Pipit: Simplifying Parallel Trace Analysis

Alexander Movsesyan, Rakrish Dhakal, Aditya Ranjan, Jordan Marry, Onur Cankur, Abhinav Bhatele

Department of Computer Science, University of Maryland
College Park, Maryland, USA
{amovsesy,rakrish,aranjan2,jmarry,ocankur}@umd.edu,bhatele@cs.umd.edu

ABSTRACT

Performance analysis is an important part of the oft-repeated, iterative process of performance tuning during the development of parallel programs. Per-process per-thread traces (detailed logs of events with timestamps) enable in-depth analysis of parallel program execution to identify various kinds of performance issues. Often times, trace collection tools provide a graphical tool to analyze the trace output. However, these GUI-based tools only support specific file formats, are difficult to scale when the data is large, limit data exploration to the implemented graphical views, and do not support automated comparisons of two or more datasets. In this poster, we present a programmatic approach to analyzing parallel execution traces by leveraging pandas, a powerful Python-based data analysis library. We have developed a Python library, Pipit, on top of pandas that can read traces in different file formats (OTF2, HPCToolkit, Projections, Nsight, etc.) and provide a uniform data structure in the form of a pandas DataFrame. Pipit provides operations to aggregate, filter, and transform the events in a trace to present the data in different ways. We also provide several functions to quickly identify performance issues in parallel executions.

1 INTRODUCTION

We fill the above mentioned gaps in performance analysis of parallel execution traces by developing Pipit, a programmatic API to analyze traces. Pipit provides trace data access to the end user so that they can explore the data programmatically instead of having to use a graphical interface. Since traces represent a time series of events (with categorical and numerical data per event), we leverage pandas [7], a powerful Python-based data analysis library for analyzing tabular data. Pipit can read traces in different file formats (OTF2 [5], HPCToolkit [1], Projections [6], Nsight [8], etc.) and provides a uniform data structure in the form of a pandas DataFrame. Pipit exposes a programmatic API to the end user with operations to aggregate, filter, and transform the events in a trace to explore, manipulate, and visualize the data in different ways.

There are several common data exploration/manipulation tasks that end users perform when analyzing parallel traces. Some examples are analyzing a heat map or matrix of communication between MPI processes, detecting load imbalance across threads or processes, detecting a critical path in the execution, identifying the most time consuming functions etc. We have designed and implemented many of these operations in the Pipit API to reduce user effort in performance analysis tasks. We also present several case studies that demonstrate the capabilities of Pipit.

SC '23, November, 2023, Denver, CO
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 THE PIPIT LIBRARY

Our goals in developing the Pipit library were the following: (1) Support several file formats in which execution traces are collected to provide users with a unified interface that works with outputs of many different tracing tools. (2) Provide a programmatic API, which allows users to write simple code for trace analysis and provides several benefits such as flexibility of exploration, scalability, reproducibility, and automation/saving of workflows. And (3) Automate certain common performance analysis tasks for analyzing single and multiple executions.

3 CASE STUDIES

Below, we present some case studies that demonstrate the utility of Pipit in simplifying parallel trace analysis.

3.1 Load imbalance analysis

Using the load_imbalance function, we can expose asymmetry in aggregated runtimes of functions across processes. The code in Figure 1 demonstrates such an example, where a Projections trace of Loimos, an epidemic simulation framework, is read. After this, with just a few lines of code, the output of the load_imbalance function is filtered by the five most time consuming functions to identify the imbalance in them.

	time.exc.imbalance	Top processes	time.exc.mean
ReceiveVisitMessages(const VisitMessage &impl_noname_1)	2.235940	[24, 21, 23, 22, 29]	1.822500e+03
ComputeInteractions()	1.985484	[21, 37, 29, 22, 23]	1.254858e+04
SendVisitMessages()	1.758879	[22, 23, 28, 35, 31]	9.691400e+03
Idle	1.291811	[110, 127, 124, 103, 105]	4.900719e+04
Computation	1.000056	[46, 84, 86, 70, 7]	1.316492e+06

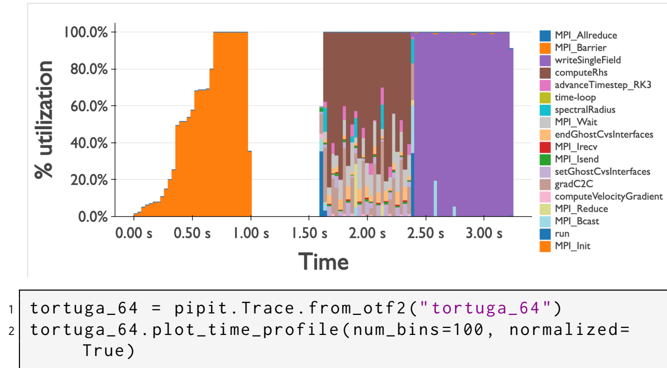
```
loimos_128 = pipit.Trace.from_projections('loimos_128')

loimos_128.calc_exc_metrics()
imbalance_df = loimos_128.load_imbalance(num_processes=5)
imbalance_df = imbalance_df.iloc[0:5].sort_values(by='
    time.exc.imbalance', ascending=False)
```

Figure 1: Analyzing Load Imbalance for the 5 Most Time Consuming Functions

3.2 Utilization over time

The time_profile function provides an overview of the activity or utilization over time, and allows the user to identify repeating patterns or functions that might be a significant portion of the total time. In Figure 2, we show a time profile of a CFD code, Tortuga, running on 64 processes. The stacked bar chart allows the user to see what functions are taking up the most amount of time in a specific bin. Focusing on the middle of the time profile, we observe

that the `computeRhs` function (in brown) makes up a significant portion of the total time. We can see that `advanceTimestep_Rk3` and `spectralRadius` have a pattern and are called periodically in the middle region. The code at the bottom shows that using two lines of Python code, a user can glean significant information from a time profile.




```
tortuga_64 = pipit.Trace.from_otf2("tortuga_64")
tortuga_64.plot_time_profile(num_bins=100, normalized=
     True)
```

Figure 2: Time profile of a Tortuga trace with 64 processes.

3.3 Pattern detection

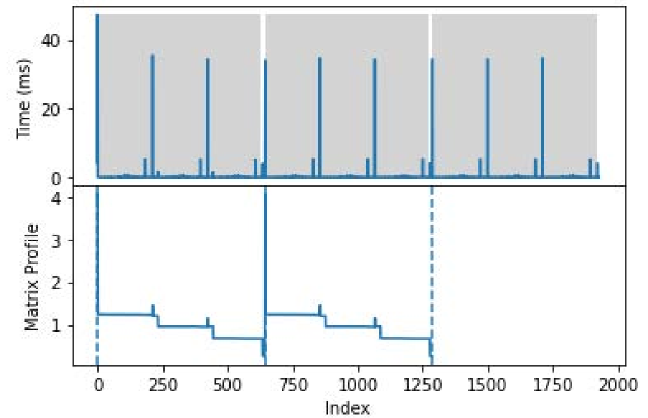
To identify patterns in a trace, we use a Score-P user annotated Tortuga execution on 16 processes and set the number of iterations to three when running the program. We pass the number of iterations and a window size (calculated by inspecting the start of each loop iteration) to the pattern_detection function. The top plot in Figure 3 presents a time series generated using the exclusive time values of each enter event in the trace. The bottom plot shows the corresponding matrix profile. The lowest points in the matrix profile indicate similar subsequences (vertical dashed lines). For details about the matrix profiles, we refer the reader to the paper by Yeh et al. [9]. As we can see, Pipit can detect patterns using this approach and identify the start of iterations.

3.4 Communication

Figure 4 shows the communication matrix of a Laghos execution on 32 processes, using both a linear colormap (on the left), and a logarithmic colormap (on the right). The code snippet required to generate the views is shown in the listing at the bottom. The heatmap shows the total data exchanged between any two processes. We observe that the matrix is symmetric, and the communication happens along diagonals. This typically suggests a near-neighbor communication pattern in an n-dimensional virtual topology. Switching to logarithmic scale for the colormap makes additional patterns visible in the data.

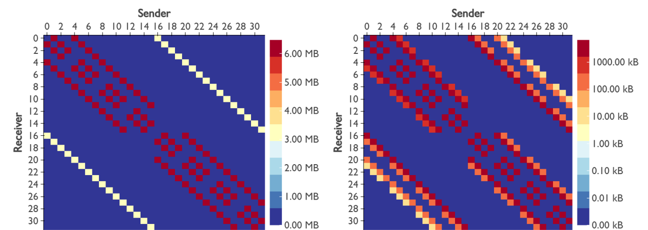
4 CONCLUSION

In this paper, we present a new Python-based performance analysis tool called Pipit for analyzing parallel execution traces. Through Pipit's design and implementation, we sought to solve the following challenges: (1) Support several file formats in which execution traces are collected to provide users with a unified interface that works with outputs of many different tracing tools. (2) Provide a programmatic API, which allows users to write simple code for trace analysis and provides several benefits such as flexibility of exploration, scalability, reproducibility, and automation/saving of workflows. And (3) Automate certain common performance analysis tasks for analyzing single and multiple executions. To the best of our knowledge, Pipit is unique in its capabilities in terms of supporting several file formats and providing a programmatic API to analyze traces. Hatchet is the tool that comes closest to it but only supports aggregated profiles [2–4]. We believe that Pipit can revolutionize how HPC developers and performance engineers analyze the performance of their codes, and improve the efficiency of both parallel programs and HPC programmers.




```
tortuga_16 = pipit.Trace.from_otf2('./tortuga_16')
matches = tortuga_16.detect_pattern(window_size,
     iterations, metric='time.exc')
tortuga_16.plot_timeline()
```

Figure 3: Detecting patterns in a trace. The y-axis in the top plot shows the exclusive time values for each enter event of a process. The gray boxes represent the patterns detected. The vertical dashed lines on the bottom plot (minimum values on the matrix profile) point to start indices of the similar subsequences.




```
laghos_32 = pipit.Trace.from_otf2('./laghos_32')

laghos_32.plot_comm_matrix(mapping='linear')
laghos_32.plot_comm_matrix(mapping='log')
```

Figure 4: Communication matrix of a Laghos execution on 32 processes, with a linear colormap (left) and logarithmic colormap (right).

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Alexandre Bergel, Abhinav Bhatele, David Boehme, Patrick Gralka, Kevin Griffin, Marc-Andre Hermanns, Dusan Okanovic, Olga Pearce, and Tom Vierjahn. 2019. Visual Analytics Challenges in Analyzing Calling Context Trees. In *Programming and Performance Visualization Tools (Lecture Notes in Computer Science, Vol. 11027)*. https://link.springer.com/chapter/10.1007/978-3-030-17872-7_14

[3] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. 2019. Hatchet: Pruning the Overgrowth in Parallel Profiles. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’19)*. http://doi.acm.org/10.1145/3295500.3356219 LLNL-CONF-772402.

[4] Stephanie Brink, Ian Lumsden, Connor Scully-Allison, Katy Williams, Olga Pearce, Todd Gamblin, Michela Taufer, Katherine E Isaacs, and Abhinav Bhatele. 2020. 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, 49–58.

[5] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. 2012. Open trace format 2: The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 481–490.

[6] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. 2006. Scaling Applications to Massively Parallel Machines Using Projections Performance Analysis Tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, Vol. 22. 347–358.

[7] Wes McKinney. 2017. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.* O’Reilly Media.

[8] NVIDIA. [n. d.]. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[9] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. 2016. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 1317–1322. https://doi.org/10.1109/ICDM.2016.0179