An Integrated Interconnection Network Model for Large-Scale Performance Prediction

Kishwar Ahmed
Mohammad Obaida
Jason Liu
Florida International University
{kahme006, mobai001, liux}@fiu.edu

Stephan Eidenbenz
Nandakishore Santhi
Guillaume Chapuis
Los Alamos National Laboratory
{eidenben, nsanthi, gchapuis}@lanl.gov

ABSTRACT

Interconnection network is a critical component of high-performance computing architecture and application co-design. For many scientific applications, the increasing communication complexity poses a serious concern as it may hinder the scaling properties of these applications on novel architectures. It is apparent that a scalable, efficient, and accurate interconnect model would be essential for performance evaluation studies. In this paper, we present an interconnect model for predicting the performance of large-scale applications on high-performance architectures. In particular, we present a sufficiently detailed interconnect model for Cray's Gemini 3-D torus network. The model has been integrated with an implementation of the Message-Passing Interface (MPI) that can mimic most of its functions with packet-level accuracy on the target platform. Extensive experiments show that our integrated model provides good accuracy for predicting the network behavior, while at the same time allowing for good parallel scaling performance.

CCS Concepts

•Networks → Network simulations; •Computing methodologies → Modeling and simulation;

Keywords

High-performance computing; interconnection network; performance prediction; hardware software co-design

1. INTRODUCTION

As we move towards exascale computing, the collapse of hardware scaling laws has led to the emergence of novel hardware architecture designs in high-performance computing (HPC) that include accelerator technologies (such as GPUs), high core-count compute nodes with shared memory, deep instruction pipelines, deep memory hierarchies with aggressive memory prefetching strategies, and sophisticated branch prediction for speculative execution. These new architectural features enable massive *parallelism* and *latency hiding* that in principle allow software and codes to scale to next-generation HPC systems. For example, Intel's Knight's Corner node features 61 cores with shared main memory (albeit at a non-uniform access speed) that enables thread-level parallelism. In contrast, NVIDIA's Tesla GPU accelerators have up to 3,000 CUDA Cores per CPU enabling vector parallelism. Different parallelization strategies were adopted in the these cases. CPU-based nodes use a significant fraction of their chip real estate to implement pipelining logic (to enable instruction-level parallelism) and memory prefetching logic at different cache levels (to enable latency hiding), whereas GPU designs tend to maximize core counts with arithmetic logic units (ALUs) for enabling vector parallelism.

These novel hardware technologies have turned out to be disruptive to existing software portfolios in many industries and government branches because simple re-compilation does not exploit these features very well. This in turn has led to massive code re-factoring in many sectors, including—and perhaps most pronounced—among users of high-performance computational physics code. Fast performance prediction of how well a new computational method or code will run on a novel architecture HPC platform is a key technique to achieve exascale computing because it allows the quick identification of algorithmic ideas that will or will not pair well with novel architecture platforms. Performance prediction on how fast and how energy-efficient a code will run on a platform is at the heart of computational co-design.

Performance prediction of large-scale parallel computers consisting thousands of node and more is a challenging task. In recent years we have witnessed the fast growth in supercomputer design that can perform operations at scale of quadrillions of calculations per second. The tremendous rise in the computational power is in part attributed to the government agencies that have been supporting (and encouraging) the growth of large-scale supercomputing infrastructures. For example, significant investment by the U.S. Department of Energy (DOE) on building state-of-the-art supercomputers through programs (such as FastForward [39], and recently FastForward 2 [38]) support the fact that exascale computing will continue to receive attention in years to come. Consequently, the community faces a significant challenge for complex large-scale scientific and engineering applications to keep up and take full advantage of the fast growth of supercomputing capabilities.

The Performance Prediction Toolkit (PPT) is a DOE co-design project that aims at developing a comprehensive pre-

SIGSIM-PADS '16, May 15-18, 2016, Banff, AB, Canada
 ISBN 978-1-4503-3742-7/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901378.2901396

diction capability for computational physics code, algorithms and methods that perform on novel hardware architectures, thus enabling fast adoption of new code by quickly identifying and ruling out unsuccessful refactoring scheme. PPT models both hardware and software at levels of abstraction that are appropriate to the concrete question at hand, by applying a mix of discrete-event simulation, stochastic and analytical models at various layers on the software and hardware stack.

With changes in HPC system so frequent, it is imperative that performance prediction of future HPC system is properly realized. Of particular importance is the model for the interconnection networks as it is critical to the understanding of the communication cost and thus the performance limitations of large-scale applications on high-performance computing infrastructures. There has been significant research effort on performance prediction and modeling of extreme-scale interconnection network (e.g., [22,23,32,33]). However, few of these research efforts consider the effect of complex, dynamic application behaviors, such as the computational physics code, on the underlying large-scale interconnection network.

The contribution of this paper is three-fold. First, we present PPT's interconnection network model, which includes as an example a sufficiently detailed model of Cray's Gemini 3-D torus network. Second, our interconnection network model has been fully integrated with an implementation of the Message Passing Interface (MPI) model, which mimics all common MPI commands, including various send and receive functions, as well as collective operations. The MPI model can achieve packet-level accuracy at the target platforms. Third, we present extensive validation studies of our MPI and interconnect models, including a trace-based study using data obtained from executing real-life computational physics code on an existing high-performance computing platform.

PPT relies on the Simian [37], a parallel discrete-event simulation engine, and essentially consists of libraries of hardware models, application models, and middleware models. PPT, along with Simian, is designed to be lean, written in Python (or alternatively Lua) with minimal reliance on third-party libraries in an effort to keep the code simple, understandable, and yet offer high performance. In this paper, we report on scaling runs of our interconnect model, which confirm the scalability of the underlying simulation engine, albeit also point to some performance weaknesses in the Python version of Simian, which suggest directions of our further efforts.

The rest of this paper is organized as follows. Section 2 provides the background and related work. Section 3 provides an overview of our design. We provide the details of our model for the Gemini interconnection network in Section 4 and for the message-passing interface in Section 5. We conducted extensive experiments to validate our integrated interconnect model. The experiments are presented in Section 6. In Section 7 we present a trace-based simulation study to demonstrate the capability of our model for incorporating realistic applications. A preliminary study on the parallel performance of the interconnect model is presented in Section 8. Finally, we conclude the paper and outline our future work in Section 9.

2. BACKGROUND AND RELATED WORK

There exist a wide selection of HPC simulators. Some simulators, such as Gem5 [4], COTSon [3], and Simics [24], model the full-system architecture. Some simulators focus only on a specific component of the system, such as Cacti [41] for caches, M5 [5] for networks, and Graphite [25] for multicore design. These simulators are not appropriate for the performance prediction of large-scale HPC applications due to their limitations in scalability and scope.

An important aspect in HPC performance prediction is *scalability*. That is, how large the HPC system can one model? For example, BigSim is an early effort for performance prediction of large-scale parallel machines (in particular, Blue Gene/L machines), based on the actual execution of the real applications [35, 43]. It is implemented using Charm++ and MPI, and applies parallel discrete-event simulation to scale up performance. BigSim adopts an optimistic approach using the inherent determinacy of the target parallel applications to reduce the overhead of the optimistic scheme. Experiments show that BigSim is capable of scaling up to 64K simulated processors. In the similar fashion, $\mu\pi$ is an MPI simulator based on an efficient conservatively-synchronized parallel simulator [32]. Experiments show that the simulator is capable of simulating hundreds of millions of MPI ranks on a Cray XTS with 216K cores. Compared to our integrated interconnect model, however, both BigSim and $\mu\pi$ provide a simpler network model [9, 43].

The Extreme-scale Simulator (xSim) is a performance-prediction toolkit for future HPC architectures [6]. xSim applies parallel discrete-event simulation using lightweight threads to achieve scalability up to millions of application processes [15, 16]. xSim also incorporates different network topologies, including star, ring, tree, mesh, and torus [17]. However, unlike our interconnect model, network congestion is omitted in xSim to gain scalability. As such, their simulator cannot accurately model the blocking behavior of the target interconnection network which may be of importance to the architecture/application co-design.

The Structural Simulation Toolkit (SST) [34] is a comprehensive simulation framework for modeling large-scale HPC systems, including processors, memory, network, and I/O systems. It attempts to achieve scalability using a conservative parallel simulation approach. SST can model hardware components with different granularity and accuracy. SST's network model in particular contains a variety of interconnect topologies: binary tree, fat-tree, hypercube, butterfly, mesh, and so on. The interconnect model, however, does not provide the necessary details for capturing important network behaviors for performance prediction. For example, it does not support network flow control and also the links are assumed to have infinite capacity. Our interconnect model, on the contrary, provides packet-level details that can support realistic network scenarios, such as the transient network congestion occurred during the execution of large complex applications.

Co-Design of Exascale Storage System (CODES) is a joint project between the Argonne National Laboratory and Rensselaer Polytechnic Institute [12]. The simulator is built upon the Rensselaer Optimistic Simulation System (ROSS), which is based on reverse computation [8]. Several detailed interconnect models have been implemented, which include torus [22], dragonfly [26], and fat-tree [23]. For example, in [27], the simulator predicts the performance of the torus

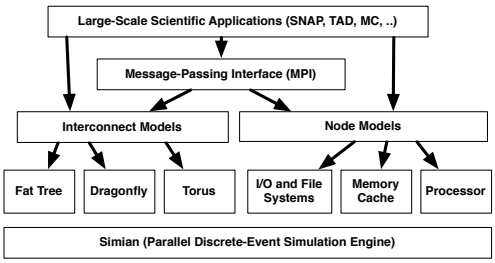

Figure 1: An architectural design of PPT.

network with high-fidelity using synthetic traffic patterns (such as diagonal pairing) on IBM's Blue Gene/P system. In [23], the simulator models large-scale fat-tree networks consisting of millions of compute nodes in a time-efficient manner. A recent paper has proposed a trace-driven simulator (TraceR) to replay large execution traces to predict and understand network performance and behavior [1]. TraceR is built upon ROSS-based CODES simulator and has been shown to be able to simulate a network consisting of half million nodes using traces produced by running BigSim applications.

The CODES project aims at enabling co-design of exascale storage systems. Although complementary to our approach in examining the communication cost of parallel applications (especially computation physics applications), CODES has a slightly different focus on storage systems. Our project aims at providing *fast evaluation* of computational physics algorithms and methods on novel large-scale parallel architectures. We have adopted a minimalistic design to facilitate easy integration of the interconnect model with the target applications (using interpreted languages, like Python and Lua) and provide scalability to accommodate large-scale parallel applications and high-performance architectures.

3. DESIGN OVERVIEW

To design an interconnect model for performance prediction, one need to take several important factors into account:

• *Scale*: The interconnect model must be able to accommodate high-performance computing platforms and applications at extreme scale.

• *Performance:* The interconnect model must run reasonably fast so that it can be used to explore design alternatives of system architectures, software, and parallel applications.

• *Accuracy:* The interconnect model must provide high fidelity sufficient to represent the effect of important design decisions, constraints and optimizations. Simple analytical models may not be sufficient for projecting the performance of dynamic, complex applications.

• *Integration:* The interconnect model must be easy to integrate with other models, including those for processors, memory, and file systems. It is also important that the model can be readily integrated with common software tools, such as MPI, so that various scientific applications can be easily incorporated in the performance study.

Fig. 1 presents an architectural design of our Performance Prediction Toolkit. The majority of target large-scale scientific applications use MPI. Consequently, we designed and implemented an MPI model, which makes it easy for us to incorporate various application models. An instance of the MPI model can be instantiated at the simulated compute nodes, connected via the interconnect model. There are different interconnection network topologies, such as fat-tree, dragonfly, and torus. In this paper, we focus on the specific interconnect model for torus.

All our models are developed based on Simian, which is an open-source, process-oriented parallel discrete-event simulation (PDES) engine [37]. Simian has two independent implementations written in two interpreted languages, Python and Lua, respectively. Simian uses a conservative barrier-based synchronization algorithm [29] for parallel execution.

Simian has several distinct features. First, Simian adopts a minimalistic design. For example, the Python implementation of Simian consists of only around 500 lines of code. As a result, it requires low effort to understand the code and it is thus easy for model development and debugging. Second, Simian features a very simplistic application programming interface (API). To maximize portability, Simian requires minimal dependency on third-party libraries. Third, Simian takes advantage of just-in-time (JIT) compilation for interpreted languages. For certain models, Simian has demonstrated that it can even outperform the C/C++ based simulation engine.

To develop models on Simian, it is necessary to understand the Simian API, which contains only three main modules: the simulation engine, entities, and processes. A *simulation engine* is a logical process responsible for synchronizing with other logical processes. A simulation engine starts with the `run` method, which continuously pops events with the minimum timestamp from the event queue and invokes the corresponding event handler functions. The logical processes are synchronized using a simple window-based conservative synchronization mechanism. In particular, at the start of each window (beginning at simulation time zero), all logical processes find the timestamp of the event at the head of the event queue, add a system-wide minimum delay (lookahead), and then perform a min-reduction to determine the start time of the next synchronization window.

Entities are containers for state (such as a network switch or a compute node). Entities contain event handlers (called services in Simian) that may change the state. An entity can communicate with others by scheduling services at the other entities. Important methods for the entity include: `attachService`, `reqService`, `createProcess`, and `startProcess`. The `attachService` method attaches an event-handler function to the entity, while the `reqService` method schedules an event to be processed at a future simulation time. The methods `createProcess` and `startProcess` creates and starts a process, respectively.

Processes are independent threads of execution. Each process is associated with an entity. Simian uses lightweight threads to implement the processes—greenlets in Python and coroutines in Lua. The user can create a child process using the the `spawn` method and terminates one using the `kill` method. In addition, the user can put a process to sleep for a certain amount of simulation time (using the `sleep` method), suspend a process from execution (using the `hibernate` method), and later resume its execution to continue from the previous suspension (using the `wake` method).

In this paper, we focus on the torus interconnect model. In particular, we describe a model for the Cray's Gemini interconnect that has been commonly used by many supercomputing systems today. We also focus on an MPI model

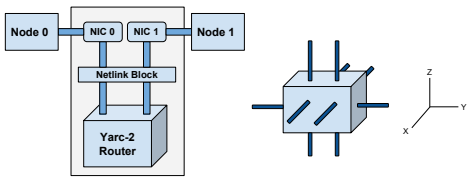

Figure 2: Cray Gemini ASIC block diagram.

that integrates with the Gemini interconnect model, allowing various scientific application models to be readily incorporated for performance evaluation and analysis.

4. GEMINI INTERCONNECT MODEL

We designed and implemented a relatively detailed model for the Gemini interconnection network. Gemini is a part of the Cray's XE6 architecture. Cray XE6 is a system currently used by many large-scale high-performance computing systems, including, for example, Hopper at National Energy Research Scientific Computing Center (NERSC), Cielo at Los Alamos National Laboratory (LANL), Blue Waters at the National Center for Supercomputing (NCSA), Titan at the Oak Ridge National Laboratory, and ISTeC at Colorado State University.

Each Cray XE6 compute node has two AMD Opteron processors, coupled with its own memory (either 32 GB or 64 GB) and communication interface. The Gemini network was first introduced in 2010 in Cray XE6 systems and was the most notable difference from the earlier Cray XT systems. In Gemini, the two AMD Opteron nodes are connected to the Gemini Application-Specific Integrated Circuit (ASIC) through two Network Interface Controllers (NICs). The NICs have their own HyperTransport (HT) 3 link to connect to the nodes, where the link offers up to 8 GB/s bandwidth per node and direction [31]. The NICs within an ASIC are connected through a Netlink block, enabling internal communication between the NICs. At the heart of Gemini is a 48-port YARC router (shown in Fig. 2), which is configured to construct a 3D torus topology. The router is connected to Netlink block through 8 links. Each router gives ten torus connection: two connections per direction in the "X" and "Z" dimension and one connection per direction in the "Y" direction.

Unlike some other interconnection networks, such as fat-tree, torus is a blocking network. It is possible that congestion may happen in the network where queuing delays may negatively affect the performance of parallel applications in a significant fashion. It is thus important to model the traffic behavior in the network imposed by high-level applications. To do that, we need to provide a detailed queuing model to capture the interactions of network transactions.

We implemented each compute node (which is also called a host) or interconnect switch as a Simian entity. Fig. 3 shows a diagram of the design. The hosts and switches are connected via network interfaces that simulate the queuing behavior. A network interface may consists of multiple ports to handle parallel connections between the switches (e.g., in the "X" and "Z" dimensions). Each port consists of an output port ("outport") and an input port ("inport") for sending and receiving messages. To send data from the output port, Simian schedules a service (i.e., an event handler), called

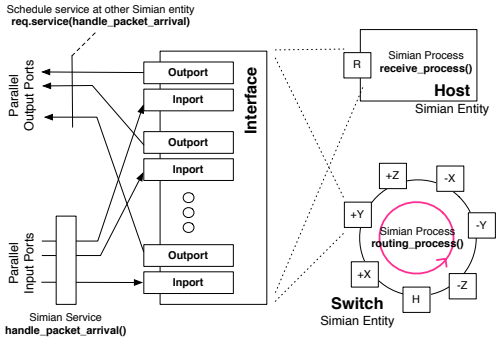

Figure 3: Interconnect model using Simian entities, processes, and services.

`handle_packet_arrival`, at the next node (which can be either a switch or a host), with a delay that is the sum of the current queuing delay at the output port, the packet transmission time, and the link propagation delay between the two nodes. Upon a packet's arrival at a switch, the `handle_packet_arrival` service inserts the packet into the buffer of the corresponding input port and informs the routing process. The routing process is a Simian process that takes packets from the input ports, calculates the next hop using the selected routing algorithm, and then forwards the packet to the corresponding output port. If a host receives the packet, the `handle_packet_arrival` service inserts the packet into the input buffer of the host interface and informs the receive process, which hands the packet to the corresponding MPI receiver accordingly.

Gemini supports multiple routing algorithms, such as deterministic, hashed, and adaptive [40]. Each routing algorithm follows dimension-order routing, where "X" dimension is always traversed first, then "Y" dimension, and finally "Z" dimension [30]. Different routing algorithms provide different level of flexibility in using links at dimensions. For example, the deterministic dimension-order routing provides least amount of flexibility, where links are predetermined in each dimension. The adaptive dimension-order routing provides most flexibility, allowing packets to be adaptively scheduled to lightly-loaded links. In our implementation, the user can explicitly select the routing algorithm when configuring the interconnect model.

5. THE MPI MODEL

The Message Passing Interface (MPI) is one of the most popular parallel programming tools on today's HPC platforms. A good MPI model is essential to studying the design and implementation of scientific applications.

The design of an MPI implementation is intricately influenced by the underlying interconnection network. For example, Cray's MPI implementation for the Gemini interconnect uses Fast Memory Access (FMA). It allows a maximum of 64 bytes of data transfer for each network transaction. A network transaction initiates a single request from the source to the destination, which triggers a response from the destination back to the source. A large message will get broken down into many individual 64-byte transactions. There are two types of transactions. A typical PUT transaction sends 64 bytes of data from a source to a destination. A PUT message consists of a 32-phit request packet (i.e., 96 bytes, where

```
from ppt import *

# config hopper (17x8x24 gemini interconnect)
model_cfg = { # a dictionary
  "intercon_type" : "gemini",
  "host_type" : "mpihost",
  "torus" : configs.hopper_intercon,
  "mpiopt" : configs.gemini_mpiopt,
}
model = HPCSim(model_cfg, ..)

# mpi main function, n is matrix dimension
def cannon(mpi_comm_world, n):
  ... # we describe this later

# start 16 mpi ranks, pass matrix dimension
model.start_mpi(range(16), cannon, 10000)

# simulation starts
model.run()
```

Figure 4: An example showing running 16 MPI processes on Hopper.

each phit is 24 bits). Each PUT message is followed by a 3-phit response packet (9 bytes) from destination to source. A typical GET transaction consists of a 8-phit request packet (24 bytes), followed by a 27-phit response packet (81 bytes), including 64 bytes of data.

To design an MPI model for Gemini, we need to incorporated the FMA request and response scheme at the level of each network transaction. Cray's MPI implementation uses both PUT and GET protocols, the decision of which to use depends on the data size [30]. It was observed that, for data size up to 4K bytes and also beyond 256K bytes, Cray's MPI uses PUT. For data size between 4K and 256K bytes, MPI chooses GET. In our implementation, we only use PUT for simplicity. Since both PUT and GET transactions have a total of 105 bytes of traffic for each request and response pair between the source and the destination, we expect that the effect of selecting between PUT and GET, both in terms of network latency and bandwidth, would be rather insignificant.

In our model, upon receiving a send request of a large MPI message, the MPI sender needs to break down the message into individual PUT requests of at most 64 bytes each. Each message will be sent over the network with an extra 32-byte message overhead. Upon receiving the PUT request, the MPI receiver responses with a 9-byte ACK. We implemented a message retransmission mechanism to ensure reliable data delivery of the MPI messages.

To easily incorporate scientific applications that use MPI, we take advantage of Simian's process oriented design. As we mentioned earlier, each compute node (host) is by itself a Simian entity. Different compute nodes communicate by sending and receiving events (via scheduling services in Simian). We implemented each user MPI process as a Simian process on the compute node. This allows each user MPI process to run independently from other MPI ranks as well as other system-level simulation processes.

Fig. 4 provides an example showing how to start the MPI processes on a simulated HPC cluster. The program starts by calling `HPCSim` to instantiate the model for the entire cluster, including the interconnect model and the compute nodes. Model parameters are passed as an argument in the form of a python dictionary. Most common hardware configurations are preset in PPT for easy reuse and customization, including those parameters that are needed by the MPI implementation for specific interconnection networks.

```
# cannon's algorithm on matrix multiplication
def cannon(mpi_comm_world, n):
  p = mpi_comm_size(mpi_comm_world)
  id = mpi_comm_rank(mpi_comm_world)
  # use p, id to calc i, j, and neighbor ranks

  # time for reading/initing submatrics
  sleep(sometime) # proportional to m^2

  # shift A(i,j) left by i columns
  mpi_sendrecv(left_i, None, m*m*8,
               right_i, mpi_comm_world)
  # shift B(i,j) up by j rows
  mpi_sendrecv(up_j, None, m*m*8,
               down_j, mpi_comm_world)

  for r in range(sqrt(p)-1):
    # time for multiplying A(i,j) and B(i,j)
    sleep(sometime) # proportional to m^3

    # shift A(i,j) to the left
    mpi_sendrecv(left, None, m*m*8,
                 right, mpi_comm_world)
    # shift B(i,j) upward
    mpi_sendrecv(up, None, m*m*8,
                 down, mpi_comm_world)

  mpi_finalize(mpi_comm_world)
```

Figure 5: Simulating Cannon's matrix multiplication.

The `start_mpi` function creates the MPI processes on the designated compute nodes. To allow maximum flexibility, we require the users to specify a mapping from the MPI ranks to the host IDs. The first argument to `start_mpi` is a list. In the example, the simulator creates 16 MPI processes and maps them to 16 compute nodes. On the other hand, if a compute node contains multiple cores (say, 4), one may want to allocate as many MPI ranks to the compute node. This can be easily achieved by specifying a list in python, like: `[i/4 for i in range(n)]`.

Each MPI process is simply a python function that takes at least one argument: `mpi_comm_world`. Like in a real MPI implementation, it is an opaque data structure that represents the set of MPI processes among which communication may take place. Our design, to a large extent, resembles the MPI API. To illustrate its use, we use a simple example of Cannon's matrix multiplication algorithm [7]. The algorithm applies a 2-D block decomposition of the matrices. Suppose the dimension of the matrices is $n \times n$, each processor would be in charge of calculating a sub-matrix of size $m \times m$, where $m = n/\sqrt{p}$, and p is the total number of MPI ranks (assuming it is a square number).

Fig. 5 shows a simulation of the Cannon's algorithm. As we see, the program captures the main execution skeleton of the algorithm. The timing calculation for loading and initializing the sub-matrices and for multiplying the sub-matrices depends on the processor, cache/memory, and file system models that we ignore here. The MPI calls are mapped to the real MPI functions. We implemented most common MPI functions. Table 1 summarizes the main func-

Table 1: Implemented MPI Functions

MPI_Send	blocking send (until message delivered to destination)
MPI_Recv	blocking receive
MPI_Sendrecv	send and receive messages at the same time
MPI_Isend	non-blocking send, return a request handle
MPI_Irecv	non-blocking receive, return a request handle
MPI_Wait	wait until given non-blocking operation has completed
MPI_Waitall	wait for a set of non-blocking operations
MPI_Reduce	reduce values from all processes, root has final result
MPI_Allreduce	reduce values from all, everyone has final result
MPI_Bcast	broadcast a message from root to all processes
MPI_Barrier	block until all processes have called this function
MPI_Gather	gather values form all processes at root
MPI_Allgather	gather values from all processes and give to everyone
MPI_Scatter	send individual messages from root to all processes
MPI_Alltoall	send individual messages from all to all processes
MPI_Alltoallv	same as above, but each can send different amount
MPI_Comm_split	create sub-communicators
MPI_Comm_dup	duplicate an existing communicator
MPI_Comm_free	deallocate a communicator
MPI_Comm_group	return group associated with communicator
MPI_Group_size	return group size
MPI_Group_rank	return process rank in group
MPI_Group_incl	create new group including all listed
MPI_Group_excl	create new group excluding all listed
MPI_Group_free	reclaim the group
MPI_Cart_create	add cartesian coordinates to communicator
MPI_Cart_coords	return cartesian coordinates of given rank
MPI_Cart_rank	return rank of given cartesian coordinates
MPI_Cart_shift	return shifted source and destination ranks

tions included in our MPI model, including blocking and non-blocking point-to-point communications, most collective operations, groups and sub-communicators.

6. VALIDATION EXPERIMENTS

In this section, we describe the experiments for validating our interconnect model. We measure the model-predicted MPI performance on Cray's Gemini network and compare that with published results in the literature to validate our interconnect model.

We consider a large-scale interconnect system in real deployment. Hopper was built by National Energy Research Scientific Computing Center/NERSC (a high-performance computing facility of the U.S. Department of Energy (DOE) [42]). It is a Cray XE6 system that consists of $6,384$ compute nodes connected via the Gemini interconnect[1]. Each compute node contains two 12-core AMD Magny Cours processors running at 2.1 GHz, and DDR3 1.3 GHz RAM (32 GB for each of the $6,000$ nodes and 64 GB for each of the rest 384 nodes). The entire system contains a total of $153,216$ cores, 212 terabytes of memory, and 2 petabytes of disk. The peak floating point operations per node is 201.6 Gflops. The peak performance of the system has been demonstrated to reach 1.3 petaflops [20].

As mentioned earlier, Cray's Gemini interconnect is a 3-D torus interconnect of high performance [2]. Dimensions of Hopper's torus network are $17 \times 8 \times 24$. As outlined in the original design and considered in various literature [19], the peak link speed across the X and Z dimensions is 9.375 GB/sec and in the Y dimension is 4.68 GB/sec. Inter-node latency is measured about 1.27 μs between the nearest nodes and 3.88 μs between the farthest nodes across the system. Although topologically it is a regular 3-D torus, Hopper's interconnect is wired specifically to optimize for the application performance, in which case the hosts are not necessarily named consecutively. To account for that in our model, we provide a mapping from the host IDs to the 3-D torus coordinates of the corresponding interconnect switches [28]. Using this one-to-one mapping, we design the hopper interconnect to closely represent the communication behavior of the applications running on the compute nodes.

The end-to-end latency between two end nodes is determined by the link (propagation) delay and the number of hops between the nodes. For Hopper, the inter-node latency has been reported to be 1.27 μs between the nearest nodes. Consequently, we configure the link delay between the compute nodes and the corresponding switch in our model to be half of that, which is 635 nanoseconds. The inter-node latency for the farthest nodes on Hopper is measured to be 3.88 μs. Since the network diameter for a $17 \times 8 \times 24$ torus is 24, we can subtract two node-switch link delays from the inter-node latency and divide the results by the network diameter. In this way, we obtain the link delay between the adjacent torus switches to be 108.75 nanoseconds. The result per-hop latency seems to be consistent with the empirical measurement reported in the original design paper [2].

[1]Cray XE6 has been used by many of the largest supercomputing systems over the last decade [19].

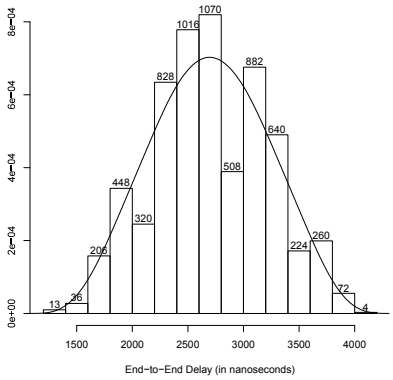

Figure 6: A histogram of end-to-end delay between compute nodes of the simulated HPC cluster.

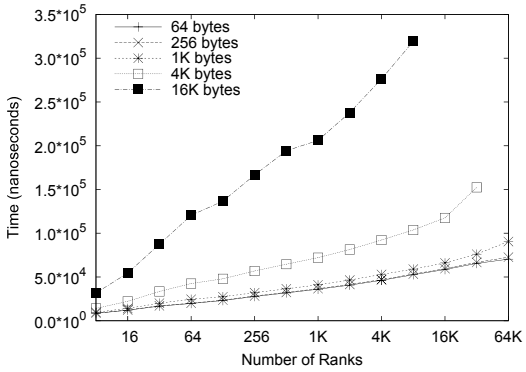

Figure 7: Duration of the `MPI_Allreduce` call for different number of ranks and data size on the simulated HPC cluster.

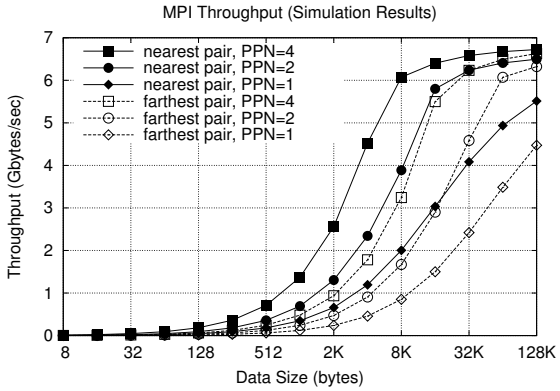

Figure 8: MPI throughput from simulation as a function of message size for 1, 2 and 4 MPI processes per node.

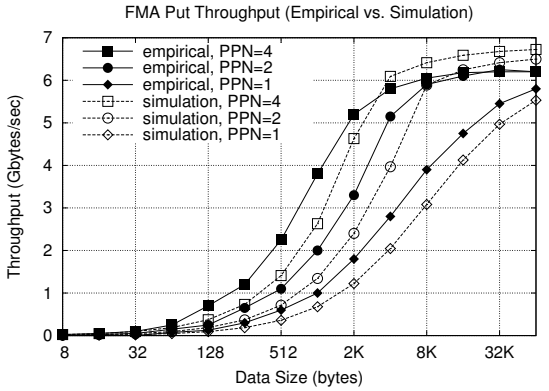

Figure 9: Gemini FMA put throughput (as reported in [2]) versus simulated throughput as a function of transfer size for 1, 2, and 4 processes per node.

We did a latency test by having an MPI process to send a 4-byte data to all other MPI processes mapped on different compute nodes and measure the end-to-end delay. Fig. 6 shows the histogram of the end-to-end delay. The delays are measured between 1.27 μs and 4.07 μs, which are considered within expectation. We also conducted a latency measurement for MPI collective operations. In particular, we measured the duration of a call to `MPI_Allreduce`, as we vary the number of MPI ranks and the data size. Fig. 7 shows the results. As expected, the collective operation has a logarithmic cost in the number of processes under the normal situation. When the number of processes increases along with the data size, part of the network becomes congested and the delay increase superlinearly.

To measure the MPI throughput, we select two compute nodes to run multiple MPI processes; we designate one compute node to run only the MPI senders and the other only the MPI receivers. We vary the number of the sender and receiver pairs (i.e., the number of processes per node, PPN) to be 1, 2 and 4. Each MPI sender sends a series of MPI messages of a given fixed size back-to-back, using `MPI_Send` call, to the designated MPI receiver on the other host. The MPI receiver simply loops and calls the `MPI_Recv`. We run different experiments varying the size of the MPI messages from 8 bytes to 128K bytes doubling each time between the experiments. To get reasonable bounds of the throughput, we select two extremes: one with the two compute nodes next to each other, and the other with the two nodes farthest apart over the interconnect network.

Fig. 8 shows the aggregate throughput of all MPI senders as a function of MPI message size. The performance levels off at 6.75 GB/s when the traffic becomes largely bandwidth constrained. As expected, multiplexing MPI sends at the source host achieves proportionally higher aggregate throughput for small data sizes when the total is less than the bandwidth cap. The throughput between the farthest nodes is lower than that between the nearest nodes due to the increased end-to-end latency.

In Gemini, Fast Memory Access (FMA) is a mechanism for user processes to generate network transactions. In our model, we implemented MPI only as FMA put, where the source can write up to 64 bytes at a time. In Fig. 9, we reproduce the Gemini FMA put throughput (solid lines) as a function of transfer size for 1, 2 and 4 processes per node (as published in [2]). We noticed that the FMA put throughput is significantly higher than what we have achieved using MPI, especially at small transfer sizes, although both level off at above 6 MB/s for large transfer sizes. We speculated that this is due to the MPI overhead. On a quiet network, remote put has an end-to-end latency of less than 700 nanoseconds. But with MPI, the end-to-end latency increases to 3.88 μs between the farthest nodes. To verify that this is indeed the cause of the lowered throughput of our MPI performance, we artificially reconfigured the link delay so that the end-to-end delay for MPI becomes 700 nanoseconds. The results are shown in Fig. 9 (dashed lines), which clearly indicates a much closer match of the simulated results with the empirical measurements.

7. TRACE-DRIVEN MPI SIMULATION

In this section, we present a trace-based simulation study to demonstrate the capability of our interconnect model of incorporating realistic application behaviors, and further validate our model by comparing the communication cost predicted by our model against the actual performance of running the scientific applications on target HPC platforms.

In this study, we use real application communication traces provided by the National Energy Research Scientific Computing Center (NERSC). These traces are used for characterizing the demand of various DOE (US Department of Energy) mini-apps run at various large-scale computing facilities [13]. The traces contain single-node execution profiles of the mini-apps, which include the execution time, the execution speed (the number of instructions per second), the workload (the number of floating-point operations), as well as other cache/memory performance metrics, such as cache miss ratios at different levels. The traces also provide parallel speedup performance and MPI communication operations. The latter is of particular interest in our study.

The DOE mini-apps in the trace collection were run at DOE's three co-design centers, each covering two main applications. ExMatEx (Extreme Materials at Extreme Scale) [14] contains the traces of the Neutron Transport Evaluation and Test Suite (HILO) and the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). CESAR (Center for Exascale Simulation of Advanced Reactors) [10] contains the traces for the MOC emulator and

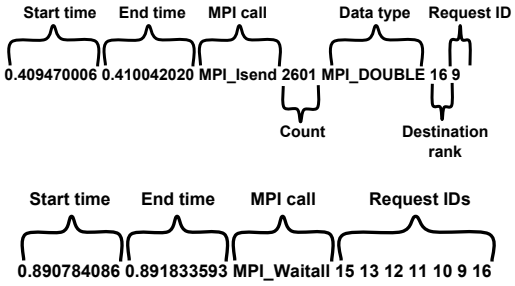

Figure 10: Format of MPI calls in the processed trace file (there is one trace file for each MPI rank).

Nekbone, which solves a poison equation using conjugate gradient iteration with no preconditioner on a block or linear geometry. ExaCT (Exascale Simulation of Combustion in Turbulence) [11] contains traces for a multigrid solver and CNS, a stencil-based algorithm for computing the Compressible Navier-Stokes equations.

The MPI traces was performed on Hopper (described earlier) using the open-source DUMPI toolkit [36] for different number of cores (e.g., 64, 256, and 1024 cores). For each run of the given application, there are a set of trace files, one for each MPI rank. The original trace files are in a binary format. We converted the binary files to text files, using the SST DUMPI toolkit [36] and then processed the files to assemble the necessary information of each MPI call in order, which includes the measured start and end time of the MPI call, and the specific parameters associated with the call, such as the source or destination rank, data size, etc. As an example, Fig. 10 shows two entries of a processed trace file, one for `MPI_Isend` and the other for `MPI_Waitall`. Note that, since the conversion from DUMPI traces to text format is done during pre-processing, this step does not contribute to the simulation runtime. An alternative solution is to parse the binary information directly. We did not choose this option as it depends on the knowledge of DUMPI's internal trace file format.

`MPI_Isend` is a non-blocking send; the function is expected to return immediately with a request handle, which the user can later use to query or wait for the completion of the corresponding non-blocking MPI operation. An entry associated with the `MPI_Isend` call includes the start time and the end time of the MPI call. The count indicates the number of data elements to be sent. Using the count and the data type, one can easily determine the true size of the MPI message. In the example, $2,601$ elements of the `MPI_DOUBLE` type (8 bytes each) would give $20,808$ bytes of data which is scheduled to be transferred for this MPI non-blocking call. The entry also provides the destination MPI rank and an ID to represent the request handle returned by the MPI call. `MPI_Wallall` waits for a list of MPI requests to complete. Accordingly the corresponding entry in the trace provides a list of the request IDs. The MPI function will not return until all corresponding non-blocking operations (which may include both `MPI_Isend` and `MPI_Irecv` calls) are completed.

To run the trace, we start the simulation with the same number of simulated MPI ranks. At each MPI rank, we read the corresponding processed trace file for the rank, one entry at a time. For each entry, we first advance the simulation

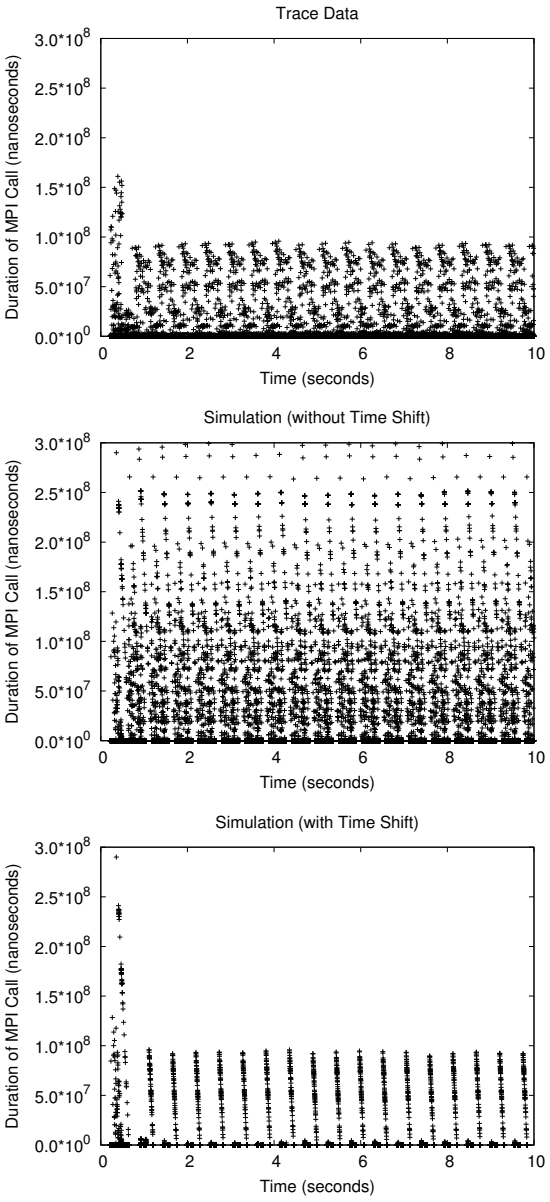

Figure 11: Comparing the duration of MPI calls between trace and simulation with and without time shift.

clock to the exact start time of the MPI call shown in the trace, by having the simulation process to sleep for the exact amount time equal to the difference between the MPI start time and current simulation clock. We then call the same MPI routine in our model and measure the time it takes to complete the MPI call in simulation. We record the time and later compare it against the end time of the MPI call in the trace.

Fig. 11 shows the results of our trace-driven simulation for LULESH from ExMatEx running on 64 MPI processes. Our method can be generally applied to all other traces. LULESH is a mini-app that approximates a typical hydrodynamics model and solves Sedov blast wave problem in 3-D [21]. It is a widely-studied proxy application, which can efficiently run on various platforms and has been ported to a

number of programming models (including MPI, OpenMPI, Chapel, and Charm++) [18]. The particular trace runs for approximately 55 seconds. There are a total of 123,336 calls to `MPI_Isend` and the same number for `MPI_Irecv` and `MPI_Wait`. There are 12,864 calls to `MPI_Waitall`, 6,336 calls to `MPI_Allreduce`, 64 calls each to `MPI_Barrier` and `MPI_Reduce`.

The top plot of Fig. 11 shows the duration of MPI calls observed from the trace (by subtracting the start time from the end time). For easy exposition, we show only the first 10 seconds of the experiment (later time exhibits similar behavior). The middle plot Fig. 11 shows the trace-driven simulation result. At first glance, the simulation shows very similar pattern, yet the duration of the MPI calls spreads as much as three times of the empirical results. A closer inspection shows that the simulation clock sometimes may go beyond the start time of the MPI calls in trace. This is possible since the simulated process may take longer time to complete the previous MPI operation.

To eliminate this bias for comparing the duration of the MPI calls between the simulation and the empirical measurements, we introduce time shift for the trace. When the simulation process detects that its clock goes beyond the time of the trace, we shift the start time of all subsequent MPI calls in the trace by the difference so that the delay of the previous MPI calls in the simulation will not affect the subsequent calculations of the duration of the MPI calls.

The bottom plot of Fig. 11 shows the result of simulation with this time shift. We observe that the duration of the MPI calls becomes much lower. The outstanding spikes (up to around 100 milliseconds) are from `MPI_Waitall`. The staggering pattern seems to be related to the skew in the wall-clock time of the participating compute nodes in the original trace. This would explain the spread of the durations of the MPI calls observed in the original trace (in the top plot).

8. PARALLEL PERFORMANCE

To assess the parallel performance of our integrated model, we conducted a set of experiments on a 1,500-node compute cluster located at Los Alamos National Laboratory. Each compute node in the cluster is equipped with a 12-core Opteron 6176 12C 2.3GHz CPU. The compute nodes are connected by an Infiniband QDR interconnect.

To obtain strong-scaling results, we simulated 156,672 MPI processes running on the Hopper. That is, there is one MPI process running at each core of the target supercomputer platform. For the experiment, the MPI processes perform a collective operation, using `MPI_Allreduce`, with different data size (1K or 4K bytes).

Fig. 12 shows the performance results. We ran the model varying the number of compute nodes, from 1 (12 cores) to 256 nodes (that's 3,072 cores). For data size of 4KB, we ran the model with at least 48 cores to save compute time. The results demonstrate decent parallel performance of the simulator as we see the run time steadily decreases as we increase parallelism. However, the cost of using Simian's Python implementation is also obvious. The aggregate event rate is low, even for 3,072 cores. For this experiment, we did not use Python just-in-time (JIT) compilation, which is expected to significantly improve the performance. We are in the process of translating our model to Lua, for which Simian has demonstrated superior performance. Using JIT

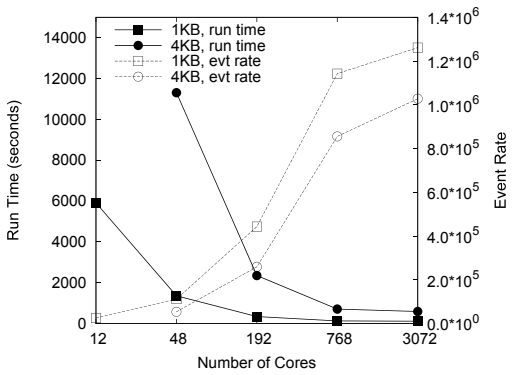

Figure 12: Observed run time and event rate for running Simian with an 156K-rank MPI model on a parallel compute cluster.

and with sufficient event granularity, Simian has been shown to achieve as much as three times the event rate of an optimized C++ parallel simulator [37].

9. CONCLUSIONS

In this paper, we presented an integrated HPC interconnect model for performance prediction. Performance prediction for large-scale scientific applications require an accurate representation of the communication cost between an extremely large number of compute nodes. Our interconnect model is fully integrated with an MPI implementation that includes all common point-to-point communication functions and collective operations with packet-level accuracy. We conducted extensive validation study of our integrated model, including a trace-driven simulation of real-life scientific application communication patterns. Results show that our model provides reasonably good accuracy.

For future work, we plan to include more interconnect topologies and integrate the interconnect model with detailed system models, including processors, cache, memory, and file systems. We plan to incorporate target scientific applications and perform scalability studies using large-scale application communication patterns (e.g., using those included on the DOE website [13]). We are currently in the process of translating our interconnect model to Simian Lua. By then, we will be able to study the parallel performance of our integrated models on large-scale HPC platforms.

Acknowledgments

We gratefully acknowledge the support of the U.S. Department of Energy through the LANL/LDRD Program for this work. We would also like to thank the anonymous reviewers for their constructive comments and suggestions.

10. REFERENCES

[1] B. Acun, N. Jain, A. Bhatele, M. Mubarak, C. D. Carothers, and L. V. Kale. Preliminary evaluation of a parallel trace replay tool for HPC network simulations. In *Euro-Par 2015: Parallel Processing Workshops*, pages 417–429. Springer, 2015.

[2] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 83–87. IEEE, 2010.

[3] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, (4):52–60, 2006.

[6] S. Böhm and C. Engelmann. xSim: The extreme-scale simulator. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 280–286. IEEE, 2011.

[7] L. E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, MT, 1969.

[8] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: A high-performance, low-memory, modular Time Warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[9] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[10] Center for Exascale Simulation of Advanced Reactors (CESAR). https://cesar.mcs.anl.gov/.

[11] Center for Exascale Simulation of Combustion in Turbulence (ExaCT). http://exactcodesign.org/.

[12] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross. CODES: Enabling co-design of multilayer exascale storage architectures. In *Proceedings of the Workshop on Emerging Supercomputing Technologies*, pages 303–312, 2011.

[13] Department of Energy. Design forward characterization of DOE mini-apps. http://portal.nersc.gov/project/CAL/doe-miniapps.htm, Accessed December 1, 2015.

[14] DoE Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). ExMatEx: Extreme Materials at Extreme Scale. http://www.exmatex.org/.

[15] C. Engelmann. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems*, 30:59–65, 2014.

[16] C. Engelmann and F. Lauer. Facilitating co-design for extreme-scale systems through lightweight simulation. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[17] I. S. Jones and C. Engelmann. Simulation of large-scale HPC architectures. In *Parallel Processing Workshops (ICPPW), 2011 40th International Conference on*, pages 447–456. IEEE, 2011.

[18] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 919–932. IEEE, 2013.

[19] D. J. Kerbyson, K. J. Barker, A. Vishnu, and A. Hoisie. A performance comparison of current HPC systems: Blue Gene/Q, Cray XE6 and InfiniBand systems. *Future Generation Computer Systems*, 30:291–304, 2014.

[20] V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science & Engineering*, 13(3):92–95, 2011.

[21] Lawrence Livermore National Laboratory. Livermore unstructured lagrangian explicit shock hydrodynamics (lulesh). https://codesign.llnl.gov/lulesh.php.

[22] N. Liu and C. D. Carothers. Modeling billion-node torus networks using massively parallel discrete-event simulation. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–8. IEEE Computer Society, 2011.

[23] N. Liu, A. Haider, X.-H. Sun, and D. Jin. FatTreeSim: Modeling large-scale fat-tree networks for HPC systems and data centers using parallel and discrete event simulation. In *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation*, pages 199–210. ACM, 2015.

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[25] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[26] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 366–376. IEEE, 2012.

[27] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. A case study in using massively parallel simulation for extreme-scale torus network codesign. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation*, pages 27–38. ACM, 2014.

[28] National Energy Research Scientific Computing Center (NERSC). Hopper system. http://www.nersc.gov/users/computational-systems/hopper/.

[29] D. M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

[30] K. Pedretti, C. Vaughan, R. Barrett, K. Devine, and K. S. Hemmert. Using the Cray Gemini performance counters. *Proc Cray User Group (CUG)*, 2013.

[31] A. J. Peña, R. G. C. Carvalho, J. Dinan, P. Balaji, R. Thakur, and W. Gropp. Analysis of topology-dependent MPI performance on Gemini networks. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 61–66. ACM, 2013.

[32] K. S. Perumalla. $\mu\pi$: a scalable and transparent system for simulating MPI programs. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 62. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[33] K. S. Perumalla and A. J. Park. Simulating billion-task parallel programs. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2014), International Symposium on*, pages 585–592. IEEE, 2014.

[34] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[35] N. Saboo, A. K. Singla, J. M. Unger, and L. V. Kalé. Emulating petaflops machines and Blue Gene. In *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium*, IPDPS '01, pages 195–, Washington, DC, USA, 2001. IEEE Computer Society.

[36] Sandia National Laboratories. Dumpi: The mpi trace library. http://sst.sandia.gov/about_dumpi.html.

[37] N. Santhi, S. Eidenzenz, and J. Liu. The Simian concept: parallel discrete event simulation with interpreted languages. In L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, editors, *Proceedings of the 2015 Winter Simulation Conference*, 2015.

[38] US Department of Energy. Department of Energy Awards $425 Million in Next Generation Supercomputing Technologies. http://energy.gov/articles/department-energy-awards-425-million-next-generation-supercomputing-technologies, 2014.

[39] B. Van Straalen and P. Collela. Resiliency and codesign. In *DOE Exascale Research Conference*, 2012.

[40] A. Vishnu, J. Daily, and B. Palmer. Designing scalable PGAS communication subsystems on Cray Gemini interconnect. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10. IEEE, 2012.

[41] S. J. Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, 1996.

[42] N. Wright, H. Shan, F. Blagojevic, H. Wasserman, T. Drummond, J. Shalf, K. Fuerlinger, K. Yelick, S. Ethier, M. Wagner, et al. The NERSC-Cray center of excellence: Performance optimization for the multicore era. *CUG Proceedings*, 2011.

[43] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 78. IEEE, 2004.