Automatic Generation of Executable Communication Specifications from Parallel Applications

Xing Wu
North Carolina State University
xwu3@ncsu.edu

Frank Mueller
North Carolina State University
mueller@csc.ncsu.edu

Scott Pakin
Los Alamos National Laboratory
pakin@lanl.gov

ABSTRACT

Portable parallel benchmarks are widely used and highly effective for (a) the evaluation, analysis and procurement of high-performance computing (HPC) systems and (b) quantifying the potential benefits of porting applications for new hardware platforms. Yet, past techniques to synthetically parametrized hand-coded HPC benchmarks prove insufficient for today's rapidly-evolving scientific codes particularly when subject to multi-scale science modeling or when utilizing domain-specific libraries.

To address these problems, this work contributes novel methods to automatically generate highly portable and customizable communication benchmarks from HPC applications. We utilize ScalaTrace, a lossless, yet scalable, parallel application tracing framework to collect selected aspects of the run-time *behavior* of HPC applications, including communication operations and execution time, while abstracting away the *details* of the computation proper. We subsequently generate benchmarks with identical run-time behavior from the collected traces. A unique feature of our approach is that we generate benchmarks in CONCEPTUAL, a domain-specific language that enables the expression of sophisticated communication patterns using a rich and easily understandable grammar yet compiles to ordinary C+MPI. Experimental results demonstrate that the generated benchmarks are able to preserve the run-time behavior—including both the communication pattern and the execution time—of the original applications. Such automated benchmark generation is particularly valuable for proprietary, export-controlled, or classified application codes: when supplied to a third party, our auto-generated benchmarks ensure performance fidelity but without the risks associated with releasing the original code. This ability to automatically generate performance-accurate benchmarks from parallel applications is novel and without any precedence, to our knowledge.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

General Terms

Measurement, Performance

ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.

Keywords

application-specific benchmark generation, ScalaTrace, CONCEPTUAL, trace compression, communication, performance, domain-specific languages

1. INTRODUCTION

Evaluating and analyzing the performance of high-performance computing (HPC) systems generally involves running complete applications, computational kernels, or microbenchmarks. Complete applications are the truest indicator of how well a system performs. However, they may be time-consuming to port to a target machine's compilers, libraries, and operating system, and their size and intricacy makes them time-consuming to modify, for example, to evaluate the performance of different data decompositions or parallelism strategies. Furthermore, with intense competition to be the first to scientific discovery, computational scientists may be loath to risk granting their rivals access to their application's source code; or, the source code may be more formally protected as a corporate trade secret or as an export-controlled or classified piece of information. Computational kernels address some of these issues by attempting to isolate an application's key algorithms (e.g., a conjugate-gradient solver). Their relative simplicity reduces the porting effort, and they are generally less encumbered than a complete application. While their performance is somewhat indicative of how well an application will perform on a target machine, isolated kernels overlook important performance characteristics that apply when they are combined into a complete application. Finally, microbenchmarks stress individual machine components (e.g., memory, CPU, or network). While they are easy to port, distribute, modify, and run, and they precisely report characteristics of a target machine's performance, they provide little information about how an application might perform when the primitive operations they measure are combined in complex ways in an application.

The research question we propose to answer in this paper is the following: Is it possible to combine the best features of complete applications, computational kernels, and microbenchmarks into a single performance-evaluation methodology? That is, can one evaluate how fast a target HPC system will run a given application without having to migrate it and all of its dependencies to that system, without ignoring the subtleties of how different pieces of an application perform in context, without forsaking the ability to experiment with alternative application structures, and without restricting access to the tools needed to perform the evaluation?

Our approach is based on the insight that application performance is largely a function of the sorts of primitive operations that microbenchmarks measure and that if these operations can be juxtaposed as they appear in an application, the performance ought to be nearly identical. We therefore propose generating *application-specific performance benchmarks*. In fact, by "generating," we imply a fully automatic

approach in which a parallel application can be treated as a black box and mechanically converted into an easy-to-build, easy-to-modify, and easy-to-run program with the same performance as the original but absent the original's data structures, numerical methods, and other algorithms.

We take as input an MPI-based [7] message-passing application. To convert this into a benchmark, we utilize the approach illustrated in Figure 1. We begin by tracing the application's communication pattern (including intervening computation time) using ScalaTrace [12]. The resulting trace is fed into the benchmark generator that is the focus of this paper. The benchmark generator outputs a benchmark written in coNCePTuaL, a domain-specific language for specifying communication patterns [15]. The coNCePTuaL code can then be compiled into ordinary C+MPI code for execution on a target machine.

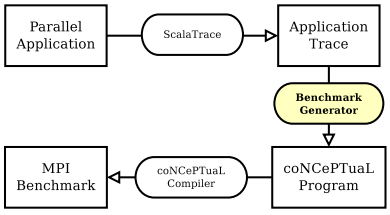


Figure 1: Our benchmark generation system

We utilize ScalaTrace [12] for communication trace collection because ScalaTrace represents the state of the art in parallel application tracing. It benefits benchmark generation in two aspects. First, due to its pattern-based compression techniques, ScalaTrace generates application traces that are lossless in communication semantics yet small and scalable in size. For example, ScalaTrace can represent all processes performing the same operation (e.g., each MPI rank sending a message to rank+4) as a single event, regardless of the number of ranks. Because the application trace is the basis for benchmark generation, this feature helps reduce the size of the generated code, making it more manageable for subsequent hand-modification. In contrast, previous application tracing tools, such as Extrae/Paraver [16], Tau [21], OpenlSpeedShop [19], Vampir [11], and Kojak [25], are less suitable for benchmark generation because their traces increase in size with both the number of communication events and the number of MPI ranks traced. Second, ScalaTrace is aware of the structure of the original program. It utilizes the stack signature to distinguish different call sites. Its loop-compression techniques can detect the loop structure of the source code. For example, if an iteration comprises a hundred iterations, and each iteration sends five messages of one size and ten of another, ScalaTrace represents that internally as a set of nested loops rather than as 1500 individual messaging events. These pattern-identification features help benchmark generation maintain the program structure of the original application so that the generated code will be not only be semantically correct but also human comprehensible and editable.

We use the domain-specific coNCePTuaL language [15] instead of a general-purpose language such as C or Fortran as the target language for benchmark generation. (coNCePTuaL does, however, compile to C source code.) Because coNCePTuaL is designed specifically for the expression of communication patterns, benchmarks generated in coNCePTuaL are highly readable. coNCePTuaL code includes almost exclusively communication specifications. Mundane benchmarking details such as error checking, memory allocation, timer calibration, statistics calculation, MPI subcommunicator creation, and so forth are all handled implicitly, which reduces code clutter.

We evaluated our benchmark generation approach with the NAS Parallel Benchmark suite [3] and the Sweep3D code [10]. We performed experiments to assess both the correctness and the timing accuracy of the generated parallel benchmarks. Experimental results show that the auto-generated benchmarks preserve the application's semantics, including the communication pattern, the message count and volume, and the temporal ordering of communication events as they appear in the original parallel applications. In addition, the total execution times of the generated codes are very similar to those of the original applications; the mean absolute percentage error across all of our measurements is only 2.9%. Given these experimental results, we conclude that the generated benchmarks are able to reproduce the communication behavior and wall-clock timing characteristics of the source applications.

The contributions of this work are (1) a demonstration and evaluation of the feasibility of automatically converting parallel applications into human-readable benchmark codes, (2) an algorithm for determining precisely when separately appearing collective-communication calls in fact belong to the same logical operation, and (3) an approach and algorithm for ensuring performance repeatability by introducing determinism into benchmarks generated from nondeterministic applications.

We foresee our work benefiting application developers, communication researchers, and HPC system procurers. Application developers can benefit in multiple ways. First, they can quickly gauge what application performance is likely to be on a target machine before exerting 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 accelerators (reducing computation time without directly affecting communication time). Communication researchers can benefit by being able to study the impact of novel messaging techniques without incurring the burden of needing to build complex applications with myriad dependencies and without requiring access to codes that are not freely distributable. Finally, people tasked with procuring HPC systems benefit by being able to instruct vendors to deliver specified performance on a given application without having to provide those vendors with the application itself.

This paper is structured as follows. Section 2 contrasts our approach to others' related efforts. Section 3 introduces ScalaTrace and the coNCePTuaL language with respect to their abilities to support benchmark generation. The salient features of our benchmark-generation approach are detailed in Section 4. Section 5 empirically confirms the correctness and accuracy of the benchmarks we generate and presents sample usage of our framework. Finally, Section 7 draws some conclusions from our findings.

2. RELATED WORK

The following characteristics of our benchmark-generation approach make it unique:

- The size of the benchmarks we generate increases sublinearly in the number of processes and in the number of communication operations.
- We exploit run-time information rather than limit ourselves to information available at compile time.
- We preserve all communication performed by the original application.

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 [4], Extrae/Paraver [16], and tools based on the Open Trace Format [9] lack structure-aware compression. As a result, the size of a trace file grows linearly with the number of MPI calls and the number of MPI processes, and so too would the size of any benchmark generated from such a trace, making it inconvenient for processing long-running applications executing on large-scale machines. This lack of scalability is addressed in part by call-graph compression techniques [8] but still falls short of our structural compression, 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 [5]. While this approach could facilitate trace analysis, it is lossy and thus not suitable for benchmark generation.

Xu et al.'s work on constructing coordinated *performance skeletons* to estimate application execution time in new hardware environments [27, 28] 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 reproduce 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. In addition, we generate benchmarks in coNCePTuaL instead of C so that the generated benchmarks are more human-readable and editable.

Program slicing, statically reducing a program to a minimal form that preserves key properties of the original, offers an alternative approach to generating benchmarks from application traces. Ertvelde et al. utilize program slicing to generate benchmarks that preserve an application's performance characteristics while hiding its functional semantics [6]. This work focuses on resembling the branch and memory access behaviors 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 [20], 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 [29]. Program slicing and static benchmark generation in general have a number of shortcomings relative to our run-time, trace-based approach: Their reliance on inter-procedural analysis requires that *all* source code—the application's and all its dependencies—be available; they lack run-time timing information; they cannot accurately handle loops with data-dependent trip counts ("**while not** converged **do**…"); and they produce benchmarks that are neither human-readable nor editable.

3. BACKGROUND

Our benchmark generation approach utilizes the ScalaTrace infrastructure [12] to extract the communication behavior of the target application. Based on the application trace, we generate benchmarks in coNCePTuaL [15], a high-level domain-specific language (with an associated compiler and run-time system) designed for testing the correctness and performance of communication networks. This section introduces the features of ScalaTrace and coNCePTuaL that enable our benchmark generation methodology.

3.1 ScalaTrace

ScalaTrace is chosen as the trace collection framework because it generates near constant-size communication traces for a parallel applications regardless of the number of nodes while preserving structural information and temporal ordering. This is important because

```
for(i=0; i<1000; i++){
    MPI_Irecv(LEFT, ...);
    MPI_Isend(RIGHT, ...);
    MPI_Waitall(...);
}
```

Figure 2: Sample code for RSD and PRSD generation

it makes the size of the generated benchmarks reasonably small and independent of node count.

ScalaTrace achieves near constant-sized traces through pattern-based compression. It uses extended regular section descriptors (RSDs) 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. Power-RSDs (PRSDs) recursively specify RSDs nested in loops. For example, the program fragment shown in Figure 2 establishes a ring-style communication across N nodes. The three RSDs,

RSD1: {⟨*rank*⟩, MPI_Irecv, LEFT}
RSD2: {⟨*rank*⟩, MPI_Isend, RIGHT}
RSD3: {⟨*rank*⟩, MPI_Waitall}

denote the MPI_Send, MPI_Receive, and MPI_Waitall operations in a single loop iteration, where ⟨*rank*⟩ takes on each value from 0 to $N-1$ in turn. ScalaTrace then detects the loop structure and outputs the single PRSD, {1000, RSD1, RSD2, RSD3}, to concisely denote a single, 1000-iteration loop. Note that the intra-node loop compression is done on-the-fly to reduce memory overhead and compression time. Finally, the local traces are combined into a single global trace upon application completion (i.e., within the PMPI interposition wrapper for MPI_Finalize). This inter-node compression detects similarities among the per-node traces and merges the RSDs by combining their lists of participating nodes. For example, in Figure 2, because each MPI routine is called with the same parameters on each node, the RSDs within the PRSD are consequently merged across nodes as

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$\}$

Besides communication tracing, ScalaTrace also stores application computation times in a scalable way [17]. Computation is defined as the time between consecutive MPI calls. Rather than store individual computation-time measurements, ScalaTrace compresses into a histogram the time taken by all instances of a particular computation (identified by its unique call path) across all loop iterations and all nodes. By grouping computation times in this manner, ScalaTrace achieves good compression while still addressing the time variations that are expected on different call paths. For example, the time spent in computation prior to the first statement of a loop generally differs significantly from the time spent in the first iteration, which generally differs significantly from the times spent in subsequent iterations.

3.2 coNCePTuaL

coNCePTuaL is a tool designed to facilitate rapid generation of network benchmarks. coNCePTuaL includes a compiler for a high-level specification language and an accompanying run-time library. coNCePTuaL programs are understandable even to non-experts because of its English-like grammar. For example, the following is a *complete* coNCePTuaL benchmark program corresponding to the code snippet presented in Figure 2:

```
FOR 1000 REPETITIONS {
  ALL TASKS RESET THEIR COUNTERS THEN
  ALL TASKS t ASYNCHRONOUSLY SEND A 1 KILOBYTE
    MESSAGE TO TASK t+1 THEN
  ALL TASKS AWAIT COMPLETION THEN
  ALL TASKS LOG THE MEDIAN OF elapsed_usecs
```

```
    AS "Time (us)".
}
```

Note in the above that no variable or function declarations are required; no buffer allocation is required; no MPI_Request or MPI_Status objects need to be defined; no MPI communicators need to be queried for rank and size; no files need to be opened and written to; no statistics-calculating routines need to be implemented; no error codes need to be checked; no matching receive needs to be posted for each send (but can be if the programmer requires more precise control over posting order); and no special cases for the first and last task (rank) need to be specified. Nevertheless, coNCePTuaL is able to express sophisticated communication patterns utilizing a variety of collective and point-to-point communication primitives, looping constructs, and conditional operations. When executed, the generated code produces log files that contain a wealth of information about the measured communication performance, code build characteristics, execution environment, and other information needed to yield reproducible performance measurements [14].

The aforementioned features make coNCePTuaL an ideal language for benchmark generation. In the following section, we present our approach to producing coNCePTuaL output from ScalaTrace input.

4. BENCHMARK GENERATION

4.1 Overview

The process of automatic code generation from traces is the process of traversing the parallel application trace, interpreting the RSDs and PRSDs, and generating the corresponding coNCePTuaL program. We designed a trace traversal framework that walks through the trace and invokes a language-dependent code generator for each RSD and PRSD. A code generator is a pluggable function that conforms to a predefined interface. By implementing a generator for a different target language, we can easily generate code for languages other than coNCePTuaL as well.

Most of the conversion from RSDs and PRSDs to coNCePTuaL code is straightforward. An RSD representing point-to-point communication (blocking or nonblocking) is converted to a coNCePTuaL `SEND` or `RECEIVE` statement; computation time encoded in an RSD is converted to a coNCePTuaL `COMPUTE` statement; and a PRSD is converted to a coNCePTuaL `FOR EACH` loop. Behavior that differs across loop iterations (message destinations, compute times, etc.) is implemented with a coNCePTuaL `IF` statement conditioned on a loop variable. There are a few subtleties involved in the mapping from ScalaTrace to coNCePTuaL; Section 4.2 discusses these.

Our view, however, is that a naive conversion from a trace to benchmark code has two important shortcomings. First, one of our goals is for the generated benchmark code to be *readable*, so a human can easily examine, understand, and modify the code. Our second goal is for the performance reported by the benchmark program to be *reproducible*, to make it a more suitable vehicle for experimentation. In short, we want it to be possible to reason about a generated benchmark's behavior and performance. However, achieving the goals of readability and reproducibility is a challenging research problem and is the subject of this section.

One difficulty in improving benchmark readability is the elimination of constructs whose behavior cannot statically be determined. Consider the following snippet of C code:

```
if (rank == 0)
  MPI_Reduce(⟨argument list⟩);
else
  MPI_Reduce(⟨the same argument list⟩);
```

It is not possible to know if those two MPI_Reduce() calls are part of the same collective operation without knowing the complete, run-time control flow of the program—on each rank individually—that led to the execution of the code shown above. The challenge is how to merge per-rank collective operations found in a trace into a single collective operation whose participants can be identified *statically*. An example of such an operation expressed in coNCePTuaL is "`TASKS xyz SUCH THAT 3 DIVIDES xyz REDUCE A DOUBLEWORD TO TASK 0`"; no further information is required to know that tasks 0, 3, 6, 9, ... are the participants in that reduction operation. Section 4.3 presents our algorithm for matching collective operations specified separately on each node.

An MPI feature that hinders performance reproducibility is nondeterminism. MPI supports "wildcard receives" (MPI_ANY_SOURCE), which can receive messages from any sender. While this feature *can* lead to correctness issues [23], and we do address this, we are concerned primarily with the different performance that can result from different messages matching a set of wildcard receives. Consider, for example, the following use of the MPI_Recv receive operation:

```
MPI_Recv(..., MPI_ANY_SOURCE, ..., status);
if (status.MPI_SOURCE == 0)
  ⟨Do some long-running computation.⟩
else
  ⟨Do some short-running computation.⟩
MPI_Recv(..., MPI_ANY_SOURCE, ..., status);
```

Depending on the sender's MPI rank (`status.MPI_SOURCE`), the preceding code can take either a long time or a short time to run. Because the sender whose message matches the MPI_Recv can vary from run to run, the execution time of the preceding code also varies from run to run. While this behavior may be reasonable for an application, we deem it inappropriate for a benchmark program. As benchmarks are commonly used to evaluate system performance, small changes in a target machine's hardware or system software should not result in arbitrarily large changes in a benchmark's execution time. Section 4.4 presents our algorithm for removing performance nondeterminism caused by wildcard receives in the input trace.

4.2 Engineering Details

coNCePTuaL is not designed to exactly represent MPI features. In fact, the coNCePTuaL compiler can compile the same source program to C+MPI, C+Unix sockets, or to any other language/communication library combination for which a compiler backend exists. Consequently, coNCePTuaL contains collectives that MPI lacks (e.g., arbitrary many-to-many reductions with non-overlapping source and destination task sets), and MPI contains collectives that coNCePTuaL lacks (e.g., scatters of different-sized messages to different destinations). We therefore had to "impedance match" the benchmark generator's MPI-centric input to coNCePTuaL output. Our approach is to replace each unsupported MPI collective with one or more coNCePTuaL collectives that represent a similar communication pattern (i.e., data fan in or fan out) and data volume. Table 1 presents the substitutions we made.

MPI has a notion of a "communicator," which is a subset of the available ranks, renumbered and possibly reordered. Every MPI communication operation takes a communicator as an argument and uses it to specify the participants in the operation. A disturbing consequence of communicators is that a line in the application source code that seems to be sending a message to, say, rank 3 may in fact be sending a message to rank 8 in the primordial MPI_COMM_WORLD communicator. To make the generated benchmarks more readable we keep track of the mapping of every rank within every communicator to an "absolute" rank within MPI_COMM_WORLD and express all

Table 1: Mapping of MPI collectives to coNCePTuaL

MPI collective	coNCePTuaL implementation
Allgather	`REDUCE` + `MULTICAST`
Allgatherv	`REDUCE` with averaged message size + `MULTICAST`
Alltoallv	`MULTICAST` with averaged message size
Gather	`REDUCE`
Gatherv	`REDUCE` with averaged message size
Reduce_scatter	*n* many-to-one `REDUCE`s with different message sizes and roots, where *n* is the communicator size
Scatter	`MULTICAST`
Scatterv	`MULTICAST` with averaged message size

generated computation and communication operations in terms of these absolute ranks.

4.3 Combining Per-Node Collectives

As discussed in Section 4.1, MPI allows multiple statements in the source code to represent a single, common collective operation. Because ScalaTrace differentiates call sites by call-stack signatures, this use of collectives generates distinct RSDs in the trace. To improve benchmark readability, before generating coNCePTuaL code we want to combine these separate RSDs, each representing a subset of the collective's participants, into a single RSD that represents the complete set of participants. Figure 3 illustrates the intention, using C+MPI (with the omission of most MPI arguments) instead of RSDs for clarity. Figure 3(a) presents the initial communication pattern, in which each of ranks 0 and 1 invoke MPI_Barrier from a different source-code line. Assuming these are found to be the same collective, we want to hoist the MPI_Barrier outside of all conditionals on the rank, as shown in Figure 3(b).

```
if(rank == 0) {
    MPI_Isend(1);
    MPI_Barrier();
}
if(rank == 1) {
    MPI_Barrier(1);
    MPI_Irecv(0);
}
MPI_Wait();
```

(a) C+MPI program

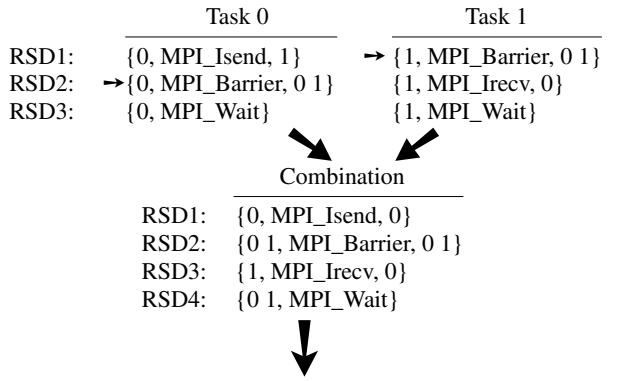
```
if(rank == 0)
    MPI_Isend(1);
MPI_Barrier();
if(rank == 1)
    MPI_Irecv(0);
MPI_Wait();
```

(b) Aligned collectives

Figure 3: Combining collectives across separate source-code statements

To perform this transformation, recall that our benchmark generator operates on communication traces, not on application source code; it therefore does not literally perform the source-code transformation shown in Figure 3. Rather, it follows the sequence of steps presented in Algorithm 1 to align in time the RSDs of the same collective operation across nodes then combine these RSDs into a single RSD specifying the complete set of nodes to which the collective operation applies.

The main idea, illustrated in Figure 4 for RSDs corresponding to the C+MPI code in Figure 3, is to stop the trace traversal for a node at each collective in which it participates until all of the other participating nodes have arrived at the same collective. Algorithm 1 guarantees that (1) a collective operation corresponds to only one RSD in the output trace, (2) the ordering of MPI events for each node is preserved in the trace, and (3) the output trace is still in a compressed format. This algorithm tracks the traversal on different nodes by maintaining a *traversal context* for each node. The traversal context stores the current RSD the node is executing, the loop stack the execution is in, and the iteration count for each loop in the stack. Upon startup, the algorithm traverses the trace on behalf of node 0, which is called the current *running node*. For each RSD of non-collective MPI routines that the running node is involved in, the algorithm extracts the current MPI event and appends an RSD to the output queue. (Note that an RSD can contain multiple MPI events across loop iterations and across nodes due to compression.) For collectives, however, the traversal stops for the current running node and switches to the next node in the communicator (indicated by the small arrows in Figure 4). When the last node in the communicator arrives at the collective, the algorithm appends the RSD for all the nodes to the output queue and switches the traversal back to the first node that is blocked on the same collective. We treat MPI_Finalize as a collective so that the algorithm cannot finish until the traversal is done for all the nodes. To guarantee that the new trace is scalable in length, we apply ScalaTrace's loop compression algorithm [12] to the output RSD queue each time a new RSD is appended to the queue. The complexity of this algorithm is $O(p \cdot e)$, where p is the number of MPI tasks and e is the number of communication events per task. Nevertheless, we do not blindly run this algorithm for arbitrary input traces. Before applying the algorithm we first check the trace to see if there are unaligned collectives. This check costs only $O(r)$, where r is the number of RSDs in the trace and is typically much smaller than e due to compression.




```
TASK 0 ASYNCHRONOUSLY SENDS AN x-BYTE MESSAGE TO
  UNSUSPECTING TASK 1 THEN
ALL TASKS SYNCHRONIZE THEN
TASK 1 ASYNCHRONOUSLY RECEIVES AN x-BYTE MESSAGE
  FROM TASK 0 THEN
ALL TASKS AWAIT COMPLETION.
```

Figure 4: Operation of Algorithm 1

4.4 Eliminating Nondeterminism

MPI supports the use of a wildcard value, MPI_ANY_SOURCE, for the *source* parameter of point-to-point receives. For example, in the NAS Parallel Benchmarks's implementation of LU decomposition [3], nodes use MPI_ANY_SOURCE to receive messages in arbitrary order from their neighbors in a 2-D stencil. The problem with the use of MPI_ANY_SOURCE from a benchmarking perspective is that it has the potential to introduce performance artifacts, as discussed in Section 4.1. That is, each run of LU may stress the communication subsystem slightly differently based on the order in which messages happen to be received. To promote reproducibility of empirical measurements, our benchmark generator removes nondeterminism by replacing wildcard receives with arbitrary but valid non-wildcard receives.

As in Section 4.3's algorithm for combining collectives, Algorithm 2 utilizes a trace-traversal approach to resolve wildcard receives. Let e_{ijk} represent an MPI event k that is issued by node i and has

Algorithm 1 Algorithm to align collectives

Precondition: T_{in}: input trace, N: total number of nodes
Postcondition: T_{out}: the trace for CONCEPTUAL code generation

```
 1: function INITIALIZATION(T_in, N)
 2:     for i ← 1, N do
 3:         Allocate traversal context C[i]
 4:         C[i].RSD ← T_in.head
 5:     end for
 6:     Initialize T_out to am empty trace
 7:     T_out ← ALIGN(0, T_out)                    ▷ Start with node 0
 8:     return T_out
 9: end function

10: function ALIGN(n, T_out)
11:     iter ← C[n].RSD
12:     while iter do
13:         if node n is not in iter.rank_list then
14:             iter ← iter.next
15:         else
16:             if iter.op is not a collective then
17:                 Extract current MPI event
18:                 Append a new RSD to T_out
19:                 Compress T_out
20:                 iter ← iter.next
21:                 continue
22:             end if
23:             if iter.op is a collective or MPI_Finalize then
24:                 if some participants have not arrived yet then
25:                     C[n].RSD ← iter
26:                     next ← the next node in the communicator
27:                     ALIGN(next, T_out)
28:                 else
29:                     Append an RSD for all participants to T_out
30:                     Compress T_out
31:                     C[n].RSD ← iter
32:                     for each i ∈ {participants} do
33:                         C[i].RSD ← C[i].RSD.next
34:                     end for
35:                     first ← the first node in the communicator
36:                     ALIGN(first, T_out)
37:                 end if
38:             end if
39:         end if
40:     end while
41:     return T_out
42: end function
```

node j as its peer. We maintain two lists for each node x: a list L_1 of the to-be-matched MPI events $e_{xj_1 1}, e_{xj_2 2}, e_{xj_3 3}, \ldots$ that were issued by node x itself and a list L_2 of the MPI events $e_{i_1 xk_1}, e_{i_2 xk_2}, e_{i_3 xk_3}, \ldots$ specifying the events issued by other nodes that should be matched by node x. Upon startup, this algorithm traverses the input trace on behalf of an arbitrary node x. During the traversal, it adds the unmatched point-to-point operations to list L_1 of node x and to list L_2 of each peer node. The traversal for node x stops when the execution is blocked on (1) a blocking send/receive, (2) a collective, or (3) a wait operation. It then switches the traversal to a node y whose execution will potentially unblock the execution on node x. In order to be selected as the target node to which the traversal switches (i.e., node y), a node must be (1) the destination/source of the blocking send/receive on node x, (2) a node in the same communicator with node x, or (3) the destination/source of one of the nonblocking sends/receives that node x is waiting on, respectively. During the traversal for node y, we look up every MPI operation we arrived at in list L_2 of node y to detect matches. When a match is found, we delete the event from both lists. If possible, we unblock the execution on node x so that the traversal for it can proceed later on. If the receiver of a match uses MPI_ANY_SOURCE, this value is replaced with the rank of the (first) matching sender so that the wildcard source is resolved. Collectives are handled in a similar way as Algorithm 1 by blocking the traversal until every participating node arrives. We treat MPI_Finalize as a collective that all the nodes participate in, so that every node is traversed before the algorithm finishes. Because Algorithm 2 is again based on traversing a trace and each MPI event is evaluated exactly once, the complexity is also $O(p \cdot e)$, where p is the number of MPI tasks and e is the number of communication events per task. Similarly, the use of wildcard receives is checked at a cost of $O(r)$ before applying this algorithm, where r is the number of RSDs in the trace and, typically, $r \ll e$.

A ScalaTrace trace is obtained from an instance of a correct execution of the original parallel application. However, ScalaTrace does not represent this or any other specific execution because it does not replace the wildcard *source* value with the rank of the actual sender. Consequently, if the original application potentially deadlocks, Algorithm 2 suffers from the same risk. As an example, the code fragment in Figure 5(a) deadlocks if the wildcard receive is matched with node 0 but completes if matched with node 2. One possible execution generates the trace shown in Figure 5(b), which causes Algorithm 2 to hang because node 0 is blocked on MPI_Finalize and node 1 is blocked on MPI_Recv(0) during trace traversal.

```
if(rank == 1){
    MPI_Recv(MPI_ANY_SOURCE);
    MPI_Recv(0);
}
if(rank == 0 || rank == 2){
    MPI_Send(1);
}
```

(a) MPI program with potential deadlock

RSD1:	{1, MPI_Recv, MPI_ANY_SOURCE}
RSD2:	{1, MPI_Recv, 0}
RSD3:	{0, MPI_Send, 1}
RSD4:	{2, MPI_Send, 1}

(b) The trace of (a) that makes Algorithm 2 hang

Figure 5: Potential deadlock

To avoid potential hangs in Algorithm 2 caused by nondeterminism in the original application, our benchmark generator extends Algorithm 2 to detect deadlock conditions during trace traversal. Notice that these deadlocks stem from incorrect MPI semantics of the application, not from our tracing or code-generation framework. We decided to identify such incorrect MPI programs and report the existence of deadlocks to the user. To this end, we track another two types of events during traversal: (1) T_{ijk}, the transfer of traversal from node i to node j due to MPI event e_k, and (2) U, the unblocking event. We append these events to a global list, L_3, in the order they were encountered during the traversal. If the traversal is switched to node n while node n is blocked on an MPI event e_k, the deadlock detection algorithm traverses L_3 to determine if any unblocking event U has taken place since the last time the traversal left node n due to the same MPI event e_k. If there is no unblocking event found, a potential cyclic dependency is detected. If e_k is a blocking send/receive, then a deadlock potential has been uncovered and the algorithm terminates. If e_k is a wait operation blocked on multiple requests, the traversal is

Algorithm 2 Algorithm to resolve wildcard receive (without deadlock detection)

Precondition: T: input trace, N: total number of nodes
Postcondition: T: trace without wildcard receive

```
 1: function INITIALIZATION(T, N)
 2:     for i ← 1, N do
 3:         Allocate list L1 and list L2 for node i
 4:         Allocate traversal context C[i]
 5:         C[i].RSD ← T.head
 6:     end for
 7:     T ← Match(0, T)                          ▷ Start with node 0
 8:     return T
 9: end function

10: function MATCH(n, T)
11:     iter ← C[n].RSD
12:     while iter do
13:         if node n is not in iter.rank_list then
14:             iter ← iter.next
15:         else
16:             if iter.op is point-to-point operation then
17:                 if match with an event e_ink in L2 then
18:                     L2.delete(e_ink)
19:                     node_i.L1.delete(e_ink)
20:                     if node_i.L1 is empty then
21:                         C[i].RSD ← C[i].RSD.next   ▷ unblock
22:                     end if
23:                     if iter.peer is MPI_ANY_SOURCE then
24:                         iter.peer = i          ▷ resolve the wildcard
25:                     end if
26:                     iter ← iter.next
27:                     continue
28:                 else
29:                     p ← iter.peer
30:                     L1.add(e_np(k_n++))
31:                     node_p.L2.add(e_npk_n)
32:                     if iter.op is blocking operation then
33:                         C[n].RSD ← iter
34:                         MATCH(p, T)
35:                     else
36:                         iter ← iter.next
37:                         continue
38:                     end if
39:                 end if
40:             end if
41:             if iter.op is collective or MPI_Finalize then
42:                 ...                            ▷ refer to Algorithm 1
43:             end if
44:             if iter.op is wait operation then
45:                 if L1 is not empty then
46:                     MATCH(L1.first.getPeer(), T)
47:                 else
48:                     iter ← iter.next
49:                     continue
50:                 end if
51:             end if
52:         end if
53:     end while
54:     return T
55: end function
```

proxied to the peer of another nonblocking communication on which node n is waiting. If the peers of all the pending nonblocking sends/receives have been traversed and the cyclic dependency still exists, a deadlock potential has been detected and the algorithm terminates. This algorithm implements a *sufficient* deadlock detection scheme. As a result, Algorithm 2 is guaranteed to be deadlock-free. However, unlike the DAMPI algorithm [23], Algorithm 2 does not establish or test the permutations of all execution interleavings and thus does not present a *necessary* condition for a deadlock as the approach is based on a single trace sequence of events. It may therefore fail to identify deadlocks in the original application that were not uncovered by the specific trace execution.

4.5 Sources of Performance Inaccuracy

As indicated, there are a number of ways in which our benchmark generator trades off performance fidelity for an improved ability to reason about the generated code and its performance: computation times are summarized across ranks instead of being specified individually (Section 3.1); some complex MPI collectives are implemented in terms of more basic coNCePTuaL collectives (Section 4.2); and nondeterministic receive ordering is replaced with an arbitrary deterministic ordering (Section 4.4). In Section 5 we examine the impact of these design decisions in the context of a suite of test programs.

5. EVALUATION

5.1 Experimental Framework

To evaluate our benchmark-generation methodology, we generated coNCePTuaL codes for the NAS Parallel Benchmarks (NPB) suite (version 3.3 for MPI, comprising BT, CG, EP, FT, IS, LU, MG, and SP) using the class C input size [3] and for the Sweep3D neutron-transport kernel [24]. These benchmarks all have either a mesh-neighbor communication pattern or rely heavily on collective communication. Some of them (e.g., Sweep3D) require collective alignment (Section 4.3), and some (e.g., LU) require the resolution of wildcard receives (Section 4.4). Hence, the key features of our code-generation framework are fully tested in this set of experiments. We believe results from the NPB and Sweep3D in this paper, combined with previous ScalaTrace experiments [13, 26], are sufficient to demonstrate the correctness of our approach, and we do not foresee any algorithmic or technical problems with generating code for larger applications. Moreover, these benchmarks are sufficient to demonstrate our ability to retain an application's performance characteristics. In particular, several kernels in the NPB suite, including CG, FT, and MG, are known to be memory-bound [18], which stresses our generated benchmarks' ability to mimic computation with spin loops of the same duration.

Benchmark generation is based on traces obtained on (a) Ocracoke, an IBM Blue Gene/L [1] with 2,048 compute nodes and 1 GB of DRAM per node and (b) ARC, a cluster with 1728 cores on 108 compute nodes, 32 GB memory per node, and an Ethernet interconnect. Due to limited access to these systems our experiments generally run on only a subset of the available nodes. Benchmark generation is performed on a standalone workstation.

5.2 Communication 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 Blue Gene/L, generated coNCePTuaL benchmarks, and executed these benchmarks also on Blue Gene/L. To verify the correctness of the generated benchmarks, we linked both them and the original applications with mpiP [22], 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 benchmarks 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 are never bit-for-bit identical. Therefore, we replayed both traces with the ScalaTrace-based ScalaReplay tool [26] to eliminate spurious structural differences and thereby fairly compare the pairs of traces. The results (again, not presented here) show that the original applications and the generated benchmarks generated equivalent traces. That is, the semantics of each of the original applications was precisely reproduced by the corresponding generated benchmark.

5.3 Accuracy of Generated Timings

Having determined that benchmarks produced using our benchmark generator faithfully represent the communication performed by the original applications, we then assessed the generated benchmarks' ability to retain the original applications' performance. To measure the total execution time of the original applications, we extended the PMPI profiling wrappers of MPI_Init and MPI_Finalize to obtain timestamps). The corresponding CONCEPTUAL timing calls were also added to the generated benchmarks. We ran both the original application and the generated benchmark on the Blue Gene/L system and compared the total elapsed times. Figure 6 shows that the timing accuracy is qualitatively extremely good. Quantitatively, the mean absolute percentage error (i.e., $100\% \times |(T_{\text{CONCEPTUAL}} - T_{\text{app}})/T_{\text{app}}|$) across all of Figure 6 is only 2.9%, and only two data points exhibit worse than 10% deviation: LU at 256 nodes observes a deviation of 22% (40 s for the benchmark versus 52 s for the original application), and SP at 16 nodes observes a deviation of 10% (980 s for the benchmark versus 1092 s for the original application).

5.4 Applications of the Benchmark Generator

The experimental results presented in Sections 5.2 and 5.3 indicate that the performance of the generated benchmarks can be trusted. We now present an example of some what-if analysis that is made practical by automatic benchmark generation.

A current trend in high-performance computing is to supplement general-purpose CPUs with more special-purpose computational accelerators (e.g., GPUs).[1] However, by Amdahl's Law [2], accelerating only an application's computational phases does not always lead to proportional overall speedup. Unfortunately, it is nontrivial both to predict how fast a parallel application will run once accelerated and to port a parallel application to an accelerated architecture. Application developers may also optimize performance by overlapping communication and computation. This too takes time to implement and leads to a reduction in execution time that can be difficult to predict.

Because the CONCEPTUAL benchmarks produced by our generator are easy to modify, we can use our framework to estimate how fast an application can be expected to run once accelerated or once communication and computation fully overlap. We generated a benchmark from the NPB BT code on 64 cores using the class C input. We then modified the CONCEPTUAL code to vary the time spent in all computation phases from 100% down to 0% of their original time to simulate different expected improvements due to acceleration. We ran the resulting benchmark variations on the ARC cluster (cf. Section 5.1) and plotted the results in Figure 7.

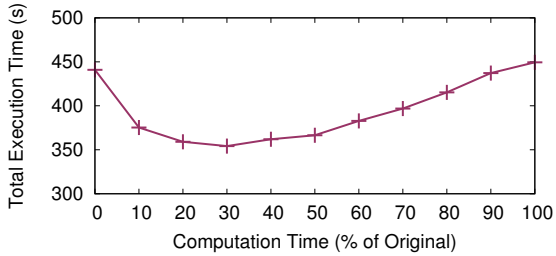


Figure 7: Communication performance of BT

Reading Figure 7 from right to left, the data points ranging from 100% down to 30% of the original application's compute time are essentially what one might expect: a steady but sublinear decrease in total execution time. That is, a fabricated 3.3x speedup of computation leads to only a 21% reduction in total execution time for BT. However, as computation time continues to decrease, rather than reach a plateau, the total execution time *increases*. At the 0% computation mark, which represents infinitely fast processors on a modern Ethernet network, there is essentially *no* speedup over the unmodified BT execution time.

To understand this puzzling behavior, note that BT is a stencil code consisting almost exclusively of asynchronous point-to-point communication operations, with only a few collectives at the beginning and end of the execution. Reducing the time between subsequent communication operations alters the dynamics of the messaging layer and leads to the observed increase in performance. For example, if messages begin arriving faster than they can be processed, they will start being directed to the MPI implementation's unexpected-receive queue, which incurs a performance cost in the form of an extra memory copy to transfer unexpected messages to the target buffer. Once all available space for storing incoming messages on a given node is exhausted, the MPI implementation's flow-control mechanism must stall any senders and later pay a cost in network latency to resume them. It is the nonlinear effects such as those that make it important to quantify potential performance improvements using a framework such as ours before investing the effort to accelerate an application.

We should note that the experimental result presented in Figure 7 is both application-specific and platform-specific. Yet, with our benchmark-generation approach, the experiment can easily be repeated on different platforms without ever needing to port the original application. In addition, our BT experiment can easily be refined to utilize different speedup factors for different computational phases. We foresee this type of performance experimentation, enabled by our benchmark generator, becoming increasingly important as HPC hardware increases in complexity and requires expanded efforts to port large applications (for potentially small performance gains).

6. DISCUSSION AND FUTURE WORK

This work has demonstrated the feasibility of automatically generating performance-accurate and highly readable benchmarks from application traces. The ability to generate benchmarks that can be executed with arbitrary number of MPI processes still remains an open problem. Our prior publication contributed a set of algorithms and techniques to extrapolate a trace of a large-scale execution of an application from traces of several smaller runs [26]. We intend to incorporate that effort into benchmark generation.

Currently, our work focuses on the generation of communication benchmarks. Our approach guarantees that the generated communi-

[1] In fact, four of the world's ten fastest supercomputers contain accelerators (`http://www.top500.org/`, November 2010).

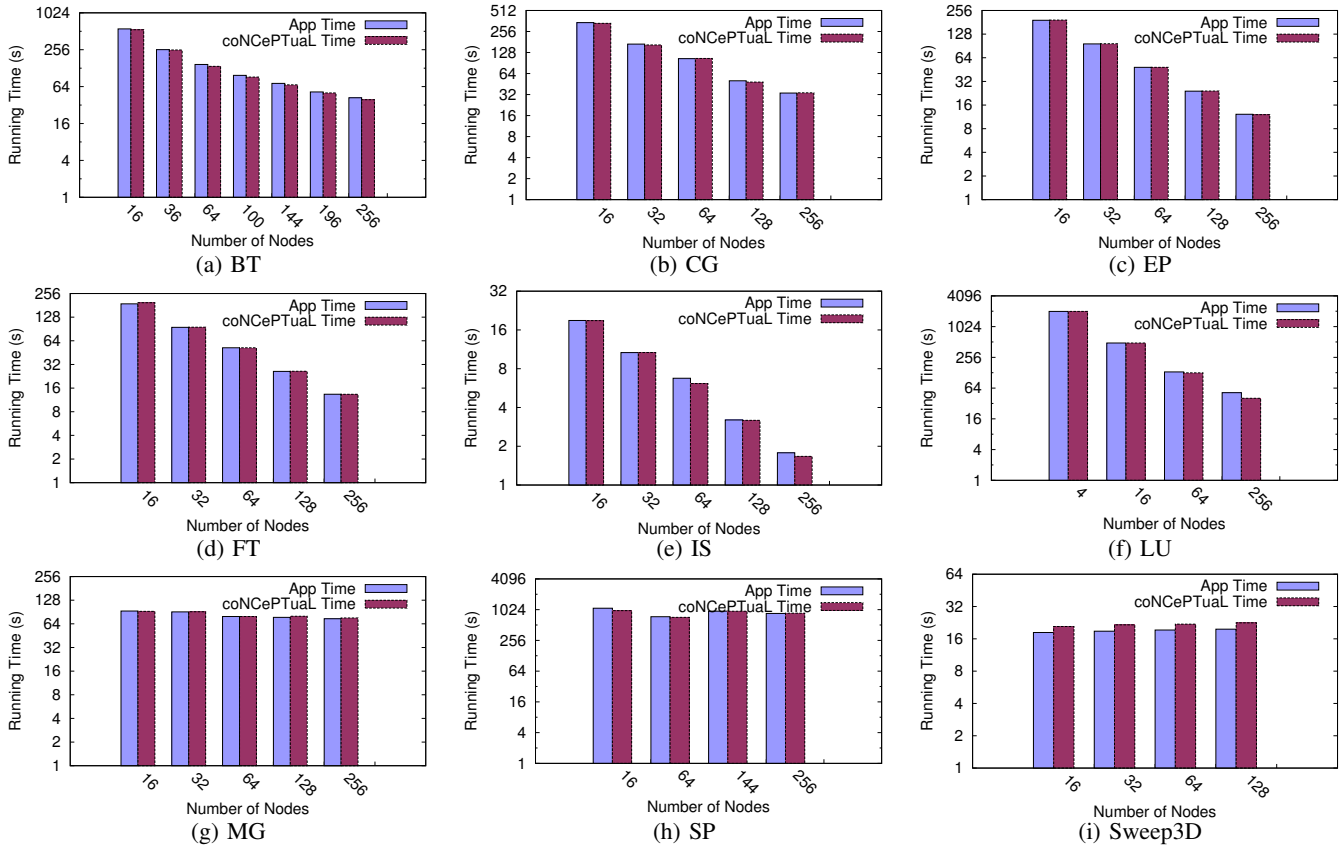


Figure 6: Time accuracy for generated benchmarks

cation is cross-platform performance-portable because we preserve the original communication pattern and can execute it natively on a target machine. However, since computation times are taken from the source machine, the computation performance does not reflect architecture-specific effects of a different platform. One advantage of mimicking computation with spin loops is that this enables studies in which computation time is explicitly varied, as in Section 5.4. Meanwhile, we are also working on scalable memory tracing to complement communication tracing. Automatic generation and replay of memory-access behavior within ScalaTrace is a subject of future work.

7. CONCLUSIONS

To bridge the gap between the performance realism of a complete application and the convenience of porting and modifying a benchmark code, we have designed, implemented, and evaluated a benchmark-generation framework that automatically generates portable, customizable communication benchmarks from parallel applications. Our approach is based on an application's dynamic behavior rather than its statically identifiable characteristics. We use ScalaTrace [12] to recover application structure from a communication trace and CONCEPTUAL [15] to express the resulting benchmarks in a readable, editable, yet executable format.[2] Algorithms we developed to assist in this process merge collective operations described by disparate source-code lines into a single call point and eliminate nondeterminism caused by wildcard receives. Empirical measurements indicate that the performance of the generated benchmarks is faithful to that of the original application.

[2]ScalaTrace and CONCEPTUAL are freely available from, `http://moss.csc.ncsu.edu/~mueller/ScalaTrace/` and `http://conceptual.sourceforge.net/`.

There are two main conclusions one can draw from this work. First, it is in fact feasible to automatically convert parallel applications into benchmark codes that accurately reproduce the applications' performance yet are easy to port, read, edit, and reason about. Second, as demonstrated in Section 5.4, nonlinear performance effects come into play as applications are modified for nascent architectures, and performance-accurate, application-specific benchmarks are an important new technology for quantifying these effects before exerting the effort involved in application porting.

The benchmarks we generate preserve all communication operations, represent applications' actual run-time behavior, and do not grow proportionally to the process count or message volume. To our knowledge, our work is the first successful attempt at automatically converting parallel applications into performance-accurate benchmarks that exhibit all of those features.

Acknowledgments

This work was supported in part by NSF grants 0937908 and 0958311 and by the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC52-06NA25396 with Los Alamos National Security, LLC.

8. REFERENCES

[1] N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Baltimore, Maryland, Nov. 16–22, 2002. IEEE Computer Society Press.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, Atlantic City, New Jersey, Apr. 18–20, 1967. ACM.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[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 *Int'l Conf. on Computational Science*, pages 526–533, May 2006.

[10] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65(108):198–199, 1992.

[11] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[12] 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.

[13] 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.

[14] S. Pakin. Reproducible network benchmarks with CONCEPTUAL. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Proceedings of the 10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 64–71, Aug. 31–Sept. 3, 2004.

[15] 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.

[16] V. Pillet, V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualize and analyze parallel code. In *Proceedings of the 18th Technical Meeting of WoTUG-18: Transputer and Occam Developments*, pages 17–31, 1995.

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

[18] S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, and R. Biswas. Scientific application-based performance comparison of sgi altix 4700, ibm power5+, and sgi ice 8200 supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 7:1–7:12, Piscataway, NJ, USA, 2008. IEEE Press.

[19] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2–3):105–121, 2008.

[20] 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.

[21] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int'l Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[22] 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.

[23] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, New Orleans, Louisiana, Nov. 13–19, 2010.

[24] 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.

[25] F. Wolf and B. Mohr. KOJAK—a tool set for automatic performance analysis of parallel applications. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, volume 2790 of *Lecture Notes in Computer Science*, pages 1301–1304, Klagenfurt, Austria, August 2003. Springer. Demonstrations of Parallel and Distributed Computing.

[26] 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.

[27] 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.

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

[29] 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.