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 [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: Experiments: Methodology and Results
- Sec 4: Related Work
- Sec 5: Concluding Remarks

II. Background

There are two major phases in any "Program Understanding" tool: *data collection* and *data analysis*.

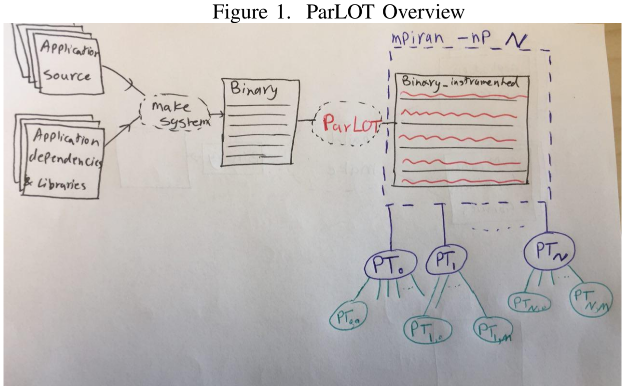
- *data collection:* To understand the runtime behavior of applications, an efficient tracing mechanism is required to collect informative data during execution of the application.
- *data analysis:* Upon failure or observing unexpected behavior of the program (e.g., a wrong answer), studying the collected execution data would reveal insight about how the program behaved dynamically and what went wrong.

In this section, we provide some background that explains our methodology for data collection and data analysis towards debugging and locating potential root causes of the unexpected behavior.

A. ParLOT: Efficient Trace Collection

The executable of HPC applications is often a combination of a large code base and a complex build system with numerous dependencies and libraries. Injecting instrumentation code to the source code is difficult in the HPC space. Also, recompilation of the application with compiler wrappers, as in TAU [9] and Score-p [10], may break the build system. The instrumentation and tracing mechanism of existing tools are often dependent on other libraries that are need to be present on the target system for trace collection. For example STAT [11] and AutomaDeD [12] require Dyninst [13] for instrumentation, and MRNet[14] and TBON[15] for providing a distributed communication of tracing mechanism. To enable comprehensive data collection in combination with low time and space overhead, HPC program analysis tools often sacrifice one for the other. However, ParLOT collects whole-program function call traces at as low as library level, while incrementally compressing traces on-the-fly and leaving the majority of the system bandwidth for the application. ParLOT collects *whole* program function call traces with the mindset of *paying a little upfront and saving resource and time cost of reproducing the bug later*. ParLOT instruments the entry and exit point of each function in the binary using Pin [16] (fig. 1). A unique id is assigned to each distinct function at runtime and every time the function is called (returned), its id is pushed (popped) from the compressed stack. In this fashion, each ParLOT trace contains highly compressed sequence of function calls and returns for every thread of the application code, reflecting the dynamic control flow and call stack. Formally, a ParLOT Trace (PT) is a sequence of ordered integers $< f_i, ..., f_j >$ where f_0 represents a *return* and f_k is the ID of *function k* $(k \neq 0)$. Note that $PT_{p.t}$ refers to the PT that belongs to process p and thread t of that process. Enabling ParLOT on top of execution of HPC applications would generate PTs for every running thread, enabling analyzers to dig into dynamic behavior of application.

Figure 1. ParLOT Overview



B. Loop structure detection

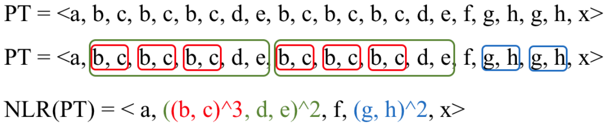
HPC applications and resources are of great interest to scientists and engineers for simulating *iterative* kernels. Computer simulation of fluid dynamics, partial differential equations, the Gauss-Seidel method, and finite element methods in form of stencil codes, all include a main outer loop that iterates over some elements (i.e., timesteps) and updates the elements. This character of typical HPC applications makes PTs very long (often millions or billions of entries) with a relatively small number (hundreds to thousands) of distinct elements (i.e., function IDs). We propose a representation of PT elements (intra-PT compression) in form of *loop structures*, such that PT = sequence of repetitive patterns (i.e., loops). In other words, each PT is a sequence of *Loop Bodies (LB)* that repeated *Loop Count (LC)* times, consecutively.

According to Makoto Kobayashi's [17] definition of loops, an occurrence of a *loop* is defined as *a sequence of elements* in which a particular sequence of *distinct elements* (called the *cycle* of the loop) is successively repeated. Later, Alain Ketterlin [3] expanded this definition to numerical values for compressing and predicting memory access addresses and designed the Nested Loop Recognition (NLR) algorithm. The basic idea behind the NLR algorithm is that a linear function model can be extracted from the linear progression in a sequence of numbers, and these linear functions form a tree in which the depth of each node is the depth of a *nested* loop (the outermost loop's function is the root of the tree with depth 0). We have modified the NLR algorithm to make it suitable for PTs. Each repetitive pattern and its frequency of consecutive appearances would be compressed to a single *Loop Structure (LS)* entry.

Definition 1: Loop Structure $LS = LB\hat{\ }LC$ where $LB =< pt_i, ..., pt_j >$ $(0 <= i < j < len(PT))$ that occur LC times is in a sub-sequence $< pt_i, ..., pt_k >$ $(k = 3j, k < len(PT))$ By converting each PT into a sequence of LS_i, we reduce the length of PT by a factor of $\sum_i len(LB_i) * LC_i$.

As an example, figure 2 shows how the sample PT with length 23 can be reduced to 5 after detecting its loop structures. In addition to reducing the size of easy-to-read PTs, NLR can also help with detecting broken loop structures due to a bug. For example, when a node fails to send or receive messages

Figure 2. 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>

in a supercomputer after some number of iterations, the PTs that belong to that node would show loop structures with loop counts lower than what was expected. In other word, NLR is enabling us to "measure progress" of each PT and detecting the cause of divergence in the control flow that leads to failure. Later we will explain how this lossless representation of PTs eases the process of diffing between a pair of PTs.

C. Equivalencing behavior via FCA

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 decompression 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) [4] 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)[8] and machine learning and data mining [7])

Table I
CONTEXT

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

A concept lattice is based on a *formal context* [4], 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 II-C2). 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 [5] 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.

1) Concept Lattice Construction:

- Batch vs. Incremental [18]
- 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).

2) FCA example: Executing the sample code below would result in PTs with the context in table II-C2.

.

```
main(){
 int rank;
 int src;
 MPI_Init()
 MPI_Comm_size(MPI_COMM_WORLD)
 MPI_Comm_rank(MPI_COMM_WORLD,&rank)
 if (rank != 0) {
  MPI_Send(0) // Send to rank 0
 } else{ /* rank = 0
  MPI_Recv(1) // Receive from rank 1
  MPI_Recv(2) // Receive from rank 2
  MPI_Recv(3) // Receive from rank 3
 }
 MPI_Finalize()
}
```

3) Jaccard Similarity Scores:

- Some background about Jaccard Similarity Score
- How to obtain full pair-wise Jaccard Similarity Matrix (JSM) from a concept lattice (e.g., LCA approach)

D. diffNLR: Reflecting differences

- Inspired by `diff` original algorithm[20] that has bin used in Git and GNU Diff, we visualize the differences of a pair of PT as shown in fig 6.

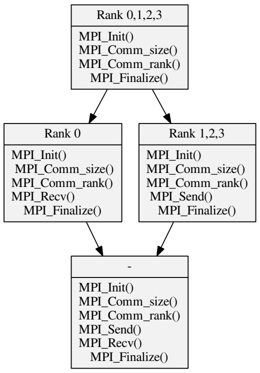


Figure 3. Sample Concept Lattice from Obj-Atr Context in tableII-C2

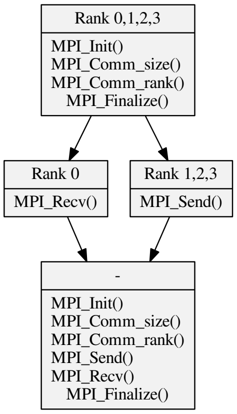


Figure 4. Concept Lattice with reduced labels

- This visualization reflects of the differences of **occurrences** of PT elements and their **orders**.
- In section III we show how this visualization can help us locating the points of divergence in PTs, and potential bug manifestation and root cause.

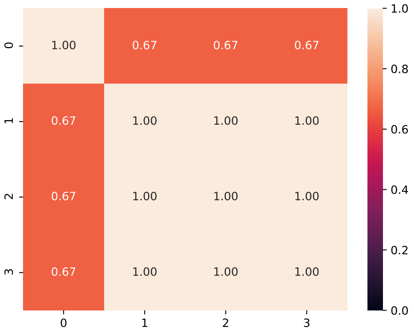


Figure 5. Pair-wise Jaccard Similarity Matrix (JSM) of MPI processes in Sample code

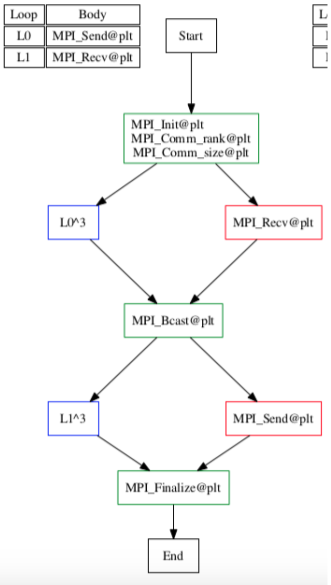


Figure 6. Sample diffNLR

III. Experiments: Methodology and Results

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 [52], 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 III 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.

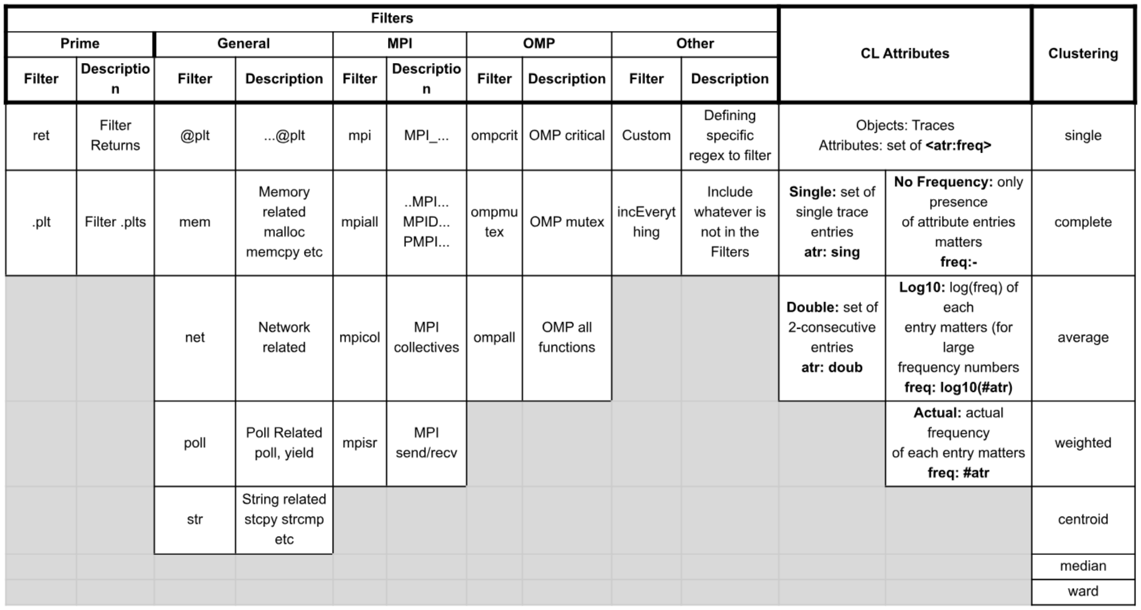
B. Case Study: ILCS-TSP

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 II
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

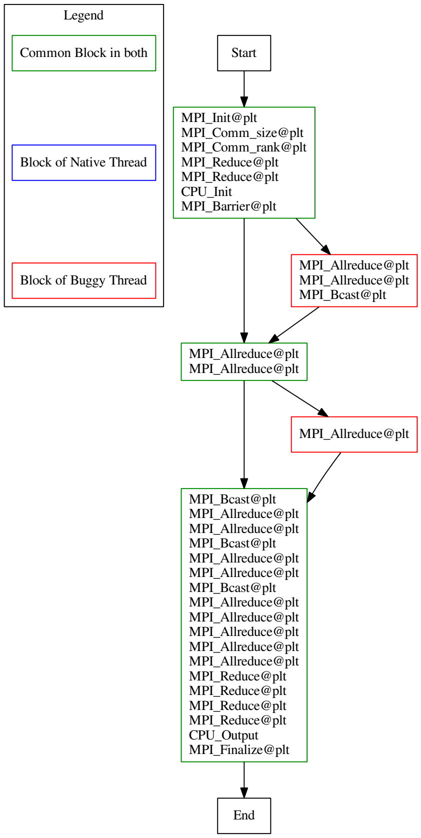


Figure 8. Bug1: diffNLR $PT_{0,0}$ - buggy vs. native

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

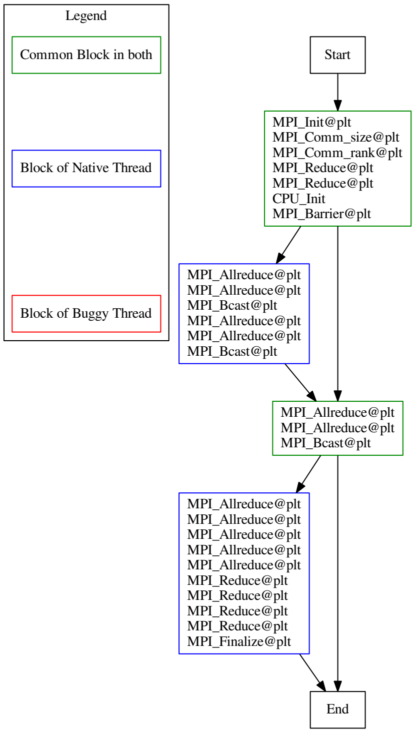


Figure 9. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

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

Similar to table III, 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::

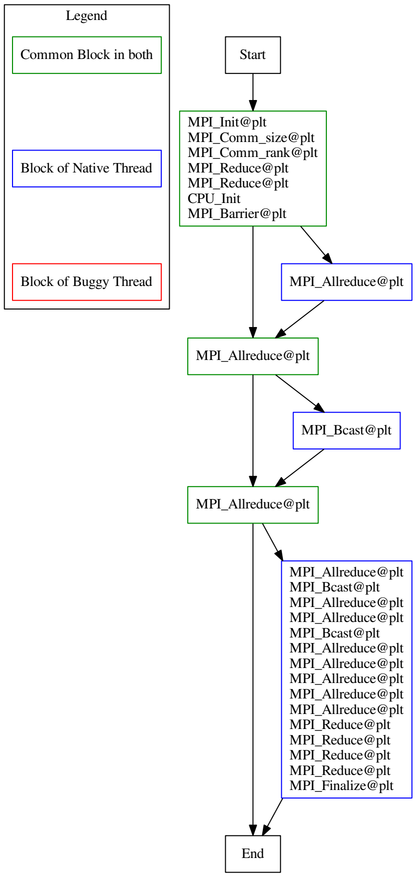


Figure 10. Bug2: diffNLR $PT_{1,0}$ - buggy vs. native

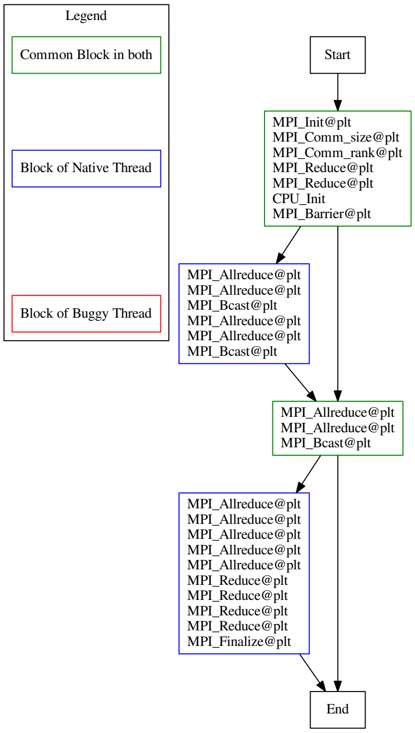


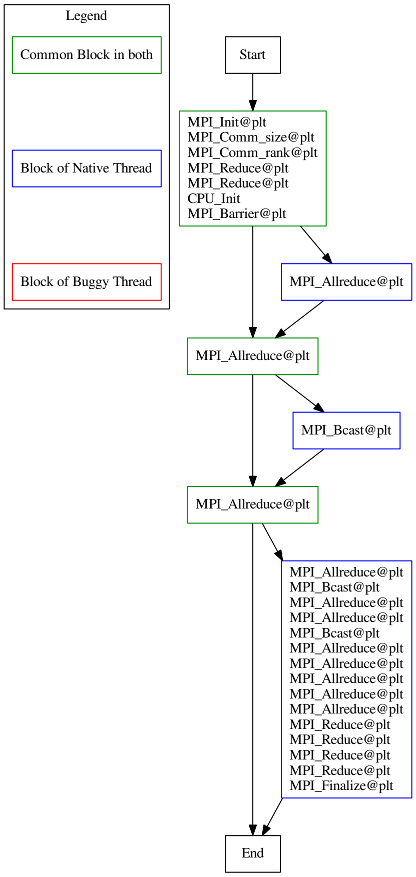
Figure 11. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

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

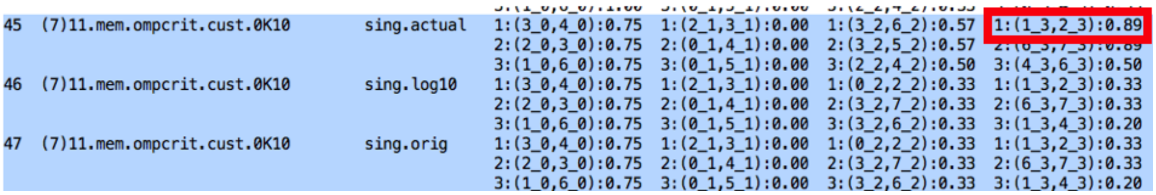
5) Bug5: Wrong Size in next MPI AllReduce()(one process)::no effect, program terminates fine: maybe all images show some reflection

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

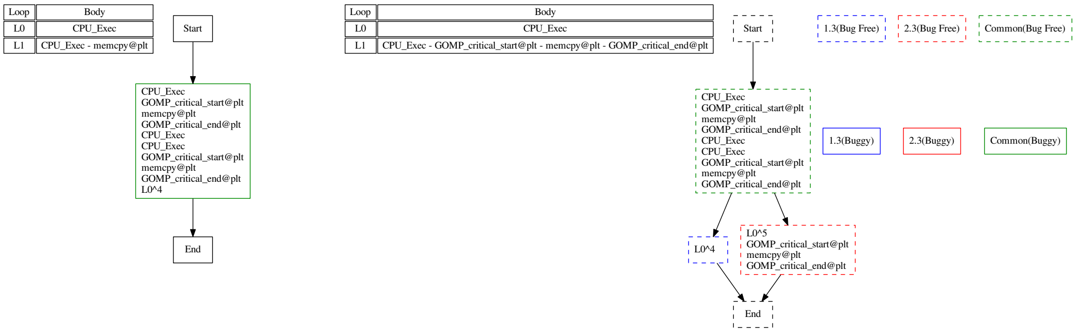
8) Bug8: Wrong Size in next MPI Bcast()(all processes)::no effect,program terminates fine: maybe all images show some reflection

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



```
                                                       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



Loop	Body
L0	CPU_Exec
L1	CPU_Exec - memcpy@plt

Loop	Body
L0	CPU_Exec
L1	CPU_Exec - GOMP_critical_start@plt - memcpy@plt - GOMP_critical_end@plt

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

IV. Related Work

A. Program Understanding

- Score-P [10]
- TAU [9]
- ScalaTrace: Scalable compression and replay of communication traces for HPC [21]
- Barrier Matching for Programs with Textually unaligned barriers [22]
- Pivot Tracing: Dynamic causal monitoring for distributed systems - Johnathan mace [23]
- Automated Charecterization of parallel application communication patterns [24]
- Problem Diagnosis in Large Scale Computing environments [25]
- Probablistic diagnosis of performance faults in large-scale parallel applications [26]
- detecting patterns in MPI communication traces - robert preissl [27]
- D4: Fast concurrency debugging with parallel differntial analysis - bozhen liu [28]
- Marmot: An MPI analysis and checking tool - bettina krammer [29]
- MPI-checker - Static Analysis for MPI - Alexandrer droste [30]
- STAT: stack trace analysis for large scale debugging - Dorian Arnold [11]
- DMTracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements [31]
- SyncChecker: Detecting synchronization errors between MPI applications and libraries - [32]
- Model Based fault localization in large-scale computing systems - Naoya Maruyama [33]
- Synoptic: Studying logged behavior with inferred models - ivan beschastnikh [34]
- Mining temporal invariants from partially ordered logs - ivan beschastnikh [35]
- Scalable Temporal Order Analysis for Large Scale Debugging - Dong Ahn [36]
- Inferring and asserting distributed system invariants - ivan beschastnikh - stewart grant [37]
- PRODOMETER: Accurate application progress analysis for large-scale parallel debugging - subatra mitra [38]
- Automaded : Automata-based debugging for dissimilar parallel tasks - greg [39]
- Automaded : large scale debugging of parallel tasks with Automaded - ignacio [12]
- Inferring models of concurrent systems from logs of their behavior with CSight - ivan [40]

B. Trace Analysis

- Trace File Comparison with a hierarchical Sequence Alignment algorithm [41]
- structural clustering : matthias weber [6]
- building a better backtrace: techniques for postmortem program analysis - ben liblit [42]
- automatically charecterizing large scale program behavior - timothy sherwood [43]

C. Visualizations

- Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time - katherine e isaacs [44]
- recovering logical structure from charm++ event traces [45]
- ShiViz - Debugging distributed systems - [46]

D. Concept Lattice and LCA

- Vijay Garg - Applications of lattice theory in distributed systems
- Dimitry Ignatov [**?**] - Concept Lattice Applications in Information Retrieval
- [4] [18] [19] [8] [20]

E. Repetitive Patterns

- [47] [48] [49] [50] [51]

F. STAT

Parallel debugger STAT[11]

- 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

V. 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 *7th Workshop on Extreme-Scale Programming Tools, ESPT@SC 2018, Dallas, TX, USA, November 16, 2018*.

[3] 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

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

[5] 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

[6] 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/

[7] 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

[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] 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

[10] 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.

[11] 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.

[12] 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

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

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

[15] T. Hilbrich, B. R. de Supinski, F. Hänsel, M. S. Müller, M. Schulz, and W. E. Nagel, "Runtime mpi collective checking with tree-based overlay networks," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: ACM, 2013, pp. 129–134. [Online]. Available: http://doi.acm.org/10.1145/2488551.2488570

[16] 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.

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

[18] 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.

[19] 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

[20] 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

[21] 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).

[22] 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

[23] 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

[24] 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

[25] 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.

[26] 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

[27] 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.

[28] 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

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

[30] 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

[31] 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.

[32] 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.

[33] 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.

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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.

[40] 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

[41] 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.

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

[43] 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

[44] 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.

[45] 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.

[46] 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

[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] 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

[49] 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

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

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

[52] 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

Appendix

A. Case Study: Hybrid (MPI+OMP) Matmul

Below is the JSM of bug free version of hybrid matmul applying different filters to understand its behavior.

B. Bug: AllReduce wrong op - proc 2

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

After collecting traces from bug-free version and this buggy version of ILCS-TSP, and after applying filters and detecting loops, my ranking system, with the highest possible score, recommends to look at figure 18 diffNLR. This diffNLR shows that in the bug free version, we are expecting that at least one of the processes (in this case process 6 to win the championship, copy the result of that process to broadcast_buffer and broadcast it to all other processes. However, in the buggy version, the champion never gets updated (never goes through OMP critical section).

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

C. Bug: AllReduce wrong size - one and all

This bug made the code crash with showing an error in MPI_Bcast(), although the bug was injected to MPI_AllReduce(). Figure 19 $CMPdiffNLR(P_{2.0}, P_{6.0})$ and 23 CMPdiffNLR($P_{1.0}, P_{6.0}$), where CMPdiffNLR(P_i, P_j) shows the comparison of

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

and

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

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

D. Bug: Missing Critical Section one thread in on process

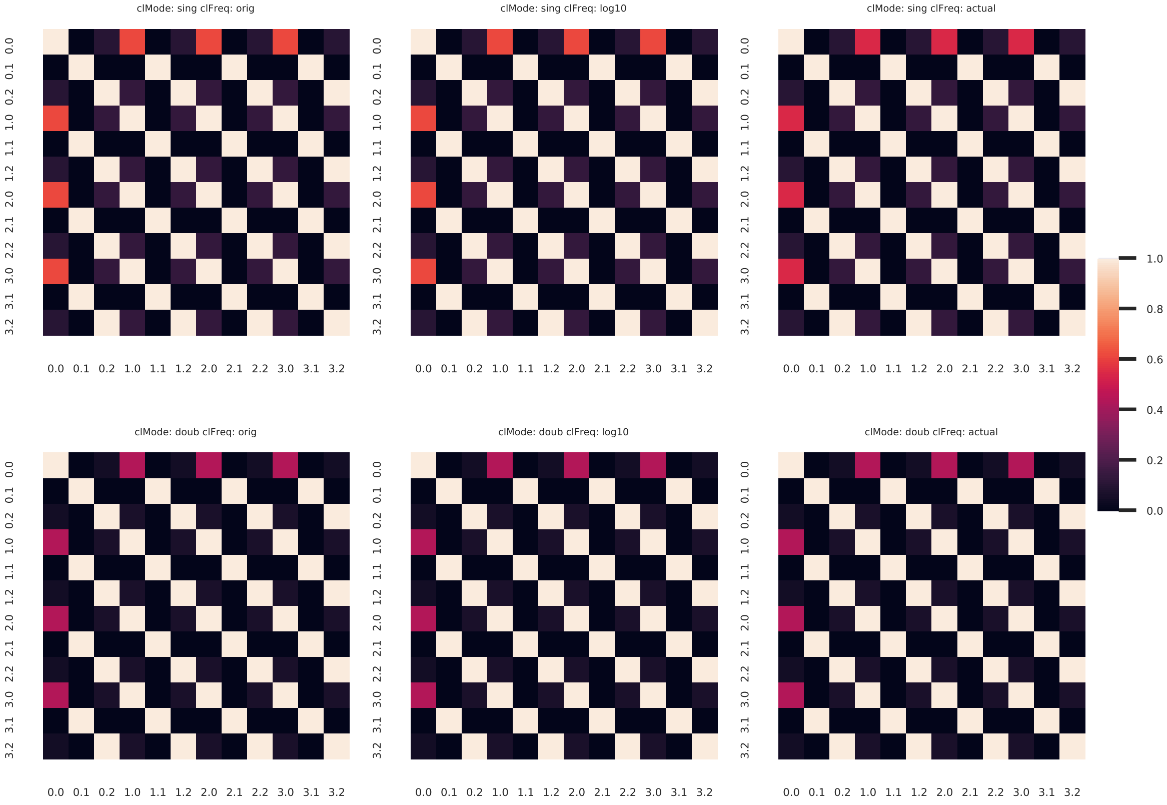


Figure 14. JSM of bug-free version of hybrid Matmul (all functions)

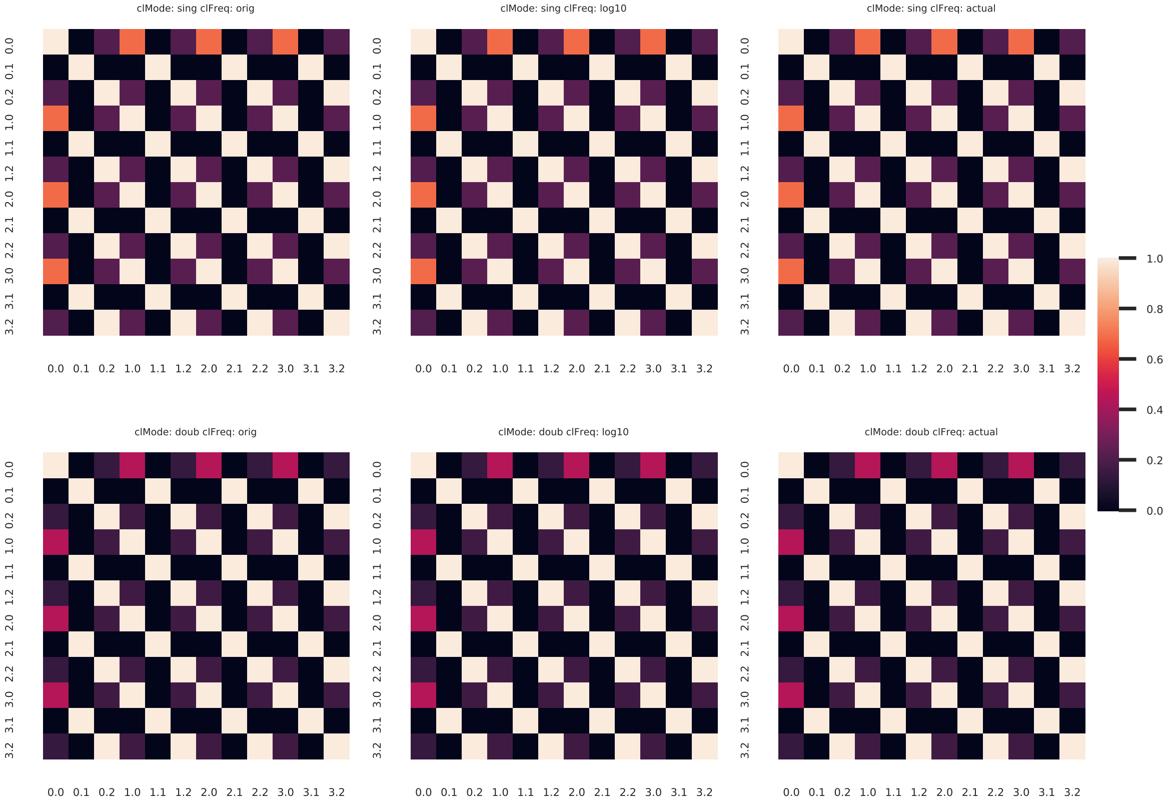


Figure 15. JSM of bug-free version of hybrid Matmul (MPI and OMP functions)

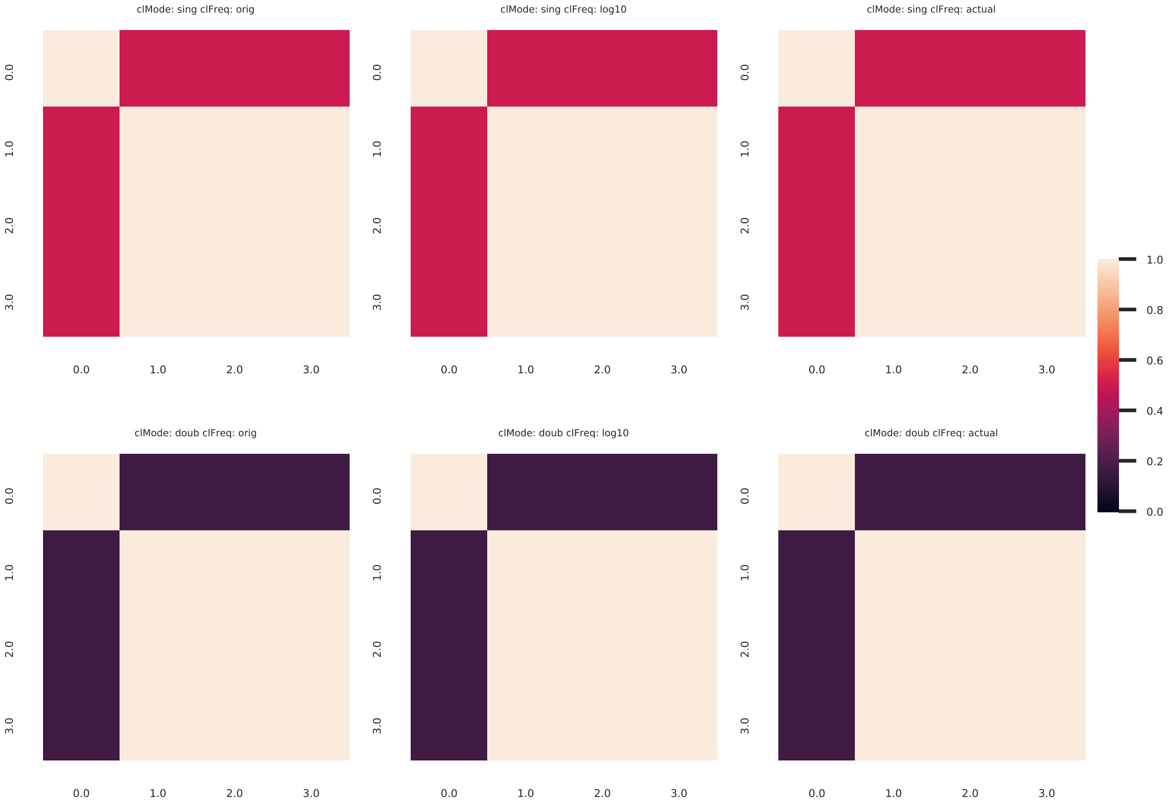
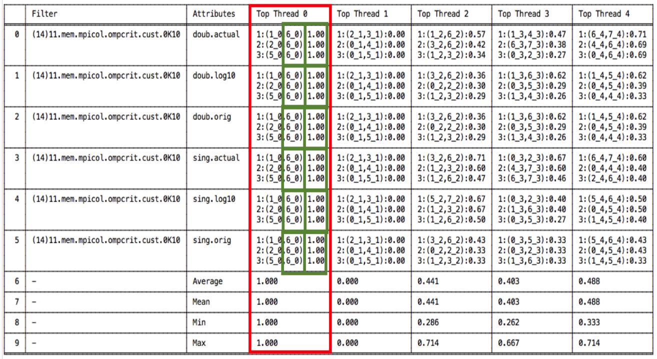
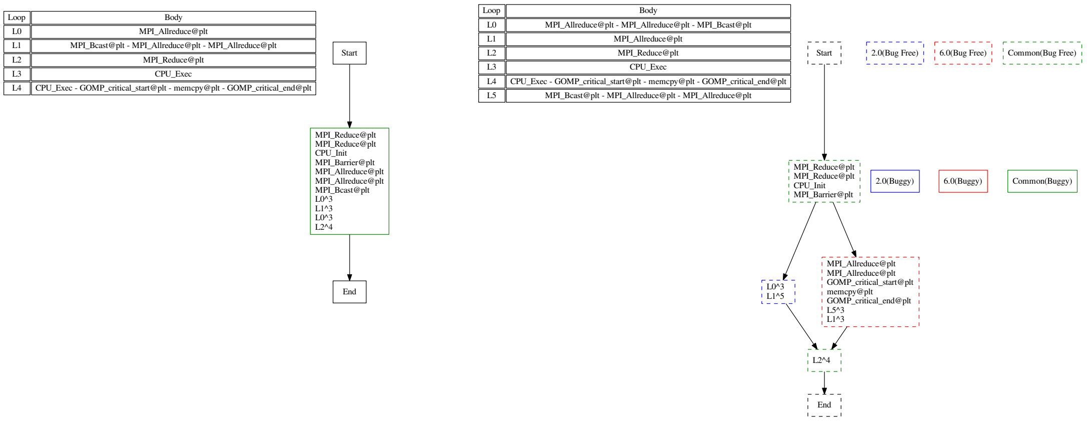


Figure 16. JSM of bug-free version of hybrid Matmul (MPI Only)



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

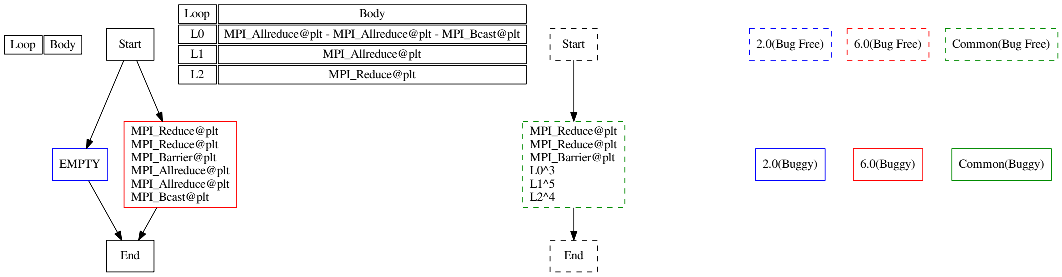
Figure 17. Recommendation Table Bug 1



Loop	Body
L0	MPI_Allreduce@plt
L1	MPI_Bcast@plt - MPI_Allreduce@plt - MPI_Allreduce@plt
L2	MPI_Reduce@plt
L3	CPU_Exec
L4	CPU_Exec - GOMP_critical_start@plt - memcpy@plt - GOMP_critical_end@plt

Loop	Body
L0	MPI_Allreduce@plt - MPI_Allreduce@plt - MPI_Bcast@plt
L1	MPI_Allreduce@plt
L2	MPI_Reduce@plt
L3	CPU_Exec
L4	CPU_Exec - GOMP_critical_start@plt - memcpy@plt - GOMP_critical_end@plt
L5	MPI_Bcast@plt - MPI_Allreduce@plt - MPI_Allreduce@plt

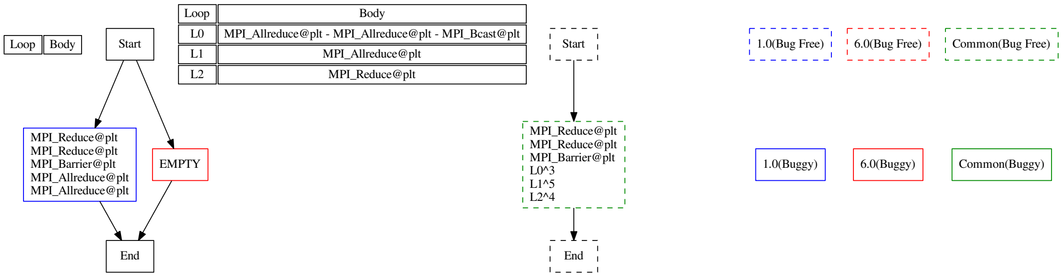
Figure 18. diffNLR of process 2 and process 6 buggy(AllReduce() wrong op) vs. bug-free



Loop	Body

Loop	Body
L0	MPI_Allreduce@plt - MPI_Allreduce@plt - MPI_Bcast@plt
L1	MPI_Allreduce@plt
L2	MPI_Reduce@plt

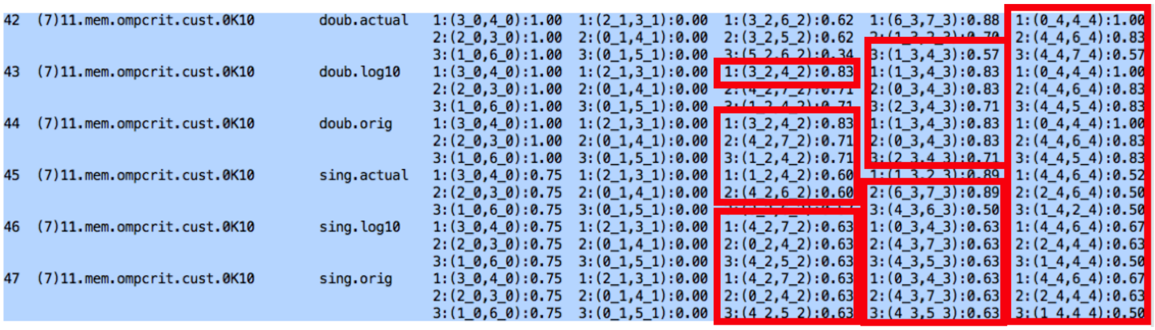
Figure 19. diffNLR of process 2 and process 6 buggy(AllReduce() wrong size) vs. bug-free



Loop	Body

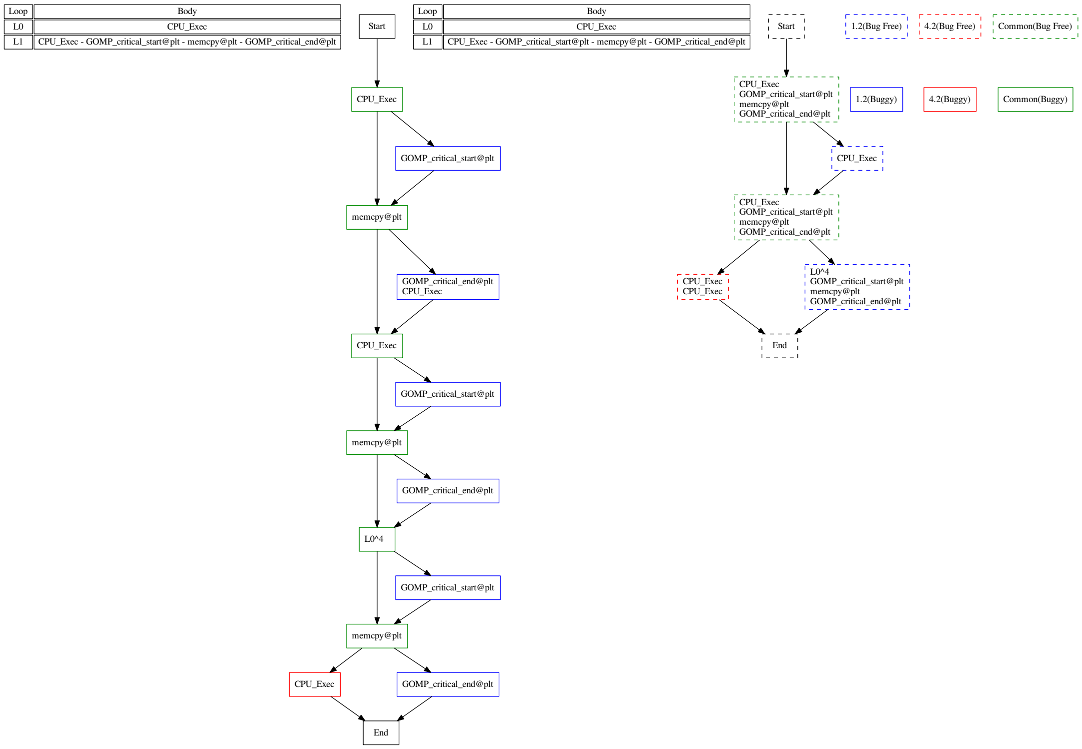
Loop	Body
L0	MPI_Allreduce@plt - MPI_Allreduce@plt - MPI_Bcast@plt
L1	MPI_Allreduce@plt
L2	MPI_Reduce@plt

Figure 20. diffNLR of process 1 and process 6 buggy(AllReduce() wrong size) vs. bug-free



```
42  (7)11.mem.ompcrit.cust.0K10      doub.actual  1:(3_0,4_0):1.00  1:(2_1,3_1):0.00  1:(3_2,6_2):0.62  1:(6_3,7_3):0.88  1:(0_4,4_4):1.00
                                                 2:(2_0,3_0):1.00  2:(0_1,4_1):0.00  2:(3_2,5_2):0.62  2:(1_3,2_3):0.70  2:(4_4,6_4):0.83
                                                 3:(1_0,6_0):1.00  3:(0_1,5_1):0.00  3:(5_2,6_2):0.34  3:(1_3,4_3):0.57  3:(4_4,7_4):0.57
43  (7)11.mem.ompcrit.cust.0K10      doub.log10   1:(3_0,4_0):1.00  1:(2_1,3_1):0.00  1:(3_2,4_2):0.83  1:(1_3,4_3):0.83  1:(0_4,4_4):1.00
                                                 2:(2_0,3_0):1.00  2:(0_1,4_1):0.00  2:(4_2,7_2):0.71  2:(0_3,4_3):0.83  2:(4_4,6_4):0.83
                                                 3:(1_0,6_0):1.00  3:(0_1,5_1):0.00  3:(1_2,4_2):0.71  3:(2_3,4_3):0.71  3:(4_4,5_4):0.83
44  (7)11.mem.ompcrit.cust.0K10      doub.orig    1:(3_0,4_0):1.00  1:(2_1,3_1):0.00  1:(3_2,4_2):0.83  1:(1_3,4_3):0.83  1:(0_4,4_4):1.00
                                                 2:(2_0,3_0):1.00  2:(0_1,4_1):0.00  2:(4_2,7_2):0.71  2:(0_3,4_3):0.83  2:(4_4,6_4):0.83
                                                 3:(1_0,6_0):1.00  3:(0_1,5_1):0.00  3:(1_2,4_2):0.71  3:(2_3,4_3):0.71  3:(4_4,5_4):0.83
45  (7)11.mem.ompcrit.cust.0K10      sing.actual  1:(3_0,4_0):0.75  1:(2_1,3_1):0.00  1:(1_2,4_2):0.60  1:(1_3,2_3):0.89  1:(4_4,6_4):0.52
                                                 2:(2_0,3_0):0.75  2:(0_1,4_1):0.00  2:(4_2,6_2):0.60  2:(6_3,7_3):0.89  2:(2_4,6_4):0.50
                                                 3:(1_0,6_0):0.75  3:(0_1,5_1):0.00  [illegible]       3:(4_3,6_3):0.50  3:(1_4,2_4):0.50
46  (7)11.mem.ompcrit.cust.0K10      sing.log10   1:(3_0,4_0):0.75  1:(2_1,3_1):0.00  1:(4_2,7_2):0.63  1:(0_3,4_3):0.63  1:(4_4,6_4):0.67
                                                 2:(2_0,3_0):0.75  2:(0_1,4_1):0.00  2:(0_2,4_2):0.63  2:(4_3,7_3):0.63  2:(2_4,4_4):0.63
                                                 3:(1_0,6_0):0.75  3:(0_1,5_1):0.00  3:(4_2,5_2):0.63  3:(4_3,5_3):0.63  3:(1_4,4_4):0.50
47  (7)11.mem.ompcrit.cust.0K10      sing.orig    1:(3_0,4_0):0.75  1:(2_1,3_1):0.00  1:(4_2,7_2):0.63  1:(0_3,4_3):0.63  1:(4_4,6_4):0.67
                                                 2:(2_0,3_0):0.75  2:(0_1,4_1):0.00  2:(0_2,4_2):0.63  2:(4_3,7_3):0.63  2:(2_4,4_4):0.63
                                                 3:(1_0,6_0):0.75  3:(0_1,5_1):0.00  3:(4_2,5_2):0.63  3:(4_3,5_3):0.63  3:(1_4,4_4):0.50
```

Figure 21. Part of ranking table for MisCrit 1-all



Loop	Body
L0	CPU_Exec
L1	CPU_Exec - GOMP_critical_start@plt - memcpy@plt - GOMP_critical_end@plt

Loop	Body
L0	CPU_Exec
L1	CPU_Exec - GOMP_critical_start@plt - memcpy@plt - GOMP_critical_end@plt

Figure 22. diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free