Two-phase IO Enabling Large-scale Introspection

Ke Fan
kfan23@uic.edu
University of Illinois at Chicago
Chicago, Illinois, USA

Sidharth Kumar
sidharth@uic.edu
University of Illinois at Chicago
Chicago, Illinois, USA

1 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 the complexity of problems that can be studied [8]. However, these growing possibilities also necessitate optimal use of computational resources, in order to achieve faster time-to-solution, lower (monetary) cost of computing, and lower carbon footprint. Performance profiling and visualization are critical to 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 simultaneously exposing the semantic context. Commonly used metrics measure runtime, data movement, cache misses, and other costs. Analysis and visualization of the resulting profiles is the second key step. Whereas summary statistics and pre-scripted, static plots can provide some indication of the overall performance behavior, a more-thorough understanding typically requires support from interactive visualization tools.

State-of-the-art performance analysis systems struggle to keep up with the increasing scale of today's applications. Their limitations can be broadly grouped into two categories: (1) I/O overhead and scalability challenges of parallel profilers; and (2) limited interactivity, portability, and scalability of visualization and analysis systems. There exists several profiling tools [2, 7] for capturing performance-related metrics from parallel applications. Although flexible and mostly easy to integrate into applications, existing solutions strive to reduce the runtime overhead of the profiler, largely ignoring the I/O overhead and the size of the profiles. As demonstrated in this paper, the I/O of profiling reports can be expensive for large-scale parallel applications (with hundreds or thousands of processes). State-of-the-art profiling systems like Caliper [2] and HPCToolkit [1] either support single-file I/O or file-per-process I/O modes which are known to have limitations at higher scales. A common way to analyze these profiles is via interactive visualization that can promote seamless hypothesis-driven exploration. A major challenge faced by existing systems stems from the increasing scale of the HPC applications that lead to proliferation in collected performance data. In order to be effective, these visualization systems must scale and encode data from all processes. For example, exploring per-process behavior [5] is crucial for diagnosing load imbalance issues and discovering interesting patterns (e.g., every n^{th} process performs poorly). However, depicting thousands of processes simultaneously is a difficult problem. As demonstrated in [7], few visualization methods can handle thousands of simultaneous tasks and some that can do so only for statistical plots.

Meeting the scalability criteria of both the profiling and the visualization systems, we develop *Viveka*, an end-to-end performance introspection framework comprising two essential components: (1) a lightweight and scalable parallel profiling system (Section 2); (2) a scalable, companion visualization system that can load the performance metric files and facilitate interactive analysis (Section 3). Our profiler collects performance metrics effectively from all running processes with minimal overhead. It facilitates annotating the HPC code, *e.g.,* loops and code snippets, and logging the associated application-specific metadata. To ensure that the profiler has low overhead (especially at high core counts), we deploy a two-phase data aggregation strategy with an optimal number of files that find a middle ground between single-file I/O and file-per-process I/O. We have also developed a compact file format that eliminates repeated metadata in the file, making the I/O easier and cheaper for both parallel logging and any downstream analysis, further improving the scalability of what can be interactively explored. Finally, we have developed a web-based, interactive dashboard with the ability to analyze and visualize profiles at high scales.

2 *VIVEKA*'S PARALLEL PROFILING

Our main innovation is in developing a scalable, low-overhead runtime, that can facilitate the profiling of parallel applications at high process counts *Viveka* presents (1) an easy-to-use annotation API, (2) a compact file format of profiles, and (3) a two-phase data aggregation strategy with sub-filing. The entire workflow of *Viveka* can be seen in Fig. 1.

Listing 1: Viveka minimal API listing

```
Viveka::Profiler(string filename, double io_frequency,
    int file_count);
Viveka::Event(string event_name, int is_common = 0, int
    mode=0, int ite=0, string tag="");
Viveka::~Event();
bool Viveka::flush();
```

Our minimal API can be seen in Listing 1. The `Profiler` is used to set application-specific contexts that only need to be set once for an application at the beginning. The `Event` class is instantiated for every code region to be annotated and referred to as an event. The calling sequence of events is generated automatically. `is_common` indicates if the event is called by every process (0 means true). Furthermore, a profiler must also be able to tackle loops. We design some optimal arguments for loop events: (1) `mode` (default 0, ≤ 2) and (2) `ite` (default 0). `mode > 0` indicates that an event is a loop. Its value 1 indicates measuring runtime for each iteration, whereas 2 means calculating the sum of the runtime for all loop iterations.

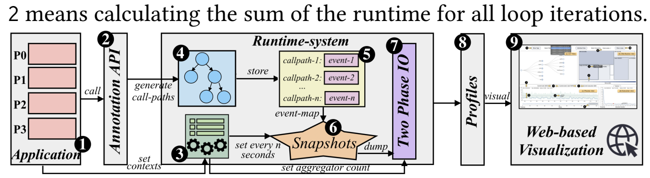

Figure 1: The workflow of *Viveka*.

All events are internally maintained in a hash-map called the *event-map* where the *callpath* serves as the *key* and the *value* corresponds to a *list* of *attribute*-class objects. The attribute class stores all relevant data for an event, including its time profile and other metadata. Executions repeat the entire program, whereas loops repeat specific functions, resulting in numerous events with the same *callpath*. Our usage of *event-map* takes advantage of them, eliminating data redundancies. Internally, we do not create a new key-value pair for every event with the same *callpath*; instead, we add it to the existing attribute list correspondingly. We translate this directly to our file format to minimize the size of output profiles.

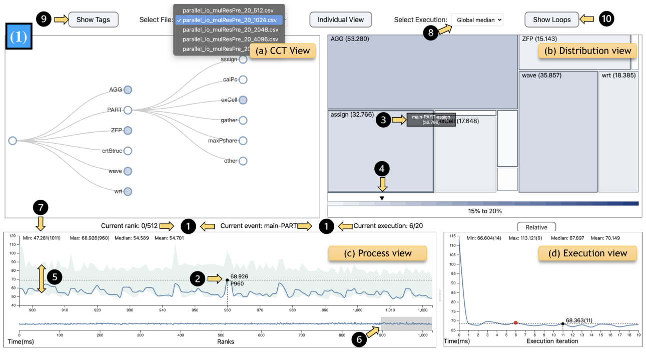

Figure 2: (1) Visualization with linked views.

Subfiling [4] is a known technique, wherein the number of files outputted is kept as a tunable parameter between 1 and P (P: process count). We observe the performance degradation with Caliper's file I/O (single-file IO). This problem can be alleviated via data aggregation [6], where only a selected set of processes (known as aggregators) perform all the necessary I/O operations. In order to attain a balance between all these performance-related parameters, we implement a customized two-phase data aggregation strategy along with sub-filling. We aggregate data from all processes to a tunable number of aggregators process, each of which writes data to an independent file concurrently.

3 *VIVEKA* WEB-BASED VISUALIZATION

Our visualization tool contains four linked views, including (1) the CCT view (for understanding and analyzing the structure of programs) (Fig. 2(a)), (2) the distribution view (complements the CCT view by helping in understanding how the runtime is distributed across the events) (Fig. 2(b)), (3) the process view (illustrates the per-process performance for a selected CCT node) (Fig. 2(c)), and (4) the execution view (demonstrates the total runtimes of the whole program across executions) (Fig. 2(d)), all working in conjunction to show profiled data in an intuitive way .

Example. The execution view helps users to perceive the stability of the application's performance. The best, average, and median execution results are offered by selection box without requiring users to calculate themselves. For example, in Fig. 2(d), the first execution result is noisy. The CCT view, in conjunction with the distribution view, can reveal potential performance bottlenecks in terms of events (e.g., expensive events) by investigating the views progressively. For example, in Fig. 2, we show all children of the event "main" at first, and the "PART" event is the most expensive in the distribution view. We then collapsed all the offspring of this event by clicking on it in the CCT view and found that the event "assign" is the most expensive and a leaf node. Therefore, the event "assign" in this case could be a potential performance bottleneck that needs to be optimized. In addition, we are able to assess its runtime percentage based on the colormap below. The process view can then assist users in identifying load imbalance issues for each given event (e.g., outliers) or finding out some interesting communication patterns (e.g., odd processes are slower than even processes.).

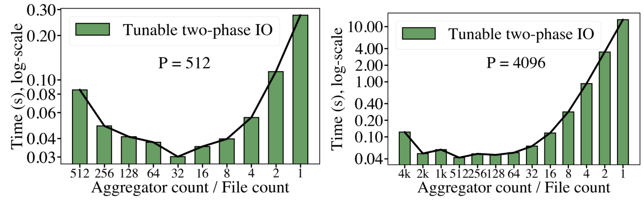

Figure 3: Results for two-phase I/O with varying aggregators.

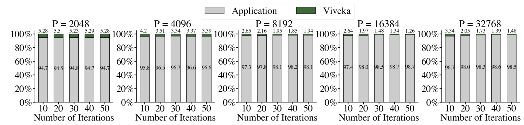

Figure 4: *Viveka* incurs minimal overhead (around 3%) at varying scales and for different iteration counts.

4 EVALUATION

We conducted a series of experiments on the *Theta* Supercomputer at Argonne National Laboratory to evaluate the efficacy and performance of both *Viveka*'s profiling and visualization capabilities. We use Parallel I/O [3] as our target application. The parallel I/O framework is a *comprehensive* application that consists of a sequence of iterable computation, communication, and I/O phases. To evaluate the performance of the profiling capabilities of *Viveka*, we conduct (a) a benchmark of the two-phase I/O to demonstrate the importance of the tunable data aggregation scheme and (b) overhead analysis, to compute the overhead of profiling in a production environment.

We evaluate the efficacy of *Viveka*'s tunable two-phase 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). The results for these two sets of runs are shown in Figure 3. Both results show a similar U - shaped trend, where the optimal performance is reached roughly around $A = P/16$ aggregators.

We then evaluate the efficacy of *Viveka* in a production setting where applications typically run for several time steps. In this experiment, we ran the application for a varying number of timesteps and recorded the overhead that *Viveka* 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 Fig. 4. We observe that the overhead added by *Viveka* remains consistent to around $2 - 5\%$ for all scales and iteration settings.

5 CONCLUSION

In this paper, we presented *Viveka*, an end-to-end system for profiling and visualizing large-scale applications. Our simple annotation API enables easy integration with parallel applications, resulting in a compact file format. *Viveka* includes an interactive web-based visualization system that allows the exploration of application profiles using a set of linked views.

REFERENCES

[1] Laksono Adhianto et al. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Conc. and Comp.: Prac. and Exp.* (2010).

[2] David Boehme et al. 2016. Caliper: performance introspection for HPC software stacks. In *Proc. of the Int. Conf. for High Perf. Comp., Net., Stor. and Anal.* IEEE.

[3] Ke Fan, Duong Hoang, Steve Petruzza, Thomas Gilray, Valerio Pascucci, and Sidharth Kumar. 2021. Load-balancing Parallel I/O of Compressed Hierarchical Layouts. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE.

[4] Kui Gao, Wei-keng Liao, Arifa Nisar, Alok Choudhary, Robert Ross, and Robert Latham. 2009. Using subfiling to improve programming flexibility and performance of parallel shared-file I/O. In *2009 International Conference on Parallel Processing*. IEEE, 470–477.

[5] Suraj Kesavan, Harsh Bhatia, Abhinav Bhatele, Stephanie Brink, Olga Pearce, Todd Gamblin, Peer-Timo Bremer, and Kwan-Liu Ma. 2021. Scalable Comparative Visualization of Ensembles of Call Graphs. *Trans. on Vis. and Comp. Graph.* (2021).

[6] Sidharth Kumar et al. 2018. Scalable data management of the Uintah simulation framework for next-generation engineering problems with radiation. In *Asian Conference on Supercomputing Frontiers*. Springer, Cham, 219–240.

[7] Jonathan R Madsen et al. 2020. TiMemory: modular performance analysis for HPC. In *Int. Conf. on High Perf. Comp.* Springer, 434–452.

[8] Jie Shen et al. 2015. Workload partitioning for accelerating applications on heterogeneous platforms. *IEEE Trans. on Para. and Dist. Sys.* 27 (2015).