ABSTRACT

WU, XING. Scalable Communication Tracing for Performance Analysis of Parallel Applications. (Under the direction of Frank Mueller.)

Performance analysis and prediction for parallel applications is important for the design and development of scientific applications, and for the construction and procurement of high-performance computing (HPC) systems. As one of the most important approaches, application tracing is widely used for this purpose for being able to provide the computation and communication details of an application. Recent progress in communication tracing has tremendously improved the scalability of tracing tools and reduced the size of the trace file, and thereby opened up novel opportunities for trace-based performance analysis for parallel applications.

This work focuses on domain-specific trace compression methodology and puts forth fundamentally new approaches to improve the communication tracing techniques. Facilitated by the advances in this area, novel algorithms are further designed to address the hard problem of performance analysis, prediction, and benchmarking at scale. Specifically, this work makes the following contributions:

1. This work contributes ScalaExtrap, a fundamentally novel performance modeling scheme and tool. With ScalaExtrap, we synthetically generate the application trace for large numbers of MPI tasks by extrapolating from a set of smaller traces. We devise an innovative approach for topology extrapolation of SPMD (Single Program Multiple Data) codes with stencil or mesh communication. The extrapolated trace can subsequently be used for trace-based simulation, visualization, and detection of communication inefficiencies and scalability limitations at scale.

2. This work contributes novel methods to automatically generate highly portable and customizable communication benchmarks from HPC applications. We utilize ScalaTrace to collect selected aspects of the run-time behavior of HPC applications. We then generate portable and easy-to-read benchmarks with identical run-time behavior from the collected traces with C and the rich-featured coNCePTuaL network benchmarking language. Because our approach supports code obfuscation, it is particularly valuable for proprietary, export-controlled, or classified applications.

3. This work contributes novel algorithms to improve the trace compression and replay for SPMD applications. Built on our past experience with ScalaTrace, a spectrum of compression techniques, including elastic data element representation, approximate loop matching, loop agnostic inter-node compression, and so on, are designed to improve the

trace compression for applications with iteration-specific program behavior and diverging parallel control flow. A fully distributed replay tool for probabilistic traces is also developed for the reproduction of the computation performance of the original application. The respective design has been implemented in ScalaTrace 2, the next generation of the ScalaTrace tracing infrastructure.

Overall, this work is centered around scalable tracing of parallel applications. Built upon the prior research, it contributes novel approaches on communication trace compression and trace-based performance analysis. To the best of our knowledge, the algorithms and techniques proposed in this work are without precedence.

Scalable Communication Tracing for Performance Analysis of Parallel Applications

by
Xing Wu

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

Xiaosong Ma	Yan Solihin
Xiaohui Helen Gu	Scott Pakin

Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents and my lovely wife, Yi Tang, for their tremendous support and constant love.

BIOGRAPHY

Xing Wu was born and raised in Chengdu, China. He attended ShiShi high school in Chengdu. Thereafter, he began his undergraduate studies in University of Electronic Science and Technology of China (UESTC). In 2005, he interned with Infosys Limited in Bangalore, India. In 2006, he received his Bachelor of Engineering degree in Computer Science from UESTC. He then joined the master's program in Computer Science in the same university, where he developed a keen interest in parallel and distributed computing. In 2008, Xing joined the Department of Computer Science in North Carolina State University (NCSU) to pursue a Ph.D degree. Under the guidance of Dr. Frank Mueller, he specialized in the performance analysis of parallel applications with a focus on scalable approaches for application tracing. During his doctoral studies, he also interned in Los Alamos National Laboratory and Amazon.com Inc. He will be joining Amazon.com Inc. after graduation.

ACKNOWLEDGEMENTS

All of a sudden, when I finally had the opportunity to write down my acknowledgement for this dissertation, I came to realize that it is almost the end of my Ph.D journey. In retrospect, the memories of so many first times are still fresh: the first time fighting a twelve-hour jet lag in an operating systems class that I could not really understand, the first time sitting in Room 3266 struggling to comprehend a complication called ScalaTrace, the first time receiving a rejection for an ambitious paper submission, the first time standing behind a podium giving a conference presentation, the first time working in a national lab at an once secret location, the first time passing a job interview and getting a position in a dream company ... . Throughout the journey, there were disappointment, depression, frustration, and suffering, but there were also excitement, cheerfulness, brilliance, and most importantly, achievements, which are never possible without the encouragement, guidance, and support from the people to whom I shall show my gratitude.

Firstly, I would like to thank my family. I thank my parents for supporting their only child to pursue a graduate degree in a country that is thousands of miles away. I thank my wonderful wife, Yi, for enduring the loneliness of being geographically apart for ten years, and for all the unselfish sacrifice thereafter. Their constant understanding and tremendous support encouraged me to move on.

Secondly, I would like to thank my advisor Dr. Frank Mueller, for his patience and encouragement to a beginner in the first few years, and for his professional guidance and invaluable advice throughout my studies. His wisdom, expertise, and insightful thoughts motivated me and inspired my research. I would also thank Dr. Xiaosong Ma, Dr. Xiaohui Gu, Dr. Yan Solihin, and Dr. Scott Pakin for serving on my advisory committee and giving me invaluable suggestions on my dissertation. Particularly, I would like to give special thanks to Dr. Scott Pakin for being my mentor during my internship at Los Alamos National Laboratory and being my collaborator of one of my papers. Since we worked together, he has given me substantial help on my research.

Last but not least, I would like to thank Abhik Sarkar, Chris Zimmer, Yongpeng Zhang, David Fiala, Arash Rezaei, James Elliott, Feng Ji, and Fei Meng for making Room 3226 a forum, a coffee/tea house, a soccer field, or anything other than a research lab. I would like to thank Karthik Vijayakumar and Vivek Deshpande for being my research collaborators. I would like to thank Zhengzhang Chen, Zhenhuan Gong, Yongmin Tan, and Yaogong Wang, for the nights with "music and beer". I am also indebted to my 16-year friend, Yangyang Liu, who encouraged me to pursue a Ph.D and offered me great help thereafter.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

1.1.1 The Recent History of Supercomputers

Processor counts and Flops (FLoating-point Operations Per Second) in modern supercomputers are rising exponentially. Back in June 1993, when the first Top500 list [1] was announced, the CM-5/1024, the world's fastest supercomputer from Los Alamos National Laboratory at the time had only 1,024 cores and a maximal LINPACK [21] performance (R_{max}) of 59.7 GFlops. Four years later, in June 1997, the world's first teraflop supercomputer, ASCI Red from Sandia National Laboratories with 7,264 cores and an R_{max} of 1,068 GFlops, became the number one system in the world. In June 2008, the Roadrunner system at Los Alamos National Laboratory with 122,400 cores and an R_{max} of 1,026.00 TFlops brought the global high performance computing community into the era of petascale for the first time. As of the publication of the latest Top500 list (November 2012), all of the top 20 systems have achieved petaflop/s performance. Titan, the Cray XK7 supercomputer at the Oak Ridge National Laboratory capable of performing more than 17 quadrillion calculations per second (PFlops), currently occupies the first place in the list. It (together with the June 2012 champion, Sequoia BlueGene/Q with 1,572,864 cores and an R_{max} of 16.3 PFlops at the Lawrence Livermore National Laboratory) exceeds the maximum computation capability of Roadrunner by 16 times after a period of just 4 years — equivalent to a doubling of performance every year! Based on the historical statistics of the past two decades, it is not unreasonable to expect the advent of exascale computing in the near future.

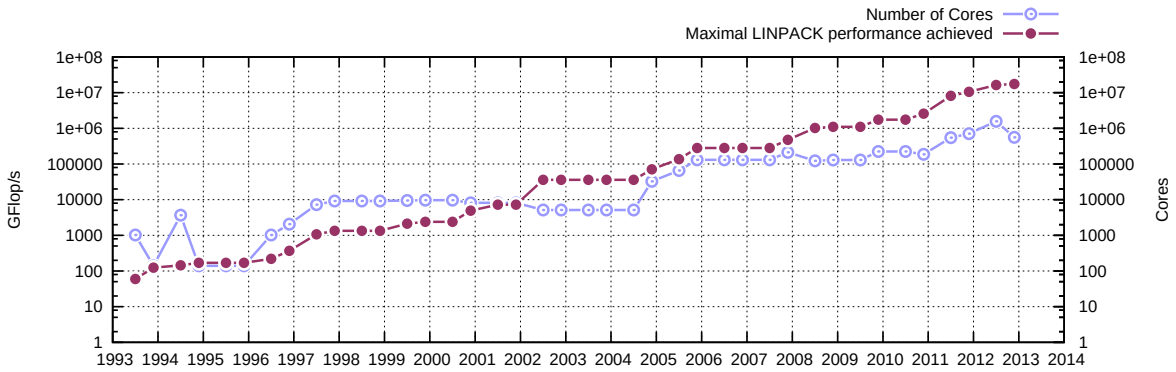


Figure 1.1: Performance Development of Supercomputers Since June 1993

1.1.2 Application Trace for Performance Analysis and Prediction

Performance analysis and prediction for scientific applications is important for assessing potential application performance and HPC systems procurement. However, as supercomputers progress in scale and capability toward exascale levels, characterization of communication behavior and its impact on the overall application performance is becoming increasingly difficult due to the application size and system complexity.

Performance modeling is an important approach to predict application behavior. Generally, this approach takes a number of machine and application parameters as input. It utilizes a set of formulae to assess the performance of an application when it is executed on a particular architecture and at a certain scale. Nonetheless, measuring the system and application performance parameters is non-trivial given the complexity of supercomputers and large-scale scientific applications. In addition, this approach provides only the predicted overall statistics for an application. Without detailed application runtime information, neither more sophisticated static analysis nor post-mortem performance debugging is possible.

Profiling is another widely used method for performance analysis and debugging of scientific codes that utilize MPI-style message passing [34]. Through binary instrumentation or utilizing the MPI profiling layer, profiling tools, such as mpiP [75], are able to gather runtime information such as the execution times of the functions and the message volume exchanged in the network. Nevertheless, as a light-weighted approach, profiling provides only the aggregated statistics of an application. Without the structural and temporal ordering of events and the information about each communication event and each computation region, in-depth and fine-grained performance analysis can hardly be done.

Alternatively, application tracing can be used to capture complete application behavior. Typically, MPI tracing is implemented with the MPI profiling interface, which is a set of wrap-

per functions for the actual MPI functions. By intercepting MPI calls at the profiling interface, the MPI profiling system can record useful information including the parameters passed to a subroutine, timestamps, the duration of a subroutine, any statistics from hardware performance counters, etc. Since event logs are stored in a trace file in the order that the corresponding MPI calls were issued, the application trace is often able to preserve the temporal and causal ordering of events. With complete information about the runtime behavior of an application, tracing can be used to pinpoint the root cause of the performance problems. However, while a number of tracing tools for communication exist, their storage requirements do not scale well. A full-blown communication trace with per-event timestamps not only requires a high-bandwidth parallelized I/O backbone to collect the trace. It also mandates a parallelized approach to analyze or visualize such traces. Hence, even with the complete information about an application, performance analysis and debugging is still non-trivial for large-scale applications.

1.1.3 ScalaTrace

This work focuses on novel performance analysis techniques that are built upon ScalaTrace. ScalaTrace is a lossless yet scalable communication tracing library [56, 64]. More specifically, ScalaTrace records the communication events in a fully lossless manner with their causal ordering preserved, while it preserves the execution times of communication and computational stages with a histogram-based lossy approach to capture the original performance characteristics [64]. It utilizes the MPI profiling layer to intercept MPI communication calls and record parameter values and durations of computational regions. A unique feature that distinguishes ScalaTrace from previous approaches is its capability of performing structure-preserving compression. At runtime, ScalaTrace performs an on-the-fly intra-node loop compression on each node to obtain a scalable trace size with respect to the repeating workload caused by timestep simulation in parallel scientific applications. At application termination, ScalaTrace performs an inter-node compression along a radix tree to further merge the per-node traces to a single one. By exploiting the similarities between per-node traces caused by the SPMD nature of the scientific applications, inter-node compression allows the trace size to be scalable with respect to the total number of MPI tasks.

ScalaReplay is a replay engine designed to replay the ScalaTrace traces. At runtime, it interprets the application trace and issues MPI calls in the order that they were performed in the original application. ScalaReplay supports timed replay by sleeping for the number of time units recorded for the computational phases. With ScalaReplay, the application runtime behavior can be accurately reproduced for post-mortem performance analysis and cross-platform performance prediction.

1.2 Hypothesis

With supercomputer's computation power doubled each year and the system size continuously increasing, the HPC community will embrace the era of exascale in the near future. With such massive-scale systems, development, debugging, and performance analysis of parallel applications are becoming increasingly difficult due to the lack of methods and tools that are efficient at scale. We attempt to address this limitation in this dissertation. Hence, the hypothesis of this dissertation is:

By exploiting the repetitive nature of time step simulation and the HPC-prevalent SPMD programming style, it becomes feasible to preserve the communication and runtime behavior of parallel applications in a lossless or near lossless fashion, while still ensuring scalability of tracing capabilities. Such scalable tracing methodology has the potential to enable innovative performance analysis and benchmarking techniques that are otherwise impractical with the past approaches.

1.3 Contributions

1.3.1 Contributions

This work puts forth fundamentally new approaches to improve communication tracing techniques. We focus on domain-specific trace compression schemes for parallel applications that utilize the Message Passing Interface. Facilitated by advances in this area, novel techniques and algorithms were designed to address the hard problems of performance analysis, prediction, and benchmarking at scale. Specifically, this work makes the following contributions:

1. Scalability is one of the main challenges of scientific applications in HPC. Estimating the impact of scaling on communication efficiency is non-trivial due to execution time variations and exposure to hardware and software artifacts. This work contributes ScalaExtrap, a fundamentally novel modeling scheme and tool. We synthetically generate the application trace for large numbers of nodes by extrapolation from a set of smaller traces. We devise an innovative approach for topology extrapolation of SPMD codes with stencil or mesh communication. The extrapolated trace can subsequently be (a) replayed to assess communication requirements before porting an application, (b) transformed to auto-generate communication benchmarks for various target platforms, and (c) analyzed to detect communication inefficiencies and scalability limitations.

2. Portable parallel benchmarks are widely used and highly effective for performance analysis and evaluation of scientific codes and for HPC system procurement. Yet, past techniques to synthetically parametrized hand-coded HPC benchmarks prove insufficient for today's

rapidly-evolving scientific codes particularly when subjected to multi-scale science modeling or when utilizing domain-specific libraries. To address these problems, this work contributes novel methods to automatically generate highly portable and customizable communication benchmarks from HPC applications. We utilize ScalaTrace to collect selected aspects of the run-time behavior of HPC applications while abstracting away the details of computation. We subsequently generate portable and easy-to-read benchmarks with identical run-time behavior from the collected traces with coNCePTuaL. coNCePTuaL is a domain-specific language that enables the expression of sophisticated communication patterns using a rich and easily understandable grammar yet compiles to ordinary C+MPI. Such automated benchmark generation is particularly valuable for proprietary, export-controlled, or classified application codes: when supplied to a third party, our auto-generated benchmarks ensure performance fidelity without the risks associated with releasing the original code.

3. While a number of communication tracing tools exist, they either produce trace files with non-scalable sizes, or only gather aggregated runtime statistics without preserving the program structure and temporal event ordering. ScalaTrace introduces effective communication trace representation and compression techniques that enable scalable application tracing. This work contributes ScalaTrace 2, the next generation ScalaTrace that delivers even higher trace compression capability. In this work, a spectrum of compression techniques, including elastic data element representation, approximate loop matching, loop agnostic inter-node compression, and so on, are designed to improve the trace compression for applications with iteration-specific program behavior and diverging parallel control flow. A fully distributed replay tool for probabilistic traces is also developed for the reproduction of the computation performance of the original application. With ScalaTrace 2, we significantly improve on today's the state-of-the-art compression capabilities.

Experiments were performed to evaluate the proposed approaches for trace-based performance analysis. Results demonstrate that 1) the extrapolated trace is able to predict the performance characteristics of an application at scale, 2) the generated benchmarks can accurately preserve the runtime behavior of the original applications for performance analysis, and 3) ScalaTrace 2 achieves key improvements on trace compression for applications with inconsistent time step behavior and diverging task level behavior compared to its predecessor, ScalaTrace 1.

Overall, this work is centered around scalable tracing of parallel applications. Building upon prior research, it contributes novel approaches on communication trace compression and trace-based performance analysis that would otherwise be infeasible. To the best of our knowledge,

the algorithms and techniques proposed in this work are without precedence.

1.3.2 Assumptions and Scope

The methodologies and techniques proposed in this work target parallel applications utilizing the Message Passing Interface (MPI), the *de facto* standard for scientific computing. The principles of compression are applicable to other domains as well, e.g., memory trace compression [50, 49, 13], but this work focuses on MPI communication traces. Particularly, the work presented in Chapter 3 targets SPMD codes with regular stencil or mesh style communication patterns. It assumes that the nodes are numbered in a row-major fashion. It makes the assumption that an application's communication pattern is linearly related to its communication topology. Being a trace-based approach requiring no binary instrumentation or source code analysis, this work also assumes that the communication patterns and computational times evolve continuously with the scale of the execution in a predictable manner. Should any of these assumptions do not hold for a particular code, the ScalaExtrap approach will fail. For example, without the knowledge of a given node assignment scheme, identifying the communication pattern from the communication graph provided by a trace file is equivalent to solving the graph isomorphism problem, which is known to be NP hard [87]. Also, if the variation trend of the execution time of a certain computational stage follows a discontinuous function, our curve fitting approach will be insufficient to capture the discontinuity. The benchmark generation tool presented in Chapter 4 is generally applicable to any MPI applications that can be traced with ScalaTrace. It generates concise and easily understandable codes for applications demonstrating regular event patterns that can be exploited by ScalaTrace for trace compression. Similarly, ScalaTrace 2 (Chapter 5) is also applicable to all MPI programs in general. Nonetheless, by incorporating a set of novel compression algorithms, ScalaTrace 2 is particularly suitable for applications demonstrating inconsistent loop behavior and irregular SPMD behavior. Similar to the last generation ScalaTrace (see Chapter 2), ScalaTrace 2 also uses a hybrid approach where communication event tracing can be configured to be fully lossless, but the delta times of communication and computational stages are recorded in a lossy manner using histograms.

1.4 Organization

The remaining chapters are structured as follows. Chapter 2 introduces ScalaTrace 1, the prior research that inspired and enabled the work presented in this dissertation. Chapter 3 presents a trace-based extrapolation algorithm that extrapolates a large trace from several smaller traces for quick performance analysis at large scale. Chapter 4 describes an automatic benchmark generation framework that generates performance-accurate benchmarks for scientific applications from communication traces collected with ScalaTrace. Chapter 5 describes ScalaTrace 2,

a fundamental redesign of ScalaTrace that implements novel trace compression algorithms to cope with applications featuring complicated loop-level and task-level behavior. Chapter 6 discusses future work. Chapter 7 summarizes the contributions of this research and concludes this dissertation.

Chapter 2

An Overview of ScalaTrace

This chapter provides a brief overview of ScalaTrace. We introduce the internal compression mechanisms and the unique features of ScalaTrace that serve as the basis for the work presented in Chapter 3 and Chapter 4. Some of these algorithms and encoding schemes are also adopted in the ScalaTrace 2 work presented in Chapter 5.

2.1 Intra-node and Inter-node Compression

ScalaTrace is an MPI communication tracing framework for parallel applications. It utilizes the MPI profiling layer (PMPI) to intercept MPI calls. It collects lossless communication traces where program structure, event ordering, and temporal information of the original applications are preserved. By utilizing a set of domain-specific compression techniques, ScalaTrace is able to generate space-efficient traces for SPMD codes regardless of the number of time steps and the task count during execution.

During application execution, ScalaTrace performs on-the-fly intra-node compression to capture the loop structure and represent MPI events in a compressed manner. Upon application completion, local traces are combined into a single global trace where matching events and loop structures are merged across node. Specifically, ScalaTrace utilizes Extended Regular Section Descriptors (RSDs) to record the parameters and information of a single MPI event nested in a loop. Power-RSDs (PRSDs) are utilized to recursively specify RSDs nested in multiple loops. For example, for the 4-point stencil code shown in Figure 2.1,

```
RSD1: {<rank>, MPI_Irecv, (NORTH, WEST, EAST, SOUTH)}
```

and

```
RSD2: {<rank>, MPI_Isend, (NORTH, WEST, EAST, SOUTH)}
```

denote the alternating send/receive calls to/from the 4 neighbors, and

```
PRSD1: {<rank>, 1000, (RSD1, RSD2, MPI_Waitall)}
```

denotes the outer loop with 1000 iterations. In the loop's body, RSD1, RSD2, and a succeeding MPI_Waitall are called sequentially. During inter-node compression, matching events are compressed by forming a *ranklist* , i.e., a set of task IDs, to describe the participants of the events. For example, the aforementioned task-level RSDs and PRSD become

```
RSD1: {<ranklist = 0,1,...,n>, MPI_Irecv, (NORTH, WEST, EAST, SOUTH)}

RSD2: {<ranklist = 0,1,...,n>, MPI_Isend, (NORTH, WEST, EAST, SOUTH)}

PRSD1: {<ranklist = 0,1,...,n>, 1000, (RSD1, RSD2, MPI_Waitall)}
```

```
neighbors[] = {NORTH, WEST, EAST, SOUTH};
for(i=0; i<1000; i++) {
    for(j=0; j<4; j++) {
        MPI_Irecv(neighbors[j]);
        MPI_Isend(neighbors[j]);
    }
    MPI_Waitall();
}
```

Figure 2.1: Sample Stencil Code for RSD and PRSD Generation

2.2 ScalaTrace Encoding Schemes

The key approaches to achieve scalable inter-node compression are the location-independent encoding and communication group encoding schemes detailed in the following.

- *Location-independent encoding:* Communication endpoints in SPMD programs differ from one node to another. By encoding endpoints *relative* to the index of an MPI task on a node, a location-independent denotation is created. The location-independent encoding helps to describe the behavior of large set of nodes that exercise a common communication pattern, e.g., a 5-point stencil. In a stencil/mesh topology, only few of such distinct sets/groups tend to exist. Consequently, location-independent encoding opens up opportunities for inter-node compression to unify endpoints across different computational nodes.

- *Communication group encoding:* Similarity in communication patterns is recognized to succinctly denote sets/groups of nodes with common behavior. In a topological space, a communication group refers to a subset of nodes that have identical communication patterns. With this encoding scheme, a communication group is represented as a ranklist. Using the EBNF meta-syntax, a ranklist is represented as

  $$< dimension\ start\_rank\ iteration\_length\ stride\ \{iteration\_length\ stride\} >,$$

  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 2.2. 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.

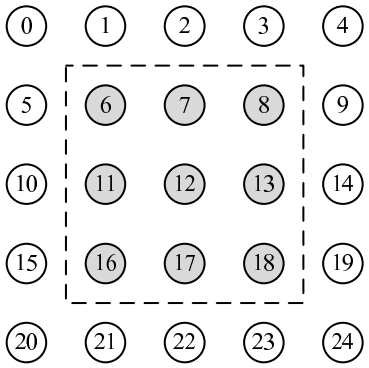


Figure 2.2: Ranklist Representation for Communication Group

2.3 Preserving Time in Communication Traces

Besides communication tracing, ScalaTrace also preserves the timing information of a parallel application in a scalable way [64]. Along with the intra-node and inter-node compression pro-

cesses, "delta" times representing the computation between communication events are recorded and compressed. For the purpose of scalability, delta times of a single MPI function call across multiple loop iterations and across MPI tasks are not recorded one by one. Instead, histograms with a fixed number of bins for delta times are dynamically constructed to provide a statistical view. Delta times are distinguished by not only the call context of recorded events, but also by their path sequence, which addresses significant variation of delta times caused by path differences, e.g., within entry/exit paths of a loop.

2.4 ScalaReplay

Finally, ScalaReplay is a replay engine operating on the application traces generated by ScalaTrace. It interprets the compressed application trace on-the-fly and issues MPI communication calls accordingly. During replay, all MPI calls are triggered over the same number of nodes with their original parameters (e.g., message payload size) but a randomly generated message content. This ensures comparable bandwidth requirements on communication interconnects. ScalaReplay emulates computation events in the original application by sleeping/busy waiting so that the communication contention characteristics are maintained during replay. In general, the replay engine can be utilized for rapid prototyping and tuning, as well as to assess communication needs of future platforms for large-scale procurements in conjunction with system simulators (Dimemas/SST) [44, 73, 65].

Chapter 3

ScalaExtrap: Trace Extrapolation for SPMD Programs

3.1 Introduction

Scalability is one of the main challenges for scientific applications in HPC. A host of automatic tools have been developed by both academia and industry to assist in communication gathering and analysis for MPI-style message passing [34]. Most of these tools either obtain lossless trace information at the price of poor scalability [52] or preserve only aggregated statistical trace information to limit the size of trace files as in mpiP [75]. Recent work on communication tracing and time recording made a breakthrough in this realm. ScalaTrace introduced an effective communication trace representation and compression algorithm [56]. It managed to preserve the structure and temporal ordering of events, yet maintains traces in a space-efficient representation. However, ScalaTrace needs to be linked to the original application and executed on a high-performance computing cluster of a *given number of compute nodes* to obtain a trace. Due to the often long application execution times and limited availability of cluster resources for large numbers of nodes, obtaining the trace information of a large-scale parallel application remains costly.

In this chapter, we describe ScalaExtrap, a methodology and tool to synthetically generate communication trace for an arbitrarily large number of MPI tasks. This work contributes a set of algorithms and techniques to extrapolate full communication traces and execution times of an application at larger scale with information gathered from smaller executions. Since extrapolation is based on analytical processing of smaller traces with mathematical transformations, this approach can be performed on a single workstation, much in contrast to analysis or visualization of large traces in contemporary tools e.g., Vampir Next Generation [10]). It thus enables, for the first time, the instant generation of trace information of an application at arbitrary

scale without necessitating time-consuming execution. Specifically, we extrapolate two aspects of the application behavior, namely (1) the communication trace events with parameters and (2) the timing information resembling computation. The extrapolation of the communication trace is based on the observation that, in many regular SPMD stencil and mesh codes, communication parameters and communication groups are related to the sizes and dimensions of the communication topology. Thus, extrapolation of communication traces becomes feasible with the detection of communication topologies and the analysis of communication parameters to infer evolving patterns. The extrapolation of timing information involves a process of analytical modeling. In order to mitigate timing fluctuations under scaling, we employ statistical methods.

Our extrapolation methodology is applicable for both strong and weak scaling applications. Weak scaling is typically defined as scaling the problem size and the number of processors at the same rate such that the problem size per processor is fixed. This should imply that the communication patterns generally evolve in a similar manner for both strong and weak scaling. Thus, we hypothesize that the same extrapolation algorithms for patterns and communication end points should apply to both. For communication parameters, such as message sizes and computation times different trends can be observed. But we hypothesize that extrapolation based on curve fitting is still applicable. In this work, we verify these hypotheses by evaluating our extrapolation algorithm with both strong and weak scaling applications.

Our extrapolation approach follows a trace analysis methodology independent of the tracing infrastructure and works for any of the existing trace formats. Nonetheless, the approach is significantly facilitated by ScalaTrace's compression scheme that preserves application structure with inherent compression that closely resembles the loop structure of an application. In contrast, extrapolation with other trace formats, such as OTF [40], would be far more tedious and time/space consuming as structure is neither established across nodes nor retained after binary-level compression.

This trace extrapolation approach has been implemented in the ScalaExtrap tool, which we utilize to evaluate our extrapolation approach with both strong and weak scaling benchmarks, including the NAS Parallel Benchmark suite [6] and Sweep3D [79]. We utilize up to 16,384 nodes of a 73,728-node IBM Blue Gene/P supercomputer to generate communication traces for extrapolation and verification. Experiments were performed to assess both the correctness of communication extrapolation and the accuracy of the timing extrapolation. Experimental results demonstrate that our topology detection algorithm is capable of identifying and characterizing stencil/mesh and collective communication patterns. Upon topology detection, the communication trace extrapolation algorithm correctly extrapolates all communication events, parameters and communication groups at an arbitrary target size for both stencil/mesh point-to-point and collective communication. The experiments also demonstrate that the extrapo-

lation of timing information resembles the running time of the original parallel application. Compared to the running time of the original application, the accuracy of replay times of the corresponding extrapolated trace is, in the majority of cases, higher than 90%, sometimes as high as 98%. Given the difficulty of extrapolating application execution time with only the time information obtained from several small executions, our approach achieves unprecedented accuracy that is sufficient for modeling, procurement and analysis tasks.

Overall, this work explores the potential to extrapolate communication behavior of parallel applications. Several novel algorithms for communication topology detection and communication trace extrapolation are introduced. Experimental results demonstrate that rapid generation of an application's trace information at arbitrary size is entirely possible, which is unprecedented. In contrast to tedious and application-centric model development, our approach opens new opportunities for automatically deriving communication models, facilitating communication analysis and tuning at any scale. Our work further enables system simulation at extreme scale based on a single file, concise communication trace representation. More specifically, HPC simulation tools (e.g., Dimemas or SST [44, 73, 65]), which currently cannot operate at petascale levels, could benefit by utilizing our extrapolated single-file traces that are just 10s of megabytes in size. Benchmark generation is important for cross-platform performance analysis due to its standard and portable source code and the platform-independent nature. Our work enables code generation at extreme scale by providing large traces that are otherwise unavailable. Furthermore, by contributing a set of detection techniques of communication patterns, our work has the potential to enable the generation of flexible and stand-alone programs that can be executed with arbitrary numbers of nodes and any possible input.

3.2 Communication Extrapolation

This work focuses on the extrapolation of communication traces and execution times. The respective design is subsequently implemented in a novel tool, ScalaExtrap. The challenge of communication trace extrapolation is to determine how the communication parameters change with node and problem scaling. The main idea is to identify the relationship between communication parameters and the characteristics of the communication topology, i.e., typically the sizes of each dimension. As a simple example, in Figure 2.2, assume *node 0* communicates with *node 4*, i.e., a node at distance of 4. If we can identify that the topological communication space is a grid consisting of 25 nodes with 5 nodes per row, we know that *node 0* actually communicates with the upper-right node. Therefore, when there are $1024 = 32 \times 32$ nodes, we can safely infer that *node 0* communicates with *node 31*, which is still the upper-right node.

Characterizing a communication pattern from one or more traces is non-trivial nonetheless. Without the knowledge of a given node assignment scheme and topology, identifying the

communication pattern from the communication graph provided by a trace file is equivalent to solving the graph isomorphism problem, which is known to be NP hard [87]. Therefore, instead of attempting to find a universal solution, we constrain our work to applications where

1. nodes execute the same program on different data, i.e., the application follow the SPMD paradigm;
2. nodes are numbered in a row-major fashion; and
3. communication is performed in stencil/mesh point-to-point manner or via collectives involving all MPI tasks.

In essence, our communication trace extrapolation algorithm first identifies the nodes at the "corner" of a topological space. It then calculates the sizes of each dimension of the topological space accordingly.

Upon acquiring the topology data, we can perform extrapolation. The extrapolation of a communication trace consists of two tasks. First, we need to match the records corresponding to the same MPI call in the source code across the traces of different node sizes. We will discuss the difficulties involved in this step and our solutions in the following sections. Second, for each MPI call in the source code, we need to determine which MPI processes execute this call and what are the values of the parameters when the application is running at the target scale. For the second task, we represent the ranklist and the communication parameters, e.g., the destination rank of MPI_Send, as a function of the known topology data and their undetermined coefficients. In order to calculate these coefficients, we correlate multiple traces and construct a set of linear equations. Finally, we employ Gaussian Elimination to solve the set of equations. With the fixed coefficients, we can extrapolate the value of the desired communication parameter by simply substituting the topology data with their values at the desired problem size.

The second aspect of this work concerns the extrapolation of program execution time. In the input trace files, computation time and communication time between (and optionally during) MPI communication events are preserved statistically with histograms. When analyzing the corresponding delta time, scaling trends can be identified across different number of nodes. Therefore, statistical curve fitting methods are utilized to model an evolving trend and extrapolate the execution time to a desired target size. In order to eliminate outliers, we further introduce several confidence coefficients to statistically determine the best extrapolated value under such constraints.

3.2.1 Topology Identification

Topology identification is the basis of communication trace extrapolation. In order to identify a topology, it is important to find the nodes at the corner or on the boundary of a topo-

logical space, which we call *critical nodes*. We devised a three-step approach to identify the communication topology.

1. We create an adjacency list of communication endpoints for each node and group nodes according to their adjacency lists.
2. We identify critical nodes by analyzing the adjacency lists.
3. We calculate the sizes of each dimension (x, y, and z) of the communication topology.

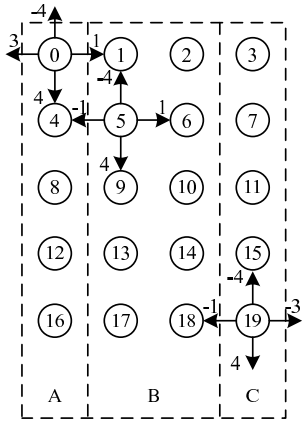


Figure 3.1: Topology Detection

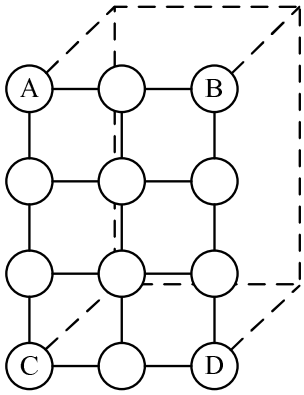


Figure 3.2: Boundary Size Calculation

First, our algorithm traverses the input trace to construct communication adjacency lists for each node. According to the relative positions (encodings) of all the communication endpoints of each node, nodes with same endpoint patterns are placed into the same group. Figure 3.1 illustrates an example of a 2D mesh topology. In this example, nodes on the boundaries communicate with nodes at the opposite side in a wrap-around manner while the internal nodes communicate with their immediate neighbors. Note that wrapping around in the vertical direction does not lead to different endpoint encoding. Therefore, the nodes are divided in to three groups (A, B, and C) with group sizes 5, 10, and 5, respectively.

Next, we analyze the adjacency list of each node to identify the critical nodes. Exploiting the row major constraint, we scan all nodes sequentially to identify loop structures with respect to communication adjacency list patterns. The underlying rationale is that critical nodes define a topology. Between corresponding critical nodes, communication patterns emerge repeatedly. According to the length of a loop structure, the sizes of the groups consist of critical nodes,

i.e.,*critical groups*, are calculated as

$$critical\ group\ size = \frac{n}{length\ of\ loop},$$

where n denotes the number of nodes engaged in MPI communication. For example, in Figure 3.1, each row has the same group distribution (A B B C) and is thus identified as a single iteration of the loop structure. Since the length of such a loop iteration is 4, the size of the critical groups (group A and C) is $20/4 = 5$. Having obtained the size of the critical groups, we then associate critical nodes with groups by matching sizes of critical groups.

Finally, we calculate the sizes of each dimension. Again exploiting the row-major constraint, in a d-dimensional topological space, the number of nodes at the *d-th* dimension is the total number of nodes. The number of nodes at the *i-th* $(i < d)$ dimension, n_i, is the inclusive range of numbers of nodes between *node 0* (*1st* critical node) and the 2^i*-th* critical node. Once we have determined the number of nodes at each dimension, the boundary size of the *i-th* dimension, s_i, is calculated as

$$s_i = \frac{n_i}{n_{i-1}}$$

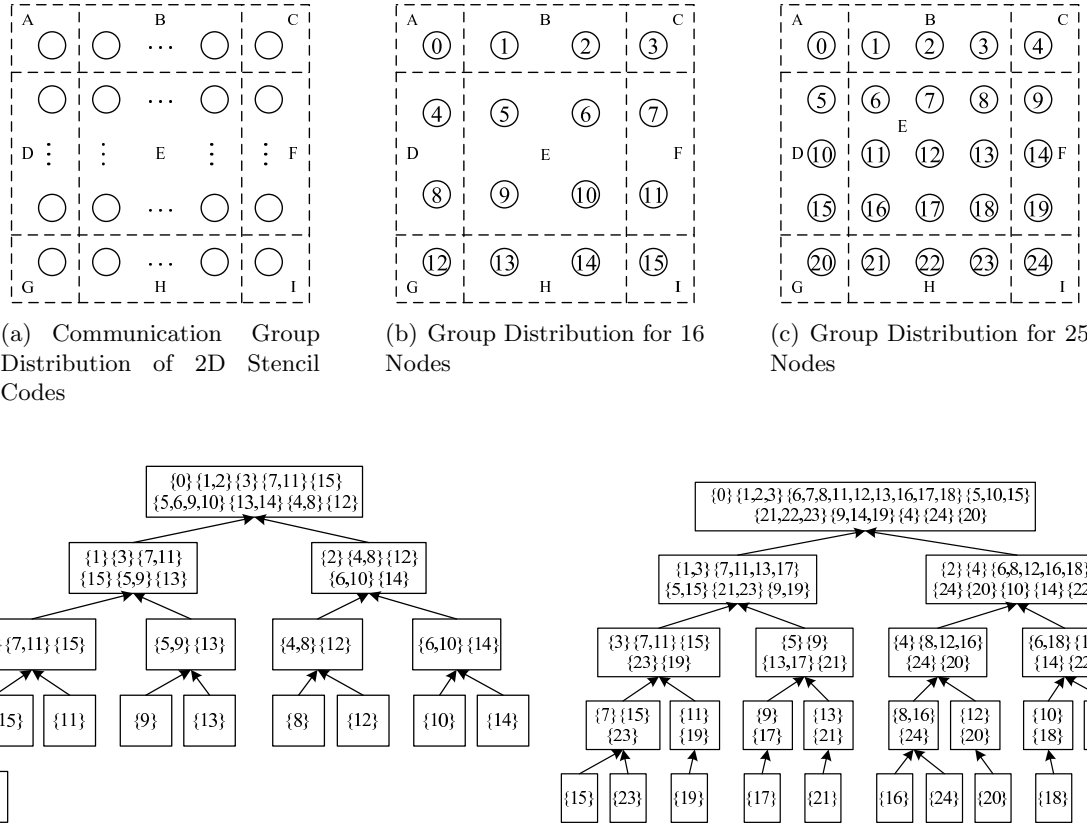
For example, in the 3D topology of Figure 3.2, the number of nodes in the *1st* dimension, n_1=*3*, is the number of nodes between A and B inclusively, the number of nodes in the second dimension, n_2=*12* , is the number of nodes between A and D, and the number of nodes in the third dimension n_3 is the total number of nodes. Hence, we have

$$\begin{cases} x = s_1 = n_1/n_0 = 3 \\ y = s_2 = n_2/n_1 = 4 \\ z = s_3 = n_3/n_2 \end{cases}$$

3.2.2 Matching MPI Events for Extrapolation

The extrapolation of a trace is performed one-by-one for each recorded MPI event of the trace. An MPI event is emitted per execution of an MPI function in the source code by the actual values of the input parameters. Therefore, the extrapolation of an MPI event is actually the process of inferring the execution of an MPI function at the target scale from its executions at smaller scales, which are represented as RSDs in the input traces. (In addition, due to the SPMD nature of parallel applications, the extrapolation also involves the prediction of the participants of an MPI event, i.e., the callers of an MPI function in the source code, which will be discussed in Section 3.2.3.) Therefore, being able to match the RSDs corresponding to calls to the same MPI function originating from source code across traces of different node sizes is the prerequisite of extrapolation.

Due to its structure-preserving representation, ScalaTrace traces are often similar to the



(a) Communication Group Distribution of 2D Stencil Codes

(b) Group Distribution for 16 Nodes

(c) Group Distribution for 25 Nodes

(d) Inter-node Compression for 16 nodes

(e) Inter-node Compression for 25 nodes

Figure 3.3: Inter-node Compression and the Positions of Communication Groups

source code. In a trace, the queue of RSDs represents the temporal ordering of the MPI events, which in turn reflects the locations of the corresponding MPI function invocations in the source code. Therefore, in most cases, traces of different node sizes are inherently aligned. However, nodes are sometimes partitioned due to differences in their communication patterns and may thus form different communication groups. For example, Figure 3.3(a) shows the distribution of the communication groups of 2D stencil codes such as Sweep3D. Since the communication behavior is different across groups, ScalaTrace cannot merge the per-node traces but appends them sequentially. Because the inter-node compression are performed with a radix tree and the order of disjoint subsequences of MPI events are not maintained during compression [55], the relative positions of RSDs originating from different communication groups are not necessarily the same in traces of different node sizes. For example, in the final 16-node trace the third group is Group C (both in Figure 3.3(b) and in the root node of Figure 3.3(d)) while it is Group E in

the 25-node trace (in the root node of Figure 3.3(e)) . This illustrates how the order of different communication groups are determined along with the radix tree style inter-node reduction (cf. Figures 3.3(d)+(e)). Clearly, extrapolating by relating RSDs of different communication groups is meaningless.

Algorithm 1 Aligning the Communication Groups

Precondition: T_{in}: input trace

Postcondition: T_{out}: output trace in which branches (RSD subsequences for communication groups) are ordered by rank

```
1: procedure REORDER_TRACE(T_in)
2:     for iter ← T_in.head, T_in.tail do
3:         if iter is a merging RSD node then
4:             merging_node ← iter
5:             find branching_node: merging_node's matching branching RSD node
6:             REORDER(merging_node, branching_node)        ▷ reorder the branches between merging_node and branching_node by rank
7:         end if
8:     end for
9: end procedure

10: procedure REORDER(merging_node, branching_node)
11:     for each branch between merging_node and branching_node do
12:         traverse branch in depth-first order
13:         if m: a merging RSD node is found in branch then
14:             find b: m's matching branching RSD node
15:             REORDER(m, b)                                ▷ recursively reorder the branches
16:         end if
17:     end for
18:     sort the branches between merging_node and branching_node by rank
19:     reorder the branches
20: end procedure
```

We utilize the dependence graph to reorder the trace. The dependence graph is a data structure used by ScalaTrace to keep track of disjoint RSD subsequences during the inter-node reduction [56]. If two per-node traces are partially different, a branching point and a merging point will be inserted before the first and after the last non-matching RSDs. We designed a recursive algorithm that traverses the dependence graph in a depth-first manner and topologically sorts each branch in the rank order (Algorithm 1). Our reordering algorithm guarantees that the RSD subsequences corresponding to different communication groups are

always organized in ascending order of the rank for the leading nodes of groups. With such an algorithm, we are able to align the communication groups in traces of different node sizes. The extrapolation is subsequently becomes possible.

3.2.3 Extrapolation of MPI Events

The extrapolation of an MPI event consists of the extrapolation of both communication groups and communication parameters to indicate who communicates and how they communicate. The extrapolation algorithm is based on the observation that, in regular SPMD stencil/mesh codes, *strong scaling* (increasing the number of nodes under a constant input size) linearly increases/decreases the value of communication parameters and the topological sizes. Given several data points, a fitting curve can be constructed to extrapolate the growth rate of the communication parameters and the topology information (the sizes of each dimension) of the communication groups.

Specifically, in an n-dimensional Cartesian space, the coordinates of node X and Y are $(X_1, X_2, ..., X_n)$ and $(Y_1, Y_2, ..., Y_n)$, where X_i and $Y_i \in [0, S_i - 1]$ and S_i is the size of the *i-th* dimension of the topological space ($1 \leq i \leq n$). Assuming the locations of node X and Y differ only in the *i-th* dimension, the distance between X and Y in the *i-th* dimension is $d_i = X_i - Y_i$. With the assumption of linear correlation between topology size and communication parameters, $d_i = X_i - Y_i = a_i \times S_i + b_i$, where a_i and b_i are two constants. Furthermore, with the row-major node placement assumption, the rank of an arbitrary node $A(A_1, A_2, ..., A_n)$ is

$$Rank_A = \sum_{i=1}^{n} A_i \prod_{j=1}^{i-1} S_j.$$

Therefore, $d_i{}'$, the rank distance between X and Y, is

$$d_i{}' = (X_i - Y_i) \times \prod_{j=1}^{i-1} S_j = (a_i \times S_i + b_i) \times \prod_{j=1}^{i-1} S_j$$

In general, for two arbitrarily selected nodes M and N, their rank distance d' is the sum of

their rank distances in each dimension,

$$\begin{aligned} d' &= d_0' + d_1' + ... + d_n' \\ &= \sum_{i=1}^{n} (N_i - M_i) \prod_{j=1}^{i-1} S_j = \sum_{i=1}^{n} (a_i \times S_i + b_i) \prod_{j=1}^{i-1} S_j \\ &= a_n \prod_{j=1}^{n} S_j + \sum_{i=1}^{n-1} (a_i + b_{i+1}) \prod_{j=1}^{i} S_j + b_1 = \sum_{i=0}^{n} c_i \prod_{j=1}^{i} S_j, \end{aligned}$$

where $c_n = a_n$, $c_0 = b_1$, and $c_i = a_i + b_{i+1} (1 \leq i \leq n-1)$.

In order to extrapolate the rank of a communication endpoint (src/dest), which is defined by the rank distance between nodes, we need to identify how the topology information is related to the communication parameter. We construct a set of linear equations to solve c_i (1≤i≤n-1). In general, for an n-dimensional topology, $n+1$ input traces are needed to solve $n+1$ coefficients. We employ Gaussian Elimination to solve the equations. Once the values of $c_i (1 \leq i \leq n-1)$ are determined, a fitting curve for the given parameter is established. In order to extrapolate the same parameter for a larger execution, we utilize the known coefficients and specify the topology information at the target task size. The desired value is then calculated accordingly.

As an example, in a 2D space, the bottom-right node in Figure 3.4 communicates with its *EAST* neighbor in a wrap-around manner. In order to extrapolate the rank of the communication endpoint, three input traces with dimensions 4×4, 5×5, and 6×6 are used to construct the set of linear equations shown in Figure 3.5, and $c_2 = 1$, $c_1 = -1$, and $c_0 = 1$ are obtained as the values of the coefficients. To extrapolate a 10×10 mesh, we re-construct the equation with coefficients and topology information assigned. Subsequently, the target value V is calculated as $V = c_2 \times 10 \times 10 + c_1 \times 10 + c_0 = 91$.

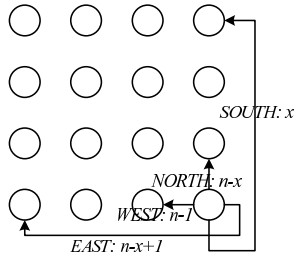


Figure 3.4: Generic Representation of Communication Endpoints

$$\begin{cases} c_2 \times 4 \times 4 + c_1 \times 4 + c_0 = 13 \\ c_2 \times 5 \times 5 + c_1 \times 5 + c_0 = 21 \\ c_2 \times 6 \times 6 + c_1 \times 6 + c_0 = 31 \end{cases}$$

Figure 3.5: Set of Equations for Communication Endpoint Extrapolation

Besides the communication parameters, communication groups are also extrapolated. The topological space of an application can be partitioned into several communication groups according to the communication endpoint pattern of each node. Under strong scaling, partitions tend to retain their position within the topological space but change their sizes for each dimension accordingly. For example, Figure 3.6 shows the distribution of 9 communication groups of a 2D stencil code. Despite the changing problem size, groups *A*, *C*, *G*, and *I* always represent corner nodes, groups *B*, *D*, *F*, and *H* are always the boundaries, and group *E* contains the remaining (interior) nodes.

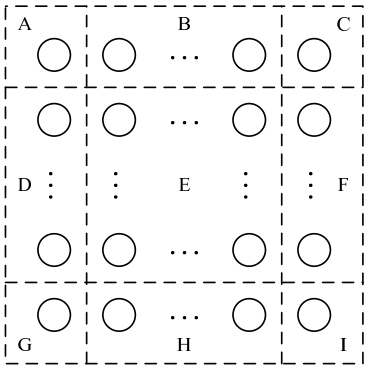


Figure 3.6: Distribution of Communication Groups of a 2D Stencil Code

This opens up the opportunity to extrapolate communication groups of the same application at arbitrary size. In order to extrapolate, we represent communication groups as ranklists, which effectively specifies the starting node and the dimension sizes of a group. Since the dimension sizes are defined by the distances between nodes (vertices), we again utilize a set of linear equations to establish the relation between the topology information of communication groups and the task sizes. Extrapolation is performed for the *start_rank*, *iteration_length*, and *stride* fields of the ranklist. The output ranklist reflects the communication group at the target size. For example, for the topology shown in Figure 3.6, when the total number of nodes is 16, the ranklist of group *E*, as defined in Section 2.2, is *<2 5 2 4 2 1>*, i.e., a 2D space starting from *node 5* with x- and y-dimensions of size 2. Similarly, the ranklists of group *E* at sizes 25 and 36 are *<2 6 3 5 3 1>* and *<2 7 4 6 4 1>*, respectively. We can thus construct the set of linear equations for each field in the ranklist to derive a generic representation of the ranklist as:

$$< 2 \quad x+1 \quad x-2 \quad x \quad x-2 \quad 1 >.$$

Subsequently, assuming that we want to extrapolate for size 10×10, let x be 10, which yields the output ranklist<*2 11 8 10 8 1*> that precisely matches the ranklist representation of communication group E at this problem size.

By combining the extrapolation of both communication groups and communication parameters, we are capable of extrapolating the communication trace for a given application at arbitrary topological sizes.

3.2.4 Lossy Extrapolation

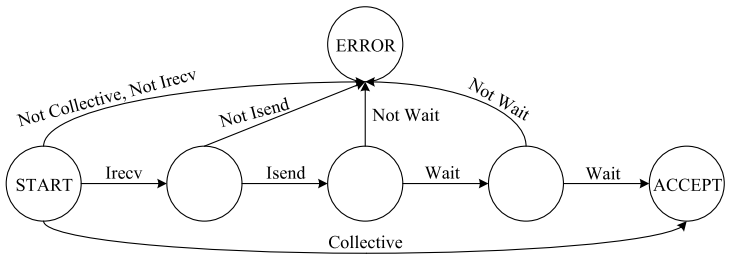
As was discussed in Section 3.2.2, aligning the matching RSDs across traces of different node sizes is critical for extrapolation. Despite being recognized as one of the existing tracing libraries that provides the best compression, there still exist some applications that ScalaTrace cannot obtain constant-size compression for. In fact, for applications that exhibit new communication patterns only at or beyond a certain node size, lossless yet constant-size compression is hardly possible. To extrapolate traces of such applications, we designed a lossy but still program structure-preserving approach. We attempt to capture and extrapolate the dominating communication pattern by optionally dropping events at three different levels: (1) within an MPI process, (2) among MPI processes of an execution, and (3) across traces of different node sizes.

The intra-node level event filtering is performed against the per-node queue of MPI events. We observe that a number of application traces contain a subsequence of events that embody the dominating communication pattern and comprise a large portion of the trace by repeating multiple times. Based on this observation, a user-provided trace snippet is utilized as the reference to drop events. Typically, such a trace snippet is a sequence of RSDs consisting of tens of MPI events, with the event type (Send, Recv, etc.) and the values of the key parameters of each event. Based on the trace snippet, we automatically generate a Finite-State Machine (FSM) to process the input stream of MPI events. At the beginning, the FSM is initialized to the START state. If the input event is a collective, the FSM directly enters the ACCEPT state. This indicates that all collectives are directly accepted while the FSM is not in the middle of accepting a sequence. If the input is neither a collective nor the first event to be accepted, the FSM enters the ERROR state and the input event is dropped. Once the FSM leaves the START state, it only accepts the next event expected in a sequence. If an unexpected event arrives, the FSM enters the ERROR state and all the pending events are dropped, including the current input if it is not a collective. Finally, if the FSM arrives at the ACCEPT state, the pending events are accepted. These events will not be affected by future ERROR states. Figure 3.7 shows a simple trace snippet and the FSM generated.

Beyond the intra-node level event filtering, if necessary, we also drop events during the inter-

```
<MPI_Irecv, (LEFT)>
<MPI_Isend, (RIGHT)>
<MPI_Wait>
<MPI_Wait>
```

(a) Trace Snippet



(b) Generated Finite-state Machine

Figure 3.7: A Simple Trace Snippet and the Generated Finite-state Machine

node compression and when aligning the traces of different node sizes for extrapolation. We designed a Longest Common Subsequence (LCS) based approach for the event filtering at these two levels. The LCS problem is to find the longest subsequence that is common to two input sequences, where a subsequence need not be consecutive in either of the original sequences. If a trace is considered as a sequence of MPI events, the LCS of two traces reflects the MPI events that nodes participate in for both traces. We adapted a well-known dynamic programming based LCS algorithm for trace comparison [9]. In a ScalaTrace trace, the loop structure is preserved and explicitly indicated. As a building block, loop structures should be evaluated in their entirety with the number of MPI events in the loop representing the weight. Therefore, we first enhanced the LCS algorithm to take into account the weight when evaluating how the length of the LCS will be affected by retaining or removing a loop structure. Second, since loops are often nested in the source code and in the trace, we further modified the LCS algorithm so that it can execute in two modes in a recursive manner. In the first pass, this algorithm only calculates the LCS but does not modify the trace. This is required because modifying the inner loop may affect the evaluation of the outer loop. Once the LCS is determined, this algorithm is applied again in the second mode such that any uncommon events are removed.

With these event filtering techniques above, we are able to extrapolate complicated and irregular applications depending on nodes counts such as the NPB MG kernel. We have developed a replay engine that infers receives from sends to replay a histogram-based communication trace [85]. In case the event filtering makes the trace incorrect, e.g., when sends and receives mismatch, this algorithm can be adapted to execute on a single machine to correct the trace. Our insight here is that instead of dropping the minor communication events that are not matched by any other MPI processes due to event filtering, preserving them by manually generating the matching events with probabilistic approaches may be a better solution [85].

3.2.5 Extrapolation of Timing Information

Besides the communication traces, we also extrapolate the timing information of the application. ScalaTrace preserves the "delta" time for each communication event and for the computation between two communication events. For a single MPI function call across multiple loop iterations, i.e., for a RSD, the delta times are recorded in multi-bin histograms. These histograms contain the overall average, minimum, and maximum delta time, the distribution of the delta execution times represented as histogram bins, and the average, minimum, and maximum delta time for each histogram bin. To extrapolate timing information, we utilize curve fitting to capture the variations in trends of the delta times with respect to the number of nodes, i.e., $t = f(n)$, where t is the delta execution time and n is the total number of nodes. Hence, the target delta time t_e is calculated as $t_e = f(n_e)$, where n_e is the total number of nodes at a given problem size. While we can extrapolate only the aggregated average delta time per RSD, to restrain the statistics of delta time, extrapolation is performed for each field of a histogram. Currently, we implemented four statistical models based on curve fitting for each extrapolation. We use a deviation-based metric to determine the best of these models to fit to a given curve.

1. Constant: This method captures constant time, i.e., $t = f(n) = c$. Before calculating the constant time, the input time t_o with the largest absolute value of deviation is excluded from the input times to mitigate the influence of outliers (which can be caused by either unstable system state or an empty bin). Subsequently, the average value of the remaining input times reflects the constant time c, and $d_1 = \textit{std. dev.}/\textit{average}$ is used to evaluate this fitting curve among the remaining values.

2. Linear: This method captures linearly increasing/decreasing trends, i.e., $t = f(n) = an + b$. We use the least-squares method to fit the curve. In order to avoid mis-classifications, such as a constant time relationship as a linear relationship with a near-zero slope, we define a threshold slope $s_{min} = 0.2$ such that $\forall a < s_{min}$, $t = f(n) = b$. For curve evaluation, $d_2 = \sqrt{\textit{residual}}/\textit{average}$ is used, where $\textit{average}$ refers to the average value of the estimated running times.

3. Inverse Proportional: This method captures inverse-proportional trends, i.e., $t = f(n) = k/n$. We observe this trend in the NAS Parallel Benchmark IS, where MPI_Alltoallv dynamically rebalances the per-node workloads even though the collective workload over all nodes is constant. Let t_i be the input times, n_i be the corresponding number of nodes, and $k_i = t_i \times n_i$. We extrapolate the constant k as the average value of k_i. Again, we exclude the outlier k_o, which has the largest absolute value within the deviation. To evaluate this fitting curve, we calculate the standard deviation of k_i and then divide by the average value of k_i, i.e., $d_3 = \textit{std. dev.}/\textit{average}$ is used for comparison.

4. Inverse Proportional + Constant: This method captures the execution time consisting of an inverse proportional phase and a constant phase, i.e., $t = f(n) = k/n + c$. Instead of directly extrapolating t, we utilize the least-squares method to extrapolate $t' = tn = cn+k$ and use $d_4 = \sqrt{residual}/average$ for the curve evaluation. With an extrapolated c and k, t is subsequently calculated as $t = t'/n = k/n + c$.

Having obtained the deviations for each curve-fitting process, we compare the values to determine the curve that best fits. For a closer approximation, we define a threshold value $d_t = 0.05$, such that if and only if $d_{min} + d_t < d_i$ holds for all d_i other than d_{min} will the corresponding candidate curve be selected as the fitting curve. Otherwise, the extrapolation for the current field is postponed until we have processed all the fields in the same histogram. Since every field in the histogram should have the same variation trend, we finalize the pending extrapolation according to the decisions of the remaining fields.

3.3 Experimental Framework

Our extrapolation methodology for communication traces was implemented as the ScalaExtrap tool that generates a synthetic trace for a freely selected number of nodes. The extrapolation is based on traces obtained from application instrumentation with ScalaTrace on a cluster. For both base traces generation and results verification, we use a subset of JUGENE, an IBM Blue Gene/P with 73,728 compute nodes and 294,912 cores, 2 GB memory per node, and the 3D torus and global tree interconnection networks.

The extrapolation process is run on a single workstation and requires only several seconds, irrespective of the target number of nodes for extrapolation. This low overhead is due to the linear time complexity of our algorithm with respect to the total number of MPI function calls in an application. Results from extrapolation are subsequently compared to traces and runtimes of an application at the same scale, where runtimes for extrapolated traces are obtained via ScalaReplay (see Section 2.4).

We conducted extrapolation experiments with the NAS Parallel Benchmark (NPB) suite (version 3.3 for MPI) [6] and Sweep3D [79]. These benchmarks have either a stencil/mesh communication pattern or collective communication, both of which are applicable to our extrapolation algorithm. Among these benchmarks, IS originally exhibited imperfect compression resulting in non-scalable trace sizes due to its dynamic load re-balancing via workload exchange through the MPI_Alltoallv communication collective. In order to utilize our extrapolation techniques, we enhanced ScalaTrace such that minor differences in MPI_Alltoallv parameters caused by load re-balancing are eliminated. The communication pattern of CG is another example of a complicated dynamic pattern. In CG, nodes are logically organized in a 2D array. Each node

communicates with the nodes in the same row with a power-of-two distance and with the node diagonally symmetric to itself, as indicated in Figure 3.8. We support such more complicated patterns by allowing programmers to provide plugin functions for compression and extrapolation on a per-parameter basis. The communication trace extrapolation for CG is facilitated by specifying the communication pattern (i.e., the communication end point described by a function) as a plugin. With this plugin, the extrapolation of timing information does not require any extra information.

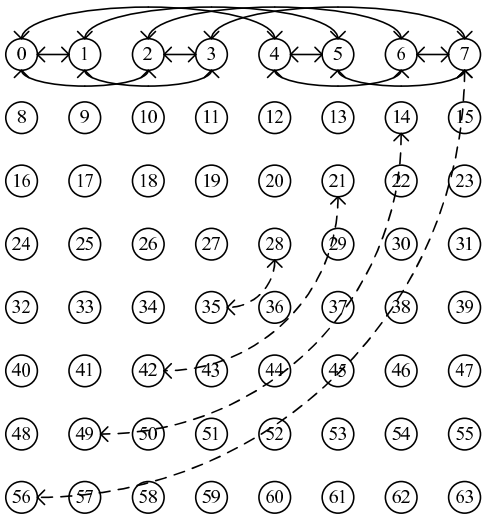


Figure 3.8: CG Communication Topology

We report the experimental results for both strong scaling and weak scaling. For the strong scaling experiments, we mostly used class D and E input sizes for the NPB codes. For the weak scaling experiments, we enhanced the input generator to provide weak scaling inputs for selected NPB codes.

3.4 Experimental Results

Experiments were conducted with respect to two aspects, namely the correctness of communication traces and the accuracy of timing information, both for extrapolations under strong scaling, i.e., when varying the number of nodes. Notice that strong scaling is actually a *harder* problem under extrapolation as it tends to affect communication parameters such as message volume size. In contrast, weak scaling (increasing the number of nodes and problem sizes at the same rate) is easier as it tends to preserve message volumes sizes irrespective of the number

of nodes.

3.4.1 Correctness of Communication Trace Extrapolation

We first evaluated our communication trace extrapolation algorithm with microbenchmarks and the NPB BT, EP, FT, CG, LU, and IS codes. We assessed the ability to retain communication semantics across the extrapolation process for these benchmarks at the target scale. The microbenchmarks perform regular stencil-style/torus-style communication in topological spaces from 1D to 3D. The NPB programs exercise both collective and point-to-point communication patterns. We verified the extrapolation results in multiple ways.

1. The extrapolated trace file T_{e_0} was compared with the trace file obtained from an actual execution at the same scale T_{target} on a per-event basis (Exp1 in Figure 3.9).

2. The extrapolated trace T_{e_0} was replayed such that aggregate statistical metrics about communication events could be compared to those of a corresponding original application run at the same problem size and node size (Exp2 in Figure 3.9).

3. After extrapolation, traces T_{e_1}, T_{e_2}, ..., T_{e_i} were collected in a sequence of replays to obtain a fixed point in the trace representation (Exp3 in Figure 3.9).

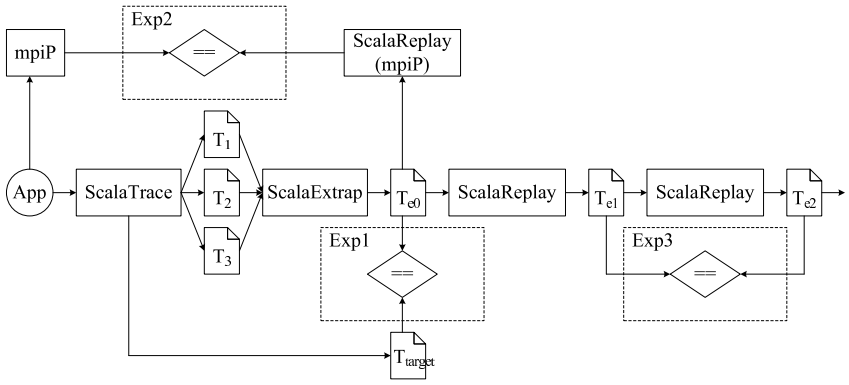


Figure 3.9: Correctness of Trace Extrapolation and Replay

First, the per-event analysis of trace files showed that extrapolated MPI parameters and communication groups perfectly matched those of the application trace for all benchmarks except one (Exp1 in Figure 3.9). In BT, the message volume of non-blocking point-to-point sends and receives *approximates* an inverse-proportional relationship with respect to the number of

nodes. However, it diverges slightly from an inverse-proportional approximation for extrapolating the message volume due to integer division (discarding the remainder) inherent to the source code. This inaccuracy is later amplified in the extrapolation process and results in message volumes that are about 13% smaller than the actual ones at a given scaling factor in the worst case. As imprecisions remain localized to certain point-to-point messages, this effect is shown to be contained in that resulting timings are deemed accurate within the considered tolerance range for extrapolation experiments (see timing results below). Such imprecisions have no side-effect on semantic correctness (causal order) of trace events whatsoever. Overall, the results of static trace analysis show that our synthetically generated extrapolation trace is equivalent to the trace obtained from actual execution of the same application at the same scaling level.

Second, we replayed the extrapolated trace T_{e_0} to assess if the MPI communication events are fully captured (see Exp2 in Figure 3.9). For this experiment, ScalaReplay is linked with mpiP [75], which yields frequency information of each MPI call distinguished by call site (using dynamic stackwalks). During replay, all MPI function calls recorded in the synthetically generated extrapolation trace were executed with the same number of nodes and their original payload size. For comparison, we instrumented the original application with mpiP and executed it at extrapolated sizes (problem and node sizes). We compared the *Aggregate Sent Message Size* reported by mpiP between the original application and the replayed extrapolated trace. Results show that the total send volumes of these experiments are identical, except for MPI_Isend in BT as discussed above. We also compared the total number of MPI calls recorded in the mpiP output files. The results allowed us to verify that the number of communication events in the actual and extrapolated traces match, i.e., the correctness of communication trace extrapolation is preserved.

Third, we evaluated the correctness of ScalaReplay by replaying the generated trace file in sequence until a fixed point is reached (see Exp3 in Figure 3.9). The fixed point approach is a well established mathematical proof method that establishes conversion, in this case of the trace data. In this experiment, instead of instrumenting ScalaReplay with mpiP, we interposed MPI calls through ScalaTrace again. As ScalaReplay issues MPI function calls, ScalaTrace captures these communication events and generates a trace file for it, just as would be done for any other ordinary MPI application. We start by replaying the extrapolated trace file T_{e_0} and obtain a new trace T_{e_1}. This trace differs from T_{e_0} in that call sites of the original program have been replaced by call sites from ScalaReplay. This affects not only stackwalk signatures but also the structure of trace files due to the recursive approach of replaying trace files in place over their internal (PRSD) structure without decompressing it. We then replay trace T_{e_1} to obtain another trace T_{e_2} and so on for T_{e_i}. We then compare pairs of trace files $T_{e_i}, T_{e_{i+1}}$. If two such traces match, a fixed point has been reached. In these experiments, we verified that

pairs of trace files, baring syntactical differences, are semantically equivalent to each other. In other words, ScalaReplay neither adds nor drops any communication events during replay, i.e., by obtaining a fixed point it was shown that all MPI communication calls are preserved during replay.

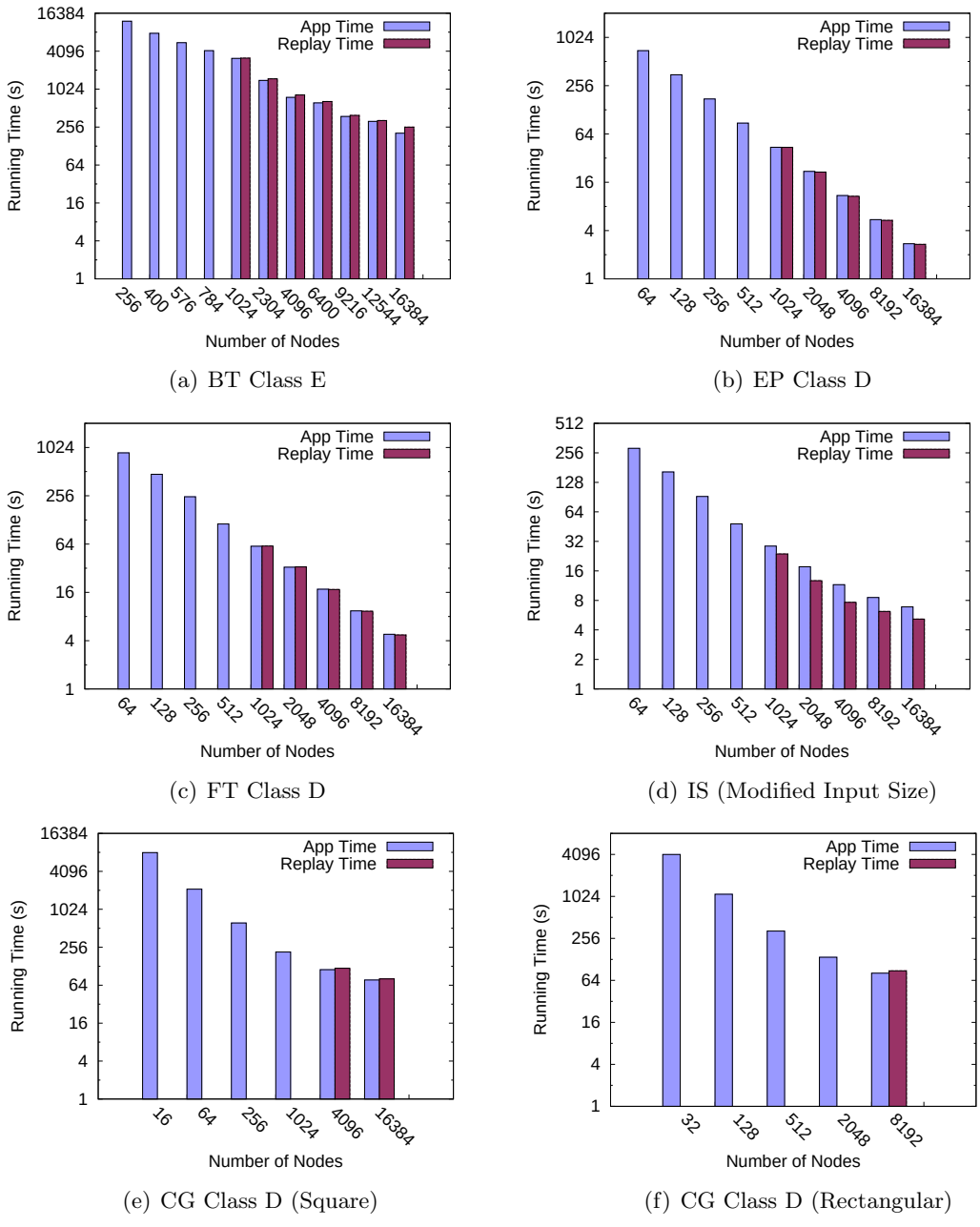
3.4.2 Accuracy of Extrapolated Timings: Timed Replay

We further analyzed the timing information of the extrapolated traces. We report the accuracy of the extrapolated timings for both strong scaling and weak scaling.

Strong Scaling: For this set experiments, we used the NPB BT, EP, FT, CG, and IS codes with a total number of nodes of up to 16,384. For CG, EP, and FT, we used class D input sizes. For BT, class E was used so that a sufficient workload is guaranteed at 16,384 nodes. For IS, we modified the input size to adapt it for 16,384 nodes (the original NPB3.3-MPI provides only class D problem size and supports a maximum of 1024 nodes). These problem sizes and node sizes were decided based on the memory constraints (for some benchmarks, memory constraints compel us to generate the base traces already at large scales, which in turn leaves fewer target sizes for evaluation) and the availability of computational resources to assess the effects and limitations of our timing extrapolation approach.

In this set of experiments, we first generated 4 trace files for each benchmark as the extrapolation basis. From these base traces, an extrapolated trace was constructed next using ScalaExtrap, including extrapolated delta time histograms. We then assess the timing accuracy by replaying the extrapolated traces. During replay, ScalaReplay parses the timing histograms of the computation periods in the trace files. It simulates computation by sleeping to delay the next communication event by the proper amount of time. In this context, the effect of load imbalance is preserved by ScalaTrace. The timing histogram records not only *minimum, maximum, average and standard deviation* values, but also the *frequency* for each timing bin, and these statistics are also extrapolated by ScalaExtrap. During replay, the sleeping time is generated according to these statistics and the unbalanced timing behavior is thus reproduced. Communication is simply replayed with the same extrapolated end points and payload sizes but a random message payload. We do not impose any delays on communication as published results indicate better accuracy with just delays for computation only [56], which we also confirmed. In this experiment, ScalaReplay is linked to neither ScalaTrace nor mpiP to avoid additional overhead caused by the instrumentation layer of these tools. Hence, the output of ScalaReplay in this experiment is the total time to replay a trace. For each extrapolated trace, we run the corresponding application at the same problem size and record its overall execution time for comparison.

Figure 3.10 depicts the extrapolation accuracy of BT, EP, FT, IS, and CG, respectively, for



(a) BT Class E

(b) EP Class D

(c) FT Class D

(d) IS (Modified Input Size)

(e) CG Class D (Square)

(f) CG Class D (Rectangular)

Figure 3.10: Replay Time Accuracy for Strong Scaling Benchmarks

a varying number of nodes. We show the extrapolation results of CG in separate figures because they have different communication topologies and thus a different extrapolation basis. As shown in Figure 3.10, the timing extrapolation accuracy is generally higher than 90%, sometimes even

higher than 98%, where accuracy is defined as

$$Accuracy = (1 - \frac{|Replay\ Time\ -\ App\ Time|}{App\ Time}) \times 100\%.$$

For BT, we observed slightly lower accuracy when the total number of nodes approaches 16,384. At such sizes the computational workload becomes so small that the influence of non-deterministic factors, such as system overheads or performance fluctuation of MPI collectives caused by different process arrival patterns [27], become dominant. Compared to the other benchmarks, IS shows a constantly lower accuracy (66%-83%). Two reasons may explain this phenomenon: (a) Although IS dynamically rebalances the workload across all nodes, the execution time of the application's sorting algorithm on each process still takes a different amount of time. Hence, collective MPI calls take unpredictable time to synchronize as the arrival times of processes at collectives varies significantly due to load imbalance. Since the degree of imbalance is determined by randomly determined delta times from histograms, it is difficult to predict/extrapolate this behavior. (b) Source code analysis shows that the most computationally intensive code section in IS consists of two phases, namely (i) an inverse-proportional phase (runtime is inverse-proportional to the number of nodes), and (ii) a relatively short constant phase (runtime does not change significantly with node sizes). When the node size is small, the inverse-proportional phase almost solely determines the computation time. As a result, our algorithm fails to uncover a small constant factor that contributes to timing for larger node sizes. ScalaExtrap instead treats it as a pure inverse-proportional timing trend. Without the short constant factor in the timing curve, the extrapolated runtime drops slightly faster than the real runtime leading to a constantly shorter replay time. However, since we are able to capture the dominating inverse-proportional timing trend, we still obtained an acceptable timing prediction accuracy.

In large, minor inaccuracies during replay stem from imprecise curve fitting for the extrapolation of computation times. For the simulation of communication duration, ScalaReplay depends only on the communication parameters such as end points and payload sizes, which are shown to be correctly extrapolated in Section 3.4.1. Overall, the extrapolated timing information precisely reflects the runtime of the original application at the target problem size and node size.

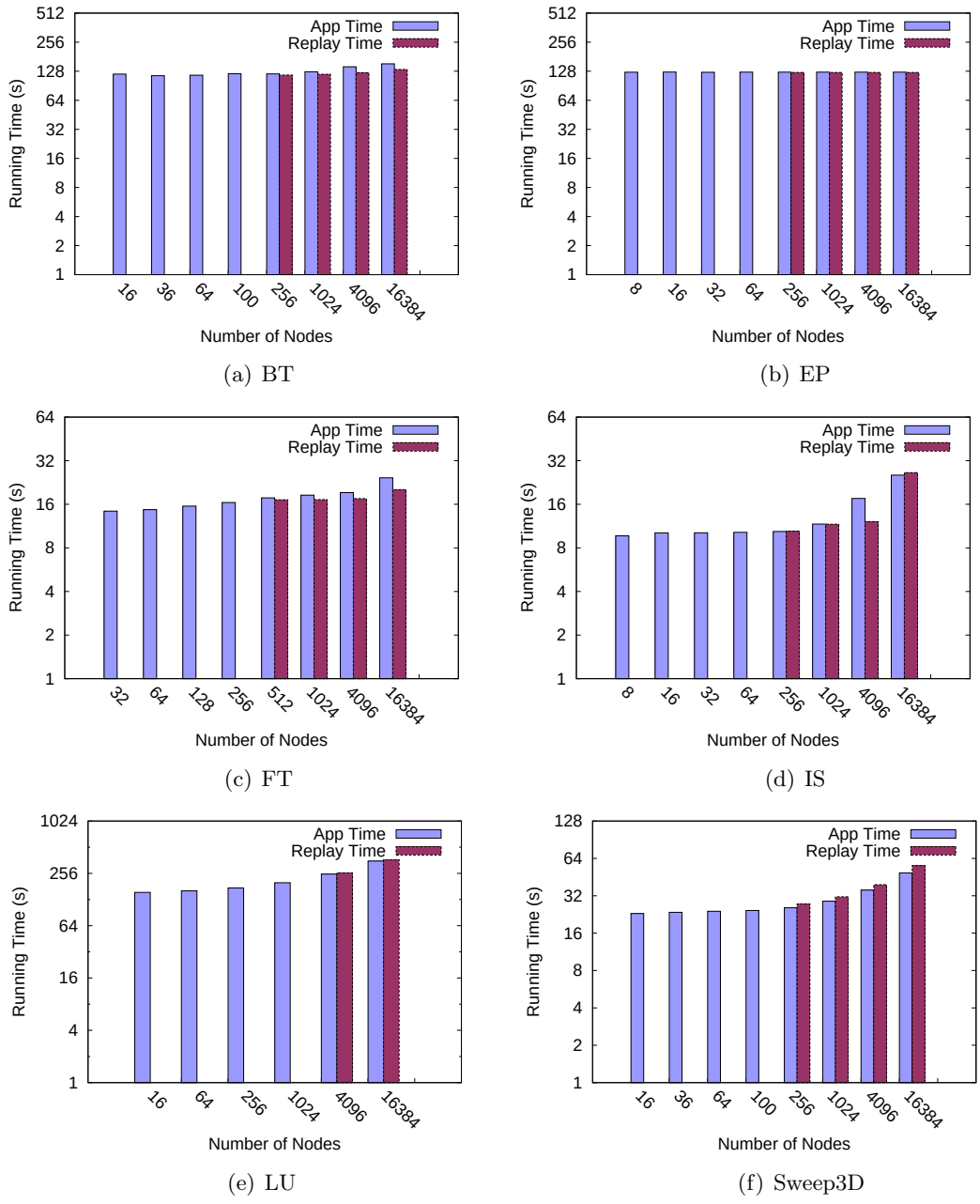
Weak Scaling: Weak scaling refers to varying the problem size and the number of processes at the same rate so that the problem size per node stays consistent during scaling [36]. Among the three factors we have to extrapolate, namely communication topologies, message sizes, and computation times, strong scaling and weak scaling generally do not affect the communication topology in different ways, i.e., the communication patterns often evolve similarly for both strong and weak scaling. Therefore, the communication topology detection and ex-

trapolation algorithms still apply to weak scaling codes. For the other two factors, compared to strong scaling codes, weak scaling codes may exhibit different runtime behavior. For example, due to a constant computational workload per node, the computation times often (but not always) follow a constant trend for weak scaling. In terms of the message sizes, the overall message volume exchanged among all the participating nodes—typically with MPI_Alltoall or MPI_Alltoallv—often increases linearly (or remain constant) under weak scaling when varying the total number of processes. Nonetheless, the curve fitting approach is still applicable, though different/additional curve fitting algorithms may have to be supplied in practice.

We verified our extrapolation approach with weak scaling codes. We conducted these experiments with the NPB BT, EP, FT, IS and LU codes, and the Sweep3D neutron-transport kernel [79]. (For other NPB codes, such as CG, weak scaling inputs could be easily be constructed.) Unlike Sweep3D, the NPB codes are originally designed as strong scaling benchmarks. Hence, we manually changed the input to provide weak scaling workloads.

In the first experiment, we verified the correctness of the generated traces with respect to the extrapolated communication topologies and the values of the communication parameters. We applied similar tests, i.e., static trace comparison and mpiP results comparison, for the synthetically generated traces under weak scaling. The results show that the extrapolated traces are able to correctly preserve the communication semantics, and hence demonstrate the applicability of our topology extrapolation algorithm for applications under weak scaling.

In the second experiment, we evaluated the accuracy of the extrapolated timings for weak scaling problems. We used BT, EP, FT, IS, LU, and Sweep3D benchmarks on runs with up to 16,384 MPI processes. We first generated four traces for each benchmark as the extrapolation basis. Because these base traces were obtained from a series of weak scaling executions, the timing information preserved in the traces also reflects the weak scaling trend. We then performed the extrapolation with ScalaExtrap and replayed the extrapolated traces to obtain and evaluate the accuracy of the total runtimes against the original runs. In these experiments, we observed that the duration of most of the computational phases remains consistent during scaling. This is because for a given weak scaling input, the per-node problem size is fixed irrespective of the node size. We observed this trend for all benchmarks in this experiment, including the simpler ones such as EP and FT as well as the more complicated ones such as Sweep3D and LU. This observation is consistent with our empirical knowledge about the nature of weak scaling codes. In BT, we also observed a more complicated timing trend for some of the computational phases caused by the 3D layout of the problem. Nevertheless, our curve-fitting approach remains still applicable. Figure 3.11 depicts the extrapolation accuracy of the benchmarks. Quantitatively, the mean absolute percentage accuracy (as defined in Section 3.4.2) across all benchmarks and test cases is 92.87%. Among all the tests, IS with 4,096 nodes has a relatively low timing accuracy. IS uses bucket sort to distribute the elements to different nodes



(a) BT (b) EP

(c) FT (d) IS

(e) LU (f) Sweep3D

Figure 3.11: Replay Time Accuracy for Weak Scaling Benchmarks

and then sorts the local elements within each node. When run with 4,096 nodes with the weak scaling input we provided, IS failed to balance the workload across nodes. Since the base traces for extrapolation were obtained from runs with good load balance, we cannot extrapolate the

inconsistent computational duration caused by the unbalanced workload. We thus simply could not reproduce the unbalanced timing behavior during replay. Overall, our timing extrapolation approach is able to accurately predict the runtimes without experiencing additional challenges beyond those observed under strong scaling. With such a high timing accuracy and the proven communication semantics, the extrapolated weak scaling traces can be used for trace-based system simulation or other performance analysis experiments.

3.4.3 Lossy Extrapolation

Compared to all other benchmarks discussed so far, MG demonstrates the most complicated communication pattern. Overall, MG has a 3D communication topology. All nodes participate in a regular 7-point 3D torus-style communication. However, as a minor communication pattern, nodes at particular positions also communicate to nodes one hop (distance of 2) away in the 3D space. This communication is rank dependent even for those participating nodes, which makes the per-node traces highly divergent and the size of the final trace non-scalable. What is even more challenging is that the non-SPMD communication pattern does not exhibit any regular spacial property in forming the communication groups, i.e., there is little information to be derived from smaller traces about how a node will behave at larger scale. As a result, we cannot do the extrapolation by utilizing a general approach.

To extrapolate MG, we applied the lossy extrapolation approach. Specifically, we applied the snippet-based trace event filtering at the intra-node level to eliminate the minor rank dependent communication events. With events dropped, the generated traces consist of only the dominating regular 3D torus communication and the collectives. When compared across different node sizes, the traces are structurally identical, which indicates a perfect compression. As the result, the extrapolation of MG is largely simplified and becomes equivalent to the extrapolation of the 3D torus micro-benchmark.

We evaluated the lossy extrapolation of MG with both strong scaling and weak scaling configurations. For weak scaling experiments, we changed the input generator to provide weak scaling inputs. We first evaluated the ability of extrapolated traces to preserve communication events. We replayed each extrapolated trace with mpiP instrumentation under ScalaReplay. We subsequently compared the generated mpiP results (profile counts) with those obtained by executing the mpiP-instrumented original benchmark over the same number of MPI processes. The experimental results show that for all the extrapolated sizes the number of dropped events is constantly less than 5% of the total number of MPI events, which indicates the communication workload is well preserved. Figure 3.12 compares the replay times of the extrapolated traces to the runtimes of the original benchmark. Due to inaccurate curve fitting, the replay times of the extrapolated traces are slightly longer than the original runtimes at large scales. Nonetheless,

even after increasing the replay times by 5% (given less than 5% of the events were filtered), the extrapolated traces were still able to reflect the total runtime of the original benchmark.

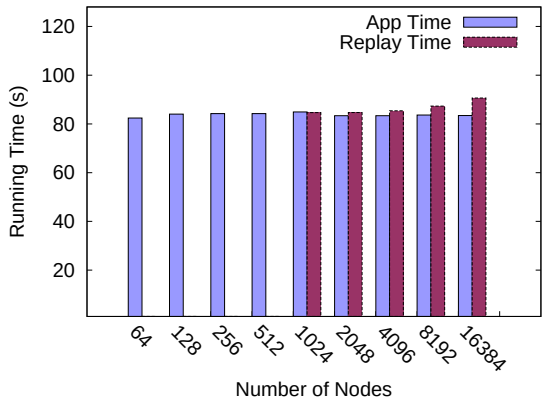


Figure 3.12: Timing Accuracy of Lossy Extrapolation of Weak Scaling MG

When configured for strong scaling, MG falls into the category of applications that do not follow the dominating communication pattern at smaller scales, which presents a challenge. On the positive side, we are able to preserve and extrapolate all the collectives and the regular 7-point 3D torus communication pattern, the latter of which is measured to be the dominating pattern at smaller scales. However, as the problem size per node decreases, the number of the 3D torus communication events also decreases. Meanwhile, the number of the MPI events corresponding to the minor communication pattern stays constant and starts to domineer at larger scales. For example, with the lossy tracing approach, 96.98% of the MPI_Send operations were preserved in the 64-node trace of MG while only 77.56% were preserved in the trace of 1024 nodes. As a result, the extrapolated trace loses its ability to preserve most of the communication workload, even though one of the two overlapping communication patterns is fully captured. Since our trace-based extrapolation is a black box approach relying only on information in the input traces instead of knowledge about the source code of the application, the extrapolation of the strong scaling for MG is beyond the scope of ScalaExtrap's current capabilities.

3.5 Application of the Extrapolated Trace

As shown with the experiments, the extrapolated traces are able to correctly predict the communication semantics and timing behavior of the original applications at large scales. This capability enables the extrapolated traces to be used for performance analysis. For example,

they can be fed into discrete-event-based simulators such as Dimemas for performance simulation, or be used as the input for visualization tools such as Vampir Next Generation. In this section, we describe two case studies to demonstrate the potential use of the extrapolated traces. The experiments in this section were performed on ARC, a cluster with 1728 cores on 108 compute nodes, 32 GB memory per node and both Infiniband and Ethernet Interconnect.

3.5.1 Extrapolated Trace for Code Generation

In the first case study, we use the extrapolated traces to generate parallel benchmarks. The main idea of benchmark generation is to automatically generate 1) MPI events with the same parameter values and temporal ordering, and 2) sleeps that mimic the computation stages in the original application, according to the information in the input trace [84, 20]. The drawback of the trace-based benchmark generation approach is that the generated code can only be launched with the same number of MPI processes with which the trace was collected. Nonetheless, this is complimented by our extrapolation algorithm — with the extrapolated trace, the benchmark generator is able to generate code for arbitrary number of MPI processes that is valid for an application.

To demonstrate this idea, we generated C+MPI code from the extrapolated Sweep3D traces. In this experiment, we first executed the ScalaTrace-instrumented Sweep3D at the node sizes of 16, 36, 64, and 100, to collect the base traces for extrapolation. We then extrapolated a series of traces for the node sizes of 144, 196, 256, 324, and 400, which were subsequently fed into the benchmark generator to generate C+MPI parallel benchmarks. Figure 3.13 compares the total execution times of the generated benchmarks to that of the original applications at the same node size. Quantitatively, the mean absolute percentage accuracy (as defined in Section 3.4.2) across all test cases is 93.76%. With such a high timing accuracy, the generated benchmarks can be used for performance experiments or as the substitute of the classified applications.

3.5.2 Extrapolated Trace for Performance Experiments

In the second case study, we use the extrapolated trace to analyze the impact of computational speedup on the overall performance. Computational speedup can be achieved in multiple ways. For example, application developers can optimize performance by overlapping communication and computation. They may also manually or automatically parallelize their code to take advantage of the compute power of the multi-core/many-core architectures. A current trend in high-performance computing is to supplement general-purpose CPUs with more special-purpose computational accelerators (e.g., GPUs). Unfortunately, it is nontrivial both to predict how fast a parallel application will run once accelerated and to port a parallel application to an accelerated architecture.

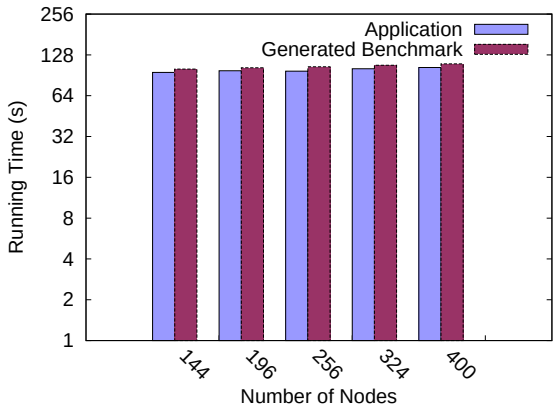


Figure 3.13: Timing Accuracy for the Extrapolated Benchmarks

To readily assess the effect of computational speedup before implementation, application developers can perform a quick what-if analysis by modifying the application trace. A unique feature of ScalaTrace is that the collected trace is concise and structure-preserving. This enables people to manually modify the trace for performance analysis. Since the extrapolated trace is equivalent to the trace obtained from a real execution, it shares the same merit.

As an example, we assess the impact of computational speedup on the overall performance of Sweep3D with the extrapolated Sweep3D trace. In this experiment, we used the Ethernet interconnect on ARC, which is slower than Infiniband. We first extrapolated the Sweep3D trace of 400 MPI processes from smaller traces. We then changed the computation times and message volumes recorded in the trace to simulate different expected improvements due to acceleration and different communication workload, respectively. Total execution time was measured to reflect the change of the overall performance. The results are shown in Figure 3.14, where the x-axis indicates the length of the generated computational phases in percentage of the original delta time, and different curves correspond to different message volumes. As shown in Figure 3.14, an interesting observation is that reducing the computation times, i.e., "accelerating" the computational stages, does not always lead to a better overall performance. For example, when the message volume is increased to 1, 2, and 4 time(s) of the actual values, the best overall performance is achieved when the sleep time is set to be 10%, 20%, and 40% of the original, i.e., a 10x, 5x, and 2.5x computational speedup, respectively. Particularly, when the message volume is set to 8 times of the actual value, the optimal sleep time is 180% of the original computation times, which indicates that, instead of trying to accelerate, application developers actually should slow down the computation stages to achieve the best overall performance. To understand this puzzling behavior, note that Sweep3D is a stencil

code consisting almost exclusively of synchronous point-to-point communication operations. Shortening the sleep times increases the contention in the network, which in turn offsets the time saved with computational speedup. Moreover, the larger the message volume, the heavier the network congestion would be. As a result, the optimal sleep time increases with the message volume.

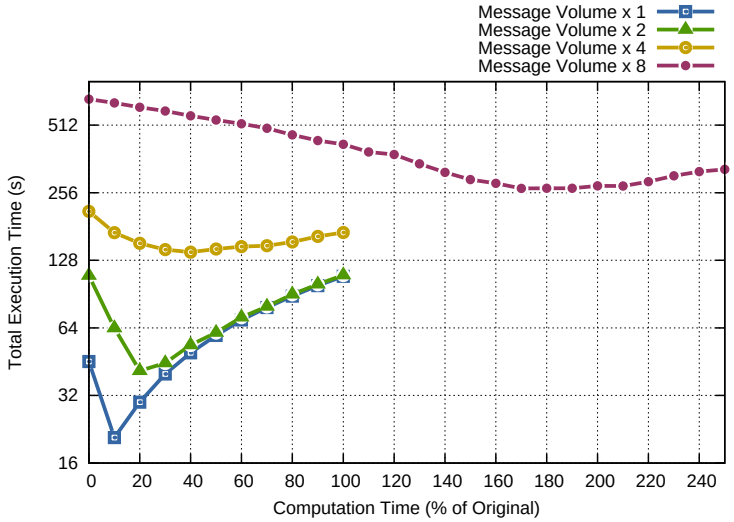


Figure 3.14: The Impact of Computational Speedup on the Overall Performance

We should note that the experimental result presented in Figure 3.14 is both application-specific and platform-specific. Yet, with the application trace, what-if analysis on application performance can easily be done without ever needing to implement a parallel algorithm or port the original application. Besides, with the trace-based extrapolation, performance analysis can be done even without the large-scale input data set or the necessary hardware, *e.g.,* large amount of GPUs.

3.6 Related Work

ScalaTrace is an MPI trace-gathering framework. It generates near constant-size communication traces for a parallel application with regular SPMD behavior regardless of the number of nodes while preserving structural information and temporal ordering [56, 64] (see Chapter 2). Our extrapolation work builds on the trace representation of ScalaTrace.

Xu et al. construct coordinated performance skeletons to estimate application execution

time in new hardware environments [86, 87]. They detect dominant communication topologies by comparing an application communication matrix against a predefined set (library) of reference patterns. In this work, complicated communication patterns, such as the NAS benchmark CG, are handled by manually provided specifications of the new patterns. Moreover, the graph spectrum analysis and graph isomorphism tests utilized in this work lack scalability in terms of time complexity and thus limit the applicability of this work at large sizes. Most significantly, their work does not capture all communication events.

Zhai et al. collect MPI communication traces and extract application communication patterns through program slicing [90]. This work utilizes a set of source code analysis techniques to build a program slice that only contains the variables and code sections related to MPI events, and then executes the program slice to acquire communication traces. While removing the computation in the original application enables a fast and cheap trace collection, it also causes the loss of temporal information that is essential for characterizing the application runtime behavior. In addition, the lack of trace compression limits its feasibility for large-scale application tracing. Based on the FACT framework, Zhai et al. employ a deterministic replay technique to predict the sequential computation time of one process in a parallel application on a target platform [89]. The main idea is to use the information recorded in the trace to simulate the execution result of MPI calls when there is actually only one MPI process, and utilize the deterministic data replay to simulate the runtime of the computation phases on the target platform. While this approach manages to predict the computation time, it fails to capture the communication related effects. In addition, this work focuses on cross-platform performance prediction but cannot predict the application performance on a cluster that is larger than the available host platform.

Dimemas is a discrete-event-based network performance simulator that uses Paraver traces as input [60]. It simulates the application behavior on the target platform with specified processor counts and network latency. However, Dimemas simulations are infeasible for peta-/exascale simulations due to a lack of hardware resources to generate the input trace and the sheer size of traditional application traces. Our work, in contrast, focuses on the trace extrapolation for larger platforms that applications have not yet been ported to or even future platforms (exascale). The extrapolated traces can then be either replayed with ScalaReplay (former case) or used as the input trace for simulators (Dimemas/SST) in the latter case for performance prediction. Ronsse et al. presented $Rolt^{MT}$, an extension of ROLT (Reconstruction of Lamport Timestamps) for message passing systems [66]. With the $Rolt^{MT}$ approach, send events related with a promiscuous receive event are attached with Lamport timestamps incremented with a value larger than one. Since only timestamps for these events are recorded in the trace, the trace size and the program perturbation caused by tracing are minimized. The recorded Lamport timestamps are then used in replay with additional synchronizations to allow a deterministic

replay of programs with non-deterministic receives. In contrast, ScalaTrace and ScalaReplay keep the wildcard values such as MPI_ANY_SOURCE in tracing and extrapolation so that the nondeterministic features of the original programs are preserved, although eliminating nondeterminism can also be easily done with our systems.

Preissl et al. extract communication patterns, i.e., the recurring communication event sets, from MPI traces [62]. They first search for repeating occurrences of identical events in the trace of each individual process and then iteratively grow them into global patterns. The output of this algorithm can be used to identify potential bottlenecks in parallel applications. Preissl et al. further utilize the detected communication patterns to automate source code transformations such as automatic introduction of MPI collectives [63]. Our method, in contrast, focuses on the spatial aspect of communication events, i.e., the identification and extrapolation of communication topology.

Eckert and Nutt [22, 23] extrapolate traces of parallel shared-memory applications. They take as input the traces collected on an existing architecture and extrapolate them to a target platform with different architectural parameters, without re-executing the original application. This work analyzes the causal event stream. It focuses on the correctness of the extrapolated trace given the existence of program-level non-determinism, e.g., the interleaving of events or modifications in the actual set of events caused by moving the trace across different architectures. In contrast, our work is based on deterministic application execution. We also preserve the causal ordering of communication events but our focus is on the communication behavior at arbitrary problem sizes.

Performance modeling has traditionally taken the approach of algorithmic analysis, often combined with tedious source code inspection and hardware modeling for floating-point operations per second, memory hierarchy analysis from caches over buses to main memory and interconnect topology, latency and bandwidth considerations. In particular, Kerbyson et al. present a predictive performance and scalability model of a large-scale multidimensional hydrodynamics code [39]. This model takes application, system, and mapping parameters as input to match the application with a target system. It utilizes a multitude of formulae to characterize and predict the performance of a scientific application. Snavely et al. model and predict application performance by 1) characterizing a system with machine profiles, namely single processor performance and network latency and bandwidth, 2) collecting the operations in an application to generate application signatures, and 3) mapping signatures to profiles to characterize performance [73, 7]. Gruber et al. describes PatternTool, an interactive tool for creating scalable hierarchical graphs to define the communication patterns and control flows of a parallel algorithm graphically [35]. The created performance model can be used as the input to PAPS (Performance Prediction of Parallel Systems), a simulator that can be parameterized for different computer systems, for performance simulation. İpek et al. follow a completely

different approach by utilizing artificial neural networks (ANNs) to predict the performance when application configuration varies [38]. This approach employs repeated sampling of a small number of points in the design space that are statistically determined through SimPoint [72]. Only these points are then simulated and results are utilized to teach the ANNs, which are subsequently utilized to predict the performance for other design points. In contrast, our work explores the potential of extrapolating the application runtime according to its evolving trend across increasing problem sizes. Since this method requires neither measurement of performance metrics nor intense computation, it provides a simple and highly efficient approach to study the effect of scaling across a large numbers of compute nodes. In contrast to all of the above approaches, our ScalaExtrap does not just simulate communication behavior at scale but allows such behavior to be observed in practice through replaying on a target platform with large numbers of nodes, even if the corresponding application itself has not been ported yet.

3.7 Summary

Scalability is one of the main challenges of scientific applications in HPC. Advanced communication tracing techniques achieve lossless trace collection, preserve event ordering and encapsulate time in a scalable fashion. However, estimating the impact of scaling on communication efficiency is still non-trivial due to execution time variations and exposure to hardware and software artifacts.

This work contributes a set of algorithms and analysis techniques to extrapolate communication traces and execution times of an application at large scale with information gathered from smaller executions. The extrapolation of communication traces depends on an analytical method to characterize the communication topology of an application. Based on the observation that problem scaling increases/decreases communication parameters and topology at a certain rate, we utilize a set of linear equations to capture the relation between communication traces for changing number of nodes and extrapolate communication traces accordingly. For the extrapolation of communication traces, the detection of communication topology is vital but non-trivial. We currently focus on stencil/mesh topology with nodes arranged in a row-major fashion. While a large amount of parallel applications fall into this category, we observed more complex communication topologies that are hard to detect with a generic approach, which thus limits the applicability of this work. In future work, we plan to support user plugins so that the extrapolation of complicated and unique communication patterns can be facilitated by user-supplied information.

For the extrapolation of timing information, we utilize curve fitting approaches to model trends in delta times over traces with varying number of nodes. Statistical methods are further employed to mitigate timing fluctuations under scaling. Currently, we capture four categories

of the most commonly seen timing trend. However, the prediction of more complicated timing trends, *e.g.,* the detection of the combination of multiple types of timing trends, may require more sophisticated algorithms.

Experiments were conducted using an implementation through our ScalaExtrap tool and with the NAS Parallel Benchmark suite and Sweep3D. We utilized up to 16,384 nodes of a 73,728-node IBM Blue Gene/P. Experimental results show that our algorithm is capable of extrapolating stencil/mesh and collective communication patterns for both strong scaling and weak scaling configurations. Extrapolation of timing information is further shown to provide good accuracy.

We believe that extrapolation of communication traces for parallel applications at arbitrary scale is without precedence. Without porting applications, communication events can be replayed and analyzed in a timed manner at scale. This has the potential to enable otherwise infeasible system simulation at the exascale level.

Chapter 4

Automatic Generation of Parallel Benchmarks from Applications

4.1 Introduction

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

The research question we propose to answer in this chapter is the following: Is it possible to combine the best features of complete applications, computational kernels, and microbench-

marks into a single performance-evaluation methodology? That is, can one evaluate how fast a target HPC system will run a given application without having to migrate it and all of its dependencies to that system, without ignoring the subtleties of how different pieces of an application perform in context, without forsaking the ability to experiment with alternative application structures, and without restricting access to the tools needed to perform the evaluation?

Our approach is based on the insight that application performance is largely a function of the sorts of primitive operations that microbenchmarks measure and that if these operations can be juxtaposed as they appear in an application, the performance ought to be nearly identical. We therefore propose generating *application-specific performance benchmarks*. In fact, by "generating", we imply a fully automatic approach in which a parallel application can be treated as a black box and mechanically converted into an easy-to-build, easy-to-modify, and easy-to-run program with the same performance as the original but absent the original's data structures, numerical methods, and other algorithms.

We take as input an MPI-based [34] message-passing application. To convert this into a benchmark, we utilize the approach illustrated in Figure 4.1. We begin by tracing the application's communication pattern (including intervening computation time) using ScalaTrace [55]. The resulting trace is fed into the benchmark generator that is the focus of this chapter. The benchmark generator outputs a benchmark written in coNCePTuaL, a domain-specific language for specifying communication patterns [58]. The coNCePTuaL code can then be compiled into ordinary C+MPI code for execution on a target machine.

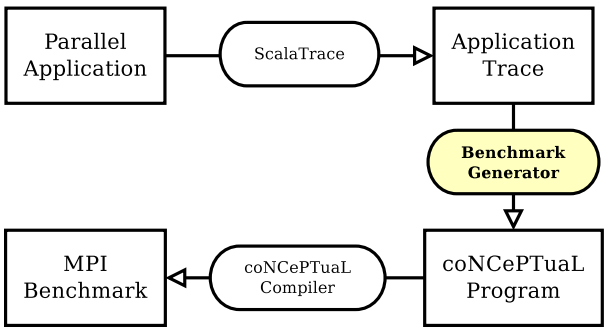


Figure 4.1: Benchmark Generation System

We utilize ScalaTrace 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 [60], Tau [70], Open|SpeedShop [68], Vampir [52], and Kojak [80], 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 [58] instead of a general-purpose language such as C or Fortran as the target language for benchmark generation. (coNCePTuaL does, however, compile to C source code.) Because coNCePTuaL is designed specifically for the expression of communication patterns, benchmarks generated in coNCePTuaL are highly readable. coNCePTuaL code includes almost exclusively communication specifications. Mundane benchmarking details such as error checking, memory allocation, timer calibration, statistics calculation, MPI subcommunicator creation, and so forth are all handled implicitly, which reduces code clutter.

We evaluated our benchmark generation approach with the NAS Parallel Benchmark suite [6] and the Sweep3D code [42]. We performed experiments to assess both the correctness and the timing accuracy of the generated parallel benchmarks. Experimental results show that the auto-generated benchmarks preserve the application's semantics, including the communication pattern, the message count and volume, and the temporal ordering of communication events as they appear in the original parallel applications. In addition, the total execution times of the generated codes are very similar to those of the original applications; the mean absolute percentage error across all of our measurements is only 2.9%. Given these experimental results, we conclude that the generated benchmarks are able to reproduce the communication behavior and wall-clock timing characteristics of the source applications.

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

performance repeatability by introducing determinism into benchmarks generated from nondeterministic applications.

We foresee our work benefiting application developers, communication researchers, and HPC system procurers. Application developers can benefit in multiple ways. First, they can quickly gauge what application performance is likely to be on a target machine before exerting the effort to port their applications to that machine. Second, they can use the generated benchmarks for performance debugging, as the benchmarks can separate communication from computation to help isolate observed performance anomalies. Third, application developers can examine the impact of alternative application implementations such as different data decompositions (causing different communication patterns) or the use of computational accelerators (reducing computation time without directly affecting communication time). Communication researchers can benefit by being able to study the impact of novel messaging techniques without incurring the burden of needing to build complex applications with myriad dependencies and without requiring access to codes that are not freely distributable. Finally, people tasked with procuring HPC systems benefit by being able to instruct vendors to deliver specified performance on a given application without having to provide those vendors with the application itself.

4.2 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 [11], Extrae/Paraver [60], and tools based on the Open Trace Format [40] 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 [41] 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 [15]. While this approach could facilitate trace analysis, it is lossy and thus not suitable for benchmark generation.

Xu et al.'s work on constructing coordinated *performance skeletons* to estimate application execution time in new hardware environments [86, 87] exhibits many similarities with our work. However, a key aspect of performance skeletons is that they filter out "local" communication (communication outside the dominant pattern). As a result, the generated code does not fully reproduce the original application, which may cause subtle but important performance characteristics to be overlooked. Because our benchmark generation framework is based on lossless application traces it is able to generate benchmarks with identical communication behavior to the original application. In addition, we generate benchmarks in coNCePTuaL instead of C so that the generated benchmarks are more human-readable and editable.

Program slicing, statically reducing a program to a minimal form that preserves key properties of the original, offers an alternative approach to generating benchmarks from application traces. Ertvelde et al. utilize program slicing to generate benchmarks that preserve an application's performance characteristics while hiding its functional semantics [25]. 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 [69], 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 [90]. 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 [8, 81, 74]. For example, Bell et al. [8] 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 [81]. Sreenivasan et al. generate representative synthetic workload by matching the joint probability density of the real workload with that of the synthetic workload [74].

Automatic code generation approaches can also be utilized to assist in characterizing I/O performance. Logan et al. generate skeletal I/O applications automatically from XML files with ADIOS I/O specifications and manually provided runtime parameters [47, 4]. ScalaIOTrace

collects trace information for MPI-IO and POSIX I/O operations [77]. As an extension to ScalaTrace, ScalaIOTrace retains the compression capabilities and structure-preserving features of ScalaTrace. Thereby, it is feasible to utilize the code generation framework proposed in this work to generate I/O benchmarks from the ScalaIOTrace traces.

Besides benchmark generation and synthesis, our work is also relevant to performance modeling and prediction [17, 38, 39, 7, 73]. For example, in [17], a modeling and analysis framework is designed to automatically estimate the resource demand for a given performance target using program characteristics. In [38], artificial neural networks (ANNs) are utilized to predict the application performance when configuration varies.

4.3 coNCePTuaL

Our benchmark generation approach utilizes the ScalaTrace infrastructure [55] to extract the communication behavior of the target application. Based on the application trace, we generate benchmarks in coNCePTuaL [58], 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 coNCePTuaL that enable our benchmark generation methodology.

coNCePTuaL is a tool designed to facilitate rapid generation of network benchmarks. It 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, Figure 4.3 shows a *complete* coNCePTuaL benchmark program corresponding to the 1D torus communication pseudo MPI code snippet presented in Figure 4.2.

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

Figure 4.2: Pseudo MPI Code for 1D Torus Communication

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

```
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)".
}
```

Figure 4.3: coNCePTuaL Code for the Pseudo MPI Code in Figure 4.2

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 [57].

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.

4.4 Benchmark Generation

4.4.1 Overview

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

Most of the conversion from RSDs and PRSDs to coNCePTuaL code is straightforward. An RSD representing point-to-point communication (blocking or nonblocking) is converted to a coNCePTuaL `SEND` or `RECEIVE` statement; computation time encoded in an RSD is converted to a coNCePTuaL `COMPUTE` statement; and a PRSD is converted to a coNCePTuaL `FOR`

EACH loop. Behavior that differs across loop iterations (message destinations, compute times, etc.) is implemented with a coNCePTuaL IF statement conditioned on a loop variable. There are a few subtleties involved in the mapping from ScalaTrace to coNCePTuaL; Section 4.4.2 discusses these.

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

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

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

It is not possible to know if those two MPI_Reduce() calls are part of the same collective operation without knowing the complete, run-time control flow of the program—on each rank individually—that led to the execution of the code shown above. The challenge is how to merge per-rank collective operations found in a trace into a single collective operation whose participants can be identified *statically*. An example of such an operation expressed in coNCePTuaL is "TASKS xyz SUCH THAT 3 DIVIDES xyz REDUCE A DOUBLEWORD TO TASK 0"; no further information is required to know that tasks 0, 3, 6, 9, ... are the participants in that reduction operation. Section 4.4.3 presents our algorithm for matching collective operations specified separately on each node.

An MPI feature that hinders performance reproducibility is nondeterminism. MPI supports "wildcard receives" (MPI_ANY_SOURCE), which can receive messages from any sender. While this feature *can* lead to correctness issues [78], and we do address this, we are concerned primarily with the different performance that can result from different messages matching a set of wildcard receives. Consider, for example, the following use of the MPI_Recv receive operation:

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

```
else
  ⟨Do some short-running computation.⟩
MPI_Recv(..., MPI_ANY_SOURCE, ..., status);
```

Depending on the sender's MPI rank (`status.MPI_SOURCE`), the preceding code can take either a long time or a short time to run. Because the sender whose message matches the MPI_Recv can vary from run to run, the execution time of the preceding code also varies from run to run. While this behavior may be reasonable for an application, we deem it inappropriate for a benchmark program. As benchmarks are commonly used to evaluate system performance, small changes in a target machine's hardware or system software should not result in arbitrarily large changes in a benchmark's execution time. Section 4.4.4 presents our algorithm for removing performance nondeterminism caused by wildcard receives in the input trace.

4.4.2 Engineering Details

coNCePTuaL is not designed to exactly represent MPI features. In fact, the coNCePTuaL compiler can compile the same source program to C+MPI, C+Unix sockets, or to any other language/communication library combination for which a compiler backend exists. Consequently, coNCePTuaL contains collectives that MPI lacks (e.g., arbitrary many-to-many reductions with non-overlapping source and destination task sets), and MPI contains collectives that coNCePTuaL lacks (e.g., scatters of different-sized messages to different destinations). We therefore had to "impedance match" the benchmark generator's MPI-centric input to coNCePTuaL output. Our approach is to replace each unsupported MPI collective with one or more coNCePTuaL collectives that represent a similar communication pattern (i.e., data fan in or fan out) and data volume. Table 4.1 presents the substitutions we made.

Table 4.1: Mapping of MPI Collectives to coNCePTuaL

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

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

4.4.3 Combining Per-Node Collectives

As discussed in Section 4.4.1, MPI allows multiple statements in the source code to represent a single, common collective operation. Because ScalaTrace differentiates call sites by call-stack signatures, this use of collectives generates distinct RSDs in the trace. To improve benchmark readability, before generating coNCePTuaL code we want to combine these separate RSDs, each representing a subset of the collective's participants, into a single RSD that represents the complete set of participants. Figure 4.4 illustrates the intention, using C+MPI (with the omission of most MPI arguments) instead of RSDs for clarity. Figure 4.4(a) presents the initial communication pattern, in which each of ranks 0 and 1 invoke MPI_Barrier from a different source-code line. Assuming these are found to be the same collective, we want to hoist the MPI_Barrier outside of all conditionals on the rank, as shown in Figure 4.4(b).

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

(a) C+MPI Program

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

(b) Aligned Collectives

Figure 4.4: Combining Collectives Across Separate Source-code Statements

To perform this transformation, recall that our benchmark generator operates on communication traces, not on application source code; it therefore does not literally perform the

source-code transformation shown in Figure 4.4. Rather, it follows the sequence of steps presented in Algorithm 2 to align in time the RSDs of the same collective operation across nodes then combine these RSDs into a single RSD specifying the complete set of nodes to which the collective operation applies.

The main idea, illustrated in Figure 4.5 for RSDs corresponding to the C+MPI code in Figure 4.4, is to stop the trace traversal for a node at each collective in which it participates until all of the other participating nodes have arrived at the same collective. Algorithm 2 guarantees that (1) a collective operation corresponds to only one RSD in the output trace, (2) the ordering of MPI events for each node is preserved in the trace, and (3) the output trace is still in a compressed format. This algorithm tracks the traversal on different nodes by maintaining a *traversal context* for each node. The traversal context stores the current RSD the node is executing, the loop stack the execution is in, and the iteration count for each loop in the stack. Upon startup, the algorithm traverses the trace on behalf of node 0, which is called the current *running node*. For each RSD of non-collective MPI routines that the running node is involved in, the algorithm extracts the current MPI event and appends an RSD to the output queue. (Note that an RSD can contain multiple MPI events across loop iterations and across nodes due to compression.) For collectives, however, the traversal stops for the current running node and switches to the next node in the communicator (indicated by the small arrows in Figure 4.5). When the last node in the communicator arrives at the collective, the algorithm appends the RSD for all the nodes to the output queue and switches the traversal back to the first node that is blocked on the same collective. We treat MPI_Finalize as a collective so that the algorithm cannot finish until the traversal is done for all the nodes. To guarantee that the new trace is scalable in length, we apply ScalaTrace's loop compression algorithm [55] to the output RSD queue each time a new RSD is appended to the queue.

The complexity of Algorithm 2 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 2 traverses every event in the trace exactly once for each node. In Algorithm 2, the *for* loop in line 2 initializes the iterator to the head RSD for each node. During execution, the *while* loop in line 12 always moves the iterator forward by exactly one event in each iteration. In 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.

Algorithm 2 Algorithm to Align Collectives

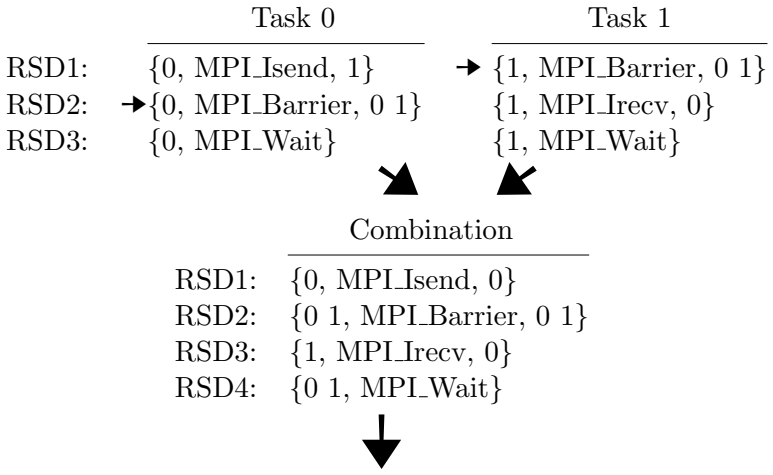
Precondition: T_{in}: input trace, N: total number of nodes
Postcondition: T_{out}: the trace for coNCePTuaL code generation

```
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
```

4.4.4 Eliminating Nondeterminism

MPI supports the use of a wildcard value, MPI_ANY_SOURCE, for the *source* parameter of point-to-point receives. For example, in the NAS Parallel Benchmarks's implementation of LU decomposition [6], nodes use MPI_ANY_SOURCE to receive messages in arbitrary order from their neighbors in a 2-D stencil. The problem with the use of MPI_ANY_SOURCE from a benchmarking perspective is that it has the potential to introduce performance artifacts, as discussed in Section 4.4.1. That is, each run of LU may stress the communication subsystem




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

Figure 4.5: Operation of Algorithm 2

slightly differently based on the order in which messages happen to be received. To promote reproducibility of empirical measurements, our benchmark generator removes nondeterminism by replacing wildcard receives with arbitrary but valid non-wildcard receives.

As in Section 4.4.3's algorithm for combining collectives, Algorithm 3 utilizes a trace-traversal approach to resolve wildcard receives. Let e_{ijk} represent an MPI event k that is issued by node i and has node j as its peer. We maintain two lists for each node x: a list L_1 of the to-be-matched MPI events $e_{xj_1 1}, e_{xj_2 2}, e_{xj_3 3}, \ldots$ that were issued by node x itself and a list L_2 of the MPI events $e_{i_1 x k_1}, e_{i_2 x k_2}, e_{i_3 x k_3}, \ldots$ specifying the events issued by other nodes that should be matched by node x. Upon startup, this algorithm traverses the input trace on behalf of an arbitrary node x. During the traversal, it adds the unmatched point-to-point operations to list L_1 of node x and to list L_2 of each peer node. The traversal for node x stops when the execution is blocked on (1) a blocking send/receive, (2) a collective, or (3) a wait operation. It then switches the traversal to a node y whose execution will potentially unblock the execution on node x. In order to be selected as the target node to which the traversal switches (i.e., node y), a node must be (1) the destination/source of the blocking send/receive on node x, (2) a node in the same communicator with node x, or (3) the destination/source of one of the nonblocking sends/receives that node x is waiting on, respectively. During the traversal for node y, we look up every MPI operation we arrived at in list L_2 of node y to

detect matches. When a match is found, we delete the event from both lists. If possible, we unblock the execution on node x so that the traversal for it can proceed later on. If the receiver of a match uses MPI_ANY_SOURCE, this value is replaced with the rank of the (first) matching sender so that the wildcard source is resolved. Collectives are handled in a similar way as Algorithm 2 by blocking the traversal until every participating node arrives. We treat MPI_Finalize as a collective that all the nodes participate in, so that every node is traversed before the algorithm finishes. Because Algorithm 3 is again based on traversing a trace and each MPI event is evaluated exactly once in the *while* loop at line 12, the complexity is $O(e)$, where $e = \Sigma_{i=0}^{n-1} e_i$ is the total number of MPI events on all the nodes. Similarly, the use of wildcard receives is checked at a cost of $O(r)$ before applying this algorithm, where r is the number of RSDs in the trace and, typically, $r \ll e$.

A ScalaTrace trace is obtained from an instance of a correct execution of the original parallel application. However, ScalaTrace does not represent this or any other specific execution because it does not replace the wildcard *source* value with the rank of the actual sender. Consequently, if the original application potentially deadlocks, Algorithm 3 suffers from the same risk. As an example, the code fragment in Figure 4.6(a) deadlocks if the wildcard receive is matched with node 0 but completes if matched with node 2. One possible execution generates the trace shown in Figure 4.6(b), which causes Algorithm 3 to hang because node 0 is blocked on MPI_Finalize and node 1 is blocked on MPI_Recv(0) during trace traversal.

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

(a) MPI Program with Potential Deadlock

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

(b) The Trace of (a) that Makes Algorithm 3 Hang

Figure 4.6: Potential Deadlock

To avoid potential hangs in Algorithm 3 caused by nondeterminism in the original application, our benchmark generator extends Algorithm 3 to detect deadlock conditions during trace traversal. Notice that these deadlocks stem from incorrect MPI semantics of the application, not from our tracing or code-generation framework. We decided to identify such incorrect MPI programs and report the existence of deadlocks to the user. To this end, we track another two

Algorithm 3 Algorithm to Resolve Wildcard Receive (Without Deadlock Detection)

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

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

10: function MATCH(n, T)
11:     iter ← C[n].RSD
12:     while iter do
13:         if node n is not in iter.rank_list then
14:             iter ← iter.next
15:         else
16:             if iter.op is point-to-point operation then
17:                 if match with an event e_ink in L2 then
18:                     L2.delete(e_ink)
19:                     node_i.L1.delete(e_ink)
20:                     if node_i.L1 is empty then
21:                         C[i].RSD ← C[i].RSD.next                ▷ unblock
22:                     end if
23:                     if iter.peer is MPI_ANY_SOURCE then
24:                         iter.peer = i                           ▷ resolve the wildcard
25:                     end if
26:                     iter ← iter.next
27:                     continue
28:                 else
29:                     p ← iter.peer
30:                     L1.add(e_np(k_n++))
31:                     node_p.L2.add(e_npk_n)
32:                     if iter.op is blocking operation then
33:                         C[n].RSD ← iter
34:                         MATCH(p, T)
35:                     else
36:                         iter ← iter.next
37:                         continue
38:                     end if
39:                 end if
40:             end if
41:             if iter.op is collective or MPI_Finalize then
42:                 ...                                             ▷ refer to Algorithm 2
43:             end if
44:             if iter.op is wait operation then
45:                 if L1 is not empty then
46:                     MATCH(L1.first.getPeer(), T)
47:                 else
48:                     iter ← iter.next
49:                     continue
50:                 end if
51:             end if
52:         end if
53:     end while
54:     return T
55: end function
```

types of events during traversal: (1) T_{ijk}, the transfer of traversal from node i to node j due to MPI event e_k, and (2) U, the unblocking event. We append these events to a global list, L_3, in the order they were encountered during the traversal. If the traversal is switched to node n while node n is blocked on an MPI event e_k, the deadlock detection algorithm traverses L_3 to determine if any unblocking event U has taken place since the last time the traversal left node n due to the same MPI event e_k. If there is no unblocking event found, a potential cyclic dependency is detected. If e_k is a blocking send/receive, then a deadlock potential has been uncovered and the algorithm terminates. If e_k is a wait operation blocked on multiple requests, the traversal is proxied to the peer of another nonblocking communication on which node n is waiting. If the peers of all the pending nonblocking sends/receives have been traversed and the cyclic dependency still exists, a deadlock potential has been detected and the algorithm terminates. This algorithm implements a *sufficient* deadlock detection scheme. As a result, Algorithm 3 is guaranteed to be deadlock-free. However, unlike the DAMPI algorithm [78], Algorithm 3 does not establish or test the permutations of all execution interleavings and thus does not present a *necessary* condition for a deadlock as the approach is based on a single trace sequence of events. It may therefore fail to identify deadlocks in the original application that were not uncovered by the specific trace execution.

4.4.5 The Generation of Scalable Benchmarks

An inherent drawback of the trace-based benchmark generation approach is that the generated code is not scalable in a parametric sense; it can be executed only with the exact number of MPI tasks with which the trace was collected. To alleviate this shortcoming and allow an arbitrary number of MPI tasks for invocation, we incorporated our benchmark generator with ScalaExtrap, our prior work that extrapolates a large communication trace (a trace with large number of MPI tasks) from a series of smaller traces (see Chapter 3).

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

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

In addition, the fitting curves for the computation times are also stored in the auxiliary trace. The auxiliary trace is structurally similar to a normal ScalaTrace trace so that the formulae

in the auxiliary trace can be easily mapped to parameters once a trace size is selected for an actual run.

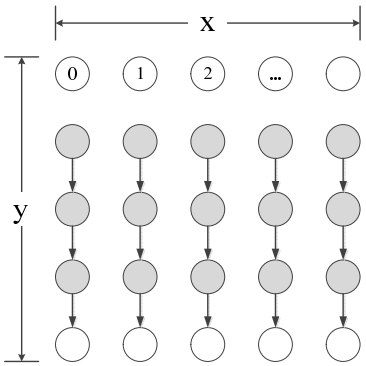


Figure 4.7: Communication Pattern of a 2D Stencil Code

To auto-generate the extrapolation benchmark of an application, both the extrapolated trace of a specific size and the auxiliary trace are read into the code generator. During generation, whenever there is a formula in the auxiliary trace for a certain parameter in the normal trace, the code generator generates a coNCePTuaL communication statement that uses the scalable representation in the auxiliary trace. As an example, Figure 4.7 shows a 2D communication pattern in which the gray nodes send messages to the nodes below them. ScalaExtrap identifies that the relative distance d of the receiving nodes is $d = x$, i.e., a linear correlation with the size of the x dimension. Hence, a scalable coNCePTuaL send statement using a formula instead of a particular value as the destination will be generated as

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

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

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

indicating the participating MPI tasks. During code generation, this ranklist is used to generate the range expression that defines the source tasks of this MPI event. coNCePTuaL implements the ranklist with a list comprehension using n variables to recursively define the iteration count and stride for each of the n dimensions in a ranklist. By utilizing list comprehensions, we can

conveniently generate the scalable ranklist representation above with the coNCePTuaL range expression shown below:

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

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

By introducing ScalaExtrap's parameter-extrapolation functionality into our benchmark-generation framework, we are able to generate coNCePTuaL communication benchmarks that can be executed with an arbitrary valid number of MPI tasks while performance remains accurate at different processor counts. Note that generating scalable benchmarks is nontrivial due to the challenges in trace-based extrapolation such as detecting communication topology, matching trace events across scales, and inferring scale-dependent communication events. ScalaExtrap currently focuses on stencil/mesh topology with nodes arranged in a row-major fashion, which represents the structure of many parallel applications.

4.4.6 Sources of Performance Inaccuracy

As indicated, there are a number of ways in which our benchmark generator trades off performance fidelity for an improved ability to reason about the generated code and its performance: computation times are summarized across ranks instead of being specified individually; some complex MPI collectives are implemented in terms of more basic coNCePTuaL collectives (Section 4.4.2); and nondeterministic receive ordering is replaced with an arbitrary deterministic ordering (Section 4.4.4). In Section 5.3 we examine the impact of these design decisions in the context of a suite of test programs.

4.5 Evaluation

4.5.1 Experimental Framework

To evaluate our benchmark-generation methodology, we generated coNCePTuaL codes for the NAS Parallel Benchmarks (NPB) suite (version 3.3 for MPI, comprising BT, CG, EP, FT,

IS, LU, MG, and SP) using the class C input size [6] and for the Sweep3D neutron-transport kernel [79]. These benchmarks all have either a mesh-neighbor communication pattern or rely heavily on collective communication. Some of them (e.g., Sweep3D) require collective alignment (Section 4.4.3), and some (e.g., LU) require the resolution of wildcard receives (Section 4.4.4). Hence, the key features of our code-generation framework are fully tested in this set of experiments. We believe results from the NPB codes and Sweep3D, combined with previous ScalaTrace experiments [56, 83], are sufficient to demonstrate the correctness of our approach, and we do not foresee any algorithmic or technical problems with generating code for larger applications. Moreover, these benchmarks are sufficient to demonstrate our ability to retain an application's performance characteristics. In particular, several kernels in the NPB suite, including CG, FT, and MG, are known to be memory-bound [67], which stresses our generated benchmarks' ability to mimic computation with spin loops of the same duration.

Benchmark generation is based on traces obtained on (a) Ocracoke, an IBM Blue Gene/L [3] with 2,048 compute nodes and 1 GB of DRAM per node and (b) ARC, a cluster with 1728 cores on 108 compute nodes, 32 GB memory per node, and an Ethernet interconnect. Due to limited access to these systems our experiments generally run on only a subset of the available nodes. Benchmark generation is performed on a standalone workstation.

4.5.2 Communication Correctness

Our first set of experiments verifies the correctness of the generated benchmarks, i.e., the benchmark generator's ability to retain the original applications' communication pattern. For these experiments, we acquired traces of our test suite on Blue Gene/L, generated coNCePTuaL benchmarks, and executed these benchmarks also on Blue Gene/L. To verify the correctness of the generated benchmarks, we linked both them and the original applications with mpiP [75], 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 [83] 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.

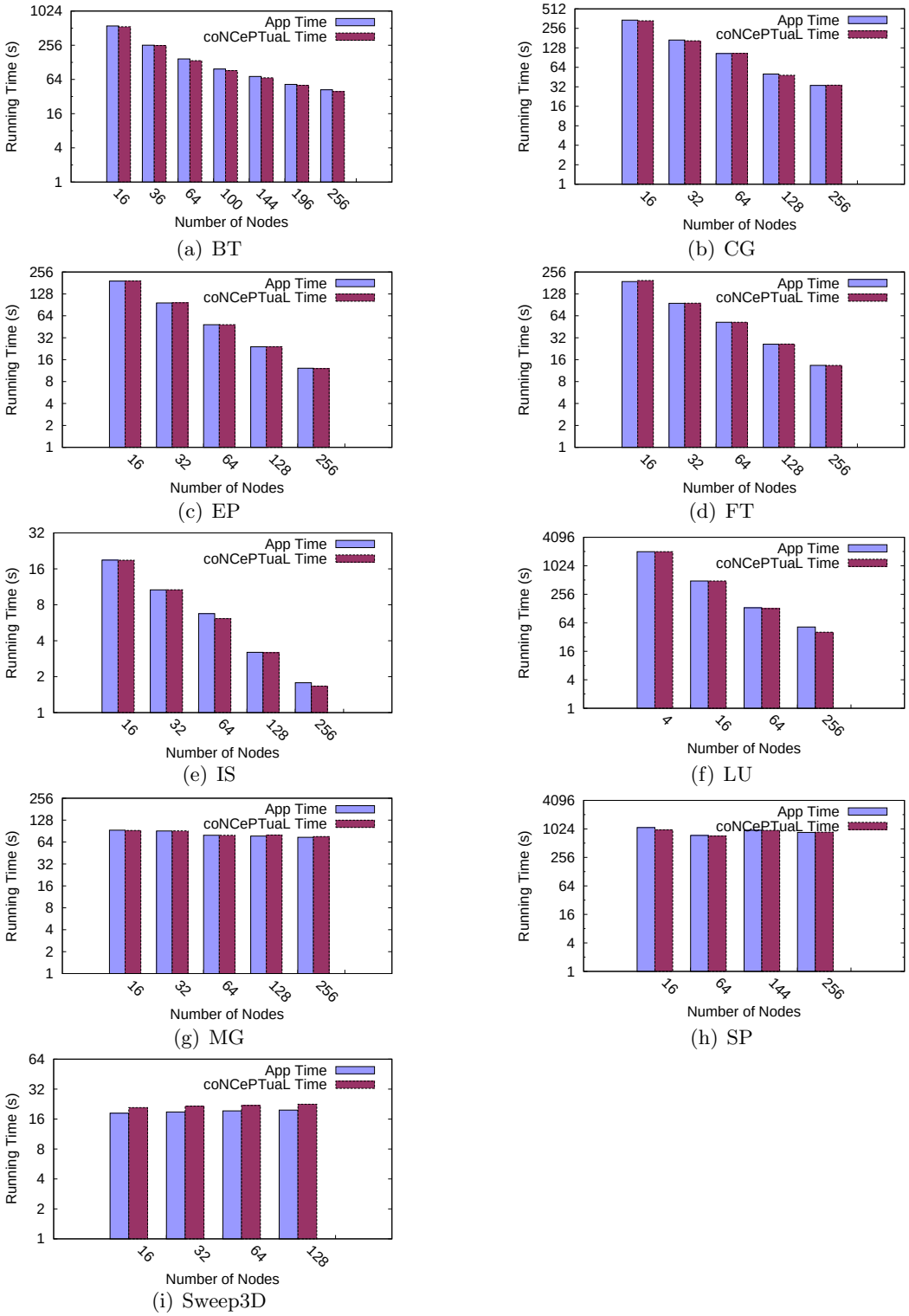
4.5.3 Accuracy of Generated Timings

Having determined that benchmarks produced using our benchmark generator faithfully represent the communication performed by the original applications, we then assessed the generated benchmarks' ability to retain the original applications' performance. To measure the total execution time of the original applications, we extended the PMPI profiling wrappers of MPI_Init and MPI_Finalize to obtain timestamps). The corresponding coNCePTuaL timing calls were also added to the generated benchmarks. We ran both the original application and the generated benchmark on the Blue Gene/L system and compared the total elapsed times. Figure 4.8 shows that the timing accuracy is qualitatively extremely good. Quantitatively, the mean absolute percentage error (i.e., $100\% \times |(T_{\text{coNCePTuaL}} - T_{\text{app}})/T_{\text{app}}|$) across all of Figure 4.8 is only 2.9%, and only two data points exhibit worse than 10% deviation: LU at 256 nodes observes a deviation of 22% (40 s for the benchmark versus 52 s for the original application), and SP at 16 nodes observes a deviation of 10% (980 s for the benchmark versus 1092 s for the original application).

4.5.4 Correctness and Timing Accuracy of Generated Scalable Benchmarks

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

In the first experiment, we generated an extrapolation benchmark under coNCePTuaL for BT of the Class D input size. We used traces of 16, 64, 144, and 256 tasks as the input of

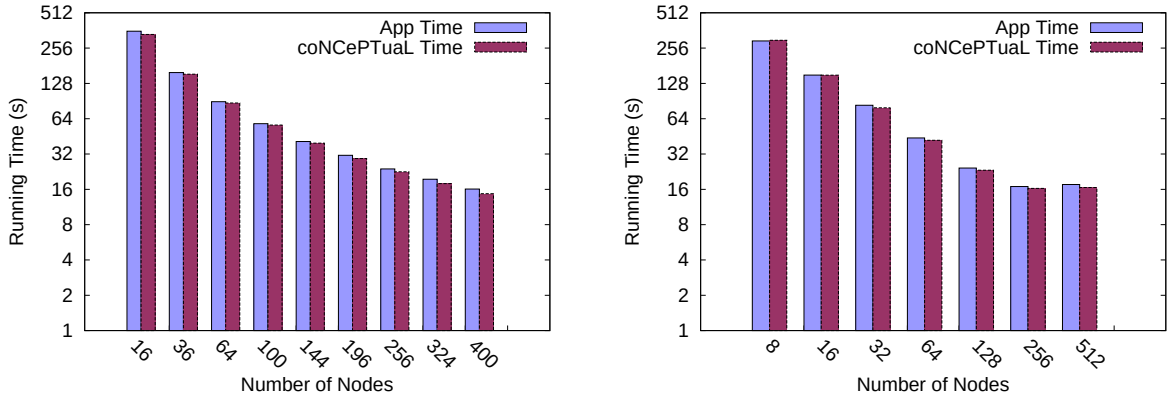


(a) BT (b) CG (c) EP (d) FT (e) IS (f) LU (g) MG (h) SP (i) Sweep3D

Figure 4.8: Time Accuracy for Generated Benchmarks

ScalaExtrap. With an extrapolated trace and the auxiliary trace generated during extrapolation, we generated an extrapolation benchmark under coNCePTuaL. We then executed the extrapolation benchmark at different scales from 16 to 400 MPI tasks and evaluated their communication correctness and timing accuracy. To demonstrate the communication correctness, we collected the message density matrices for both the original application and the extrapolation benchmark. The heat maps of the original application and the extrapolation benchmark are identical, showing that the generated extrapolation benchmark is able to preserve the communication pattern of the original application under scaling.

In the second experiment, we evaluated the timing accuracy of the generated extrapolation benchmarks under coNCePTuaL with different numbers of MPI tasks. For BT, we used the same scalable code that we used in the experiment described above. For FT, we used the Class C input size instead of Class D so that we can collect the input traces for ScalaExtrap starting with a minimum of 8 MPI tasks. With traces collected for 8, 16, 32, and 64 MPI processes, we generated an extrapolation benchmark for FT under coNCePTuaL. We then executed both the original application and the extrapolation benchmarks at different scales and compared their total execution times. Figure 4.9 shows the experimental results. As demonstrated, the auto-generated extrapolation codes under coNCePTuaL have total execution times that closely resemble those of the original applications at each tested scale. Quantitatively, across all the tested node sizes, the mean absolute percentage errors for BT and FT are only 5.12% and 3.42%, respectively.



(a) BT, Class D

(b) FT, Class C

Figure 4.9: Timing Accuracy of the Scalable coNCePTuaL Benchmarks

4.5.5 Applications of the Benchmark Generator

The experimental results presented in Sections 4.5.2 and 4.5.3 indicate that the performance of the generated benchmarks can be trusted. We now present examples of what-if analysis and cross-platform performance prediction that are made practical by automatic benchmark generation.

Impact of Computational Speedup

A current trend in high-performance computing is to supplement general-purpose CPUs with more special-purpose computational accelerators (e.g., GPUs).[1] However, by Amdahl's Law [5], accelerating only an application's computational phases does not always lead to proportional overall speedup. Unfortunately, it is nontrivial both to predict how fast a parallel application will run once accelerated and to port a parallel application to an accelerated architecture. Application developers may also optimize performance by overlapping communication and computation. This too takes time to implement and leads to a reduction in execution time that can be difficult to predict.

Because the coNCePTuaL benchmarks produced by our generator are easy to modify, we can use our framework to estimate how fast an application can be expected to run once accelerated or once communication and computation fully overlap. We generated a benchmark from the NPB BT code on 64 cores using the class C input. We then modified the coNCePTuaL code to vary the time spent in all computation phases from 100% down to 0% of their original time to simulate different expected improvements due to acceleration. We ran the resulting benchmark variations on the ARC cluster (cf. Section 4.5.1) and plotted the results in Figure 4.10.

Reading Figure 4.10 from right to left, the data points ranging from 100% down to 30% of the original application's compute time are essentially what one might expect: a steady but sublinear decrease in total execution time. That is, a fabricated 3.3x speedup of computation leads to only a 21% reduction in total execution time for BT. However, as computation time continues to decrease, rather than reach a plateau, the total execution time *increases*. At the 0% computation mark, which represents infinitely fast processors on a modern Ethernet network, there is essentially *no* speedup over the unmodified BT execution time.

To understand this puzzling behavior, note that BT is a stencil code consisting almost exclusively of asynchronous point-to-point communication operations, with only a few collectives at the beginning and end of the execution. Reducing the time between subsequent communication operations alters the dynamics of the messaging layer and leads to the observed increase in

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

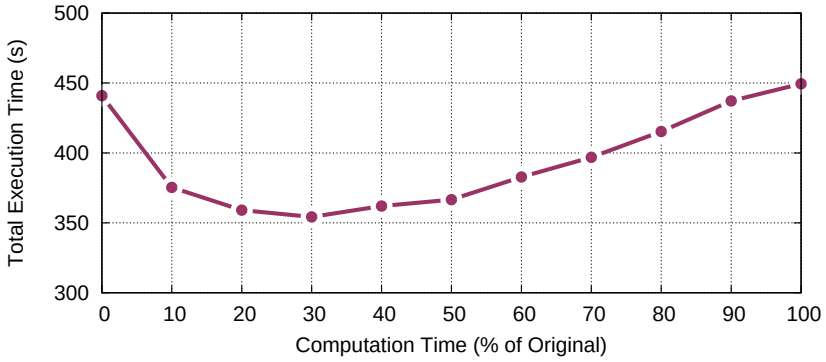


Figure 4.10: Communication Performance of BT

performance. For example, if messages begin arriving faster than they can be processed, they will start being directed to the MPI implementation's unexpected-receive queue, which incurs a performance cost in the form of an extra memory copy to transfer unexpected messages to the target buffer. Once all available space for storing incoming messages on a given node is exhausted, the MPI implementation's flow-control mechanism must stall any senders and later pay a cost in network latency to resume them. It is the nonlinear effects such as those that make it important to quantify potential performance improvements using a framework such as ours before investing the effort to accelerate an application.

We should note that the experimental result presented in Figure 4.10 is both application-specific and platform-specific. Yet, with our benchmark-generation approach, the experiment can easily be repeated on different platforms without ever needing to port the original application. In addition, our BT experiment can easily be refined to utilize different speedup factors for different computational phases. We foresee this type of performance experimentation, enabled by our benchmark generator, becoming increasingly important as HPC hardware increases in complexity and requires expanded efforts to port large applications (for potentially small performance gains).

Impact of Communication Performance

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

For this experiment, we used the ARC cluster with an Infiniband QDR interconnect. We

generated the coNCePTuaL benchmark for the NPB BT code with 16 MPI tasks and class A input size, which allows us to modify the generated benchmark by manually customizing message sizes from 0.1x to 256x of their original sizes. We then executed the modified codes with a node count from 1 to 16, which increases the available bandwidth per MPI task as node counts become higher. Figure 4.11 shows the overall execution time of the modified benchmark for each configuration. Each curve represents the results obtained for a different message size.

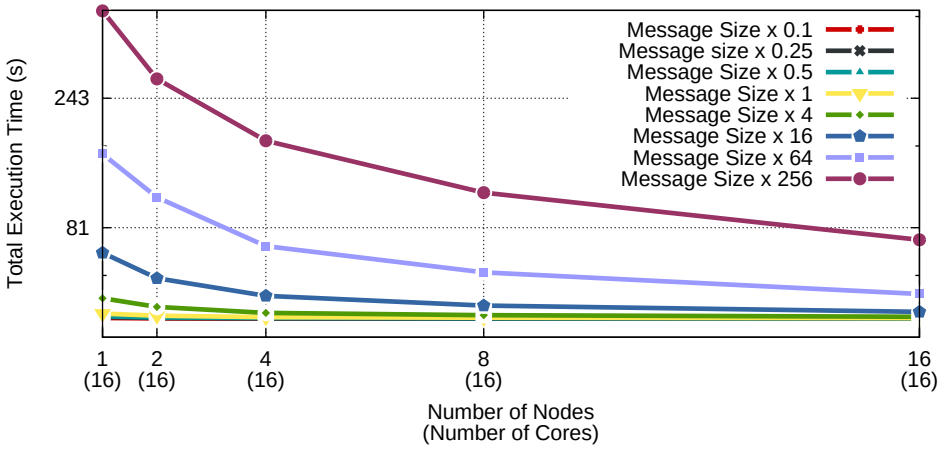


Figure 4.11: Impact of Communication Performance on BT

We observe that, in the figure, the curves for 0.1x, 0.25x, 0.5x, 1x, and 4x message sizes overlap with each other. This indicates, for this particular application and this particular system, that neither increasing the available bandwidth nor decreasing the message size may improve overall performance. Therefore, optimizations in the computation design are required to accelerate this application. Yet, when the message size is 16 times the original size, the communication starts to saturate the network bandwidth. From there on, notable speedup can be observed if more nodes and hence higher overall bandwidth are allocated for this application.

Impact of Collective Implementation

With easy-to-modify coNCePTuaL benchmarks, application developers not only can modify the parameter values for what-if analysis, they can also quickly modify the communication implementation for performance comparison. This is made practical by the conciseness of coNCePTuaL programs. For example, no buffer allocation or explicit request handle manage-

ment is necessary. In addition, because computation is removed and replaced with idle spinning in the generated benchmarks, the generated benchmarks tend to be much shorter and simpler than the original application. For example, the coNCePTuaL version of FT has only 22 lines of code (Figure 4.12) while the original FT code has 2,131 lines (approximately 35 pages) of code. As Figure 4.8(d) shows, there is no qualitative difference between the performance of the original code and the generated benchmark, even though the latter is only 1% of the length of the former. With the generated communication skeleton, application developers can readily assess a communication design without having to modify the entire parallel algorithm and keep track of the changes in multiple source files.

```
01:    ALL TASKS SYNCHRONIZE THEN
02:    TASK 0 RESETS ITS COUNTERS THEN
03:    ALL TASKS COMPUTE FOR 52 MICROSECONDS
04:    TASK 0 MULTICASTS A 12-BYTE MESSAGE TO ALL OTHER TASKS
05:    ALL TASKS COMPUTE FOR 0 MICROSECONDS
06:    TASK 0 MULTICASTS A 4-BYTE MESSAGE TO ALL OTHER TASKS
07:    ALL TASKS COMPUTE FOR 295410 MICROSECONDS
08:    ALL TASKS MULTICASTS A 32768-BYTE MESSAGE TO ALL OTHER TASKS
09:    ALL TASKS COMPUTE FOR 133044 MICROSECONDS
10:    ALL TASKS SYNCHRONIZE
11:    ALL TASKS COMPUTE FOR 317381 MICROSECONDS
12:    ALL TASKS MULTICASTS A 32768-BYTE MESSAGE TO ALL OTHER TASKS
13:    FOR EACH i1 IN {1, ..., 20} {
14:        IF i1 <> 1 THEN ALL TASKS COMPUTE FOR 167332 MICROSECONDS THEN
15:        IF i1 = 1 THEN ALL TASKS COMPUTE FOR 312861 MICROSECONDS THEN
16:        ALL TASKS MULTICASTS A 32768-BYTE MESSAGE TO ALL OTHER TASKS THEN
17:        ALL TASKS COMPUTE FOR 254450 MICROSECONDS THEN
18:        ALL TASKS REDUCE 4 INTEGERS TO TASK 0
19:    }
20:    ALL TASKS COMPUTE FOR 24 MICROSECONDS
21:    ALL TASKS SYNCHRONIZE THEN
22:    TASK 0 LOGS ELAPSED_USECS/1E6 AS "Seconds"
```

Figure 4.12: Complete coNCePTuaL Code for NPB FT (Class C) of 256 MPI Tasks

In this experiment, we evaluate different communication implementations for the NPB FT code. FT solves a three-dimensional partial differential equation using the fast Fourier transform (FFT). It uses MPI_Alltoall to exchange data among all the participating MPI tasks in each timestep. Alternatively, point-to-point communication routines can also be used to imple-

ment the same communication pattern. We compared the performance of these two different implementations by modifying the generated FT benchmark. To migrate from the MPI_Alltoall implementation to the point-to-point implementation with MPI_Isend, we do not need to understand the FFT algorithm to find out which buffer should be changed or to implement the all-to-all style point-to-point communication for the communicators representing the user-defined processor layout. Instead, only one line of the coNCePTuaL code needs to be changed from

```
ALL TASKS MULTICAST A xxx-BYTE MESSAGE TO ALL OTHER TASKS
```

to

```
ALL TASKS ASYNCHRONOUSLY SEND A xxx-BYTE MESSAGE TO ALL OTHER TASKS
ALL TASKS AWAIT COMPLETION
```

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

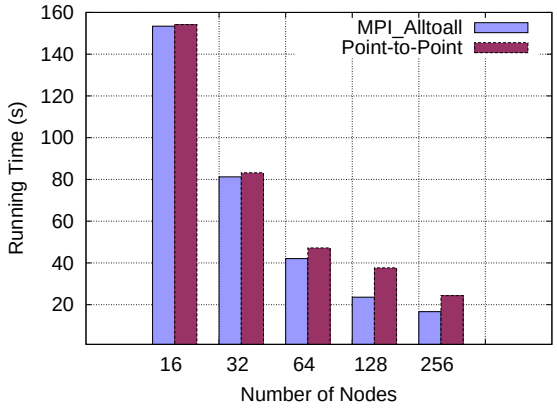


Figure 4.13: Performance of All-to-all Implementations for FT

We compared the performance for different implementations by executing the coNCePTuaL FT codes on ARC. Figure 4.13 plots the overall execution times for different implementation strategies with different number of nodes. Not surprisingly, the MPI_Alltoall implementation outperforms the point-to-point implementation. Because the point-to-point implementation blindly exchanges data between each pair of nodes without any communication pattern optimization, it suffers from scalability constraints: the overall runtime of the point-to-point version with 256 MPI tasks is 46.2% longer than the collective version even though its performance is 0.5% slower for 16 nodes.

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

Cross-platform Performance Prediction

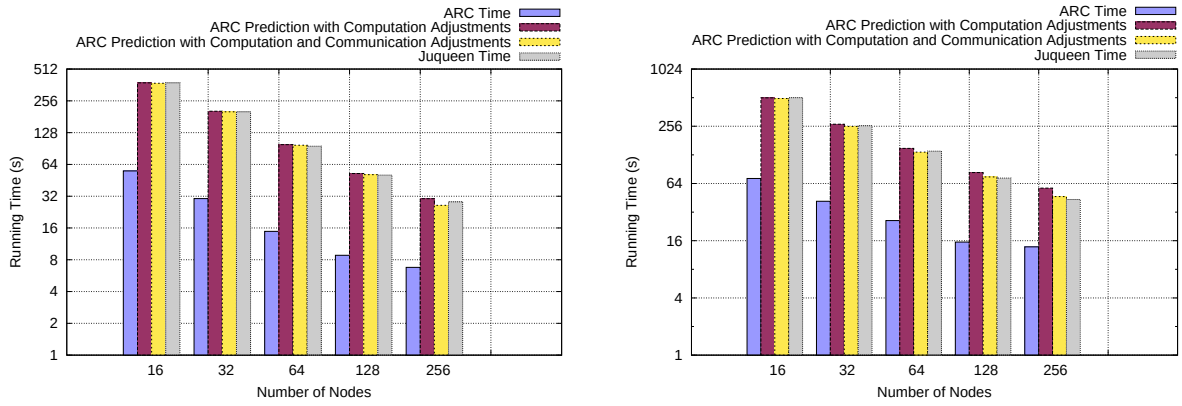
The benchmark generation approach presented in this work may also benefit system designers and procurers by providing a means for fast cross-platform performance prediction for existing or even future systems. In this section, we present our prediction results for two different architectures, ARC and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

The runtime of a parallel application consists of the sequential computation time in each process, the communication time between processes, and their convolution. To predict the runtime of an application on a future platform, we generate coNCePTuaL benchmarks of different node sizes that reflect the computational speedup with modified sleep times, execute the generated benchmarks on an existing system, and adjust the communication times to estimate the total runtime. In this set of experiments, we obtained the computational speedup by executing the application on only one node on each of the existing (ARC) and the future (Juqueen) platforms. The communication speedup was calculated by performing a ping-pong test on both systems. To adjust the total runtime, mpiP was utilized to measure the time spent in MPI communication events. Hence, with the known communication time T_{comm}, communication speedup s, and the computation time T_{comp} of the simulation run, the predicted runtime

T can be calculated with equation $T = T_{comm} \times s + T_{comp}$. In case that the interconnect on the future platform is not yet available, estimation or analytical modeling results can be used instead.

Figure 4.14 shows the cross-platform prediction results. According to the single-node computational speedup tests, ARC is 5.9 times faster than Juqueen for MG and 6.2 times faster for CG. Accordingly, the coNCePTuaL benchmarks were generated to reflect the speedup. We then measured the communication speedup by performing ping-pong tests for messages of different sizes on both systems. The message sizes chosen for this test are the send volumes of the dominating send operations in the applications. The communication speedup is then used together with the mpiP results (not presented) to calculate the adjusted total runtime prediction shown in Figure 4.14. Compared to actual runtimes for MG and CG on Juqueen, our execution time predictions match closely, with an accuracy of 97.72% for MG and 96.81% for CG.

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



(a) MG, Class C

(b) CG, Class C

Figure 4.14: Cross-platform Prediction

4.6 Summary

To bridge the gap between the performance realism of a complete application and the convenience of porting and modifying a benchmark code, we have designed, implemented, and

evaluated a benchmark-generation framework that automatically generates portable, customizable communication benchmarks from parallel applications. Our approach is based on an application's dynamic behavior rather than its statically identifiable characteristics. We use ScalaTrace [55] to recover application structure from a communication trace. Subsequently, coNCePTuaL [58] is used to express the resulting benchmarks in a readable, editable, yet executable format.[2] Algorithms we developed to assist in this process merge collective operations described by disparate source-code lines into a single call point and eliminate nondeterminism caused by wildcard receives. Empirical measurements indicate that the performance of the generated benchmarks is faithful to that of the original application.

There are two main conclusions one can draw from this work. First, it is in fact feasible to automatically convert parallel applications into benchmark codes that accurately reproduce the applications' performance yet are easy to port, read, edit, and reason about. Second, as demonstrated in Section 4.5.5, nonlinear performance effects come into play as applications are modified for nascent architectures, and performance-accurate, application-specific benchmarks are an important new technology for quantifying these effects before exerting the effort involved in application porting.

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

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

Chapter 5

ScalaTrace 2

5.1 Introduction

The compute power of supercomputers has been doubling each year in the past two decades. The era of exascale computing is projected to arrive in the near future. With such large systems, recording the program behavior of parallel applications for post-mortem performance analysis is becoming increasingly difficult. On the one hand, analyzing complicated scientific applications requires complete and accurate performance data. On the other hand, the large number of processors/cores and the increasing gap between computational power and I/O performance pose great challenges on the efficiency and scalability of performance analysis tools. Consequently, traditional analysis tools either collect lossless traces by sacrificing scalability [52] or report only aggregated statistical information that might be insufficient for in-depth performance analysis and debugging [75]. To address this discrepancy, we designed ScalaTrace, a scalable parallel communication and I/O tracing library that features on-the-fly trace compression [56, 64] (see Chapter 2). For single program, multiple data (SPMD) parallel applications, ScalaTrace is able to collect lossless traces that are much more space-efficient than the past approaches.

ScalaTrace represents the state-of-the-art of parallel application tracing for high performance computing (HPC). In this chapter, we present ScalaTrace 2, the next generation ScalaTrace that features a fundamental redesign in every aspect. ScalaTrace 2 is designed to address the shortcomings of the previous work. It targets inefficiencies in the compression of communication traces for applications with inconsistent program behavior across time steps and diverging parallel control flow. For example, coupled large-scale scientific codes such as the Community Earth System Model (CESM) [16] exhibit multiple program, multiple data (MPMD) behavior. They perform multi-physics simulation with different modules using different inputs, executing different algorithms, and running on different sets of processors. To generate scalable traces for these applications, methodologies that better exploit the trace similarities across time steps

and MPI tasks are in demand.

With ScalaTrace 2, we contribute a spectrum of novel compression and replay techniques that are fundamentally different from our past approaches. In ScalaTrace 2, MPI parameters and loop information are stored with the elastic data element representation, a redesigned low-level encoding scheme that is automatically evolving and self-explanatory. By annotating the loop information with participant information, we designed a loop agnostic inter-node compression scheme that guarantees the optimal event matching for applications with task-specific communication patterns. We also redesigned the task-level loop compression to perform approximate loop iteration matching, which is particularly effective for applications exhibiting inconsistent behavior across time steps.

ScalaTrace 2 inherits the lossy philosophy proposed in our previous work [85]. In essence, ScalaTrace 2 can be configured to compress MPI parameters to the utmost using probabilistic methods while still preserving most of the advantages of the lossless approach. As part of the redesign, we developed ScalaReplay 2, a brand-new probabilistic replay engine that is compatible with the loop agnostic trace format. With ScalaReplay 2, we improved the coordinated random value selection of the replay algorithm. Optimizations such as multi-context traversal are also designed to boost the robustness, replay accuracy, and scalability of the replay engine.

We evaluated ScalaTrace 2 with regard to two aspects: (1) the effectiveness of trace compression and (2) the correctness and the timing accuracy of the probabilistic replay. We conducted experiments with a subset of the NAS Parallel Benchmark suite, the Sweep3D neutron-transport kernel, and the Parallel Ocean Program (POP). Experimental results demonstrate that ScalaTrace 2 achieves key improvement on trace compression for benchmarks exhibiting task-specific communication pattern, inconsistent loop behavior, and/or diverging parallel control flow. Results on probabilistic replay show that ScalaReplay 2 is able to accurately reproduce the execution times of the original applications. Across all the test cases, the mean absolute percentage error of the replay times is only 5.7%. Given such accuracy, we conclude that the lossy compression scheme, which is powerful for reducing the trace size, is equally applicable to the scenarios where timing accuracy is required.

To summarize, we made the following contributions in this work:

- We proposed ScalaTrace 2, a fundamental redesign of ScalaTrace that features a spectrum of novel compression techniques to improve the trace compression for applications with inconsistent loop-level and task-level behavior.
- We designed ScalaReplay 2, a probabilistic replay engine compatible with ScalaTrace 2 that is more robust, accurate, and scalable.
- By comparing the compression ratio of two generations of ScalaTrace with computational kernels and real-world applications, we studied the crux of compression inefficiency and

demonstrated potential solutions to obtain better compression.

5.2 Communication Trace Compression and Replay

5.2.1 Elastic Data Element Representation

ScalaTrace 2 features a complete redesign of ScalaTrace, ranging from the very low-level data structures, to the core trace compression algorithms. In this section, we introduce the new elastic data element representation.

A ScalaTrace trace file is a human-readable text file. A data element in the trace file is an integer that represents the value of either an MPI event parameter, such as the destination of a send, or a program control flow parameter, such as the trip count of a loop. From our experience with the previous version of ScalaTrace, we learned that even though a data element is apparently simple as an integer, it may become complicated when the trace is compressed to a high degree. For example, assume a scalar integer value d represents the destination of an MPI_Send operation. When the same MPI_Send is called twice with destinations d_1 and d_2 in consecutive loop iterations, the two events will be compressed due to ScalaTrace's loop compression mechanism (see Section 5.2.2). Thus the scalar value d evolves into a vector (d_1, d_2). When the same loop has multiple iterations, the destination vector may grow in a non-scalable fashion. Hence, a vector compression mechanism is activated. Finally, the inter-node compression further imposes another level of complexity on the data element representation. In a nutshell, a compressed data element ought to contain not only the parameter value, but also the loop information and participant information.

In ScalaTrace 2, we introduce the elastic data element representation and apply this representation to all the data elements in the trace. The elastic data element representation is a list of $< value\ vector, ranklist >$ pairs, where a *value vector* is simply a C++ vector of primitive integer values. On initialization, there is only one such pair in the list consisting of a value vector of a single parameter value and a ranklist of a single participant. During loop compression, new values are appended to the value vector in the order they are generated so that the replay engine or other trace analyzers can traverse the values in the correct order. During inter-node reduction, we merge the ranklists when the value vectors fully match, otherwise a new $< value\ vector, ranklist >$ pair is added to the list. As such, the data element is fully self-explanatory; no additional information is required to resolve a data element for a particular rank and loop iteration.

To keep the size of the value vector scalable, we constantly perform a loop compression against the entries in the vector. Whenever possible, a vector of $m \cdot n$ elements is represented as a vector of m elements and n iterations. Note that the choice of the vector compression algorithm

is not unalterable. For example, run-length encoding might be more efficient for loop parameters when an MPI event is specific only to certain loop iterations (see Section 5.2.2). Since the vector compression mechanism is encapsulated in the elastic data element representation, it is possible to intelligently choose the best compression strategies and convert to the most space-efficient format when necessary.

As the fundamental data structure of ScalaTrace 2, it is vital to guarantee its scalability even under extreme circumstances. We realized that there always exist cases where even a sophisticated algorithm fails to compress the value vectors. We therefore further enhanced the elastic data element representation to exploit probability-based compression using histograms (see Section 2.3) for selected parameter types, such as SOURCE, DEST, and COUNT of point-to-point communication routines. Nonetheless, utilizing such lossy compression techniques poses challenges for the trace replay. We have updated our past probabilistic trace replay technique to address these challenges. The new approach is discussed in detail in Section 5.2.6.

5.2.2 Compressing Partially Matching Loops

ScalaTrace utilizes the MPI profiling layer (PMPI) to intercept MPI calls during application execution. It performs loop detection and compression by searching for consecutive repeating patterns in the MPI event sequence. In contrast to the instruction-level binary instrumentation, which is able to pinpoint the entry and exit points of loop structures, the ScalaTrace approach relies heavily on recognizing repeating patterns. With ScalaTrace 2, we redesigned the task-level loop compression algorithm from the ground up to support the compression of loops with iteration-specific behavior.

Handling Iteration-specific Behavior

Production-grade scientific applications such as the Parallel Ocean Program (POP) demonstrate inconsistent behavior across time steps. POP performs a set of computations and communications of an inner loop in multiple iterations in each time step. Due to inconsistent data-dependent convergence in the computation , the trip counts of the inner loop varies across different time steps. In addition, branches inside loop structures also lead to loop iterations with different event counts and unmatching event sequences. This behavior can also be observed in many Adaptive Mesh Refinement (AMR) applications in which the input set is dynamically rebalanced on a periodic basis. Due to the iteration-specific behavior, ScalaTrace's task-level loop compression fails to compress the loop iterations because its loop detection algorithm requires the loop iterations to be identical event sequences with matching inner loop structures.

In ScalaTrace 2, we overcome the shortcoming by loosening the iteration matching criteria. In order for two consecutive sequences of events $E1, ..., Ea$ and $Eb, ..., En$ to be considered

matching loop iterations, ScalaTrace 2 only requires their beginning and ending events to match, i.e., $E1 == Eb$ and $Ea == En$. Under such criteria, once two matching loop iterations are identified, ScalaTrace 2's aggressive longest common subsequence (LCS) based loop compression algorithm will merge the rest of the events, irrespective of the loop lengths or the inner loop structures. As an example, if a certain node executes the following code,

```
for(int i=0; i<2; i++){
    MPI_Barrier(); // E1
    if(i == 0)
        MPI_Isend(); // E2
    if(i == 1)
        MPI_Irecv(); // E3
    MPI_Barrier(); // E4
}
```

Figure 5.1: Loop with Iteration-specific Behavior

the trace after loop compression will be

$$E1_{(4,2)}\ E2\ E3\ E4,$$

where the subscript of $E1$ indicates that $E1$ is the beginning event of a loop structure of 4 member events and 2 iterations. In general, we use the following mnemonic to describe the loop stack associated with a loop head event E:

$E_{(m_1,i_1)(m_2,i_2)...(m_n,i_n)}$*: E is the head event of a series of n-nested loops, where the outermost loop has a loop length (member event count) of* m_1 *and an iteration count of* i_1*, the second outermost loop has a loop length of* m_2 *and an iteration count of* i_2*, and so on.*

A unique challenge of forcibly merging partially matching loop iterations is to preserve the information of in which iteration a certain event was actually called. As in the example above, a mechanism is needed to tell that $E2$ was only called in the first iteration whereas $E3$ was only called in the second iteration. To address this problem, we represent the loop information as elastic data elements. Recall that the value vector grows as more values are appended to the vector due to loop compression. For example, assuming event E is the head event of loop L, $E_{(a_1\ a_2,b_1\ b_2)}$ indicates that when the first time loop L is executed, it has a_1 member events and b_1 iterations, whereas in the second time, it has a_2 member events and b_2 iterations. (This

is possible when loop L is executed in two iterations of its parent loop). We hence treat every event as a loop of length 1 and iteration 1. During loop compression, we merge the events according to the results of the longest common subsequence analysis and manipulate the loop information according to the following rules:

1. An event $E_{(1,0)}$ is called a *dummy event* because the loop information indicates that it is executed zero times.

2. For event $E_{(a_1\ a_2...,b_1\ b_2...)(c_1\ c_2...,d_1\ d_2...)...}$, adding an outer loop of one iteration to obtain a new event $E_{(a_1\ a_2...,1\ 1...)(a_1\ a_2...,b_1\ b_2...)(c_1\ c_2...,d_1\ d_2...)...}$ does not change the loop structure in terms of the times and order the events in the nested loops are executed. The added outer loop is thus called a *pseudo-loop* with just one iteration.

3. Assume $I1$ and $I2$ are matching loop iterations. If event $E1_{(a_1\ a_2...,b_1\ b_2...)(c_1\ c_2...,d_1\ d_2...)...}$ in iteration $I1$ matches with event $E2_{(i_1\ i_2...,j_1\ j_2...)(k_1\ k_2...,l_1\ l_2...)...}$ in iteration $I2$, merge $E1$ and $E2$ by merging the loop information at corresponding levels. Merging the value vectors is accomplished by appending the value vector of $I2$ to the value vector of $I1$. If the loop stack depth d_1 at $E1$ does not match the depth d_2 at $E2$, e.g., $d_1 > d_2$, align the loop stacks by adding pseudo-loops to the top of the loop stack at $E2$ as placeholders to avoid mismatching the extra outer loops at $E1$ with $E2$ during traversal.

4. If event E is in iteration $I1$ but not in iteration $I2$, create a dummy event E' of E and insert it into $I2$ immediately before the matching event M of $I1$ and $I2$ returned by the longest common subsequence analysis. The loop stack of E' is created according to that of E by adding pseudo-loops. The vector values at each nest level of E' are generated by referring to M for the number of times it would be encountered if it were in $I2$. As such, E' acts as a placeholder to avoid mistakenly calling E when executing $I2$. After the insertion of all the iteration-specific events, merge iterations $I1$ and $I2$ according to the third rule.

5. When merging iteration $I2$ into iteration $I1$, if the outermost loop of the head event E of $I1$ is not a loop of the entire event sequence of $I1$, a new outer loop is identified and a new loop descriptor $(n, 2)$ is added to the top of the loop stack of E, where n equals to number of events in both $I1$ and $I2$.

In essence, the rules above ensure that the iteration-specific events will only be executed in the correct iterations. The introduction of the pseudo-loops as placeholders guarantees that 1) iteration-specific events are evaluated but not executed in the loop iterations they do not belong to, and, therefore, 2) meaningful loop information is evaluated and fetched in the correct loop iteration. Nevertheless, the core of the loop compression algorithm is still the longest common

subsequence analysis performed against the two sequences of MPI events. The matching event pairs returned by the LCS analysis are the basis for loop information adjustment. Therefore, the complexity of this algorithm is $O(m \cdot n)$, where m and n are the numbers of events in the two event sequences, respectively.

By applying the aforementioned loop compression guidelines, ScalaTrace 2 is able to compress loops with iteration-specific behavior. For example, the MPI code shown in Figure 5.1 is eventually compressed as follows:

$$E1_{(4,2)}\ E2_{(1,1\ 0)}\ E3_{(1,0\ 1)}\ E4$$

Handling the Trailing Iteration

The NAS Parallel Benchmarks (NPB) BT code exemplifies a pattern of an MPI event sequence that a multitude of stencil codes share at the end of a time step. Each MPI task communicates with its neighbors with a series of send, receive, and wait operations in a loop, as illustrated with the simplified example in Figure 5.2.

```
/* m time steps */
for(int i=0; i<m; i++){
    ... // MPI events
    for(int j=0; j<n; j++){
        MPI_Isend();
        MPI_Irecv();
        MPI_Waitall();
    }
}
```

Figure 5.2: Loop with Trailing Iterations

The original ScalaTrace compresses the outer loop only if n, the trip count of the inner loop, is a constant. In contrast, ScalaTrace 2's more aggressive loop compression discussed in Section 5.2.2 can always compress the outer loop even if n is not a constant, e.g., $n = f(i)$. However, since ScalaTrace 2 eagerly compresses the detected loops, the second iteration of the outer loop will be compressed immediately when the first iteration of the inner loop terminates. As a result, the remaining $n-1$ iterations of the inner loop cause the *trailing iteration* problem.

To address the trailing iteration problem, we redesigned the entire loop detection and compression algorithm to perform a delayed merge, as shown by Algorithm 4. Essentially, Algorithm 4 does not eagerly merge a newly identified loop iteration. It rather marks it as a

pending iteration so that a potential trailing iteration can be merged with the pending iteration later without having to perform any decompression. Specifically, after an event E is appended to the trace, DetectLoop() is called to find the *target_head*, *merge_tail*, and *merge_head* for two matching loop iterations ending with *target_tail* == E. Assuming a loop structure is found, the sequence *merge_head*, ..., *merge_tail* can be 1) the ending subsequence of a pending iteration detected previously, 2) a pending iteration whose match has already been found, or 3) an independent sequence in no loop structures. In the first case, the sequence *target_head*, ..., *target_tail* is identified as a trailing iteration. An event sequence can simultaneously be the trailing iteration of multiple pending iterations in a nested manner, so line 16 - 21 of Algorithm 4 updates all the pending iterations by appending the trailing iteration to each of them. In the second case, where the sequence *merge_head*, ..., *merge_tail* is itself a pending iteration, the sequence *target_head*, ..., *target_tail* is then identified as the next iteration of the pending iteration. Since it is now safe to conclude that there will be no more trailing events for the pending iteration *merge_head*, ..., *merge_tail*, Algorithm 4 performs the delayed iteration merge by calling MergePendingIteration() (line 22 - 24). Finally, the newly detected iteration *target_head*, ..., *target_tail* is always marked as a pending iteration in either case. The function MergePendingIteration() merges the iteration between the events *pending_head* and *pending_tail* with the iteration between the memorized events *head* and *tail*. Before calling LCSLoopCompression() which implements the algorithm introduced in Section 5.2.2, MergePendingIteration() first compresses the pending iterations of all the nested inner loops (line 32 - 51) by recursively calling itself. Finally, when MPI_Finalize is called, all loops have either zero or one pending iteration. To eventually merge them, the function FinalizePendingIterations() is called, which treats the entire trace as an iteration and recursively merges the inner loops by calling MergePendingIteration().

5.2.3 Approximate Stack Signature Matching

ScalaTrace preserves a call stack signature by logging the call sites of the calling stack for each event. Using these stack signatures, ScalaTrace is able to distinguish MPI calls of the same type by their locations in the program. The stack signature therefore serves as the only basis for comparison of events, which then makes loop detection possible. Nonetheless, strictly enforced stack signature comparison may not always benefit trace compression. For example, POP wraps MPI_Bcast of different data types with different functions, which are then invoked in 36 different files at approximately 400 different locations. Experimental results show that due to such usage, the size of the trace of the initialization stage — a stage that is usually less important for performance analysis — accounts for 26% of the size of the final trace. More commonly, a number of scientific applications, including POP and the NPB BT and SP

Algorithm 4 Loop Compression with Delayed Merge

Precondition: T: the trace after a new event was appended as the new tail

```
1: function DetectLoop(T)
2:     target_tail ← T.tail
3:     merge_tail ← T.FindMatch(target_tail)
4:     target_head ← merge_tail.next
5:     while true do
6:         merge_head ← T.FindMatch(target_head)
7:         if merge_head.isPendingMember == true then                ▷ potential trailing iteration, more checks
8:             pending_tail ← FindPendingTail(merge_head)
9:             if pending_tail == merge_tail then   ▷ trailing iteration of a pending iteration must follow the pending iteration immediately
10:                break
11:            end if
12:        else                                                      ▷ not a trailing iteration, no more check
13:            break
14:        end if
15:    end while

16:    if target_head...target_tail is a trailing iteration then
17:        PIs ← FindPendingsForTrailing(target_head, target_tail)
18:        for pendingIteration ← PIs.first, PIs.last do
19:            pendingIteration.AddTrailing(target_head, target_tail)
20:        end for
21:    end if

22:    if merge_head...merge_tail is a pending iteration then  ▷ perform the delayed merge: merge a pending iteration only when the next iteration (namely, target_head...target_tail) is found
23:        MergePendingIteration(merge_head, merge_tail)
24:    end if

25:    for event ← target_head, target_tail do
26:        event.isPendingMember ← true
27:    end for
28: end function

29: function MergePendingIteration(pending_head, pending_tail)
30:    head ← FindKnownMergeHead(pending_head)
31:    tail ← FindKnownMergeTail(pending_tail)

32:    for event ← head, tail do
33:        if event.IsKnownMergeHead() == true then
34:            h ← FindPendingHead(event)
35:            t ← FindPendingTail(event)
36:            new_tail ← MergePendingIteration(h, t)
37:            if tail == t then
38:                tail ← new_tail
39:            end if
40:        end if
41:    end for

42:    for event ← pending_head, pending_tail do
43:        if event.IsKnownMergeHead() == true then
44:            h ← FindPendingHead(event)
45:            t ← FindPendingTail(event)
46:            new_tail ← MergePendingIteration(h, t)
47:            if pending_tail == t then
48:                pending_tail ← new_tail
49:            end if
50:        end if
51:    end for

52:    LCSLoopCompression(head, tail, pending_head, pending_tail)
53:    DeleteEvents(pending_head, pending_tail)
54:    return tail
55: end function

56: function FinalizePendingIterations(T)
57:    ...
58: end function
```

codes, are coded according to the data decomposition, their communication topology, or their simulation stages, with the MPI events hidden deep in the call stack, as shown in Figure 5.3. As a result, these applications also create trace events with various stack signatures even though

they are functionally symmetric. In both cases, there exists the need to trade the call stack information for better compression.

```
simulate(){
    while(i<s){
        solve_x();
        solve_y();
        i++;
    }
}
```

```
solve_x(){
    // compute
    MPI_Isend();
    MPI_Irecv();
    MPI_Wait();
    MPI_Wait();
}
```

```
solve_y(){
    // compute
    MPI_Isend();
    MPI_Irecv();
    MPI_Wait();
    MPI_Wait();
}
```

Figure 5.3: The Simplified NPB BT Code

In ScalaTrace 2, we loosened the stack signature comparison criteria to tolerate a pre-defined number of different frames. When comparing two stack signatures, we start from the first call site (the *main* function) and compare the call sites in corresponding frames one by one. The comparison returns *true* only if the number of different frames is less than the user-defined threshold. As an ongoing improvement, we also allow users to specify a range of instruction addresses, so that the call sites within it will always be considered a match. During loop compression, we compare event $E1$ in iteration $I1$ with event $E2$ in iteration $I2$ by both event type and stack signature. If their signatures differ less than the pre-defined limit, we replace $E2$'s signature with that of $E1$ and update all the signature-annotated statistics of $E2$'s succeeding events accordingly. With such user-configurable stack signature imprecision, ScalaTrace 2 is able to better exploit the potential of trace compression than the original ScalaTrace for applications like POP or BT, as illustrated in Figure 5.3. The user-defined limit serves as a tuning parameter to trade off compression against accuracy.

5.2.4 Loop Agnostic Inter-node Compression

ScalaTrace performs inter-node trace compression to exploit the single-program multiple-data (SPMD) paradigm of scientific applications. However, the compression capability of the original ScalaTrace is limited by the fact that loop structures must be treated as an indivisible unit. For example, the code in Figure 5.4 cannot be compressed across nodes because the for loops on different nodes have mismatching trip counts and event sequences. The heavy dependence on the perfect matching of loop structures is partially alleviated by the recursive loop matching algorithm proposed in our previous work [85], but it still cannot handle the case where loop trip counts do not match.

```
Rank 0:                          Rank 1:
1:   for(i=0;i<5;i++){           1:   for(i=0;i<6;i++){
2:       MPI_Isend(1);           2:       MPI_Isend(0);
3:       MPI_Irecv(1);           3:       MPI_Irecv(0);
4:   }                           4:       MPI_Waitall(2);
5:   MPI_Isend(1);               5:   }
6:   MPI_Irecv(1);
7:   MPI_Waitall(12);
```

Figure 5.4: Code Needs Loop Agnostic Inter-node Compression

The restrictions on loop structures for inter-node compression is eliminated in ScalaTrace 2 due to the introduction of the elastic data element representation. Close examination shows that the crux of the compression problem stated above is the coupling of the loop information and the event participants information. Specifically, the loop structure is formed during task-level loop compression and only applies to the same task. Given two loop head events from different MPI tasks, they cannot be merged if the loop structures diverge, because there is no mechanism to recover the task-specific loop information once it is compressed. By representing the loop information with the elastic data elements, a separate ranklist is attached for each loop data element (including the trip count and the loop length). When merging two loop head events, we simply merge the loop structures at each corresponding loop level by following the general rules of compression of the elastic data element representation. Namely, if the loop structures match, they are compressed by merging their ranklists. If the loop structures at a certain loop level do not match, they are merged by adding another $< value\ vector, ranklist >$ pair to distinguish the task-specific loop information.

By decoupling the loop information and the event participants information, ScalaTrace 2 is able to perform loop structure agnostic inter-node compression. During inter-node compression, the longest common subsequence is determined by evaluating only the stack signatures of the events from different MPI tasks, and the events are then merged accordingly. For example, the events of the code shown in Figure 5.4 will be merged into the trace shown in Figure 5.5, where the number in [and] shows the ranklist of the loop information before it, and the MPI parameters are ignored.

Lastly, since the new inter-node compression algorithm is loop agnostic, it does not have to perform the LCS analysis recursively for each level of the nested loops. Hence, assuming m and n are the lengths of two task-level traces, the complexity of the inter-node compression algorithm is $O(m \cdot n)$.

Rank	Event
0 1	$MPI\_Isend()_{(2,5)[0](3,6)[1]}$
0 1	$MPI\_Irecv()$
0	$MPI\_Isend()$
0	$MPI\_Irecv()$
0 1	$MPI\_Waitall()$

Figure 5.5: Final Trace of the Code in Figure 5.4

5.2.5 Customizable Instrumentation

Diagnosis of an application's performance problem requires the collection and analysis of performance data describing the application's behavior. To ensure tool scalability, it is important that one collects only the data relevant to the performance problems at hand. This is because blindly collecting useless performance data inevitably introduces unnecessary perturbation, and thus may cause the user to mis-identify the real cause of the problem. In addition, collecting excessive performance data also makes data analysis difficult as more resources are required for analysis.

We propose to solve this problem with dynamically customizable instrumentation. ScalaTrace utilizes the MPI profiling interface to collect the performance data about the MPI events and the preceding computation at the entry and exit points of MPI routines. With ScalaTrace 2, we allow the users to conveniently supply customized profiling functions that will be called at the same locations as the native profiling code by simply inheriting from a base class named *Stat* and overriding two virtual methods, *start*() and *end*(). The functions *start*() and *end*() operate on a data structure that is internal to the user-provided child class to gather the performance data. When being called automatically by ScalaTrace 2 at runtime, *start*() initializes or resets the data structure, and *end*() calculates the performance data and returns the result as a floating point number to the runtime framework. Once the performance data is returned to the framework, it is placed into the histogram with the correct predecessor stack signature. During trace compression, the base class *Stat* invokes the $Stat :: merge()$ function to merge the histograms of matching execution paths to engage in context-aware statistical data compression. In ScalaTrace 2, we also made the *Event* class accessible to the *Stat* class (and thus all of its children) associated with it, so that the user-defined profiling code will have additional information to selectively collect the performance data for particular messages/functions/libraries.

Currently, ScalaTrace 2 only supports the collection of numerical performance data with customizable instrumentation. Internally, ScalaTrace 2 utilizes histograms to store and compress performance statistics. Numerical data is sufficient for expressing most types of performance data, such as execution time and performance counter values. In fact, the *StatTime* class —

a class that records the execution times of the computation and communication stages of each event — is implemented as a child class of *Stat* using numerical data. Nonetheless, we are also working on supporting various complicated data types. In the case where the histogram representation is incapable of storing the performance data, the users can define their customized data types T with a compression scheme $T :: merge(T)$ provided. As such, the ScalaTrace 2 runtime framework can merge the customized data type by referring to the user-provided compression function while still being responsible for the context-aware statistical data compression integrated within the built-in algorithm.

5.2.6 Replaying Non-deterministic Trace

For scalability reasons, ScalaTrace 2's elastic data element representation may internally transform from the lossless $< value\ vector, ranklist >$ pair representation to the lossy histogram representation for pre-configured parameters including SOURCE, DEST, and COUNT. As was discussed in our prior work, representing non-performance data, such as DEST, as histograms still preserves meaningful information regarding the communication topology [85]. However, it poses a great challenge to the re-creation of the program behavior from the probabilistic trace because the critical communication parameters are not accurate anymore.

ScalaReplay 2 is the new replay engine designed to cope with probabilistic traces. In contrast to the previous version of ScalaReplay, we redesigned the trace traversal algorithm to support ScalaTrace 2's loop agnostic traces. We have also made key improvements in ScalaReplay 2 to boost the robustness and replay accuracy. ScalaReplay 2 utilizes the coordinated random value selection approach described in our previous work. In essence, during replay, nodes parse send events but skip receive events in the trace. At send events, senders select receivers from the DEST histograms by referring to random numbers. In order to generate matching receives for the send operations, each node parses the traces of the other nodes to locate the send operations addressed to itself. This is made possible by the fact that the senders and the potential receivers agree on the random numbers used for value selection. In this way, the overhead of exchanging control messages via back-channel communication is avoided.

Improvements for ScalaReplay 2 center around a novel trace traversal strategy. In the past approach, each node used a single pointer to traverse a global trace; whenever there is a loop structure, the node traverses it as a participant. However, this is impossible with ScalaTrace 2's new trace format because loop structures interleave in the final trace, as shown in Figure 5.5. In addition, traversing with a single pointer also causes timing accuracy problems. With the previous approach, a node parsed every event in the order it was seen during the traversal. However, due to a stack signature mismatch, events that happened simultaneously may be recorded far apart in the trace, as shown in Figure 5.6. With the single pointer approach, task

0 will not issue MPI_Irecv(1) until it reads event 4 after having performed some computation for 10 seconds at event 2. As a result, task 1 will be blocked at the blocking send (event 4) for 10 seconds and the total runtime of the program approximates 20 seconds, i.e., almost twice as much as it ought to be.

```
1:   if(rank==0){
2:       MPI_Irecv(1);
3:       compute(10s);
4:       MPI_Wait();
5:   }else if(rank==1){
6:       MPI_Send(0);
7:       compute(10s);
8:   }
```

ID	Rank	Event
1	0	$MPI\_Irecv(1)$
2	0	$compute(10s)$
3	0	$MPI\_Wait()$
4	1	$MPI\_Send(0)$
5	1	$compute(10s)$

Figure 5.6: Trace Needs Multiple Context Pointers for Replay

To address these issues, ScalaReplay 2 utilizes multiple traversal context pointers during replay. Intuitively, a trace reflects the result of a parallel execution, where the parallel processes progress concurrently through potentially distinct control flows with occasional synchronizations. The replay engine mimics this process by replaying with one primary traversal pointer while keeping track of the other nodes' parallel executions with multiple additional traversal context pointers. In ScalaReplay 2, a traversal context is a lightweight data structure that keeps track of the progress of the traversal on behalf of a certain MPI task. It consists of an event pointer, a loop information manager, a random number manager, and a timer. The event pointer always points to the next event to be replayed/evaluated. It supports the operations such as $hasNext()$ and $next()$. The loop information manager keeps track of the traversal by memorizing the current loop stack as well as the iteration counts at each loop level. The random number manager guarantees that contexts of the same rank on different nodes always agree on the same series of random numbers. Lastly, the timer is used to calculate the aggregated execution time at a certain event according to the recorded times of the events already traversed so far.

During replay, each node progresses according to its primary context, i.e., the context with the rank of the node. It issues all the events other than receives, and sleeps for all the recorded computational phases within its context as a normal replay. To post matching receives for potential senders, each node also traverses the trace of the other nodes by maintaining secondary traversal contexts for them. When traversing the secondary contexts, all the non-

send events and computations are ignored so that the current node can quickly identify the send operations addressed to itself. However, the current node does not post the receive immediately when a send is identified. Instead, it postpones the receive until approximately the time the corresponding send operation is issued at the sender side, which can be estimated by referring to the timer of the sender's secondary context. By posting receives in this way, ScalaReplay 2 manages to clear the system receiving buffer in a timely manner and thus improves the replay time accuracy.

To further improve the performance and scalability of ScalaReplay 2, additional optimizations are implemented. In practice, most parallel applications are designed such that each node only has a limited number of point-to-point communication destinations (otherwise, collectives are used). We therefore have introduced a negotiation stage before the replay in which each node calculates a destination set of a configurable size and informs the selected receivers so that each node only has to maintain a limited number of traversal contexts during the replay. In addition, we also improved the performance of the replay engine by overlapping the context management with simulated compute times that the primary context has to perform anyhow. With these optimizations, the replay engine manages to scale to a large number of nodes.

5.3 Evaluation

We evaluated ScalaTrace 2 with regard to two aspects: (1) its effectiveness of trace compression, as well as the advantages of different compression optimizations, and (2) the correctness and the timing accuracy of the probabilistic replay. For experiment (1), we used a subset of the NAS Parallel Benchmark suite (version 3.3 for MPI) [6], including BT, CG, LU, MG, and SP, the Sweep3D neutron-transport kernel [79], and the Parallel Ocean Program [61]. We chose these benchmarks because they exercise both collectives and point-to-point communications in multiple time steps. Besides, some of these benchmarks either do not have consistent loop behavior or do not show strict SPMD regularity. Consequently, these benchmarks poses great challenges to the existing trace compression libraries, including the last generation ScalaTrace. In these experiments, we configured ScalaTrace 2 to use different compression algorithms and we study the impact of each option. In experiment (2), we assessed ScalaTrace 2 and ScalaReplay 2's capability of preserving and re-producing the computational performance with respect to wall clock execution times. Particularly, we conducted all the replay experiments with probabilistic traces, which is significantly more challenging than replaying with lossless traces. For the second experiment, we still used the same benchmarks as in experiment (1).

We conducted all the experiments on ARC, a cluster with 1,728 cores on 108 compute nodes, 32 GB memory per node, and an Infiniband QDR interconnect. Due to limited access to the system, our experiments generally run on a subset of the available nodes that is sufficient to

reflect the trend of the trace size with respect to the increasing execution scale.

5.3.1 Trace File Size

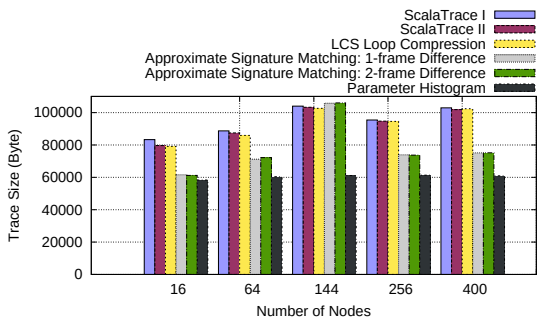
In the first experiment, we evaluated ScalaTrace 2's compression effectiveness with the NPB BT, CG, LU, MG, SP codes, Sweep3D, and POP. We chose these benchmarks because they are all stencil codes exhibiting multi-dimensional communication topologies and complicated loop structures. Among these benchmarks, MG, SP, and POP demonstrate inconstant message sizes and irregular SPMD behavior, and are hence particularly challenging for lossless and structure-preserving trace compression. In these experiments, we compared ScalaTrace 2 with our past approach. In order to demonstrate the effect of different configurations, we itemize the optimizations (as explained below) and collected traces by applying the options in an incremental manner:

- *ScalaTrace II*: features only the loop agnostic inter-node compression enabled by the elastic data element representation;
- *LCS Loop Compression*: additionally performs the longest common subsequence based loop compression;
- *Approximate Signature Matching*: adds a finer-grained optimization that matches and merges events when stack signatures differ by no more than a pre-defined threshold;
- *Parameter Histogram*: adds lossy compression that converts overlong value vectors into histograms.

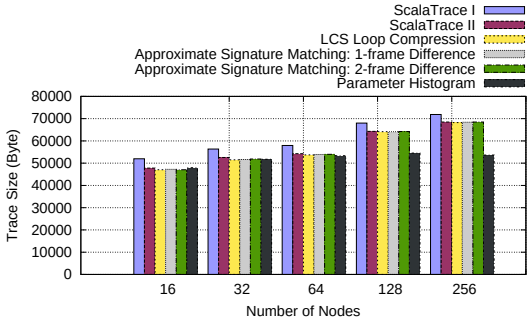
According to the improvement achieved with ScalaTrace 2, benchmarks are divided into three categories and the results are shown in Figure 5.7.

The first category consists of BT and CG. These benchmarks either demonstrate perfectly matching loop iterations or regular SPMD behavior that leads to structurally identical task-level traces. Hence, our past approach is able to capture the loop structures and has little difficulty merging the time step loops across tasks. Consequently, ScalaTrace 2 only shows limited improvement in trace size for BT and CG when configured to be fully lossless. Nevertheless, there is still room for improvement when fuzziness is allowed. For example, by applying the approximate stack signature matching, ScalaTrace 2 manages to deliver another 22% trace size reduction on average for BT. More importantly, when parameter histograms are enabled for the elastic data elements, we eventually obtained constant sized traces for BT and CG — a critical improvement that makes key difference at large scale.

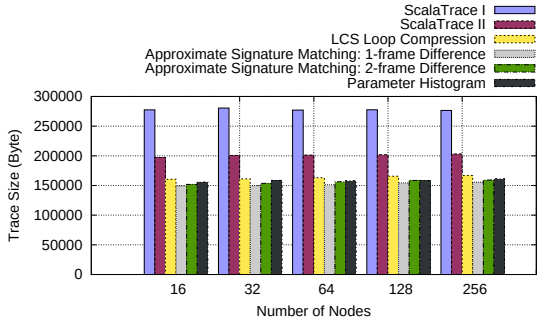
LU and Sweep3D constitute the second category for which ScalaTrace 2 improves the trace compression by forcibly merging the task-level traces in a loop agnostic way. Both LU and



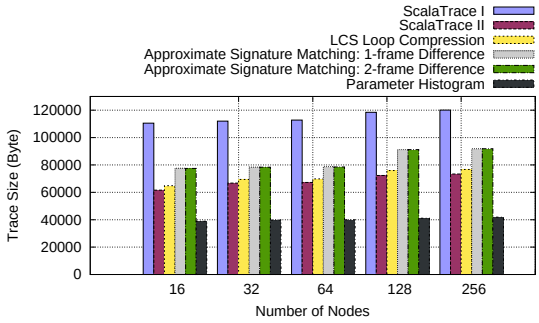
(a) BT



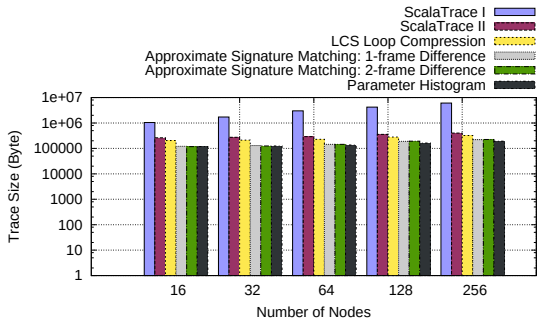
(b) CG



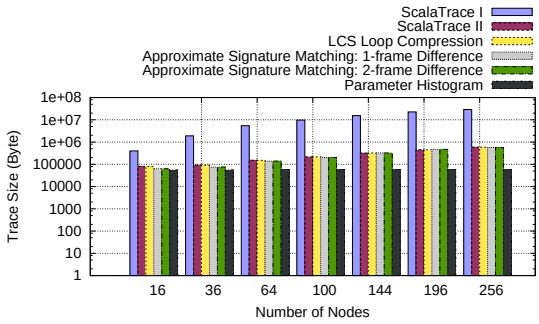
(c) LU



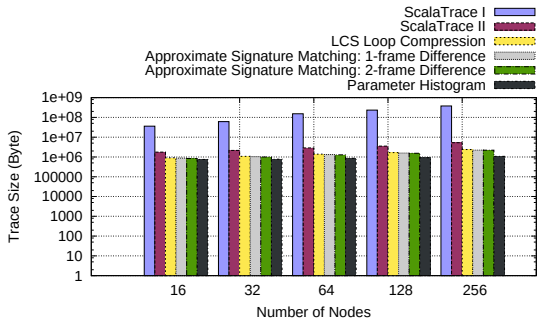
(d) Sweep3D



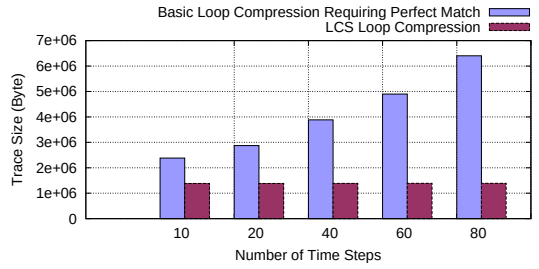
(e) MG



(f) SP



(g) POP (Increasing Node Counts)



(h) POP (Increasing Time Steps)

Figure 5.7: Trace File Sizes for NPB BT, CG, LU, MG, SP, Sweep3D, and POP

Sweep3D are stencil codes with a 2D task layout. Depending on the location in the 2D communication topology, a node may have a different number of neighbors, and thus follow a communication pattern that is unique to one of the nine communication groups (4 corners, 4 boundaries, and the interior nodes). Because the past approach matches the loop structures as an entirety during the inter-node reduction, it fails to exploit the similarities in the time step loops across communication groups. ScalaTrace 2 takes advantages of these similarities. Consequently, by applying the loop agnostic inter-node compression alone, ScalaTrace 2 manages to reduce the trace size by 28% for LU and 41% for Sweep3D. By enabling extra optimizations, ScalaTrace 2 eventually generates traces that are 43% and 65% smaller, for LU and Sweep3D, respectively.

The most compelling improvement, however, is observed for the NPB MG and SP codes, and the Parallel Ocean Program. Among these benchmarks, SP sends messages with fluctuating sizes in loops. This prevents loop compression of the last generation ScalaTrace but can be handled with ScalaTrace 2 where all parameters are represented as elastic data elements. MG is the most challenging test case of the NAS Parallel Benchmark suite. It features a complicated communication pattern consisting of a primary 7-point 3D torus and a secondary nested 3D torus among the nodes at particular positions in the topological space. Due to the interleaving of the two patterns, MG demonstrates both iteration-specific behavior and task-specific behavior, and thus poses great challenges for both intra-node and inter-node compression. By utilizing the approximate loop iteration matching and the loop agnostic inter-node compression, ScalaTrace 2 produces lossless traces that are orders of magnitude smaller for MG and SP, as shown in Figure 5.7(e) and 5.7(f) on a logarithmic y axis. To further compress the varying message sizes for SP, we enabled lossy tracing with parameter histograms and thus obtained near constant sized traces that are collectively only 0.5% of the trace size of the past approach.

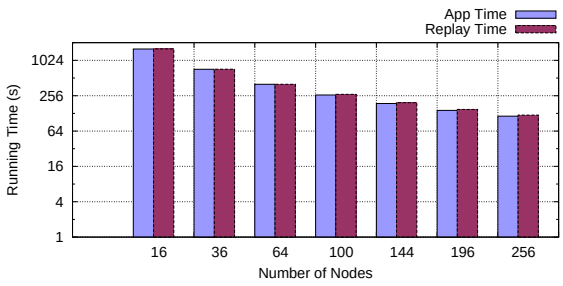
POP performs ocean simulations for multiple time steps. It demonstrates both inconsistent loop behavior across time steps and diverging task-level behavior that hinders inter-node compression. In previous work, we applied the probabilistic compression technique at the MPI parameter level and managed to greatly reduce POP's trace size [85]. With ScalaTrace 2, we improve our previous work by more systematically exploiting the structural properties in the trace. As shown in Figure 5.7(g), by utilizing the loop agnostic inter-node compression, we reduced the trace size by a maximum of two orders of magnitude. This is almost as much of an improvement as we obtained with the previous lossy approach, yet still maintain lossless traces. Furthermore, after additional optimizations are enabled, we eventually obtained near constant sized traces that are 48 to 351 times smaller. Besides, we also conducted experiments to assess the time step scalability, namely how efficient ScalaTrace 2's longest common subsequence based loop compression can detect and compress the time step loops. For this experiment, we keep the execution scale at 64 MPI tasks and increase the number of time steps. Experimental

results in Figure 5.7(h) show that, by utilizing the LCS based approximate loop matching, ScalaTrace 2 is able to produce a constant sized trace. In contrast, the traditional approach is sensitive to iteration-specific events and thus is not time step scalable for POP. From these experimental results, we conclude that even with the parameter-level probabilistic compression techniques, it is still possible to analyze the structural properties in the trace systematically and perform compression in a top-down manner.

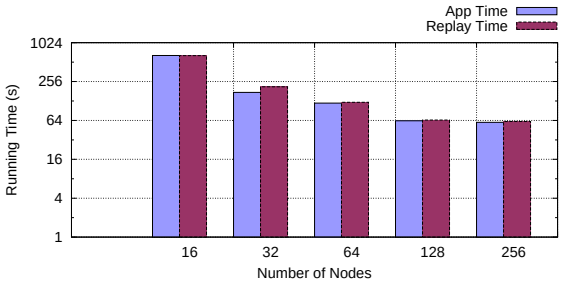
5.3.2 Probabilistic Replay Time Accuracy

In the second set of experiments, we assess ScalaTrace 2 and ScalaReplay 2's capability of preserving and re-producing computational times. We focus on 1) whether ScalaReplay 2 is able to correctly coordinate the random value selection across nodes to replay the probabilistic traces without deadlock, and 2) how accurate the probabilistic replay can re-produce the execution times of the original applications. We conducted these experiments with the same set of benchmarks that we used for the first experiment. Among these benchmarks, the NAS Parallel Benchmarks and POP are strong-scaling codes and Sweep3D is a weak-scaling code. The problem sizes were chosen for the strong-scaling benchmarks to ensure that there is a reasonable amount of computation across all the tested scales. To provide input for the replay engine, we configured ScalaTrace 2 to collect probabilistic traces where the critical MPI parameters, including SOURCE, DEST, and COUNT, are represented with histograms. Particularly, we set the histogram-triggering threshold to 1 to forcibly convert all the elastic data elements into histograms irrespective of how concise the lossless value vectors actually are.

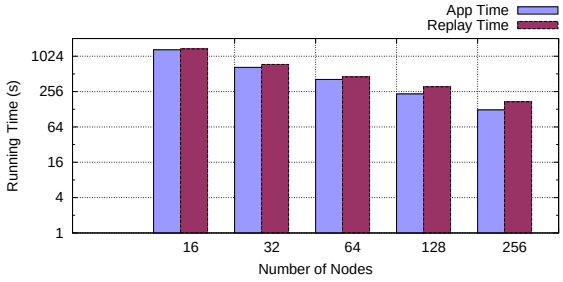
Figure 5.8 compares the probabilistic replay times with the execution times of the original applications. First, being able to obtain these results allows us to validate that the coordinated random value selection of the replay algorithm is deadlock free for the evaluated benchmarks. Besides, the experimental results also show that ScalaReplay 2 can accurately re-produce the computational performance of the original applications. Quantitatively, the mean absolute percentage error of the replay times (i.e., $100\% \times |(T_{\text{replay}} - T_{\text{app}})/T_{\text{app}}|$) across all the test cases in Figure 5.8 is only 5.7%. Such high replay timing accuracy indicates that 1) the execution times are accurately preserved by the lossy traces and 2) the probabilistic replay approach is able to re-produce the runtimes of the original applications without introducing unmanageable control overhead. Overall, given the accurately preserved and re-produced performance characteristics, we conclude that the histogram-based lossy compression, which has been shown to produce powerful reductions in the trace size, is equally applicable to the scenarios where timing accuracy is required.



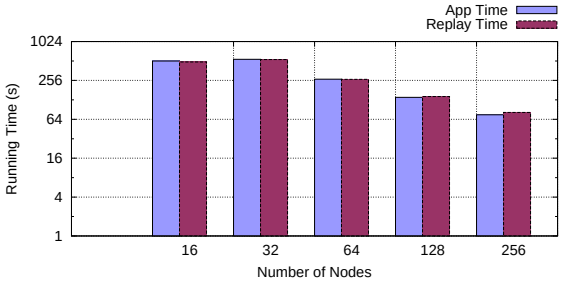
(a) BT, Class C



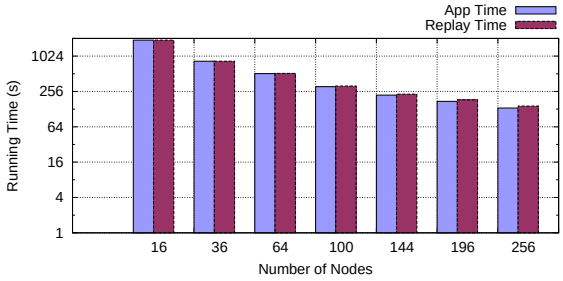
(b) CG, Class D



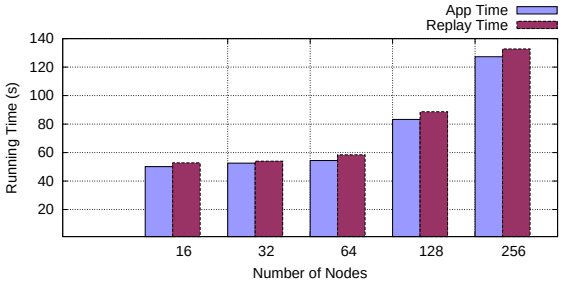
(c) LU, Class C



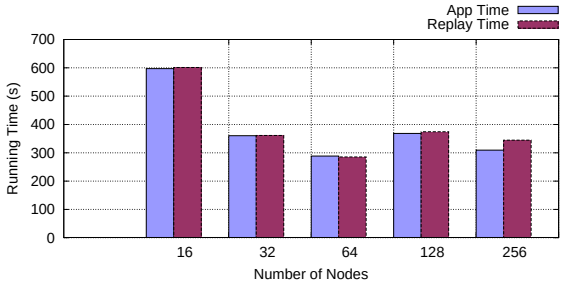
(d) MG, Class D



(e) SP, Class C



(f) Sweep3D



(g) POP

Figure 5.8: Probabilistic Replay Time Accuracy

5.4 Related Work

Our work is closely related to prior research in the area of parallel application tracing and profiling [52, 59, 32, 71]. Traditional tracing tools such as Vampir [52], Extrae [26], and Paraver/Dimemas [59] collect plain application traces that are not scalable due to the sheer size of the performance data gathered. The Open Trace Format (OTF) aims at scalable tracing [40]. However, since OTF utilizes regular zlib compression, the tools based on it generally lack structure-aware compression. Consequently, these tool cannot fully exploit the structural similarities, nor are they suitable for trace-based scalability analysis and code generation [83, 84, 82].

SEQUITUR exploits hierarchical structures in sequences of discrete symbols for compression [53, 54]. It constructs a context-free grammar for a given sequence by representing repeating digrams as non-terminals. Because SEQUITUR excels in both data compression and structural inference, it is employed by an array of algorithms and tools as the compression infrastructure for a variety of purposes. For example, Marathe et al. utilize SEQUITUR to compress data access instructions in their memory tracing work [51]. Larus proposes whole program paths (WPP) to capture a program's dynamic control flow, where an enhanced SEQUITUR algorithm is designed to compress acyclic path traces [45]. Krishnamoorthy et al. present a trace compression algorithm that is largely based on SEQUITUR [43]. To fully exploit the pattern detection capability of SEQUITUR, this work performs trace compression at argument level instead of event level. While this optimization is effective in improving the compression, it also makes the final trace unreadable and not structure-preserving. In general, if the program structure is not preserved at the event level, most post-processing or trace-based performance analysis becomes difficult or even infeasible because decompressing and effectively rendering the trace may require large amounts of memory and computing power that are not available on commodity desktops or laptops.

Recent advances in online trace compression utilize domain-specific techniques to achieve trace size reduction. ScalaTrace performs task-level loop compression and cross-node trace compression in a memory-efficient manner [56]. It generates near constant sized or orders of magnitude smaller traces for SPMD codes. Xu et al. construct coordinated performance skeletons from traces to estimate application execution time in new hardware environments [86, 87]. They adapt a pattern analysis algorithm from bioinformatics to perform loop analysis. Nonetheless, due to the lack of on-the-fly loop compression, this tool is subject to limitations on time step scalability. Knupfer et al. utilize Complete Call Graph (CCG) to hierarchically store an application trace according to the call stack [41]. By comparing and merging similar subgraphs, the trace is compressed in a bottom-up fashion. In contrast to our work, the CCG based approach cannot handle inconsistent program behavior that leads to mismatching low-level sub-structures in the CCGs.

Besides lossless or near lossless tracing techniques, our work is also related to the work that provides lightweight application profiling functionalities. For example, mpiP collects aggregated statistical information about MPI functions and computation times [75]. Gprof measures the durations and frequencies of procedures using a hybrid approach of instrumentation and sampling [33]. HPCToolkit collects call path profiles [28, 2]. To further reduce the overhead involved in profiling, Gamblin et al. utilize statistical sampling and parallel clustering techniques to reduce the number of parallel processes from which the performance data is collected, and thus improve the scalability of parallel profiling tools [31, 30, 29]. In contrast to the lossless tracing approach, tools like mpiP generally report simple and high-level information that is only suitable for a superficial understanding of performance problems. For in-depth performance debugging or complicated analysis, application tracing is still necessary. As an instrumentation framework for both communication event tracing and performance data collection, ScalaTrace 2 can employ the statistical methods proposed in prior research to improve its numerical performance data collection and compression. This is left as a future work.

A unique approach for quick acquisition of communication traces involves program slicing. Program slicing is a source code analysis technique that effectively reduces a program to a subset of the statements (a program slice) that is relevant to a target statement or variable. As an example, Zhai et al. proposed the FACT approach [90]. This approach constructs a program slice for MPI calls and strips out the computation. With the communication-only program slice, it then becomes feasible to obtain the communication trace readily without executing the computation. While it is possible to combine the program slicing approach with existing communication trace compression techniques, an inherent shortcoming of program slicing is that it neither captures the execution times, nor can it handle data-dependent control flows. Consequently, this technique is only applicable in limited scenarios.

In addition to techniques on communication tracing and profiling, our work is also relevant to prior research on parallel replay. For example, Rolt^{MP} proposes a Lamport timestamp-based approach for the deterministic replay of programs with non-deterministic receives [66]. MPIWiz is a deterministic replay method that is able to replay only a subset of the tasks of an MPI application [88]. Phantom employs deterministic replay and cross-node performance clustering techniques to predict the performance of parallel applications for future systems [89].

In a broader sense, prior research on memory tracing and memory trace-based performance analysis supplements this work. As the next generation of ScalaTrace, ScalaTrace 2 continues to utilize the Regular Section Descriptors (RSDs) and the Power-RSDs (PRSDs) to describe the nested loop structures in trace (see Chapter 2). RSDs were originally proposed to track inter-procedural side effects on common substructures of arrays to promote compiler-aided parallelization [37]. Marathe et al. adapted the RSD representation and proposed PRSDs for memory trace compression [50, 49]. Budanur et al. further designed Extended-PRSD

to perform multi-level scalable parallel memory tracing in their work ScalaMemTrace [13]. SIGMA employs online trace compression to collect lossless memory traces for simulation and performance tuning [19]. Elnozahy et al. utilize loop detection and reduction for address trace compression [24]. VPC3 employs value predictors to compress events comprising program counter values and extended data fields [14].

5.5 Summary

Application tracing is one of the most important and useful vehicle for performance analysis and debugging of parallel applications. Yet, designing scalable and efficient parallel tracing tools for exascale systems and grand-challenge HPC applications remains an open problem. In this work, we contribute ScalaTrace 2, a fundamental redesign of ScalaTrace that features a spectrum of innovative lossless compression techniques aiming at scalable trace compression of large-scale scientific codes with irregular SPMD behavior or even MPMD characteristics. We designed an elastic data element representation to address compression inefficiencies of previous work. Enabled by the new encoding scheme, novel algorithms are devised to perform approximate loop matching and loop agnostic cross-node trace compression. We also incorporated the parameter histogram-based lossy compression capability into ScalaTrace 2 and adapted the replay subsystem to achieve a more accurate probabilistic replay. As the first step towards customizable instrumentation, ScalaTrace 2 also supports the collection of performance statistics with user-defined plugins. We evaluated ScalaTrace 2 with the NAS Parallel Benchmarks, Sweep3D, and a real-world application, the Parallel Ocean Program. Experimental results demonstrate that the redesigned trace compression algorithms are particularly effective for applications with inconsistent behavior across time steps and MPI tasks. In comparison to the prior research, we deem ScalaTrace 2 a solid improvement towards exascale performance analysis.

Chapter 6

Future Work

This decade is projected to usher in the period of exascale computing with the advent of systems featuring more than 500 million concurrent tasks. Harnessing such hardware with coordinated computing in software poses significant challenges. To facilitate exascale performance analysis, this work contributes novel techniques for communication tracing and trace-based analysis. In this chapter, we discuss additional directions that can be pursued in the future.

6.1 Customizable Instrumentation

To ensure tool scalability, it is important that one collects only the data needed for the diagnosis of a particular performance problem. ScalaTrace 2 is our first step towards a customizable tracing framework. It currently supports the collection of numerical performance data, such as execution times and hardware counter values, at the entry and exit points of communication and computational stages. To further improve the flexibility and granularity of the customizable instrumentation, we propose to improve our current work along two orthogonal directions: (1) the actual instrumentation of the applications or libraries and (2) its customization and dynamic selection. In addition to relying on the existing PMPI interface layer, we can also employ the binary instrumentation and interpositioning techniques to handle other libraries or sets of API functions, e.g., systems libraries like libc or libpthreads. Several relevant technologies exist that may be used to implement such a capability, including the Dyninst [12] dynamic instrumentation library, Pin [48], the Program Database Toolkit [46], and the GNU linker's capability to automatically interpose wrapper functions. We propose to investigate the scalability and performance properties of each solution and then integrate the best suited mechanism with ScalaTrace 2. Once a function call is intercepted, we can dynamically decide whether we record the observed event or not. Such filter predicates will be light weight and can be based on one of several conditions, including the function name (to select particular function

groups), the passed arguments (to extract only certain use cases, like large messages or short I/O write operations), the context in which the function was called (to separate callpaths and only record events caused by specific libraries), or simply time (to allow sampling techniques). This approach further allows us to control the instrumentation to trace only those events originating from a particular library. This not only reduces the data, but provides users with the ability to disambiguate overlapping message patterns, a common occurrence in many complex scientific codes. This ability to dynamically adjust the instrumentation provides us with the fine granularity required to control the data volume and to restrict the performance data to only the data required for the particular analysis targeted by the user.

6.2 A Versatile Tracing Framework with Tunable Precision

We propose to design a framework to create customized tools for trace extraction. Our approach generalizes existing concepts of trace compression so that they become applicable to arbitrary trace events, ranging from system calls over I/O, communication and memory references, to performance counters. Being a fully customizable tracing framework, we also propose to use event specification to guide trace compression. A trace event specification not only consists of the trace entry (e.g., name of a call, instruction type) but also of a parameter mask. This mask indicates which parameters should be captured and what compression approach should be applied to each.

ScalaTrace 2 focuses on systematically exploiting the structural properties in the sequences of events to achieve lossless or near-lossless trace compression. In the future, we propose to employ approximate tracing techniques to handle applications with drastically irregular event patterns that are not suitable for being traced in a lossless or near-lossless manner. Particularly, we propose to support user-specified precision levels for the lossy compression of not just the MPI parameters, but also the loop structures, the participant lists, and even the trace events. For example, event sequences themselves can be represented as histograms effectively indicating a probability for a data-dependent event to be issued by a given node. This recursive definition of histograms on the trace structure itself departs with conventional PRSDs in the sense that the PRSD becomes conditional (C-PRSD), which creates another dimension in our data representation. A C-PRSD also represents the analytical behavior of an application relative to its input data, and thus opens up the opportunity to unify data reduction and program behavior analysis. By pursuing multi-level approximate tracing, we propose to design a set of relevant lossy compression algorithms and to investigate the effectiveness of such a scheme compared to the current approaches.

6.3 Scalable Numerical Data Analysis Techniques

With the massive core count on a leadership-class supercomputer, even a carefully designed instrumentation strategy may add considerable volumes of numerical data to traces, which can lead to data volumes too high to record. We propose to solve this problem using prior work on in-situ clustering and in-situ wavelet analysis [31, 30, 29] to discover the relationships between performance data elements distributed across many processes. CAPEK [30] is a generic algorithm and has been shown to work well for simple data such as vector traces, simple points, and objects in well-defined, low-dimensional spaces. It is also known to scale to hundreds of thousands of processes. We propose to use CAPEK as a foundation for structurally clustering numerical trace data and structural trace elements. First, we propose to develop distance metrics that will enable CAPEK to be applied to traces in the ScalaTrace trace format. To accomplish this goal, we will need to deduce appropriate measures for "similarity" between structural trace components. We will develop graph comparison algorithms to compare behavioral patterns among similar processes, and we will develop error measures to quantify how well these measures differentiate actual program behavior. Next, we propose to design statistical approaches to perform root cause analysis for performance problems. For example, the performance data obtained with the custom instrumentation APIs and the collected trace will be used as inputs to two runs of CAPEK. With the representatives from the resulting equivalence classes, we can then perform root cause analysis by employing correlation analysis techniques [76, 18].

Chapter 7

Conclusion

With the advent of systems with millions of cores and exascale performance in the near future, performance analysis for parallel applications becomes increasingly important for the design and development of scientific applications, and for the construction and procurement of high-performance computing systems. Recent progress on scalable communication tracing opens up opportunities for novel performance analysis approaches. This work advances the scalable communication tracing techniques and contributes a spectrum of trace-based algorithms for performance analysis, prediction, and benchmarking at scale. To facilitate extreme-scale performance analysis and prediction, this work introduces ScalaExtrap. ScalaExtrap is a trace-based algorithm that automatically extrapolates traces at large scales from smaller traces for stencil and mesh style SPMD codes. It enables — for the first time — fast trace-based performance prediction and simulation at arbitrarily large scales, yet without requiring running the application at those scales. To bridge the gap between the performance realism of a complete application and the convenience of obtaining, porting and modifying a benchmark code, this work proposes a trace-based benchmark generation framework that automatically extracts performance-accurate communication benchmarks from parallel applications. By supporting application logic obfuscation, this code generation tool is particularly valuable for proprietary, export-controlled, or classified application codes. Moreover, this work also contributes ScalaTrace 2, the next generation ScalaTrace that targets inefficiencies in the compression of communication traces for applications with irregular SPMD behavior or even MPMD characteristics. By incorporating a spectrum of novel trace compression algorithms, such as elastic data element encoding, approximate loop matching, and loop agnostic inter-node trace compression, ScalaTrace 2 demonstrates key improvement in trace compression. With ScalaTrace 2, we bring our prior research to the next stage and prepare the ScalaTrace framework for exascale performance analysis.

Overall, by utilizing domain-specific knowledge on scientific computing and exploiting the unique features of ScalaTrace, this work puts forth fundamentally new methodologies for com-

munication trace compression and for trace-based performance analysis, prediction, and benchmarking at scale. Thereby, it validates the hypothesis of this dissertation.

REFERENCES

[1] Top 500 list. http://www.top500.org/, June 2011.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.

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

[4] ADIOS 1.3 user's manual. http://users.nccs.gov/ pnorbert/ADIOS-UsersManual-1.3.pdf.

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

[6] 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.

[7] D.H. Bailey and A. Snavely. Performance modeling: Understanding the present and predicting the future. In *Euro-Par Conference*, August 2005.

[8] R Bell and L. John. Improved automatic testcase synthesis for performance model validation. In *Int'l Conf. on Supercomputing*, pages 111–120, June 2005.

[9] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, pages 39–, Washington, DC, USA, 2000. IEEE Computer Society.

[10] H. Brunst, D. Kranzlmüller, and W. Nagel. Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. *The International Series in Engineering and Computer Science, Distributed and Parallel Systems*, 777:92–102, 2005.

[11] 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.

[12] Bryan Buck and Jeffrey Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[13] Sandeep Budanur, Frank Mueller, and Todd Gamblin. Memory trace compression and replay for SPMD systems using Extended PRSDs. *SIGMETRICS Perform. Eval. Rev.*, 38(4):30–36, March 2011.

[14] M. Burtscher. Vpc3: A fast and effective trace-compression algorithm. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 167–176, N.Y., June 2004.

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

[16] Community earth system model. http://www.cesm.ucar.edu/index.html.

[17] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. *SIGMETRICS Perform. Eval. Rev.*, 39(1):1–12, June 2011.

[18] Cristian Coarfa, John Mellor-Crummey, Nathan Froyd, and Yuri Dotsenko. Scalability analysis of spmd codes using expectations. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 13–22, New York, NY, USA, 2007. ACM.

[19] L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Supercomputing*, November 2002.

[20] Vivek Deshpande. Automatic Generation of Complete Communication Skeletons from Traces. Master's thesis, North Carolina State University, Raleigh, NC, USA, 2011.

[21] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[22] Z. Eckert and G. Nutt. Trace extrapolation for parallel programs on shared memory multiprocessors. Technical Report TR CU-CS-804-96, Department of Computer Science, University of Colorado at Boulder, Boulder, CO, 1996.

[23] Zulah K. F. Eckert and Gary J. Nutt. Parallel program trace extrapolation. In *International Conference on Parallel Processing*, pages 103–107, 1994.

[24] E. N. Elnozahy. Address trace compression through loop detection and reduction. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '99, pages 214–215, New York, NY, USA, 1999. ACM.

[25] 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.

[26] Extrae. http://www.bsc.es/computer-sciences/extrae.

[27] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. A study of process arrival patterns for MPI collective operations. In *Int'l Conf. on Supercomputing*, June 2007.

[28] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 81–90, New York, NY, USA, 2005. ACM.

[29] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Rob Fowler, and Daniel A. Reed. Scalable load-balance measurement for spmd codes. In *Supercomputing*, pages 1–12, 2008.

[30] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Rob Fowler, and Daniel A. Reed. Clustering performance data efficiently at massive scales. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 243–252, New York, NY, USA, 2010. ACM.

[31] Todd Gamblin, Robert J. Fowler, and Daniel A. Reed. Scalable methods for monitoring and detecting behavioral equivalence classes in scientific codes. In *IPDPS*, pages 1–12, 2008.

[32] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr. The scalasca performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing*, June 2008.

[33] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Symposium on Compiler Construction*, pages 276–283, June 1982.

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

[35] B. Gruber, G. Haring, D. Kranzlmueller, and J. Volkert. Parallel programming with capse – a case study. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:0130, 1996.

[36] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[37] 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.

[38] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 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*, pages 195–206, 2006.

[39] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing*, November 2001.

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

[41] 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.

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

[43] Sriram Krishnamoorthy and Khushbu Agarwal. Scalable communication trace compression. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 408–417, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Jesús Labarta, Sergi Girona, and Toni Cortes. Analyzing scheduling policies using dimemas. *Parallel Computing*, 23(1-2):23–34, 1997.

[45] James R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5):259–269, May 1999.

[46] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Bernd Mohr, Reid Rivenburgh, and Craig Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing*, pages 68–68, 2000.

[47] Jeremy Logan, Scott Klasky, Jay F. Lofstead, Hasan Abbasi, Stéphane Ethier, Ray W. Grout, Seung-Hoe Ku, Qing Liu, Xiaosong Ma, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. Skel: Generative software for producing skeletal i/o applications. In *e-Science Workshops*, pages 191–198, 2011.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[49] J. Marathe, F. Mueller, and B. R. de Supinski. A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks. In *Int'l Conf. on Supercomputing*, pages 21–30, June 2005.

[50] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *International Symposium on Code Generation and Optimization*, pages 289–300, March 2003.

[51] J. Marathe, F. Mueller, T. Mohan, S. A. McKee, B. R. de Supinski, and A. Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems*, 29(2):1–36, April 2007.

[52] 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.

[53] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.

[54] Craig G. Nevill-Manning and Ian H. Witten. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the Conference on Data Compression*, DCC '97, pages 3–, Washington, DC, USA, 1997. IEEE Computer Society.

[55] 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.

[56] 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.

[57] Scott Pakin. Reproducible network benchmarks with coNCePTuaL. In Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, editors, *Proceedings of the 10th International Euro-Par Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 64–71, August 31–September 3, 2004.

[58] 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.

[59] Vincent Pillet, Vincent Pillet, Jesus Labarta, Toni Cortes, and Sergi Girona. PARAVER: A tool to visualize and analyze parallel code. In *Proceedings of the 18th Technical Meeting of WoTUG-18: Transputer and Occam Developments*, pages 17–31, 1995.

[60] Vincent Pillet, Vincent Pillet, Jess Labarta, Toni Cortes, Toni Cortes, Sergi Girona, Sergi Girona, and Departament D'arquitectura De Computadors. Paraver: A tool to visualize and analyze parallel code. Technical report, In WoTUG-18, 1995.

[61] The parallel ocean program (POP), 1996. http://climate.lanl.gov/Models/POP/.

[62] Robert Preissl, Thomas Köckerbauer, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Detecting patterns in mpi communication traces. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 230–237, Washington, DC, USA, 2008. IEEE Computer Society.

[63] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. Supinski, and Daniel J. Quinlan. Using mpi communication patterns to guide source code transformations. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*, pages 253–260, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[65] Arun F Rodrigues, Richard C Murphy, Peter Kogge, and Keith D Underwood. The structural simulation toolkit: exploring novel architectures. In *Poster at the 2006 ACM/IEEE Conference on Supercomputing*, page 157, 2006.

[66] M.A. Ronsse and D.A. Kranzlmueller. Roltmp-replay of lamport timestamps for message passing systems. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:0087, 1998.

[67] Subhash Saini, Dale Talcott, Dennis Jespersen, Jahed Djomehri, Haoqiang Jin, and Rupak Biswas. Scientific application-based performance comparison of sgi altix 4700, ibm power5+, and sgi ice 8200 supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 7:1–7:12, Piscataway, NJ, USA, 2008. IEEE Press.

[68] 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.

[69] 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.

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

[71] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[72] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[73] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing*, November 2002.

[74] K. Sreenivasan and A. J. Kleinman. On the construction of a representative synthetic workload. *Commun. ACM*, 17(3):127–133, March 1974.

[75] 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.

[76] Jeffrey S. Vetter and Michael O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 123–132, New York, NY, USA, 2001. ACM.

[77] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable i/o tracing and analysis. In *Workshop on Petascale Data Storage*, pages 26–31, 2009.

[78] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, New Orleans, Louisiana, November 13–19, 2010.

[79] 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.

[80] 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.

[81] W.S. Wong and R.J.T. Morris. Benchmark synthesis using the lru cache hit function. *Computers, IEEE Transactions on*, 37(6):637 –645, jun 1988.

[82] Xing Wu, Vivek Deshpande, and Frank Mueller. ScalaBenchGen: Auto-generation of communication benchmarks traces. In *IPDPS*, pages 1250–1260, 2012.

[83] 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.

[84] Xing Wu, Frank Mueller, and Scott Pakin. Automatic generation of executable communication specifications from parallel applications. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 12–21, New York, NY, USA, 2011. ACM.

[85] Xing Wu, Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Probabilistic communication and i/o tracing with deterministic replay at scale. In *ICPP*, pages 196–205, 2011.

[86] 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.

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

[88] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey Voelker. Mpiwiz: subgroup reproducible replay of mpi applications. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 251–260, New York, NY, USA, 2009. ACM.

[89] J. Zhai, W. Chen, and W. Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, 2010.

[90] 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.