Using Sample-Based Time Series Data for Automated Diagnosis of Scalability Losses in Parallel Programs

Lai Wei*
lai.wei@rice.edu
Department of Computer Science
Rice University
Houston, Texas, USA

John Mellor-Crummey†
johnmc@rice.edu
Department of Computer Science
Rice University
Houston, Texas, USA

Abstract

The performance of many parallel applications has failed to scale as fast as successive generations of hardware on which these applications execute. To understand the cause of scalability losses, experts use performance tools to monitor and analyze application behavior. Profiles generated by performance tools can usually indicate the presence of scalability losses while time series data are generally necessary to pinpoint the root causes of such losses. However, manual analysis of time series data can be difficult in executions with a large number of processes, long running times, and deep call chains. This paper describes an automated framework that analyzes sample-based time series data to diagnose scalability losses in parallel executions. The framework's automated diagnosis of scalability losses indicates their symptoms, severity, and causes. Two case studies illustrate the effectiveness of this framework. When compared to a tool that analyzes performance using instrumentation-based traces, our overhead for collecting sample-based time series is 1/28 in time and 1/1600 in space while our automated analysis takes 1/25 of the time.

CCS Concepts • **General and reference → Performance**.

Keywords Performance, automated diagnosis, scalability losses, sample-based time series data

*This work were done while the author was a Ph.D. student at Rice University. The author's present affiliation is Pony.ai, Fremont, CA, USA.
†Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374538

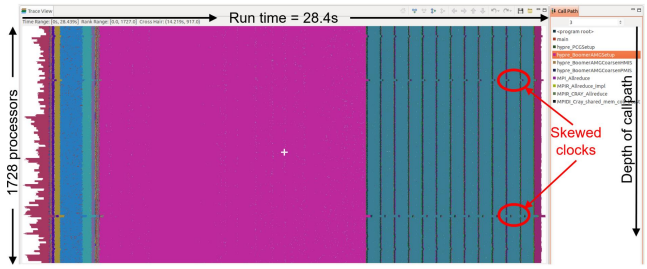

Figure 1. Annotated screenshot of HPCToolkit's hpctraceviewer examining a 1728-process execution of AMG2013 on Titan at Oak Ridge National Laboratory.

1 Introduction

Each generation of supercomputers is more powerful than the last in an attempt to keep up with the growing ambition of scientific inquiry. Despite improvements in computational power, however, the performance of many parallel applications has failed to scale.

Many factors degrade the parallel performance of applications. To help application developers understand causes of scalability and performance losses, people have developed a collection of tools to measure and analyze application performance. Commonly used performance tools include HPCToolkit [1], MAP [2], Paraver [33], Scalasca [16], TAU [37], Vampir [26], and VTune [23]. These tools monitor parallel executions to collect measurements, employ post-mortem analysis of measurement data, and provide visualizations for exploring measurements and analysis results. Experts use such tools to manually identify causes of performance losses.

While profiles generated by performance tools can usually identify the presence of scalability losses, time series data are generally necessary to pinpoint the root causes of such losses. Figure 1 shows an annotated screenshot from HPCToolkit's hpctraceviewer that examines time series data from a 1728-process execution of AMG2013 on Titan at ORNL. In the figure, processes are arranged along the vertical axis and time flows left to right. Each pixel in a horizontal line for a process shows a procedure frame on the call path of the process at that time. Each procedure has a unique color. The depth of the frame in the call path can be selected on the

right. The red circles highlight skewed clocks on some MPI ranks.

Manually analyzing performance of the parallel execution in Figure 1 is challenging. Vertically, the number of pixels is smaller than the number of processes; as a result, not all processes are shown. Horizontally, the execution apparently consists of several phases and the series of vertical bands near the end suggests iterative behavior; each of these phases needs careful examination. One would need to zoom in to address challenges in both dimensions, and at the same time, select the appropriate call path depths in the pane on the right to improve insight. On some machines, including ORNL's Titan, skewed clocks only complicate the situation.

In many cases, performance issues only appear at large scales. Automated analysis is necessary to understand the performance of large scale parallel executions. Prior tools [5, 16, 31] focus on collection and analysis of instrumentation-based traces. However, recording instrumentation-based traces has much higher overhead in time and space than recording sample-based time series data with HPCToolkit. As described in more detail in Section 6.3, two experiments to compare Scalaca's instrumentation-based tracing with HPCToolkit's collection of sample-based time series show that Scalasca's measurement overhead was more than 28× larger than HPCToolkit and the size of its traces was more than 1600× larger than HPCToolkit's sample-based time series. The space and time overhead of instrumentation-based approaches can make them problematic to apply at large scales to long-running applications.

In this paper, we describe an automated framework that analyzes sample-based time series data to diagnose scalability losses in parallel executions. Collecting and analyzing sample-based time series data incurs low overhead in time and space, making it appropriate for large scale parallel executions. We highlight four contributions of our work.

- We develop a new strategy for correcting clock skews with sample-based time series data to support robust identification of execution phases.
- We use a taxonomy of scalability issues to drive analysis of symptoms of scalability losses, identify their root causes, and rate their severity.
- Our analysis is applicable to both SPMD and MPMD codes.
- Our framework employs distributed-memory parallelism to accelerate the analysis and diagnosis of performance problems in large scale executions.

The rest of this paper is organized as follows. Section 2 describes prior work on analysis of parallel program performance and handling clock skews. Section 3 describes background for this work. Section 4 describes how we fix skewed clocks and identify execution phases in time series data. Section 5 describes how our framework generates diagnoses related to scalability issues. Section 6 illustrates the effectiveness of our automated framework by applying it to two parallel executions. Section 7 summaries our work and future directions.

2 Related Work

Many performance tools support collection and visualization of performance data for parallel executions, including HPCToolkit [1], Paraver [33], Scalasca [16], TAU [37], Vampir [26], and VTune [23]. These tools enable users to manually analyze measurement data to identify performance issues; however, support for automated analysis is limited.

Some work focuses on automated analysis of profiles. In HPCToolkit, Tallent et al. [38] analyze profiles of parallel executions to locate load imbalance. More recent work [39] improves on this approach by comparing differences between profiles of representative paths to identify load imbalances. TAU's PerfExplorer [22] enables users to build thread clusters according to metric profiles that consist of the time spent in each function. Users can analyze the min, max, and average profiles of each cluster to identify potential performance issues. All of these approaches focus on analysis of profiles; however, analyzing profiles alone is generally insufficient to pinpoint the root causes of performance issues.

Other work analyzes instrumentation-based traces automatically. In Paraver, Gonzalez et al. [17, 18, 30] use traces of MPI communications to divide parallel executions into execution regions. These regions are then clustered according to their metrics, such as instructions per cycle and cache misses, to highlight the outliers. Users need to manually analyze outliers to identify any potential performance problems. Scalasca [16] supports automated trace analysis. Boehme et al. [5] attribute wait states in parallel executions to their root causes by analyzing traces. Unfortunately, their approach only applies to short execution intervals due to the time and space cost associated with instrumentation-based traces. Users would need to analyze coarse performance data first before utilizing their automated analysis of an execution interval. In Paradyn, Miller et al. [31] implement dynamic instrumentation to measure application performance and employ the W^3 search model to automate the search for performance problems. However, they offer no study on the cost of dynamic instrumentation against the accuracy of their performance diagnosis. We believe that a good diagnosis would still require a much higher overhead compared to sampling.

In our prior work [43], we developed extensions to HPCToolkit to analyze the sample-based time series it collects for parallel program executions to identify potential indicators of scalability losses. This prior work provides the foundation for the work described in this paper. We review this work in detail in Section 3.

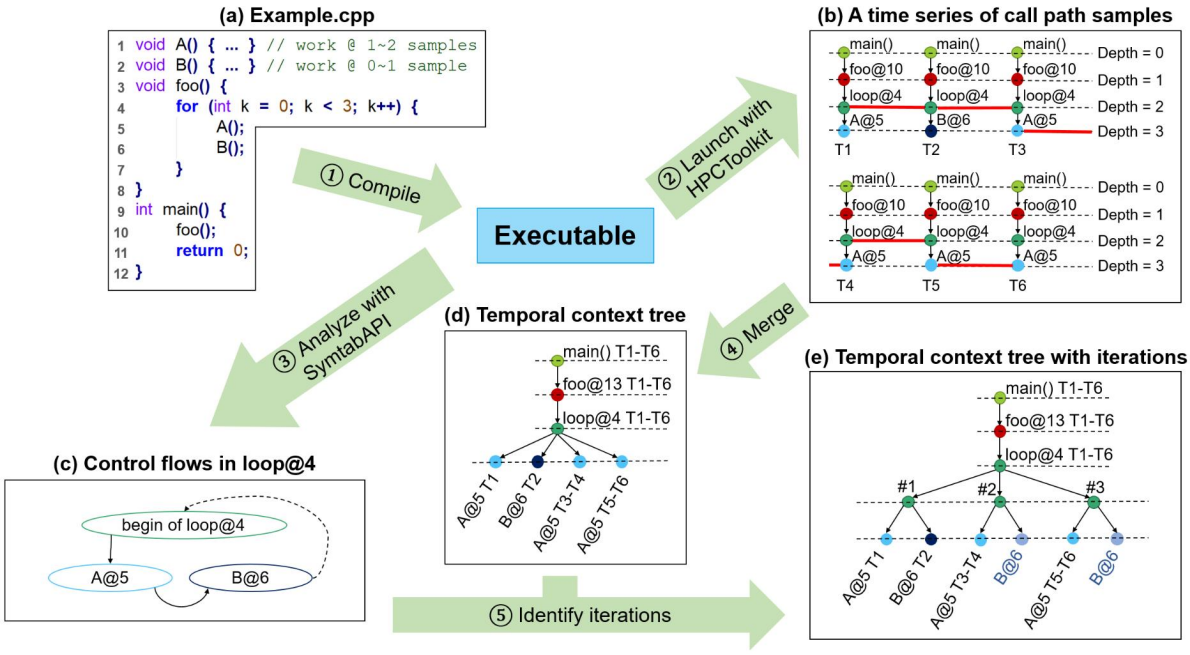

Figure 2. How we collect and digest time series data in our prior work [43].

Curtsinger et al. [10] employ *causal profiling* to assess performance. In their work, they evaluate the potential benefit of speeding up a source line through virtual experiments that slows down the execution of other code. The cost of virtual experiments is proportional to both the original execution cost and the number of source lines evaluated. As a result, this approach is only feasible on small scale parallel runs and users need to supply a short list of functions or source lines for evaluation.

The PerfExpert performance diagnosis tool [7] assesses node-level performance bottlenecks. PerfExpert uses performance counter measurements and system parameters to assess performance of functions and loops over six categories, including data accesses, branch instructions, floating-point operations, etc. Good node-level performance is an essential on parallel architectures. Our work focuses on scalability issues that are generally more important to the performance of large-scale runs.

Prior work on clock synchronization is related to our work in Section 4. We briefly describe a few lines of that work here.

The Network Time Protocol [32] is widely used to synchronize clocks in computer systems over the network; however, its accuracy is not enough for performance measurement. To achieve better precision, Jones and Koenig [24] propose an algorithm that leverages MPI collectives and synchronizations to synchronize clocks at runtime. They use MPI's PMPI wrapping interface to intercept MPI methods and augment them to synchronize clocks and provide the synchronized time.

Instead of synchronizing clocks at runtime, it is also feasible to revise timestamps in post-mortem analysis. The controlled logical clock [34] algorithm corrects the timestamp of messages whose send time is later than receive time. An extension to the algorithm [3] can correct timestamps in parallel based on point-to-point messages and collective operations. Both approaches need complete traces of messages.

3 Background

In prior work, we extended HPCToolkit [1] to analyze time series of asynchronous call path samples to highlight call paths that are potential indicators of performance loss [43]. Compared to methods that digest instrumentation-based traces, this approach to collecting and analyzing sample-based time series data incurs lower overhead in time and space, which makes it feasible for large scale parallel executions.

In this prior work, we use HPCToolkit to collect time series data and process the data to form a *temporal context tree* for each thread, as shown in Figure 2.

In step #2, the executable compiled from the example code in step #1 is launched with HPCToolkit. HPCToolkit monitors the execution and collects six asynchronous call path samples, as shown in Figure 2b. Each sample has three components – a timestamp, a call path, and an LCA (Least Common Ancestor). A timestamp indicates when a sample is taken. Timestamps `T1-T6` in Figure 2b are associated with the samples above them. A call path consists of a sequence of procedure frames on the thread's stack when the sample was recorded. Each frame is tagged with the function name and line number from which the function is called. Besides

procedure frames, HPCToolkit augments call paths with loop contexts when calls to functions are made inside loops. Each loop is tagged with line numbers in Figure 2b. An LCA of a sample represents the lowest procedure frame that remained on the call stack since the previous sample. Red lines indicate LCAs of neighboring samples in Figure 2b. Notice that `loop@4` is the LCA between samples at `T4` and `T5`. `A@5` in `T4` is called from the second iteration of `loop@4`. After timestamp `T4`, `A@5` is popped from the stack and `B@6` is pushed. No sample is taken in `B@6` and it is popped from the stack at the end of the second iteration. The third (last) iteration pushes a new instance of `A@5` onto the stack and a sample is taken at `T5`. As a result, `loop@4` is the lowest frame that remains on the call stack between `T4` and `T5`.

In step #3, we use Dyninst's ParseAPI [8] to compute control flow graphs (CFGs) for procedures and loops. Figure 2c shows the control flows inside `loop@4`. In the figure, concrete lines represent forward control flows while dashed lines represent back edges in loops. Those back edges indicate start of new iterations.

In step #4, call path samples in Figure 2b are merged to a temporal context tree in Figure 2d. In this process, common prefixes of temporally-adjacent call path samples are merged. Notice that `A@5 T3–T4` is not merged with `A@5 T5–T6` as `A@5` is not a common prefix of samples at `T4` and `T5`.

In step #5, an iteration identification algorithm parses loops in the temporal context tree according to CFGs and inserts a layer of iterations between each loop and its children. New iterations are introduced whenever back edges in CFGs are taken. In the example, the back edge in `loop@4` is taken in the middle of `T2` and `T3` as well as between `T4` and `T5`. The result of iteration identification is shown in Figure 2e. Notice that while no sample is taken in `B@6` of iterations #2 and #3, the algorithm infers their existence based on the CFG.

Subsequently, we cluster threads and loop iterations to highlight their differences, which are potential indicators of performance losses. We attribute those differences to call paths and render a display that highlights them to help users diagnose performance issues.

While our prior work can identify similarities and differences between threads and over time, it lacks the ability to analyze the nature of such differences and provide comprehensive diagnosis of performance losses. In addition, its analysis has limited parallelism and is not applicable to large scale executions. These are the problems we set out to address in this work.

3.1 Digesting time series data in parallel

Our prior work [43] employs shared memory parallelization to digest time series data. While this was sufficient for analyzing modest-scale executions, the scale of typical shared-memory platforms limited the scale of time series data that could be analyzed in reasonable time. To accelerate analysis and enable analysis of larger-scale executions, the new analysis framework we describe in this paper is a scalable MPI application that digests time series data using distributed-memory parallelism, which is ubiquitous in clusters and supercomputers.

Users can launch our framework with `K` MPI ranks to digest the time series data of `N` threads or processes. Hereafter, we use the term threads rather than threads or processes for brevity. Our framework assigns each analysis rank the time series data of `N/K` threads. Each analysis rank builds a temporal context tree from the time series data of every assigned thread by following steps `#1–#5` in Figure 2. After the time series data of `N/K` threads is digested, each analysis rank builds a thread cluster from its assigned threads.

Next, our framework employs a binary tree to reduce thread clusters on all analysis ranks to the root analysis rank. Our framework takes advantage of Boost C++ Libraries [36] to serialize and deserialize thread clusters, making it feasible to exchange thread clusters across analysis ranks through MPI point-to-point communication.

As a result of this parallel data digestion, each analysis rank stores the temporal context trees of `N/K` threads. A cluster across all threads, which serves as a summary of the parallel execution, is available on the root analysis rank for further analysis in Section 5.

Besides digesting time series data in parallel, we also devised a series of optimizations to reduce the memory usage of temporal context trees. Those optimizations speed up our analysis and are critical for analyzing long executions. Memory optimization is out of the scope of this paper; details of memory optimizations can be found in Wei's dissertation [42].

4 Identifying Execution Phases

Here, we describe three extensions to our original strategy for analyzing time series data in HPCToolkit [43]. First, our framework generates comprehensive automated diagnoses that indicate the symptoms, severity, and causes of scalability losses. This greatly reduces the amount of manual effort that is needed to diagnose performance. Second, we parallelize our analysis using MPI so that it can employ a cluster or supercomputer to analyze time-series data from large-scale parallel executions collected on such systems. With this, our framework is able to analyze and diagnose performance issues in a 4096-process parallel execution in the case studies (Section 6). Third, our analysis is applicable to both SPMD and MPMD parallel codes based on MPI. Many scientific applications launch processes that run different pieces of code to finish a task. Adding support for MPMD applications enhances the applicability of our analysis framework.

Our automated analysis starts with the detection of execution phases. We define a *phase* as a segment in the parallel execution that begins with the exit of a global synchronization

and ends with the exit of a subsequent global synchronization. Under this definition, processes enter and exit a *phase* at the same time as they exit global synchronization at the same time.[1] Many parallel executions consist of several phases. For example, we observe several vertical lines that indicate the transition of phases in Figure 1. Identifying execution phases is necessary in our framework for two reasons: it helps our framework present users with performance diagnosis in accordance with the logical structure of the application, and it helps isolate performance losses in each phase, making it easier to associate symptoms of scalability losses in a phase to their root causes.

4.1 Fixing clock skews

To detect execution phases in time series data, we must cope with the fact that clocks in parallel systems are not always well synchronized. Variance in clocks can impair recognition of execution phases and other performance analysis efforts that rely on accurate time synchronization. From our perspective, fixing clock skews, such as the example shown in Figure 1, brings two benefits: it makes it possible to render more intuitive visualizations, and accurate timestamps enable us to infer global synchronizations, which are used to identify execution phases.

Times on different nodes may not be synchronized for two reasons. First, clocks can be skewed by a constant offset that persists over a long period of time. Second, small fluctuations in the drift of clocks can affect clock accuracy. Prior studies [3, 12] indicate that drift fluctuations can lead to variances of up to 20 μs, which would be significant in instrumentation-based traces. However, in sample-based time series data, 20 μs is much smaller than a typical sampling period of a few milliseconds. As a result, we only need to infer the skew offset to get adequately synchronized timestamps.

Fixing clock skews in sample-based time series data has its challenges. Prior work [3, 24, 34] only applies to full event traces that have a time stamp associated with each endpoint of a communication event. In sample-based time series data, we aren't guaranteed that all communication operations will be observed: a communication operation may start and finish between two samples. Furthermore, in our sample-based time series, one has no knowledge of the target rank of *MPI_Send* or the source rank of *MPI_Recv*, making it infeasible to utilize happened-before relationships to analyze message events. Using MPI collectives and synchronizations isn't easy either. In our time series data, we have no direct knowledge about the sets of threads or processes participating in such operations.

To avoid the space and time overhead associated with intercepting and tracing information about messaging events

[1]Processes don't exit global synchronization exactly at the same time but the variance is negligible compared to our typical sampling interval of a few milliseconds.

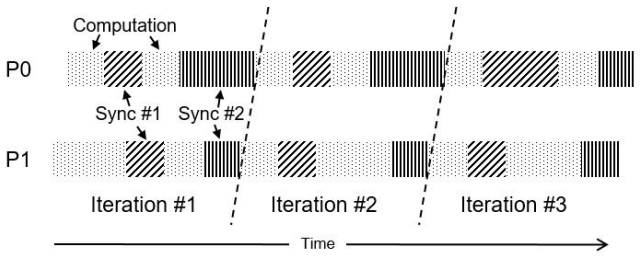

Figure 3. Time series data of two processes running in the same loop. Clocks are skewed.

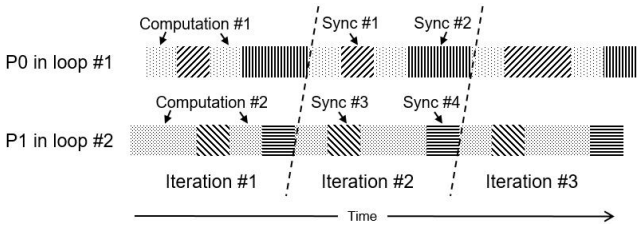

Figure 4. Time series data of two processes running in distinct loops. Clocks are skewed.

to determine process relationships, we have found that it is possible to infer if two MPI ranks belong to the same communicator under certain scenarios. Figure 3 visualizes the time series data of two processes running in the same loop. The loop contains three iterations and each iteration has two synchronization events. We compare the ending time between `P0` and `P1` for all instances of `Sync #1`. In the first iteration, `P0` ends earlier than `P1` in `Sync #1`. In the second iteration, however, `P1` ends earlier than `P0` in `Sync #1`, and the gap becomes larger in the last iteration. As a result, we can infer that `P0` and `P1` are in different communicators for `Sync #1`. In contrast, when we perform the same comparison on `Sync #2`, `P1` ends earlier than `P0` in all three iterations and the gap remains a constant. As a result, we infer that `P0` and `P1` are in the same communicator for `Sync #2`. We can adjust timestamps by the constant gap to align the clocks of `P0` and `P1`.

In this way, our framework exploits calls to MPI collectives and synchronizations within loops to fix skewed clocks. For a pair of MPI ranks, gaps between the end time of such calls must be a constant across all loop iterations. Calls satisfying this condition are used to align the clocks of the pair. Our framework aligns the clock of every MPI rank with the root rank to fix the clock skews across all MPI ranks.

4.1.1 Fixing clock skews for MPMD codes

So far, we have described how our analysis framework fixes clock skews when processes follow the same control flow in the execution (SPMD). Further adaptations are needed when processes follow diverged control flows (MPMD),

Figure 4 visualizes the time series data of two processes running in distinct loops. While P0 and P1 are running different code, they synchronize with each other through calls to the same MPI primitive. There are four different call sites to the same primitive in Figure 4, noted as Sync #1–#4. Our framework needs to find pairs of synchronization call sites that are used to align the clocks of P0 and P1. Such pair of call sites, denoted as S0 and S1, needs to meet the following criteria –

- S0 is from P0 and S1 is from P1,
- number of instances of S0 and S1 are equal,
- and the gap between the ending time of each instance pair of S0 and S1 is a constant.

Our framework would test every possible pair against the above conditions. In this example, only the pair of Sync #2 and Sync #4 meets the requirement. They would be used to fix the skewed clocks of P0 and P1.

4.1.2 Fixing clock skews in parallel

To fix clock skews in parallel, our framework broadcasts the time series data (in the form of a temporal context tree) of thread P0 to every analysis rank. Each analysis rank aligns the clocks of assigned threads to P0. In this way, with P0 as the reference, clocks of all threads are aligned in parallel.

4.2 Phase detection with aligned clocks

After clock skews are fixed, our analysis framework checks every call site to MPI primitives to see if all ranks leave it at the same time.[2] If so, such nodes are used as global synchronizations to divide the parallel executions into a series of independent phases. Processes enter and exit each phase at the same time, which makes it a perfect unit for performance diagnosis.

We want to highlight that our implementation of clock alignment and phase detection is able to tolerate scenarios where calls to MPI primitives are too short to be sampled by HPCToolkit. Our framework uses CFGs to infer the existence of these calls even if they are not sampled.

5 Generating Performance Diagnoses

Our framework analyzes time series data to diagnose causes of scalability losses in parallel executions. We define the scalability issues that our framework detects in Section 5.1; and we discuss semantic labels used by our framework to identify scalability issues in Section 5.2.

Section 4 divides a parallel execution into a series of phases for further analysis. Sections 5.3 and 5.4 describe how our framework analyzes each phase in SPMD and MPMD programs, respectively, to locate scalability losses and report their symptoms, severity, and causes.

[2]Our implementation takes care of marginal errors as ranks never leave at the same time in the real world. For example, some ranks would leave earlier than others in a global synchronization call to MPI_Barrier.

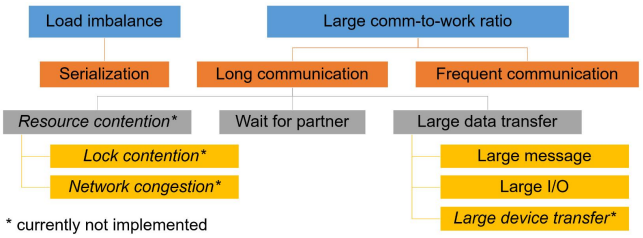

Figure 5. Taxonomy of scalability issues.

5.1 Taxonomy of Scalability Issues

We define *scalability issues* as performance issues that prevent applications from scaling accordingly to the increase in the number of cores or threads that are used. Applications with scalability issues spend a large portion of their execution time inside communication libraries or sitting idle.

To learn the set of scalability issues that should be detected by our automated framework, we inspected time series data that we previously collected for a range of parallel applications and benchmarks, including AMG2013 [21], CESM [15], FLASH [14], Kripke [28], Laghos [11], LULESH [25], NWChem [41], and PFLOTRAN [20]. We also surveyed the literature [9, 27, 29, 35, 40, 44] on various types of scalability issues and solutions to fix them. We identified scalability issues of various kinds in our study.

Load imbalance is defined as uneven distribution of work across threads. It is commonly observed in applications for various purposes, such as [27] and [29]. We also observe *load imbalance* in executions of AMG2013 [21] and FLASH [14].

Serialization is a special case of *load imbalance* in which one thread is responsible for the work while all others are sitting idle, as if the application is executed in serial. As an example, Gaussian Elimination with partial pivoting [13] can suffer from serialization issues as partial pivoting that helps minimize rounding errors is hard to parallelize. We observe *serialization* in the time series data of an MPI+OpenMP execution of AMG2013 [21].

Frequent communication refers to cases where processes make a large number of short but frequent invocations of communication libraries. We observe *frequent communication* in executions of CESM [15], Kripke [28], and NWChem [41].

Wait for partner represents scenarios where processes spend a long time waiting inside a call to communication library for its partner to show up. We observe *wait for partner* in executions of CESM [15].

Large data transfer refers to cases where threads spend a long time transferring data inside communication libraries. We observe large parallel I/O cost in a parallel execution of PFLOTRAN [20].

Based on our survey, we propose a taxonomy of scalability issues as shown in Figure 5. Our taxonomy is similar to the

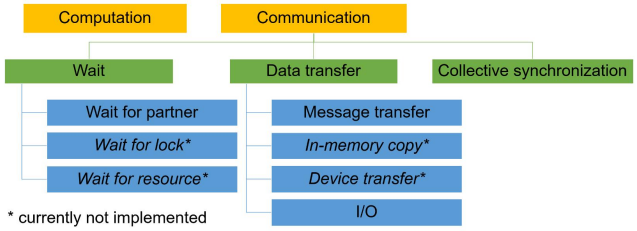

Figure 6. Hierarchy of semantic labels.

W^3 search model employed by Paradyn [31] and the top-down analysis method proposed by Yasin [45]. Unlike these prior work, our focus is on scalability issues. Our automated framework uses this taxonomy as a guideline to detect and categorize scalability losses in parallel executions.

We introduce a few intermediate nodes in the taxonomy to arrange scalability issues discussed above. *Large communication-to-work ratio* summarizes cases where all processes exhibit a large communication overhead, which is unlike load imbalance, where only some processes have a lot of wait. *Long communication* refers to scenarios when calls to communication library routines are longer and less frequent compared to *frequent communication*. *Resource contention* is defined as performance losses due to lack of certain resources, which can be the network, a lock, etc.

Our automated framework detects scalability losses according to the taxonomy in Figure 5. However, due to lack of performance data from various sources, some of the issues, as noted in the figure, cannot be detected at this stage.

5.2 Semantic Labels for Tree Nodes

Generating diagnosis of scalability issues requires knowledge of semantic meanings of nodes in the temporal context tree. Prior work [5, 30, 38] employs a binary semantic division scheme: work vs. wait. However, this scheme is insufficient for diagnosing performance problems according to our taxonomy in Figure 5.

In Figure 6, we propose a hierarchy of semantic labels that satisfies our needs. A binary division is enforced at the top level, separating *computation* from *communication*. Next, *communication* is decomposed into three classes. *Collective synchronization* is designed for collective and synchronization function calls where all participating ranks leave at the same time.[3] *Data transfer* is assigned to procedures in communication libraries that are responsible for transferring data. *Wait* is reserved for functions in communication libraries that handles wait. *Message transfers*, *in-memory copies*, *device transfers*, and *I/O* are four different kinds of *data transfer*. Reasons for *wait* are also listed, including *wait for partner*, *wait for lock*, and *wait for resource*. Again, as noted in Figure 6, not all labels are used at this stage due to lack of performance data from various sources.

[3]For some MPI collective operations, participating ranks may leave at different time, such as MPI_Bcast or MPI_Gather.

Table 1. Initial Mapping from Functions to Semantic Labels.

Functions	Initial semantic label
User defined functions	*Computation*
MPI_Barrier	*Collective synchronization*
MPI all-to-all collectives	*Collective synchronization*
MPI file operations	*I/O*
Other MPI interfaces	*Communication*
MPIDI_CH3I_Progress (MPICH only)	*Wait for partner*
MPID_nem_lmt_shm_start_send (MPICH only)	*Message transfer*

We assign semantic labels to nodes in the temporal context tree in two steps. First, we use a predefined mapping table to assign initial semantic labels. Part of the mapping table is shown in Table 1. User defined functions are considered as *computation* while MPI operations are mapped to *communication*. In particular, MPI primitives where all participating ranks leave at the same time are labeled as *Collective synchronization* whereas calls to MPI file operations are treated as *I/O*. To understand how time is spent inside MPI libraries, we investigate the implementation of MPICH [19] and create special mappings for important internal functions in the MPICH implementation.[4] Two examples are shown in Table 1. *MPIDI_CH3I_Progress* is called when an incoming package need to be retrieved. A majority of time spent in this function can be treated as *wait for partner*. *MPID_nem_lmt_shm_start_send* is used to start message sending after the corresponding rendezvous package has been retrieved; we label it as *message transfers* to reflect its semantics.

Second, we infer derived semantic labels for non-leaf nodes in the temporal context tree. As a motivating example, suppose *MPIDI_CH3I_Progress* accounts for 95% of time spent inside a call to *MPI_Send*, we should better assign *wait for partner* rather than *communication* as the semantic label of *MPI_Send*. In a formal definition, suppose we have a non-leaf node N and a predefined ratio R (currently set to 70%), if there exists a semantic label L such that

- the condition below holds for all processes and threads that have node N,

$$\frac{\sum_{C \in children(N) \wedge DerivedLabel(C)=L} duration(C)}{duration(N)} \geqslant R$$

[4]For the long term, we want the MPI standard to require that MPI implementations expose enough state information for automated analysis. Certain information is not available by labeling internal functions, which prevents us from diagnosing scalability losses due to resource contention.

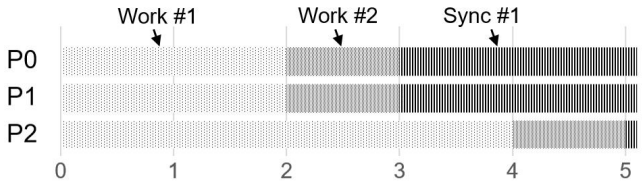

Figure 7. An example of load imbalance.

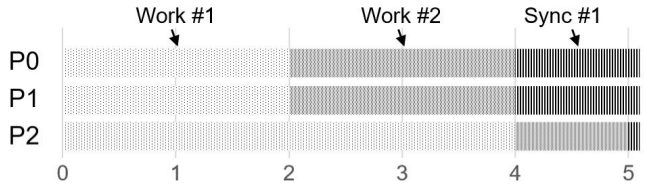

Figure 8. Another example of load imbalance. Processes with less work in **Work #1** have more work in **Work #2**.

- and no descendants of `L` in the label hierarchy satisfies the above condition,

we assign `L` as the derived semantic label of `N`. This inference scheme assigns semantic labels based on the execution and enables more accurate diagnosis to be generated.

5.3 Performance Diagnosis of SPMD Phases

Section 4 divides a parallel execution into a series of phases for further analysis. In this section, we introduce how we diagnose performance in SPMD phases. SPMD phases are those where all processes follow the same control flow. To diagnose performance of such phases, we define three helper metrics for nodes in the temporal context tree. We define `avg(N)` as the average run time of node `N` across all processes. Similarly, we define `min(N)` and `max(N)` as the minimum and maximum run time of node `N`. Our framework calculates these metrics in parallel while reducing thread clusters to the root analysis rank, as described in Section 3.1.

5.3.1 Diagnosis of Load Imbalance

Load imbalance is a common scalability issue in parallel applications. Figure 7 shows an example of load imbalance. In the figure, processes spend an uneven amount of time in `Sync #1`, which is the symptom of load imbalance. The cause of the imbalance is uneven work in `Work #1`.

Our framework calculates two metrics to quantify the severity of load imbalance. First, for each phase, we define `imb_symptom(S)` as the maximum run time reduction if uneven work before its synchronization node `S` is balanced. Then, `imb_cause(C)` is defined as the maximum run time reduction if uneven work in node `C` is balanced. Calculation of `imb_cause(C)` is simple. It would take every process `avg(C)` in execution time if the work is well distributed while the process with the most work spends `max(C)` when a load imbalance exists. Therefore, we have `imb_cause(C) = max(C) - avg(C)`. To calculate `imb_symptom(S)`, we donate all nodes before `S` as N_s and the time spent in the phase as `T`. We have `imb_symptom(S) = imb_cause(`N_s`) = max(`N_s`) - avg(`N_s`) = (T - min(S)) - (T - avg(S)) = avg(S) - min(S)`.

Our automated analysis uses these two metrics to generate diagnosis of load imbalance. Each phase ends with an synchronization node `S` and its `imb_symptom(S)` is used to indicate the presence and severity of load imbalance in the phase. If the imbalance is significant (greater than 1% of total execution time), our framework will calculate `imb_cause(C)` for all other nodes to locate causes of the imbalance. When the sum of `imb_cause(C)` is the same as `imb_symptom(S)`, causes of load imbalance are ones with high `imb_cause(C)` values (greater than 10% of `imb_symptom(S)`). Figure 8 shows a more complicated example. In this example, the sum of `imb_cause(C)` is greater than `imb_symptom(S)` as the imbalance in `Work #1` is partially evened by the imbalance in `Work #2`. In such cases, our framework would report nodes with high `imb_cause(C)` values as causes while noting that the imbalance has been partially evened out.

5.3.2 Diagnosis of Large Comm-to-Work Ratio

Many parallel applications spent a significant amount of time inside communication libraries. To diagnose scalability losses due to *large comm-to-work ratio*, our framework analyzes *communication* nodes in temporal context trees in two ways.

First, our framework calculates how the time is spent in a communication node `N` by inspecting the sub-tree rooted at `N`. It breaks the run time of `N` to the time spent in each semantic label. For example, a call to *MPI_Send*() may spend 75% of time in *wait for partner*, 20% of time in *message transfer*, 5% of time elsewhere.

Second, our framework distinguishes long but less frequent communication library calls from short and frequent ones. Every node in the temporal context tree has a duration and a return count. The return count is a lower bound on the number of call instances to the node. We compute *duration per instance* of a node `N` by dividing its duration with return count. Due to the way we collect return counts in the time series data, the result would be a close reflection of the actual duration per instance when most invocations are longer than the sampling period. However, it can be greater than the actual value when most invocations are shorter than the sampling period, in which case the computed *duration per instance* would be equal to or slightly larger than the sampling period.

Our analysis framework scans nodes in a phase to pick every *communication* node `N` whose `avg(N)` is significant (greater than 1% of total execution time). Any picked node with a small *duration per instance* (less than 5× sampling period) will be reported as frequent communication. Other than that, depending on the breakdown of run time `N`, we would

Table 2. Performance Diagnosis of Selected Scalability Issues.

Type of issue	Symptom	Severity	Cause	Potential solutions
Load imbalance	For a synchronization node `S` at the end of a phase, `imb_symptom(S)` is significant.	`avg(S) - min(S)`	Imbalanced work in nodes from the same phase whose `imb_cause` is significant.	Distribute the work in causal nodes evenly.
Frequent Communication	For a *communication* node `N`, `avg(N)` is significant while *duration per instance* of `N` is low.	`avg(N)`	Loops on the call chain to reach `N`. Work in each iteration can be too fine-grained.	Reduce the number of calls to `N`. Condense iterations if possible.
Wait for partner	For a *wait for partner* node `N`, `avg(N)` is significant while *duration per instance* of `N` is high.	`avg(N)`	The communication partner who arrived late.	Substitute blocking calls with non-blocking ones, or check why partner arrived late.
Large message	For a *message transfer* node `N`, `avg(N)` is significant while *duration per instance* of `N` is high.	`avg(N)`	The data being transferred is large.	Overlap with computation, or cut it into smaller messages for balanced network utilization.

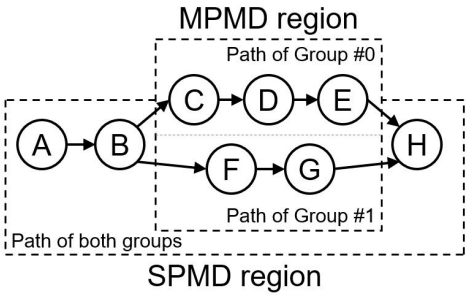

Figure 9. An example of two groups of processes diverge in the control flow.

report one or more scalability loss under *long communication* in the taxonomy.

Table 2 is a summary of how our automated framework diagnoses four different types of scalability issues. The symptom column shows the criteria used by our framework to detect the issue in the corresponding row. After an issue is detected, our framework generates a diagnosis reporting the symptom, severity, and cause of the issue and provides users with potential solutions. Diagnosis of *large I/O* is omitted in the table as it is very similar to *large message*. As noted in Figure 5, diagnosis of other scalability issues is currently unavailable because we lack the performance data needed to diagnose them. *Lock contention* and *network congestion* need state information from lock and NIC respectively; while *large device transfer* will be available in the future when we integrate the support for acceleration devices.

5.4 Performance Diagnosis of MPMD Phases

To extend performance diagnosis to MPMD phases, our framework use CFGs to identify equivalent classes of processes who follow the same control flow in a phase. Figure 9 shows an example where two process groups diverge in the control flow of a phase.

Performance diagnosis of MPMD phases are arranged in three steps. First, in MPMD regions, our framework applies the analysis in Section 5.3 to each group of processes. If a scalability issue of the same type is detected in all groups, our framework would report the issue and list the cause from each group. Second, our framework calculates the average work load of each process group. If the average work load of a group is significantly larger than the average of all processes, our framework would report load imbalance across MPMD groups. Third, our framework applies the analysis in Section 5.3 to the SPMD region.

Table 3. Theta and Titan HW and SW Configurations

	Theta	Titan
Hardware Configurations		
Core	Intel Xeon Phi 7230	AMD Opteron Processor 6274
Clock frequency	1.3GHz	2.2 GHz
# cores per node	64	16
# threads per core	4	1
Software Configurations		
Operating system	SUSE Linux 12.3	SUSE Linux 11
Compiler	gcc 7.3.0	gcc 7.3.0
MPI implementation	cray-mpich 7.7.3	cray-mpich 7.6.3

6 Case Studies

We evaluate our automated framework by applying it to time series data for two parallel executions on the Theta supercomputer at Argonne National Laboratory (Table 3). We use HPCToolkit to collect a time series of call path samples for

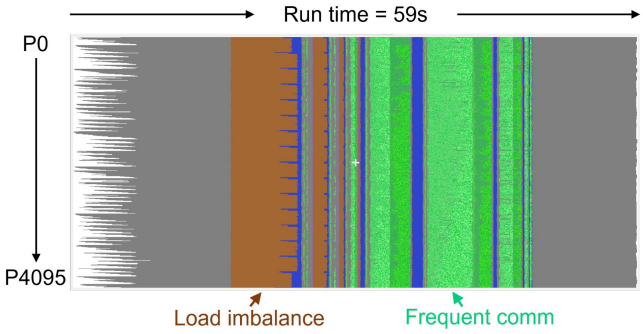

Figure 10. Annotated screenshots from our revised version of HPCToolkit's hpctraceviewer rendering a 16×16×16 execution of AMG2013 on Theta.

each process in the parallel executions. After that, we apply our automated framework to generate diagnosis report of scalability losses in the executions. We present the visualization of time series data using a version of HPCToolkit's hpctraceviewer modified to highlight scalability issues in the diagnosis report.

In addition, we used the time series data collected on Oak Ridge National Laboratory's Titan supercomputer[5] (Table 3) to verify our approach for fixing clock skews as we have never observed clock skew on Theta.

6.1 Analysis of an AMG2013 execution

Algebraic MultiGrid (AMG) 2013 [21] is a parallel iterative solver of unstructured linear systems. We run the default solver on a 16×16×16 processor grid using 64 nodes with 64 cores per node and one thread per core on Theta.

Our framework reports two losses in the AMG2013 execution as shown in Table 4. Figure 10 shows the visualization of the execution from our revised hpctraceviewer. Distinct colors are assigned to call frames that are symptoms or causes of scalability losses. The depth of each call path is automatically selected to simultaneously highlight these call frames that arise at different call path depths. Other call frames are colored in gray. Scalability losses in Table 4 are annotated at the bottom of the figure to show their location in the visualization.

The *frequent communication* loss in Table 4 and Figure 10 is the most severe. An optimal fix that removes the communication entirely would see a reduction of 32% in total execution time. Our automated diagnosis breaks down the *frequent communication* into three calls – *MPI_Iprobe*, *MPI_Testall*, and *MPI_Test*. They are located in *loop at line 331* of function *hypre_DataExchangeList*. Function *hypre_DataExchangeList* is responsible for sending and receiving messages. However, a process never knows the exact number of messages it will receive. As a result, distributed termination using a binary tree is implemented to make sure no process would leave

[5]Titan was decommissioned on August 1st, 2019.

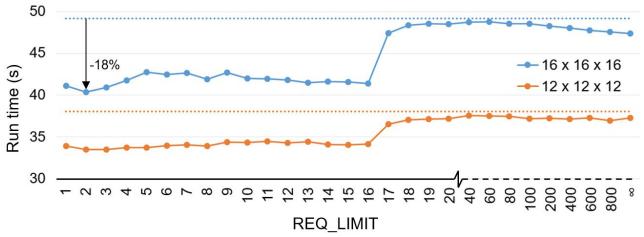

Figure 11. Effect of AMG2013 optimizations on two processor configurations on Theta.

this function before everyone finishes. In the function, a process first initializes a list of asynchronous send requests for outgoing messages and asynchronous receive requests for responses to these messages. After that, in *loop at line 331*, it actively checks incoming messages with *MPI_Iprobe*, tests the status of asynchronous send and receive requests with *MPI_Testall*, and invokes *MPI_Test* for distributed termination after all requests have retired.

Every rank communicates with many partners in the green regions in Figure 10 – sending messages to as much as 80% of all ranks. We identify two potential problems in the code that may lead to performance loss. First, each rank initiates all outgoing message requests before it retrieves incoming ones, which could lead to large overhead in the MPICH progress engine. Second, every rank initiates outgoing messages in the order of destination rank ID. Many of them would initiate the message request to rank #0 first, which could result in network congestion.

We implemented two optimizations based on these discoveries. First, we have every rank initiate messages in a random order. Second, we limit the number of requests that can be issued. Every rank is allowed to initiate no more than *REQ_LIMIT* asynchronous send requests at first. More requests can be issued only if some requests retire. At any time, no more than *REQ_LIMIT* send requests are allowed. Each send request is associated with an asynchronous receive request for the response. The same *REQ_LIMIT* applies to these receive requests.

Figure 11 shows the effect of our optimizations on two processor configurations. All run time presented in the figure is the median over five experimental runs. In the figure, two horizontal dotted lines represent execution time without optimizations. On concrete lines, each dot shows the execution time of the corresponding processor configuration on a *REQ_LIMIT* value. Notice that the y-axis starts at 30 and the x-axis is not contiguous from 20 to ∞. *REQ_LIMIT* = ∞ means no limit on the number of pending request and the performance gain comes from initiating messages in a random order.

In Figure 11, the lowest run time is achieved at *REQ_LIMIT* = 2 for both processor configurations. The run time reduction of the 16×16×16 execution is 18%, which is less than the

Table 4. Scalability Losses Identified by Automated Diagnosis in the AMG2013 Execution. Ranked by Severity.

Type of loss	Symptom	Severity	Cause
Frequent communication	Three significant *communication* call paths whose *duration per instance* are low.	32%	Too frequent calls inside *loop at line 331* in *hypre_DataExchangeList*.
Load imbalance	`imb_symptom` of synchronization *hypre_NewCommPkgCreate* is significant.	2.1%	Imbalanced work distribution in *hypre_BoomerAMGBuildCoarseOperator*.

severity of 32% as shown in our automated diagnosis report since we are speeding up, not removing, these communications. Another interesting observation is the big performance degradation when *REQ_LIMIT* is incremented from 16 to 17. We suspect this degradation is due to the limited capacity of nemesis queue in the MPICH implementation. According to the study by Buntinas et al. [6], the nemesis channel offers top or near-top performance for both intra-node and inter-node communication. As a result, the performance would not be optimal when message requests don't fit into the nemesis queue. Unfortunately, we are unable to verify our hypothesis as we cannot configure the size of nemesis queue in the cray-mpich implementation on Theta.

Compared to the *frequent communication*, the *load imbalance* loss in Table 4 is less severe. The pattern and cause of imbalance are straightforward. Work is unevenly distributed as processes on the boundary of the 3D grid have less work compared to ones at the center.

The benefits of our new analysis framework can be appreciated by comparing our performance loss diagnosis of AMG2013 with our new framework and the case study of the same application in our prior work [43]. Because our original framework only exploited shared-memory parallelism when analyzing time series data for an execution, we could only apply it to an 8×8×8 processor execution of AMG2013. The automated summary produced for this execution provided no indication of the *frequent communication* loss that we observed in Table 4, as the loss was not significant in small scale executions. This shows that some performance issues only occur at large scales and support for automated performance analysis at large scales is necessary to be able to identify causes of such losses. In addition, our original analysis framework could only highlight a few call paths with imbalance while our framework generates a complete diagnosis of *load imbalance*, as shown in Table 4.

6.2 Analysis of a Laghos execution

The Laghos (LAGrangian High-Order Solver) [11] mini-app solves the time-dependent Euler equations of compressible gas dynamics in a moving Lagrangian frame. We run the problem 1 on an 8×8×8 processor grid using 8 nodes with 64 cores per node and one thread per core on Theta.

The Laghos execution consists of 10 loop iterations that exhibit similar behaviors. Figure 12 visualizes one of the iterations on a selected set of processes. As annotated at the bottom of the figure, our framework detects four instances of MPMD regions. These MPMD regions are in four separate calls to *mfem::RK4Solver::Step*. In the step function, some processes call *RAPOperator::Mult* while others are diverted to *MassPAOperator::Mult*, leading to the MPMD behavior. Our framework reports no significant performance loss in these regions.

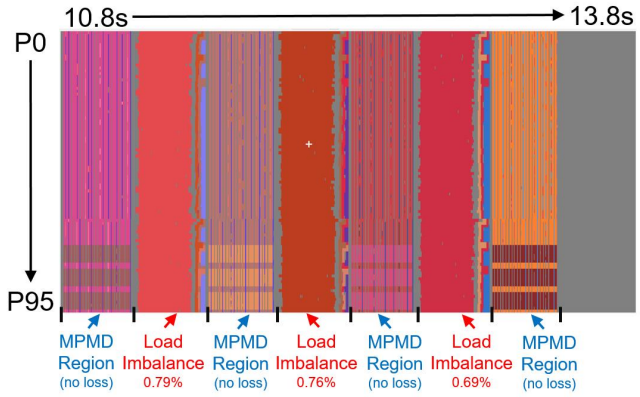

Figure 12. Annotated expanded view from our revised version of HPCToolkit's hpctraceviewer rendering part of a 8×8×8 execution of Laghos on Theta.

Table 5. Performance Measurement Overhead

8×8×8 executions	AMG2013 @ 30.3s		Laghos @ 31.3s	
	Time	Space	Time	Space
hpcrun -t	31.3s *(+3.3%)*	381M	32.9s *(+5.1%)*	318M
scalasca -q -t	69.0s *(+128%)*	631G	76.6s *(+145%)*	902G

Besides MPMD, our framework reports three instances of *load imbalance* in the execution. As shown in Figure 12, a few processes, such as `P0` and `P64`, has slightly more work than others due to the nature of the computation. Fortunately, the sum of the severity of these *load imbalance* losses is only 2.24%. Overall, this 512-process execution of Laghos is not suffering from significant scalability losses.

6.3 Measurement and Analysis Overhead

To compare our work against automated tools that analyze instrumentation-based traces, we choose Scalasca [16] as the representative as it can collect and analyze traces in parallel

Table 6. Automated Performance Analysis Overhead

4×4×4 executions	AMG2013 @ 25.1s	Laghos @ 29.8s
our framework	14.6s *(0.57×)*	51.0s *(1.7×)*
scalasca	376s *(15×)*	crashed

and can diagnose scalability losses. We are unable to apply Scalasca to 16×16×16 executions of AMG2013 due to its large overhead in time and space. As an alternative, Table 5 shows the performance measurement overhead of HPCToolkit [1] and Scalasca [16] on 8×8×8 executions of AMG2013 and Laghos on Theta. Both 512-process executions in Table 5 take around 30 seconds without performance measurement. We use HPCToolkit's hpcrun to collect sample-based time series data at a frequency of 4 millisecond per sample. On both executions, hpcrun's measurement overhead in time is nearly ignorable and the size of generated measurement databases are less than 512M, or 1M per process. On the other hand, tracing with Scalasca incurrs much higher overhead. The measurement overhead in time is greater than 1× for both executions and the generated measurement databases are extremely large – greater than 1G per process. While Scalasca has support for trace filtering that can help reduce measurement overhead, this process generally involves conducting one profiling experiment, analyzing generated profiles, and manually choosing which functions to filter. Collecting sample-based time series data would still have an edge even if trace filtering reduces the measurement overhead in time by 90% and space by 99%, which would require great user expertise if it is ever possible.

Applying Scalasca's automated analysis to traces from 512-process executions results in crashes due to the data volume. As a result, we apply our framework and Scalasca to performance measurement data from 4×4×4 executions of AMG2013 and Laghos for comparison. The automated analysis overhead is shown in Table 6. Both automated analysis is launched in parallel on 64 processes – the same number of processors as the parallel execution. Our framework spend a reasonable amount of time to generate performance diagnosis report for both executions – *0.57*× and *1.7*× of the original execution time. On the other hand, Scalasca spends much more time to diagnose the AMG2013 execution and crashes for the Laghos execution.

6.4 Effect of fixing clock skews

We never observed any clock skew in our case studies on Theta. As a result, we use the time series data collected from a 1728-processor execution of AMG2013 on Titan to test the effectiveness of our approach for fixing clock skews that we described in Section 4.1. The time series visualization in Figure 1 shows skewed clocks on several processors. We applied our analysis to fix these clock skews. As shown

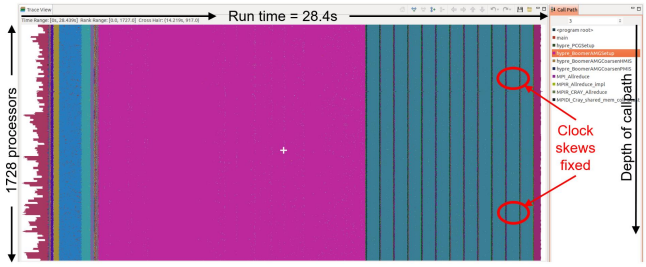

Figure 13. Annotated screenshot of HPCToolkit's hpctraceviewer examining the same 1728-processor execution of AMG2013 on Titan as in Figure 1, except that clock skews has been fixed by applying our analysis.

in Figure 13, all clock skews visible in Figure 1 have been eliminated, which shows the effectiveness of our approach.

So far, we have demonstrated the effectiveness of our automated framework with case studies on AMG2013 and Laghos. In the 4096-process AMG2013 execution, we were able to use the automated insights provided by our framework to implement optimizations that reduces the execution time by 18%. In the 512-process Laghos execution, our framework detects the MPMD regions correctly and provides no indication of significant losses. Our framework exhibits much lower performance measurement and analysis overhead compared to Scalasca, which validates our claim that analyzing sample-based time series data is a more practical approach for large scale parallel executions.

7 Conclusions and Future Work

This paper describes an automated analysis framework that uses time series of call path samples to diagnose scalability losses in parallel executions. We developed a new strategy for correcting clock skews with sample-based time series data to support robust identification of execution phases. Our framework uses a taxonomy of scalability issues to drive analysis of symptoms of scalability losses, identify their root causes, and rate their severity. Our analysis is applicable to both SPMD and MPMD codes and runs in parallel to support diagnosis of large scale executions. Case studies on AMG2013 and Laghos demonstrate the effectiveness of our framework.

Our new analysis framework has clear space and time advantages over prior frameworks. When compared to Scalasca [16], which is representative of automated tools that digest instrumentation-based traces, our overhead for collecting sample-based time series is 1/28 in time and 1/1600 in space. Our automated analysis takes 1/25 of the time spent by Scalasca. With a much lower overhead, we are able to apply our automated framework to analyze executions at large scales. Compared to our prior framework for analyzing sample-based time series data on a shared memory system [43], our new scalable analysis framework both employs

distributed-memory parallelism to analyze large scale executions and provides comprehensive diagnoses of scalability losses, which our prior framework could not.

There are some limitations of our approach. To benefit from the low overhead of sample-based measurement, we lose the high accuracy of instrumentation-based tracing. Using sample-based time series data, our framework has no knowledge of the source and destination of MPI point-to-point operations. As a result, our framework can identify the cost due to *wait for partner* but won't be able to locate a late sender. Using complete communication traces, Scalasca [16] can pinpoint late senders, but requires much larger time and space overhead to do so. In this case, one can apply our automated framework to quickly locate where the losses are and implement a trace filter so that a tracing tool such as Scalasca can collect and analyze traces for a narrow window of execution with a lower overhead to provide a more detailed diagnosis.

We plan several extensions to the framework described in this paper. First, we plan to add support for hierarchical parallel models, such as MPI+OpenMP and MPI+CUDA. Many applications use hybrid models to introduce parallelism at various levels; support for such applications is important. Second, we plan to enhance diagnosis within loops. In the scope of this paper, symptoms and causes of scalability issues are reported as call paths; every loop iteration with those call paths is highlighted in the visualization. As future work, we want to pick a few representative iterations that are able to summarize losses in a loop. Users would only need to focus on those representatives to address scalability losses. Third, we want to integrate performance data from other sources into our analysis. In many cases, our framework doesn't have access to semantic or state information required to diagnose performance losses. For example, a user may implement a busy-wait lock of which our framework has no knowledge. In addition, our framework won't be able to diagnose *network congestion* unless certain state information in MPI is exposed. We plan to integrate data from tools such as Caliper [4], with which users can supply semantic and state information for the code. We also will continue to advocate that the MPI standard define interfaces that expose state information within MPI libraries as current interfaces are insufficient for detailed performance introspection.

Acknowledgments

We would like to acknowledge all past and current members of the HPCToolkit team for their contributions to the software infrastructure used as the foundation for this research. We also want to thank the reviewers for their valuable feedback.

This work was supported in part by the National Science Foundation under Grant No.1450273. Access to Titan at Oak Ridge National Laboratory and to Theta at Argonne National Laboratory were through allocations provided by the DOE Exascale Computing Project.

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] Allinea. 2017. Allinea Forge User's Guide. http://content.allinea.com/downloads/userguide-forge.pdf

[3] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. 2009. Scalable Timestamp Synchronization for Event Traces of Message-passing Applications. *Parallel Comput.* 35, 12 (Dec. 2009), 595–607. https://doi.org/10.1016/j.parco.2008.12.012

[4] David Boehme et al. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 47, 11 pages. https://dl.acm.org/doi/10.5555/3014904.3014967

[5] 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

[6] D. Buntinas, G. Mercier, and W. Gropp. 2006. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, Vol. 1. 10 pp.–530. https://doi.org/10.1109/CCGRID.2006.31

[7] Martin Burtscher et al. 2010. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. 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.2010.41

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

[9] J. Corbalan, A. Duran, and J. Labarta. 2004. Dynamic load balancing of MPI+OpenMP applications. In *International Conference on Parallel Processing, 2004. ICPP 2004.* 195–202 vol.1. https://doi.org/10.1109/ICPP.2004.1327921

[10] 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

[11] V. Dobrev, T. Kolev, and R. Rieben. 2012. High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics. *SIAM Journal on Scientific Computing* 34, 5 (2012), B606–B641. https://doi.org/10.1137/120864672

[12] Jens Doleschal, Andreas Knüpfer, Matthias S. Müller, and Wolfgang E. Nagel. 2008. Internal Timer Synchronization for Parallel Event Tracing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 202–209.

[13] Simplice Donfack et al. 2015. A survey of recent developments in parallel implementations of Gaussian elimination. *Concurrency and Computation: Practice and Experience* 27, 5 (2015), 1292–1309.

[14] A. Dubey, L.B. Reid, and R. Fisher. 2008. Introduction to FLASH 3.0, with application to supersonic turbulence. *Physica Scripta* T132 (2008). Topical Issue on Turbulent Mixing and Beyond, results of a conference at ICTP, Trieste, Italy, August 2008.

[15] University Corporation for Atmospheric Research. 2010. Community Earth System Model (CESM) 1.0. http://www.cesm.ucar.edu/models/cesm1.0/

[16] 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

[17] 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

[18] 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

[19] William Gropp. 2002. MPICH2: A New Start for MPI Implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, London, UK, UK, 7–. https://dl.acm.org/doi/10.5555/648139.749473

[20] 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/10.2174/978160805306311201 01

[21] 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

[22] 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

[23] Intel. 2017. Intel VTune Amplifier 2019 User Guide. https://software.intel.com/en-us/vtune-amplifier-help

[24] Terry Jones and Gregory A. Koenig. 2010. A Clock Synchronization Strategy for Minimizing Clock Variance at Runtime in High-End Computing Environments. In *Proceedings of the 2010 22Nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '10)*. IEEE Computer Society, Washington, DC, USA, 207–214. https://doi.org/10.1109/SBAC-PAD.2010.33

[25] I. Karlin et al. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 919–932. https://doi.org/10.1109/IPDPS.2013.115

[26] Andreas Knüpfer et al. 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

[27] Christian Koziar, Robert Reilein, and Gudula Runger. 2005. Load Imbalance Aspects in Atmosphere Simulations. *Int. J. Comput. Sci. Eng.* 1, 2-4 (May 2005), 215–225. https://doi.org/10.1504/IJCSE.2005.009705

[28] Adam J Kunen, Teresa S Bailey, and Peter N Brown. 2015. *KRIPKE - a Massively Parallel Transport Mini-App*. Technical Report. Lawrence Livermore National Laboratory.

[29] Zhiling Lan, Valerie E Taylor, and Greg Bryan. 2002. Dynamic load balancing of SAMR applications on distributed systems. *Scientific Programming* 10, 4 (2002), 319–328.

[30] 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

[31] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. 1995. The Paradyn parallel performance measurement tool. *Computer* 28, 11 (Nov 1995), 37–46. https://doi.org/10.1109/2.471178

[32] David Mills, Jack Burbank, and William Kasch. 2010. Network Time Protocol Version 4: Protocol and Algorithms Specification. https://tools.ietf.org/html/rfc5905

[33] 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.

[34] Rolf Rabenseifner. 1997. The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters. In *In Proc. of the 5th EUROMICRO Workshop on Parallel and Distributed Processing (PDP)*. 477–484.

[35] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. 2006. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 17–17. https://doi.org/10.1109/SC.2006.51

[36] Boris Schling. 2011. *The Boost C++ Libraries*. XML Press.

[37] 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

[38] 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

[39] 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

[40] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing Lock Contention in Multithreaded Applications. *SIGPLAN Not.* 45, 5 (Jan. 2010), 269–280. https://doi.org/10.1145/1837853.1693489

[41] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477 – 1489. https://doi.org/10.1016/j.cpc.2010.04.018

[42] Lai Wei. 2020. *Automated Diagnosis of Scalability Losses in Parallel Applications*. Ph.D. Dissertation. Rice University, Houston, TX, USA. Advisor(s) John Mellor-Crummey.

[43] Lai Wei and John Mellor-Crummey. 2018. Automated Analysis of Time Series Data to Understand Parallel Program Behaviors. In *Proceedings of the 2018 International Conference on Supercomputing (ICS '18)*. ACM, New York, NY, USA, 240–251. https://doi.org/10.1145/3205289.3205308

[44] M. H. Willebeek-LeMair and A. P. Reeves. 1993. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems* 4, 9 (Sep. 1993), 979–993. https://doi.org/10.1109/71.243526

[45] A. Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

A Artifact Appendix

This artifact illustrates our automated framework that analyzes sample-based time series data to diagnose scalability losses in parallel executions. The artifact consists of a virtual machine image based on Ubuntu 18.04.3 that includes:

- an implementation of AMG2013[6],
- a version of the HPCToolkit performance tools that measure and analyze sample-based time series of parallel programs[7],
- a version of HPCToolkit's hpctraceviewer that visualizes results of automated analysis of sample-based time series of parallel programs[8], and
- a version of Scalasca and Score-P that instrument parallel codes to collect and analyze their execution traces[9].

A.1 Downloading and installing VMWare Player

The artifact virtual machine image is intended for use with VMware Workstation Player 15.5[10], which is available for both Windows and Linux and is free for non-commercial download and use. Note that installing VMware Workstation Player requires administrator privileges on Windows and root privileges on Linux.

The artifact is designed for use on a laptop, desktop, or server that includes one or more x86_64 processors. Using the software in the artifact includes launching a 12 process execution of a message-passing parallel program. The virtual machine image requires a system with at least 12 Hyper-Threads.

A.2 Downloading and running the artifact virtual machine image

The 5.75GB virtual machine image containing the artifact can be downloaded with a web browser by visiting the following URL and clicking the `download` button: https://rice.box.com/s/hq0z7xwr62i04a6ip8jlqye4fy0hktep. Unpack the downloaded ppopp174.zip file and the zip file will unpack into the ppopp174 directory.

Launch VMware Workstation Player and load the virtual machine image at ppopp174/ppopp174.vmx. Log into the virtual machine by using ppopp174 as the username and ppopp2020 as the password.

A.3 Running automated analysis of time series data

To run a quick experiment, change to the directory that contains the AMG2013 executable:

```
> cd $HOME/ppopp174/software/AMG2013/test
```

Run our version of HPCToolkit to collect time series data for a 12-process AMG2013 execution:

```
> mpirun -np 12 hpcrun -e REALTIME@2000 -t
  ./amg2013 -pooldist 0 -r 10 10 10 -P 3 2 2
  2>&1 | tee amg2013_hpcrun.out
```

The measurement data will be saved to folder hpctoolkit-amg2013-measurements/. The execution should take 10 - 20 seconds. If the execution is too short, delete hpctoolkit-amg2013-measurements/ and increase the values after -r (e.g. replacing '-r 10 10 10' with '-r 12 12 12'); decrease the values after -r (e.g. replacing '-r 10 10 10' with '-r 8 8 8') if the execution is too long.

In the next step, use `hpcstruct` to analyze the AMG2013 executable:

```
> hpcstruct amg2013
```

This command will generate the file amg2013.hpcstruct.

Next, run `hpcprof` to render the measurement data and perform automated analysis:

```
> hpcprof -S amg2013.hpcstruct -T
  hpctoolkit-amg2013-measurements -o
  hpctoolkit-amg2013-with-analysis-database
  2>&1 | tee automated_analysis.out
```

A database would be generated at hpctoolkit-amg2013-database/. Stdout (also saved to automated_analysis.out) should contain the following:

- lines starting with "msg" and "Analyzing executable",
- followed by a line stating "Trace analysis started at",
- followed by a few lines starting with "Analyzing file",
- followed by a line stating "Inefficient execution segments:"
- followed by all execution phases with performance losses, where
  - each inefficient execution phase consists of multiple call paths;
  - call frames that start with "**" indicate call paths that are symptoms or causes of performance losses;
  - the end of each phase has a summary of severity and symptom-cause relationship of performance losses in the phase; note that `IR` stands for *inefficiency ratio*, which refers to the maximum runtime reduction one can achieve if the corresponding losses can be eliminated.
- and the last line stating "Trace analysis finished at ***".

The output of our automated analysis can vary on different measurements, as the behavior of AMG2013 won't be exactly

[6] AMG2013 is available at https://computing.llnl.gov/projects/co-design/download/amg2013.tgz

[7] Source code associated with this version of HPCToolkit is available at https://github.com/hpctoolkit/hpctoolkit, branch trace-analysis.

[8] Source code associated with this version of hpctraceviewer is available at https://github.com/hpctoolkit/hpcviewer, branch trace-analysis.

[9] Scalasca v2.5 release can be downloaded from http://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/dist/scalasca-2.5.tar.gz. Scalasca v2.5 requires Score-P v5.0 for instrumentation, which is available at https://www.vi-hps.org/cms/upload/packages/scorep/scorep-5.0.tar.gz

[10] A copy of VMware Workstation Player 15.5 for Windows or Linux can be downloaded from the following webpage: https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html. See the following webpage for instructions how to install VMWare Player https://kb.vmware.com/s/article/2053973.

the same in each run. In our experiment across multiple runs, it is most likely that our automated analysis would report one inefficient segment. Imbalance will be reported in the segment, with `hypre_BoomerAMGBuildCoarseOperator` as the cause, `hypre_NewCommPkgCreate` as the symptom, with an `IR` of around 1%. This imbalance reproduces the one we presented in Table 4 with a severity of 2.1%. There are also times where our automated analysis reports nothing, which means the corresponding execution is efficient. It is also possible that our automated analysis reports more than one loss or more than one inefficient segment; nevertheless, imbalance due to `hypre_BoomerAMGBuildCoarseOperator` should be the loss with the highest `IR`.

Next, use `hpctraceviewer` to visualize the time series data (with automated analysis) by executing:

```
> hpctraceviewer
  hpctoolkit-amg2013-with-analysis-database
```

This visualization is similar to Figure 10, which highlights performance losses in the parallel execution according to our automated analysis.

Until now, we have run a case study of AMG2013 at a much smaller scale (12 MPI ranks) compared to the one in Section 6.1 that uses 4096 MPI ranks. Nevertheless, we show that our automated analysis can pinpoint the same load imbalance as presented in Table 4. Unfortunately, frequent communication in Table 4 only becomes significant when the scale is large enough (>512 MPI ranks); as a result, we are unable to reproduce a similar result for frequent communication in this artifact.

A.4 Comparison with manual analysis

To compare visualization based on automated diagnosis with the one that requires manual analysis, execute `hpcprof` without automated analysis by running

```
> hpcprof -S amg2013.hpcstruct
  hpctoolkit-amg2013-measurements -o
  hpctoolkit-amg2013-no-analysis-database
```

A database will be generated at hpctoolkit-amg2013-no-analysis-database/.

Next, use `hpctraceviewer` to visualize the time series data (no automated analysis) by executing:

```
> hpctraceviewer
  hpctoolkit-amg2013-no-analysis-database
```

This visualization is similar to Figure 1, which requires manual effort to explore the time series data in three dimensions to locate performance losses.

A.5 Comparison with Scalasca

To compare the overhead of performance measurement and analysis between our tool and Scalasca, run AMG2013 without performance data collection:

```
> mpirun -np 12 ./amg2013 -pooldist 0 -r 10 10 10
  -P 3 2 2 2>&1 | tee amg2013.out
```

Next, collect traces of AMG2013 execution with Scalasca:

```
> scalasca -analyze -q -t mpirun -np 12
  ./amg2013_inst -pooldist 0 -r 10 10 10 -P 3 2 2
  2>&1 | tee amg2013_scalasca.out
```

Scalasca database scorep_amg2013_inst_12_trace/ should be generated.

To compare performance measurement overhead in time, compare the run time of AMG2013 in amg2013.out, amg2013_hpcrun.out, and amg2013_scalasca.out. Please be aware that the execution time can be affected by other factors, such as disk caching, and you should run the measurement a few times until the run time stabilizes. Runtime in amg2013.out, amg2013_hpcrun.out, and amg2013_scalasca.out should be mostly similar to each other, except that Scalasca's measurement incurs higher overhead in the execution of *PCG Setup* when compared to HPCToolkit. Scalasca's measurement overhead in time is less severe compared to data we presented in Table 5 as the scale of this case study is fairly small (12 MPI ranks). Scalasca's measurement overhead grows rapidly with the increase in scale of parallel executions.

To compare overhead of performance measurement in space, run:

```
> du -sh *
```

Size of the database generated by our tool should be around 1.5M while Scalasca uses around 50M.

To compare overhead of performance analysis in time, check automated_analysis.out for the time consumed by our automated analysis and compare it with time consumed by Scalasca in amg2013_scalasca.out. Scalasca should consume a bit more time compared to our tool.

In this case study of AMG2013 on 12 MPI ranks, our tool incurs lower performance measurement overhead in time and space compared to Scalasca, which is in accordance with the data we presented in Table 5. Please note that Scalasca's measurement and analysis overhead grows rapidly with the increase in scale of parallel executions; as a result, its overhead is overwhelming when measuring 512 ranks (Table 5) and analyzing 64 ranks (Table 6).

A.6 Artifact summary

In this artifact, we run a case study of AMG2013 on 12 MPI ranks to show that our automated analysis is capable of providing useful performance insight and our visualization based on automated insight is able to highlight performance losses in a parallel execution, as covered in Section 6.1. We also show that our automated framework has a lower performance measurement overhead compared to Scalasca, as covered in Section 6.3.

In the virtual machine image, we are unable to reproduce similar results on Laghos in Section 6.2 as the MPMD behavior of Laghos only appears at larger scales.