ABSTRACT

VIJAYAKUMAR, KARTHIK. Probabilistic Communication and I/O Tracing with Deterministic Replay at Scale. (Under the direction of Dr. Frank Mueller and Dr. Xiaosong Ma.)

As supercomputer performance approached and surpassed the peta-flop level, concerns about low application efficiency have led application developers to rely increasingly on analysis tools to diagnose problems with the performance and scalability of their codes. Root cause analysis for performance problems often identifies poor communication and I/O performance as a major problem for many scientific applications. Several tools exist to collect communication and I/O traces to assist in such root cause analysis. However, these tools either produce extremely large trace files that complicate performance analysis, or sacrifice accuracy to collect high-level statistical information using crude averaging.

This work contributes Scala-H-Trace, a multi-level event tracing tool that collects communication and I/O traces at several levels in the High Performance Computing (HPC) I/O stack and also features new techniques for more aggressive trace compression than any previous approach, particularly for applications that do not show strict regularity in Single Program Multiple Data (SPMD) behavior.

Such code characteristics are increasingly common due to data-dependent communication, I/O and internal dynamic computational rebalancing such as adaptive mesh refinement (AMR), that require redistribution of data via communication. Past approaches fail to provide scalable tracing and experience rapid increases in traces sizes for such scenarios. Scala-H-Trace uses histograms expressing the probabilistic distribution of arbitrary communication and I/O parameters to capture variations. Yet, where other tools fail to scale, Scala-H-Trace guarantees trace files of near constant size, even for variable communication and I/O patterns, producing trace files orders of magnitudes smaller than using prior approaches. We demonstrate the ability to collect traces of applications running on thousands of processors with the potential to scale well beyond this level.

Aggressively compressed traces create significant challenges to accurately replay traced events for application analysis. We present the first approach to deterministically replay such probabilistic traces (a) without deadlocks and (b) in a manner closely resembling the original applications. We embrace the challenge of lack of knowledge to match senders with their receivers due to the loss of information in histograms but compensate with a distributed, orchestrated replay that does not require back-channel communication to preserve causal event ordering for correctness. Our contributions also include automated trace analysis to collect selected statistical information of I/O calls by parsing the compressed trace on-the-fly.

We evaluated our approach with the Parallel Ocean Program (POP) climate simulation

code, the FLASH parallel I/O benchmark and two benchmarks, CG and MG, from the NAS suite. Our results show either near constant sized traces or only sub-linear increases in trace file sizes for POP, FLASH I/O benchmark and CG, irrespective of the number of nodes utilized. Even with the aggressively compressed histogram-based traces, our replay times are within 12% to 15% of the runtime of original codes in most cases. Statistical information gathered via the automated trace analysis reveals insight on the number of I/O and communication calls issued in POP and FLASH I/O. Such concise traces resembling the behavior of production-style codes closely and our approach of deterministic replay of probabilistic traces are without precedence.

Probabilistic Communication and I/O Tracing with Deterministic Replay at Scale

by
Karthik Vijayakumar

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

Computer Science

Raleigh, North Carolina

2010

APPROVED BY:

Dr. Xiaosong Ma Co-chair of Advisory Committee	Dr. Xuxian Jiang Member of Committee

Dr. Frank Mueller
Co-chair of Advisory Committee

DEDICATION

To my entire family.
Amba, aunt Jothi and uncle Mahesh.

BIOGRAPHY

Karthik Vijayakumar was born on March 22, 1983 to Mrs. P.V.Umadevi and Mr. P.S.Vijayakumar in Madurai (Tamilnadu, India) and grew up in Chennai, India. He received his Bachelor of Technology degree in 2004 from Madras Institute of Technology, which is a constituent college of Anna University, Chennai, India. He then worked at Infosys Technologies Limited as a Programmer Analyst for four years.

Karthik joined the Masters program in the department of Computer Science at North Carolina State University in Fall 2008. Since then, he has been working with Dr. Frank Mueller and Dr. Xiaosong Ma. With the defense of this thesis, he is receiving Master of Science degree in Computer Science from NCSU, in December 2010.

ACKNOWLEDGEMENTS

I would like to thank my advisors Dr. Frank Mueller and Dr. Xiaosong Ma for their support, guidance and faith in me. I am very grateful towards Philip C. Roth for letting me work at the Oak Ridge National Laboratory in the summer of 2009 and guiding me throughout my thesis work. I would like to thank Dr. Xuxian Jiang for serving on my advisory committee. This research used resources of the National Center for Computational Sciences(NCCS) at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725 and I would like to thank NCCS for providing the access to their resources. I would also like to thank Abhik, Chris, Manav, Amey and my other labmates for their help and inputs. I am also very grateful towards my uncle Mahesh and aunt Jothi for supporting me throughout my degree program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Supercomputers already surpassed petaflop level performance and their system design is becoming increasingly complex year after year. These computers are used extensively in the research and development for simulations in various domains like physics, chemistry and nuclear engineering. Many countries around the world spend millions of dollars every year in constructing faster supercomputers for the advancement in research. The supercomputing community even ranks supercomputers based on their performance. As of June 2010, ORNL's Jaguar is rated as the fastest supercomputer in the world on Top500 [4] website. Many scientific applications are developed to harness these system resources. For example, Streitz *et al.* [22] studied solidification process in metals at high pressure using supercomputers. The simulations performed in supercomputers enabled them to obtain totally different insight in the solidification process, which otherwise would not have been possible to obtain.

Most scientific simulations perform complex computations utilizing thousands of cores available in supercomputers. One important reason for using these systems is to achieve good speedup in the overall application execution time. These applications utilize parallelization techniques to obtain very high speedup and utilize system resources quite effectively. Major performance gains are obtained by splitting large simulation input data into small chunks across multiple compute cores and by working on different data sets in parallel. These small chunks from different compute cores are later combined to get the desired result. One important factor that affects speedup in such applications is that the communication involved across cores to transfer intermediate simulation results. These intermediate results are then used by neighboring cores for their local calculation.

Another major factor affecting speedup is due to the file input/output (I/O) involved in scientific applications. Since simulations run for tens of hours and also for multiple days, these applications take checkpoints of current execution context at regular time intervals. In the event of system failure in the middle of application execution, these regular checkpoints

help in restarting application runs from that point of time instead of restarting the entire run. Hundreds of nodes write these checkpoint files simultaneously, clearly affecting the entire execution time. A significant amount of work is being carried out by the research community on improving the I/O efficiency for scientific applications utilizing supercomputers, which lead to the development of parallel I/O technologies. Several improvements have been made in parallel I/O technology to improve the overall I/O processing time. Current supercomputers employ parallel file systems and also use separate I/O nodes connected to compute cores using special high speed networks. Compute cores then transfer all I/O data to I/O nodes, which writes into the underlying file systems.

1.1 Motivation

As supercomputers progress in scale and capability toward exascale levels, characterization of communication and I/O behavior is becoming increasingly difficult due to system size and complexity. Today, many scientific applications execute in ten thousands of cores or more. Moreover, modern supercomputers are equipped with complex network interconnects to improve the speedup of parallel applications. Apart from the network complexity, different vendors employ different interconnect designs to improve the overall communication performance, thereby achieving better speedup. For example, the IBM Blue Gene family of supercomputers employs five different network interconnects [5]. Such interconnects mandate performance study of applications for efficient use of available resources. Even finding the most efficient task mapping to nodes has become difficult with complex, new system designs.

The large numbers of processors/cores, increased aggregate memory capacity, and optimized interconnects allow applications to grow not only in targeting larger problem sizes, but to explore more sophisticated communication models. Extreme-scale applications are often complex codes, integrating multiple software components exercising vastly different computation/communication models. Such codes are becoming more dynamic and diverging from strict, regular single program, multiple data (SPMD) behavior. Examples include multi-physics or coupled codes, where partitions of nodes implement different simulation models, work on separate datasets, or even conduct analytics tasks such as data reduction. Such applications exhibit multiple program, multiple data (MPMD) behavior as multiple nodes work on multiple sections of the program. *E.g.*, in climate simulations, some nodes simulate climate changes over land and while other nodes work on sea models. Hence, different modules, like land and sea, use different input data and algorithms resulting in different communication behavior within each module. Other examples include Adaptive Mesh Refinement (AMR) codes, where the data set is dynamically re-constructed (refined) and periodically re-balanced.

Apart from complex communication behavior, these applications possess I/O behavior with

different complexities due to the multi-level layering of I/O modules and architectural variations in storage systems as shown in Figure 1.1. Although these layers provide essential abstractions to the end user by hiding complex implementation details behind simpler interfaces, these details ultimately impact the actual behavior of any I/O calls issued at high-level layers. These abstractions may also hide any inherent performance bottleneck caused by poor implementation at lower layers, such as low-level synchronization issues.

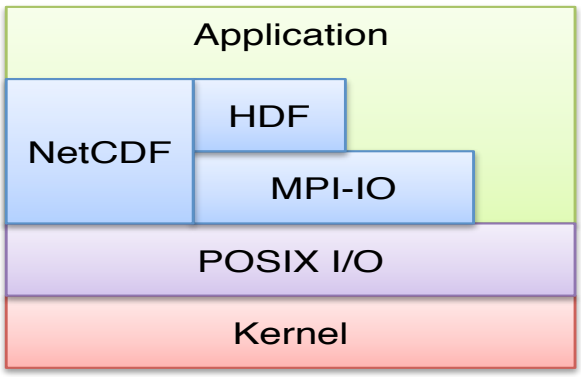

Figure 1.1: Typical Compute Node I/O Stack

Several studies have investigated the communication and I/O characteristics of applications. They focus on three main classes: tracing tools, capable of capturing and recording all message events at the cost of high storage requirements; profiling tools, designed to provide low-overhead performance summaries trading off storage space for detail level; and communication and I/O kernels that eliminate computation and retain only application communication and I/O behavior. Although application kernels are designed to capture the exact application behavior, it is difficult to keep these kernels up-to-date since the applications constantly evolve over time. Application traces, in contrast, can be readily generated by simple instrumentation of an application, to keep up with a changing code base. This makes performance analysis via traces a preferred method to analyze parallel applications in practice.

The combination of job scale and application complexity, however, creates unique challenges for parallel tracing tools. On one end of the spectrum, traditional tracing tools (such as Vampir [15]) record all events sequentially for each parallel process. For large application runs on leadership-class supercomputers, this approach generates unmanageable trace file sizes, introducing prohibitive overheads, *e.g.*, for copying trace files from temporary to permanent storage, hitting the maximum storage limit, and even the need for a cluster plus another parallelized tool to perform trace analysis [6]. On the other end of the spectrum, tools that only

report statistical information (such as mpiP [23]) may fail to deliver the level of detail needed in performance analysis or debugging.

1.2 Our Approach

In this work, we propose Scala-H-Trace, which provides two novel capabilities to collect concise communication and I/O traces. First, we provide the capability to collect I/O traces at multiple levels of the I/O stack. Next, we provide a unique histogram-based trace collection approach for applications exhibiting *non-SPMD* behavior. Past approaches [17, 20] utilize lossless on-the-fly trace compression techniques to collect communication traces and dramatically reduce trace file sizes. However, effective compression builds on the homogeneous behavior across processes (inter-node compression) and repetitive behavior within a process (intra-node compression). With complex, irregular, or self-adjusting applications, such assumptions do not hold and compression fails due to mismatches between traced events. In other words, while past approaches proved effective for the easier problems of tracing SPMD codes, this work focuses on the much harder problems of tracing communication and I/O calls for non-SPMD codes.

Scala-H-Trace is motivated by the tradeoff between exact details and manageability of trace file size. Although having exact details helps in root cause analysis, lossless tracing becomes increasingly unaffordable on ultra-scale machines. *Histograms* in Scala-H-Trace provide an opportunity to collect overall statistical details, *e.g.*, data send volume, which can be useful in studying network characteristics of the application. They provide an overall "big picture" of an application's communication behavior. Scala-H-Trace employs histograms with multiple bins whose value ranges are dynamically adapted as trace data is recorded on-the-fly. In addition to representing a distribution, each bin also retains certain crude statistical information (min/max/avg/stddev), potentially useful for root cause analysis.

Scala-H-Trace also enables the user to set a *precision level* during trace collection. This precision level drives the compression efficiency by collecting statistical information for varying traces in unique histogram bins. The trace precision is also ensured to fall within the user set value. If the trace precision falls below the specified precision, mismatching trace events are recorded without histogram-based compression, *i.e.*, traditional structural compression techniques are employed and may fail to provide concise traces in the absence of SPMD behavior of the code [17] but result in exact event recording. The size of such a trace file then becomes a function of the desired precision level, which can be tuned to obtain a manageable size while retaining trace artifacts suitable for performance analysis or even detect root causes in performance. At the same time, our unique approach to collect histogram-based statistical information captures the overall trend in communication and I/O behavior of applications executing on thousands of nodes.

While histogram-based tracing can effectively reduce trace data volumes, it creates several challenges for accurate replay of the traced events. To ensure the correctness of the captured trace and to reproduce the communication and I/O behavior, we also provide a novel replay facility. This replay tool reissues the recorded trace events without decompressing the compressed trace. If the compressed trace is lossless, sender-destination node information and communication volume are recorded precisely. Also, the causal ordering of the original application is preserved. For lossy, histogram-based traces, our tool employs a distributed, orchestrated and deterministic replay capability.

1.3 Hypothesis

We contend that, using the above mentioned approaches, it is possible to collect communication and I/O traces of either near constant size trace files or trace files with only a sub-linear increase in size for applications that exhibit not only perfect SPMD behavior but also non-SPMD behavior. We also hypothesize that histogram-based trace compression techniques suffice to capture the communication and I/O characteristics of applications close to the original behavior and will assess this claim by accuracy of replay times relative to the original application.

1.4 Contributions

Our contributions are as follows:

- We provide the capability to record lossless and concise communication and I/O traces for non-SPMD programs.
- We provide the capability to record traces at several layers of abstraction in the software stack of system and library I/O interfaces;
- We create novel capabilities for more aggressive trace compression based on a precision level, selected by the user, that drives both compression efficiency and trace accuracy.
- We support a replay technique that reissues trace events without decompressing the original trace file.
- we create novel capabilities to automate trace analysis for collecting statistical information from the traces;
- We provide a distributed approach to deterministically replay statistical traces that does not require back-channel communication to preserve causal event ordering for correctness.

We evaluated our approach with the Parallel Ocean Program (POP), Flash I/O benchmark and two benchmarks from NAS parallel benchmark suite. Our results provide one to two orders of magnitude smaller trace files than any previous approach. We also evaluated our replay tool by reissuing histogram-based traced events. The replay time only deviated 12% to 15% from the original application's time in most cases, even for most aggressively merged histogram-based traces.

Chapter 2

Background

Scala-H-Trace is a novel design of a communication and I/O tracing tool that shares its methodology for representing the resulting trace on a single file (instead of one file per node), both otherwise relies on histogram-based data collection. While Scala-H-Trace was derived from the publicly available code of ScalaTrace [17, 20], Scala-H-Trace provides entirely novel compression capabilities and also provides capabilities to collect I/O traces at multiple levels of the I/O software stack.

ScalaTrace collects communication traces using the MPI Profiling layer (PMPI) [14] through Umpire [24] to intercept MPI calls and to collect MPI traces. It features aggressive trace compression that generates a single, concise and lossless trace file from any large-scale parallel application run. It also preserves timing information in the compressed form along with the calling context of events being traced. In this paper, we develop Scala-H-Trace, which provides even more aggressive trace compression techniques to serve real-world scientific applications that do not show strict SPMD regularity.

ScalaTrace performs two types of compression: *intra-node* and *inter-node*. The former exploits the repetitive nature of timestep simulation in parallel scientific applications. The latter exploits the homogeneity in behavior (SPMD) among different processes running the application. Intra-node compression is performed on-the-fly within a node. Inter-node compression is performed across nodes by forming a radix tree structure among all nodes and sending all intra-node compressed traces to respective parents in the radix tree. This results in a single compressed trace file capturing the entire application run across all nodes. The compression algorithm is discussed in detail elsewhere [17, 20]. Scala-H-Trace employs a different intra- and inter-node compression algorithm due to its reliance on histograms but still shares the reduction over a radix tree with ScalaTrace.

2.1 Trace Compression

We briefly introduce several techniques used in ScalaTrace to allow a later comparison with Scala-H-Trace. Repetitive events in different iteration of loops are collected as Regular Section Descriptors (RSD) [9] and power-RSDs capture RSD events in nested loops [13], both of which are represented in constant size. Consider the following code snippet:

```
for( i = 0; i < 10; i++ ) {
   for( j = 0; j < 100; j++) {
      compute1();
      MPI_Irecv(...); // Receive from left neighbor
      MPI_Isend(...); //Send to right neighbor
      MPI_Waitall(...);
   }
   MPI_Allreduce(...); //Collective reduction operation
}
```

Trace compression results in the following tuples: RSD1:{100, MPI_Irecv, MPI_Isend, MPI_-Waitall} representing 100 iterations of MPI_Irecv, MPI_Isend and MPI_Waitall in the inner loop, PRSD1: {10, RSD1, MPI_Allreduce} denoting 10 iterations RSD1 followed by MPI_Allreduce in the outermost loop. The algorithm uses the calling context of events to match repetitious behavior. This ensures that identical MPI functions originating from different call paths of the application are not compressed together. Since trace events from different nodes are collected and merged in a single output trace file, the *task rank* information of nodes participating in an event is also compressed and encoded concisely in the compressed trace. This participant node information is represented in a tuple containing starting rank, total number of participants and an offset value separating ranks. Even multi-dimensional information is captured in this encoding format.

Apart from matching calling contexts, the compression algorithm matches function parameters and merges them along with compressed events ensuring that no information is lost. In typical parallel applications, communication end-points differ across nodes as a result of communication with neighboring nodes. These varying end-points inhibit event compression. ScalaTrace uses a unique location-independent encoding to represent communication end-points in events like MPI_Send and MPI_Recv. It also encodes MPI opaque pointers like MPI_File and MPI_Comm, which do not exhibit repetitive patterns, potentially inhibiting effective compression. There are special cases in which events with matching calling context can have non-matching function parameters. These non-matching function parameters are compressed using a vector representation, so that the particular event can be concisely represented in the

trace.

2.2 Time Preservation

Another important feature of ScalaTrace is the time preservation of captured traces. Instead of recording absolute timestamps, the tool records delta time of computation durations between adjacent communication calls. During RSD formation, instead of accumulating exact delta timestamps, statistical histogram bins are utilized to concisely represent timing details across the loop. These bins are comprised of statistical timing data (minimum, maximum, average and standard deviation). More details on collecting statistical timing information are provided elsewhere [20].

2.3 Timed Replay

ScalaTrace not only supports scalable tracing, it also supports a scalable replay engine. Given a single, compressed trace file, the replay engine allows all communication calls to be reissued without trace decompression while preserving event ordering. The replay engine runs as an MPI job with the same number of tasks as its original application. It replays communication events in each node with their original parameters except for actual file content/message payloads. Instead, a random buffer of the same size as the original file/message buffer is used. Additionally, computation time on each node is simulated by a delay between traced events based on recorded delta time.

Chapter 3

Multilevel I/O Trace Collection

File I/O operations in large-scale scientific applications typically go through a multi-level software stack, such as the one depicted in Figure 1.1. A parallel application often performs file I/O through high-level scientific data format libraries, such as HDF5 [2] and netCDF [3], where shared file access capabilities may be built on general-purpose parallel I/O libraries such as MPI-IO. Applications may also directly use MPI-IO interfaces to perform parallel I/O, where the I/O operations are passed to the parallel file system clients running on each compute node. Eventually, I/O calls are made through the system I/O libraries.

With such an increasingly deep I/O stack, I/O performance depends not just on application access behavior, but also on the interaction between different abstraction layers. It is important to isolate an application's behavior at a certain level, or to correlate activities at multiple levels. With our prototype system, we make initial effort to expose these layers and enable the analysis of multi-level traces in a scalable way, to better understand I/O behavior of an application on a specific architecture.

In our implementation, Scala-H-Trace collects traces of MPI-IO and low-level POSIX I/O function calls. Both MPI-IO and POSIX calls represent the high and low levels of the I/O software stack. Scala-H-Trace can, of course, be extended further to collect traces at any layer. Figure 3.1 below shows different modules assisting in trace collection.

3.1 MPI-IO Trace Generation

At the surface, the methodology to collect MPI-IO traces resembles trace collection of MPI communication calls. Scala-H-Trace contains an interposition engine based on the Umpire tool [24] to automatically create wrappers for trace events from a specification of the corresponding instrumentation actions. The wrapper engine generates modules that assist in trace collection (header files, a wrapper file containing all MPI function overrides, and lookup files containing

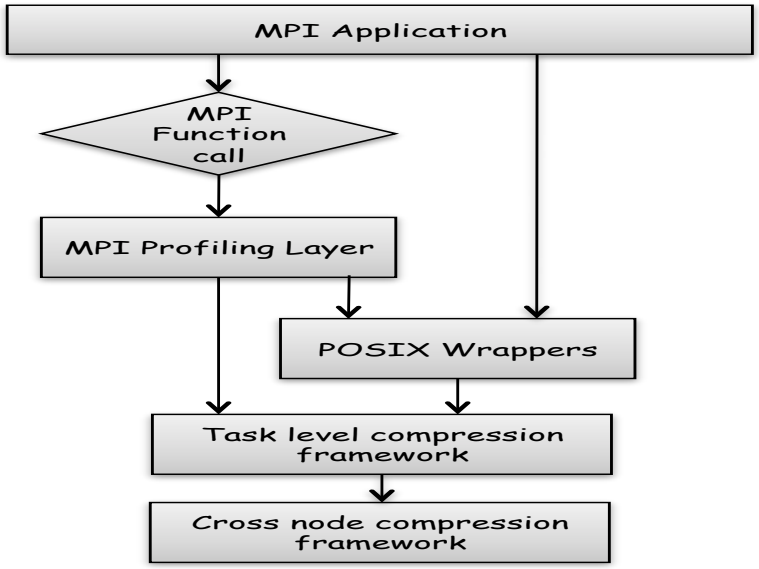

Figure 3.1: Scala-H-Trace Design

details of MPI functions used internally).

However, certain I/O function parameters, such as file name, offset and MPI_File opaque objects, require special handling to achieve scalable trace compression, as detailed below.

Regarding file names, we consider several widely used approaches for performing periodic I/O in parallel applications. In many applications, all processes send output data to a root process (process 0), which then performs I/O. Alternatively, all processes may use parallel I/O to write one or more shared checkpoint/snapshot files, either with collective or individual I/O calls. In a third and currently less common approach, each process creates its own output file. The checkpoint files and/or snapshot files are written periodically, typically once per c (where c is some constant) timesteps, identified by a timestep number in the file name. In case separate files are created per process, files are typically differentiated by encoding the process/node rank in file names. For efficient intra- and inter-node trace compression, Scala-H-Trace parses the file names to identify the "static" and "dynamic" component strings. For example, file names checkpoint-001-0.nc, checkpoint-001-1.nc, etc. will be recognized as having static components of "checkpoint-001-" and ".nc". During compression, file names from disjoint nodes are merged into a single event if the static file name components match. Process ranks can be substituted by RSDs, which are expanded during replay (see below).

Similarly, Scala-H-Trace needs to identify and merge parallel MPI-IO accesses to shared files across I/O timesteps and across processes. For example, assume each of 10 nodes writes a disjoint range of 1000 bytes in a shared file, *i.e.* node 0 accesses the byte range of [0,999], node

1 accesses [1000,1999], etc. Scala-H-Trace encodes such access pattern into three fields (start position, stride, and the total number of elements) during intra- and inter-node compression, so that multiple file accesses with the same call stack will be compressed into a single event.

Finally, MPI_File handles representing file objects are opaque pointers handled internally by the MPI library and do not exhibit repetitive patterns. Scala-H-Trace stores these handles in a buffer, added upon the file open operation. Subsequent accesses to open files are recorded by referencing the corresponding handle's offset in the buffer rather than the handle itself. This allows us to compress and replay the I/O traces appropriately.

Apart from MPI-IO calls, Scala-H-Trace provides support to record traces for creating custom data types, such as MPI_Type_create_darray, which are widely used in collective file accesses. These custom data type handlers are also opaque pointers and are treated in the same manner as file handlers.

3.2 POSIX I/O Trace Generation

Scala-H-Trace also collects traces from POSIX I/O calls. Compared to MPI-IO, POSIX I/O calls belong to the lower level in the I/O stack and potentially can provide more information on the actual requests made to the parallel file system. Tracing POSIX calls can help in identifying performance bottlenecks in the middle and lower I/O stack layers, as well as in capturing I/O activities that do not go through a higher-level I/O library. Many of Scala-H-Trace's techniques in MPI-IO trace collection, compression, and replay can be applied to POSIX I/O as well. We discuss several POSIX-specific design and implementation issues below.

We exploited GNU linker's link time function interposition facility to provide instrumentation for POSIX I/O calls and to collect traces. The "–wrap" option enables function calls, such as open, write, etc., to be redirected to corresponding interposition functions (*e.g.*, __wrap_open()). The interposition wrappers implement trace collection and call the corresponding native (actual) function (*e.g.*, __real_open()). Scala-H-Trace provides a separate library for POSIX wrappers, which, together with the Scala-H-Trace library, is statically linked with the application using the link switch "–wrap" to signify which I/O functions are interposed.

Most of the function parameters in POSIX calls are similar to those of MPI-IO calls (*e.g.*, file name, number of bytes read/written). During our initial testing with Scala-H-Trace, we discovered that several files had been opened even before application execution and thus prior to our I/O inter-positioning. These activities are accessing the internally managed resources and sockets opened by the MPI runtime system. We observed many I/O calls on these files during the initialization phase to coordinate application execution and system activity. Since these I/O calls were outside the application scope, we filtered them out by recording the traces with the files opened only after MPI_Init.

Chapter 4

Histogram Based Trace Collection

Noeth *et al.* [17] provide trace compression techniques resulting in an almost constant sized trace file or sublinear increases in trace file size with strong scaling (increasing number of nodes). Yet, these results only hold for SPMD-style benchmarks, not for production size applications with non-SPMD patterns. ScalaIOTrace [25] provides mechanisms to collect both the communication and I/O traces for scientific applications like the Parallel Ocean Program (POP) [19]. But for some scientific applications, including POP, the inter-node compression technique fails to obtain a near-constant sized trace file with increasing number of nodes. Instead, we see a linear increase in the trace size due to non-SPMD style programming.

POP performs ocean simulation for multiple time steps. Each time step performs a set of computations and communications of an inner loop in multiple iterations. Due to different data-dependent convergence points in the computation across different timesteps, the number of inner loop iterations varies from timestep to timestep. Even though all MPI events originate from the same calling sequence (call stack), varying loop iteration counts in each timestep inhibit intra-node compression and thus negatively impact inter-node compression across all nodes. 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.

To address these problems, we propose a novel method of tracing. We promote histogram-based trace information for a predefined user-tunable precision level to obtain higher compression rates of trace events — at the expense of accuracy. For example, if the user sets the precision level to 95%, then events with matching calling contexts but with non-matching function parameters will be merged if and only if that parameter differs at most a margin of 5%. Should the parameters differ by more than 5%, we fall back to lossless tracing. This provides an opportunity for users to work with a significantly smaller trace file than other previous techniques.

Our approach uses histograms to collect probabilistic information on varying trace events

and event parameters that otherwise inhibit trace compression. Histogram-based collection employs a technique to collect statistical information in dynamically balanced bins. The online balancing algorithm equalizes the number of items per bin while adjusting their value range constraints. This ensures that the histogram captures outliers and other statistical distribution properties missed by simple aggregate statistical collection like maximum, minimum, average and variance. We also collect maximum and minimum participant rank information along with the frequency in bins so as to enable root cause detection, *e.g.*, due to load imbalance. Even with this lossy trace information, histograms help in providing more insight into the general characteristics of the traced application. Histogram details can be collected at various levels in the trace. The following explains what trace information is collected as histogram and discusses possible tradeoffs in collecting statistical information versus non-lossy information.

4.1 Intra-node Event Histogram

The loop iteration count denoted by PRSDs can be collected as a histogram. This enables better compression of repeating events in many scientific applications that otherwise would fail to compress due to data dependencies. Although the exact iteration count is lost in the final trace, the number of loop iterations directly depends on the computation, which, in turn, varies with different input sets. Hence, collecting statistical loop iteration counts only has a minor impact in capturing the communication behavior of the application. The main advantage of this approach is the ability to obtain a concise trace file by allowing a small percentage of lossy trace collection that otherwise would have resulted in a trace file of unmanageable size.

Consider the code snippet:

```
for( i = 0; i < 50; i++ ) {
   //Perform calculation till the result converges
   while(result > convergence_factor) {
        do_calculation();
        MPI_Irecv(...); //Receive from neighbors
        MPI_Send(...); //Send to neighbors
        MPI_Wait(...);
   }
}
```

In the above example, if the iteration count matches across time-steps, the resulting PRSD will be of form PRSD1:{50, RSD1}. Due to mismatching convergence points across different time-steps, the following sample events can occur:

```
RSD1: <39, MPI_Irecv, MPI_Send, MPI_Wait>
RSD2: <40, MPI_Irecv, MPI_Send, MPI_Wait>
```

```
RSD3: <39, MPI_Irecv, MPI_Send, MPI_Wait>
RSD4: <42, MPI_Irecv, MPI_Send, MPI_Wait>
.... till RSD50
```

The expected PRSD is not formed due to mismatching RSDs across time steps. This leads to cascading compression failures across nodes. As a result, the trace file size increases linearly with the increase in participating nodes. Histogram-based trace collection ensures that the varying iteration count is captured in histogram bins. Hence, the resulting trace will have just one PRSD for the entire time-step calculation.

4.2 Inter-node Event Histogram

With inter-node event compression, compressed traces from different nodes are merged together. During this process, a radix tree structure is formed among all nodes. Child nodes send their respective intra-node compressed traces to their parents. A parent node performs compression of matching events from its child nodes. For each and every event from the parent node, a matching child event is searched. If there is a match, the parent event's participant list is updated with the rank of the child node and the child event is discarded. Other unmatched events will be reordered according to its dependency with other events.

In applications with non-SPMD behavior, loops created during intra-node compression can have matching events across nodes, but fail to compress across nodes due to a mismatch in the loop iteration count. This prevents the entire loop from being merged, increasing the trace file size linearly with the number of nodes.

As an example, consider the code snippet from the Section 4.1 again. Table 4.1 shows one such scenario in which computation dependent loop iterations fail to merge across nodes. By collecting loop iterations in histogram, all events merge successfully across nodes. Note that we enable merge only when all events inside the loop match perfectly. If events from two loop candidates do not match, then these loops are considered to represent *different* scenarios in the original application and are hence not merged.

Table 4.1: Varying Loop Iteration

Participants:0-3	Participants:4-7
Loop 50 times	Loop 51 times
MPI_Irecv from 0-3	MPI_Irecv from 4-7
MPI_Send to 0-3	MPI_Send to 4-7
MPI_Wait	MPI_Wait

4.3 Function Parameter Histogram

Apart from collecting loop iteration counts in histograms, MPI function parameters, such as Send/Recv volume, tag and sender/destination ranks, can also be recorded in histograms. The Send/Recv data volume is important to capture the network load due to the communication calls issued by the application. Send/Recv volumes often vary across different timesteps in AMR applications. This variation in volume inhibits compression of communication calls originating from the same call stack, thereby inhibiting compression across an entire loop due to a small deviation in the data volume parameters. There are other methods to collect exact volumes. One such method is to collect the volume information in a vector along with the rank information of participating nodes. But this results in a linear growth with the increase in number of participating nodes, which is non-scalable.

For applications that do not exhibit a regular communication pattern, it is impossible to compress repeating communication events originating from the same call stack with different sender/destination ranks. The approach of location-independent relative encoding of communication end-points provides a novel opportunity for event compression. But even this approach only succeeds in the case of applications with regular communication patterns. There are approaches in which the communication function call can be expressed as a PRSD but different end-points in different loop iterations have to be collected as a vector. Again, such an approach is not scalable for applications executed on thousands of nodes. An example of collecting function parameters as a vector is given below:

```
Loop iterations: 1 - 5
Participants: 0 - 9 (node ranks)
Event: MPI_Send
Data volume: 90 bytes [ranks: 0,1,4,5,8,9],
             92 bytes [ranks: 2,3,6,7]
Destination: relative rank 1 [ranks: 0,2,4,6,8],
             relative rank 9 [ranks: 1,3,5,7,9]
......
```

Assume MPI_Send is executed in a loop 5 times. With lossless trace collection, both data volume and destination will be recorded along with the rank information of the corresponding participant. The relative ranks shown above is location independent: 1 represents "the next right neighbor" and 9 represents "the next left neighbor". This compression results in a more concise representation than its uncompressed equivalent, but it still suffers from increases in the trace size proportional to the number of nodes since no regularity for rank lists could be deduced.

Using histograms to collect relative end-points and data volume allows better compression

of repeating events originating from the same call stack. For this example, histograms will record both destinations 1 and 9 in bins along with its frequency. In addition to binning communication end-points, we also collect relative ranks in a bitmap and encode it in the trace file. This provides information on exact values that are missing from the histograms and aids post-mortem analysis tools. In the above example, an analysis tool may choose relative ranks of either 1 or 9 while relative ranks between 2 and 8 are excluded from the pseudo-random selection. We reiterate that we provide this lossy trace collection as a feature and the decision to use this feature is entirely upon the users. Users may choose to enable histogram-based tracing and configure the precision level in response to their application analysis needs, overheads and storage availability.

4.4 Histogram Construction

We have designed our system in a way to collect exact trace information as much as possible. Users can set a target precision level expressed as a percentage. Our compression algorithm attempts to match events originating from the same call stack and compresses events only if all function parameters match. Histogram collection is triggered only if there is a mismatch in function parameters or in the loop information. In such cases, the difference between two non-matching values is checked in terms of the user specified precision level. If the difference is within the target precision range, then events are merged and the non-matching parameters are recorded in a histogram from there on. If the difference falls out of the target precision range, then either event compression will fail or data is recorded in a vector as shown in the example in 4.3.

In our current implementation, the number of histogram bins is fixed at the start of the application run, but the value ranges in bins are dynamically adjusted. We provide an option to set an interval after which bins are adjusted. Two bins with the lowest frequencies are combined and the bin with maximum frequency is split into two bins. We further store auxiliary information in bins, such as minimum/maximum/average/variance, which are adjusted accordingly. Apart from per-bin statistical information, we also collect maximum/minimum values over the entire value range (all bins) and the node ranks associated with those. This provides outlier information and can be used in the replay studies and other performance analysis tools.

Chapter 5

Deterministic Replay and Trace Analysis

5.1 Deterministic Replay

While histogram-based trace collection is powerful in compressing irregular or dynamically changing events, the collected traces themselves create challenges for replaying and subsequent performance analysis. The core challenge of histogram-based replay is to ensure that events are issued in a deterministic manner across nodes and with coordinated parameter value selections for common communication end-points of sends and receives. Since Scala-H-Trace collects statistical values for communication volume, tags, and end-points, the conventional ScalaTrace replay design for lossless traces, which takes an independent, uncoordinated approach among nodes, can lead to potential deadlocks due to statistical uncertainty, or may fail to re-create the original communication or I/O pattern with reasonable proximity. The nature of histogram-based traces mandates a distributed, orchestrated replay with coordination among all participating nodes to ensure deterministic event sequences during replay for Scala-H-Trace. All nodes must read all trace events. They need to agree on a specific value selected from the statistical information found in the trace.

We have fundamentally redesigned the replay tool [17] to reissue MPI calls from lossless traces such that the trace data collected using histograms is honored during event replay. Our replay tool issues MPI calls using the compressed trace independent of the original application and without decompressing the trace. This tool verifies the correctness of the collected trace. It can also assist in the performance tuning of MPI communication and facilitates projections of network requirements for future procurements. Apart from replaying MPI calls, this basic replay framework can also be extended to integrate with other performance analysis/tuning tools and it can be used to perform automated communication and network metric calculations.

Before we discuss the design of our new Scala-H-Trace replay tool, we first review the conventional design of replay for lossless traces in ScalaTrace [17]. For lossless traces, all participating nodes parse the trace file and *only act on events if the current node is a member of the participant list*. Then all nodes reissue MPI events one by one by identifying loops using the PRSD information and extracting individual MPI function parameters from the recorded trace. This replay tool also reads the delta time information from the trace and simulates the computation time by sleeping in place of computation. This simulates the exact communication and I/O behavior of the application in terms of interconnect characteristics, such as contention. The replay tool helps to verify the correctness of the trace. By design, it ensures absence of deadlocks if the original application did not have any deadlocks for a given trace. Replay also preserves the time taken in terms of the original application's runtime.

5.1.1 Scala-H-Trace Replay

With the histogram-based trace, the existing parallel replay functionality requires a complete overhaul to cope with statistical data instead of precise data. In our Scala-H-Trace approach, all participating nodes parse the entire trace file during replay. In contrast to ScalaTrace, *all nodes read and interpret* all MPI events. Such interpretation amounts to the selection of a random value following the histogram distribution of any recorded events, for each node in the trace. All nodes "know" the random values used by the other nodes. However, a given node only issues MPI calls if the current node is a member of the participant list in the recorded trace. The interpretation of histogram values for events that are not issued is crucial to provide efficient replay with histograms: It obviates a need to coordinate value selection across nodes and, hence, back-channel communication that might otherwise be required due to randomization, as discussed after the next paragraph.

During the random selection of replay parameters, end-points of MPI_Send/MPI_Isend events are selected. Upon encountering an MPI_Send, once a node identifies itself as a receiver, the receiver node issues a receive call (MPI_Irecv) instead of a MPI_Send. Hence, all receive communication events like MPI_Recv and MPI_Irecv are ignored. Since a particular receiver can also be a sender, only MPI_Irecv calls are internally issued followed by an internal MPI_Wait call when a node rank identifies itself as a receiver of a recorded MPI_Send event. Such internal MPI_Wait calls are issued last, after all ranks have been parsed and all MPI_Send/MPI_Irecv calls have been issued. Any MPI_Wait/Waitall calls in the original recorded trace are ignored.

The selection of a random value for histogram-recorded parameter for any event parameters, such as send/destination rank and data volume, requires that sending and receiving nodes make the same decision on matching end-points for a message exchanged between them. To ensure

that sender and receiver nodes agree on their end-points for a message exchange, all nodes use the same random seed during initialization. Hence, all nodes agree on the random value upon each selection of a replay parameter within the range of 0 and total number of elements in the histogram. No coordination via communication is required as all nodes interpret all events in the same order, even if only a subset of (one or more) nodes actually issues an MPI call.

The selected random value is internally used to select an appropriate bin. The average value recorded for that bin is then chosen as a parameter for the MPI event. This approach ensures that the value selected from the histogram is uniformly distributed to replicate the original application behavior as closely as possible. The following section discusses distributed coordination for random selection in more detail.

5.1.2 Challenges for Deterministic Replay: Point-to-Point Messages

The following code snippet is an example of climate simulation in which first 50 nodes work on land simulation and the next 50 nodes work on sea simulation. These simulations are performed in multiple time steps in which nodes perform calculations and communicate the result to surrounding neighbors. The destination nodes and communication volume can vary for land and sea simulation.

```
//Land simulation participants - Node 0 to 49
//Sea simulation participants - Node 50-99

int * resultbuf; //Buffer to hold results
for(timesteps from 0 to 100) {
  int destination[100]; //Array to hold dest. ranks
  int source[100]; //Array to hold source ranks
  do_calculations(resultbuf);
  if(simulation == land) {
     volume = 80 bytes;
     get_my_land_neighbors(destination, source);
  else {
     volume = 90 bytes;
     get_my_sea_neighbors(destination, source);
  }
  for ( i = 0 to total_neighbors_count ) {
     MPI_Irecv (resultbuf,volume,source[i],...);
     MPI_Isend(resultbuf,volume,destination[i],...);
     MPI_Waitall (...); //Wait for Irecv
     MPI_Waitall (...); //Wait for Isend
```

```
  }
}
```

In the code above, all participating nodes perform calculations and communicate the results to corresponding neighbors. All MPI function calls originate from the same call stack but communication volume and source/destination endpoints vary across nodes. This results in perfectly compressed intra-node traces with the following events:

```
RSD1: {MPI_Irecv, MPI_Isend, MPI_Waitall, MPI_Waitall}
PRSD1: {total_neighbors_count, RSD1}
PRSD2: {total_timesteps, PRSD1}
```

Since communication volume and endpoints vary across nodes, the inter-node compression fails for the above section. With an appropriate user-specified precision level, communication volume and endpoints are collected in histogram bins during inter-node compression and the trace is compressed across all nodes. Hence, all nodes need to agree upon the message payload (data volume) and send/receive endpoints during replay. With such an agreement between nodes for the selection of a particular value for replay, potential deadlocks could occur.

For example, an original send/receive pair for sender (node 1) / receiver (node 2) might result in arbitrary selection of communication end-points without our distributed coordination scheme. In other words, the sender (node 10) may issue a message to node 20, both randomly selected on node 10, while node 20 interprets the send event as a message originating from node 13 and directed at node 23 per uncoordinated random selection. In such a case, node 10 would deadlock as the message is never received. Similarly, receives (or waits for completion of receives) may deadlock if no corresponding send is ever issued.

Our distributed, coordinated approach to randomized selection ensures that all nodes interpret the send event as originating from node 10 and being directed at node 20. While node 10 issues a send (and node 20 a receive), all other nodes (13 and 23) will not issue any MPI call. The fact that the original event was a message from node 1 to 2 is probabilistically replayed as a message from one node (here: 10) to another (here: 20), *i.e.*, histograms do result in randomized end-points but retain the original number of messages for the example.

5.1.3 Challenges for Deterministic Replay: Collective Communication

With the coordinated replay approach, there are situations in which deadlocks can occur due to causal ordering of uncompressed traces. Consider the sample trace below with 8 nodes:

Both columns in Table 5.1 contain an uncompressed sequence of MPI events originating from the same call stack. Each MPI call is preceded by a sequence number as recorded by intra-node compression. The first set of nodes, 0 to 3, issues 2 sets of MPI_Irecv/Send/Wait calls followed by a MPI_Bcast. The second set of nodes, 4 to 7, issues only one MPI_Irecv/Send/Wait

Table 5.1: Uncompressed Trace

Participants:0-3	Participants:4-7
110-111: MPI_-Irecv(2 iterations) 1st. iter: from 0-3, 2nd iter: from 4-7	130: MPI_Irecv from 0-3
112-113: MPI_-Send(2 iterations) 1st. iter to 0-3, 2nd iter: to 4-7	131: MPI_Send to 0-3
114: MPI_Wait (2 counts)	132: MPI_Wait
115: MPI_Bcast	133: MPI_Bcast

followed by MPI_Bcast. These events fail to compress due to mismatching Send/Recv counts across different sets of nodes. This results in the final trace with events 110-115 followed by events 130-133.

With the coordinated compression of Scala-H-Trace and its corresponding replay, the MPI_-Send in sequence 112 will be issued and the corresponding MPI_Irecv will be issued internally by respective destination ranks as shown in the sample trace. (MPI_Irecv/Wait are ignored during histogram-based replay.) Next, MPI_Bcast will be issued by ranks 0 to 3. This will block ranks 0 to 3 until the corresponding MPI_Bcast (seq. 133) is issued by ranks 4-7. But before issuing the broadcast in sequence 133, nodes with ranks 4-7 issue the MPI_Send in sequence 131 with destination 0-3. Since nodes of ranks 0-3 are already blocked in MPI_Bcast (seq. 115), they cannot issue an corresponding internal MPI_Irecv, eventually leading to a situation in which nodes 4-7 cannot proceed to other events. This situation occurs frequently. In many scientific applications, two sets of nodes can execute different sections of a program leading to compression failure interspersed with collectives, such as barriers. This causal ordering of events in the trace can lead to deadlock when replayed using the above approach. We employ a novel design for the inter-node compression algorithm to forcibly merge collectives even if an entire PRSD loop of other events does not merge properly.

Inter-node compression attempts to match an entire sequence of events subject to the same PRSD loop across nodes. Even if there is a single mismatch, the entire sequence would conventionally not be merged but rather be written consecutively as shown in the sample trace above. We employ a novel design for inter-node compression to greedily merge any subset of events, *e.g.*, collectives inside a loop. We then rearrange other communication calls with collectives as

synchronization points. This ensures that deadlocks cannot be introduced during the replay of MPI events.

We next prove that our novel merge algorithm, which rearranges non-merging communication calls with a collective as a synchronization point, will not introduce deadlocks. We assume that the original application is deadlock free (which is reasonable since the event trace was collected from a terminating application) and provide the proof below:

Theorem 5.1.1 *A replayed trace of a program with events reorder to synchronized collectives does not result in deadlock if the original trace was deadlock free.*

Proof Follows from Lemmas 1-3.

Lemma 5.1.2 *Lemma 1: A replayed trace of a program with only collectives will not deadlock.*

Proof By construction of traces, all recorded participants engage in a collective during replay in the same order as recorded. Collectives are blocking. Hence, all participants complete a collective at (nearly) the same time (as collectives provide global/group synchronization).

Lemma 5.1.3 *A replayed trace of a program with only point-to-point communication will not deadlock.*

Proof Blocking/non-blocking point-to-point calls are replayed in the same order as recorded. Hence, if the original trace did not deadlock, replayed point-to-point messages in the same order will not deadlock either.

Lemma 5.1.4 *A replayed traces with mixed events of collectives and point-to-point messages will not deadlock.*

Proof (a) If collectives provide a fence where all point-to-point messages are consumed prior to a collective, a trace has alternating phases of only point-to-point messages and only collectives. Replaying such a trace is deadlock free for each region by Lemmas 1 and 2 and thereby also for the entire trace since each section is causally independent.

(b) If point-to-point messages are crossing collectives (sent before but received after a collective across a pair of nodes that also participates in the collective), then the same send will be issued during replay before the corresponding collective, and the corresponding receives will be issued before the collective. Hence, if the application with its traced events was deadlock free, the replay will also be deadlock free.

(c) By structural induction over (b), replays of traces with (i) multiple send/receive pairs crossing a collective, (ii) a send/receive pair crossing multiple collectives and (iii) a combination of (i) and (ii) will not deadlock if the original application did not deadlock during trace recording.

(d) Single/multiple event pairs crossing a sequence of collectives and point-to-point messages that do not cross collectives will not deadlock during replay due to structural induction over (b) plus Lemma 1 and 2.

5.2 Trace Analysis

Based on the specialized replay facility discussed in 5.1, we designed a generic and novel automated post-mortem trace analysis framework. Our design provides generic event handlers for all recorded trace functions. These handles can then be interposed or substituted by user-specific code when the trace is traversed in our generic analysis engine. While trace analysis could also be performed in a more conventional manner during application execution with interposed events within Scala-H-Trace, post-mortem generic trace analysis provides several advantages.

Conventional trace analysis requires a priori knowledge about performance problems in order to collect and correlate the subset of I/O and communication events that may contribute to performance problems or application characterization footprints. Short of such a prior knowledge, conventional trace analysis is generally repeated with different refinement steps, which requires a separate application execution each time. In contrast, our generic post-mortem trace analysis facilitates the detection of anomalies or identification of communication patterns in the applications by iterative refinement without re-running the application. Instead, different trace analysis interposition functions at the replay level operate directly on compressed traces, and by omitting time-accurate replay, can be traversed rapidly taking only a fraction of the original application's execution time.

In this work, we exploit generic trace replay to demonstrate its capabilities in one specialization example. By providing aggregation interposition functionality, we obtain statistical details on the number of I/O operations and collective/blocking/non-blocking communication calls across all nodes. This could easily be refined for group-specific analysis of MPI communicators or subsets of traced events originating from certain code sections based on per-call stack backtrace information.

Chapter 6

Experimental Results

We evaluated Scala-H-Trace in four aspects: (1) its effectiveness of trace file compression, (2) its statistical trace replay feature, (3) its trace compression sensitivity to precision level settings, and (4) its capability to collect statistical information on I/O and communication activities via replaying the compressed traces. Experiments (1) and (2) utilize forced histograms at a precision level of 0%, which is discussed in more detail in the context of experiment (3).

Our experiments were conducted on Jaguar, the Cray XT4 system at ORNL. Each of compute node features a 2.1 GHz quad-core AMD Opteron 1354 processor and 8GB of DDR2 memory. The login nodes run a full-featured Linux version while the compute nodes run the Compute Node Linux microkernel.

We analyze the efficacy of Scala-H-Trace using a production-scale application, the Parallel Ocean Program (POP) [10], as the main challenge. The Parallel Ocean Program (POP) is an ocean circulation model developed at Los Alamos National Laboratory. Our experiments exercise a one degree grid resolution in which the problem size is 320x384 blocks and the individual block size is 5x6 resulting in a total of 4096 (64x64) blocks distributed to individual nodes. POP exhibits non-SPMD behavior, which leads to trace file size increases with the number of nodes for conventional trace tools, including ScalaTrace. Hence, this application provides an opportunity to show-case the effectiveness of histogram-based trace collection of Scala-H-Trace. We conducted experiments by varying the maximum number of blocks assigned to each node.

We further utilize the CG and MG benchmarks from the NAS parallel benchmark suite for inputs sizes C to study the efficacy of Scala-H-Trace for different types of application behavior. Both CG and MG mostly exhibit SPMD behavior but differ significantly in the communication pattern impacting the compression effectiveness during trace collection. These benchmarks are also selected from the NAS benchmarks as these two were the challenging cases for ScalaTrace's lossless compression: Both were reported to results in sub-linear increases in the trace file size for

ScalaTrace [17]. We also utilized the Flash I/O benchmark, which simulates the I/O behavior of the FLASH application, to check the effectiveness of I/O trace compression. The FLASH application is a block-structured adaptive mesh hydrodynamics code [1].

6.1 Trace Compression Effectiveness

We collected traces based on two different compression techniques. First, lossless trace compression featuring ScalaTrace is used, in which all events are recorded with exact loop details and function parameters captured. Second, our novel histogram-based trace compression featuring Scala-H-Trace is used, in which trace information is collected in histograms for events and parameters that otherwise would not have compressed with the lossless trace compression. Trace file sizes are assessed under strong scaling, where we vary the number of nodes while keeping the overall problem size fixed. Lossless traces, obtained from ScalaTrace, are useful to identify exact details of the communication and I/O patterns exhibited by the application. Histogram-based traces are obtained from Scala-H-Trace, attempting to capture lossless information for trace events where feasible while non-matching events are recorded in histogram bins. We hypothesize that histograms suffice to capture the "big picture" of the application behavior and will assess this claim by accuracy of replay times relative to the original application. For applications exhibiting non-SPMD behavior, such as POP, histogram-based trace collection (Scala-H-Trace) collects concise traces, which could not otherwise be obtained with lossless trace compression (ScalaTrace).

First, we will discuss the effectiveness of compression for both communication and I/O calls for the FLASH I/O benchmark. Figure 6.1 depicts the size of trace files generated by two approaches, uncompressed flat traces and lossless inter-node compressed traces, over increasing number of nodes, both on log scale. The size of the uncompressed trace files grows linearly with increasing number of nodes. The reason for this behavior is that each node writes its own trace file and the number of files created grows with the increase in the number of nodes. In contrast, the size of the inter-node compressed traces is almost constant under strong scaling. This behavior of perfect inter-node compression is attributed to the SPMD programming style in the Flash I/O benchmark without any data-dependent conditional statements. FLASH I/O does not contain any loops. Hence, intra-node compressed traces are similar to flat traces and thus omitted in the results. FLASH I/O depicts the best case of perfect SPMD behavior, in which case even the earlier lossless trace compression resulted in trace file of constant file size.

Figure 6.2 depicts the trace file size for both lossless and histogram-based traces when varying the number of nodes. Note that the y axis is in log scale. Since POP exhibits non-SPMD behavior, we observe a linear increase in the trace file size in the case of lossless trace collection up to 256 nodes. The trace file size then stabilizes for 512 nodes and even declines

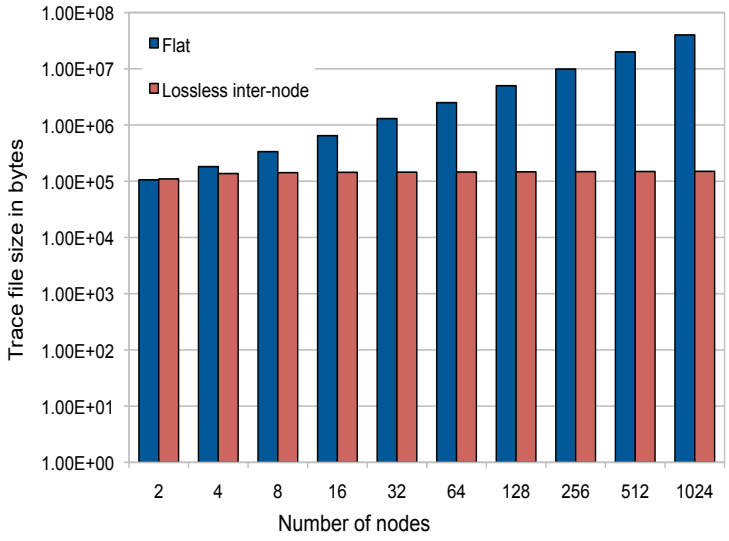

Figure 6.1: FLASH I/O Benchmark

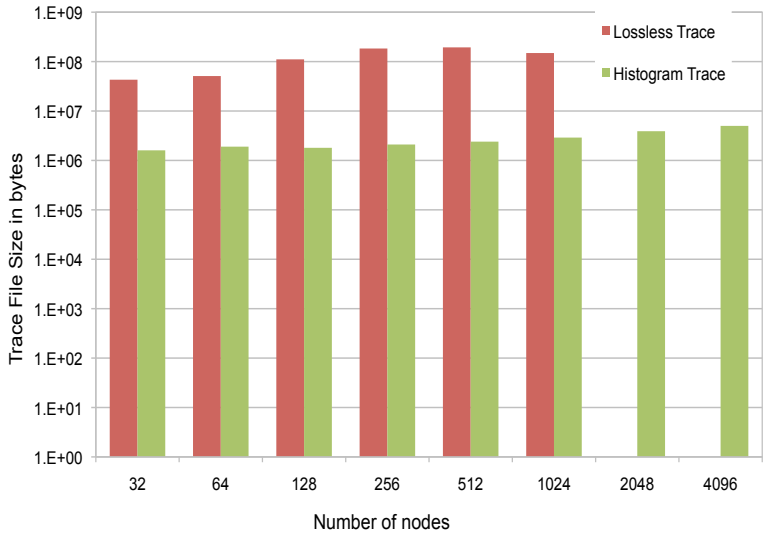

Figure 6.2: Parallel Ocean Program

for 1024 nodes. We identified that the timestep behavior becomes more regular at these levels, resulting in more effective inter-node compression. But we again observed an increasing trend in the case of 2048 nodes. For 2048 nodes and above, we could not even collect traces anymore as the trace file size was growing unmanageably fast and the time taken to merge hundreds of megabytes of per-node traces became prohibitive. With the histogram-based approach, there is a sub-linear increase in the trace file size. Moreover, histogram-based trace files are two orders of magnitude smaller than the lossless traces. This considerable reduction is obtained by aggressive compression of events and their associated function parameters in histograms. This clearly shows the efficacy of Scala-H-Trace to collect concise trace files even with applications exhibiting irregular behavior.

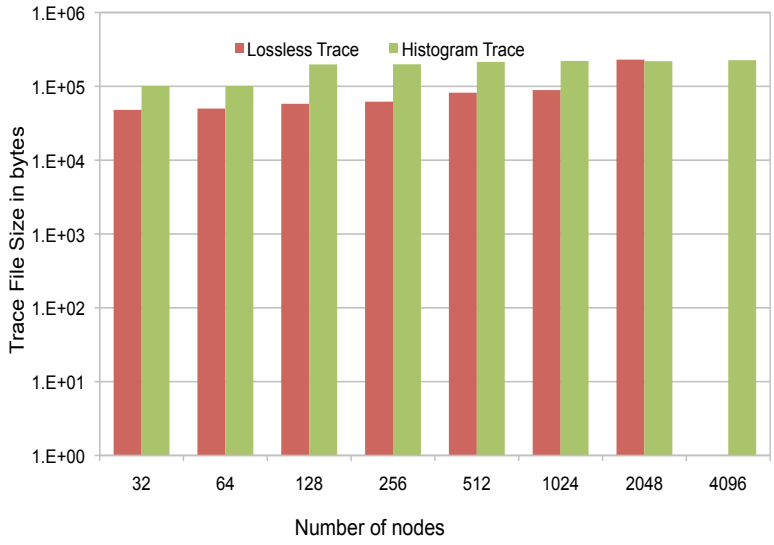

Figure 6.3: CG Benchmark

Figures 6.3 depict trace file sizes for the CG benchmark. We observe an interesting trend in CG in which the trace file size for lossless traces is consistently 50% less than that of the histogram traces up to 1024 nodes, yet sizes match at 2048 node. Even though lossless traces are initially smaller than histogram traces, there is a consistent increase in the trace file size for the lossless case. In contrast, the size of histogram traces is almost constant with the increase in number of nodes. For lossless traces, non-matching function parameters for events with the same call stack are collected in vectors associated with a participant rank list. This representation is more concise than histograms for smaller number of nodes. With thousands of nodes, the vector-participant list pair for each event has increased in size to where it is at

par with histogram traces. Unlike vector-participant lists, histogram representation is constant with the increase in number of nodes as the number of bins is fixed during the application run and even the outlier participant rank information is absorbed as constants in bins. It should also be noted that the trace file size for CG is in the order of hundreds of kilobyte. For larger applications with a similar communication behavior as CG yet with trace file sizes in hundreds of megabytes, such a linear (or even sub-linear) growth for lossless traces may simply not be scalable due to inter-node merge overheads, as discussed.

Figure 6.4: MG Benchmark

Figure 6.4 depicts the results for MG. Both lossless and histogram traces grow linearly in size with the increasing number of nodes. But the file size of histogram traces is always 50% to 60% smaller than that of lossless traces. The linear increase in trace file size is attributed to failure in inter-node compression across different subsets of nodes. While analyzing the trace file, we discovered that loops identified during intra-node compression differ across individual nodes. An explanation for such differences is that boundary nodes in communication patterns may exhibit different loop identification than internal nodes of the pattern for MG. Our inter-node compression algorithm does not compress loops with different events even if there is a matching loop iteration count.

6.2 Histogram-based Trace Replay

We studied the replay effectiveness of histogram-based traces by comparing the original application execution time with the time taken to replay the recorded events. We discuss the effectiveness of the distributed approach of replaying statistical traces. We also discuss the impact of trace compression on the replay behavior for histogram-based traces. We show that even with statistical histogram-based traces, replay can still be employed to check the correctness of a recorded trace and also to perform "what-if" analysis for system procurements.

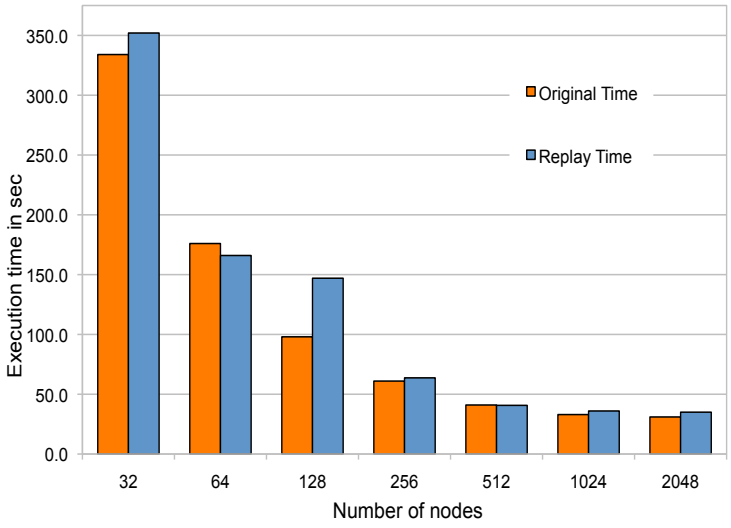

Figure 6.5: POP Replay

Figure 6.5 depicts the replay time of histogram-based trace events compared to that of the application's original execution time. The compressed traces are fully forced histogram trace events where any non-matching function parameters or loop iterations are collected as histograms. Even with these traces, we see that the replay time for traces collected for 32-512 number of nodes are within 5% of the original execution time (with the exception of replay time for 128 nodes). Replay time accuracy drops to 12% for 1024 and 2048 nodes. Due to our experiment with strong scaling for POP, the original execution time for both 1024 and 2048 nodes (30 seconds) is much lower than that for fewer nodes (>100 seconds) so that even small deviations in absolute values during replay increase the error percentage. We conjecture that such deviations are unrealistic as POP for this particular input does not scale beyond 512 nodes so that such short times are unrealistic. Similarly, this problem would not occur under weak

scaling as runtimes would not decrease with larger number of nodes. Overall, we observe that the replay time is close to the original execution time, even for fully forced histograms, due to two reasons: (1) Since our histograms are dynamically balanced, a random value selected from histogram bins during replay falls within a commonly used value range in the original application run. (2) The inter-node compression algorithm effectively merged events across nodes so that communication calls are not split in the trace file. The impact due to ineffective inter-node compression is discussed below for MG benchmark.

We observe nearly 50% deviation in the case of 128 nodes for POP. To investigate the cause, we calculated the time spent by nodes in other communication calls and found that some nodes are engaged in more communication calls than the majority of nodes. This created load imbalances where the remaining nodes wait at collectives for nodes participating in larger number of events. This type of behavior was rare for POP but more pronouncedly evident for the MG benchmark. The reason for this replay time deviation is discussed in detail in the context of MG below.

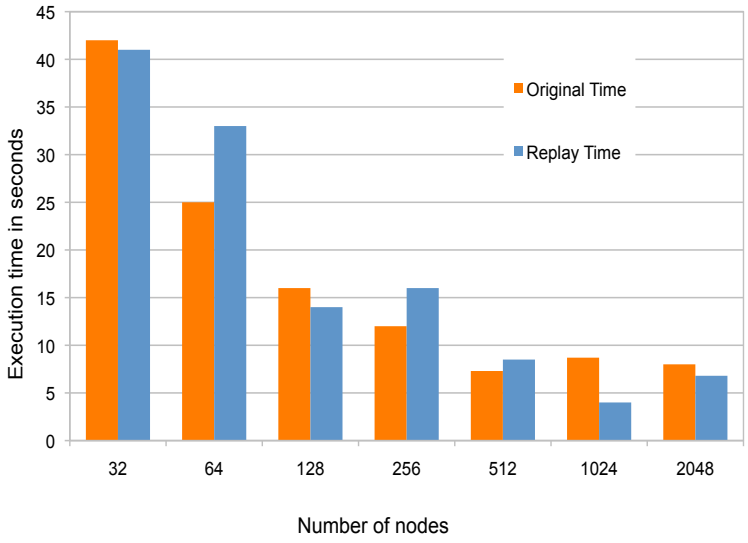

Figure 6.6: CG Replay

Figure 6.6 depicts the replay time for the CG benchmark. In the majority of cases, the replay time is with 10% to 15% of the original application runtime. Since the original execution time for CG is within 10 seconds for 1024 and 2048 nodes, even small changes in the absolute replay execution time increase the error percentage considerably. The replay time deviation can be attributed to the loop iterations recorded in histograms. Again, CG stops scaling at

512 nodes for this input size so that larger application runs are unrealistic. Furthermore, if the random loop iteration selected from histograms is close to the maximum value, then all nodes participate in more communication calls than in the original application. This is a fundamental trade-off between accuracy and manageability of trace file sizes.

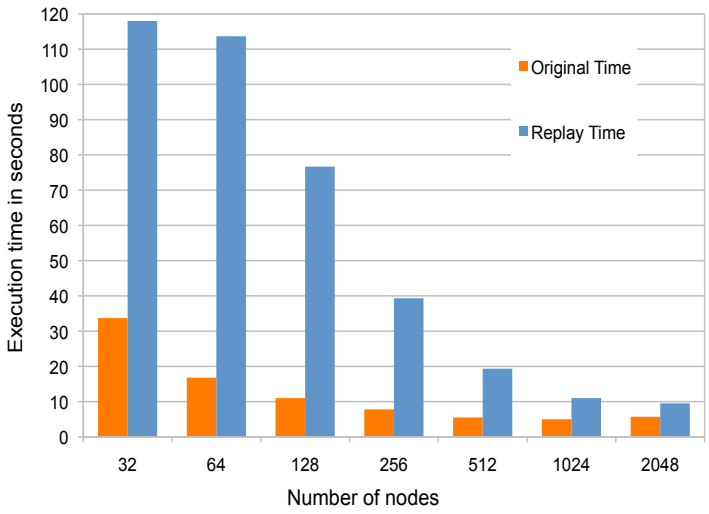

Figure 6.7: MG Replay

The replay time for MG benchmark is depicted in Figure 6.7. We observe that the replay time deviates by a factor of 3 to 4 from the original application runtime. This issue is similar to the deviation in POP for 128 nodes, but the behavior is more pronounced in MG. The reason for the big deviation is two fold: First, with imperfect inter-node compressed traces, many nodes coordinating one Send/Recv call in the original applications will be split into multiple unmerged events in the recorded trace file. With our new replay approach of all nodes interpreting every MPI_Send call, these unmerged events introduce increased synchronization. This results in some nodes having to wait for other nodes to join in a particular Send event. Second, MG uses a 7-point stencil communication pattern in a 3-dimensional space. With this type of communication pattern, nodes on the boundaries behave differently than the internal nodes. Even though we record relative communication end-points in histograms, we lose details such as which boundary node communicates with a particular end-point.

Since Scala-H-Trace's replay picks a random value from end-point histograms, boundary nodes may be selected in communication calls that would otherwise not have been on boundaries in the original application. Such additional communication with boundary nodes also increases

the time to synchronize with other nodes. This behavior is more prominently seen for lower number of nodes where the majority of nodes act as boundary nodes. With thousands of nodes, the ratio of boundary to internal nodes is lower. This is the primary reason for the exponential decrease in replay time deviation with an increasing number of nodes.

We plan to address these challenges in the future by enhancing Scala-H-Trace's replay tool. Instead of all nodes interpreting every trace event, nodes will only need to interpret events they actually participate in for lossless trace portions. The histogram-based events will still be interpreted by all nodes for the sake of distributed coordination, yet events are reissued only by randomly chosen participants (if the participant list was a histogram). We can also increase the precision level for MG to collect more exact traces. This would eventually improve the replay accuracy when combined with the proposed replay enhancements. The increase in precision level should only have a minor impact in the trace file size for MG as events do not even compress properly with the most aggressive forced histogram approach.

Exception for MG, as discussed, replay for Scala-H-Trace generated traces with forced histograms results in runtimes that are within 12-15% of the original application for most cases. This result is interesting as forced histograms are equivalent to a 0% precision level, which is the most aggressive compression possible with Scala-H-Trace. More accurate replay may result from higher precision levels at the cost of slightly larger traces, as discussed next.

6.3 Trace Sensitivity Study

Next, we study the effect of varying trace precision levels on trace file sizes. This experiment serves as an illustration for the benefits of user-specified precision levels as a means to steer compression, which should improve as precision decreases. Precision levels provide a tunable parameter to select target trace file size as required by operating environments or performance analysis experiments. Lossless traces may be desirable for exact analysis of application behavior and users with access to excessive storage may happily utilize this feature even if the trace file size becomes large. When desiring a more compact trace file and when inter-node merges become prohibitive for lossless traces, users can decrease trace precision to target a desired trace file size and tracing overhead.

Figure 6.8 depicts the impact of verifying trace precision levels on the final trace file size. We fixed the number of nodes to 512 for POP and measured trace file sizes for varying precision levels. We observe that even with a small decrease in the precision from 100% to 95%, the trace size reduced by more than a factor of three. This significant reduction is due to merging events with varying numbers of loop iterations for the timestep in POP. With lossless traces, two different sets of loops with the exact same events fail to compress due to varying numbers of loop iterations across the timestep. This variation is data dependent and induced by computation as

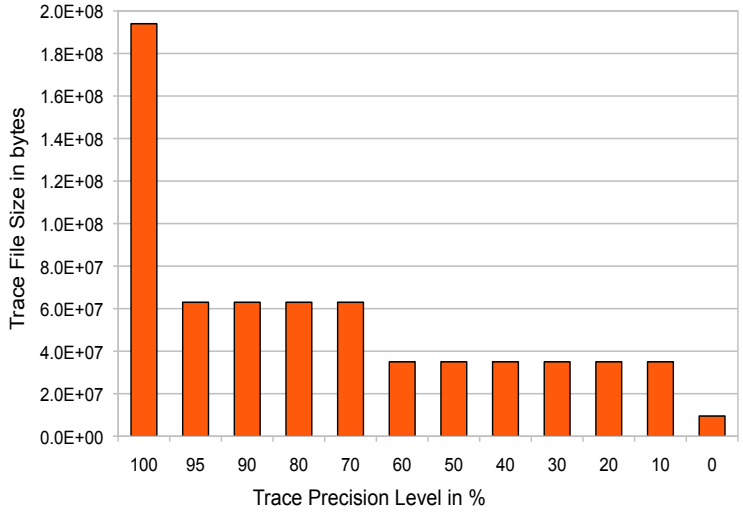

Figure 6.8: POP Trace Sensitivity for 512 nodes

explained in the section 4.1. The trace file size is constant up to a 70% precision level. At 60% precision, sizes drop again by almost 50%. This second reduction has been attributed to function parameters collected as histograms. Many events with varying function parameters are not combined under lossless tracing or result in vectors collected to represent varying parameters. Both contribute to the significant increase in the trace file size and prevent trace scalability with increasing numbers of nodes. Finally, another three-fold reduction in trace sizes is observed for forced histograms (0% precision level). At the 0% precision level, all non-matching values are represented as histograms, which results in the most concise trace possible with Scala-H-Trace. Overall, sensitivity experiments to precision levels show that small reductions in precision can significantly reduce the overall trace sizes. This particularly aids production-scale codes like POP, which otherwise cannot be feasibly traces without loss of information for thousands of nodes.

6.4 Trace Analysis

Finally, we demonstrate that our proof-of-concept trace analysis module is able to perform post-mortem analysis to examine the traces, such as how many calls have been issued and how I/O operations behave under strong scaling. While such statistics are common information items, Scala-H-Trace has the unique advantage of avoiding either rerunning the application or going through large-sized trace files. Tool features can be easily extended in the future to plug

in user supplied trace processing routines for customized analysis.

Table 6.1: Number of Multi-Scale MPI/POSIX I/O and Communication Calls for Flash I/O

# nodes	MPI-IO at 0	POSIX I/O at 0	Comm. at 0	MPI-IO Other	POSIX I/O Others	Comm. Other
2-1024	194	171	299	85	56	299

Table 6.1 shows the statistical information collected exploiting Scala-H-Trace's generic analysis feature with a specialized module for aggregation of event statistics. It shows that FLASH I/O issues the same number of I/O and communication calls under weak scaling with increasing number of nodes while the overall output file size increase due to the larger number of nodes. Since parallel HDF5 uses MPI-IO, which in turn uses POSIX I/O, separate aggregation results are reported for both MPI-IO and POSIX I/O. We observe that the number of MPI-IO calls for both node 0 and all others is greater than that of corresponding POSIX I/O calls. This is due to MPI_File_set_view calls issued before writing to the file.

Table 6.2: Number of I/O & Communication calls in POP

# nodes	I/O at 0	Coll. at 0	Block. at 0	NB at 0	Coll. Other	Block. Other	NB Other
2	1589	21247	129034	231714	21247	0	385350
4	1573	21257	179284	308952	21257	0	388838
8	1573	21277	210140	308952	21277	0	393393
16	1573	21317	1444912	386190	21317	0	447680
32	1573	21397	858648	386190	21397	0	451373
64	1573	12225	858648	386190	8575	0	382512
128	1573	21877	463372	386190	21877	0	441344
256	1573	22517	470288	386190	22517	0	426550
512	1573	23797	239932	386190	23797	0	424329
1024	1573	26357	240198	386190	26357	0	412485

Table 6.2 shows the statistical information collected similar to the one reported for FLASH I/O benchmark. Since only node 0 performs I/O operations, we report results for I/O operations and collective (Coll.)/blocking (Block.)/non-blocking (NB) communication operations for node

0 separately and the average number of operations for all other nodes. We identified that all of the blocking communication calls at node 0 are I/O induced. This confirms that using any parallel I/O methodology would have definitely reduced the communication overhead involved in I/O.

With more nodes, the average number of non-blocking calls does not change significantly. This shows that the fraction of non-blocking receives relative to the corresponding blocking sends from node 0 is almost negligible. We also observe that the communication overhead increases due to strong scaling to solve the problem across even larger number of nodes. We further infer from trace analysis that communication is performed in sub-groups as the average non-blocking calls in other nodes is greater than that for node 0. We derived from the trace analysis that even collective operations, such as MPI_Allreduce, are performed in sub-groups. This is also the explanation for the difference in collective communication between node 0 and other nodes for the 64-node experiment. Here, the collective operations at node 0 operate in one sub-group while the average number of collectives across all the other nodes is lower (dominated by other sub-groups). We manually verified the correctness of this result.

Chapter 7

Related Work

There are several tools, such as TAU [21], Vampir [15], Paraver [18] and SCALASCA [8], that capture communication and/or I/O trace events using library instrumentation similar to Scala-H-Trace. But only few employ trace compression techniques to control the trace file size. Many of these tools depend on zlib for compression, which compresses blocks of data without preserving the structure of the trace, *i.e.*, post-processing/analysis only becomes feasible after decompression. This also increases the memory requirements, effectively rendering trace analysis infeasible on commodity desktops or laptops and sometimes even high-end workstations, depending on the uncompressed trace size. Unlike these techniques, ScalaTrace [17] compresses traces while preserving the trace structure in terms of order of events. As a result, post-processing/analysis can be performed without decompression. We utilize this concept of structure preserving compression in Scala-H-Trace. Yet while ScalaTrace and any of the aforementioned tracing tools record lossless traces with a subset or all event parameters, Scala-H-Trace establishes a different methodology. Parameters, event frequencies and participant lists of nodes are recorded as histograms when lossless compressing cannot be established within a user-specified precision level. Employing statistical methods results in more concise traces even for non-SPMD programs at the expense of loss of information. Our replay tool uses an algorithm to issue events on-the-fly using the compressed traces, much like ScalaTrace. Yet recorded parameters are replayed in a probabilistic manner, which creates novel challenges that are met by our distributed approach to coordinate event replay across nodes.

The mpiP tool, a lightweight profiling library for MPI applications, collects statistical information about MPI functions [23]. It collects aggregate metrics like number of MPI events issued by the application and average execution times. This is useful to provide very high-level information on communication and I/O calls. Scala-H-Trace, in contrast, captures all events in traces and employs more sophisticated histogram bins only when the need arises for applications exhibiting non-SPMD behavior. Beside the histogram information, we also record outlier

information associated with each bin to detect communication bottlenecks and to provide a "big picture" of communication and I/O events in applications.

Kluge *et al.* [11] employ pattern matching techniques similar to ours to capture POSIX I/O calls in parallel programs. Unlike our approach, they perform post-mortem pattern matching only after collecting the application traces. They read the collected trace and create an I/O dependency graph thereby preserving the event order to do pattern matching. Even though post mortem pattern matching reduces the trace volume, this approach limits its usefulness in memory constrained systems like the IBM BlueGene family. Without online compression, either the memory footprint increases by holding the recorded trace or trace events are frequently written to disk, which affects the application execution behavior. They also do not employ pattern matching across nodes so that they require a trace file per node. This limits their approach in that they struggle with applications utilizing thousands of nodes due to parallel file system constraints. Our approach is immune to such limitations as a single trace file captures the behavior of all nodes with statistical information on a per-event and per-parameter basis.

Gao *et al.* [7] developed an event trace compression technique that performs static analysis on the application binary and collects loops and functions as structures. Along with these structure, a path grammar is constructed on-the-fly. Path grammars are then utilized to encode paths taken during execution. These structures are compressed individually and stored. Even the iteration count is stored along with the compressed structure traces. This loosely resembles the RSD and PRSD technique used in related work [9, 13, 16, 20]. But unlike Gao *et al.*'s work, our tool does not require the construction of grammars for individual applications separately. Our work employs a generalized trace compression approach based on call path stacks and records parameters exploiting statistical means. It is sufficient to link the tool library along with the application to collect traces. This generalization also enables comparative trace studies between two different applications.

Lu and Shen [12] proposed multi-layer event tracing and analysis to identify the system layers responsible for performance bottlenecks. Their evaluation was limited to very small systems (at most 16 nodes) and their approach does not employ any compression mechanism. With traces collected from many different layers of I/O subsystems, there will be unmanageable increases in the trace file size when 100s or 1000s of nodes are used. They have also provided interesting results on performance issues on parallel file system due to constraints at the operating system level using trace analysis. In contrast, our tool enables trace analysis using compressed traces and with even the potential to automate trace analysis by performing a stack walk to identify the layering information using recorded stack signatures along with event traces.

Chapter 8

Conclusion

We presented the design and implementation of Scala-H-Trace, which provides novel capabilities for more aggressive trace compression than any previous approach, collects I/O traces at multiple levels of I/O software stack and also a generic analysis tool that allows rapid post-mortem traversal of tracing in their compressed format to gather statistics, detect performance bottlenecks or analyze event precedence orders. Scala-H-Trace utilizes histograms based on a user-specified precision level. It features a distributed approach to deterministically replay statistical histogram traces where events are reissued without decompressing the original trace file.

Experimental results demonstrate the ability to obtain a single, near constant sized trace file or only sub-linear increases in the trace file size, even for production-scale scientific applications such as POP with non-SPMD behavior, the FLASH I/O benchmark and the CG benchmark of the NAS suite. Results also show that replay time for traced events are within 12%-15% of the original application execution time in majority of the cases, even for the most aggressive "forced" histograms. Such concise traces are unprecedented for isolated I/O and combined I/O plus communication tracing.

Thus, we have achieved the goals we had set and have proven our hypothesis stated initially. We expect that our trace collection approaches will aid in the study of communication and I/O behavior of complex extreme-scale applications running on thousands of cores.

REFERENCES

[1] FLASH I/O benchmark routine. http://www.ucolick.org/ zingale/flash_benchmark_io.

[2] Hierarchical data format. http://www.hdfgroup.org/HDF5.

[3] Network common data form. http://www.unidata.ucar.edu/software/netcdf/.

[4] Top 500 list. http://www.top500.org/, June 2002.

[5] N. Adiga and et al. An overview of the BlueGene/L supercomputer. In *Supercomputing*, November 2002.

[6] 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.

[7] X. Gao, A. Snavely, and L. Carter. Path grammar guided trace compression and trace approximation. *High-Performance Distributed Computing, International Symposium on*, 0:57–68, 2006.

[8] 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.

[9] 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.

[10] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque. Practical performance portability in the parallel ocean program (pop): Research articles. *Concurr. Comput. : Pract. Exper.*, 17(10):1317–1327, 2005.

[11] Michael Kluge, Andreas Knüpfer, Matthias Müller, and Wolfgang E. Nagel. Pattern matching and i/o replay for posix i/o in parallel programs. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 45–56, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Pin Lu and Kai Shen. Multi-layer event trace analysis for parallel i/o performance tuning. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 12, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, April 1995.

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

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

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

[22] Frederick H Streitz, James N Glosli, Mehul V Patel, Bor Chan, Robert K Yates, Bronis R de Supinski, James Sexton, and John A Gunnels. Simulating solidification in metals at high pressure: The drive to petascale computing. *Journal of Physics: Conference Series*, 46(254), 2006.

[23] 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.

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

[25] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable i/o tracing and analysis. In *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 26–31, New York, NY, USA, 2009. ACM.