Durango: Scalable Synthetic Workload Generation for Extreme-Scale Application Performance Modeling and Simulation

Christopher D. Carothers
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180
chrisc@cs.rpi.edu

Jeremy S. Meredith
Oak Ridge National Laboratory
Oak Ridge, TN 37831
jsmeredith@gmail.com

Mark P. Blanco
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180
mark.p.blanco@gmail.com

Jeffery Vetter
Oak Ridge National Laboratory
Oak Ridge, TN 37831
vetter@ornl.org

Misbah Mubarak
Argonne National Laboratory
Lemont, IL 60439
mmubarak@anl.gov

Justin LaPre
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180
laprej@cs.rpi.edu

Shirley Moore
Oak Ridge National Laboratory
Oak Ridge, TN 37831
mooresv@ornl.gov

ABSTRACT

Performance modeling of extreme-scale applications on accurate representations of potential architectures is critical for designing next generation supercomputing systems because it is impractical to construct prototype systems at scale with new network hardware in order to explore designs and policies. However, these simulations often rely on static application traces that can be difficult to work with because of their size and lack of flexibility to extend or scale up without rerunning the original application. To address this problem, we have created a new technique for generating scalable, flexible workloads from real applications, we have implemented a prototype, called *Durango*, that combines a proven analytical performance modeling language, *Aspen*, with the massively parallel HPC network modeling capabilities of the CODES framework.

Our models are compact, parameterized and representative of real applications with computation events. They are not resource intensive to create and are portable across simulator environments. We demonstrate the utility of Durango by simulating the LULESH application in the CODES simulation environment on several topologies and show that Durango is practical to use for simulation without loss of fidelity, as quantified by simulation metrics. During our validation of Durango's generated communication model of LULESH, we found that the original LULESH miniapp code had a latent bug where the `MPI_Waitall` operation was used incorrectly. This finding underscores the potential need for a tool such as Durango, beyond its benefits for flexible workload generation and modeling.

Additionally, we demonstrate the efficacy of Durango's direct integration approach, which links *Aspen* into CODES as part of the running network simulation model. Here, Aspen generates the application-level computation timing events, which in turn drive the start of a network communication phase. Results show that Durango's performance scales well when executing both torus and dragonfly network models on up to 4K Blue Gene/Q nodes using 32K MPI ranks, Durango also avoids the overheads and complexities associated with extreme-scale trace files.

SIGSIM-PADS'17 May 24–26, 2017, Singapore, Singapore

ACM ISBN 978-1-4503-4489-0/17/05…15.00

DOI: http://dx.doi.org/10.1145/3064911.3064923

CCS Concepts

•Networks → Network simulations; •Computer systems organization → Parallel architectures; Multicore architectures;

Keywords

Massively Parallel Simulation, HPC Networks Models, Structural Analytic Models

1. INTRODUCTION

Performance modeling of extreme-scale applications using accurate representations of potential architectures is critical for designing next generation supercomputing systems because it is impractical to construct prototype systems at

scale with new network hardware in order to explore designs and policies. However, simulations often rely on static application traces that can be difficult to work with because of their size and lack of flexibility to extend or scale up without rerunning the original application. For example, the application traces available as part of the Design Forward program (see: http://www.exascaleinitiative.org/design-forward) can be hundreds of gigabytes. Moreover, once traces are created, they are fixed and cannot be changed Also, traces require a system and time where the trace can be created.

On the other hand, well-known patterns [10] coded in a simulator-specific language also have shortcomings. First, these patterns typically are synthetic and not often representative of real application behaviors, thus driving the need for real application traces. Second, these patterns often do not include any computation for the processors, so there is limited ability to inject realistic processor behaviors between communication events.

To address this problem, we have created a new technique for generating scalable workloads from real applications, and we have implemented a prototype, named Durango, that combines a proven analytical performance modeling language, *Aspen*, with the massively parallel HPC network modeling capabilities of the CODES framework. Our models are compact, parameterized and representative of real applications with computation events. They are not resource intensive to create and are portable across simulator environments. Specifically, we make the following two contributions in this paper:

- Comparison of the Aspen-generated network communication patterns for the LULESH miniapp with real LULESH application network communications via traces that are run through the CODES packet-level network simulation framework. Durango shows identical agreement with the real application trace data for key network performance statistics. During our validation of Durango's generated communication model of LULESH, we found that the original LULESH miniapp code had a latent bug where the `MPI_Waitall` operation was used incorrectly. This finding underscores the potential need for a tool such as Durango, beyond its benefits for flexible workload generation and modeling.

- A scaling study of Durango's direct integration approach, which links Aspen into CODES as part of the running network simulation model. Here, Aspen generates the application-level computation timing events as part of an overall discrete-event system model, which in turn drive the start of a network communication phase. Performance results show that Durango's performance scales well when executing both torus and dragonfly network models on upto 4K Blue Gene/Q nodes using 32K MPI ranks and avoids the overheads and complexities associated with extreme-scale trace files.

2. Durango OVERVIEW

To motivate Durango, we refer to Figure 1, which illustrates the multiple workflows for generating a simulation workload. In this paper, we consider four scenarios when capturing the characteristics of the application to be simulated. In the first scenario, we simply execute the application, capturing the communication events, and transferring execution control between the application and simulation as appropriate (path 1-2-3-9 in Figure 1). The application is built as it normally would be, but the communication events are intercepted by the simulator and then simulated on the theoretical network. This method is relatively straightforward and easy to perform, but it has the disadvantage that it uses considerable resources in terms of execution time and memory.

In the second scenario, we use a tracing tool, such as DUMPI, to capture an event trace of all the communicating tasks and communication events in the application. This trace is digested by the simulator (path 1-4-5-3-9 in Figure 1) and the trace information is typically stored in a huge file. Moreover, all the parameters of the trace are fixed at capture time; that is, the architect cannot change the problem size or the number of processors after the trace has been created.

In the third scenario, we use synthetic communication patterns as a proxy for the application communication patterns (path 1-6-7-4-5-3-9 in Figure 1). In the fourth scenario, the proxy trace generator is glued directly into the simulator (path 1-6-8-2-3-9 in Figure 1). These last two approaches are the focus of this paper and are combined into a system we call *Durango*.

2.1 Aspen Overview

Aspen is a domain-specific language designed for analytical performance modeling [28, 31]. The models represented in Aspen comprise application models and abstract machine models. The machine models describe the hierarchy of a machine as well as speeds and feeds of its components for processing computation and communication. Application models contain descriptions of resource usage of an algorithm (such as the computation and communication requirements) and control flow (iteration, sequential and parallel dependencies, and kernel nesting).

The COMPASS framework described in [17] generates a parameterizable performance model from a target application's source code using OpenARC [18] for automated static analysis and then evaluates this model using various performance prediction techniques available in Aspen. Prior to using OpenARC, a small amount of manual annotation is required in order to specify regions of interest and declare parameters to be directly exposed in the generated Aspen application model. The Aspen control flow walker can dynamically instantiate values of parameters that cannot be determined statically. Generation of applications models for the LULESH proxy application and the matrix multiplication kernel used in the Durango research is described and validation of the Aspen resource usage and runtime predictions with measurements on a real system is given in [18]. Aspen currently combines a the throughput-based node performance model based on the Roofline model [32] with a simple latency plus bandwidth communication model, where computation and communication can be overlapped to a specified degree. Part of the motivation of the Durango research is to achieve more accurate communication modeling than is possible using Aspen's analytical methods.

Aspen, as an analytical tool, was initially designed to compute symbolic results for analysis queries, such as the number of floating-point operations at a given problem size or the performance of a kernel as a function of bus bandwidth. However, the expansion of Aspen's capabilities has allowed for more complex uses; while an Aspen model is not source code and cannot be executed per se, the representation of a control flow is sufficiently rich to allow Aspen-based tools to traverse an application model with the same control flow as

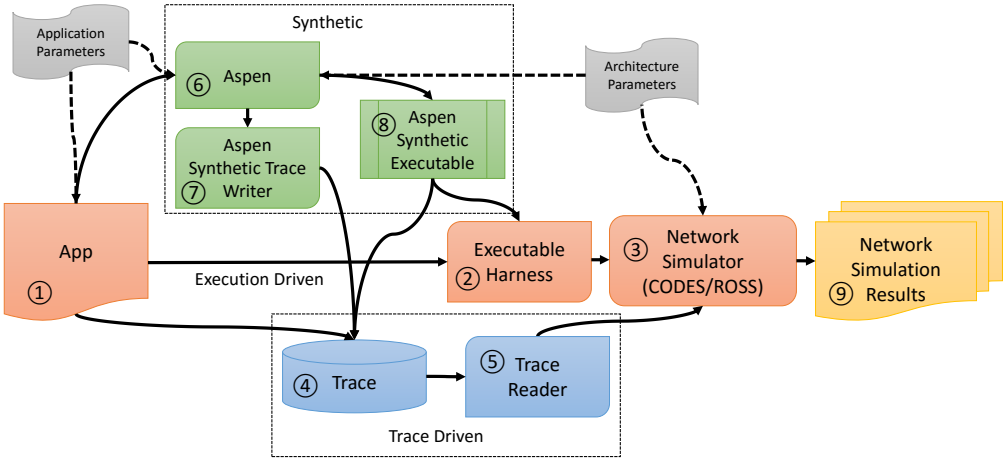

Figure 1: Durango overview.

with the original application. We build this new capability to provide synthetic communication trace generation along with direct integration in Durango.

2.2 Durango Approach

In contrast to the first two scenarios described above, our new approach allows the architect to create a parameterized model of an application, and then instantiate the application with specific parameters at simulation time. The potential benefit of this approach is multifaceted. First, the model is malleable: the user can easily change the application and architecture parameters. In our approach, a model can be instantiated for 16 MPI tasks as well as 1M MPI tasks. Second, the models are compact, typically being only a dozen lines for miniapplications and up to a few hundred lines for real applications. Third, the Durango models include computational and I/O events in addition to the communication events so that computation and I/O demands can change in concert with communication parameters. This approach has multiple benefits: the synthetic workload can be generated dynamically as necessary with the required architecture and application parameters; the description is compact; and the Aspen model can be as detailed or as sparse as required, including computation, communication, I/O, and other important events.

More specifically, Durango requires that the user create a parameterized Aspen model (6) of the application (1) and then use the Aspen model to create a synthetic workload, either by generating a compatible trace directly (7) or by generating a synthetic MPI application (8) that can then be traced (4-5-3-9 in Figure 1) or directly included in the simulation executable that avoids the need to perform expensive read operations of large trace datasets.

2.3 Representing Communication in Aspen

Resource descriptions in Aspen are user-defined. There are common conventions such as `flops`, `loads`, and `stores` for floating point operations and memory traffic, but these must merely match what is defined in the abstract machine model for Aspen to be able to return predictive costs such as runtimes and power consumption.

A commonly used resource for MPI communication is `messages`, but this results in a simplistic mapping. Extensions via traits enable more expressive message passing patterns. For example, a trait description such as "as positive_x" can direct an Aspen-based tool to generate MPI calls if given a regular processor decomposition. However, these trait descriptions are low-level, however, and require significant effort from modelers to achieve communication patterns like 3D nearest-neighbor; for example, requiring up to 26 separate `message` resource calls to generate message traffic for each neighboring processor.

Listing 1: Example Aspen model with communication

```
1 model example
2 {
3   param nelem = 23
4   param wordsize = 8
5
6   kernel main
7   {
8     execute{flops [8*nelem^3] from Domain as
9             simd}
10    execute{comm [1] of size [word] as allreduce,
11            min}
12    execute{comm [nelem] of size [3*word] as nn3d,
13            face}
14  }
15 }
```

Instead, we created a new resource convention, `comm`, that represents communication patterns at a more semantic level. An abstract machine model can still easily interpret these in the same manner as the simpler `message` construct for the purposes of determining costs, but we are now free to interpret this new `comm` resource at a higher level in an Aspen tool in order to generate synthetic patterns for executables and traces.

Listing 1 shows an example Aspen application model using this new `comm` resource. Note how our new Aspen `comm` resource corresponds to popular synthetic communication patterns such as nearest-neighbor but can be customized for application kernels as well. In particular, the pattern is listed in the traits for the resource, and any options are listed as additional traits. For example, line 10 has the *allreduce* collective pattern with a *min* operation, and these are represented in the traits as `as allreduce, min`. Combined with the quantity and size, that resource description is sufficient to generate a synthetic MPI call. Line 12 has

the nearest-neighbor 3D point-to-point pattern (`nn3d`) for a domain size of `nelelm` with three double-precision fields (`3*word`) and communication only along the six faces of each domain (`face`).

2.4 Synthetic Program Execution

Aspen models are not designed to be "executed" per se but have control flow such as iteration loops, parallel maps, and kernel invocations. This control flow information was sufficient for us to create a "walker" tool that uses the Aspen library to traverse application models in control flow order, as if they are executable programs.

The tool we created supports two major types of traversal: implicit and explicit. The difference typically manifests in control structures such as iteration. For example, if kernel K is called from within an iteration control with count 7, the explicit traversal will descend into the kernel call seven times, while the implicit traversal will descend into the kernel call only once but knows that it is executing with multiplicity seven with a sequential dependence.

Some types of analyses are amenable to implicit traversal. For counting floating-point operations, we can typically count how many operations are in kernel K in our example above and multiply by seven. This is not possible in all cases, however, for example, if floating point operation counts vary stochastically, we must use explicit traversal and sample values at each iteration. In the case of synthetic trace generation, both implicit and explicit traversal have their place.

In typical analyses, implicit traveral will be faster because it performs less analysis and accumulates results in bulk to provide the correct answer. In the context of generating a synthetic MPI executable, in the implicit mode we can output a C `for` loop containing the appropriate MPI calls, while the explicit mode would require generating many C copies of the same function calls. If Aspen were to hook up directly to a simulator, we would, by necessity, switch to explicit traversal because Aspen would need to feed the simulator every MPI call generated by the synthetic program.

Listing 2: Aspen model with control flow and communication.

```
model mpitest {
  kernel main {
    iterate [10] {
      execute { comm [2] of size [word] as
                allreduce, min }
    }
  }
}
```

2.5 Durango-Instantiated Executable

Our first option for generating a synthetic workload is to instantiate an MPI-based source code file that captures the parameters and patterns of the application via the Aspen model. Listing 2 shows an example of a simple Aspen model with an iteration loop along with a single communication pattern — in this case, an Allreduce. Listing 3 shows the output code (minus boilerplate) for this small Aspen model. Note that this output was captured in implicit mode; in explicit mode we would get ten copies of the body of the `for` loop, instead of a `for` loop with a count of ten. After generating the source code instantiating the model's control flow and communication patterns, we can compile and execute it, optionally capturing a trace for study as output from the Durango tool.

Listing 3: Source generated by model in Listing 2 using implicit traversal.

```
for (int loopctr=0; loopctr<10; loopctr++)
  {
    int nwords=2;
    std::vector<float> sendvec(nwords, rank*1.0f);
    std::vector<float> recvvec(nwords);
    MPI_Allreduce(&(sendvec[0]), &(recvvec[0]),
                  nwords, MPI_FLOAT,MPI_MIN,
                  MPI_COMM_WORLD);
  }
```

2.6 CODES: An Extreme-Scale Systems Modeling and Simulation Framework

To demonstrate our new Durango's methodology and functionality, we have integrated Durango with a popular massively parallel simulation system for interconnection networks, CODES/ROSS.

CODES enables the design-space exploration of HPC networks and storage systems with the help of scalable discrete-event simulations of interconnection networks and storage systems. CODES uses ROSS as its underlying parallel discrete-event simulation framework, which enables efficient and scalable network models thanks to the optimistic event scheduling capability of ROSS [22]. The core object within a ROSS model is a *logical process (LP)*, which models some distinct component of a network such as a terminal or router. Simulation time is advanced by LPs exchanging time-stamped event messages. The optimistic parallel synchronization approach used by ROSS guarantees that events are processed in time stamp order.

CODES supports high-fidelity network models for dragonfly, torus, and SlimFly interconnect topologies. It uses an abstraction layer on top of the network models that allows users to conveniently plugin multiple network topologies while making minimal changes to their simulation code. The network topologies are simulated at a packet-level detail with congestion control being modeled through a credit-based flow control methodology on the virtual channels. The dragonfly network model in CODES is built on the high-radix, low-cost network configuration proposed by Kim et al. [15, 16]. It models four forms of routing algorithms: minimal, non-minimal, adaptive, and progressive adaptive routings. Multiple virtual channels are used for deadlock avoidance with different routing algorithms. The dragonfly model has been validated against the *Booksim* interconnect simulator using synthetic traffic patterns [21]. The torus network model is inspired by the Blue Gene architecture. It uses a bubble-escape virtual channel for deadlock avoidance with deterministic dimension-order routing. Validation of the torus model has been carried out against the Blue Gene/P and Blue Gene/Q architectures [20]. The SlimFly network model is based on another high-radix network topology proposed by Besta and Hoefler to reduce the network cost and diameter [6]. The CODES SlimFly simulation results were validated against the simulator by Besta and Hoefler.

The CODES network models report detailed statistics about network performance for each simulated network node and router. Using metrics such as the average number of hops traversed, average packet latency, data transmitted, and number of packets completed. Detailed statistics are reported at the network link level, including the amount of

data transmitted at each network link and the time that the link gets saturated during the simulation. These metrics can be used to get detailed insight into the network performance with different workloads.

To replay the MPI operations on CODES network models, one needs a mechanism that avoids transmitting back to back MPI send messages on the network. Figure 2 shows how the MPI simulation layer interacts with the CODES network abstraction layer to replay the workload operations on top of the simulated networks. The MPI simulation layer in CODES digests the MPI operations from the workloads and simulates them on top of the network models. It tracks the queues of MPI sends and receives, matches sends with the receives, and simulates MPI wait and ait-all operations. This functionality is vital for maintaining the correct causality order of MPI operations coming from the traces.

2.7 Durango Direct Integration: Aspen with CODES

At the core of the Durango direct integration approach is a CODES discrete-event model that drives a network simulation component, coupled to an Aspen-based runtime estimator for parallel applications. The model defines Aspen Server Logical Processes (Aspen LPs), which are the entities in the simulation responsible for driving the creation of network traffic and for performing Aspen-related computations. Each Aspen LP is paired to a corresponding CODES network terminal LP to facilitate communication with the network layer within the CODES model. The CODES network LPs are generated and organized based on the network topology chosen for the current runtime estimation. When Aspen is called through the Aspen Server LPs, the estimation parameters are passed from the primary configuration file for Durango. In return, Aspen returns the runtime estimate based on the application and the machine details specified by the configuration file.

Listing 4: Aspen LP kickoff event handler.

```
static void handle_kickoff_event(
  aspen_svr_state * ns,
  tw_bf * b,
  aspen_svr_msg * m,
  tw_lp * lp)
{
    int dest_id;
    aspen_svr_msg m_local;
    aspen_svr_msg m_remote;

    m_local.aspen_svr_event_type = LOCAL;
    m_local.src = lp->gid;
    m_remote.aspen_svr_event_type = REQ;
    m_remote.src = lp->gid;

    /* record when transfers started
    // on this server */
    ns->start_ts = tw_now(lp);

    dest_id = get_next_server(lp);

    model_net_event(net_id, "test",
        dest_id, payload_sz, 0,
        sizeof(aspen_svr_msg),
        (const void*)&m_remote,
        sizeof(aspen_svr_msg),
        (const void*)&m_local, lp);
    ns->msg_sent_count++;
}
```

Under the current direct integration mode, Durango simulates a given application in two-step rounds of network simulation and computation runtime estimation, handled by the internal CODES model and Aspen respectively.

Figure 3 illustrates the runtime of a network-computation round in greater detail. First a CODES network simulation is executed, and then the 0th Aspen Server LP, labeled "Aspen Master," reduces and processes all network data. Once the data have been reduced, the Master LP passes control of the simulation to the Aspen runtime in order to estimate the computation cost for the current round. Aspen returns a runtime estimate, and then returns control to the Master LP, which resumes subsequent simulator rounds by sending network restart events to all other Aspen LPs.

Listing 5: Aspen computation event handler.

```
static void handle_computation_event(
    aspen_svr_state * ns,
    tw_bf * b,
    aspen_svr_msg * m,
    tw_lp * lp)
{
  // Compute overall network time elapsed:
  delta += end_global - start_global;

  /* Call ASPEN framework to estimate
  // computation cost: */
  delta += runtimeCalc(Aspen_App_Path[
    roundsExecuted - computationRollbacks],
    Aspen_Mach_Path,
    Aspen_Socket[roundsExecuted -
    computationRollbacks]);

  // Save totalRuntime and then update it:
  m->end_ts = totalRuntime;
  totalRuntime += delta;

  // Increment number of rounds executed:
  roundsExecuted ++;
  .
  .
  .
}
```

Each CODES network simulation phase begins with "kickoff" events sent by the Master LP to all Aspen Server LPs, itself included. The code for handling kickoff events is shown in Listing 4. In response to kickoff events, Aspen LPs record the current simulation time and use `get_next_server(lp)`, a mapping function that uses the underlying CODES API for organizing the logical processes to retrieve the identity of the LP that they will communicate with for the duration of the network-computation round. Depending on the network traffic type specified in the configuration file, `get_next_server(lp)` returns a different ID. If nearest neighbor is specified, for example, then the next logical process ID that corresponds to an Aspen LP is returned. This is an important distinction, since some of the logical processes in the simulation serve other purposes, such as network terminals, rather than Aspen Servers. In the case of a random network traffic pattern, a random identity corresponding to any other Aspen LP is returned. This means that multiple Aspen LPs may communicate with one Aspen LP for the duration of the round, but it does not mean that they may communicate with themselves. The kickoff handler ends after the CODES API is used with `model_net_event(...)` to send a "ping" to the LP whose ID was returned by `get_next_server(lp)`.

Upon receipt of a "ping" request, Aspen LPs respond with an acknowledgment event message to the sender. For every "ping" and "ack" event that an Aspen LP receives, counters

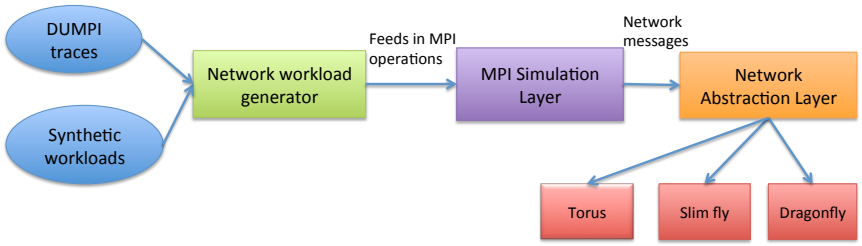

Figure 2: CODES network simulations.

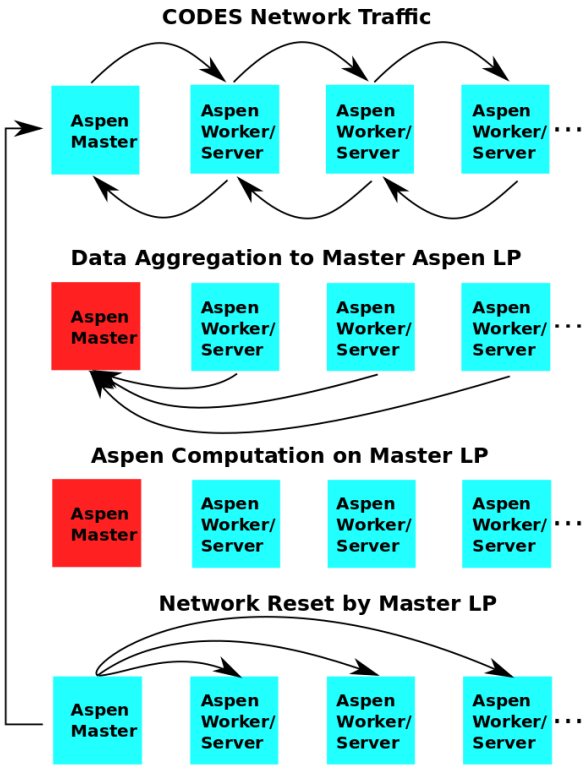

Figure 3: Durango hybrid runtime.

are incremented and saved in the logical process state. The request-acknowledgment interchange continues until the Aspen LP that initiated the exchange has sent the configured number of requests and received the correct number of acknowledgments. The Aspen Server that initiated the exchange records the simulation timestamp at which network communication ended with its counterpart.

With the first half of the network-computation round completed, the Aspen LPs cease driving network communications and send their start and end timestamps to the 0th Aspen LP. Note that the Aspen LPs could simply send their own total elapsed network communication time instead of the separate start and end values; however, then the overall longest communication time might not be accurately measured if the LP with the longest communication time begins simulation before some other LPs but also ends before other LPs that began their network conversations later. In all cases, the total network time elapsed is the difference between the globally earliest and globally latest start and end timestamps, respectively. As a result, the 0th LP is responsible for finding the longest time spent in the network phase overall and therefore keeps track only of the earliest and latest values it receives from all of the Aspen LPs, itself included. This process is handled by the `handle_data_event(...)` handler. Once all timestamps have been received, the Master LP sends an "Aspen Computation" event to itself. The "Aspen Computation" event handler performs a function call to the Aspen simulation engine with paths to the application model and machine model sourced from the Durango configuration file. In Listing 5, an excerpt of the computation event handler, the total network runtime, is calculated. Then a call to Aspen is made, passing the parameters of the application and machine model as well as which compute socket to run on. The computation runtime is returned and added to the global running counter, marking the end of the network-computation round.

If only one round has been configured in the Durango configuration file, then the simulation will terminate with only one iteration of CODES network simulation and Aspen runtime computation estimation. Otherwise, the Master LP re-sends kickoff messages to each Aspen Server LP, signaling the start of a fresh network-computation round. Upon receiving the re-kickoff message, all Aspen Server LPs then find a new LP with which to communicate and start the next network simulation phase.

3. SYNTHETIC TRACE AND PERFORMANCE EVALUATION

In this paper, we describe two series of experiments. The first uses the LULESH miniapp and compares it with Durango's generated facsimile of its communication pattern. For these experiments, a 64-way multicore system is used with 64 GB of RAM and 2.2 TB of disk space. Here, the real LULESH DUMPI trace for a 4x4x4 grid (64 MPI ranks) configuration is compared with that of the Durango generator, which also runs and creates a DUMPI trace. For the comparison, both DUMPI traces are run through the CODES torus network simulator, which is configured with a torus network topology in a 4x4x4 configuration.

In the second series of experiments, we demonstrate the efficacy of our direct integration approach. Performance results are shown for Durango when a computational kernel implemented in Aspen is linked into a executable network model. For this series of experiments, both a torus and a

dragronfly network are used. All simulations are run in parallel on the "AMOS" IBM Blue Gene/Q supercomputer located at the Center for Computational Innovations at Rensselaer.

3.1 Durango Generated vs. Real LULESH Results

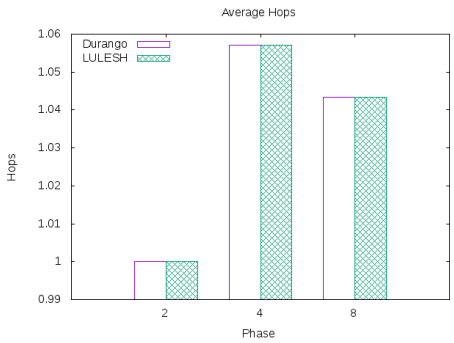

Figure 4: LULESH vs. Durango: Average torus network packet hop count as a function of the different LULESH phases.

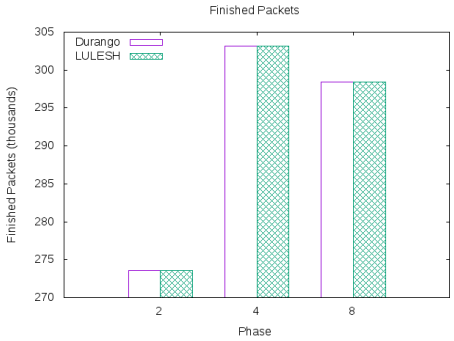

Figure 5: LULESH vs. Durango: Finished torus network packet count as a function of the different LULESH phases.

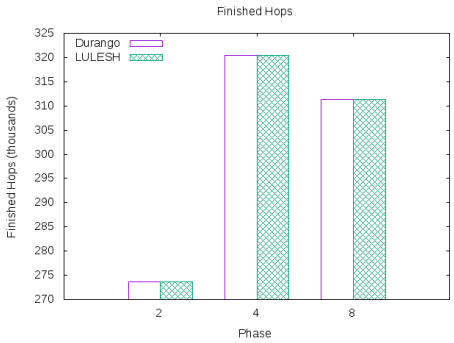

Figure 6: LULESH vs. Durango: Finished torus network packet hop count as a function of the different LULESH phases.

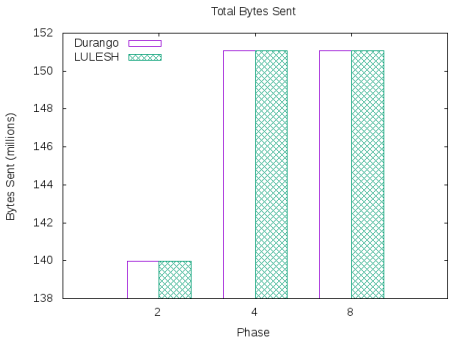

Figure 7: LULESH vs. Durango: Total torus network bytes sent as a function of the different LULESH phases.

3.1.1 *LULESH Proxy Application*

LULESH [1, 13] is a scientific computing application that performs explicit shock hydrodynamics calculations on an unstructured grid. It has been ported to several programming models. To explore modeling of the communication behavior, we studied the parallel version of LULESH implemented by using MPI for interprocess communication.

LULESH has a processor decomposition that is regular. While the mesh elements are defined with explicit connectivity allowing for unstructured elements other than hexahedra, the implementation of communication between the problem domains has a logical structure that allows nearest neighbor communication along the three-dimensional mesh $i/j/k$ directions.

LULESH has three styles of communication:

> **(A)** bidirectional nearest-neighbor communication across only faces;
> **(B)** bidirectional nearest-neighbor communication across faces, edges, and corners and
> **(C)** unidirectional communication across faces, edges, and corners, from a lower-rank task to a higher-rank task.

LULESH has four phases of communication, each conforming to one of these three styles. We label experiments using a bit pattern representing which phase is active: "1" for the first phase (style B), "2" for the second phase (style A), "4" for the third phase (style B), and "8" for the fourth phase (style C). This allows us to describe the behavior of the full application with multiple phases active by summing the bit patterns of every active phase. Because the first communication phase in LULESH is used only for problem initialization, we focused on the analysis of the other three phases, which occur every time step. The analysis results are labeled "2," "4" and "8" for their respective LULESH phases.

3.1.2 *Experimental Results*

In this first series of experiments, we compare the CODES network output statistics for the Durango generated and the real LULESH miniapp MPI communication traces. The key network statistics are as follows:

- **Average packet hop count:** the total number of hops traversals divided by the total number of packets sent into the network

- **Finished packets:** the total number of packets that reached their final destination
- **Finished packet hop count:** the total number of hop traversals divided by the total number of packets that reached their final destination
- **Total bytes sent:** total amount of data sent into the network

These statistics avoid any measure of time either absolute or relative such as an interarrival time. While such information is important, it largely depends on the amount of compute time between MPI messages and/or architectural features of the underlying machine model. The four statistics we use capture the communication behaviors that are independent from computation.

In validating Durango's generated LULESH communication patterns relative to the original LULESH miniapp code, we discovered an incorrect use of the `MPI_Waitall` operation by LULESH, whereas the Aspen-generated code was correct. Specifically, the number of wait requests sent is hard coded to 26 when only a small fraction (e.g., about 8) of the 26 available `MPI_Isend` and `MPI_Irecv` requests were used in this specific scenario. This error resulted in the CODES simulation prematurely terminating because of unmatched requests within the `MPI_Waitall`, which is a correct behavior for the simulator since it denotes a "bad" trace. Once provided the right number of active requests, the `MPI_Waitall` trace events were correct, and the CODES simulator completed without error. This finding underscores the potential need for a tool such as Durango beyond its benefits for flexible workload generation and modeling.

With regards to the network results, Figure 4 shows the average packet hop count as a function of different LULESH phases. The average ranges from 1.0 in phase 2 to nearly 1.06 in phase 4. Across all the phases, we observe identical results between the Durango generated and the real LULESH communication patterns.

Figures 5 and 6 show the number of finished packets and hops as a function of LULESH phases. In Figure 5 the number of finished packets ranges between 272K and 320K packets; similarly, the finished hop counts are in the same range because the hop count for the majority of LULESH packets is a single hop away in the 3D torus network. Across all the phases, we observe identical results the Durango finished packets/hop count and the real LULESH finished packets/hop count.

Figure 7 reports the total number of packets sent as a function of LULESH phases. As before, we observe no difference between the Durango generator and the real LULESH miniapp. The range in packets is 140MB for phase 2 up to 151MB for phases 4 and 8.

3.2 Evaluation of Durango Direct Integration

The Durango direct integration approach was executed on *AMOS*, an IBM Blue Gene/Q supercomputer located at the Rensselaer Polytechnic Institute Center for Computational Innovation. *AMOS* has 5 racks, each with 1,024 nodes. Each node contains 16 GB of DDR3 RAM and one IBM A2 processor, clocked at 1.6 GHz with 16 compute cores and 64 hardware threads.

Across all tests, Durango was run using BG/Q node counts ranging from 4 to 4096 nodes and 32 to 32,768 MPI ranks. For the network simulation component, torus and dragonfly network models were configured with a nearest-neighbor traffic pattern, and a custom Aspen machine model based on the AMD processors was written for calculating runtimes with a matrix multiplication kernel model. The torus network is a 5D (8^5) topology yielding 32K nodes with each link having 2 GB/sec bandwidth. The dragonfly network's configuration is taken from [20] and has 1.3M terminal nodes.

Unlike the previous test, Aspen is driving the compute time between phases of nearest-neighbor communications. The purpose of this performance study is to demonstrate that the Aspen compute model does impede the parallel network simulation. Future work on Durango will enable the Aspen system model to drive both compute node timing activity and network patterns at the same time.

3.2.1 Runtime Configuration

Configuration of Aspen when executed in this direct integration mode with the CODES parallel simulation framework is accomplished through a primary and secondary configuration file. The primary configuration file is used to specify the Aspen compute kernel and machine models that will be utilized during each network-computation round. This configuration file also allows per-round selection of the socket on each node of the Aspen machine to be used to "run" the compute kernel, as well as the number of rounds to be simulated. The number and size of the "ping" requests sent between Aspen Server logical processes are configured as well. A sample primary configuration file is shown in part in Listing 6, where Durango has been configured to simulate two rounds of matrix multiplication on an AMD-based cluster using the CPU in the first round and the GPU in the second round over the CODES-provided "SimpleNet" testing topology. Note that in the primary configuration file, the level of debug information can be adjusted as follows:

- Level 0: supresses all debug output
- Level 1: allows configuration details to be printed
- Level 2: allows runtime progress messages to be printed

Listing 6: Aspen parameter configuration excerpt.

```
ASPEN_PARAMS
{
   debug_output="1";
   network_conf_file="simplenet.conf";
   network_traffic_pattern="random";
   num_rounds="2";
   aspen_mach_path="./models/
                machine/BigTestRig.aspen";
   socket_choice000="amd_830";
   socket_choice001="amd_HD5770";
   aspen_app_path000="./models/matmul/
                matmul.aspen";
   aspen_app_path001="./models/matmul/
                matmul.aspen";
}

server_pings
{
    num_reqs="128";
    payload_sz="1024";
}
```

The second configuration file used by Durango is included in the simulator runtime by a parameter in the primary configuration file, called `network_conf_file`. The parameters in the network configuration file are directly passed on to the underlying CODES-Net framework, and allow for network topology and size settings to be adjusted. The network configuration file also controls how logical processes

are organized in the simulator. Durango contributes the Aspen Server LP. Required by the CODES framework are the network-level "server" LPs and the chosen network topology's routing LPs. Listing 7 shows the basic network configuration file for the SimpleNet network topology.

LPGROUPS contains the *ASPEN_SERVERS* subcategory, and lists the classes and organization of the LPs in the simulation. For the SimpleNet topology the only two LP types needed are the CODES-level "modelnet_simplenet" LPs and the "server" LPs, which drive the overall Aspen-CODES simulation layer. The other two supported topologies, torus and dragonfly, also require two CODES-level LPs to function.

Listing 7: Aspen SimpleNet network configuration.

```
LPGROUPS
{
   # simplenet has a set of servers, each with
   # point-to-point access to each other
   ASPEN_SERVERS
   {
      # required: number of times to repeat
      the following key-value pairs
      repetitions="4096";
      # LP types
      server="1";
      modelnet_simplenet="1";
   }
}
# Network Params:
PARAMS
{
   # ROSS-specific parmeters:
   # - message_size:
   message_size="340";
   pe_mem_factor="512";
   # model-net-specific parameters:
   # - individual packet sizes for network
   #     operations where each "packet" is
   #     represented by an event
   # - independent of underlying network being
   #    used
   packet_size="512";
   # - order that network types will be presented
   #     to the user in
   #   model_net_set_params. In this example,
   #     we're only using a single
   #     network topology
   modelnet_order=( "simplenet" );
   # - packet scheduling algorithm
   modelnet_scheduler="fcfs";
   # - simplenet-specific parameters
   net_startup_ns="1.5";
   net_bw_mbps="20000";
}
```

PARAMS includes topology-specific configuration details; for the excerpt shown here include the packet size, network bandwidth, latency, and packet scheduling algorithm. The dragonfly network topology adds to this list the number of routers in each subgroup, the global and local channel bandwidths, and several other topology-specific parameters. The torus topology also adds several unique parameters to this section, including the torus dimensionality (and corresponding dimension sizes).

3.2.2 Performance Results

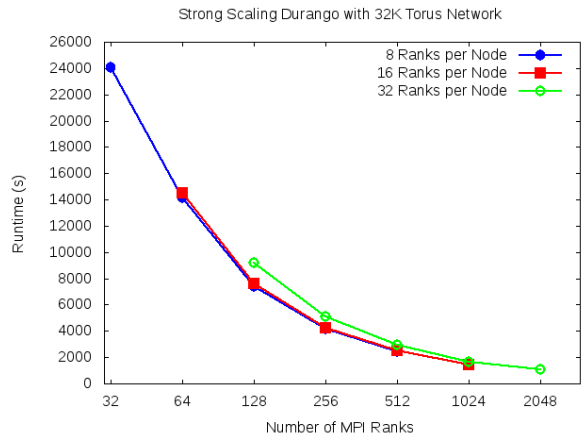

Figure 8: Durango in direct integration mode with 32K node torus network and Aspen compute node generator for 32 to 2,048 MPI ranks.

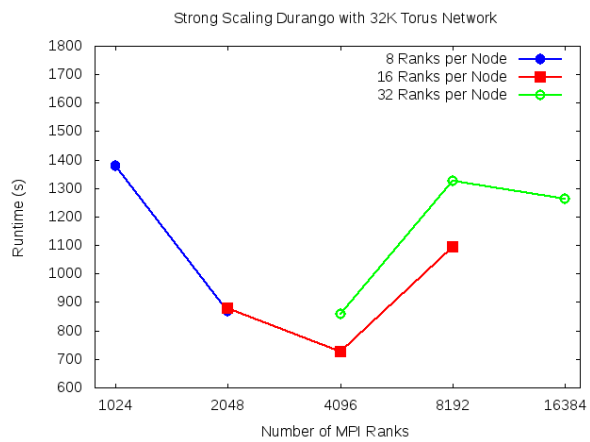

Figure 9: Durango in direct integration mode with 32K node torus network and Aspen compute node generator for 1K to 16K MPI ranks.

Figure 8 shows strong-scaling results for Durango when configured with a 32K node torus network and executed using 32 to 2048 MPI ranks across a varing number of ranks per compute node on the Blue Gene/Q supercomputer. Here, the execution time ranges from 24,000 seconds down to just over 1,000 seconds. This implies an overall worst-case to best case speedup of 24x using just 16x the hardware (e.g., 4 nodes to 64 nodes). This superlinear performance is attributed to performance gains because of a smaller memory "working set" which yields higher cache hit rates as the core counts increase. Similar performance gains were reported by Barnes et al. [5]. Parallel simulation efficiency ranges from a peak of 92% on 4 nodes with 32 ranks to a low of 61% on 64 nodes using 2048 ranks. This is to be expected because of the likelihood of out-of-order event computations grows as the number of nodes increase.

The out-of-order event computations become more problematic at larger MPI rank counts, as shown Figure 9. Here,

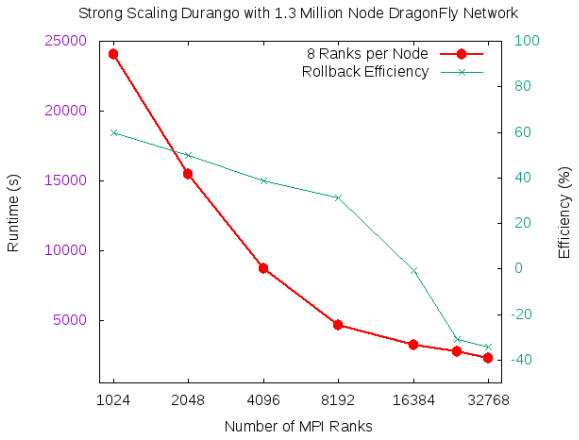

Figure 10: Durango in direct integration mode with 1.3M node dragonfly network and Aspen compute node generator for 1K to 32K MPI ranks.

the same 32K node torus network configured with an Aspen LP for determining the compute phase timing for the matrix-multiplication kernel is being scaled from 1K to 16K MPI ranks and 128 to 512 Blue Gene/Q compute nodes. The worst-case efficiency of -314% is report when using 512 nodes and 16K MPI ranks. This implies that nearly three events are being rolled back for each forward event per the efficiency definition used in [5]. Clearly, the simulation has become overly speculative. The fastest execution case is with 256 nodes and 4,096 MPI ranks and completes in 726 seconds.

If we increase the network simulation workload by using a much larger network, we observe much better Durango performance, as shown in Figure 10. Here, a 1.3M node dragonfly network is configured with the Aspen computation timing event generation. The parallel simulation run with 128 nodes and 1K MPI ranks completes in 24,000 seconds while the run with 4,096 node and 32K MPI ranks completes in just 2200 seconds. The observed performance is nearly a 11x speedup for 32x the hardware, which is in line with results reported in previous dragonfly network simulation studies [22].

Overall, the integration of Aspen's code timing event generation does not impact the overall Durango performance. Also, we avoid the performance penalty of reading in large, memory-intensive traces datasets.

4. RELATED WORK

Durango touches on many research areas but especially tracing tools and systems simulators.

Tracing Tools: Tracing is an important and traditional technique for capturing communication, memory [29], and instruction events. Relevant here are tracing tools that capture communication events in large scale systems including DUMPI [25], which is part of the Structural Simulation Toolkit (SST). DUMPI has been used to create a trace database for the key miniapps used in performance benchmarking of network design for the DOE Design Forward and Fast Forward programs. In addition to DUMPI traces, Tracer [2] provides an alternative approach to replaying trace-based communication workloads within the CODES network modeling and simulation framework.

ScalaTrace is an MPI tracing toolset that uses intra- and internode lossless compression techniques to produce near-constant-size communication traces regardless of the number of nodes while, preserving communication structure information [23]. Recent enhancements include MPI-IO and POSIX I/O tracing and improved extrapolation of MPI, MPI-IO, and POSIX I/O traces. Synthetic benchmarks that reflect application behavior can be generated from the compressed tracefiles in different ways: (1) by generating from the tracefile a high-level abstract conceptual pseudocode from which a specific implementation (e.g,. C+MPI) can be generated [34] or (2) by directly generating the C+MPI synthetic benchmark from the trace using ScalaBenchGen [33].

Compiler-Based Approaches: A compiler framework that can identify communication patterns for MPI-based parallel applications is described in [27]. The compiler uses a communication pattern representation scheme that captures the properties of communication patterns and allows manipulations of these patterns. Communication phases can be detected and logically separated within the application. The predicted LBMHD, CG and MG communication patterns were verified by comparing with application trace data.

HPC System Simulators: Dimemas [24, 26], a performance analysis tool for MPI applications, is used in conjunction with Venus, an Omnest-based network simulator [30] to perform trace-driven simulations [19, 8]. Initial results are shown at a scale of up to 512 cores on the Blue Gene/P platform.

BigSim [35] is a parallel discrete-event simulation framework built on the POSE PDES engine [2]. It has been used to explore intelligent topology-aware job mappings for the PERSC architecture [35, 7]. BigSim also can generate traces by emulating application behavior on architectures that do not yet exist. The generated application traces can then be replayed on network simulations for performance prediction. However, the POSE PDES engine imposes performance penalties, which makes it difficult to scale the simulation to large core counts. A recent study by the BigSim and CODES team demonstrates that CODES is an order of magnitude faster than BigSim [2].

Booksim [9] is a serial, cycle-accurate interconnection network simulation framework that supports multiple network topologies such as torus, fat tree, and dragonfly. Kim et al. used *Booksim* to propose the dragonfly network topology on a scale of 1,056 network nodes [15, 16]. *Booksim* supports a detailed router model for the networks along with detailed routing algorithms for each topology. A modified version of *Booksim* was also used to simulate the performance of a Slim Fly network topology [6, 14].

The WARwick Performance Prediction Toolkit (WARPP) [11] uses the following modeling approach: (1) manual or automated source code analysis to identify basic blocks and MPI calls, (2) machine benchmarking on the target machine to measure runtime for basic blocks and to do MPI benchmarking, and (3) input of the application model and the benchmarking results to a discrete event simulator. Currently, only the 2008 version of the simulation engine, which is written in Java with its accompanying documentation, is available.

POEMS [3] is an end-to-end performance analysis framework for HPC applications. This includes the modeling of the application software, runtime and operating software, and hardware architecture. Central to this framework is the POEMS Specification Language compiler that generates an end-to-end system automatically from a system specifica-

tion. Unlike POEMS, Durango enables the prediction of an application's computational and network performance on future or hypothetical hardware configurations. This feature of Durango is attributed to structural analytic modeling approach of Aspen on the application calculation side, and packet-level modeling approach of CODES on the network side. In contrast, POEMS relies on a simple latency/bandwidth hardware network model that is unable to track the detailed congestion events that can happen across different network topologies.

The Structural Simulation Toolkit (SST) is a parallel discrete-event simulation framework that uses a conservative synchronization approach to model a number of components such as network, processors and memory [12, 25]. Detailed SST models can run at small scale with a few network nodes. Large-scale network simulations use a low fidelity model buit in the SST macro layer. Here, the DUMPI trace library is used to capture the communication behavior of MPI applications.

The most recent related results are in [4]. Here, a new Python-based HPC network modeling framework that is built on the Simian parallel discrete-event engine demonstrates accurate performance predication of a Cray 3D torus network across a number of MPI application traces. A network model using 156K MPI rank trace as input is shown to efficently run in parallel on up to 3K AMD Opteron cores. Unlike Durango, this approach currently does not appear to include models for predicting the computational overhead associated with the scientific applications.

5. CONCLUSIONS

In this paper, we introduce a new performance analysis tool called *Durango* that integrates the analytical performance modeling capabilities of the Aspen domain specific language with the efficient, massively parallel network simulation capabilities of CODES. Aspen has been extended to enable communication pattern specification. The efficacy of Durango is demonstrated as a new approach to the performance modeling of extreme-scale systems in two ways:

- Comparing the Aspen generated communication patterns with real application network communications via traces that are run through the CODES packet-level network simulation framework. Durango shows strong agreement with the real application trace data for key network performance statistics.

- Performing a scaling study of Durango's direct integration approach that links Aspen with CODES as part of the running network simulation model. Here, Aspen generates the application-level computation timing events, which in turn drive the start of a network communication phase. Results show that Durango's performance scales well when executing both torus and dragonfly network models on up to 4K Blue Gene/Q nodes using 32K MPI ranks.

We plan to extend Durango's capabilities by enabling Aspen to drive both the compute kernel timing and the network communication patterns for key supercomputing applications. This extension will enable end-to-end performance prediction capabilities for current and future extreme-scale systems.

6. ACKNOWLEDGMENTS

The CODES project is supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computer Research (ASCR), under contract DE-AC02-06CH11357 managed by Lucy Nowell. The Aspen/Durango work is supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computer Research (ASCR), under contract number DE-SC0012636 managed by Richard Carlson. Additional support is provided by the Air Force Research Laboratory under contract number FA8750-15-2-0078 managed by Mark Barnell. Computational resources for this work are provided by the Center for Computational Innovations at Rensselaer Polytechnic Institute.

7. REFERENCES

[1] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[2] 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 *Workshop on Parallel and Distributed Agent-Based Simulations*, 2015.

[3] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *IEEE Trans. Software Engineering*, 26(11):1027–1048, 2000.

[4] K. Ahmed, M. Obaida, J. Liu, S. Eidenbenz, N. Santhi, and G. Chapuis. 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 '16, pages 177–187, New York, NY, USA, 2016. ACM.

[5] P. D. Barnes, C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing time warp on 1,966,080 cores. In *Proc. of the 2013 ACM SIGSIM Conf. on Principles of Advanced Discrete Simulation (PADS)*, pages 327–336, May 2013.

[6] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *Proc. of the Int. Conf. for High Performance Comput., Networking, Storage and Anal. (SC)*, pages 348–359, 2014.

[7] A. Bhatele, N. Jain, W. Gropp, and L. V. Kale. Avoiding hot-spots on two-level direct networks. In *Int. Conf. for High Performance Comput., Networking, Storage and Anal. (SC).*, pages 1–11, 2011.

[8] R. Birke, G. Rodriguez, and C. Minkenberg. Towards massively parallel simulations of massively parallel high-performance computing systems. In *Proc. of the 5th Int. ICST Conf. on Simulation Tools and Techn.*, pages 291–298, 2012.

[9] W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers, San Francisco, 2003.

[10] W. J. Dally and B. P. Towles. *Principles and Practices of Interconnection Networks*. Burlington, MA, USA: Morgan Kaufmann, 2004.

[11] S. Hammond, G. Mudalige, J. Smith, S. Jarvis, J. Herdman, and A. Vadgama. WARPP: A toolkit for simulating high performance parallel scientific codes.

In *2nd International Conference on Simulation Tools and Techniques*, Rome, Italy, Mar. 2009. ACM SIGSIM.

[12] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo. A simulator for large-scale parallel computer architectures. *Technology Integration Advancements in Distributed Systems and Computing*, 179, 2012.

[13] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[14] G. Kathareios, C. Minkenberg, B. Prisacari, G. Rodriguez, and T. Hoefler. Cost-effective diameter-two topologies: analysis and evaluation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 36. ACM, 2015.

[15] J. Kim, W. Dally, S. Scott, and D. Abts. Cost-efficient dragonfly topology for large-scale systems. *Micro, IEEE*, 29(1):33–40, Feb. 2009.

[16] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Comput. Architecture News*, 36(3):77–88, June 2008.

[17] S. Lee, J. S. Meredith, and J. S. Vetter. COMPASS: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 405–414, Newport Beach, California, USA, 2015. ACM.

[18] S. Lee and J. S. Vetter. OpenARC: extensible openACC compiler framework for directive-based accelerator programming study. In *Proceedings of the First Workshop on Accelerator Programming using Directives (with SC14)*, pages 1–11, New Orleans, 2014. IEEE Press.

[19] C. Minkenberg and G. Rodriguez. Trace-driven co-simulation of high-performance computing systems using OMNeT++. In *Proc. of the 2nd Int. Conf. on Simulation Tools and Techn.*, pages 65–72, 2009.

[20] M. Mubarak, C. Carothers, et al. A case study in using massively parallel simulation for extreme-scale torus network codesign. In *Proc. of the 2nd ACM SIGSIM/PADS Conf. on Principles of Advanced Discrete Simulation*, pages 27–38, 2014.

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

[22] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. Enabling parallel simulation of large-scale hpc network systems. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):87–100, Jan. 2017.

[23] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel Distrib. Comput.*, 69(8):696–710, 2009.

[24] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31. March, 1995.

[25] A. F. Rodrigues et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Rev.*, 38(4):37–42, Mar. 2011.

[26] G. Rodriguez, R. M. Badia, and J. Labarta. Generation of simple analytical models for message passing applications. In *Euro-Par 2004 Parallel Processing*, pages 183–188. Springer, 2004.

[27] S. Shao, A. K. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 85–85, Washington, DC, USA, 2006. IEEE Computer Society.

[28] K. L. Spafford and J. S. Vetter. Aspen: A domain specific language for performance modeling. In *SC12: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Salt Lake City, 2012.

[29] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *Acm Computing Surveys*, 29(2):128–170, 1997.

[30] A. Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESMâĂŹ2001)*, volume 9, page 65. sn, 2001.

[31] J. S. Vetter and J. S. Meredith. Synthetic program analysis with aspen. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, pages 1–6. University of Edinburgh, 2015.

[32] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[33] X. Wu, V. Deshpande, and F. Mueller. ScalaBenchGen: Auto-generation of communication benchmarks traces. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 1250–1260, Washington, DC, USA, 2012. IEEE Computer Society.

[34] X. Wu, F. Mueller, and S. Pakin. A methodology for automatic generation of executable communication specifications from parallel mpi applications. *ACM Trans. Parallel Comput.*, 1(1):6:1–6:30, Oct. 2014.

[35] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *Proc. of 18th IEEE Int. Parallel and Distributed Process. Symp.*, pages 78–87, 2004.