Pipit: Simplifying Parallel Trace Analysis

Alexander Movsesyan, Rakrish Dhakal, Aditya Ranjan, Jordan Marry, Onur Cankur, Abhinav Bhatele
Department of Computer Science, University of Maryland

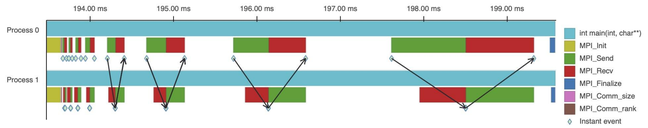
Abstract

Performance analysis is an imperative part of performance tuning during the development of parallel programs. Parallel execution traces enable in-depth analysis of the program's performance. Current trace-analysis tools/workflows have some gaps

- Most trace analysis tools support different formats and analyses.
- GUI-based tools limit data exploration to their graphical views

We have developed Pipit, a Python-based tool, to fill in the gaps in trace-analysis:

- Can read traces from different file formats (OTF2, HPCToolkit, Projections, etc.)
- Provides a uniform data structure in the form of a pandas DataFrame
- Provides a programmatic API to analyze traces
- Provides interactive visual functions to display the traces



Background and Structure

Traces: Time series data representing all the events that occur during the program's execution

- When functions are entered and exited
- When messages are shared between processes
- Different metrics (such as hardware performance counters)

```
Timestamp (s), Event Type, Name, Process
0, Enter, main(), 0
1, Enter, foo(), 0
3, Enter, MPI_Send, 0
5, Leave, MPI_Send, 0
8, Enter, baz(), 0
18, Leave, baz(), 0
25, Leave, foo(), 0
100, Leave, main(), 0
```

How does Pipit store trace data?

- A pandas DataFrame: two-dimensional labeled table-like data structure
- Every trace event is stored as a row in the DataFrame
- DataFrame is sorted by event timestamps

	Timestamp (ns)	Event Type	Name	Process
0	0	Enter	main()	0
1	1000000000	Enter	foo()	0
2	3000000000	Enter	MPI_Send	0
3	5000000000	Leave	MPI_Send	0
4	8000000000	Enter	baz()	0
5	18000000000	Leave	baz()	0
6	25000000000	Leave	foo()	0
7	100000000000	Leave	main()	0

The Calling Context Tree

- Represents caller-callee relationship between functions
- Stored as a graph in Pipit, and each event in the DataFrame corresponds to a node in the calling context tree

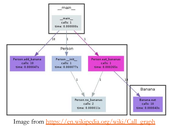
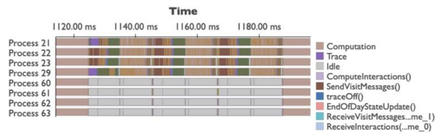


Image from https://en.wikipedia.org/wiki/Call_graph

Case Studies Using Pipit

1. Finding Load Imbalance

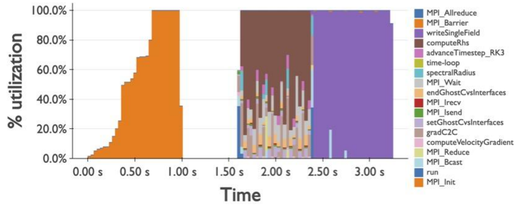
We analyze traces of a Loimos (a Charm++-based epidemiology simulator) execution on 64 threads. We want to find which processes are idling the most while the others are overloaded. Pipit's idle_time function can help us with this task. We then plot a timeline filtered to the most and least idling processes.




```
loimos_64 = pipit.Trace.from_projections('./loimos_64')
idle_times = loimos_64.idle_time()
idle_times = idle_times.sort_values(by=['Idle Time'], ascending=False)
bad_procs = idle_times["Process"].head(4)
good_procs = idle_times["Process"].tail(4)
loimos_64.filter("Process", "in", bad_procs + good_procs).plot_timeline()
```

2. Analyzing Overall Performance

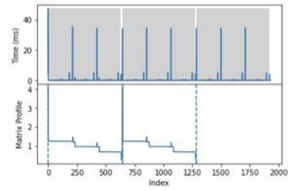
Here we analyze a trace of a Tortuga execution on 64 cores. We use pipit to understand an execution's machine utilization over time. Pipit's time_profile function provides an overview of the executions activity/utilization over time.




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

3. Pattern Detection

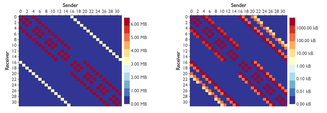
We can use pipit's detect_pattern function to find recurring sets of events in the trace. Below, we analyze a tortuga execution on 16 cores.



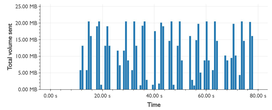

```
tortuga_16 = pipit.Trace.from_otf2('./tortuga_16')
matches = tortuga_16.detect_pattern(window_size , iterations , metric='time.exc')
tortuga_16.plot_timeline()
```

4. Analyzing Communication

Here, we analyze the 32-core executions of laghos. We can use pipit's *plot_comm_matrix* and *plot_comm_over_time* functions to examine the communication between ranks and over time respectively.




```
laghos_32 = pipit.Trace.from_otf2('./laghos_32')
laghos_32.plot_comm_matrix(mapping='linear')
laghos_32.plot_comm_matrix(mapping='log')
```

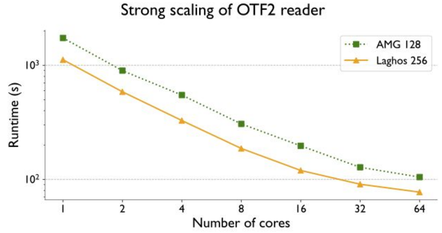



```
laghos_32.plot_comm_over_time()
```

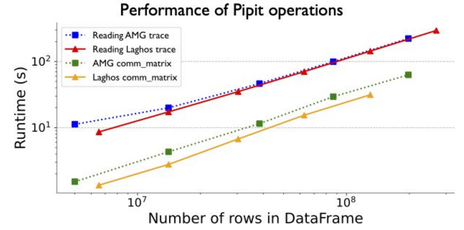
Performance of Pipit

All the experiments in this section were performed on a single node of an HPC cluster with a dual 64-core AMD EPYC 7763 processor.

Time spent by the Pipit OTF2 reader in reading traces of two different applications, AMG (128 processes) and Laghos (256 processes).



Time spent in the OTF2 reader and the comm_matrix function with AMG and Laghos traces of different sizes.



Getting Started

Install pipit with pip

```
pip install pipit
```

Scan the QR code for pipit on GitHub

Acknowledgements

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

References

[1] Lukasz 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] 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.
[3] Leemhuis V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. 2008. 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.
[4] Wes McKinney. 2017. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. O'Reilly Media.