Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

SIMULATION OF HPC JOB SCHEDULING AND LARGE-SCALE PARALLEL WORKLOADS

Mohammad Abu Obaida
Jason Liu

School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA

ABSTRACT

The paper presents a simulator designed specifically for evaluating job scheduling algorithms on large-scale HPC systems. The simulator was developed based on the Performance Prediction Toolkit (PPT), which is a parallel discrete-event simulator written in Python for rapid assessment and performance prediction of large-scale scientific applications on supercomputers. The proposed job scheduler simulator incorporates PPT's application models, and when coupled with the sufficiently detailed architecture models, can represent more realistic job runtime behaviors. Consequently, the simulator can evaluate different job scheduling and task mapping algorithms on the specific target HPC platforms more accurately.

1 INTRODUCTION

The size of today's high-performance computing (HPC) systems increases with their complexity. New architectural changes and new software capabilities have in general complicated the design of scientific applications running on these systems in order to achieve their performance objectives. In the mean time, the growth in resource demand (compute nodes, memory, power, etc.) for the new generation of HPC systems needs to be justified by the capabilities of running more and larger applications with improved performance. Many job scheduling and resource management algorithms have been proposed that aim to improve both the resource utilization and the job turnaround time for HPC systems.

Simulation plays an important role in evaluating the performance of these algorithms. Many simulators have been developed to study HPC job scheduling and resource management. Simple job scheduler simulators often provide a detail model of the queueing behavior of the jobs as they arrive at the system upon submission, wait for available resources as they compete with other jobs in the system, deploy for execution when scheduled, and eventually depart from the system upon job completion. For example, PYSS is a trace-driven scheduler simulator developed in Python (Parallel Systems Lab 2017); the simulator implements quite some scheduling algorithms, including several backfilling algorithms. The problem with simple simulators is that they do not really model the target HPC system or the runtime behavior of the applications. PYSS takes the job runtime directly from the job trace, although in reality a job's runtime should be affected by the specific resources allocated to the job and by the application's runtime behavior, which can be affected other simultaneously running jobs.

More sophisticated job scheduler simulators exist. For example, GridSim (Buyya and Murshed 2002) is a simulator that can model a wide range of computing resources, including multiprocessors, distributed-memory machines, and networks of different configurations. GridSim provides support for simulation of job scheduling on different types of systems, including clusters, Grids, and P2P networks. Alea-2 (Klusáček and Rudová 2010), for example, extends GridSim with several popular job scheduling algorithms. GridSim provides a preliminary method to model applications. An application can be described as a number of tasks, as in the task/channel model or as a direct acyclic graph (DAG). Each task corresponds with a "gridlet", which contains the information pertaining to its execution: the length (in the number of instructions), disk I/O operations, and the size of the input and output data. These parameters can help determine the job's execution time as the tasks are run on the given resources. Although the gridlet approach does provide a rough estimate of the application's runtime (such as the differences between CPU-intensive and I/O-intensive applications), and thus can be used to simulate resource contentions on target HPC systems for job scheduling and task mapping algorithms, the task-level description may not be sufficient to capture detailed application behavior, such as the cost of specific numeric kernels, nested loop structures in the program, or the communication overhead, for example, resulted from particular use of MPI functions.

The most ambitious approach is to add job scheduling capabilities to detailed HPC simulators. TraceR (Acun et al. 2015) is a trace replay and parallel workload modeling tool built on CODES (Cope et al. 2011). CODES provides several detailed interconnection network models. TraceR can take a large execution trace generated by the BigSim's emulation framework (Zheng et al. 2005) and run it on the simulated interconnection networks in CODES to assess the application's communication behavior. Unfortunately, TraceR and CODES do not support full-fledged job scheduling.

The Structured Simulation Toolkit, or SST (Rodrigues et al. 2011), has added a scheduling component to simulate job scheduling and placement on target HPC platforms (Rodrigues et al. 2012). The simulator can also replay job traces from the parallel workload archives. However, the scheduling component has not been validated in studies; and we found that it only uses the fixed runtime from the job trace. That is, the scheduling module has not been incorporated with the SST's rich architecture and application models. SST/macro (Janssen et al. 2012) is a separate effort of SST, which focuses on coarse-grained modeling of HPC applications; however, it cannot dynamically launch and terminate applications, which would make it difficult to add a job scheduler module.

In this paper, we present our job scheduler simulator, which has been seamlessly incorporated with our full-scale HPC system simulator, called the Performance Prediction Toolkit (PPT). PPT supports rapid assessment and performance prediction of large-scale scientific applications on HPC architectures (Ahmed et al. 2016). We extended the simulator with various job scheduling and task mapping algorithms, and with trace-driven simulation capabilities using large-scale job traces from real parallel workload archives. PPT is a full-scale HPC simulator, which provides models for various applications (e.g., computational physics code), middleware (MPI and threads), and hardware (compute nodes, interconnection networks) of various HPC architectures. In particular, the simulator can capture a job's high-level runtime behavior using stylized pseudo-code (with preserved control branches, loops, and communication routines). Consequently, our job scheduler simulator can be used to study the performance of various job scheduling and resource provisioning algorithms with sufficiently detailed models for resource contention, job interaction and interference on specific target HPC systems.

2 SIMULATOR DESIGN

The overall design of the proposed simulator for HPC job scheduling and parallel workloads is illustrated in Fig. 1. Upon arrival, all jobs are first placed into the waiting job queues. There could be more than one such queues, depending on the scheduling algorithm which we choose to study. The job scheduler selects a job from one of the waiting queues, one job at a time, according to the specified scheduling algorithm, and then allocates the corresponding resources from the target HPC system (i.e., the compute nodes and processors), in accordance with the specific job placement policy (also called tasking mapping). The job is then executed at the target HPC system: the simulator models the job's execution on the designated compute nodes by running the job's application model with the requested number of simulated MPI processes. The job's application describes the job's execution behavior, which can be modeled at various level of details; for example, a scientific application can describe the runtime for each iteration through its computation kernel, as well as the specific functions used for communicating with other MPI processes. Once the job has completed execution, it is placed among all finished jobs, for statistics collection and inspection.

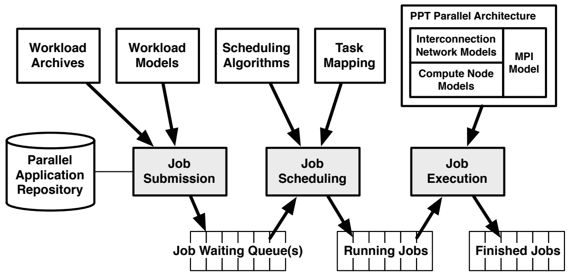

Figure 1: The simulator design.

Table 1: Specification of an HPC Job.

job attribute	description
`job_id`	unique identifier of the job
`job_size`	requested number of processes for the job
`submit_time`	arrival time of the job
`request_time`	requested time for running the job; longer jobs may be terminated prematurely
`app_name`	name of the application associated with the job
`app_args`	arguments to start the application with
`run_time`	runtime of the time keeper application
`queue_name`	name of the waiting queue to place the job
`sched_args`	arguments related to job scheduling

Our simulator consists of three modules, which correspond to the three stages of an HPC job: job submission, job scheduling, and job execution. The job submission module describes the arrival process for the HPC jobs. Depending on the purpose of the study, one can either use synthetic workload models or adopt job traces from existing HPC workload archives. Each job presented to the system needs to be associated with an application model, which describes the runtime behavior of the job as it is scheduled to run on the target HPC platform. The application model can be as simple as containing only a single value of time for which the job is expected run on the target machine, or it can be more sophisticated in describing the skeleton code, including the communication logic, of the application. Our simulator has dedicated a repository for the HPC application models, so that in time, the modelers are given the capability to evaluate job scheduling and tasking mapping algorithms using a wide variety of existing models for common parallel applications. We discuss the details on the creation of the job traces in section 2.1 and describe the application models in section 2.3.

The job scheduling module manages job scheduling and task mapping. It keeps track of both the machines and processors occupied by the running jobs and those available for new jobs. When a new job is submitted or when the resources are returned to the system due to the completion of a running job, the job scheduler is invoked. The job scheduling algorithm selects the best candidate among all jobs waiting in the queues and, if enough resource is available, removes the candidate job from the queue and allocates the requested number of machines and processors to run the job. To to that, the selected task mapping algorithm determines the best placement for the job before launching the job's application. We discuss the details of the job scheduling module in section 2.2.

Finally, the job execution module is called for simulating the execution of the jobs on the target machine. For this, we extend the Performance Prediction Toolkit (PPT), which is designed to model large-scale scientific applications on current and future HPC architectures (Ahmed, Liu, Eidenbenz, and Zerr 2016). PPT is a Python library of models for applications (e.g., computational physics code), middleware (MPI and threads), and hardware (compute nodes, interconnection networks). The jobs are associated with the applications models, which describe the applications runtime behavior (e.g., branches and loops, communications, and I/O) as stylized pseudo-code. PPT is a full-scale system simulator, which allows us to study the interaction and interference of the running jobs on the target HPC architecture. The details on the job execution module are described in section 2.3.

2.1 Job Submission

All job submitted to the system must be described properly. The required attributes of a job are shown in Table 1. Each submitted job is given a unique identifier (`job_id`) in the system. Each job must also specify the number of processes (i.e., MPI ranks) to run the job (`job_size`). In the current implementation, the processes will be packed onto the compute nodes consecutively, one for each core, although alternative methods for more sophisticated mapping can be added later. Each time a job is scheduled to run, a unique set of unoccupied compute nodes will be allocated, which is determined by the chosen task mapping algorithm. We discuss job placement and task mapping in the next section.

The required job attributes also include the job's submission time (`submit_time`), which is the time at which the job enters the waiting queue, and the requested runtime (`request_time`), which is the maximum time allowed for the job to run on the designated nodes. A job (with all the processes) will be be terminated if the job runs longer than the specified time. Each submitted job also needs to specify the name of the job waiting queue (`queue_name`). The waiting queues depend on the scheduling algorithms (also known as scheduling policies), which may require additional arguments (`sched_args`). For example, the score-based priority scheduling algorithm would require an integer value for the initial priority of the job.

The exact execution time of a job depends on the job's application. Each job must be associated with an application. In the simplest form, by default, the application is just a time keeper (if `app_name` is not specified). When the time keeper application is launched on the designated nodes, each process will simply wait for the given simulation time (`run_time`) before the job completes. To create a more realistic runtime scenario, one can use detailed applications models to capture their runtime behavior on the target platform. The application models can be included in the application repository retrievable by name (`app_name`). An application may required additional parameters (such as the size of the matrices for multiplication); they are provided through (`app_args`). We discuss the application models in section 2.3

The job trace needed to drive the simulator can be created with a synthetic workload model, for example, using an exponential distribution for the inter-arrival time of the jobs, a normal distribution for the job size, and yet another for the job's runtime. For a more realistic setting, however, one can derive a job trace from existing workload archives. For example, the Parallel Workloads Archive (PWA), by Feitelson et al. (2014), contains a fairly large number of job-level logs collected for nearly two decades from 38 production systems. They include clusters, grids, and large-scale supercomputers, with job sizes ranging from 64 to over 163,000 processes. PWA data has been widely used in many studies.

The job-level workload logs can be parsed and translated into the trace logs required by our simulator. There are multiple ways to use the workload archive:

1. One can directly use the runtime information from the logs for the time keeper application described above. In this case, we use fixed runtime from the logs and we do not model the detailed application runtime behavior.
2. One can use workload models derived from analyzing the logs in the parallel workload archive, for example, by estimating the distribution of job inter-arrival time, job runtime, job size, and so on. Examples of existing workload models can be found at: http://www.cs.huji.ac.il/labs/parallel/workload/models.html. In this case, we still use the time keeper application with the estimated runtime as the argument. Again, this method does not include detailed application runtime behavior.
3. One can categorize and assign applications to jobs that appear in the parallel workload logs. The accuracy of this method would certainly depend on whether the applications can be represented to some extent using existing models from our application model repository. One possibility is to use the runtime models derived from the real workload data as described in the previous method, and combine with high-level traffic models (such as Roth, Meredith, and Vetter 2015) to capture the statistical distribution of job workload. We have not fully investigated this approach and will do so in the future.

Being able to capture the application runtime behavior will allow us to study the interaction or interference of jobs on the target machine architecture. This can be achieved using either synthetic or real workload data combined with detailed application models. For example, one can study the performance of a target application for different runtime conditions of the target parallel system. To do that, the target application (which we call the *foreground* application) must be mixed with other applications (which we call the *background* applications). One possible method is to randomly associate the target application and a mix of background applications with the jobs in the job trace. We conducted an experiment, which is described in section 3.4, to show the feasibility of the approach.

2.2 Job Scheduling

In this section, we describe the module that selects jobs in the waiting queues and allocates resources to execute the jobs' applications. The module contains two components: the job scheduling algorithms, which select the jobs among the waiting candidates, and the task mapping algorithms, which determines the set of compute nodes to run the selected job.

2.2.1 Job Scheduling Algorithms

The design of the job scheduling algorithms is driven by specific performance and optimization goals. Since there are many job scheduling algorithms, it is infeasible to design a simulator to include them all. A more attractive feature would be to have the simulator support most common scheduling algorithms where new ones, possibly more sophisticated, can be developed by extending from the basic algorithms. The same can be said about the job placement algorithms, which we discuss in the next section. In our simulator, the existing job scheduling and task mapping modules can be replaced relatively easily, so that one can evaluate and compare the performance of different algorithms, using either synthetic and real parallel workload on different target HPC architectures.

In the simulator, the scheduling algorithms are run in a separate simulation process. When a job is submitted, it enters one of the job queues and the scheduling process is notified so that it can select an eligible job to run next. Similarly, when a job has completed its execution, the system reclaims the resources and the scheduling process is notified for it to choose the next eligible job. A job is eligible to run if there are sufficient resources (in the number of unoccupied compute nodes) to support the job's execution. If there are more than one eligible jobs, they will be selected one after another in the order stipulated by the specific scheduling policy. Our simulator currently supports the following scheduling algorithms:

- *First Come First Serve (FCFS):* The waiting jobs are stored in FIFO (first-in-first-out) order; the algorithm will not consider jobs other than the job at the head of the queue, even if there may be sufficient resources to run the other jobs.
- *First Come First Serve with Best Fit (FCFS/BF):* The jobs are processed first come first serve in general, but it is non-blocking. If the system does not have sufficient resources to run the oldest waiting job, the algorithm will select the next eligible job, This algorithm is the default scheduling algorithm used in the simulator.
- *Shortest Job First (SJF):* The waiting jobs are sorted according to their node-hours, which is the product of the requested time (in hours) and the job size (in the number of requested compute nodes). The algorithm selects the eligible job with the lowest node-hours. If multiple jobs have the same lowest node-hours, the algorithm uses FCFS for tie-breaking.
- *Longest Job First (LJF):* The algorithm does the opposite of SJF. The algorithm selects the eligible job with the highest node-hours. If multiple jobs have the same highest node-hours, the algorithm selects the oldest waiting job.
- *Score-Based Priority (SBP):* Each job is initially assigned with a score, and the score may change over time. The algorithm sorts the jobs according to their scores; a job with a higher score has a higher priority to be selected for execution. The scores may change algorithmically in order to achieve certain resource utilization or performance goals. For example, one can use SBP to achieve fair share scheduling, where we prioritize jobs based on the user's current "share": a job's score can be adjusted based on the total number of compute nodes requested by the user, the wait time of all user's jobs, the number of running jobs, the fraction of completed jobs, and so on, in the recent history.
- *Multi-Queue Priority (MPP):* There are multiple waiting queues with distinct priorities. Jobs in a higher priority queue are scheduled more preferably. The user can specify which queue to submit jobs, as long as the jobs satisfy the specific conditions imposed for the queues. For example, Mira is a 10-petaflops IBM Blue Gene/Q supercomputer at the Argonne Leadership Computing

Facility. Mira has several job queues, e.g., `prod-capability`, `prod-long`, `prod-short`, and `backfill`, for high capacity, long running, and short running, and low priority jobs, respectively. The `prod-capability` queue, for example, stores only jobs whose request size must make a significant portion of the system (say, 10% or 20% of all compute nodes).

One of the common scheduling algorithms we have not included in our simulator is backfilling. Backfilling is a technique of opportunistically running low-priority jobs when there are insufficient resources to run high-priority jobs. If the remaining runtime of currently running jobs can be estimated with good accuracy, a time window could be found for low-priority jobs on the available resources, such that it will not negatively affect the launch of next high-priority job. Backfilling has been applied successfully to improve system utilization and job turnaround time. An important aspect of backfilling is the ability to accurately estimate the job's runtime. EASY (Kifka 1995) uses user provided runtime estimation. EASY++ (Tsafrir et al. 2007) also considers historic data for runtime prediction. Recently, machine learning techniques have also been proposed for better predicting runtime (Gaussier et al. 2015). We plan to add these backfilling algorithms in future work.

Once the scheduling algorithm finds the best candidate job, the scheduling process invokes the job placement algorithm to determine the specific compute nodes to run the job.

2.2.2 Task Mapping Algorithms

Job placement (also known as task mapping) algorithms play a crucial role in determining the performance of applications. Good task mapping algorithms can assign compute nodes to jobs so as to minimize the communication overhead of the processes and reduce cross-job interference on the target machine. For this reason, task mapping can be of great importance in improving both job turnaround time and throughput of the system. Our simulator currently supports the following task mapping algorithms:

- *Random:* The algorithm chooses a random set of unoccupied compute nodes. The size of the set is the number of requested compute nodes, which can be calculated from the requested job size (i.e., the number of processes) divided by the number of cores at each compute node and take the ceiling. (Currently, we assume a homogeneous architecture). For most interconnection networks, such as torus, random task mapping may not be a good choice, since the parallel processes are scattered throughout the entire system. In this case, the random task mapping can be used as a worst-case scenario for applications. In the case of the dragonfly interconnection network, however, the random task mapping, coupled with a proper routing algorithm, can indeed improve the overall communication performance of applications (Jain 2016).
- *Round-Robin:* The system starts with an ordered list of all compute nodes. A job request is fulfilled by taking out the nodes from the head of the list. When a job terminates, the reclaimed nodes are appended at the tail of the list. Round-robin task mapping can be a good choice for a small set of applications. However, when the system runs for a longer period of time, fragmentation may become significant, the communication distance may increase, and consequently the communication overhead may get worse.
- *Dual-End:* This task mapping algorithm was designed originally to improve communication performance and resource utilization on the torus interconnection network (Zimmer et al. 2016). The system starts with an ordered list of compute nodes, all marked as unoccupied. The jobs are classified as either short jobs or long jobs, using a threshold value. The algorithm places the short jobs and long jobs at the either ends of the ordered list. To place a short job, the algorithm will search for unoccupied nodes from one end of the ordered list; a long job would go from the other end. The selected compute nodes are then marked as occupied to run the job. When the job has completed execution, the nodes are again marked as unoccupied. This task mapping strategy has shown capable of achieving more balanced utilization of the system (Zimmer et al. 2016).

There are several methods for deriving the ordered list used by either the round-robin or the dual-end task mapping algorithms. A straightforward method is *topological ordering*. For example, in a 3D torus network, we can use the XYZ coordinates of the corresponding switch to sort the nodes: we order the compute nodes attached to the same switch, and we order the switches, by going in the X dimension first, and then Y and then Z; that is, in the XYZ ordering. Specifically for the torus network, it is possible that the dimensions have different bandwidth and delay. For example, Cray's Gemini interconnect, used in the Titan supercomputer at ORNL, has double the number of connections in the X and Z dimensions than in the Y dimension. In this case, one would prefer using XZY rather than XYZ topological ordering, so that adjacent nodes in the ordered list would have better connectivity. There are similar ways to derive topological ordering of the switches in a hierarchical network, such as the fat-tree or dragonfly network, using the switch level in the network and the switch index within the level.

Another ordering method for the compute nodes is *Cartesian splitting*. This particularly applies for torus networks. For example, one can split a $8 \times 6 \times 8$ torus into 32 equal sized boxes with $2 \times 3 \times 2$ in dimension. It will allow neighbor nodes to be grouped together for better communication performance.

The simulator also supports *virtual partitioning* in addition to the two ordering methods above. Virtual partitioning divides the ordered list into several blocks of consecutive nodes. A job is evenly straddled across all partitions. Suppose p is the number of partitions and n is the number of nodes requested by the job (job size). Assuming n is divisible by p, the tasking mapping algorithm would allocate n/p nodes from each partition.

2.3 Job Execution

Once the job scheduler selects the next job to run, it will launch the job's application on the designated compute nodes. To make accurate performance prediction of a parallel application, it is necessary that the simulator provide sufficiently accurate computation and communication models for potentially large number of jobs with large job size. We chose to use the Performance Prediction Toolkit (PPT), which has been designed for rapid assessment and performance prediction of large-scale scientific applications on high-performance computing platforms (Ahmed et al. 2016).

Our decision is based on two main concerns. First, PPT uses interpreted languages (Python, LUA, or Javascript). We chose Python, which would allow us to easily incorporate different scheduling and task mapping algorithms. PPT is developed based on Simian, which is a process-oriented, parallel discrete-event simulator (Santhi et al. 2015). The process-oriented world view is implemented using lightweight threads (greenlets in Python), which makes it easy to implement the scheduling algorithms and simulating the MPI processes for the applications. Simian also supports parallel simulation, and has shown capable of running large-scale models on parallel platforms.

Second, PPT already has a library of models of hardware, middleware, and applications. The hardware includes high-level models of different machine architectures and interconnection networks. Although it does not yield cycle-accurate performance metrics, the hardware models is able to capture high-level application runtime behavior with decent accuracy. The hardware node model uses information, such as the number of (integer and floating-point) instructions, memory operations, and so on, to predict the execution time of a specific code segment on the target machine architecture. The interconnect model includes most interconnection networks, such as torus, fat-tree, and dragonfly, and its accuracy has been validated when compared with empirical studies on production networks.

PPT middleware consists of MPI and multithread models. In particular, the MPI model include all MPI functions that are commonly used in parallel applications (Ahmed et al. 2016), including point-to-point functions (with both blocking and immediate send and receive versions), collective operations (reduce, broadcast, gather, scatter, all-to-all, and their variants), and those related to process groups and communicators (creating and splitting groups, and Cartesian communicators). A full-fledged MPI model allows one to write application models as a stylized version of the actual application: the skeleton code

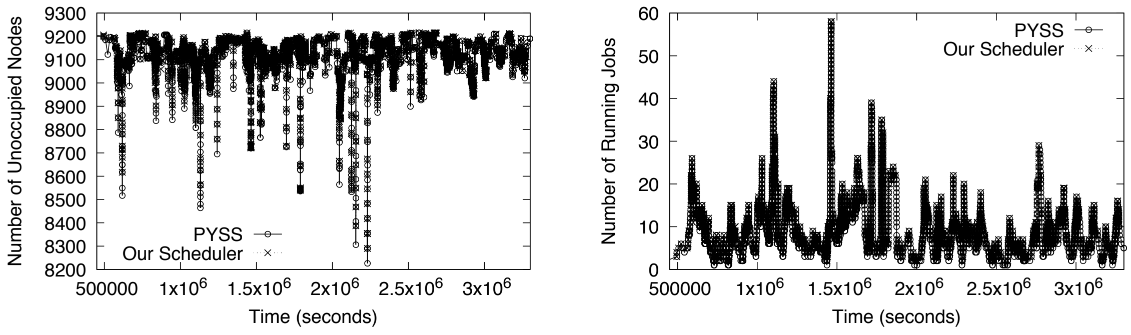

Figure 2: Comparing between PYSS and our simulator.

can maintain the loop structures and conditional branches pertaining to important numerical calculations, while preserving the detailed communication operations.

PPT already has a rich set of application models, mostly computational physics code, including both benchmark applications and production applications, in fluid dynamics, radiative transport, molecular dynamics, Monte Carlo methods, and so on. We only began to incorporate these application models into our application model repository. We have also started to develop applications that can generate basic traffic patterns, such as allreduce and other collective operations, random and neighbor communications (such as 2D and 3D stencils), and so on. We also added basic numerical applications, such as matrix multiplication, and plan to include other linear algebra solvers. The application model repository currently is quite elementary, but we expect it to grow as we plan to focus more on modeling application behavior in next phase when we study the performance of different scheduling algorithms.

3 EXPERIMENTS

We conducted some preliminary experiments to validate the basic functions of our simulator and to evaluate the performance of job scheduling and task mapping algorithms.

3.1 Simple Validation

Our first experiment is designed to validate the basic functions of the job scheduler. To do that, we compare the results against an existing well-established simulator running with real workload job traces. In particular, we compare with the results from PYSS (Python Scheduler Simulator), which provides support for many job scheduling algorithms (Parallel Systems Lab 2017). We use the SDSC SP2 log from the Parallel Workloads Archive (Dror Feitelson 2017). The job trace was collected on a 128-node IBM SP2 system between May 1998 and April 2000, and contains 73,496 jobs with information about the user, application, and job size, as well as the submit, wait, and run times. Note that PYSS does not have application models to capture the job's runtime (including computation and communication) behavior. For this experiment, we simply use the job run time provided in the job trace. We target an HPC system with 9,216 compute nodes connected via a 3D torus interconnection network. Since we do not need to model the application behavior (we use the default time keeper application), the particular choice of interconnection network does not affect the results.

The results are shown in Fig. 2. In particular, the plot on the left of the figure shows the system's resources in terms of the number of unoccupied nodes in the system change over time as jobs are scheduled to run. The plot on the right shows the total number of running jobs over time. For illustration, we only show the results during an arbitrary time period. Our simulator was able to produce similar results as those from PYSS. The figure shows the only results from using the FCFS scheduling algorithm, although other scheduling algorithms can provide the same conclusion.

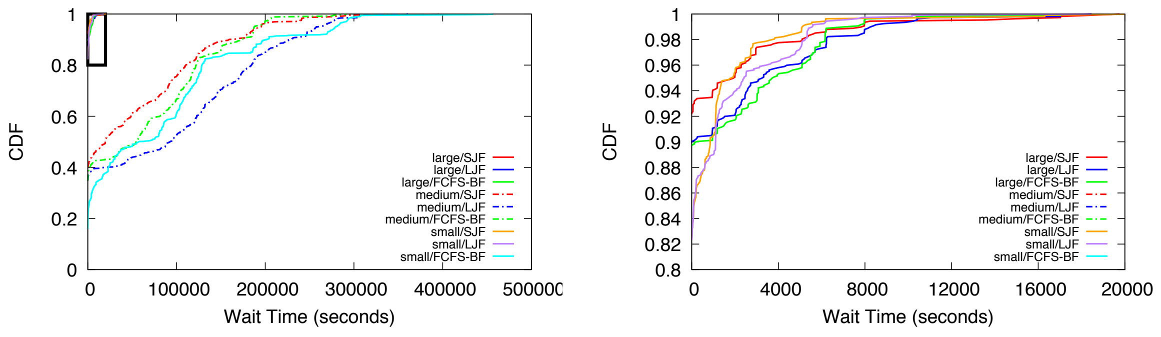

Figure 3: Comparing scheduling algorithms on systems of different size.

3.2 Evaluating Scheduling Algorithms

Our next experiment compares the results from different scheduling algorithms. For this experiment we choose to use the ANL Intrepid log from the Parallel Workloads Archive (Feitelson 2017). The job trace was collected on a Blue Gene/P system called Intrepid at the Argonne National Laboratory. The system is the largest one in the archive with 40,960 quad-core compute nodes, that is, 163,840 cores in total. The logs contains 68,936 jobs submitted from January 2009 to September 2009, with job size ranging widely from merely several cores up to 163K cores. The purpose of this experiment is to test whether our job scheduler simulator is able to handle real job workload for large-scale HPC systems.

We run the job trace for three different scheduling algorithms (FCFS/BF, SJF, and LJF) on a target system configured to represent three different clusters: a small-size cluster with $36,864$ cores, a medium-size cluster with $73,728$ cores, and a large-size cluster with $147,456$ cores. Oversized jobs in the trace were discarded accordingly. Fig. 3 shows the cumulative distribution function (CDF) of the job wait time of different scheduling algorithms on the three target machines. The plot on the right is zoomed-in top-left corner of the plot on the left.

This experiment shows that our scheduler simulator is able to handle large job traces and can obtain meaningful performance results of different scheduling algorithms using large-scale models of parallel systems. A detailed study to compare various job scheduling algorithms is planned for future work.

3.3 Evaluating Task Mapping

We conducted two experiments to test the task mapping algorithms. We chose the target machine as Cielo, which is a 96-rack Cray XE6 system at the Los Alamos National Laboratory (LANL). The machine has a $16 \times 12 \times 24$ Gemini interconnection network, where two compute nodes are connected with each router and each compute node has 16 cores. The entire cluster has 147,456 cores in total.

We used a simple application that performs an all-reduce operation (of 128-byte data in size) among all MPI processes. We varied the job size for 1K, 2K, 4K, and 8K processes and measured the average hop count for the all-reduce operation for different configurations.

In the first experiment, we test the effect of Cartesian splitting on performance. As described earlier, Cartesian split can be an effective technique to improve application performance on torus interconnection networks. The torus topology is divided into smaller boxes (for 3D torus) and the compute nodes are ordered by the boxes. In the experiment, we tested four Cartesian splitting box sizes: $1 \times 1 \times 1$, $2 \times 1 \times 2$, $2 \times 2 \times 2$, and $4 \times 3 \times 4$. Fig. 4 shows the average number of hops vary for different job sizes and for different Cartesian box dimensions. The runtime varies as well, but not as significantly, and there is no evidence for a strong correlation between the number of hops and the runtime.

In the second experiment, we test the effect of virtual partitioning. Here we vary the number of virtual partitions from 1 to 8, doubling each time. Fig. 5 shows the results in the average number of hops and the

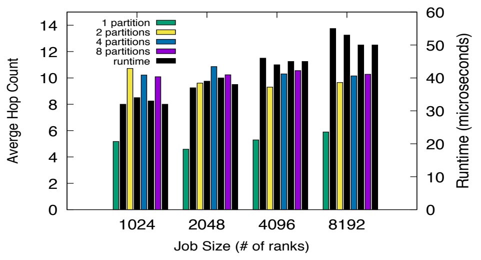

Figure 4: Cartesian Splitting Effect.

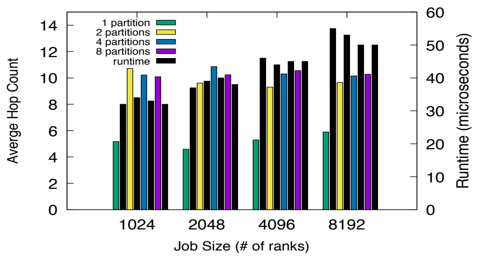

Figure 5: Virtual Partitioning Effect.

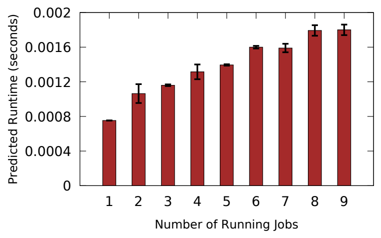

Figure 6: Job Interference.

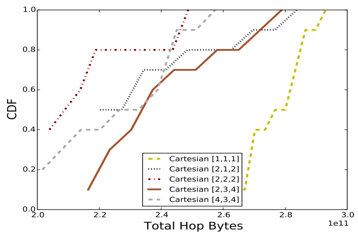

Figure 7: Stencil-3D Performance.

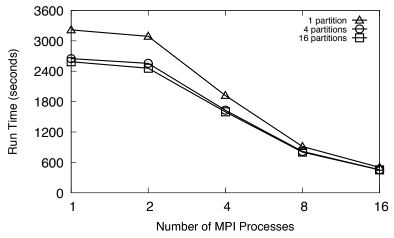

Figure 8: Parallel Performance.

corresponding runtime for different job sizes and for different partitions. In general, the runtime improves with higher partitions.

3.4 Application Performance under Runtime Conditions

Jobs running concurrently share network resources and may interfere with one another if congestion occurs in some part of the interconnection network. We design an experiment to evaluate the performance of a target application under different runtime conditions. The target application is mixed with the other jobs which we call background applications. This is achieved by inserting the job with the target application at random points in the job trace. The target application will eventually be scheduled along with other background applications on the target machine. The job scheduling and resource provisioning algorithms will determine the exact resources used to run the target application and the other applications. Depending on the job placement, it is possible that significant traffic from all applications may cause contentions for the available network bandwidth and therefore impact their performance.

To measure the influence of runtime conditions on application performance we designed an experiment that exposes an application to different runtime conditions with various degree of network congestion. Fig 6 shows predicted runtime for an application performing MPI `even-odd` rank communication among 1024 ranks. The system was configured with a torus interconnect with 9,216 compute nodes. We vary the number of jobs running the same application from 1 to 9. We use random job placement. The result shows that the application experiences 2.5x slowdown under heavily congested network conditions, which underlines the importance of system runtime conditions in predicting application performance.

The next experiment was designed to measure the stress on the network resulted from running a job at different system runtime state. For this experiment, we chose a target application running with 2,048 ranks, where each process performs point-to-point communication with its neighbors in a 3D arrangement; the application is called `stencil-3d`. In this experiment, we randomly insert 10 stencil-3d jobs into a set of 300 jobs from the ANL Intrepid workload trace log. The background jobs were chosen randomly between two simple applications: an application performing an all-reduce operation and an application performing a `stencil-2d` operation. We chose to use five different Cartesian splitting strategies for task mapping. We measured the “hop-bytes” of the target application. The hop-bytes of a message is the product of the number of hops taken by the message and the number of bytes carried by the message. The hop-bytes of an application is the sum of hop-bytes of all messages transmitted by the application. Fig. 7

shows the cumulative distribution in total hop bytes among the 10 runs of the target application; they vary significantly. This demonstrates that the state of the system and task mapping can significantly influence the application behavior.

The last experiment tries to assess the parallel performance of our scheduler simulator. As mentioned earlier, the simulator is built using PPT in Python; the simulator supports parallel discrete-event simulation using MPI. In this experiment we measure the simulation runtime of `mpi_allreduce` with 1 KB data for 8K ranks on Cielo (one on each node) with different virtual partitions (1, 4, and 16). We conducted the experiment on a workstation equipped with two 8-core Intel Xeon E5-2450 processors (at 2.1 GHz clock frequency) and 48 GB of shared memory. Fig. 8 shows the result of running the simulation with various number of MPI processes (cores). The result shows a speedup as much as 6.5x on 16 cores. Extended performance studies are warranted and planned for future investigation.

4 CONCLUSION

In this paper we propose a simulator for scheduling parallel workloads and model application placements on large-scale high-performance computing platforms. Our preliminary experiments show that the application performance can vary significantly for various job scheduling and task mapping algorithms, as well as the runtime conditions of the target platform. The proposed simulator is unique in that it incorporates full-scale models for parallel applications and parallel architectures.

In the next step, we plan to carefully study different job scheduling and task mapping solutions, including various backfilling algorithms. Some of the available placement policies such as random and round-robin are somewhat independent of underlying interconnection networks. As dragonfly and fat-tree based interconnection networks become more common in newer system design, we would like to focus more on the newer architectures, design and evaluate various advanced dynamic job scheduling and job placement strategies.

As the HPC community is advancing toward exascale, better solutions are needed to improve both system utilization and application performance. To a large extent, better scheduling and placement algorithms depends on the accurate prediction of application performance and overall system state. Machine learning techniques has been used to predict application behavior and we are investigating various methods to help predict the runtime state of the HPC systems. We plan to extend our job scheduler simulator to incorporate these advanced methods for large-scale performance evaluation studies.

ACKNOWLEDGMENTS

This research is supported in part by the NSF grant CNS-1563883, the DOD grant W911NF-13-1-0157, the DOE LANL LDRD Program, and a USF/FC2 SEED grant.

REFERENCES

Acun, B., N. Jain, A. Bhatele, M. Mubarak, C. D. Carothers, and L. V. Kale. 2015. “Preliminary Evaluation of a Parallel Trace Replay Tool for HPC Network Simulations”. In *European Conference on Parallel Processing*, 417–429.

Ahmed, K., J. Liu, S. Eidenbenz, and J. Zerr. 2016. “Scalable Interconnection Network Models for Rapid Performance Prediction of HPC Applications”. In *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 1069–1078.

Ahmed, K., M. Obaida, J. Liu, S. Eidenbenz, N. Santhi, and G. Chapuis. 2016. “An Integrated Interconnection Network Model for Large-Scale Performance Prediction”. In *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS)*, 177–187.

Buyya, R., and M. Murshed. 2002. “GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing”. *Concurrency and Computation: Practice and Experience* 14 (13-15): 1175–1220.

Cope, J., N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross. 2011. "CODES: Enabling Co-design of Multilayer Exascale Storage Architectures". In *Proceedings of the Workshop on Emerging Supercomputing Technologies*.

Dror Feitelson 2017. "Parallel Workloads Archive". http://www.cs.huji.ac.il/labs/parallel/workload/.

Feitelson, D. G., D. Tsafrir, and D. Krakov. 2014. "Experience with Using the Parallel Workloads Archive". *J. Parallel Distrib. Comput.* 74 (10): 2967–2982.

Gaussier, E., D. Glesser, V. Reis, and D. Trystram. 2015. "Improving Backfilling by Using Machine Learning to Predict Running Times". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 64:1–64:10.

Jain, N. 2016. *Optimization of Communication Intensive Applications on HPC Networks*. Ph. D. thesis, Dept. of Computer Science, University of Illinois.

Janssen, C. L., H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo. 2012. "A Simulator for Large-Scale Parallel Computer Architectures". *Technology Integration Advancements in Distributed Systems and Computing* 179:57–73.

Kifka, D. 1995. "The ANL/IBM SP Scheduling System". In *IPPS Workshop on Job Scheduling Strategies*, 295–303.

Klusáček, D., and H. Rudová. 2010. "Alea 2: Job Scheduling Simulator". In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 61:1–61:10.

Parallel Systems Lab 2017. "PYSS – The Python Scheduler Simulator". https://code.google.com/archive/p/pyss/.

Rodrigues, A., E. Cooper-Balis, K. Bergman, K. Ferreira, D. Bunde, and K. S. Hemmert. 2012. "Improvements to the Structural Simulation Toolkit". In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 190–195.

Rodrigues, A. F., K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. 2011. "The Structural Simulation Toolkit". *SIGMETRICS Perform. Eval. Rev.* 38 (4): 37–42.

Roth, P. C., J. S. Meredith, and J. S. Vetter. 2015. "Automated Characterization of Parallel Application Communication Patterns". In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 73–84.

Santhi, n., S. Eidenbenz, and J. Liu. 2015. "The Simian Concept: Parallel Discrete Event Simulation with Interpreted Languages and Just-In-Time Compilation". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, 3013–3024. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Tsafrir, D., Y. Etsion, and D. G. Feitelson. 2007, June. "Backfilling Using System-Generated Predictions Rather Than User Runtime Estimates". *IEEE Trans. Parallel Distrib. Syst.* 18 (6): 789–803.

Zheng, G., T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. 2005. "Simulation-Based Performance Prediction for Large Parallel Machines". *International Journal of Parallel Programming* 33 (2): 183–207.

Zimmer, C., S. Gupta, S. Atchley, S. S. Vazhkudai, and C. Albing. 2016. "A Multi-faceted Approach to Job Placement for Improved Performance on Extreme-scale Systems". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 87:1–87:11.

AUTHOR BIOGRAPHIES

MOHAMMAD ABU OBAIDA is a Ph. D. Candidate at the School of Computing and Information Sciences, Florida International University. His areas of research interests include HPC performance prediction, runtime systems, real-time simulation and emulation. His email address is mobai001@fiu.edu.

JASON LIU is an Associate Professor at the School of Computing and Information Sciences, Florida International University. He received a Ph.D. degree from Dartmouth College in Computer Science. His research focuses on parallel simulation and high-performance modeling of computer systems and communication networks. His email address is liux@cis.fiu.edu.