TOOLS FOR DEBUGGING AND ANALYSIS OF CONCURRENT PROGRAMS

by

Saeed Taheri

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computing

The University of Utah

July 2021

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Saeed Taheri**
has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan,	Chair(s)	**—– 2021** Date Approved
Zvonimir Racamaric,	Member	**—– 2021** Date Approved
Hari Sundar,	Member	**—– 2021** Date Approved
Alexander Lex,	Member	**—– 2021** Date Approved
Martin Burtscher,	Member	**—– 2021** Date Approved

by **Mary Hall**, Chair/Dean of
the Department/College/School of **Computer Science**
and by **David B. Keida**, Dean of The Graduate School.

ABSTRACT

With the high growth in computation power and the invention of modern languages, concurrent software testing and debugging is vital to deliver reliable software. Finding bugs in concurrent/parallel software is notoriously challenging because 1) the interleaving space grows exponentially with the number of processing units (e.g., CPU cores), 2) the non-deterministic nature of concurrent software makes concurrent bugs difficult to reproduce, and 3) root-causing misbehaved executions of a concurrent program is non-trivial due to the complex interactions between concurrent components of the program. In this work, we have designed and implemented several frameworks and toolchains to overcome large-scale and real-world concurrent/parallel software debugging challenges. Our methods aim to facilitate the concurrent debugging process by providing efficient data collection and effective information retrieval mechanisms to target real-world software and bugs. First, we introduce PARLOT, a whole-program call tracing framework for HPC applications (MPI+X) that highly compresses the traces (up to more than 21000 times) while adding minimal overhead with an average required bandwidth of just 56 kB/s per core. Second, we present DiffTrace, a series of automated data abstraction, representation, and visualization techniques that differentiates the collected ParLOT traces and narrows the search down to just a few candidates of buggy traces. Finally, we illustrate GOAT, an end-to-end framework for automated tracing, analysis, and testing of concurrent Go applications. We also propose a set of coverage metrics to measure the quality of testing in CSP-like concurrent languages.

For my family

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Concurrent Software Correctness

In the life-cycle of computer programs, software testing and debugging play essential roles. Many approaches have been developed to test and debug software at different layers and from different aspects. Traditional and basic debugging techniques such as "printf" debugging, interactive debugging, memory dumps, and profiling have been used widely in software development. However, with the high growth in computation power and modern languages, programs are getting more sophisticated, complex, large, and heterogeneous to efficiently exploit the processing power (e.g., Cloud computing, High Performance Computing - HPC). Locating bugs in concurrent software is notoriously challenging because: 1) the interleaving space grows exponentially with the number of processing units (e.g., CPU cores), 2) the non-deterministic nature of concurrent software makes concurrent bugs difficult to reproduce, and 3) root-causing misbehaved executions of a concurrent program is non-trivial due to the complex interactions between concurrent components of the program.

1.1.1 Concurrent Bugs

1.1.1.1 Deadlocks

1.1.1.2 Misuse of Serialization/Synchronization

1.1.1.2.1 Race

1.1.1.2.2 Atomicity Violations

1.1.2 Concurrent Debuggers

1.1.2.1 Race Checkers

1.1.2.2 Static Analyzers

1.1.2.2.1 Assertion-based Debugging

1.1.2.2.2 Model Checking

1.1.2.2.3 Theorem Proving

1.1.2.3 Dynamic Analyzers

1.1.2.3.4 Tracing

1.1.2.3.5 Visualization

1.1.2.4 Hybrid Analyzers

1.1.2.4.6 Record and Replay

1.1.2.4.7 Testing

1.2 Dissertation Statement

Today's programs are designed to exploit the power of modern processing units. Efficiently collecting data and systematically analyze them is an effective way to gain insight into the behavior of complex programs and help developers fix the flaws of the program.

1.3 Background

Goal one: Heterogeneity — Many computation-based scientific problems like gene sequence alignment in biology, climate simulations and stencil codes are implemented to execute on supercomputers. Depending on the problem domain, HPC applications apply a wide range of programming languages, compilers, and optimizations to solve problems. Besides, HPC applications are often implemented for a particular architecture to maximize performance. Moreover, different types of faults might occur at different levels. There exist debugging tools that target a specific domain, architecture, programming language, or particular classes of bugs. However, the applied techniques in such tools might not prove effective for other domains or bug types. In this work, we propose a general-purpose infrastructure to collect comprehensive data from an application regardless of its domain, compiler, and programming language by tracing at the binary level.

Goal two: Scalability — Powerful supercomputers are deployed to execute the HPC application on a massive scale to achieve high throughput. The high throughput implies that many processing units are busy with executing instructions or communicating with each other in parallel. Enough evidence needs to get gathered during the execution from each processing unit (process, thread, task) to understand the computation and communication behavior of the program. Also, some abnormal behaviors may only get triggered and manifest when executed on larger scales. Thus it is essential to have a mechanism to collect sufficient data during the execution of HPC applications on a massive scale. However, capturing the per-processing-unit events requires *control events* to attach to each original event (e.g., function calls). Placing and executing these other control events adds overhead to the application execution and might hurt the performance. In addition, the collected data from each processing unit have to transfer through system and network bandwidth for the analysis phase. Special care for instrumenting and gathering data is essential to preserve the performance of the native application. On the other hand, during the execution of HPC applications, each processing unit or node contains a large number of *events* to capture. Thus long-running large scale applications often leave an overwhelming amount of data to analyze. The search space rapidly grows as the scale of application execution increases. Having an efficient *data collection* mechanism to overcome the trade-off between *comprehensive information* and *low overhead*, is another motivation of this dissertation.

Goal Three: Intuitition — The effort towards saving significant energy causes the non-intuitive behavior of HPC applications. Floating-point reduction, low-buffering message passing, non-determinism of application, and collective operations are just some examples of uncertain behavior of HPC applications. Root causing the unsuccessful execution of these kinds of applications is not achievable by ad hoc debuggers. The efficient data collection during program execution enables developers to post-mortem analyze the collected data from different aspects. Given the one-time gathered data, the behavior of the program execution can be studied iteratively, from various angles on each iteration. In this dissertation, different data abstraction and visualization techniques are applied to bring intuition and reveal interesting facts about the program behavior from a purposeful point of view.

1.4 Contributions of the Dissertation

- We introduce ParLoT, a whole-program call tracing framework for HPC applications (MPI+X) that highly compresses the traces (up to more than 21000 times) while adding minimal overhead with average required bandwidth of just 56 kB/s per core.

- We present DiffTrace, a series of automated data abstraction, representation and visualization techniques that differntiates the collected ParLOT traces and narrows the search down to just a few candidates of buggy traces.

- We illustrate GOAT, an end-to-end framework for automated tracing, analysis and testing of concurrent Go applications.

- We propose a set of coverage metrics to measure the quality of testing in CSP-like concurrent languages.

1.5 Organization of the Dissertation

- Efficient Whole-Program Tracing (PARLOT)

- Whole-Program Trace Analysis for Debugging (DiffTrace)

- End-to-end Analysis Framework for Concurrent Go (Goat)

CHAPTER 2

WHOLE-PROGRAM DYAMIC TRACING

This chapter is based on the work published at the Workshop on Programming and Performance Visualization Tools (ESPT) 2018 [58]. We present PARLOT, a framework for efficient whole-program call tracing for HPC (MPI+X) applications using Intel Pin [39] dynamic binary isntrumentation that includes following key features: (1) It describes a technique that makes low-overhead on-the-fly compression of whole-program call traces feasible. (2) It presents a new, efficient, incremental trace-compression approach that reduces the trace volume dynamically, which lowers not only the needed bandwidth but also the tracing overhead. (3) It collects all caller/callee relations, call frequencies, call stacks, as well as the full trace of all calls and returns executed by each thread, including in library code. (4) It works on top of existing dynamic binary instrumentation tools, thus requiring neither source-code modifications nor recompilation. (5) It supports program analysis and debugging at the thread, thread-group, and program level. PARLOT establishes that comparable capabilities are currently unavailable. Our experiments with the NAS parallel benchmarks running on the Comet supercomputer with up to 1,024 cores show that ParLoT can collect whole-program function-call traces at an average tracing bandwidth of just 56 kB/s per core.

2.1 Introduction

Understanding and debugging HPC programs is time-consuming for the user and computationally inefficient. This is especially true when one has to track control flow in terms of function calls and returns that may span library and system codes. Traditional software engineering quality assurance methods are often inapplicable to HPC where concurrency combined with large problem scales and sophisticated domain-specific math can make programming very challenging. For example, it took months for scientists to debug an MPI laser-plasma interaction code [22].

HPC bugs may be a combination of both flawed program logic and unspecified or illegal interactions between various concurrency models (e.g., PThreads, MPI, OpenMP, etc.) that coexist in most large applications [22]. Moreover, HPC software tends to consume vast amounts of CPU time and hardware resources. Reproducing bugs by rerunning the application is therefore expensive and undesirable. A natural and field-proven approach for debugging is to capture detailed execution traces and compare the traces against corresponding traces from previous (stable) runs [3,13]. A *key requirement* is to do this collection *as efficiently as possible* and in *as general and comprehensive a manner* as possible.

Existing tools in this space do not meet our criteria for efficiency and generality. The highly acclaimed STAT [3] tool has helped isolate bugs based on building equivalence classes of MPI processes and spotting outliers. We would like to go beyond the capabilities offered by STAT and support the collection of *whole-program* traces that can then be employed by a gamut of back-end tools. Also, STAT is usually brought into the picture when a failure (e.g., a deadlock or hang) is encountered. We would like to move toward an "always on" collection regime, as we cannot anticipate when a failure will occur – or, more importantly, *whether the failure will be reproducible.* There are no reported debugging studies on using STAT in continuous collection ("always on") mode. In CSTG [13], the collection is orchestrated by the user around chosen collection points and employs heavy-weight unix `backtrace` calls. These again are different from ParLoT, where collection points would not be a priori chosen.

The thrust of the work in this paper is to avoid many of the drawbacks of existing tracing-based tools. We are interested in avoiding source-code modifications and recompilation — thus making binary instrumentation-based tools the only practical and widely deployable option. We also believe in the value of creating tools that are *portable across a wide variety of platforms*.

Our goal is to use *compression* for trace aggregation and to offer a generic and low-overhead tracing method that (1) collects dynamic call information during execution (all function calls and returns) for debugging, performance evaluation, phase detection [51], etc., (2) has low overhead, (3) and requires little tracing bandwidth. *Providing all these features in a single tool that operates based on binary instrumentation is an unsolved problem.* In this paper, we describe a new tracing approach that fulfills these requirements, which we implemented in our proof-of-concept PARLOT tool.

With PARLOT, users can easily build a host of post-processors to examine executions from many vantage points. For instance, they can write post-processors to detect unexpected (or "outlier") executions. If needed, they can drill down and detect abnormal behaviors *even in the runtime and support library stack* such as MPI-level activities. In HPC, it is well-known (especially on newer machines) that bugs are often due to broken libraries (MPI, OpenMP), a broken runtime, or OS-level activities. Having a single low-overhead tool that can "X-ray" an application to this depth is a goal met by PARLOT— a unique feature in today's tool eco-system.

To further motivate the need for whole-program function call traces, consider the expression `f()+g()`. In C, there is no sequence point associated with the + operator [48]. If these function calls have inadvertent side-effects causing failure, it is important to know in which order `f()` and `g()` were invoked—something that is easy to discern using PARLOT's traces. One may be concerned that such a tool introduces excessive execution slowdown. PARLOT goes to great lengths to minimize these overheads to a level that we believe most users will find acceptable. The mindset is to *"pay a little upfront to dramatically reduce the number of overall debug iterations"*.

As proof of concept, we gathered preliminary results from using the PARLOT tracing mechanism to compare different runs. We injected various bugs into the MPI-related functions of ILCS [9], a parallelization framework for iterative local searches. We ran PARLOT on top of executions of buggy and bug-free versions of ILCS and collected traces. Since PARLOT's traces maintain the order of the function calls, we were able to split the traces at multiple points of interest and to feed different chunks of traces to a Concept Lattice data structure [17] [21]. Having the totally ordered sequence of function calls of the whole program for each active process/thread enabled us to quickly narrow down the search space to locate the cause of the abnormal behavior in the buggy version of ILCS.

This paper does not pursue debugging per se but rather a thorough benchmarking of PARLOT. It makes the following main contributions:

- It introduces a new tracing approach that makes it possible to capture the whole-program call-return, call-stack, call-graph, and call-frequency information, including all library calls, for every thread and process of HPC applications at low overhead in both space and time.

- It describes a new incremental data compression algorithm to drastically reduce the required tracing bandwidth, thus enabling the collection of whole-program traces, which would be infeasible without on-the-fly compression.

- It presents PARLOT, a proof-of-concept tool that implements our compression-based low-overhead tracing approach. PARLOT is capable of instrumenting x86 applications at the binary level (regardless of the source language used) to collect whole-program call traces.

The remainder of this paper is organized as follows. Section 2.2 introduces the basic ideas and infrastructure behind PARLOT and other tracing tools. Section 2.3 describes the design of PARLOT in detail. Sections 2.4 and 2.5 present our evaluation of different aspects of PARLOT and compare it with a similar tool. Section 2.6 concludes the paper with a summary and future work.

2.2 Background and Related Work

Recording a log of events during the execution of an application is essential for better understanding the program behavior and, in case of a failure, to locate the problem. Recording this type of information requires instrumentation of the program either at the source-code or the binary-code level. Instrumenting the source code by adding extra libraries and statements to collect the desired information is easy for developers. However, doing so modifies the code and requires recompilation, often involving multiple different tools and complex hierarchies of makefiles and libraries, which can make this approach cumbersome and frustrating for users. Instrumenting an executable at the binary level using a tool is typically easier, faster, and less error prone for most users. Moreover, binary instrumentation is language independent, portable to any system that has the appropriate instrumentation tool installed, and provides machine-level insight into the behavior of the application.

2.2.1 Binary Instrumentation

Executables can be instrumented *statically*, where the additional code is inserted into the binary before execution, which results in a persistent modified executable, or *dynamically*, where the modification of the executable is not permanent. In dynamic binary instrumentation, code behavior can be monitored at runtime, making it possible to handle dynamically-generated and self-modifying code. Furthermore, it may be feasible to attach the instrumentation to a running process, which is particularly useful for long-running applications and infinite loops.

Many different tools for investigating application behavior have been designed on top of such Dynamic Binary Instrumentation (DBI) frameworks. For instance, Dyninst [40] provides a dynamic instrumentation API that gives developers the ability to measure various performance aspects. It is used in tools like Open-SpeedShop [55] and TAU [56] as well as correctness debuggers like STAT [3]. Moreover, VampirTrace [30] uses it to provide a library for collecting program execution logs.

Valgrind [46] is a shadow-value DBI framework that keeps a copy of every register and memory location. It provides developers with the ability to instrument system calls and instructions. Error detectors such as Memcheck [47] and call-graph generators like CALLGRIND [61] are built upon Valgrind.[1]

We implemented PARLOT on top of PIN [39], a DBI framework for the IA-32, x86-64, and MIC instruction-set architectures for creating dynamic program analysis tools. There is also version of PIN available for the ARM architecture [24]. PARLOT mutates PIN to trace the entry (call) and exit (return) of every executed function. Note that our tracing and compression approaches can equally be implemented on top of other instrumentation tools. For example, PMaC [59] is a DBI tool for the PowerPC/AIX architecture upon which PARLOT could also be based.

[1]Given the absence of tools similar to PARLOT, we employ CALLGRIND as a "close-enough" tool in our comparisons elaborated in §**??**. In this capacity, CALLGRIND is similar to PARLOT(M), a variant of PARLOT that only collects traces from the `main` image. We perform such comparison to have an idea of how we fare with respect to one other tool. In §2.5, we also present a self-assessment of PARLOT separately.

2.2.2 Efficient Tracing

When dealing with large-scale parallel programs, any attempt to capture reasonably frequent events will result in a vast amount of data. Moreover, transferring and storing the data will incur significant overhead. For example, collecting just one byte of information per executed instruction yields on the order of a gigabyte of data per second on a single high-end core. Storing the resulting multi-gigabyte traces from many cores can be a challenge, even on today's large hard disks.

Hence, to be able to capture whole-program call traces, we need a way to decrease the space and runtime overhead. *Compression* can encode the generated data using a smaller number of bits, help reduce the amount of data movement across the memory hierarchy, and lower storage and network demands. Although the encoded data will later have to be decoded for analysis, compressing them during tracing enables the collection of *whole-program* traces.

The use of compression by itself is not new. Various performance evaluation tools [1, 33, 56] already employ compression during the collection of performance analysis data. Tools such as ScalaTrace [49] also exploit the repetitive nature of time-step simulations [15]. Aguilar et al. [2] proposed a lossy compression mechanism using the Nami library [16] for online MPI tracing. Mohror and Karavanic [42] investigated similarity-based trace reduction techniques to store and analyze traces at scale.

Many performance and debugging tools for HPC applications [3, 45] rely on mechanisms such as MRNet [53] to accelerate the collection and aggregation of traces based on an overlay network to overcome the challenge of massive data movement and analysis. However, our experiments show that, due to the high compression ratio of ParLoT traces, such mechanisms for data movement and aggregation may be unnecessary.

The novelty offered by ParLoT lies in the combination of compression speed, efficacy, and low timing jitter made possible by its *incremental* lossless compression algorithm, which is described in §2.3. It immediately compresses all traced information while the application is running, that is, ParLoT does not record the uncompressed trace in memory. As a result, just a few kilobytes of data need to be written out per thread and per second, thus requiring only a small fraction of the disk or network bandwidth. The traces are decompressed later when they are read for offline analysis. From the decompressed full function-call trace, the complete call-graph, call-frequency, and caller-callee information can be extracted. This can be done at the granularity of a thread, a group of threads, or the whole application. We now elaborate on the design of ParLoT that makes these innovations possible.

2.3 Design of ParLoT

Our experimental results in §2.5 highlight why *compression* is essential to make our approach work. We used ParLoT to record a unique 16-bit identifier for every function call and return. Tracing just this small amount of information without compression when running the Mantevo miniapps [25] on Stampede 1 resulted in about 2 MB/s of data per core on average. Extrapolating this value to all 102,400 cores of Stampede 1 (not counting the accelerators) yields 205 GB/s of trace data, which exceeds the Lustre filesystem's parallel write performance of 150 GB/s. Enabling ParLoT's compression algorithm reduced the emitted trace data by a factor of 100 on average, a ratio that is quite stable w.r.t scaling, making it possible to trace full-scale programs while leaving over 98% of the I/O bandwidth to the application. Therefore, ParLoT should also work for codes with higher bandwidth requirements than the ones we tested.

Figure 2.1 provides a general overview of ParLoT's workflow. Basic blocks within program executables are *dynamically* instrumented before being executed. The collected data are compressed on-the-fly at runtime.

2.3.1 Tracing Operation

ParLoT uses the PIN API as its instrumentation mechanism to gather traces. In particular, it instructs PIN to instrument every thread launch and termination in the application as well as every function entry and exit. The thread-launch instrumentation code initializes the per-thread tracing variables and opens a file into which the trace data from that thread will be written. The thread-termination code finalizes any ongoing compression, flushes out any remaining entries, and closes the trace file. ParLoT assigns every static function in each image (main program and all libraries) a unique unsigned 16-bit ID, which it records in a separate file together with the image and function name. This file allows the trace reader to map IDs back to function-name/image pairs.

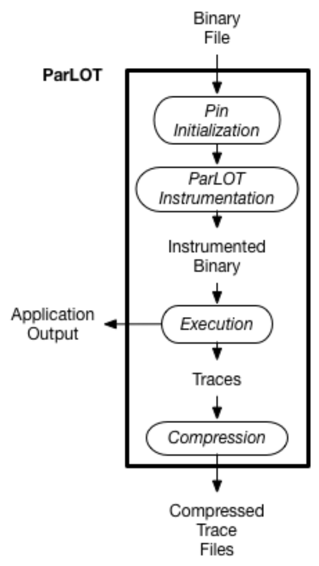


Figure 2.1: Overview of PARLOT

For every function *entry*, PARLOT executes extra code that has access to the thread ID, function ID, and current stack-pointer (SP) value. Based on the SP value, it performs call-stack correction if necessary (see §2.3.4), adds the new function to a data structure it maintains that holds the call stack (which is separate from the application's runtime stack), and emits the function ID into the trace file via an incremental compression algorithm (see §2.3.2). All of this is done independently for each thread. Similarly, for every function *exit*, PARLOT also executes extra code that has access to the thread ID, function ID, and current SP value. Based on the SP value, it performs call-stack correction if necessary, removes the function from its call-stack data structure, and emits the reserved function ID of zero into the trace file to indicate an exit. As before, this is done via an incremental compression algorithm. We use zero for all exits rather than emitting the function ID and a bit to specify whether it is an entry or exit because using zeros results in more compressible output. This way, half of the values in the trace will be zero.

2.3.2 Incremental Compression

PARLOT immediately compresses the traced information even before it is written to memory. It does, however, keep a sliding window (circular buffer) of the most recent uncompressed trace events, which is needed by the compressor. It compresses each function ID before the next function ID is known. The conventional approach would be to first record uncompressed function IDs in a buffer and later compress the whole buffer once it fills up. However, this makes the processing time very non-uniform. Whereas almost all function IDs can be recorded very quickly since they just have to be written to the buffer, processing a function ID that happens to fill the buffer takes a long time as it triggers the compression of the entire buffer. This results in sporadic blocking of threads during which time they make no progress towards executing the application code. Initial experiments revealed that such behavior can be detrimental when one thread is polling data from another thread that is currently blocked due to compression. For example, we observed a several order of magnitude increase in entry/exit events of an internal MPI library function when using block-based compression.

To remedy this situation, the compressor must operate incrementally, i.e., each piece of trace data must be compressed when it is generated, without buffering it first, to ensure that there is never a long-latency compression delay. Few existing compression algorithms have been implemented in such a manner because it is more difficult to code up and probably a little slower. Nevertheless, we were able to implement our algorithm (discussed next) in this way so that each trace event is compressed with similar latency.

2.3.3 Compression Algorithm

We used the CRUSHER framework [8,10,12,62] to automatically synthesize an effective and fast lossless compression algorithm for our traces. CRUSHER is based on a library of data transformations extracted from various compression algorithms. It combines these transformations in all possible ways to generate algorithm candidates, which it then evaluates on a set of training data. We gathered uncompressed traces from some of the Mantevo miniapps [25] for this purpose. This evaluation revealed that a particular word-level Lempel-Ziv (LZ) transformation followed by a byte-level Zero-Elimination (ZE) transformation works well. In other words, ParLot's trace entries, which are two-byte words, are first transformed using LZ. The output is interpreted as a sequence of bytes, which is transformed using ZE for further compression. The output of ZE is written to secondary storage.

LZ implements a variant of the LZ77 algorithm [64]. It uses a 4096-entry hash table to identify the most recent prior occurrence of the current value in the trace. Then it checks whether the three values immediately before that location match the three trace entries just before the current location. If they do not, the current trace entry is emitted and LZ advances to the next entry. If the three values match, LZ counts how many values following the current value match the values following that location. The length of the matching substring is emitted and LZ advances by that many values. Note that all of this is done incrementally. The history of previous trace entries available to LZ for finding matches is maintained in a 64k-entry circular buffer.

ZE emits a bitmap in which each bit represents one input byte. The bits indicate whether the corresponding bytes are zero or not. Following each eight-bit bitmap, ZE emits the non-zero bytes.

As mentioned above, we had to implement the two transformations incrementally to minimize the maximum latency. This required breaking them up into multiple pieces. Depending on the state the compressor is in when the next trace entry needs to be processed, the appropriate piece of code is executed and the state updated. If the LZ code produces an output, which it only does some of the time, then the appropriate piece of the ZE code is executed in a similar manner.

2.3.4 PIN and Call-Stack Correction

To be able to decode the trace, i.e., to correctly associate each exit with the function entry it belongs to, our trace reader maintains an identical call-stack data structure. Unfortunately, and as pointed out in the PIN documentation [27], it is not always possible to identify all function exits. For example, in optimized code, a function's instructions may be inlined and interleaved with the caller's instructions, making it sometimes infeasible for PIN to identify the exit. As a consequence, we have to ensure that PARLOT works correctly even when PIN misses an exit. This is why the SP values are needed.

During tracing, PARLOT not only records the function IDs in its call stack but also the associated SP values. This enables it to detect missing exits and to correct the call stack accordingly. Whenever a function is entered, it checks if there is at least one entry in the call stack and, if so, whether its SP value is higher than that of the current SP. If it is lower, we must have missed at least one exit since the runtime stack grows downwards (the SP value decreases with every function entry and increases with every exit). If a missing exit is detected in this manner, PARLOT pops the top element from its call stack and emits a zero to indicate a function exit. It repeats this procedure until the stack is empty or its top entry has a sufficiently high SP value. The same call-stack correction technique is applied for every function exit whose SP value is inconsistent. Note that the SP values are only used for this purpose and are not included in the compressed trace.

The result is an internally consistent trace of function entry and exit events, meaning that parsing the trace will yield a correct call stack. This is essential so that the trace can be decoded properly. Moreover, it means that the trace includes exits that truly happened in the application but that were missed by PIN. Note, however, that our call-stack correction is a best-effort approach and may, in rare cases, temporarily not reflect what the application actually did. For example, this can happen for functions that do not create a frame on the runtime stack. When implementing PARLOT on top of another DBI framework, call-stack correction may not be needed, resulting in even lower PARLOT overhead.

2.4 Evaluation Methodology

2.4.1 Benchmarks and System

We performed our evaluations on the MPI-based NAS Parallel Benchmarks (NPB) [5]. NPB includes four inputs sizes. To keep the runtimes reasonable, we show results for the class *B* (small-medium) and class *C* (medium-large) inputs.

We compiled the NPB codes with the mpicc and mpif77 wrappers of MVAPICH 2.2.1, which are based on icc/ifort 14.0.2 using the prescribed -g and -O1 optimization flags. Quick tests showed that higher optimization levels do not significantly improve the performance.

We ran all experiments on Comet at the San Diego Supercomputer Center [57], whose filesystem is NFS and Lustre. Comet has 1944 compute nodes, each of which has dual-socket Intel Xeon E5-2680 v3 processors with a total of 28 cores (14 per socket) and 128 GB of main memory. Note that we only used 16 cores per node as many of the NPB programs require a core count that is a power of two. To study the scaling behavior, we ran experiments on 1, 4, 16 and 64 compute nodes, i.e., on up to 1024 cores.

2.4.2 Metrics

We use the following metrics to quantify and compare the performance of the tracing tools. Unless otherwise noted, all results are based on the median of three identical experiments.

- The **tracing overhead** is the runtime of the target application when it is being traced divided by the runtime of the same application without tracing. This lower-is-better ratio measures by how much the tracing (and the compression when enabled) slows down the target application.

- The **tracing bandwidth** is the size of the trace information divided by the application runtime. To make the results easier to compare, we generally list the tracing bandwidth per core, i.e., the tracing bandwidth divided by the number of cores used. This lower-is-better metric is expressed in kilobytes per second (kB/s) per core. It specifies the average needed bandwidth to record the trace data.

- The **compression ratio** is the size of the uncompressed trace divided by the size of the generated (compressed) trace. This higher-is-better ratio measures the factor by which the compression reduces the trace size. In other words, without compression, the tracing bandwidth would be higher by this factor.

2.4.3 Tracing Tools

We compare our ParLoT tool, implemented on top of PIN 3.5, with Callgrind 3.13. ParLoT was compiled with gcc 4.9.2 using PIN's make system and Callgrind with Valgrind's make system. We created the following versions of ParLoT to evaluate different aspects of its design.

- **ParLoT(m)** is the normal ParLoT tool configured to only collect function-call traces from the main image of the application.

- **ParLoT(a)** is the normal ParLoT tool configured to collect function-call traces from all images of the application, including library function calls.

- **Pin-Init** is a crippled version of ParLoT from which the tracing code has been removed. The purpose of Pin-Init is to see how much of the overhead is due to PIN.

- **ParLoT-NC** is the normal ParLoT tool but with compression disabled. It writes out the captured data in uncompressed form. The purpose of ParLoT-NC is to show the performance impact of the compression.

It proved surprisingly difficult to find a tool that is similar to ParLoT because there appear to be no other tools that generate whole program call traces. In the end, we settled on Callgrind as the most similar tool we could find and used it for our comparisons.

Table 2.1: Overhead added by each tool

Input	Tool	# Nodes	bt	cg	ep	ft	is	lu	mg	sp	GM
B	ParLoT(m)	1	1.6	1.8	2.6	2.1	2.5	1.3	2.5	1.3	1.9
		4	1.8	1.9	1.9	1.7	1.8	1.8	1.5	1.7	1.8
		16	2.2	2.6	2.0	1.9	1.8	2.7	2.4	2.2	2.2
		64	2.1	2.2	2.4	2.0	4.3	4.4	2.0	2.1	2.5
		AVG	1.9	2.1	2.2	1.9	2.6	2.6	2.1	1.8	**2.1**
	ParLoT(a)	1	1.8	2.7	4.2	2.8	4.2	1.7	4.8	1.7	2.8
		4	2.6	3.1	3.4	2.8	3.0	2.8	2.8	2.7	2.9
		16	3.5	4.2	3.4	2.9	2.8	4.3	4.5	3.7	3.6
		64	3.1	3.3	3.8	3.0	5.4	4.7	3.2	3.3	3.7
		AVG	2.8	3.3	3.7	2.9	3.9	3.4	3.8	2.8	**3.2**
	Callgrind	1	8.6	6.0	8.9	10.1	2.5	7.5	3.3	6.6	6.1
		4	6.0	3.6	2.9	3.5	1.5	5.2	1.2	5.8	3.2
		16	4.3	3.3	2.2	2.2	1.7	4.6	1.8	4.3	2.8
		64	2.3	2.0	1.7	2.1	4.1	4.0	1.5	2.5	2.3
		AVG	5.3	3.7	3.9	4.5	2.4	5.3	2.0	4.8	**3.6**
C	ParLoT(m)	1	1.4	1.3	2.5	1.9	2.3	1.1	1.7	1.1	1.6
		4	1.6	1.7	1.8	1.6	1.7	1.3	1.8	1.4	1.6
		16	1.8	2.4	2.5	1.5	1.8	2.2	2.4	1.8	2.0
		64	2.2	2.7	2.4	1.6	4.5	3.4	2.4	2.2	2.6
		AVG	1.8	2.0	2.3	1.7	2.6	2.0	2.1	1.6	**1.9**
	ParLoT(a)	1	1.5	1.6	3.2	2.0	2.8	1.2	2.5	1.2	1.9
		4	1.9	2.4	2.6	2.1	2.6	1.7	3.1	1.7	2.2
		16	2.7	3.5	4.1	2.1	2.8	3.2	4.0	2.5	3.0
		64	3.6	4.1	4.2	2.2	5.5	4.4	4.2	3.0	3.8
		AVG	2.4	2.9	3.5	2.1	3.4	2.6	3.5	2.1	**2.7**
	Callgrind	1	8.5	4.4	13.2	13.1	3.3	7.9	5.9	5.1	6.9
		4	8.7	4.5	4.8	6.4	1.7	6.4	2.8	6.3	4.6
		16	6.9	3.9	3.1	2.8	1.8	6.4	2.1	6.1	3.7
		64	4.4	3.5	2.1	2.5	4.2	5.2	2.1	3.8	3.3
		AVG	7.1	4.1	5.8	6.2	2.8	6.5	3.2	5.3	**4.6**

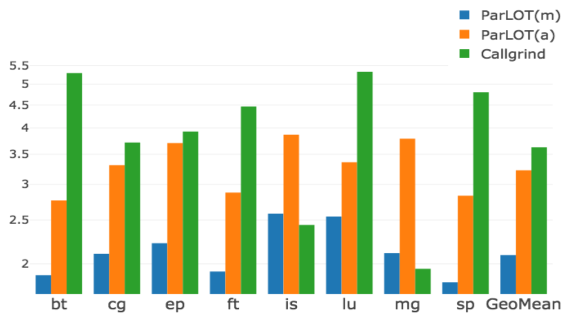


Figure 2.2: Average tracing overhead on the NPB applications - Input B

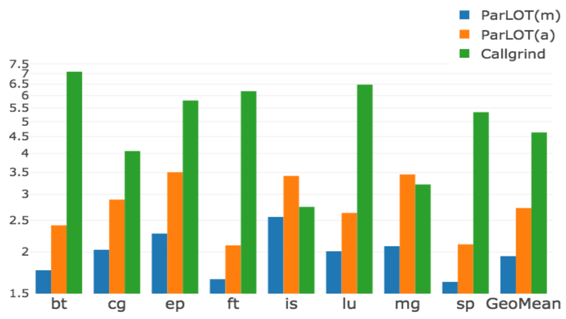


Figure 2.3: Average tracing overhead on the NPB applications - Input C

CALLGRIND is based on the Valgrind DBI tool. It collects function-call graphs combined with performance data to show the user what portion of the execution time has been spent in each function.

Each CALLGRIND trace file contains a sequence of function names (or their code) plus numerical data for each function on its caller-callee relationship with other functions. Moreover, it contains cost information for each function in terms of how many machine instructions it read. This information is collected using hardware performance counters. The format of the file is plain ASCII text. Interestingly, all numerical values are expressed relative to previous values, i.e., they are delta (or difference) encoded. This simple form of compression is enabled by default in CALLGRIND.

We believe the information traced by CALLGRIND is reasonably similar to the information traced by PARLOT(M). Whereas CALLGRIND's traces include performance data that PARLOT does not capture, PARLOT records the whole-program call trace, which CALLGRIND does not capture. The full function-call trace is a strict superset of the call-graph

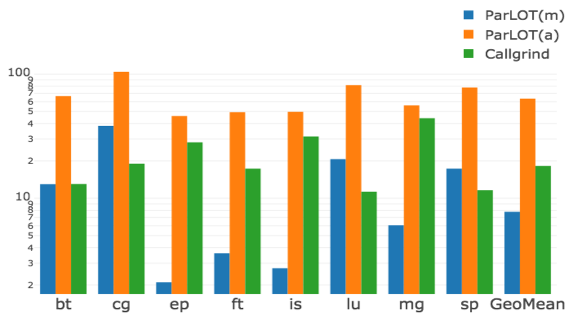


Figure 2.4: Average required bandwidth per core (kB/s) on the NPB applications - Input B

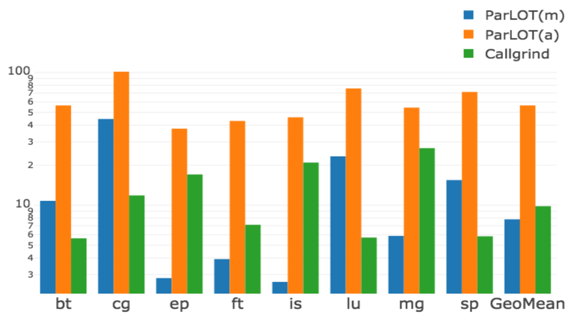


Figure 2.5: Average required bandwidth per core (kB/s) on the NPB applications - Input C

information that CALLGRIND records because the call graph can be extracted from the function-call trace but not vice versa. In particular, CALLGRIND cannot recreate the order of the function calls a thread made whereas PARLOT can.

2.5 Results

2.5.1 Tracing Overhead

Table 2.1 shows the tracing overhead of PARLOT(M), PARLOT(A), and CALLGRIND on each application of the NPB benchmark suite for different node counts. The last column of the table lists the geometric mean over all eight programs. The AVG rows show the average over the four node counts.

On average, both PARLOT(M) and PARLOT(A) outperform CALLGRIND. The bolded numbers in Table 2.1 for input C show that the average overhead is 1.94 for PARLOT(M), 2.73 for PARLOT(A), and 4.63 for CALLGRIND. Figures 2.2 and 2.3 show these results in visual form.

Table 2.2: Required bandwidth per core (kB/s)

Input	Tool	# Nodes	bt	cg	ep	ft	is	lu	mg	sp	GM
B	PARLOT(M)	1	4.7	21.9	3.8	1.5	0.8	2.4	5.6	5.4	3.7
		4	14.3	41.1	1.9	3.5	2.2	21.5	6.5	15.9	8.1
		16	14.3	46.6	1.5	4.9	3.4	31.8	6.5	18.6	9.4
		64	18.6	43.6	1.3	4.6	4.5	27.1	5.6	29.6	9.9
		AVG	13.0	38.3	2.1	3.6	2.7	20.7	6.1	17.4	**7.8**
	PARLOT(A)	1	48.7	89.4	47.2	45.6	60.0	53.6	60.8	54.3	56.2
		4	61.8	101.2	45.2	55.1	53.2	71.1	54.9	73.6	62.7
		16	74.0	116.9	47.4	48.9	47.8	100.9	55.8	84.6	68.0
		64	81.8	110.2	44.2	48.0	37.8	100.3	52.7	99.9	66.5
		AVG	66.6	104.4	46.0	49.4	49.7	81.5	56.0	78.1	**63.3**
	CALLGRIND	1	1.6	7.7	7.4	4.6	39.5	2.6	34.4	2.7	6.7
		4	6.5	16.0	22.1	15.7	45.5	8.6	45.5	7.8	16.3
		16	17.2	24.6	37.4	23.8	29.9	16.2	51.5	15.8	24.9
		64	26.8	27.7	45.9	25.1	11.0	17.8	45.3	20.2	25.0
		AVG	13.0	19.0	28.2	17.3	31.5	11.3	44.2	11.6	**18.2**
C	PARLOT(M)	1	1.8	17.0	5.2	1.2	0.7	0.8	3.6	1.4	2.2
		4	7.5	44.9	3.0	2.5	2.1	20.1	7.1	13.7	7.6
		16	16.3	55.0	1.8	6.1	3.4	34.1	7.2	20.7	10.7
		64	17.5	61.4	1.3	5.9	4.4	38.3	5.6	26.1	10.9
		AVG	10.8	44.6	2.8	3.9	2.7	23.3	5.9	15.5	**7.8**
	PARLOT(A)	1	17.8	53.4	26.3	20.9	48.3	25.3	52.6	19.5	30.0
		4	51.8	95.8	36.8	43.8	51.4	58.4	54.2	65.8	55.2
		16	75.4	121.0	44.3	61.4	46.9	101.1	56.5	101.3	71.4
		64	80.6	135.2	43.5	46.3	37.1	117.9	54.1	99.0	69.0
		AVG	56.4	101.4	37.7	43.1	45.9	75.7	54.3	71.4	**56.4**
	CALLGRIND	1	0.4	3.1	2.0	1.1	14.6	0.7	7.0	0.8	1.9
		4	1.8	8.9	7.7	4.5	31.7	2.8	21.0	2.8	6.4
		16	6.0	15.8	22.9	10.8	26.5	7.5	39.1	7.0	13.7
		64	14.3	19.6	35.8	12.2	11.1	11.9	40.7	12.8	17.4
		AVG	5.6	11.8	17.1	7.1	21.0	5.7	26.9	5.8	**9.8**

The key takeaway point is that the overhead of PARLOT is roughly a factor of two to three, which we believe users may be willing to accept, for example, if it helps them debug their applications. This is promising especially when considering how detailed the collected trace information is and that most of the overhead is due to PIN (see §2.5.4). Note that PARLOT's overhead is typically lower than that of CALLGRIND, which collects less information.

The overhead of PARLOT increases as we scale the applications to more compute nodes. However, the increase is quite small. Going from 16 to 1024 cores, a 64-fold increase in parallelism, only increases the average overhead by between 1.3- and 2.1-fold. In contrast, CALLGRIND's overhead decreases with increasing node count, making it more scalable. Having said that, CALLGRIND's overhead is larger for the C inputs whereas PARLOT's overhead is larger for the smaller B inputs. In other words, PARLOT scales better to larger inputs than CALLGRIND.

PARLOT's scaling behavior can be explained by correlating it with the expected function-call frequency. When distributing a fixed problem size over more cores, each core receives less work. As a consequence, less time is spent in the functions that process the work, resulting in more function calls per time unit, which causes more work for PARLOT. In contrast, when distributing a larger problem size over the same number of cores, each core receives more work. Hence, more time is spent in the functions that process the work, resulting in fewer function calls per time unit, which causes less work for PARLOT and therefore less tracing overhead. Hence, we believe PARLOT's overhead to be even lower on long-running inputs, which is where our tracing technique is needed the most.

In summary, PARLOT's overhead is in the single digits for all evaluated applications and configurations, including for 1024-core runs. It appears to scale reasonably to larger node counts and well to larger problem sizes.

2.5.2 Required Bandwidth

Table 2.2, Fig. 2.4 and Fig. 2.5 show how much trace bandwidth each tool requires during the application execution. On average, PARLOT(M) requires less bandwidth than CALLGRIND, especially for smaller inputs. PARLOT(A)'s bandwidth is much higher as it collects call information from all images and not just the main image like PARLOT(M)

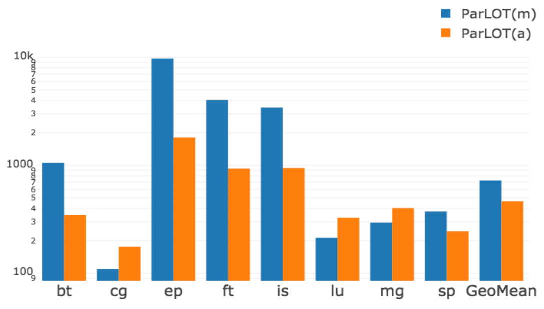


Figure 2.6: Average compression ratio of PARLOT on the NPB applications - Input B

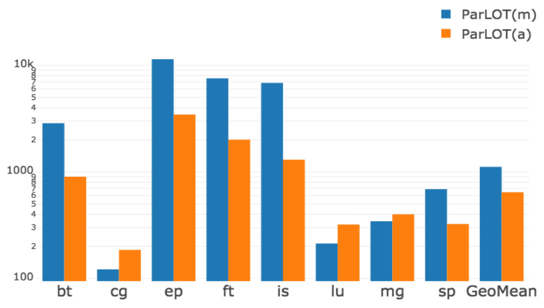


Figure 2.7: Average compression ratio of PARLOT on the NPB applications - Input C

does.

We see that the required bandwidth for different input sizes of the NPB applications are almost equal in PARLOT. According to the NPB documentation, the number of iterations for inputs B and C are the same for all applications. They only differ in the number of elements or the grid size. It is clear that the required bandwidth of PARLOT is independent of the problem size, unlike CALLGRIND, where the input size has a linear impact on the results.

2.5.3 Compression Ratio

Table 2.3 shows the compression ratios for all configurations and inputs. On average, PARLOT stores between half a kilobyte and a kilobyte of trace information in a single byte. We observe that the average compression ratio for PARLOT(A) on input C is 644.3, and its corresponding required bandwidth from Table 2.2 is 56.4 kB/s. This means PARLOT can collect **more than 36 MB** worth of data per core per second while only needing 56 kB/s of the system bandwidth, *leaving the rest of the available bandwidth to the application.* In

Table 2.3: Compression ratio

Input	Tool	# Nodes	bt	cg	ep	ft	is	lu	mg
B	ParLoT(m)	1	3 035.9	94.4	12 456.2	12 173.5	9 718.4	167.7	99.1
		4	586.6	82.5	10 368.4	1 737.1	909.2	140.3	255.0
		16	346.7	113.3	8 563.9	1 077.4	1 200.6	179.0	387.6
		64	252.2	147.8	7 611.0	1 122.6	1 908.0	366.8	437.3
		AVG	1 055.4	109.5	9 749.9	4 027.6	3 434.0	213.5	294.7
	ParLoT(a)	1	514.5	137.4	3 335.8	1 226.7	543.2	314.6	260.9
		4	315.7	137.2	1 266.9	436.2	316.2	287.3	329.6
		16	226.9	181.6	1 246.7	1 026.5	927.1	299.3	469.3
		64	329.2	247.3	1 394.1	1 043.9	1 984.6	410.3	548.5
		AVG	346.6	175.9	1 810.9	933.3	942.8	327.9	402.1
C	ParLoT(m)	1	8 619.0	111.2	13 068.0	21 335.6	21 856.5	350.0	247.4
		4	1 910.6	110.5	12 418.7	6 520.3	2 256.6	112.8	268.0
		16	580.8	133.2	11 017.4	1 239.3	1 347.9	164.5	396.9
		64	322.8	131.9	9 155.0	1 065.1	1 896.3	223.7	465.7
		AVG	2 858.3	121.7	11 414.7	7 540.1	6 839.3	212.7	344.5
	ParLoT(a)	1	2 579.4	181.8	7 377.0	5 143.1	1 520.4	408.2	314.8
		4	448.6	161.3	3 194.6	1 062.9	527.3	274.7	319.4
		16	285.1	185.7	1 765.5	588.9	1 106.3	273.6	467.4
		64	290.0	214.7	1 512.9	1 237.3	2 038.7	329.0	496.2
		AVG	900.8	185.9	3 462.5	2 008.1	1 298.2	321.4	399.4

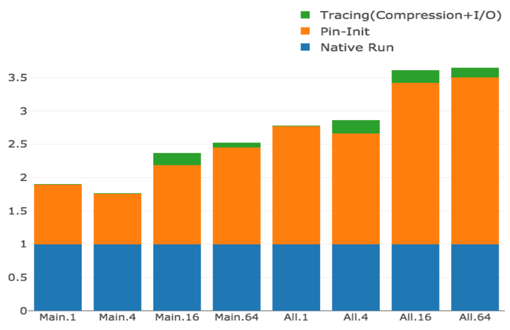


Figure 2.8: Tracing overhead breakdown - Input B

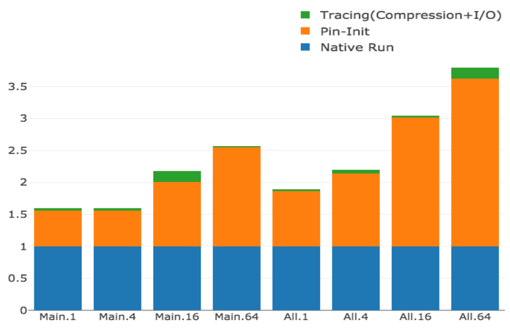


Figure 2.9: Tracing overhead breakdown - Input C

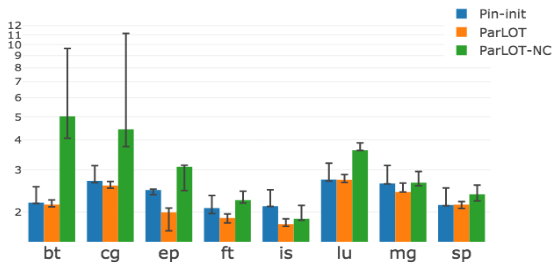


Figure 2.10: Variability of PARLOT(M) overhead on 16 nodes - Input B

comparison, CALLGRIND collects **less than 100 kB** of data but still adds more overhead compared to either PARLOT(A) or PARLOT(M). The average amount of trace data that can be collected by PARLOT(A) is **360x** (85x for PARLOT(M)) larger than that for CALLGRIND. In the best observed case, the compression ratio of PARLOT exceeds 21000. This is particularly impressive because it was achieved with relatively low overhead and incremental on-the-fly compression. Generally, the compression ratios of PARLOT(M) are higher than

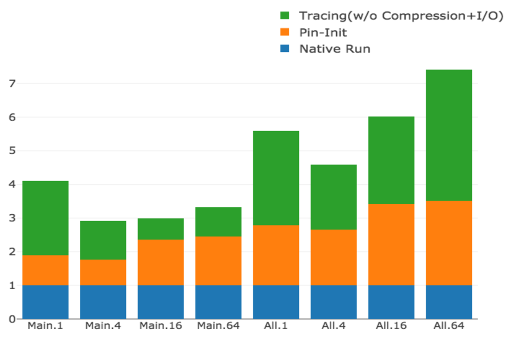


Figure 2.11: PARLOT-NC tracing overhead breakdown - Input B

those of PARLOT(A) because the variety of distinct function calls on the main image is smaller than when tracing all images, thus compression performs better on PARLOT(M). Also by looking at Fig. 2.4, Fig. 2.5, Fig. 2.6 and Fig. 2.7, we find EP to have the highest compression ratio of the NPB applications. At the same time, it has the minimum required bandwidth. The opposite is true for CG, which exhibits the lowest compression ratio and the highest required bandwidth. CG is a conjugate gradient method with irregular memory accesses and communications whereas EP is an embarrassingly parallel random number generator. CG's whole-program trace contains a larger number of distinct calls and more complex patterns than that of EP, thus resulting in a higher bandwidth and lower compression ratio.

PARLOT's compression mechanism works better on larger input sizes because larger inputs tend to result in longer streams of similar function calls (e.g., a call that is made for every processed element).

2.5.4 Overheads

Tables 2.4 and 2.5 present the average overhead added to each application for different versions of PARLOT. The last row of these two tables presents the geometric mean. This information captures how much each phase of PARLOT slows down the native execution.

In general, one expects the following inequality to hold: the overhead of PIN-INIT should be less than that of PARLOT, which should be less than that of PARLOT-NC. This is not always the case because of the non-deterministic runtimes of the applications. In fact, the variability across three runs of each experiment is shown in Fig. 2.10 where we present the minimum, maximum and median overheads. These overheads are for input size B and

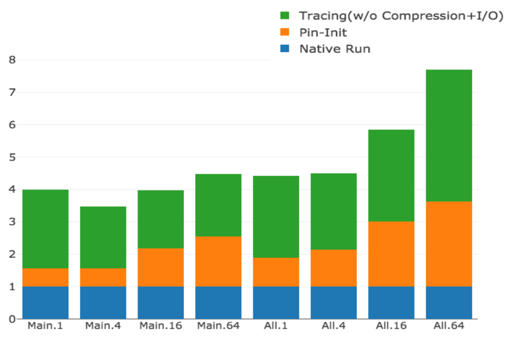


Figure 2.12: PARLOT-NC tracing overhead breakdown - Input C

16 nodes. This variability explains the seeming inconsistencies in Tables 2.4 and 2.5.

On average, PIN-INIT adds an overhead of 3.28 and PARLOT(A) adds an overhead of 3.42. This means that **almost 96% of PARLOT(A)'s overhead is due to PIN**. The results of PARLOT(M) and other inputs follow the same pattern as shown in Fig. 2.8 and 2.9. The overhead that PARLOT (excluding the overhead of PIN-INIT) *adds* to the applications is very small. If we were to switch to a different instrumentation tool that is not as general as PIN but more lightweight, the overhead would potentially reduce drastically.

2.5.5 Compression Impact

Fig. 2.11 and Fig. 2.12 show the overhead breakdown of PARLOT-NC, which illustrate the impact of compression. They also highlight the importance of incorporating compression directly in the tracing tool. On average, PARLOT-NC slows down the application execution almost **2x** more than PARLOT(A). The average overhead across Table 2.5 for PARLOT(A) is **3.4**. The corresponding factor for PARLOT-NC is **6.6**. The numbers of PARLOT(M) and input C follow the same pattern. For example, PARLOT-NC slows down the application execution almost **1.66x** more than PARLOT(M).

Clearly, compression not only lowers the storage requirement but also the overhead. This is important as it shows that the extra computation to perform the compression is more than amortized by the reduction in the amount of data that need to be written out.

This result validates our approach and highlights that incremental, on-the-fly compression is likely essential to make whole-program tracing possible at low overhead.

2.6 Discussion and Conclusion

In this paper, we present ParLoT, a portable low overhead dynamic binary instrumentation-based whole-program tracing approach that can support a variety of dynamic program analyses, including debugging. Key properties of ParLoT include its on-the-fly trace collection and compression that reduces timing jitter, I/O bandwidth, and storage requirements to such a degree that whole-program call/return traces can be collected efficiently even at scale.

We evaluate various versions of ParLoT created by disabling/enabling compression, not collecting any traces, etc. In order to provide an intuitive comparison against a well known tool, we also compare ParLoT to Callgrind. Our metrics include the tracing overhead, required bandwidth, achieved compression ratio, initialization overhead, and the overall impact of compression. Detailed evaluations on the NAS parallel benchmarks running on up to 1024 cores establish the merit of our tool and our design decisions. ParLoT can collect more than 36 MB worth of data per core per second while only needing 56 kB/s of bandwidth and slowing down the application by 2.7x on average. These results are highly promising in terms of supporting whole program tracing and debugging, in particular when considering that most of the overhead is due to the DBI tool and not ParLoT.

The traces collected by ParLoT cut through the entire stack of heterogeneous (MPI, OpenMP, PThreads) calls. This permits a designer to project these traces onto specific APIs of interest during program analysis, visualization, and debugging.

A number of improvements to ParLoT remain to be made. These include allowing users to selectively trace at specific interfaces: doing so can further increase compression efficiency by reducing the variety of function calls to be handled by the compressor. We also discuss the need to bring down initialization overheads, i.e., by switching to a less general-purpose DBI tool.

Acknowledgment

This research was supported by the NSF. We thank our colleague Dr. Hari Sundar from the University of Utah who provided insight and expertise that greatly assisted the research. We also thank the Texas Advanced Computing Center (TACC) and the San

Diego Supercomputer Center (SDSC) for the infrastructure they provided for running our experiments.

Table 2.4: Tracing overhead of versions of PARLOT(M)- Input B

Input: B	Nodes :	1			4			16			64		
	Detail Tools:	PIN-INIT	PARLOT	PARLOT-NC	PIN-INIT	PARLOT	PARLOT-NC	PIN-INIT	PARLOT	PARLOT-NC	PIN-INIT	PARLOT	PARLOT-NC
Main	bt	1.5	1.5	5.6	1.7	1.7	5.0	2.1	2.1	5.0	1.8	2.1	3.5
	cg	1.7	1.8	2.3	1.8	1.8	2.6	2.7	2.5	4.4	2.3	2.1	4.6
	ep	2.9	2.6	20.4	1.9	1.8	5.3	2.4	1.9	3.0	2.6	2.3	2.6
	ft	1.8	2.1	6.1	1.7	1.7	2.7	2.0	1.8	2.2	2.1	1.9	2.1
	is	2.4	2.4	4.8	1.7	1.7	2.0	2.1	1.7	1.8	4.5	4.3	5.7
	lu	1.3	1.3	1.4	1.7	1.7	2.2	2.7	2.7	3.6	3.0	4.3	6.1
	mg	2.5	2.5	2.7	1.5	1.5	1.5	2.6	2.4	2.6	1.9	1.9	1.8
	sp	1.3	1.3	2.4	1.7	1.7	3.5	2.1	2.1	2.3	1.9	2.0	2.5
	GM	1.8	1.9	4.1	1.7	1.7	2.9	2.3	2.1	3.0	2.4	2.5	3.3

Table 2.5: Tracing overhead of versions of PARLOT(A)- Input B

Input: B	Nodes :	1			4			16			64		
	Detail Tools:	PIN-INIT	PARLOT	PARLOT-NC	PIN-INIT	PARLOT	PARLOT-NC	PIN-INIT	PARLOT	PARLOT-NC	PIN-INIT	PARLOT	PARLOT-NC
All	bt	1.7	1.8	6.1	2.3	2.5	6.1	3.2	3.5	9.0	2.8	3.1	7.5
	cg	2.6	2.7	3.8	2.8	3.0	4.4	4.0	4.2	11.3	3.3	3.2	10.3
	ep	4.3	4.1	22.2	3.1	3.4	7.1	3.1	3.3	4.5	4.1	3.8	4.1
	ft	2.8	2.7	6.8	2.6	2.7	3.8	2.8	2.9	3.6	3.1	3.0	3.5
	is	4.4	4.2	7.0	2.8	2.9	3.4	2.9	2.8	3.2	5.3	5.4	8.8
	lu	1.7	1.7	2.3	2.5	2.7	4.8	3.9	4.3	10.4	4.4	4.6	23.4
	mg	4.8	4.7	5.3	2.5	2.7	3.0	4.3	4.4	5.2	2.7	3.1	3.2
	sp	1.7	1.7	3.0	2.4	2.6	5.0	3.2	3.6	5.6	2.7	3.3	11.6
	GM	2.7	2.7	5.5	2.6	2.8	4.5	3.4	3.6	6.0	3.5	3.6	7.4

CHAPTER 3

WHOLE-PROGRAM TRACE ANALYSIS

We present a tool called DiffTrace that approaches debugging via *whole program* tracing and diffing of typical and erroneous traces. After collecting these traces, a user-configurable front-end filters out irrelevant function calls and then summarizes loops in the retained function calls based on state-of-the-art loop extraction algorithms. Information about these loops is inserted into concept lattices, which we use to compute salient dissimilarities to narrow down bugs. DiffTrace is a clean start that addresses debugging features missing in existing approaches. Our experiments on an MPI/OpenMP program called ILCS and initial measurements on LULESH, a DOE miniapp, demonstrate the advantages of the proposed debugging approach.

3.1 Introduction

Debugging high-performance computing code remains a challenge at all levels of scale. Conventional HPC debuggers [4,23] excel at many tasks such as examining the execution state of a complex simulation in detail and allowing the developer to re-execute the program close to the point of failure. However, they do not provide a good understanding of why a program version that worked earlier failed upon upgrade or feature addition. Innovative solutions are needed to highlight the salient differences between two executions in a manner that makes debugging easier as well as more systematic. A recent study conducted under the auspices of the DOE [22] provides a comprehensive survey of existing debugging tools. It classifies them under four software organizations (serial, multithreaded, multi-process, and hybrid), six method types (formal methods, static analysis, dynamic analysis, nondeterminism control, anomaly detection, and parallel debugging), and lists a total of 30 specific tools. Despite this abundance of activity and tools, many significant problems remain to be solved before debugging *can be approached by the HPC community as a collaborative activity* so that HPC developers can extend a common framework.

Almost all debugging approaches seek to find outliers ("unexpected executions") amongst thousands of running processes and threads. The approach taken by most existing tools is to look for symptoms in a specific bug-class that they cover. Unfortunately, this approach calls for a programmer having a good guess of what the underlying problem might be, and to then pick the right set of tools to deploy. If the guess is wrong, the programmer has no choice but to refine their guess and look for bugs in another class, re-executing the application and hoping for better luck with another tool. This iterative loop of re-execution followed by applying a best-guess tool for the suspected bug class can potentially consume large amounts of execution cycles and wastes an expert developer's time. More glaring is the fact that these tools must recreate the execution traces yet again: they do not have means to hand off these traces to another tool or cooperate in symbiotic ways.

We cannot collect all relevant pieces of information necessary to detect all possible bug classes such as resource leaks, deadlocks, and data races. Each such bug requires its attributes to be kept. Also, debugging is not fully automatable (it is an undecidable problem in general) and must involve human thinking: at least to reconcile what is observed against the deeper application-level semantics. However, (1) we believe that it is still possible to collect one standard set of data and use it to make an initial triage in such a way that it can guide a later, deeper debugging phase to locate which of the finer bug gradations (e.g., resource leaks or races) brought the application down. Also, (2) we believe that it is possible to engage the human *with respect to understanding structured presentations of information*.

Our DiffTrace framework addresses both issues. The common set of data it uses is a *whole program function call trace* collected per process/thread. DiffTrace relies on novel ways to diff a normal trace and a fault-laden trace to guide the debugging engineer closer to the bug. While our work has not (yet) addressed situations in which millions of threads and thousands of processes run for days before they produce an error, we strongly believe that we can get there once we understand the pros and cons of our initial implementation of the DiffTrace tool, which is described in this paper. The second issue is handled in DiffTrace by offering a novel collection of modalities for understanding program execution diffs. We now elaborate on these points by addressing the following three problems.

3.1.0.0.1 Problem 1 – Collecting Whole-Program Heterogeneous Function-Call Traces Efficiently Not only must we have the ability to record function calls and returns at one API such as MPI, increasingly we must collect calls/returns at multiple interfaces (e.g., OpenMP, PThreads, and even inner levels such as TCP). The growing use of heterogeneous parallelization necessitates that we understand MPI and OpenMP activities (for example) to locate cross-API bugs that are often missed by other tools. Sometimes, these APIs contain the actual error (as opposed to the user code), and it would be attractive to have this debugging ability.

Solution to Problem 1: In DiffTrace, we choose Pin-based whole program binary tracing, with tracing filters that allow the designer to collect a suitable mixture of API calls/returns. We realize this facility using ParLOT, a tool designed by us and published earlier [58]. In our research, we have thus far demonstrated the advantage of ParLOT with respect to collecting both MPI and OpenMP traces from a *single run of a hybrid MPI/OpenMP program*. We demonstrate that, from this single type of trace, it is possible to pick out MPI-level bugs and/or OpenMP-level bugs. While whole-program tracing may sound extremely computation and storage intensive, ParLOT employs lightweight on-the-fly compression techniques to keep these overheads low. It achieves compression ratios exceeding 21,000 [58], thus making this approach practical, demanding only a few kilobytes per second per core of bandwidth.

3.1.0.0.2 Problem 2 – Need to Generalize Techniques for Outlier Detection Given that outlier detection is central to debugging, it is essential to use efficient representations of the traces to be able to systematically compute *distances* between them without involving human reasoning. The representation must also be versatile enough to be able to "diff" the traces with respect to *an extensible number of vantage points*. These vantage points could be diffing traces concerning process-level activities, thread-level activities, a combination thereof, or even finite sequences of process/thread calls (say, to locate *changes* in caller/callee relationships).

Solution to Problem 2: DiffTrace employs *concept lattices* to amalgamate the collected traces. Concept lattices have previously been employed in HPC to perform structural clustering of process behaviors [60] to present performance data more meaningfully to users. The authors of that paper use the notion of *Jaccard distances* to cluster performance results that are closely related to process structures (determined based on caller/callee relationships). In DiffTrace, we employ incremental algorithms for building and maintaining concept lattices from the ParLOT-collected traces. In addition to Jaccard distances, in our work, we also perform hierarchical clustering of traces and provide a tunable threshold for outlier detection. We believe that these uses of concept lattices and refinement approaches for outlier detection are new in HPC debugging.

3.1.0.0.3 Problem 3 – Loop Summarization Most programs spend most of their time in loops. Therefore, it is important to employ state-of-the-art algorithms for loop extraction from execution traces. It is also important to be able to diff two executions with respect to changes in their looping behaviors. In our experience, presenting such changes using good visual metaphors tends to highlight many bug types immediately.

Solution to Problem 3: DiffTrace utilizes the rigorous notion of Nested Loop Representations (NLRs) for summarizing traces and representing loops. Each repetitive loop structure is given an identifier, and nested loops are expressed as repetitions of this identifier exponentiated (as with regular expressions). This approach to summarizing loops can help manifest bugs where the program does not hang or crash but nevertheless runs differently in a manner that informs the developer engaged in debugging.

Organization: §**??** illustrates the contributions of this paper on a simple example. §**??** presents the algorithms underlying DiffTrace in more detail. §**??** summaries the experimental methodology before showing a medium-sized case study involving MPI and OpenMP. §**??** shows initial measurements and examples on LULESH [**?**], a DOE common mini app. §**??** summarizes selected related works. §**??** concludes the paper with a discussion.

3.2 DiffTrace Overview

3.2.1 High-level Overview

DiffTrace employs ParLOT's whole-program function-call and return trace-collection mechanism, where ParLOT captures traces via Pin [39] and incrementally compresses them using a new compression scheme [58]. ParLOT can capture functions at two levels: the *main image* (which does not include library code) and *all images* (including all library code). As the application runs, ParLOT generates per-thread trace files that contain the compressed sequence of the IDs of the executed functions. The compression mechanism is light-weight yet effective, thus reducing not only the required bandwidth and storage but also the runtime relative to not compressing the traces. As a result, ParLOT can capture whole-program traces at low overhead while leaving most of the disk bandwidth to the application. Using whole-program traces substantially reduces the number of overall debug iterations because it allows us to repeatedly analyze the traces offline with different filters.

Figure 3.1 provides an overview of the DiffTrace toolchain in terms of the blue flows (fault-free) and red flows (faulty). In a broad sense, code-level faults in HPC applications (e.g., the use of wrong subscripts) turn into observable code-level misbehaviors (e.g., an unexpected number of loop iterations), many of which turn into application-level issues. In our study of DiffTrace, we evaluate success merely in terms of the efficacy of observing these misbehaviors in response to injected code-level faults (we rely on a rudimentary fault injection framework complemented by manual fault injection).

The preprocessing stage removes calls/returns at the ignored APIs. The nested loop recognition (NLR) mechanism then extracts loops from traces. The resulting information not only serves as a lossless abstraction to ease the rest of the trace analysis but also serves as a *per-thread measure of progress*. The FCA (Formal Concept Analysis) stage conducts a systematic way to arrange objects (in our case threads) and attributes (we support a rich collection of attributes including the set of function calls a thread makes, the set of *pairs* of function calls made—this reflects calling context—etc.). Weber et al.'s work [17, 60] employs FCA exactly in this manner (including the use of pairs of calls), however, for grouping performance information. Our new contribution is showing that FCA can play a central role in debugging HPC applications.

While faults induce asymmetries ("aberrations") in program behaviors, one cannot locate faults merely by locating the asymmetries in an overall collection of process traces. The reason is that even in a collection of MPI processes or threads within these processes, some processes/threads may serve as a master while others serve as workers [**?**]. Thus, we must have a base level of similarities computed even for normal behaviors and then compute how *this similarity relation changes* when faults are introduced. This is highlighted by the blue and red rectangular patches in Figure 3.1 that, respectively, iconify the *Jaccard similarity matrices* computed for the normal behavior (above) and the erroneous behavior (below). This is shown as the "diff Jaccard similarity matrix" in greyscale at the juncture of JSM_{normal} and JSM_{faulty}.

After the JSM_D matrix is computed, we invoke a hierarchical clustering algorithm that computes the "B-score" and helps rank suspicious traces/processes. The diffNLR representation is then extracted. Intuitively, this is a diff of the loop structures of the normal and abnormal threads/processes. This diagram shows (as with git diff and text diff) a *main stem* comprised of green rectangles ("common looping structure") and red/blue *diff rectangles* showing how the loop structures of the normal and erroneous threads differ with respect to the main stem. We show that this presentation often helps the debugging engineer locate the faults.

Last but not least, we strongly believe that a framework such as DiffTrace can serve as an important HPC community resource. Each debugging tool designer who uses DiffTrace can extend it by incorporating new attributes and clustering methods, but otherwise retain the overall tool structure. Such a "playground" for developing and exploring new methods for debugging does not exist in HPC. There is also the intriguing possibility that many of the 30-odd tools mentioned in §**??** *can be made to focus on the problems highlighted by diffNLR*, thus gaining efficiency (this will be part of our future work).

In this paper, we describe DiffTrace as a *relative debugging* [52] tool, in that bugs are caught with respect to JSM_D which is a *change* from the previous code version found working. However, many types of faults may be apparent just by analyzing JSM_{faulty}: for instance, processes whose execution got truncated will look highly dissimilar to those that terminated normally. In those use cases of DiffTrace, the B-score based ranking can then be made on JSM_{faulty} directly.

3.2.2 Example Walk-through

We now employ Figure 3.2—a textbook MPI odd/even sorting example—to illustrate DiffTrace. Odd/even sorting is a parallel variant of bubble sort and operates in two alternating phases: in the *even phase*, the even processes exchange (conditionally swap) values with their right neighbors, and in the *odd phase*, the odd processes exchange values with their right neighbors.

A waiting trap in this example is this: the user may have swapped the `Recv; Send` order in the `else` part, creating head-to-head ``‘‘Send || Send’’`` deadlock under low-buffering (MPI EAGER limit). We will now show how DiffTrace helps pick out this root-cause.

3.2.3 Pre-processing

Using ParLOT's decoder, each trace is first decompressed. Next, the desired functions are extracted based on predefined (Table 3.1) or custom regular expressions (i.e., *filters*) and kept for later phases. Table 3.2 shows the pre-processed traces (T_i) of odd/even sort with four processes. T_i is the trace that stores the function calls of process i.

3.2.4 Nested Loop Representation

Virtually all dynamic statements are found within loops. Function calls within a loop body yield *repetitive patterns* in ParLOT traces. Inspired by ideas for the detection of repetitive patterns in strings [44] and other data structures [31], we have adapted the Nested Loop Recognition (NLR) algorithm by Ketterlin et al. [32] to detect repetitive patterns in ParLOT traces (cf. Section 3.3.1). Detecting such patterns can be used to measure the progress of each thread, revealing unfinished or broken loops that may be the consequence of a fault.

For example, the loop in line 3 of `oddEvenSort()` (Figure 3.2) iterates four times when run with four processes. Thus each T_i contains four occurrences of either [`MPI_Send-MPI_Recv`] (even i) or [`MPI_Recv-MPI_Send`] (odd i). By keeping only MPI functions and converting each T_i into its equivalent NLR, Table 3.2 can be reduced to Table 3.3 where **L0** and **L1** represent the *loop body* [`MPI_Send-MPI_Recv`] and [`MPI_Recv-MPI_Send`], respectively. The integer after the ˆ symbol in NLR represents the *loop iteration count*. Note that, since the first and last processes only have one-way communication with their neighbors, T_0 and T_3 perform only half as many iterations.

3.2.5 Hierarchical Clustering via FCA

Processes in HPC applications are known to fall into predictable equivalence classes. The widely used and highly successful STAT tool [3] owes most of its success for being able to efficiently collect stack traces (nested sequences of function calls), organize them as prefix-trees, and equivalence the processes into teams that evolve in different ways.

Figure 3.1: DiffTrace Overview

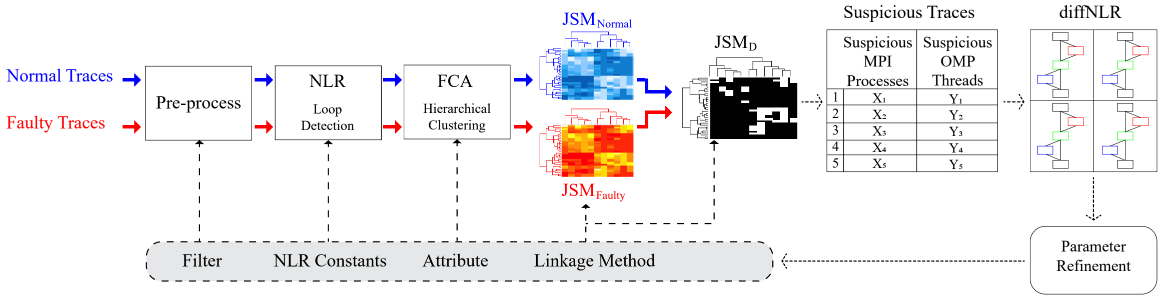
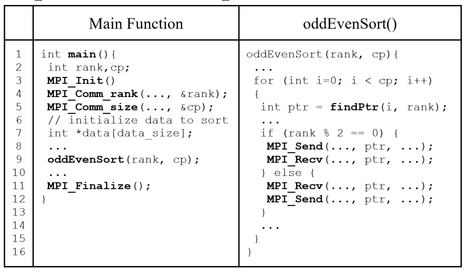


Figure 3.2: Simplified MPI implementation of Odd/Even Sort



Main Function

```
int main(){
 int rank,cp;
 MPI_Init()
 MPI_Comm_rank(..., &rank);
 MPI_Comm_size(..., &cp);
 // initialize data to sort
 int *data[data_size];
 ...
 oddEvenSort(rank, cp);
 ...
 MPI_Finalize();
}
```

oddEvenSort()

```
oddEvenSort(rank, cp){
 ...
 for (int i=0; i < cp; i++)
 {
  int ptr = findPtr(i, rank);
  ...
  if (rank % 2 == 0) {
   MPI_Send(..., ptr, ...);
   MPI_Recv(..., ptr, ...);
  } else {
   MPI_Recv(..., ptr, ...);
   MPI_Send(..., ptr, ...);
  }
  ...
 }
}
```

Table 3.1: Pre-defined Filters

Category	**Sub-Category**	**Description**
Primary	Returns	Filter out all returns
	PLT	Filter out the ".plt" function calls for external functions/procedures that their address needs to be resolved dynamically from Procedure Linkage Table (PLT)
MPI	MPI All	Only keep functions that start with "MPI_"
	MPI Collectives	Only keep MPI collective calls (MPI_Barrier, MPI_Allreduce, etc)
	MPI Send/Recv	Only keep MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv and MPI_Wait
	MPI Internal Library	Keep all inner MPI library calls
OMP	OMP All	Only keep OMP calls (starting with GOMP_)
	OMP Critical	Only keep OMP_CRITICAL_START and OMP_CRITICAL_END
	OMP Mutex	Only keep OMP_Mutex calls
System	Memory	Keep any memory related functions (memcpy, memchk, alloc, malloc, etc)
	Network	Keep any network related functions (network, tcp, sched, etc)
	Poll	Keep any poll related functions (poll, yield, sched, etc)
	String	Keep any string related functions (strlen, strcpy, etc)
Advanced	Custom	Any regular expression can be captured
	Everything	Does not filter anything

Table 3.2: The generated traces for odd/even execution with four processes

T_0	T_1	T_2	T_3
...	...	...	...
main	main	main	main
MPI_Init	MPI_Init	MPI_Init	MPI_Init
MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank
MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size
...	...	...	...
oddEvenSort	oddEvenSort	oddEvenSort	oddEvenSort
...	...	...	...
findPtr	findPtr	findPtr	findPtr
MPI_Send	MPI_Recv	MPI_Send	MPI_Recv
MPI_Recv	MPI_Send	MPI_Recv	MPI_Send
...	...	...	...
findPtr	findPtr	findPtr	findPtr
MPI_Send	MPI_Recv	MPI_Send	MPI_Recv
MPI_Recv	MPI_Send	MPI_Recv	MPI_Send
...	...	...	...
MPI_Finalize	MPI_Finalize	MPI_Finalize	MPI_Finalize

Table 3.3: NLR of Traces

T_0	T_1	T_2	T_3
MPI_Init	MPI_Init	MPI_Init	MPI_Init
MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank	MPI_Comm_Rank
MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size	MPI_Comm_Size
L0 ˆ 2	L1 ˆ 4	L0 ˆ 4	L1 ˆ 2
MPI_Finalize	MPI_Finalize	MPI_Finalize	MPI_Finalize

Table 3.4: Formal Context of odd/even sort example

	MPI_Init()	MPI_Comm_Size()	MPI_Comm_Rank()	L0	L1	MPI_Finalize()
Trace 0	×	×	×	×		×
Trace 1	×	×	×		×	×
Trace 2	×	×	×	×		×
Trace 3	×	×	×		×	×

Coalesced stack trace graphs (CSTG, [13]) have proven effective in locating bugs within Uintah [**?**] and perform stat-like equivalence class formation, albeit with the added detail of maintaining calling contexts. Inspired by these ideas, FCA-based clustering provides the next logical level of refinement in the sense that (1) we can pick any of the multiple attributes one can mine from traces (e.g., pairs of function calls, memory regions accessed by processes, locks held by threads, etc.), and (2) form this equivalencing relation quite naturally by computing the Jaccard distance between processes/threads. In general, such a classification is powerful enough to distinguish structurally different threads from one another (e.g., MPI processes from OpenMP threads in hybrid MPI+OpenMP applications) and reduce the search space for bug location to a few representative classes of traces that are distinctly dissimilar.[1]

A *formal context* is a triple $K = (G, M, I)$ where G is a set of **objects**, M is a set of **attributes**, and $I \subseteq G \times M$ is an incidence relation that expresses *which objects have which attributes*. Table 3.4 shows the formal context of the preprocessed odd/even-sort traces. We can employ as attributes either the function calls themselves or the detected loop bodies (each detected loop is assigned a unique ID, and one can diff with respect to these IDs). The context shows that all traces include the functions MPI_Init(), MPI_Comm_size(), MPI_Comm_rank() and MPI_Finalize(). The even traces contain the loop *L0* and the odd traces the loop *L1*.

Figure 3.3 shows the concept lattice derived from the formal context in Table 3.4 and is interpreted as follows:

- The top node indicates that all traces share MPI_Init(), MPI_Comm_size(), MPI_Comm_rank() and MPI_Finalize().
- The bottom node signifies that none of the traces share all attributes.
- The middle nodes show that T_0 and T_2 are different from T_1 and T_3.

The complete pairwise Jaccard Similarity Matrix (JSM) can easily be computed from concept lattices. For large-scale executions with thousands of threads, it is imperative to employ incremental algorithms to construct concept lattices (detailed in Section 3.3.2). Figure 3.4 shows the heatmap of the JSM obtained from the concept lattice in Figure 3.3.

[1]As emphasized earlier, we perform "sky subtraction" as in astronomy to locate comets; in our case, we diff the diffs, which is captured in JSM$_D$.

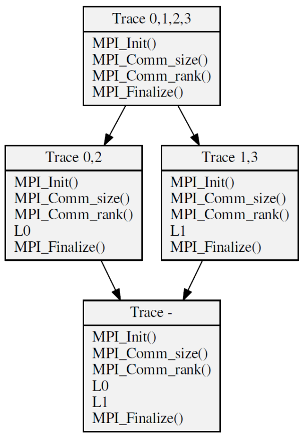


Figure 3.3: Sample Concept Lattice from Object-Attribute Context in Table 3.4

DiffTrace uses the JSM to form equivalence classes of traces by hierarchical clustering. Next, we show how the differences between two hierarchical clusterings from two executions (faulty vs. normal) reveal which traces have been affected the most by the fault.

3.2.6 Detecting Suspicious Traces via JSM$_D$

JSM$_{normal}$[i][j] (JSM$_{faulty}$[i][j]) shows the Jaccard similarity score of T_i and T_j from the normal trace (T'_i and T'_j). As explained earlier, we compute JSM$_D$ to detect outlier executions, where JSM$_D$ = $|$JSM$_{faulty}-$JSM$_{normal}|$.

We sort the suggestion table based on the *B-score* similarity metric of two hierarchical clusterings [14] (cf. Section 3.3.3). A single iteration through the DiffTrace loop (with a single set of parameters shown as a dashed box in Figure 3.1) may still not detect the root-cause of a bug. The user can then (1) alter the linkage method employed in computing the hierarchical clustering (reorder the dendrograms built to achieve the clustering), (2) alter the FCA attributes, (3) adjust the NLR constants (loops are extracted with realistic complexity by observing repetitive patterns inside a preallocated buffer), and/or (4) the front-end filters. This is shown in the iterative loop in Figure 3.1.

3.2.7 Evaluation

To evaluate the effectiveness of DiffJSM, we planted two artificial bugs (*swapBug* and *dlBug*) in the code from Figure 3.2 and ran it with 16 processes. *swapBug* swaps the order of MPI_Send and MPI_Recv in rank 5 after the seventh iteration of the loop in line 3 of `oddEvenSort`, simulating a potential deadlock. *dlBug* simulates an actual deadlock in the

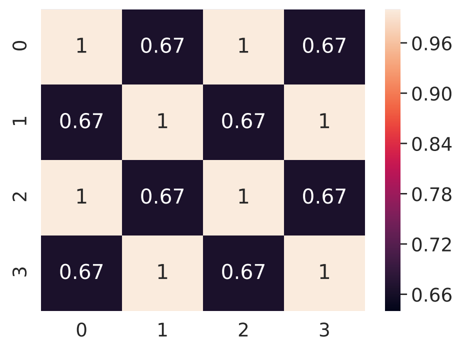
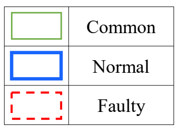
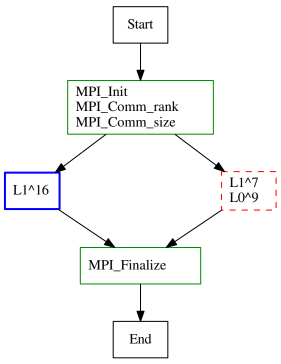


Figure 3.4: Pairwise Jaccard Similarity Matrix (JSM) of MPI Processes in Sample Code



(a) Legend



(b) swapBug

Figure 3.5: diffNLR Example

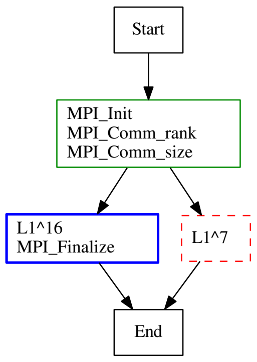


Figure 3.6: dlBug

same location (rank 5 after the seventh iteration). Upon collection of ParLOT traces from the execution of the buggy code versions, DiffTrace first decompresses them and filters out all non-MPI functions. Then two major loops are detected, **L0** and **L1** (identical to the ones in Table 3.3), that are supposed to loop 16 times in the even and odd traces, respectively (except for the first and last traces, which loop just eight times).

After constructing concept lattices and their corresponding JSMs, trace 5 appears as the trace that got affected the most by the bugs because row 5 (showing the similarity score of T_5 relative to all other traces) (JSM$_{normal}$[5][i] for $i \in [0, 16)$) changed the most after the bug was introduced. The differences between the suggested suspicious trace (T_s') and its corresponding normal trace (T_s) is visualized by *diffNLR*.

3.2.7.1 diffNLR

To highlight the differences in an easy-to-understand manner, DiffTrace visually separates the common and different blocks of a pair of pre-processed traces via *diffNLR*, a graphical visualization of the `diff` algorithm [43].

`diff` takes two sequences S_A and S_B and computes the minimal *edit* to convert S_A to S_B. This algorithm is used in the GNU `diff` utility to compare two text files and in git for efficiently keeping track of file changes. Since ParLOT preserves the order of function calls, each trace T_i is totally ordered. Thus *diff* can expose the differences of a pair of Ts. *diffNLR* aligns common and different blocks of a pair of sequences (e.g., traces) horizontally and vertically, making it easier for the analyst to see the differences at a glance. For simplicity, our implementation of *gdiff* only takes one argument x that denotes *the suspicious trace*.

diffNLR$(x) \equiv$ diffNLR(T_x, T'_x) where T_x is the trace of thread/process x of a normal execution and T'_x is the corresponding trace of the faulty execution.

Figure 3.5b shows the diffNLR(5) of *swapBug* where T_5 iterates over the loop [MPI_Recv - MPI_Send] 16 times (L1^16) after the MPI initialization while the order swap is well reflected in T'_5 (L1^7 - L0^9). Both processes seem to terminate fine by executing MPI_Finalize(). However, diffNLR(5) of *dlBug* (Figure 3.6) shows that, while T_5 executed MPI_Finalize, T'_5 got stuck after executing L1 seven times and never reached MPI_Finalize.

This example illustrates how our approach can locate the part of each execution that was impacted by a fault. Having an understanding of *how the application should behave normally* can reduce the number of iterations by picking the right set of parameters sooner.

3.3 Algorithms Underlying DiffTrace

3.3.1 Nested Loop Recognition (NLR)

We build NLRs based on the work by Ketterlin and Clauss [32], who use this algorithm for trace compression, and the work by Kobayashi and MacDougall [34], who propose a similar bottom-up strategy to build loop nests from traces, replacing each recognized loop with a new symbol. We adapt these algorithms to function-call traces wherein we record identical loops at different locations by introducing a single new (made-up) function ID that represents the entire loop. This process is restarted once the whole trace has been analyzed for depth-2 loops and so on until a function-ID replacement is performed. DiffTrace-NLR works by incrementally pushing trace entries (function IDs) onto a stack of *elements* (i.e., function IDs representing detected loop structures). Whenever an element is pushed onto the stack S, the upper elements of the stack are recursively examined for potential loop detection or loop extensions (Procedure 1).

We store all distinct loop bodies (LBs) in a hash-table, assigning each a unique ID, which can be applied as a heuristic to detect loops not only in the current trace but also in other traces of the same execution. The maximum length of the subsequences to examine is decided by a fixed K. The complexity of the NLR algorithm is $\Theta(K^2N)$ where N is the size of the input. While loop detection has been researched in other contexts, its use to support debugging is believed to be novel.

```
Reduce($S$):
    for $i$ : 1 ... $3K$ do
        $b = i/3$
        if Top 3 $b$-long elements of $S$ are isomorphic then
            pop $i$ elements from $S$
            $LB = S[b:1]$, $LC = 3$
            $LS = (LB, LC)$
            push $LS$ to $S$
            add $LB$ to the Loop Table
            Reduce($S$)
        end
        if $S[i]$ is a loop (LS) and $S[i-1:1]$ isomorphic to its loop body$LB$ then
            $LC = LC + 1$
            pop $i - 1$ elements from $S$
            Reduce($S$)
        end
    end
```

Procedure 1: `Reduce` procedure adapted from the NLR algorithm

3.3.2 Concept Lattice Construction

The efficiency of algorithms for concept lattice construction depends on the sparseness of the formal context [35]. Ganter's *Next Closure* algorithm [17] constructs the lattice from a *batch* of contexts and requires the whole context to be present in main memory and is, therefore, inefficient for long HPC traces.

We have implemented Godin's *incremental* algorithm [21] to extract attributes (Table 3.5) from each trace (object) and inject them into an initially empty lattice. Notice that our representation already includes compression of the attributes as (1) either the observed frequency is recorded, (2) the log10 of the frequency is recorded, or (3) "no frequency" (presence/absence) of a function call is recorded. *These are versatile knobs to adjust for bug-location and similarity calculation.*

Every time a new object with its set of attributes is added to the lattice, an *update* procedure minimally modifies/adds/deletes edges and nodes of the lattice. The extracted attributes are in the form {*attr:freq*}. *attr* is either a single entry of the trace NLR or a consecutive pair of entries. *freq* is a parameter to adjust the impact of the frequency of each *attr* in the concept lattice. The complexity of Godin's algorithm is $O(2^{2K}|G|)$, where K is an upper bound for the number of attributes (e.g., distinct function calls in the whole execution) and $|G|$ is the number of objects (e.g., the number of traces).

3.3.3 Hierarchical Clustering, Construction, and Comparison

DiffJSMs provide pair-wise dissimilarity measurements that can be used to combine traces (forming initial clusters). To obtain outliers (suspicious traces), we form dendrograms for which a *linkage* function is required to measure the distance between sets of traces. We currently employ SciPy (version 1.3.0. [29]) for these tasks. SciPy provides a wide range of linkage functions such as single, complete, average, weighted, centroid, median, and ward.

3.3.3.1 Ranking Table

As shown in Figure 3.1, each component of DiffTrace has some tunable parameters and constants, and the suggested suspicious traces are a function of them. Thus, a metric is needed to serve as the sorting key of the suspicious traces. Each parameter combination, in essence, creates a different DiffJSM, giving us "the distance between two hierarchical clusterings". Fowlkes et al. [14] proposed a method for comparing two hierarchical clusterings by computing their *B-score*. While we have not evaluated the full relevance of this idea, our initial experiments show that sorting suspicious traces based on the B-score of DiffJSMs is effective and brings interesting outliers to attention.

3.4 Case Study: ILCS

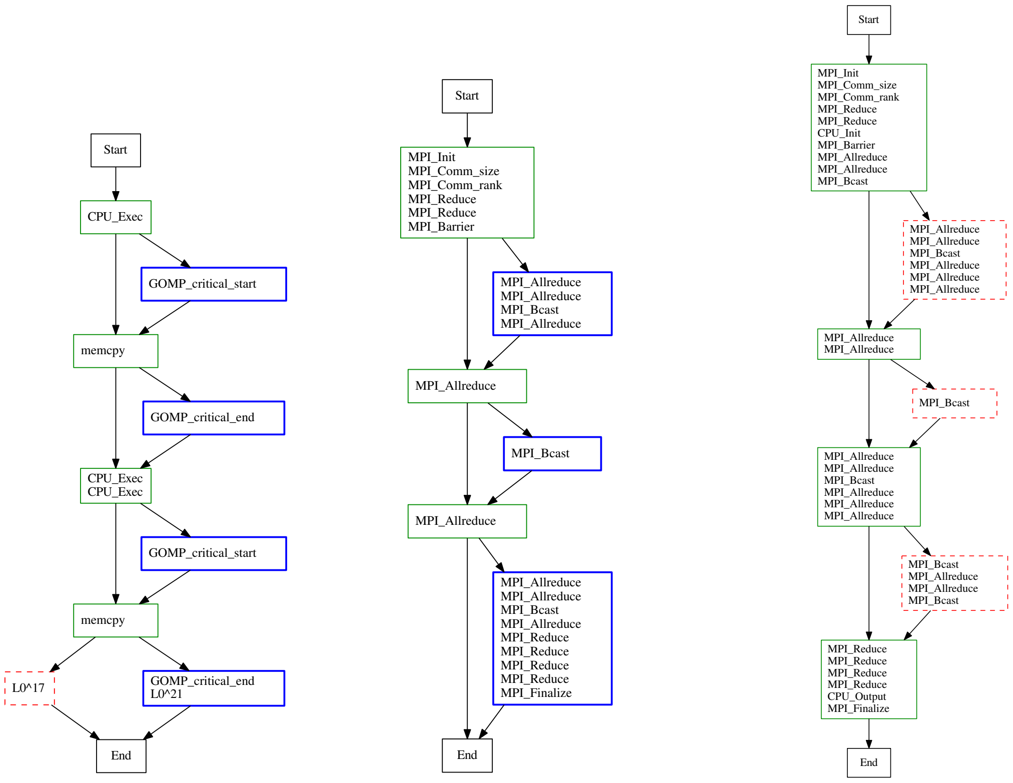
ILCS is a scalable framework for running iterative local searches on HPC platforms [9]. Providing serial CPU and/or single-GPU code, ILCS executes this code in parallel between compute nodes (MPI) and within them (OpenMP and CUDA).

To evaluate DiffTrace, we manually injected MPI-level and OMP-level bugs into the Traveling Salesman Problem (TSP) running on ILCS (Listing 3.1). The injected bugs simulate real HPC bugs such as deadlocks. Moreover, we inserted "hidden" faults that do not crash the program such as violations of critical sections and semantic bugs. The goal was to see how effectively DiffTrace can analyze the resulting traces and how close it can get to the root cause of the fault. .

```
main(argc, argv) {
 ... // initialization
 MPI_Init();
 MPI_Comm_size();
 MPI_Comm_rank(my_rank);
 ... // Obtain number of local CPUs and GPUs
 MPI_Reduce(lCPUs, gCPUs, MPI_SUM); // Total # of CPUs
 MPI_Reduce(lGPUs, gGPUs, MPI_SUM); // Total # of GPUs
```

Table 3.5: Attributes mined from traces

Attributes {attr:freq}			
attr		freq	
Single	each entry of the trace	**Actual**	observed frequency
		Log10	log10 of the observed frequency
Double	each pair of consecutive entries	**noFreq**	no frequency



(a) diffNLR(6.4) (b) diffNLR(4) (c) diffNLR(5)

Figure 3.7: Three diffNLR outputs

```
 champSize = CPU_Init();
 ... // Memory allocation for storing local and global champions w.r.t. champSize
 MPI_Barrier();
 #pragma omp parallel num_threads(lCPUs+1)
 {rank = omp_get_thread_num();
  if (rank != 0) { // worker threads
   while (cont) {
    ... // calculate seed
    local_result = CPU_Exec();
    if (local_result < champ[rank]) { // update local champion
     #pragma omp critical
     memcpy(champ[rank], local_result);}}
  } else { //master thread
   do {
    ...
    MPI_AllReduce(); //broadcast the global champion
    ...
    MPI_AllReduce(); //broadcast the global champion P_id
    ...
    if (my_rank == global_champion_P_id) {
     #pragma omp critical
     memcpy(bcast_buffer, champ[rank]);
    }
    MPI_Bcast(bcast_buffer); // broadcast the local champion to all nodes
   } while (no_change_threshold);
   cont = 0; // signal worker threads to terminate
  }}
 if (my_rank == 0) {CPU_Output(champ);}
 MPI_Finalize();}

/* User code for TSP problem */
CPU_Init() {/* Read coordinates, calculate distances, initialize champion structure,
    return structure size */}
CPU_Exec() {/* Find local champions (TSP tours) */}
CPU_Output() {/* Output champion */}
```

Listing 3.1: ILCS Overview

We collected ParLOT (main image) traces from the execution of ILCS-TSP with 8 MPI processes and 4 OpenMP threads per process on the XSEDE-PSC Bridges supercomputer whose compute nodes have 128 GB of main memory and contain 2 Intel Haswell (E5-2695 v3) CPUs with 14 cores each running at 2.3 - 3.3 GHz. Note that we did not provide any GPU code to ILCS.

The collected traces (faulty and normal) are fed to DiffTrace. We enabled the MPI, OpenMP, and custom (ILCS-TSP user code) filters and set the NLR constant K to 10 for all experiments. The current version of DiffTrace is implemented and built using C++ GCC 5.5.0, Pin 3.8, Python 2.7, and Scipy 1.3.0.

We present the results in the form of ranking tables that show which traces (processes and threads) DiffTrace considers "suspicious". Since DiffTrace output is highly dependent to "parameters", each row in ranking tables starts with parameters whom the suspicious traces are the result of.

The linkage method that converts JSMs to flat clustering is "ward" for all of the top reported suspicious traces that we removed from tables for better readability. Ward linkage function in SciPy uses *Ward variance minimization algorithm* to calculate the distance between newly formed clusters [29]. Furthermore, we show diffNLRs for selected traces.

3.4.1 ILCS-TSP Workflow

The TSP code starts with a random tour and iteratively shortens it using the 2-opt improvement heuristic [28] until a local minimum is reached. ILCS automatically and asynchronously distributes unique seed values to each worker thread, runs the TSP code, reduces the results to find the best solution, and repeats these steps until the termination criterion is met. It employs two types of threads per node: a *master* thread (MPI process) that handles the communication and local work distribution and a set of *worker* threads (OpenMP threads) that execute the provided TSP code. The master thread forks a worker thread for each detected CPU core. Each worker thread continually calls `CPU_Exec()` to evaluate a seed and records the result (lines 14-20). Once the worker threads are running, the master thread's primary job is to scan the results of the workers to find the best solution computed so far (i.e., the local champion). This information is then globally reduced to determine the current system-wide champion (lines 22-32). ILCS terminates the search when the quality has not improved over a certain period (lines 33-34).

3.4.2 OpenMP Bug: Unprotected Memory Access

The memory accesses performed by the `memcpy` calls on lines 20 and 30 are protected by an OpenMP critical section. Not protecting them results in a data race that might lead to incorrect final program output. To simulate this scenario, we modified the ILCS source code to omit the critical section in worker thread 4 of process 6.

Table 3.6 lists the top suspicious traces that DiffTrace finds when injecting this bug. Each row presents the results for different filters and attributes. For example, the filter "11.mem.ompcit.cust.0K10" removes all function returns and .plt calls from the traces and only keeps memory-related calls, OpenMP critical-section functions, and the custom function "CPU_Exec". The "K10" at the end of filter means that the filtered traces are converted into an NLR with K=10. The bold numbers in the rightmost column of the table flag trace 6.4 (i.e., process 6, thread 4) as the trace that was affected the most by the bug.

The corresponding diffNLR(6.4) presented in Figure 3.7a clearly shows that the normal execution of ILCS (green and blue blocks) protects the `memcpy` while the buggy execution (green and red blocks) does not. Here, L0 represents `CPU_Exec`, which is called multiple times in both the fault-free and the buggy version (the call frequencies are different due to the asynchronous nature of ILCS).

3.4.3 MPI Bug: Deadlock Caused by Fault in Collective

By forcing process 2 to invoke MPI_Allreduce (line 24) with a wrong size, we can inject a *real deadlock*. Because the deadlock happens early in the execution, the resulting traces are very different from their fault-free counterparts. Consequently, DiffTrace marks almost all processes as suspicious (cf. Table 3.7). Clearly, this is not helpful for debugging. Nevertheless, diffNLR still yields useful information. Since most of the traces are suspicious, we do not know which one the real culprit is and randomly selected trace 4. By looking at the diffNLR(4) output shown in Figure 3.7b, we immediately see that both the normal and the buggy trace are identical up to the invocation of MPI_Allreduce. This gives the user the first (correct) hint as to where the problem lies. Beyond this point, the bug-free process continues to the end of the program (it reaches the MPI_Finalize call) whereas the buggy process does not. The last entry in the buggy trace is a call to MPI_Allreduce (the last green box), indicating that this call never returned, that is, it deadlocked. This provides the user with the second (correct) hint as to the type of the underlying bug.

3.4.4 MPI Bug: Wrong Collective Operation

By changing the MPI_MIN argument to MPI_MAX in the MPI_Allreduce call on line 24 of Listing 3.1, the semantics of ILCS change. Instead of computing the best answer, the modified code computes the worst answer. Hence, this code variation terminates but is likely to yield the wrong result. We injected this bug into process 0.

The first few suspicious processes listed in Table 3.8 are inconclusive. However, the filters that include MPI all agree that process 5 changed the most. Looking at the corresponding diffNLR(5) output in Figure 3.7c makes it clear why process 5 was singled out. In the buggy run, it executes many more MPI_Bcast calls than in the bug-free run because the frequency in which local "optimums" are produced has changed. Though this should affect all traces equally, which has reflected in the diffNLR of other traces. We

Table 3.6: Ranking table - OpenMP bug: unprotected shared memory access by thread 4 of process 6

Filter	Attributes	B-score	Top Processes	Top Threads
11.plt.mem.cust.0K10	doub.noFreq	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
11.plt.mem.cust.0K10	doub.log10	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
01.plt.mem.cust.0K10	doub.noFreq	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
01.plt.mem.cust.0K10	doub.log10	0.244	7, 3, 4	**6.4**, 7.3, 1.4, 3.3, 3.4, 4.2
01.mem.ompcrit.cust.0K10	sing.log10	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
01.mem.ompcrit.cust.0K10	sing.noFreq	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
11.mem.ompcrit.cust.0K10	sing.log10	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
11.mem.ompcrit.cust.0K10	sing.noFreq	0.262	3	**6.4**, 7.1, 3.3, 4.1, 5.1, 6.1
11.plt.mem.cust.0K10	doub.actual	0.273	7	**6.4**, 2.4, 3.4, 4.2, 4.4
01.plt.mem.cust.0K10	doub.actual	0.273	7	**6.4**, 2.4, 3.4, 4.2, 4.4

Table 3.7: Ranking table - MPI bug: wrong collective size in process 2

Filter	Attributes	B-score	Top Processes	Top Threads
11.mpicol.cust.0K10	sing.log10	0.439	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpicol.cust.0K10	sing.noFreq	0.439	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpi.cust.0K10	doub.noFreq	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpi.cust.0K10	doub.actual	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpiall.cust.0K10	doub.noFreq	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpiall.cust.0K10	doub.actual	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpicol.cust.0K10	doub.noFreq	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpicol.cust.0K10	doub.actual	0.457	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpi.cust.0K10	sing.log10	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpi.cust.0K10	sing.noFreq	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpiall.cust.0K10	sing.log10	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpiall.cust.0K10	sing.noFreq	0.465	0, 7, 2, 4, 5, 6	1.1, 1.3, 3.1, 3.2, 3.4
11.mpi.cust.0K10	doub.noFreq	0.543	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4
11.mpi.cust.0K10	doub.actual	0.543	0, 7, 2, 4, 5, 6	1.4, 3.3, 3.4

are presenting these tables and figures to show that DiffTrace can reveal the impact of silent bugs like the wrong operation. Such data representation via suggested tables and diffNLRs helps developers to gain insight into the general behavior of the execution. More accurate results can be obtained by refining the parameters and collecting more profound traces (e.g., ParLOT(all images)). This would be part of our future work to find the set of parameters for different classes of bugs to maximize accuracy.

3.5 LULESH2 Examples

Our ultimate goal is to apply DiffTrace to complex HPC codes. As a more complex example, we have executed the single-cycle LULESH2[**?**] with 8 MPI processes and 4 OMP threads (system configuration described in §**??**) and collected ParLOT (main image) function calls.

Before bug injection, we analyzed LULESH2 traces and computed some statistics to gain insight into the general control flow of LULESH2 and also to evaluate DiffTrace's performance and effectiveness. Our primary results show that ParLOT instruments and captures **410 distinct function** calls on average per process, and stores them in compressed trace files of size less than **2.8 KB** on average per thread. Upon decompression, each per process trace file turns into a sequence of **421503** function calls on average. The equivalent NLR of each trace file reduces the sequence size by a factor of **1.92** and **16.74**, for constant K set to 10 and 50, respectively.

For further evaluation of DiffTrace, we injected a fault into the LULESH source code so that the process with rank 2 would not invoke the function `LagrangeLeapFrog` that is in charge of updating "domain" distances and send/receive MPI messages from other processes.

Table 3.9 reflects the ID of processes (rightmost column) that DiffTrace's ranking system suggests as the most affected traces by the bug. Since the fault in process 2 prevents other processes from making progress and successfully terminate, all of the process IDs appeared in the table. The generated diffNLRs clearly showed the point at which each process stopped making progress. Due to lack of space, we did not include the relatively large diffNLRs of LULESH in this paper. However, all diffNLRs and related observations are available online via [**?**].

Table 3.8: Ranking Table - MPI-Bug: Wrong Collective Operation ,Injected to Process 0

Filter	Attributes	B-score	Top Processes	Top Threads
01.plt.cust.0K10	doub.log10	0.271	2	6.2, 7.3, 2.2, 5.2, 5.3
11.plt.cust.0K10	doub.log10	0.271	2	6.2, 7.3, 2.2, 5.2, 5.3
01.plt.cust.0K10	sing.actual	0.276	1	3.1, 1.4, 6.4, 3.4
11.plt.cust.0K10	sing.actual	0.276	1	3.1, 1.4, 6.4, 3.4
01.plt.cust.0K10	doub.noFreq	0.285	2	6.2, 7.3, 2.2, 5.2, 5.3
11.plt.cust.0K10	doub.noFreq	0.285	2	6.2, 7.3, 2.2, 5.2, 5.3
01.plt.cust.0K10	sing.log10	0.292	1, 4, 5	3.1, 4.3
11.plt.cust.0K10	sing.log10	0.292	1, 4, 5	3.1, 4.3
01.**mpicol**.cust.0K10	sing.actual	0.312	**5**	3.2, 6.4, 5.4, 4.2
11.**mpicol**.cust.0K10	sing.actual	0.312	**5**	3.2, 6.4, 5.4, 4.2
11.**mpi**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
11.**mpiall**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
01.**mpiall**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
01.**mpi**.cust.0K10	sing.actual	0.331	**5**	3.2, 6.4, 5.4, 4.2
11.**mpi**.cust.0K10	sing.actual	0.371	**5**	3.2, 6.4, 5.4, 4.2
11.**mpiall**.cust.0K10	sing.actual	0.371	**5**	3.2, 6.4, 5.4, 4.2

Table 3.9: Ranking Table for LULESH

Filter	Attributes	B-score	Top Processes
11.1K10	sing.noFreq	0.295	**2** , 3 , 4 , 5 , 6 , 7
01.1K10	sing.noFreq	0.354	0 , 1 , **2** , 3 , 4 , 5
01.1K10	sing.actual	0.383	**2** , 3 , 4 , 5 , 6 , 7
11.1K10	sing.noFreq	0.408	**2** , 3 , 4 , 5 , 6 , 7
11.1K10	sing.noFreq	0.408	**2** , 3 , 4 , 5 , 6 , 7
01.1K10	doub.noFreq	0.433	4 , 5 , 6
01.1K10	doub.noFreq	0.433	4 , 5 , 6
11.1K10	doub.noFreq	0.433	5 , 1 , 6
01.1K10	doub.noFreq	0.455	1 , **2** , 3 , 4 , 7
11.1K10	doub.noFreq	0.458	5 , 1 , 6
11.1K10	doub.noFreq	0.458	4 , 5 , 6 , 7
01.1K10	sing.log10	0.459	1 , **2** , 3 , 4 , 5 , 6
01.1K10	doub.noFreq	0.472	0 , 1 , **2** , 3 , 4 , 5
01.1K10	sing.log10	0.475	1 , 3 , 4 , 5 , 6 , 7
01.1K10	sing.log10	0.478	1 , **2** , 3 , 4 , 5 , 6
01.1K10	sing.log10	0.478	1 , **2** , 3 , 4 , 5 , 6

3.6 Related Work

Three major recent studies have emphasized the need for better debugging tools *and* the need to build a community that can share debugging methods and infrastructure: the DOE report mentioned earlier [22], an NSF workshop [11], and an ASCR report on extreme heterogeneity [**?**]. Our key contribution in this paper is a fresh approach to debugging that (1) incorporates methods to debug across the API-stack by resorting to binary tracing and thereby being able to "dial into" MPI bugs and/or OpenMP bugs (as shown in the ILCS case study), (2) makes initial triage of debugging methods possible via function-call traces, and (3) enables the verification community to cohere around DiffTrace by allowing other tools to extend our toolchain (they can tap into it at various places).

Many HPC debugging efforts have emphasized the need to highlight dissimilarities and incorporate progress measures on loops. We now summarize a few of them. AutomaDeD [7][36] captures the application's control flow via Semi Markov Models and detects outlier executions. PRODOMETER [41] detects loops in AutomaDeD models and introduces the notion of *least progressed tasks* by analyzing *progress dependency graphs*. DiffTrace's DiffNLR method does not (yet) incorporate progress measures; it only computes changes in loop *structure*. Prodometer's methods are ripe for symbiotic incorporation into DiffTrace. We also plan to incorporate *happens-before* computation as a progress measure using FCA-based algorithms by Garg et al. [19, 20]. FCA-based approaches have been widely used in data mining [**?**], machine learning [**?**], and information retrieval [26].

In terms of computing differences with previous executions, we draw inspirations from Zeller's delta-debugging [63] and De Rose et al.'s relative debugging [52]. The power of equivalence classes for outlier detection is researched in STAT [3], which merges stack traces from processes into a prefix tree, looking for equivalence-class outliers. STAT uses the StackWalker API from Dyninst [40] to gather stack traces and efficiently handles scaling issues through tree-based overlay networks such as MRnet [53]. D4 [38] detects concurrency bugs by statically analyzing source-code changes, and DMTracker [18] detects anomalies in data movement. The communication patterns of HPC applications can be automatically characterized by diffing the communication matrix with common patterns [54] or by detecting repetitive patterns [50]. ScalaTrace [49] captures and compresses communication traces for later replay. Synoptic [6] is applied to distributed system logs to find bugs.

3.7 Discussions & Future Work

DiffTrace is the first tool we know of that situates debugging around *whole program* diffing, and (1) provides user-selectable front-end filters of function calls to keep; (2) summarizes loops based on state-of-the-art algorithms to detect loop-level behavioral differences; (3) condenses the loop-summarized traces into concept lattices that are built using incremental algorithms; (4) and clusters behaviors using hierarchical clustering and ranks them by similarity to detect and highlight the most salient differences. We deliberately chose the path of a clean start that addresses missing features in existing tools and missing collectivism in the debugging community. Our initial assessment of this design is encouraging.

In our future work we will improve DiffTrace components as follows: (1) Optimizing them to exploit multi-core CPUs, thus reducing the overall analysis time; (2) Converting ParLOT traces into Open Trace Format (OTF2) [**?**] by logically timestamping trace entries to mine temporal properties of functions such as *happened-before* [37]; (3) Conducting systematic bug-injection to see whether concept lattices and loop structures can be used as elevated features for precise bug classifications via machine learning and neural network techniques; and (4) Taking up more challenging and real-world examples to evaluate DiffTrace against similar tools, and release it to the community.

Acknowledgements: Supported in part by NSF awards CCF 1817073 and 1704715.

CHAPTER 4

AUTOMATED CONCURRENCY TESTING FRAMEWORK FOR GO

CHAPTER 5

REFERENCES

[1] X. Aguilar, K. Fürlinger, and E. Laure, *Online mpi trace compression using event flow graphs and wavelets*, Procedia Computer Science, 80 (2016), pp. 1497 – 1506. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[2] ——, *Online mpi trace compression using event flow graphs and wavelets*, Procedia Computer Science, 80 (2016), pp. 1497 – 1506. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[3] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, *Scalable temporal order analysis for large scale debugging*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Nov 2009, pp. 1–11.

[4] A. Allinea, *Allinea DDT*.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, *The nas parallel benchmarks&mdash;summary and preliminary results*, in Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, New York, NY, USA, 1991, ACM, pp. 158–165.

[6] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst, *Synoptic: Studying logged behavior with inferred models*, in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, New York, NY, USA, 2011, ACM, pp. 448–451.

[7] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz, *Automaded: Automata-based debugging for dissimilar parallel tasks*, in 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), June 2010, pp. 231–240.

[8] M. Burtscher, H. Mukka, A. Yang, and F. Hesaaraki, *Real-time synthesis of compression algorithms for scientific data*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, Piscataway, NJ, USA, 2016, IEEE Press, pp. 23:1–23:12.

[9] M. Burtscher and H. Rabeti, *A scalable heterogeneous parallelization framework for iterative local searches*, in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, May 2013, pp. 1289–1298.

[10] S. Claggett, S. Azimi, and M. Burtscher, *SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data*, in 2018 Data Compression Conference, 2018.

[11] A. Cohen, X. Shen, J. Torrellas, J. Tuck, Y. Zhou, S. Adve, I. Akturk, S. Bagchi, R. Balasubramonian, R. Barik, M. Beck, R. Bodik, A. Butt, L. Ceze, H. Chen, Y. Chen, T. Chilimbi, M. Christodorescu, J. Criswell, C. Ding, Y. Ding, S. Dwarkadas, E. Elmroth, P. Gibbons, X. Guo, R. Gupta, G. Heiser, H. Hoffman, J. Huang, H. Hunter, J. Kim, S. King, J. Larus, C. Liu, S. Lu, B. Lucia, S. Maleki, S. Mazumdar, I. Neamtiu, K. Pingali, P. Rech, M. Scott, Y. Solihin, D. Song, J. Szefer, D. Tsafrir, B. Urgaonkar, M. Wolf, Y. Xie, J. Zhao, L. Zhong, and Y. Zhu, *Inter-Disciplinary Research Challenges in Computer Systems for the 2020s*, USA, 2018.

[12] J. Coplin, A. Yang, A. Poppe, and M. Burtscher, *Increasing Telemetry Throughput Using Customized and Adaptive Data Compression*, in AIAA SPACE and Astronautics Forum and Exposition, 2016.

[13] D. de Oliveira, A. Humphrey, Q. Meng, Z. Rakamaric, M. Berzins, and G. Gopalakrishnan, *Systematic debugging of concurrent systems using coalesced stack trace graphs*, in Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC), September 2014.

[14] E. B. Fowlkes and C. L. Mallows, *A method for comparing two hierarchical clusterings*, Journal of the American Statistical Association, 78 (1983), pp. 553–569.

[15] F. Freitag, J. Caubet, and J. Labarta, *On the Scalability of Tracing Mechanisms*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 97–104.

[16] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, *Scalable load-balance measurement for spmd codes*, in SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Nov 2008, pp. 1–12.

[17] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st ed., 1997.

[18] Q. Gao, F. Qin, and D. K. Panda, *Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements*, in SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Nov 2007, pp. 1–12.

[19] V. K. Garg, *Maximal antichain lattice algorithms for distributed computations*, in Distributed Computing and Networking, D. Frey, M. Raynal, S. Sarkar, R. K. Shyamasundar, and P. Sinha, eds., Berlin, Heidelberg, 2013, Springer Berlin Heidelberg, pp. 240–254.

[20] ——, *Introduction to lattice theory with computer science applications*, Wiley, 2015.

[21] R. Godin, R. Missaoui, and H. Alaoui, *Incremental concept formation algorithms based on galois (concept) lattices*, Computational Intelligence, 11, pp. 246–267.

[22] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama, *Report of the HPC correctness summit, jan 25-26, 2017, washington, DC*, CoRR, abs/1705.07478 (2017).

[23] C. Gottbrath and P. Thompson, *Totalview tips and tricks*, in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, New York, NY, USA, 2006, ACM.

[24] K. Hazelwood and A. Klauser, *A dynamic binary instrumentation engine for the arm architecture*, in Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '06, New York, NY, USA, 2006, ACM, pp. 261–270.

[25] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, *Improving performance via mini-applications*, Sandia National Laboratories, Tech. Rep. SAND2009-5574, 3 (2009).

[26] D. I. Ignatov, *Introduction to formal concept analysis and its applications in information retrieval and related fields*, CoRR, abs/1703.02819 (2017).

[27] Intel, *Pin, A Dynamic Binary Instrumentation.*

[28] D. Johnson and L. A. McGeoch, *The traveling salesman problem: A case study in local optimization*, Local Search in Combinatorial Optimization, 1 (1997).

[29] E. Jones, T. Oliphant, P. Peterson, et al., *SciPy: Open source scientific tools for Python*, 2001–. [Online; accessed ¡today¿].

[30] M. Jurenz, R. Brendel, A. Knupfer, M. Muller, and W. E. Nagel, *Memory allocation tracing with vampirtrace*, in Proceedings of the 7th International Conference on Computational Science Part II, ICCS '07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 839–846.

[31] R. M. Karp, R. E. Miller, and A. L. Rosenberg, *Rapid identification of repeated patterns in strings, trees and arrays*, in Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC '72, New York, NY, USA, 1972, ACM, pp. 125–136.

[32] A. Ketterlin and P. Clauss, *Prediction and trace compression of data access addresses through nested loop recognition*, in Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '08, New York, NY, USA, 2008, ACM, pp. 94–103.

[33] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, *Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir*, in Tools for High Performance Computing 2011 - Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, ZIH, Dresden, September 2011, 2011, pp. 79–91.

[34] M. Kobayashi and M. MacDougall, *Dynamic characteristics of loops*, IEEE Transactions on Computers, C-33 (1984), pp. 125–132.

[35] S. O. Kuznetsov and S. A. Obiedkov, *Comparing performance of algorithms for generating concept lattices*, Journal of Experimental & Theoretical Artificial Intelligence, 14 (2002), pp. 189–216.

[36] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree, *Large scale debugging of parallel tasks with automaded,*

in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, New York, NY, USA, 2011, ACM, pp. 50:1–50:10.

[37] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Commun. ACM, 21 (1978), pp. 558–565.

[38] B. Liu and J. Huang, *D4: Fast concurrency debugging with parallel differential analysis*, in Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, New York, NY, USA, 2018, ACM, pp. 359–373.

[39] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, New York, NY, USA, 2005, ACM, pp. 190–200.

[40] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, *The paradyn parallel performance measurement tool*, IEEE Computer, 28 (1995), pp. 37–46.

[41] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin, *Accurate application progress analysis for large-scale parallel debugging*, in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, New York, NY, USA, 2014, ACM, pp. 193–203.

[42] K. Mohror and K. L. Karavanic, *Evaluating similarity-based trace reduction techniques for scalable performance analysis*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, 2009, ACM, pp. 55:1–55:12.

[43] E. W. Myers, *An O(ND) difference algorithm and its variations*, Algorithmica, 1 (1986), pp. 251–266.

[44] A. Nakamura, T. Saito, I. Takigawa, M. Kudo, and H. Mamitsuka, *Fast algorithms for finding a minimum repetition representation of strings and trees*, Discrete Applied Mathematics, 161 (2013), pp. 1556 – 1575.

[45] A. Nataraj, A. Malony, A. Morris, D. C. Arnold, and B. Miller, *A framework for scalable, parallel performance monitoring*, 22 (2009), pp. 720–735.

[46] N. Nethercote and J. Seward, *Valgrind: A program supervision framework*, Electr. Notes Theor. Comput. Sci., 89 (2003), pp. 44–66.

[47] N. Nethercote and J. Seward, *How to shadow every byte of memory used by a program*, in Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE '07, New York, NY, USA, 2007, ACM, pp. 65–74.

[48] M. D. Network, *C Sequence Points*.

[49] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, *Scalatrace: Scalable compression and replay of communication traces for high-performance computing*, Journal of Parallel and Distributed Computing, 69 (2009), pp. 696 – 710. Best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

[50] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, *Detecting patterns in mpi communication traces*, 2008 37th International Conference on Parallel Processing, (2008), pp. 230–237.

[51] P. Ratanaworabhan and M. Burtscher, *Program phase detection based on critical basic block transitions*, in ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software, April 2008, pp. 11–21.

[52] L. D. Rose, A. Gontarek, A. Vose, R. Moench, D. Abramson, M. N. Dinh, and C. Jin, *Relative debugging for a highly parallel hybrid computer system*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015, 2015, pp. 63:1–63:12.

[53] P. C. Roth, D. C. Arnold, and B. P. Miller, *Mrnet: A software-based multicast/reduction network for scalable tools*, in Supercomputing, 2003 ACM/IEEE Conference, Nov 2003, pp. 21–21.

[54] P. C. Roth, J. S. Meredith, and J. S. Vetter, *Automated characterization of parallel application communication patterns*, in Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '15, New York, NY, USA, 2015, ACM, pp. 73–84.

[55] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, *Open | speedshop: An open source infrastructure for parallel performance analysis*, Scientific Programming, 16 (2008), pp. 105–121.

[56] S. S. Shende and A. D. Malony, *The Tau parallel performance system*, International Journal on High Performance Computer Applications, 20 (2006), pp. 287–311.

[57] S. M. Strande, H. Cai, T. Cooper, K. Flammer, C. Irving, G. von Laszewski, A. Majumdar, D. Mishin, P. Papadopoulos, W. Pfeiffer, R. S. Sinkovits, M. Tatineni, R. Wagner, F. Wang, N. Wilkins-Diehr, N. Wolter, and M. L. Norman, *Comet: Tales from the long tail: Two years in and 10,000 users later*, in Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact, PEARC17, New York, NY, USA, 2017, ACM, pp. 38:1–38:7.

[58] S. Taheri, S. Devale, G. Gopalakrishnan, and M. Burtscher, *ParLOT: Efficient whole-program call tracing for HPC applications*, in Programming and Performance Visualization Tools - International Workshops, ESPT 2017 and VPA 2017, Denver, CO, USA, November 12 and 17, 2017, and ESPT 2018 and VPA 2018, Dallas, TX, USA, November 16 and 11, 2018, Revised Selected Papers, 2018, pp. 162–184.

[59] M. M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, *A.: Pmac binary instrumentation library for powerpc/aix*, in In: Workshop on Binary Instrumentation and Applications, 2006.

[60] M. Weber, R. Brendel, T. Hilbrich, K. Mohror, M. Schulz, and H. Brunst, *Structural Clustering: A New Approach to Support Performance Analysis at Scale*, IEEE, May 2016, pp. 484–493.

[61] J. Weidendorfer, *Sequential performance analysis with callgrind and kcachegrind*, in Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart, 2008, pp. 93–113.

[62] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, *MPC: A massively parallel compression algorithm for scientific data*, in 2015 IEEE International Conference on Cluster Computing, Sept 2015, pp. 381–389.

[63] A. Zeller, *Yesterday, my program worked. today, it does not. why?*, in Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings, 1999, pp. 253–267.

[64] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. Inf. Theor., 23 (2006), pp. 337–343.