DiffTrace: Efficient Whole-Program Trace Analysis and Diffing

Saeed Taheri
School of Computing
University of Utah
Salt Lake City, Utah, USA
staheri@cs.utah.edu

Sindhu Devale
Department of Computer Science
Texas State University
San Marcos, Texas, USA
sindhu.devale@gmail.com

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, Utah, USA
ganesh@cs.utah.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, Texas, USA
burtscher@cs.txstate.edu

***Abstract*— Abstract to be written**
***Index Terms*—diffing, tracing, debugging**

I. Introduction

When the next version of an HPC software system is created, logical errors often get introduced. To maintain productivity, designers need effective and efficient methods to locate these errors. Given the increasing use of hybrid (MPI + X) codes and library functions, errors may be introduced through a usage contract violation at any one of these interfaces. Therefore, tools that record activities at multiple APIs are necessary. Designs find most of these bugs manually, and the efficacy of a debugging tool is often measured by how well it can highlight the salient differences between the executions of two versions of software. Given the huge number of things that could be different – individual iterative patterns of function calls, groups of functions calls, or even specific instruction types (e.g., non-vectorized versus vectorized floating-point dot vector loops) – designers cannot often afford to rerun the application multiple times to collect each facet of behavior separately. These issues are well summarized in many recent studies [**?**]

One of the major challenges of HPC debugging is the huge diversity of the applications, which encompass domains such as computational chemistry, molecular dynamics, and climate simulation. In addition, there are many types of possible "bugs" or, more precisely, errors. An **error** may be a deadlock or a resource leak. These errors may be caused by different **faults**: an unexpected message reordering rule (for a deadlock) or a forgotten free statement (for a resource leak). There exists a collection of scenarios in which a bug can be introduced: when developing a brand new application, optimizing an existing application, upscaling an application, porting to a new platform, changing the compiler, or even changing compiler flags. Unlike traditional software, there are hardly any bug-repositories, collection of trace data or debugging-purpose benchmarks in HPC community. The heterogeneous nature of HPC bugs make developers come up with their own solutions to resolve specific class of bugs on specific architecture or platforms that are not usable on other [**?**].

When a failure occurs (e.g., deadlock or crash) or the application outputs an unexpected result, it is not economic to rerun the application and consume resources to reproduce the failure. In addition, HPC bugs might not be reproducible due to non-deterministic behavior of most of HPC applications. Also the failure might be caused by a bug present at different APIs, system levels or network, thus multiple reruns might be needed to locate the buggy area. In our previous work[**?**], we have introduced ParLOT that collects whole program function call traces efficiently using dynamic binary instrumentation. ParLOT captures function calls and returns at different levels (e.g., source-code and library) and incrementally compress them on-the-fly, resulting in low runtime overhead and significantly low required bandwidth, preserving the whole-program dynamic behavior for offline analysis.

In the current work, we introduce DiffTrace, a tool-chain that post-mortem analyze ParLOT traces in order to supply developers with information about dynamic behavior of HPC applications at different levels towards debugging. Topology of HPC tasks on both distributed and shared memory often follow a "symmetric" control flow such as SPMD, master/worker and odd/even where multiple tasks contain *similar* events in their control flow. HPC bugs often manifest themselves as divergence in the control flow of processes comparing to what was expected. In other words, HPC bugs violates the rule of "symmetric" and "similar" control flow of one or more thread/process in typical HPC applications based on the original topology of the application. We believe that finding the dissimilarities among traces is the essential initial step towards finding the bug manifestation, and consequently the bug root cause.
Large-scale HPC application execution would result in thousands of ParLOT trace files due to execution of thousands of processes and threads. Since HPC applications spend most of their time in an outer main loop, every single trace file also may contain million-long sequence of trace entries (i.e., function calls and returns). Finding the bug manifestation (i.e., dissimilarities caused by the bug) among large number of long traces is the problem of finding the needle in the haystack.

Decompressing ParLOT traces collected from long-running large-scale HPC applications for offline analysis produce overwhelming amount of data. However, missing any piece of collected data may result in loosing key information about the application behavior. We propose a variation of NLR (Nested

Loop Recognition) algorithm [?] that takes a sequence of trace entries as input and by recursively detecting repetitive patterns, re-compresses traces into "iterative sets" in a lossless fashion (intra-trace compression). Analyzing the application execution as a whole is another goal that we are pursuing in this work. By extracting *attributes* from pre-processed traces, we inject them a concept hierarchy data structure called Concept Lattice [?]. Concept lattices give us the capability of reducing the search space from thousands of instances to just a few *equivalent behavior classes of traces* by measuring the similarity of traces[?]

** TODO: Highlights of results obtained as a result of the above thinking should be here. THis typically comes before ROADMAP of paper.

In summary, here are our main contributions:

- we have a powerful combination of ideas to locate bugs
- A variation of NLR algorithm to compress traces in lossless fashion for easier analysis and detecting (broken) loop structures
- FCA-based clustering of similar behavior traces, efficient,
- Ranking system based on delta-sim
- Visualization diffNLR

The rest of the paper is as follows:
- Sec 2: Background
- Sec 3: Related Work
- Sec 4: DiffTrace Components and Design
- Sec 5: DiffTrace Evaluation and Experiments
- Sec 6: Discussion, Limitations, Conclusion, Future Work

II. Background

There are two major phases in any "Program Understanding" tool: *data collection* and *data analysis*. To understand the runtime behavior of applications, an efficient tracing mechanism is required to collect informative data during execution of the application. Upon failure or observing unexpected behavior of the program (e.g., wrong answer), studying collected execution data would reveal insight about how program dynamically behaved and what went wrong. In this section, we explain our methodology of data collection and data analysis towards debugging and locating potential root causes of unexpected behavior. The main source of whole-program dynamic behavior is provided by ParLOT, a dynamic binary tracing tool that captures all function calls and returns and compress them on the fly (section II-A). After pre-processing ParLOT traces, a loop-detection-based lossless data reduction mechanism applies to each trace to simplify collected data and reflect facts about loop structures (section II-B). Whole-program analysis in HPC applications only makes sense when the analysis includes cross-thread and cross-process as well as analysis of sequential control flow of every single running thread. Inspired by many works[?] [?] [?][?], we have used FCA[?] to integrate collected data into a single data structure as a whole, and extract valuable information about different aspect of the execution (section II-C) The major advantage of FCA, is that we can extract a full pair-wise similarity score matrix for all traces of a single execution in an efficient and scalable way, based on attributes that we extract from pre-processed traces. Relying on similarities of traces, we classify traces (i.e., threads) into equivalent-behavior groups. This way we reduce the search space from thousands of long traces to just a few classes of simplified trace representation Any abnormal behavior of a thread (e.g., outlier) or group of threads can be detected (section II-E), and for more-in-depth analysis, any pair of traces can be studied with respect to their entry orders and differences using a visualization (section II-D).
Section II-F discusses some additional remarks (limitations, effectiveness of our approach)

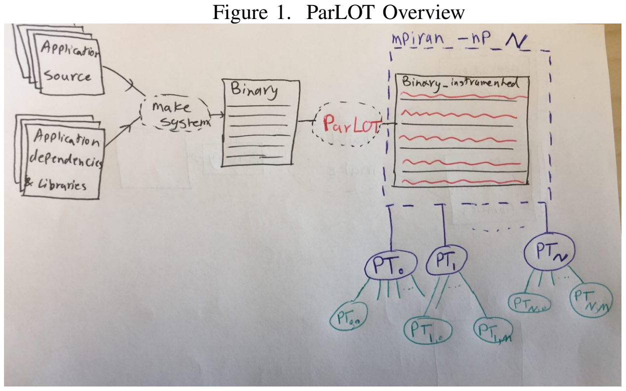
A. ParLOT: Efficient Trace Collection

The final executable of real HPC applications are often a production of a large code base and a complex build system with numerous dependencies and libraries. Injecting instrumentation code to the source code, as in traditional tools like [????], is not feasible in HPC space. Also recompilation of the application with tools' compile-wrappers, as in TAU[?] and Score-p[?], may break the build system. Also instrumentation and tracing mechanism of existing tools are often dependent to other libraries that are need to be present on the supercomputer for trace collection. [[Example: STAT[?] and AutomaDeD[?] that requires Dyninst[?] for instrumentation and MRNet[?] and TBON [?]]] To overcome the trade-off of comprehensive data collection while adding low time and space overhead, HPC program analysis tools often sacrifice one for the other. However, ParLOT collects whole-program function call traces at as low as library level, while incrementally compressing traces on-the-fly and leave majority of the system bandwidth for the application. ParLOT collects *whole* program function call traces with the mindset of *paying a little upfront and save resource and time cost of reproducing the bug*. ParLOT instruments the entry and exit point of each function in the binary using Pin[?] (fig. 1). Each ParLOT Trace contain full sequence of function calls and returns for every single thread that running the application code, reflecting the dynamic control flow and call-stack of the application individually. Here we define ParLOT Trace, as we refer to it in the paper: ParLOT Trace (PT) A ParLOT Trace (PT) is a sequence of ordered integers $< f_i, ..., f_j >$ where f_0 is *return* and f_k is the id of *function k* ($k! = 0$). Note that the PT is the pre-processed (decompressed and filtered) version of immediate ParLOT traces. The fresh output of ParLOT traces are highly compressed byte-codes. Also Note that $PT_{p.t}$ refers to the PT that belongs to process p and thread t of that process.

B. Loop structure detection

HPC applications and resources are in interest of scientists and engineers for simulating *iterative* kernels. Computer simulation of fluid dynamics, partial differential equations, Gauss-Seidel method and finite element methods in form of stencil codes do include a main outer loop that iterates over some

Figure 1. ParLOT Overview



elements (i.e., timesteps) and updates elements. This character of typical HPC applications make PT lengths too long (millions) with much smaller number (hundreds) of distinct elements (i.e., function IDs). We propose a representation of PT elements (intra-PT compression) in form of *loop structures*, such that PT = sequence of repetitive patterns (i.e, loops). In other words, each PT is a sequence of *Loop Bodies (LB)* that repeated *Loop Count (LC)* times, consecutively.

1) Loops definition: According to Makoto Kobayashi[**?**] definition of loops, an occurrence of a *loop* is defined as *a sequence of elements* in which a particular sequence of *distinct elements* (called the *cycle* of the loop) is successively repeated. Later Alain Ketterlin in [**?**] expanded this definition to numerical values for compressing and predicting memory access addresses and designed Nested Loop Recognition (NLR) algorithm. The basic idea behind NLR algorithm is that a linear function model can be extracted from the linear progression in a sequence of numbers and these linear functions form a tree in which depth of each node is the depth of *nested* loop(e.g., most outer loop's function is the root of the tree with depth 0). We have modified NLR algorithm to make it suitable for PTs. Each repetitive pattern and its frequency of consecutive appearances would be compressed to a single *Loop Structure (LS)* entry. Loop Structure $LS = LB\,\hat{}\,LC$ where $LB =< pt_i, ..., pt_j >$ $(0 <= i < j < len(PT))$ that occur LC times is in a subsequence $< pt_i, ..., pt_k > (k = 3j, k < len(PT))$

By converting each PT to a sequence of LS_i , we reduce the length of PT by a factor of $\sum_i len(LB_i) * LC_i$. Later we will explain how this lossless representation of PTs eases the process of diffing between a pair of PTs.

C. Equivalencing behavior via FCA

To Reduce the search space from thousands of PTs to just a few groups of equivalent PTs (i.e., inter-PT compression not only requires a similarity measure based on a call matrix, but also requires a scheme that is efficient even for large process counts. Since a pair-wise comparison of all processes is highly inefficient, we use *concept lattices* that stem from *formal concept analysis*(FCA)[**?**] to store and compute groups of similar PTs. A concept lattice is based on a *formal context*[**?**], which is a triple (O, A, I), where O is a set of **objects**, A a set of **attributes**, and $I \subseteq O \times A$ an incidence relation.

Table I
CONTEXT

	MPI_Init()	MPI_Comm_Size()	MPI_Comm_Rank()	MPI_Send()	MPI_Recv()	MPI_Finalize()
Rank 0	×	×	×		×	×
Rank 1	×	×	×	×		×
Rank 2	×	×	×	×		×
Rank 3	×	×	×	×		×

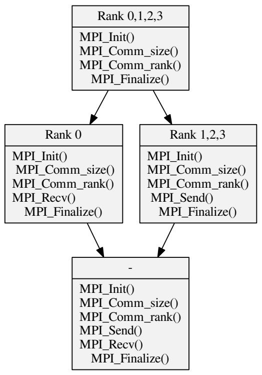


Figure 2. Sample Concept Lattice from Obj-Atr Context in tableII-C

The incidence relation associates each object with a set of attributes. Due to its valuable properties,especially its *partial order*, FCA has been used widely in computer science fields from machine learning and data mining [**?**] to distributed systems [**?**]. However, since we are only interested in grouping similar PTs in this work, we only take advantage of similarity measures [**?**] of concept lattices, and left rest of its properties for our future work. Due to typical HPC application common topologies such as SPMD, master/worker and odd/even where multiple processes/threads behave similarly, our experiments show that large number of PTs can be reduced to just a few groups.

.

```
main(){
 int rank;
 int src;
 MPI_Init()
 MPI_Comm_size(MPI_COMM_WORLD)
 MPI_Comm_rank(MPI_COMM_WORLD,&rank)
 if (rank != 0) {
  MPI_Send(0) // Send to rank 0
 } else{ /* rank = 0
  MPI_Recv(1) // Receive from rank 1
  MPI_Recv(2) // Receive from rank 2
  MPI_Recv(3) // Receive from rank 3
 }
 MPI_Finalize()
}
```

An example with
- Source code [done]
- Simple attributes and Context [done]
- Concept Lattice [done]

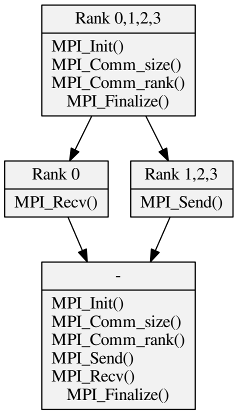


Figure 3. Concept Lattice with reduced labels

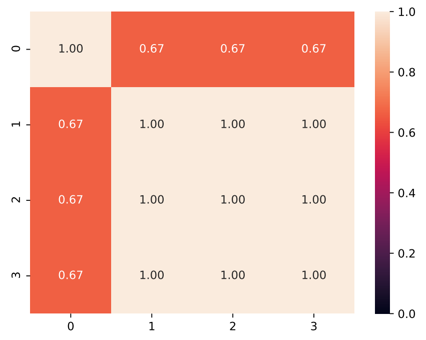


Figure 4. Pair-wise Jaccard Similarity Matrix (JSM) of MPI processes in Sample code

- Jaccard Similarity (heatmap or matrix) [done]
- How we obtained JSM elements
- Visualization of LCA [not yet]
- CL construction [?] [?]
- Attribute Table

D. *diffNLR: Reflecting differences*

-From fig we can see that PT0 and PT1 are in different groups with similarity of XX.
Inspired by `diff` original algorithm[?] that has bin used in Git and GNU Diff, we visualize the differences of a pair of PT as shown in fig 5.
This visualization reflects of the differences of **occurrences** of PT elements and their **orders**.
In section **??** we show how this visualization can help us locating the points of divergence in PTs, and potential bug manifestation and root cause.

E. *Experimental Methodology*

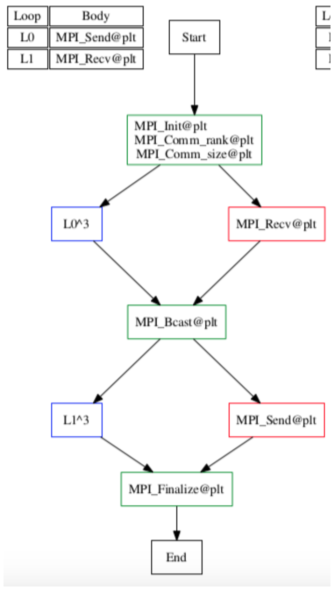


Figure 5. Sample diffNLR

- Candidates to look at and observe their diffNLR: $(PT_i, PT_j) = Max|JSM[i][j](buggy) - JSM[i][j](bugfree)|$
- It means the similarity of (PT_i, PT_j) of buggy execution have the highest difference with its corresponding similarity of the bug-free execution.
- Figures are showing two *diffNLR* side by side, one is $diffNLR(i,j)_{buggy}$ and the other one is $diffNLR(i,j)_{bugfree}$ where i,j obtained from candidate tables.
- Two additional tables that show the LB (loop body) of each detected loop. L_i

F. *Additional Remarks*