# *TinyProf*: Towards Continuous Performance Introspection through Scalable Parallel I/O

Ke Fan*§, Suraj Kesavan†, Steve Petruzza‡, Sidharth Kumar*§
* University of Illinois at Chicago, Chicago, IL, USA
† University of California at Davis, Davis, CA, USA
‡ Utah State University, Logan, UT, USA
§ Email: {kfan23, sidharth}@uic.edu

***Abstract*—Performance profiling tools are crucial for HPC specialists to identify performance bottlenecks in parallel codes at various levels of granularity (i.e., across nodes, ranks, and threads). Although numerous sophisticated profiling tools have been developed, achieving scalable performance introspection on large scales remains a challenge. This is particularly evident in efficiently *writing* profiles to disk during runtime and subsequently *reading* them with constrained computing resources for post-hoc analysis. In this paper, we present *TinyProf*, a performance introspection framework that tackles I/O-related challenges in profiling performance data at *scale*. *TinyProf*'s scalability is attributed to an optimal runtime that consists of three key components: (1) an efficient in-memory data structure that minimizes memory consumption and decreases communication overhead during parallel file I/O; (2) a customizable three-phase I/O scheme that generates optimal I/O patterns capable of scaling with high core counts; and (3) a streamlined data format for profiles, which guarantees minimal sizes for profile files. These three techniques instill scalability into the profiler, making it low overhead, even at high process counts (less than $5\%$). This low overhead makes it possible for the profiler to be run with an application as a default (whenever the application is running)—enabling *continuous* introspection of performance. We demonstrate the efficiency of our framework using large-scale parallel applications and perform a thorough evaluation against existing systems up to $32k$ processes.**

## I. Introduction

Rapid advancements in computing technologies, especially the arrival of exascale machines, are pushing the frontiers of computational sciences in terms of both the scale and complexity of problems that can be studied [1]. However, the increasing possibilities also necessitate optimal use of computational resources to achieve faster time-to-solution, a lower (monetary) cost of computing, and a lower carbon footprint. Performance profiling tools are critical to achieving these goals. Broadly, profiling entails measuring key metrics for the performance of parallel programs and their specific portions (*e.g.,* lines of code, loops, and functions), while preserving the semantic context. Commonly employed metrics measure execution runtime, data movement, cache misses, and other associated costs. This typically follows the use of performance visualization tools, enabling a post-hoc analysis [2], [3] of the generated profiles.

Although performance profiling aims to improve the overall performance of applications, significant advantages can be reaped through *continuous* introspection, *i.e., always* running the parallel code with the profiling capabilities turned on. Continuous introspection can be key to understanding and eventually *explaining* one of the bigger challenges in HPC – *performance variability* [4], [5], [6], *i.e.,* the phenomenon of the same code exhibiting different performance during various runs, potentially due to the presence of a straggler process or resource contention, such as the network or the I/O system. In this paper, we strive towards continuous introspection through our *scalable* performance profiling system.

State-of-the-art performance analysis frameworks [7], [8], [9] face challenges in keeping pace with the escalating scale of modern applications. These challenges fall into two broad categories: (1) I/O overhead and scalability challenges of writing profiles in parallel; and (2) I/O overhead while reading those profiles using web-based or standalone visualization systems. Several profiling tools [10], [11], [9] exist for capturing performance-related metrics from parallel applications. Although these tools are flexible and mostly easy to integrate into applications, existing solutions strive to reduce the runtime overhead of the profiler (e.g., annotation overhead), largely ignoring the file I/O overhead and the size of the profiles. However, the I/O of profiling reports can be expensive for large-scale parallel applications with hundreds or thousands of processes (see Figure 7). State-of-the-art profiling systems like Caliper [10] and HPCToolkit [12] either support single-file I/O or file-per-process I/O or both, which are known to have limitations at higher scales [13], [14]. The analysis and visualization systems for such profiles may incur significant I/O overhead, particularly for profiles collected at high scales. This is primarily because these systems typically operate in memory-constrained environments, such as web browsers, which can further contribute to the overall performance cost. Most existing profiling frameworks do not account for this overhead, often producing profile data formats with many redundancies and a high footprint.

To meet the scalability needs of performance profiling systems, we developed *TinyProf*, a *lightweight* and *scalable* parallel profiling system. *TinyProf* is designed to collect performance metrics effectively from all running processes with minimal overhead. It is an instrumentation-based profiling system that facilitates annotating the HPC code, e.g., loops and code snippets, and logging the associated application-specific metadata. The profiles generated by *TinyProf* contain compre-

hensive information required to perform essential performance analysis tasks. The included data encompasses call sequences of the profiled code, enabling users to gain insights into the program's structure. The profiles also provide detailed data for each process, allowing for an in-depth examination of process-specific behavior. *TinyProf* produces output files that can be easily converted to formats compatible with widely-used analysis and visualization tools, such as CallFlow [3], [15] and Hatchet [16], [17], through the use of a straightforward conversion script.

*TinyProf* addresses data movement challenges at scale with three technical components: (i) an efficient in-memory data structure (Section IV), (ii) a customizable three-phase I/O strategy (Section V), (iii) and a compact data format (Section VI). The in-memory data structure consolidates all profiled code regions, referred to as events, along with their associated metrics. This aggregation process minimizes redundancies within each individual process, ensuring efficient processing of the profiling data. The three-phase I/O approach consists of three phases: the events synchronization phase, the data aggregation phase, and the file I/O phase. These three phases work together to generate amenable I/O access patterns that can scale to high core counts. The compact file format minimizes profile sizes, reducing the burden on I/O for both parallel profiling and downstream performance analysis and visualization applications. Our contributions can be listed as follows:

- Designed a data structure that enhances *TinyProf*'s scalability by reducing communication overhead during the data aggregation stage, a component of the three-phase I/O approach.
- Developed a tailored three-phase I/O system that efficiently scales parallel file I/O for performance data to high core counts. ($100\times$ faster than Caliper at $32k$ processes).
- Designed a lightweight data format for the generated logs that minimizes metadata, leading to a smaller storage footprint and faster load times. (on average $3\times$ lightweight than Caliper).
- performed a comprehensive evaluation of *TinyProf* using real-world applications [14], [18] on the Theta supercomputer at Argonne National Laboratory. The results demonstrated the feasibility of continuous introspection, with an overhead of less than $5\%$ when running the application on $32k$ processes.

*TinyProf* has overhead similar to in-situ analysis tasks, which are heavily adopted in production. Therefore, we want to position *TinyProf* as a low-overhead, in-situ-profiling tool that can potentially be run with any parallel application by default without compromising performance, thus facilitating *continuous* introspection for performance analysis. Additionally, our work is open-source, and the methodologies detailed in the paper are versatile and could be applied to other tools (such as Caliper) to improve their runtime performance.

| Profilers | P-strategy | Scale (Processes) |
|---|---|---|
| HPCToolkit [12] | S | 10k |
| Open\|SpeedShop [19] | S | 256 |
| Score-P [11] | I | 10k |
| TAU [20] | I | 16k |
| Timemory [9] | I | - |
| INAM [21] | I | 4k |
| Caliper [10] | I | 256 |

TABLE I: Summary of a subset of existing popular profiling (P) tools. P-strategy denotes the profiling strategy of tools, including statistical sampling (S) and instrumentation (I). Scalability represents the largest number of processes or threads reported in the corresponding paper.

## II. Related Work

This section comprehensively reviews the state-of-the-art profiling tools and systems commonly employed in high-performance computing (HPC) environments. We begin by summarizing a subset of existing popular profiling tools in Table I, also showing the total number of processes/threads reported in each of their foundational papers.

***Profiling tools classification:*** Profiling tools collect fine-grained execution profiles and generate calling contexts to help facilitate several performance analysis tasks, such as load balancing analysis detection [22], roofline analysis [23] or variability analysis [3], [24]. Numerous profiling tools have been developed for performance profiling of parallel applications [12], [10], [9], [25], [26]. Broadly, profiling tools can be classified into two categories: (1) *statistical sampling*, where system-generated interrupts are triggered using either time-based or event-based sampling, and (2) *instrumentation*, where measurement probes are placed directly either in the source code or by modifying the generated binary program.

HPCToolkit [12] and Open|SpeedShop [19] use statistical sampling to collect measurement data through interval timers, hardware counters, and overflow interrupts. Choosing an appropriate sampling interval is challenging since a low sampling rate can miss critical information, while high sampling can generate large profiles. This can further degrade the performance analysis experience as users must profile once again to improve the data quality. In contrast, instrumentation profilers like Score-P [11], TAU [20], Timemory [9], INAM [21] and Caliper [10] are flexible, customizable, and incur fewer logging overheads. They employ source-code annotation and adopt several compiler mechanisms to record the performance data across several domains (*e.g.,* hardware, application, communication). The annotation functionality allows users to adjust the granularity at which profiles are generated (*e.g.,* module, class, function, or a specific line in code). However, the usability of these profilers is affected by the steep learning curve needed to tweak the performance collection process to perfection. This incurs considerable development time and training costs. Caliper, for example, is a cross-stack, general-purpose introspection framework that provides users with multiple data-collection strategies based on a user-provided policy at runtime. It is quite powerful and can be used to collect the

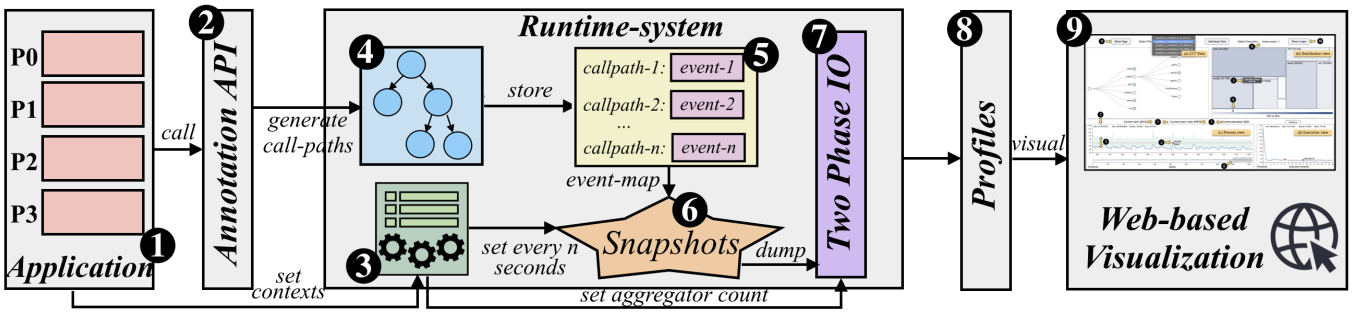


Fig. 1: The workflow of *TinyProf*. An application begins by setting *one-time, global* application-specific contexts (③). Then the *annotation API* (②) invokes a series of profiled code regions (events). The runtime is responsible for generating the unique key called as *callpath* of every event (④) which then gets stored in an *event-map* (⑤). *TinyProf* provides the ability to *snapshot* (⑥) the entire state of performance metric every $n$ seconds, an important fault tolerant measure. To reduce the file I/O overhead, we have developed a three-phase I/O technique (⑦) to dump performance metrics in a customized compact file format (⑧). The profiles can then be read by most existing visualization tools (⑨).

entire software stack, all the way to the operating system. Caliper also offers built-in measurement configurations to make it easier to use, such as *loop-report*. However, these configurations report the basic profiles independently, such as time-series information for loops or a region time profile across all processes. To merge these two profiles into a single file, as our profiler does, it is necessary to create a custom configuration file.

***I/O overhead of profilers:*** At large scales, file I/O plays a crucial role in the performance of profiling tools, as demonstrated in Figure 7, where it constitutes most of the runtime. However, most existing profiling frameworks lack an optimized file I/O system. They typically employ file-per-process I/O, single-file I/O, shared-file I/O, or their combination (e.g., Caliper [10] uses both file-per-process I/O and single-file I/O). In file-per-process I/O, each process writes to a separate file. Single-file I/O involves one process collecting data from all others and writing a single file. Shared-file I/O permits all processes to access and write to a single shared file. However, it is well known that none of these I/O strategies scale effectively at high scales [14]. Our work presents a customized three-phase I/O scheme to reduce the overhead of parallel file I/O.

***I/O overhead of post-hoc analysis:*** File I/O overhead can also be a significant penalty for performance analysis and visualization systems, especially when dealing with profiles of large-scale runs. The I/O challenge is exacerbated for visualization systems that run entirely in resource-constrained environments, such as web browsers. The generated profiles typically contain performance data across all processes, and the data for each process must include specific program entities with many operation iterations (e.g., repeated executions and loops), resulting in numerous amounts of data at high scales. Analyzing individual process behavior and performance variability for various program entities is essential and required by performance visualization systems (e.g., Callflow [15]). In fact, the collected data includes massive local (within each process) and global (across processes) redundancies. However, although existing profiling tools have devised various data formats for profiles, they tackle only a limited number of redundancies. For example, Hatchet [16] utilizes unique integers to encode event names to reduce the size of logs. This paper introduces a compact data format that systematically eliminates all redundancies in profiles to minimize their size.

## III. Challenges with profiling at high scales

A robust performance analysis framework consists of two essential components: first, a profiling mechanism that collects performance metrics from large-scale applications, and second, an analysis and visualization toolkit that enables users to investigate and gain insights from the gathered data. Both components present unique I/O requirements. Profilers gather performance metrics simultaneously while parallel HPC applications are running, which requires an efficient parallel output mechanism across nodes to store the collected data. In contrast, visualization is an interactive process performed on individual workstations or through web browsers, necessitating selective and on-demand retrieval of the relevant data. Two major challenges arise when scaling performance analysis frameworks. Firstly, profilers must minimize the I/O overhead incurred while writing profile data, considering the resource contention with the target application (Section III-A). Secondly, visualization tools must provide quick and responsive access to the collected data to enable meaningful post-hoc analysis and insights (Section III-B).

### A. Parallel file I/O challenge at scale

Instrumentation-based profilers, including ours, collect performance data and application-related metadata; unlike other applications, they do not have to deal with huge volumes of data. However, the performance of parallel I/O depends not only on the total data size under consideration but also on a range of other factors, such as file access pattern, burst size, number of concurrent file accesses, file size, and count. A successful parallel I/O strategy must be both flexible and

tunable, striking a balance across all these parameters to effectively scale to high core counts. Unfortunately, existing frameworks like Caliper and HPCToolKit lack this adaptability and do not scale gracefully to high core counts (see Table II). A low-overhead parallel file I/O strategy becomes even more essential for profiling tools aspiring to offer *continuous introspection*, as they generate performance profiles at regular intervals throughout application execution.

Popular I/O strategies that are used in existing profiling systems include file-per-process I/O, single-file I/O, and shared-file I/O (detailed in Section II, *I/O overhead of profilers*). However, these strategies are known to suffer from performance issues at scale. Additionally, shared-file I/O, built on top of MPI collective I/O, is primarily used for dealing with large volumes of data in binary format. This I/O method is not suited for our needs as the collected metadata by our system is typically composed of different data types (making conversion to binary or raw format cumbersome), and profiles are preferred to be written in a certain format (e.g., JSON) that can be analyzed directly.

In the parallel I/O community, subfiling [27] and data aggregation [28] are two known techniques that mitigate the challenges associated with file-per-process and single-file I/O. With subfiling, the number of files outputted becomes a tunable parameter between 1 and $P$ (process count). With data aggregation, only a selected set of processes (known as aggregators) perform all the necessary I/O operations. This happens in two phases, where all processes transmit their data to chosen aggregators, which then perform all necessary I/O. This process also guarantees optimal I/O burst size.

We aim to develop a parallel I/O system for profiling data, incorporating sub-filing and data aggregation techniques. To implement an effective two-phase I/O technique, we must reduce both the communication overhead associated with aggregating data from processes to aggregators and the file I/O overhead associated with writing data to disk. Reducing the communication overhead necessitates an efficient in-memory data structure (see Section IV) that can minimize the exchanging of local data within each process. In addition, HPC applications run at large scales, where processes or threads typically execute a similar set of tasks with just a few exceptions. This leads to a significant amount of redundant performance data being collected across processes and threads, referred to as global redundancies. To address this, we incorporate an event synchronization phase with a two-phase I/O scheme, resulting in a three-phase I/O scheme (see Section V). This phase synchronizes the event (profiled code region) list along with their calling sequences across all processes to eliminate global redundancies. It minimizes the profile size, consequently reducing the file I/O overhead of writing data.

### B. Storage footprint of profile

To expedite data access when performing post-hoc analysis, minimizing the size of the output profiles is essential. This can be done by adopting a compact data format that eliminates redundancies (both local and global) from the profiles. All existing profiling tools must, at least, capture the names and runtimes of profiled events while also keeping track of their calling sequences. Consequently, each entry in the output file should contain an event's name, parent, and runtime. Furthermore, a profiler must be able to tackle different program entities, including (1) repeated executions and (2) loops. To accurately assess the performance of parallel applications, HPC developers typically run programs multiple times to remove ambient effects like OS noise and network instability. Additionally, many HPC applications rely heavily on loops and have temporal behavior where certain functions get executed repeatedly. These two entities generate numerous profiling entries for a certain event with the same name and other possible measured metadata, resulting in many local redundancies.

Current compact formats address specific instances of redundancy by assigning unique IDs to event names and repeatedly using these encoded IDs instead of the full event names. They often create distinct entries for repeated executions, loops, and processes/threads, each featuring a unique execution ID, loop iteration, and rank/thread ID (see Figure 3(a)). Thus, this leads to an undesirably inflated output file size. To address this, we present an efficient in-memory data structure (Section IV), which is instrumental in reducing local redundancies within each process/thread. We then propose the customizable three-phase I/O scheme (Section V), eliminating global redundancies across processes/threads. These two techniques pave the way for the compact data format (Section VI), yielding minimal file sizes for profiles, low storage overhead, and fast data access in post-hoc analysis.

Listing 1: TinyProf minimal API listing

```
TinyProf::Profiler(string filename, double
    io_frequency, string selfkeys);
TinyProf::Event(string event_name);
TinyProf::Event(string event_name, int is_common);
TinyProf::Event(string event_name, int is_common,
    int loop);
TinyProf::Event(string event_name, int is_common,
    int loop, string selfvalues);
TinyProf::~Event();
TinyProf::flush();
```

## IV. Efficient In-memory data structure

In this section, we delineate an efficient in-memory data structure, termed the *event map*, to efficiently organize local data within each process. This structure eliminates local redundancies and minimizes the communication overhead for the customizable three-phase I/O scheme. To demonstrate the *event map* clearly, it is necessary first to describe the process of generating measured performance data through our simple source-code annotation API (Section IV-A). Following that, we elaborate on the design of the *event map*, which serves as an effective in-memory storage for the collected metrics (Section IV-B).

```
0   // example code
1   void compute() {
2     for (int i = 0; i < 3; i++) {
3       // M is a variable for memory cost
4       Event e ("compute", 1, i, "COMP;M");
5       // do some computation
6     }
7     if ( rank % 2 == 1) {
8       Event e ("update", 0, "COMP;M");
9       ...
10    }
11  }
12  void exchange() {
13    Event e ("exchange");
14    for (int i = 0; i < 3; i++) {
15      { Event e ("comm", 2, i);
16        MPI_Sendrecv(); }
17      ...
18      { Event e ("write", 2, i,"IO;M");
19        MPI_File_write_at();
20      }
21    }
22    ...
23  }
24  void main() {
25    TinyProf::Profiler("out", 5, "tag:0;mem:1");
26    for (int i = 0; i < 2; i++) {
27      Event e("main");
28      compute(); exchange();
29      ...
30    }
31    TinyProf::flush();
32  }
```

Fig. 2: Example of MPI code using our annotation API.

### A. Profiling performance data using annotation API

*TinyProf*'s annotation API can be seen in Listing 1. One of the main components of the API (Figure 1②) is a C++ `Event` class. It is instantiated for any profiling code region enclosed within a pair of curly braces and designated as an event. We leverage the *Resource Acquisition Initialization* (RAII) feature of C++ to guarantee that code regions within any local scope invoke the constructor and destructor of the `Event` class when entering and departing the scope, respectively. To be easy to use, an instant of the `Event` class only requires a `name` of an event in most cases (see first constructor in `Event` class in Listing 1). To accommodate different program entities, such as executions and loops, the `Event` class incorporates two key features. Firstly, it automatically detects each execution by monitoring the invocation of the first event, typically the `main` function. Secondly, it offers an optimized argument `loop` for loop events, providing two distinct measurement modes to cater to various requirements: (1) measuring the runtime for each individual iteration, and (2) calculating the cumulative runtime across all iterations (see the third constructor for `Event` class in Listing 1). In addition, to minimize the communication overhead in the synchronization phase (see Section V-A), the `Event` class includes an optional argument `is_common`, which specifies whether the event is invoked by all processes (`1`) or not (`0`) (see the second constructor for `Event` class in Listing 1).

The runtime system manages the details for every event class instance, storing its *callpath* and its runtime. The *callpath* of an event comprises all the preceding events, originating from the root event (e.g., *main*), which is automatically generated by our API (Figure 1④). As an illustration, consider the event *comm* depicted in Figure 2 (line 15); its *callpath* is *main*<*exchange*<*comm*.

The *Profiler* class is another crucial component of the runtime system, tasked with handling the data gathered in the backend in an efficient manner. It is instantiated as a singleton class and invoked by applications once at the beginning to establish application-specific contexts, including the profile name, I/O frequency dumps, and an optional user-defined list of measured metrics (Figure 1③). The profile name determines the filename for the dumping profiles, the I/O frequency sets the intervals at which intermediate checkpoints are saved, and the user-defined list may include any metrics specific to the measured application.

***Customizing profiling performance metrics:*** Since different HPC applications may require the profiling of unique performance metrics, our annotation API allows users to tailor measurements for collecting application-specific data. These metrics should be defined when setting up the application-specific context (*TinyProf::Profiler(..., string selfkeys)*) in the format *name:is_consistent*. *name* is the name of the performance metric, which can be any arbitrary string. *is_consistent* indicates whether this metric maintains a constant value for an event, regardless of the processes or distinct program entities (e.g., different iterations in a for loop) involved. For example, the *callpath* of an event is a *consistent* metric, while its runtime is an *inconsistent* metric.

***Example:*** In Figure 2, the illustrative MPI code begins by initializing the *Profiler* class and configuring the application-specific context (line 25). This setup includes specifying the output profile names, establishing intermediate checkpoints every 5 seconds, and selecting two specific performance metrics for customization. The *tag* metric is used to aggregate events that share the same high-level semantic meaning across different code snippets (e.g., *compute-only*, *network-comm*, and *file-I/O*). Its value is consistent (`0`) across different processes and program entities. On the other hand, the *mem* metric represents the memory usage for each event, but its user-specified value ($M$) can vary with different executions, loop iterations, and processes, making it inconsistent (`1`). The *Event* class is then instantiated multiple times for various profiled code regions (events), highlighted in green. The only necessary argument for these instances is the event's name (see line 27). Two modes for loop events are distinguished by 1 and 2 modes (line 4 vs. 15). Also, the inclusion of values for *tag* and *mem* is optional for events (line 15 vs. 18). An example of an event that is not common across all processes is shown in line 8, where the *is_common* argument is set to 0. Ultimately, the program executes the `flush()` function to transfer all the measured data into the profiles for analysis.

Additionally, because the annotations are independent of one another, they can be incrementally added anywhere in the software stack. Having the freedom to annotate the code man-

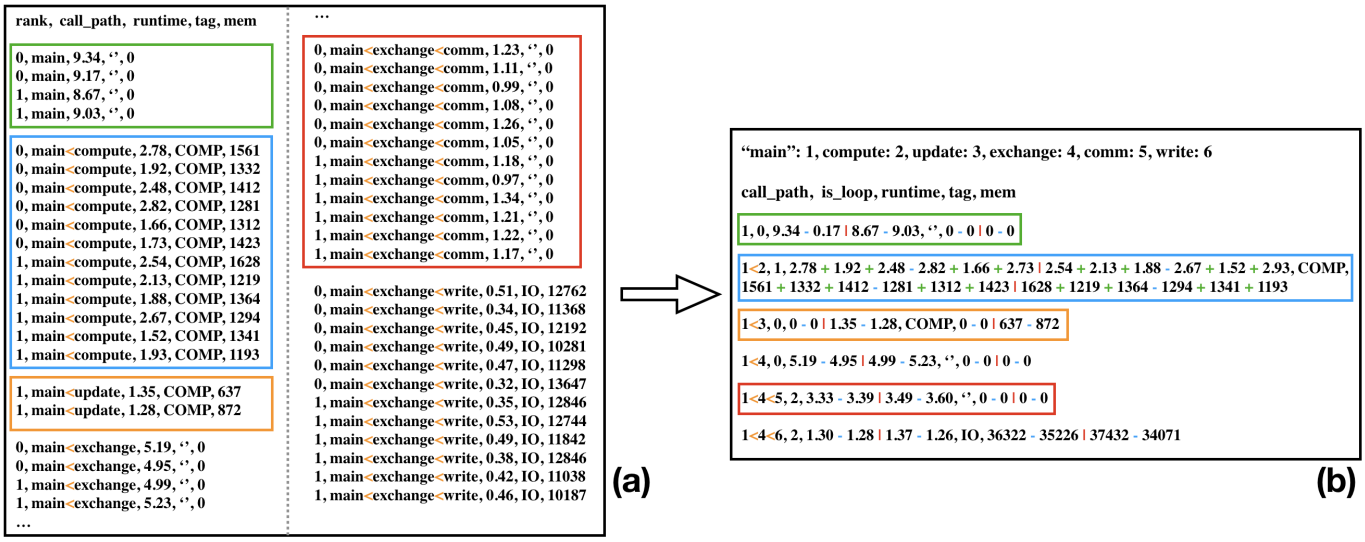


Fig. 3: (a) A profile that aligns with the example code in Figure 2, presented in the standard CSV format; (b) a corresponding profile in our more condensed format. The colored boxes in section (c) are created by merging the similarly colored boxes from section (b).

ually offers additional flexibility, as application developers can decide to annotate and, hence, profile only the performance-critical source-code regions or use coarser granularity for less important events to reduce the annotation overhead.

### B. Event-map

As we have seen, an event represents the most granular level of performance measurement. Consequently, the final performance data consists of all recorded events. For most HPC applications, the code annotation API is bound to create large performance data, consisting of several thousands of events. It is therefore essential to have an efficient in-memory data structure to store the details of an event, its callpath, and its measurement metrics. We implement this internally using an ordered hash map called the *event-map* with a key-value format (Figure 1⑤). The key of the *event-map* is the *callpath*, and the corresponding value is a list of attribute-class objects. For example, the callpath for the event *comm* in Figure 2 (line 15) will be *main<exchange<comm*. *Callpaths* of all events combined are used in constructing the calling context tree (CCT) when the profiles are loaded for performance analysis, revealing the clear code structure of the program. The attribute class stores all relevant data for an event, including its runtime and other self-defined metadata.

Executions involve repeating the entire codebase, while loops iterate over specific functions; both lead to multiple events sharing the same *callpath*. Our *event-map* data structure leverages this characteristic to eliminate data redundancies. Instead of creating a new key-value pair for each event with an identical *callpath*, we append the event to the existing list of attributes associated with that *callpath* (see Figure 4). This approach eliminates the need to store repeated *callpaths* in memory. Furthermore, we also eliminate redundancies in the corresponding list of attribute-class objects, which is the

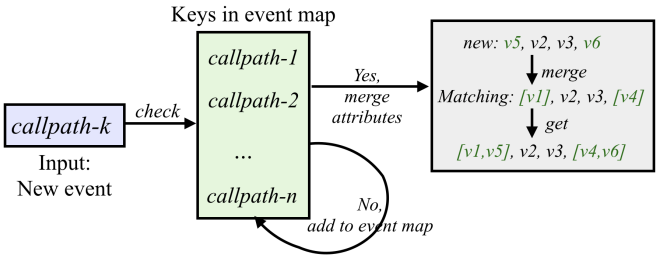


Fig. 4: Removing local redundancies using event map within each process. For a new profiled event, *TinyProf* initially verifies whether its *callpath* exists in the current *event map*. If so, *TinyProf* consolidates its attribute list with the existing one in the *event map*. Note that, *TinyProf* merges the *inconsistent* attributes, e.g., v5 and v6, highlighted in green. Otherwise, the new event is appended to the event map.

matching value of the *callpath* in the *event-map*. As shown in Figure 4, we only concatenate the values of *inconsistent* attributes highlighted as green. For example, the lines in Figure 3(b) highlighted with green boxes only concatenate attributes *runtime* and *mem* for event *main*.

## V. Custom three-phase I/O scheme

One of our main contributions is developing a scalable, low-overhead, three-phase file I/O scheme that can facilitate the profiling of parallel applications at high scales. The three phases correspond to (see Figure 5): (1) the events synchronization phase (Section V-A); (2) data aggregation (Section V-B); and (3) file I/O (Section V-B).

### A. Efficient events synchronization (phase 1)

Processes or threads in HPC applications may concurrently perform different tasks, leading to different lists of profiled

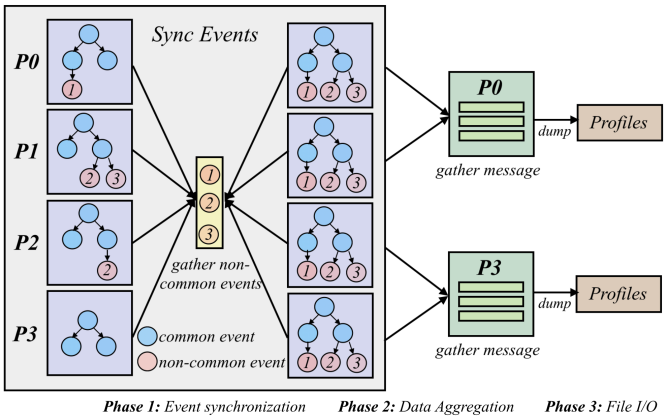


Fig. 5: The three-phase I/O strategy comprises (1) event synchronization, (2) data aggregation, and (3) file input/output. During the event synchronization phase, only the unique events, which are indicated in red circles (①, ②, and ③), are collected. In the data aggregation phase, two aggregators ($P0$ and $P3$) are chosen to gather data from other processes. Finally, these aggregators are responsible for writing the collected data to profiles.

events and their associated calling sequences. For example, in Figure 5, each of the four processes maintains a unique event list and the corresponding calling sequences. The *common* events that are executed by all processes are colored blue, while the *non-common* events that are called by only a subset of processes are colored red. To eliminate global redundancies across processes, reducing I/O overhead and log size, we execute an event synchronization phase to maintain a consistent view of all events and their calling sequences (see phase 1 in Figure 5).

The implementation of this synchronization involves three stages: (1) *callpaths* synchronization across processes; (2) event names encoding within each process; and (3) rearranging local data within each process. The initial stage is synchronizing the *callpaths* of events (keys of the *event map*) using `MPI_Allgather`. All processes are then guaranteed to possess a consistent view of the events list associated with calling sequences by merging all *non-common* events from other processes. Only the *non-common* events from all processes are gathered to minimize the communication overhead. Recall, from subsection IV-A, *common* events are guaranteed to be present on every process, and would therefore already be part of the *event map*. Every process scans through these *non-common* events to add them to the local *event-map* if it already does not exist. The relevant attributes of these events are labeled as null, indicating that they do not represent actual process events.

With this consistent events list, the second stage encodes the event names (strings) into unique integral indices, facilitating the encoding of *callpaths* of events. This further reduces memory and file storage overheads. The third phase eventually rearranges the local data within each process according to the consistent events list by padding the relevant attributes of *non-common* events with 0, indicating that they do not have these events. As an illustration, the event *update* in Figure 2 (line 8) is exclusively invoked by rank 1. Its corresponding output is highlighted in orange in Figure 3(b), with data padded for rank 0. The purpose of this padding is to transform the *non-uniform* problem (`MPI_Gatherv`), which involves varying message sizes in the following aggregation phase, into a *uniform* problem (`MPI_Gather`) with consistent message sizes. While the *uniform* approach may exchange slightly more data with others, its communication overhead is lower than that of the *non-uniform* approach.

### B. Data aggregation and file I/O (phases 2 and 3)

Following event synchronization, a data aggregation phase is performed (see phase 2 in Figure 5) to collect data from processes for selected aggregators. This follows a file I/O phase in which each aggregator writes to an independent profile file (see phase 3 in Figure 5).

Data aggregation evenly partitions processes based on the predefined number of aggregators, with the first process in each group designated as the aggregator. To reduce the communication overhead, we concatenate values of *inconsistent* attribute for all events into a string-typed message, necessitating only a single communication. Moreover, the event synchronization phase (Section V-A) ensures that all processes share the consistent view (order) of the events list; therefore, there is no explicit need to transmit the keys of *event map* during the communication phase. Finally, we use `MPI_Gather` to gather the messages from processes since we converted the *non-uniform* problem into a *uniform* problem in Section V-A. We do this based on the observation that the degree of imbalance in these messages is often low due to a small number of *non-common* events. Also, *non-uniform* communication requires an extra communication phase to gather the necessary metadata, which adds an extra cost.

Finally, each aggregator writes all its data to an independent file using just one write operation, ensuring a large I/O burst size (see phase 3 in Figure 5). Therefore, the number of output profiles equals the number of aggregators.

## VI. Compact Data Format of Profiles

As demonstrated in Section III-B, there are several local and global redundancies in the collected data. We then present an efficient data structure (Section IV) and a three-phase I/O scheme (Section V) to remove local and global redundancies, respectively. In this section, we describe the process of translating them into our compact file format to minimize the size of the output profiles.

Existing profiling methods have created a variety of data formats for profiles in *JSON* and *CSV*. Meanwhile, *CSV* is a more compact format than *JSON* since it uses a headline for all entries instead of repetitive attributes in every entry. Thus, in this section, we compare our compact data format (see Figure 3(b)) with the standard *CSV* format (see Figure 3(a)). The standard *CSV* format (see Figure 3(a)) stores

| A | P | T | A-cost | N-cost | I-cost | O (%) |
|---|---|---|---|---|---|---|
| FFT-20 | 4096 | *TinyProf* | 57.70 | 0.0031 | 0.056 | 0.102 |
| FFT-20 | 4096 | Caliper | 57.40 | 0.0058 | 30.89 | 53.82 |
| FFT-3000 | 512 | *TinyProf* | 272.61 | 0.529 | 0.086 | 0.226 |
| TC | 1024 | *TinyProf* | 161.36 | 0.123 | 0.120 | 0.150 |
| TC | 1024 | Caliper | 163.08 | 0.414 | 1401.19 | 1718.55 |
| TC (ET) | 4096 | *TinyProf* | 246.45 | 1.250 | 2.382 | 1.474 |

TABLE II: Profiling with applications. A: application, P: process count, T: profiling tool, N: annotation, I: I/O, O: overhead. All costs are presented in seconds (s). The overhead ($O$) is calculated by *((N-cost + I-cost) / A-cost* $\times$ 100*)*.

performance data across executions, loops, and processes as separate entries, while our format combines them into a single line. Figure 3 depicts how we aggregate the data using different colored boxes. For example, the entire program is run twice, and the event *"main<compute"* is a three-iteration loop. Each of the two processes generates three loop-iteration entries twice (highlighted in blue boxes in Figure 3(a)). The four blue boxes are then combined into a single line in Figure 3(b) (also indicated with a blue box). Note that, as we indicated in Section IV-B, only the values of *inconsistent* attributes are concatenated. Other colored boxes in Figure 3 (a) and (b) follow the same combination pattern.

In addition, to be able to parse the data correctly when performing post-hoc analysis, we must add delimiters between timings to distinguish between the specific program entities. In our format (Figure 3(b)), the *inconsistent* values are grouped by processes (increasing rank order) and separated by the delimiter "|" at the very outer level. Next, the values are grouped by repeated executions (increasing iteration order) using the "−" delimiter. Finally, they are grouped by loops "+". An example of our format can be seen in Figure 3(b). When compared to a standard *CSV* format(Figure 3(a)), our approach is much more compact, with no redundancies.

## VII. Case studies

In this section, we first demonstrate the performance of our profiling system with two open-source applications: (1) a transitive closure (TC) built on top of balanced parallel relational algebra (BPRA) [18], and (2) a parallel Fast Fourier Transform (FFT) built on top of the FFTW-3 [29]. We then demonstrate the scalability and efficacy of our profiling framework for identifying bottlenecks at scale in a parallel I/O library [14].

FFTW3 is a widely used open-source library that computes the discrete Fourier transform (DFT) in parallel using MPI. We repeatedly run the parallel FFT application (the size of DFT complex $N = P^2 \times 1024$ bytes) with 20 and $3,000$ iterations, which we refer to as *FFT-20* and *FFT-3000*. *FFT-3000*, runs for approximately 5 minutes and represents a typical application running for many iterations. We profiled *FFT-20* and *FFT-3000* with *TinyProf* and *Caliper*, shown in the first three rows in Table II. For *FFT-20*, *TinyProf* outperforms *Caliper* significantly at $P = 4{,}096$ ($Overhead$: $0.102\%$ vs. $53.82\%$), where the total cost is dominated by I/O ($0.056$s vs. $30.89$s). For *FFT-3000*, we only report the result at

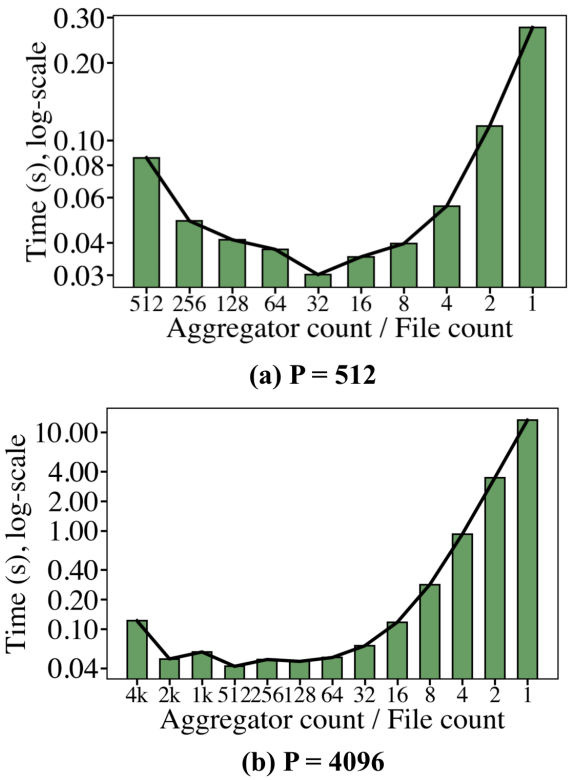


Fig. 6: Results for the performance of the three-phase I/O with varying number of aggregators when (a) $P = 512$ and (b) $P = 4096$.

$P = 512$ for *TinyProf* ($3^{rd}$ row) since the I/O time of *Caliper* was too huge to capture. Our tool still runs well for a nearly 5-minute execution with only $0.226\%$ overhead.

Balanced parallel relational algebra (BPRA) [18] is an open-source library that maps database tuples to MPI processes in a dynamically balanced manner. A classic application built on this library computes an input graph's transitive closure (TC) [30]. We use a graph of $412{,}148$ edges obtained from the Suite Sparse Matrix collection [31] as our input graph. The performance results of profiling with *TinyProf* and *Caliper* at $P = 1{,}024$ are shown in the $4_{th}$ and $5_{th}$ rows of the table. The results demonstrate that *TinyProf* is significantly faster than *Caliper*, whose I/O overhead dominates total runtime ($1401.19$s). This performance can be attributed to the file size and the I/O scheme. For example, Caliper's file size at $1{,}024$ processes is $162$MB compared to $27$MB of *TinyProf*. Furthermore, Caliper uses no file I/O optimizations and has many small-size sub-optimal file accesses. In addition, we enable the error-tolerance mechanism for this application (referred to as *TC(ET)*), which dumps an intermediate output every $2$ seconds (in total, $122$ outputs are dumped). The result is shown in the last row of the table. Our I/O overhead is still pretty low ($2.382$s).

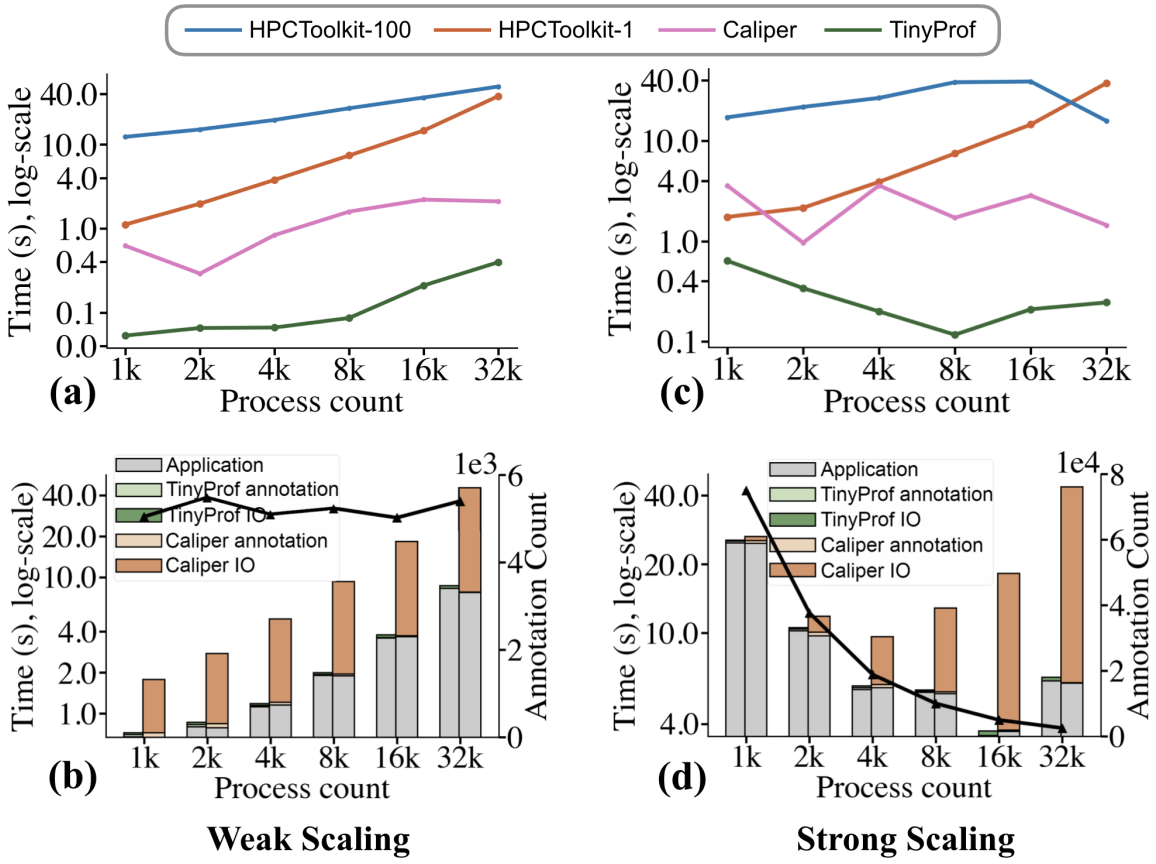


Fig. 7: Results for weak (left) and strong (right) scaling are presented. In figures (a) and (c), *TinyProf*'s profiling cost *(= total-cost - application-cost)* demonstrates superior efficiency over HPCToolkit-100, HPCToolkit-1, and Caliper. Figures (b) and (d) detail application, annotations, and I/O overheads. Notably, *TinyProf* outperforms Caliper, with a particular advantage in reducing I/O costs. The black lines depicted in figures (b) and (d) represent the frequency of calls to the annotation API.

## VIII. Evaluation

In this section, we evaluate the performance of *TinyProf* via a series of experiments using a parallel I/O based application. The goal of our experiments is to demonstrate the salient features of our profiling framework– low performance and storage overhead, eventually positioning *TinyProf* as a system to enable continuous introspection for large scale applications.

***Target platform:*** We conducted a series of experiments on the *Theta* [32] supercomputer at Argonne National Laboratory to evaluate the efficacy and performance of both *TinyProf*'s profiling. *Theta* is a Cray machine with a peak performance of 11.69 petaflops, offers 281,088 compute cores, 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM, and 10 PiB of storage.

***Target application:*** We use a parallel I/O library [14], [33] that enables I/O for multiresolution data format as a use-case. With increasing high-resolution scientific data being generated, it puts pressure on existing analysis tools to derive scientific insights. One solution to address it is to transform the data on the fly (before writing) to a multiresolution layout that supports progressive access to data at varying scales. Additionally, data can be compressed before the actual file I/O to reduce the burden on data movement. Although this compressed-multiresolution data layout has obvious advantages in fast data-movement, it creates a load imbalance.

The data movement pipeline of the parallel I/O system for a compressed, hierarchical data layout comprises five phases: (1) patch distribution, (2) hierarchy creation, (3) compression, (4) aggregation, and (5) actual file I/O. Patch distribution partitions the entire simulation data into smaller patch sizes (each patch is a data chunk) that eventually get transformed using discrete wavelet transform and compression [14]. Compression creates imbalanced data-loads across processes, as some processes can achieve better compression ratios than others. Data aggregation [28], which is a commonly used parallel I/O optimization technique, if performed agnostic of this imbalance, will lead to sub-optimal performance.

***Experiments configuration:*** To evaluate the performance of the profiling capabilities of *TinyProf*, we conduct four sets of experiments: (a) benchmark the three-phase I/O to demonstrate the importance of the tunable data aggregation scheme (Section VIII-A), (b) perform weak and strong scaling experiments to compare performance against Caliper and HPCToolkit (Section VIII-B), (c) overhead analysis, to compute the overhead

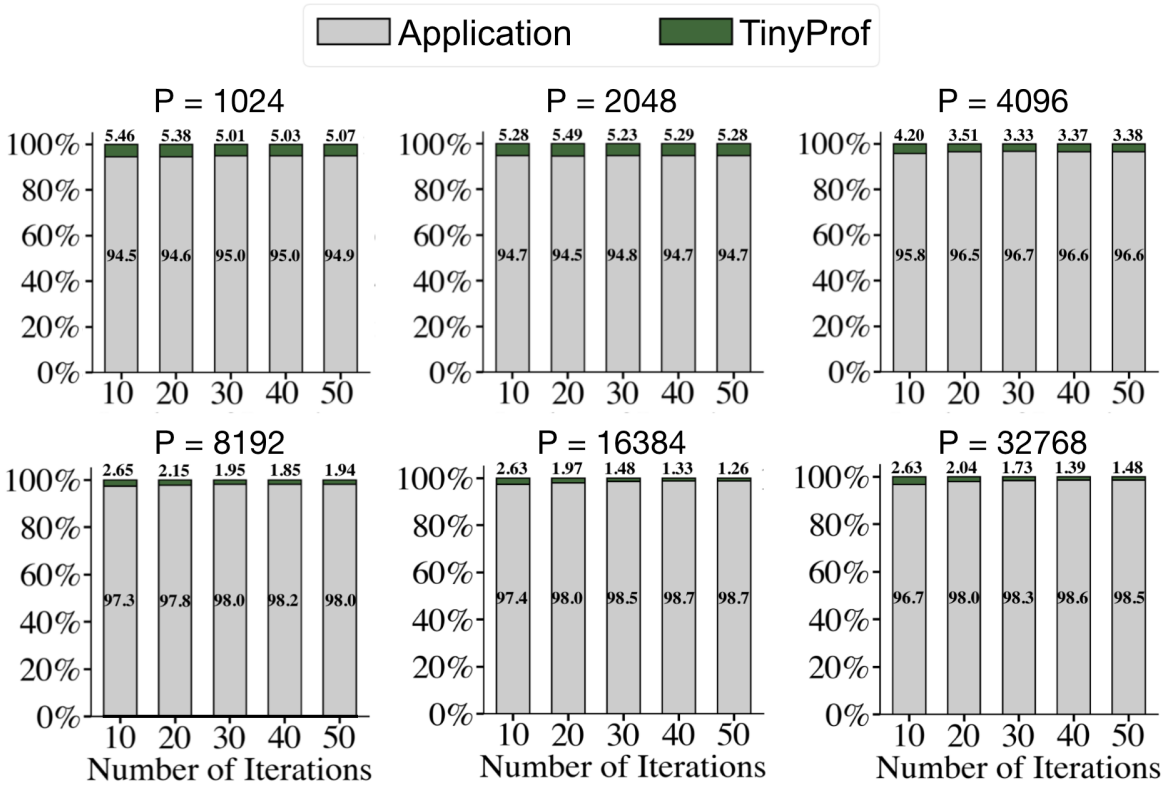


Fig. 8: *TinyProf* incurs minimal overhead (around 3%) at varying scales and for different iteration counts.

of profiling in a production environment (Section VIII-C). and (d) I/O overhead of reading, to demonstrate the efficacy of our compact data format for post-hoc analysis systems against Caplier (Section VIII-D). We used the MPI-everywhere programming model that maps one rank per core (e.g., we have 64 processes per node on Theta). Additionally, for a fair comparison with Caliper and HPCToolkit, we refrain from doing temporal checkpointing feature in our experiments. Instead, the program only outputs the final profiles at the end of its execution.

### A. Tunable Three-phase I/O

Caliper and HPCToolkit support two parallel I/O modes: single-process I/O (one process gathers all the data and writes a single file) or file-per-process I/O, while *TinyProf* uses a tunable three-phase I/O. We evaluate the efficacy of *TinyProf*'s I/O scheme at two process counts, $P = 512$ and $P = 4{,}096$. At both these scales, we vary the total number of aggregators (and thus the total number of files) and measure the aggregate I/O time. We vary the aggregator count (A) from $P$ down to 1 ($P$, $P/2$, $P/4$, ..., 4, 2, 1), which respectively corresponds to file-per-process I/O and single-file I/O mode. We measure the total time to perform I/O which includes both the inter-process data aggregation time and the actual file I/O time. The results for these two sets of runs are shown in Figure 6. Both results show a similar `U` - shaped trend, where the optimal performance is reached roughly around $A = P/16$ aggregators. These results show the importance of the three-phase I/O, as both file-per-process I/O and single-file I/O demonstrate sub-optimal performance. At $P = 4{,}096$, the optimal I/O time observed at $A = P/16$ is $2\times$ faster than file-per-process I/O and $10\times$ faster than single-file I/O. Although the aggregator count (file count) is a tunable parameter and must be set based on the target file system, we used $A = P/16$ as the default for all following experiments (best for Theta).

### B. Scaling Studies

We perform weak and strong scaling experiments comparing performance of *TinyProf* against Caliper and HPCToolkit. *TinyProf* and Caliper annotate the same regions of code to generate the same list of events, while HPCTookit uses the recommended configuration (by theta) that samples 100 times per second [34] (referred to as *HPCToolkit-100*). Additionally, we also ran HPCToolkit with one sample per second (referred to as *HPCToolkit-1*), a configuration that will have the best running time, but will capture performance data very sparsely. Meanwhile, Caliper's "hatchet-region-profile" configuration is used to dump the profile in hatchet format, resulting in the most compact file format. We generate weak and strong scaling characteristics for the profiles by running the target application in weak and strong scaling modes. In both sets of experiments, the total process counts varied from $1k$ to $32k$. For all three profiling frameworks, we capture both the application time and the profiling time. The profiling time is the time taken by these libraries to collect performance-related metrics, including annotation and file I/O time.

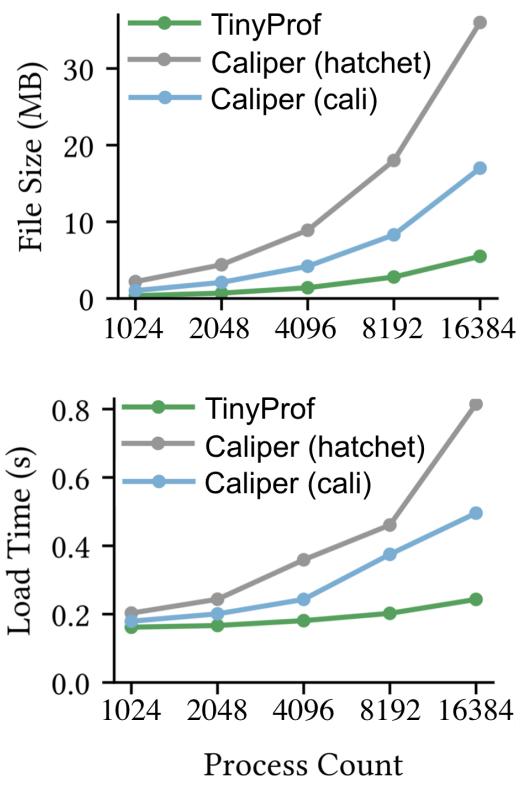


Fig. 9: *TinyProf*'s compact format offers significant gains over Caliper, both for file sizes (top) and reading performance for post-analysis (bottom).

***Results analysis:*** We plot the weak and strong scaling results in Figures 7. We also record the total number of per-process annotations made by Caliper and *TinyProf* for all runs and plot them as black trendlines in Figures 7(b)(d). Figure 7(b) shows that the annotation counts are roughly the same for all process counts in weak scaling runs. For strong scaling runs, as the total global workload remains fixed, we observe a decrease in the total number of annotations with increasing scale. Focusing just on *TinyProf*'s performance, for both weak and strong scaling, we observe near-perfect scaling till $8k$ processes – for strong scaling, we observe a reduction in time with increasing process counts, and we observe roughly the same time with increasing process counts with weak scaling. The observed performance degradation at $16k$ and $32k$ processes can be attributed to communication overhead and workload starvation.

Furthermore, we observe *TinyProf* outperforms both Caliper and HPCToolkit at all scales for both weak and strong scaling. For weak scaling, *TinyProf* (0.397s) is $95.11\times$ faster than Caliper (37.76s) and $5.29\times$ and $123.48\times$ faster than *HPCToolkit-1* (2.10s) and *HPCToolkit-100* (49.03s) at $32k$ processes. The performance trends of *TinyProf* and Caliper can be further explained by breaking down the total profiling time into two components: (1) annotation time, which is the time taken to capture the performance metrics; and (2) I/O time, which is the time taken by the three-phase I/O scheme. We plot this time breakdown along with the application time for Caliper and *TinyProf* in Figure 7(b)(d). Here we can make the following observations: (a) the actual annotation cost is negligible for both Caliper (light orange bars) and *TinyProf* (light green bars); (b) Caliper, which by default follows the single-file I/O paradigm, takes significantly more time than the three-phase I/O of *TinyProf* (dark green bar vs. dark orange bar); and (c) Caliper adds significant overhead to the application (large orange bar on top of grey bar) as opposed to *TinyProf*, which adds negligible overhead (small green bar on top of grey bar). Overall, *HPCToolkit-100* performs poorly, while *HPCToolkit-1* beats Caliper for all process counts. Although *HPCToolkit-1* performs well, it is likely that the amount of data recorded is not enough for accurate profiling.

### C. Overhead Analysis

Here, we evaluate the efficacy of *TinyProf* in a production setting where applications typically run for several time steps. A true test of a profiling system is when it continues to incur low overhead at scale, even when run for several iterations. To evaluate *TinyProf* on this front, we ran the application for a varying number of timesteps and recorded the overhead that *TinyProf* added for all these runs. We varied the total number of iterations from 10 to 50 and ran our experiments at $P = 1k$ to $32k$. The results are plotted in Figure 8. We observe that the overhead added by *TinyProf* remains consistent at around $2-5\%$ for all scales and iteration settings. This can be attributed to three key factors: (a) efficient data structure; (b) optimal parallel I/O using the three-phase I/O scheme; and (c) the compact file format that reduces data redundancies. Therefore, we can also make our case for using *TinyProf* to achieve *continuous introspection* for HPC applications at scale.

### D. I/O Overhead of Reading

To demonstrate the efficacy of our compact data format for post-hoc analysis and visualization systems, we evaluated the I/O overhead of loading the profiles in this section. To make a fair comparison, we only compare the file sizes (Figure 9 top) and loading times (Figure 9 bottom) for the profiles generated by Caliper. Our profiles are easy to convert into the required formats for any web-based or standalone visualization system. To simplify this comparison, we estimate the loading time using D3 [35] read functions for CSV and JSON formats, since D3 is widely used in many web-based visualization systems. Furthermore, Caliper can output the performance data in two formats: a) hatchet (in JSON) and b) cali (a compressed format). Comparing against these two formats, our proposed format (green lines) is both space-efficient and therefore has a lower loading time (45x speedup for $16k$ process counts).

## IX. Conclusion

In this paper, we presented *TinyProf*, a lightweight end-to-end system for profiling the performance of large-scale applications. Our simple annotation API enables easy integration with parallel applications. *TinyProf* incorporates a

custom file format and deploys an efficient in-memory data structure, along with a scalable three-phase I/O scheme, to scale I/O to high process counts. Experimental results demonstrate how the *TinyProf* introduces a smaller runtime overhead and requires less storage when compared to state-of-the-art profiling libraries such as Caliper and HPCToolkit. While *TinyProf* currently mainly supports MPI-based applications, the customizable performance metrics feature of the system can be used to coarsely profile the performance of other shared memory programming models such as OpenMP or CUDA (when run in MPI+X mode). As part of our future work, we look forward to extending *TinyProf* to directly interface and profile such programming models through tools (e.g., OMPT).

## X. Acknowledgement

This work was partly funded by NSF Collaborative Research Awards 2401274 and 2221812, and NSF PPoSS Planning and Large awards 2217036 and 2316157. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on Theta located at the Argonne National Laboratory.

## References

[1] J. Shen *et al.*, "Workload partitioning for accelerating applications on heterogeneous platforms," *IEEE Trans. on Para. and Dist. Sys.*, vol. 27, 2015.

[2] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization." in *EuroVis (STARs)*, 2014.

[3] H. T. Nguyen, A. Bhatele, N. Jain, S. P. Kesavan, H. Bhatia, T. Gamblin, K.-L. Ma, and P.-T. Bremer, "Visualizing hierarchical performance profiles of parallel codes using callflow," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 27, no. 4, pp. 2455–2468, 2019.

[4] D. Skinner and W. Kramer, "Understanding the causes of performance variability in hpc workloads," in *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE, 2005, pp. 137–149.

[5] L. Zheng *et al.*, "Vapro: performance variance detection and diagnosis for production-run parallel applications," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 150–162.

[6] D. Nichols, A. Marathe, K. Shoga, T. Gamblin, and A. Bhatele, "Resource utilization aware job scheduling to mitigate performance variability," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 335–345.

[7] X. Aguilar, "Performance monitoring, analysis, and real-time introspection on large-scale parallel systems," Ph.D. dissertation, KTH Royal Institute of Technology, 2020.

[8] D. Boehme, P. Aschwanden, O. Pearce, K. Weiss, and M. LeGendre, "Ubiquitous performance analysis," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings 36*. Springer, 2021, pp. 431–449.

[9] J. R. Madsen *et al.*, "Timemory: modular performance analysis for hpc," in *Int. Conf. on High Perf. Comp.* Springer, 2020, pp. 434–452.

[10] D. Boehme *et al.*, "Caliper: performance introspection for hpc software stacks," in *Proc. of the Int. Conf. for High Perf. Comp., Net., Stor. and Anal.* IEEE, 2016.

[11] D. an Mey *et al.*, "Score-p: A unified performance measurement system for petascale applications," in *Comp. in High Perf.* Springer, 2011, pp. 85–97.

[12] L. Adhianto *et al.*, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Conc. and Comp.: Prac. and Exp.*, 2010.

[13] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[14] K. Fan, D. Hoang, S. Petruzza, T. Gilray, V. Pascucci, and S. Kumar, "Load-balancing parallel i/o of compressed hierarchical layouts," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2021, pp. 343–353.

[15] S. Kesavan, H. Bhatia, A. Bhatele, S. Brink, O. Pearce, T. Gamblin, P.-T. Bremer, and K.-L. Ma, "Scalable comparative visualization of ensembles of call graphs," *Trans. on Vis. and Comp. Graph.*, 2021.

[16] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proc. of the Int. Conf. for High Perf. Comp., Net., Stor. and Anal.*, 2019, pp. 1–21.

[17] S. Brink, I. Lumsden, C. Scully-Allison, K. Williams, O. Pearce, T. Gamblin, M. Taufer, K. E. Isaacs, and A. Bhatele, "Usability and performance improvements in hatchet," in *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 2020, pp. 49–58.

[18] S. Kumar and T. Gilray, "Distributed relational algebra at scale," in *International Conference on High Performance Computing, Data, and Analytics*, 2019.

[19] The Open—SpeedShop Team, "Open—SpeedShop for Linux," https://openspeedshop.org/, online; accessed 29 January 2022.

[20] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. Jrnl. of High Perf. Comp. App.*, vol. 20, no. 2, pp. 287–311, 2006.

[21] P. Kousha *et al.*, "Inam: Cross-stack profiling and analysis of communication in mpi-based applications," in *Practice and Experience in Advanced Research Computing*, 2021, pp. 1–11.

[22] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–35, 2015.

[23] Y. J. Lo *et al.*, "Roofline model toolkit: A practical tool for architectural and program analysis," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014.

[24] A. D. Malony, S. Ramesh, K. Huck, C. Wood, and S. Shendey, "Towards runtime analytics in a parallel performance system," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 559–566.

[25] S. Brink, M. McKinsey, D. Boehme, C. Scully-Allison, I. Lumsden, D. Hawkins, T. Burgess, V. Lama, J. Lüttgau, K. E. Isaacs *et al.*, "Thicket: Seeing the performance experiment forest for the individual run trees," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 281–293.

[26] T. Islam, A. Ayala, Q. Jensen, and K. Ibrahim, "Toward a programmable analysis and visualization framework for interactive performance analytics," in *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 2019, pp. 70–77.

[27] K. Gao, W.-k. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham, "Using subfiling to improve programming flexibility and performance of parallel shared-file i/o," in *2009 International Conference on Parallel Processing*. IEEE, 2009, pp. 470–477.

[28] S. Kumar *et al.*, "Scalable data management of the uintah simulation framework for next-generation engineering problems with radiation," in *Asian Conference on Supercomputing Frontiers*. Springer, Cham, 2018, pp. 219–240.

[29] S. G. Johnson and M. Frigo, "A modified split-radix fft with fewer arithmetic operations," *IEEE Transactions on Signal Processing*, pp. 111–119, 2006.

[30] S. Kumar and T. Gilray, "Load-balancing parallel relational algebra," in *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, Proceedings 35*. Springer, 2020, pp. 288–308.

[31] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[32] S. Parker, V. Morozov, S. Chunduri, K. Harms, C. Knight, and K. Kumaran, "Early evaluation of the cray xc40 xeon phi system 'theta'at argonne," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[33] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout, "Pidx: Efficient parallel i/o for multi-resolution multi-dimensional scientific datasets," in *2011 IEEE International Conference on Cluster Computing*.

[34] HPCToolkit on Theta. [Online]. Available: https://www.alcf.anl.gov/support-center/training-assets/hpctoolkit-theta

[35] D3 Home Page. [Online]. Available: https://d3js.org/