DiffTrace: Efficient Whole-Program Trace Analysis and Diffing

Saeed Taheri
staheri@cs.utah.edu
University of Utah
Salt Lake City, Utah, United States

Ian Briggs
ianbriggsutah@gmail.com
University of Utah
Salt Lake City, Utah, United States

Martin Burtscher
burtscher@cs.txstate.edu
Texas State University
San Marcos, Texas, U.S.A.

Ganesh Gopalakrishnan
ganesh@cs.utah.edu
University of Utah
Salt Lake City, Utah, U.S.A.

ABSTRACT

Abstract to be written

KEYWORDS

diffing, tracing, debugging

ACM Reference Format:
Saeed Taheri, Ian Briggs, Martin Burtscher, and Ganesh Gopalakrishnan. 2019. DiffTrace: Efficient Whole-Program Trace Analysis and Diffing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 [19]

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 âĂIJbugsâĂİ 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 [19].

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[45], 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

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [25] 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 [15]. 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[3]

** 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

2 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. 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*.

[[ToBeADDED to the background story]]:

- Nested Loop Recognition (NLR): Easier to analyze, information about loop structures, how many iteration each loop executed matters, broken loop matters

- Concept lattices, inter-process information, similarity, clustering, efficient, scalable, can be used as execution model, can be used in machine learning, can be used to extract HB relation in events.

- diffNLR visualizes pairs of traces and reflect their differences where they are supposed to be equal.

2.1 Parlot Summary (changeme)

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 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[43] and Score-p[26], 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[1] and AutomaDeD[29] that requires Dyninst[34] for instrumentation and MRNet[41] and TBON [21]]]

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.

Each ParLOT Trace (from now on in this paper, PT refers to *single 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.

2.2 Equivalencing behavior via FCA

this is how it is cleanly defined (use our context or examples)

e.g. high-level : objects to attributes

in our case objects can be... and attributes can be

how we use them is ...

- Vijay Garg - Applications of lattice theory in distributed systems
- Dimitry Ignatov [**?**] - Concept Lattice Applications in Information Retrieval
- [15] [18] [4] [17]
  Similarity measurement using FCA [3]

2.2.1 Attribute creation... There are many of them (others have not

2.2.2 incremental algos. challenges : need incremental algo...

end

2.3 Loop structure detection

loop detection has been addressed in xyz

challenges in our context are xyz...

highlights of what you did (briefly) and why it can help

2.4 diffing (changeme)

what you adapted

how does it help

[37]

3 RELATED WORK

3.1 Program Understanding

- Score-P [26]
- TAU [43]
- ScalaTrace: Scalable compression and replay of communication traces for HPC [39]
- Barrier Matching for Programs with Textually unaligned barriers [48]
- Pivot Tracing: Dynamic causal monitoring for distributed systems - Johnathan mace [32]
- Automated Charecterization of parallel application communication patterns [42]
- Problem Diagnosis in Large Scale Computing environments [35]
- Probablistic diagnosis of performance faults in large-scale parallel applications [28]
- detecting patterns in MPI communication traces - robert preissl [40]
- D4: Fast concurrency debugging with parallel differntial analysis - bozhen liu [31]
- Marmot: An MPI analysis and checking tool - bettina krammer [27]
- MPI-checker - Static Analysis for MPI - Alexandrer droste [14]
- STAT: stack trace analysis for large scale debugging - Dorian Arnold [1]
- DMTracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements [16]
- SyncChecker: Detecting synchronization errors between MPI applications and libraries - [10]
- Model Based fault localization in large-scale computing systems - Naoya Maruyama [33]
- Synoptic: Studying logged behavior with inferred models - ivan beschastnikh [5]
- Mining temporal invariants from partially ordered logs - ivan beschastnikh [7]
- Scalable Temporal Order Analysis for Large Scale Debugging - Dong Ahn [2]
- Inferring and asserting distributed system invariants - ivan beschastnikh - stewart grant [20]
- PRODOMETER: Accurate application progress analysis for large-scale parallel debugging - subatra mitra [36]
- Automaded : Automata-based debugging for dissimilar parallel tasks - greg [9]
- Automaded : large scale debugging of parallel tasks with Automaded - ignacio [29]
- Inferring models of concurrent systems from logs of their behavior with CSight - ivan [6]

3.2 Trace Analysis

- Trace File Comparison with a hierarchical Sequence Alignment algorithm [46]
- structural clustering : matthias weber [47]
- building a better backtrace: techniques for postmortem program analysis - ben liblit [30]
- automatically charecterizing large scale program behavior - timothy sherwood [44]

3.3 Visualizations

- Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time - katherine e isaacs [23]
- recovering logical structure from charm++ event traces [22]
- ShiViz - Debugging distributed systems - [8]

3.4 Concept Lattice and LCA

- Vijay Garg - Applications of lattice theory in distributed systems
- Dimitry Ignatov [?] - Concept Lattice Applications in Information Retrieval
- [15] [18] [4] [17] [37]

3.5 Repetitive Patterns

- [11] [24] [38] [13] [12]

3.6 STAT

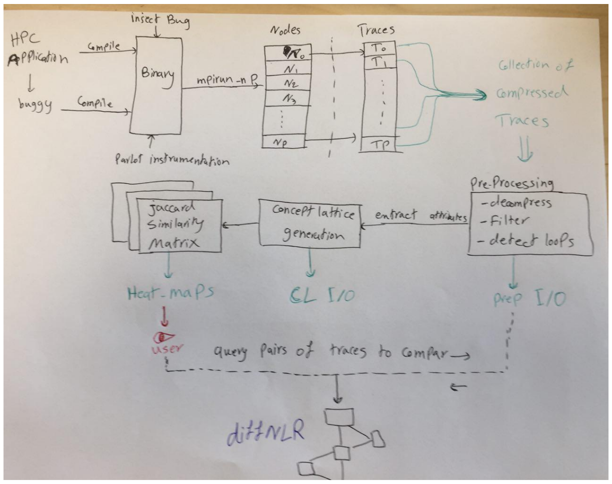
Parallel debugger STAT[1]

- STAT gathers stack traces from all processes
- Merge them into prefix tree
- Groups processes that exhibit similar behavior into equivalent classes
- A single representative of each equivalence can then be examined with a full-featured debugger like TotalView or DDT

What STAT does not have?

- FP debugging
- Portability (too many dependencies)
- Domain-specific
- Loop structures and detection

Figure 1: diffTrace Overview



4 DFFTRACE TOOL ARCHITECTURE

4.1 General Idea

Here is a general overview of DiffTrace and its components

- Motivating example
- Problem statement
- Potential Approaches and Related Work
- Next subsections will explain the components that we have in our framework and the corresponding related work and background

4.2 Fault Injection

4.3 ParLOT

2-3 paragraph explanation about ParLOT and its mechanism [45]

4.4 Filter

Include a table with all filters and their regular expressions

4.4.1 General Filters.

- Returns
- .plt
- Memory
- Network
- Polling
- String
- Customize
- IncludeEverything

4.4.2 Target Filters.

- MPI_
- MPIall
- MPI_ Collectives
- MPI_ Send/Recv
- OMPall
- OMPcritical
- OMPmutex

4.5 Nested Loop Recognition

4.5.1 Background.

4.5.2 Implementation.

4.6 Concept Lattice Analysis

4.6.1 Background.

- FCA (formal concept analysis) background and citations
- FCA applications in all areas
- FCA applications in Data Mining and Information Retrieval
- FCA applications in distributed systems (Garg's work)
- intro to Concept, Object, Attribute and other definitions

4.6.2 Objects/Attributes. Mapping of Object/Attribute (general) to Trace/Attribute (clTrace)

What do we expect to gain by doing so

- Single entity represents the whole execution of HPC application (can be used as signature/model in ML)
- Classifying similar behavior objects(traces)
- Efficient Incremental CL building makes it scalable
- Efficient full pair-wise Jaccard Similarity Matrix extraction

4.6.3 CL generation.

- background
- current approach

4.6.4 Jaccard Similarity Matrix.

- background
- LCA
- Benefits

4.7 diffNLR

- motivation
- diff algorithm
- visualization

4.8 FP-Trace

5 EXPERIMENTAL STUDIES

5.1 Case Study: ILCS-TSP

Here is the ILCS framework pseudo-code. User needs to write `CPU_Init()`,`CPU_Exec()` and `CPU_Output()`.

.

```
int main(argc,argv){
 MPI_Init();
 MPI_Comm_size()
 MPI_Comm_rank(my_rank)
 //Figuring local number of CPUs
 MPI_Reduce() // Figuring global number of CPUs
 CPU_Init();
 //For storing local champion results
 champ[CPUs] = malloc();
 MPI_Barrier();
 #pragma omp parallel num_threads(CPUs+1)
 {
  rank = omp_get_thread_num()
  if (rank == 0){ //communication thread
   do{
    //Find and report the thread with
    //local champion, global champion
    MPI_AllReduce();
    //Find and report the process with
    //global champion
    MPI_AllReduce();
    //The process with the global champion
    //copy its results to bcast_buffer
    if (my_rank == global_champion){
     #pragma omp cirtical
     memcpy(bcast_buffer,local_champ)
    }
    //Broadcast the champion
    MPI_Bcast(bcast_buffer)
   } while (no_change_threshold);
   cont=0 // signal worker threads to stop
  } else{ // worker threads
   while(cont){
    //Calculate Seed
    local_result = CPU_Exec()
    if (local_result < champ[rank]){
     #pragma omp cirtical
     memcpy(champ[rank],local_result)
    }
   }
  }
 }
 //Find and report the thread with
 //local champion, global champion
 MPI_AllReduce();
 //Find and report the process with
 //global champion
 MPI_AllReduce();
 // The process with the global champion
 // copy its results to bcast_buffer
 if (my_rank == global_champion){
  #pragma omp cirtical
  memcpy(bcast_buffer,local_champ)
 }
 //Broadcast the champion
 MPI_Bcast(bcast_buffer)
 if(my_rank==0){
  CPU_Output(champ)
 }
 MPI_Finalize()
}

/* User code for TSP problem */

CPU_Init(){
 // Read In data from cities
 // Calculate distances
 // Return data structure to store champion
}

CPU_Exec(){
 // Find local champions (TSP tours)
}

CPU_Output(){
 // Output champion
}
```

	Filter	Attributes	Top Thread 0	Top Thread 1	Top Thread 2	Top Thread 3	Top Thread 4
0	(14)11.mem.mpicol.ompcrit.cust.0K10	doub.actual	1:(1_0,6_0):1.00 2:(2_0,6_0):1.00 3:(5_0,6_0):1.00	1:(2_1,3_1):0.00 2:(0_1,4_1):0.00 3:(0_1,5_1):0.00	1:(1_2,6_2):0.57 2:(3_2,6_2):0.42 3:(1_2,3_2):0.34	1:(1_3,4_3):0.47 2:(6_3,7_3):0.38 3:(0_3,2_3):0.27	1:(6_4,7_4):0.71 2:(4_4,6_4):0.69 3:(0_4,6_4):0.69
1	(14)11.mem.mpicol.ompcrit.cust.0K10	doub.log10	1:(1_0,6_0):1.00 2:(2_0,6_0):1.00 3:(5_0,6_0):1.00	1:(2_1,3_1):0.00 2:(0_1,4_1):0.00 3:(0_1,5_1):0.00	1:(3_2,6_2):0.36 2:(0_2,2_2):0.30 3:(1_2,3_2):0.29	1:(1_3,6_3):0.62 2:(0_3,5_3):0.29 3:(1_3,4_3):0.26	1:(1_4,5_4):0.62 2:(0_4,5_4):0.39 3:(0_4,4_4):0.33
2	(14)11.mem.mpicol.ompcrit.cust.0K10	doub.orig	1:(1_0,6_0):1.00 2:(2_0,6_0):1.00 3:(5_0,6_0):1.00	1:(2_1,3_1):0.00 2:(0_1,4_1):0.00 3:(0_1,5_1):0.00	1:(3_2,6_2):0.36 2:(0_2,2_2):0.30 3:(1_2,3_2):0.29	1:(1_3,6_3):0.62 2:(0_3,5_3):0.29 3:(1_3,4_3):0.26	1:(1_4,5_4):0.62 2:(0_4,5_4):0.39 3:(0_4,4_4):0.33
3	(14)11.mem.mpicol.ompcrit.cust.0K10	sing.actual	1:(1_0,6_0):1.00 2:(2_0,6_0):1.00 3:(5_0,6_0):1.00	1:(2_1,3_1):0.00 2:(0_1,4_1):0.00 3:(0_1,5_1):0.00	1:(3_2,6_2):0.71 2:(1_2,3_2):0.60 3:(1_2,6_2):0.47	1:(0_3,2_3):0.67 2:(4_3,7_3):0.60 3:(6_3,7_3):0.46	1:(6_4,7_4):0.60 2:(0_4,4_4):0.40 3:(2_4,6_4):0.40
4	(14)11.mem.mpicol.ompcrit.cust.0K10	sing.log10	1:(1_0,6_0):1.00 2:(2_0,6_0):1.00 3:(5_0,6_0):1.00	1:(2_1,3_1):0.00 2:(0_1,4_1):0.00 3:(0_1,5_1):0.00	1:(5_2,7_2):0.67 2:(1_2,3_2):0.67 3:(1_2,6_2):0.50	1:(0_3,2_3):0.40 2:(1_3,6_3):0.40 3:(0_3,5_3):0.27	1:(5_4,6_4):0.50 2:(0_4,5_4):0.50 3:(1_4,5_4):0.40
5	(14)11.mem.mpicol.ompcrit.cust.0K10	sing.orig	1:(1_0,6_0):1.00 2:(2_0,6_0):1.00 3:(5_0,6_0):1.00	1:(2_1,3_1):0.00 2:(0_1,4_1):0.00 3:(0_1,5_1):0.00	1:(3_2,6_2):0.43 2:(0_2,2_2):0.33 3:(1_2,3_2):0.33	1:(0_3,5_3):0.33 2:(0_3,2_3):0.33 3:(1_3,6_3):0.33	1:(5_4,6_4):0.43 2:(0_4,5_4):0.43 3:(1_4,5_4):0.33
6	–	Average	1.000	0.000	0.441	0.403	0.488
7	–	Mean	1.000	0.000	0.441	0.403	0.488
8	–	Min	1.000	0.000	0.286	0.262	0.333
9	–	Max	1.000	0.000	0.714	0.667	0.714

Figure 2: Recommendation Table Bug 1

Table 1 describes the bug that I injected to ILCS-TSP

5.1.1 1: AllReduce wrong op - proc 2. Bug injected within to MPI_AllReduce() in line 21 where the local champions of all processes are about computed and their MIN is going to be distributed to all other processes. However, with injecting the wrong operation (MAX instead of MIN) to one (or more) of the processes, the logic of if statement in line 24 would be violated and the control flow of the code would not take this branch which is the the core part of ILCS (deciding champion to broadcast to other processes).

After collecting traces from bug-free version and this buggy version of ILCS-TSP, and after applying filters and detecting loops, my ranking system, with the highest possible score, recommends to look at figure 3 diffNLR. This diffNLR shows that in the bug free version, we are expecting that at least one of the processes (in this case process 6 to win the championship, copy the result of that process to broadcast_buffer and broadcast it to all other processes. However, in the buggy version, the champion never gets updated (never goes through OMP critical section).

The recommendation system that I designed tries to find the pairs of traces that their similarities changed the most. In other word, this recommendation system wants to suggest top diffNLRs that are candidates to show the impact of the bug based on trace similarities. (table

5.1.2 2: AllReduce wrong size - one and all. This bug made the code crash with showing an error in MPI_Bcast(), although the bug was injected to MPI_AllReduce(). Figure 4 $CMPdiffNLR(P_{2.0}, P_{6.0})$ and 10 CMPdiffNLR($P_{1.0}, P_{6.0}$), where CMPdiffNLR(P_i, P_j) shows the comparison of

$diffNLR(P_{i.buggy}, P_{j.buggy})$

and

$diffNLR(P_{i.bugFree}, P_{j.bugFree})$

Ranking table is showing all 1.00 since most of the traces did not go through.

5.1.3 3: Missing Critical Section one thread in on process. I planted the bug (missing critical section) in process 2

5.1.4 4: Missing Critical Section one thread in on process.

Table 1: Injected Bugs to ILCS-TSP

ID	Level	Bugs	Description
1	MPI	allRed1wrgOp-1-all-x	Different operation (MPI_MAX) in only one process (buggyProc = 2) for MPI_ALLREDUCE() in Line 21
2		allRed1wrgSize-1-all-x	Wrong size in only one process (buggyProc = 2) for MPI_ALLREDUCE() in Line 21
3		allRed1wrgSize-all-all-x	Wrong Size in all processes for MPI_ALLREDUCE() in Line 21
4		allRed2wrgOp-1-all-x	Different operation (MPI_MAX) in only one (buggyProc) for first MPI_ALLREDUCE() – L277:ilcsTSP.c
5		allRed2wrgSize-1-all-x	Wrong size in only one (buggyProc) for first MPI_ALLREDUCE() – L277:ilcsTSP.c
6		allRed2wrgSize-all-all-x	Wrong Size in all processes for second MPI_ALLREDUCE() – L277:ilcsTSP.c
7		bcastWrgSize-1-all-x	Wrong Size in only one (buggyProc) of MPI_Bcast() – L290:ilcsTSP.c
8		bcastWrgSize-all-all-x	Wrong Size n all processes for MPI_Bcast() – L240:ilcsTSP.c
9	OMP	misCrit-1-1-x	Missing Critical Section in buggyProc and buggyThread – L170:ilcsTSP.c
10		misCrit-all-1-x	Missing Critical Section in buggyThread and all prcoesses – L170:ilcsTSP.c
11		misCrit-1-all-x	Missing Critical Section in buggyProc and all threads – L170:ilcsTSP.c
12		misCrit-all-all-x	Missing Critical Section in all procs and threads – L170:ilcsTSP.c
13		misCrit2-1-1-x	Missing Critical Section in buggyProc and buggyThread – L230:ilcsTSP.c
14		misCrit2-all-1-x	Missing Critical Section in buggyThread – L230:ilcsTSP.c
15		misCrit2-1-all-x	Missing Critical Section in buggyProc and all threads – L230:ilcsTSP.c
16		misCrit2-all-all-x	Missing Critical Section in all procs and threads – L230:ilcsTSP.c
17		misCrit3-1-all-x	Missing Critical Section in buggyProc and all threads – L280:ilcsTSP.c
18		misCrit3-all-all-x	Missing Critical Section in all procs and threads – L280:ilcsTSP.c
19	General	infLoop-1-1-1	Injected an infinite loop after CPU_EXEC() in buggyProc,buggyThread & buggyIter L164:ilcsTSP.c

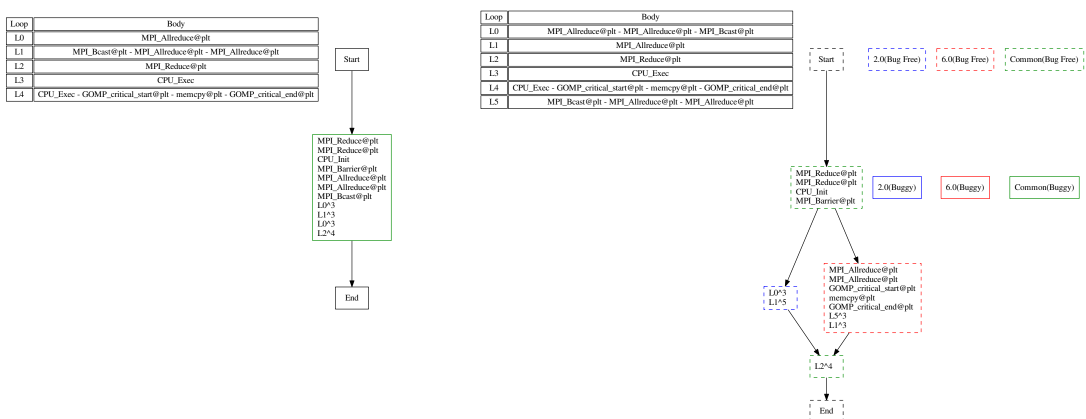


Figure 3: diffNLR of process 2 and process 6 buggy(AllReduce() wrong op) vs. bug-free

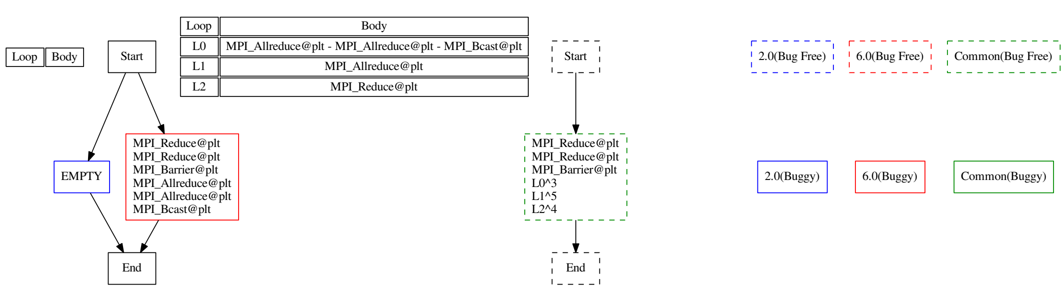


Figure 4: diffNLR of process 2 and process 6 buggy(AllReduce() wrong size) vs. bug-free

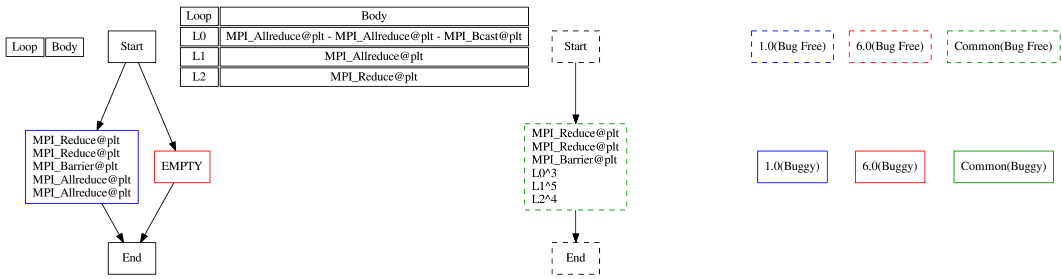
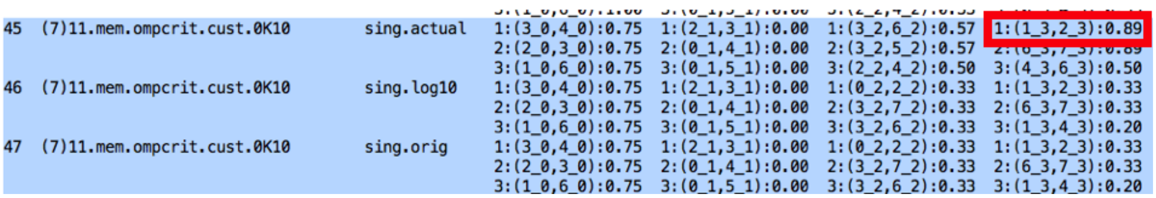


Figure 5: diffNLR of process 1 and process 6 buggy(AllReduce() wrong size) vs. bug-free



			[illegible]	[illegible]	[illegible]	[illegible]
45	(7)11.mem.ompcrit.cust.0K10	sing.actual	1:(3_0,4_0):0.75	1:(2_1,3_1):0.00	1:(3_2,6_2):0.57	1:(1_3,2_3):0.89
			2:(2_0,3_0):0.75	2:(0_1,4_1):0.00	2:(3_2,5_2):0.57	[illegible]
			3:(1_0,6_0):0.75	3:(0_1,5_1):0.00	3:(2_2,4_2):0.50	3:(4_3,6_3):0.50
46	(7)11.mem.ompcrit.cust.0K10	sing.log10	1:(3_0,4_0):0.75	1:(2_1,3_1):0.00	1:(0_2,2_2):0.33	1:(1_3,2_3):0.33
			2:(2_0,3_0):0.75	2:(0_1,4_1):0.00	2:(3_2,7_2):0.33	2:(6_3,7_3):0.33
			3:(1_0,6_0):0.75	3:(0_1,5_1):0.00	3:(3_2,6_2):0.33	3:(1_3,4_3):0.20
47	(7)11.mem.ompcrit.cust.0K10	sing.orig	1:(3_0,4_0):0.75	1:(2_1,3_1):0.00	1:(0_2,2_2):0.33	1:(1_3,2_3):0.33
			2:(2_0,3_0):0.75	2:(0_1,4_1):0.00	2:(3_2,7_2):0.33	2:(6_3,7_3):0.33
			3:(1_0,6_0):0.75	3:(0_1,5_1):0.00	3:(3_2,6_2):0.33	3:(1_3,4_3):0.20

Figure 6: Part of ranking table for MisCrit 1-1

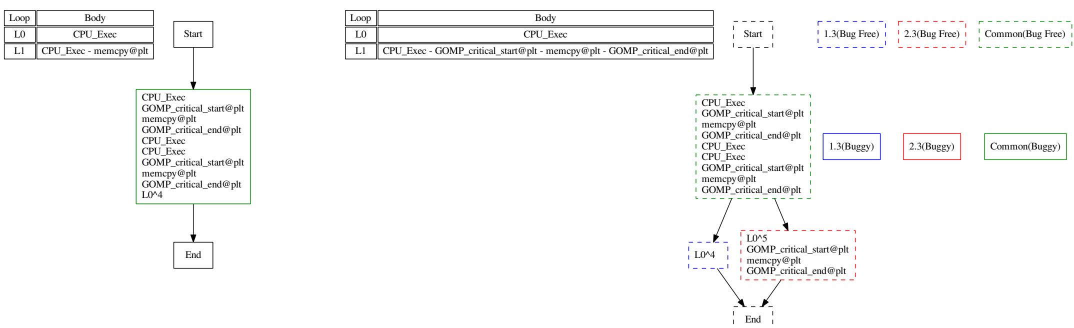
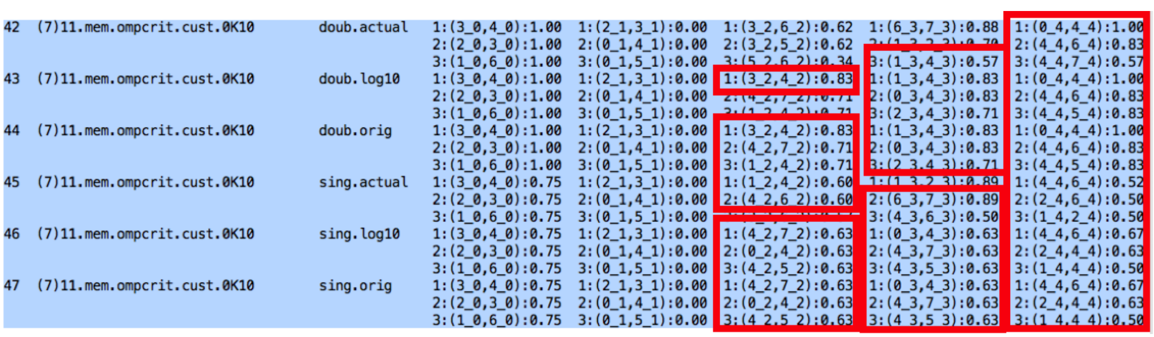


Figure 7: diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free



42	(7)11.mem.ompcrit.cust.0K10	doub.actual	1:(3_0,4_0):1.00	1:(2_1,3_1):0.00	1:(3_2,6_2):0.62	1:(6_3,7_3):0.88	1:(0_4,4_4):1.00
			2:(2_0,3_0):1.00	2:(0_1,4_1):0.00	2:(3_2,5_2):0.62	[illegible]	2:(4_4,6_4):0.83
			3:(1_0,6_0):1.00	3:(0_1,5_1):0.00	3:(5_2,6_2):0.34	3:(1_3,4_3):0.57	3:(4_4,7_4):0.57
43	(7)11.mem.ompcrit.cust.0K10	doub.log10	1:(3_0,4_0):1.00	1:(2_1,3_1):0.00	1:(3_2,4_2):0.83	1:(1_3,4_3):0.83	1:(0_4,4_4):1.00
			2:(2_0,3_0):1.00	2:(0_1,4_1):0.00	[illegible]	2:(0_3,4_3):0.83	2:(4_4,6_4):0.83
			3:(1_0,6_0):1.00	3:(0_1,5_1):0.00	[illegible]	3:(2_3,4_3):0.71	3:(4_4,5_4):0.83
44	(7)11.mem.ompcrit.cust.0K10	doub.orig	1:(3_0,4_0):1.00	1:(2_1,3_1):0.00	1:(3_2,4_2):0.83	1:(1_3,4_3):0.83	1:(0_4,4_4):1.00
			2:(2_0,3_0):1.00	2:(0_1,4_1):0.00	2:(4_2,7_2):0.71	2:(0_3,4_3):0.83	2:(4_4,6_4):0.83
			3:(1_0,6_0):1.00	3:(0_1,5_1):0.00	3:(1_2,4_2):0.71	3:(2_3,4_3):0.71	3:(4_4,5_4):0.83
45	(7)11.mem.ompcrit.cust.0K10	sing.actual	1:(3_0,4_0):0.75	1:(2_1,3_1):0.00	1:(1_2,4_2):0.60	1:(1_3,2_3):0.89	1:(4_4,6_4):0.52
			2:(2_0,3_0):0.75	2:(0_1,4_1):0.00	2:(4_2,6_2):0.60	2:(6_3,7_3):0.89	2:(2_4,6_4):0.50
			3:(1_0,6_0):0.75	3:(0_1,5_1):0.00	[illegible]	3:(4_3,6_3):0.50	3:(1_4,2_4):0.50
46	(7)11.mem.ompcrit.cust.0K10	sing.log10	1:(3_0,4_0):0.75	1:(2_1,3_1):0.00	1:(4_2,7_2):0.63	1:(0_3,4_3):0.63	1:(4_4,6_4):0.67
			2:(2_0,3_0):0.75	2:(0_1,4_1):0.00	2:(0_2,4_2):0.63	2:(4_3,7_3):0.63	2:(2_4,4_4):0.63
			3:(1_0,6_0):0.75	3:(0_1,5_1):0.00	3:(4_2,5_2):0.63	3:(4_3,5_3):0.63	3:(1_4,4_4):0.50
47	(7)11.mem.ompcrit.cust.0K10	sing.orig	1:(3_0,4_0):0.75	1:(2_1,3_1):0.00	1:(4_2,7_2):0.63	1:(0_3,4_3):0.63	1:(4_4,6_4):0.67
			2:(2_0,3_0):0.75	2:(0_1,4_1):0.00	2:(0_2,4_2):0.63	2:(4_3,7_3):0.63	2:(2_4,4_4):0.63
			3:(1_0,6_0):0.75	3:(0_1,5_1):0.00	3:(4_2,5_2):0.63	3:(4_3,5_3):0.63	3:(1_4,4_4):0.50

Figure 8: Part of ranking table for MisCrit 1-all

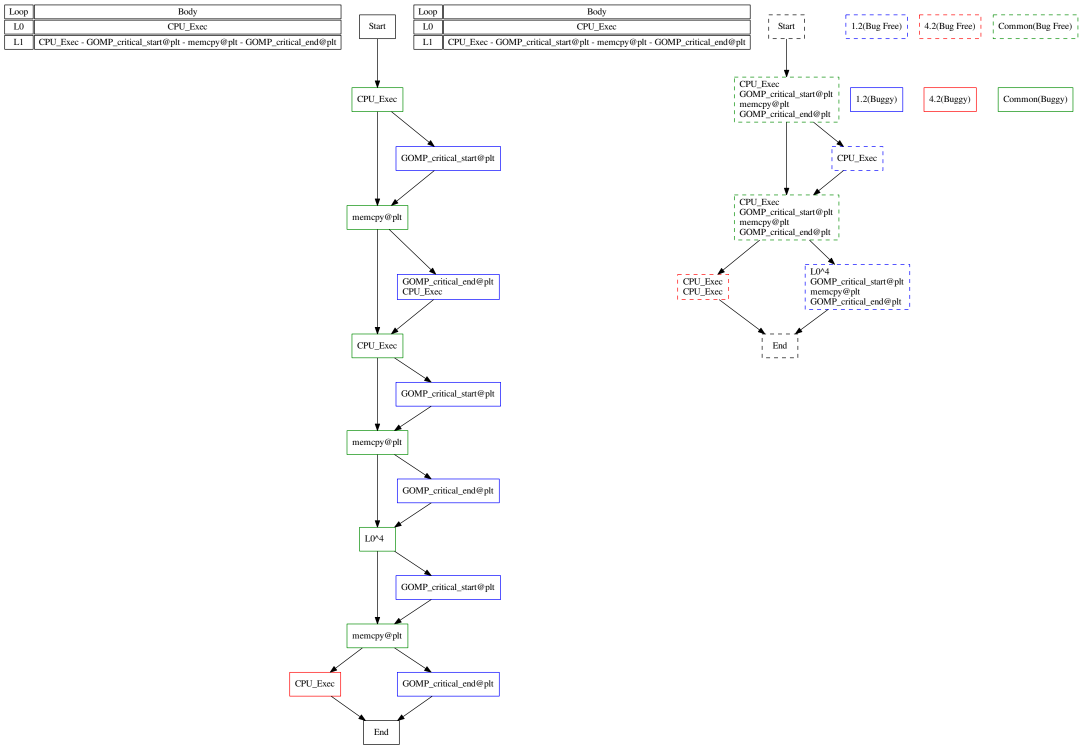


Figure 9: diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free

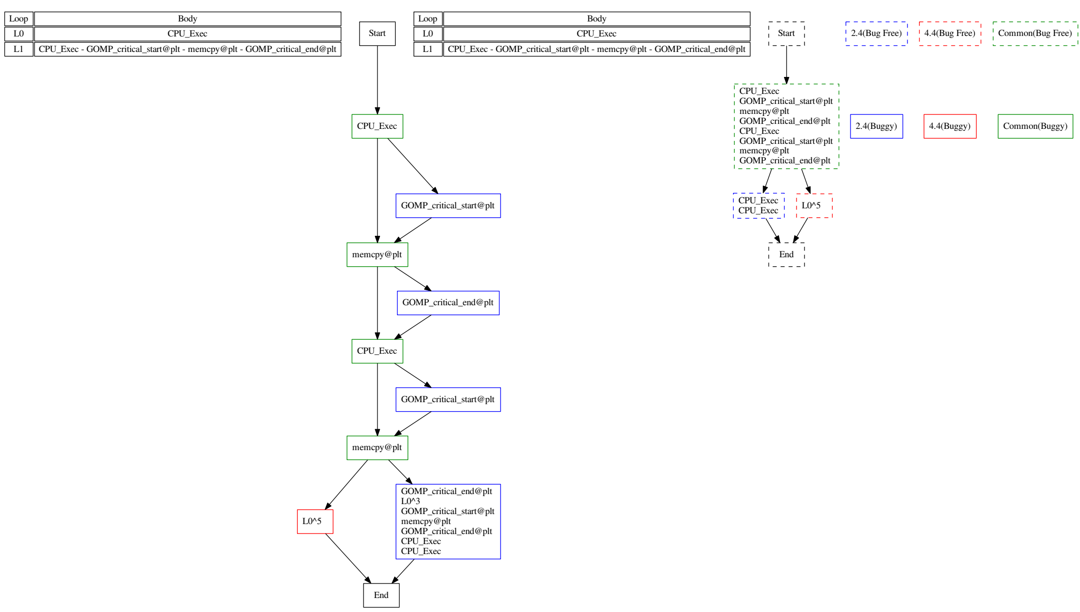


Figure 10: diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free

6 DISCUSSION AND FUTURE WORK

REFERENCES

[1] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. 2009. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/1654059.1654104

[2] Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. 2009. Scalable Temporal Order Analysis for Large Scale Debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, NY, USA, Article 44, 11 pages. https://doi.org/10.1145/1654059.1654104

[3] Faris Alqadah and Raj Bhatnagar. 2011. Similarity measures in formal concept analysis. *Annals of Mathematics and Artificial Intelligence* 61, 3 (01 Mar 2011), 245–256. https://doi.org/10.1007/s10472-011-9257-7

[4] Michael A. Bender, MartÃŋn Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. 2005. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms* 57, 2 (2005), 75 – 94. https://doi.org/10.1016/j.jalgor.2005.08.001

[5] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D. Ernst. 2011. 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)*. ACM, New York, NY, USA, 448–451. https://doi.org/10.1145/2025113.2025188

[6] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 468–479. https://doi.org/10.1145/2568225.2568246

[7] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. 2011. Mining Temporal Invariants from Partially Ordered Logs. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML '11)*. ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2038633.2038636

[8] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging Distributed Systems. *Commun. ACM* 59, 8 (July 2016), 32–37. https://doi.org/10.1145/2909480

[9] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz. 2010. AutomaDeD: Automata-based debugging for dissimilar parallel tasks. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 231–240. https://doi.org/10.1109/DSN.2010.5544927

[10] Z. Chen, X. Li, J. Chen, H. Zhong, and F. Qin. 2012. SyncChecker: Detecting Synchronization Errors between MPI Applications and Libraries. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 342–353. https://doi.org/10.1109/IPDPS.2012.40

[11] M. Crochemore and W. Rytter. 1991. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theoretical Computer Science* 88, 1 (1991), 59 – 82. http://www.sciencedirect.com/science/article/pii/030439759190073B

[12] Maxime Crochemore and Wojciech Rytter. 1994. *Text Algorithms*. Oxford University Press, Inc., New York, NY, USA.

[13] Maxime Crochemore and Wojciech Rytter. 2002. *Jewels of Stringology*. World Scientific. https://books.google.com/books?id=ipuPQgAACAAJ

[14] Alexander Droste, Michael Kuhn, and Thomas Ludwig. 2015. MPI-checker: Static Analysis for MPI. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/2833157.2833159

[15] Bernhard Ganter and Rudolf Wille. 1997. *Formal Concept Analysis: Mathematical Foundations* (1st ed.). Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[16] Q. Gao, F. Qin, and D. K. Panda. 2007. 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*. 1–12. https://doi.org/10.1145/1362622.1362643

[17] Vijay K. Garg. 2013. Maximal Antichain Lattice Algorithms for Distributed Computations. In *Distributed Computing and Networking*, Davide Frey, Michel Raynal, Saswati Sarkar, Rudrapatna K. Shyamasundar, and Prasun Sinha (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 240–254.

[18] Robert Godin, Rokia Missaoui, and Hassan Alaoui. [n. d.]. INCREMENTAL CONCEPT FORMATION ALGORITHMS BASED ON GALOIS (CONCEPT) LATTICES. *Computational Intelligence* 11, 2 ([n. d.]), 246–267.

[19] Ganesh Gopalakrishnan, Paul D. Hovland, Costin Iancu, Sriram Krishnamoorthy, Ignacio Laguna, Richard A. Lethin, Koushik Sen, Stephen F. Siegel, and Armando Solar-Lezama. 2017. Report of the HPC Correctness Summit, Jan 25-26, 2017, Washington, DC. *CoRR* abs/1705.07478 (2017). arXiv:1705.07478 http://arxiv.org/abs/1705.07478

[20] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and Asserting Distributed System Invariants. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1149–1159. https://doi.org/10.1145/3180155.3180199

[21] Tobias Hilbrich, Bronis R. de Supinski, Fabian Hänsel, Matthias S. Müller, Martin Schulz, and Wolfgang E. Nagel. 2013. Runtime MPI Collective Checking with Tree-based Overlay Networks. In *Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI '13)*. ACM, New York, NY, USA, 129–134. https://doi.org/10.1145/2488551.2488570

[22] K. E. Isaacs, A. Bhatele, J. Lifflander, D. BÃűhme, T. Gamblin, M. Schulz, B. Hamann, and P. Bremer. 2015. Recovering logical structure from Charm++ event traces. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807634

[23] Katherine E. Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. 2014. Combing the Communication Hairball: Visualizing Parallel Execution Traces using Logical Time. *IEEE Transactions on Visualization and Computer Graphics* 20 (2014), 2349–2358.

[24] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. 1972. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (STOC '72)*. ACM, New York, NY, USA, 125–136. https://doi.org/10.1145/800152.804905

[25] Alain Ketterlin and Philippe Clauss. 2008. 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)*. ACM, New York, NY, USA, 94–103. https://doi.org/10.1145/1356058.1356071

[26] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2011. 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*. 79–91.

[27] Bettina Krammer, Matthias MÃČÅŠller, and Michael Resch. 2004. MPI application development using the analysis tool MARMOT, Vol. 3038. 464–471. https://doi.org/10.1007/978-3-540-24688-6_61

[28] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, and Todd Gamblin. 2012. Probabilistic Diagnosis of Performance Faults in Large-scale Parallel Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 213–222. https://doi.org/10.1145/2370816.2370848

[29] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Anh, Martin Schulz, and Barry Rountree. 2011. Large Scale Debugging of Parallel Tasks with AutomaDeD. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 50, 10 pages. https://doi.org/10.1145/2063384.2063451

[30] Ben Liblit and Alex Aiken. 2002. *Building a Better Backtrace: Techniques for Postmortem Program Analysis*. Technical Report. Berkeley, CA, USA.

[31] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3192366.3192390

[32] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Trans. Comput. Syst.* 35, 4, Article 11 (Dec. 2018), 28 pages. https://doi.org/10.1145/3208104

[33] N. Maruyama and S. Matsuoka. 2008. Model-based fault localization in large-scale computing systems. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2008.4536310

[34] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 1995. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28, 11 (1995), 37–46. https://doi.org/10.1109/2.471178

[35] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. 2006. Problem Diagnosis in Large-Scale Computing Environments. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 11–11. https://doi.org/10.1109/SC.2006.50

[36] Subrata Mitra, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. 2014. 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)*. ACM, New York, NY, USA, 193–203. https://doi.org/10.1145/2594291.2594336

[37] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266. https://doi.org/10.1007/BF01840446

[38] Atsuyoshi Nakamura, Tomoya Saito, Ichigaku Takigawa, Mineichi Kudo, and Hiroshi Mamitsuka. 2013. Fast algorithms for finding a minimum repetition representation of strings and trees. *Discrete Applied Mathematics* 161, 10 (2013), 1556 – 1575. https://doi.org/10.1016/j.dam.2012.12.013

[39] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. 2009. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel and Distrib. Comput.* 69, 8 (2009), 696 – 710. https://doi.org/10.1016/j.jpdc.2008.09.001 Best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

[40] Robert Preissl, Thomas Köckerbauer, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. 2008. Detecting Patterns in MPI Communication Traces. *2008 37th International Conference on Parallel Processing* (2008), 230–237.

[41] P. C. Roth, D. C. Arnold, and B. P. Miller. 2003. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Supercomputing, 2003 ACM/IEEE Conference.* 21–21. https://doi.org/10.1145/1048935.1050172

[42] Philip C. Roth, Jeremy S. Meredith, and Jeffrey S. Vetter. 2015. Automated Characterization of Parallel Application Communication Patterns. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15).* ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2749246.2749278

[43] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *International Journal on High Performance Computer Applications* 20 (May 2006), 287–311. Issue 2. https://doi.org/10.1177/1094342006064482

[44] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X).* ACM, New York, NY, USA, 45–57. https://doi.org/10.1145/605397.605403

[45] S. Taheri, S. Devale, G. Gopalakrishnan, and M. Burtscher. [n. d.]. PARLOT: Efficient Whole-Program Call Tracing for HPC Applications. In *7th Workshop on Extreme-Scale Programming Tools, ESPT@SC 2018, Dallas, TX, USA, November 16, 2018.*

[46] M. Weber, R. Brendel, and H. Brunst. 2012. Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications.* 247–254. https://doi.org/10.1109/ISPA.2012.40

[47] Matthias Weber, Ronny Brendel, Tobias Hilbrich, Kathryn Mohror, Martin Schulz, and Holger Brunst. 2016. Structural Clustering: A New Approach to Support Performance Analysis at Scale. IEEE, 484–493. https://doi.org/10.1109/IPDPS.2016.27

[48] Yuan Zhang and Evelyn Duesterwald. 2007. Barrier Matching for Programs with Textually Unaligned Barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07).* ACM, New York, NY, USA, 194–204. https://doi.org/10.1145/1229428.1229472