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@txstate.edu

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

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

***Index Terms*—Whole-program tracing, HPC debugging, trace diffing, nested loop recognition, formal concept analysis**

I. Introduction

Debugging high-performance computing code remains a challenge at all levels of scale. Conventional HPC debuggers [1], [2] 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 [3] provides a comprehensive survey of existing debugging tools. It classifies them under four software organizations (serial, multithreaded, multi-process, and hybrid), six method types (formal methods, static analysis, dynamic analysis, nondeterminism control, anomaly detection, and parallel debugging), and lists a total of 30 specific tools. Despite this abundance of activity and tools, many significant problems remain to be solved before debugging *can be approached by the HPC community as a collaborative activity* so that HPC developers can extend a common framework.

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

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

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

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 [4]. 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 [4], 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 essential to use efficient representations of the traces to be able to systematically compute *distances* between them without involving human reasoning. The representation must also be versatile enough to be able to "diff" the traces with respect to *an extensible number of vantage points*. These vantage points could be diffing traces concerning process-level activities, thread-level activities, a combination thereof, or even finite sequences of process/thread calls (say, to locate *changes* in caller/callee relationships).

Solution to Problem 2: DiffTrace employs *concept lattices* to amalgamate the collected traces. Concept lattices have previously been employed in HPC to perform structural clustering of process behaviors [5] 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 highlight many bug types immediately.

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

Organization: §II illustrates the contributions of this paper on a simple example. §III presents the algorithms underlying DiffTrace in more detail. §IV summaries the experimental methodology before showing a medium-sized case study involving MPI and OpenMP. §V shows initial measurements and examples on LULESH [6], a DOE common mini app. §VI summarizes selected related works. §VII concludes the paper with a discussion.

II. DiffTrace Overview

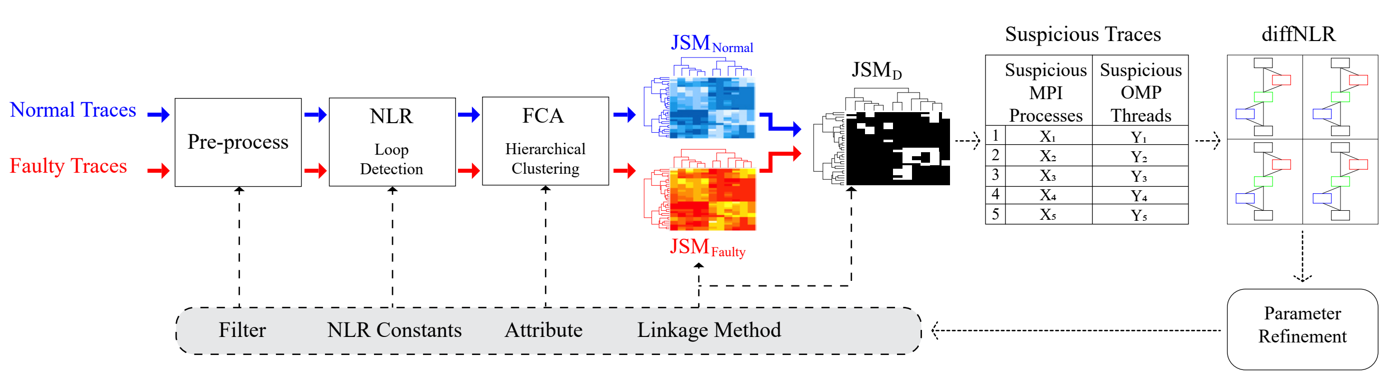
A. High-level Overview

DiffTrace employs ParLOT's whole-program function-call and return trace-collection mechanism, where ParLOT captures traces via Pin [7] and incrementally compresses them using a new compression scheme [4]. ParLOT can capture functions at two levels: the *main image* (which does not include library code) and *all images* (including all library code). As the application runs, ParLOT generates per-thread trace files that contain the compressed sequence of the IDs of the executed functions. The compression mechanism is light-weight yet effective, thus reducing not only the required bandwidth and storage but also the runtime relative to not compressing the traces. As a result, ParLOT can capture whole-program traces at low overhead while leaving most of the disk bandwidth to the application. Using whole-program traces substantially reduces the number of overall debug iterations because it allows us to repeatedly analyze the traces offline with different filters.

Figure 1 provides an overview of the DiffTrace toolchain in terms of the blue flows (fault-free) and red flows (faulty). In a broad sense, code-level faults in HPC applications (e.g., the use of wrong subscripts) turn into observable code-level misbehaviors (e.g., an unexpected number of loop iterations), many of which turn into application-level issues. In our study of DiffTrace, we evaluate success merely in terms of the efficacy of observing these misbehaviors in response to injected code-level faults (we rely on a rudimentary fault injection framework complemented by manual fault injection).

The preprocessing stage removes calls/returns at the ignored APIs. The nested loop recognition (NLR) mechanism then extracts loops from traces. The resulting information not only serves as a lossless abstraction to ease the rest of the trace analysis but also serves as a *per-thread measure of progress*. The FCA (Formal Concept Analysis) stage conducts a systematic way to arrange objects (in our case threads) and attributes

Figure 1: DiffTrace Overview



	Suspicious MPI Processes	Suspicious OMP Threads
1	X_1	Y_1
2	X_2	Y_2
3	X_3	Y_3
4	X_4	Y_4
5	X_5	Y_5

(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 [5], [8] employs FCA exactly in this manner (including the use of pairs of calls), however, for grouping performance information. Our new contribution is showing that FCA can play a central role in debugging HPC applications.

While faults induce asymmetries ("aberrations") in program behaviors, one cannot locate faults merely by locating the asymmetries in an overall collection of process traces. The reason is that even in a collection of MPI processes or threads within these processes, some processes/threads may serve as a master while others serve as workers [9]. Thus, we must have a base level of similarities computed even for normal behaviors and then compute how *this similarity relation changes* when faults are introduced. This is highlighted by the blue and red rectangular patches in Figure 1 that, respectively, iconify the *Jaccard similarity matrices* computed for the normal behavior (above) and the erroneous behavior (below). This is shown as the "diff Jaccard similarity matrix" in greyscale at the juncture of JSM$_{normal}$ and JSM$_{faulty}$.

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

Last but not least, we strongly believe that a framework such as DiffTrace can serve as an important HPC community resource. Each debugging tool designer who uses DiffTrace can extend it by incorporating new attributes and clustering methods, but otherwise retain the overall tool structure. Such a "playground" for developing and exploring new methods for debugging does not exist in HPC. There is also the intriguing possibility that many of the 30-odd tools mentioned in §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* [10] 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

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

oddEvenSort()

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

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.

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

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

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 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 [11] and other data structures [12], we have adapted the Nested Loop Recognition (NLR) algorithm by Ketterlin et al. [13] 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

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 ^ 2	L1 ^ 4	L0 ^ 4	L1 ^ 2
MPI_Finalize	MPI_Finalize	MPI_Finalize	MPI_Finalize

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	×	×	×		×	×

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.

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 [14] owes most of its success for being able to efficiently collect stack traces (nested sequences of function calls), organize them as prefix-trees, and equivalence the processes into teams that evolve in different ways. Coalesced stack trace graphs (CSTG, [15]) have proven effective in locating bugs within Uintah [16] and perform stat-like equivalence class formation, albeit with the added detail of maintaining calling contexts. Inspired by these ideas, FCA-based clustering provides the next logical level of refinement in the sense that (1) we can pick any of the multiple attributes one can mine from traces (e.g., pairs of function calls, memory regions accessed by processes, locks held by threads, etc.), and (2) form this equivalencing relation quite naturally by computing the Jaccard distance between processes/threads. In general, such a classification is powerful enough to distinguish structurally different threads from one another (e.g., MPI processes from OpenMP threads in hybrid MPI+OpenMP applications) and reduce the search space for bug location to a few representative classes of traces that are distinctly dissimilar.[1]

A *formal context* is a triple $K = (G, M, I)$ where G is a set of **objects**, M is a set of **attributes**, and $I \subseteq G \times M$ is an incidence relation that expresses *which objects have which attributes*. Table 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). 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().

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

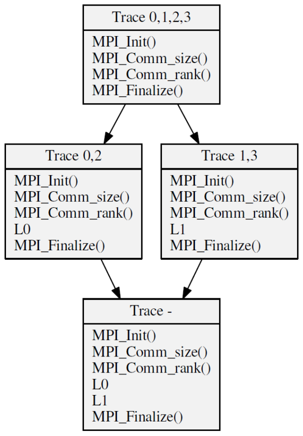


Figure 3: Sample Concept Lattice from Object-Attribute Context in Table IV

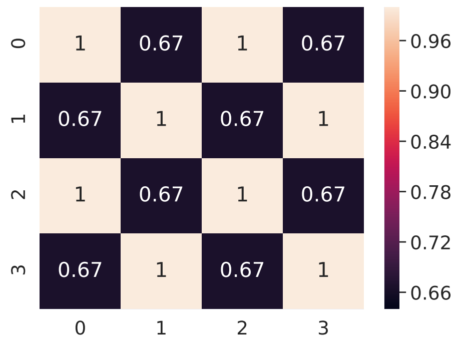


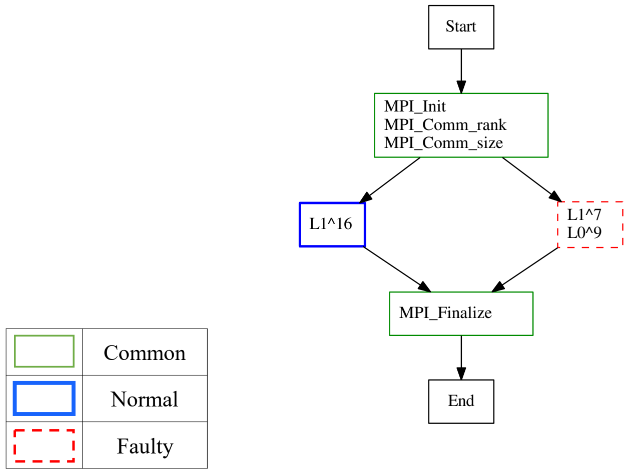
Figure 4: Pairwise Jaccard Similarity Matrix (JSM) of MPI Processes in Sample Code

- The bottom node signifies that none of the traces share all attributes.
- The middle nodes show that T_0 and T_2 are different from T_1 and T_3.

The complete pairwise Jaccard Similarity Matrix (JSM) can easily be computed from concept lattices. For large-scale executions with thousands of threads, it is imperative to employ incremental algorithms to construct concept lattices (detailed in Section 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 JSM$_D$

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



(a) Legend (b) swapBug

Figure 5: diffNLR Example

We sort the suggestion table based on the *B-score* similarity metric of two hierarchical clusterings [17] (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** (identical to the ones in Table III), 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*.

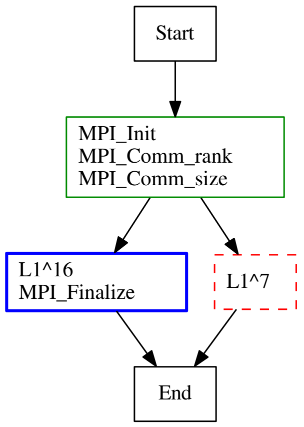


Figure 6: dlBug

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 [18].

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

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

Figure 5b 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 6) shows that, while T_5 executed MPI_Finalize, T'_5 got stuck after executing L1 seven times and never reached MPI_Finalize.

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

III. Algorithms Underlying DiffTrace

A. Nested Loop Recognition (NLR)

We build NLRs based on the work by Ketterlin and Clauss [13], who use this algorithm for trace compression, and the work by Kobayashi and MacDougall [19], 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).

```
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

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.

B. Concept Lattice Construction

The efficiency of algorithms for concept lattice construction depends on the sparseness of the formal context [20]. 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 [21] to extract attributes (Table V) from each trace (object) and inject them into an initially empty lattice. Notice that our

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

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. [22]) 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. [17] 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 [23]. Providing serial CPU and/or single-GPU code, ILCS executes this code in parallel between compute nodes (MPI) and within them (OpenMP and CUDA).

To evaluate DiffTrace, we manually injected MPI-level and OMP-level bugs into the Traveling Salesman Problem (TSP) running on ILCS (Listing 1). The injected bugs simulate real HPC bugs such as deadlocks. Moreover, we inserted “hidden” faults that do not crash the program such as violations of critical sections and semantic bugs. The goal was to see how effectively DiffTrace can analyze the resulting traces and how close it can get to the root cause of the fault. .

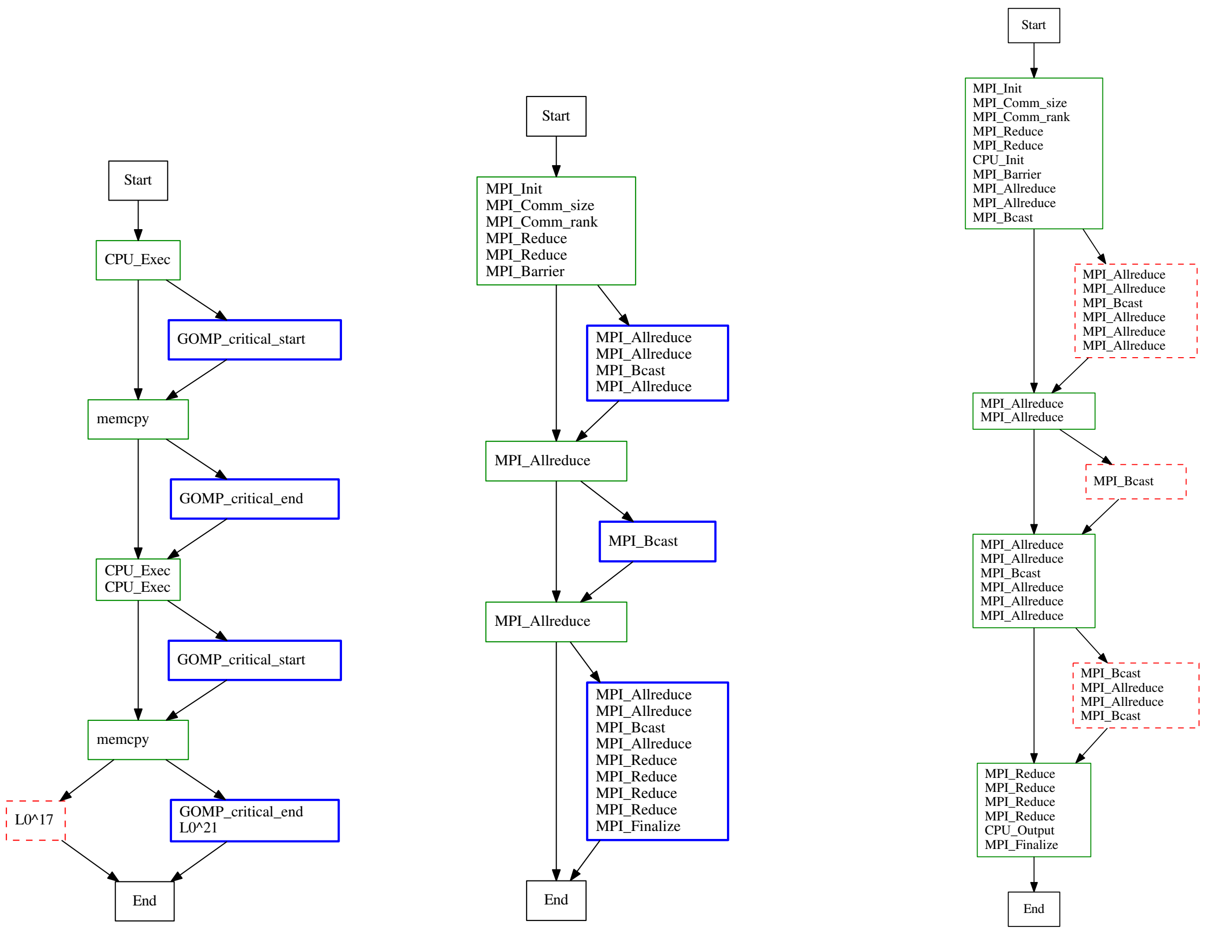
```
main(argc, argv) {
 ... // initialization
 MPI_Init();
 MPI_Comm_size();
 MPI_Comm_rank(my_rank);
 ... // Obtain number of local CPUs and GPUs
 MPI_Reduce(lCPUs, gCPUs, MPI_SUM); // Total # of CPUs
 MPI_Reduce(lGPUs, gGPUs, MPI_SUM); // Total # of GPUs
 champSize = CPU_Init();
 ... // Memory allocation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

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

Listing 1: ILCS Overview

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

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

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



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

Figure 7: Three diffNLR outputs

The linkage method that converts JSMs to flat clustering is "ward" for all of the top reported suspicious traces that we removed from tables for better readability. Ward linkage function in SciPy uses *Ward variance minimization algorithm* to calculate the distance between newly formed clusters [22]. Furthermore, we show diffNLRs for selected traces.

A. ILCS-TSP Workflow

The TSP code starts with a random tour and iteratively shortens it using the 2-opt improvement heuristic [24] until a local minimum is reached. ILCS automatically and asynchronously distributes unique seed values to each worker thread, runs the TSP code, reduces the results to find the best solution, and repeats these steps until the termination criterion is met. It employs two types of threads per node: a *master* thread (MPI process) that handles the communication and local work distribution and a set of *worker* threads (OpenMP threads) that execute the provided TSP code. The master thread forks a worker thread for each detected CPU core. Each worker thread continually calls `CPU_Exec()` to evaluate a seed and records the result (lines 14-20). Once the worker threads are running, the master thread's primary job is to scan the results of the workers to find the best solution computed so far (i.e., the local champion). This information is then globally reduced to determine the current system-wide champion (lines 22-32). ILCS terminates the search when the quality has not improved over a certain period (lines 33-34).

B. OpenMP Bug: Unprotected Memory Access

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

Table VI lists the top suspicious traces that DiffTrace finds when injecting this bug. Each row presents the results for different filters and attributes. For example, the filter "11.mem.ompcit.cust.0K10" removes all function returns and .plt calls from the traces and only keeps memory-related calls,

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

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

OpenMP critical-section functions, and the custom function “CPU_Exec”. The “K10” at the end of filter means that the filtered traces are converted into an NLR with K=10. The bold numbers in the rightmost column of the table flag trace 6.4 (i.e., process 6, thread 4) as the trace that was affected the most by the bug.

The corresponding diffNLR(6.4) presented in Figure 7a clearly shows that the normal execution of ILCS (green and blue blocks) protects the `memcpy` while the buggy execution (green and red blocks) does not. Here, L0 represents `CPU_Exec`, which is called multiple times in both the fault-free and the buggy version (the call frequencies are different due to the asynchronous nature of ILCS).

C. MPI Bug: Deadlock Caused by Fault in Collective

By forcing process 2 to invoke MPI_Allreduce (line 24) with a wrong size, we can inject a *real deadlock*. Because the deadlock happens early in the execution, the resulting traces are very different from their fault-free counterparts. Consequently, DiffTrace marks almost all processes as suspicious (cf. Table VII). Clearly, this is not helpful for debugging. Nevertheless, diffNLR still yields useful information. Since most of the traces are suspicious, we do not know which one the real culprit is and randomly selected trace 4. By looking at the diffNLR(4) output shown in Figure 7b, we immediately see that both the normal and the buggy trace are identical up to the invocation of MPI_Allreduce. This gives the user the first (correct) hint as to where the problem lies. Beyond this point, the bug-free process continues to the end of the program (it reaches the MPI_Finalize call) whereas the buggy process does not. The last entry in the buggy trace is a call to MPI_Allreduce (the last green box), indicating that this call never returned, that is, it deadlocked. This provides the user with the second (correct) hint as to the type of the underlying bug.

D. MPI Bug: Wrong Collective Operation

By changing the MPI_MIN argument to MPI_MAX in the MPI_Allreduce call on line 24 of Listing 1, the semantics of ILCS change. Instead of computing the best answer, the modified code computes the worst answer. Hence, this code variation terminates but is likely to yield the wrong result. We injected this bug into process 0.

The first few suspicious processes listed in Table VIII are inconclusive. However, the filters that include MPI all agree

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

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

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

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

that process 5 changed the most. Looking at the corresponding diffNLR(5) output in Figure 7c makes it clear why process 5 was singled out. In the buggy run, it executes many more MPI_Bcast calls than in the bug-free run because the frequency in which local “optimums” are produced has changed. Though this should affect all traces equally, which has reflected in the diffNLR of other traces. We are presenting these tables and figures to show that DiffTrace can reveal the impact of silent bugs like the wrong operation. Such data representation via suggested tables and diffNLRs helps developers to gain insight into the general behavior of the execution. More accurate results can be obtained by refining the parameters and collecting more profound traces (e.g., ParLOT(all images)). This would be part of our future work to find the set of parameters for different classes of bugs to maximize accuracy.

V. LULESH2 Examples

Our ultimate goal is to apply DiffTrace to complex HPC codes. As a more complex example, we have executed the single-cycle LULESH2[6] with 8 MPI processes and 4 OMP threads (system configuration described in §IV) and collected ParLOT (main image) function calls.

Before bug injection, we analyzed LULESH2 traces and computed some statistics to gain insight into the general control flow of LULESH2 and also to evaluate DiffTrace’s performance and effectiveness. Our primary results show that ParLOT instruments and captures **410 distinct function** calls on average per process, and stores them in compressed trace

Table IX: Ranking Table for LULESH

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

files of size less than **2.8 KB** on average per thread. Upon decompression, each per process trace file turns into a sequence of **421503** function calls on average. The equivalent NLR of each trace file reduces the sequence size by a factor of **1.92** and **16.74**, for constant K set to 10 and 50, respectively.

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

Table IX reflects the ID of processes (rightmost column) that DiffTrace's ranking system suggests as the most affected traces by the bug. Since the fault in process 2 prevents other processes from making progress and successfully terminate, all of the process IDs appeared in the table. The generated diffNLRs clearly showed the point at which each process stopped making progress. Due to lack of space, we did not include the relatively large diffNLRs of LULESH in this paper. However, all diffNLRs and related observations are available online via [25].

VI. Related Work

Three major recent studies have emphasized the need for better debugging tools *and* the need to build a community that can share debugging methods and infrastructure: the DOE report mentioned earlier [3], an NSF workshop [26], and an ASCR report on extreme heterogeneity [27]. Our key contribution in this paper is a fresh approach to debugging that (1) incorporates methods to debug across the API-stack by resorting to binary tracing and thereby being able to "dial into" MPI bugs and/or OpenMP bugs (as shown in the ILCS case study), (2) makes initial triage of debugging methods possible via function-call traces, and (3) enables the verification community to cohere around DiffTrace by allowing other tools to extend our toolchain (they can tap into it at various places).

Many HPC debugging efforts have emphasized the need to highlight dissimilarities and incorporate progress measures on loops. We now summarize a few of them. AutomaDeD [28][29] captures the application's control flow via Semi Markov Models and detects outlier executions. PRODOMETER [30] detects loops in AutomaDeD models and introduces the notion of *least progressed tasks* by analyzing *progress dependency graphs*. DiffTrace's DiffNLR method does not (yet) incorporate progress measures; it only computes changes in loop *structure*. Prodometer's methods are ripe for symbiotic incorporation into DiffTrace. We also plan to incorporate *happens-before* computation as a progress measure using FCA-based algorithms by Garg et al. [31], [32]. FCA-based approaches have been widely used in data mining [33], machine learning [34], and information retrieval [35].

In terms of computing differences with previous executions, we draw inspirations from Zeller's delta-debugging [36] and De Rose et al.'s relative debugging [10]. The power of equivalence classes for outlier detection is researched in STAT [14], which merges stack traces from processes into a prefix tree, looking for equivalence-class outliers. STAT uses the StackWalker API from Dyninst [37] to gather stack traces and efficiently handles scaling issues through tree-based overlay networks such as MRnet [38]. D4 [39] detects concurrency bugs by statically analyzing source-code changes, and DMTracker [40] detects anomalies in data movement. The communication patterns of HPC applications can be automatically characterized by diffing the communication matrix with common patterns [41] or by detecting repetitive patterns [42]. ScalaTrace [43] captures and compresses communication traces for later replay. Synoptic [44] is applied to distributed system logs to find bugs.

VII. Discussions & Future Work

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

In our future work we will improve DiffTrace components as follows: (1) Optimizing them to exploit multi-core CPUs, thus reducing the overall analysis time; (2) Converting ParLOT traces into Open Trace Format (OTF2) [45] by logically timestamping trace entries to mine temporal properties of functions such as *happened-before* [46]; (3) Conducting systematic bug-injection to see whether concept lattices and loop structures can be used as elevated features for precise bug classifications via machine learning and neural network techniques; and (4) Taking up more challenging and real-world examples to evaluate DiffTrace against similar tools, and release it to the community.

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

References

[1] A. Allinea. Allinea ddt. [Online]. Available: https://www.allinea.com/products/ddt

[2] C. Gottbrath and P. Thompson, "Totalview tips and tricks," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188465

[3] 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

[4] 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

[5] 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/

[6] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[7] 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.

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

[9] T. G. Mattson, B. A. Sanders, and B. Massingill, *Patterns for parallel programming*. Pearson Education, 2013.

[10] L. D. Rose, A. Gontarek, A. Vose, R. Moench, D. Abramson, M. N. Dinh, and C. Jin, "Relative debugging for a highly parallel hybrid computer system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, 2015, pp. 63:1–63:12. [Online]. Available: https://doi.org/10.1145/2807591.2807605

[11] 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

[12] 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

[13] 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

[14] 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.

[15] D. de Oliveira, A. Humphrey, Q. Meng, Z. Rakamaric, M. Berzins, and G. Gopalakrishnan, "Systematic debugging of concurrent systems using coalesced stack trace graphs," in *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, September 2014. [Online]. Available: http://www.sci.utah.edu/publications/Oli2014a/OliveiraLCPC2014.pdf

[16] J. D. de St. Germain, S. G. Parker, J. McCorquodale, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing, HPDC'00, Pittsburgh, Pennsylvania, USA, August 1-4, 2000.*, 2000, pp. 33–42. [Online]. Available: https://doi.org/10.1109/HPDC.2000.868632

[17] 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

[18] 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

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

[20] 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

[21] 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.

[22] 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/

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

[24] D. Johnson and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization," *Local Search in Combinatorial Optimization*, vol. 1, 01 1997.

[25] "IEEE Cluster'19 DiffTrace Material." [Online]. Available: http://bit.ly/IEEECluster19-DiffTrace

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

[27] J. S. Vetter, "Extreme Heterogeneity 2018: Productive Computational Science in the Era of Extreme," Jan. 2018, report for DOE ASCR Basic Research Needs Workshop on Extreme Heterogeneity, https://science.energy.gov/ /media/ascr/pdf/programdocuments/docs/-2018/2018-09-26d-Extreme_Heterogeneity_BRN_report.pdf.

[28] 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.

[29] 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

[30] 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

[31] 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.

[32] V. K. Garg, *Introduction to lattice theory with computer science applications*. Wiley, 2015.

[33] B. Ganter, P. A. Grigoriev, S. O. Kuznetsov, and M. V. Samokhin, "Concept-Based Data Mining with Scaled Labeled Graphs," in *Conceptual Structures at Work*, K. E. Wolff, H. D. Pfeiffer, and H. S. Delugach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 94–108.

[34] S. O. Kuznetsov, "Machine Learning and Formal Concept Analysis," in *Concept Lattices*, P. Eklund, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 287–312.

[35] D. I. Ignatov, "Introduction to formal concept analysis and its applications in information retrieval and related fields," *CoRR*, vol. abs/1703.02819, 2017. [Online]. Available: http://arxiv.org/abs/1703.02819

[36] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, 1999, pp. 253–267. [Online]. Available: https://doi.org/10.1007/3-540-48166-4_16

[37] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, 1995. [Online]. Available: https://doi.org/10.1109/2.471178

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

[39] 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

[40] 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.

[41] 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

[42] 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.

[43] 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).

[44] 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

[45] OTF2 Developer Community, "Open trace format version 2 (otf2)," Jul. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3356709

[46] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563