodes are assigned to a specific bin according to which of those they currently have available. If they do not have any, they will be assigned to a special *nul* bin; conversely, they will be assigned to the bin for which they have the maximum availability, if multiple critical resources are present (note that in Eurora, a node has only one type of critical resource). The bins are then combined in a final node list, which is built as follows: nodes belonging to the *nul* bin, which do not have any critical resource, are placed at the head. The rest of the list is built incrementally by picking a node from the *currently largest* bin until they are all empty. An example of BALANCED node ordering in a small system with 8 nodes can be seen in Figure 1.

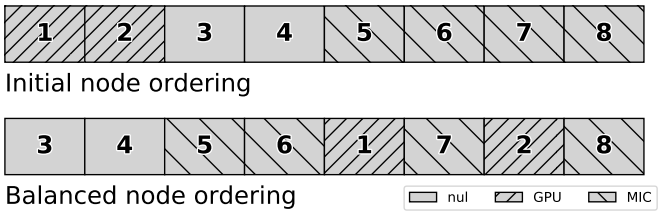


Fig. 1. An example of BALANCED node ordering on a small system with 8 nodes, each labeled with an ID and colored according to its corresponding bin.

This type of reasoning is expected to be beneficial in an architecture like Eurora, where two continuous blocks of nodes have either GPU or MIC resources and are thus prone to unbalanced node usage. The BALANCED allocator does not consider the distribution of the resource requirements of jobs submitted to the system, and assumes that all critical resource types are used uniformly. This design choice ignores that some resources can become critical at runtime when the distribution is heavily skewed towards a specific resource type, but at the same time it increases the algorithm's robustness against sudden bursts of jobs requiring that specific resource. While BALANCED can be used in any system by suitably defining the critical resource types that must be protected, it is most effective on systems that are known to possess multiple critical resource types.

Weighted heuristic. This algorithm is more general than BALANCED as it is able to detect the critical resources at runtime, as opposed to them being statically defined by the user, and focuses on avoiding their wastage. It is based on the popular BF heuristic, which at each allocation time sorts the nodes in non-decreasing order with respect to the total amount of available resources. BF can easily waste some resources as it does not distinguish between different resource types. WEIGHTED is instead aware of heterogeneous resources and adds lookahead features to allocation to increase the chance of success. For each job during allocation, it sorts the nodes in non-decreasing order of their ranking. A node is ranked based on the criticality of its resource types and their availability after a potential allocation. Consequently, nodes with highly critical resource types and nodes which will be providing high availability after allocation are pushed towards the end of the list, in order to be protected against jobs that do not need such resources, similar to what BALANCED does with nodes containing user-defined critical resources.

More formally, for a job allocation, after the number of job units that fit on a certain node is calculated, the *impact* of the allocation is defined for each resource type to be the amount of resources still available in the node after such allocation. We thus have, for each node i and for each resource type $k \in r$, an impact value $imp_{i,k}$. The impact serves to measure the resource wastage after allocation in the presence of multiple resource types. The ranking R_i of a node

i is then computed by summing the $imp_{i,k}$ of each resource type k available on node i weighted by w_k:

$$R_i = \sum_{k \in r} w_k * imp_{i,k} \qquad w_k = \frac{\overline{req_k} * load_k}{cap_k} \tag{1}$$

A weight w_k is computed at the system level and quantifies the level of criticality of a certain resource type k using three parameters as in Equation 1. The first parameter $\overline{req_k}$ is the average amount requested for k by jobs in the queue. A highly requested resource type is considered critical. This average is computed over all jobs in the queue, weighted by the jobs' duration prediction. Considering the job duration as well in the average is mostly a fairness consideration, since most of our schedulers, like SJF, tend to favor shorter jobs. The second parameter $load_k$ is the *load ratio* for k, which is the ratio between the amount of resources used at a certain time and the total resource capacity of the system for k, assuming that resources assigned to jobs are always used fully [8]. A high load ratio means low availability, which makes the resource type critical. The $load_k$ parameter, however, does not consider the total capacity cap_k which can influence the criticality of k. We therefore use cap_k as a normalization factor.

With multiple factors assessing the criticality of resource types, Weighted is expected to perform well in many scenarios. Weighted is thus more flexible than Balanced, even though it does not possess its interleaving capability.

Priority-Weighted heuristic. The Weighted and Balanced allocators are expected to be most effective in different scenarios, with Balanced performing better in the presence of bursts of jobs requiring critical resources, and vice versa. Priority-Weighted is a hybrid strategy, trying to combine the strengths of both allocators in order to obtain optimal performance. This algorithm extends Weighted, by adding a new multiplicative parameter p_k to w_k. Specifically, p_k acts as a bounded *priority value*, used only for user-defined critical resource types like in Balanced. For the other resource types, it is assumed to be always equal to 1. Such a priority value is calculated at runtime in a simple way: starting with the value 1, every time the allocation for a job requiring a critical resource type k fails, its priority value p_k is increased by 1. Conversely, when the allocation succeeds, p_k is decreased by 1. If a job requires multiple critical resource types, all the related p_k values are affected. The bound of p_k is user-defined and is set to 10 by default.

This solution allows us to take into account the runtime criticality of resources (like in Weighted) and to protect user-defined critical resources (like in Balanced) in a rather dynamic way by adjusting to the system's behavior. Various other solutions were tried for p_k, such as the average number of allocation failures per job or per allocation time, or the number of jobs in the queue for which allocation has previously failed. Out of all of these, our priority mechanism emerged to be the best technique, despite its simplicity.

4 Experimental Results

In this section, we present the experimental results obtained by using the Eurora workload dataset described in Section 2.1. All the data available for the Eurora system has been considered in the experiments. Due to space limitations, we cannot report tests performed on other systems.

4.1 Experimental setup

Simulation of the Eurora system along with job submission and job dispatching were carried out using the open-source AccaSim HPC Simulator [10]. A total of 20 dispatchers were employed, which were obtained by combining the 4 scheduling algorithms (SJF, EBF, PRB, CPH) described in Section 2 together with 5 allocation algorithms: the three described in Section 3 (B, W, P-W) together with First-Fit (FF) and Best-Fit (BF). FF and BF are included solely for the purpose of comparison. Interpreted in the context of Eurora, FF searches the nodes with available resources in a static order, while BF sorts the nodes in non-decreasing order of the amount of available resources. The experiments were performed on a dedicated server with a 16-core Intel Xeon CPU and 8GB of RAM, running Linux Ubuntu 16.04. All the dispatchers along with their source code in Python are available on the AccaSim website.[8]

In the experiments, we evaluate dispatchers in terms of their impact on job response times and system throughput, characterized by two metrics. The first is the *job slowdown*, a common indicator for evaluating job scheduling algorithms [9], which quantifies the effect of a dispatching method on the jobs themselves and is directly perceived also by the HPC users. The slowdown of a job j is a normalized response time and is defined as $slowdown_j = (T_{w,j} + T_{r,j})/T_{r,j}$, where $T_{w,j}$ and $T_{r,j}$ are the waiting time and duration of job j, respectively. A job waiting more than its duration has a higher slowdown than a job waiting less than its duration. The second metric is the *queue size*, which counts the number of queued jobs at a certain dispatching time. This metric is a measure of the effects of dispatching on the computing system itself. The lower these two metrics are, the better job response times and system throughput are.

We also compared the dispatchers in terms of their resource utilization. The metric we adopt for this purpose is the popular *system load ratio* [8] which considers the ratio between the amount of used resources in the HPC system at a certain time and its total resource capacity, assuming that resources assigned to jobs are always used fully.

4.2 Results over the entire workload dataset

We first discuss the results obtained over the entire workload dataset. All the dispatchers are run using three different job duration predictions: wall-time (W), the data-driven prediction heuristic presented in Section 2 (D), and real duration

[8] http://accasim.readthedocs.io/en/latest/

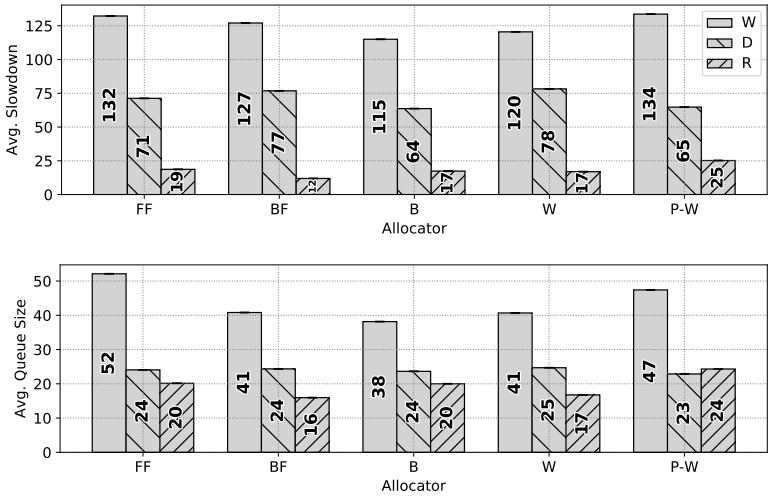


Fig. 2. Average slowdown and queue size results over the entire workload dataset using the CPH scheduler and five different allocators with wall-time (W), data-driven (D) and real duration (R) predictions for job durations.

(R). The purpose here is to assess the importance of using data-driven prediction for response time and throughput of a dispatcher, with respect to crude prediction (wall-time) and best prediction (real duration). Due to lack of space, we here present only the results related to the CPH scheduler, as it is the best performing among all and is highly representative of the behavior of the schedulers in conjunction with the allocators in question.

Figure 2 shows the average slowdown over all jobs and average queue size values over all dispatching times of the CPH scheduler for all available allocators. We make two observations. First, across all allocators, the data-driven job duration prediction has notable impact on job response times and system throughput, leading to results better than using the wall-time. Therefore, in the next sections we only present results using the data-driven job duration prediction for all jobs. Second, we do not note substantial performance variations among the various allocators. We believe that this could be due to certain time periods in our workload where the corresponding data does not possess significant characteristics for our study. For instance, there could be time periods in which few jobs are submitted to the system, keeping it in a low utilization state, or conversely periods in which large amounts of jobs are submitted, overloading it. In either case, an allocator is not expected to enhance the job response times and system throughput considerably.

4.3 Results over specific time periods

We now use only the data-driven prediction of job duration. We then restrict our study to shorter time periods in the workload, such as months, to be able

Scheduler	Allocator										Best gain %			
	FF		BF		B		W		P-W		FF		BF	
	s	q	s	q	s	q	s	q	s	q	s	q	s	q
CPH	10	4	10	4	**7**	**4**	8	4	**7**	**4**	30%	0%	30%	0%
SJF	16	5	16	5	**10**	**5**	**10**	**5**	**10**	**5**	37%	0%	37%	0%
EBF	28	5	28	5	**18**	**4**	20	4	20	4	35%	20%	35%	20%
PRB	28	5	28	5	**21**	**4**	22	4	22	4	25%	20%	25%	20%

(a) April 2015 dataset.

Scheduler	Allocator										Best gain %			
	FF		BF		B		W		P-W		FF		BF	
	s	q	s	q	s	q	s	q	s	q	s	q	s	q
CPH	251	57	271	55	**238**	**63**	315	82	254	70	5%	-10%	12%	-12%
SJF	1269	230	1270	229	1266	233	1276	218	**1253**	**216**	1%	6%	1%	6%
PRB	1852	615	2023	640	**1778**	**594**	1829	627	1910	599	4%	3%	12%	7%
EBF	3004	697	2563	702	**2197**	**619**	2378	686	2313	674	26%	11%	14%	12%

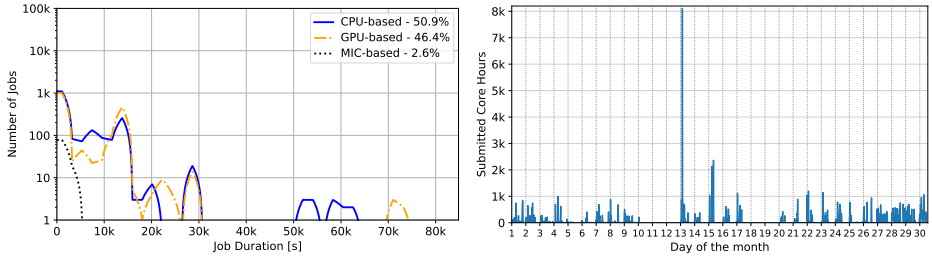
(b) April 2014 dataset.

Table 2. Average slowdown (s) and queue size (q) results for the April 2015 (a) and April 2014 (b) datasets.

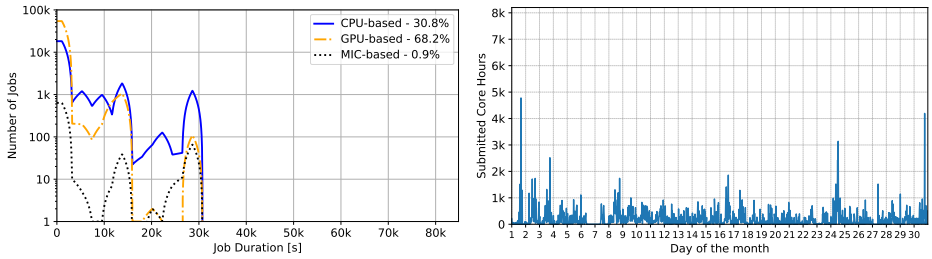
to understand the operating conditions in which we can benefit from the new allocators. For this purpose, we extracted the results of the jobs that were submitted during a certain month after running the experiments once on the entire workload. Thus, the months never start in an idle state but carry on from the previous month, like in a real scenario.

Here we present some insights derived by analyzing data from four particular months. These months are not only interesting due to their peak job submissions, they are also representative in the sense that their job submission patterns and the corresponding experimental results are found in other months as well (not shown in the paper due to space reasons).

Tables 2 and 3 give the average slowdown (s) and queue size (q) of every dispatcher (composed by a scheduler in the first column and an allocator in the next 5 columns). For each scheduler, the best allocator result is indicated in bold, in addition to the best gain obtained in percentage by the new allocators compared to FF and BF. Figures 3 and 4 demonstrate instead job duration distributions, for each job class as in Section 2.1, as well as job submission patterns in terms of the total CPU core hours of the submitted jobs in every 30-minute time window of a day. Sudden spikes in the job submission patterns are caused by the arrival of jobs that either have a high duration or require a large amount of resources. We do not consider the distributions of the amount of resources requested by jobs, as no significant differences across months have been observed.



(a) April 2015 dataset.



(b) April 2014 dataset.

Fig. 3. Job duration distributions (left) and job submission patterns (right) for the April 2015 (a) and April 2014 (b) datasets.

Allocation with moderate gains. We start with the workloads related to April 2015 and April 2014 where the new allocators bring about relatively moderate gains in response times and throughput with respect to FF and BF. This can be immediately witnessed in Table 2. For the April 2015 dataset, while the slowdown values are reduced between 20% to 37%, the queue size remains the same with the two best-performing schedulers, CPH and SJF. In the case of the April 2014 dataset, the gains in slowdown are between 1% and 26%, while the queue size increases slightly with the best performing scheduler, CPH. Hence, in all cases we witness an improvement in slowdown, however queue size values improve only when not using the CPH scheduler.

Analyzing the characteristics of the workload in Figure 3, we can understand the reason for having only moderate improvements. The April 2015 dataset contains 3,740 jobs with few of them long (duration $>$ 5h), while the majority are short (duration $<$ 1h) or medium (1 $\leq$ duration $\leq$ 5h). We would therefore expect low slowdown and queue size values without the need of dedicated allocators. Around half of the jobs in the workload require only CPU and memory, which do not need heterogeneity-aware allocators. Moreover, the system is rarely put under pressure, reducing the importance of complex dispatching algorithms. The only exception is in the sudden burst towards the middle of the month, in which over 8,000 core hours worth of jobs are submitted to the system in a very

Scheduler	Allocator										Best gain %			
	FF		BF		B		W		P-W		FF		BF	
	s	q	s	q	s	q	s	q	s	q	s	q	s	q
CPH	41	27	42	28	11	11	10	10	**8**	**8**	80%	70%	81%	71%
SJF	34	28	29	23	19	20	23	19	**14**	**14**	58%	50%	50%	39%
PRB	43	24	47	27	**30**	**16**	36	16	40	20	30%	33%	36%	40%
EBF	51	33	48	34	39	19	**37**	**20**	53	37	27%	39%	22%	41%

(a) September 2014 dataset.

Scheduler	Allocator										Best gain %			
	FF		BF		B		W		P-W		FF		BF	
	s	q	s	q	s	q	s	q	s	q	s	q	s	q
CPH	8	7	7	6	11	7	**4**	**4**	7	6	50%	42%	42%	33%
PRB	20	13	22	14	14	10	8	6	**5**	**3**	75%	76%	77%	78%
EBF	22	15	26	16	21	15	**14**	**10**	18	13	36%	33%	46%	37%
SJF	26	17	32	22	18	14	18	13	**15**	**11**	42%	35%	53%	50%

(b) August 2014 dataset.

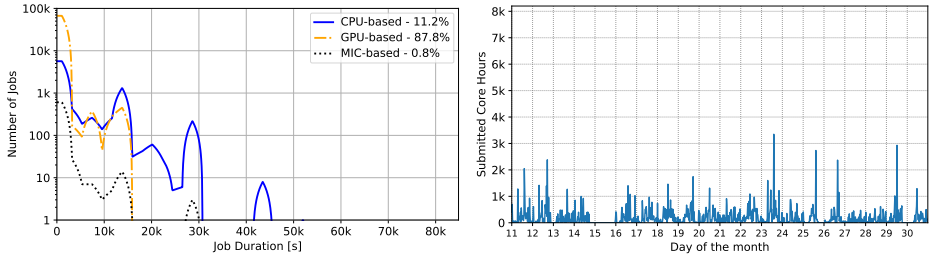
Table 3. Average slowdown (s) and queue size (q) results for the September 2014 (a) and August 2014 (b) datasets.

short time. This, however, overloads a small-scale HPC system like Eurora and is hardly managed by any of the dispatchers.

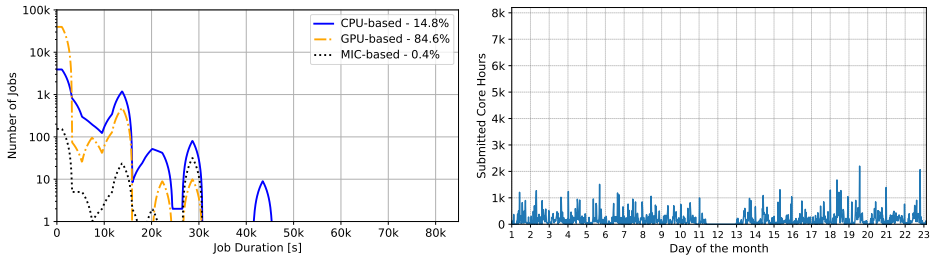
April 2014 is instead a big dataset of 85,245 jobs, with many more medium and long jobs compared to the April 2015 dataset. We would expect here high slowdown and queue size values even with dedicated allocators. The share of jobs requiring only CPU and memory resources is 30.8%, which is still high compared to the 22.8% share for the entire workload, reducing the contribution of allocators specifically designed for heterogeneous systems. In addition, job pressure on the system is always high, with frequent bursts that amount to more than 1,000 core hours. The main problem in this month seems to be the 5,000-hours burst at its beginning: due to the large size of the workload, the early position of the burst results in a cascade effect, severely delaying all subsequent jobs.

Allocation with high gains. We now discuss the datasets related to September 2014 and August 2014 for which significant improvements in response times and throughput are observed, as can be seen in in Table 3. The gains in slowdown and queue size reach up to 81% and 71% in the September 2014 dataset, and up to 77% and 78% in the August 2014 dataset, respectively.

September 2014 is a big dataset of 77,786 jobs, while the August 2014 dataset has medium size with 47,967 jobs. The two datasets, however, share common traits that help understand the relevant results. As can be seen in Figure 4, both datasets contain a high number of short and medium jobs. The number of long jobs is neither low, as in the case of the April 2015 dataset, nor too high as in the case of the April 2014 dataset. In addition, unlike the April 2015 and April 2014



(a) September 2014 dataset.

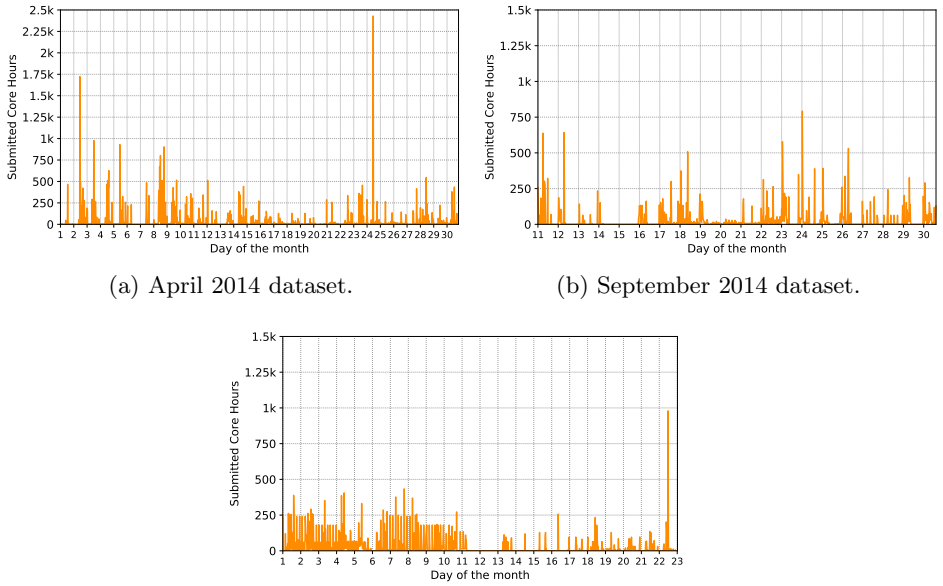


(b) August 2014 dataset.

Fig. 4. Job duration distributions (left) and job submission patterns (right) for the September 2014 (a) and August 2014 (b) datasets.

datasets, GPU-based jobs constitute the vast majority of the workload. All these mean that we can expect considerable improvements in slowdown and queue size values with allocators for heterogeneous systems. Finally, from Figure 4 we can see that the September 2014 and August 2014 datasets are very bursty, much like the April 2014 one. Yet in the case of the September 2014 and August 2014 datasets, the bursts are much tamer in intensity, corresponding to the normal day-night usage cycles present in the HPC system. Moreover, unlike the April 2015 dataset, the system is often put under pressure, but such pressure is not as high as in the April 2014 dataset.

Comparison of allocators. So far we have only studied the impact of the new allocators B, W and P-W in response times and throughput with respect to FF and BF in an heterogeneous system, but we did not contrast them. To do this, we show in Figure 5 job submission patterns in the selected datasets, this time only for GPU-based jobs. These plots do not show the job pressure on the system, which we already illustrated in Figures 3 and 4, but rather demonstrate the distribution of the jobs requiring accelerator resources over the workloads' time spans. Such resources are peculiar to heterogeneous systems and it is interesting to see when the new allocators behave differently in the presence of jobs requiring them. We are omitting the distributions for the April 2015 dataset due to its small size and the small variance in the behavior among the various allocators.



(a) April 2014 dataset.

(b) September 2014 dataset.

(c) August 2014 dataset.

Fig. 5. Submission patterns of jobs requiring GPUs for the April 2014 (a), September 2014 (b) and August 2014 (c) datasets.

Intuitively, one may expect the W and P-W allocators to perform better than B since they take into account several resource criticality parameters and can adapt to different workload characteristics. However, as explained in Section 3, the B allocator is more robust than W against sudden bursts of jobs requiring critical resources, accelerators in this case, due to its simple nature: B always tries to limit the use of nodes equipped with critical resources, even if they are not actively needed or they are scarcely used by jobs, resulting in a fairly consistent behavior. This is reflected in our results. As seen in Table 2, B is consistently the best performer for the April 2014 dataset which contains several big bursts of GPU-based jobs (Figure 5a). In the September 2014 dataset, some bursts of GPU-based jobs are still present, though less intense compared to those of the April 2014 dataset (Figure 5b). In this case, as shown in Table 3, the W or P-W allocator performs better than B except when used with the PRB scheduler. In the August 2014 dataset instead, where W and P-W are consistently the best performers, GPU-based jobs are evenly distributed over most days of the month, with very few bursts (Figure 5c).

Overall, B is more suited for extreme scenarios where critical resources must be protected at all costs and at all times. Otherwise, W and P-W are the best allocators. The gain offered by P-W over W is less clear. This can be attributed to the fact that P-W is primarily an hybridization strategy between B and W,

and it can be better or worse than its constituting components depending on the workload characteristics and the type of scheduler.

4.4 Resource utilization

We conclude our evaluation with the resource utilization results. In our analysis, we first looked at the distribution of the fraction of used resources, as a function of the fraction of used nodes in the system for all time points, separately for each resource type and combined for all resource types. The results obtained over the entire workload, as well on the four individual months are mainly homogeneous across all dispatchers and therefore are omitted here. This is still good news because we can see that the new allocators can improve response times and throughout without degrading resource utilization with respect to FF and BF, resulting in the best overall compromise between system performance and resource utilization. It is worth mentioning however a particular case when considering the GPU resources for the September 2014 dataset. Looking at the average distribution over all the nodes and all time points, referred to as system load ratio previously, the best-performing scheduler CPH has an improvement of 5.95% when resources are allocated with W instead of BF. Similar improvements are observed when CPH is used with B or P-W. This result may suggest that heterogeneity-aware allocators can lead to better usage of critical resources.

5 Conclusions

We have presented three allocation algorithms suitable for heterogeneous HPC systems aimed at intelligent management of critical, accelerator-like resources. The algorithms are general enough to be applied to any heterogeneous HPC system where critical resources need to be managed efficiently, they are based on simple heuristics that exhibit good performance with low computational overhead, and they can easily be integrated in live queueing systems (like PBS or SLURM) as they do not rely on features beyond those found in common heuristics (like Best-Fit). In order to assess their effectiveness, we modeled the Eurora HPC system with the AccaSim simulator driven by a real workload trace extracted from the same system. We conducted extensive evaluation of our allocators by coupling them with different scheduling algorithms and job duration predictors. We observed up to 81% improvements in average job response times and 78% in system throughput, compared to common solutions like First-Fit and Best-Fit. Also, all of the state-of-the-art schedulers we considered together with our allocators significantly benefited from our algorithms, while no degradation in resource utilization was observed compared to First-Fit and Best-Fit, thus confirming our algorithms as effective alternatives to them.

Although our study is based on a particular HPC system (Eurora) and its workload, our results help us to characterize the operating conditions in which heterogeneity-aware resource allocation becomes crucial for heterogeneous HPC systems in general. A system may go through different workload types, ranging

from light loads with a small number of jobs requesting few critical resources, to heavy loads with a high number of jobs requesting large amounts of critical resources; and the majority of the jobs in the workload may range from being short, occupying critical resources for short periods, to long, blocking the critical resources for long periods of time. In addition, job submission patterns may fluctuate, keeping the system under different amounts of pressure, ranging from rare to heavy. We observed that protecting the critical resources is most useful when (i) the workload contains a significant amount of long jobs requiring critical resources, without dominating the workload; (ii) the system is under pressure consistently without sudden peaks in job submission patterns.

As future work, we plan to test our allocation algorithms on data from different heterogeneous architectures. We are also interested in the performance on a wider set of operating conditions, which can be tested also by employing a synthetic workload generator. The algorithms can also be improved further. For instance, allocating nodes fully may cause saturation of shared resources such as memory, which could result in decreased system performance. We plan to take this into account and allocate resources in a way that avoids saturation, something which we do not consider at the moment.

Acknowledgements. We thank Dr. A. Bartolini, Prof. L. Benini, Prof. M. Milano, Dr. M. Lombardi and the SCAI group at Cineca for providing access to the Eurora data. We also thank the IT Center of the University of Pisa (Centro Interdipartimentale di Servizi e Ricerca) for providing access to computing resources for simulations. A. Netti has been supported by a research fellowship from the *Oprecomp-Open Transprecision Computing* project. C. Galleguillos has been supported by Postgraduate Grant PUCV 2017. A. Sîrbu has been partially funded by the EU project *SoBigData Research Infrastructure — Big Data and Social Mining Ecosystem* (grant agreement 654024).

References

1. Ashby, S., Beckman, P., Chen, J., Colella, P., Collins, B., Crawford, D., et al.: The opportunities and challenges of exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee pp. 1–77 (2010)
2. Bartolini, A., Borghesi, A., Bridi, T., Lombardi, M., Milano, M.: Proactive workload dispatching on the EURORA supercomputer. In: Proc. of CP 2014. LNCS, vol. 8656, pp. 765–780. Springer (2014)
3. Bhattacharya, S., Tsai, W.: Lookahead processor allocation in mesh-connected massively parallel multicomputer. In: Proc. of IPPS 1994. pp. 868–875. IEEE (1994)
4. Borghesi, A., Collina, F., Lombardi, M., Milano, M., Benini, L.: Power capping in high performance computing systems. In: Proc. of CP 2015. LNCS, vol. 9255, pp. 524–540. Springer (2015)
5. 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 (2016)

6. Buddhakulsomsiri, J., Kim, D.S.: 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) (2007)
7. Cavazzoni, C.: Eurora: a european architecture toward exascale. In: Future HPC Systems: the Challenges of Power-Constrained Performance. ACM (2012)
8. Emeras, J., Ruiz, C., Vincent, J.M., Richard, O.: Analysis of the jobs resource utilization on a production system. In: Proc. of JSSPP 2013. LNCS, vol. 8429, pp. 1–21. Springer (2013)
9. Feitelson, D.G.: Metrics for parallel job scheduling and their convergence. In: Proc. of JSSPP 2001. LNCS, vol. 2221, pp. 188–206. Springer (2001)
10. Galleguillos, C., Kiziltan, Z., Netti, A.: Accasim: an HPC simulator for workload management. In: Proc. of CARLA 2017. Springer (2017)
11. Galleguillos, C., Sirbu, A., Kiziltan, Z., Babaoglu, O., Borghesi, A., Bridi, T.: Data-driven job dispatching in HPC systems. In: Proc. of MOD 2017. Springer (2017)
12. Guim, F., Rodero, I., Corbalán, J.: The resource usage-aware backfilling. In: Proc. of JSSPP 2009. LNCS, vol. 5798, pp. 59–79. Springer (2009)
13. Guim, F., Rodero, I., Corbalan, J., Parashar, M.: Enabling gpu and many-core systems in heterogeneous hpc environments using memory considerations. In: Proc. of HPCC 2010. pp. 146–155. IEEE (2010)
14. Henderson, R.L.: Job scheduling under the portable batch system. In: Proc. of JSSPP 1995. pp. 279–294. Springer (1995), https://pbsworks.com
15. Hentenryck, P.V., Bent, R.: Online stochastic combinatorial optimization. The MIT Press (2009)
16. Wasi-ur Rahman, M., Islam, N.S., Lu, X., Panda, D.K.D.: A comprehensive study of mapreduce over lustre for intermediate data placement and shuffle strategies on HPC clusters. IEEE Transactions on Parallel and Distributed Systems 28(3), 633–646 (2017)
17. Reuther, A., Byun, C., Arcand, W., Bestor, D., Bergeron, B., Hubbell, M., et al.: Scalable system scheduling for HPC and big data. arXiv:1705.03102 (2017)
18. Shmueli, E., Feitelson, D.G.: Backfilling with lookahead to optimize the packing of parallel jobs. Journal of Parallel and Distributed Computing 65(9), 1090–1107 (2005)
19. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 9th Edition. Wiley (2014)
20. Villa, O., Johnson, D.R., Oconnor, M., Bolotin, E., Nellans, D., Luitjens, J., et al.: Scaling the power wall: a path to exascale. In: Proc. of SC 2014. pp. 830–841. IEEE (2014)
21. Wong, A.K.L., Goscinski, A.M.: Evaluating the EASY-backfill job scheduling of static workloads on clusters. In: Proc. of CLUSTER 2007. pp. 64–73. IEEE (2007)
22. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Proc. of JSSPP 2003. pp. 44–60. Springer (2003), http://www.slurm.schedmd.com
23. Zeldes, Y., Feitelson, D.G.: On-line fair allocations based on bottlenecks and global priorities. In: Proc. of ICPE 2013. pp. 229–240. ACM (2013)