DiffTrace: Efficient Whole-Program Trace Analysis and Diffing

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

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

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

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

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

I. Introduction

BEGIN Ganesh

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 at a detailed level 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 [**?**] 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 tools and approaches, 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 share their solutions and extend a common framework **Need more build-up; will do later.**

In this paper, we provide our fundamentally fresh look at debugging. We point out three significant problems that we have addressed in our work, and provide our preliminary solutions backed up by case studies. While our work has not (yet) addressed the situations in which millions of threads and thousands of processes run for days and produce an error, we strongly believe that we can get there only through a series of *rigorous* approaches that overcome key limitations found in conventional debugging approaches in a step-by-step manner, accompanied by careful measurements of the merits of the new approach. The main contribution of this paper is the first such critical measurements of our proposed approach.

a) Problem-1: Need to Generalize Approaches for Outlier Detection:: 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 also waste an expert developer's time.

Solution to Problem-1: Whole Program Tracing for Debugging: In this setting, our first contribution is a debugging approach in which the application is not merely run with a single symptom-specific tool attached as described earlier. Instead, we collect *whole program traces* of function calls and returns, using a PIN-based function call tracing facility called ParLoT that we have developed and previously reported [**?**]. We store these traces for potential examination by multiple tools and approaches. The advantage of whole program binary tracing supported in ParLoT is that we can collect function calls at *any desired level of abstraction*. For instance, if the programmer wants to cover activities at the MPI level, the OpenMP level and perhaps even lower levels (e.g., the MPI library or the OpenMP runtime), they can do so using ParLoT.

Clearly, the more APIs at which function calls are recorded, the more burdensome trace collection becomes. However, the advantage is that correspondingly more tools can then be applied to the collected traces. There is always a sweet-spot in this trade-off space, depending on the particular debugging situation. However, our fundamental insight is that *given the inevitability of heterogeneous programming* (the use of multiple concurrency models), it is important to be collecting traces from a few related APIs at a time, so that one can study bugs in one of the concurrency models or a bug resulting from a bad cross-model interaction from a *single run of the program*.

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 traces, it is possible to pick out MPI-level bugs or OpenMP-level bugs.

While we have not covered all these combinations in our work so far, the main contribution claimed is the ability to cover multiple APIs while debugging, and without re-executions or guess-work.

While this approach to whole-program tracing may sound extremely computation intensive, we employ novel on-the-fly compression techniques within ParLoT. In our previous study [?], we report compression efficiencies exceeding 16,000. This allows us to bring out the function call traces without significantly burdening the memory subsystem or I/O networks in the HPC cluster.

b) Problem-2: Need to Generalize Approaches for Outlier Detection:: Given that outlier detection is central to debugging, it is important to be employing efficient representations of the traces collected from threads and processes so that one can compute *distances* between these traces more systematically, without involving human reasoning in the loop. The representation must also be versatile enough to be able to "Diff" the traces[1] with respect to *an extensible number of vantage points*. These vantage points could be diffing with respect to process level activities, diffing with respect to 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: Use of Concept Lattices in Debugging: In DiffTrace, we employ *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 employ 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 more refinement approaches for outlier detection are new in HPC debugging.

c) Problem-3: Loop Detection:: 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.

Solution to Problem-3: Rigorous Approaches to Loop Analysis: In DiffTrace, we employ the notion of NLRs (what does it stand for?) 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 run differently in a manner that informs the developer engaged in debugging.

[1]Hence the name of our tool, **DiffTrace**.

To summarize, the key contributions of this paper are the following [[fix the section numbers later]]:

- A method to organize function call traces collected from processes and threads into concept lattices, and a method to detect loops from dynamic traces (Section **??**).
- Details of the algorithms employed in DiffTrace (Section **??**).
- Experimental studies on a heterogeneous program called Iterated Local Champion Search (ILCS, Section **??**).
- Strengths and limitations of DiffTrace, plans for future work (Section **??**).

END Ganesh

[[Ganesh and Saeed have written some text before for the intro which is available in v0/intro.tex (also available but commented in current file). Current version is based on our discussion on May 8th]]

- Importance of whole program diffing : understand changes, debug (DOE REPORT [1])
- Efficient tracing supports selective monitoring at multiple levels
  - Bugs not there at a predictable API level
  - Prior work (ParLoT) supports whole program tr.
- Dissimilarity is important to know: bugs, changes during porting,...
- Key enablers of meaningful diffing:
  - Formal concepts (novel contrib to debugging)
  - Loop detection (loop diffing can help)
- Importance, given the growing heterogeneity

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

In summary, this paper makes the following main contributions:

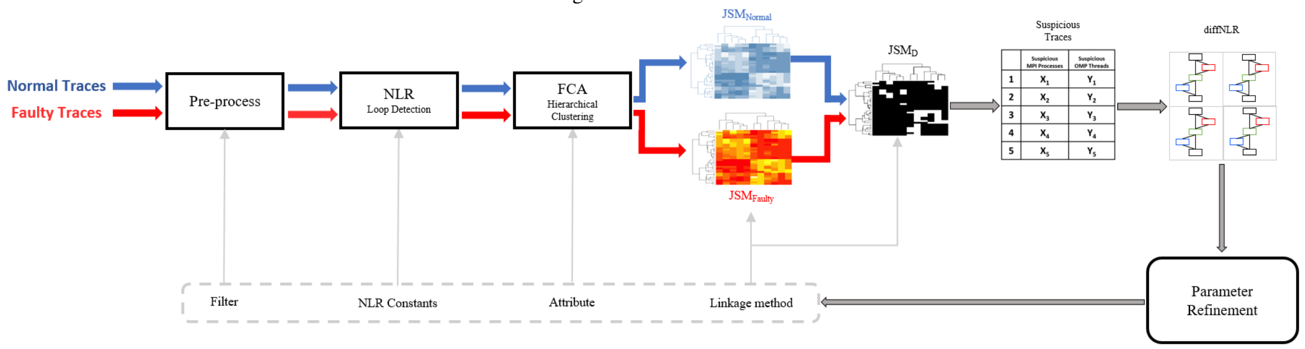
- A tunable tracing and trace-analysis tool-chain for HPC application program understanding and debugging
- A variation of the NLR algorithm to compress traces in lossless fashion for easier analysis and detecting (broken) loop structures
- An FCA-based clustering approach to efficiently classify traces with similar behavior
- A tunable ranking mechanism to highlight suspicious trace instances for deeper study
- A visualization framework that reflects the points of differences or divergence in a pair of sequences.

The rest of the paper is as follows:
- Sec 2: Background
- Sec 3: Components
- Sec 4: Case Study: ILCS
- Sec 5: Related Work
- Sec 6: Concluding Remarks

II. DiffTrace Overview

DiffTrace employs ParLOT's [2] whole-program function-call and return traces for fault detection and localization.

Figure 1. DiffTrace Overview



ParLOT is based on the dynamic binary instrumentation toolkit Pin [3] and incrementally compresses the generated traces on-the-fly. It can capture functions at two levels: the *main image*, which does not include library code, and *all images*, which includes 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 the majority 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. Despite these benefits, raw ParLOT traces are just compressed sequences of executed function IDs. Moreover, using ParLOT on a large-scale HPC application tends to yield thousands of long traces. These traces need to be filtered, grouped, and transformed into a meaningful representation that reveals important facts about the dynamic behavior of the program and hides uninteresting details. This is what DiffTrace does.

The DiffTrace tool-chain (Figure 1) provides the sophisticated infrastructure needed for *iterative and configurable trace search space reduction for localizing errors* in a set of traces. Considering a "successful" termination of the application as *normal behavior*, DiffTrace iteratively compares the traces from faulty runs against those from normal runs to progressively zero in on any *abnormal behavior*. Each abnormal behavior is considered a potential fault cause or manifestation. However, faults may occur or influence the program behavior at different locations and granularities. A fault may get triggered at some point in the code and be immediately visible, or it may only manifest itself some time later. DiffTrace feeds the normal and faulty traces through a sequence of data transformation and classification steps. It then suggests a few traces for a more detailed study (e.g., comparing the traces from a different vantage point) based on *measuring and locating points of differences*. Since each DiffTrace step can be parameterized in multiple ways, additional evidence about what went wrong may be obtained

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, ...);
  }
  ...
 }
}
```

iteratively. In each iteration, DiffTrace puts the spotlight on a separate aspect of the program's dynamic behavior.

DiffTrace starts by *pre-processing* the traces. ParLOT traces are highly compressed and typically contain functions that are not of interest to the current analysis. Thus, DiffTrace first decompresses and prunes the traces. Loops appear as *repetitive patterns*, resulting in often long but redundant trace information. A "nested loop recognition" mechanism then extracts the loops from the traces. The resulting information not only serves as a lossless abstraction to ease the rest of the trace analysis but also as a per-thread "*measure of progress*". The control flow in parallel programs often follows a *specific pattern* such as SPMD, odd-even, or master/slave [**?**]. As a consequence, the traces of a single execution tend to fall into just *a few* "equivalence classes". definition of "fault" maybe? a software bug, node failure, a library version update or porting to a new system that causes failure to a working software To study the impact of a fault on the execution, DiffTrace classifies the collected traces and measures the *similarity* between the equivalence classes from the faulty and the fault-free execution. The basic idea is to find out which traces (and, consequently, processes/threads) are falling into a different class (i.e., cluster) when the fault is introduced. Based on the observed similarity among the two clusterings, DiffTrace suggests the topmost suspicious traces, i.e., the traces that suffer the most from the fault, for deeper analysis.

The rest of this section illustrates the DiffTrace components on sets of ParLOT traces from the aforementioned MPI

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

odd/even sort example (Figure 2). Odd/even sort is a parallel version 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. The for loop in line 4 of `oddEvenSort()` iterates over the phases of the algorithm. Based on the phase, the appropriate partner for each rank is computed by the function `findPtr()` (line 6). The odd/even ranks then exchange their chunks of data (lines 9-13) and sort, merge, and copy operations are performed on the received data (denoted by `...` in line 15 for simplicity).

According to the MPI standard, `MPI_Send` may *block*, meaning MPI may buffer the outgoing message, in which case the send call completes before a matching receive is invoked, or it may choose not to buffer the message. In the latter case, the send call will not complete until a matching receive has been posted and the data has been transferred to the receiver. So, based on the dynamic behavior, swapping the statements on lines 11 and 12 of Figure **??** might end up causing a deadlock. DiffTrace can extract evidence from the traces to locate the cause of such deadlocks and other faults.

A. 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. All other trace information is discarded.

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.

B. Nested Loop Representation

Most programs, and HPC applications in particular, spend most of their runtime in *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.

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

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

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 (Nested Loop Representation), 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 ^ 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.

connect this section to next

C. Hierarchical Clustering via FCA

HPC applications often follow a *regular pattern* of executed functions. Due to this characteristic, per-thread function-call traces can generally be classified into *a few equivalence groups* based on their content. Such a classification would distinguish structurally different threads from each other (e.g., MPI processes from OpenMP threads in hybrid MPI+OpenMP applications) and reduce the search space into just a few representative classes of traces. In addition, the set of per-thread traces should be studied as "a whole" since there is a strong conceptual and causal relation among threads/processes. To integrate the collected traces into a *single model of execution* and forming "equivalence classes of traces", we have adapted the idea of *Structural Clustering* [7] by applying *formal concept analysis* (FCA) [8] techniques to our traces.

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*

Figure 3. Formal Concept Definition

For subsets A ⊆ G of objects and subsets B ⊆ M of attributes, one defines two derivation operators as follows:
A' = {m ∈ M | (g,m) ∈ I for all g ∈ A}, and dually
B' = {g ∈ G | (g,m) ∈ I for all m ∈ B}.
Applying either derivation operator and then the other constitutes two closure operators:
A ↦ A'' = (A')' for A ⊆ G (extent closure), and
B ↦ B'' = (B')' for B ⊆ M (intent closure).

Definition of a formal concept: a pair (A,B) is a formal concept of a context (G, M, I) provided that:
A ⊆ G, B ⊆ M, A′ = B, and B′ = A.
Equivalently and more intuitively, (A,B) is a **formal concept** precisely when:

- every object in A has every attribute in B,
- for every object in G that is not in A, there is some attribute in B that the object does not have,
- for every attribute in M that is not in B, there is some object in A that does not have that attribute.

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	×	×	×		×	×

which attributes. Table IV shows the formal context of the preprocessed odd/even sort traces.

In this context, attributes are trace entries (function calls or detected loop bodies) without their frequency (e.g., iteration counts). 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*. Definition of formal concept (needed?) figure 3 :

A *concept lattice* can be derived from a *formal context* by specifying *formal concepts* (Figure 3) and a *partial order* on them. Concept lattices are represented as a directed acyclic graph where concepts are nodes and the order on them determines the edges. Figure 4 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.

Once the redundant labels are removed from the lattice, each object (trace) and attribute appears in the lattice exactly once. Consequently, the nodes of the lattice form the desired grouping, since it is guaranteed that each trace belongs to exactly one group. However, the concept lattice itself does not provide similarity values for the distinct groups of traces.

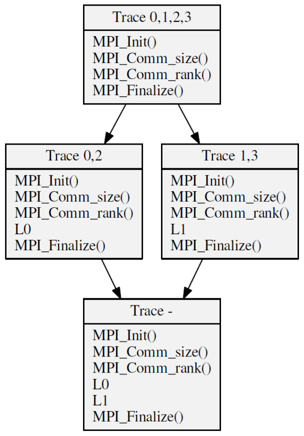


Figure 4. Sample Concept Lattice from Obj-Atr Context in Table IV

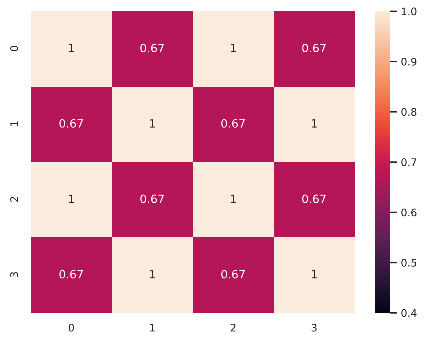


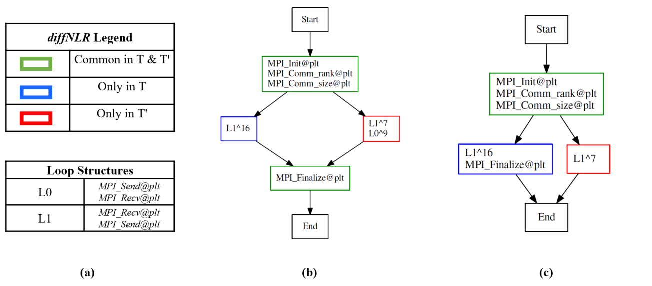
Figure 5. Pairwise Jaccard Similarity Matrix (JSM) of MPI processes in sample code

The *Jaccard Index*, also known as *Intersection over Union*, measures the *distance* between sets A and B in terms of the ratio of the *intersection* size of A and B over the size of their *union*. The complete pairwise Jaccard Similarity Matrix (JSM) can easily be computed from the concept lattice.

For any pair of (T_i, T_j), the number of attributes in the Lowest Common Ancestor (LCA) node of T_i and T_j in the lattice is the number of attributes that T_i and T_j have in common (intersection). The sum of the number of attributes of the nodes on the path from each T_i and T_j to their LCA is the union. This property is one of the motivations for using concept lattices as classifier since it makes computing the union and intersection easy and fast.

Some algorithms for extracting concepts from contexts and constructing the concept lattice require the whole context to be present in the memory. For large-scale executions with thousands of threads, this is not feasible. Through an incremental concept-lattice-construction approach, DiffTrace extracts attributes (table that shows attributes) from NLR traces and injects them into a concept lattice one trace at a time (cf. Section III-B). Figure 5 shows the heatmap (explain this) of the JSM obtained from the concept lattice in Figure 4. DiffTrace uses the JSM to form equivalence classes of traces

Figure 6. (a) The legend of *diffNLR* and the list of loop structures (b) *diffNLR(5)* of *swapBug* (c) *diffNLR(5)* of *dlBug*



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.

D. Detecting Suspicious Traces via DiffJSM

So far, we have explained how DiffTrace can narrow down the search space from numerous long traces to just a few equivalent JSMs (i.e., clusters). $JSM_{normal}[i][j]$ ($JSM_{faulty}[i][j]$) shows the Jaccard similarity score of T_i and T_j from the normal trace (T'_i and T'_j).

However, we are interested in detecting what changed the most due to the fault. The hierarchical clustering based on *DiffJSM*, followed by the subtraction of the faulty JSM from its corresponding normal JSM, typically puts the trace(s) that changed the most in a separate, often singleton, cluster and thus detects the outlier.

$DiffJSM = |JSM_{faulty} - JSM_{normal}|$

The resulting outlier traces are candidates for the potential cause of the change in the program behavior and thus a potential fault root cause or fault manifestation. However, a single iteration of DiffTrace (with a single set of parameters shown as a dashed box in Figure 1) may not detect the problem. To improve the accuracy, we have sorted the suggestion table based on the *B-score* similarity metric of two hierarchical clusterings [9] (cf. Section III-C).

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 6-(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 6-(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 6-(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. NLR

Nested Loop Recognition (NLR) algorithm [6] is originally designed for prediction and compression of data access traces (memory addresses) through detecting the “linear progression function” in the sequence of addresses. Two main operations of NLR algorithm are 1) recognizing the start of a loop and forming its initial syntactic structure and 2) recognizing if what follows a loop is just another iteration and extend the upper bound. These operations are performed incrementally and recursively to recognize loops at different depths (i.e., a linear function of the loop index and depth) until the input sequence is completely consumed and no more loop is detected.

Previous to NLR, Kobayashi [11] proposed a similar bottom-up strategy to build loop nests from a trace (sequence of string instructions) where each recognized loop is replaced with a new symbol. Identical loops at different locations can be identified by remembering the new symbol. This process is restarted once the whole trace has been analyzed for depth-2 loops, and so on until some replacement is performed.

Inspired by Kobayashi approach, we have re-implemented the NLR algorithm (DiffTrace-NLR) for detecting repetitive patterns in a sequence of function calls (trace) and summarizing the trace to NLR representation. DiffTrace-NLR works by incrementally pushing trace entries (function IDs) into a stack of *elements* (i.e., function IDs and already detected loop structures). Whenever an element is pushed to 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

Each loop structure **LS** is a tuple of (loop body **LB**, loop count **LC**) where LB is a sequence of elements and LC is an integer showing the frequency of consecutive occurrence of LB. To avoid coincidental regularity, the algorithm needs at least *three* consecutive terms to form an LS. The Reduce procedure checks to see if any three consecutive subsequences on top of the stack are *isomorphic*. In our context, two sequences are isomorphic if both have equal lengths and identical corresponding elements. If the check passes, then the top three consecutive subsequences (LB) are popped from the stack, and the LS=(LB,3) is pushed onto the stack. Otherwise, top elements of the stack are compared against the potential LS behind them to increment the LC in case of isomorphism. If either of the above examinations successes, the procedure restarts on the recently modified stack for detecting or extending potential nested loops.

We store all distinct LBs into 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 in other traces of the same execution. The maximum length of subsequences to examine is decided by a fixed priori K. The complexity of the NLR algorithm is $\Theta(K^2N)$ where N is the size of the input.

Table V
ATTRIBUTES MINED FROM TRACES

Attributes {atr:freq}			
atr		freq	
Single	Each entry of the trace	**Actual**	The observed frequency
		Log10	log10 of the observed frequency
Double	Each pair of consecutive entry	**noFreq**	No frequency

B. Concept Lattice Construction

There exist algorithms that extract concepts and their partial order from a formal context; each has its pros and cons. Given the density/sparseness of the formal context, the efficiency of algorithms varies [12]. The basic Ganter's *Next Closure* algorithm [8] construct the lattice from the *batch* of context and requires the whole context to be present in the main memory. For large scale HPC traces, such an approach is inefficient.

We have implemented Godin's *incremental* algorithm [13] to extract attributes (Table V) from each trace (object) and inject them to an initially empty lattice. Every time a new object with its set of attributes added to the lattice, an *update* procedure minimally modifies/adds/deletes edges and nodes.

This procedure is guaranteed to preserve the validity of the concept lattice properties.

The extracted attributes are in form of $\{$*atr:freq*$\}$. *atr* is either a single entry of the trace NLR or a consecutive pair of entries. *freq* is a parameter to adjust the impact of frequency of each *atr* 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., number of traces).

C. Hierarchical Clustering, Construction and Comparison

1) SciPy: The table of suspicious traces is derived from the DiffJSM by hierarchical clustering algorithms in SciPy API version 1.3.0. [14]. DiffJSMs provide pair-wise dissimilarity measurement that can be used to combine traces (forming initial clusters). However, to derive a flat clustering of traces to detect the outlier(s)(i.e., suspicious traces), a *linkage* function is required to measure the distance between sets of traces. SciPy provides a wide range of linkage functions: single, complete, average, weighted, centroid, median, ward.

2) Ranking Table: As shown in figure 1, each component of DiffTrace has some tunable parameters and constants. Consequently, the suggested suspicious traces might not be accurate or drastically changes with a slight change in the parameters. Thus a metric is needed to sort the selected suspicious traces based on. Since each parameter combination would map to a DiffJSM, the metric can be " the distance between two hierarchical clusterings". Fowlkes et al. [9] proposed a method for comparing two hierarchical clusterings by computing their *B-score*. The B-score of two clusterings is computed by counting the number of objects that fall into different clusters. We have not evaluated the accuracy of the proposed idea. However, our initial experiments show that sorting suspicious traces based on the B-score of DiffJSMs is effective and brings interesting traces to attention.

Table VI
ILCSTSP.MC1-MC-6-N1.M.8.AUTO

Filter	Attributes	Link Method	Thresh	B-score	Top Procs (JSMD)	TOP Threads(JSMD)
01.mem.ompall.cust.0K10	sing.actual	average	4	0.308594		6.2 , 6.4 , 5.2 , 5.4 , 7.3 ,
11.mem.ompall.cust.0K10	sing.actual	average	4	0.308594		6.2 , 6.4 , 5.2 , 5.4 , 7.3 ,
01.mem.ompmutex.cust.0K10	sing.actual	weighted	4	0.321883		6.2 , 3.4 , 5.3 , 4.2 , 7.4 ,
11.mem.ompmutex.cust.0K10	sing.actual	weighted	4	0.321883		6.2 , 3.4 , 5.3 , 4.2 , 7.4 ,
11.plt.mem.mpi.ompcrit.cust.0K10	doub.actual	weighted	4	0.324427	3 ,	0.2 , 1.1 , 3.1 , 4.3 , 4.4 , 5.2 ,
01.plt.mem.mpi.ompcrit.cust.0K10	doub.actual	weighted	4	0.324427	3 ,	0.2 , 1.1 , 3.1 , 4.3 , 4.4 , 5.2 ,
01.mem.ompall.cust.0K10	doub.actual	weighted	4	0.33266	3 ,	6.4 , 7.1 , 3.4 , 4.3 , 4.4 , 5.2 ,
11.mem.ompall.cust.0K10	doub.actual	weighted	4	0.33266	3 ,	6.4 , 7.1 , 3.4 , 4.3 , 4.4 , 5.2 ,
01.mem.ompcrit.cust.0K10	sing.actual	weighted	4	0.354396	6 ,	7.2 , 7.4 , 3.4 , 4.2 , 4.3 , 4.4 ,
11.mem.ompcrit.cust.0K10	sing.actual	weighted	4	0.354396	6 ,	7.2 , 7.4 , 3.4 , 4.2 , 4.3 , 4.4 ,
11.plt.mem.mpi.ompcrit.cust.0K10	doub.actual	average	4	0.381646		3.3 , 2.4 ,

Table VII
ILCSTSP.MC1-MC-2-2.M.8.AUTO

Filter	Attributes	Link Method	Thresh	B-score	Top Procs (JSMD)	TOP Threads(JSMD)
01.mem.ompall.cust.0K10	doub.actual	weighted	4	0.309391	3 ,	6.2 , 6.4 , 7.1 , 7.4 , 1.3 , 3.1 ,
11.mem.ompall.cust.0K10	doub.actual	weighted	4	0.309391	3 ,	6.2 , 6.4 , 7.1 , 7.4 , 1.3 , 3.1 ,
11.plt.mem.cust.0K10	doub.actual	weighted	4	0.318003	7 , 4 ,	1.3 , 2.2 , 3.3 , 3.4 , 4.2 , 4.3 ,
01.plt.mem.cust.0K10	doub.actual	weighted	4	0.318003	7 , 4 ,	1.3 , 2.2 , 3.3 , 3.4 , 4.2 , 4.3 ,
01.mem.ompall.cust.0K10	doub.actual	average	4	0.34462	7 , 3 ,	7.1 , 1.3 , 1.4 , 2.2 , 2.3 , 3.1 ,
11.mem.ompall.cust.0K10	doub.actual	average	4	0.34462	7 , 3 ,	7.1 , 1.3 , 1.4 , 2.2 , 2.3 , 3.1 ,
11.plt.mem.mpi.ompcrit.cust.0K10	doub.actual	average	4	0.350702	7 , 3 ,	6.2 , 6.3 , 7.2 , 2.4 , 3.3 , 4.2 ,
01.plt.mem.mpi.ompcrit.cust.0K10	doub.actual	average	4	0.350702	7 , 3 ,	6.2 , 6.3 , 7.2 , 2.4 , 3.3 , 4.2 ,
11.plt.mem.cust.0K10	doub.actual	weighted	3	0.357334	7 ,	2.2 , 3.3 , 3.4 , 4.3 , 4.4 ,
01.plt.mem.cust.0K10	doub.actual	weighted	3	0.357334	7 ,	2.2 , 3.3 , 3.4 , 4.3 , 4.4 ,
01.mem.ompall.cust.0K10	doub.actual	weighted	3	0.380481	3 ,	7.1 , 1.3 , 3.1 , 4.3 , 4.4 ,

IV. Case Study: ILCS

Table VIII
ILCSTSP.AR2-WO-2-NN.M.8.AUTO

Filter	Attributes	Link Method	Thresh	B-score	Top Procs (JSMD)	TOP Threads(JSMD)
01.plt.cust.0K10	doub.actual	average	4	0.392446	6 ,	2.3 , 2.4 , 4.2 , 4.3 , 4.4 ,
11.plt.cust.0K10	doub.actual	average	4	0.392446	6 ,	2.3 , 2.4 , 4.2 , 4.3 , 4.4 ,
01.plt.cust.0K10	sing.log10	centroid	4	0.945946	0 ,	6.2 , 0.4 , 1.1 , 0.1 , 2.3 , 2.4 ,
01.plt.cust.0K10	sing.log10	median	4	0.945946	0 ,	6.2 , 0.4 , 1.1 , 0.1 , 2.3 , 2.4 ,
11.plt.cust.0K10	sing.log10	centroid	4	0.945946	0 ,	6.2 , 0.4 , 1.1 , 0.1 , 2.3 , 2.4 ,
11.plt.cust.0K10	sing.log10	median	4	0.945946	0 ,	6.2 , 0.4 , 1.1 , 0.1 , 2.3 , 2.4 ,
01.plt.cust.0K10	sing.log10	median	3	0.947368	0 ,	6.2 , 0.4 , 1.1 , 0.1 , 2.3 , 2.4 ,
11.plt.cust.0K10	sing.log10	median	3	0.947368	0 ,	6.2 , 0.4 , 1.1 , 0.1 , 2.3 , 2.4 ,
01.plt.cust.0K10	sing.log10	complete	3	1	0 ,	0.1 , 7.1 , 1.1 , 3.1 , 3.3 ,
01.plt.cust.0K10	sing.log10	complete	4	1	0 ,	0.1 , 7.1 , 1.1 , 3.1 , 3.3 ,
01.plt.cust.0K10	sing.log10	average	4	1	0 ,	0.1 , 7.1 , 1.1 , 3.1 , 3.3 ,

Table IX
ILCSTSP.BC2-WR-3-NN.M.8.AUTO

Filter	Attributes	Link Method	Thresh	B-score	Top Procs (JSMD)	TOP Threads(JSMD)
01.plt.cust.0K10	doub.actual	centroid	4	0.512309	2 ,	6.4 , 7.3 , 1.2 , 1.3 , 2.1 , 2.2 ,
11.plt.cust.0K10	doub.actual	centroid	4	0.512309	2 ,	6.4 , 7.3 , 1.2 , 1.3 , 2.1 , 2.2 ,
01.plt.cust.0K10	sing.actual	average	4	0.513221	7 ,	3.4 , 5.2 , 4.2 , 4.3 ,
11.plt.cust.0K10	sing.actual	average	4	0.513221	7 ,	3.4 , 5.2 , 4.2 , 4.3 ,
11.mpi.cust.0K10	sing.actual	median	4	0.544807	0 ,	0.1 , 6.4 , 7.3 , 7.4 , 1.3 , 0.2 ,
11.mpiall.cust.0K10	sing.actual	median	4	0.544807	0 ,	0.1 , 6.4 , 7.3 , 7.4 , 1.3 , 0.2 ,
01.mpiall.cust.0K10	sing.actual	median	4	0.544807	0 ,	0.1 , 6.4 , 7.3 , 7.4 , 1.3 , 0.2 ,
01.mpicol.cust.0K10	sing.actual	median	4	0.544807	0 ,	0.1 , 6.4 , 7.3 , 7.4 , 1.3 , 0.2 ,
11.mpicol.cust.0K10	sing.actual	median	4	0.544807	0 ,	0.1 , 6.4 , 7.3 , 7.4 , 1.3 , 0.2 ,
01.mpi.cust.0K10	sing.actual	median	4	0.544807	0 ,	0.1 , 6.4 , 7.3 , 7.4 , 1.3 , 0.2 ,
01.plt.cust.0K10	doub.actual	centroid	3	0.639268	2 ,	6.4 , 7.3 , 1.2 , 1.3 , 2.1 , 2.2 ,

Table X
ILCSTSP.BC1-WS-3-NN.M.8.AUTO

Filter	Attributes	Link Method	Thresh	B-score	Top Procs (JSMD)	TOP Threads(JSMD)
11.mpi.cust.0K10	sing.actual	weighted	4	0.385229	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
11.mpiall.cust.0K10	sing.actual	weighted	4	0.385229	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
01.mpiall.cust.0K10	sing.actual	weighted	4	0.385229	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
01.mpicol.cust.0K10	sing.actual	weighted	4	0.385229	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
11.mpicol.cust.0K10	sing.actual	weighted	4	0.385229	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
01.mpi.cust.0K10	sing.actual	weighted	4	0.385229	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
01.plt.cust.0K10	sing.actual	weighted	4	0.448188	7 , 3 ,	6.2 , 6.4 , 7.1 , 7.4 , 3.3 , 3.4 ,
11.plt.cust.0K10	sing.actual	weighted	4	0.448188	7 , 3 ,	6.2 , 6.4 , 7.1 , 7.4 , 3.3 , 3.4 ,
11.mpi.cust.0K10	sing.actual	weighted	3	0.465043	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
11.mpiall.cust.0K10	sing.actual	weighted	3	0.465043	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,
01.mpiall.cust.0K10	sing.actual	weighted	3	0.465043	6 ,	6.2 , 6.3 , 6.4 , 7.2 , 7.3 , 7.4 ,

V. Case Study: ILCS

A. Experimental Methodology

So far, we are able to collect whole-program execution traces, preprocess them (decompress, filter, detect loops, extract attributes) and inject each PT to concept lattice data structure. Concept lattices help us having a single model for the execution of HPC application with thousands of processes/threads. Concept lattices also classify PTs based on their Jaccard distance. Full pair-wise Jaccard distance matrix can be extracted from the concept lattice in linear time and reduces the search space from thousands of PTs to just a few equivalent classes of PTs. Studying JSM by itself helps the user to understand the program behavior as a whole, and how each process/thread behaving. However, comparing the JSM of the bug-free version of the application versus the buggy version would reveal insights about how the bug impacted the behavior of the application. In particular, we are interested to see how the bug changes the formation of equivalent classes of PTs. Inspired by a method for comparing two different clustering [9], we count the number of objects (PTs) in each cluster and see which PT(s) fall into different clusters once the bug is introduced. A set of candidate PTs then would be reported to the user for more in-depth study. Here is where we take advantage of diffNLR to see how does the bug changes the control flow of a candidate PT comparing to its corresponding PT of native run.

Table 7 shows different parameters that we can pre-process PTs with. Each combination of these parameters would result in a different concept lattice, thus different JSM and different clusterings. A table similar to XII is created for each injected bug. Each row of the table is showing the set of parameters used to create JSMs. Then by calculating $|JSM(buggy) - JSM(bugfree)|$ we are interested to see which PT changes the most after the bug injected and falls into a single cluster. The object(s) in the cluster with the fewest members (below a threshold) are potential candidates of *threads that are manifesting the bug* and the diff(buggy,bug-free) is in our interest to see how does the bug changes its control flow.

B. Case Study: ILCS-TSP

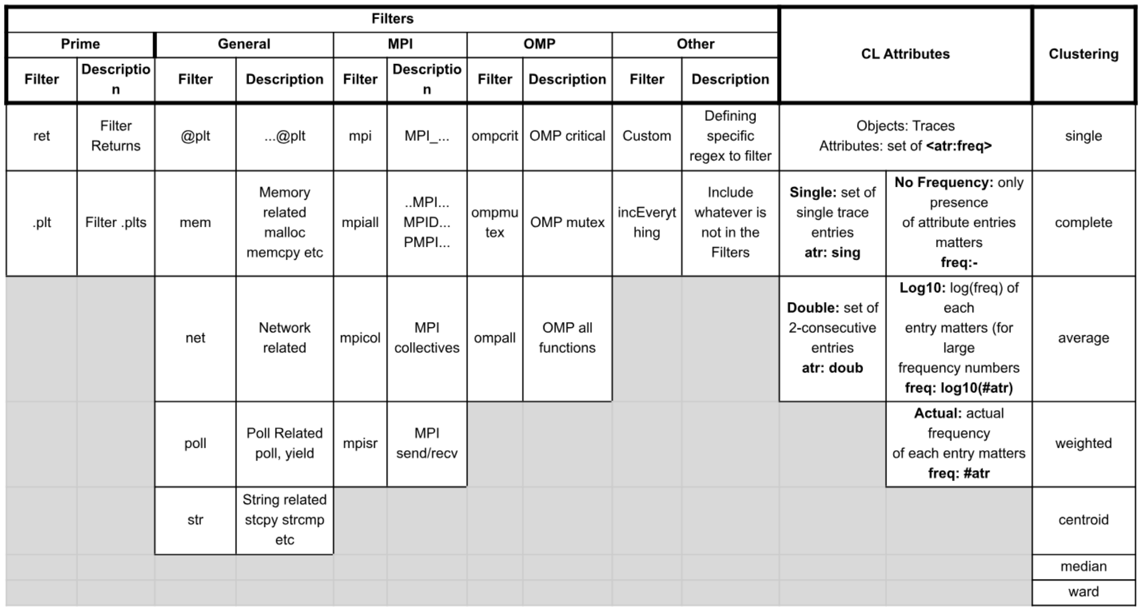
Here is the ILCS framework pseudo-code. User needs to write `CPU_Init()`,`CPU_Exec()` and `CPU_Output()`.

.

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

/* User code for TSP problem */

CPU_Init(){
 // Read In data from cities
 // Calculate distances
```



Filters										CL Attributes		Clustering
Prime		General		MPI		OMP		Other				
Filter	Description	Filter	Description	Filter	Description	Filter	Description	Filter	Description			
ret	Filter Returns	@plt	...@plt	mpi	MPI_...	ompcrit	OMP critical	Custom	Defining specific regex to filter	Objects: Traces Attributes: set of **<atr:freq>**		single
.plt	Filter .plts	mem	Memory related malloc memcpy etc	mpiall	..MPI... MPID... PMPI...	ompmutex	OMP mutex	incEverything	Include whatever is not in the Filters	**Single:** set of single trace entries **atr: sing**	**No Frequency:** only presence of attribute entries matters **freq:-**	complete
		net	Network related	mpicol	MPI collectives	ompall	OMP all functions			**Double:** set of 2-consecutive entries **atr: doub**	**Log10:** log(freq) of each entry matters (for large frequency numbers **freq: log10(#atr)**	average
		poll	Poll Related poll, yield	mpisr	MPI send/recv						**Actual:** actual frequency of each entry matters **freq: #atr**	weighted
		str	String related stcpy strcmp etc									centroid
												median
												ward

Figure 7. Filters, Attributes and other Parameters used to pre-process ParLOT Traces (PTs)

```
 // Return data structure to store champion
}

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

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

Table **??** describes the bug that I injected to ILCS-TSP

Table XI
INJECTED BUGS TO ILCS-TSP

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

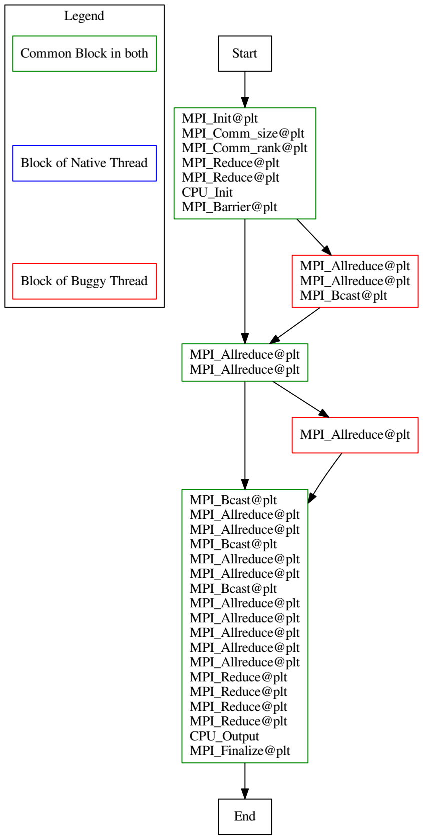


Figure 8. Bug1: diffNLR $PT_{0,0}$ - buggy vs. native

1) Bug1: Wrong Operation in MPI AllReduce(): We have injected a bug (row 1 table XI) where `MPI_Allreduce()` had been invoked with a wrong operation in one of the processes (P_2)(`MPI_MAX` instead of `MPI_MIN`).

::What is the runtime reaction to this bug:: Program terminated well without any error, crash, hang or throwing any exception. But the results might be corrupted. This might be a silent bug that diffTrace could reveal

The last row of table XII is telling us that among all combinations of parameters (filters, attributes, etc.) PT 0 (ParLOT trace that belongs to thread 0 of process 0 got impacted the most after we inject the bug.

The target MPI_Allreduce() that we injected the bug to, finds the rank (i.e., process) that has the "champion" result among all of ranks using MPI_MIN operator. Then that champion rank copies its "champion results" to a global data-structure and broadcast the "champion results" to all other ranks for the next time step. However, since we changed the MPI_MIN to MPI_MAX in only one of the ranks, the true champion rank would get lost, instead a false champion rank (which turned to be rank 0 or $PT_{0,0}$) would broadcast its results as champion in the first time-step, causing a potential wrong answer.

Table XII
BUG 1: WRONG MPI OPERATION IN ALLREDUCE() CANDIDATE TABLE

Filter	Attribute	K: # of diff Clusters	# Objects in each Cluster (CL i)	Candidate PT Outliers
11.mpi.cust.0K10	sing.orig	2	CL 0:34	-
			CL 1:6	-
11.mpi.cust.0K10	sing.orig	3	CL 0:34	-
			CL 1:5	{2 3 4 27 29 }
			CL 2:1	{22 }
11.mpi.cust.0K10	sing.orig	4	CL 0:3	{24 38 39 }
			CL 1:31	-
			CL 2:5	{2 3 4 27 29 }
			CL 3:1	{22 }
11.mpi.cust.0K10	sing.log10	2	CL 0:34	-
			CL 1:6	-
11.mpi.cust.0K10	sing.log10	3	CL 0:34	-
			CL 1:5	{2 3 4 27 29 }
			CL 2:1	{22 }
11.mpi.cust.0K10	sing.log10	4	CL 0:3	{24 38 39 }
			CL 1:31	-
			CL 2:5	{2 3 4 27 29 }
			CL 3:1	{22 }
11.mpi.cust.0K10	sing.actual	2	CL 0:34	-
			CL 1:6	-
11.mpi.cust.0K10	sing.actual	3	CL 0:34	-
			CL 1:5	{2 3 4 27 29 }
			CL 2:1	{22 }
11.mpi.cust.0K10	sing.actual	4	CL 0:3	{24 38 39 }
			CL 1:31	-
			CL 2:5	{2 3 4 27 29 }
			CL 3:1	{22 }
11.mpi.cust.0K10	doub.orig	2	CL 0:39	-
			CL 1:1	**{0}**
11.mpi.cust.0K10	doub.orig	3	CL 0:32	-
			CL 1:7	-
			CL 2:1	**{0}**
11.mpi.cust.0K10	doub.orig	4	CL 0:32	-
			CL 1:7	-
			CL 2:1	**{0}**
11.mpi.cust.0K10	doub.log10	2	CL 0:39	-
			CL 1:1	**{0}**
11.mpi.cust.0K10	doub.log10	3	CL 0:32	-
			CL 1:7	-
			CL 2:1	**{0}**
11.mpi.cust.0K10	doub.log10	4	CL 0:32	-
			CL 1:7	-
			CL 2:1	**{0}**
11.mpi.cust.0K10	doub.actual	2	CL 0:39	-
			CL 1:1	**{0}**
11.mpi.cust.0K10	doub.actual	3	CL 0:32	-
			CL 1:7	-
			CL 2:1	**{0}**
11.mpi.cust.0K10	doub.actual	4	CL 0:32	-
			CL 1:7	-
			CL 2:1	**{0}**
		TOP Suspicious Traces to check	**1-0** **2-2** **3-3**	-

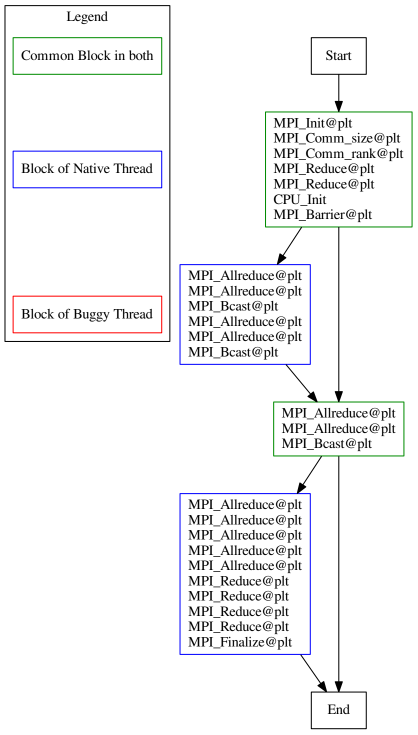


Figure 9. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

2) Bug2: Wrong Size in MPI AllReduce() (one process): We have injected a bug (row 2 table XI) where `MPI_Allreduce()` had been invoked with a wrong size. **::MORE EXPLANATIONS ABOUT WHAT THE BUG IS::**

Similar to table XII, the same ranking system tells us to check $PT_{1,0}$ and $PT_{3,0}$. Note that the bug injected to only one process (P_3) to have the wrong size.)

::EXPLANATIONS OF OBSERVATIONS::

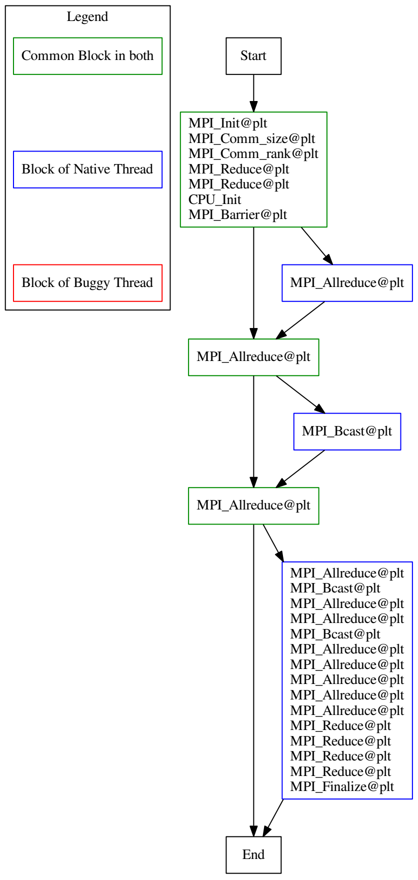


Figure 10. Bug2: diffNLR $PT_{1,0}$ - buggy vs. native

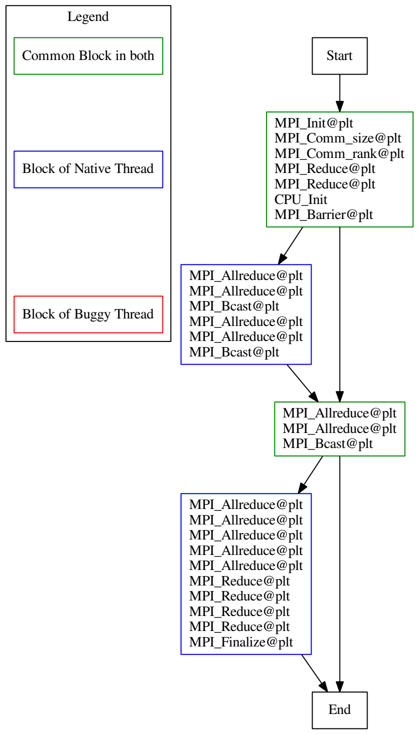


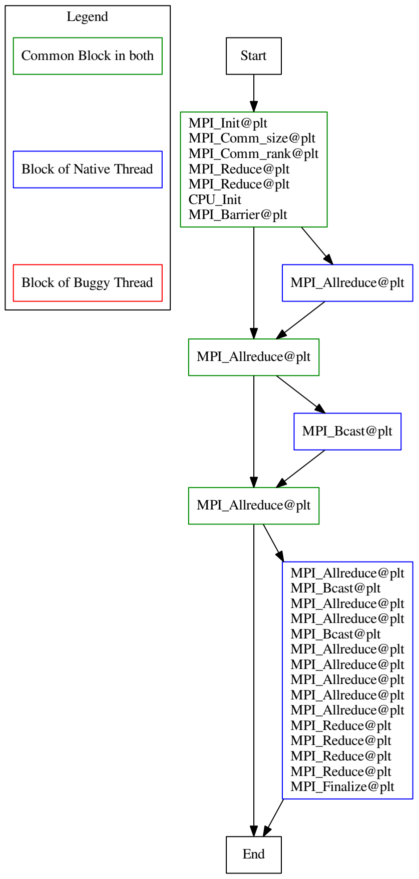
Figure 11. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

3) Bug3: Wrong Size in MPI AllReduce() (all processes): We have injected a bug (row 3 table XI) where `MPI_Allreduce()` had been invoked with a wrong size. **::MORE EXPLANATIONS ABOUT WHAT THE BUG IS::**

::What is the runtime reaction to this bug:: on node 3 (rank 3 in comm 0): Fatal error in PMPI_Bcast: Invalid root

Similar to table XII, the same ranking system tells us to check $PT_{1,0}$ and $PT_{3,0}$. Note that the bug injected to only one process (P_3) to have the wrong size.)

::EXPLANATIONS OF OBSERVATIONS::



4) Bug4: Wrong Op in MPI AllReduce(): no effect!,program terminates fine: maybe all images show some reflection

5) Bug5: Wrong Size in next MPI AllReduce()(one process)::no effect, program terminates fine: maybe all images show some reflection

6) Bug6: Wrong Size in next MPI AllReduce()(all processes)::no effect,program terminates fine: maybe all images show some reflection

7) Bug7: Wrong Size in MPI Bcast()(one process)::: maybe all images show some reflection

8) Bug8: Wrong Size in next MPI Bcast()(all processes)::no effect,program terminates fine: maybe all images show some reflection

9) 3: Missing Critical Section one thread in on process: I planted the bug (missing critical section) in process 2

VI. Related Work

A. Program Understanding

- Score-P [15]
- TAU [16]
- ScalaTrace: Scalable compression and replay of communication traces for HPC [17]
- 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]
- D4: Fast concurrency debugging with parallel differntial analysis - bozhen liu [24]
- Marmot: An MPI analysis and checking tool - bettina krammer [25]
- MPI-checker - Static Analysis for MPI - Alexandrer droste [26]
- STAT: stack trace analysis for large scale debugging - Dorian Arnold [27]
- DMTracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements [28]
- SyncChecker: Detecting synchronization errors between MPI applications and libraries - [29]
- Model Based fault localization in large-scale computing systems - Naoya Maruyama [30]
- Synoptic: Studying logged behavior with inferred models - ivan beschastnikh [31]
- Mining temporal invariants from partially ordered logs - ivan beschastnikh [32]
- Scalable Temporal Order Analysis for Large Scale Debugging - Dong Ahn [33]
- Inferring and asserting distributed system invariants - ivan beschastnikh - stewart grant [34]
- PRODOMETER: Accurate application progress analysis for large-scale parallel debugging - subatra mitra [35]
- Automaded : Automata-based debugging for dissimilar parallel tasks - greg [36]
- Automaded : large scale debugging of parallel tasks with Automaded - ignacio [37]
- Inferring models of concurrent systems from logs of their behavior with CSight - ivan [38]

B. 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]

C. 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]

D. 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] [10]

E. Repetitive Patterns

- [47] [5] [4] [48] [49]

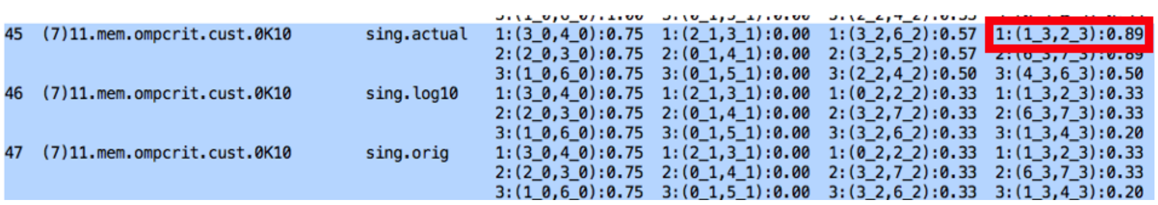
F. STAT

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



```
                                                     3:(1_0,6_0):1.00  3:(0_1,5_1):0.00  3:(2_2,4_2):0.33
45  (7)11.mem.ompcrit.cust.0K10         sing.actual  1:(3_0,4_0):0.75  1:(2_1,3_1):0.00  1:(3_2,6_2):0.57  1:(1_3,2_3):0.89
                                                     2:(2_0,3_0):0.75  2:(0_1,4_1):0.00  2:(3_2,5_2):0.57  2:(6_3,7_3):0.89
                                                     3:(1_0,6_0):0.75  3:(0_1,5_1):0.00  3:(2_2,4_2):0.50  3:(4_3,6_3):0.50
46  (7)11.mem.ompcrit.cust.0K10         sing.log10   1:(3_0,4_0):0.75  1:(2_1,3_1):0.00  1:(0_2,2_2):0.33  1:(1_3,2_3):0.33
                                                     2:(2_0,3_0):0.75  2:(0_1,4_1):0.00  2:(3_2,7_2):0.33  2:(6_3,7_3):0.33
                                                     3:(1_0,6_0):0.75  3:(0_1,5_1):0.00  3:(3_2,6_2):0.33  3:(1_3,4_3):0.20
47  (7)11.mem.ompcrit.cust.0K10         sing.orig    1:(3_0,4_0):0.75  1:(2_1,3_1):0.00  1:(0_2,2_2):0.33  1:(1_3,2_3):0.33
                                                     2:(2_0,3_0):0.75  2:(0_1,4_1):0.00  2:(3_2,7_2):0.33  2:(6_3,7_3):0.33
                                                     3:(1_0,6_0):0.75  3:(0_1,5_1):0.00  3:(3_2,6_2):0.33  3:(1_3,4_3):0.20
```

Figure 12. Part of ranking table for MisCrit 1-1

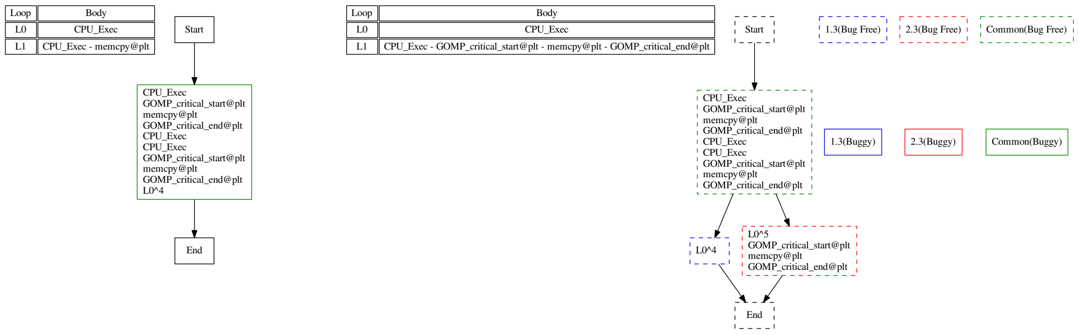


Figure 13. diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free

VII. Concluding Remarks

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