ABSTRACT

DESHPANDE, VIVEK RAJENDRA. Automatic Generation of Complete Communication Skeletons from Traces. (Under the direction of Dr. Frank Mueller.)

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

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

Automatic Generation of Complete Communication Skeletons from Traces

by
Vivek Rajendra Deshpande

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

Dr. Xiaosong Ma

Dr. Xiaohui (Helen) Gu

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

Vivek was born on December 14, 1986 in Pune (Maharashtra, India), a Cultural Capital of State Maharashtra and a Knowledge Hub a.k.a. Oxford of East. He received his Bachelor of Engineering degree in 2008 from University of Pune. After graduating with a degree in Computer Engineering, he worked for TCS Ltd. for almost one year. Being interested in more education and research, he pursued his Master's degree in Computer Science at NC State from Fall 2009. Since then, he has been working with Dr. Frank Mueller in systems research.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Frank Mueller for his support, guidance and faith in me. I would like to thank Dr. Xiaosong Ma and Dr. Xiaohui (Helen) Gu for serving on my advisory committee. This research used resources at Oak Ridge National Laboratory (ORNL) and I would like to thank ORNL for providing the access to their resources. I would like to thank Xing Wu for his useful inputs and help. I would like to thank Kishor, Abhik, Shobhit, David and my other labmates for their help and inputs. I am grateful towards Mrinal, my roommates and friends in Raleigh for their encouragement and help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

The “Supercomputer” term was coined because of the high processing capacity in terms of speed of calculations. In 1960s, commercial supercomputers were introduced. Those were primarily designed by Seymour Cray. He is called as the father of supercomputing and created a supercomputing industry.

Early supercomputers in 1960s were scalar machines 10 fold faster than contemporary computers. Vector processors dominated in 1970s in supercomputers. The early and mid-1980s saw machines with a modest number of vector processors working in parallel. Later in the 1980s and 1990s, the focus shifted to thousands of ordinary, off-the-shelf processors connected in parallel to form massive parallel processing systems. Today, parallel systems are based on server class microprocessors such as Opteron, Xeon and coprocessors such as GPUs (Nvidia Tesla), IBM Cells and FPGAs. Most modern supercomputers are highly tuned computer clusters that use commodity processors with custom interconnects [1].

Supercomputers are used for highly calculation-intensive tasks such as problems involving quantum physics, weather forecasting, climate research, molecular modeling (computing the structures and properties of chemical compounds, biological macromolecules, polymers, and crystals), physical simulations (such as simulation of airplanes in wind tunnels, simulation of the detonation of nuclear weapons, and research into nuclear fusion). Such experiments are carried out at places like Oak Ridge National Laboratory, in Oak Ridge, TN, on facilities such as their so called Jaguar and Kraken installations. According to www.top500.org, these two machines are among the fastest 10 machines in the world as of June 2011. Such petascale machines help in speeding up the process of computation by orders of magnitude. Thus, such machines are essential to scientific research simulations.

These supercomputers at National Labs are expensive and typically cannot be used for

Computer Science (CS) experimentations. The applications that run on such machines are robust, well tested and execute under conservative permissions in a well controlled environment. But there is need for systems that can be used for CS experimentations and innovations leading to highly tuned applications in terms of performance and efficiency. This approach is taken at NC State with the installation of A Root Cluster (ARC) [2]. ARC serves as a "crash-test dummy" for potential new CS solutions to address the major obstacles faced by next generation HPC systems.

Different programming models are available for parallel programming today, such as using shared memory and message passing. For shared memory, the address space is shared /w multiple threads running in parallel on different processors. For message passing multiple processes run simultaneously on different processors and communicate using message passing for data transfer or synchronization. In practice, these processes execute a single program on different sets of data that is multiple data a.k.a. the SPMD model. The message passing is used in high performance computing (HPC) because it can scale up to a large number of processors connected by the high-speed of interconnect, which can be called as distributed memory systems. For message passing, the Message Passing Interface (MPI) [21] is the de facto industry standard and specifies the API that allows the processes to communicate.

1.2 Motivation

To assess the capability of HPC systems for parallel applications, software subsystems and hardware are evaluated and analyzed for performance. This is done to the best by using actual applications. But porting an HPC application to target machine is tedious and time consuming, as such applications need compatible compilers and libraries for numerical operations and domain specific tasks. Application tuning for better performance, such as data decomposition or transformations for parallelism, are also difficult. Thus, using HPC application directly as a benchmark is impractical because of large overheads. Furthermore, some HPC applications are legacy codes that cannot be easily ported or may be considered intellectual property or classified sometimes so that their sources are not public.

Benchmarks are important for parallel I/O and HPC storage, thus are widely used for evaluating and analyzing storage systems and assessing migration costs of HPC applications to new platforms with different I/O subsystems. They are easy to port, modify and run. They provide an indication for characteristics as of HPC applications. Most existing benchmarks do not capture the complexity and scale of realistic HEC applications as they do not feature the intricate interplay of I/O operations, computation and communication.

To address those challenges, we propose a more viable solution to the requirement of evaluation of HPC systems, incl. storage, network and CPU speed without actually migrating HPC applications to those platforms. In our approach, we only need to observe an application run on given HPC platform and capture its dependencies to that platform.

1.3 Our Approach

HPC applications performance is largely dominated by large amount of numerical operations. These operations are captured microbenchmarks in the order they appear. We propose to generate communication benchmarks in an automated approach. These benchmarks are human readable, compact, easy to generate and port. They closely resemble behavior in terms of execution time and communication volume of the original application.

As an input, we take an HPC application with message passing communication using MPI (Message Passing Interface). The applications communication patterns are captured in traces using ScalaTrace [13]. The obtained trace is given as an input to the benchmark generator, which is the central focus of this work. It outputs the communication benchmark in C incl. MPI for communication and can be executed on target machine. This is illustrated in the following Figure 1.1.

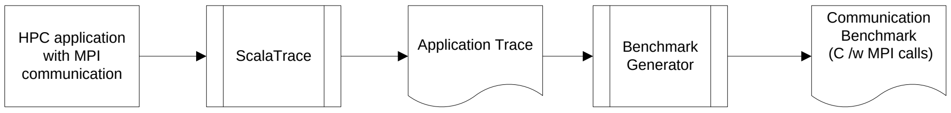

HPC application with MPI communication
ScalaTrace
Application Trace
Benchmark Generator
Communication Benchmark (C /w MPI calls)

Figure 1.1: Benchmark Generation System - Block Diagram

We utilize ScalaTrace [13] for communication trace collection. ScalaTrace is a unique approach to parallel application tracing as this scalable framework captures the communication in lossless and near constant size in terms of trace representation independent of the number of the nodes while keeping the structural information of the nodes and iterations. It also employs a pattern based intra-node and inter-node compression techniques extracting the applications communication structure.

For example, ScalaTrace can represent all processes performing the same operation (e.g., each MPI rank sending a message to rank+2) as a single event, regardless of the number of ranks. Because the application trace is the basis for benchmark generation, this feature helps to reduce the size of the generated code, making it more manageable for subsequent manual modifications. In contrast, previous application tracing tools, such as Extrae/Paraver [16], Tau [20], Open|SpeedShop [18], Vampir [12], and Kojak [25], are less suitable for benchmark generation because their traces increase in size with both the number of communication events and the number of MPI ranks traced. Second, ScalaTrace is aware of the structure of the original program. It utilizes the stack signature to distinguish different call sites. Its loop compression techniques can detect the loop structure of the source code. For example, if an iteration comprises a hundred iterations and each iteration sends five messages of one size and ten of another, ScalaTrace represents that internally as a set of nested loops rather than as 1500 individual messaging events. These pattern-identification features help benchmark generation maintain the program structure of the original application so that the generated code will be not only be semantically correct but also human comprehensible and editable.

1.4 Hypothesis

We contend that it is feasible to automatically generate communication benchmarks in C with MPI calls from a full-scale HPC application using the above mentioned scalable tracing methods. The generated code preserves the timing behavior and structural behavior of the original application. We also argue that the generated code is human readable, portable can be used as a communication benchmark.

1.5 Evaluation

We evaluate our generated communication benchmark using the NAS Parallel Benchmark Suite [4] and Sweep3D [24]. We perform evaluations by experimenting and testing the correctness and timing accuracy of the generated benchmarks. The obtained results show that auto-generated benchmarks preserve the applications semantics in terms of their communication pattern along with communication volume and the ordering of events relative to the original HPC application. Furthermore, the overall execution time of benchmarks is close to that of their original applications. Thus, communication benchmark generator is able to generate benchmarks that are similar to the original application in terms of communication behavior and execution time.

1.6 Contribution

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

Our work benefits application developers, communication researchers, and HPC system designers. Application developers can benefit in multiple ways. First, they can quickly gauge the application performance of a target machine before doing 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 the need to build complex applications and without access to source code that is not freely distributed or even classified. Finally, procurement of HPC systems can benefit by contracting vendors to deliver a specified performance on a given auto-generated benchmark without having to provide those vendors with the actual application.

1.7 Summary

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

Chapter 2

Background

The work on communication benchmark generation builds on previous work on MPI tracing. This scalable trace-compression framework is referred to as ScalaTrace. Previous work on ScalaTrace showed online trace compression can result in trace file sizes orders of magnitude smaller than previous approaches or, in some cases, even near constant size regardless of the number of nodes or application run time.

ScalaTrace collects communication traces using the MPI Profiling layer (PMPI) [3] through Umpire [23] to intercept MPI calls during application execution. On each node, profiling wrappers trace all MPI functions, recording their call parameters, such as source and destination of communications, but without recording the actual message content. ScalaTrace features aggressive trace compression that generates a single, concise and lossless trace file from any large-scale parallel application run. The resulting tracs preserve timing information, event ordering in the compressed form along with the calling context of traced events. The way ScalaTrace works and captures the traces is shown in the Figure 2.1.

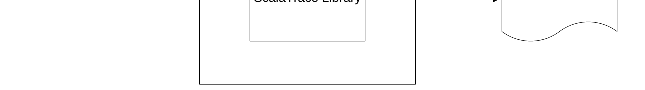


Figure 2.1: Working of ScalaTrace - Block Diagram

2.1 Trace Compression

ScalaTrace performs two types of compression: $intra - node$ and $inter - node$. For the intra node compression, the repetitive nature of timestep simulation in parallel scientific applications is used. Intra-node compression is performed on-the-fly within a node. Further, the inter-node merge, exploits the homogeneity in behavior (SPMD) among different processes running the application. Inter-node compression is performed across nodes by forming a radix tree structure among all nodes and sending all intra-node compressed traces to respective parents in the radix tree. This results in a single compressed trace file capturing the entire application execution across all nodes. The compression algorithm is discussed in detail in other papers [14, 17].

ScalaTrace achieves near constant size traces by applying pattern based compression. It uses extended regular section descriptors (RSD) to record the participating nodes and parameter values of multiple calls to a single MPI routine in the source code across loop iterations and nodes in compressed manner [8]. Power-RSDs (PRSD) recursively specify RSDs nested in a loop [11].

For example, consider the code snippet shown below, which has ring-style communication across N nodes.

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

We can see that three RSDs present in the code snippet are:
RSD1: $\{\langle rank \rangle$, MPI_Irecv, Left$\}$
RSD2: $\{\langle rank \rangle$, MPI_Isend, Right$\}$
RSD3: $\{\langle rank \rangle$, MPI_Waitall$\}$
These RSDs denote the non-blocking send and receive and waitall MPI operations in a single loop iteration, where $\langle rank \rangle$ represents a value within $0 \ldots N - 1$ in each per-node trace. ScalaTrace then detects the loop structure and ouputs the single PRSD

PRSD: {100, RSD1, RSD2, RSD3}

to denote a single loop of 100 iterations. This intra-node compression is performed on-the-fly to reduce the time for trace generation and the memory overhead.

Further, during the inter-node compression, the local traces on each node are combined into a single global trace when the application is terminated (i.e., within the PMPI interposition wrapper for MPI_Finalize). Inter-node compression detects similarities among the per-nodes traces and merges the RSDs by combining their participant lists in a final participant list. For example, in the given code snippet, because each MPI routine is called on each node with same parameters, these RSDs across the nodes are merged within the PRSD 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}

The participant node information is encoded and represented in a tuple containing starting rank, total number of participants and an offset value separating ranks. Even multi-dimensional information is captured in this encoding format. This is explained more in later Section 3.8. There are special cases in which events with matching calling context can have non-matching function parameters. These non-matching function parameters are compressed using a vector representation so that the particular event can be concisely represented in the trace.

2.2 Time Preservation

Another important feature of ScalaTrace is the time preservation of captured traces. Instead of recording absolute timestamps, the tool records delta time of computation durations between adjacent communication calls. During RSD formation, instead of accumulating exact delta timestamps, statistical histogram bins are utilized to concisely represent timing details across the loop. These bins are comprised of statistical timing data (minimum, maximum, average and standard deviation). ScalaTrace compresses into a histogram of times taken by all instances of a particular computation (identified by its unique call path). For example, consider the code snippet shown below:

```
for(i=0; i<10; i++){
   if (i==0)
      (Long running computation)
   else
      (Short running computation)
   MPI_Recv(...)
   .
   .
   .
}
```

The time spent in computation in first iteration prior to first MPI instruction differs in the later iterations. More details on collecting statistical timing information are provided elsewhere [17].

Chapter 3

Benchmark Generation

3.1 Overview

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

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

Our goal is to generate benchmark code that is compact, portable, human-readable, and accurately reflects the original application's communication characteristics and performance. It is human readable, so a human can easily examine, understand the communication in the application and perform modifications. It is portable and compact to facilitate migration of applications from one platform to another. Based on the performance on one platform, one can assess performance on another platform at a high level with regard to communication and interconnect capabilities without actually porting the application code. The generated code preserves the communication features and semantics of the original application in terms of structure, making it available to new platforms as well. This assists in performance analysis of software, hardware and also in easy migration of applications. There are different challenges involved in this implementation of code generation because of readability and reproducibility goals along with handling target language intricacies. Thus, these problems are a subject of research and discussed in this section.

3.2 File Composition

The generator takes a single trace file as input and expands it to C files with MPI calls for the communication benchmark. The files generated are skeleton_code.c, skeleton_type.c and skeleton_code.h. The skeleton_code.c is the main file with all the MPI events representing the communication pattern of the actual application. skeleton_type.c is the supporting file and contains supporting functions. skeleton_code.h is the header file and has all the variables declared to assist RSD/PRSD traversal. While generating the code, some variables are dynamically created, e.g. index variables, as the PRSD representing the loop is traversed or new communicators are created by communicator split or duplication events. These variables are added to the header file.

Along with this, two text files, skel_rank and skel_alltoallv, are generated. skel_alltoallv file is an optional file and generated only if MPI_Alltoallv is one of the events in the original application and not converted to MPI_Alltoall by averaging of count arrays. Thus, the skel_alltoallv file contains all the information required by the MPI_Alltoallv event such as send count, receive count, displacements in receive and send buffers. The skel_rank file contains all the ranklists and MPI event numbers of participating tasks. Ranklist is an encoded list of the MPI tasks taking part in a particular MPI event. Handling of this information is explained in the later sections. Figure 3.1 depicts the composition of files in the generated code.

3.3 The Framework and Process

The framework is designed in such a way that it can be used to both replay the trace and generate the code. While generating the code, the trace is traversed only once, including the part of the trace which belongs to PRSDs (iterations). While traversing, the RSDs and PRSDs are parsed. The detailed process of parsing each described in the following sections.

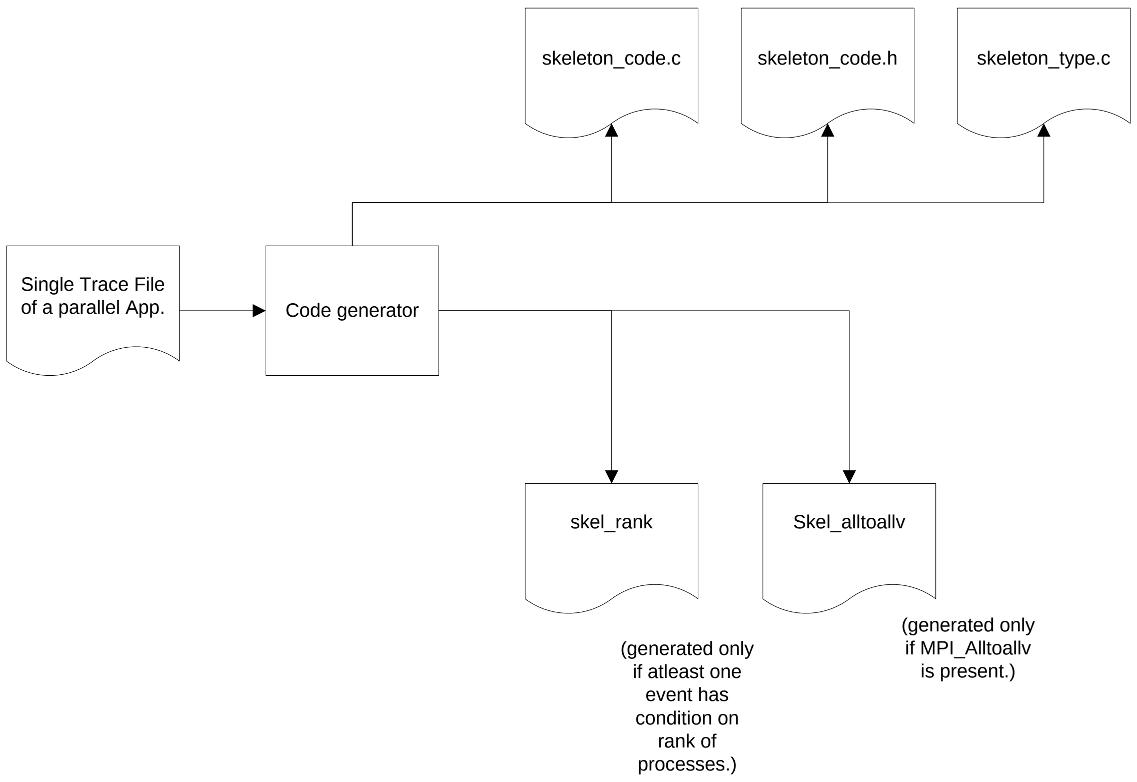

skeleton_code.c
skeleton_code.h
skeleton_type.c
Single Trace File of a parallel App.
Code generator
skel_rank
Skel_alltoallv
(generated only if atleast one event has condition on rank of processes.)
(generated only if MPI_Alltoallv is present.)

Figure 3.1: The Composition of Files in a Generated Code

3.3.1 Parsing RSD

By parsing one RSD, we generate one MPI event such as synchronous MPI_Send, MPI_Recv or asynchronous MPI_Irecv, MPI_Isend, collectives such as MPI_Alltoall, MPI_Alltoallv, MPI_-Reduce, MPI_Barrier and communicator based events such as MPI_Split and MPI_Dup.

In each RSD, the parameters of each event are captured, such as relative source or relative destination, datatype, amount of data (count), tag, communicator and so on. The MPI events handled in this tool are enlisted in Table 3.1 below along with the parameters captured in the trace.The information about the parameters captured in the trace is given Table 3.2.

Using these parameters for the RSDs, the MPI events are generated. Along with this information, the delta times for the computation between the communication events is captured in the RSDs. Each RSD has the delta time encoded, which represents the time between the current MPI event and previous MPI event. Using this delta time, sleep calls are generated before each MPI event.

Whenever an MPI event from a RSD is generated, the count and datatype of the data being sent is used to find the maximum size of the buffer which can be used in the generated code as the size of send and receive buffers.

Table 3.1: MPI events

MPI event	Type	Parameters Captured in Trace
MPI_Send	Synchronous point to point	count, datatype, dest, tag, comm
MPI_Recv	Synchronous point to point	count, datatype, source, tag, comm
MPI_Isend	Asynchronous point to point	count, datatype, dest, tag, comm
MPI_Irecv	Asynchronous point to point	count, datatype, source, tag, comm
MPI_Barrier	Collective	count, comm
MPI_Bcast	Collective	count, datatype, root, comm
MPI_Reduce	Collective	count, datatype, op, root, comm
MPI_Allreduce	Collective	count, datatype, op, comm
MPI_Alltoall	Collective	sendcount, sendtype, recvcount, recvtype, comm
MPI_Alltoallv	Collective	sendcount, sdispls, sendtype, recvcount, rdispls recvtype, comm
MPI_Barrier	Collective	count, comm
MPI_Comm_split	Communicator based	comm, color, key
MPI_Comm_dup	Communicator based	comm

Table 3.2: MPI event parameters

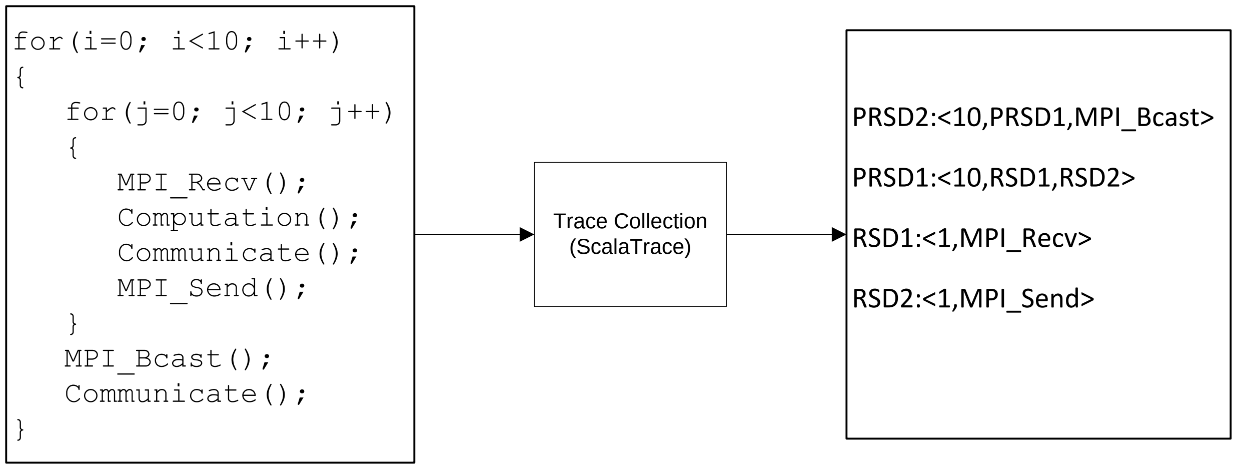
MPI Parameter	Information
count	number of elements in send/receive buffer (nonnegative integer)
datatype	datatype of each send/receive buffer element (handle)
dest	rank of destination (integer)
source	rank of source (integer)
tag	message tag (integer)
comm	communicator (handle)
root	rank of broadcast root (integer)
op	operation (handle)
sendcount/recvcount	integer array equal to the group size specifying the number of elements to send/receive to each processor
sendtype/recvtype	type of send/receive buffer elements (handle)
sdispls/rdispls	integer array (of length group size). Entry j specifies the displacement relative to sendbuf/recvbuf from which to take the outgoing data destined for process j
color	control of subset assignment (nonnegative integer). Processes with the same color are in the same new communicator
key	control of rank assigment (integer)

The details of handling parameters such as communicator, requests and corresponding waits are discussed in later sections. Also because of inter node compression, the parameters such as count, source and destination are merged forming complex data structures. Handling of those data structures is also discussed in later sections.

3.3.2 Parsing PRSD

By parsing a single PRSD, we generate C-style "for loops". In the trace, a PRSD can be mapped to a loop in the original source. Thus, nested loops and sequences of loops can be represented using Power RSDs.

For example, consider the RSD tuples as RSD1:⟨*1,MPI_Recv*⟩, RSD2:⟨*1,MPI_Send*⟩, which capture MPI_Recv and MPI_Send calls in those RSDs respectively. The PRSD as PRSD1:⟨*10,RSD1,RSD2*⟩ denotes a loop of 10 iterations each with RSD1 and RSD2. Furthermore, PRSD2:⟨*10,PRSD1,MPI_Bcast*⟩ denotes a loop of PRSD1 followed by broadcast. (see Figure 3.2).



```
for(i=0; i<10; i++)
{
   for(j=0; j<10; j++)
   {
      MPI_Recv();
      Computation();
      Communicate();
      MPI_Send();
   }
   MPI_Bcast();
   Communicate();
}
```

```
PRSD2:<10,PRSD1,MPI_Bcast>
PRSD1:<10,RSD1,RSD2>
RSD1:<1,MPI_Recv>
RSD2:<1,MPI_Send>
```

Figure 3.2: The Trace Collection

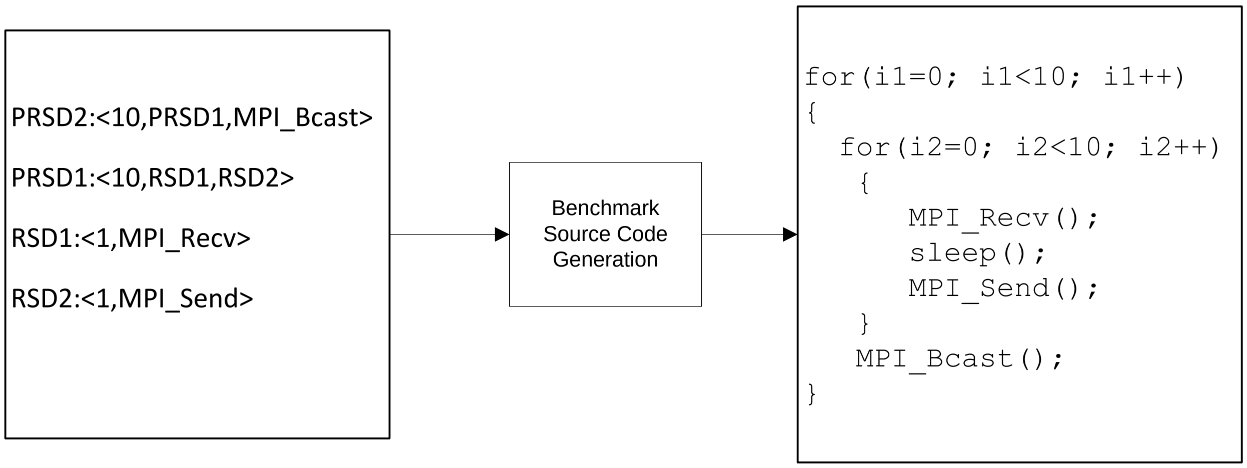
This repetitive nature of PRSDs to capture the nesting of loops is exploited in the benchmark code generation. The generated trace is then traversed using a recursive function. The depth of nesting is captured by the recursive function.

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

The algorithm for loop handling from MPI traces is given below:

```
function Trace_queue_and_generate(head,depth)
   current =head;
   if (current == PRSD)
         Generate a for loop of nesting *depth* with a new index created using *depth*.
   end if
   while(current != null)
      if(current == RSD)
         Generate the MPI events in the RSD.
      end if
      if(current == PRSD)
         Trace_queue_and_generate(head,depth+1)
      end if
   end while
end function
```

The benchmark code generated using trace from Figure 3.2 can be illustrated in Figure 3.3,



```
for(i1=0; i1<10; i1++)
{
  for(i2=0; i2<10; i2++)
   {
      MPI_Recv();
      sleep();
      MPI_Send();
   }
   MPI_Bcast();
}
```

Figure 3.3: The Code Generation

The index variable is generated using the depth of nesting of the loops. Thus, we can reuse variables as the number of index variables is limited by the maximum depth of nesting. This facilitates code generation for different control structures such as straight line code, multiple loops at different places inside a single loop and nesting of loops.

The sleeps are generated inside and at the beginning of the *for loop* and before the *for loop* using the delta times between the events of the *for loop* and the event before the loop.

3.4 Variables and Derived Datatypes

Variables are generated as a part of the generated code during the process of code generation. These variables are written to the skeleton_code.h. For this, while traversing through the trace, variables are added in a list and then emitted to this file at the end. These variables are non-MPI primitive datatypes such as int, char, etc. These dynamically created variables during the process of code generation are added to the list of variables in the structure as given below:

```
typedef struct var_dump {
  char* var_detail[VAR_LIST_SIZE];
  int is_global[VAR_LIST_SIZE];
  int total_count;
} var_dump_t;
```

The examples of the variables added to this structure are as follows: myrank is a an integer variable for each process containing the rank of the process in the communicator during MPI communication; comm_size is an integer variable has the size of communicator during MPI communication and index variables such as *int i1=0;* and so on.

The derived datatypes are based upon sequences of MPI primitive datatypes. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous. The datatypes created using the MPI calls given for derived datatypes are added to the list of derived datatypes in a data structure depicted below:

```
typedef struct derived_type_dump {
  char* type_detail[TYPE_LIST_SIZE];
  char* type_name[TYPE_LIST_SIZE];
  int type_usage_count[TYPE_LIST_SIZE];
  int is_derived_type[TYPE_LIST_SIZE];
  int type_trace_index[TYPE_LIST_SIZE];
  int total_count;
} derived_type_dump_t;
```

3.5 Communicators

The MPI_Comm_split and MPI_Comm_dup events create new communicators. These communicator names are declared in the skeleton_code.h similar to other variables with the datatype MPI_Comm. These communicators are dynamically created during the process of code generation. They are added in list and then emitted to this file at the end. During the process of code generation these dynamically created communicators are added to the list of communicators in the structure as given below:

```
typedef struct comm_dump {
  char* comm_name[COMM_LIST_SIZE];
  int comm_trace_index[COMM_LIST_SIZE];
  int split_tracker_flag[COMM_LIST_SIZE];
  int total_count;
}comm_dump_t;
```

The value of comm_trace_index from the trace is used to create a name for the new communicator by appending the postfix comm_trace_index to the prefix *comm_out*. For example, the name for the communicator with the comm_trace_index value 1000 would be *comm_out1000*. This would be added to skeleton_code.h file as *MPI_Comm comm_out1000;*

The color and key in the trace of split communicators are used to generate the MPI_Comm_-split. The color and key could be absolute or could be an offset. If an offset is encountered then the offset is added to *myrank*.

3.6 Requests and Waits

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

The code generated in the skeleton_type.c and skeleton_code.h is used to handle these requests during the execution of the benchmark.The following per-process structure is used to handle the dynamically created requests:

```
typedef struct req_handler{
           MPI_Request *req_buf;
           MPI_Status  *statuses;
           int req_bufsize;
           int req_bufcur;
      } * req_handler_t;
```

A Small example for referencing requests handles and waiting on those handles is shown in the code snippet below:

```
current = add_request(req);
MPI_Irecv(buffer2, recv_count, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
                    4, MPI_COMM_WORLD,&(req->req_buf[current]));
MPI_Send(buffer, send_count, MPI_DOUBLE_PRECISION,
                     (511+myrank)%comm_size, 4, MPI_COMM_WORLD);
MPI_Wait(&req->req_buf[0],&req->statuses[0]);
reset_req(req,9999,1);
```

In the code snippet above, during the execution of the benchmark, a request handle is added to the structure using the pointer *req* to structure req_handler and a position is returned in the variable *current*. This is used as a reference within MPI_Wait and MPI_Waitall. After the wait is executed, the request handle is reset to MPI_REQUEST_NULL using the handle obtained from the trace of corresponding to the wait. Thus the same request handle can be reused in future waits. Here in the example, *511* is a relative destination recorded in the trace of MPI_Send that is added to the *myrank*.

3.7 MPI_Alltoallv

When using MPI_Alltoallv, each process sends data to all other processes. Each process may send a different amount of data and provide displacements for the input and output data.

The parameters send count, receive count, send displacement, receive displacement, send and receive datatypes are recorded in the trace. These parameters are written to the file *skel_-alltoallv* for all the *MPI_Alltoallv* events in the trace. This is a text file and generated only if MPI_Alltoallv is encountered. The code generated for the MPI_Alltoallv event is emitted to the file skeleton_code.c. This consists code for selecting correct values of all the required parameters from the file skel_alltoallv and calling the MPI_Alltoallv function. The generated code snippet is shown below:

```
skel_alltoallv_fh=fopen("skel_alltoallv","r");
.
.
.
fscanf(skel_alltoallv_fh,"%d",&no_counts);
        rdispls = (int*)realloc(rdispls,no_counts*sizeof(int));
```

```
for(it=0;it<no_counts;it++)
        fscanf(skel_alltoallv_fh,"%d",&rdispls[it]);
fclose(skel_alltoallv_fh);
skel_sleep(0, 25000);
MPI_Alltoallv(buffer, counts, displs, MPI_INT,buffer2, recvcounts,
         rdispls, MPI_INT, MPI_COMM_WORLD);
```

In the above code snippet, the way parameters are read in to the variables is shown as an example for one parameter that is a receive displacement in the variable *rdispls*.

Since reading parameters from file is inefficient in terms of time and increases the contention, we implemented another approach to generate MPI_Alltoallv. In this approach, we average the count values in the counts array and generate MPI_Alltoallv with that same send, receive count values as the parameter. The basic idea behind this is the total amount of data transferred over the interconnect remains constant resulting in an equivalent overhead and execution time as the original. This works well for IS of NAS Parallel Benchmarks or other codes that use algorithmic load balancing intrinsic to the application.

3.8 Multivalue Parameters and Ranklists

While obtaining the trace using ScalaTrace, the compression is performed also on non-matching parameters. At the *intra-node merge level*, parameter queues are created for source (for receives), destination (for sends), request (for waits), counts and datatypes. Thus, intra-node merge constructs an array of parameters in case of a parameter mismatch while most (other) events may match. This can be represented as a vector:
$parameter\ value_1\ value_2\ value_3\ \ldots\ value_n$

This array of values could be a basic array such as (count or displacement arrays in MPI_-Alltoallv) or it could result from an intra-node merge discussed above and can be called as intra-node arrays. In case of intra-node arrays, the value in the array corresponds to the iteration step of the closest loop in which the RSD is being parsed.

For example, consider the code snippet given below:

```
for(i=1;i<=3;i++){
   MPI_Send(..., i,...);   //count = i
   MPI_Recv(..., i,...);
}
```

In the above code snippet, the send and receive in each iteration has different value in count parameter. While intra-node merge, the sends and receives have different count value generating a intra-node array and is recorded in the trace as: *count 1 2 3*. Each value in the count parameter corresponds to the iteration step of the closest loop.

During the *inter-node merge*, the previous method is taken one step further. Even when the parameters do not match across all nodes, they tend to be the same for groups of nodes. This property is exploited during inter-node merge as ⟨*value, ranklist*⟩ tuples are created. The parameter *value* can be an array or a scalar.

The ranklists are created for an MPI event, when only a subset of ranks or nodes is taking part in an event. Then a participatant list is created for that event and is part of the RSD for that event. Also, ranklists are constructed as a parameter as tuples ⟨*value, ranklist*⟩ because of the inter-node merge as discussed above.

The generated skel_rank file holds all ranklists where each ranklist has a unique identification number. During the execution, this identification number is used to select a particular ranklist for parsing and to identify whether the current rank is part of that ranklist.

The ranklist is encoded as sequence of integers using a recursive pattern during the process of compression. The first element specifies the number of recursive lists. The subsequent numbers specify those recursive lists. Within a list, the first number specifies the start rank. The start rank is the lowest processs rank in that list. The next number is depth indicating the number of iterations required to fully unwind the ranklist. At each depth, a stride and number of iterations are specified. The illustration of ranklist is given in the Figure 3.4 below:

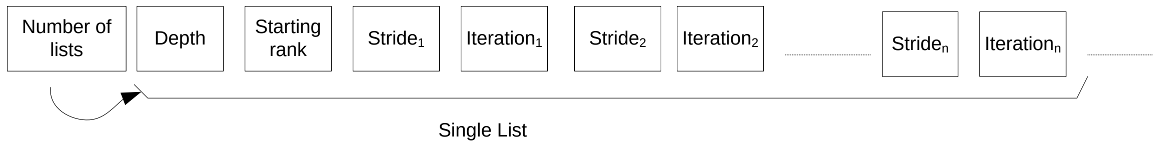


Figure 3.4: The Illustration of a Ranklist

We next discuss the process of the parsing the ranklist and the generation of the corresponding code. During code generation, the function is_member(identification number, current rank) is constructed in skeleton_type.c. This function parses the ranklist for each event/for tuples ⟨*value, ranklist*⟩ during execution of the benchmark. For the ranklist of an event the parsing decides whether the current rank is part of the process or not. For ranklist tuples ⟨*value, ranklist*⟩ the parser iterates through the tuple and extracts correct value for the current rank.

The algorithm for the is_member() function is as given below. Current rank is the rank of the process calling this function with the identification number for the ranklist from the skel_rank file. This function returns true if the current rank is part of the participant list or rank list.

```
function is_member(identification number, current rank)
for k = 1 to number of ranklists do
      if (current rank == start rank)
         return true
      end if
      if (current rank  < start rank)
         return false
      end if
      if(depth > 1)                  //depth indicates the number of iterations
          for j = 1 to depth do
                iters = $iteration_j$
                 stride = $stride_j$
                 for i=start to(start+((iters)*stride)) do
                       if (rank < i)
                            i = i - stride
                            break
                       end if
                        i = i+ stride
                 end for
                 start = i
            end for
       end if
       iters = $iteration_n$
       stride = $stride_n$
      if(stride > 1 && iters > 1) do
             return_val = (((rank-start)%stride==0) &&
                                      (rank <= (start+((iters-1)*stride)))) ? 1 : 0;
      else if(stride == 1 && iters > 1)
             return_val = (rank >= start && rank <=(start+((iters-1)*stride))) ? 1 : 0;
      else if(stride == 0 && iters == 1)
             return_val = (rank == start) ? 1 : 0;
      end if
      if( return_val == 1)
             return 1
     end if
end for
return return_val
end function
```

Thus this algorithm can be explained using a recursive function in the Figure 3.5 as follows:

```
function is_member(depth, rank, start)
read stride and iterations.
if( depth > 1)
   i = start
   start = find_start(rank, i, iterations,stride)
   f(depth -1, rank, start)
end if
if(depth == 1)
   if(stride  > 1 && iterations > 1 && (rank-start)%stride
                          && rank<=start+(iterations-1)*stride)
      return 1
   else
     return 0
   if(stride == 1 && iterations > 1 && rank>=start
                          && rank<=start+(iterations-1)*stride)
      return 1
   else
     return 0
end if
end function is_member

//This returns start rank for the given depth.
fucntion find_start(rank, i, iterations,stride)
if( rank <  i)
   return (i-stride)
if(iterations = 0)
   return  i
return find_start(rank,i+stride,iterations -1, stride)
end function find_start
```

Figure 3.5: Recursive Function to Check Participation in Ranklist

Here, at each depth, the highest *start rank* is found using function find_start, which is less than the rank for which we need check its participation in the event. Then using this start rank and the stride and iterations at that depth, the function returns 1 if the rank is participating in the event, else 0.

Consider a ranklist as 1 -2 0 4 2 1 3.
One ranklist indicated by first “1”. Then the depth of 2 indicated by “-2” and the start rank is indicated by “0”. The unwinding of the ranklist for 8 nodes is shown in Table 3.3.

Table 3.3: Unwinding of Ranklist

Node number	Depth = 2			Depth = 1			Return value
	start	stride	iterations	start	stride	iterations	
0, 1, 2	0	4	2	0	1	3	1
3	0	4	2	0	1	3	0
4, 5, 6	0	4	2	4	1	3	1
7	0	4	2	4	1	3	0

3.9 Preservation of Behavior

Some applications use the MPI feature of MPI_ANY_SOURCE known as *wild card receives* (e.g. in the LU benchmark of the NAS Parallel Benchmark Suite). Such features introduce non determinism in the application. Such non determinism might be resolved by using TAGS in sends and receives. But the MPI_ANY_SOURCE is recorded as an abstract(special value) in the trace. This special value is used to generate the code with MPI_ANY_SOURCE. Thus, it accurately preserves the behavior of the application. If the original application is non deterministic in nature, then it is also captured in the benchmark. Thereby, reflecting the actual communication in the benchmark as that of the original application.

If the original application shows diverging timing behavior over the course of execution because of non- determinism, then the generated benchmark will conserve the overall *proportional behavior*. For example, consider the code snippet shown below:

```
MPI_Recv(..., MPI_ANY_SOURCE, ...,status)
if (status. MPI_SOURCE == 0)
     (Long running computation)
else
     (Short running computation)
```

Depending on the senders rank, the code shown above can take a long time or a short time to run. Thereby, changing the time of execution of code from run to run. But after many runs, the proportion between two patterns can be obtained. Since the generated code from this application will have same structure and semantics, it will conserve the proportion of the two different execution times including the non-determinism of the original application.

Chapter 4

Experimental Framework

To evaluate our communication benchmark-generation tool, we generated C code with MPI calls for the NAS Parallel Benchmarks (NPB) suite (version 3.3 for MPI, comprising BT, CG, EP, FT, IS, LU, MG, and SP) using the class C and D input sizes [4] and for the Sweep3D neutron-transport kernel [24]. These benchmarks all have either a mesh-neighbor communication pattern or rely heavily on collective communication. Some of them (e.g., SP and BT) require communicator handling (Section 3.5), others (e.g. IS) require averaging of parameters in MPI_Alltoallv (Section 3.7) and some (e.g., LU) require the recording of wildcard receives (Section 3.9). Hence, the key features of our code-generation framework are fully tested in this set of experiments.

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

Chapter 5

Evaluation

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

5.1 Correctness

Our first set of experiments verifies the correctness of the generated benchmarks, i.e., the benchmark generator's ability to retain the original applications' communication pattern. For these experiments, we acquired traces of our test suite on ARC, generated communication benchmarks, and executed these benchmarks also on ARC. To verify the correctness of the generated benchmarks, we linked both the generated codes and the original applications with mpiP [22], a lightweight MPI profiling library that gathers run-time statistics of MPI event counts and the message volumes exchanged. Experimental results (not presented here) showed that, for each type of MPI event, the event count and the message volume measured for each generated benchmark matched perfectly with those measured for the original application.

We then conducted experiments to verify that the generated benchmarks not only resemble the original applications in overall statistics but also that they preserve the original semantics on a per-event basis. To this end, we instrumented each generated benchmark with ScalaTrace and compared its communication trace with that of its respective original application. Due to differences in the call-site stack signatures between the original application and the generated benchmark, these traces cannot be identical, they can only be semantically equivalent. Therefore, we replayed both traces with the ScalaTrace-based ScalaReplay tool [26] to eliminate spurious structural differences and thus allow a fair comparison 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. The experimental framework is depicted in the Figure 5.1.

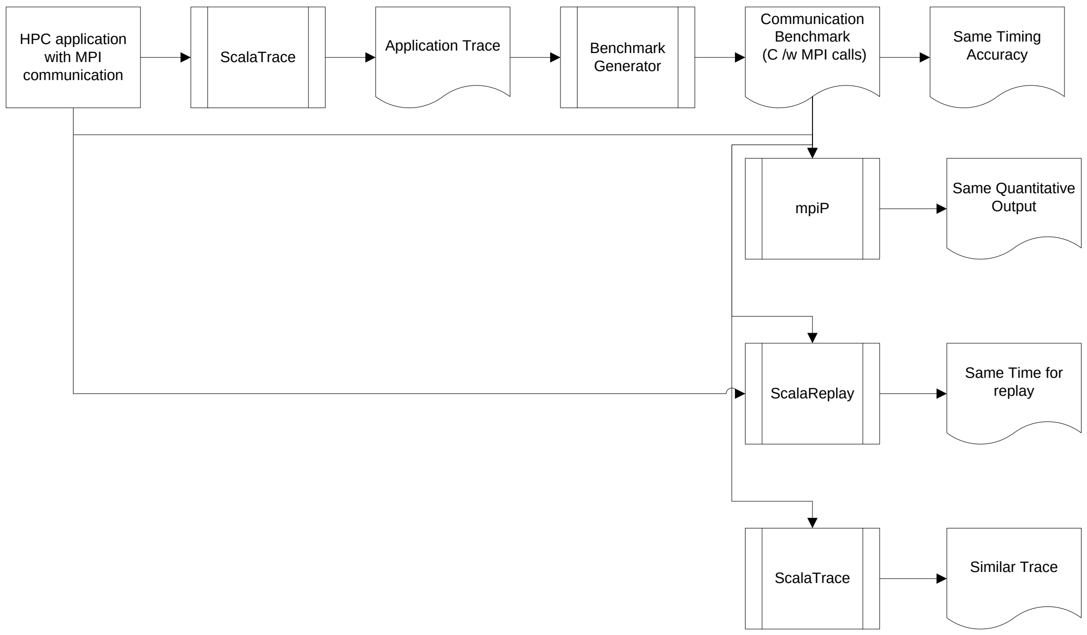


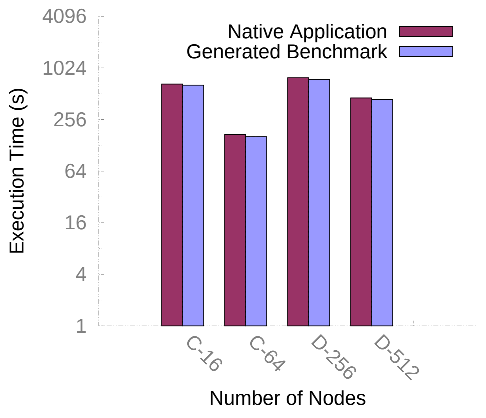
Figure 5.1: Experimental Framework

5.2 Accuracy of Timing Results

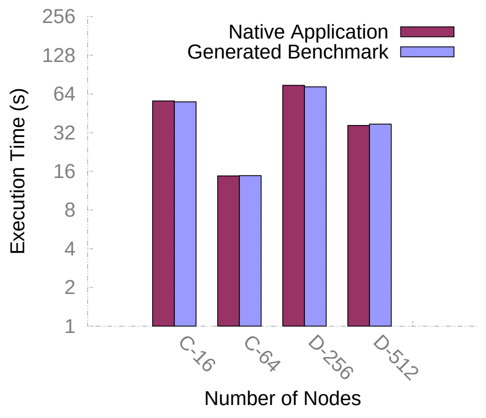
After evaluating that the generated code preserves the communication of the original application in terms of ordering of events and message volumes, we assessed ability of generated benchmark to retain the performance in terms of wall-clock time relative the orginal application. To measure the execucution times of the original applications, we extended the PMPI profiling wrappers of MPI_Init and MPI_Finalize to obtain the start and end timestamps, respectively. The corresponding timing calls were also added to the generated benchmarks. We executed both the original application and the generated benchmark on the ARC system, measured and compared the elapsed times. The results obtained are shown in the Figures 5.2 and 5.3.

We can observe from the graphs that the timings obtained for the generated benchmarks are very close to that of the original applications indicating very high accuracy. Quantitatively, the mean percentage error obtained by formula $|T_{gen} - T_{app}|/T_{app} * 100$ across all the graphs is only 6.7% and only one deviation with less timing accuracy observed is: class D FT for 512 nodes (110 seconds for benchmark and 145 seconds for original application) with a difference 24%. The average delta time is used to resemble computation via busy wait. FT uses collectives heavily, which may result in computational imbalance. Hence, computation is more closely resembled by the maximum recorded time instead of the average for collectives.

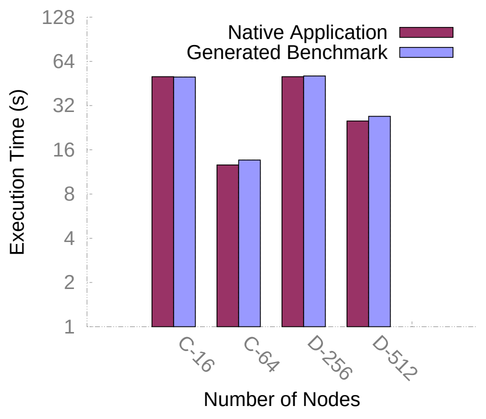
The results for the class C IS benchmark varying from 16 to 512 processors are shown in the Figure 5.4. We observe that the execution time reduces from 16 processors to 64 processors



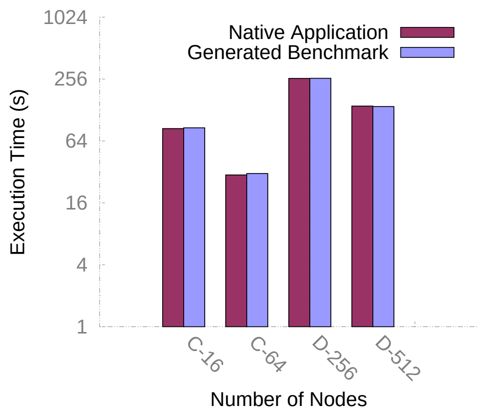
(a) LU



(b) MG



(c) EP



(d) CG

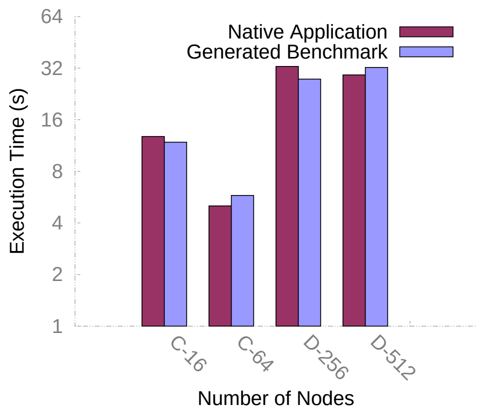
Figure 5.2: Graphs for Timing Accuracy of LU, MG, EP and CG.

and then increases as the number of processors increase from 256 and onwards. This can be explained through strong scaling in which the solution time varies with the number of processors for a fixed total problem size. The reason for the increase in execution time with the increase in the number of processors beyond certain number is likely due to increase in the communication and decrease in the per-processor computation.

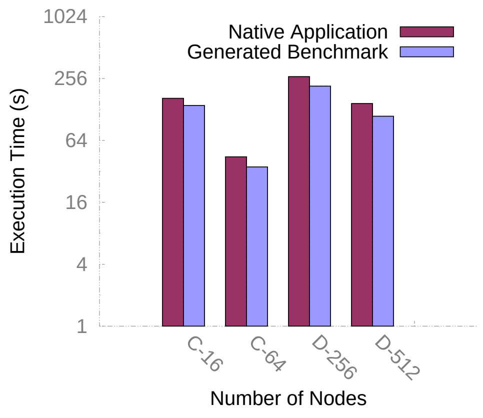
5.3 Cross Platform Results

We obtained cross platform results by porting the generated benchmark of IS and MG on ARC on Jaguar. The results are shown in the graphs from figures 5.5 and 5.6.

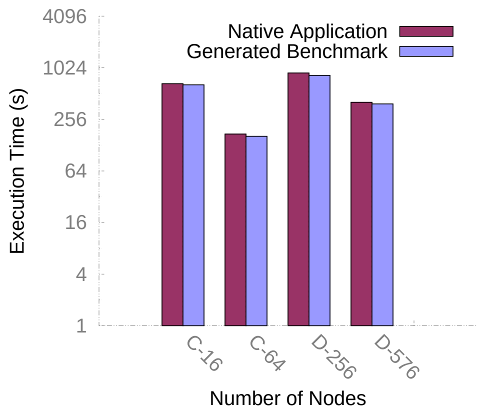
From Figure 5.5 we can observe that, in case of IS benchmark, the difference between the



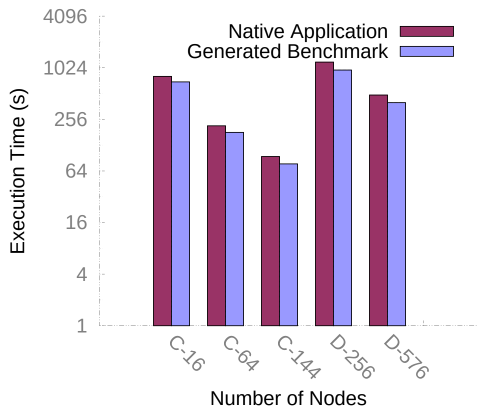
(a) IS



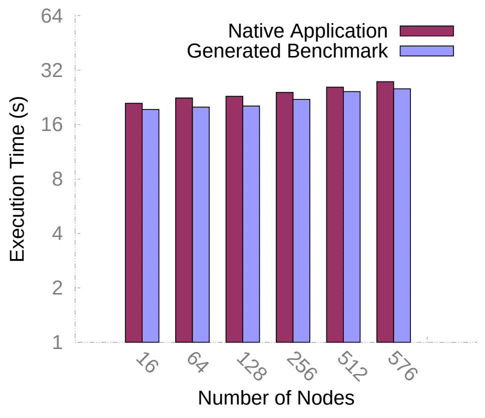
(b) FT



(c) BT



(d) SP



(e) Sweep3D

Figure 5.3: Continued Graphs for Timing Accuracy of IS, FT, BT, CG and Sweep3D.

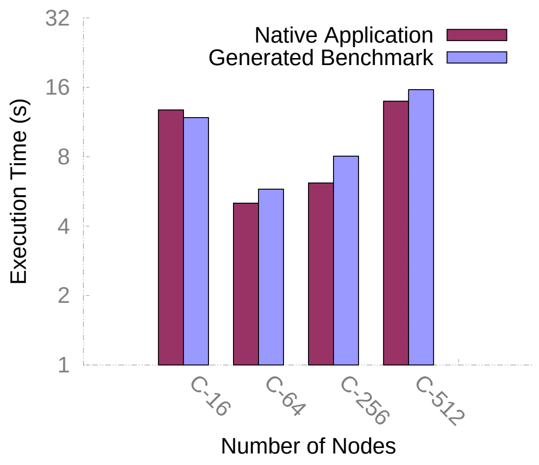


Figure 5.4: Timing Results for IS for class C on ARC

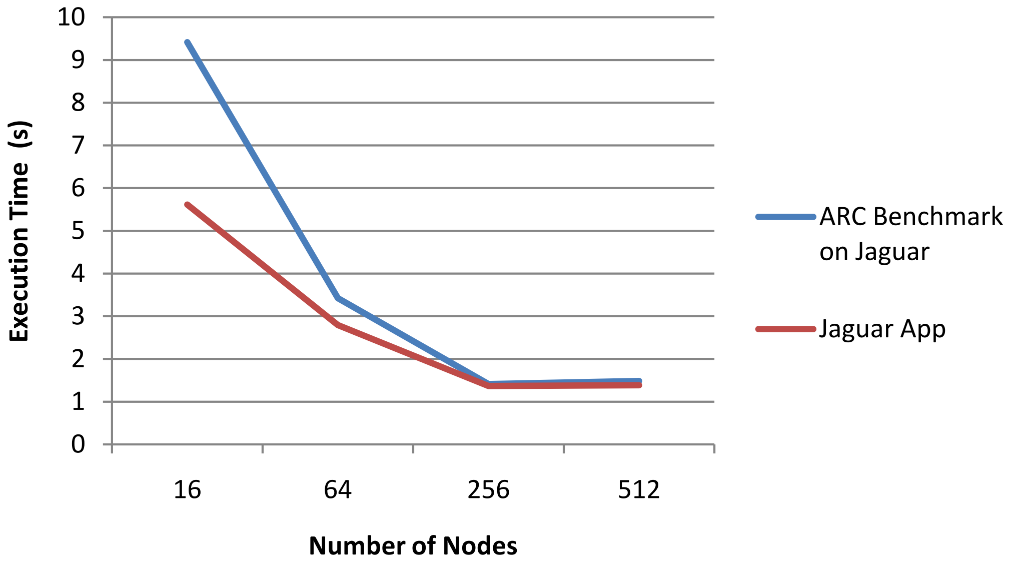


Figure 5.5: Timing Results of Cross Platform Experiments of IS

execution times of benchmark from ARC and the original application on Jaguar reduces as the number of processors increase. This is because the computation is splitt accross a larger number of processors reducing the per-processor computation to communication ratio and thus reducing the effect of higher processing capacity of Jaguar. Also, for the IS benchmark, the lowest time is obtained for 64 processors up to 512 processors on the ARC cluster resembling the actual application behavior. The same benchmark obtained on ARC indicated the lowest time for 256 processors on Jaguar and this resembles the execution time of the original application on Jaguar with the lowest time for 256 processors up to 512 processors.

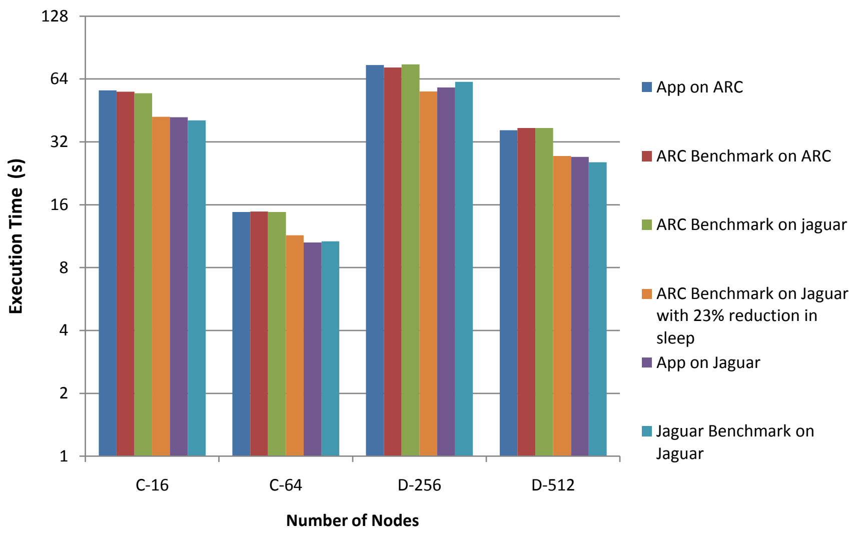


Figure 5.6: Timing Results of Cross Platform Experiments of MG

From Figure 5.6 we observe that the execution time of MG benchmark obtained on ARC takes similar time as that of the original application on Jaguar, whereas the execution time for the MG benchmark obtained on Jaguar itself very closely resembles it. The difference is observed because of the difference in the CPU speeds of the both ARC and Jaguar. Jaguar with a faster CPU than ARC, finishes the computation earlier than the sleeps indicated by the benchmark obtained on ARC. We reduced the sleeps in the MG benchmark obtained on ARC by 23% empirically and by observing the delta times in the traces from ARC and Jaguar, to obtain the fabricated speedup. With this, we observed that the execution time of the fabricated benchmark matches very closely to that of MG benchmark obtained from Jaguar and actual MG application, when executed on Jaguar.

To verify the speedup of Jaguar over ARC, we executed a computational kernel that performs matrix multiplication on a single processor for square matrices of size 100 with iterations ranging from 3000 to 9000. Execution times are given in the Table 5.1. The CPU speedup of Jaguar over ARC is very close to 23% which conforms our observations from traces.

Table 5.1: Execution Times for Matrix Multiplication on ARC and Jaguar

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

Thus, such performance experiments could be performed with the benchmarks generated by our tool and could help in gauging different performance aspects related to communication for HPC systems with increasing complexities without actually porting the real applications to those platforms.

5.4 Lines of Code

We also measured the number of lines of code(LOC) in the generated code to gauge the conciseness of the generated code.

Table 5.2: Comparision of Number Lines of Code

	App (LOC)	Benchmark (LOC)	Change(%)
LU	5937	2868	-51.69277413
MG	2580	11402	341.9379845
EP	325	306	-5.846153846
CG	1796	1658	-7.683741648
IS	1141	365	-68.01051709
FT	2165	320	-85.21939954
BT	9217	1547	-83.2157969
SP	4922	6657	35.24989842
Sweep3D	2096	14624	597.7099237

Table 5.2 shows that the number of lines in the generated code are lower than for the native application except for MG, SP and Sweep3D. This makes the generated code more precise and readable in just a single file. For MG, SP and Sweep3D, the number of lines increases due to different compression techniques that result in conditionals in the auto-generated code. We plan to address this issue by applying more aggressive compression techniques in ScalaTrace.

Chapter 6

Related Work

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

- The size of the benchmarks we generate increases sublinearly as the number of processes and in the number of communication operations increase.
- The run-time information is exploited rather than limiting ourselves to information available at compile time.
- We preserve all communication events and their ordering 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 which is discussed in Chapter 2. Other tools for acquiring communication traces, such as Vampir [5], Extrae/Paraver [16], and tools based on the Open Trace Format [9], do not have structure-aware compression. This results in the size of a trace file that grows linearly with the number of MPI calls and the number of MPI processes. This also increases 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 [10] but still falls short of structural compression of ScalaTrace, which extends to any event parameters. Casas et al. utilize techniques of signal processing to detect internal structures of Paraver traces and extract meaningful parts of the trace files [6]. This facilitates trace analysis in compressed manner but does not allow one to capture full information and becomes lossy and thus not suitable for benchmark generation.

Xu et al.'s work on constructing coordinated *performance skeletons* to estimate application execution time in new hardware environments [28, 29] exhibits many similarities with our work.

However, a key aspect of performance skeletons is that they filter out "local"' communication (communication outside the dominant pattern). As a result, the generated code does not fully reflect the original application, which may cause subtle but important performance characteristics to be overlooked. Because our benchmark generation framework is based on lossless application traces, it is able to generate benchmarks with identical communication behavior to the original application.

Program slicing, statically reducing a program to a minimal form that preserves key properties of the original one, offers an 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 [7]. 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 [19], 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 [30]. Program slicing and static benchmark generation in general have a number of shortcomings relative to our run-time, trace-based approach: (a) Their reliance on inter-procedural analysis requires that *all* source code be available. This includes application's complete source code along with the source codes of all its dependencies including libraries. (b) They lack run-time timing information. (c) They cannot accurately handle loops with data-dependent trip counts ("**while not** converged **do**…"). (d) They produce benchmarks that are neither human-readable nor editable.

Wu et al.'s work of generating the Conceptual benchmark [27] closely resembles to our work. ScalaTrace is used to collect the traces from the application in their work. A trace traversal framework, which is similar to our traversal framework, is used to generate the source code in Conceptual, a domain specific language [15]. This language focuses on generating networking/communication benchmarks. This work does not generate all MPI calls but maps the MPI events from the trace to the corresponding combination of communication routines. The Conceptual language does not have the concept of "communicator"" as in MPI. Thus it cannot form the subsets of ranks based on a communicator. Since our work generates C code with MPI calls, it can translate all MPI events captured in the trace accurately. The Conceptual language does not have provision like MPI_ANY_SOURCE, thus generated code needs to be resolved for the source in the send and receive communication calls. This eliminates the non-determinism present in the source code but changes runtime behavior relative to internal MPI queues which are used to buffer the receives till the matching sends are encountered. In our work, we reproduce the non-determinism present in the original application which is discussed in Section 3.9, thus accurately preserving the behavior of the application. Our work, generates lossless, accurate and human readable MPI communication calls in C source code from single trace file obtained from ScalaTrace, which is easily portable to any platform as opposed to Conceptual with the need to interpret Conceptual code, which more closely resembles trace replay.

Chapter 7

Conclusion

Benchmarks are required for assessing the capabilites of HPC systems. They aid in evaluating and analyzing software subsystems and hardware. Benchmarks are also used for application tuning to improve performance, e.g., by variations in data decomposition and transformations for parallelism. Using real HPC applications for such tuning is impractical because of large overheads. Also, sometimes sources of such applications are not public. Benchmarks are easy to port, modify, run and also provide an indication for characteristics of HPC applications. But most of the existing benchmarks do not capture the complexity and scale of realistic HPC applications.

In this work, we have presented a solution to the task of evaluating HPC systems and application tuning. We have designed and implemented a novel communication benchmark code generator, which generates benchmark code in C with MPI calls from communication traces. These traces are generated by ScalaTrace, a lossless and scalable framework to extract communication, I/O operations and execution time while abstracting away the computations. These benchmarks are human readable, compact, easy to generate and port. They also preserve the behavior of the original application in terms of execution time, communication volume and ordering of events.

Experimental results demonstrate the ability of our code generator to generate the communication benchmarks from NAS Parallel Benchmark Suite and Sweep3D. The obtained results show that the benchmarks accurately preserve the application semantics and overall execution time.

Thus, we have achieved the goals we had set and have proven our hypothesis stated initially. The developed tool can benefit application developers, communication researchers and HPC system designers. This tool assists in perfomance analysis of software, hardware and also easy migration of applcations across different platforms.

REFERENCES

[1] `http://en.wikipedia.org/wiki/Supercomputer`.

[2] `http://moss.csc.ncsu.edu/~mueller/arc.html`.

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

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[5] Holger Brunst, Hans-Christian Hoppe, Wolfgang E. Nagel, and Manuela Winkler. Performance optimization for large scale computing: The scalable VAMPIR approach. In *International Conference on Computational Science (2)*, pages 751–760, 2001.

[6] Marc Casas, Rosa Badia, and Jesus Labarta. Automatic structure extraction from mpi applications tracefiles. In *Euro-Par Conference*, August 2007.

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

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

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

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

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

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

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

[14] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces in high performance computing. *Journal of Parallel Distributed Computing*, 69(8):969–710, August 2009.

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

[16] Vincent Pillet, Vincent Pillet, Jesus Labarta, Toni Cortes, and Sergi Girona. PARAVER: A tool to visualize and analyze parallel code. In Patrick Nixon, editor, *Proceedings of the 18th Technical Meeting of WoTUG-18: Transputer and Occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, Manchester, United Kingdom, April 9–12, 1995. IOS Press.

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

[18] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 16(2–3):105–121, 2008.

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

[20] Sameer S. Shende and Allen D. Malony. The Tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, Summer 2006.

[21] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1. MIT Press, 2 edition, 1998.

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

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

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

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

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

[27] Xing Wu, Frank Mueller, and Scott Pakin. Automatic generation of executable communication specifications from parallel applications. In *ICS*, pages 12–21, 2011.

[28] Qiang Xu, Ravi Prithivathi, Jaspal Subhlok, and Rong Zheng. Logicalization of MPI communication traces. Technical Report UH-CS-08-07, Dept. of Computer Science, University of Houston, 2008.

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

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