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

When the next version of an HPC software system is created, logical errors often get introduced. To maintain productivity, designers need effective and efficient methods to locate these errors. Given the increasing use of hybrid (MPI + X) codes and library functions, errors may be introduced through a usage contract violation at any one of these interfaces. Therefore, tools that record activities at multiple APIs are necessary. Designs find most of these bugs manually, and the efficacy of a debugging tool is often measured by how well it can highlight the salient differences between the executions of two versions of software. Given the huge number of things that could be different – individual iterative patterns of function calls, groups of functions calls, or even specific instruction types (e.g., non-vectorized versus vectorized floating-point dot vector loops) – designers cannot often afford to rerun the application multiple times to collect each facet of behavior separately. These issues are well summarized in many recent studies [**?**]

One of the major challenges of HPC debugging is the huge diversity of the applications, which encompass domains such as computational chemistry, molecular dynamics, and climate simulation. In addition, there are many types of possible "bugs" or, more precisely, errors. An **error** may be a deadlock or a resource leak. These errors may be caused by different **faults**: an unexpected message reordering rule (for a deadlock) or a forgotten free statement (for a resource leak). There exists a collection of scenarios in which a bug can be introduced: when developing a brand new application, optimizing an existing application, upscaling an application, porting to a new platform, changing the compiler, or even changing compiler flags. Unlike traditional software, there are hardly any bug-repositories, collection of trace data or debugging-purpose benchmarks in HPC community. The heterogeneous nature of HPC bugs make developers come up with their own solutions to resolve specific class of bugs on specific architecture or platforms that are not usable on other [**?**].

When a failure occurs (e.g., deadlock or crash) or the application outputs an unexpected result, it is not economic to rerun the application and consume resources to reproduce the failure. In addition, HPC bugs might not be reproducible due to non-deterministic behavior of most of HPC applications. Also the failure might be caused by a bug present at different APIs, system levels or network, thus multiple reruns might be needed to locate the buggy area. In our previous work[**?**], we have introduced ParLOT that collects whole program function call traces efficiently using dynamic binary instrumentation. ParLOT captures function calls and returns at different levels (e.g., source-code and library) and incrementally compress them on-the-fly, resulting in low runtime overhead and significantly low required bandwidth, preserving the whole-program dynamic behavior for offline analysis.

In the current work, we introduce DiffTrace, a tool-chain that post-mortem analyze *ParLOT Traces* (PTs) in order to supply developers with information about dynamic behavior of HPC applications at different levels with different granularities towards debugging. Topology of HPC tasks on both distributed and shared memory often follow a "symmetric" control flow such as SPMD, master/worker and odd/even where multiple tasks contain *similar* events in their control flow. HPC bugs often manifest themselves as divergence in the control flow of processes comparing to what was expected. In other words, HPC bugs violates the rule of "symmetric" and "similar" control flow of one or more thread/process in typical HPC applications based on the original topology of the application. We believe that finding the dissimilarities among traces is the essential initial step towards finding the bug manifestation, and consequently the bug root cause.

Large-scale HPC application execution would result in thousands of PTs due to execution of thousands of processes and threads. Since HPC applications spend most of their time in an outer main loop, every single PT also may contain million-long sequence of trace entries (i.e., function calls and returns). Finding the bug manifestation (i.e., dissimilarities caused by the bug) among large number of long PTs is the problem of finding the needle in the haystack.

Decompressing PTs collected from long-running large-scale HPC applications for offline analysis produce overwhelming amount of data. However, missing any piece of collected data may result in loosing key information about the application behavior. We propose a variation of NLR (Nested Loop

Recognition) algorithm [**?**] that takes a sequence of trace entries as input and by recursively detecting repetitive patterns, re-compresses traces into "iterative sets" in a lossless fashion (intra-PT compression).

Analyzing the application execution as a whole (inter-PT compression) is another goal that we are pursuing in this work. By extracting *attributes* from pre-processed traces, we inject them into a concept hierarchy data structure called Concept Lattice [**?**]. Concept lattices give us the capability of reducing the search space from thousands of instances to just a few *equivalent behavior classes of traces* by measuring the similarity of traces[**?**], making the process of finding the needle more feasible. Comparison of bug-free concept lattice and its equivalent classes with the buggy version of the same application would reveal insights about the dynamic behavior of the program and how the bug changed the classes and their members.

Fowlkes et. al. [**?**] proposed a method for comparing two hierarchical clustering, that is counting the objects that fall into different or same clusters in two different clusterings. Inspired by Fowlkes's approach, we believe that the PTs that fall into different classes before and after the bug are the potential PTs that either manifest the bug impact or reflect the bug root cause. These candidate PTs then require further deeper *observing* and *diffing* with their corresponding bug-free PTs to see what has been changed after the bug introduced.
Our results show ** TODO: Highlights of results obtained as a result of the above thinking should be here. This typically comes before ROADMAP of paper.

In summary, here are our main contributions:

- A tunable tracing and trace analysis tool-chain for HPC application program understanding and debugging
- A variation of NLR algorithm to compress traces in lossless fashion for easier analysis and detecting (broken) loop structures
- A FCA-based clustering approach to efficiently classify similar behavior traces
- A Tunable ranking mechanism to bold 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: Experimental Methodology
- Sec 4: Case Studies
- Sec 5: Related Work
- Sec 6: Discussion, Limitations, Conclusion, Future Work

II. Background

There are two major phases in any "Program Understanding" tool: *data collection* and *data analysis*. To understand the runtime behavior of applications, an efficient tracing mechanism is required to collect informative data during execution of the application. Upon failure or observing unexpected behavior of the program (e.g., wrong answer), studying collected execution data would reveal insight about how program dynamically behaved and what went wrong. In this section, we explain our methodology of data collection and data analysis towards debugging and locating potential root causes of the unexpected behavior. The main source of whole-program dynamic behavior is provided by ParLOT, a dynamic binary tracing tool that captures all function calls and returns and compress them on the fly (section II-A), producing a set of highly compact *ParLOT Traces* (PTs). After preprocessing PTs, a loop-detection-based lossless data reduction mechanism applies to each trace to simplify collected data and reflect facts about loop structures (section II-B). Whole-program analysis in HPC applications only makes sense when the analysis includes cross-thread and cross-process as well as analysis of sequential control flow of every single running thread. Inspired by many works[**?**] [**?**] [**?**][**?**], we have used FCA[**?**] to integrate collected data into a single data structure as a whole, and extract valuable information about different aspect of the execution (section II-C) The major advantage of FCA, is that we can extract a full pair-wise similarity score matrix for all traces of a single execution in an efficient and scalable way, based on attributes that we extract from preprocessed traces. Relying on similarities of traces, we classify PTs into equivalent-behavior classes. This way we reduce the search space from thousands of long PTs to just a few classes of simplified trace representation

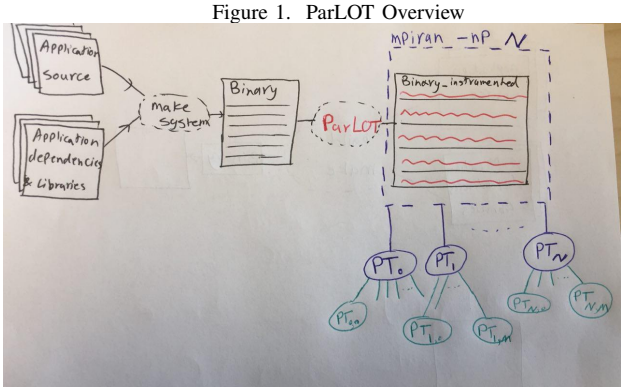
Section II-D talk about how we rank PTs based on their comparison with corresponding PTs from bug-free execution and how we pick some PTs for deeper study.

Section II-E explain why visualizing diffs of a pair of PT is useful and some backgrounds about it.

A. ParLOT: Efficient Trace Collection

The final executable of real HPC applications are often a production of a large code base and a complex build system with numerous dependencies and libraries. Injecting instrumentation code to the source code, as in traditional tools like [????], is not feasible in HPC space. Also recompilation of the application with tools' compile-wrappers, as in TAU[**?**] and Score-p[**?**], may break the build system. Also instrumentation and tracing mechanism of existing tools are often dependent to other libraries that are need to be present on the supercomputer for trace collection. Example: STAT[**?**] and AutomaDeD[**?**] that requires Dyninst[**?**] for instrumentation and MRNet[**?**] and TBON [**?**]. To overcome the trade-off of comprehensive data collection while adding 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

Figure 1. ParLOT Overview



traces on-the-fly and leave majority of the system bandwidth for the application. ParLOT collects *whole* program function call traces with the mindset of *paying a little upfront and save resource and time cost of reproducing the bug*. ParLOT instruments the entry and exit point of each function in the binary using Pin[**?**] (fig. 1). Each ParLOT Trace contain full sequence of function calls and returns for every single thread that running the application code, reflecting the dynamic control flow and call-stack of the application individually. Here we define ParLOT Trace, as we refer to it in the paper:

Definition 1: ParLOT Trace (PT) A ParLOT Trace (PT) is a sequence of ordered integers $< f_i, ..., f_j >$ where f_0 is *return* and f_k is the id of *function k* $(k! = 0)$.

Note that the PT is the pre-processed (decompressed and filtered) version of immediate ParLOT traces. The fresh output of ParLOT traces are highly compressed byte-codes. Also Note that $PT_{p.t}$ refers to the PT that belongs to process p and thread t of that process.

What else do we need here?

B. Loop structure detection

HPC applications and resources are in interest of scientists and engineers for simulating *iterative* kernels. Computer simulation of fluid dynamics, partial differential equations, Gauss-Seidel method and finite element methods in form of stencil codes do include a main outer loop that iterates over some elements (i.e., timesteps) and updates elements. This character of typical HPC applications make PT lengths too long (millions) with much smaller number (hundreds) 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.

1) Loops definition: According to Makoto Kobayashi[**?**] 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 in [**?**] expanded this definition to numerical values for compressing and predicting memory access addresses and designed Nested Loop Recognition (NLR) algorithm. The basic idea behind 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 depth of each node is the depth of *nested* loop(e.g., most outer loop's function is the root of the tree with depth 0). We have modified 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 2: 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 to a sequence of LS_i , we reduce the length of PT by a factor of $\sum_i len(LB_i) * LC_i$. Later we will explain how this lossless representation of PTs eases the process of diffing between a pair of PTs.

What else do we need here?

C. Equivalencing behavior via FCA

To Reduce 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 requires 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. 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. Due to its valuable properties,especially its *partial order*, FCA has been used widely in computer science fields from machine learning and data mining [**?**] to distributed systems [**?**]. However, since we are only interested in grouping similar PTs in this work, we only take advantage of similarity measures [**?**] of concept lattices, and left rest of its properties for our future work. Due to typical HPC application common topologies such as SPMD, master/worker and odd/even where multiple processes/threads behave similarly, our experiments show that large number of PTs can be reduced to just a few groups.

.

```
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()
}
```

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	×	×	×	×		×

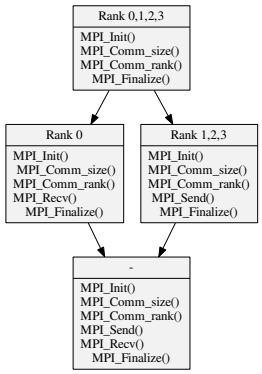


Figure 2. Sample Concept Lattice from Obj-Atr Context in tableII-C

1) Jaccard Similarity Scores: - Some background about Jaccard Similarity Score

- How to obtain full pair-wise Jaccard Similarty Matrix (JSM) from a concept lattice (e.g., LCA approach)

More explanations about concept lattices, CL construction appears in DiffTrace Components section. What else do we need here?

D. Ranking Suspicious PTs

Explanations about how we rank PTs based on their significant difference with respect to bug-free version

E. diffNLR: Reflecting differences

From fig we can see that PT0 and PT1 are in different groups with similarity of XX.

Inspired by `diff` original algorithm[**?**] that has bin used in Git and GNU Diff, we visualize the differences of a pair of PT as shown in fig 5.

This visualization reflects of the differences of **occurrences** of PT elements and their **orders**.

In section IV we show how this visualization can help us locating the points of divergence in PTs, and potential bug manifestation and root cause.

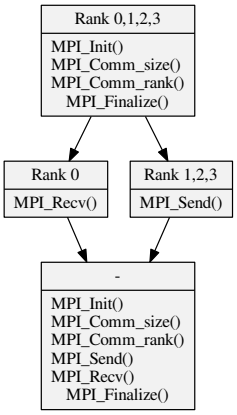


Figure 3. Concept Lattice with reduced labels

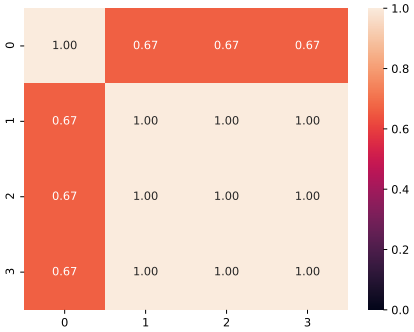


Figure 4. Pair-wise Jaccard Similarity Matrix (JSM) of MPI processes in Sample code

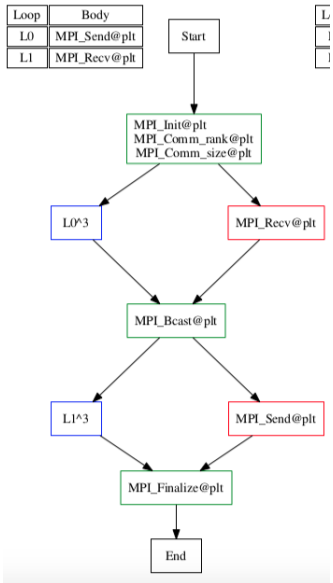
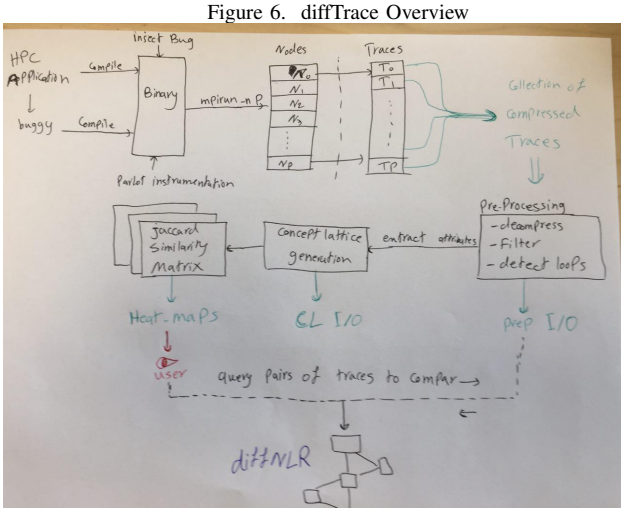


Figure 5. Sample diffNLR

Figure 6. diffTrace Overview



III. DiffTrace Tool Architecture

IS THIS SECTION NECESSARY?

A. General Idea

Figure 6 shows a general overview of DiffTrace and its components.

B. Fault Injection??

C. ParLOT

How we collect ParLOT traces? Is that necessary?

D. Filter

Include a table with all filters and their regular expressions

1) General Filters:

- Returns
- .plt
- Memory
- Network
- Polling
- String
- Customize
- IncludeEverything

2) Target Filters:

- MPI_
- MPIall
- MPI_ Collectives
- MPI_ Send/Recv
- OMPall
- OMPcritical
- OMPmutex

E. Nested Loop Recognition

1) Background:

2) Implementation:

F. Concept Lattice Analysis

1) Background:

- FCA (formal concept analysis) background and citations
- FCA applications in all areas
- FCA applications in Data Mining and Information Retrieval
- FCA applications in distributed systems (Garg's work)
- intro to Concept, Object, Attribute and other definitions

2) Objects/Attributes: Mapping of Object/Attribute (general) to Trace/Attribute (clTrace)

What do we expect to gain by doing so

- Single entity represents the whole execution of HPC application (can be used as signature/model in ML)
- Classifying similar behavior objects(traces)
- Efficient Incremental CL building makes it scalable
- Efficient full pair-wise Jaccard Similarity Matrix extraction

3) CL generation:

- background
- current approach

4) Jaccard Similarity Matrix:

- background
- LCA
- Benefits

G. diffNLR

- motivation
- diff algorithm
- visualization

H. FP-Trace

IV. Experimental Methodology

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 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
```

<table>
<tr><th colspan="10">Filters</th><th rowspan="3">CL Attributes</th><th rowspan="3">Clustering</th></tr>
<tr><th colspan="2">Prime</th><th colspan="2">General</th><th colspan="2">MPI</th><th colspan="2">OMP</th><th colspan="2">Other</th></tr>
<tr><th>Filter</th><th>Description</th><th>Filter</th><th>Description</th><th>Filter</th><th>Description</th><th>Filter</th><th>Description</th><th>Filter</th><th>Description</th></tr>
<tr><td>ret</td><td>Filter Returns</td><td>@plt</td><td>...@plt</td><td>mpi</td><td>MPI_...</td><td>ompcrit</td><td>OMP critical</td><td>Custom</td><td>Defining specific regex to filter</td><td colspan="2">Objects: Traces
Attributes: set of <atr:freq></td><td>single</td></tr>
<tr><td>.plt</td><td>Filter .plts</td><td>mem</td><td>Memory related malloc memcpy etc</td><td>mpiall</td><td>..MPI...
MPID...
PMPI...</td><td>ompmutex</td><td>OMP mutex</td><td>incEverything</td><td>Include whatever is not in the Filters</td><td>Single: set of single trace entries
atr: sing</td><td>No Frequency: only presence of attribute entries matters
freq:-</td><td>complete</td></tr>
<tr><td></td><td></td><td>net</td><td>Network related</td><td>mpicol</td><td>MPI collectives</td><td>ompall</td><td>OMP all functions</td><td></td><td></td><td>Double: set of 2-consecutive entries
atr: doub</td><td>Log10: log(freq) of each entry matters (for large frequency numbers
freq: log10(#atr)</td><td>average</td></tr>
<tr><td></td><td></td><td>poll</td><td>Poll Related poll, yield</td><td>mpisr</td><td>MPI send/recv</td><td></td><td></td><td></td><td></td><td></td><td>Actual: actual frequency of each entry matters
freq: #atr</td><td>weighted</td></tr>
<tr><td></td><td></td><td>str</td><td>String related stcpy strcmp etc</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>centroid</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>median</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>ward</td></tr>
</table>

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

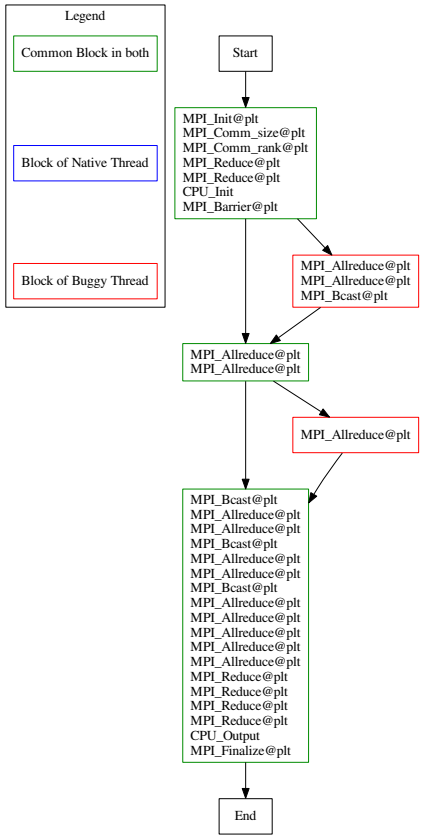


Figure 8. Bug1: diffNLR $PT_{0,0}$ - buggy vs. native

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**	-

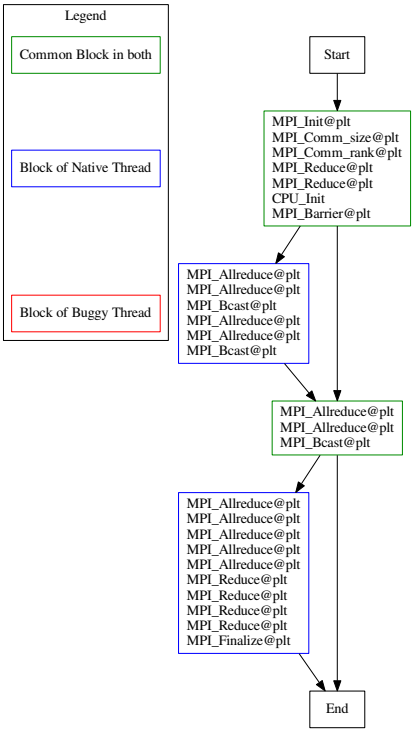


Figure 9. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

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::

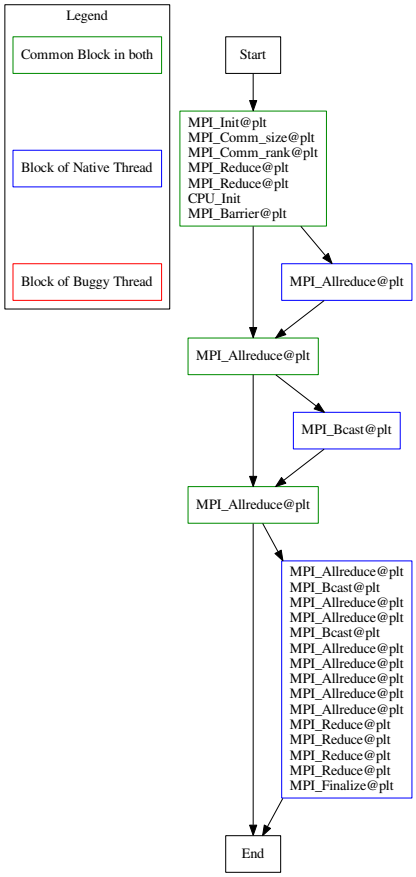


Figure 10. Bug2: diffNLR $PT_{1,0}$ - buggy vs. native

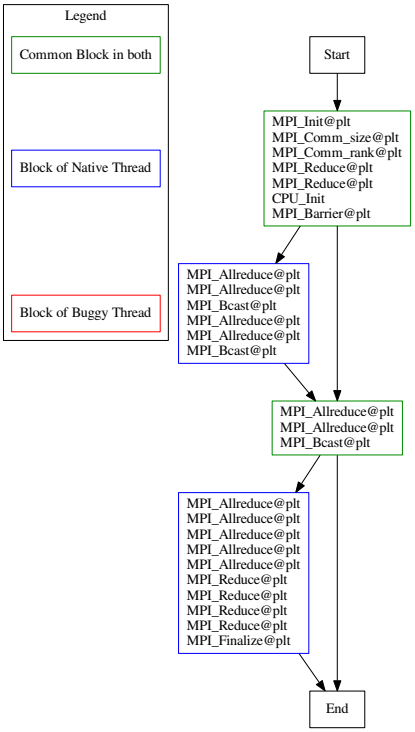


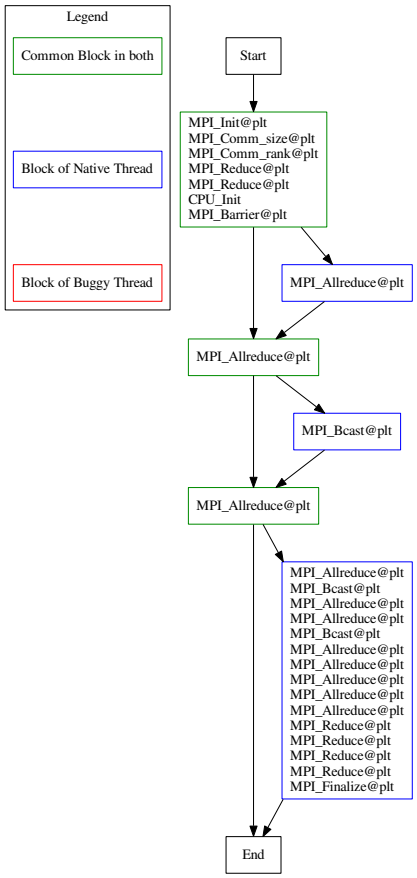
Figure 11. Bug2: diffNLR $PT_{3,0}$ - buggy vs. native

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

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

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

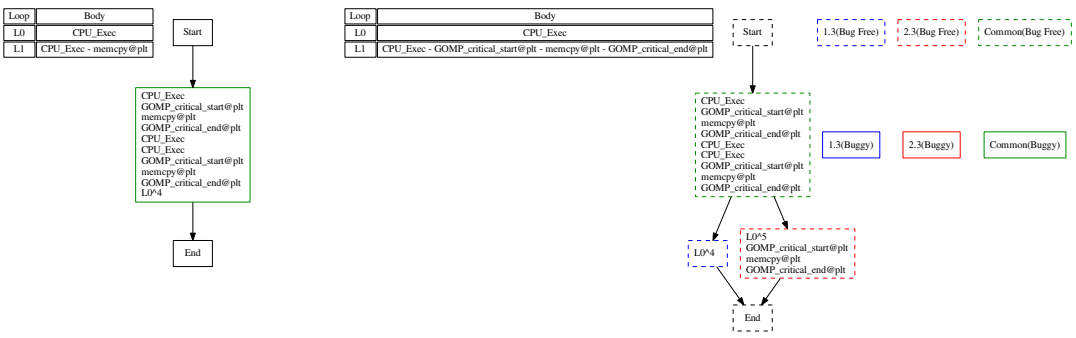


Figure 13. diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free

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

VI. Discussion and Future Work

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)

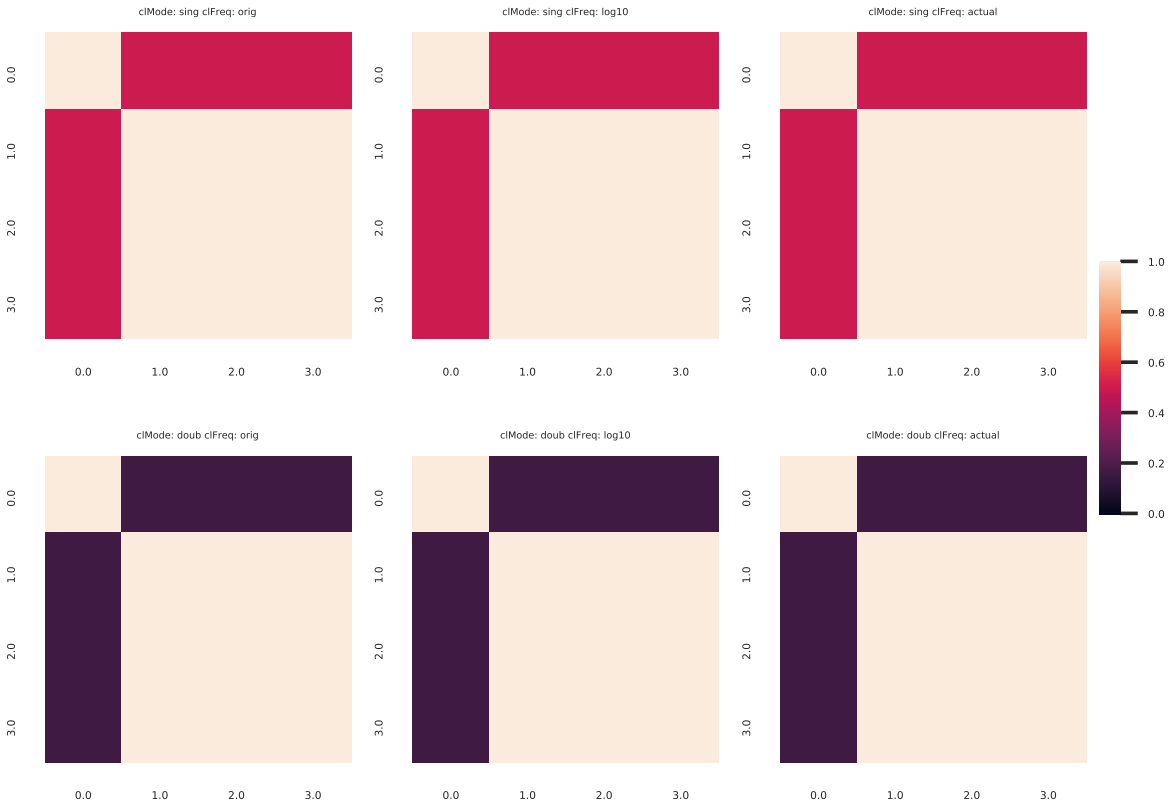


Figure 16. JSM of bug-free version of hybrid Matmul (MPI Only)

	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

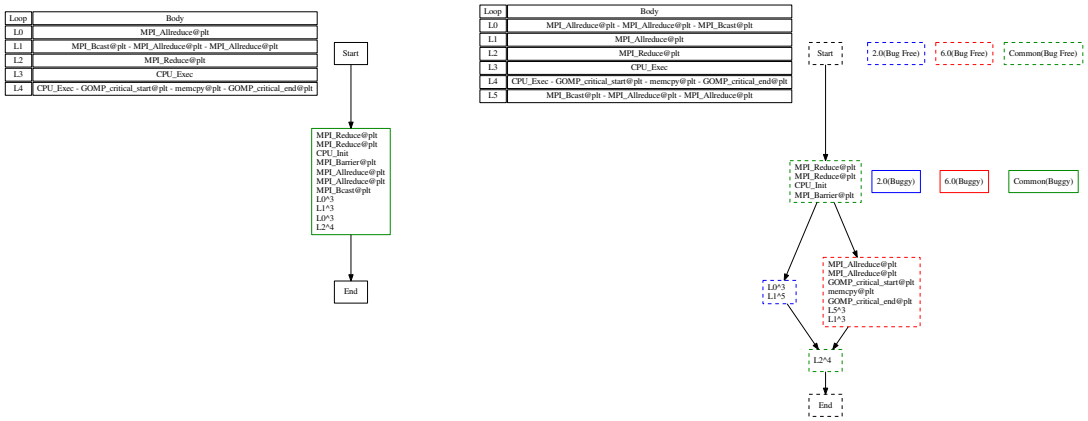


Figure 18. diffNLR of process 2 and process 6 buggy(AllReduce() wrong op) vs. bug-free

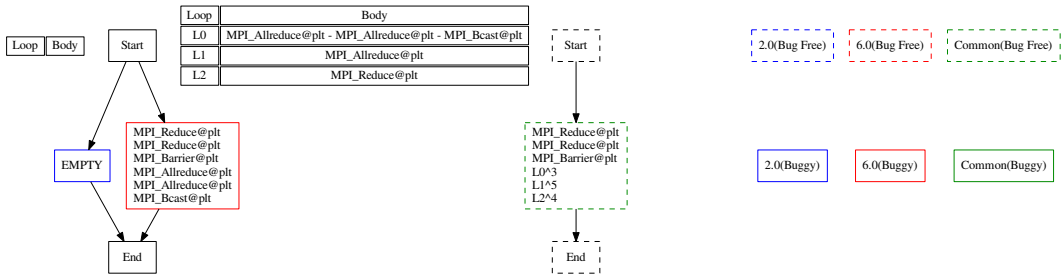


Figure 19. diffNLR of process 2 and process 6 buggy(AllReduce() wrong size) vs. bug-free

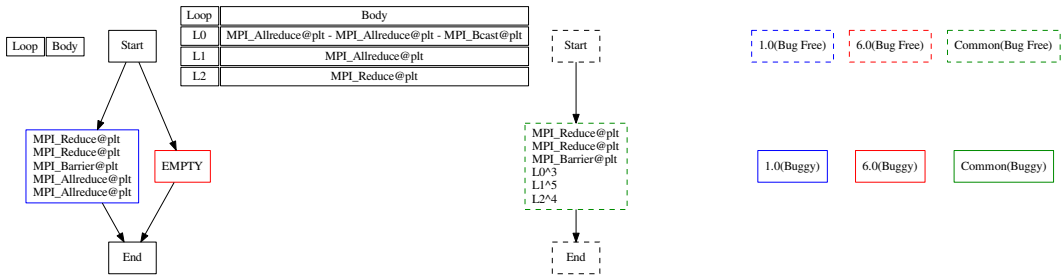


Figure 20. diffNLR of process 1 and process 6 buggy(AllReduce() wrong size) vs. bug-free

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	[illegible]	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	[illegible]	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	[illegible]	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	[illegible]	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

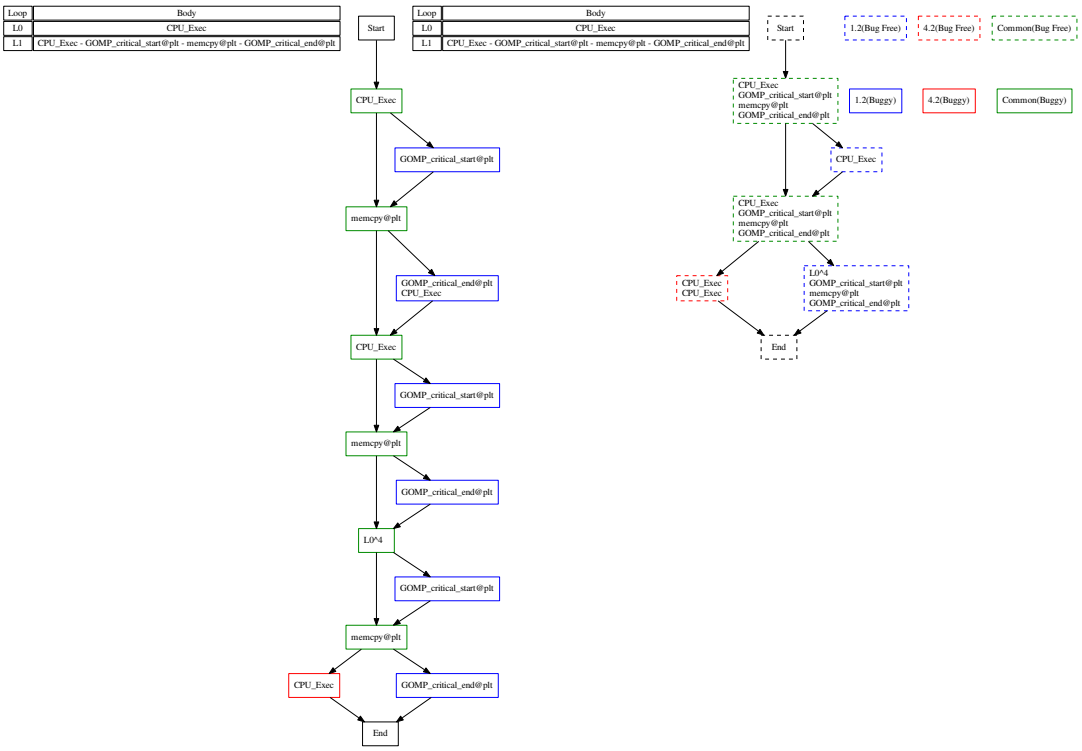


Figure 22. diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free

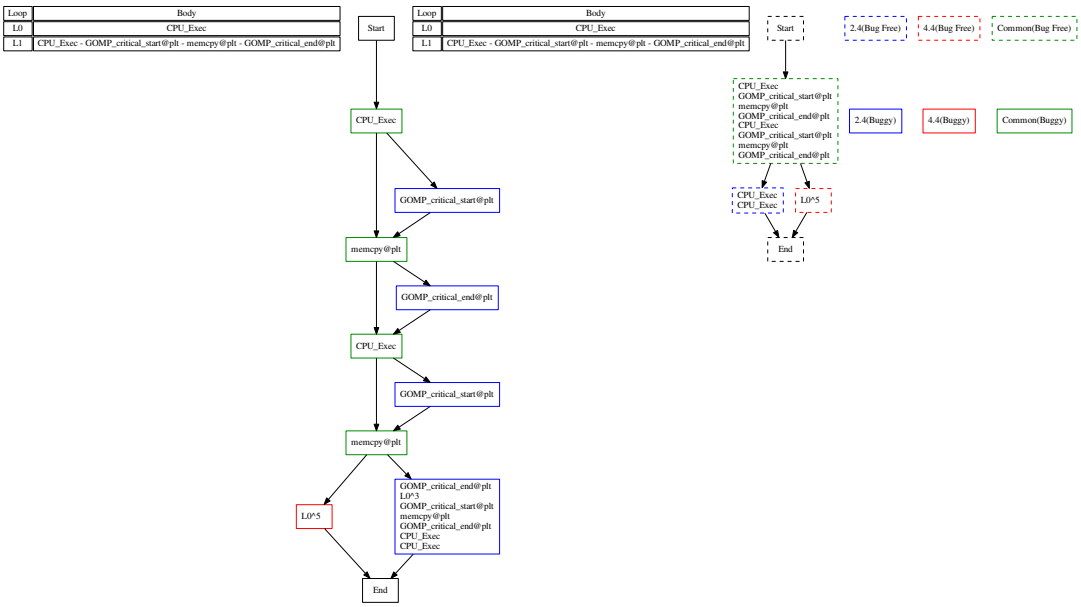


Figure 23. diffNLR of process 1 thread 3 and process 2 thread 3 buggy(missing critical section) vs. bug-free