Auto-Generation of Communication Benchmark Traces *

Vivek Deshpande
North Carolina State University
Raleigh, NC, USA
vrdeshpa@ncsu.edu

Xing Wu
North Carolina State University
Raleigh, NC, USA
xwu3@ncsu.edu

Frank Mueller
North Carolina State University
Raleigh, NC, USA
mueller@cs.ncsu.edu

ABSTRACT

Benchmarks are essential for evaluating HPC hardware and software for petascale machines and beyond. But benchmark creation is a tedious manual process. As a result, benchmarks tend to lag behind the development of complex scientific codes.

Our work automates the creation of communication benchmarks. Given an MPI application, we utilize ScalaTrace, a lossless and scalable framework to trace communication operations and execution time while abstracting away the computations. A single trace file that reflects the behavior of all nodes is subsequently expanded to C source code by a novel code generator. This resulting benchmark code is compact, portable, human-readable, and accurately reflects the original application's communication characteristics and performance. Experimental results demonstrate that generated source code of benchmarks preserves both the communication patterns and the run-time behavior of the original application. Such automatically generated benchmarks not only shorten the transition from application development to benchmark extraction but also facilitate code obfuscation, which is essential for benchmark extraction from commercial and restricted applications.

1. INTRODUCTION

Benchmarks are widely used for evaluating and analyzing system performance and assessing migration costs of HPC applications to new platforms with different architectures. They are easy to port, modify and run, and they closely resemble the characteristics of HPC applications. But most benchmarks do not capture the complexity and scale of realistic HPC applications as they do not feature the intricate interplay of computation, communication and I/O operations.

We generate communication benchmarks in an *automated* approach. These benchmarks are human readable, compact, easy to generate and port. They closely resemble the execution time and communication volume of the original application.

As an input, we take an HPC application with message passing communication using MPI (Message Passing Interface) [6]. The application's communication patterns are captured in traces. The obtained trace is given as an input to the benchmark generator, which is the central focus of this work. It outputs the communication benchmark in C code (including MPI calls for communication) that can be executed on target machine, as illustrated in Figure 1.

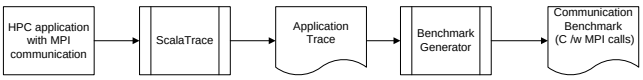

Figure 1: Benchmark Generation System - Block Diagram

We utilize ScalaTrace [11] for communication trace collection. ScalaTrace is a unique approach to parallel application tracing as this scalable framework captures the communication in lossless and near constant size in terms of trace representation independent of the number of the nodes while preserving the structural information of the nodes and iterations. It also employs a pattern based intra-node and inter-node compression techniques extracting the application's communication structure.

We evaluate our communication benchmark generator using the NAS Parallel Benchmark Suite [2] and Sweep3D [19]. We show that auto-generated benchmarks preserve the application's semantics in terms of their communication pattern along with communication volume and the ordering of events relative to the original HPC application. Furthermore, the overall execution time of benchmarks is close to that of their original applications. Thus, the communication benchmark generator is able to generate benchmarks that are similar to the original application in terms of communication behavior and execution time.

The contributions of this work are (1) a demonstration and evaluation of the feasibility of automatically converting parallel applications into human-readable benchmark codes and (2) an approach and algorithm for resembling the original performance by generating benchmarks from communication traces.

Our work benefits application developers, communication researchers, and HPC system designers. Application developers can benefit in multiple ways. First, they can quickly gauge the application performance of a target machine before investing in the effort to port their applications to that machine. Second, they can use the generated benchmarks for performance debugging as the benchmarks can separate communication from computation to help isolate observed performance anomalies. Third, application developers can examine the impact of alternative application implementations such as different data decompositions (causing different communication patterns) or the use of computational accel-

*This work was supported in part by NSF grants 1058779, 0958311, 0937908. It used resources of the National Center for Computational Sciences at ORNL and was performed in part at ORNL, managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

erators (reducing computation time without directly affecting communication time). Communication researchers can benefit by being able to study the impact of novel messaging techniques without the need to build complex applications and without access to source code that is not freely distributed or even classified. Finally, procurement of HPC systems can benefit by contracting vendors to deliver a specified performance on a given auto-generated benchmark without having to provide those vendors with the actual application.

In summary, we have developed a tool that automatically generates the communication benchmark C code with MPI calls from HPC applications such that the characteristics of the original application are preserved in terms of time and structure. The generated code is human readable, compact, easy to compute and portable.

2. BACKGROUND

Our work builds on ScalaTrace, an MPI tracing toolkit with aggressive and scalable trace compression. ScalaTrace's compression can result in trace file sizes orders of magnitude smaller than previous approaches or, in some cases, even near constant size regardless of the number of nodes or application run time [11].

The tool collects communication traces using the MPI Profiling layer (PMPI) [1] through Umpire [18] to intercept MPI calls during application execution. On each node, profiling wrappers trace all MPI functions, recording their call parameters, such as source and destination of communications, but without recording the actual message content.

ScalaTrace performs two types of compression: $intra-node$ and $inter-node$. For the intra node compression, the repetitive nature of timestep simulation in parallel scientific applications is used. Intra-node compression is performed on-the-fly within a node. Further, the inter-node merge exploits the homogeneity in behavior (SPMD) among different processes running the application. Inter-node compression is performed across nodes by forming a radix tree structure among all nodes and sending all intra-node compressed traces to respective parents in the radix tree. At the parent, the respective trace representations are merged, reduced and then compressed exploiting domain-specific properties of MPI. This results in a single compressed trace file capturing the entire application execution across all nodes. The compression algorithms are discussed in detail in other papers [12, 15].

ScalaTrace achieves near constant size traces by applying pattern based compression. It uses extended regular section descriptors (RSD) to record the participating nodes and parameter values of multiple calls to a single MPI routine in the source code across loop iterations and nodes in a compressed manner [7]. Power-RSDs (PRSD) recursively specify RSDs nested in a loop [10].

Example: Consider the code snippet shown in Figure 2 with ring-style communication across N nodes.

```
for(i=0; i<100; i++){
   MPI_Irecv(LEFT, ...);
   MPI_Isend(RIGHT, ...);
   MPI_Waitall(...);
}
```

Figure 2: Ring-style MPI Communication Code

ScalaTrace represents these events as three RSDs in the trace (see Figure 3) to denote the non-blocking send, receive and waitall MPI operations of a single loop iteration, where $\langle rank \rangle$ represents a value within $0 \ldots N-1$ in each per-node trace. ScalaTrace then detects the loop structure and outputs a single PRSD to denote a single loop of 100 iterations. This intra-node compression is performed on-the-fly to reduce the time for trace generation and the memory overhead.

RSD1: {$\langle rank \rangle$, MPI_Irecv, LEFT}
RSD2: {$\langle rank \rangle$, MPI_Isend, RIGHT}
RSD3: {$\langle rank \rangle$, MPI_Waitall}
PRSD: {100, RSD1, RSD2, RSD3}

Figure 3: Intra-node Compressed Trace

Further, during the inter-node compression, the local traces on each node are combined into a single global trace when the application is terminates (i.e., within the PMPI interposition wrapper for MPI_Finalize). Inter-node compression detects similarities among the per-nodes traces and merges the RSDs by combining their participant lists in a final participant list. For the example above, each MPI routine is called on each node with same parameters resulting in the following inter-node trace depicted in Figure 4.

RSD1: {$0, 1, \ldots, N-1$, MPI_Irecv, LEFT}
RSD2: {$0, 1, \ldots, N-1$, MPI_Isend, RIGHT}
RSD3: {$0, 1, \ldots, N-1$, MPI_Waitall}

Figure 4: Inter-Node Compressed Trace

The participant node information is encoded and represented in a fixed-sized tuple containing starting rank, total number of participants and an offset value separating ranks. Even multi-dimensional information is captured in this encoding format. There are special cases in which events with matching calling context can have non-matching function parameters. These non-matching function parameters are compressed using a vector representation so that the particular event can be concisely represented in the trace.

Another important feature of ScalaTrace is the time preservation of captured traces. Instead of recording absolute timestamps, the tool records delta time of computation durations between adjacent communication calls. During RSD formation, instead of accumulating exact delta timestamps, statistical histogram bins are utilized to concisely represent timing details across the loop. These bins are comprised of statistical timing data (minimum, maximum, average and standard deviation). ScalaTrace records histograms of delta times for each instance of a particular computation, i.e., distinguishing disjoint call paths by separate histogram instances.

3. BENCHMARK GENERATOR DESIGN

We next describe our trace-based benchmark generation approach. Our goal is to generate benchmark code that is compact, portable, human-readable, and accurately reflects the original application's communication characteristics and performance. This is made possible by using the timing information for the computation regions in the application trace, as opposed to the compiler-based approaches that lack

the runtime information. Notice that the benchmark generated with our approach can only be executed with the same number of MPI processes with which the trace was collected.

The process of automatic benchmark source code generation from communication traces is accomplished by traversing through the trace of a parallel application obtained from ScalaTrace. The trace traversal framework is designed to walk through all the RSDs and PRSDs. For each RSD and PRSD, the code generator is invoked to generate the respective C code and MPI calls. The code generator uses the predefined interfaces provided by the traversal framework, making the code generator a pluggable module. Thus, the same platform can be used to generate the code for different languages by writing code generators for those languages providing flexibility in generating code beyond C.

The RSDs that represent point-to-point communication are converted to respective point-to-point MPI calls in C code. For example, blocking sends and receives are transformed to MPI_Send and MPI_Recv and non-blocking ones are transformed to MPI_Isend and MPI_Irecv. Collective calls are generated using MPI collective routines in C such as MPI_Barrier, MPI_Reduce, MPI_Alltoall and so on. The communicator-based MPI events are converted to the respective routines such as MPI_Comm_split and MPI_Comm_dup. PRSDs representing loops are converted to C-style *for* loops. Behavioral constraints captured by traces are imposed in the generated code using conditionals on loop index variables and on ranks of the processes participating in a particular event.

The generator takes a single trace file as input and generates a set of files that comprise the auto-generated benchmarks. This is depicted in Figure 5. This includes a C source file with MPI calls for the communication benchmark, skeleton code and type (C code) plus a header file, which contains MPI events of communication patterns, support functions and variable declarations to assist RSD/PRSD traversal. The header file is dynamically expanded during the traversal as a need for new variables is encountered (e.g., communicator split or duplication). An additional rank file records node participants in communication events (plus an optional file to record MPI_Alltoallv parameters, if used).

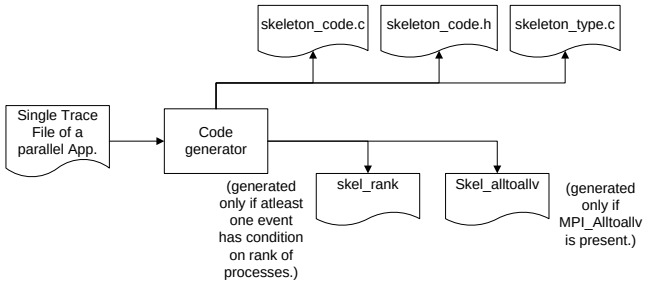

Figure 5: The Composition of Files in a Generated Code

The framework is designed in such a way that it can be used to both replay the trace and generate the code. While generating the code, the trace is traversed only once, including the part of the trace which belongs to PRSDs (iterations). While traversing, the RSDs and PRSDs are parsed and MPI events are generated one at a time, e.g., MPI_Send. Each parameter of the trace is expanded accordingly. The auto-generator maintains high watermarks for certain parameters, such as count (message length), and it creates declaration for each unique data type (including derived data types), communicator and other parameters. Upon termination, declarations are emitted to the header file, e.g., for receive buffer sizes shared across MPI calls for the maximum message length encountered.

During traversal, we also generate delta times for the computational regions and store them in trace events of the subsequent communication event. In a compressed trace, delta times before the leading event of nested loops are represented by a list of histograms to distinguish between different execution paths. During code generation, conditionals on loop iterator variables are generated so that different execution paths will lead to different sleep times. Histograms contain the statistical information, e.g, minimum, maximum, and mean of the recorded delta times. Currently, we use the mean to generate sleep statements. For applications exhibiting heavy load imbalance, more fine-grained approaches can be used to generate sleeps. For example, the distribution information captured by histogram bins can be used to preserve the unbalanced program behavior. Particularly, rank lists can be attached to histogram bins so that outliers will be captured and imbalance will not be averaged out.

By parsing a single PRSD, we generate C-style *for* loops. For example, consider the tuples of RSD1 and RSD2 in Figure 6. They capture MPI_Recv and MPI_Send calls in the respective RSDs. PRSD1 denotes a loop of 10 iterations each with RSD1 and RSD2. PRSD2 denotes a loop of PRSD1 followed by a broadcast.

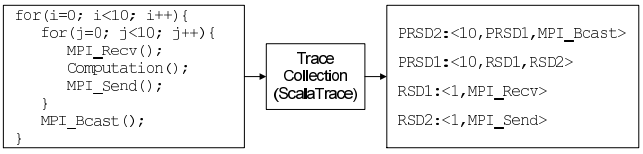

Figure 6: The Trace Collection

This repetitive nature of PRSDs to capture the nesting of loops is exploited in the benchmark code generation. The traces are stored as linked lists of RSDs/PRSDs. The queue in the form of linked lists is traversed to generate the MPI events. The traversal function calls itself recursively whenever a nested loop is found. The nesting depth is tracked and used in generating the index for the loop.

An example is given in Figure 7, where nested PRSDs generated from trace of Figure 6 are auto-expanded to source code interspersed by delta-time sleeps resembling computation of the original application.

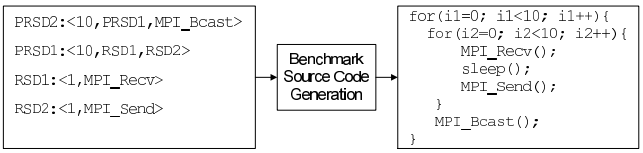

Figure 7: The Code Generation

MPI_Comm_split and MPI_Comm_dup events create new communicators. These communicator names are declared in the header file similar to other variables with the data type

MPI_Comm. These communicators are dynamically created during the process of code generation. The color and key in the trace of split communicators are used to generate MPI_Comm_split calls. Color and key could be absolute or could be an offset. If an offset is encountered then the offset is added to *myrank*.

MPI_Isend and MPI_Irecv generate request handles. Subsequently, MPI_Wait and MPI_Waitall use those handles to block the processes until the sending or receiving is complete.

During benchmark execution, a request handle is added to a ring buffer containing pointers to requests. An index into this buffer signifies the location of the request pointer for a non-blocking event. This pointer is used within matching MPI_Wait and MPI_Waitall calls. After the wait call, the request handle is invalidated in the request buffer to ensure that the same request handle can be reused (reinitialized) by subsequent wait scenarios.

Some applications use receive wildcards (MPI_ANY_SOURCE), which may introduce non-determinism. ScalaTrace records both wildcards (as a unique value) and the actual sender of the wildcard receive. During code generation, the real sender is used so that the application behavior observed in the tracing run is preserved.

4. EXPERIMENTAL FRAMEWORK

To evaluate our communication benchmark generation tool, we generated C code with MPI calls for the NAS Parallel Benchmarks (NPB) suite (version 3.3 for MPI) using class C and D input sizes [2] and for the Sweep3D neutron-transport kernel [19]. These codes all have either a mesh-neighbor communication patterns or rely heavily on collective communication. Some of them (e.g., SP and BT) require communicator handling, others (e.g., IS) require averaging of parameters in MPI_Alltoallv and some (e.g., LU) require the recording of wildcard receives. Hence, the key features of our code-generation framework are thoroughly tested in these experiments.

Benchmark generation is based on traces obtained on (a) ARC, a cluster with 1728 cores on 108 compute nodes, 32 GB memory per node and an Infiniband Interconnect and (b) Jaguar, a petascale HPC installation at Oak Ridge National Laboratory with 18,668 compute nodes where each compute node contains dual hex-core processors, 16 GB memory, and a SeaStar2+ router. Benchmark generation is performed on a stand-alone workstation.

5. EXPERIMENTAL RESULTS

We performed the following experiments for the evaluation of our benchmark generation tool.

5.1 Correctness

Our first set of experiments verifies the correctness of the generated benchmarks, i.e., the benchmark generator's ability to retain the original applications' communication pattern. For these experiments, we acquired traces of our test suite on ARC, generated communication benchmarks, and executed these benchmarks also on ARC. To verify the correctness of the generated benchmarks, we linked both the generated codes and the original applications with mpiP [17] (see Figure 8, upper half). The mpiP tool is packaged as a lightweight MPI profiling library that gathers run-time statistics of MPI event counts and the message volumes exchanged. Experimental results (not presented here) showed that, for each type of MPI event, the event count and the message volume measured for each generated benchmark matched perfectly with those measured for the original application.

We then conducted experiments to verify that the generated benchmarks not only resemble the original applications in overall statistics but also that they preserve the original semantics on a per-event basis. To this end, we instrumented each generated benchmark with ScalaTrace and compared its communication trace with that of its respective original application. Due to differences in the call-site stack signatures between the original application and the generated benchmark, these traces cannot be identical, they can only be semantically equivalent. Therefore, we replayed both traces with the ScalaTrace-based ScalaReplay tool [20] to eliminate spurious structural differences and thus allow a fair comparison of traces as depicted in Figure 8 (lower half). The results (again, not presented here) show that the original applications and the generated benchmarks have equivalent traces. That is, the semantics of each of the original applications was precisely reproduced by the corresponding generated benchmark.

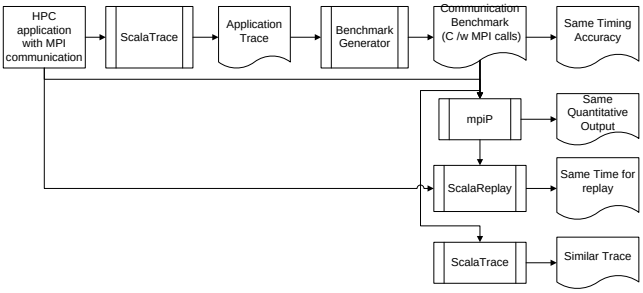

Figure 8: Experimental Framework

5.2 Accuracy of Timing Results

After evaluating that the generated code preserves the communication of the original application in terms of ordering of events and message volumes, we assessed the ability of a generated benchmark to retain the performance in terms of wall-clock time relative the original application. To measure the execution times of the original applications, we extended the PMPI profiling wrappers of MPI_Init and MPI_Finalize to obtain the start and end timestamps, respectively. The corresponding timing calls were also added to the generated benchmarks. We executed both the original application and the generated benchmark on the ARC system, measured and compared the elapsed times. The results obtained are shown in the Figure 9.

We observe from the graphs that the timings obtained for the generated benchmarks are very close to that of the original applications indicating very high accuracy. Quantitatively, the mean percentage error obtained by the formula

$$|T_{gen} - T_{app}|/T_{app} * 100$$

across all the graphs, is only 6.7%. Only one deviation with less timing accuracy was observed: class D FT for 512 nodes (110 seconds for benchmark and 145 seconds for original application) with a difference of 24%. The average delta time

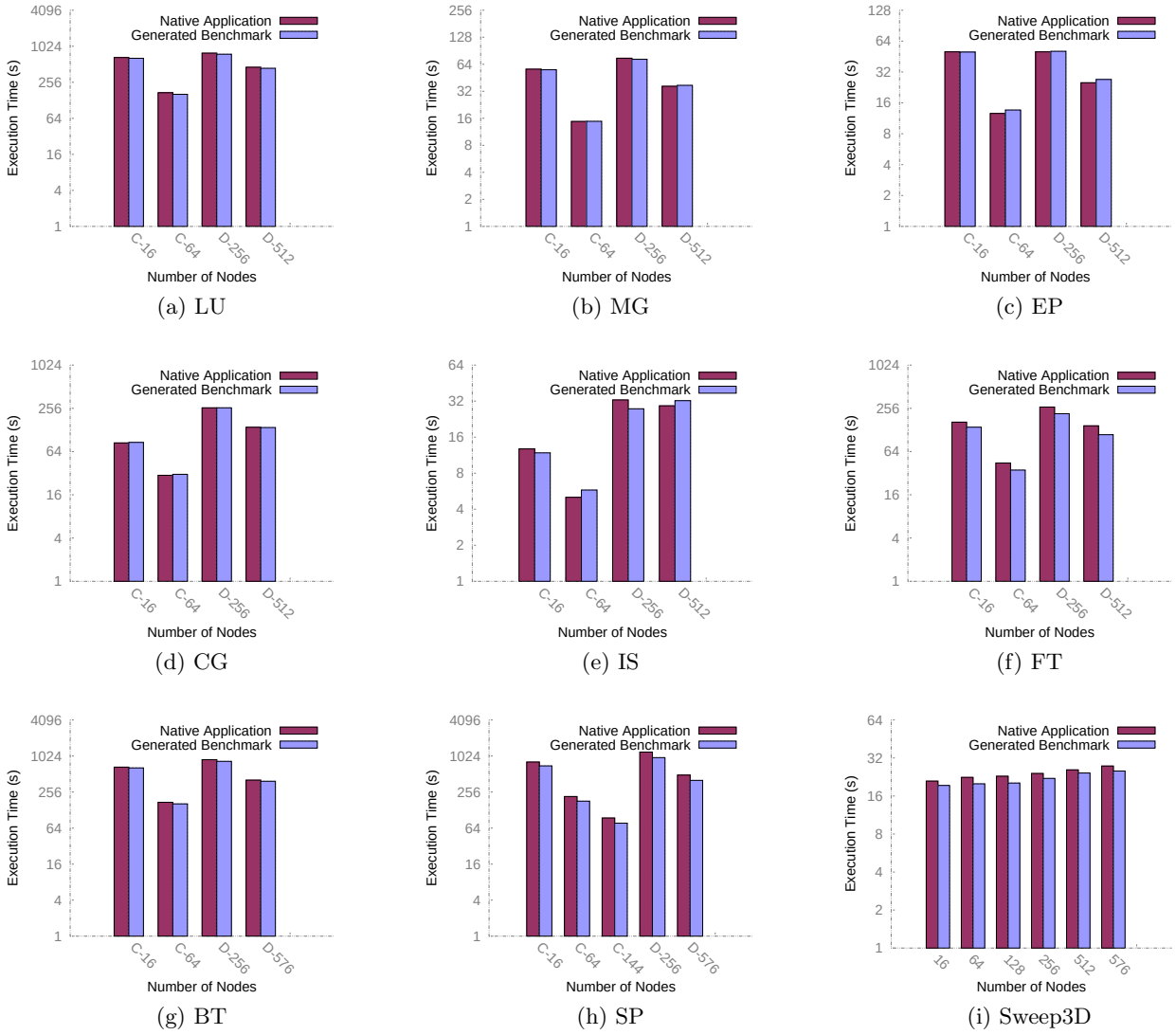

Figure 9: Graphs for Timing Accuracy of NAS PB Codes for Input Class C/D.

is used to resemble computation via busy wait. FT uses collectives heavily, which may result in computational imbalance. Hence, computation is more closely resembled by the maximum recorded time instead of the average for collectives.

The results for the class C IS benchmark varying from 16 to 512 processors are shown in the Figure 10. We observe that the execution time reduces from 16 processors to 64 processors and then increases as the number of processors increase from 256 and onwards. This can be explained through strong scaling in which the solution time varies with the number of processors for a fixed total problem size. The reason for the increase in execution time with the increase in the number of processors beyond certain number is due to increase in communication overhead and decrease in the per-processor computation.

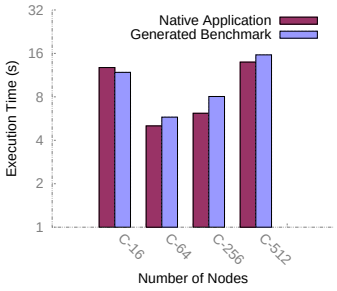

Figure 10: Timing Results for IS Class C Inputs on ARC

5.3 Cross Platform Results

We obtained cross platform results by running the benchmarks generated from IS and MG on ARC and Jaguar. The results are depicted in Figures 11 and 12.

Figure 11 shows that, in case of the IS benchmark, the difference between the execution times of benchmark from ARC and the original application on Jaguar reduces as the

number of processors increases. This is because the computation is split across a larger number of processors reducing the per-processor computation to communication ratio and thus reducing the effect of higher processing capacity of Jaguar. Also, for the IS benchmark, the lowest time in the 16-512 processor range is obtained for 64 processors on the ARC cluster resembling the actual application behavior. The same benchmark with delta times from ARC but executed on Jaguar resulted in the lowest time for 256 processors on the latter platform. This is also matching the lowest runtime (at 256 processors) of the original application on Jaguar (within the 15-512 processor range).

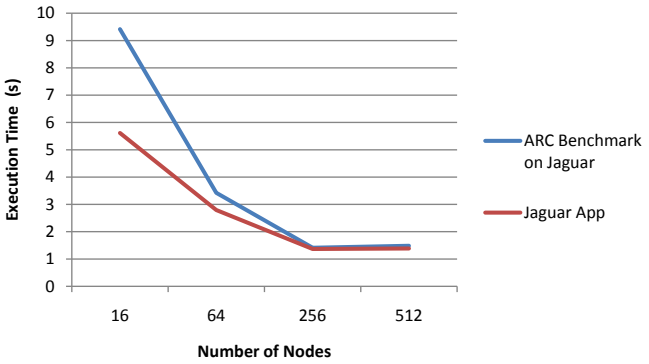

Figure 11: Cross-Platform Timing Results of IS

Figure 12 shows that the execution time of the MG benchmark obtained on ARC takes is close to that of the original application on Jaguar, whereas the execution time for the MG benchmark obtained on Jaguar itself very closely resembles it. The difference is due to diverging CPU speeds between ARC and Jaguar. Since Jaguar has a higher processor frequency than ARC, it finishes the computation earlier than indicated by the delta time for sleeps obtained by tracing on ARC.

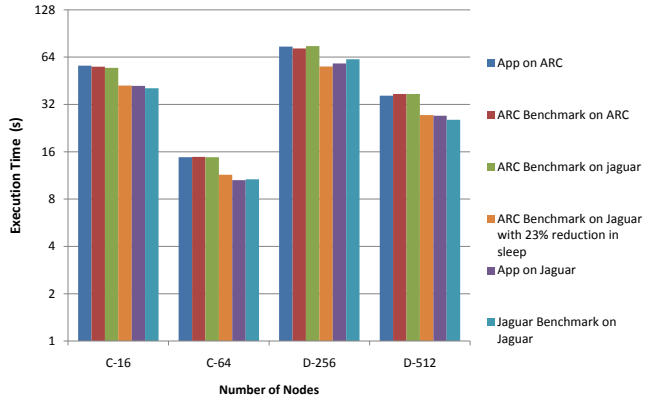

Figure 12: Cross-Platform Timing Results of MG

To verify the speedup of Jaguar over ARC, we executed a computational kernel that performs matrix multiplication on a single processor for square matrices of size 100x100 with iterations ranging from 3000 to 9000. Execution times are given in the Table 1. The CPU speedup of Jaguar over ARC is around 23%, which conforms our observations from traces.

Table 1: Execution Times for Matrix Multiplication on ARC and Jaguar

# Iterations	Time (ARC)	Time (Jaguar)	Speedup(%)
3000	44.168	34.259	22.43479442
6000	88.314	68.565	22.36225287
9000	132.443	102.614	22.5221416

We then reduced the sleeps in the MG benchmark obtained on ARC by 23% by proportionally shortening the delta times in the traces from ARC. The resulting MG run on Jaguar (see Figure 12, 4th bar) shows that the execution time then matches very closely to that the actual MG application on Jaguar.

Such performance experiments performed with the benchmarks generated by our tool could help in gaging different performance aspects related to communication on HPC systems with increasing complexities without actually porting the real applications to those platforms.

6. RELATED WORK

We utilize ScalaTrace to collect the communication trace of parallel applications. With a set of sophisticated domain-specific trace-compression techniques, ScalaTrace is able to generate traces that preserve the original source-code structure while ensuring scalability in trace size. Other tools for acquiring communication traces, such as Vampir [3], Extrae/ Paraver [14] and tools based on the Open Trace Format [9], do not have structure-aware compression. This results in trace file sizes that grow at least linearly with the number of MPI calls and the number of MPI processes. This also increases the size of any benchmark generated from such a trace, making it not only inconvenient for processing long-running applications executing on large-scale machines but also losing the ability to resembles the original loop structure of an application. This lack of scalability is addressed in part by call-graph compression techniques [8] but still falls short of the structural compression of ScalaTrace, which extends to any event parameters. Casas et al. utilize techniques of signal processing to detect internal structures of Paraver traces and extract meaningful parts of the trace files [4]. This facilitates trace analysis in a compressed manner but does not allow one to capture full information and becomes lossy and thus is not suitable for benchmark generation.

Xu et al.'s work on constructing coordinated *performance skeletons* to estimate application execution time in new hardware environments [22, 23] exhibits many similarities with our work. However, a key aspect of performance skeletons is that they filter out "local" communication (communication outside the dominant pattern). As a result, the generated code does not fully reflect the original application, which may cause subtle but important performance characteristics to be overlooked. Because our benchmark generation framework is based on lossless application traces, it is able to generate benchmarks with identical communication behavior to the original application.

Program slicing, statically reducing a program to a minimal form that preserves key properties of the original one, offers an alternate approach to generating benchmarks from application traces. Ertvelde et al. utilize program slic-

ing to generate benchmarks that preserve application performance characteristics while hiding its functional semantics [5]. This work focuses on resembling the branch and memory access behavior for sequential applications and may therefore complement our benchmark generator for parallel applications. Shao et al. designed a compiler framework to identify communication patterns for MPI-based parallel applications through static analysis [16], and Zhai et al. built program slices that contain only the variables and code sections related to MPI events and subsequently executed these program slices to acquire communication traces [24]. Program slicing and static benchmark generation in general have a number of shortcomings relative to our run-time, trace-based approach: (a) Their reliance on inter-procedural analysis requires that *all* source code be available. This includes complete source code of an application along with the source codes of all its dependencies, such as libraries, which is often unrealistic. (b) They lack execution time information. (c) They cannot accurately handle loops with data-dependent trip counts ("**while not** converged **do**..."). (d) They produce benchmarks that are neither human-readable nor editable.

Wu et al.'s work of generating the Conceptual benchmark [21] closely resembles to our work. ScalaTrace is used to collect the traces from application in their work. A trace traversal framework, which is similar to our traversal framework, is used to generate the source code in Conceptual, a domain specific language [13]. This language focuses on generating networking/communication benchmarks. This work does not generate all MPI calls but maps the MPI events from the trace to the corresponding combination of communication routines. The Conceptual language does not have the concept of "communicators" as in MPI. Thus, it cannot form the subsets of ranks based on a communicator. Since our work generates C code with MPI calls, it can translate all MPI events captured in the trace accurately. The Conceptual language does not have provisions like wildcard receives, thus generated code needs to be resolved for the source in the send and receive communication calls. Our work, generates lossless, accurate and human readable MPI communication calls in C source code from a single trace file obtained from ScalaTrace, which is easily portable to any platform, as opposed to Conceptual with the need to interpret Conceptual code, which more closely resembles trace replay.

7. CONCLUSION

We have designed and implemented a novel communication benchmark code generator that generates benchmark code in C with MPI calls from communication traces. These traces are generated by ScalaTrace, a lossless and scalable framework to extract communication, I/O operations and execution time while abstracting away the computations. These benchmarks are human readable, compact, easy to generate and port. They also preserve the behavior of the original application in terms of execution time, communication volume and ordering of events. Furthermore, application code is obfuscated by our benchmark generation process, which allows auto-generated benchmarks of otherwise restricted / distribution-controlled applications to be released to the public. And such benchmarks can be generated and released more frequently due to the automated generation process so that such benchmark releases can keep up more closely with fast development cycles of full-scale applications.

Experimental results demonstrate the ability of our code generator to generate the communication benchmarks from codes of the NAS Parallel Benchmark Suite and Sweep3D. The obtained results show that the benchmarks accurately preserve not only application semantics but also overall execution time. Our benchmark generator can benefit application developers, communication researchers and HPC system designers. It may assist in performance analysis of software, hardware and can also ease migration of applications across different platforms.

8. REFERENCES

[1] MPI-2: Extensions to the message passing interface, July 1997.

[2] 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*, 5(3):63–73, Fall 1991.

[3] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler. Performance optimization for large scale computing: The scalable VAMPIR approach. In *International Conference on Computational Science (2)*, pages 751–760, 2001.

[4] M. Casas, R. Badia, and J. Labarta. Automatic structure extraction from mpi applications tracefiles. In *Euro-Par Conference*, Aug. 2007.

[5] L. V. Ertvelde and L. Eeckhout. Dispersing proprietary applications as benchmarks through code mutation. In *Architectural Support for Programming Languages and Operating Systems*, pages 201–210, 2008.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[7] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[8] A. Knupfer. Construction and compression of complete call graphs for post-mortem program trace analysis. In *International Conference on Parallel Processing*, pages 165–172, 2005.

[9] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the Open Trace Format (OTF). In *International Conference on Computational Science*, pages 526–533, May 2006.

[10] J. Marathe and F. Mueller. Detecting memory performance bottlenecks via binary rewriting. In *Workshop on Binary Translation*, Sept. 2002.

[11] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *International Parallel and Distributed Processing Symposium*, Apr. 2007.

[12] 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*, 69(8):969–710, Aug. 2009.

[13] S. Pakin. The design and implementation of a domain-specific language for network performance testing. *IEEE Transactions on Parallel and Distributed Systems*, 18(10):1436–1449, Oct. 2007.

[14] V. Pillet, V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualize and analyze parallel code. In P. Nixon, editor, *Proceedings of the 18th Technical Meeting of WoTUG-18: Transputer and Occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, Manchester, United Kingdom, Apr. 9–12, 1995. IOS Press.

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

[16] S. Shao, A. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *In International Parallel and Distributed Processing Symposium*, 2006.

[17] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[18] J. S. Vetter and B. R. de Supinski. Dynamic software testing of mpi applications with umpire. In *Supercomputing*, page 51, 2000.

[19] H. Wasserman, A. Hoisie, and O. Lubeck. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14:330–346, 2000.

[20] X. Wu and F. Mueller. ScalaExtrap: Trace-based communication extrapolation for SPMD programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011.

[21] X. Wu, F. Mueller, and S. Pakin. Automatic generation of executable communication specifications from parallel applications. In *ICS*, pages 12–21, 2011.

[22] Q. Xu, R. Prithivathi, J. Subhlok, and R. Zheng. Logicalization of MPI communication traces. Technical Report UH-CS-08-07, Dept. of Computer Science, University of Houston, 2008.

[23] Q. Xu and J. Subhlok. Construction and evaluation of coordinated performance skeletons. In *International Conference on High Performance Computing*, pages 73–86, 2008.

[24] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. FACT: Fast communication trace collection for parallel applications through program slicing. In *Proceedings of SC'09*, pages 1–12, 2009.