A Methodology for Automatic Generation of Executable Communication Specifications from Parallel MPI Applications

XING WU and FRANK MUELLER, North Carolina State University
SCOTT PAKIN, Los Alamos National Laboratory

Portable parallel benchmarks are widely used for performance evaluation of HPC systems. However, because these are manually produced, they generally represent a greatly simplified view of application behavior, missing the subtle but important-to-performance nuances that may exist in a complete application. 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 computation time, while abstracting away the *details* of the computation proper. We subsequently generate benchmarks with nearly identical run-time behavior to the original applications. Results demonstrate that the generated benchmarks are in fact 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 without precedent and particularly valuable for proprietary, export-controlled, or classified application codes.

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*

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, network). While easy to port, distribute, modify, and run on a target machine, 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 [Gropp et al. 1996] 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 [Noeth et al. 2007]. 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 [Pakin 2007]. The CONCEPTUAL code can then be compiled into ordinary C+MPI code for execution on a target machine.

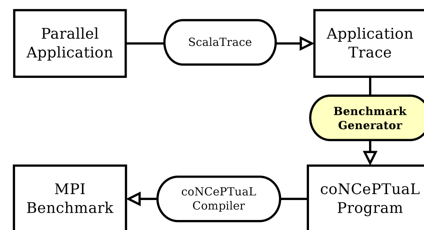


Fig. 1. Our Benchmark Generation System

We utilize ScalaTrace [Noeth et al. 2007] 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 [Pillet et al. 1995], Tau [Shende and Malony 2006], Open|SpeedShop [Schulz et al. 2008], Vampir [Nagel et al. 1996], and Kojak [Wolf and Mohr 2003], 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 [Pakin 2007] 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.) CONCEPTUAL is designed specifically to facilitate the rapid creation of benchmarks that assess communication design and network performance. We embrace CONCEPTUAL for its unique features that benefit benchmark generation in various aspects.

— CONCEPTUAL has a powerful, yet concise grammar 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 and helps benchmark designers to focus on the core functionality.
— The CONCEPTUAL runtime library automatically generates an execution log to facilitate analysis and reproduction of experimental results.
— Benchmarks in CONCEPTUAL are also more portable as they can be compiled to not only C+MPI but also other combinations of languages and messaging layers.

Later in this paper we present the complete CONCEPTUAL program that our tool generated for NPB FT (Class C) of 256 MPI tasks (Figure 12). This program reproduces the communication behavior and performance characteristics of FT, which is a 2,131-line program, in only 22 lines of CONCEPTUAL code. We believe the conciseness of CONCEPTUAL makes it an ideal language for benchmark generation.

Beyond the naive conversion from traces to CONCEPTUAL codes, we also focus on eliminating ambiguity and non-determinism in the generated code. In particular, MPI allows constructs whose behavior cannot statically be determined. The use of "wildcard receives" (MPI_ANY_SOURCE) introduces randomness to the performance of the application. While application developers are encouraged to exploit the rich features of MPI, we deem some of them inappropriate for benchmarks because they degrade performance reproducibility. For this work, we designed trace-based algorithms to combine per-node collectives and to eliminate non-determinism in MPI applications. With these optimizations, we managed to make the generated benchmarks more readable and performance-reproducible.

We evaluated our benchmark generation approach with the NAS Parallel Benchmark (NPB) suite [Bailey et al. 1991] and the Sweep3D code [Koch et al. 1992]. 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%.

Beyond the straightforward benchmark generation, we also combined the benchmark generator with ScalaExtrap [Wu and Mueller 2011] to generate scalable codes that can be executed with arbitrary numbers of MPI processes. ScalaExtrap captures the communication patterns and computation times as functions of mesh dimensions from several small-scale input traces. With these functions — instead of the exact parameter values for a particular scale — we are able to generate codes that execute at different scales. We evaluated the generated scalable codes with NPB FT and BT. The experimental results show that the generated scalable codes are able to correctly reproduce the communication patterns at different scales. For the timing accuracy, the mean absolute percentage error for a single benchmark across different scales is only 5.12% for BT and 3.42% for FT. 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, (3) an approach and algorithm for ensuring performance repeatability by introducing determinism into benchmarks generated from nondeterministic applications, and (4) an approach for generating scalable codes from application traces by utilizing trace extrapolation.

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.

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 [Brunst et al. 2001], Extrae/Paraver [Pillet et al. 1995], and tools based on the Open Trace Format [Knüpfer et al. 2006] 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 [Knupfer 2005] 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 [Casas et al. 2007]. While this approach could facilitate trace analysis, it is lossy and thus not suitable for benchmark generation.

Xu et al. construct coordinated *performance skeletons* to predict application execution time in new hardware environments [Xu et al. 2008; Xu and Subhlok 2008]. This work exhibits the following fundamental differences from ours: 1) A key aspect of performance skeletons is that they drop "local" communication (communication outside the dominant pattern) and only capture a single, dominant communication pattern by filtering a trace into aggregate information equivalent to profiling (communication matrix). Similarly, PAS2P [Panadero et al. 2013] extracts a subset of communication patterns, but not all of them, as we do, and generates signature kernels with associated weights per pattern to reflect an application mix for execution time prediction. We handle local/irregular communication via lossless tracing and generate concise and readable benchmarks from such lossless traces, which is non-trivial; Xu et al. generate code only for a single hot-spot communication pattern. 2) Fully preserving all the communication events ensures the correctness of the generated benchmarks while dropping events may introduce errors such as deadlock and/or mismatching send/receive operations. We carefully address these problems to ensure that our tool is applicable to real-world scenarios. 3) In some applications, such as NPB MG, minor communication patterns can become the dominant pattern as the application scales. To generate performance-accurate benchmarks, no communication events should be dropped. 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 [Ertvelde and Eeckhout 2008]. 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 [Shao et al. 2006], 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 [Zhai et al. 2009]. 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.

Previous work also focused on benchmark synthesis using low-level workload characteristics [Bell and John 2005; Wong and Morris 1988; Sreenivasan and Kleinman 1974]. For example, Bell et al. [Bell and John 2005] synthesize representative test cases from workload characteristics, such as instruction sequences, branch predictability, and cache miss rates, of an application binary. Wong et al. concentrate on the locality of references and use the LRU cache hit function as a workload characterization for benchmark synthesis [Wong and Morris 1988]. Sreenivasan et al. generate representative synthetic workload by matching the joint probability density of the real workload with that of the synthetic workload [Sreenivasan and Kleinman 1974].

Besides benchmark generation and synthesis, our work is also relevant to performance modeling and prediction [Chen et al. 2011; İpek et al. 2006; Kerbyson et al. 2001; Bailey and Snavely 2005; Snavely et al. 2002]. For example, Chen et al. describe a modeling and analysis framework designed to automatically estimate the resource demand for a given performance target using program characteristics [Chen et al. 2011]. İpek et al. use artificial neural networks (ANNs) to predict application performance when the configuration varies [İpek et al. 2006].

3. BACKGROUND

Our benchmark generation approach utilizes the ScalaTrace infrastructure [Noeth et al. 2007] to extract the communication behavior of the target application. Based on the application trace, we generate benchmarks in coNCePTuaL [Pakin 2007], 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 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: $\{\langle rank \rangle$, MPI_Irecv, LEFT$\}$
RSD2: $\{\langle rank \rangle$, MPI_Isend, RIGHT$\}$
RSD3: $\{\langle rank \rangle$, MPI_Waitall$\}$

denote the MPI_Send, MPI_Receive, and MPI_Waitall operations in a single loop iteration, where $\langle rank \rangle$ takes on each value from 0 to $N-1$ in turn. ScalaTrace then detects the loop structure and

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

Fig. 2. Sample Code for RSD and PRSD Generation

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}

Of course, enumerating all participating ranks one by one is not scalable. Hence, ScalaTrace further compresses the participating list with a *ranklist* representation. Using the EBNF meta-syntax, a ranklist is represented as

$$\langle dimension\ start_rank\ iteration_length\ stride\{iteration_length\ stride\}^{*}\rangle,$$

where *dimension* is the dimension of the group, *start_rank* is the rank of the starting node, and the *iteration_length stride* pair is the iteration and stride of the corresponding dimension. As an example, consider the row-major grid topology in Figure 3. The shaded nodes form a communication group. This group is represented as *ranklist* <*2 6 3 5 3 1*>, where the tuple indicates that this communication group is a 2-dimensional area starting at node 6 with 3 iterations of stride 5 in the *y* dimension and 3 iterations of stride 1 in the *x* dimension, respectively. Since this encoding scheme takes node placement into account, it naturally reflects the spatial characteristics of a communication group.

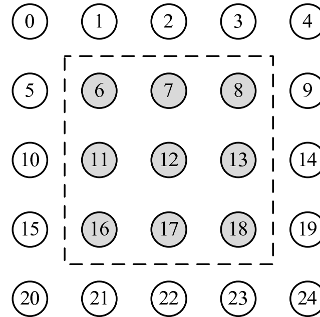


Fig. 3. Ranklist Representation

Besides communication tracing, ScalaTrace also stores application computation times in a scalable way [Ratn et al. 2008]. 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 across all loop iterations and all nodes. ScalaTrace's path-aware compression distinguishes delta times of different execution paths. Therefore, in the cases where the time spent in computation prior to the first statement of a loop differs significantly from the time spent in the subsequent iterations, ScalaTrace can still achieve good compression without sacrificing performance accuracy. This feature of ScalaTrace enables the generated CONCEPTUAL code always to reflect the loop structures and capture the path-specific

execution times irrespective of their time variance by having conditionals on loop iterators, as illustrated by the following coNCePTuaL code snippet:

```
FOR EACH i IN {1, ..., n} {
    IF i<>1 THEN ALL TASKS COMPUTE FOR t1 THEN
    IF i=1 THEN ALL TASKS COMPUTE FOR t2 THEN
    ...
}
```

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 [Pakin 2004].

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 compressed trace formats as RSDs and PRSDs (see Section 3.1), 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 [Wu et al. 2012].

Most of the conversion from RSDs and PRSDs to coNCePTuaL code is straightforward. An RSD representing point-to-point communication (blocking or non-blocking) 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 (see Section 4.2).

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*. As an example, if a reduce operation in which tasks 0, 3, 6, 9, ... participate is implemented with multiple statements in the source code and recorded as multiple events in the trace, we expect these statements to be combined into a single line of generated CONCEPTUAL code: "`TASKS t SUCH THAT 3 DIVIDES t REDUCE A DOUBLEWORD TO TASK 0`". As such, it suffices 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, which is invoked separately per node.

MPI programs are allowed to be nondeterministic. That is, they can observe different communication patterns and different performance from run to run even given the same inputs. The primary source of nondeterministic behavior in MPI is "wildcard receives" (MPI_ANY_SOURCE), which can receive messages from any sender. 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.

Consider, for example, if the code shown above were used to compare the performance of two clusters, one containing slow processors and a slow network and one containing fast processors and a fast network. If the first cluster happened to deliver rank 1's message before rank 0's while the second cluster happened to deliver those messages in the reverse order, one might conclude that the first cluster is the faster of the two, although this conclusion is based on happenstance, not on fundamental performance characteristics of the two clusters.

To ensure that our generated benchmarks lead to fair, reproducible performance comparisons our benchmark generator selects *one* of the possible executions and forces that *always* to be used. Section 4.4 presents our algorithm for identifying a valid execution. The result from running our algorithm is a deterministic, performance-reproducible benchmark that accurately captures what

the original application *might* do without being susceptible to the complete set of vagaries that impact the application's performance.

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). Therefore, for the MPI collectives that are directly supported by CONCEPTUAL, such as MPI_Bcast, MPI_Reduce, and MPI_Alltoall, we generate the corresponding CONCEPTUAL MULTICAST and REDUCE statements. For the unsupported MPI collectives, we 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 I presents the substitutions we made.

Table I. Mapping of MPI Collectives to CONCEPTUAL

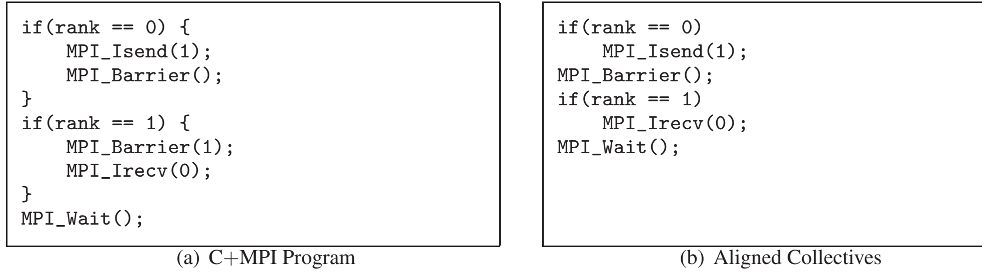
MPI collective	CONCEPTUAL implementation
Allgather	`REDUCE` + `MULTICAST`
Allgatherv	`REDUCE` with averaged message size + `MULTICAST`
Alltoallv	`MULTICAST` with averaged message size
Gather	`REDUCE`
Gatherv	`REDUCE` with averaged message size
Reduce_scatter	n many-to-one `REDUCE`s with different message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

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 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 4 illustrates the intention, for clarity using C+MPI (with the omission of most MPI arguments) instead of RSDs. Figure 4(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 4(b).

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 4. Rather, it follows the sequence of steps presented in Algorithm 1 to



(a) C+MPI Program (b) Aligned Collectives

Fig. 4. Combining Collectives across Statements

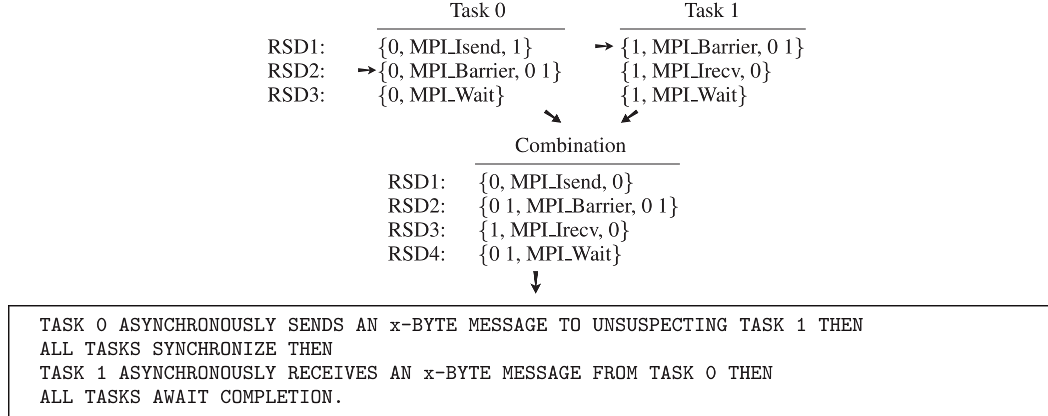


Fig. 5. Operation of Algorithm 1

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 5 for RSDs corresponding to the C+MPI code in Figure 4, 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 5). 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 [Noeth et al. 2007] to the output RSD queue each time a new RSD is appended to the queue.

The complexity of Algorithm 1 is $O(e)$, where $e = \Sigma_{i=0}^{n-1} e_i$ is the total number of MPI events on all the nodes and e_i is the number of communication events per node. This can be derived from the fact that Algorithm 1 traverses every event in the trace exactly once for each node. In Algorithm 1, the *for* loop in line 2 initializes the iterator to the head RSD for each node. During execution, the *while* loop in line 12 always moves the iterator forward by exactly one event in each iteration. In

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
```

case the traversal is blocked at a collective, a context switch happens at line 27. When the call to *Align* returns, the traversal proceeds to the next event. In addition, since MPI_Finalize is handled as a collective that all nodes participate in (line 23), the traversal is performed for all the nodes.

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.

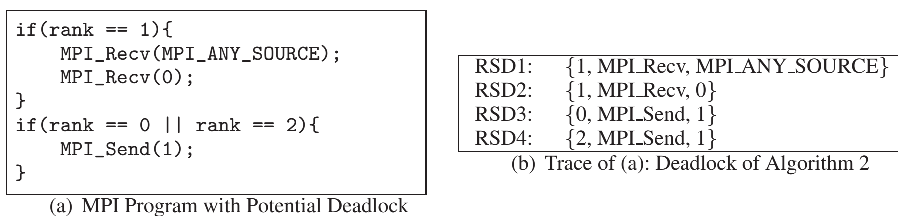
4.4. Eliminating Non-determinism

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 decom-

position [Bailey et al. 1991], 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, our algorithm for eliminating nondeterminism (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 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 $e_{i_1 x k_1}, e_{i_2 x k_2}, e_{i_3 x k_3}, \ldots$ 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 non-blocking 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 in the *while* loop at line 12, the complexity is $O(e)$, where $e = \Sigma_{i=0}^{n-1} e_i$ is the total number of MPI events on all the nodes. 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 6(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 6(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) Trace of (a): Deadlock of Algorithm 2

Fig. 6. Potential Deadlock

To prevent Algorithm 2 from hanging when nondeterminism in the original application introduces a deadlock condition, our benchmark generator extends Algorithm 2 to detect deadlock conditions

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
```

during trace traversal. Notice that these deadlocks stem from incorrect MPI semantics of the application, not from flaws in our tracing or code-generation frameworks. 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 proxied to the peer of another non-blocking communication on which node n is waiting. If the peers of all the pending non-blocking 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 [Vo et al. 2010], 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. The Generation of Scalable Benchmarks

An inherent drawback of the trace-based benchmark generation approach is that the generated code is not scalable in a parametric sense; it can be executed only with the exact number of MPI tasks with which the trace was collected. To alleviate this shortcoming and allow an arbitrary number of MPI tasks for invocation, we incorporated our benchmark generator with ScalaExtrap, our prior work that extrapolates a large communication trace (a trace with large number of MPI tasks) from a series of smaller traces.

4.5.1. ScalaExtrap. This section briefly summarizes ScalaExtrap. A complete discussion on ScalaExtrap can be found in our prior work [Wu and Mueller 2011]. ScalaExtrap is a tool that implements a methodology to automatically extrapolate a large trace from a series of smaller traces for SPMD codes with a stencil/mesh communication pattern. ScalaExtrap assumes that MPI parameters such as *source*, *dest*, and *count* are linearly correlated with the dimension sizes x, y, and z of the communication topology. Given a set of input traces of different node sizes, ScalaExtrap constructs a set of linear equations in which unknown coefficients of x, y, z reflect the correlation. ScalaExtrap uses Gaussian elimination to solve the set of linear equations. The obtained coefficients are later used with the known x, y, z sizes of an application at large scale to calculate actual values of the MPI parameters of interest. In addition, ScalaExtrap utilizes the curve fitting approach to extrapolate the lengths of the computational regions in the application so that the timing behavior under scaling is also captured in the extrapolated trace.

We combined our benchmark generator with ScalaExtrap by introducing the use of an auxiliary trace. We extended ScalaExtrap such that for each MPI parameter, a function of the processor mesh's x, y, and z dimensions and the solved coefficients is stored in a separate trace in addition to the extrapolated trace of a specific node size. We store formulae for all the trace parameters including

(1) MPI parameters such as *source, dest, count, etc.*,
(2) application parameters such as the loop iteration counts, and
(3) trace parameters such as the ranklists (see Section 3.1).

In addition, the fitting curves for the computation times are also stored in the auxiliary trace. The auxiliary trace is structurally similar to a normal ScalaTrace trace so that the formulae in the auxiliary trace can be easily mapped to parameters once a trace size is selected for an actual run.

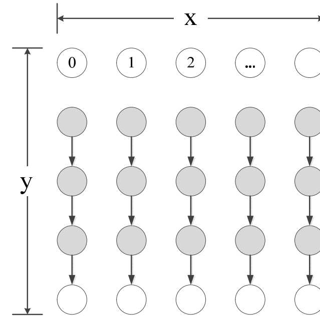


Fig. 7. Communication Pattern of a 2D Stencil Code

To auto-generate the extrapolation benchmark of an application, both the extrapolated trace of a specific size and the auxiliary trace are read into the code generator. During generation, whenever there is a formula in the auxiliary trace for a certain parameter in the normal trace, the code generator generates a CONCEPTUAL communication statement that uses the scalable representation in the auxiliary trace. As an example, Figure 7 shows a 2D communication pattern in which the gray nodes send messages to the nodes below them. ScalaExtrap identifies that the relative distance d of the receiving nodes is $d = x$, *i.e.,* a linear correlation with the size of the x dimension. Hence, a scalable CONCEPTUAL send statement using a formula instead of a particular value as the destination will be generated as

```
TASKS t1 SUCH THAT ... SEND ... TO
        TASKS t2 SUCH THAT t2 = t1 + x,
```

where x is initialized according to the total number of MPI tasks of a particular run. Moreover, since all the gray nodes follow the same communication pattern, their send events are merged into a single trace record with an associated ranklist

$\langle 2 \quad x \quad y-2 \quad x \quad x \quad 1 \rangle$

indicating the participating MPI tasks. During code generation, this ranklist is used to generate the range expression that defines the source tasks of this MPI event. CONCEPTUAL implements the ranklist with a list comprehension using n variables to recursively define the iteration count and stride for each of the n dimensions in a ranklist. By utilizing list comprehensions, we can conveniently generate the scalable ranklist representation above with the CONCEPTUAL range expression (in this case, a list comprehension [Turner 1982]) shown below:

```
TASKS t1 SUCH THAT t1 IS IN {i1+i2
                              FOR EACH i1 IN {x,2x,...,x+((y-2)-1)*x}
                                FOR EACH i2 IN {0,1,...,(x-1)*1}} SEND ...
```

Parameters other than the *source/dest* and ranklists, such as *count*, loop iterations, and computation times, are also generated with the auxiliary formulae. Because a generated benchmark cannot automatically infer the user's intentions when selecting a processor layout, the user must explicitly provide the processor mesh's x, y, and z dimensions through command-line arguments. The generated benchmark then automatically uses these values to extrapolate the various other parameters described above.

By introducing ScalaExtrap's parameter-extrapolation functionality into our benchmark-generation framework, we are able to generate CONCEPTUAL communication benchmarks that can be executed with an arbitrary valid number of MPI tasks while performance remains accurate at different processor counts. Note that generating scalable benchmarks is nontrivial due to the challenges in trace-based extrapolation such as detecting communication topology, matching trace events across scales, and inferring scale-dependent communication events. ScalaExtrap currently

focuses on stencil/mesh topology with nodes arranged in a row-major fashion, which represents the structure of many parallel applications.

4.6. 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 [Bailey et al. 1991] and for the Sweep3D neutron-transport kernel [Wasserman et al. 2000]. 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 that the results from the NPB suite and Sweep3D in this paper, combined with previous ScalaTrace experiments [Noeth et al. 2009; Wu and Mueller 2011], 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 [Saini et al. 2008], 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 [Adiga et al. 2002] 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 [Vetter and McCracken 2001], 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 [Wu and Mueller 2011] 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 shown that auto-generated benchmarks faithfully preserve 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 8 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 8 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). Close examination of the SP traces shows that the inaccuracy might be caused by the use of the average histogram value as the computation time. When the variance is large for a histogram, the replay time tends to be shorter than the original execution time if the immediately succeeding operation is a wait operation. This is a tradeoff between the conciseness and performance fidelity of the generated benchmarks.

5.4. Correctness and Timing Accuracy of Auto-Generated Extrapolation Benchmarks

By combining the benchmark generator with ScalaExtrap, we are able to generate extrapolation benchmarks under CONCEPTUAL that can be executed with an arbitrary number of MPI tasks. In this section, we evaluate the correctness of the extrapolation benchmarks in terms of their ability to retain the communication pattern under scaling. In addition, we also assess the timing accuracy of the extrapolation benchmarks. In this set of experiments, we generated extrapolation benchmarks under CONCEPTUAL for the NPB BT and FT codes. We chose BT and FT because they represent two widely used communication patterns: stencil/mesh codes and applications with collective communication, and thus demonstrate the ability of generating scalable codes for such patterns in general. BT is a 2-dimensional 7-point stencil code. Due to strong scaling, various application parameters vary across different node sizes, including the MPI parameters *source, dest,* and *count*, the loop iteration count, the ranklists for MPI events, and the computational delta times. FT performs a fast Fourier transform (FFT). Its communication workload is mainly comprised of repetitive calls of MPI_Alltoall in multiple iterations.

In the first experiment, we generated an extrapolation benchmark under CONCEPTUAL for BT of the Class D input size. We used traces of 16, 64, 144, and 256 tasks as the input of ScalaExtrap. With an extrapolated trace and the auxiliary trace generated during extrapolation, we generated an extrapolation benchmark under CONCEPTUAL. We then executed the extrapolation benchmark at different scales from 16 to 400 MPI tasks and evaluated their communication correctness and timing accuracy. To demonstrate the communication correctness, we collected the message density matrices for both the original application and the extrapolation benchmark. The heat maps of the original application and the extrapolation benchmark are identical, showing that the generated extrapolation benchmark is able to preserve the communication pattern of the original application under scaling.

In the second experiment, we evaluated the timing accuracy of the generated extrapolation benchmarks under CONCEPTUAL with different numbers of MPI tasks. For BT, we used the same scalable code that we used in the experiment described above. For FT, we used the Class C input size instead of Class D so that we can collect the input traces for ScalaExtrap starting with a minimum of 8 MPI tasks. With traces collected for 8, 16, 32, and 64 MPI processes, we generated an extrapolation benchmark for FT under CONCEPTUAL. We then executed both the original application

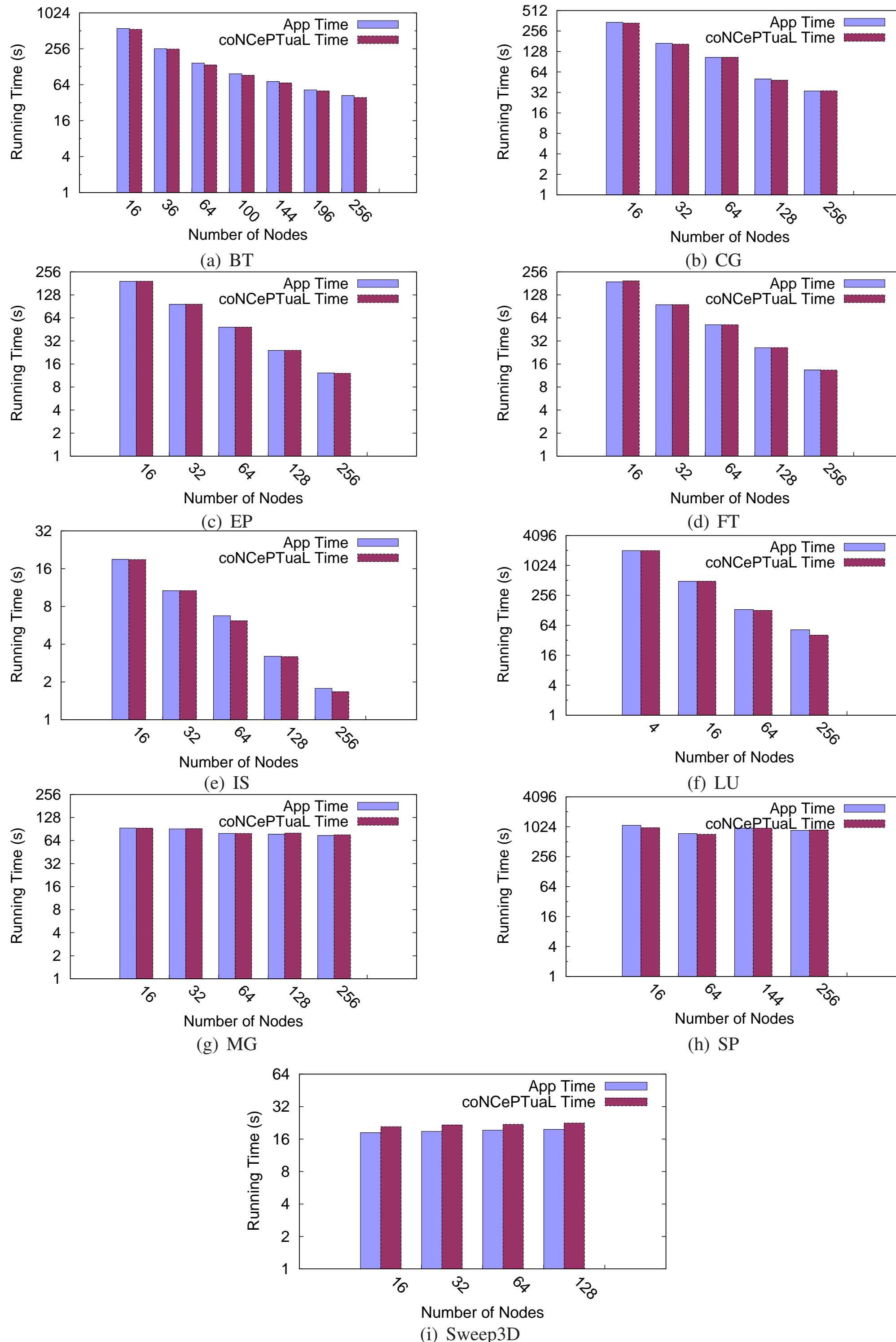
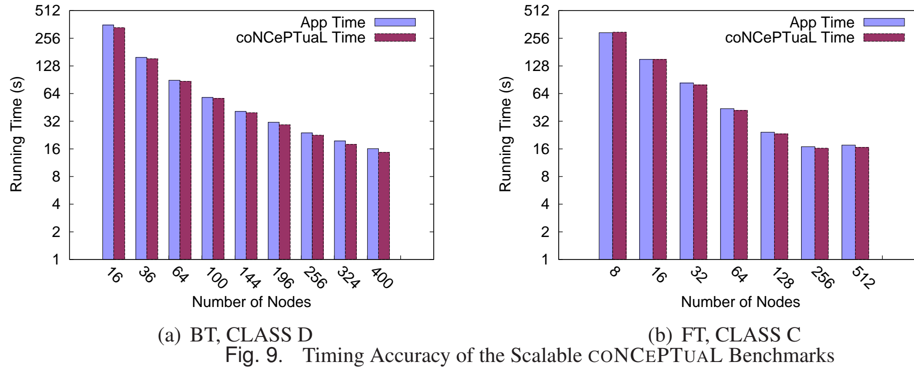


Fig. 8. Time Accuracy for Generated Benchmarks

and the extrapolation benchmarks at different scales and compared their total execution times. Figure 9 shows the experimental results. As demonstrated, the auto-generated extrapolation codes under coNCePTuaL have total execution times that closely resemble those of the original applications at each tested scale. Quantitatively, across all the tested node sizes, the mean absolute percentage errors for BT and FT are only 5.12% and 3.42%, respectively.



(a) BT, CLASS D (b) FT, CLASS C

Fig. 9. Timing Accuracy of the Scalable coNCePTuaL Benchmarks

5.5. Case Studies: 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 a set of case studies featuring what-if analysis and cross-platform performance prediction that become practical due to the novel capabilities of automatic benchmark generation. Since each scenario requires detailed analysis to interpret their behavior and the respective causes, we demonstrate these capabilities just for one exemplary program at a time.

5.5.1. Impact of Computational Speedup. 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 [Amdahl 1967], 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, which is computation bound in nature at this input size and core count. We chose BT because it is a stencil code consisting almost exclusively of asynchronous point-to-point communication operations, which leads to the optimal computation speedup that we want to study. 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 10.

Reading Figure 10 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 sub-linear 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

[1]In fact, four of the world's ten fastest supercomputers contain accelerators (http://www.top500.org/, June 2014).

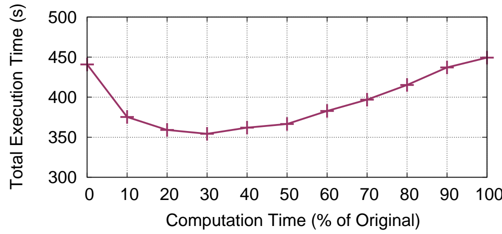


Fig. 10. Communication Performance of BT

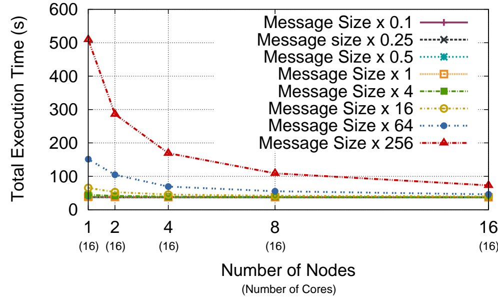


Fig. 11. Impact of Communication Performance on BT

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 10 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).

5.5.2. Impact of Communication Performance. In the second case study, we evaluate the impact of network bandwidth on the overall performance of an application. We keep the computation times and communication pattern as they are but vary the message sizes to mimic the impact of different problem sizes or different data decomposition on the communication behavior. For each configuration, we further evaluate the impact of the maximum bandwidth by varying the number of MPI tasks on each node.

For this experiment, we used the ARC cluster with an Infiniband QDR interconnect. We generated the CONCEPTUAL benchmark for the NPB BT code with 16 MPI tasks and class A input size. BT is computation bound in nature at this input size and core count (just like the preceding BT experiment), but we will assess the point where it becomes communication bound. The generated benchmark allows us to modify the generated benchmark by manually customizing message sizes from 0.1x to 256x of their original sizes. We then executed the modified codes with a node count from 1 to 16, which increases the available bandwidth per MPI task as node counts become higher. Figure 11 shows the overall execution time of the modified benchmark for each configuration. Each curve represents the results obtained for a different message size.

We observe that, in the figure, the curves for 0.1x, 0.25x, 0.5x, 1x, and 4x message sizes overlap with each other. This indicates, for this particular application and this particular system, that neither increasing the available bandwidth nor decreasing the message size may improve overall

performance. Therefore, optimizations in the computation design are required to accelerate this application. Yet, when the message size is 16 times the original size, the communication starts to saturate the network bandwidth. From there on, notable speedup can be observed if more nodes and hence higher overall bandwidth are allocated for this application.

5.5.3. Impact of Collective Implementation. With easy-to-modify CONCEPTUAL benchmarks, application developers not only can modify the parameter values for what-if analysis, they can also quickly modify the communication implementation for performance comparison. This is made practical by the conciseness of CONCEPTUAL programs. For example, no buffer allocation or explicit request handle management is necessary. In addition, because computation is removed and replaced with idle spinning in the generated benchmarks, the generated benchmarks tend to be much shorter and simpler than the original application. For example, the CONCEPTUAL version of FT has only 22 lines of code (Figure 12) while the original FT code has 2,131 lines of code. As Figure 8(d) shows, there is no qualitative difference between the performance of the original code and the generated benchmark, even though the latter is only 1% of the length of the former. With the generated communication skeleton, application developers can readily assess a communication design without having to modify the entire parallel algorithm and keep track of the changes in multiple source files.

```
All tasks synchronize then
Task 0 resets its counters then
All tasks compute for 52 microseconds
Task 0 multicasts a 12-byte message
        to all other tasks
All tasks compute for 0 microseconds
Task 0 multicasts a 4-byte message
        to all other tasks
All tasks compute for 295410 microseconds
All tasks multicasts a 32768-byte message
        to all other tasks
All tasks compute for 133044 microseconds
All tasks synchronize
All tasks compute for 317381 microseconds
All tasks multicasts a 32768-byte message
        to all other tasks
```

```
For each i1 in {1, ..., 20} {
  If i1 <> 1 then All tasks compute
          for 167332 microseconds then
  If i1 = 1 then All tasks compute
          for 312861 microseconds then
  All tasks multicasts a 32768-byte message
          to all other tasks then
  All tasks compute
          for 254450 microseconds then
  All tasks reduce 4 integers to task 0
}
All tasks  compute for 24 microseconds
All tasks synchronize then
Task 0 logs elapsed_usecs/1E6 as "Seconds"
```

Fig. 12. Complete auto-generated CONCEPTUAL Program for NPB FT (Class C) of 256 MPI Tasks

In this experiment, we evaluate different communication implementations for the NPB FT code. FT solves a three-dimensional partial differential equation using the fast Fourier transform (FFT). It uses MPI_Alltoall to exchange data among all the participating MPI tasks in each timestep. Alternatively, point-to-point communication routines can also be used to implement the same communication pattern. We compared the performance of these two different implementations by modifying the generated FT benchmark. To migrate from the MPI_Alltoall implementation to the point-to-point implementation with MPI_Isend, we do not need to understand the FFT algorithm to find out which buffer should be changed or to implement the all-to-all style point-to-point communication for the communicators representing the user-defined processor layout. Instead, only one line of the CONCEPTUAL code needs to be changed from

```
All tasks multicasts a xxx-byte message to all other tasks
```

to

```
All tasks asynchronously send a xxx-byte message to all other tasks
All tasks await completion
```

where the matching receive operations will be posted automatically by the CONCEPTUAL runtime framework.

We compared the performance for different implementations by executing the CONCEPTUAL FT codes on ARC. Figure 13 plots the overall execution times for different implementation strategies with different number of nodes. Not surprisingly, the MPI_Alltoall implementation outperforms the point-to-point implementation. Because the point-to-point implementation blindly exchanges data between each pair of nodes without any communication pattern optimization, it suffers from scalability constraints: the overall runtime of the point-to-point version with 256 MPI tasks is 46.2% longer than the collective version even though its performance is 0.5% slower for 16 nodes.

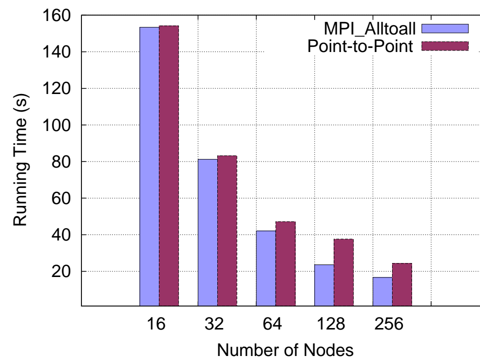


Fig. 13. All-to-all Variants for FT

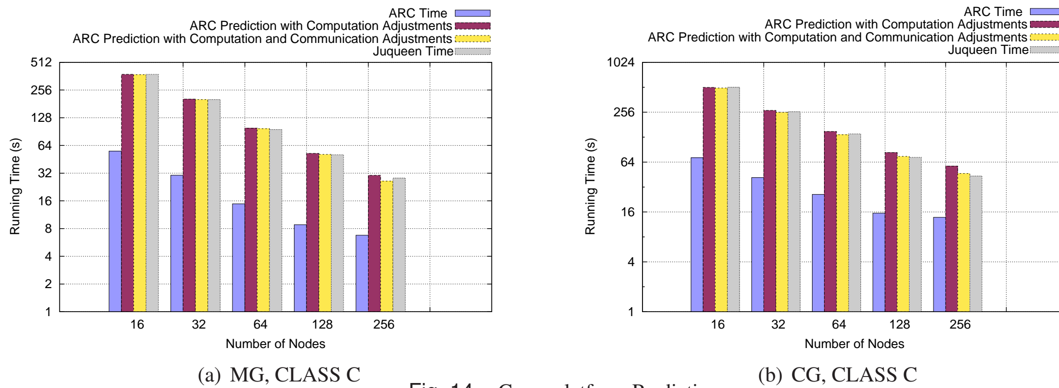
With this experiment, we show the ability to utilize the generated benchmark to assess the efficiency of different communication options in rapid prototyping. This becomes feasible due to the ease of modifying the generated CONCEPTUAL benchmark for what-if analysis. In practice, developers may evaluate various optimizations, such as replacing point-to-point message-based multicasting with collectives within a communicator or replace a single collective with sets of group-based collectives using modified communication patterns, which is facilitated by CONCEPTUAL benchmarks. Moreover, developers may even completely change the communication design for evaluation before modifying the implementation of the associated data and problem decomposition in the original application. Modifying the output of conventional MPI tracing tools is tedious and error-prone. For example, given that MPI_Alltoall is called 22 times by each MPI process, altering the communication pattern of a 256-rank FT run requires that 22×256 locations be changed. In contrast, our approach of using CONCEPTUAL benchmarks generated from compressed traces requires modifying only one statement and hence significantly facilitates this process.

5.5.4. Cross-platform Performance Prediction. The benchmark generation approach presented in this paper may also benefit system designers and procurers by providing a means for fast cross-platform performance prediction for existing or even future systems. In this section, we present our prediction results for two different architectures, ARC and Juqueen. Juqueen is an IBM Blue Gene/Q system with 131,072 cores on 8,192 compute nodes, 16 GB SDRAM-DDR3 per node, and a 5D Torus interconnect. In this set of experiments, we predict the performance of the NPB CG and MG codes on Juqueen by modifying and executing the generated CONCEPTUAL benchmarks on ARC. For these experiments, we pretend that Juqueen is a future supercomputer for which one compute node is available and the network performance is known. In practice, the same approach can also be used for cross-platform prediction between existing systems.

The runtime of a parallel application consists of the sequential computation time in each process, the communication time between processes, and their convolution. To predict the runtime of an application on a future platform, we generate CONCEPTUAL benchmarks of different node sizes that reflect the computational speedup with modified sleep times, execute the generated benchmarks on an existing system, and adjust the communication times to estimate the total runtime. In this set of experiments, we obtained the computational speedup by executing the application on only one node on each of the existing (ARC) and the future (Juqueen) platforms. The communication speedup was

calculated by performing a ping-pong test on both systems. To adjust the total runtime, mpiP was utilized to measure the time spent in MPI communication events. Hence, with the known communication time T_{comm}, communication speedup s, and the computation time T_{comp} of the simulation run, the predicted runtime T can be calculated with equation $T = T_{comm} \times s + T_{comp}$. In case that the interconnect on the future platform is not yet available, estimation or analytical modeling results can be used instead.

Figure 14 shows the cross-platform prediction results. According to the single-node computational speedup tests, ARC is 5.9 times faster than Juqueen for MG and 6.2 times faster for CG. Accordingly, the CONCEPTUAL benchmarks were generated to reflect the speedup. We then measured the communication speedup by performing ping-pong tests for messages of different sizes on both systems. The message sizes chosen for this test are the send volumes of the dominating send operations in the applications. The communication speedup is then used together with the mpiP results (not presented) to calculate the adjusted total runtime prediction shown in Figure 14. Compared to actual runtimes for MG and CG on Juqueen, our execution time predictions match closely, with an accuracy of 97.72% for MG and 96.81% for CG.



(a) MG, CLASS C (b) CG, CLASS C

Fig. 14. Cross-platform Prediction

This experiment demonstrates the feasibility to perform such experiments—enabled by our benchmark generation tools—via quick cross-platform performance prediction for either existing or future HPC systems, yet without porting the actual applications to those platforms.

5.6. Limitations

Automatic generation of executables from traces is subject to limitations. As a premise, traces should be scalable in the sense that their trace file size does not significantly change as the number of time steps of iterative algorithms or the number of parallel tasks is increased. Absence of change (constant trace file sizes) irrespective of these two parameters is an indication for scalability. Sub-linear increases in trace file sizes generally result in conditionals to distinguish irregular behavior across nodes (e.g., parameter differences such as send volumes or communication events only triggered by a subset of tasks). Super-linear trace file size increases generally result in excessive conditionals that will skew the replay accuracy of generated executables relative to their original program counterpart. UMT2k [LLNL 2002] is one example of such a benchmark, where the super-linear growth in trace file size can be observed not just for uncompressed but also node-only intra compression and global inter-node compression (Figure 15).

Phase changes in applications, on the other hand, are handled by our framework as we recognize new, repetitive values as a new pattern when dynamically compressing traces. For example, if an application utilizing Adaptive Mesh Refinement (AMR) changes communication from one pattern to another, the phase change is recognized. However, some ARM codes contain inner loops whose iteration counts vary from one timestep to another. Such behavior results in super-linear trace size growth and causes generated executions from traces to become inaccurate. This behavior can be countered by probabilistic trace encodings [Wu et al. 2011]. The POP code [Jones et al. 2005] is

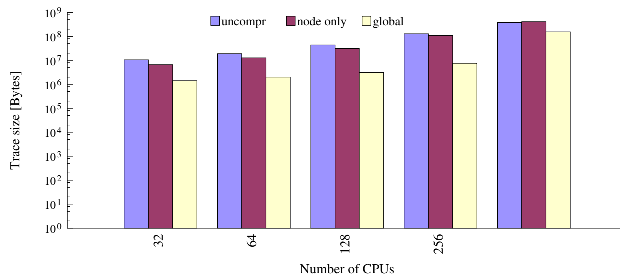


Fig. 15. UMT2k Trace File Size on BlueGene/L

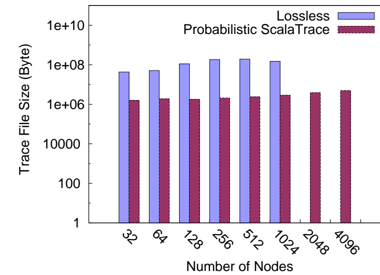


Fig. 16. POP on BlueGene/L

an example for such a code with varying inner loop trip counts, where only probabilistic traces provide linear growth of trace files (see Figure 16) while precise (lossless) traces increase super-linearly. In the future generation of executables could be extended to probabilistic traces to extend the applicability of our approach to a wider class of programs.

In general, any generated executable reflects the behavior of its original benchmark for a specific set of inputs for its traced execution. Different inputs generally result in different traces for the same application and, hence, different generated executables. If, however, a parameter is changing on each event invocation in a randomized manner, trace compression will deteriorate in terms of trace size explosion making the generation of executables infeasible and their runtimes inaccurate.

6. DISCUSSION AND FUTURE WORK

Currently, our work focuses on the generation of communication benchmarks. Our approach guarantees that the generated communication 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 5.5 of the paper. 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 small 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 [Noeth et al. 2007] to recover application structure from a communication trace and CONCEPTUAL [Pakin 2007] 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.

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 Sec-

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

tion 5.5, 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.

REFERENCES

N. R. Adiga and others. 2002. An Overview of the BlueGene/L Supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, Baltimore, Maryland. DOI:http://dx.doi.org/10.1109/SC.2002.10017

Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*. ACM, Atlantic City, New Jersey, 483–485. DOI:http://dx.doi.org/10.1145/1465482.1465560

D.H. Bailey and A. Snavely. 2005. Performance Modeling: Understanding the Present and Predicting the Future. In *Euro-Par Conference*.

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. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications* 5, 3 (Fall 1991), 63–73. citeseer.ist.psu.edu/article/bailey94nas.html

R Bell and L. John. 2005. Improved Automatic Testcase Synthesis for Performance Model Validation. In *Int'l Conf. on Supercomputing*. 111–120.

Holger Brunst, Hans-Christian Hoppe, Wolfgang E. Nagel, and Manuela Winkler. 2001. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach.. In *International Conference on Computational Science (2)*. 751–760.

Marc Casas, Rosa Badia, and Jesus Labarta. 2007. Automatic structure extraction from MPI applications tracefiles. In *Euro-Par Conference*.

Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. 2011. Modeling program resource demand using inherent program characteristics. *SIGMETRICS Perform. Eval. Rev.* 39, 1 (June 2011), 1–12. DOI:http://dx.doi.org/10.1145/2007116.2007118

Luk Van Ertvelde and Lieven Eeckhout. 2008. Dispersing proprietary applications as benchmarks through code mutation. In *Architectural Support for Programming Languages and Operating Systems*. 201–210.

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

Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2006. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 195–206.

P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque. 2005. Practical performance portability in the Parallel Ocean Program (POP): Research Articles. *Concurr. Comput. : Pract. Exper.* 17, 10 (2005), 1317–1327. DOI:http://dx.doi.org/10.1002/cpe.v17:10

D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. 2001. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Supercomputing*.

Andreas Knupfer. 2005. Construction and Compression of Complete Call Graphs for Post-Mortem Program Trace Analysis. In *International Conference on Parallel Processing*. 165–172.

A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. 2006. Introducing the Open Trace Format (OTF). In *Int'l Conf. on Computational Science*. 526–533.

K. R. Koch, R. S. Baker, and R. E. Alcouffe. 1992. 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 (1992), 198–199.

LLNL 2002. The ASCI Purple Benchmarks. (2002). http://www.llnl.gov/asci/purple/benchmarks.

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

M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. 2007. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *International Parallel and Distributed Processing Symposium*. DOI:http://dx.doi.org/10.1145/1188455.1188605

M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. 2009. ScalaTrace: Scalable Compression and Replay of Communication Traces in High Performance Computing. *Journal of Parallel Distributed Computing* 69, 8 (Aug. 2009), 969–710.

Scott Pakin. 2004. Reproducible Network Benchmarks with CONCEPTUAL. In *Proceedings of the 10th International Euro-Par Conference (Lecture Notes in Computer Science)*, Marco Danelutto, Domenico Laforenza, and Marco Vanneschi (Eds.), Vol. 3149. 64–71.

Scott Pakin. 2007. The Design and Implementation of a Domain-Specific Language for Network Performance Testing. *IEEE Transactions on Parallel and Distributed Systems* 18, 10 (Oct. 2007), 1436–1449. DOI:http://dx.doi.org/10.1109/TPDS.2007.1065

Javier Panadero, Alvaro Wong, Dolores Rexachs, and Emilio Luque. 2013. A Tool for Selecting the Right Target Machine for Parallel Scientific Applications. *Procedia Computer Science* 18 (2013), 1824–1833.

Vincent Pillet, Vincent Pillet, Jesus Labarta, Toni Cortes, and Sergi Girona. 1995. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of the 18th Technical Meeting of WoTUG-18: Transputer and Occam Developments*. 17–31.

P. Ratn, F. Mueller, Bronis R. de Supinski, and M. Schulz. 2008. Preserving Time in Large-Scale Communication Traces. In *Int'l Conf. on Supercomputing*. 46–55.

Subhash Saini, Dale Talcott, Dennis Jespersen, Jahed Djomehri, Haoqiang Jin, and Rupak Biswas. 2008. 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*. IEEE Press, Piscataway, NJ, USA, 7:1–7:12.

Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. 2008. Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. *Scientific Programming* 16, 2–3 (2008), 105–121. DOI:http://dx.doi.org/10.3233/SPR-2008-0256

Shuyi Shao, Alexk. Jones, and Rami Melhem. 2006. A compiler-based communication analysis approach for multiprocessor systems. In *In International Parallel and Distributed Processing Symposium*.

Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int'l Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. 2002. A Framework for Performance Modeling and Prediction. In *Supercomputing*.

K. Sreenivasan and A. J. Kleinman. 1974. On the construction of a representative synthetic workload. *Commun. ACM* 17, 3 (March 1974), 127–133. DOI:http://dx.doi.org/10.1145/360860.360863

D. A. Turner. 1982. Recursion Equations as a Programming Language. In *Functional Programming and Its Applications: An Advanced Course*, J. Darlington, Peter Henderson, and D. A. Turner (Eds.). Cambridge University Press, 1–28.

J. Vetter and M. McCracken. 2001. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2010. 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. DOI:http://dx.doi.org/10.1109/SC.2010.7

Harvey Wasserman, Adolfy Hoisie, and Olaf Lubeck. 2000. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures using Multidimensional Wavefront Applications. *The International Journal of High Performance Computing Applications* 14 (2000), 330–346.

F. Wolf and B. Mohr. 2003. KOJAK—A Tool Set for Automatic Performance Analysis of Parallel Applications. In *Proc. of the European Conference on Parallel Computing (Euro-Par) (Lecture Notes in Computer Science)*, Vol. 2790. Springer, Klagenfurt, Austria, 1301–1304. Demonstrations of Parallel and Distributed Computing.

W.S. Wong and R.J.T. Morris. 1988. Benchmark synthesis using the LRU cache hit function. *Computers, IEEE Transactions on* 37, 6 (jun 1988), 637 –645. DOI:http://dx.doi.org/10.1109/12.2202

X. Wu, V. Deshpande, and F. Mueller. 2012. ScalaBenchGen: Auto-Generation of Communication Benchmark Traces. In *International Parallel and Distributed Processing Symposium*. DOI:http://dx.doi.org/DOI10.1109/IPDPS.2012.114

Xing Wu and Frank Mueller. 2011. ScalaExtrap: Trace-based Communication Extrapolation for SPMD Programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. 2011. Probabilistic Communication and I/O Tracing with Deterministic Replay at Scale. In *International Conference on Parallel Processing*. 196–205.

Qiang Xu, Ravi Prithivathi, Jaspal Subhlok, and Rong Zheng. 2008. *Logicalization of MPI Communication Traces*. Technical Report UH-CS-08-07. Dept. of Computer Science, University of Houston.

Qiang Xu and Jaspal Subhlok. 2008. Construction and Evaluation of Coordinated Performance Skeletons. In *International Conference on High Performance Computing*. 73–86.

J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. 2009. FACT: Fast Communication Trace Collection for Parallel Applications Through Program Slicing. In *Proceedings of SC'09*. 1–12.