Enabling Graph-Based Profiling Analysis of HPC Applications using Hatchet

Ian Lumsden
University of Tennessee
Knoxville, Tennessee

ACM Reference Format:
Ian Lumsden. 2020. Enabling Graph-Based Profiling Analysis of HPC Applications using Hatchet. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC20).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ABSTRACT

We augment the Hatchet performance data analysis tool to leverage relational caller-callee data in the analysis of HPC applications. Specifically, we design a query language that enables data reduction using graph path pattern matching. We demonstrate the capabilities of our augmented Hatchet by examining the performance of MPI calls in three High Performance Computing (HPC) benchmarks (i.e., AMG2013, Kripke, and Lammps) when using MVAPICH and Spectrum-MPI. In doing so, we identify hidden performance losses in specific MPI functions.

1 INTRODUCTION

Profilers measure code performance on HPC systems, allowing users to identify and mitigate performance and scalability bottlenecks. Numerous HPC profilers exist (e.g., TAU, Calliper, and HPCToolkit). Unfortunately, most profilers use their own unique format for storing profiling data. As a result, users are required to use the analysis tools provided by the profiling software. These tools are typically GUI-based, and they do not allow the user to analyze performance data *programmatically*. This ultimately limits the kinds of analysis users can perform on their data.

Hatchet [2] is a Python library that overcomes these analysis constraints by allowing users to read the hierarchical call graph data generated by **different HPC profilers** into a new data model that builds upon the combination of the *pandas* Python library and graph-based hierarchical data representations. Although Hatchet resolves the issues associated with HPC profilers and their unique data formats, it currently does not provide a way to utilize the relational caller-callee data collected by profilers in analysis, thus limiting the types of analysis a user can perform.

In this work, we augment Hatchet to enable analysis using the relational data collected by HPC profilers. To achieve this, we design and implement a new graph-based filtering query language in Hatchet. We also present a case study in which we use this query language to identify potential root causes for differing performance of MPI calls in several HPC benchmarks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC20, November 2020, Atlanta, GA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

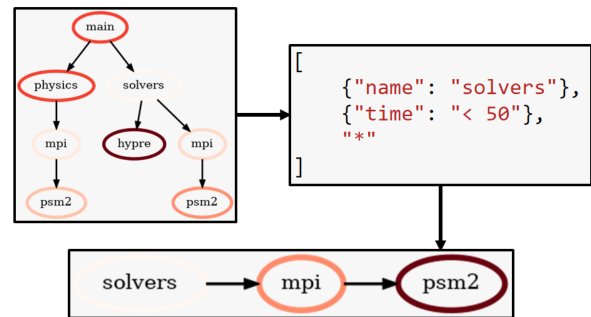

Figure 1: Example of using the new Hatchet query language to filter a graph.

2 METHODOLOGY

To leverage relational data in analysis using Hatchet, we design a graph query language that filters performance data through graph path pattern matching. Our query language is based on Cypher [3] and GQL [4]. In our query language, users provide a *query path* in the form of a list of *abstract graph nodes*. A node consists of two elements: (1) a wildcard specifying the number of real graph nodes to match to the *abstract graph node* and (2) a filter determining whether a real graph node matches the *abstract graph node*. We filter data graphs in three steps. First, we match all real nodes in the graph to the *abstract nodes* in the user-provided *query path*. Next, we collect an exhaustive list of all paths in the graph that match the entire *query path*. Finally, we use the exhaustive list to create a new graph containing only the real nodes found in the list of matched paths. An example of this is shown in Figure 1.

Our query language consists of two API levels. The "high-level" API represents the *query path* as a Python list in which each element is an *abstract graph node*. Filters in the high-level API are represented as Python dictionaries keyed on the attribute names of real nodes in the graph. The "low-level" API represents the *query path* as a set of chained function calls in which each function call represents a single *abstract graph node*. Filters in the low-level API are represented by Python callables that accept a *pandas* Series representing a row and return a boolean. In both API levels, we represent wildcards as either a number or a regex-style wildcard string.

3 EVALUATION

We evaluate the effectiveness of Hatchet once augmented with our query language in identifying sources of performance losses associated with MPI calls in three HPC benchmarks (i.e., AMG2013,

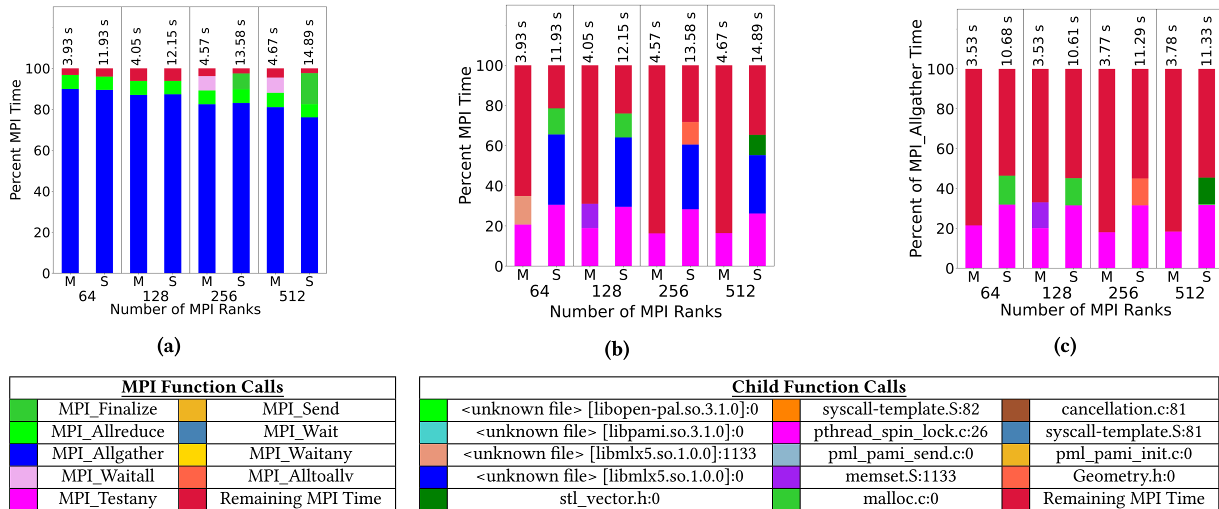

Figure 2: The percent of (a) total MPI time in AMG2013 spent in MPI functions, (b) total MPI time in AMG2013 spent in the children calls of MPI functions, and (c) total MPI_Allgather time in AMG2013 spent in children calls. We denote the MVAPICH and Spectrum-MPI libraries by *M* and *S*, respectively.

Kripke, and Lammps). We use two different MPI libraries (i.e., MVAPICH and Spectrum-MPI) with 64, 128, 256, and 512 ranks on LLNL's Lassen supercomputer. We profile all the benchmark runs using HPCToolkit [1]. Because of our query language, we can extract the subgraphs rooted at standard MPI function calls from the generated profiles. Using the subgraphs we obtain, we examine the percentage of the total MPI time spent in each MPI function call. We also examine the percentage of total MPI time spent in each child call of the MPI functions. Using this data, we determine the MPI calls and children calls that are most important to the performance of the benchmark running with a particular MPI library.

In this work, we only show results related to the MPI_Allgather function in AMG2013 due to space constraints. Our reasons for this are twofold. First, as shown in Figure 2a, MPI_Allgather clearly comprises the majority of the MPI time spent in the AMG2013 benchmark. Second, the AMG2013 benchmark has the largest performance difference between MVAPICH and Spectrum-MPI. This suggests that, if we can determine a likely primary cause for the performance difference in MPI_Allgather across the two MPI libraries, we can also determine the likely primary cause for the overall performance difference across the libraries.

To determine a likely cause for the performance difference in MPI_Allgather, we first use the query language to obtain the subgraphs of the AMG2013 data rooted at MPI_Allgather calls. We further reduce the data to consider only the children calls of this MPI function that we previously identified in Figure 2b as most important to the performance of the program. The results of this reduction are shown in Figure 2c.

4 LESSONS LEARNED

In our tests with MVAPICH and Spectrum-MPI, we determine that the *pthread_spin_lock* function is consistently a major contributor to MPI runtime (i.e., 10% or more of the MPI time, usually 20% or more). Additionally, when considering *MPI_Allgather*, we conclude that the worse performance of Spectrum-MPI may be due to differences in its use of *pthread_spin_lock* compared to MVAPICH. Overall, our augmented Hatchet supports these new analysis capabilities: extracting all call paths specific to a given library; determining the performance contributions of function calls used internally in a library; correlating children function calls to specific important library API calls in an application; using this correlation to determine children function calls that contribute the most to the performance of the targeted library API call; and comparing the correlation of children and API calls across libraries to determine possible causes for performance differences in these libraries.

ACKNOWLEDGMENT

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-ABS-813303).

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] A. Bhatele, S. Brink, and T. Gamblin. 2019. Hatchet: Pruning the Overgrowth in Parallel Profiles. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*.

[3] N. Francis, A. Green, P. Guagliardo, L Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proc. of the 2018 International Conference on Management of Data (SIGMOD18)*.

[4] A. Green, P. Furniss, P. Lindaaker, P. Selmer, H. Voigt, and S. Plantikow. 2019. *GQL Scope and Features.* Technical Report. ISO.