DiffTrace: Efficient Whole-Program Trace Analysis and Diffing for Debugging

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

Ian Briggs
School of Computing
University of Utah
Salt Lake City, Utah, USA
ian.briggs@gmail.com

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

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

***Abstract*— Abstract to be written**

***Index Terms*—tracing, diffing, debugging**

I. Introduction

Debugging high-performance computing code remains a challenge at all levels of scale. Conventional HPC debuggers [**?**], [**?**] 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 [1] 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 own 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 common 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 are 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.

a) 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 [2]. 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 [2], thus making this approach practical, demanding only a few kilobytes per second per core of bandwidth.

b) Problem 2 – Need to Generalize Techniques for Outlier Detection: Given that outlier detection is central to debugging, it is important 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 with respect to 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 [**?**] 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.

c) 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 immediately highlight many bug types.

Solution to Problem 3: DiffTrace utilizes the rigorous notion of Nested Loop Representations (NLRs) for extracting 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: §II illustrates the contributions of this paper on a simple example. §III presents the algorithms underlying DiffTrace in more detail. §IV shows a medium-sized case study involving MPI and OpenMP. §**??** summaries the experimental methodology before presenting results for LULESH [**?**], a DOE common mini app. §VI summarizes selected related works. §VII concludes the paper with a discussion.

II. DiffTrace Overview

A. High-level Overview

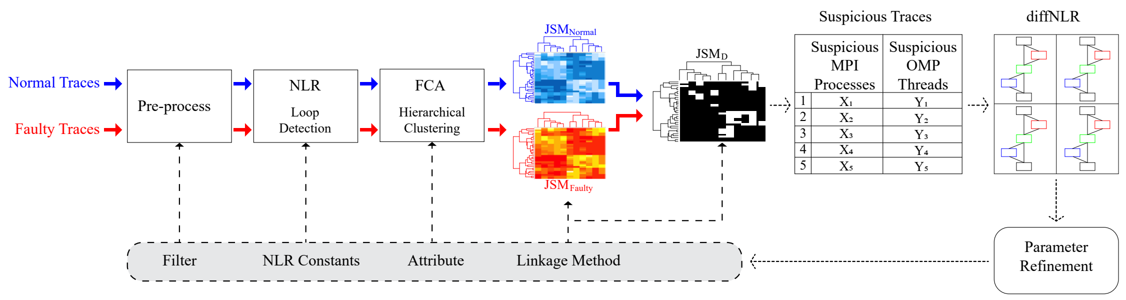
DiffTrace employs ParLoT's [**?**] whole-program function-call and return trace collection mechanism, where ParLoT captures traces via Pin [3] and incrementally compresses them using new compression schemes [**?**]. ParLoT can capture functions at two levels: the *main image* (which does not include library code) and *all images* (including all application 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 not only reducing 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 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 stage conducts *formal concept analysis*, which is 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 [7], [8] employs FCA exactly in this manner (including the use of pairs of calls), but 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

Figure 1. DiffTrace Overview



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 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 grey scale 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 §I *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* [?] 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.

Figure 2. Simplified MPI implementation of Odd/Even Sort

	Main Function	oddEvenSort()
1	`int main(){`	`oddEvenSort(rank, cp){`
2	` int rank,cp;`	` ...`
3	` MPI_Init()`	` for (int i=0; i < cp; i++)`
4	` MPI_Comm_rank(..., &rank);`	` {`
5	` MPI_Comm_size(..., &cp);`	`  int ptr = findPtr(i, rank);`
6	` // initialize data to sort`	`  ...`
7	` int *data[data_size];`	`  if (rank % 2 == 0) {`
8	` ...`	`   MPI_Send(..., ptr, ...);`
9	` oddEvenSort(rank, cp);`	`   MPI_Recv(..., ptr, ...);`
10	` ...`	`  } else {`
11	` MPI_Finalize();`	`   MPI_Recv(..., ptr, ...);`
12	`}`	`   MPI_Send(..., ptr, ...);`
13		`  }`
14		`  ...`
15		` }`
16		`}`

B. Example Walk-through

We now employ Figure 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.[1]

A waiting trap in this example is this: the `MPI_Send` call may need to *block* if there is not enough system buffer space available (EAGER limit is set low). Thus, in low-buffer situations, the statements on lines 11 and 12 of this figure can end up in a deadlock. We will now show how DiffTrace helps picks out this root-cause.

C. Pre-processing

Using ParLoT's decoder, each trace is first decompressed. Next, the desired functions are extracted based on predefined (Table I) or custom regular expressions (i.e., *filters*) and kept

[1]The details of this algorithm are unimportant for this paper and may be found in standard MPI textbooks such as by Pacheco [?].

Table I
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 II
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

for later phases. Table II 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.

D. 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 [4] and other data structures [5], we have adapted the Nested Loop Recognition (NLR) algorithm by Ketterlin et al. [6] to detect repetitive patterns in ParLoT traces (cf. Section III-A). 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 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 II can be reduced to Table III 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.

Table III
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

E. Hierarchical Clustering via FCA

Processes in HPC applications are known to fall into predictable equivalence classes. The widely used and highly successful STAT tool [27] owes most of its success to 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. Coalesced stack trace graphs (CSTG, [**?**]) 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.[2]

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 IV 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). Table (the table that shows attributes in the next section) shows the attributes that we have extracted from the traces. 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 shows the concept lattice derived from the formal context in Table IV 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.

[2]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$_D$.

Table IV
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	×	×	×		×	×

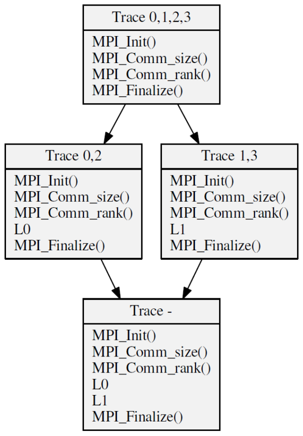


Figure 3. Sample Concept Lattice from Obj-Atr Context in Table IV

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 III-B). Figure 4 shows the heatmap of the JSM obtained from the concept lattice in Figure 3. 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.

F. Detecting Suspicious Traces via DiffJSM

$JSM_{normal}[i][j]$ ($JSM_{faulty}[i][j]$) 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

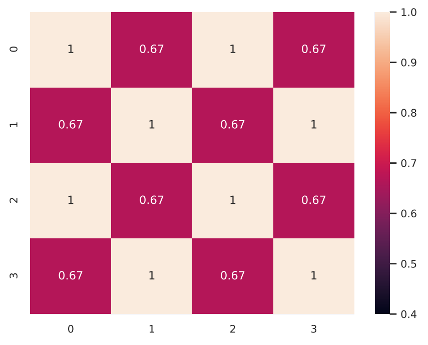
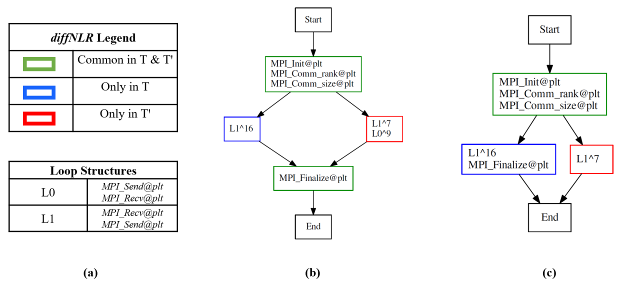


Figure 4. Pairwise Jaccard Similarity Matrix (JSM) of MPI processes in sample code

Figure 5. (a) The legend of *diffNLR* and the list of loop structures (b) *diffNLR(5)* of *swapBug* (c) *diffNLR(5)* of *dlBug*



executions. The construction of JSM$_D$ involves obtaining the hierarchical clustering based on *DiffJSM*, followed by the subtraction of the faulty JSM from its corresponding normal JSM. $DiffJSM = |JSM_{faulty} - JSM_{normal}|$. We sort the suggestion table based on the *B-score* similarity metric of two hierarchical clusterings [9] (cf. Section III-C). A single iteration through the DiffTrace loop (with a single set of parameters shown as a dashed box in Figure 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 1.

G. Evaluation

To evaluate the effectiveness of DiffJSM, we planted two artificial bugs (*swapBug* and *dlBug*) in the code from Figure 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 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** (Figure 5-(a)), 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*.

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 [10].

`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) $\equiv$ diffNKR(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 5-(b) shows the $diffNLR(5)$ of *swapBug* where T_5 iterates over the loop [MPI_Recv - MPI_Send] 16 times (L1ˆ16) after the MPI initialization while the order swap is well reflected in T'_5 (L1ˆ7 - L0ˆ9). Both processes seem to terminate fine by executing MPI_Finalize(). However, $diffNLR(5)$ of *dlBug* (Figure 5-(c)) 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.

III. Algorithms Underlying DiffTrace

A. Nested Loop Recognition (NLR)

We build NLRs based on the work by Ketterlin and Clauss [6], who use this algorithm for trace compression, and the work by Kobayashi and MacDougall [11], 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

Table V
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

B. Concept Lattice Construction

The efficiency of algorithms for concept lattice construction depends on the sparseness of the formal context [12]. Ganter's *Next Closure* algorithm [8] 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 [13] to extract attributes (Table V) 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).

C. 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. [14]) for these tasks. SciPy provides a wide range of linkage functions such as single, complete, average, weighted, centroid, median, and ward.

1) Ranking Table: As shown in Figure 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. [9] 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.

IV. Case Study: ILCS

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

To evaluate the effectiveness of ideas behind DiffTrace, we have manually injected MPI-level and OMP-level bugs to the Traveling Salesman Problem (TSP) implementation on ILCS framework (Listing 1). The injected bugs are tend to simulate real HPC bugs such as deadlocks. Also bugs are close to common mistakes that HPC developers usually make during developing HPC codes. In addition, there exist "hidden" faults that does not alarm anything during execution such as violation of critical sections or semantic-based bugs.

.

```
1  main(argc,argv){
2   ... // initialization
3   MPI_Init();
4   MPI_Comm_size()
5   MPI_Comm_rank(my_rank)
6   ... // Obtain number of local CPUs and GPUs
7   MPI_Reduce(lCPUs, gCPUs,MPI_SUM) // Total # of CPUs
8   MPI_Reduce(lGPUs, gGPUs,MPI_SUM) // Total # of GPUs
9   champSize = CPU_Init();
10  ... // Memory allocation for storing local and global
        champions w.r.t. champSize
11  MPI_Barrier();
12  #pragma omp parallel num_threads(lCPUs+1)
13  {rank = omp_get_thread_num()
14   if (rank != 0){ // worker threads
15    while(cont){
16      ...//Calculate Seed
17      local_result = CPU_Exec()
18      if (local_result < champ[rank]){ // update local
         champion
19       #pragma omp critical
20       memcpy(champ[rank],local_result)}}
21   } else{ //master thread
22    do{
23     ...
24     MPI_AllReduce(); //broadcast the global champion
25     ...
26     MPI_AllReduce(); //broadcast the global champion P_id
27     ...
28     if (my_rank == global_champion_P_id){
29      #pragma omp circital
30      memcpy(bcast_buffer,local_champ); //
31     }
32     MPI_Bcast(bcast_buffer); // broadcast the local
        champion to all nodes
33   } while (no_change_threshold);
34    cont=0; // signal worker threads to terminate
35  }}
36  if(my_rank==0){ CPU_Output(champ);}
37  MPI_Finalize();}
38
39 /* User code for TSP problem */
40 CPU_Init(){ /* Read coordinations, calculate distances,
      Initialize the champion structure, Return structure
      size */}
41 CPU_Exec(){ /* Find local champions (TSP tours) */}
42 CPU_Output(){/* Output champion */ }
```

Listing 1. ILCS Overview

The injected bugs are planted in a way that might get triggered in only one or more threads (master and worker threads, one thread, every other thread, all threads except one, all threads). Generally, the goal is to see how effective DiffTrace can analyze and diff traces, and how close it can get to the fault root cause or its manifestation.

We have collected ParLOT (main image) traces from the execution of ILCS-TSP with 8 MPI processes and 4 OpenMP threads on each process. PSC Config will be added. Note that the GPU-related activities of ILCS are out of the scope of this paper, and we have not touched them in our experiments. Here, after a general explanation of ILCS behavior, we explain the injected bugs and the observations from DiffTrace.

A. ILCS-TSP workflow

2-3 sentences about how ILCS finds local champions in TSP problem

There are two types of threads in ILCS: a *master* thread per node (MPI process) and a set of *worker* threads per compute node (OpenMP threads). Master threads of compute nodes are in charge of handling local working threads and communicating with master threads on other nodes. For each detected CPU core, the master thread forks worker OpenMP threads. Each worker thread continually calls `CPU_Exec()` to evaluate a range of seeds and record the results (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). Since scanning the entire seed range in a reasonable amount of time is not feasible, ILCS terminates the search when the quality has not improved over a certain period (lines 33-34).

B. OpenMP Bug: Unprotected Memory Access

The memory accesses of `memcpy` in line 20 and 30 are protected by OpenMP critical section. If under some scenario,

Table VI
RANKING TABLE - OMP-BUG: UNPROTECTED SHARED MEMORY ACCESS, INJECTED TO THREAD 4 OF PROCESS 6

Filter	Attributes	Link Method	Thresh	B-score	Top Procs	TOP Threads
11.plt.mem.cust.0K10	doub.noFreq	ward	4	0.244	7 , 3 , 4 ,	**6.4** , 7.3 , 1.4 , 3.3 , 3.4 , 4.2 ,
11.plt.mem.cust.0K10	doub.log10	ward	4	0.244	7 , 3 , 4 ,	**6.4** , 7.3 , 1.4 , 3.3 , 3.4 , 4.2 ,
01.plt.mem.cust.0K10	doub.noFreq	ward	4	0.244	7 , 3 , 4 ,	**6.4** , 7.3 , 1.4 , 3.3 , 3.4 , 4.2 ,
01.plt.mem.cust.0K10	doub.log10	ward	4	0.244	7 , 3 , 4 ,	**6.4** , 7.3 , 1.4 , 3.3 , 3.4 , 4.2 ,
01.mem.ompcrit.cust.0K10	sing.log10	ward	4	0.262	3 ,	**6.4** , 7.1 , 3.3 , 4.1 , 5.1 , 6.1 ,
01.mem.ompcrit.cust.0K10	sing.noFreq	ward	4	0.262	3 ,	**6.4** , 7.1 , 3.3 , 4.1 , 5.1 , 6.1 ,
11.mem.ompcrit.cust.0K10	sing.log10	ward	4	0.262	3 ,	**6.4** , 7.1 , 3.3 , 4.1 , 5.1 , 6.1 ,
11.mem.ompcrit.cust.0K10	sing.noFreq	ward	4	0.262	3 ,	**6.4** , 7.1 , 3.3 , 4.1 , 5.1 , 6.1 ,
01.plt.mem.mpi.ompall.cust.0K10	sing.actual	ward	4	0.266		2.4 , 4.3 ,
11.plt.mem.mpi.ompall.cust.0K10	sing.actual	ward	4	0.266		2.4 , 4.3 ,
11.plt.mem.cust.0K10	doub.actual	weighted	4	0.273	7 ,	**6.4** , 2.4 , 3.4 , 4.2 , 4.4 ,
01.plt.mem.cust.0K10	doub.actual	weighted	4	0.273	7 ,	**6.4** , 2.4 , 3.4 , 4.2 , 4.4 ,
11.plt.mem.mpi.ompcrit.cust.0K10	doub.noFreq	ward	4	0.276	3 ,	3.3 , **6.4** ,
11.plt.mem.mpi.ompcrit.cust.0K10	doub.log10	ward	4	0.276	3 ,	3.3 , **6.4** ,
01.plt.mem.mpi.ompcrit.cust.0K10	doub.noFreq	ward	4	0.276	3 ,	3.3 , **6.4** ,
01.plt.mem.mpi.ompcrit.cust.0K10	doub.log10	ward	4	0.276	3 ,	3.3 , **6.4** ,

this shared memory location becomes unprotected, a race condition might happen and invalidate the ILCS final output. We have simulated such a scenario and modified the ILCS source code so that the control flow of the program skip the critical section in some specific OpenMP threads. In one case where we inject this bug to the worker thread 4 of process 6, DiffTrace generated Table VI as top suspicious traces for further analysis. Each table entry contains the parameter that leads to the last two column suggestions. For example, filter "11.mem.ompcit.cust.0K10" briefly means that all returns and .plt calls have been removed from traces of both faulty and normal executions, and only memory-related functions, OpenMP critical section functions and custom function "CPU_Exec" are kept in traces. The K10 at the end of filter means that all filtered traces are converted to their equivalent NLR with K=10. The rest of the parameters have been explained in previous sections. I will remove two unnecessary columns from the tables (threshold and linkage function) to save space and add 2-3 sentences explaining what was them

The bold numbers in the last column are suggesting trace **6.4** (process 6, thread 4) as the trace that changed the most after we planted the bug. diffNLR(6.4) in Figure 6 clearly shows that the normal execution of ILCS (blue blocks) protects the `memcpy` while the buggy execution does not. In this figure, L0 is `CPU_Exec`, which has been executed several times in both versions but never reaches the optimal solution until the end.

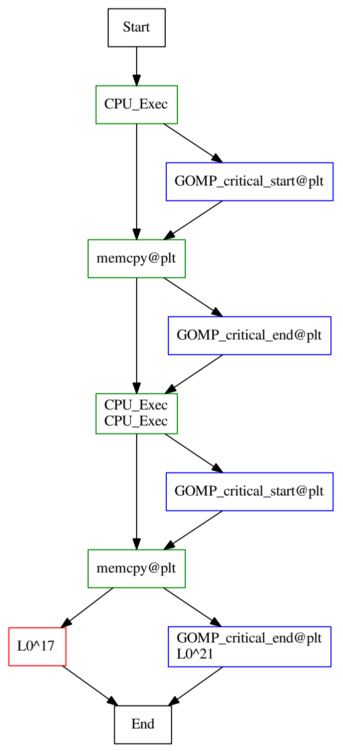


Figure 6. OpenMP Bug: diffNLR(6.4)

C. MPI Bug: Deadlock Caused by Fault in Collectives

By forcing only one of the processes (process 2) to invoke MPI_Allreduce (line 24) with a wrong size, we have simulated a *real deadlock*. Table VII shows that almost all processes are suspicious. It turned out that ParLOT did not happen to capture function calls from all processes since the bug happens too early in the code. Thus except for process 1 and 4, all other traces are empty. By looking at the diffNLR(1) (Figure 7), we can see that both normal and the buggy trace of process 1 are identical until an invocation of MPI_Allreduce(). After that, normal trace hits the end of the program and terminates while the buggy process is waiting for the return from the actual point of fault (process 2) and never ends (i.e., deadlocks). diffNLRs of other processes look the same.

D. MPI Bug: Wrong Collective Operation

By changing the operation MPI_MIN to MPI_MAX in the input arguments of MPI_Allreduce(), we have changed the semantics of ILCS. The execution of this variation terminated well, but the results might be corrupted.

Table VII
RANKING TABLE - MPI-BUG: WRONG COLLECTIVE SIZE, INJECTED TO PROCESS 2

Filter	Attributes	Link Method	Thresh	B-score	Top Procs (JSMD)	TOP Threads(JSMD)
11.mem.mpicol.ompcrit.cust.0K10	sing.log10	ward	4	0.383	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 1.4 , 3.1 , 3.2 , 3.4 ,
11.mem.mpicol.ompcrit.cust.0K10	sing.noFreq	ward	4	0.383	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 1.4 , 3.1 , 3.2 , 3.4 ,
11.mpicol.cust.0K10	sing.log10	ward	4	0.439	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 3.1 , 3.2 , 3.4 ,
11.mpicol.cust.0K10	sing.noFreq	ward	4	0.439	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 3.1 , 3.2 , 3.4 ,
11.mpi.cust.0K10	doub.noFreq	ward	4	0.457	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,
11.mpi.cust.0K10	doub.actual	ward	4	0.457	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,
11.mpiall.cust.0K10	doub.noFreq	ward	4	0.457	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,
11.mpiall.cust.0K10	doub.actual	ward	4	0.457	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,
11.mpicol.cust.0K10	doub.noFreq	ward	4	0.457	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,
11.mpicol.cust.0K10	doub.actual	ward	4	0.457	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,
11.mpi.cust.0K10	sing.log10	ward	4	0.465	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 3.1 , 3.2 , 3.4 ,
11.mpi.cust.0K10	sing.noFreq	ward	4	0.465	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 3.1 , 3.2 , 3.4 ,
11.mpiall.cust.0K10	sing.log10	ward	4	0.465	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 3.1 , 3.2 , 3.4 ,
11.mpiall.cust.0K10	sing.noFreq	ward	4	0.465	0 , 7 , 2 , 4 , 5 , 6 ,	1.1 , 1.3 , 3.1 , 3.2 , 3.4 ,
11.mpi.cust.0K10	doub.noFreq	ward	3	0.543	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,
11.mpi.cust.0K10	doub.actual	ward	3	0.543	0 , 7 , 2 , 4 , 5 , 6 ,	1.4 , 3.3 , 3.4 ,

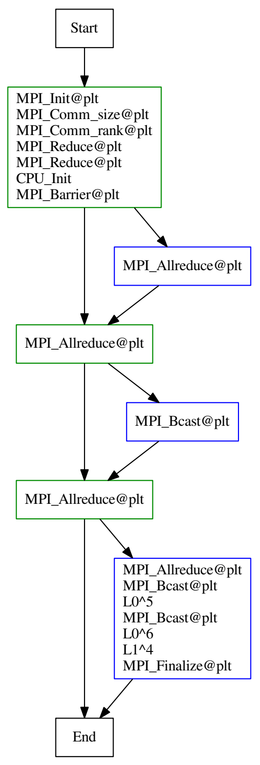


Figure 7. diffNLR(0)

The MPI_Allreduce() in line 24 of Listing 1 broadcasts the best-calculated answer among all processes. However, by the change that we made to ILCS, now the "worst" answer is getting stored. We injected the bug only to process 0. Among all suggested suspicious processes (Table VIII), only process 5 (bold numbers) are making sense since their filters are more relevant to the aspect that we are interested (MPI-level activities) to study deeper. Our observation from diffNLR(5) (Figure 8) is that process 5, in comparison with its corresponding normal process, involves more in updating and broadcasting the champion among all traces. Similar to the deadlock bug, this is another instance of "bug manifestation" detection by DiffTrace.

Table VIII
RANKING TABLE - MPI-BUG: WRONG COLLECTIVE OPERATION ,INJECTED TO PROCESS 0

Filter	Attributes	Link Method	Thresh	B-score	Top Procs (JSMD)	TOP Threads(JSMD)
01.plt.cust.0K10	doub.log10	ward	4	0.271	2 ,	6.2 , 7.3 , 2.2 , 5.2 , 5.3 ,
11.plt.cust.0K10	doub.log10	ward	4	0.271	2 ,	6.2 , 7.3 , 2.2 , 5.2 , 5.3 ,
01.plt.cust.0K10	sing.actual	ward	4	0.276	1 ,	3.1 , 1.4 , 6.4 , 3.4 ,
11.plt.cust.0K10	sing.actual	ward	4	0.276	1 ,	3.1 , 1.4 , 6.4 , 3.4 ,
01.plt.cust.0K10	doub.noFreq	ward	4	0.285	2 ,	6.2 , 7.3 , 2.2 , 5.2 , 5.3 ,
11.plt.cust.0K10	doub.noFreq	ward	4	0.285	2 ,	6.2 , 7.3 , 2.2 , 5.2 , 5.3 ,
01.plt.cust.0K10	sing.log10	ward	4	0.292	1 , 4 , 5 , 6 ,	3.1 , 4.3 ,
11.plt.cust.0K10	sing.log10	ward	4	0.292	1 , 4 , 5 , 6 ,	3.1 , 4.3 ,
01.**mpicol**.cust.0K10	sing.actual	ward	4	0.312	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,
11.**mpicol**.cust.0K10	sing.actual	ward	4	0.312	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,
11.**mpi**.cust.0K10	sing.actual	ward	4	0.331	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,
11.**mpiall**.cust.0K10	sing.actual	ward	4	0.331	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,
01.**mpiall**.cust.0K10	sing.actual	ward	4	0.331	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,
01.**mpi**.cust.0K10	sing.actual	ward	4	0.331	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,
11.**mpi**.cust.0K10	sing.actual	ward	3	0.371	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,
11.**mpiall**.cust.0K10	sing.actual	ward	3	0.371	**5** ,	3.2 , 6.4 , 5.4 , 4.2 ,

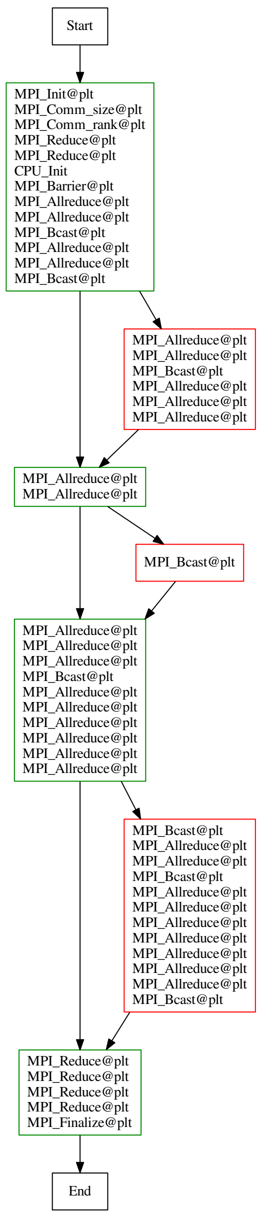


Figure 8. diffNLR(5)

V. Larger Examples, Limitations, Future Avenues

VI. Related Work

1) HPC debugging: The general idea of most HPC debugging tools are first collecting some information from the applications Top: STAT: stack trace analysis for large scale debugging - Dorian Arnold [27] PRODOMETER: Accurate application progress analysis for large-scale parallel debugging - subatra mitra [35] Automaded : Automata-based debugging for dissimilar parallel tasks - greg [36][37] D4 : real time concurrency debugging, detecting changes in source code iteratively Marmot : mpi debugger deadlock detection MPI checker static analysis for MPI

DM-tracker Detecting anomaly in data movements

Other: Inferring models of concurrent systems from logs of their behavior with CSight - ivan [38]

- Inferring and asserting distributed system invariants - ivan beschastnikh - stewart grant [34]
- Mining temporal invariants from partially ordered logs - ivan beschastnikh [32]
- Model Based fault localization in large-scale computing systems - Naoya Maruyama [30]
- Synoptic: Studying logged behavior with inferred models - ivan beschastnikh [31]
- Barrier Matching for Programs with Textually unaligned barriers [18]
- Pivot Tracing: Dynamic causal monitoring for distributed systems - Johnathan mace [19]
- Automated Charecterization of parallel application communication patterns [20]
- Problem Diagnosis in Large Scale Computing environments [21]
- Probablistic diagnosis of performance faults in large-scale parallel applications [22]
- detecting patterns in MPI communication traces - robert preissl [23]
- Score-P [15]
- TAU [16]
- ScalaTrace: Scalable compression and replay of communication traces for HPC [17]

A. Trace Analysis

- Trace File Comparison with a hierarchical Sequence Alignment algorithm [39]
- structural clustering : matthias weber [7]
- building a better backtrace: techniques for postmortem program analysis - ben liblit [40]
- automatically charecterizing large scale program behavior - timothy sherwood [41]

B. Visualizations

- Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time - katherine e isaacs [42]
- recovering logical structure from charm++ event traces [43]
- ShiViz - Debugging distributed systems - [44]

C. Concept Lattice and LCA

- Vijay Garg - Applications of lattice theory in distributed systems
- Dimitry Ignatov [**?**] - Concept Lattice Applications in Information Retrieval
- [8] [13] [45] [46]

D. Repetitive Patterns

E. STAT

Parallel debugger STAT[27]

- 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

VII. Concluding Remarks

GANESH: In the discussion section at the end, please make a note now itself that we will have an absolute-debugging story also. I'll also add it. In the JSM, we can see truncated vs. non-truncated executions. Easy.

REFERENCES

[1] 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*, vol. abs/1705.07478, 2017. [Online]. Available: http://arxiv.org/abs/1705.07478

[2] 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. [Online]. Available: https://doi.org/10.1007/978-3-030-17872-7_10

[3] 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*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[4] 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*, vol. 161, no. 10, pp. 1556 – 1575, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X13000024

[5] 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*, ser. STOC '72. New York, NY, USA: ACM, 1972, pp. 125–136. [Online]. Available: http://doi.acm.org/10.1145/800152.804905

[6] 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*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 94–103. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356071

[7] 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. [Online]. Available: http://ieeexplore.ieee.org/document/7516045/

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

[9] E. B. Fowlkes and C. L. Mallows, "A method for comparing two hierarchical clusterings," *Journal of the American Statistical Association*, vol. 78, no. 383, pp. 553–569, 1983. [Online]. Available: https://amstat.tandfonline.com/doi/abs/10.1080/01621459.1983.10478008

[10] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986. [Online]. Available: https://doi.org/10.1007/BF01840446

[11] "Dynamic characteristics of loops," *IEEE Transactions on Computers*, vol. C-33, no. 2, pp. 125–132, Feb 1984.

[12] S. O. Kuznetsov and S. A. Obiedkov, "Comparing performance of algorithms for generating concept lattices," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 14, no. 2-3, pp. 189–216, 2002. [Online]. Available: https://doi.org/10.1080/09528130210164170

[13] R. Godin, R. Missaoui, and H. Alaoui, "Incremental concept formation algorithms based on galois (concept) lattices," *Computational Intelligence*, vol. 11, no. 2, pp. 246–267.

[14] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed ¡today¿]. [Online]. Available: http://www.scipy.org/

[15] 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.

[16] S. S. Shende and A. D. Malony, "The Tau parallel performance system," *International Journal on High Performance Computer Applications*, vol. 20, pp. 287–311, May 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1125980.1125982

[17] 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*, vol. 69, no. 8, pp. 696 – 710, 2009, best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

[18] Y. Zhang and E. Duesterwald, "Barrier matching for programs with textually unaligned barriers," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 194–204. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229472

[19] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, pp. 11:1–11:28, Dec. 2018. [Online]. Available: http://doi.acm.org/10.1145/3208104

[20] 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*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749278

[21] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Nov 2006, pp. 11–11.

[22] I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, and T. Gamblin, "Probabilistic diagnosis of performance faults in large-scale parallel applications," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 213–222. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370848

[23] 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*, pp. 230–237, 2008.

[24] 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*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 359–373. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192390

[25] B. Krammer, M. MÃŒller, and M. Resch, "Mpi application development using the analysis tool marmot," vol. 3038, 12 2004, pp. 464–471.

[26] A. Droste, M. Kuhn, and T. Ludwig, "Mpi-checker: Static analysis for mpi," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2833157.2833159

[27] 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.

[28] 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.

[29] Z. Chen, X. Li, J. Chen, H. Zhong, and F. Qin, "Syncchecker: Detecting synchronization errors between mpi applications and libraries," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 342–353.

[30] N. Maruyama and S. Matsuoka, "Model-based fault localization in large-scale computing systems," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–12.

[31] 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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 448–451. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025188

[32] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "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*, ser. SLAML '11.

New York, NY, USA: ACM, 2011, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2038633.2038636

[33] 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*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 44:1–44:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654104

[34] S. Grant, H. Cech, and I. Beschastnikh, "Inferring and asserting distributed system invariants," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1149–1159. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180199

[35] 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*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 193–203. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594336

[36] 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.

[37] 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*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 50:1–50:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063451

[38] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 468–479. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568246

[39] M. Weber, R. Brendel, and H. Brunst, "Trace file comparison with a hierarchical sequence alignment algorithm," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, July 2012, pp. 247–254.

[40] B. Liblit and A. Aiken, "Building a better backtrace: Techniques for postmortem program analysis," Berkeley, CA, USA, Tech. Rep., 2002.

[41] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 45–57. [Online]. Available: http://doi.acm.org/10.1145/605397.605403

[42] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing parallel execution traces using logical time," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, pp. 2349–2358, 2014.

[43] K. E. Isaacs, A. Bhatele, J. Lifflander, D. Böhme, T. Gamblin, M. Schulz, B. Hamann, and P. Bremer, "Recovering logical structure from charm++ event traces," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.

[44] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Commun. ACM*, vol. 59, no. 8, pp. 32–37, Jul. 2016. [Online]. Available: http://doi.acm.org/10.1145/2909480

[45] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin, "Lowest common ancestors in trees and directed acyclic graphs," *Journal of Algorithms*, vol. 57, no. 2, pp. 75 – 94, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0196677405000854

[46] 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: Springer Berlin Heidelberg, 2013, pp. 240–254.

[47] M. Crochemore and W. Rytter, "Usefulness of the karp-miller-rosenberg algorithm in parallel computations on strings and arrays," *Theoretical Computer Science*, vol. 88, no. 1, pp. 59 – 82, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/030439759190073B

[48] ——, *Jewels of Stringology*. World Scientific, 2002. [Online]. Available: https://books.google.com/books?id=ipuPQgAACAAJ

[49] ——, *Text Algorithms*. New York, NY, USA: Oxford University Press, Inc., 1994.

Appendix