Benchmark Generation and Simulation at Extreme Scale

Mahesh Lagadapati[1]
Dept. of Computer Science,
North Carolina State University,
Raleigh, NC 27695-7534,
Email: mlagada@ncsu.edu

Frank Mueller[1]
Dept. of Computer Science,
North Carolina State University,
Raleigh, NC 27695-7534,
Email: mueller@cs.ncsu.edu

Christian Engelmann[2,3]
Computer Science and Mathematics Division,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831-6016,
Email: engelmannc@ornl.gov

***Abstract*—The path to extreme scale high-performance computing (HPC) poses several challenges related to power, performance, resilience, productivity, programmability, data movement, and data management. Investigating the performance of parallel applications at scale on future architectures and the performance impact of different architectural choices is an important component of HPC hardware/software co-design. Simulations using models of future HPC systems and communication traces from applications running on existing HPC systems can offer an insight into the performance of future architectures. This work targets technology developed for scalable application tracing of communication events. It focuses on extreme-scale simulation of HPC applications and their communication behavior via lightweight parallel discrete event simulation for performance estimation and evaluation. Instead of simply replaying a trace within a simulator, this work promotes the generation of a benchmark from traces. This benchmark is subsequently exposed to simulation using models to reflect the performance characteristics of future-generation HPC systems. This technique provides a number of benefits, such as eliminating the data intensive trace replay and enabling simulations at different scales. The presented work features novel software co-design aspects, combining the ScalaTrace tool to generate scalable trace files, the ScalaBenchGen tool to generate the benchmark, and the xSim tool to assess the benchmark characteristics within a simulator.**

I. Introduction

This decade is projected to usher in the period of exascale computing with the advent of high-performance computing (HPC) systems of up to one billion tasks and possibly as many cores. To assess the requirements for future hardware platforms and to investigate the application/algorithm performance at extreme scale, hardware simulation plays an important role. Significant challenges exist even at the single node level, the network interconnect and at the system level when trying to orchestrate the execution of such extensive numbers of cores. Hardware simulators are vital in assessing the potential of different approaches under these challenges. Yet, these simulators need to be subjected to realistic application workloads that originate in the HPC realm.

The Extreme-scale Simulator (xSim) [2], [3], [4], [7], a performance investigation toolkit at extreme scale, can execute an application in a simulated HPC environment and thus facilitates HPC hardware/software co-design. To simulate an application, xSim needs to perform all the computation and communication involved in it. This, coupled with overhead to simulate network and processor models, makes simulation slow and necessitates that a significant amount of resources be allocated for simulation. The compounding effect of resources used by the application and the simulation, such as the additional network traffic to coordinate the simulation and memory accesses to facilitate the virtualization needed for the simulated environment, typically results in an overhead of several orders of magnitude. This overhead could be significantly reduced if computational overhead were only simulated as the time spent on computational resources, and not as the computation performed. This approach, however, would require the availability of skeleton benchmarks for communication at extreme scale derived from HPC application programs, which currently do not exist. This work tries to fill this gap by generating such communication benchmarks from scalable traces suitable for rapid replay within event-based simulators.

ScalaTrace [10], [12], [15] is a state-of-the-art tracing framework that produces near constant size communication traces for parallel applications based on the Message Passing Interface (MPI) irrespective of number of nodes. ScalaBenchGen [14] demonstrated that accurate and readable benchmarks can be generated from the traces of ScalaTrace. The generated benchmark is a normal parallel C program that issues MPI calls corresponding to every event in the trace. Experimental results have shown that the generated benchmark closely resembles that of the actual application, both in terms of execution time and communication messages exchanged. ScalaBenchGen [14] is based on the initial version of ScalaTrace [10], [12] that generates scalable traces for applications with SPMD (Single Program Multiple Data) behavior. The new version of ScalaTrace [15] (from here on referred as ScalaTrace II) redesigned the trace format such that scalable traces can be generated even for applications that exhibit irregular SPMD behavior,

[1]This work was supported in part by the U.S. National Science Foundation via grants 1217748, 0937908 and 0958311.

[2]This work was sponsored in partby the U.S. Department of Energy's Office of Advanced Scientific Computing Research and by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory.

which is essential for most large-scale HPC applications. To produce benchmarks from this trace format, we contribute novel algorithms for benchmark generation from elastic traces within *ScalaBenchGen II*.

To improve the simulation process, we co-design the benchmarks produced by ScalaBenchGen II to mesh with discrete event simulation tools, such as xSim. Simulating a benchmark program instead of the actual application improves the simulator performance in many ways. First, the payload exchange of an MPI event across native ranks can be avoided. In the benchmark program, a payload of an event is transmitted only to match with the communication behavior of the actual application, but is not utilized elsewhere in the program. xSim supports an *application model mode* in which the payload of an event is not transmitted. Instead, only metadata is exchanged. Still, this accurately captures the performance of events by updating the timing information within the simulated network model. Benchmarks can be simulated in this mode and, hence, huge data transfers can be avoided without loss of accuracy.

Second, computation can be completely virtualized in the simulator. The computation between two MPI events in the application is reproduced in the benchmark through a *sleep()* call with a duration equal to the time elapsed between them. xSim simulates a *sleep()* by simply advancing the simulated processor clock. Therefore, computation can be simulated with almost zero overhead, which otherwise adds significant overhead. xSim thus avoids excessive data exchanges and tedious computation. This enables xSim to gaze benchmarks for simulation and requires fewer resources, thus facilitating simulation at extreme scale.

Third, utilizing a skeleton benchmark instead of the original application obfuscates the application's implementation details to a degree that permits analyzing the performance characteristics of proprietary applications on future-generation HPC systems without revealing the implementation details.

Finally, the computational part can be enhanced to adapt to future architectures. ScalaBenchGen [14] showed that benchmarks generated on one platform can be adapted to another platform by adjusting the recorded overheads resembling the computational part.

The following are the main contributions of this work:

- We have designed and implemented ScalaBenchGen II, a tool to generate concise and accurate benchmark programs from the traces of ScalaTrace II by utilizing innovative algorithms to expand elastic traces.
- The ScalaBenchGen II and xSim capabilities have been combined in a software co-design approach to pave the way for HPC systems simulation at extreme scale.

II. Background

ScalaTrace [10], [12] produces scalable and often even constant size communication traces from MPI programs. These traces preserve both structural information and temporal ordering of events, and can be replayed to reproduce the original application behavior. It uses the MPI profiling layer (PMPI) to trace MPI functions and to record their parameters (such as source, destination, etc.) without actually recording the message content. It performs compression in two stages: *intra node* and *inter node*. Intra node compression is performed locally on each node on-the-fly. Loops in the program are identified and represented in the trace using extended regular section descriptors (RSD) [6]. Power-RSDs (PRSD) [9] are used to describe RSDs nested in a loop. Inter node compression is performed during *MPI_Finalize()* to produce a global trace file by combining the local trace records of each node. Another important feature of ScalaTrace is the preservation of the computation time of an application. It records the time elapsed between two communication events. Instead of exact values, statistical histogram bins are used to store delta times such that differences in execution time of an event across a loop or nodes is captured in the trace.

In a parallel program, an event can be executed in only some ranks or its parameters may vary with the rank. This information is captured in the trace by associating a *ranklist*, a list of ranks expressed in a recursive manner, to the event. A ranklist is encoded (in BNF notation) as: $dimension, start, (iters, stride)+$

where *dimension* indicates the number of pairs of the list, *start* is the starting node rank, and a sequence of (*iters, stride*) pairs represent the iteration count and stride, one each per dimension.

The original ScalaTrace tool compresses event sequences only if their call stack signatures, MPI parameters and loop structures match. The new version of ScalaTrace [15] (ScalaTrace II) is designed to generate scalable traces for applications that have inconsistent task level and loop level behavior. Event parameters, including loop information, are represented through a novel format, the *elastic data element representation.*

III. Framework Overview

Figure 1 outlines the process of generating a benchmark skeleton, and simulating it using xSim. First, the application is linked with the ScalaTrace library and executed to produce a trace file. The trace file is fed into the benchmark generator, which outputs a corresponding benchmark program. The benchmark generator can be run on a standalone machine. The generated benchmark is a normal *C* program that can be executed like any other parallel MPI program, reproducing the timing and communication behavior of the original application.

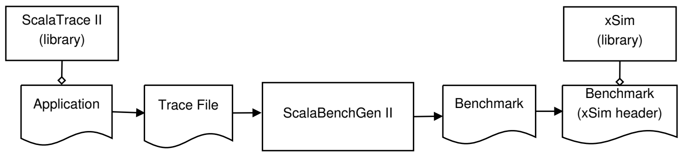


Fig. 1. Benchmark Generation and Simulation Process

As the generated benchmark is a normal *C* program, it can also be executed in the simulated environment of xSim like any other MPI application. xSim's header file is included, the benchmark is recompiled and linked against the simulator libraries. To perform studies using xSim, the simulated processor and network need to be configured. To accelerate these studies, xSim's *application model mode* (explained later) is utilized to remove unnecessary simulation overhead by transferring the MPI message envelope only, without the payload.

IV. SCALABENCHGEN II

ScalaBenchGen [14] generates communication benchmarks from the traces of ScalaTrace [10], [12]. Each event in the trace maintains MPI call parameters along with the time elapsed from the previous event. The benchmark program is a *C* program that issues MPI calls for each event in the trace. Computation between two events is abstracted by sleeping for a duration equal to the elapsed delta time since the previous event. This makes the wall clock time of both application and generated benchmark program closely resemble each other. Wrapper functions are designed for MPI events to hide the details of parsing the parameters and to invoke the actual MPI function with appropriate values. Retrieving communicators and generating request handles are also implemented in the wrapper functions. Thus, the generated benchmarks are both accurate and readable. A detailed description of the benchmark generator is presented in [14].

In ScalaTrace II, the trace format is completely redesigned to be able to capture more complex application patterns. Hence, ScalaBenchGen also requires a redesign including novel algorithms for expanding elastic representations to generate benchmark programs from the traces of ScalaTrace II. In this work, we describe ScalaBenchGen II, a tool to generate benchmarks from the traces of ScalaTrace II. The following sections contribute novel algorithms required for benchmark generation specifically for elastic traces.

A. Elastic Data Element Representation

ScalaTrace II introduced a new format, the *elastic data element representation*, to capture all data elements such as source, destination and count. An elastic data element is a list of $< valuevector, ranklist >$ pairs, where the former denotes a list of values for each iteration corresponding to the ranklist. This allows an event in the trace to assume different values of a parameter when values vary with the current rank or iteration of the enclosed loop. For instance, consider the scenario when an MPI_Send communicates 5 and 10 elements in two successive iterations of a loop for ranklist R1, but sends 15 and 20 elements for Ranklist R2. Then, the count parameter of the event is represented as: $\mathbf{5, 10 : R1; 15, 20 : R2}$

The benchmark generator passes elastic data elements of a parameter as a character string to the MPI event wrapper function. Inside the wrapper function, this string is parsed for the first time and an integer value vector is constructed. To avoid parsing the string every time, a value vector and the last accessed index are maintained in the benchmark code. Thus, ScalaBenchGen II can generate scalable benchmarks without much overhead by using the elastic data element representation.

B. Generating Benchmark Code for Loops

ScalaTrace identifies the loops in a program and represents them using the PRSD format [9]. Whenever a loop is encountered in the trace, ScalaBenchGen generates a *for* loop and places all members of the loop in it. ScalaTrace compresses repetitive events into a loop only when all events match across iterations. Loops across ranks are merged if and only if all its loop members are identical. ScalaTrace II redesigns the compression algorithms, such that a sequence of events is compressed as a loop even if the loop members differ across ranks or iterations. Loop information is also encoded using the elastic data element representation to achieve good compression. Hence, to generate benchmark code for loops, new techniques need to be introduced in the benchmark generator.

Inter-node compression in ScalaTrace II is loop structure agnostic in nature. The loop head maintains the information about a loop, *i.e.,* the number of members (events) and iterations, as elastic data elements. While merging two loop heads, loop information is compressed as elastic data elements such that the difference in loop structure across tasks is captured through the elastic data element representation. But this compression mechanism provides the following unusual properties to loop members, which makes the code generation process difficult:

- The number of events in a loop can be different per rank and iteration of an outer loop.
- An event can be part of a loop in one rank but may not be part of the loop in another rank.

```
Rank 0:                         Rank 1:
for(i=0;i<10;i++){              for(i=0;i<20;i++){
    E1;                             E1;
    E2;                             E2;
    E3;                         }
}                               E3;
```

Fig. 2. Rank specific behavior in loops

Figure 2 illustrates the aforementioned task-specific behavior in loops. Rank 0 has a loop of 10 iterations with three events: $E1, E2$, and $E3$. But Rank 1 has only two events, $E1$ and $E2$, in the loop with a trip count of 20. ScalaTrace II merges these two loops by annotating loop structure information with the loop head event, *i.e.,* $E1$. The trace for Figure 2 is $\mathbf{E1}_{(\mathbf{3},\mathbf{10})[\mathbf{0}],(\mathbf{2},\mathbf{20})[\mathbf{1}]}$ $\mathbf{E2}$ $\mathbf{E3}$, where the subscript of $E1$ indicates that rank 0 has a loop with 3 events and 10 iterations, whereas the loop in rank 1 has 2 events and 20 iterations.

ScalaBenchGen II uses the elastic data element representation to reflect any task specific behavior in loops. Whenever a loop is encountered in the trace, a *for* loop is placed in the benchmark code. Like other MPI data parameters, elastic data element strings are constructed for both iteration and member counts of a loop. They are parsed before executing the loop to determine the current iteration and member counts of the loop. The loop tail, *i.e.,* the last event in the trace that can be part of the loop across all ranks, will also be identified. All the events from loop head to loop tail are placed in the generated *for* loop. This may result in a placement of events in the loop such that certain events are executed in the loop only for a subset of ranks. For the example in Figure 2, events $E1, E2$, and $E3$ are placed in a loop, but $E3$ is not part of the loop for rank 1 and, hence, should only execute once. Thus, to guarantee the correctness of a program, a conditional check is placed before every loop member, $is\_loop\_member(rank, event\_ranklist, nesting\_depth)$. This predicate ensures that events are executed in their correct order and for the correct number of times. The predicate evaluates to *true* only if the following two conditions are satisfied. First, the current rank should be a member of the ranklist. Second, for a given iteration, the number of events executed so far should be less than the total member count of the loop, except for the last iteration. For instance, Figure 3 shows the simplified benchmark code for the example in Figure 2. Here, for rank 1, events $E1$ and $E2$ will be executed in all iterations of the loop but $E3$ will be executed only in the last iteration. However, for rank 0, all events are executed in each iteration of the loop.

```
iter_cnt = get_param('10,0;20,1');
mem_cnt = get_param('3,0;2,1');
for(iter=0; iter<iter_cnt; iter++){
    if(is_loop_member(...))
        E1;
    if(is_loop_member(...))
        E2;
    if(is_loop_member(...))
        E3;
}
```

Fig. 3. Benchmark code for the loop with rank specific behavior

C. Generating Benchmark Code for Computation

Each event in the trace records the time taken for computation, *i.e.,* the time elapsed in execution since the last event. The benchmark generator simulates this computation time by introducing a *sleep()* call before the event for the recorded time. An event can have multiple time records as its previous event can be different based on the rank and iteration of the loop. The initial version of ScalaBenchGen [14] places conditional checks such that the correct value is chosen for computation based on the entry path. This approach does not hold for the traces of ScalaTrace II. As described in sections IV-A and IV-B, ScalaTrace II aggressively compresses events even though their parameters and loop membership vary with the rank and iteration. This could lead to many entry paths for an event. For example, in the program shown in Figure 2, event $E1$ has an entry from $E3$ for rank 0, but it also has an another entry from $E2$ for rank 1. Compressing nested loops might increase entry paths even more. Placing conditional checks for every entry path increases the size of the generated benchmark program and reduces its readability.

ScalaBenchGen II avoids multiple conditional checks by maintaining the timing records of an event in the benchmark code itself. Each timing record of an event in the trace includes the previous event's stack signature along with the computation time. Instead of maintaining incomprehensible stack signatures, the benchmark generator annotates every event with a sequence number and uses it as a reference in the timing record. The number of the most recently executed event is maintained as a state in the benchmark code. A wrapper function, *do_compute()*, is placed before every event to parse the list of timing records and to simulate the required amount of computation based on the event that was executed previously. As the parsing logic is completely embedded in the wrapper function, the generated benchmarks can still be concise and readable, even if there are multiple entry paths. For the program given in Figure 2, the benchmark code for the computation at event $E1$ is:

```
times={{t(E2),E2},{t(E3),E3},{t(P),P}};
do_compute(times,...);
```

where the first field in the *times* array contains the computation time, and the second field indicates the corresponding previous event number. Here, P represents the event before the loop.

V. xSim

The Extreme-scale Simulator (xSim) [2], [3], [4], [7] is a performance investigation toolkit that permits running native HPC applications or proxy/mini applications in a controlled environment with millions of concurrent execution threads, called simulated MPI processes, while observing application performance in a simulated extreme-scale system for hardware/software co-design. Using a lightweight parallel discrete event simulation (PDES), xSim executes an application on a much smaller HPC system in a highly oversubscribed fashion with a virtual wall clock time, such that performance data can be extracted based on a processor and a network model with the appropriate simulation scalability/accuracy trade-off.

xSim is designed like a traditional performance tool, as an interposition library that sits between the MPI application and the MPI layer, using the MPI performance tool interface. It has essential support for simulated MPI point-to-point communication and full support for simulated MPI data types, groups, communicators, and collective communication. In total, xSim supports 88 simulated MPI functions for each supported programming language, *C* and *Fortran*. An application is run in the simulator using the following steps:

- Add *#include xsim-c.h* to the *C* source code, or *#include xsim-f.h* to the *Fortran* source code.
- Recompile the application and link it with the xSim library, *i.e.*, *-lxsim*, and the respective xSim programming language interface library, *i.e,* *-lxsim-c* for *C* or *-lxsim-f* for *Fortran*.
- Run the application with: *mpirun -np <real process cout> <application> -xsim-np <virtual process count> [other xSim arguments] [application arguments]*.

The PDES-driven simulation accounts for the execution time for each simulated MPI process using a processor model. It is based on the actual execution time on the real hardware platform scaled to the simulated processor speed. It does support heterogeneous processor cores with different speeds and operating system noise simulation. Calls by the application to *sleep()* and *usleep()* correspondingly advance the simulated process time. Calls to *gettimeofday()* perform a native execution time measurement and apply the processor model to return the absolute execution time since process start.

The simulation also accounts for the wait time incurred by communication for each simulated MPI process using a network model. It offers latency and bandwidth restrictions with different network architectures, such as star, ring, mesh, torus, twisted torus and tree. It also supports hierarchical combinations, such as to simulate network-on-chip and network-on-node, as well as, rendezvous protocols and sender/receiver process contention simulation. For scalability reasons, the network model does not provide full contention modeling for shared network interfaces or routers at this point.

For better simulation scalability, xSim supports the execution of application models in the *application model mode*. Similar to an MPI trace replay, application models, or the generated benchmarks in this paper, feed the simulator with the same timing and communication behavior, but do not require certain resources, such as memory and processor usage, to scale with the simulation. The simulated time is simply advanced according to the time the application would have spent between MPI calls on actual execution using the simulated *sleep()* and *usleep()* calls. Simulated MPI calls are executed without actually sending MPI message payloads, as the MPI message envelope containing the metadata is enough to accurately simulate communication behavior.

xSim has been run up to 134,217,728 (2^{27}) communicating MPI tasks, each with its own process context, using just a 960-core Linux-based cluster. The toolkit is relatively easy to use and the scalability/accuracy trade-off offered by xSim provides a unique opportunity for extreme-scale studies.

VI. Evaluation

The developed framework is evaluated on a cluster using a subset of the NAS parallel benchmarks [1] (version 3.3.1). These benchmarks are chosen as they perform both point-to-point and collective communication, which are prevalent in HPC applications. All experiments are conducted with 256, 512 and 1024 MPI ranks, except for SP and BT. As the number of MPI ranks needs to be a perfect square for these benchmarks, those two are executed with 64, 256 and 1024 MPI ranks. All benchmarks are executed with class D input problem sizes to produce sufficient computation and communication at this scale. Execution time metrics are reported as averages over three runs with a relative standard deviation of no more than 4.88% across all benchmarks. For the wall clock time taken for simulation, we observed relative standard deviations as high as 64% for outliers as simulation time can be more erratic for extremely short runs (less than 20 seconds), which is due to dominant system overheads for very short runtimes that tend to skew results. The average relative standard deviation (without outliers) across all benchmarks is only 8.36%, which is more meaningful and, most significantly, applies to longer runs. We also observed stability issues for simulation at 1024 MPI ranks with an oversubscription factor of 1 for certain benchmarks (MG, SP and BT), which result in lost heartbeats within Open MPI's runtime layer (ORTE). We reported the results of a single run in this case (just for simulation under xSim), and we are in the process of obtaining additional runs and of identifying the cause of these problems.

Initially, trace files are collected using ScalaTrace II by executing the selected NAS parallel benchmarks on the cluster. From these traces, corresponding benchmark skeletons are generated using ScalaBenchGen II. The generated programs are executed on the cluster and on xSim in a simulated platform that matches the configuration of the cluster. The simulation experiments using xSim are performed on the cluster as well. This methodology offers a good perspective on the performance and accuracy of the generated benchmark skeletons on the evaluation platform and on the simulated platform. It also offers insight into the capabilities of xSim using generated benchmark skeletons.

A. Evaluation Platform

The evaluation cluster consists of 108 compute nodes, each with 2 AMD Opteron 6128 (Magny Core) processors. Each processor has 8 compute cores, i.e., each node has 16 cores and 32 GB DRAM. The nodes are connected via InfiniBand with a fat tree network topology as follows: The 108 compute nodes are connected through 6 layer 1 (L1) switches and 3 layer 2 (L2) switches. Each L1 switch has 36 ports, where the lower 18 ports are connected to the compute nodes and the upper 18 ports are connected to L2 switches (6 per switch). For MPI communication, the raw InfiniBand (IB) or TCP over IB may be used. Table I shows a significant difference in effective bandwidth between both. The NAS parallel benchmarks and the generated skeleton benchmarks are executed over IB with Open MPI 1.5.4.

The simulator is configured to match the IB configuration. However, due to the requirement for multithreading in xSim, xSim itself is executed using the 8-times slower IB-over-TCP with Open MPI 1.5.4 and MPI multithreading support (since native IB does not support MPI multithreading). As xSim supports oversubscription, its performance is evaluated by increasing the oversubscription from a 1:1 ratio of physical-to-simulated MPI processes to a 1:16 ratio, *i.e.,* reducing the number of physical processes by half in each step.

TABLE I. IB vs. TCP-over-IB Effective Bandwidth

Type	Within Switch	Across Switches
InfiniBand (IB)	2547MB/sec	2546MB/sec
TCP over IB	308MB/sec	303MB/sec

B. Simulated Platform

xSim is configured to simulate the cluster's performance characteristics. As xSim's *application model mode* is used, the processor model is not needed. The generated benchmark skeletons simply call *usleep()* to advance the simulated MPI process clock for the execution time of computational phases. The network model uses a combination of hierarchical networks to simulate the on-node communication between cores at the lowest level, the off-node communication within the same switch at the middle level, and the off-node communication across different switches at the highest level. At each level of the hierarchy, a star network model is employed that matches the latency and bandwidth performance of the cluster. The simulation parameters are shown in Table II.

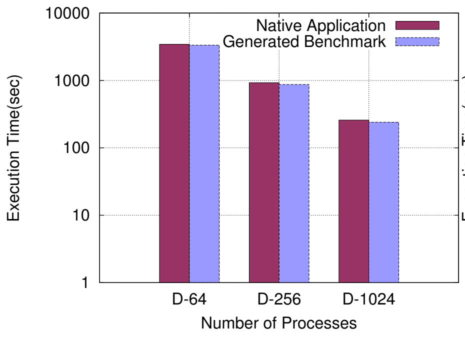
TABLE II. Cluster Network Parameters

Level	Latency	Bandwidth	Rendezvous Threshold
On-node	12 μs	1834 MB/sec	4 kB
Off-node (same switch)	13 μs	2547 MB/sec	64 kB
Off-node (diff. switch)	14 μs	2546 MB/sec	64 kB

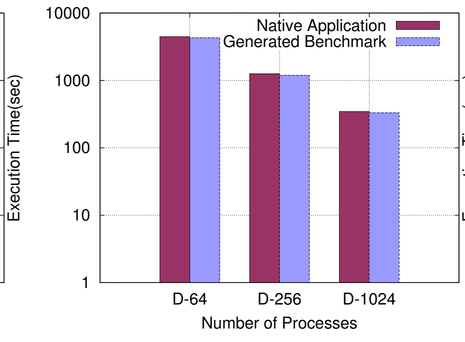
As noted earlier, communication contention is only simulated for each individual simulated MPI process, *i.e.,* a simulated MPI process cannot send and/or receive multiple MPI messages without incurring the required cost to send and/or receive them. However, xSim neither simulates network contention at the node interface nor at router level at this point. Furthermore, xSim's collective MPI calls are implemented in a linear fashion at the moment. There is no simulation support for tree-based collectives.

C. Generated Benchmarks Accuracy and Conciseness

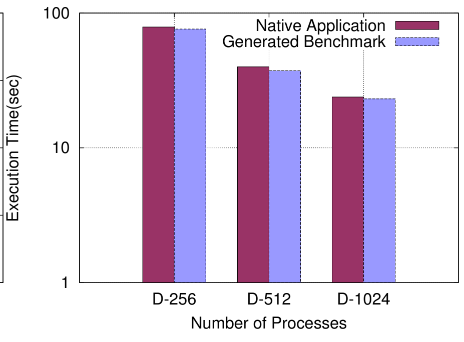
Figure 4 compares the wall clock execution times (on a logarithmic scale on the y-axis) of the benchmark programs (x-axis) generated by ScalaBenchGen II with the corresponding original applications. Both are executed on the same evaluation



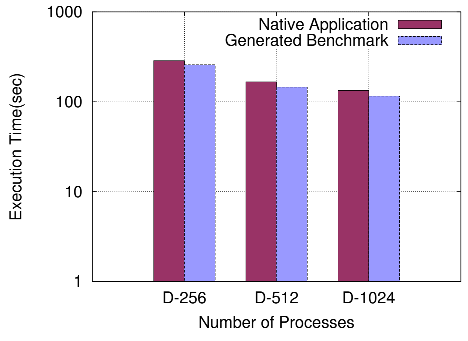
(a) BT



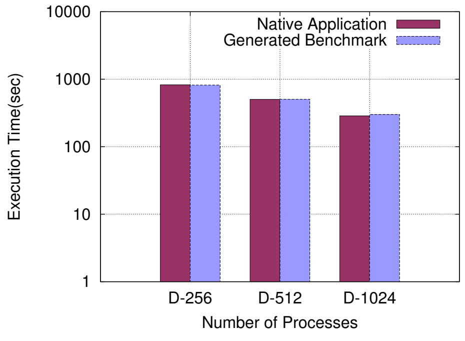
(b) SP



(c) MG



(d) CG



(e) LU

Fig. 4. Accuracy of ScalaBenchGen II Benchmarks

platform. From the graphs in Figure 4, we observe that the execution times of the generated benchmark and the corresponding application are close to each other. Quantitatively, the mean percentage error across the benchmarks is 5.5%, indicating a high accuracy. The maximum percentage error observed is 13.29%. We have calculated the percentage error using the formula:

$$|(T_{bm} - T_{app})|/T_{app} \times 100$$

where T_{bm} and T_{app} represent the execution time of benchmark program and the corresponding application, respectively.

Table III demonstrates the conciseness of the benchmarks produced by ScalaBenchGen II. The generated programs are smaller than the corresponding applications, less than half the size in most of the cases, even though these were benchmark programs to begin with. (More significant reductions in size would be obtained when automatically generating benchmarks from full-sized HPC applications.) The scalability column in Table III indicates how the number of lines of code changes with number of MPI ranks. The numbers of lines of code remain constant for all the benchmarks, irrespective of the number of MPI ranks, except for MG. MG performs a complex 7-point stencil communication that results in imperfect trace compression. However, the increase is only sub-linear with number of MPI ranks. Here, we reported the change in number of lines for 256 to 1024 ranks. Most significantly, BT and CG show a constant size irrespective of the number of ranks, which could only be obtained by benchmark generation with ScalaBenchGen II from the elastic trace format (cf. prior sub-linear results for these benchmarks in [10]).

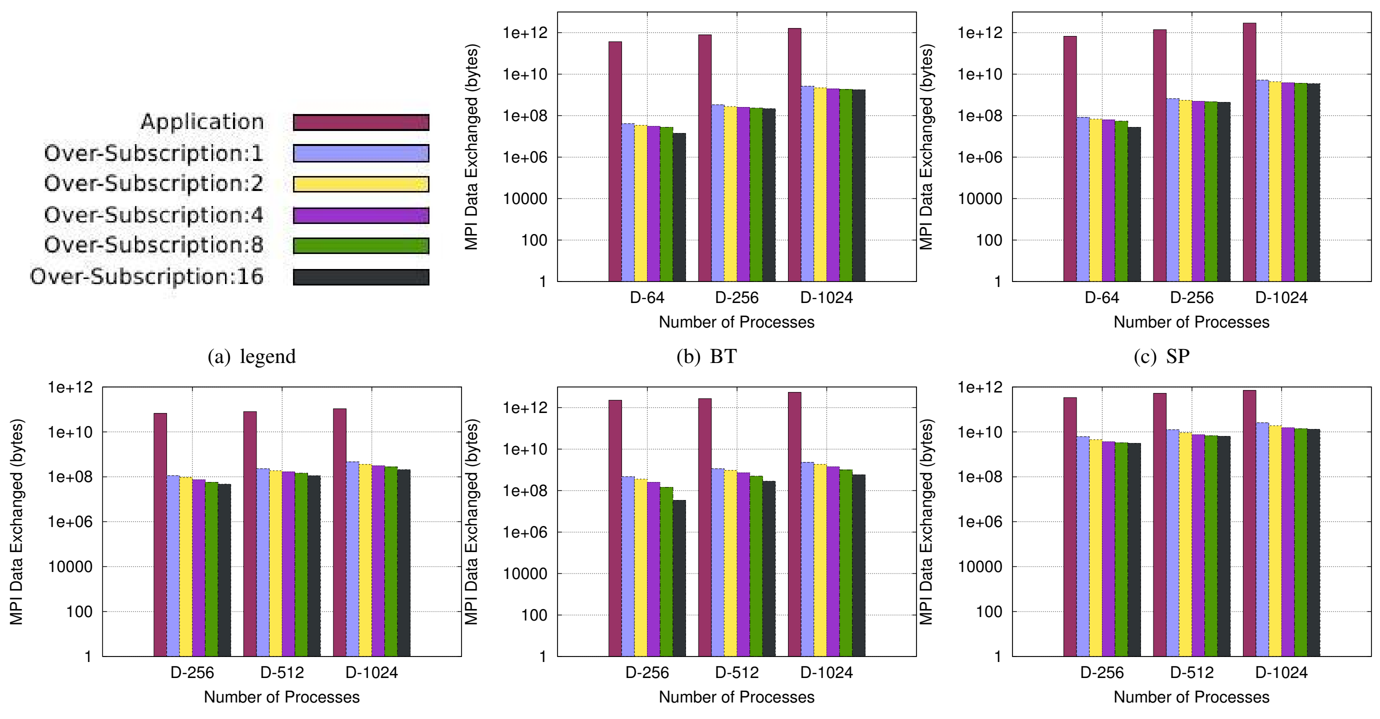
TABLE III. COMPARISON OF NUMBER OF LINES OF CODE

Program	Application	Benchmark	Reduction (%)	Scalability
BT	9383	1076	88.53	constant
SP	5067	1022	79.83	constant
CG	1918	954	50.26	constant
LU	6024	1298	78.45	constant
MG	2678	2496· · · 2888	-7.84· · · 6.8	sub-linear

D. Simulation Performance and Accuracy

We evaluated the effectiveness of our simulation approach using two metrics: the amount of MPI data communicated during simulation and the time taken to complete the simulation. Figure 5 presents the amount of MPI data communicated (on a log scale on the y-axis) under the simulation for the different generated skeleton benchmarks (x-axis). Note that the generated skeleton benchmarks communicate the same amount of MPI data as the corresponding NAS benchmarks when not using xSim for simulation. We observe that MPI data transmitted during simulation is 2-4 orders of magnitude smaller than the amount of data communicated in the actual application. This is due to the fact that the MPI payload is not transmitted during simulation. In all benchmarks, a total of only a few giga bytes are communicated during simulation, even though the MPI data exchanged in native application execution is in the order of tera bytes. Also, the amount of transmitted MPI data decreases with increasing oversubscription, because xSim does not need to send MPI messages with corresponding metadata for simulated MPI processes that reside within the same physical MPI process (using user-space threading). Figure 6 shows the effect of decreasing MPI message counts (on a log scale) over increasing oversubscription. We observe a reduction of MPI messages in the order of millions as we increase the oversubscription factor.

Figure 7 presents the total wall clock time elapsed to perform the simulation (on a log scale) with different oversubscription factors. We observe an initial decrease of simulation



(a) legend (b) BT (c) SP

(d) MG (e) CG (f) LU

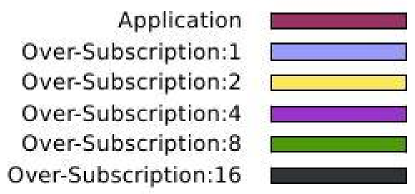
Fig. 5. MPI Data Exchanged for Different Over-Subscription Scenarios

time with increasing oversubscription and an eventual increase again in most of the scenarios. We have shown in Figures 5 and 6 that an increase of oversubscription will lower the number of MPI messages and the amount of MPI data that is communicated. However, more simulated MPI processes reside on the same physical MPI process with higher oversubscription, resulting in additional overheads for context switching between them. Each individual simulated MPI message that is received by a simulated MPI process requires the simulator to switch to the context of the receiving simulated MPI process. While the context switch itself consists of a jump in the user-space stack and a copy-out/in of the data segments, it is not necessarily by itself a huge performance issue. The context switch, however, may trigger cache pollution. In addition, the network model is applied for each simulated MPI message whether it is sent to a locally residing simulated MPI process or to a remote one, incurring a small computational overhead. Programs with high message volumes, such as LU with over 100 million messages, create a noticeable overhead by forcing context switches and network model computations. Figure 7 further shows an outlier. The simulation of MG at 1024 MPI ranks takes longer than the application independent of oversubscription, while it is shorter for 256 and 512 MPI ranks. It appears that its 7-point stencil communication pattern is causing performance degradation in xSim's MPI message processing that is compounded at larger scales.

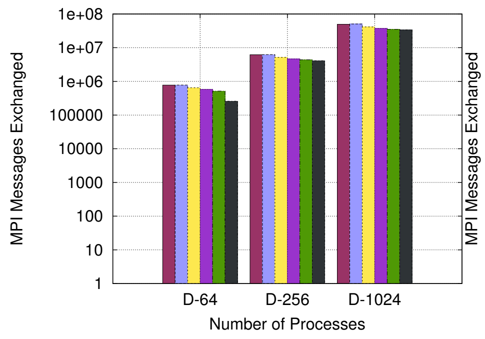
Figure 8 presents the comparison of the simulated benchmark skeleton execution time with the corresponding application execution time. The mean percentage error across the simulated benchmark skeletons is 22.41%. Both times are decreasing with an increase in the number of MPI ranks. Hence, even though the simulated benchmark skeleton's time is not highly accurate, it follows the same scaling pattern as the original application. This demonstrates that the presented approach permits assessing the performance characteristics of applications using generated benchmark skeletons in a simulated HPC system of this size.

The simulation accuracy is affected by a number of factors. xSim does not model network interface and router contention, which are likely the biggest influence. Furthermore, MPI collectives are simulated in a linear fashion, which may not reflect the algorithms used internally by the MPI implementation at scale. Also, xSim executes the simulated benchmark skeleton in these experiments in an entirely clean environment, without simulated operating system (OS) noise and without any simulated local contention within the MPI layer for matching messages during receives. This results in lower execution times for the simulated benchmark skeletons, especially when they communicate a large amount of data for which the network contention is not modeled. For instance, we observed maximum error of 54% for CG for 1024 MPI ranks in our experiments, where nearly 5.5 TB of data are transferred. As an extension to this work, a probabilistic approach will be designed to simulate network contention by changing network bandwidth following a normal distribution. A less scalable but accurate network contention model is planned as well.

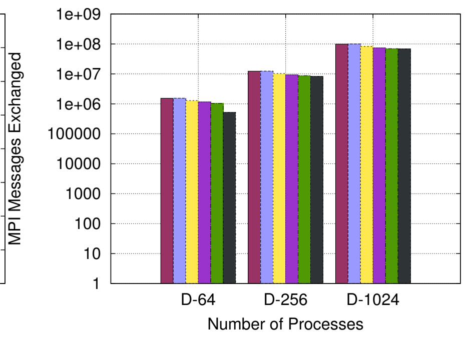
Our methodology can also be utilized to obtain a trace on one platform and then simulate another platform as follows. Processor/memory/network characteristics are provided from configurations. Even non-linear differences to the original timings can be modeled. This would require, e.g., tracing to obtain PAPI cache metrics to allow non-linear changes for different cache sizes.



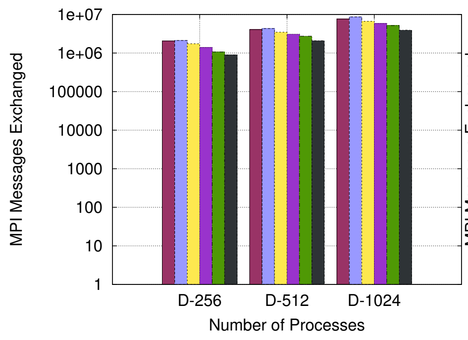
(a) legend



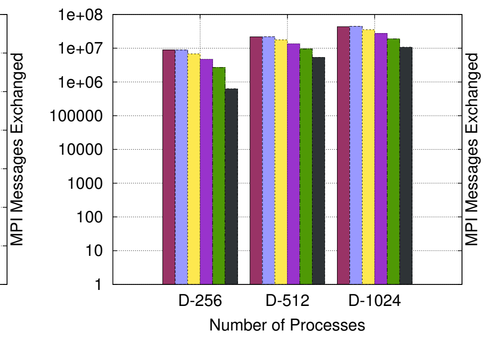
(b) BT



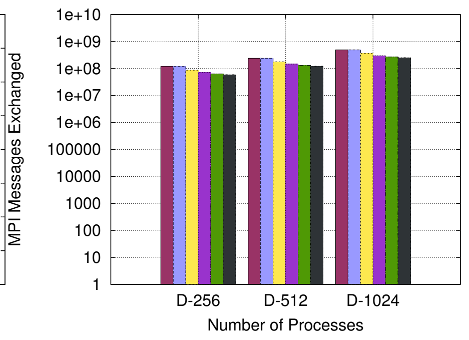
(c) SP



(d) MG



(e) CG



(f) LU

Fig. 6. MPI Messages Exchanged for Different Over-Subscription Scenarios

VII. Related Work

In the context of the presented research, related work focuses on MPI performance tools that extract and analyze MPI traces and simulation tools that execute applications or trace replays in a simulated HPC system.

Our benchmark generator framework, ScalaBenchGen II, utilizes traces produced by ScalaTrace II that are constant in size irrespective of number of MPI ranks. In contrast to our approach, the size of the traces produced by traditional tracing tools, such as PARAVER [11] and Vampir [8], increases linearly with number of MPI ranks.

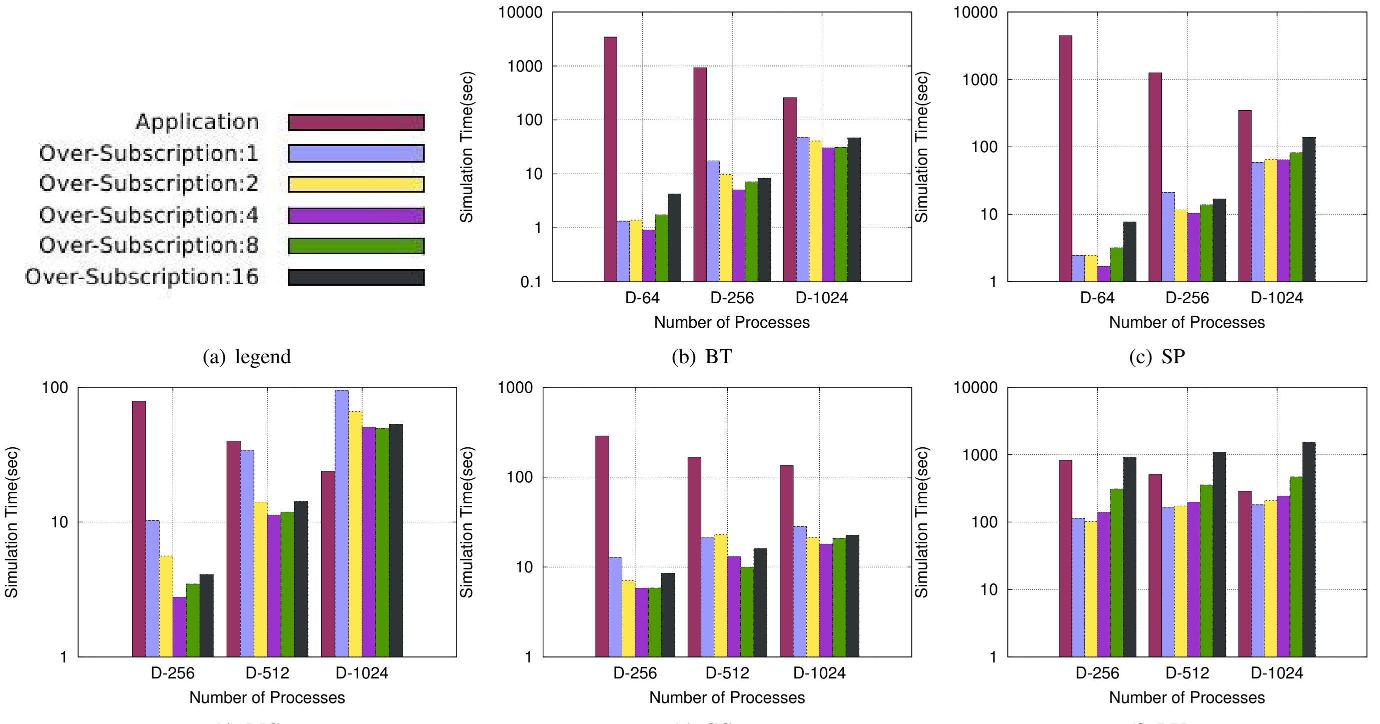
Xu et al. [16], [17] proposed a framework to construct *performance skeletons* to evaluate the performance of corresponding application on a future HPC system. But MPI events that are not part of a globally dominant communication pattern are not included in performance skeletons. Hence, communication behavior of these skeletons may differ from the original application, unlike benchmarks produced by ScalaBenchGen II.

The Structural Simulation Toolkit (SST) [13] offers simulation of novel compute-node architectures of future-generation HPC systems. It scales to a few hundred nodes with different levels of accuracy, utilizing external modeling and simulation tools. It can generate traces for larger-scale system simulations. SST/macro is a complementary simulation toolkit that can process output from the MPI tracing library DUMPI (http://sst.sandia.gov/about_dumpi.html) for performance evaluation. SST and SST/macro enable a synergy between small-scale cycle-accurate and large-scale communication-accurate simulations by generating traces at smaller scale and extrapolating performance at extreme scale. The solution presented in this paper offers a more practical, versatile and efficient approach as skeleton benchmarks are generated and executed in a simulation environment. The data-intensive trace replay is not necessary and, in contrast to running real applications, there is no need for computation or communication workload within the simulation. Furthermore, application code itself is obfuscated, permitting analysis of the performance characteristics for proprietary applications.

Other trace-driven PDES solutions exist for performance estimation of HPC applications on future-generation HPC systems. For example, DIMEMAS [5] is a simulation tool that processes MPIDTrace traces acquired from HPC applications and generates output for performance tools, such as PARAVER [11] and Vampir [8]. Performance estimations of architectural changes can be simulated and visualized. While the presented solution does not feature trace output and visualization, it does offer simulation-based performance estimation without the overhead of trace-driven simulation.

VIII. Conclusion

This work has demonstrated the capability to utilize benchmarks generated from ScalaBenchGen II to drive HPC architectural simulations. The ScalaTrace II framework is used to produce near lossless scalable and elastic communication traces. The resulting traces are transformed by ScalaBenchGen II into a benchmark code, which is of constant size for most benchmarks irrespective of the number of nodes and sub-linear for one benchmark, benefiting from novel benchmark generation algorithms from elastic traces. This code is subsequently fed into xSim to run the benchmark within a simulated environment. Simulating generated benchmarks instead of the original application enables xSim to avoid the



(a) legend (b) BT (c) SP

(d) MG (e) CG (f) LU

Fig. 7. Simulation Time for Different Over-Subscription Scenarios

payload exchange of an MPI event and virtualizes computation completely. Experimental results showed that with benchmark skeletons, the total amount of MPI data communicated in simulation is reduced by several orders of magnitude, and the time taken for simulation is close to the application's native execution time.

The work differs from prior ScalaTrace and xSim papers in that it is the first description of challenges due to HPC simulation when *compressed* traces are used, which is an important contribution to make extreme-scale systems simulations more feasible. This may become even more important in the future as such simulation can be derived from traces of real benchmarks, which can then be extrapolated and analyzed in a simulator at different levels of *granularity*. Finding the right granularity is the challenge, as this paper also shows. xSim was *adapted* (see Section 5) to better reflect the details of trace information, a necessary step for subsequently determining the right level of detail in simulation.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1991. [Online]. Available: citeseer.ist.psu.edu/article/bailey94nas.html

[2] S. Böhm and C. Engelmann, "xSim: The extreme-scale simulator," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS) 2011*. Istanbul, Turkey: IEEE Computer Society, Los Alamitos, CA, USA, Jul. 4-8, 2011, pp. 280–286, acceptance rate 28.1% (48/171). [Online]. Available: http://www.christian-engelmann.info/publications/boehm11xsim.pdf

[3] C. Engelmann, "Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale," *Future Generation Computer Systems (FGCS)*, vol. 30, no. 0, pp. 59–65, Jan. 2014. [Online]. Available: http://www.christian-engelmann.info/publications/engelmann13scaling.pdf

[4] C. Engelmann and F. Lauer, "Facilitating co-design for extreme-scale systems through lightweight simulation," in *Proceedings of the* 12^{th} *IEEE International Conference on Cluster Computing (Cluster) 2010:* 1^{st} *Workshop on Application/Architecture Co-design for Extreme-scale Computing (AACEC)*. Hersonissos, Crete, Greece: IEEE Computer Society, Los Alamitos, CA, USA, Sep. 20-24, 2010, pp. 1–8. [Online]. Available: http://www.christian-engelmann.info/publications/engelmann10facilitating.pdf

[5] S. Girona, J. Labarta, and R. M. Badia, ""Validation of dimemas communication model for MPI collective operations"," in *Lecture Notes in Computer Science: Proceedings of the* 7^{th} *European PVM/MPI Users' Group Meeting (EuroPVM/MPI) 2000*, vol. 1908, Sep. 10-13 2000, pp. 39–46.

[6] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, Jul. 1991.

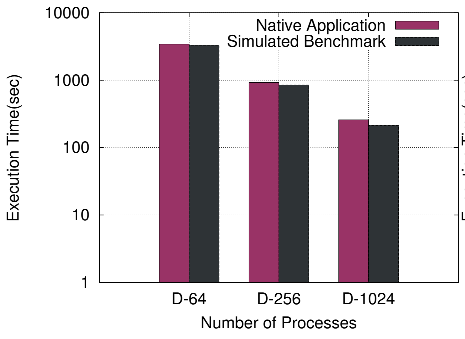
[7] I. S. Jones and C. Engelmann, "Simulation of large-scale HPC architectures," in *Proceedings of the* 40^{th} *International Conference on Parallel Processing (ICPP) 2011:* 2^{nd} *International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*. Taipei, Taiwan: IEEE Computer Society, Los Alamitos, CA, USA, Sep. 13-19, 2011, pp. 447–456. [Online]. Available: http://www.christian-engelmann.info/publications/jones11simulation.pdf

[8] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis toolset," in *Tools for High Performance Computing*, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer Berlin Heidelberg, 2008, pp. 139–155.

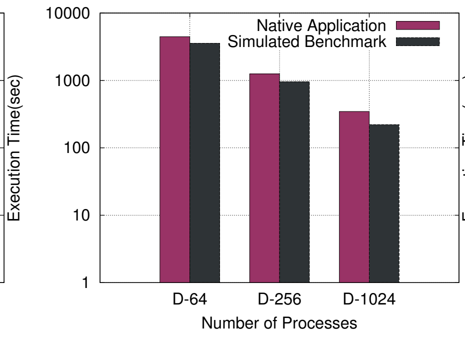
[9] J. Marathe and F. Mueller, "Detecting memory performance bottlenecks via binary rewriting," in *Workshop on Binary Translation*, Sep. 2002.

[10] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces in high performance computing," *Journal of Parallel Distributed Computing*, vol. 69, no. 8, pp. 969–710, Aug. 2009.

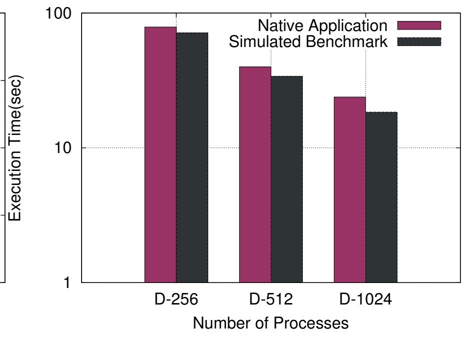
[11] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A Tool to



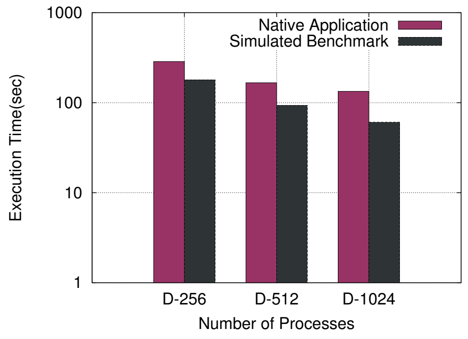
(a) BT



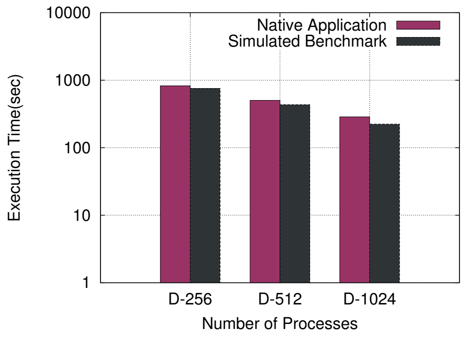
(b) SP



(c) MG



(d) CG



(e) LU

Fig. 8. Accuracy of Simulation

Visualize and Analyze Parallel Code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, mar 1995, pp. 17–31.

[12] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *International Conference on Supercomputing*, Jun. 2008, pp. 46–55.

[13] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1964218.1964225

[14] X. Wu, V. Deshpande, and F. Mueller, "Scalabenchgen: Auto-generation of communication benchmark traces," in *International Parallel and Distributed Processing Symposium*, Apr. 2012.

[15] X. Wu and F. Mueller, "Elastic and scalable tracing and accurate replay of non-deterministic events," in *International Conference on Supercomputing*, Jun. 2013, pp. 59–68.

[16] Q. Xu, R. Prithivathi, J. Subhlok, and R. Zheng, "Logicalization of mpi communication traces," Dept. of Computer Science, University of Houston, Tech. Rep. UH-CS-08-07, 2008.

[17] Q. Xu and J. Subhlok, "Construction and evaluation of coordinated performance skeletons," in *International Conference on High Performance Computing*, 2008, pp. 73–86.