A Methodology for Automatic Generation of Executable Communication Specifications from Parallel MPI Applications (Supplementary Material)

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

1. INTRODUCTION

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:

— 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.

Figure 3 shows the complete CONCEPTUAL program for NPB FT (Class C) of 256 MPI tasks. It uses only 22 lines of code to produce the communication workload of FT, which is a 2,131-line program. We believe the conciseness of CONCEPTUAL makes it an ideal language for benchmark generation.

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; Ïpek 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]. Ïpek et al. use artificial neural networks (ANNs) to predict application performance when the configuration varies [Ïpek 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 1 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 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 1, because each MPI routine is called with the

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

Fig. 1. Sample Code for RSD and PRSD Generation

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 2. 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.

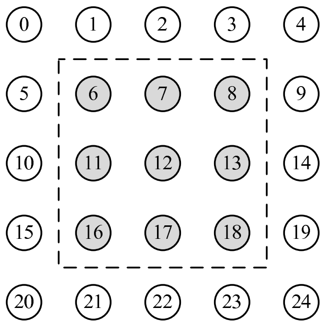

Fig. 2. Ranklist Representation

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 1:

```
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. DESIGN

4.1. 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.

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.

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

4.2. Combining Per-Node Collectives

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 farward by exactly one event in each iteration. In 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.

4.3. Eliminating Non-determinism

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, $r \ll e$.

4.4. 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.

4.5. Sources of Performance Inaccuracy

As indicated, there are a number of ways in which our benchmark generator trades off performance fidelity for an improved ability to reason about the generated code and its performance: computation times are summarized across ranks instead of being specified individually (Section 3.1); some complex MPI collectives are implemented in terms of more basic CONCEPTUAL collectives (Sec-

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
```

tion 4.1); and nondeterministic receive ordering is replaced with an arbitrary deterministic ordering (Section 4.3). In Section 5 we examine the impact of these design decisions in the context of a suite of test programs.

5. EVALUATION

5.1. 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],

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 $L_1$ and list $L_2$ 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 $L_2$ then
18:                     $L_2$.delete($e_{ink}$)
19:                     $node_i.L_1$.delete($e_{ink}$)
20:                     if $node_i.L_1$ 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:                     $L_1$.add($e_{np(k_n++)}$)
31:                     $node_p.L_2$.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 $L_1$ is not empty then
46:                     MATCH($L_1$.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
```

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.

```
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 multicast 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 multicast 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 multicast 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. 3. Complete CONCEPTUAL Program for NPB FT (Class C) of 256 MPI Tasks

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 **??**

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.

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

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

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.

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.

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.

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

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.

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

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*.

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

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*.

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.