Preliminary Evaluation of a Parallel Trace Replay Tool for HPC Network Simulations

Bilge Acun[1], Nikhil Jain[1], Abhinav Bhatele[2], Misbah Mubarak[3], Christopher D. Carothers[4], and Laxmikant V. Kale[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
[2] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
[3] Mathematics and Computer Science Division, Argonne National Laboratory
[4] Department of Computer Science, Rensselaer Polytechnic Institute
[1]{acun2, nikhil, kale}@illinois.edu,[2]bhatele@llnl.gov,
[3] mmubarak@anl.gov,[4]chrisc@cs.rpi.edu

Abstract. This paper presents a preliminary evaluation of TraceR, a trace replay tool built upon the ROSS-based CODES simulation framework. TraceR can be used for predicting network performance and understanding network behavior by simulating messaging on interconnection networks. It addresses two major shortcomings in current network simulators. First, it enables fast and scalable simulations of large-scale supercomputer networks. Second, it can simulate production HPC applications using BigSim's emulation framework. In addition to introducing TraceR, this paper studies the impact of input parameters on simulation performance. We also compare TraceR with other network simulators such as SST and BigSim, and demonstrate its scalability using various case studies.

1 Introduction

The design and deployment of large supercomputers with hundreds of thousands of cores is a daunting task. Both at the design stage and after the machine is installed, several decisions about the node architecture and the interconnection network need to be made. Application developers and end users are often interested in studying the effects of these decisions on their codes' performance for existing and future machines. Hence, tools that can predict these effects are important. This paper focuses on tools used for predicting the impact of interconnection networks on the communication performance of parallel codes.

Prediction of communication performance on a hypothetical or actual machine requires simulating the network architecture and its components. Discrete-event simulation (DES) based frameworks are often used to simulate interconnection networks. The usability and performance of these frameworks depend on many factors: sequential versus parallel (PDES) simulation, the level of detail at which the communication is simulated (e.g., flit-level or packet-level), whether the PDES uses conservative or optimistic methods, etc.

Existing state-of-the-art DES-based network simulators suffer from two major problems. First, sequential simulators have large memory footprints and long execution times when simulating large execution traces. Second, some simulators can only simulate synthetic communication patterns that do not accurately represent production high-performance computing applications. These shortcomings can be eliminated by using a scalable PDES engine, which improves performance and reduces the memory footprint per node, to replay execution traces generated from production HPC codes. Hence, we have developed a trace replay tool called TraceR for simulating messaging on HPC networks.

TraceR is designed as an application on top of the CODES simulation framework [6]. It uses traces generated by BigSim's emulation framework [16] to simulate an application's communication behavior by leveraging the network API exposed by CODES. Under the hood, CODES uses the Rensselaer Optimistic Simulation System (ROSS) as the PDES engine to drive the simulation [4].

The major contributions of this work are as follows:

- We present a trace-driven simulator which executes under an optimistic parallel discrete-event paradigm using reversible computing for real HPC codes.
- We show that TraceR outperforms state-of-the-art simulators like BigSim and SST in serial mode.
- We present a simulation parameter study to identify parameter values that maximize performance for simulating real HPC traffic workloads.
- We demonstrate the scalability of TraceR and show that it can simulate HPC workloads on half a million nodes in under 10 minutes using 512 cores.

2 Background and Related Work

TraceR is built upon several existing tools which are introduced briefly below.

BigSim's emulation framework: The first requirement of simulating a parallel execution is the ability to record the control flow and communication pattern of an application. The BigSim emulation framework [16] exploits the concept of virtualization in Charm++ [2] to execute a large number of processes on a smaller number of physical cores and generates traces. This enables trace generation for networks of sizes that have not been built yet. Using AMPI [11], this feature enables trace generation for production MPI applications as well.

ROSS PDES engine: ROSS [4] is a general purpose, massively parallel, discrete-event simulator. ROSS allows users to define logical processes (LPs) distributed among processors and to schedule time-stamped events either locally or remotely. ROSS provides two execution modes: conservative and optimistic. The *conservative* mode executes an event for an LP only when it is guaranteed to be the next lowest time-stamped event for it. On the other hand, the *optimistic* mode aggressively executes events that have the lowest time-stamps among the current set of events. If an event with time-stamp lower than the last executed event is encountered for an LP, *reverse handlers* are executed for the events executed out of order to undo their effects.

CODES: The CODES framework is built upon ROSS to facilitate studies of HPC storage and network systems [6]. The network component of CODES, Model-net, provides an API to simulate the flow of messages on HPC networks using either detailed congestion models or theoretical models such as LogP.

Model-net allows users to instantiate a prototype network based on one of these models. Such instantiations are controlled by parameters such as network type, dimensions, link bandwidth, link latency, packet size, buffer size, etc. CODES has been used to study the behavior of HPC networks for a few traffic patterns [6]. These traffic patterns have been implemented as one-off applications that use Model-net as a network driver. Recently, an application to replay DUMPI [1] based traces has also been added to CODES.

2.1 Related Work

BigSim is one of the earliest simulators that supports packet-level network simulation [16]. It is based on the POSE PDES engine [15] which has high overheads and impacts the scaling performance of the simulator. Structural Simulation Toolkit (SST) [9] provides both online (skeleton application based) and offline (DUMPI [1] trace based) modes for simulation. However, it uses a conservative PDES engine, does not support packet-level simulation in parallel builds, and has limited scalability with flow-based models. Booksim [10] is a sequential cycle accurate simulator that supports several topologies, but is extremely slow.

There are several network simulators that are either sequential and/or do not provide detailed packet-level (or flit-level) network simulation and/or trace-based simulation. These include the Extreme-scale Simulator (xSim) [3], DIMEMAS [7], LogGOPSim [8], MPI-Netsim [14], OMNet++ [12], and SimGrid [5].

3 Design and Implementation of TraceR

TraceR is designed as an application on top of the CODES simulation framework. Figure 1 (left) provides an overview of TraceR's integration with BigSim and CODES. The two primary inputs to TraceR are the traces generated by BigSim and the configuration parameters describing the interconnection network to be simulated. The meta-data in the traces is used to initialize the simulated processes. The network configuration parameters are passed to CODES to initialize the prototype network design.

We define the following terminology to describe the working of TraceR:

PE: Each simulated process (called PE) is a logical process (LP) visible to ROSS. It stores virtual time, logical state, and the status of *tasks* to be executed by it.

Task: The trace for a PE is a collection of tasks, each of which represents a sequential execution block (SEB). A task may have multiple backward dependencies to other tasks or to message arrivals. At startup, all tasks are marked *undone*. If a task has an *undone* backward dependency, it can not be executed.

Event: A unit entity that represents an action with a time-stamp in the PDES. We implement three types of events in TraceR:

– Kickoff event starts the simulation of a PE.

– Message Recv event is triggered when a message is received for a PE. The network message transmission and reception is performed by CODES.

– Completion event is generated when a *Task* execution is completed.

Reverse Handler: Another unit entity which is responsible for reversing the effect of an *event*. It is needed only if executing in optimistic mode.

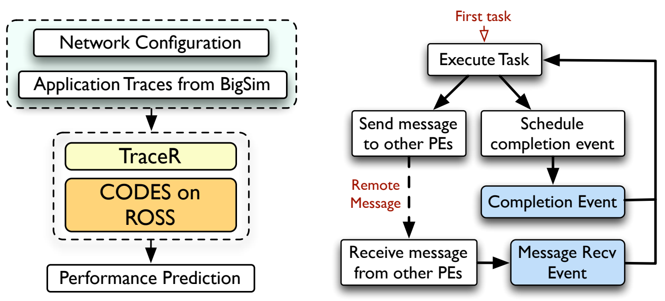


Fig. 1: Integration of TRACER with BigSim emulation and CODES (left). Forward path control flow of trace-driven simulation (right).

Let us consider an MPI application that performs an iterative 5-point stencil computation on a structured 2D grid to understand the simulation process. In each iteration, every MPI process sends boundary elements to its four neighbors and waits for ghost elements from those neighbors. When the data arrives, the MPI process performs the 5-point stencil followed by a global reduction to determine if another iteration is required. From TRACER's perspective, every MPI process is a PE. Tasks are work performed by these MPI processes locally: initial setup, sending boundary elements, the 5-point stencil computation, etc. The *Kickoff* event triggers the initial setup task. Whenever an MPI process receives ghost elements, a *Message Recv* event is generated. The dependence of the stencil computation on the receives of ghost elements is an example of a backward dependency. Similarly, posting of a receive by an MPI process is a prerequisite for TRACER to execute the *Message Recv* event.

Figure 1 (right) presents the forward path control flow of a typical simulation in TRACER. Application traces are initially read and stored in memory. When the system setup is complete, the *Kickoff* event for every PE is executed, wherein the PEs execute their first task. In the 2D stencil example, this leads to execution of the initial setup task. What happens next depends on the content of the task being executed, and the virtual time, t_s, at which the task is executed.

Every task T has an execution time t_e, which represents the virtual time T takes for executing the SEB it represents. When a task is executed, TRACER marks the PE busy and schedules a completion event for T at $t_s + t_e$ (Algorithm 1(a)). During the execution of a task, messages for other PEs may be generated. These actions are representative of what happens in real execution. When the MPI process is executing a SEB, e.g. to send boundary elements, the process is busy and no other SEB can be executed till the sending of boundary is complete. The generated messages are handed over to CODES for delivery. Note that the execution of a task in our framework only amounts to fast-forwarding of the PE's virtual time and delegation of messages to CODES; the actual computation performed by the SEB is not repeated.

When a completion event is executed, the task T is marked done and the PE is marked available (Algorithm 1(c)). Next, some of the tasks whose backward dependencies included T may now be ready to execute. Thus, those tasks are

Algorithm 1 Event handler implementation for PEs: code lines that begin with an asterisk (*) are required only in the optimistic mode.

```
$pe\_busy$: A boolean; set to true if the PE is executing a task at the current virtual time.
$ready\_tasks$: List of tasks that are ready to be executed on a PE if pe_busy is false for it.
$trigger\_task$: Map that stores the task executed at the completion of a given task.

(a) Execute_Task(task_id)
1: Get the current virtual time of the PE, $t_s$.
2: Mark the PE busy, $pe\_busy = true$.
3: Send out messages of the task with their offsets from $t_s$.
4: Get the execution time of the task, $t_e$.
5: Schedule a completion event for the PE at time $t_s + t_e$ for this task.

(b) Receive_Msg_Event(msg)
1: Find the task $T$ that depends on the message.
2: If $T$ does not have any undone backward dependencies, add $T$ to $ready\_tasks$.
3: if pe_busy == false then
4:    Get the next task, $T'$, from $ready\_tasks$.
5:    *Store $T'$ and $pe\_busy$ for possible use in the reverse handler; $trigger\_task[T] = T'$, $busy\_state[T] = pe\_busy$.
6:    Call Execute_Task($T'$).
7: end if

(c) Completion_Event(msg)
1: Get the completed task, $T$, from the $msg$.
2: Mark $T$ as $done$.
3: Set $pe\_busy = false$.
4: for every task, $f$, that depends on $T$ do
5:    if $f$ does not have any undone backward dependency then
6:       Add $f$ to $ready\_tasks$.
7:    end if
8: end for
9: Get the next task, $T'$ from $ready\_tasks$.
10: *Store $T'$ for possible use in the reverse handler, $trigger\_task[T] = T'$.
11: Call Execute_Task($T'$).
```

added to a list of pending tasks, *ready_tasks*. Finally, the task at the top of *ready_tasks* list is selected and *Execute_task* function is called (Figure 1 (right)).

As the simulation progresses, a PE may receive messages from other PEs. When a message is received, if the task dependent on the incoming message has no other undone backward dependency, it is added to the *ready_tasks* list (Algorithm 1(b)). If the PE is marked available when a message is received, the next task from the *ready_tasks* list is executed. After the initial tasks are executed, more tasks become eligible for execution. Eventually, all tasks are marked done, and simulation is terminated.

3.1 Running TraceR in Optimistic Mode

When executing in the optimistic mode, TraceR speculatively executes available events on a PE. When all the messages finally arrive, ROSS may discover that some events were executed out of order and *rolls back* the PE to rectify the error. In order to exploit the speculative event scheduling, TraceR does two things. First, during the forward execution of an event, extra information required to undo the effect of a speculatively executed event is stored. In Algorithm 1, these actions are marked with an asterisk. For both *Message Recv* and *Completion* events, the data stored includes the task whose execution is triggered by these events. For the *Message Recv* event, whether the PE was executing an SEB when the message was received is also stored. If this information is not stored by TraceR, it will get lost and hence the rollback will not be possible.

Second, as shown in Algorithm 2, reverse handlers for each of the events are implemented. These handlers are responsible for reversing the effect of forward execution using the information stored for them. For example, in the stencil code, *Reverse Handler* for a *Message Recv* event reverts the MPI process back to a state where it was still waiting for the ghost elements. In general, for a *Message Recv* event, the reverse handler marks the message as *not received*, while the reverse handler of a completion event marks the task as *undone*. In addition, the tasks that were added to the *ready_tasks* list are removed from the list. Both the reverse handlers also add the task triggered by the event to the *ready_tasks* list.

Algorithm 2 Reverse handler implementations: they use extra information stored by event handlers to undo their effect.

(a) **Message_Recv_Rev_Handler(msg)**

```
1: Find the task $T$ that depends on the message.
2: Recover the busy state of the PE, $pe\_busy = busy\_state[T]$.
3: if $pe\_busy == false$ then
4:     Add $trigger\_task[T]$ to the front of the $ready\_tasks$.
5: end if
6: Remove $T$ from the $ready\_tasks$.
```

(b) **Completion_Rev_Handler(msg)**

```
1: Get the completed task, $T$, from the $msg$.
2: Mark $T$ as undone.
3: Remove the tasks that depends on $T$ from the bottom of $ready\_tasks$.
4: Add $trigger\_task[T]$ to the front of $ready\_tasks$.
```

4 Parameter Choices for TraceR

The complexity involved in simulating real codes over a PDES engine manifests itself in a number of design and parameter choices. The first choice is the type of PDES engine: conservative versus optimistic. While the optimistic mode provides an opportunity for exploiting parallelism by speculative scheduling of events, the benefits of speculative scheduling may be offset by the repeated rollbacks for scenarios with tight coupling of LPs. Conservative mode does not pose such a risk, but spends a large amount of time on global synchronization.

Another option that is available through the BigSim emulation is defining regions of interest in the emulation traces. TRACER can exploit this functionality to skip unimportant events such as program startup. For some applications, this can speed the simulation significantly. Next, we briefly describe some other important configuration parameters:

Event granularity: This parameter decides the number of tasks to execute when an event is scheduled. We can either execute only the immediate task dependent on this event or all the tasks in the *ready_tasks* list. The former leads to one completion event per task, while in the latter method, a single completion event is scheduled for all the tasks executed as a set. The second option reduces the number of completion events to the minimum required.

Execution of one task per event may lead to a larger number of events and hence results in overheads related to scheduling and maintaining the event state. However, it simplifies the storage of information for reverse computation since only one task needs to be stored per event. In contrast, when we execute multiple tasks per event, a variable length array of all executed tasks needs to be stored. This leads to inefficiency in terms of memory usage and memory access. However, the number of total events is fewer thus reducing the PDES engine overheads. These modes are referred to as TRACER-single and TRACER-multi in Figure 5.

Parameters for optimistic mode: There are three important parameters that are available in the optimistic mode only: batch size, global virtual time (GVT) interval and number of LPs per kernel process (KP).

- *Batch size* defines the maximum number of events executed between consecutive checks on the rollback queue to see if rollbacks are required.
- *GVT interval* is the number of batches of events executed between consecutive rounds of GVT computation and garbage collection. GVT is the minimum virtual time across all LPs and its computation leads to global synchronization.
- *Number of LPs per KP*: ROSS groups LPs into kernel processes (KPs), which is the granularity at which rollbacks are performed.

5 Experimental Setup and Configuration Parameters

Proxy Applications: We use two proxy applications for evaluating and validating TraceR. *3D Stencil* is an MPI code that performs a Jacobi relaxation on a 3D process grid. In each iteration of 3D Stencil, every MPI process exchanges its boundary elements with six neighbors, two in each direction. Then, the temperature of every grid point is updated using a 7-point stencil. In our experiments, we allocate $128 \times 128 \times 128$ grid points on each MPI process, and hence have 128 KB messages. *LeanMD* is a Charm++ proxy application for the short-range force calculations in NAMD [2]. We use a molecular system of 1.2 million atoms as input to LeanMD. In our experiments, we simulate three iterations of both benchmarks.

Simulated Networks: We simulate a 3D torus of size 512 to 524,288 nodes to measure and compare simulator performance because 3D torus is the only topology available in all simulators used in the paper. For validation, we simulate a 5D torus because isolated allocations on IBM Blue Gene/Q (which has a 5D torus) allow us to collect valid performance data. Dimensions of the 3D tori are chosen to be as cubic as possible and those of the 5D tori mimic the real allocations on IBM Blue Gene/Q. For the 3D torus, we have experimented with two types of congestion models: a packet-level congestion model based on IBM's Blue Gene/P system (TorusNet) [13] and a topology-oblivious packet-level $\alpha - \beta$ model (SimpleNet). The simulation runs were performed on Blue Waters, a Cray XE6 at NCSA, while the emulation (trace generation) and validation were performed on Vulcan, an IBM Blue Gene/Q system at LLNL.

Evaluation metrics: We use three metrics to compare and analyze the performance of different simulators:

- *Execution time*: time spent in performing the simulation (excluding startup).
- *Event rate*: number of committed events executed per second
- *Event efficiency*: represents the "rollback efficiency" and is defined as:

$$\text{Event efficiency (\%)} = \left(1 - \frac{\#rolled\ back\ events}{\#committed\ events}\right) \times 100$$

Based on the equation above, when the number of events rolled back is greater than the number of committed events (events that are not rolled back, which equals the number of events executed in a sequential simulation), the efficiency is negative. A parallel simulator may be scalable even if its event efficiency is negative. This is because while using more cores may not improve event efficiency, it may reduce the execution time due to additional parallelism.

5.1 Conservative versus Optimistic Simulation

We begin with comparing the conservative and optimistic modes in TraceR. In these experiments, we simulate the execution of 3D Stencil on 4K nodes of a 3D Torus using 1 to 16 cores of Blue Waters. As shown in Figure 2, the execution time for the conservative mode increases with the number of cores, but decreases for the optimistic mode (for both TorusNet and SimpleNet). Detailed profiles of these executions show that the conservative mode performs global synchronization 43 million times which accounts for 31% of the execution time. Overall, 60% of the total execution time is spent in communication.

In contrast, the optimistic mode synchronizes only 1,239 times with communication accounting for 10% of the execution time. This is in part due to the overlap of communication with useful computation and in part due to the lazy nature of global synchronization in the optimistic mode. Based on these results, we conclude that the optimistic mode is suitable for performing large simulations using TraceR and we use it for the results in the rest of the paper.

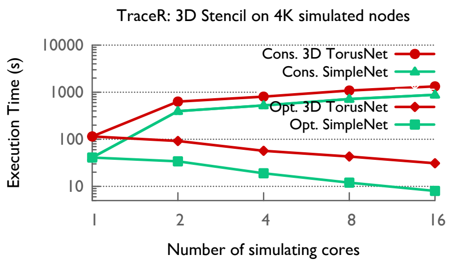


Fig. 2: Optimistic vs. conservative DES

5.2 Effect of Batch Size and GVT Interval

Figure 3 shows the impact of batch size and GVT interval on performance when simulating 3D Stencil on a 3D torus with 8K nodes and 512K nodes using 8 and 256 cores, respectively. Note that the choice of the number of simulated nodes and that of simulating cores affects the results minimally except at the limits of strong scaling. These results are for the TorusNet model; similar results were obtained for SimpleNet model also. The first observation from Figure 3 (left) is the diminishing increase in the event rate as the batch size is increased. The improvement in the event rate is because of two reasons: positive impact of spatial and temporal locality in consecutive event executions and overlap of communication with computation. However, as the batch size becomes very large, the communication engine is progressed infrequently which reduces the overlap of communication with computation. At the same time, the number of rollbacks increases due to the delay in communication and execution of pending events to be rolled back. These effects become more prominent on larger core counts as shown by the event efficiency plots in Figure 3 (right).

Next, we observe that the event rate typically improves when a large GVT interval is used. This is because as the GVT interval is increased, the time spent in performing global synchronization is reduced. Infrequent synchronization also

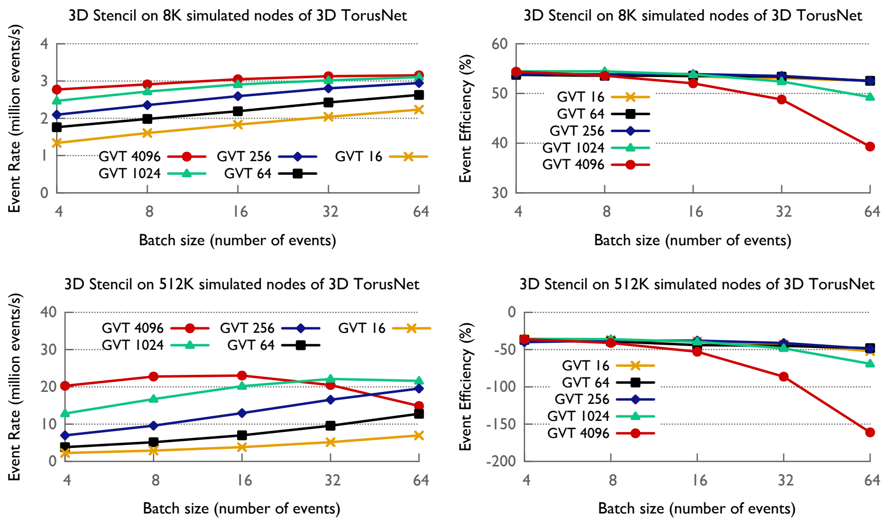


Fig. 3: Effect of batch size and GVT interval on performance: 3D Stencil on 8K simulated nodes using 8 cores (top), and on 512K simulated nodes using 256 cores (bottom).

reduces idle time since LPs with variation in load do not need to wait for one another. These results are *in contrast* to past findings that were performed on PDES with uniform loads on the LPs [4]. When a sufficiently large GVT interval is used with a large batch size, memory limits force certain LPs to idle wait till the next garbage collection. As a result, the rollback efficiency and event rates drop as shown in the Figure 3. Based on these findings, we use a batch size of 16 and GVT interval of 1024 for all simulations in the rest of the paper.

5.3 Impact of Number of LPs per KP

#LPs/KP	Efficiency (%)	Time(s)
1	51	82
2	38	92
16	2	119
128	-87	189

Fig. 4: Impact of #LPs per KP.

ROSS groups LPs together to form kernel processes (KPs) to optimize garbage collection. This causes all LPs in a KP to rollback if any one of the LPs has to rollback. In [4], it was shown that although smaller values of LPs per KP reduce the number of rolled back events, they do not have a significant impact on the execution time. Our findings, shown by a representative set in Figure 4 (obtained by simulating 3D Stencil on 8K nodes of 3D Torus with TorusNet model on 8 cores), differ – smaller values of LPs per KP *reduce* the execution time also. As we reduce the number of LPs per KP from 128 to 1, the execution time decreases by 57%.

The primary reason for the difference in impact of LPs per KP is the varying event efficiency. For synthetic benchmarks used in [4], the event efficiency is greater than 95% in all cases. As a result, any further increase caused by decreasing LPs per KP is marginal. In contrast, for real application simulations, the event efficiency is much lower. Thus, a reduction in the number of rollbacks can significantly impact the overall execution time.

6 Performance Comparison, Scaling and Validation

We now compare the performance of TraceR with other simulators and analyze its scaling performance and prediction accuracy using the packet-level model (TorusNet). Here, TraceR is executed in optimistic mode with batch size = 16, and GVT interval = 2048. The simulated network topology is 3D torus, which is the only common available topology in TraceR, BigSim, and SST.

6.1 Comparison with Sequential Executions

We first compare the sequential performance of BigSim, SST (online mode), TraceR-single, and TraceR-multi for simulating 3D Stencil's execution on various node counts. Figure 5 shows that TraceR is an order of magnitude faster than BigSim. This is primarily because of the inefficiencies in BigSim's torus model and its PDES engine. Compared to SST, the execution time of TraceR-single is lower by 50%, which we believe is due to ROSS's high performing DES engine. The performance of TraceR-single is better than TraceR-multi for these experiments. However, we found TraceR-multi to out-perform TraceR-single in other experiments for reasons that need to be explored further. In the rest of this section, we report the performance of TraceR-single as TraceR's performance.

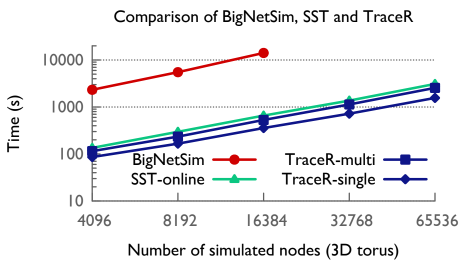


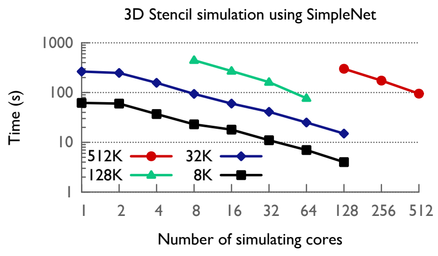
Fig. 5: Sequential simulation time

Startup Overhead in Parallel Mode: The overhead of reading traces in TraceR for the 3D Stencil code are as low as a few tens of seconds in most cases, especially on large core counts. This is because trace files can be read in parallel as the core counts increase. In general, we observe that the trace reading time is less than 5% of the total execution time.

6.2 Parallel Scaling and Validation of TraceR

Next, we present scaling results for TraceR using packet-level TorusNet and SimpleNet models. The comparison with other simulators was not possible due to the following reasons: 1) The parallel execution of BigSim, on 2-16 cores, is an order of magnitude slower than its sequential version. This is due to high overheads introduced by its PDES engine. 2) The parallel version of SST does not work with packet-level models, and hence is not available for a fair comparison.

Figure 6 presents the execution time for simulating 3D Stencil on various node counts of 3D torus. It is clear that TraceR scales well for all simulated system sizes. For the simulations with half a million (512K) nodes, the execution time is only 95s and 542s using SimpleNet and TorusNet, respectively. For smaller systems, the execution time is reduced to less than 100s, even for TorusNet.



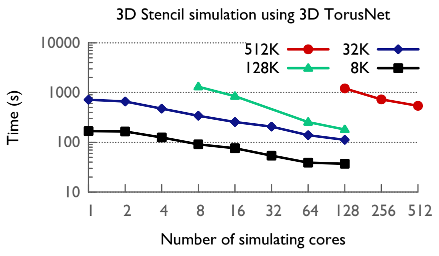
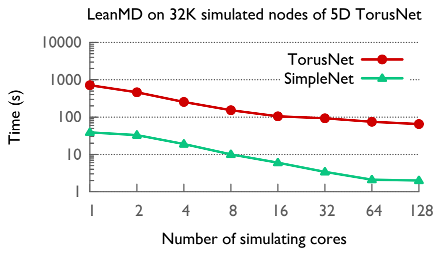


Fig. 6: Scalability of TRACER when simulating networks of various sizes.



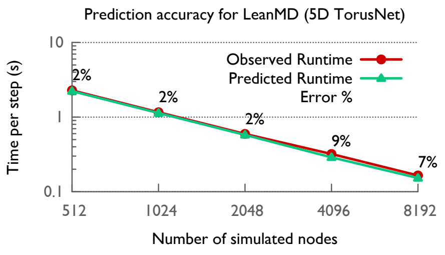


Fig. 7: Scaling and accuracy of simulating LeanMD with TRACER.

Figure 7 (left) shows the scaling behavior of TRACER when simulating LeanMD on 32K nodes of a 5D torus. Again, the simulation takes only 2s and 65s on 128 cores using SimpleNet and TorusNet, respectively. However, the speed up for simulation of LeanMD is lower in comparison to the speed up for simulating 3D Stencil. This is due to LeanMD's relatively more complicated interaction pattern and dependency graph, which causes more rollbacks on large core counts.

Validation: We simulated LeanMD on a 5D torus because we also used it to validate TRACER. IBM's Blue Gene/Q system, which has a 5D torus, provides an isolated execution environment which is ideal for comparison with the predicted performance from TRACER. For the validation, LeanMD was executed on Vulcan, a Blue Gene/Q with a 5D torus network. During these experiments, we enable deterministic routing, record the execution time and the topology of the system allocated for these jobs. Next, TRACER is configured to simulate LeanMD using a 5D TorusNet model and the topology information we obtained during the real runs. Figure 7 (right) compares the prediction by TRACER with the observed performance. We observe that for all node counts (512 to 8,192 nodes) the error in the prediction is less than 9%. For most cases, the predicted time is within 2% of the observed execution time. This proves that TRACER is able to predict the execution time of fairly complex application with a high accuracy.

7 Conclusion

We have presented a trace-driven simulator, TRACER, for studying communication performance of HPC applications on current and future interconnection networks. TRACER shows how one can leverage optimistic parallel discrete-event simulation with reversible computing to provide scalable performance. We have also shown that TRACER outperforms state-of-the-art simulators such as BigSim and SST in serial mode and significantly lowers the simulation time on large core counts. Additionally, we observed that depending on the simulated workload, the optimal set of input parameter values for a PDES-based simulator varies and needs to be identified.

References

1. Dumpi: The mpi trace library (2015), `http://sst.sandia.gov/about_dumpi.html`
2. Acun, B., et al.: Parallel Programming with Migratable Objects: Charm++ in Practice. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '14, ACM, New York, NY, USA (2014)
3. Bohm, S., Engelmann, C.: xsim: The extreme-scale simulator. HPCS (2011)
4. Carothers, C.D., Bauer, D., Pearce, S.: Ross: A high-performance, low-memory, modular time warp system. Journal of Parallel and Distributed Computing 62(11), 1648 – 1669 (2002)
5. Casanova, H., et al.: Versatile, scalable, and accurate simulation of distributed applications and platforms. Journal of Parallel and Distributed Computing (Jun 2014)
6. Cope, J., et al.: Codes: Enabling co-design of multilayer exascale storage architectures. In: Proceedings of the Workshop on Emerging Supercomputing Technologies (2011)
7. Girona, S., Labarta, J.: Sensitivity of performance prediction of message passing programs. The Journal of Supercomputing (2000)
8. Hoefler, T., Schneider, T., Lumsdaine, A.: LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. pp. 597–604. ACM (Jun 2010)
9. Janssen, C.L., et al.: A simulator for large-scale parallel computer architectures. IJDST 1(2), 57–73 (2010), `http://dblp.uni-trier.de/db/journals/ijdst/ijdst1.html#JanssenACKPEM10`
10. Jiang, N., et al.: A detailed and flexible cycle-accurate network-on-chip simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software (2013)
11. Lawlor, O., Bhandarkar, M., Kalé, L.V.: Adaptive mpi. Tech. Rep. 02-05, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign (2002)
12. Minkenberg, C., Rodriguez, G.: Trace-driven co-simulation of high-performance computing systems using omnet++. In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques. p. 65 (2009)
13. Misbah Mubarak, Christopher D. Carothers, R.B.R., Carns, P.: 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. pp. 27–38. ACM (2014)
14. Penoff, B., Wagner, A., Tuxen, M., Rungeler, I.: Mpi-netsim: A network simulation module for mpi. In: Parallel and Distributed Systems (ICPADS). IEEE (2009)
15. Wilmarth, T., Kalé, L.V.: Pose: Getting over grainsize in parallel discrete event simulation. In: 2004 International Conference on Parallel Processing. pp. 12–19 (August 2004)

16. Zheng, G., Wilmarth, T., Jagadishprasad, P., Kalé, L.V.: Simulation-based performance prediction for large parallel machines. In: International Journal of Parallel Programming. vol. 33, pp. 183–207 (2005)