AccaSim: An HPC Simulator for Workload Management

Cristian Galleguillos[1,2(✉)], Zeynep Kiziltan[1], and Alessio Netti[1]

[1] Department of Computer Science and Engineering,
University of Bologna, Bologna, Italy
zeynep.kiziltan@unibo.it, alessio.netti@studio.unibo.it

[2] Escuela de Ing. Informática, Pontificia Universidad Católica de Valparaíso,
Valparaíso, Chile
cristian.galleguillos.m@mail.pucv.cl

Abstract. We present AccaSim, an HPC simulator for workload management. Thanks to the scalability and high customizability features of AccaSim, users can easily represent various real HPC system resources, develop dispatching methods and carry out large experiments across different workload sources. AccaSim is thus an attractive tool for conducting controlled experiments in HPC dispatching research.

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 vital importance of HPC to the global economic competitiveness.

As the demand for HPC technology continues to grow, a typical HPC system receives a large number of variable requests 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 software component *Workload Management System* (WMS). Central to WMS is the *dispatcher* which has the key role of deciding when and on which resources to execute the individual requests by ensuring high system performance and Quality of Service (QoS), such as high utilization of resources and high throughput. An optimal dispatching decision is a hard problem [4], and yet suboptimal decisions could have severe consequences, like wasted resources and/or exceptionally delayed requests. Efficient HPC dispatching is thus an active research area, see for instance [9] for an overview.

One of the challenges of the dispatching research is the amount of experimentation necessary for evaluating and comparing various approaches in a controlled

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

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

E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 169–184, 2018.
https://doi.org/10.1007/978-3-319-73353-1_12

environment. The experiments differ under a range of conditions with respect to workload, the number and the heterogeneity of resources, and dispatching method. 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 approaches process the same workloads, which hinders fair comparison. Therefore, simulating a WMS in a synthetic HPC system is essential for conducting controlled dispatching experiments. Unfortunately, currently available simulators are not flexible enough to render customization in many aspects, limiting the scope of their usage.

The contribution of this paper is the design and implementation of AccaSim, an HPC simulator for workload management. AccaSim is an open source, freely available library for Python, executable in any major operating system. AccaSim is scalable and highly customizable, allowing to carry out large experiments across different workload sources, resource settings, and dispatching methods. Moreover, AccaSim enables users to design novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as power consumption and failures of the resources. The researchers can use AccaSim to mimic any real system by setting up the synthetic resources suitably, develop advanced such as power-aware, fault-resilient dispatching methods, and test them over a wide range of workloads by generating them synthetically or using real workload traces from HPC users. As such, AccaSim an attractive tool for developing dispatchers and conducting controlled experiments in HPC dispatching research.

This paper is organized as follows. After introducing the concept of WMS in Sect. 2, we present in Sect. 3 the architecture and main features of AccaSim, and recap its implementation, instantiation, and customization. In Sect. 4, we show a case study to illustrate the use of AccaSim for evaluating dispatching methods and highlight its scalability. We discuss the related work in Sect. 5 and conclude in Sect. 6.

2 Workload Management System in HPC

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 the jobs are completed, the results are communicated to the respective users. Figure 1 depicts a general scheme of a WMS.

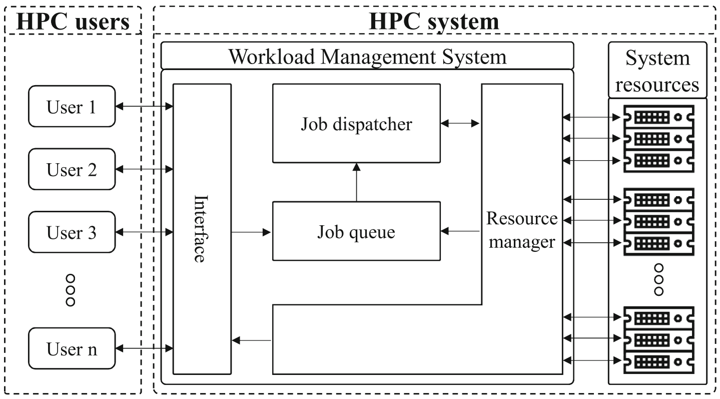


Fig. 1. HPC workload management system.

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 performance and QoS, such as high utilization of resources and high throughput. The dispatching decision is generated according 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 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 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

AccaSim enables to simulate the WMS of any real HPC system with minimum effort and facilitates the study of various issues related to dispatching methods, such as feasibility, behavior, and performance, accelerating the dispatching research process.

In the rest of this section, we first present the architecture and the main features of AccaSim, and then recap its implementation, instantiation and customization.

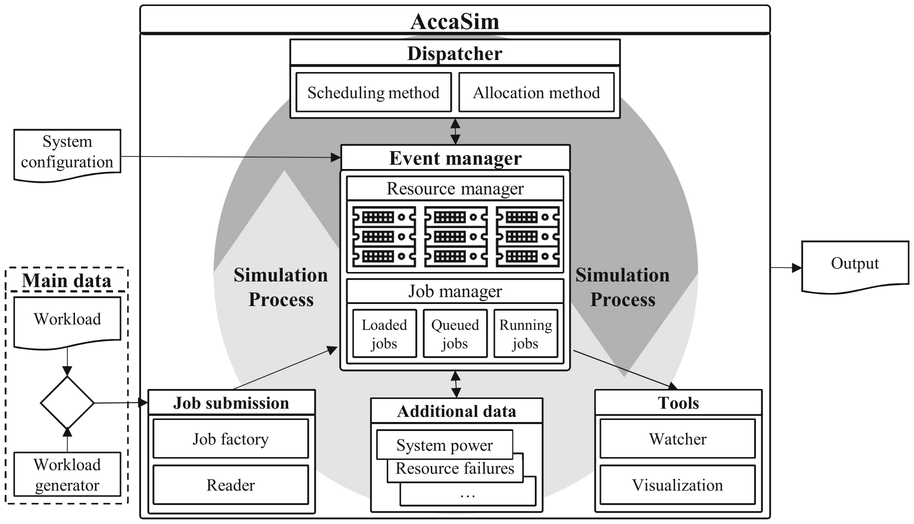


Fig. 2. AccaSim architecture.

3.1 Architecture and Main Features

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 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 Fig. 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.

The *job submission* component mimics the job submission of users. The main input data is the workload provided either in the form of a file, corresponding to the descriptions of the existing workloads, or in the form of an external workload generator producing synthetic workload descriptions based on statistical data. The default *reader* subcomponent reads the input from a file in Standard Workload Format (SWF)[11] 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 created 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 description file and any synthetic workload generator can be used. The *reader* can be adapted easily for this purpose to parse any workload description file format or to read from any workload source.

Event manager is the core component of the simulator, which mimics the behaviour of the synthetic jobs and the presence of the synthetic resources, and manages the coordination between the two. Differently from a real WMS, the *job manager* subcomponent tracks the jobs during their artificial life-cycle by maintaining all their possible states "loaded", "queued", "running" and "completed" via the events handled by the *event manager*. 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 *job 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 *job 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 the *job manager* 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). If the completion time of a job is reached, the *job 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 in 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 in the simulation.

AccaSim is designed to maintain a low consumption of memory for scalability concerns, 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.

The *dispatcher* 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 the job durations. This information is known only by the *event manager* to stop the jobs at their completion time in a simulated environment. The scheduler and the allocator subcomponents of the *dispatcher* are customizable according to the methods of interest. Currently implemented and available methods for scheduling are: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority [20]; and for allocation: First Fit (FF) which allocates to the first available resource, and Consolidate (C) 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.

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 power consumption of the resources [2,5,6,24], resource failures [7,15], and the heating/cooling conditions [3,23]. 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 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*.

We conclude the overview of the architecture with the *tools* component and the output data which help the users to follow the simulation process and analyze the results. The *watcher* 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 *visualization* instead shows in a GUI a representation of the allocation of resources by the running jobs during the simulation. The output data are of two types: (i) the data regarding the execution of the dispatching decision for each job, and (ii) the data related to the simulation process, specifically the CPU time required by the simulation tasks like the generation of the dispatching decision, job loading etc. Such data is useful for analyzing the dispatching results and the performance of the simulation process.

To sum up the main features, AccaSim is customizable in its workload source, resource types, and dispatching methods; AccaSim enables users to design novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as power consumption and failures of the resources; and Accasim provides tools and output data to follow the simulation process and analyze the results.

3.2 Implementation, Instantiation and Customization

AccaSim is implemented in Python (compatible with version 3.4 or above) which is an interpreted, object-oriented, high-level programming language, freely available for any major operating system, and is well established with a large community in academia and industry[3]. All the dependencies used by AccaSim are part of any Python distribution, except the matplotlib and psutil packages which can be easily installed using the pip management tool. The source code is available under MIT License, together with a documentation at http://accasim.readthedocs.io/en/latest/. A release version is available as a package in the PyPi repository[4].

The highly customizable characteristic of AccaSim is driven by its abstract classes and the inheritance capabilities of Python. The UML diagram of the main classes is shown in Fig. 3 where the abstract classes associated to the customizable components are highlighted in bold.

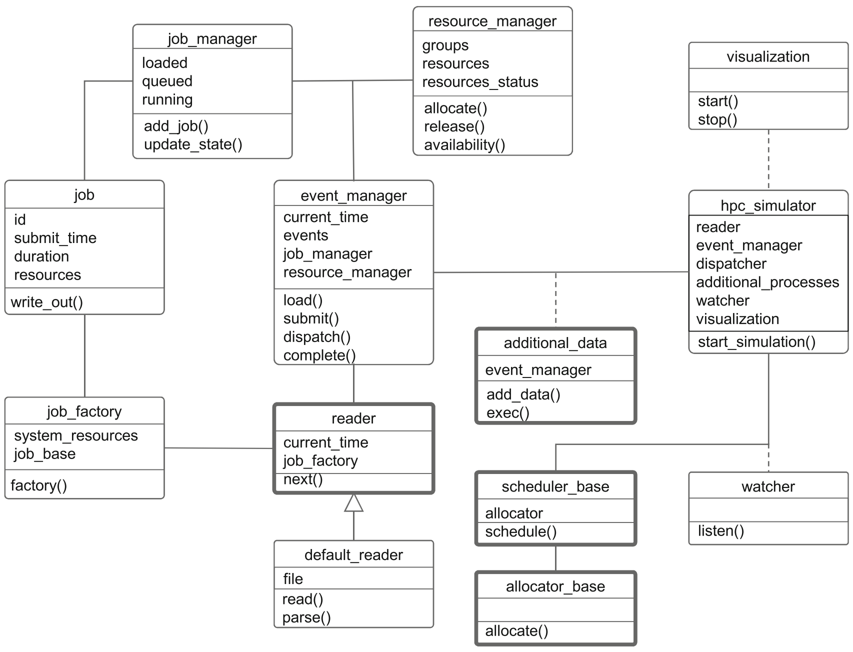


Fig. 3. AccaSim class diagram.

The starting point for launching a simulation is to instantiate the *hpc_simulator* class. It must receive as arguments at least a workload description – such as a file path in SWF format, a system configuration file path in JSON

[3] https://www.python.org/events/python-events/.
[4] https://pypi.org.

format, and a dispatcher instance, with which the synthetic system is generated and loaded with all the default features. An example instantiation is detailed in Fig. 4.

```
1   from accasim.base.simulator_class import hpc_simulator
    from accasim.base.scheduler_class import fifo_sched
3   from accasim.base.allocator_class import ff_alloc

5   workload = '/tmp/workload.swf'
    sys_cfg = '/tmp/sys_config.json'
7
    allocator = ff_alloc()
9   dispatcher = fifo_sched(allocator)
    simulator = hpc_simulator(workload, sys_cfg, dispatcher)
11  simulator.start_simulation()
```

Fig. 4. A sample AccaSim instantiation.

The workload description file is handled by an implementation of the abstract *reader* class, which is *default_reader* by default. The file is read and parsed by the *read()* and *parse()* methods. AccaSim can be customized in its workload description file format by modifying these methods suitably. AccaSim can as well be customized so as to read workloads from any source, not necessarily from a file, by implementing the abstract *reader* class appropriately.

The system configuration file, which is processed by the *resource_manager* class, defines the synthetic resources. The file has two main contents. The first specifies the resource types and their quantity in each group of nodes, which is useful for modeling heterogeneous HPC systems. The second instead determines the number of nodes of each group. See Fig. 5 for an example. The user is free to mimic any real system by customizing this configuration file suitably.

The dispatcher instance is composed by implementations of the abstract *scheduler_base* and *allocator_base* classes. Both classes must implement their main methods, *schedule()* and *allocate()* respectively, to deal with the scheduling and the allocation decisions of the dispatching. In this illustrative instantiation of the *hpc_simulator* class, *fifo_sched* implements *scheduler_base* using FIFO, whereas *ff_alloc* implements *allocator_base* using FF, and both *fifo_sched* and *ff_alloc* classes are available in the library for importing, as done in lines 2–3 of Fig. 4. AccaSim can be customized in its dispatching method by implementing the abstract *scheduler_base* and *allocator_base* classes as desired.

After instantiating the *hpc_simulator* class in line 10 of Fig. 4, the simulation process starts in line 11 with the *start_simulation()* method which has the following optional arguments:

```
simulator.start_simulation(debug=True, watcher=True,visualization=True,
    ↪ additional_data=None)
```

which serve to require the use of a debugger, as well as the *watcher*, the *visualization*, and the *additional data* components of the simulator. The additional_data argument is an array of objects where each object is an implementation of the abstract *additional_data* class, giving the possibility to customize AccaSim in terms of the extra data that the user may want to provide to the system for dispatching purposes.

4 Case Study

In this section, we show a case study to illustrate the use of AccaSim for evaluating dispatching methods in a simulated environment and highlight AccaSim's scalability. The experiments are done on a Macbook Pro machine with Intel Dual Core i5@2.2 Ghz CPU, 8 GB of RAM, and Python 3.6.

Workload source and synthetic system configuration. We rely on a public workload trace collected from the Seth cluster[5] belonging the High Performance Computing Center North (HPC2N) of the Swedish National Infrastructure for Computing. The workload trace file includes 200,735 jobs spanning through 4 years, from July 2002 to January 2006, and is available on-line[6] in the SWF format. Seth was built in 2001 and is already retired by now. It ranked 59th in Top500 list[7], the world's 500 fastest computers. It was composed of 120 nodes, each node with two AMD Athlon MP2000 + dual core processors with 1.667 GHz and 1 GB of RAM. For high parallel performance, the system was equipped with a low latency network. Because multiple jobs can co-exist on the same node, we consider a better representation of the system, made of cores instead of processors. Therefore, we define the synthetic system in the configuration file with 120 nodes each with 4 cores and 1 GB of RAM, as depicted in Fig. 5.

```
{
    "system_name": "Seth – HPC2N",
    "groups": {
        "g0": {
            "core": 4,
            "mem": 1048576
        }
    },
    "resources": {
        "g0": 120
    }
}
```

Fig. 5. System configuration of Seth.

Dispatching Methods. As we previously mentioned in Sect. 3.1, currently implemented and available methods for scheduling are: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority (EBF); and for allocation are: First Fit (FF) and Consolidate (C). In the experiments, we consider every combination of the available

[5] https://www.hpc2n.umu.se/resources/hardware/seth.
[6] http://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/index.html.
[7] http://www.top500.org/.

scheduling and allocation methods, which gives rise to the 8 dispatching methods: FIFO-FF, FIFO-C, SJF-FF, SJF-C, LJF-FF, LJF-C, EBF-FF, and EBF-C. To employ the various dispatching methods, we modify lines 2–3 and 8–9 of Fig. 4 so as to import from the library and use the corresponding implementations of the abstract *scheduler_base* and *allocator_base* classes. Then we run AccaSim 8 times on the entire workload each time with a different dispatching method.

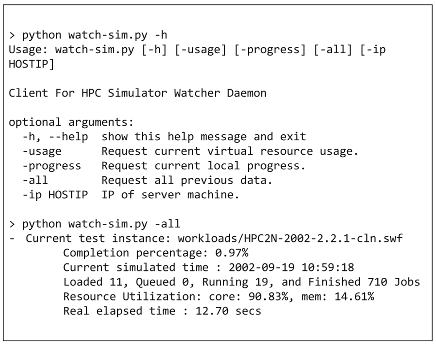
Scalability of AccaSim. We parallelize the experiments in two different threads. The usage of CPU time and memory of each experiment are reported in Table 1, where the time columns correspond to the total CPU time spent by the simulator and the time spent in generating the dispatching decision; whereas the memory columns give the average and the maximum amount of memory utilized over the total simulation time points. In the first thread, FIFO and LJF based experiments are completed in 110 min, while in the second, the SJF and EBF based experiments are completed in 193 min. Each of the experiments took around 30 min. The exceptions are the EBF-based experiments which require around an hour because the underlying dispatching methods are computationally more intensive. As can be observed in the table, the time spent by the simulator, other than generating the dispatching decision, is constant (around 22 min) across all the experiments. The total CPU usage is thus highly dependent on the complexity of the dispatcher.

Table 1. CPU time and memory usage of the simulator.

Thread 1					Thread 2				
Simulator	Time (MM:SS)		Memory (MB)		Simulator	Time (MM:SS)		Memory (MB)	
	Total	Disp.	Avg.	Max.		Total	Disp.	Avg.	Max.
FIFO-FF	24:44	03:23	27.8	41.4	SJF-FF	25:48	05:19	32.5	54.7
FIFO-C	26:31	04:43	27.5	40.9	SJF-C	27:19	06:14	32.5	55.4
LJF-FF	29:45	07:26	32.7	54.7	EBF-FF	68:27	46:15	26.8	40.2
LJF-C	30:26	08:12	32.9	54.7	EBF-C	71:43	48:31	26.89	40.2

As for the memory usage, thanks to the incremental job loading and job removal capabilities, AccaSim consumes low memory. The average memory usage is around 30 MB with a peak at 55 MB across all the experiments. Such low memory usage makes it possible to execute experiments in parallel. Considering the large size of the workload, these numbers are very reasonable, supporting the claim that AccaSim is scalable.

Evaluation of the dispatching methods. The dispatching methods can be evaluated and compared from different perspectives thanks to AccaSim's tools and output data. In Fig. 6, sample snapshots of the *watcher* and the *visualization* tools taken at certain time points during the FIFO-FF experiment are shown. The *watcher* 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 *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 resources. In our case study, the core and memory usage are shown separately.



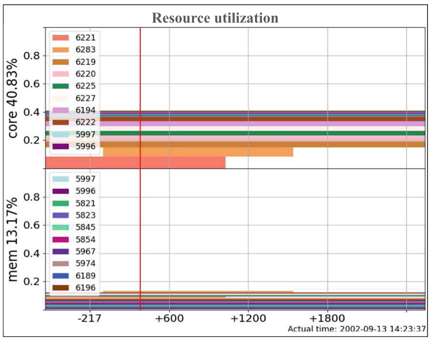
```
> python watch-sim.py -h
Usage: watch-sim.py [-h] [-usage] [-progress] [-all] [-ip
HOSTIP]

Client For HPC Simulator Watcher Daemon

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 watch-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
```

(a) Watcher tool.

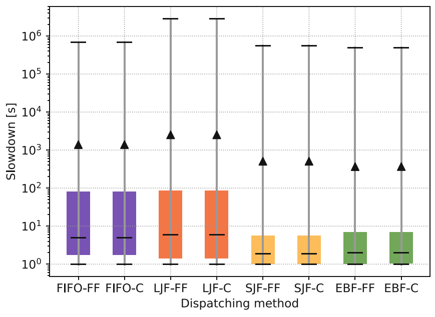


(b) Visualization tool.

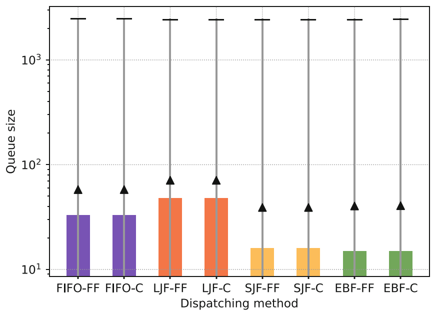
Fig. 6. Following a simulation process.

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 dispatcher methods in terms of their effect on system resource utilization: how many resources are used and to what extend they are consumed. Alternatively, the data can be utilized to compare them in terms of their impact on system throughput, using some metrics like the well-know job slowdown [10]. Slowdown of a job j is a normalized waiting 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. Another useful metric could be the queue size, which is the number of jobs waiting in the queue at a given time point. The lower the slowdown and the queue size are, the higher the throughput is.

In Fig. 7, we show the distributions of the slowdown and the queue size for each of the 8 experiments in box-and-whisker plots. We can see that SJF and EBF based dispatching methods achieve the best results, independently of their allocation methods 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 methods. SJF maintains overall lower slowdown values than the other methods, but a higher mean than the EBF. SJF maintains also slightly higher mean in the queue size than the EBF. The scheduling policy 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.



(a) Distributions for job slowdown.

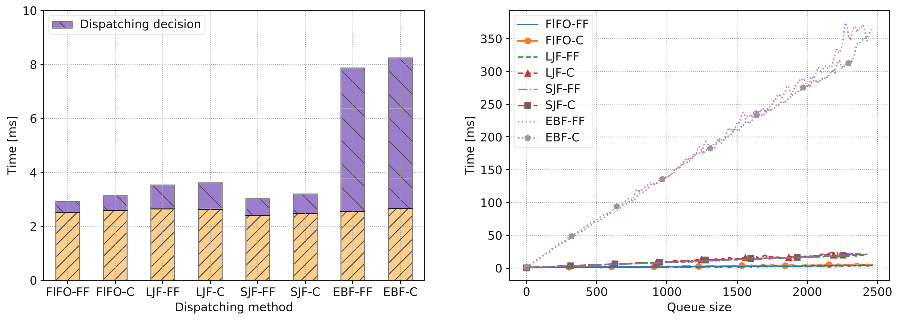


(b) Distributions of queue size.

Fig. 7. QoS of the dispatchers.

The second type of output data regards the simulation process, specifically the CPU time required by the simulation tasks like job loading and generation of the dispatching decision, which gets updated at each simulation time point. This type of data can be used to evaluate the performance of the dispatchers in terms of the time they incur for generating a decision. In Fig. 8a, we report the average CPU time required at a simulation time point for each of the 8 experiments. In accordance with Table 1, the time spent in simulation, other than generating the dispatching decision, is constant (around 2 ms in this case) across all the experiments and the EBF based methods spent much more time in generating a decision than the others. In Fig. 8b, we instead analyze the scalability. Specifically, we report for each queue size the average CPU time spent at a simulation time point in generating a dispatching decision for each of the 8 experiments. While all the dispatchers scale well, the EBF based methods require more CPU time for processing bigger queue sizes, due to their scheduling policy which tries to fit as many jobs as possible into the system.

Our analysis restricted to the considered workload and resource settings reveals that, while the EBF based dispatchers give the best throughput, they are much more costly in generating a dispatching decision. Simple dispatchers based on SJF are valid alternatives with their excellent scalability and high throughput comparable to the EBF based methods. The job durations in the current workload are distributed as 56,63% short (duration under 1 h), 34,66% medium (duration between 1 and 12 h), and 8,71% long (duration over 12 h). The synthetic system has a very homogeneous structure with all the nodes having the



(a) Average CPU time at a simulation time point.

(b) Average CPU time at a simulation time point to generate a dispatching decision w.r.t. queue size.

Fig. 8. Performance of the dispatchers.

same number of the same type of resources. It would be interesting to experiment further with the dispatching methods using workloads with different job duration distributions and more heterogeneous resource structures. Thanks to AccaSim, such experiments can be conducted smoothly.

5 Related Work

HPC systems have been simulated from distinct perspectives, for instance to model their network topologies [1,14,17] or storage systems [18,21]. There also exist HPC 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 most recent WMS simulator is presented in [19]. The Scheduling Simulation Framework (ScSF) emulates a real WMS, Slurm Workload Manager[8] which is popular in many HPC systems. In [16,22] Slurm is modified to provide job submission, resource management and job dispatching through distinct daemons which run in diverse virtual machines and which communicate over RPC calls, and dedicated simulators are implemented. ScSF extends these simulators with automatic generation of synthetic workload descriptions based on statistical data, but does not give the possibility to read real workload descriptions, for instance from a file, for job submission. The dependency on a specific WMS does not render the customization of the WMS, 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.

In [12], an energy aware WMS simulator, called Performance and Energy Aware Scheduling (PEAS) simulator is described. With the main aim being to minimize the power consumption and to increase the throughput of the system,

[8] Slurm Workload Manager: https://slurm.schedmd.com/.

PEAS uses predefined dispatching methods and workload description file format, and the system power calculations are based on a 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 descriptions for job submission, as apposed to having to generate them automatically as in ScSF, all jobs are loaded at the beginning which can hinder the performance when experimenting with a large number of jobs. Moreover, the implementation is not available which prevents any form of customization.

In [13], a WMS simulator based on a discrete event library called Omnet++[10] is introduced. Similar to ScSF, only automatically generated synthetic workload 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 extend 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, performance degrade with large workloads, no customization of the WMS, and unavailable or undocumented implementation.

6 Conclusions

In this paper, we presented AccaSim, a library for simulating WMS in HPC systems, which offers to the researchers an accessible tool to aid them in their HPC dispatching 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, making it easy to use. AccaSim is scalable and is highly customizable, allowing to carry out large experiments across different workload sources, resource settings, and dispatching methods. Moreover, AccaSim enables users to design novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as power consumption and failures of the resources.

In future work, we plan to do experimental comparison to other simulators and assess further the scalability of AccaSim in terms of distinct system configurations. Besides, in order to aid the users further in evaluating dispatchers, we are currently working on showing more information in the tools regarding system utilization, such as the amount of allocation of each resource individually, and on automatically generating the performance and QoS plots used in this paper.

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

In addition, we plan to externalize the visualization tool by executing it in an independent application to reduce the simulation resource usage.

Acknowledgments. C. Galleguillos is supported by Postgraduate Grant PUCV 2017. We thank Alina Sîrbu for fruitful discussions on the work presented here.

References

1. Acun, B., Jain, N., Bhatele, A., Mubarak, M., Carothers, C.D., Kalé, L.V.: Preliminary evaluation of a parallel trace replay tool for HPC network simulations. In: Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M.E., Scarano, V., Varbanescu, A.L., Scott, S.L., Lankes, S., Weidendorfer, J., Alexander, M. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 417–429. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_34
2. Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., Wilde, T.: A case study of energy aware scheduling on SuperMUC. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 394–409. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_25
3. Banerjee, A., Mukherjee, T., Varsamopoulos, G., Gupta, S.K.: Integrating cooling awareness with thermal aware workload placement for HPC data centers. Sustain. Comput. Inf. Syst. **1**(2), 134–150 (2011)
4. Blazewicz, J., Lenstra, J.K., Kan, A.H.G.R.: Scheduling subject to resource constraints: classification and complexity. Discrete Appl. Math. **5**(1), 11–24 (1983)
5. Bodas, D., Song, J., Rajappa, M., Hoffman, A.: Simple power-aware scheduler to limit power consumption by HPC system within a budget. In: Proceedings of E2SC@SC, pp. 21–30. IEEE (2014)
6. Borghesi, A., Collina, F., Lombardi, M., Milano, M., Benini, L.: Power capping in high performance computing systems. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 524–540. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_37
7. Brandt, J.M., Debusschere, B.J., Gentile, A.C., Mayo, J., Pébay, P.P., Thompson, D.C., Wong, M.: Using probabilistic characterization to reduce runtime faults in HPC systems. In: Proceedings of CCGRID, pp. 759–764. IEEE CS (2008)
8. Brennan, J., Kureshi, I., Holmes, V.: CDES: an approach to HPC workload modelling. In: Proceedings of DS-RT, pp. 47–54. IEEE CS (2014)
9. Bridi, T., Bartolini, A., Lombardi, M., Milano, M., Benini, L.: A constraint programming scheduler for heterogeneous high-performance computing machines. IEEE Trans. Parallel Distrib. Syst. **27**(10), 2781–2794 (2016)
10. Feitelson, D.G.: Metrics for parallel job scheduling and their convergence. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 2001. LNCS, vol. 2221, pp. 188–205. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45540-X_11
11. Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with using the parallel workloads archive. J. Parallel Distrib. Comput. **74**(10), 2967–2982 (2014)
12. Gómez-Martín, C., Vega-Rodríguez, M.A., Sánchez, J.L.G.: Performance and energy aware scheduling simulator for HPC: evaluating different resource selection methods. Concurrency Comput. Pract. Exp. **27**(17), 5436–5459 (2015)
13. Hurst, W.B., Ramaswamy, S., Lenin, R.B., Hoffman, D.: Modeling and simulation of HPC systems through job scheduling analysis. In: Conference on Applied Research in Information Technology. Acxiom Laboratory of Applied Research (2010)

14. Jain, N., Bhatele, A., White, S., Gamblin, T., Kalé, L.V.: Evaluating HPC networks via simulation of parallel workloads. In: Proceedings of SC, pp. 154–165. IEEE CS (2016)
15. Li, Y., Gujrati, P., Lan, Z., Sun, X.: Fault-driven re-scheduling for improving system-level fault resilience. In: Proceedings of ICPP, pp. 39. IEEE CS (2007)
16. Lucero, A.: Simulation of batch scheduling using real production-ready software tools. In: Proceedings of IBERGRID, pp. 345–356, Netbiblo (2011)
17. Mubarak, M., Carothers, C.D., Ross, R.B., Carns, P.H.: Enabling parallel simulation of large-scale HPC network systems. IEEE Trans. Parallel Distrib. Syst. **28**(1), 87–100 (2017)
18. Nuñez, A., Fernández, J., García, J.D., García, F., Carretero, J.: New techniques for simulating high performance MPI applications on large storage networks. J. Supercomput. **51**(1), 40–57 (2010)
19. Rodrigo, G.P., Elmroth, E., Östberg, P.-O., Lavanya, R.: ScSF: a scheduling simulation framework. To appear in the Proceedings of JSSPP. Springer (2017)
20. Skovira, J., Chan, W., Zhou, H., Lifka, D.: The EASY — LoadLeveler API project. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1996. LNCS, vol. 1162, pp. 41–47. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0022286
21. Snyder, S., Carns, P.H., Latham, R., Mubarak, M., Ross, R.B., Carothers, C.D., Behzad, B., Luu, H.V.T., Byna, S., Prabhat, S.: Techniques for modeling large-scale HPC I/O workloads. In: Proceedings of PMBS@SC, pp. 5:1–5:11. ACM (2015)
22. Stephen, T., Benini, M.: Using and modifying the BSC slurm workload simulator, Technical report, Slurm User Group Meeting (2015)
23. Tang, Q., Gupta, S.K.S., Varsamopoulos, G.: 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)
24. Zhou, Z., Lan, Z., Tang, W., Desai, N.: Reducing energy costs for IBM blue gene/P via power-aware job scheduling. In: Desai, N., Cirne, W. (eds.) JSSPP 2013. LNCS, vol. 8429, pp. 96–115. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43779-7_6