AccaSim: a customizable workload management simulator for job dispatching research in HPC systems

# AccaSim: a Customizable Workload Management Simulator for Job Dispatching Research in HPC Systems

**Cristian Galleguillos · Zeynep Kiziltan · Alessio Netti · Ricardo Soto**

**Abstract** We present AccaSim, a simulator for workload management in HPC systems. Thanks to AccaSim's scalability to large workload datasets, support for easy customization, and practical automated tools to aid experimentation, users can easily represent various real HPC systems, develop novel advanced dispatchers and evaluate them in a convenient way across different workload sources. AccaSim is thus an attractive tool for conducting job dispatching research in HPC systems.

**Keywords** HPC systems, workload management system, job dispatching problem, simulation tool, dispatcher development, dispatcher evaluation

## 1 Introduction

High Performance Computing (HPC) systems have become fundamental tools to solve complex, compute-intensive, and data-intensive problems in diverse engineering, business and scientific fields, enabling new scientific discoveries, innovation of more reliable and efficient products and services, and new insights in an increasingly data-dependent world. This can be witnessed for instance in the annual reports[1] of PRACE and the recent report[2] by ITIF which accounts for the importance of HPC to the global economic competitiveness.

Cristian Galleguillos · Ricardo Soto
Pontificia Universidad Católica de Valparaíso, 2362807 Valparaíso, Chile
E-mail: cristian.galleguillos.m@mail.pucv.cl

Cristian Galleguillos · Zeynep Kiziltan · Alessio Netti
University of Bologna, 40126 Bologna, Italy

[1] `http://www.prace-ri.eu/praceannualreports/`

[2] `http://www2.itif.org/2016-high-performance-computing.pdf`

As the demand for HPC technology continues to grow, a typical HPC system receives a large number of variable requests (jobs) by its end users. This calls for the efficient management of the submitted workload and system resources. This critical task is carried out by the *Workload Management System* (WMS) software component. Central to WMS is the *dispatcher* which has the key role of deciding when and on which resources to execute the individual jobs by ensuring high system utilization and performance. An optimal dispatching decision is a hard problem [4], and yet sub-optimal decisions could have severe consequences, like wasted resources and/or exceptionally delayed requests. Efficient job dispatching in an HPC system is thus an active research area, see for instance [9] fo r an overview.

One of the challenges of job dispatching research is the intensive experimentation necessary for evaluating and comparing various dispatchers in a controlled environment. The experiments differ under a range of conditions with respect to the workload, the number and the heterogeneity of resources, and the dispatching algorithms. Using a real HPC system for experiments is not realistic for the following reasons. First, researchers may not have access to a real system. Second, it is impossible to modify the hardware components of a system, and often unlikely to access its WMS for any type of alterations. And finally, even with a real system permitting modifications in its WMS, it is inconceivable to ensure that distinct dispatchers process the same workload, which hinders fair comparison. Therefore, simulating a WMS is essential for conducting controlled dispatching experiments.

The contribution of this paper is the design and implementation of AccaSim, a WMS simulator developed for job dispatching research in HPC systems. AccaSim is an open source, freely available library for Python,

thus compatible with any major operating system, and executable on a wide range of computers thanks to its lightweight installation and light memory footprint. AccaSim is scalable to large workload datasets and provides support for easy customization, allowing to carry out experiments across different workload sources, resource types, and dispatching algorithms. Moreover, AccaSim enables users to develop novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as power and energy consumption and failures of resources. Furthermore, AccaSim aids users in their experiments via automated tools to generate synthetic workload datasets, to run simulation experiments and to produce plots to evaluate dispatchers. The researchers can thus use AccaSim to mimic any real system, including those possessing heterogeneous resources, develop advanced dispatchers using for instance power and energy-aware, fault-resilient algorithms, and test and evaluate them in a convenient way over a wide range of workload sources by using real workload traces or by generating them.

This paper extends an earlier version [13] by providing three new automated tools for workload generation, experimentation and plot generation, as well as a detailed comparison to the relevant existing simulators. In the rest of the paper, after giving the background in Section 2 on WMS in HPC systems, we introduce the architecture and the main features of AccaSim in Section 3. We briefly describe AccaSim's implementation and customization, and show its various instantiations in Section 4. We discuss in Section 5 the related work and contrast AccaSim in Section 6 against the existing relevant simulators. We then present a case study in Section 7 where we showcase AccaSim's use in job dispatching research. We conclude in Section 8.

## 2 Workload Management in HPC Systems

A WMS is an important software of an HPC system, being the main access for the users to exploit the available resources for computing. A WMS manages user requests and the system resources through critical services. A user request consists of the execution of a computational application over the system resources. Such a request is referred to as job and the set of all jobs are known as workload. The jobs are tracked by the WMS during all their states, i.e. from their submission time, to queuing, running, and completion. Once a job is completed, the results are communicated to the respective user. Figure 1 depicts a general scheme of a WMS.

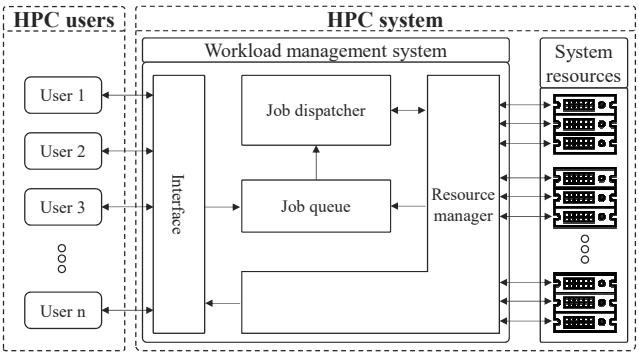


Fig. 1: WMS in an HPC system.

A WMS offers distinct ways to users for *job submission* such as a GUI and/or a command line interface. A submitted job includes the executable of a computational application, its respective arguments, input files, and the resource requirements. An HPC system periodically receives job submissions. Some jobs may have the same computational application with different arguments and input files, referring to the different running conditions of the application in development, debugging and production environments. When a job is submitted, it is placed in a *queue* together with the other pending jobs (if there are any). The time interval during which a job remains in the queue is known as waiting time. The queued jobs compete with each other to be executed on limited resources.

A *job dispatcher* decides which jobs waiting in the queue to run next (*scheduling*) and on which resources to run them (*allocation*) by ensuring high system utilization and performance. The dispatching decision is generated according to a policy using the current system status, such as the queued jobs, the running jobs and the availability of the resources. A suboptimal dispatching decision could cause resource waste and/or exceptional delays in the queue, worsening the system performance and the perception of its users. A (near-)optimal dispatching decision is thus a critical aspect in a WMS.

The dispatcher periodically communicates with a *resource manager* of the WMS for obtaining the current system status. The *resource manager* updates the system status through a set of active monitors, one defined on each resource which primarily keeps track of the resource availability. The WMS systematically calls the *dispatcher* for the jobs in the queue. An answer means that a set of jobs are ready for being executed. Then the dispatching decision is processed by the *resource manager* by removing the ready jobs from the queue and sending them to their allocated resources. Once a job starts running, the *resource manager* turns its state

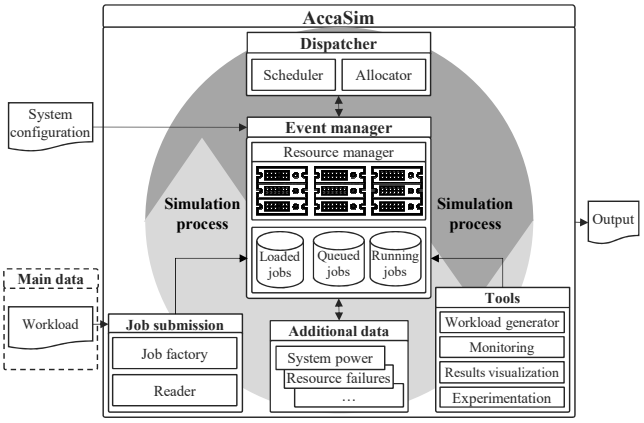


Fig. 2: AccaSim architecture.

from "queued" to "running". The *resource manager* commonly tracks the running jobs for giving to the WMS the ability to communicate their state to their users through the interface, and in a more advanced setting to (let the users) submit again their jobs in case of resource failures. When a job is completed, the *resource manager* turns its state from "running" to "completed" and communicates its result to the interface to be retrieved by the user.

## 3 AccaSim Architecture and Main Features

AccaSim enables to simulate the WMS of any real HPC system with minimum effort and facilitates the study of various issues related to dispatchers, such as feasibility, behavior, and performance, accelerating the dispatching research process. In this section, we present the architecture and highlight the main features of AccaSim.

AccaSim is designed as a discrete event simulator. The simulation is guided by certain events that belong to a real HPC system. These events are mainly collected from the workload and correspond to the job submission, starting and completion times, referred to as $T_{sb}$, $T_{st}$ and $T_c$, resp. The architecture of AccaSim is depicted in Figure 2. Since there are no real users for submitting jobs nor real resources for computation during simulation, the first step for starting a simulation is to define the synthetic system with its jobs and resources.

*Job submission.* This component mimics the job submission of users. The main input data is the workload dataset provided in the form of a file which includes job descriptions. The default *reader* subcomponent reads the input file in Standard Workload Format (SWF)[12] and passes the parsed data to the *job factory* subcomponent for creating the synthetic jobs for simulation, keeping the information related to their identification, submission time, duration and request of system resources. The *job factory* can extend this basic information with additional attributes for the synthetic jobs, such as job duration estimation which is a useful information for many dispatching algorithms [14]. The synthetic jobs are then mapped to the *event manager* component, simulating the job submission process. The main data input is customizable in the sense that any workload dataset file can be used. This is possible thanks to the *reader* which can be adapted easily to parse any workload dataset file format. Consequently, AccaSim can be employed with any workload source corresponding to an existing workload dataset or to a synthetic one produced by a workload generator.

*Event manager.* This is the core component of the simulator, which mimics the behavior of the synthetic jobs and the presence of the synthetic resources, and manages the coordination between the two. Differently from a real WMS, the *event manager* tracks the jobs during their artificial life-cycle by maintaining all their possible states "loaded", "queued", "running" and "completed" via certain events. During simulation, at each time point $t$:

- the *event manager* checks if $t = T_{sb}$ for some jobs. If the submission time of a job is not yet reached, the *event manager* assigns the job the "loaded" state meaning in the real context that the job has not yet been submitted. If instead the submission time of a job is reached, the *event manager* updates its status to "queued";
- the *dispatcher* component gives a dispatching decision on (the subset of) the queued jobs, assigning them an immediate starting time. The *event manager* reveals that $t = T_{st}$ for some waiting jobs and consequently updates their status to "running";
- the *event manager* checks if $t = T_c$ for currently running jobs. Since these jobs were dispatched in a previous time point, their starting and completion times are known. The completion time of a job is the sum of its starting time and duration, which are known from the workload data. If the completion time of a job is reached, the *event manager* updates its status to "completed".

The *resource manager* subcomponent of the *event manager* defines the synthetic resources of the system using a system configuration file as input, and then mimics their allocation and release at the job starting and completion times. Hence, at a time point $t$, if a job starts, the *resource manager* allocates for the job the resources decided by the *dispatcher*; and if it completes,

the *resource manager* releases its resources. The system configuration file can be customized according to the needed types of resources, which renders the simulation of a system possessing heterogeneous resources possible.

AccaSim is designed to maintain a low consumption of memory for scalability to large workload datasets, therefore job loading is performed in an incremental way, loading only the jobs that are near to be submitted at the corresponding simulation time, as opposed to loading them once and for all. Moreover, completed jobs are removed from the system so as to release space in the memory.

*Dispatcher.* This component, responsible for generating a dispatching decision, interacts with the *event manager* for retrieving the current system status regarding the queued jobs, the running jobs, and the availability of the resources. Note that the *dispatcher* is not aware of job durations. This information is known only by the *event manager* to stop the jobs at their completion time in a simulated environment. Therefore, the dispatching decision can be solely based on job duration estimations which are supplied as a job attribute. This has no impact on the execution of jobs, which are always allowed to run for their entire duration, despite the presence of estimation errors. The *scheduler* and the *allocator* subcomponents of the *dispatcher* are customizable according to the algorithms of interest. Currently implemented and available schedulers are: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority (EBF) [36]; and allocators are: First-Fit (FF) which allocates to the first available resource, and Best-Fit (BF) which sorts the resources by their current load (busy resources are preferred first), thus trying to fit as many jobs as possible on the same resource, to decrease the fragmentation of the system.

*Additional data.* It has been shown in the last decade that system performance can be enhanced greatly if the dispatchers are aware of additional information regarding the current system status, such as energy and power consumption of the resources [37,2,5,6], resource failures [22,7], and the heating/cooling conditions [35,3]. The *additional data* component of AccaSim provides an interface to integrate such extra data to the system which can then be utilized to develop and experiment with advanced dispatchers which are for instance energy and power-aware, fault-resilient and thermal-aware. The interface lets receive the necessary data externally from the user, make the necessary calculations together with some input from the *event manager*, all customizable according to the need, and pass back the result to the *event manager* so as to transfer it to the *dispatcher*.

*Output.* The output file contains two types of data. The first regards the execution of the dispatching decision for each job, such as the starting time, the completion time and its resource allocation, which gets updated each time a job completes its execution. This type of data can be utilized to contrast the quality of the dispatching decisions from different perspectives. An example is the effect on synthetic system resource utilization: how many and which resources are used in the system, and how they are distributed over the nodes. Another example is the impact on system performance. With the increasing trend in employing HPCs for real-time applications which cannot tolerate delays [26], some critical aspects of system performance are job response times and system throughput. The second type of output data regards the simulation process, specifically the CPU time required by the simulation tasks like job loading, generation of the dispatching decision, and the total amount of memory used during simulation, which gets updated at each simulation time point. This type of data can be used, for instance, to evaluate the performance of the simulator, as well as the performance of the dispatchers in terms of the time they incur for generating a decision.

*Tools.* The *tools* let users follow the simulation process and facilitate their dispatching experimentation. We will demonstrate their utility in Section 7. The *monitoring* subcomponent includes the *system status* and *system utilization* subcomponents. The *system status* allows tracking the current system status, such as the number of queued jobs, the running jobs, the completed jobs, the availability of the resources, etc. The *system utilization* instead shows in a GUI a representation of the allocation of resources by the running jobs during the simulation.

The *results visualization* subcomponent renders the automatic generation of different types of plots for evaluating the quality of dispatching decisions as well as the performance of the dispatchers. The *experimentation* subcomponent instead renders the automation of complex experiments. After configuring the simulator with a workload dataset, a system to simulate, and a set of dispatchers, the *experimentation* performs a simulation for each dispatcher and then produces comparative plots through the *results visualization*.

When doing dispatching research with a real workload dataset, users could face issues such as the dependency on the real system configuration which hinders

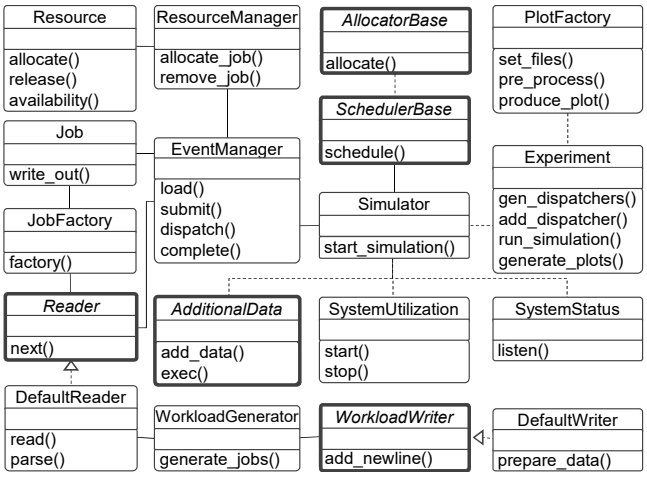


Fig. 3: AccaSim class diagram.

testing with other system configurations, the small size of the dataset preventing scalability tests, or the unavailability of certain data in the dataset for testing specific cases. To tackle this, AccaSim provides a *workload generator* subcomponent which produces a synthetic workload dataset. This subcomponent exploits the data contained in a real workload dataset by mimicking, through statistical methods, its distributions for job submission times, jobs resource requests, and job durations. The generated dataset is written to a file in the SWF format. Other file formats can as well be considered by customizing its subcomponents.

To highlight the *main features*, (i) AccaSim is designed to be scalable to large workload datasets; (ii) AccaSim is customizable in its workload source, resource types, and dispatching algorithms, providing maximum flexibility in representing a WMS; (iii) AccaSim enables users to develop novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as energy and power consumption and failures of the resources; (iv) Accasim provides output data and automated tools to analyze the results, to follow the simulation process and facilitate dispatching experimentation.

## 4 Implementation, Customization, and Instantiation

In this section, we briefly describe AccaSim's implementation and customization, and show its various instantiations. This not only serves to depict the internal organization of AccaSim, but also provides evidence on how easy it is to use and customize.

```
1  from accasim.base.simulator_class import Simulator
2  from accasim.base.scheduler_class import FirstInFirstOut
3  from accasim.base.allocator_class import FirstFit
4  from accasim.utils.plot_factory import PlotFactory
5
6  workload = 'workload.swf'
7  sys_cfg = 'sys_config.json'
8
9  allocator = FirstFit()
10 dispatcher = FirstInFirstOut(allocator)
11 simulator = Simulator(workload, sys_cfg, dispatcher)
12 output_file = simulator.start_simulation()
13
14 plot_factory = PlotFactory('decision', sys_cfg)
15 plot_factory.set_files(output_file, 'my_plot')
16 plot_factory.produce_plot('slowdown')
```

Fig. 4: A basic AccaSim instantiation.

AccaSim is implemented in Python which is an interpreted, object-oriented, high-level programming language, freely available for any major operating system, and is widely used in academia and industry.[3] All the dependencies used by AccaSim are part of Python 3.5 and newer versions, except the matplotlib, scipy, sortedcontainters and psutil packages which can be easily installed using the pip management tool. The source code is available under MIT License. User and API documentations can be found on the AccaSim website.[4] A release version is available as a package in the PyPi repository.[5] Customization is driven by the abstract classes and the inheritance capabilities of Python. The UML class diagram of the main classes is shown in Figure 3 where the abstract classes associated to the customizable components are highlighted in bold.

*The simulator.* A basic AccaSim instantiation is detailed in Figure 4. A simulator object is created in line 11 by instantiating the *Simulator* class. It receives as arguments a workload dataset file in, for instance, SWF, a system configuration file in JSON format, and a dispatcher object, with which the synthetic system is generated and loaded with all the default features.

The workload dataset file is handled by an implementation of the abstract *Reader* class, which is the SWF-based *DefaultReader* by default. The file is read and parsed by the *read()* and *parse()* methods. By implementing the *Reader* class appropriately, AccaSim can be customized to read any workload dataset file format beyond SWF, or to read workloads from any source, not necessarily from a file. The system configuration file, which is processed by the *ResourceManager* class, defines the synthetic resources. The file has two main contents. The first specifies the resource types and their

[3] https://www.python.org/events/python-events/
[4] http://accasim.readthedocs.io/en/latest/
[5] https://pypi.org

quantity in a node belonging to a group, which is useful for modeling HPC systems possessing heterogeneous resources. The second, instead, defines the number of nodes of each group. See Figure 7 for an example. The user is free to mimic any real system by customizing this configuration file suitably.

The dispatcher object is composed by implementations of the abstract *SchedulerBase* and *AllocatorBase* classes. Both classes must implement their main methods, *schedule()* and *allocate()* respectively, to deal with the scheduling and the allocation decisions of the dispatching. This illustrative instantiation exemplifies a specific instance of the *Simulator* class, using as scheduler the *FirstInFirstOut* class, which implements *SchedulerBase* with FIFO, and as allocator the *FirstFit* class, which implements *AllocatorBase* using FF. Both the *FirstInFirstOut* and *FirstFit* classes are available in the library for importing, as done in lines 2-3 of Figure 4. AccaSim can be customized in its dispatching algorithm by implementing the abstract *SchedulerBase* and *AllocatorBase* classes as desired.

In line 12, the *start_simulation()* method launches the simulation with the following optional arguments:

```
simulator.start_simulation(
                    system_status=True,
                    system_utilization=True,
                    additional_data=None)
```

which serve to require the use of the *system status*, the *system utilization*, and the *additional data* tools of the simulator. The additional_data argument is an array of objects where each object is an implementation of the abstract *AdditionalData* class, giving the possibility to customization in terms of the extra data that the user may want to provide to the system for dispatching purposes. After the simulation is finished, the output data file is returned.

The last three lines in Figure 4 serve to use the automated plot generation tool. In line 14, the *PlotFactory* class is instantiated using two arguments. The first indicates the plot type to be produced, as a decision-related or performance-related type. A decision-related plot shows metrics related to the quality of the dispatching decision, such as the job slowdown [11] or queue size, while a performance-related plot serves to show metrics related to the performance of the dispatcher, such as the average CPU time at a simulation time point. Examples of such plots will be shown in Section 7. The second argument is instead the system configuration file which is necessary for the resource specific plots. In line 15, the output file of the simulator is set to be analyzed through the *set_files()* method, together with a label to be used in the plots. Finally,

```
5  [...]
6  from accasim.base.scheduler_class import
       ↪ ShortestJobFirst
7  from accasim.experimentation.experiment import
       ↪ Experiment
8
9  experiment = Experiment('my_experiment', workload,
       ↪ sys_cfg)
10 sched_list = [FirstInFirstOut, ShortestJobFirst]
11 alloc_list = [FirstFit]
12 experiment.gen_dispatchers(sched_list, alloc_list)
13 experiment.run_simulation()
```

Fig. 5: An AccaSim instantiation using the experimentation tool.

the *produce_plot()* method produces the desired plot as specified in its argument.

*The experimentation tool.* In Figure 5, an AccaSim instantiation that uses the *experimentation* tool is detailed. The first 4 lines related to imports and assignment statements are the same as lines 2, 3, 6 and 7 in Figure 4 and are therefore omitted. An experiment object is created in line 9 by instantiating the *Experiment* class which takes as arguments the name of the experiment (which is used to name the output directory as well), the workload dataset file, and the system configuration file, along with the the optional arguments supported by the *Simulator* class. In line 12, the dispatchers of interest are generated through the *gen_dispatchers()* method, which accepts as arguments a list of scheduler and allocator classes. In this illustrative instantiation of the *Experiment* class, we use the *FirstInFirstOut* and the *ShortestJobFirst* classes which implement FIFO and SJF scheduling, as well as the *FirstFit* class which implements the FF allocation. All these classes are available in the library for importing, as done in lines 6-7 of Figure 5. The *gen_dispatchers()* method then automatically creates the dispatchers corresponding to all possible combinations between the schedulers and the allocators, facilitating greatly the conduction of experiments on large sets of dispatchers. If users wish to experiment with a specific dispatcher, it can be formed by instantiating the corresponding implementation of *SchedulerBase* and then passing the object to the *add_dispatcher()* method, similarly to what we have shown in the lines 9-11 in Figure 4 when instantiating the *Simulator* class. Finally in line 13, the experiment is launched with the *run_simulation()* method which performs simulations for all configured dispatchers and produces all the available plots.

*The workload generator tool.* The workload dataset file can refer to a real workload dataset extracted from an HPC system, or to a synthetic one generated through

```
1 from accasim.experimentation.workload_generator import
      ↪ WorkloadGenerator
2
3 workload = 'real_workload.swf'
4 sys_cfg = 'sys_config.json'
5 performance = {'core': 1.667}
6 request_limits = {'min': {'core': 1, 'mem': 256}, 'max': {'
      ↪ core': 8, 'mem': 1024}}
7
8 gen = WorkloadGenerator(workload, sys_cfg,
      ↪ performance, request_limits)
9 jobs = gen.generate_jobs(500000, 'new_workload.swf')
```

Fig. 6: A basic workload generator instantiation.

an external workload generator such as AccaSim's own *workload generator* tool. Figure 6 shows its basic instantiation. A generator object is created in line 8 via the *WorkloadGenerator* class which is available in the library for importing, as done in line 1. It receives as arguments a real workload dataset file to be mimicked, a system configuration file, and variables regarding performance and request limits. The performance variable is a dictionary storing the performance of each processing unit as a unit-value pair. The request_limits variable instead defines the minimum and maximum request of each resource type available in the system. Finally, the jobs are generated in line 9 using the *generate_jobs()* method, which receives as arguments the number of jobs and the name of the output file in which the generated workload dataset is saved.

As in the case of the simulator, the input workload dataset file is parsed by an implementation of the abstract *Reader* class, which is *DefaultReader* and implements an SWF reader by default. The output file is instead written through an implementation of the abstract *WorkloadWriter* class, which is the SWF-based *DefaultWriter* by default. Similar to the *Reader*, the output file format can be customized by implementing the *WorkloadWriter* suitably. It is also possible to customize the job generation process via the optional arguments of the *WorkloadGenerator* constructor, as detailed in the AccaSim documentation.

## 5 Related Work

HPC systems have been simulated from distinct perspectives, for instance to model their network topologies [1,18,27] or storage systems [33,31]. There also exist simulators dealing with the duties of a WMS, as in our work, which are mainly focused on job submission, resource management and job dispatching.

To the best of our knowledge, the WMS simulators most similar to AccaSim are ScSF, Batsim, and Alea. The ScSF simulator [32] emulates a real WMS, Slurm Workload Manager[6], which is popular in many HPC systems. In [25,34] Slurm is modified to provide synthetic job submission, resource management and job dispatching through distinct daemons which run in diverse virtual machines and which communicate over RPC calls, and a dedicated simulator is implemented. ScSF extends this simulator with automatic generation of synthetic job descriptions based on statistical data, but does not give the possibility to read real workload datasets. The dependency on a specific WMS complicates the customization, and together with the additional dependency on virtual Machines and MySQL, the set up of ScSF is rather complex. Moreover, ScSF requires a significant amount of resources in the machines where the simulation will be executed.

Batsim [10] is developed on top of the SimGrid simulation framework.[7] Batsim decouples the dispatcher from the simulator and allows it to be implemented in any programming language, yet both the simulator's and the dispatcher's source code and binaries are available only for GNU/Linux. Batsim takes as input a file in a JSON-based format, and provides a script to translate from SWF with which it is possible to read real workload datasets. However, all jobs are loaded in memory at the beginning of simulation which can hinder the performance when experimenting with a large number of jobs. While users can define different resource types as supported by SimGrid, the concept of a single node possessing heterogeneous resources is not natively implemented in the simulator. This calls for significant effort when users wish to model a system using heterogeneous resources. The dispatchers need to be adapted as well in order to take into account the new representation of a system. Similar to AccaSim, additional data regarding the current system status can be used in Batsim for instance, to model the energy consumption of the system. The type of data, however, depends exclusively on the capabilities of SimGrid. And finally, while Batsim includes a workload generator, it is simple, useful for testing purposes only, and is not intended for dispatching research.

Alea [19] is developed on top of the GridSim simulation framework.[8] Job submission, resource management and job dispatching are driven by the predefined workload format, resource types, and dispatchers. The implementation in Java is open-source and cross-platform. However, any customization to the simulator needs to be done at the source code level, which can be complicated and error-prone. PYSS [23,21,28] and OCS [15] have similar characteristics to Alea, but provide less ad-

[6] https://slurm.schedmd.com/
[7] http://simgrid.gforge.inria.fr/
[8] http://www.cloudbus.org/gridsim/

vanced WMS features as they are developed primarily for a specific research work in dispatching. In general, simple simulators like PYSS and OCS hinder the design of novel advanced dispatchers and their evaluation which requires a more flexible way to represent a WMS.

In [16], an energy aware WMS simulator, called Performance and Energy Aware Scheduling (PEAS) simulator is described. With the main aim being to minimize the energy consumption and to increase the throughput of the system, PEAS uses predefined dispatchers and workload dataset file format, and the system power calculations are based on fixed data from SPEC benchmark[9] considering the entire processor at its max load. PEAS is available only as GNU/Linux binary, therefore it is not customizable in any of these aspects.

Brennan et al. [8] define a framework for WMS simulation, called Cluster Discrete Event Simulator (CDES), which uses predefined scheduling algorithms and relies on specific resource types. Although CDES allows reading real workload datasets for job submission, it loads all jobs in memory at the beginning of the simulation, like Batsim does. Moreover, the implementation is not available which prevents any form of customization.

In [17], a WMS simulator based on a discrete event library called Omnet$^{++}$[10] is introduced. Similar to ScSF, only automatically generated synthetic job descriptions are accepted for job submission. Since Omnet++ is primarily used for building network simulators and is not devoted to workload management, there exist issues such as the inability to consider different types of resources as in CDES. Moreover, due to lack of documentation, it is hard to understand to what extent the simulator is customizable.

The main issues presented in the existing WMS simulators w.r.t. to AccaSim can be summarized as complex set up and need of many virtual machines and resources, inflexibility in the workload source and resource types, limited support for additional data, potential performance degrade with large workload datasets, difficulty or the impossibility of the customization of the WMS, platform restriction, and unavailable or undocumented implementation. As AccaSim is developed for facilitating job dispatching research in HPC systems, it is designed to be scalable to large workload datasets and provides maximum flexibility in representing a WMS in terms of workload source, resource types, and dispatchers. It is open-source and cross-platform, simple to install and use, and is easy to customize via abstract class implementations without having to touch the source code.

[9] `https://www.spec.org/power_ssj2008/`
[10] `http://www.omnetpp.org/`

## 6 Comparison of Simulators

In this section, we contrast AccaSim with a critical attention against ScSF, Batsim and Alea which are the most similar simulators to AccaSim.

### 6.1 Comparison to ScSF

ScSF[11] is a complex framework which needs an entire testing environment for running. The environment should have at least two real or virtual machines with dedicated resources, enough hard disk space for the simulator and its components, and external applications such as a database. The network connection is also a key point in the simulation, since it is required to have a low latency in order to maintain a fast link between its components. We do not compare AccaSim to ScSF experimentally for the following reasons. First, the physical resources needed for experimentation with ScSF are much more than those required by AccaSim. Second, the processes involved in a simulation are more complicated, and they are not encapsulated in a single parent process, as in AccaSim, which hinders a fair comparison. For instance, there are processes that are executed in the MySQL database or that depend on ssh connections, which can affect the performance evaluation. Third, job submission in ScSF is performed only by its own workload generator which restricts the experiments to the synthetic jobs generated by ScSF itself.

### 6.2 Comparison to Batsim and Alea

We here conduct an experimental study to compare the performance of AccaSim to Batsim and Alea using three real workload datasets, which are freely available in SWF. The study is performed on an Ubuntu 16.04 machine with an Intel Core i7-2600 CPU, 16 GB of RAM and a WD10EZEX HDD with 1 TB of capacity. The software used for each simulator experiment are AccaSim 1.0 with Python 3.6.5, Batsim 2.0.0 with Batsched 1.2.0, and finally Alea 4.0 with OpenJDK 1.8.0_171 and 4 GB of max. heap size. All the scripts used to setup and run to experiments, and to evaluate their results are available on the AccaSim GitHub repository.[12]

*Workload datasets* It is important to compare the simulators' performance on datasets diverse in terms of size and time span, so as to derive robust conclusions

[11] `http://frieda.lbl.gov/download`
[12] `https://git.io/fhmbM`

on their behavior, especially on how they scale up to large workload datasets. The three datasets on which the experiments are based differ in these aspects. They range from medium-size to very large-size, and they are created in time periods ranging from a decade ago to recent years. The first dataset is based on a workload trace collected from the Seth cluster[13] which was part of the High Performance Computing Center North of the Swedish National Infrastructure for Computing. The dataset file is available on-line[14] in SWF, and it contains 202,871 jobs spanning through 4 years, from July 2002 to January 2006. Seth was composed of 120 nodes, 480 cores and 120 GB of RAM in total.

The workload trace on which the second dataset is based is collected from the RICC supercomputer [29] which was part of RIKEN, an independent scientific research and technology institution of the Japanese government. The dataset file is available on-line[15] SWF, and it contains 447,794 jobs spanning through 5 months, from May 2010 to September 2010. RICC was a massively parallel cluster, which was composed of 1,024 nodes, 8192 cores and 12 TB of RAM in total.

The last workload dataset is based on a workload trace collected from the MetaCentrum Czech National Grid [20]. The dataset file is available on-line[16] in SWF, and it contains 5,731,100 jobs spanning through 2 years, from Jan 2013 to Apr 2015. The MetaCentrum grid [17] is composed of several clusters, the composition of which has changed over the time. During the recorded period, it was composed of 19 clusters with 495 nodes, 8412 cores and 10 TB of RAM in total.

*Experimental setup* Each experiment corresponds to the simulation of one of the three workload datasets using one of the three simulators. In order to isolate the core actions of a simulator from external factors, such as non-optimal dispatcher implementations, we use a dispatcher which rejects any submitted job. While the rejecting dispatcher is available in AccaSim and Batsim, we implemented it ourselves in Alea. We evaluate the simulators' performance in terms of the total CPU time required to run an experiment and memory footprint. To do so, we use a script which sequentially runs each experiment and repeats it 10 times as a child program in a new process so as to obtain reliable and representative results. The script records each experiment's start and ending time, and gathers the memory consumption every 10ms by using the Python psutil library.[18]

Batsim[19] is conveniently packaged in the Nix package manager for an easy and clean installation on any Linux distribution with superuser privileges. Batsim does not accept SWF in input, and instead provides a script to convert SWF into the required format. This script also works as a workload preprocessor which removes jobs with incomplete or erroneous data. The CPU time and memory consumption of this preprocessing phase is not considered in the Batsim performance result. Instead in AccaSim and Alea, a similar preprocessing is carried out during job submission, therefore the corresponding CPU time and memory consumption are included in the AccaSim and Alea performance results.

Alea[20] is distributed as a Netbeans Java project in which the entire source code is available. All dependencies and a sample simulation configuration are provided. As opposed to Batsim, Alea accepts SWF in input. However, Alea needs the number of expected jobs in the simulation. Since the number of jobs in the workload may reduce during the preprocessing step, a mismatch with the workload size may crash the job submission process. We indeed faced the problem with the Seth dataset and worked around it by using a number of jobs (200,500), obtained by trial and error, lower than the size of the workload (202,871). Another issue in Alea is that it includes hardcoded instructions for specific datasets or systems which may have to be modified for recent or custom datasets. This kind of implementation makes Alea rather difficult to use.

*Experimental results* We present the results in Table 1, where the MetaCentrum dataset is abbreviated as MC, the total CPU time spent in an experiment is expressed in MM:SS, and the memory usage is expressed with its average and maximum values in MB. The reported values of an experiment are aggregated across all the 10 iterations, and both mean ($\mu$) and standard deviation ($\sigma$) are shown. Across the same dataset and metric, the best results are indicated in bold.

It is clear to see that AccaSim uses up much less memory than the other simulators due to its incremental job loading and job removal capability. This approach is shared by Alea which shows better performance than Batsim. As was discussed in Section 5, Batsim loads in memory the preprocessed data from the workload at the beginning of the simulation, which clearly hinders the performance when experimenting with a large workload dataset. As for the total CPU

[13] https://www.hpc2n.umu.se/resources/hardware/seth

[14] http://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/index.html

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

[16] http://www.cs.huji.ac.il/labs/parallel/workload/l_metacentrum2/index.html

[17] https://metavo.metacentrum.cz/en/index.html

[18] https://pypi.org/project/psutil/

[19] https://github.com/oar-team/batsim

[20] https://github.com/aleasimulator/alea/

| Workload | | | | Simulator | | |
|---|---|---|---|---|---|---|
| | | | | AccaSim | Batsim | Alea |
| Seth | Total time (MM:SS) | | $\mu$ | **00:15** | 00:34 | **00:15** |
| | | | $\sigma$ | 0.2 | 0.5 | 0.5 |
| | Mem. (MB) | Avg. | $\mu$ | **18** | 596 | 161 |
| | | | $\sigma$ | 0.1 | 2.5 | 5.4 |
| | | Max. | $\mu$ | **18** | 964 | 209 |
| | | | $\sigma$ | 0.1 | 0.2 | 23.7 |
| RICC | Total time (MM:SS) | | $\mu$ | 00:27 | 01:03 | **00:24** |
| | | | $\sigma$ | 0.5 | 0.7 | 0.2 |
| | Mem. (MB) | Avg. | $\mu$ | **21** | 1,220 | 162 |
| | | | $\sigma$ | 0.1 | 5.4 | 5.6 |
| | | Max. | $\mu$ | **26** | 2,072 | 272 |
| | | | $\sigma$ | 0.1 | 0.1 | 52.3 |
| MC | Total time (MM:SS) | | $\mu$ | **06:23** | 29:29 | 09:08 |
| | | | $\sigma$ | 4.1 | 14.2 | 3.7 |
| | Mem. (MB) | Avg. | $\mu$ | **19** | 12,647 | 195 |
| | | | $\sigma$ | 0.1 | 137.2 | 17.4 |
| | | Max. | $\mu$ | **19** | 15,431 | 1,165 |
| | | | $\sigma$ | 0.2 | 6.7 | 234.4 |

Table 1: Performance comparison of AccaSim, Batsim and Alea.

time, AccaSim and Alea show competitive results. Despite AccaSim's more general and costly approach in creating synthetic jobs that can have additional attributes with respect to Alea, the results are close with the medium-size Seth and large-size RICC datasets. AccaSim shows the best results with the very large-size MetaCentrum dataset. Batsim's performance worsens, as the workload size increases. This can be explained by its high memory consumption. In general, when an application requires high amount of memory, the OS has to employ auxiliary data structures at the expense of reduced performance. In addition, Batsim is not optimized for fixed-length job execution models, but rather for models which take into account network and CPU contention.

We can conclude that, Accasim is scalable to large workload datasets, and overall it performs much better than the similar simulators Batsim and Alea.

## 7 Case Study

In this section, we present a case study to illustrate ~~'s~~ AccaSim use in job dispatching research. We here focus primarily on dispatcher evaluation and synthetic workload generation. AccaSim can as well be used to develop advanced dispatchers, see [14] for an example. We leave further examples of dispatcher development in AccaSim to future work.

The experimental study conducted in this section is performed on a CentOS 7.3 machine with two Intel Xeon E5-2630 v3 CPUs, 128GB of RAM, using Python 3.6.5 and Accasim 1.0. All the scripts used to setup and run the experiments, and to evaluate their results are available on the AccaSim GitHub repository. [21]

```
{
    "system_name": "Seth - HPC2N",
    "start_time": 1027839845,
    "equivalence": {
        "processor": {
            "core": 2
        }
    },
    "groups": {
        "g0": {
            "core": 4,
            "mem": 1000000
        }
    },
    "resources": {
        "g0": 120
    }
}
```

Fig. 7: System configuration of Seth.

### 7.1 Experimental setup for dispatcher evaluation

To conduct the experimental study regarding dispatcher evaluation, we use the Seth dataset introduced in Section 6, given its reasonable size for proof of concept. The corresponding synthetic system configuration is shown in Figure 7. Since multiple jobs can co-exist on the same node, we consider a better representation of the system, made of cores instead of processors. We note that AccaSim can as well be used to simulate an HPC system possessing heterogeneous resources, such as the Eurora system, as was shown in [30].

As for dispatchers, we employ all the implemented and available dispatchers of AccaSim which are composed of all combinations between the schedulers: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority (EBF); and the allocators: First Fit (FF) and Best Fit (BF). To run the experiments, we conveniently use the *experimentation* tool of AccaSim, as was shown in Figure 5. Each experiment corresponds to the simulation of the Seth workload using a specific dispatcher, and is repeated 10 times so as to obtain reliable and representative results.

### 7.2 Dispatcher evaluation

Dispatchers can be evaluated and compared from different perspectives thanks to AccaSim's tools and output data. In Figures 8 and 9, sample snapshots taken by the two components of the *monitoring* tool at certain

[21] `https://git.io/fhmba`

```
> python status-sim.py -h
Usage: status-sim.py [-h] [-usage] [-progress] [-all] [-ip
HOSTIP]

AccaSim System Status

optional arguments:
  -h, --help  show this help message and exit
  -usage      Request current virtual resource usage.
  -progress   Request current local progress.
  -all        Request all previous data.
  -ip HOSTIP  IP of server machine.

> python status-sim.py -all
-  Current test instance: workloads/HPC2N-2002-2.2.1-cln.swf
        Completion percentage: 0.97%
        Current simulated time : 2002-09-19 10:59:18
        Loaded 11, Queued 0, Running 19, and Finished 710 Jobs
        Resource Utilization: core: 90.83%, mem: 14.61%
        Real elapsed time : 12.70 secs
```

Fig. 8: System status.

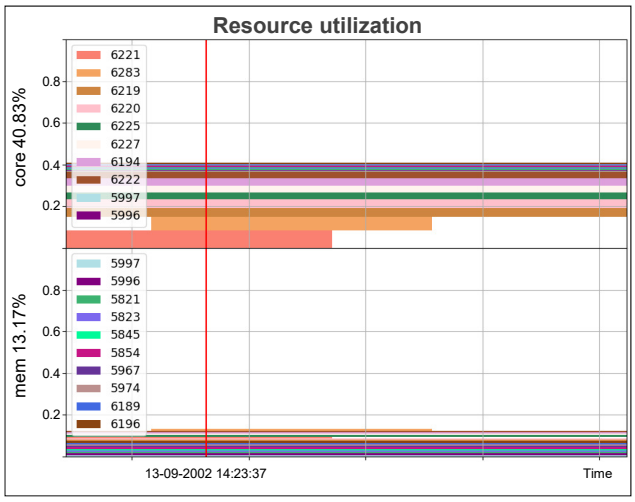


Fig. 9: System visualization.

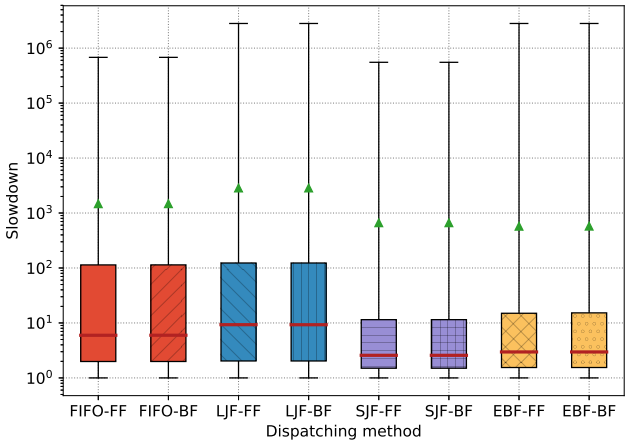


Fig. 10: Distributions for job slowdown.

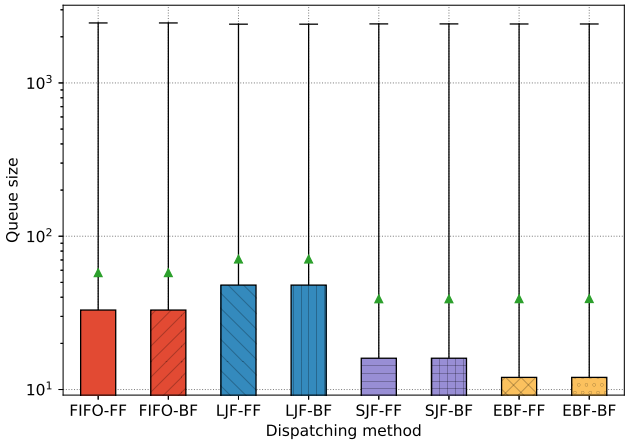


Fig. 11: Distributions of queue size.

time points during the FIFO-FF experiment are shown. The *system status* tool receives command line queries to show a variety of information regarding the current synthetic system status, such as the queued jobs, the running jobs, the completed jobs, resource utilization, the current simulation time point, as well as the total CPU time elapsed by the simulator. The *system visualization* tool summarizes the allocation of resources by the running jobs each indicated with a different color, using an estimation (such as wall-time) for job duration. The display is divided by the types of synthetic resources. In our case study, the core and memory usage are shown separately.

The *experimentation* tool automatically generates plots to compare the dispatchers according to their effect on system utilization, job response times, system throughput, and their performance in terms of the time they incur for generating a decision. For job response times and system throughput, two metrics are used. The first is the job slowdown, a common indicator for evaluating job scheduling algorithms [11], 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 $T_{w,j}$ is the waiting time and $T_{r,j}$ is the duration of job $j$. 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.

In Figures 10 and 11, we present the automatically-generated box-and-whisker plots showing the distributions of the slowdown and the queue size for each exper-

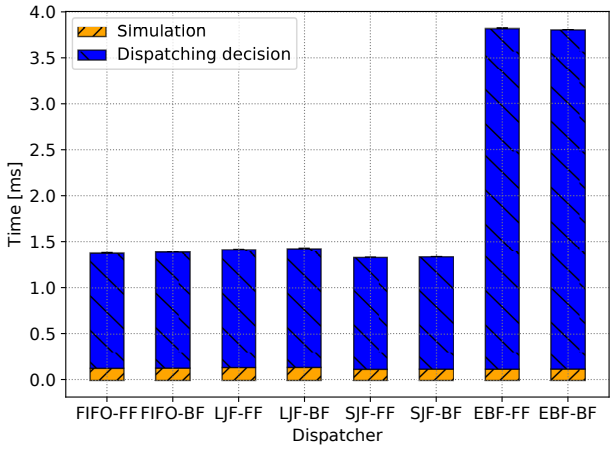


Fig. 12: Average CPU time at a simulation time point.

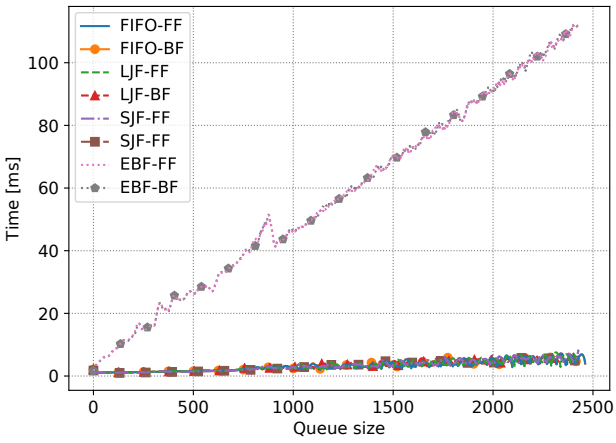


Fig. 13: Average CPU time at a simulation time point to generate a dispatching decision w.r.t. queue size.

iment. We can see that SJF and EBF-based dispatchers achieve the best results, independently of their allocators probably due to the homogeneous nature of the synthetic system. Their slowdown values are mainly lower than the median of the FIFO and LJF-based dispatchers. SJF maintains overall lower slowdown values than the others, but a higher mean than the EBF. SJF maintains also slightly higher mean in the queue size than the EBF. The scheduling algorithm of EBF does not sort the jobs, like SJF, instead it tries to fit as many jobs as possible into the system, which can explain the best average results achieved in terms of slowdown and queue size.

In Figure 12, we present the automatically-generated plot which shows the average CPU time required at a simulation time point for each dispatcher. The average CPU time of an experiment is obtained by aggregating the data from all its 10 iterations. The time spent in simulation, other than generating the dispatching decision, is constant (around 0.2 ms) across all the experiments, and the EBF-based dispatchers spend much more time in generating a decision than the others. In Figure 13, we instead present the automatically-generated plot that analyzes the scalability. Specifically, it reports for each queue size the average CPU time spent at a simulation time point in generating a dispatching decision. Also in this case, we considered the data related to all 10 iterations of the experiments. While all the dispatchers scale well, the EBF-based dispatchers require more CPU time for processing bigger queue sizes, due to their scheduling algorithm which tries to fit as many jobs as possible into the system.

AccaSim users are free to analyze the output data as they wish to evaluate the dispatchers further. For instance, to compare in more detail the dispatchers' performance, they can extract the total usage of CPU time and memory of each experiment, as reported in Table 2. In the table, the time columns correspond to the total CPU time spent by the simulator and the time spent in generating the dispatching decision; whereas the memory columns give the average and the maximum amount of memory utilized over the entire simulation time points. The reported values of an experiment are aggregated across all the 10 iterations, and both mean ($\mu$) and standard deviation ($\sigma$) are shown.

Most of the experiments took around 8 minutes. The exceptions are the EBF-based experiments which require around 22 minutes because the underlying dispatching algorithms are computationally more intensive. In accordance with Figure 12, the time spent by the simulator, other than generating the dispatching decision, is constant (around 40 seconds) across all the experiments. The total CPU usage is thus highly dependent on the complexity of the dispatcher. The average memory usage is around 80MB with a peak at 86MB across all the experiments.

Our analysis restricted to the considered dataset reveals that, while the EBF-based dispatchers give the best results in terms of response times and throughput, they are much more costly in generating a dispatching decision. Simple dispatchers based on SJF are valid alternatives with their excellent scalability and with their comparable results in response times and throughput.

### 7.3 Synthetic workload datasets

In order to generate synthetic workload datasets, and later for comparison purposes, we utilize the Seth and RICC datasets introduced in Section 6. With each, we generate four datasets using different configurations in terms of resource type, processing unit performance,

| Dispatcher | Time (MM:SS) | | | | Memory (MB) | | | |
|---|---|---|---|---|---|---|---|---|
| | Total | | Disp. | | Avg. | | Max. | |
| | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| FIFO-FF | 08:01 | 2.6 | 07:15 | 2.3 | 76 | 0.2 | 82 | 0.3 |
| FIFO-BF | 08:05 | 1.8 | 07:18 | 1.6 | 79 | 0.1 | 85 | 1.1 |
| LJF-FF | 08:13 | 2.4 | 07:24 | 2.1 | 80 | 0.7 | 86 | 0.9 |
| LJF-BF | 08:17 | 2.3 | 07:27 | 2.1 | 81 | 0.8 | 86 | 0.9 |
| SJF-FF | 07:46 | 2.2 | 07:04 | 2.0 | 82 | 0.8 | 86 | 0.5 |
| SJF-BF | 07:49 | 1.7 | 07:06 | 1.5 | 82 | 0.4 | 86 | 0.6 |
| EBF-FF | 22:24 | 2.9 | 21:41 | 2.7 | 82 | 0.6 | 85 | 0.7 |
| EBF-BF | 22:19 | 4.6 | 21:36 | 4.2 | 82 | 0.6 | 84 | 0.8 |

Table 2: Total CPU time and memory usage during the simulation.

and the number of jobs. The first dataset includes 50,000 jobs and a 1.5x improvement in core performance. The second includes 100,000 jobs with double number of nodes. The third includes 200,000 jobs, two GPU accelerator cards for a quarter of the nodes with a performance of 933 GFLOPS per second. Finally, the last includes 500,000 jobs, two GPU accelerator cards for a half of the nodes with a performance of 933 GFLOPS per second and a 1.5x improvement in the core performance. The improved performance and the change in the number of nodes are relative to the system that the workload dataset in consideration belongs to. In the following, we first briefly describe the generation process, and then show the similarity between the real and the generated datasets.

*Synthetic workload dataset generation.* The first aspect to compare between a real and a synthetic workload dataset is the job submission cycle which refers to the job submission times and reflects the usage of the system by its users. The cycles could be represented by certain periods of working time to reflect better the real usage of the system. The *WorkloadGenerator* calculates the submission time of a job $j$ based on a daily cycle model proposed in [24]. In the original algorithm, named Slot Weight Method, a day is represented by 48 slots of 30 minutes each ($s$). Thus, the first slot starts at midnight, the next one at 00:30, and so on. Each slot has a specific weight which is the ratio between the number of jobs belonging to the time slot and the total number of jobs in the real workload dataset, which represents a measure for selecting a slot for $j$. The algorithm generates a random value $v$ between 0 and 5 to represent the maximum number of days that can elapse between $j$ and its predecessor, based on the statistical distribution of the interarrival times of the real workload dataset. For selecting a slot, the algorithm starts from the slot of the predecessor of $j$. The slots are considered as a circular list. For each considered slot, if $v$ is greater or equal to the slot weight, $v$ is updated by subtracting the slot weight. Update continues with the next slot, otherwise, the algorithm stops and selects the current slot. Then, the job submission time of $j$ is calculated by summing the half hours of all the surpassed slots plus the remaining amount of $v$.

We modify this algorithm in two aspects so as to assimilate a real job submission cycle. First, we modify the fixed upper-bound $v_{max}$ of $v$ to the maximum value of the interarrival times of the dataset. Second, we add a dynamic process that modifies $v_{max}$ during a job submission time generation. For this purpose, we calculate the ratio between the number of the currently generated jobs and the required jobs in three different ways in relation to the last submitted hour, the last submitted day, and the last submitted month. This allows to keep the generation of values as similar to the real data as possible. Then, we calculate the progress ratio of each ratio by dividing it by the respective ratios in the real data. The overall progress ratio is the multiplication between all progress ratios ($pr$). Finally, $v_{max}$ is dynamically adapted at each job submission time generation as follows:

$$v_{max} \leftarrow v_{max} - (v_{max} - s) * (1 - pr)$$

If $pr = 1$, the job submission time generation of the predecessor reached the real ratios, thus for $j$, we use $v_{max}$. In addition, when the real data does not include specific months, $pr$ has only hourly and daily ratios.

The second aspect to compare is the theoretical computed FLOPs for each job during its execution in the system, which depends on, among others, its duration and resource requests in terms of resource type restricted to the processing units (e.g., cores, GPU, MIC, etc.) and quantity. These features of a job are generated in three phases. The first phase is based on an algorithm from [24] to select the job type, serial or parallel, and the number of requested nodes. Since this algorithm considers a job parallel if it runs on multiple nodes, we modify it to create parallel jobs on a single node, i.e. when the number of required cores is greater than one. In the second phase, the resource request is defined by

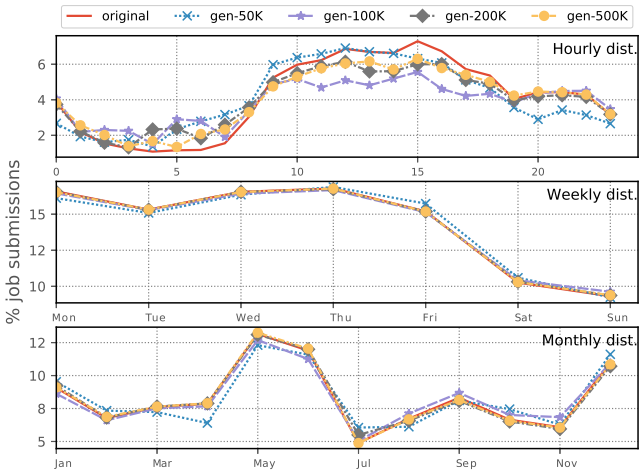


Fig. 14: Seth workload dataset.


Fig. 15: RICC workload dataset.

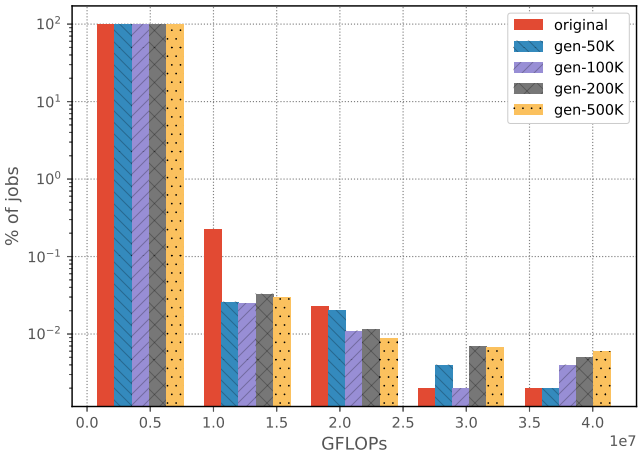


Fig. 16: Seth workload dataset.

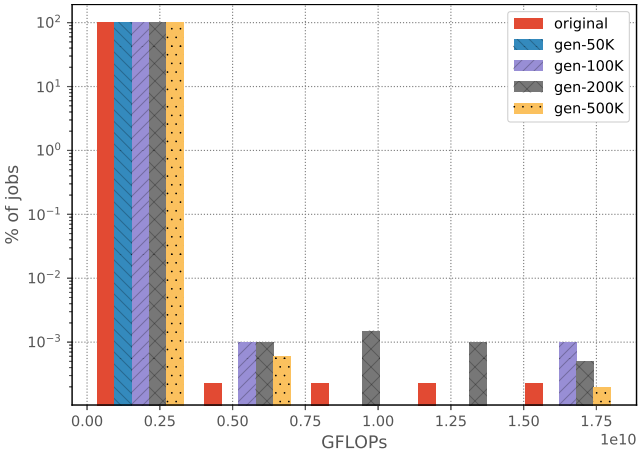


Fig. 17: RICC workload dataset.

randomly choosing among the available resource types and assigning them a quantity, using a uniform distribution and considering the request limits passed as an argument during the *WorkloadGenerator* instantiation, as shown in Figure 6. Finally, in the third phase, the job duration is calculated as the division between (i) a random FLOP value and (ii) the dot product of the resource requests and their corresponding theoretical performance, multiplied by the number of required nodes.

*Comparison to the real workload datasets.* Figures 14 and 15 show the the hourly, daily, monthly job submission distributions of the real and the generated workload datasets. The introduced modifications generate submissions that took place mainly during the working hours, weekdays, and working months, resulting in a more realistic scenario. The generated datasets look very similar to the real datasets, except in the case of the monthly distribution of the RICC dataset. The reason is that the RICC job submissions span to five months, not to an entire year.

Figures 16 and 17 show the distributions of the computed theoretical FLOPS, here represented in GFLOPS, between the real and the generated workload datasets. We observe a similar pattern also here. The usage of the FLOPS calculation for the generation of the jobs' features allows maintaining a distribution similar to the real workload dataset, independent of the configuration of the real system. In this way, the real dataset can be tested with other system configurations using the generated dataset.

## 8 Conclusions

In this paper, we presented AccaSim, a library for simulating WMS in an HPC system, which offers to the researchers an accessible tool to facilitate their job dis-

patching research. The library is open-source, implemented in Python, which is freely available for any major operating system, and works with dependencies reachable in any distribution. It is executable on a wide range of computers thanks to its lightweight installation and light memory footprint. AccaSim is scalable to large workload datasets and provides support for easy customization, allowing to carry out experiments across different workload sources, resource types, and dispatching algorithms. Moreover, AccaSim enables users to develop novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as energy and power consumption and failures of the resources. Last but not least, AccaSim aids users in their experiments via automated tools to generate synthetic workload datasets, to run the simulation experiments and to produce plots to evaluate dispatchers. The researchers can thus use AccaSim to mimic any real system, including those possessing heterogeneous resources, develop advanced dispatchers using for instance power and energy-aware, fault-resilient algorithms, and test and evaluate them in a convenient way over a wide range of workload sources by using real workload traces or by generating them.

In order to highlight the main contributions of AccaSim, we discussed the existing related simulators, presented a critical comparison to the most similar simulators, and showcased AccaSim's use in job dispatching research, specifically in dispatcher evaluation and synthetic workload generation. In future work, we plan to use AccaSim to develop advanced dispatchers using power and energy-aware, fault-resilient algorithms.

**Acknowledgements** C. Galleguillos is supported by Postgraduate Grant PUCV 2018. A. Netti is supported by a research fellowship from the *Oprecomp-Open Transprecision Computing* project. R. Soto is supported by Grant CONICYT/FONDECYT/REGULAR/1160455. We are grateful to Åke Sandgren, Motoyoshi Kurokawa, and the Czech National Grid Infrastructure MetaCentrum, for providing, respectively, the Seth, RICC and the MetaCentrum workload datasets. We thank Alina Sîrbu for fruitful discussions on the work presented here. Finally, we appreciate the precious comments of the reviewers which helped improve the paper significantly. We especially thank Millian Poquet for signing his review and giving us the possibility to interact during the revision of the paper.

## References

1. B. Acun, N. Jain, A. Bhatele, M. Mubarak, C. D. Carothers, and L. V. Kalé. Preliminary evaluation of a parallel trace replay tool for HPC network simulations. In *Proc. of Euro-Par'15 Workshops*, volume 9523 of *LNCS*, pages 417–429. Springer, 2015.
2. A. Auweter, A. Bode, M. Brehm, L. Brochard, N. Hammer, H. Huber, R. Panda, F. Thomas, and T. Wilde. A case study of energy aware scheduling on supermuc. In *Proc. of ISC'14*, volume 8488 of *LNCS*, pages 394–409. Springer, 2014.
3. A. Banerjee, T. Mukherjee, G. Varsamopoulos, and S. K. Gupta. Integrating cooling awareness with thermal aware workload placement for hpc data centers. *Sustainable Computing: Informatics and Systems*, 1(2):134 – 150, 2011.
4. J. Blazewicz, J. K. Lenstra, and A. H. G. R. Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.
5. D. Bodas, J. Song, M. Rajappa, and A. Hoffman. Simple power-aware scheduler to limit power consumption by HPC system within a budget. In *Proc. of E2SC@SC'14*, pages 21–30. IEEE, 2014.
6. A. Borghesi, F. Collina, M. Lombardi, M. Milano, and L. Benini. Power capping in high performance computing systems. In *Proc. of CP'15*, volume 9255 of *LNCS*, pages 524–540. Springer, 2015.
7. J. M. Brandt, B. J. Debusschere, A. C. Gentile, J. Mayo, P. P. Pébay, D. C. Thompson, and M. Wong. Using probabilistic characterization to reduce runtime faults in HPC systems. In *Proc. of CCGRID'08*, pages 759–764. IEEE CS, 2008.
8. J. Brennan, I. Kureshi, and V. Holmes. CDES: an approach to HPC workload modelling. In *Proc. of DS-RT'14*, pages 47–54. IEEE CS, 2014.
9. T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE Trans. Parallel Distrib. Syst.*, 27(10):2781–2794, 2016.
10. P. Dutot, M. Mercier, M. Poquet, and O. Richard. Batsim: A realistic language-independent resources and jobs management systems simulator. In *Proc. of JSSPP'16*, volume 10353 of *Lecture Notes in Computer Science*, pages 178–197. Springer, 2016.
11. D. G. Feitelson. Metrics for parallel job scheduling and their convergence. In *Proc. of JSSPP'01*, volume 2221 of *LNCS*, pages 188–206. Springer, 2001.
12. D. G. Feitelson, D. Tsafrir, and D. Krakov. Experience with using the parallel workloads archive. *J. Parallel Distrib. Comput.*, 74(10):2967–2982, 2014.
13. C. Galleguillos, Z. Kiziltan, and A. Netti. Accasim: An HPC simulator for workload management. In *Proc. of CARLA'17*, volume 796 of *Communications in Computer and Information Science*, pages 169–184. Springer, 2017.
14. C. Galleguillos, A. Sîrbu, Z. Kiziltan, Ö. Babaoglu, A. Borghesi, and T. Bridi. Data-driven job dispatching in HPC systems. In *Proc. of MOD'17*, volume 10710 of *Lecture Notes in Computer Science*, pages 449–461. Springer, 2017.
15. É. Gaussier, D. Glesser, V. Reis, and D. Trystram. Improving backfilling by using machine learning to predict running times. In *Proc. of SC'15*, pages 64:1–64:10. ACM, 2015.
16. C. Gómez-Martín, M. A. Vega-Rodríguez, and J. L. G. Sánchez. Performance and energy aware scheduling simulator for HPC: evaluating different resource selection methods. *Concurrency and Computation: Practice and Experience*, 27(17):5436–5459, 2015.
17. W. B. Hurst, S. Ramaswamy, R. B. Lenin, and D. Hoffman. Modeling and simulation of hpc systems through job scheduling analysis. In *Conference on Applied Research in Information Technology*. Acxiom Laboratory of Applied Research, 2010.
18. N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kalé. Evaluating HPC networks via simulation of parallel workloads. In *Proc. of SC'16*, pages 154–165. IEEE CS, 2016.
19. D. Klusácek and H. Rudová. Alea 2: job scheduling simulator. In *Proc. of SimuTools'10*, pages 61:1–61:10. ICST/ACM, 2010.

20. D. Klusácek, S. Tóth, and G. Podolníková. Real-life experience with major reconfiguration of job scheduling system. In *Proc. of JSSPP'15*, volume 10353 of *Lecture Notes in Computer Science*, pages 83–101. Springer, 2015.
21. J. Lelong, V. Reis, and D. Trystram. Tuning easy-backfilling queues. In *Proc. of JSSPP'17*, volume 10773 of *Lecture Notes in Computer Science*, pages 43–61. Springer, 2017.
22. Y. Li, P. Gujrati, Z. Lan, and X. Sun. Fault-driven rescheduling for improving system-level fault resilience. In *Proc. of ICPP'07*, page 39. IEEE CS, 2007.
23. F. Liu and J. B. Weissman. Elastic job bundling: an adaptive resource request strategy for large-scale parallel applications. In *Proc. of SC'15*, pages 33:1–33:12. ACM, 2015.
24. U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, 2003.
25. A. Lucero. Simulation of batch scheduling using real production-ready software tools. In *Proc. of IBERGRID'11*, pages 345–356. Netbiblo, 2011.
26. N. Mohamed and J. Al-Jaroodi. Real-time big data analytics: Applications and challenges. In *Proc. of HPCS'14*, pages 305–310. IEEE, 2014.
27. M. Mubarak, C. D. Carothers, R. B. Ross, and P. H. Carns. Enabling parallel simulation of large-scale HPC network systems. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):87–100, 2017.
28. P. Murali and S. Vadhiyar. Metascheduling of HPC jobs in day-ahead electricity markets. *IEEE Trans. Parallel Distrib. Syst.*, 29(3):614–627, 2018.
29. M. Nakata. All about RICC: RIKEN integrated cluster of clusters. In *Proc. of ICNC'11*, pages 27–29. IEEE Computer Society, 2011.
30. A. Netti, C. Galleguillos, Z. Kiziltan, A. Sîrbu, and Ö. Babaoglu. Heterogeneity-aware resource allocation in HPC systems. In *Proc. of ISC'18*, volume 10876 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2018.
31. A. Nuñez, J. Fernández, J. D. García, F. García, and J. Carretero. New techniques for simulating high performance MPI applications on large storage networks. *The Journal of Supercomputing*, 51(1):40–57, 2010.
32. G. P. Rodrigo, E. Elmroth, P. Östberg, and L. Ramakrishnan. Scsf: A scheduling simulation framework. In *Proc. of JSSPP'17*, volume 10773 of *Lecture Notes in Computer Science*, pages 152–173. Springer, 2017.
33. S. Snyder, P. H. Carns, R. Latham, M. Mubarak, R. B. Ross, C. D. Carothers, B. Behzad, H. V. T. Luu, S. Byna, and Prabhat. Techniques for modeling large-scale HPC I/O workloads. In *Proc. of PMBS@SC'15*, pages 5:1–5:11. ACM, 2015.
34. T. Stephen and M. Benini. Using and modifying the bsc slurm workload simulator. Technical report, Slurm User Group Meeting, 2015.
35. Q. Tang, S. K. S. Gupta, and G. Varsamopoulos. Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach. *IEEE Trans. Parallel Distrib. Syst.*, 19(11):1458–1472, 2008.
36. A. K. L. Wong and A. M. Goscinski. Evaluating the easy-backfill job scheduling of static workloads on clusters. In *Proc. of CLUSTER'07*. IEEE Computer Society, 2007.
37. Z. Zhou, Z. Lan, W. Tang, and N. Desai. Reducing energy costs for IBM blue gene/p via power-aware job scheduling. In *Proc. of JSSPP'13*, volume 8429 of *LNCS*, pages 96–115. Springer, 2014.