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

[[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 [**?**])
- 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. Background

The general idea is to utilize ParLOT [**?**] traces for studying HPC application behaviors towards fault detection and localization. ParLOT collects whole-program function calls and returns at different levels via dynamic binary instrumentation [**?**], and incrementally compresses them on-the-fly. Upon termination of the application, ParLOT flushes out per-thread trace files containing compressed sequence of executed function IDs. The compression mechanism of ParLOT significantly reduces the time, memory and disk overhead, leaves the majority of the system bandwidth for the application. With the mindset of "*pay a little upfront to dramatically reduce the number of overall debug iterations*", ParLOT well overcomes the challenge of whole-program *trace collection* and leaves the *trace analysis* for offline post-mortem analysis, saving HPC resources.

In this paper, we introduce DiffTrace, a tool-chain that provides an infrastructure for iterative and configurable search space reduction of the HPC whole-program function-call traces, and detection of the most impacted trace(s) and/or region of trace(s). Considering a "successful" termination of the application as *normal behavior*, DiffTrace takes steps towards *abnormal behavior* detection when an application crashes, time outs or produces a corrupted answer. Each abnormal behavior is a potential fault cause or a manifestation of the fault. However, faults in HPC applications may occur or influence the program behavior at different locations and granularities, due to high and hybrid level of parallelism. Also typical HPC applications spend most of their execution time in a main loop until a convergence or over timesteps, and a fault may get triggered or causing problems after some iteration. Thus accurate automatic fault localization is the problem of finding the needle in a haystack. Due to numerous and comprehensive whole-program function traces, a light-weight single pass of analysis has low chance to reveal interesting facts. A comprehensive heavy analysis also is often not feasible and unpractical. DiffTrace gives the HPC developers the capability of going through the pipeline of trace processing multiple times, each time putting a flash-light on various aspects of the applications' dynamic behavior, gradually collecting evidence about what has happened during execution. The journey starts

Figure 1. Simplified MPI implementation of Odd/Even Sort

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

Figure 2. A line change in oddEvenSort (left) that might cause a deadlock in oddEvenSort_DL (right)

	oddEvenSort()	oddEvenSort_DL()
1	`oddEvenSort(rank, cp){`	`oddEvenSort_DL(rank, cp){`
2	`...`	`...`
3	`for (int i=0; i < cp; i++)`	`for (int i=0; i < cp; i++)`
4	`{`	`{`
5	`int ptr = findPtr(i, rank);`	`int ptr = findPtr(i, rank);`
6	`...`	`...`
7	`if (rank % 2 == 0) {`	`if (rank % 2 == 0) {`
8	`MPI_Send(..., ptr, ...);`	`MPI_Send(..., ptr, ...);`
9	`MPI_Recv(..., ptr, ...);`	`MPI_Recv(..., ptr, ...);`
10	`} else {`	`} else {`
11	`MPI_Recv(..., ptr, ...);`	`MPI_Send(..., ptr, ...);`
12	`MPI_Send(..., ptr, ...);`	`MPI_Recv(..., ptr, ...);`
13	`}`	`}`
14	`...`	`...`
15	`}`	`}`
16	`}`	`}`

from decompressing ParLOT traces and pruning out uninteresting functions from traces. Loops in source codes reflect themselves as a sequence of *repetitive patterns*, resulting in often-long but redundant traces. A "nested loop recognition" mechanism then mines loops from traces as "*a measure of progress*" per thread, and also a loss-less abstraction to ease the rest of trace analysis. The control flow of events in parallel architectures often follow a *regular pattern* such as SPMD, odd-even and master/slave. This characteristic often makes all traces of a single execution tend to fall into just *a few* "conceptually/behaviorally equivalent classes". Adopted from the work by Webber et al [**?**], we have applied *formal concept analysis* (FCA)[**?**] techniques to reduce the trace search space into a few classes of traces, and also using the *concept lattice* (CL) data structure as the *model* of execution for further analysis. By comparison of the CL-based models of execution, a "suspicious candidate table" is generated for each set of parameters, pointing at traces that have "changed" the most after a fault is introduced. What has changed after a bug is encountered, is visualized by *gdiff*, a graphical representation of differences of a pair of sequences.

Section III explains details of DiffTrace components, but before that, we support our ideas over a simple example.

A. Introducing *gdiff*: Observing pair-wise differences

"Diff" algorithm by Meyers [**?**] takes two sequences S_A and S_B and computes the minimal *edit* to convert S_A to S_B. This algorithm has been used in GNU `diff` to compare two text

Table I

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

files and in git for efficiently keeping track of file changes. Since ParLOT preserves the order of function calls in the binary, each per thread trace T_i is totally ordered, thus *diff* can reflect the differences of a pair of Ts. *gdiff* is the graphical visualization of diff, aligning common and different blocks of a pair of sequences horizontally and vertically, making it easier for analyst to see the differences of a pair of sequences in a glance. For simplicity, our implementation of *gdiff* only takes one argument x as *the suspicious trace*

$$\textit{gdiff}(x) \equiv \textit{gdiff}(T_x, T'_x)$$

where T_x is the trace of thread/process x of a normal/successful execution and T'_x is the corresponding trace of faulty execution.

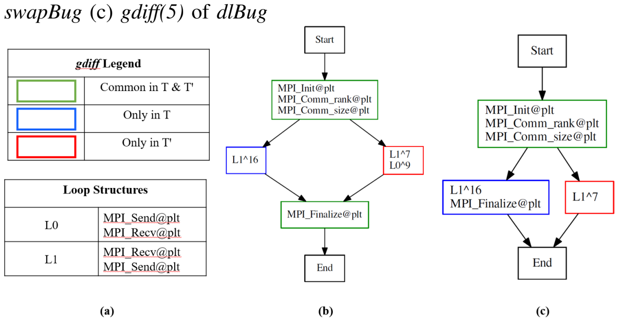
1) gdiff via example: Odd/Even sort is a variant of the bubble-sort operates in two alternate phases: *Phase-even* where even processes exchange (compare and swap) values with right neighbors and *Phase-odd* where odd processes exchange values with right neighbors. Figure 1 shows the simplified MPI implementation of the odd/even sort algorithm.

The for loop in line 4 of `oddEvenSort()` iterates over phases of the algorithm and based on the phase, the appropriate partner for each rank is getting discovered by the function `findPtr()` (line 6). The odd/even ranks then exchange their chunks of data (lines 9-13) and a set of sort, merge and copy operations would be performed on received data by each rank (which are replaced by `...` in line 15 for simplicity).

Execution of odd/even sort application with four processes (`mpirun -np 4`) while ParLOT trace collection is enabled on top of the application, would result in T_0, T_1, T_2 and T_3 (table I). This execution terminates successfully with expected results and the set of generated traces clearly reflects the expected behavior (control flow) of odd and even processes.

According to MPI Standard [cite MPI-forum or openMPI url], MPI_Send is a *blocking send* used the *standard* communication mode. In this mode, MPI may buffer outgoing messages and the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing

Figure 3. (a) The legend of *gdiff* and the list of loop structures (b) *gdiff(5)* of *swapBug* (c) *gdiff(5)* of *dlBug*



messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. This shows that, based on the MPI implementation, the `oddEvenSort_DL()` (figure 2) might end up causing a deadlock, because of the order swap of MPI_Recv and MPI_Send in lines 11-12.

We have planted two artificial bugs (*swapBug* and *dlBug*) in the code in figure 1 and launched the code with 16 processes. *swapBug* swaps the order of MPI_Send and MPI_Recv in rank 5 after 7th iteration (of the loop in line 3 of `oddEvenSort`) simulating a potential deadlock and *dlBug* simulates an actual deadlock (e.g., infinite loop) in the same location (rank 5 after 7th iteration). Upon collection of ParLOT traces from execution above buggy versions, DiffTrace first decompresses traces and filters out all non-MPI functions. Then two major loops are detected, **L0** and **L1** (figure 3-(a)) that are supposed to occur 16 times in even and odd ranks, respectively. After analysis of CL models of execution, $x = 5$ has been suggested as the most affected trace by the artificial bugs. Figure 3-(b) shows the *gdiff*(5) of *swapBug* where T_5 iterates over the loop [MPI_Recv - MPI_Send] for 16 times (L1ˆ16) after the MPI initilization while the order swap has well reflected in T'_x (L1ˆ7 - L0ˆ9). Both processes seem to be terminated fine by executing MPI_Finalize(). However, *gdiff*(5) of *dlBug* (figure 3-(c)) shows that while T_5 have executed MPI_Finalize and terminated well, T'_5 got stuck after executing L1 seven times and have never reached MPI_Finalize.

This example shows that our approach can locate the impacted part of each execution by a fault. Having a pre-understanding of *how the application should behave normally* would reduce the number of iterations by picking the right set of parameters on each pass. Relying on the knowledge of HPC developers, as the potential future users of DiffTrace, we believe our approach can post-mortem analyze of a "dead" execution and provides insight about "why the code died".

Table II
APPLICABLE FILTERS TO PT CONTENTS BASED ON REGULAR EXPRESSIONS

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

III. DIFFTRACE COMPONENTS

- 1-2 paragraph about the general overview of different components of DiffTrace
- major figure 4 showing DiffTrace components and the iterative approach

A. Trace Pre-processing

Figure 5 showing the overview of pre-processing chain (decompression, filter, nested loop recognition)

1) Decompression and Filter: As mentioned earlier, ParLOT incrementally compresses collected sequence of function calls and returns per-thread on-the-fly and store them in form of byte-codes on the disk. Each trace file contains a sequence of function IDs and an INFO file per process holds the corresponding function names.

maybe 1-2 sentences about how decompression works

Since ParLOT collects the *whole-program* function calls and does not ignore any function, traces might contain functions that we are not interested in studying them at the moment. On the other hand, any piece of information from traces might become handy in later phases of analysis. As an iterative approach, we have a set of pre-defined filters based on the regular expressions and string matching of function calls. On each iteration, we select one or more set of filters and if we get the desired results, we stop. Otherwise, we do our analysis using another, maybe more inclusive, set of filters to see what other information we might gain from traces. Table II shows the built-in filters. One can define custom filters based on the semantics of the application as well.

2) Loop Structures: [[WHOLE SUBSECTION NEEDS REWRITE]]

- 1 paragraph: Motivation of detecting loops in traces
- 1 paragraph: Loop definition and the original references of NLR algorithm
- 1-2 paragraph (or a figure): explaining our variation of NLR algorithm for detecting loop structures in ParLOT traces.
- Maybe an illustration on detecting odd/even sort loops

HPC applications and resources are of great interest to scientists and engineers for simulating *iterative* kernels. Computer simulation of fluid dynamics, partial differential equations, the Gauss-Seidel method, and finite element methods in form of stencil codes, all include a main outer loop that iterates over some elements (i.e., timesteps) and updates the elements. Loops in source codes would be reflected in traces as sequences of *repetitive patterns*. Mining these repetitive patterns in traces and replacing them with a compact loss-less representation would reveal the structure of sequence of function calls (or control flow). Also a fault in the code might cause a loop to break after some iterations. Thus "measuring progress" in each trace, as it is done as well in PRODOMETER and STAT, would become handy in later phases. To recognize loop structures in traces, we have adopted ideas from Kobayashi [**?**] paper where he defines loops in a sequence of instructions as "a string of instruction executions in which a particular sequence of distinct instructions (called the *cycle* of the loop) is successively repeated". This idea have been later expanded by Ketterline et al in [**?**] where they have introduced Nested Loop Recognition (NLR) algorithm for compressing data access addresses and predicting next accessing addresses. NLR is a memory-bounded algorithm that

start reading from the beginning of the sequence

store them in the stack

upon each push to stack

checks for top 3 equal size sub-sequence for isomorphism (equal length and equal corresponding elements)

checks if top n elements of the stack matches with any previous detected loops. if yes, increment the loop count and pop n elements from the stack

the above procedure would be repeated any time a change happens in the stack.

There is a pre-defined size for Max Stack. If stack reaches that point, a fixed number of elements would be popped from the bottom of the stack to free the space for rest of elements.

figure 6 shows the final product

complexity is $\Theta(K^2 N)$ where K is a fixed priori and N is the size of the input.

Table III
CONTEXT

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

B. Equivalencing Traces via FCA

- Construct the context table from example in figure 1
- Construct the Actual Concept Lattice from example in figure 1
- 1 paragraph background on FCA
- 1-2 paragraphs on advantages of FCA and what we would gain from FCA? Answer: Full pair-wise Jaccard Similarity Matrix (JSM)
- 1 paragraph how JSMs are going to help us (referring to the major figure at the beginning of this section)
  - Some background about Jaccard Similarity Score
  - How to obtain full pair-wise Jaccard Similarity Matrix (JSM) from a concept lattice (e.g., LCA approach)

Figure 4. DiffTrace Overview

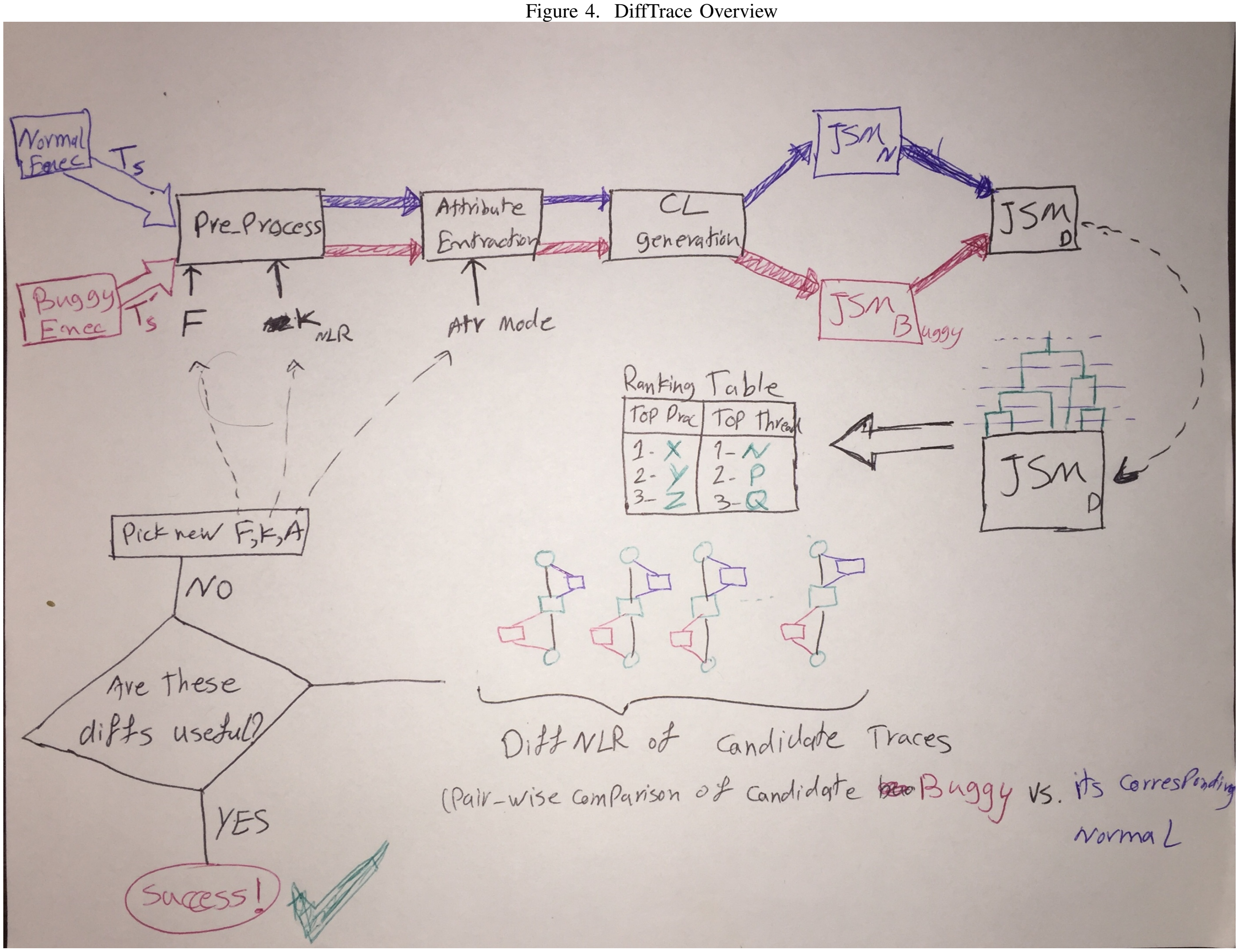


Figure 5. Pre-processing Components

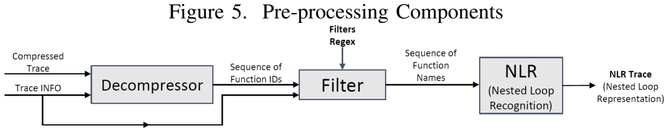


Figure 6. Sample NLR

PT = <a, b, c, b, c, b, c, d, e, b, c, b, c, b, c, d, e, f, g, h, g, h, x>

PT = <a, b, c, b, c, b, c, d, e, b, c, b, c, b, c, d, e, f, g, h, g, h, x>

NLR(PT) = < a, ((b, c)^3, d, e)^2, f, (g, h)^2, x>

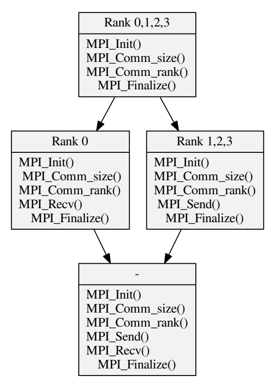


Figure 7. Sample Concept Lattice from Obj-Atr Context in tableIII-B

- 1-2 paragraphs on CL generation (related work and our approach)
  - Batch vs. Incremental [?]
  - Complexity: $O(2^{2K}||E||)$ where K is an upper bound for number of attributes (e.g., distinct function calls in the whole execution) and $||E||$ is the number of objects (e.g., number of PTs).
- 1-2 paragraphs (+ 1-2 figures) explaining the FCA ideas on odd/even sort example.

Thanks to ParLoT compression mechanism, we are able to efficiently (w.r.t. time and space) collect whole-program function call and return traces (PTs). However, post-mortem analysis of the PTs from thousands of threads requires decom-

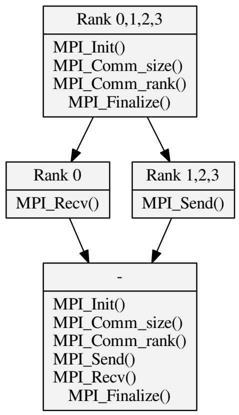


Figure 8. Concept Lattice with reduced labels

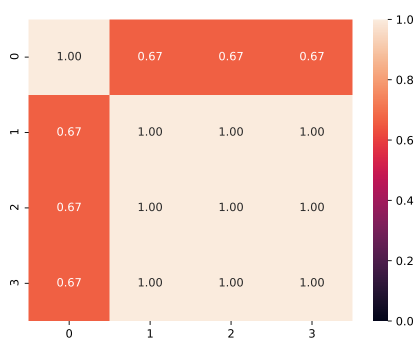


Figure 9. Pair-wise Jaccard Similarity Matrix (JSM) of MPI processes in Sample code

pression of traces, and consequently, analysis of large amount of data. Before jumping into *the huge haystack* of PTs to find *the tiny needle* (bug, bug manifestation or root cause of the failure), a middle ground data manipulation is required to simplify and organize the haystack.

Reducing the search space from thousands of PTs to just a few groups of equivalent PTs (i.e., inter-PT compression) not only requires a similarity measure based on a call matrix but also a scheme that is efficient even for large process counts. Since a pair-wise comparison of all processes is highly inefficient, we use *concept lattices* that stem from *formal concept analysis* (FCA) [**?**] to store and compute groups of similar PTs. FCA can efficiently split the large haystack into a few hay(semi)stacks with "conceptually" similar hays in each. This way conceptually isolated PTs (i.e., outliers) which are the potential bug manifestation or root cause would be detected. If no outlier detected, we only have a few distinct group of PTs to dig in, instead of thousands of large traces. With a wider perspective, here are other benefits of FCA for HPC debugging:

- FCA is scalable and efficient. It can be built incrementally and different kind of information such as full Jaccard Similarity Matrix (JSM) can be generated in linear time due to CL properties.
- Clustering is only one advantage of creating concept lattices from ParLoT traces. CLs can integrate all traces from an execution to a single entity as signature/model of good or bad execution for further analysis (e.g., prediction)
- Due to the *partial order* of nodes within CLs, valuable information can be retrieved from CLs like Happens-Before relation (Vijay Garg's book explains all applications of FCA in computer science applications)[**?**] and machine learning and data mining [**?**])

A concept lattice is based on a *formal context* [**?**], which is a triple (O, A, I), where O is a set of **objects**, A a set of **attributes**, and $I \subseteq O \times A$ an incidence relation. The incidence relation associates each object with a set of attributes (e.g., table III-B). Using FCA for clustering giving us the capability of clustering trace objects based on the "concept" of each trace object. We can characterize the "concept" (i.e., what we want to understand from the collected traces) by extracting meaningful "attributes" from traces. However, since we are only interested in grouping similar PTs in this work, we only take advantage of similarity measures [**?**] of concept lattices and leave other properties for future work. Due to typical HPC application topologies such as SPMD, master/worker and odd/even where multiple processes/threads behave similarly, our experiments show that large numbers of PTs can be reduced to just a few groups.

C. Suspicious Ranking Table via JSM diffing

2-3 paragraphs explaining JSM comparison and how it is going to give us top candidates as suspicious traces to check their gdiff

D. Parameters

2-3 paragraphs explaining the need of iterative approach to go through the tool chain multiple times, each time with different set of parameters (filters, attributes, NLR parameters) to gain insight about different aspects of the application execution (referring to the major overview figure).

IV. 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 [?], 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 **??** 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 **??** 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
 // 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

Filters: Prime		General		MPI		OMP		Other		CL Attributes		Clustering
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 10. Filters, Attributes and other Parameters used to pre-process ParLOT Traces (PTs)

Table IV
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

1) Bug1: Wrong Operation in MPI AllReduce(): We have injected a bug (row 1 table **??**) 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 **??** 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 V
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**	-

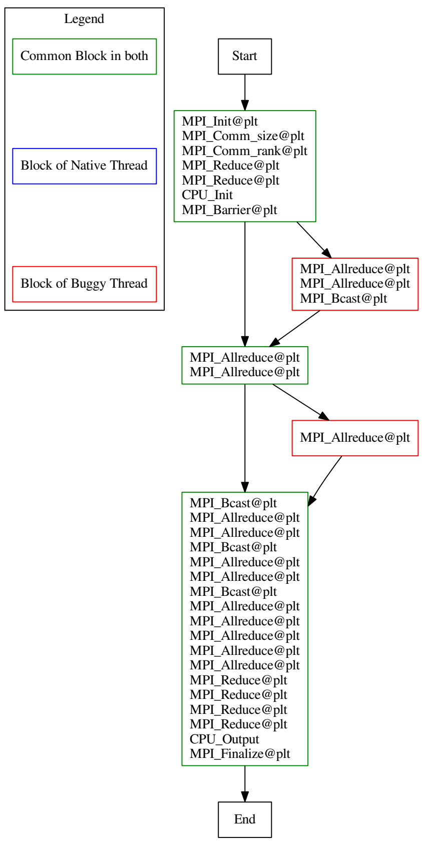


Figure 11. Bug1: diffNLR $PT_{0,0}$ - buggy vs. native

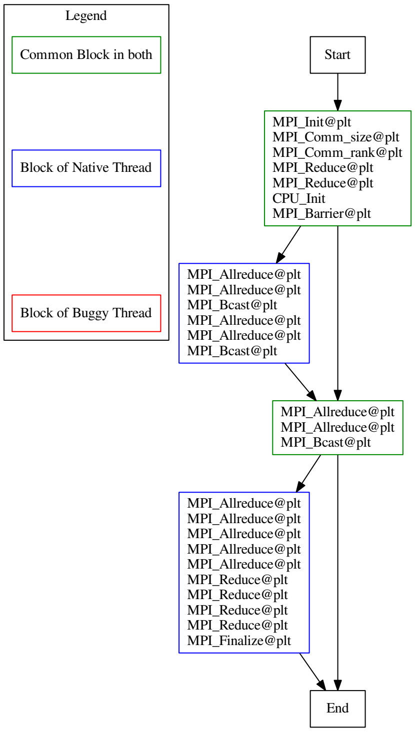


Figure 12. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

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

Similar to table **??**, 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::

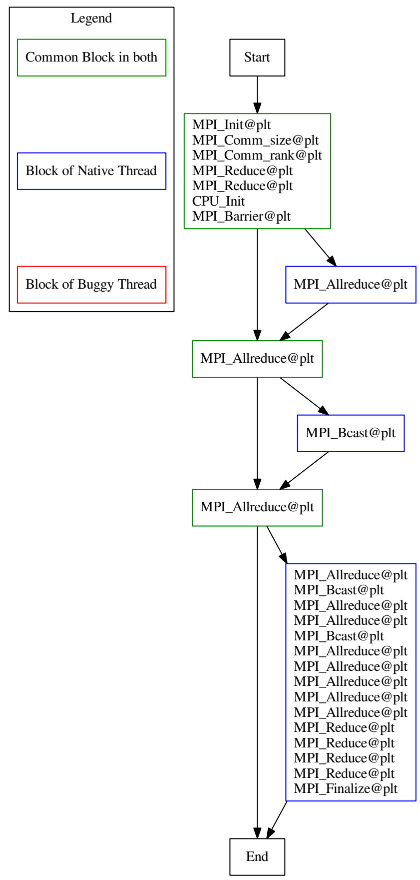


Figure 13. Bug2: diffNLR $PT_{1,0}$ - buggy vs. native

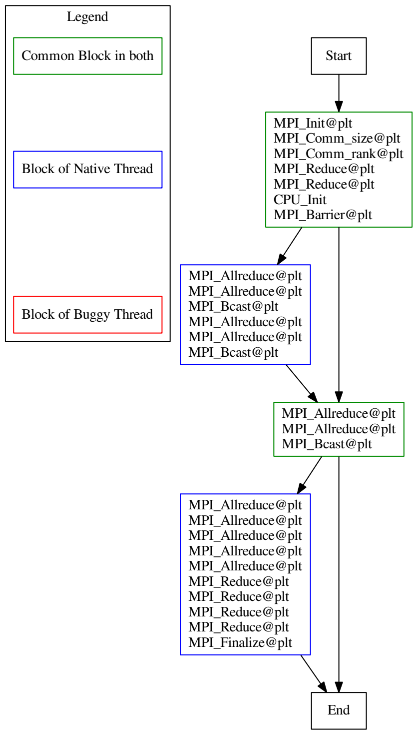


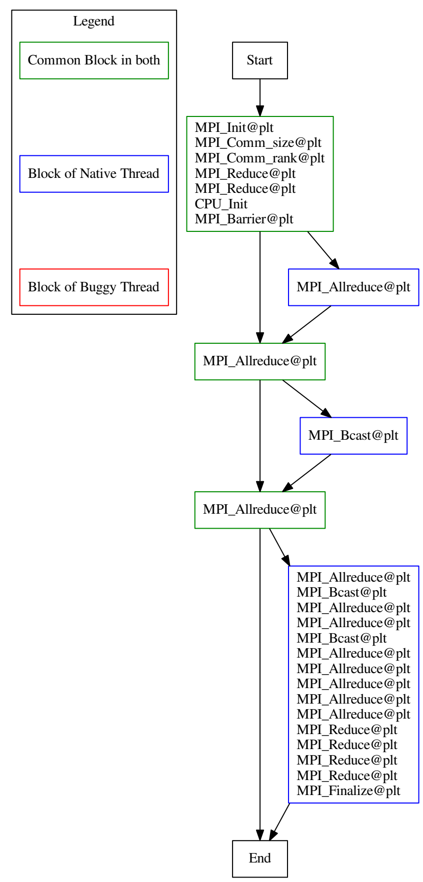
Figure 14. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

3) Bug3: Wrong Size in MPI AllReduce() (all processes): We have injected a bug (row 3 table **??**) 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 **??**, 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

V. Related Work

A. Program Understanding

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

B. Trace Analysis

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

C. Visualizations

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

D. Concept Lattice and LCA

- Vijay Garg - Applications of lattice theory in distributed systems
- Dimitry Ignatov [?] - Concept Lattice Applications in Information Retrieval
- [?] [?] [?] [?] [?]

E. Repetitive Patterns

- [?] [?] [?] [?] [?]

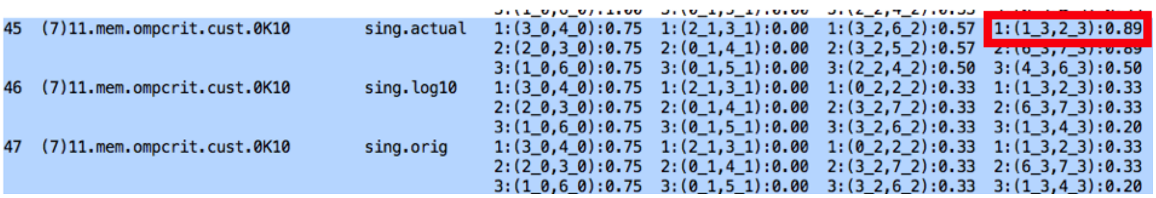
F. STAT

Parallel debugger STAT[?]

- 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



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

Figure 15. Part of ranking table for MisCrit 1-1

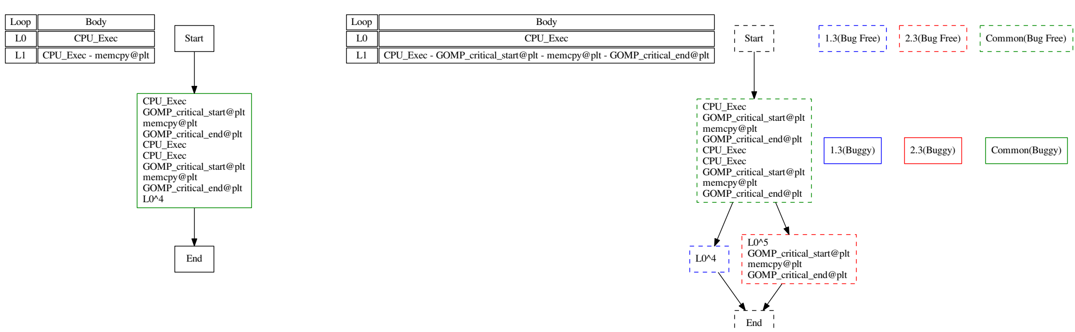


Figure 16. diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free

VI. 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] J.-D. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 210–220. [Online]. Available: http://doi.acm.org/10.1145/566172.566211

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

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

[8] 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.

[9] 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

[10] F. Alqadah and R. Bhatnagar, "Similarity measures in formal concept analysis," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 245–256, Mar 2011. [Online]. Available: https://doi.org/10.1007/s10472-011-9257-7

[11] 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.

[12] 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

[13] 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

[14] 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.

[15] 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

[16] 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).

[17] 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

[18] 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

[19] 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

[20] 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.

[21] 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

[22] 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.

[23] 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

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

[25] 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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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.

[36] 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

[37] 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

[38] 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.

[39] 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/

[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] 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

[47] 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

[48] 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

[49] M. Crochemore and W. Rytter, *Jewels of Stringology*. World Scientific, 2002. [Online]. Available: https://books.google.com/books?id=ipuPQgAACAAJ

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

APPENDIX