Automated Analysis of Time Series Data to Understand Parallel Program Behaviors

Lai Wei
Rice University
Houston, Texas, USA
lai.wei@rice.edu

John Mellor-Crummey
Rice University
Houston, Texas, USA
johnmc@rice.edu

ABSTRACT

Traditionally, performance analysis tools have focused on collecting measurements, attributing them to program source code, and presenting them; responsibility for analysis and interpretation of measurement data falls to application developers. While profiles of parallel programs can identify the presence of performance problems, often developers need to analyze execution behavior over time to understand how and why parallel inefficiencies arise. With the growing scale of supercomputers, such manual analysis is becoming increasingly difficult. In many cases, performance problems of interest only appear at larger scales. Manual analysis of time series data from executions on extreme-scale parallel systems is daunting as the volume of data across processors and time makes it difficult to assimilate. To address this problem, we have developed an automated analysis framework that generates compact summaries of time series data for parallel program executions. These summaries provide users with high-level insight into patterns in the performance data and can quickly direct a user's attention to potential performance bottlenecks. We demonstrate the effectiveness of our framework by applying it to time-series measurements of two scientific codes.

CCS CONCEPTS

• **General and reference → Performance**;

KEYWORDS

Automated performance analysis, iteration identification, clustering, performance loss attribution

ACM Reference Format:
Lai Wei and John Mellor-Crummey. 2018. Automated Analysis of Time Series Data to Understand Parallel Program Behaviors. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* 12 pages. https://doi.org/10.1145/3205289.3205308

1 INTRODUCTION

Parallel computing has become an indispensable tool for scientific inquiry. Simulations of increasing ambition and analysis of data from high resolution instruments have propelled the construction of parallel computers of increasing scale. While the computational power of such systems has increased dramatically, the performance of many applications has failed to scale proportionally.

Many factors contribute to the inefficiency of parallel applications. The need to understand application behaviors and pinpoint causes of inefficiency has led to the development of a broad array of tools for measuring and analyzing application performance. Well-known parallel performance tools include HPCToolkit [1], MAP [3], Paraver [20], Scalasca [11], TAU [21], Vampir [18], and VTune [17]. These tools measure an application's performance and employ post-mortem analysis to prepare measurement data for presentation. Using such tools, experts browse through performance data and manually identify code regions associated with inefficiencies.

While profiles of parallel program executions can identify the presence of performance problems, often developers need to analyze execution behavior over time to understand how and why parallel inefficiencies arise. Figure 1 shows an annotated screenshot from HPCToolkit's hpctraceviewer that examines the behavior over time of a 512-processor run of PFLOTRAN [14]—a code for massively parallel simulation of subsurface flow and reactive transport. Each row in the diagram represents a process; time advances from left to right. In the figure, six phases are apparent, as noted at the bottom. The first phase appears to have some load imbalance. The series of vertical bands in the fifth phase suggests an iterative behavior. Further manual analysis is needed to understand the performance of each phase. One would need to zoom in to examine the execution at a higher resolution and select appropriate call path depths to derive detailed insights. For long executions involving thousands of processes, such manual analysis is difficult if not impractical.

Automated analysis of time series data is necessary to understand the performance of large-scale executions of parallel programs. To automate such analysis, one must tackle the vastness of time-series data in three dimensions – process, time, and call-path depth. One must decide how to identify interesting features, quantify performance losses, and report them to users.

To address these challenges, we developed a novel automated framework that generates compact summaries from time series data. These summaries indicate patterns in the data and direct a developer's attention to performance losses, which are attributed to call paths where the losses occur. Visualization of these summaries enables application developers to immediately understand performance bottlenecks and locate causes.

This paper highlights three novel aspects of our framework. First, we present an algorithm that identifies iterative behavior in a time series of samples. Second, our framework computes a set of metrics to quantify the severity of performance losses and

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205308

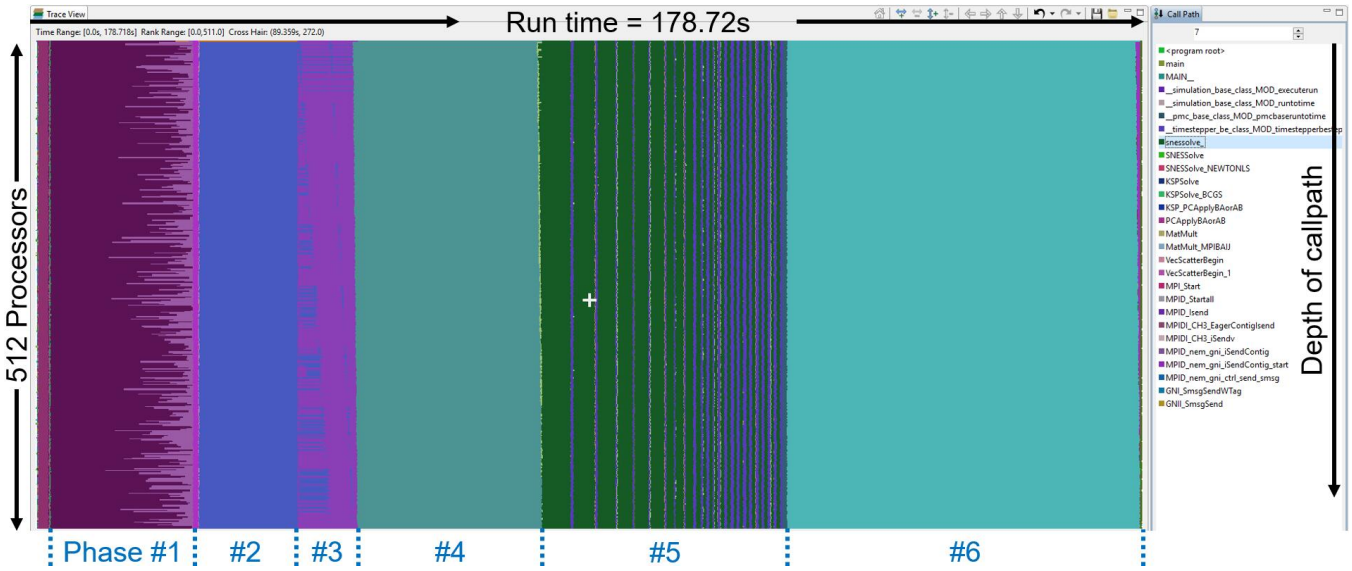


Figure 1: Annotated screenshot of HPCToolkit's hpctraceviewer examining a 512-processor execution of PFLOTRAN. Processes are arranged along the vertical axis and time flows left to right. Each pixel in a horizontal trace line for a process shows a procedure frame on the call path of the process at that time. Each procedure has a unique color. Depth of the frame in the call path can be selected on the right. Six phases are readily visible in the view, as noted at the bottom of the figure.

attribute such losses to call paths where they originate. Third, our visualization of automated summaries helps users quickly locate causes of performance loss.

The rest of the paper is organized as follows. Section 2 describes prior work analyzing parallel program performance and execution traces in particular. Section 3 describes how our tool recognizes iterative behavior in a time series of asynchronous samples and then organizes information about iterations into a structured tree-based representation. Section 4 describes how our tool evaluates variation across threads and iterations, organizes them into a set of equivalence classes, and generates summaries of potential performance losses. Section 5 illustrates the utility of our automated analysis by applying it to executions of two scientific codes. Section 6 summarizes our work and future directions.

2 RELATED WORK

Many tools collect and visualize time series data of parallel programs, including HPCToolkit [1], Paraver [20], Scalasca [11], TAU [21], Vampir [18], and VTune [17]. With such tools, users typically manually browse through measurement data to uncover performance inefficiencies. A few of these tools automate some kinds of analysis, but none of them offer the fully automated support users need.

In HPCToolkit, Tallent et al. [22] locate load imbalance by analyzing profiles of parallel program executions. Profiles can identify the presence of performance problems; however, it is often insufficient for diagnosing how and why parallel inefficiencies arise.

TAU's PerfExplorer [16] clusters threads according to metric profiles, e.g. time spent in each function. Min, max, and average profiles of each cluster are calculated for comparison. Since profiles can't explain all kinds of performance losses, clusters of profiles can't either.

In Paraver, Gonzalez et al. [12, 13, 19] divide an execution of a parallel program into regions between MPI communications and cluster these regions using their metrics, such as Instructions Per Cycle (IPC) and cache misses. Paraver colors regions according to their clusters in a visualization to highlight outliers. However, manual effort is still needed to judge if these outliers are indications of performance loss and locate the causes of any performance loss.

Scalasca [11] automates the analysis of traces to identify wait states in a parallel program. Furthermore, Boehme et al. [6] employ a delay analysis mechanism to attribute those wait states to their root causes. Unfortunately, their analysis requires complete traces of MPI communications that grow too quickly to be practical for large-scale parallel applications. As a result, selective tracing to record events only for short execution intervals is needed. Prior manual analysis is necessary to select such intervals.

Casas et al. [7] identify phases and iterations in MPI applications. They convert time-series data into signals and apply signal processing algorithms to detect program structures. Aguilar et al. [2] generate event flow graphs of MPI calls online and use these graphs to discover nested loops in a program. These two techniques identify regular patterns in traces that contain every MPI operation and would not apply to sampled time series data.

Bahmani and Mueller [5] employ an online clustering algorithm for traces of MPI events that capture communication behaviors. They apply a K-farthest clustering algorithm on a set of parameters associated with MPI events. Only the representative processor of each cluster records MPI events to reduce trace size. The clustering algorithm we describe in section 4.2 applies this idea to identify constant behaviors in time series data across threads and iterations.

Curtsinger et al. [9] try to estimate how the effects of optimizations would propagate along program paths with a technique that they call *causal profiling*. They virtually speed up a function in experimental runs to evaluate the benefit of optimizing it. Unfortunately, evaluating a function is time-consuming and users need to supply a small list of functions. Prior manual analysis is required to identify a list of interesting functions.

Tallent et al. [23] propose an efficient algorithm to collect representative paths for MPI applications. Differences between profiles of representative paths can be used to estimate parallel performance and locate load imbalance. However, a few representative paths may be insufficient to pinpoint causes of performance losses and quantify their effects over all processes.

3 ASSIMILATING TIME SERIES DATA

Prior work on automating analysis of parallel program behavior over time has focused on analysis of traces that include every event of interest (e.g., the beginning or end of an MPI call) during an interval. Collecting such events can lead to very large trace files. In this section, we describe how we collect and digest a time series of asynchronous call path samples to prepare them for automated analysis. We organize these samples in a tree consisting of program constructs that include call sites, loops, and iterations.

As input to our analysis, we use a time series of asynchronous call path samples collected using HPCToolkit [1] for each process in a parallel program execution. To help analyze this data, we developed a static binary analyzer using Dyninst's ParseAPI [8] to build control flow graphs (CFGs) for every function and loop. Such CFGs help us identify iterative patterns in HPCToolkit's traces that result from program loops; this information serves as input to construct top-down summaries, described in Section 4.

3.1 Time Series Data from HPCToolkit

As a parallel program executes, HPCToolkit collects a sequence of asynchronous call path samples for each thread. A sample for a thread consists of a timestamp, a call path, and an LCA (Least Common Ancestor). A call path corresponds to a program counter (PC) followed by a sequence of return addresses that represent each procedure frame on the thread's stack when the sample was recorded. An LCA associated with a call path sample for a thread represents the lowest procedure frame that remained on the thread's call stack since the previous sample.

To associate samples with source code contexts, HPCToolkit employs binary analysis to relate machine code addresses in an executable to source lines, call sites, loops, and procedures. Given an executable for the program in Figure 2, HPCToolkit would identify that calls to `A`, `B`, and `C` on line 7 – 9 are made inside a loop. For any call path sample containing an address in function `A`, `B`, or `C`, HPCToolkit's postmortem analysis would augment the call path with the loop context enclosing call sites to the functions. Figure 3 illustrates what a time series of call path samples might look like for an execution of the sequential program shown in Figure 2.

To derive the LCA for a pair of adjacent call path samples, we leverage *trampolines* and *mark bits* inserted by HPCToolkit when collecting a call path sample. On x86 architectures, HPCToolkit can insert a lightweight trampoline [10] to intercept procedure returns; HPCToolkit uses these trampolines to reduce unwinding overhead by remembering the call path prefix in common between two adjacent call path samples, which we refer to here as the LCA. On Power architectures, instead of inserting a trampoline when collecting a call path sample, HPCToolkit can mark each return address on its call stack by setting its lowest bit [4]. This is safe because the word-oriented Power architecture ignores the two lowest bits in a function's return address. HPCToolkit can detect the call path prefix in common between two adjacent call path samples by observing the lowest procedure frame with a marked return address.

In Figure 3, the LCA between samples at `T5` and `T6` is `loop@6`, because `A@7` in `T5` is popped from the stack when the second iteration of `loop@6` finishes while the third (last) iteration of `loop@6`

```
1 void A() { ... } // work that takes 1~2 samples
2 void B() { ... } // work that takes 1~2 samples
3 void C() { ... } // work that takes 0~1 sample
4 void foo() {
5       C();
6       for (int k = 0; k < 3; k++) {
7           A();
8           if (k==0) B();
9           if (k==2) C();
10      }
11 }
12 int main() {
13      foo();
14      return 0;
15 }
```

Figure 2: An example program in C.

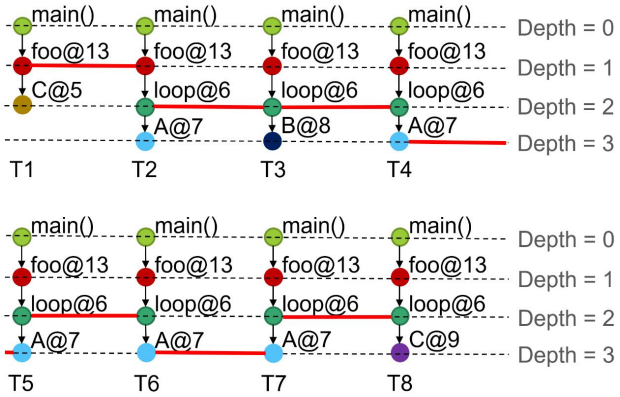


Figure 3: An illustration of a possible time series of call path samples that could be recorded by HPCToolkit for the program in Figure 2. Timestamps T1–T8 are associated with the samples above them. A call path sample consists of a sequence of nodes representing call sites and loops. Each call site is tagged with name of the callee function and line number of the call. Loops are tagged with line numbers. LCAs of neighboring samples are connected with red lines. PC values at call path leaves are omitted for all call paths.

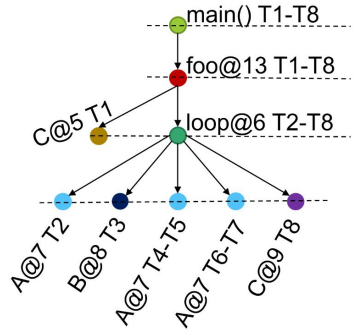


Figure 4: A temporal context tree built from samples in Figure 3. Nodes are annotated with timestamp ranges in T1–T8.

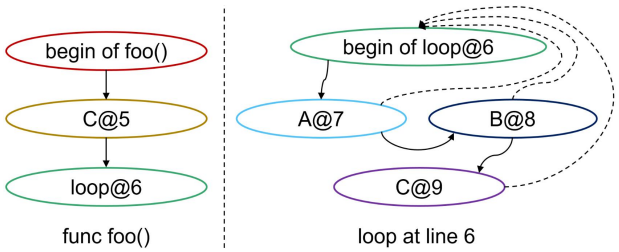


Figure 5: Control flows for function foo and loop at line 6 of the program in Figure 2. In the control flows for a routine or loop, we have an entry node indicating the begin of that routine or loop. Other nodes represents call sites and loops within that routine or loop. Concrete lines are forward flows while dashed lines are back edges in loops.

pushes a new instance of `A@7` onto stack. HPCToolkit knows at run time that `A@7` doesn't persist on the stack between `T5` and `T6` while `foo@13` is the lowest frame that persists. Our post-mortem analysis can determine that the execution did not leave `loop@6` between these two samples and pushes the LCA one level deeper. Knowledge of the LCA helps the iteration identification algorithm described in Section 3.2 identify individual iterations of loops represented by a sequence of call path samples. Without LCAs, we cannot tell precisely when the second iteration of `loop@6` finishes.

To facilitate top-down analysis of a series of call path samples, our tool builds a tree-based representation of a series of call path samples by merging common prefixes of temporally-adjacent samples. This representation, which we call a *temporal context tree*, has nodes representing call sites and loops annotated with ranges of timestamps. A temporal context tree built from the sequence of samples in Figure 3 is shown in Figure 4. Children of a node in the tree are ordered from left to right by their timestamps. Notice that `A@7 T4-T5` is not merged with `A@7 T6-T7` as the LCA at `T6` indicates that `A@7` is not a common prefix of samples at `T5` and `T6`.

3.2 Identifying Iterations

Our framework divides execution of a loop into iteration-based segments to enable automated analysis of time series data at a fine granularity. Compared to analysis of loop profiles, analyzing a sequence of loop iterations can identify more patterns that represent potential bottlenecks. Our analysis transforms the temporal context subtree rooted at `loop@6` in Figure 4 into Figure 6 by interposing a sequence of nodes between `loop@6` and its children to partition the sequence of samples it represents into a sequence of loop iterations.

Algorithm 1: Tag children of a loop with iteration identifiers.

Data: a loop node L in the temporal context tree; a graph $G = (V, E)$ containing a set of control flow edges where both the source and sink node of an edge are within the loop L.
Result: all children of L tagged with an iteration identifier.

```
1  G' ← (V, E') be a subgraph of G without back edges;
2  for v ∈ V do
3      successor[v] ← all nodes reachable from v in G'
4  children[] ← all children of L ordered by timestamps;
5  iter ← 0;
6  prev ← entry node in V;
7  for k ∈ {0, ..., |children|-1} do
       // getCFNode(N) returns the node in V that
          represents the call site or loop N.
8      current ← getCFNode(children[k]);
9      if current not in successor[prev] then
           // A back edge in G must have been taken. Assign
              a new iteration identifier.
10         iter ← iter + 1;
11     tag children[k] with iter;
12     prev ← current;
```

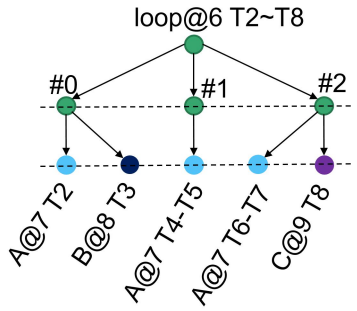


Figure 6: A loop node with explicit iterations after iteration identification using the temporal context tree in Figure 4 and the control flows in Figure 5

Dyninst's ParseAPI library analyzes binaries and computes a control flow graph (CFG) for each routine. It also identifies loops and loop nests in each CFG. We use CFGs from ParseAPI to derive control flows between call sites and loops within a routine or loop. An example of such flows for the code in Figure 2 is shown in Figure 5. For each routine or loop, besides control flows between nodes representing call sites and loops within it, we have an entry node indicating the begin of the routine or loop.

Algorithm 1 uses control flows between call sites and loops to separate iterations of a loop. With results of this analysis, our tool transforms a temporal context tree by adding a layer of iterations between each loop and its children as shown in Figure 6. Our algorithm separates iterations when back edges in control flows are

taken. Children tagged with different identifiers must belong to different iterations. However, the opposite is not guaranteed when applied to a time series of samples; children with the same identifier may belong to different iterations in the execution. To understand why, imagine that the execution time of a loop iteration in Figure 2 is smaller than the sampling period. In this case, HPCToolkit may record a sample in `A` in the first iteration, followed by a sample in `C` in the third iteration. Despite the fact that `A` and `C` are not called in the same iteration, our algorithm would tag them with the same identifier because it cannot prove that they must have been recorded in different iterations.

Fortunately, when the sampling period is not short enough to resolve iterative behavior, identifiers would only be tagged to a few children that last a few samples – in the above example, the identifier is applied to two call frames each lasting one sampling period. To determine if children with the same identifier belong to a same iteration in the execution, our analysis framework runs the algorithm and tests every identifier against the following criteria:

- for an identifier, if the number of children being tagged is $\geq$ *CHILD_NUM_ACC*, this identifier is accepted;
- the identifier is also accepted if the duration of any tagged child is $\geq$ *CHILD_DUR_ACC*;
- otherwise, the identifier is rejected.

An identifier is accepted when we are confident that all its tagged children belong to the same iteration in the execution. Our case studies show that lots of assignments to those two parameters yield good results. We currently use *CHILD_NUM_ACC* = 5 and *CHILD_DUR_ACC* = 5 × *sampling period*.

All children tagged with rejected identifiers are merged to form a set of profile nodes. Profile nodes are different from regular ones as they only have their durations accumulated instead of recording the range of timestamps for each individual instance. When the duration of children tagged with rejected identifiers are dominant in a loop – greater than half the duration of the loop, we find it less beneficial to keep detailed records of accepted identifiers. In such cases, the entire loop node will be transformed into a profile node.

In our experiments, most identifiers are accepted for loops where iterations run a sufficient amount of time, which enables further iteration analysis on these loops. For loops with very short iterations, users are unlikely to devote interest to the behavior of each iteration and a profile node that summaries all iterations is sufficient.

So far, we have described how we ingest a sequence of call path samples from HPCToolkit, assemble them into a temporal context tree, and use control flow information extracted from the application binary using Dyninst's ParseAPI to group samples into iterations. Loops in the temporal context tree either have their iterations identified or have their children merged into profile nodes.

4 GENERATING TOP-DOWN SUMMARIES

Often, time series data contains multiple instances of similar behaviors. Iterations in a loop may be comparable and threads may perform similar work. Rather than presenting users with time series data that shows thousands of threads running millions of iterations each, our framework generates concise summaries of these behaviors and highlights interesting features of the data.

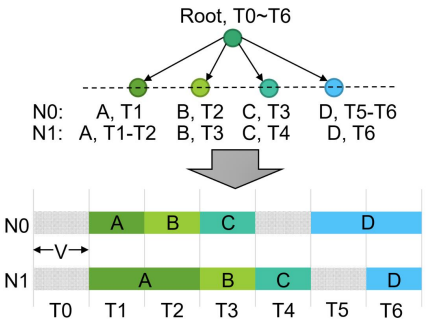


Figure 7: Comparison between N_0 and N_1: two temporal context subtree roots. Only one tree is shown as both nodes share the same sequence of children. To aid in comparing the temporal sequences represented by N_0 and N_1, we convert the sequence of their children into a sequence of bars that shows the sequence of states representing each subtree. Gray bars represent exclusive running time of N_0 or N_1, which means that a sample was recorded at an instruction in the root node, not in a child call site or a loop.

One interesting feature is variation across threads and iterations. When threads and iterations have different behaviors, investigating variation can reveal performance bottlenecks. In our work, we seek to provide answers to the following questions. Is there any significant variation? How large is it? Where does the variation arise? Answers to these questions can provide insight into performance loss and help users devise optimizations.

In the next section, we discuss quantifying pairwise differences. In Section 4.2, we use this as a building block for clustering, which identifies repetitive behaviors in time series data. In Section 4.3, we discuss how we compute and keep track of the source of variation, providing answers to the aforementioned three questions.

4.1 Quantifying Pairwise Differences

To cluster time series data, we need to quantify the differences between pairs of temporal context tree nodes. In this section, we describe how our framework quantifies such differences.

Figure 7 shows a pair of temporal context tree nodes N_0 and N_1 with some differences. We define their difference $diff(N_0, N_1)$ as the minimum number of additions or deletions needed to make them equivalent. With V as the sampling period, difference between N_0 and N_1 is $2 \times V$ as we can turn N_0 into N_1 by extending the run time of `A` by V and cutting the run time of `D` by V.

Besides using $diff(N_0, N_1)$ to denote the difference between two temporal context tree nodes, we also define $diffRatio(N_0, N_1)$ to quantify the difference relative to the duration of both tree nodes. We have

$$diffRatio(N_0, N_1) = \frac{diff(N_0, N_1)}{duration(N_0) + duration(N_1)}$$

As of Figure 7, we have $diffRatio(N_0, N_1) = 2/14 = 14.3\%$.

Algorithm 2 computes the difference between two regular temporal context tree nodes N_0 and N_1, denoted as $diff(N_0, N_1)$. At line 8, the algorithm accumulates difference between matched children of N_0 and N_1 that refer to the same call site or loop. At line 13, control

Algorithm 2: computes the difference in run time between two regular temporal context tree nodes that refer to the same call site.

Data: sampling period V; a pair of regular temporal context tree node N_0 and N_1 refering to the same call site S; the set of control flow edges CF where both the source and sink node of an edge are within S.
Result: difference in execution time between N_0 and N_1.

```
1  Function diff(N0, N1):
2      sum ← 0;
       // For simplicity, we use N[i] to denote the i_th
          child of node N (children are ordered by
          timestamps); S[N] to denote the start time of node
          N; and E[N] to denote the end time of node N.
       // Gaps are used to count differences in gray bars in
          Figure 7. S[N[i]] returns E[N] if
          i = numChild(N).
3      Gap0 ← S[N0[0]] − S[N0]; Gap1 ← S[N1[0]] − S[N1];
4      C0 ← 0; C1 ← 0;
5      while C0 < numChild(N0) and C1 < numChild(N1) do
6          if N0[C0] and N1[C1] refer to the same call site or loop then
7              sum ← sum + diffGap(Gap0, Gap1);
8              sum ← sum + diff(N0[C0], N1[C1]);
9              Gap0 ← S[N0[C0 + 1]] − E[N0[C0]];
10             Gap1 ← S[N1[C1 + 1]] − E[N1[C1]];
11             C0 ← C0 + 1; C1 ← C1 + 1;
12         else
13             if N0[C0] is a successor of N1[C1] in CF then
                   // Increment C1 as a later child of N1 may
                      match N0[C0] at line 6.
                   // dummy(N) generates a copy of N with
                      no child and zero execution time.
14                 sum ← sum + diff(N1[C1], dummy(N1[C1]));
15                 Gap1 ← Gap1 + S[N1[C1 + 1]] − E[N1[C1]];
16                 C1 ← C1 + 1;
17             else
18                 sum ← sum + diff(N0[C0], dummy(N0[C0]));
19                 Gap0 ← Gap0 + S[N0[C0 + 1]] − E[N0[C0]];
20                 C0 ← C0 + 1;
21     while C0 < numChild(N0) do
           // Same as line 18 - 20
22     while C1 < numChild(N1) do
           // Same as line 14 - 16
23     sum ← sum + diffGap(Gap0, Gap1);
24     return sum;
25 Function diffGap(Gap0, Gap1):
26     if |Gap0 − Gap1| > 2 × V then
27         return |Gap0 − Gap1| − 2 × V;
28     return 0;
```

flows are used to determine which child to process first when children of N_0 and N_1 don't match. The algorithm builds dummy nodes for unmatched children and accumulates the difference at line 14 and 18. At line 7 and 23, it accumulates difference between exclusive run time of N_0 and N_1 (gray bars in Figure 7). The algorithm runs recursively at line 8, 14, and 18 until it hits leaf nodes in the tree.

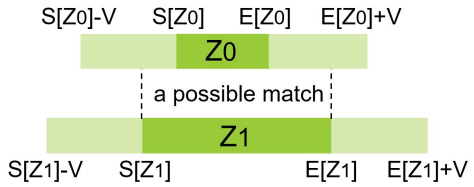


Figure 8: Comparison between two leaf nodes Z_0 and Z_1. We use V to denote the sampling period, S[Z] to denote the start time of Z, and E[Z] to denote the end time of Z. While Z_0 and Z_1 are not exactly the same, a perfect match is possible when we take the sampling period into account.

Difference between profile nodes can be computed in a similar way. Computation of difference between regular temporal context tree nodes that refer to the same loop is covered in Section 4.2.1.

In the implementation of *diffGap* at line 25 of Algorithm 2, we deduct two times the sampling period from the actual difference. The reason is illustrated by Figure 8. With a sampling period of V, a node Z could have started anytime between $S[Z] - V$ and $S[Z]$ and ended anytime between $E[Z]$ and $E[Z] + V$. As a result, Z_0 and Z_1 in Figure 8 can have a perfect match even if they are not exactly the same. In our experiments, we find it essential to take the sampling period into account when computing *diffGap*. If not, the result becomes useless as the true variation will be dominated by noise that arises from not accounting for the sampling period properly. In practice, we also notice that the sampling period is not constant over all samples – it varies to some degree. Algorithm 2 was designed with such variation in mind.

4.2 Clustering

Clustering iterations or threads in the time series data brings several benefits. First, it saves space and time spent in analysis by condensing many instances into a few equivalence classes. Second, it reduces analysis effort by summarizing thousands of instances with one representative. Third, it directs attention to variation among distinct clusters which may be the source of performance losses.

Algorithm 3 employs the K-farthest clustering algorithm proposed by Bahmani et al. [5] to cluster iterations and threads. We choose this algorithm because it has a low time complexity and is easy to parallelize (discussed in Section 4.2.2). The algorithm uses a divide-and-conquer method to return no more than K clusters representing all instances. Computing the difference between a pair of clusters at line 13 and 24 of Algorithm 3 takes $O(G)$ time, where G is the size of the temporal context tree. Those functions are called $O(K^2)$ times by *mergeClusterSet*; as a result, time complexity for *mergeClusterSet* is $O(K^2G)$. Function *mergeClusterSet* is called $O(M/K)$ times by *getClusterSet*, where M is the number of instances in node N. Therefore, the overall time complexity is $O(MKG)$. We currently set $K = O(\log_2 M)$.

In our implementation, a representative is associated with each cluster, which is the average of all instances in that cluster. For a cluster with one instance, its representative is that instance. For a cluster with more than one instance, a representative is computed when

Algorithm 3: K-farthest clustering for temporal context tree.

Data: a temporal context tree node N which is a loop node with iterations or a root node with threads; h and e represent the start and end index of the instances to be clustered; K indicates the maximum number of clusters.

// For simplicity, we use $N[i]$ to denote the i_{th} iteration or thread of N.

// Return a set of clusters representing instances from $N[h]$ to $N[e-1]$.

1 **Function** getClusteSet(N, h, e, K):
2 **if** $e - h \leq K$ **then**
3 **for** $i \in \{h, \ldots, e\text{-}1\}$ **do**
4 $clusters[i-h] \leftarrow$ N[h] ;
5 **return** $clusters$;
6 $mid \leftarrow (h+e)/2$;
7 $S_0 \leftarrow getClusterSet(N, h, mid, K)$;
8 $S_1 \leftarrow getClusterSet(N, mid, e, K)$;
9 $clusters \leftarrow mergeClusterSet(S_0, S_1, K)$;
10 **return** $clusters$;

// Merge two sets of clusters to one containing no more than K clusters.

11 **Function** mergeClusterSet(S_0, S_1, K):
12 Copy S_0 and S_1 to S;
13 Compute pairwise *diffRatio* for clusters in S;
14 (C_0,C_1) $\leftarrow$ the pair of clusters in S with lowest *diffRatio*;
15 $min \leftarrow$ the lowest *diffRatio* computed;
16 $max \leftarrow$ the highest *diffRatio* computed;
17 M $\leftarrow$ number of clusters in S;
18 **while** $((min < Ratio_{min})$ // continue when min is below a threshold
19 *or* $(min/max < Ratio_{rel})$ // or when min is small compared to max
20 *or* $(M > K))$ // or when M exceeds K
21 **do**
22 $C \leftarrow merge(C_0, C_1)$;
23 Insert C into S and delete C_0 and C_1 from S ;
24 Compute *diffRatio* between C and other clusters;
25 Update C_0, C_1, min, max, and M as line 14–17;
26 **return** S;

$merge(C_0, C_1)$ is called at line 22 of Algorithm 3. The implementation of $merge(C_0, C_1)$ is very similar to $diff(N_0, N_1)$ in Algorithm 2, where we would substitute recursive calls to $diff(N_0, N_1)$ with calls to $merge(C_0, C_1)$ that computes weighted average of input nodes.

Two values are configurable in Algorithm 3. $Ratio_{min}$ is a threshold such that all cluster pairs (C_0, C_1) satisfying $diffRatio(C_0, C_1) < Ratio_{min}$ would always be merged, which avoids dividing similar instances into different clusters. $Ratio_{rel}$ instructs the algorithm to merge cluster pairs when their *diffRatio* is insignificant compared to other pairs, which serves to keep only the clusters with significant variances. Through several experiments, we set $Ratio_{min}$ = 0.02 and $Ratio_{rel}$ = 0.25.

Clusters generated by our algorithm serve as summaries of the corresponding loop or a parallel program. When behavior is regular, time series data for a loop or an entire parallel program can be replaced with a few iteration or thread clusters that emphasize the most distinct equivalent classes.

4.2.1 Multi-level Clustering. Our clustering algorithm supports multi-level clustering and is applied in a bottom-up fashion: inner loops are clustered before outer loops; loops in individual threads are processed before the entire execution.

To extend Algorithm 3 to support multi-level clustering, we implement $diff(S_0, S_1)$, which can be called at line 13 and 24 in Algorithm 3, to compute the difference in run time between two sets of clusters that are originally loops. We define $diff(S_0, S_1)$ to be the sum of differences in execution time across every matching pair of iterations. In our implementation of $diff(S_0, S_1)$, we compute the difference in a few groups by utilizing clustering result.

4.2.2 Parallelization of Clustering. Clustering in parallel is necessary to uncover performance insights within a reasonable amount of time. Currently, we employ a shared memory parallelization.

Our framework begins by processing the temporal context tree of every thread to cluster all loops. We parallelize this step by assigning each analysis process a set of thread trees. Next, our framework clusters the entire execution. We employ two kinds of parallelism in this step: (1) task parallelism within the divide-and-conquer implementation of *getClusterSet* in Algorithm 3, and (2) data parallelism when computing pairwise *diffRatio* at line 13 and 24 in Algorithm 3.

4.3 Identifying Performance Losses

Variation across threads and iterations provides clues to performance loss. In this section, we describe how we quantify and attribute variations across threads to locate potential performance losses.

4.3.1 Quantifying Imbalance and Waiting. To optimize a serial program, one typically identifies functions and loops that take a significant amount of time and optimizes them to improve the overall performance. For parallel programs, identifying work imbalance and unnecessary waiting is typically the first step.

We compute two metrics to quantify performance loss from work imbalance and unnecessary waiting respectively. We define $imb(N)$ as the projected run time reduction if work imbalance in node N is fixed; and $wait(N)$ as the projected run time reduction if balanced wait in node N is eliminated. To compute these two metrics, we attach three extra metrics to every node in the temporal context tree. We define $avg(N)$ as the average run time of node N across all processes and threads. Similarly, we define $min(N)$ and $max(N)$ as the minimum and maximum run time of N. These three metrics can be easily calculated during clustering.

We divide nodes in the temporal context tree into three semantic categories – computation, waiting, and synchronization – based on several heuristics. User-level functions and loops are considered as computation; functions involving synchronization, such as *MPI_Barrier* and the OpenMP barrier pragma, are classified as synchronization; functions involving waiting, such as *MPI_Send* and *omp_set_lock*, are treated as waiting.

We compute $imb(N)$ and $wait(N)$ for a node according to its semantic category, as shown in Table 1. Explanation for formulas in

Table 1: Computing imb(N) and wait(N). imb(N) quantifies performance loss due to load imbalance while wait(N) quantifies performance loss due to waiting.

Node category	imb(N)	wait(N)
Computation (C)	$max(C) - avg(C)$	0
Wait (W)	$max(W) - avg(W)$	$avg(W)$
Synchronization (S)	$avg(S) - min(S)$	$min(S)$

Figure 9: Example estimation of performance improvement. Time series data of P0 – P2 are visualized in the same way as Figure 7. Run time of each region is tagged. Assuming X as computation, Y and Z as synchronization, the bottom bar shows the time series data of all processes if load-imbalance in X is fixed, leading to an improvement of 4s.

Table 1 are as follows. Assume the absence of region Y and Z in Figure 9, if region X is computation, the execution time will be reduced from $max(X)$ to $avg(X)$ by balancing the work in X. Therefore, we have $imb(X) = max(X) - avg(X)$ and $wait(X) = 0$. Similarly, if region X is waiting, the execution time will be reduced by $max(X)$ if waiting in X is fully eliminated. The reduction has two sources: $max(X) - avg(X)$ from balancing the wait and $avg(X)$ from eliminating the balanced wait. $imb(S)$ of a synchronization node is defined in a slightly different way, as the projected run time reduction if imbalance between the previous synchronization node and the current one is fixed. In Figure 9, if region X is computation and region Y is synchronization, we have $imb(Y) = imb(X) = max(X) - avg(X) = (length(X+Y) - min(Y)) - (length(X+Y) - avg(Y)) = avg(Y) - min(Y)$. After X is balanced, eliminating waiting in Y would bring an extra run time reduction of $min(Y)$; therefore, we have $wait(Y) = min(Y)$.

4.3.2 Attributing Performance Loss to Significant Call Paths. In this section, we attribute performance losses to call paths. We identify call path depths where losses arise and pick significant call paths that are major contributors to the overall inefficiency.

We define $sumImb(N)$ as the sum of imbalance in all children of a temporal context tree node N, which is computed as follows:

$$sumImb(N) = \begin{cases} imb(N) & child(N) = \emptyset \\ \sum_{M \in child(N)} sumImb(M) & child(N) \neq \emptyset \end{cases}$$

Figure 10 shows an example on the value of $imb(N)$ and $sumImb(N)$ in a temporal context tree.

With $imb(N)$ and $sumImb(N)$, we are able to identify call path depths where imbalance arise and pick call paths that are significant contributors to the overall imbalance. Figure 11 shows an example

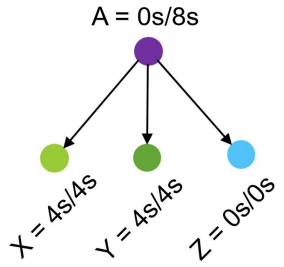


Figure 10: Example of imbalance metrics in the tree assuming that X, Y, and Z in Figure 9 are computation nodes called by A. Two values are shown for each node. The first value is imb(N) and the second value is sumImb(N).

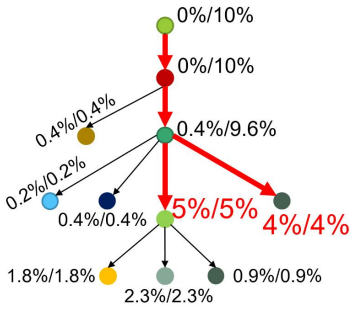


Figure 11: Example of imbalance attribution. Imbalance is shown in percentiles with respect to the total execution time. Two percentiles are shown for each node. The first one is imb(N) and the second one is sumImb(N). Out of all call paths, two are highlighted with appropriate depth selected.

on highlighting major sources of imbalance. All highlighted call paths ending with node N must meet the following two criteria:

- $imb(N)/RunTime > significanceRatio$ so that the imbalance is significant compared to execution time;
- $imb(N)/sumImb(N) > originDepthRatio$ such that node N can explain imbalance originated from its subtree.

significanceRatio prunes call paths with little imbalance. *originDepthRatio* helps select call path depths where imbalance arise and avoid reporting too many children of N with diluted imbalance. In our experiments, lots of assignments for the two parameters work well and we currently pick *significanceRatio* = 0.1% and *originDepthRatio* = 70%. Note that while *significanceRatio* is small, the sum of several call paths in an inefficient segment (discussed in the next section) can become significant.

To highlight sources of waiting, our tool employs a similar strategy based on the approach described above. We compute $wait(N)$ and $sumWait(N)$ and use *significanceRatio* and *originDepthRatio* to select call paths that are significant contributors to wait time. In this way, our framework highlights call paths that are significant sources of performance loss and identifies call path depths where losses arise.

Table 2: Performance Diagnosis of Inefficient Execution Segments

	imb(S)	∑imb(N)	∑wait(N)	Explanations and further investigation steps
#1	Low	Low	High	Lots of waiting. Investigate contributers of wait to identify unnecessary waiting.
#2	Low	High	Low	Locate contributers of imbalance. Make sure they have correct semantic classifications and their imbalances trade off when added up. Good performance if so; otherwise, significant imbalance.
#3	High	High	Low	Load imbalance. Locate causes of the imbalance by investigating contributers of imbalance.
#4	Low	High	High	First, follow steps in the 2nd row. If imbalance in computation call paths trade off with imbalance in waiting, investigate if those computation call paths are the causing the wait. If not, follow steps in the 1st row.
#5	High	High	High	Can be a mix of the above performance inefficiencies. Follow steps in the 1st and 3rd row first. If no clues are found, follow steps in the 4th row.

4.3.3 Dividing Execution into Inefficient Segments. Execution of a parallel program can be divided into phases between synchronizations. Similarly, our automated framework divides an execution into inefficient segments based on significant call paths. Each segment contains multiple significant call paths and ends in synchronization. For example, the execution in Figure 9 would be divided into two segments of `(X,Y)` and `(Z)`, assuming X as computation and Y and Z as synchronization.

For each inefficient segment, we can diagnose its performance according to Table 2. In the table, we have $imb(S)$, imbalance of the synchronization call path; we sum up $imb(N)$ of all significant call paths except $imb(S)$; and we also sum up $wait(N)$ of these call paths. As an example, for Figure 9 where `X` is computation while `Y` is synchronization, the user should follow instructions in the 3rd row of the diagnosis table. Under another scenario, assuming `X` and `Y` are computation while `Z` is synchronization, the user would follow steps in the 2nd row.

To summarize, our clustering algorithm divides threads and iterations into a few equivalence classes to generate compact summaries. Furthermore, our framework computes metrics to quantify performance losses, which it attributes to call paths, and identifies call path depths where losses arise.

5 EVALUATION

We evaluate the capabilities of our analysis framework by applying it to executions of two scientific codes on the Titan supercomputer at ORNL (Table 3). For both executions, we use HPCToolkit to collect a series of call path samples for each process in the execution. We then employ our automated framework to analyze the sequence of call path samples collected. We present analysis results using a version of HPCToolkit's hpctraceviewer modified to render novel views that simultaneously highlight multiple bottlenecks that arise at different call path depths.

5.1 Case Study: Analysis of PFLOTRAN

PFLOTRAN [14] is a code for massively parallel simulation of subsurface flow and reactive transport. We study the `100_10_10` example problem on 512 cores using 32 nodes with 16 cores per node.

Table 4 shows our iteration identification results for the top three loops (ranked by loop duration) in PFLOTRAN. We divide the average loop duration across all processes by execution time to produce the percentiles in the second column. The first loop covers phase #5 and #6 in Figure 1 – iterative behaviors are seen in phase #5 while phase #6 belongs to the last iteration of the loop. With plenty of time-series data for each iteration, our framework is able to identify and accept 23 identifiers, which matches the number of actual iterations. Our framework also yields the correct result for two instances of the loop at line 3 in Algorithm 4, which is discussed later in this section.

Table 3: Titan HW and SW Configurations

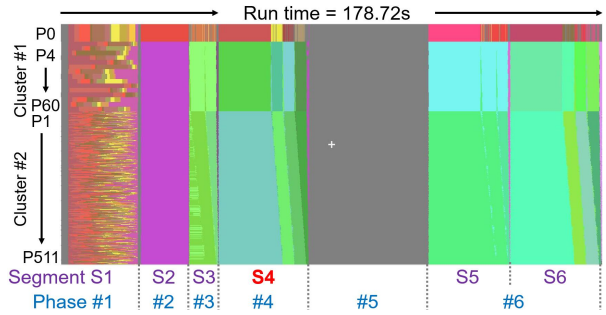
Core	AMD Opteron Processor 6274
Clock frequency	2.2GHz
# sockets per node	2
# cores per socket	8
# threads per core	1
Operating system	Linux 3.0.101
Compiler	gcc 4.9.3
MPI implementation	cray-mpich 7.5.2

Table 4: Iteration Identification Result for Selected Loops in PFLOTRAN.

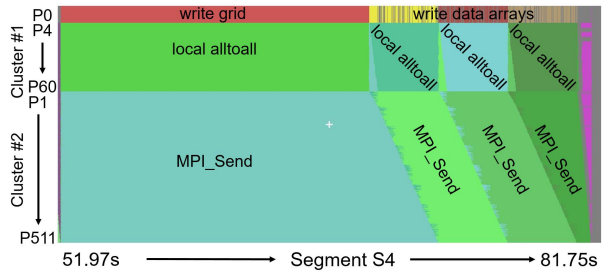
Loop	Average Duration	# Accepted Identifiers	# Rejected Identifiers	# Actual Iterations
Phase #5 and #6 in Figure 1	51.8%	23	0	23
Line 3 in Algorithm 4	15.4%	3	0	3
Line 3 in Algorithm 4	14.7%	3	0	3

Table 5: Six Inefficient Execution Segments for PFLOTRAN.

	imb(S)	∑imb(N)	∑wait(N)	Phase of Figure 1
S1	2.01%	61.6%	0%	#1
S2	8.80%	8.80%	0%	#2
S3	0.20%	11.3%	5.20%	#3
S4	0.13%	33.4%	16.6%	#4
S5	0.20%	30.3%	14.8%	first half of #6
S6	0.20%	33.3%	16.6%	second half of #6



(a) Annotated screenshot showing the full execution of PFLOTRAN. Segment (as of Table 5) and phase (as of Figure 1) numbers are noted at the bottom.



(b) Expanded and annotated view of segment S4 in (a).

Figure 12: Annotated screenshots from our revised version of HPCToolkit's hpctraceviewer rendering automated summaries of the same PFLOTRAN run shown in Figure 1. Processes are arranged first by clusters and then by MPI rank. Each pixel in a horizontal trace line for a process shows the call path of the process at that time. The depth of each call path displayed is selected based on the automatic summaries we compute. Significant call paths are assigned colors according to their semantic category: computation in red, yellow, or brown, wait in green or blue, and synchronization in purple. Insignificant call paths are colored in gray. Functionality of call paths are labeled where possible in (b).[1]

Our framework identifies six inefficient segments in the PFLOTRAN execution as shown in Table 5. We divide the actual metric values by total execution time to produce the percentiles in the table. Besides the metrics, relationships between these segments and phases in Figure 1 are specified. Applying our clustering algorithm to time series data from 512 processes yields 3 clusters:

- cluster #0 contains 1 process: P0;
- cluster #1 contains 15 processes: P4, P8, P12, ..., P60;
- cluster #2 contains the remaining 496 processes.

Figure 12a shows the reorganized time series data of all processes for the same PFLOTRAN execution as in Figure 1. The revised hpctraceviewer arranges processes according to the automated clustering result. Processes are ordered first by clusters and then by MPI rank. The height of each cluster is proportional to $\log_2(size+1)$ so that processes with distinct behaviors stand out. At each pixel, instead of showing a procedure frame on the call path at some depth manually selected by the user, the revised hpctraceviewer selects the depth based on automated summaries. Significant call paths are assigned colors according to their semantic category: computation in red, yellow, or brown, wait in green or blue, and synchronization in purple. Insignificant call paths are colored in gray. This coloring scheme directs attention to performance issues worth understanding and highlights imbalances and waiting that cause inefficiencies.

Algorithm 4: Sketch of PFLOTRAN serialized I/O.

```
Data: Each process has a total of three local arrays, donated as A[0],
      A[1], and A[2].
1  if I'm P0 then
2      Write the global grid to visualization file ;
3  for k ∈ {0, 1, 2} do
4      MPI_Alltoall A[k] within local MPI group, turning A[k] from a
       3D local partition to several contiguous rows in the grid;
5      if I'm not P0 then
6          MPI_Send A[k] to P0;
7      else
8          Write A[k] to visualization file;
9          for i ∈ {1, . . . , numProc − 1} do
10             MPI_Recv temp from Pi;
11             Write temp to visualization file;
12 MPI_Barrier across all processes;
```

Figure 12b shows an expanded view of S4 from Figure 12a. The view is manually annotated with significant call frames.[1] Serialization is obvious in the figure. In cluster #2, processes execute an *MPI_Send* and retire in order as P0 receives their data and writes a visualization file. This highlights that having P0 perform all I/O is a bottleneck.

Significant call paths in S4 identify less than ten functions of interest out of 300K lines of code. Our investigation into the functions highlighted by our automated analysis revealed the nature of the I/O serialization. Algorithm 4 sketches how PFLOTRAN performs I/O. The computation is serialized at line 10, where P0 receives data from other processes in order. P0 is responsible for all file writes at line 2, 8, and 11, which is the root cause of waiting by other processes. Processes in cluster #1 are in the same local MPI group as P0; therefore, they spend more time waiting at line 4 whereas processes in cluster #2 wait at line 6.

Our investigation into S5 and S6 in Table 5 shows they are explained by the same piece of code, except that the code is called from different contexts and writes distinct visualization files. S2 and S3 can also be explained by Algorithm 4, where a call to *MPI_Barrier* takes place between line 2 and 3, cutting the code into two halves. The first half explains S2 while the second half explains S3. Investigation into S1 reveals work imbalance when processes write an output data file in parallel. Multiple call paths contributed to the total imbalance. Work in every call path is imbalanced but they are almost balanced when added up. That's why we see a big total imbalance but a modest $imb(S)$ in S1.

[1]Each manual annotation highlights the key frame from call paths associated with a region. The tool provides the complete call path associated with each pixel in the visualization in another pane that is not shown.

Original Time (Serialized I/O)	Estimated Time (No Serialization)	Optimized Time (Parallel I/O)
178s	66s	70s

Figure 13: Comparison of PFLOTRAN execution times.

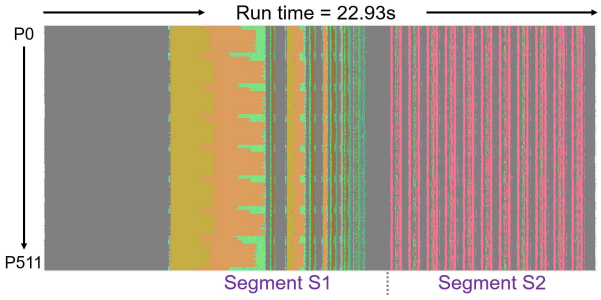


Figure 14: Annotated screenshot from hpctraceviewer examining automated summaries on AMG2013. The view is organized in the same way as Figure 12 except that processes are solely arranged by MPI rank.

Table 6: Two Inefficient Execution Segments for AMG2013.

	imb(S)	∑imb(N)	∑wait(N)
S1	0.11%	19.6%	5.40%
S2	0.24%	2.01%	1.54%

Table 7: Iteration Identification Result for Selected Loops in AMG2013.

Loop	Average Duration	# Accepted Identifiers	# Rejected Identifiers	# Actual Iterations
Loop in S1	48.6%	8	1	9
Loop in S2	32.2%	11	0	11

If the serialization in Algorithm 4 is eliminated and the work on writing files is evenly distributed, imb(S) and total wait of S2 – S6 would disappear, suggesting a 63% reduction of execution time. We expect an extra 2% reduction if work in S1 can be balanced.

Based on these insights, we replaced the serialized writes shown in Algorithm 4 with MPI parallel I/O. As shown in Figure 13, the optimized code is 61% faster than the original. Performance of optimized code is consistent with the estimation: the extra 4*s* corresponds to the overhead of using MPI parallel I/O and *MPI_Scan* to calculate offsets for file writes.

5.2 Case Study: Analysis of AMG2013

Algebraic MultiGrid (AMG) 2013 [15] is a parallel iterative solver of unstructured linear systems. We run the default solver on an 8×8×8 processor grid using 32 nodes with 16 cores per node.

Our framework identifies two inefficient segments in the AMG2013 execution as shown in Table 6. Figure 14 shows the visualization of the execution from our revised hpctraceviewer. The view is organized in the same way as the PFLOTRAN example in Figure 12 except that we choose to arrange processes by MPI rank to highlight the distribution of imbalance across processes.

One can easily identify load imbalance in segment S1 of Figure 14. Processes with fewer neighbors on the 8×8×8 processor grid have less work in call paths colored in gold and orange when the application coarsens the data grid. These processes spend more time waiting in MPI communications colored in green. By adding up the imbalance metric of the two computation call paths colored in gold and orange, we conclude that balancing the work could lead to a 3% performance improvement. An extra 2% is possible if certain waits in MPI communications of S1 can be eliminated. A quick examination of S2 shows the imbalance in S2 has little influence on the overall performance.

Table 7 highlights the iterative behaviors in S1 and S2. Our framework identifies 9 iterations for the loop in S1, where duration of each iteration decreases over time. The last iteration is rejected as its duration is smaller than twice the sampling period (less than 0.02% of execution time). Nonetheless, our framework is still able to apply further analysis to the first 8 iterations that are accepted. Iterations of loop in S2 have constant durations and our framework identifies and accepts all 11 iterations.

Our automated framework groups processes into four clusters:

- cluster #0 has 122 processes that are the first and last 8 processes of every group of 64 processes (but with 6 exceptions);
- cluster #1 has 384 processes that are the middle 48 processes of every group of 64 processes;
- cluster #2 contains 4 processes: P56, P120, P184, and P455;
- cluster #3 contains 2 processes: P327 and P391.

Processes in cluster #2 and #3 are separated from ones in #0 as they spent more time waiting in *PMPI_Recv* for messages from neighbors rather than a combination of some other MPI communications that function as global synchronization in S1. Such a difference in the distribution of waiting time doesn't challenge our conclusion of load imbalance in S1.

With all these insights, one may decide to optimize AMG2013 to achieve estimated performance improvement or choose to verify the effect of the imbalance in S1 on other input problems before further actions. After a quick investigation, we concluded that the small potential improvement doesn't worth the effort to gather more detailed information about the bottleneck and implement optimizations to address the load imbalance.

With PFLOTRAN and AMG2013 as examples, we demonstrate how users can employ our analysis framework to quickly locate potential performance bottlenecks and reason about causes of performance losses. We demonstrated the utility of our findings and the accuracy of our tool's estimates of possible improvement by fixing the serialization bottleneck in PFLOTRAN to achieve considerable performance improvement. Manually exploring the time series data for these codes to understand the causes of performance losses using the original version of HPCToolkit's hpctraceviewer requires considerably significantly more effort than our automated analysis.

6 CONCLUSIONS AND FUTURE WORK

This paper describes an automated analysis framework that generates compact summaries of a time series of asynchronous call path samples. These summaries highlight patterns in the data and can quickly direct a user's attention to potential performance bottlenecks. Case studies of PFLOTRAN and AMG2013 executions demonstrate the effectiveness of our approach.

We are pursuing several extensions to our work on automated time series data analysis. First, we aim to further revise hpctraceviewer for more effective exploration of automated summaries. We envision additional views that (1) highlight inefficient execution segments and their associated metrics and (2) visualize summarized iterative behaviors over time. Second, we want to exploit semantic information in our analysis. Our framework only works on SPMD programs where processes run the same code. To extend our analysis to MPMD programs, semantic information is required to build a map between processes running different code. Semantic information can also help our framework derive more accurate diagnosis. Third, we want to automatically derive causes for performance losses highlighted in the summary. By matching behaviors of inefficient execution segments with certain patterns, we may be able to determine the nature of performance loss, such as load-imbalance and serialization, as well as provide suggestions on optimizing the code.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No.1450273. Access to Titan at Oak Ridge National Laboratory was provided through a 2017 DOE INCITE award.

REFERENCES

[1] L. Adhianto et al. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 685–701. https://doi.org/10.1002/cpe.v22:6

[2] Xavier Aguilar, Karl Fürlinger, and Erwin Laure. 2015. Automatic On-Line Detection of MPI Application Structure with Event Flow Graphs. In *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 70–81. https://doi.org/10.1007/978-3-662-48096-0_6

[3] Allinea. 2017. Allinea Forge User's Guide. (2017). http://content.allinea.com/downloads/userguide-forge.pdf Accessed Apr 25, 2018.

[4] Matthew Arnold and Peter F. Sweeney. 2000. *Approximating the Calling Context Tree via Sampling*. Technical Report. IBM.

[5] Amir Bahmani and Frank Mueller. 2015. ACURDION: An Adaptive Clustering-based Algorithm for Tracing Large-scale MPI Applications. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data) (BIG DATA '15)*. IEEE Computer Society, Washington, DC, USA, 785–792. https://doi.org/10.1109/BigData.2015.7363823

[6] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. 2016. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. *ACM Trans. Parallel Comput.* 3, 2, Article 11 (July 2016), 24 pages. https://doi.org/10.1145/2934661

[7] Marc Casas, Rosa M. Badia, and Jesús Labarta. 2010. Automatic Phase Detection and Structure Extraction of MPI Applications. *Int. J. High Perform. Comput. Appl.* 24, 3 (Aug. 2010), 335–360. https://doi.org/10.1177/1094342009360039

[8] Computer Sciences Department, University of Wisconsin-Madison and Computer Science Department, University of Maryland. 2016. ParseAPI Programmer's Guide. (June 2016). http://www.dyninst.org/sites/default/files/manuals/dyninst/parseAPI.pdf Accessed Apr 25, 2018.

[9] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 184–197. https://doi.org/10.1145/2815400.2815409

[10] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. 2005. Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 81–90. https://doi.org/10.1145/1088149.1088161

[11] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca Performance Toolset Architecture. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 702–719. https://doi.org/10.1002/cpe.v22:6

[12] J. Gonzalez, J. Gimenez, and J. Labarta. 2009. Automatic Detection of Parallel Applications Computation Phases. In *2009 IEEE International Symposium on Parallel Distributed Processing*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/IPDPS.2009.5161027

[13] J. Gonzalez, K. Huck, J. Gimenez, and J. Labarta. 2012. Automatic Refinement of Parallel Applications Structure Detection. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. IEEE Computer Society, Washington, DC, USA, 1680–1687. https://doi.org/10.1109/IPDPSW.2012.209

[14] G. E. Hammond, P. C. Lichtner, C. Lu, and Mills R.T. 2012. PFLOTRAN: Reactive Flow and Transport Code for Use on Laptops to Leadership-Class Supercomputers. In *Groundwater Reactive Transport Models*, Fan Zhang, G.T. Yeh, and Jack C. Parker (Eds.). Bentham Science Publishers, Sharjah, UAE, 141–159. https://doi.org/10.2174/97816080530631120101

[15] Van Emden Henson and Ulrike Meier Yang. 2002. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Appl. Numer. Math.* 41, 1 (April 2002), 155–177. https://doi.org/10.1016/S0168-9274(01)00115-5

[16] Kevin A. Huck and Allen D. Malony. 2005. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, Washington, DC, USA, 41–52. https://doi.org/10.1109/SC.2005.55

[17] Intel. 2017. Intel VTune Amplifier 2018 Help. (2017). https://software.intel.com/en-us/vtune-amplifier-help Accessed Apr 25, 2018.

[18] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. 2008. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 139–155. https://doi.org/10.1007/978-3-540-68564-7_9

[19] G. Llort, J. Gonzalez, H. Servat, J. Gimenez, and J. Labarta. 2010. On-Line Detection of Large-Scale Parallel Application's Structure. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/IPDPS.2010.5470350

[20] Vincent Pillet, Toni Cortes, Vincent Pillet, Jesus Labarta, Toni Cortes, and Sergi Girona. 1995. Paraver: a Tool to Visualize and Analyze Parallel Code. In *Transputer and Occam Developments*, Vol. 44. IOS Press, Clifton, VA, USA, 17–31. Issue 1.

[21] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311. https://doi.org/10.1177/1094342006064482

[22] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.47

[23] Nathan R. Tallent, Darren J. Kerbyson, and Adolfy Hoisie. 2017. Representative Paths Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 34, 12 pages. https://doi.org/10.1145/3126908.3126962