Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications

Ian Lumsden*, Jakob Luettgau*, Vanessa Lama*, Connor Scully-Allison§,
Stephanie Brink‡, Katherine E. Isaacs§, Olga Pearce‡, Michela Taufer*

*Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA
‡Lawrence Livermore National Laboratory, Livermore, CA, USA
§SCI Institute and the School of Computing, University of Utah, UT, USA

2022 IEEE 18th International Conference on e-Science (e-Science) | 978-1-6654-6124-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ESCIENCE55777.2022.00039

***Abstract*—As computational science applications benefit from larger-scale, more heterogeneous high performance computing (HPC) systems, the process of studying their performance becomes increasingly complex. The performance data analysis library Hatchet provides some insights into this complexity, but is currently limited in its analysis capabilities. Missing capabilities include the handling of relational caller-callee data captured by HPC profilers. To address this shortcoming, we augment Hatchet with a Call Path Query Language that leverages relational data in the performance analysis of scientific applications. Specifically, our Query Language enables data reduction using call path pattern matching. We demonstrate the effectiveness of our Query Language in identifying performance bottlenecks and enhancing Hatchet's analysis capabilities through three case studies. In the first case study, we compare the performance of sequential and multi-threaded versions of the graph alignment application Fido. In doing so, we identify the existence of large memory inefficiencies in both versions. In the second case study, we examine the performance of MPI calls in the linear algebra mini-application AMG2013 when using MVAPICH and Spectrum-MPI. In doing so, we identify hidden performance losses in specific MPI functions. In the third case study, we illustrate the use of our Query Language in Hatchet's interactive visualization. In doing so, we show that our Query Language enables a simple and intuitive way to massively reduce profiling data.**

I. Problem and Motivation

Computational science applications studying a variety of phenomena in nature have gradually become more complex, both in terms of problem size and algorithms. To deal with this complexity, scientists run their applications on large-scale, high performance computing (HPC) systems. The performance of application executions on supercomputers (e.g., the time to completion or runtime, the amount of resources and power used) has gained relevance as scientists pursue short turnaround scientific discovery. To assist with the need for high performance, measurement tools, also called profilers, are used to provide insights on sources of performance bottlenecks in applications. Numerous profilers specialized for HPC applications exist [1]–[5]. As HPC systems increase in size and heterogeneity of their resources, the study of performance bottlenecks through application profiling has also become increasingly complex, as illustrated in Figure 1. The figure shows an example of the complexity of performance visualization using a well-known tool named HPCToolkit for the profiling of the Fido application [6], [7] for aligning graph structures (i.e., DNA, protein, and codon) on Lawrence Livermore National Laboratory's (LLNL) Lassen supercomputer. This complexity leads to difficulties in extracting useful information on how to improve application performance and accelerate scientific discovery.

One key challenge when selecting a suitable profiler is that most profilers use specialized solutions to assist in performance analysis and tuning of scientific applications. To manage the increased complexity of scientific applications and HPC systems, HPC profilers utilize solutions which introduce difficulties in analyzing performance data. For example, most profilers use their own unique file format for storing profiling data. As a result, users are required to deploy 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. Another example is how HPC profilers attribute execution time to source code. Simple profilers correlate the execution time to functions or statements in the source code. More advanced profilers may distinguish between invocations of a function in different call paths (i.e., the series of function calls that led to the current function) and correlate execution time to each unique invocation. As a result, profiling data generated by different profilers often represents the code in different ways, making performance data analysis tedious.

Hatchet [8], [9] is an open-source Python library for the performance analysis of profiling data that overcomes the above-mentioned limitations by allowing users to read the hierarchical data generated by different HPC profilers (e.g., HPCToolkit [5], Caliper [2], and TAU [4]) into a new data model that builds upon the combination of the pandas Python library [10], [11] and graph-based hierarchical data representations. However, Hatchet has a major shortcoming when considering extremely large and complex profiling data such as the one illustrated in Figure 1. It does not provide easy-to-use, programmatic ways to leverage the hierarchical nature of profiling data. When dealing with such data, users often need only a subset of the available data to identify and examine performance phenomena such as bottlenecks. As a result, users need a way to reduce their profiling data to a meaningful subset. Hatchet provides predicate-based filters to perform data

978-1-6654-6124-5/22/$31.00 ©2022 IEEE
DOI 10.1109/eScience55777.2022.00039

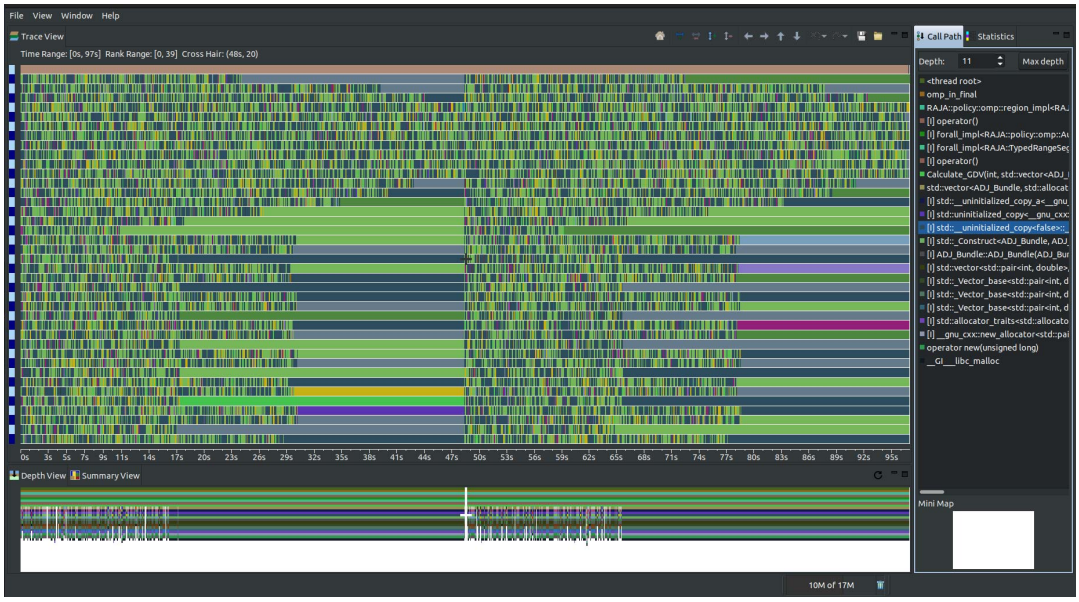


Fig. 1: Example of the complexity of HPCToolkit's performance visualization for the profiling of the Fido application [6], [7] on Lawrence Livermore National Laboratory's (LLNL) Lassen supercomputer.

reduction. These filters can capture individual nodes satisfying a single predicate, but they cannot perform more complex operations such as capturing sequences of nodes that satisfy a set of predicates. Only more sophisticated filtering allows users to capture both a sequence of nodes and the information about their call paths.

In this work, we augment Hatchet with a Call Path Query Language that enhances analysis capabilities by leveraging the hierarchical data collected by HPC profilers. To achieve this, we first design a Query Language, using the Cypher [12] query language for graph databases as inspiration. Our Query Language extracts call paths from the application's profiling data using queries. A query is a description of the properties of one or more paths. To facilitate the use of our Call Path Query Language from object-oriented Python and string-based JavaScript, we define two dialects for the definition of more expressive queries, the Object-based Dialect and the String-based Dialect. We integrate the Query Language and its dialects into Hatchet to enhance Hatchet's analysis capabilities.

We present three case studies to illustrate the effectiveness of our Query Language in identifying performance bottlenecks and enhancing Hatchet's analysis capabilities. In our first case study, we examine profiling data of the Fido application. In this case, our query language allows us to identify memory inefficiencies across both Fido's sequential and multi-threaded versions. In our second case study, we examine the AMG2013 mini-application, which is derived from a linear system solver that is commonly used in computational science applications. Our query language allows us to identify hidden performance losses in specific MPI functions of AMG2013 that otherwise were hidden to the developers. Finally, in our third case study, we illustrate how our Query Language enables Hatchet's Jupyter notebook-based interactive visualization. In this case, our Query Language allows the visualization to provide users with a simple and intuitive way to massively reduce their profiling data interactively.

The contributions of this work are as follows:

- We design and implement a new Call Path Query Language in Hatchet.
- We define Object-based and String-based Dialects for our Query Language to simplify its use under certain circumstances (e.g., when building queries in JavaScript).
- We classify the capabilities of our Query Language and its dialects to show their differences and how to choose among them based on the user's requirements.
- We demonstrate the benefits of our Query Language through three case studies.

The paper is organized as follows: Section II presents Hatchet and HPCToolkit, the two tools this work builds on. Section III describes the design of our Call Path Query Language. Section IV defines the dialects of the Query Language and their purpose. Section V provides the capability classifications of the Query Language and its dialects. Section VI demonstrates the capabilities of the Query Language and its dialects through the three case studies. Section VII discusses related work and Section VIII concludes the paper.

II. HPCToolkit and the Hatchet Library

We briefly describe key tools on which we build the work presented in this paper. We use HPCToolkit to collect profiling data and we use the Hatchet library to analyze the data. Note that prior to our work, Hatchet did not integrate capabilities to effectively deal with the hierarchical nature of profiling data.

A. HPCToolkit

HPCToolkit [5] is the profiling tool we use throughout this work to profile applications. HPCToolkit is a suite of profiling, analysis, and visualization tools designed primarily for HPC applications. It uses sampling at both the thread and process level to measure performance metrics of applications with hybrid CPU parallelism. We selected this tool because it does not require any code annotations and thus can be easily used across runs of different applications. After collecting performance metrics, HPCToolkit associates them with the full call path in which they occur and stores the final profile in a calling context tree.

B. The Hatchet Library

Users can feed profiling data from various tools, including HPCToolkit, into the Hatchet library. Hatchet's primary data structure is the `GraphFrame`, which is comprised of two parts: a custom `Graph` and a pandas `DataFrame`. The `Graph` data structure stores the caller-callee relationships that define call paths. The `DataFrame` stores the performance metrics associated with each node in the `Graph`. To combine both the `Graph` and pandas `DataFrame` into a single canonical data model, Hatchet provides a structured index that allows nodes in the `Graph` to be used as an index in the pandas `DataFrame`. Besides providing a single canonical data model for profiling data, Hatchet also provides readers to ingest data gathered from several popular profiling tools, such as HPCToolkit [5], Caliper [2], GNU gprof [1], and many others. Once the data has been read into a Hatchet `GraphFrame`, users can deploy provided operations such as filtering and comparisons across `GraphFrames`, or they can extract the pandas `DataFrame` and utilize other pandas-compatible Python data analysis tools.

III. Call Path Query Language

The foundational contribution of our work is the design and implementation of a Query Language. We design the Query Language to enable a Hatchet user to extract a set of paths from a call graph (e.g., function calls). We define a *query* as a sequence of query nodes. A *query node* is comprised of a quantifier and a predicate. A *quantifier* defines how many real nodes in a call path to match to a query node. A *predicate* defines what conditions must be satisfied for a real node to match a query node.

By applying a query to profiling data, our Query Language finds all paths in a call graph that match properties described by the query. We frame this problem as a version of the subgraph isomorphism problem. To this end, we use a modified version of the Ullmann's algorithm [13], one of the most important algorithms for solving the subgraph isomorphism problem. If used in its original version, the Ullmann's algorithm has two shortcomings. First, the Ullmann's algorithm is not designed to handle graphs in which nodes have attributes or metrics. Second, Ullmann's algorithm cannot process quantifiers. Thus, we modify the algorithm in two ways. First, we replace degree-based node comparison with the use of predicates to account for the fact that nodes in profiling data have metrics. Second, we add support for quantifiers. Algorithms 1 and 2 show our modified Ullmann's algorithm. `MatchPaths` finds all paths that match the query and start with a given node. `ApplyQuery` uses `MatchPaths` to capture a set of all paths in the profiling data that match the query. After applying our modified Ullmann's algorithm, we output a new Hatchet GraphFrame containing only the nodes in the captured paths and any edges that connect these nodes.

Algorithm 1 Apply Query to a given Call Graph

Input:
$graphframe$: the call graph data to query
$query$: the query being applied
Output: all paths in $graphframe$ that match $query$

function ApplyQuery($graphframe, query$)
  $matches \leftarrow [\,]$
  for each node $node$ in $graphframe$ **do**
    if $node$ satisfies the predicate of $query[0]$ **then**
      $new\_paths \leftarrow MatchPaths(node, query)$
      if $matches \neq \varnothing$ **then**
        add paths in new_paths to $matches$
      end if
    end if
  end for
  return unique paths in $matches$
end function

A. Constructing Call Path Queries using the Query Language

The `QueryMatcher` class in Hatchet defines how to construct a call path query. Using this class, we build queries using the `match` and `rel` methods. The `match` method sets the first node of the query. The `rel` method is called iteratively, each time adding a new node to the end of the query. Both methods take a quantifier and a predicate as input.

A quantifier can have one of four possible values:

- `"."`: match one node
- `"*"`: match zero or more nodes
- `"+"`: match one or more nodes
- An integer: match exactly that number of nodes

The `"."` quantifier matches one node, and the `"*"` quantifier matches zero or more nodes. The `"+"` and integer quantifiers are both implemented in terms of the `"."` and `"*"` quantifiers. Specifically, the `"+"` quantifier is implemented as two nodes: one with a `"."` quantifier followed by one with a `"*"` quantifier. The integer quantifier is implemented as a sequence of nodes with `"."` quantifiers. If a quantifier is not provided for a given query node, the default `"."` quantifier is used.

A predicate is represented as a Python `Callable` that takes the data for a node in a Hatchet GraphFrame as input and returns a Boolean. The returned Boolean is used to determine whether a GraphFrame node satisfies the predicate. If a predicate is not provided for a given query node, the default predicate is a function that always returns `True`.

Algorithm 2 Match Paths for a given Starting Node

Input:
$node$: a node that matches the first query node
$query$: the query being applied
Output: all matches to the query starting with $node$

```
function MatchPaths(node, query)
    query_idx ← 0
    matches ← [[node]]
    while query_idx < number of query nodes do
        q ← quantifier of query[query_idx]
        p ← predicate of query[query_idx]
        new_matches ← [ ]
        for each partial match, m in matches do
            prev_node ← last node of m
            if q indicates "match 1 node" then
                if prev_node satisfies p then
                    add m to new_matches
                end if
            else
                sub_matches ← all sequences of nodes that
                              start with prev_node in which
                              every node satisfies p
                if sub_matches is not empty then
                    remove prev_node from start of each
                                sequence in sub_matches
                    prepend m to each sequence in
                                sub_matches
                    add each sequence in sub_matches to
                                new_matches
                end if
            end if
        end for
        matches ← all unique sequences in new_matches
        if matches is empty then
            return ∅
        end if
        query_idx ← query_idx + 1
    end while
    return matches
end function
```

To illustrate the composition of queries, we consider the Hatchet query in Figure 2a as an example. This query uses two query nodes to find all subgraphs in the call graph rooted at MPI (or PMPI) function calls that have more than five L2 cache misses (as measured by PAPI [14]). Specifically, the first query node has the quantifier `"."` and one predicate that checks two conditions. The predicate first checks if the "name" metric matches the regular expression `"P?MPI_.*"`. Then, the predicate checks if the "PAPI_L2_TCM" metric is greater than five. Both parts of the predicate are combined using conjunction. The second query node has the quantifier `"*"`. Since no predicate is provided for the second query node, the default predicate is used. As a result, the second query node finds all nodes in the call graph between MPI functions and the leaf nodes (i.e., functions that invoke no other functions).

IV. Query Language Dialects

To simplify the use of our Query Language under diverse circumstances (e.g., creating queries in JavaScript that will be moved into Python code), we define an Object-based Dialect and a String-based Dialect. These two dialects enable alternative representations of predicates; these predicates are translated into predicates in our Query Language.

$e \in$ (Python-Style) Regular Expression
$i \in I$, where I is the set of integers
$r \in \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers

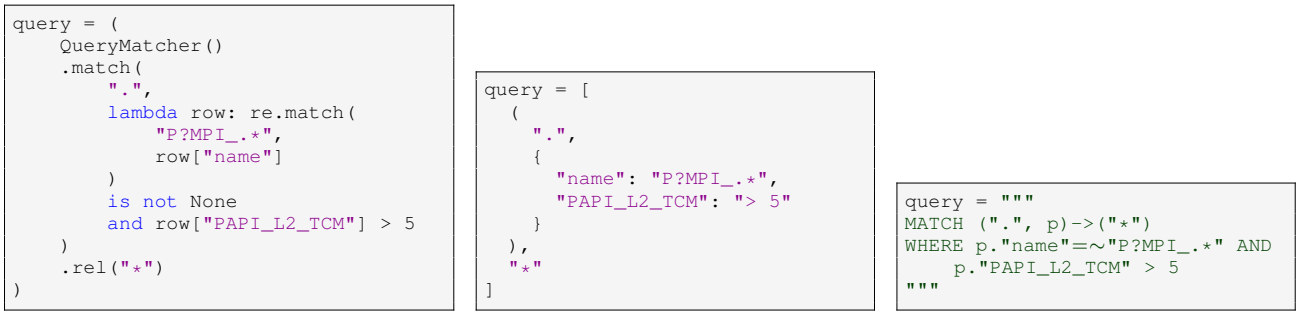
⟨*query*⟩	::=	[⟨*node_tuple*⟩]
⟨*node_tuple*⟩	::=	⟨*node*⟩
	|	⟨*node*⟩, ⟨*node_tuple*⟩
⟨*node*⟩	::=	(⟨*quantifier*⟩, ⟨*condition*⟩)
	|	⟨*quantifier*⟩
	|	⟨*condition*⟩
⟨*quantifier*⟩	::=	`"."` | `"*"` | `"+"` | i
⟨*condition*⟩	::=	⟨*cond_expr*⟩
⟨*cond_expr*⟩	::=	⟨*sing_cond*⟩
	|	⟨*sing_cond*⟩, ⟨*cond_expr*⟩
⟨*sing_cond*⟩	::=	met: ⟨*cond_val_str*⟩
	|	met: ⟨*cond_val_num*⟩
⟨*cond_val_num*⟩	::=	`r` | `< r` | `<= r` | `== r` | `> r` | `>= r`
⟨*cond_val_str*⟩	::=	e

Grammar 1: Syntax of the Object-based Dialect.

A. Object-based Dialect

The Object-based Dialect is a formal language that is built around Python's built-in objects. In the Object-based Dialect, quantifiers are represented in the same way as in the Query Language (see Section III-A). The rest of the query syntax for the Object-based Dialect is unique in terms of its predicate representation and composition. Specifically, in the Object-based Dialect, queries are composed using Python's `list`, `tuple`, and `dict` built-in data structures. A predicate is a key-value pair where the key is a metric name and the value is a Boolean expression generated by using Grammar 1. For a given query node, one or more predicates are combined into a single Python dictionary. Multiple predicates are combined using only conjunctions (i.e., AND).

To illustrate the Object-based Dialect, consider the query in Figure 2b. This query identifies the same set of call paths as the query in Figures 2a. It consists of two query nodes. The first node has the quantifier `"."` and two predicates. The first predicate is the key-value pair `"name": "P?MPI_.*"`, and the second predicate is `"PAPI_L2_TCM": "> 5"`. As explained in



```
query = (
    QueryMatcher()
    .match(
        ".",
        lambda row: re.match(
            "P?MPI_.*",
            row["name"]
        )
        is not None
        and row["PAPI_L2_TCM"] > 5
    )
    .rel("*")
)
```

```
query = [
  (
    ".",
    {
      "name": "P?MPI_.*",
      "PAPI_L2_TCM": "> 5"
    }
  ),
  "*"
]
```

```
query = """
MATCH (".", p)->("*")
WHERE p."name"=~"P?MPI_.*" AND
    p."PAPI_L2_TCM" > 5
"""
```

(a) Query Language (b) Object-based Dialect (c) String-based Dialect

Fig. 2: Examples of a query identifying the same set of call paths but using different languages (i.e., Query Language, the Object-based Dialect, and the String-based Dialect).

Section IV-B, the second query node has the quantifier `"*"` and the default "always-true" predicate.

B. String-based Dialect

The String-based Dialect is a formal language that can be used to create queries using a syntax derived from Cypher [12]. In the String-based Dialect, a query quantifier has the same representation as in the Query Language and the Object-based Dialect. On the other hand, predicates are represented as Boolean expressions that are created using Grammar 2. To extract one or more paths from profiling data, users can deploy one or more quantifiers and predicates.

Queries generated using the String-based Dialect contain two main syntactic pieces: a `MATCH` statement and a `WHERE` statement. The `MATCH` statement starts with the `MATCH` keyword and defines the quantifiers and variable names used to refer to query nodes in the predicates. The `WHERE` statement starts with the `WHERE` keyword and defines one or more predicates. Multiple predicates can be combined using three Boolean operators: conjunction (i.e., AND), disjunction (i.e., OR), and complement (i.e., NOT). Each individual predicate takes the form of `<variable name>."<metric name>"<comparison operation>`. Grammar 2 shows the full String-based Dialect syntax.

To illustrate the String-based Dialect, consider the query in Figure 2c. This query identifies the same set of call paths as the queries in Figures 2a and 2b. Once again, it consists of two query nodes. The first node has the quantifier `"."` and two predicates. The first predicate is the expression `p."name"=~ "P?MPI_.*"`, and the second predicate is `p."PAPI_L2_TCM"> 5`. As explained in Section III-A, the second query node has the quantifier `"*"` and the default "always-true" predicate.

V. Capability Classifications

Through quantifiers and predicates, we capture all the paths that match the properties expressed in the query. To facilitate the use of our Query Language and its dialects as well as to illustrate the differences across the three query representations, we classify queries in terms of their properties and

$a \in A$, where A is the set of variable names
$i \in I$, where I is the set of integers
$m \in M$, where M is the set of metric names
$s \in S$, where S is the set of string literals
$e \in$ (Python-style) Regular Expression
$r \in \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers

⟨*query*⟩ ::= ⟨*path_expr*⟩ | ⟨*path_expr*⟩ ⟨*cond_expr*⟩
⟨*path_expr*⟩ ::= `MATCH` ⟨*path*⟩
⟨*path*⟩ ::= ⟨*node_expr*⟩ | ⟨*node_expr*⟩ `->` ⟨*path*⟩
⟨*node_expr*⟩ ::= (⟨*node*⟩)
⟨*node*⟩ ::= ⟨*quantifier*⟩, a | ⟨*quantifier*⟩ | a
⟨*quantifier*⟩ ::= `"."` | `"*"` | `"+"` | i
⟨*cond_expr*⟩ ::= `WHERE` ⟨*condition*⟩
⟨*condition*⟩ ::= ⟨*unary_cond*⟩ | ⟨*binary_cond*⟩
⟨*binary_cond*⟩ ::= ⟨*cond*⟩ `AND` ⟨*cond*⟩ | ⟨*cond*⟩ `OR` ⟨*cond*⟩
⟨*unary_cond*⟩ ::= ⟨*sing_cond*⟩ | `NOT` ⟨*sing_cond*⟩
⟨*sing_cond*⟩ ::= ⟨*str_cond*⟩ | ⟨*num_cond*⟩ | ⟨*exists_cond*⟩
⟨*str_cond*⟩ ::= a.m = s
| a.m `STARTS WITH` s
| a.m `ENDS WITH` s
| a.m `CONTAINS` s
| a.m =~ e
⟨*num_cond*⟩ ::= a.m = r
| a.m < r | a.m <= r
| a.m > r | a.m >= r
| a.m `IS NAN`
| a.m `is NOT NAN`
| a.m `IS INF`
| a.m `IS NON INF`
⟨*exists_cond*⟩ ::= a.m `IS NONE` | a.m `IS NOT NONE`

Grammar 2: Syntax of the String-based Dialect.

logical operators. Specifically, we classify properties into five categories, one for quantifiers and four for predicates. They are as follows:

- *Quantifier Capabilities*: match one, zero or more, one or more, or an exact number of nodes;
- *String Equivalence and Regex Matching Predicates*: match if the value of the specified string metric is equal to a provided string or matches a provided regular expression;
- *String Containment Predicates*: match if the value of the specified string metric starts with, ends with, or contains a provided string;
- *Basic Numeric Comparison Predicates*: match if the value of the specified numeric metric satisfies the numeric comparison (e.g., equal to, greater than, greater than or equal to); and
- *Special Value Identification Predicates*: match if the value of the specified metric is equivalent to the provided "special value" (i.e., NaN, infinity, or None).

We can match multiple properties by combining multiple predicates with logical operators. We classify the logical operators using three categories. They are as follows:

- *Predicate Combination through Conjunction*: combine predicates using conjunction (i.e., logical AND);
- *Predicate Combination through Disjunction and Complement*: combine predicates using disjunction (i.e., logical OR) or find the complement (i.e., logical NOT) to a single predicate; and
- *Predicate Combination through Other Operations*: combine predicates through other means, such as exclusive disjunction (i.e., logical XOR).

Table I presents the five property categories and the three categories of logical operators in relation to the Query Language and its dialects. Note that not all categories are supported across all three ways of creating queries. For the *Property Category*, when using the Object-based Dialect, we do not support the generation of queries with string containment predicates and special value identification predicates. Adding them would introduce syntactic complexity to queries not justified by the intended simplifications of the Object-based Dialect. For the *Logical Operation Category*, Object-based Dialect predicates are represented through Python dictionaries, and thus only one operation can be supported (i.e., conjunction). Furthermore, predicate combinations through other operations besides conjunction, disjunction, and complement are currently not supported in the dialects. We provide users with a suite of Jupyter notebooks containing use cases for all these categories in [15].

Property Category	*QL*	*Object*	*String*
Quantifier Capabilities	✓	✓	✓
String Equivalence and Regex Matching Predicates	✓	✓	✓
String Containment Predicates	✓		✓
Basic Numeric Comparison Predicates	✓	✓	✓
Special Value Identification Predicates	✓		✓
Logical Operator Category	*QL*	*Object*	*String*
Predicate Combination through Conjunction	✓	✓	✓
Predicate Combination through Disjunction and Complement	✓		✓
Predicate Combination through Other Operations	✓		

TABLE I: Support for each property and logical operator category in Query Language (QL), Object-based Dialect, and String-based Dialect.

VI. Demonstrating the New Hatchet Capabilities

To demonstrate the new capabilities of Hatchet with our Query Language and its dialects, we present three case studies. In the first case study, we use the Query Language to compare sequential and multi-threaded versions of a graph alignment application called Fido [6], [7]. In the second case study, we use the Object-based Dialect to examine the performance of two MPI libraries in the AMG2013 [16] mini-application and locate a potential root cause of performance differences between the libraries. In the third case study, we examine the use of the String-based Dialect to reduce profiling data through Hatchet's call path visualization tool [17].

A. Case Study 1: Sequential vs. Multi-threaded Fido

In this case study, we evaluate the effectiveness of Hatchet once augmented with our Query Language to compare and contrast the performance of the sequential and parallel implementations of the graph alignment application Fido [6], [7]. Fido builds on the GRAAL algorithm [18]. Tools such as Fido can be used for the alignment of DNA, protein, and codon in bioinformatics and medicine. We consider two different versions of Fido: the original, sequential version and a multi-threaded version implemented using the portability library RAJA [19]. We run both versions of the code on a single node of LLNL's Lassen supercomputer, where each node contains two IBM Power9 CPUs and four NVIDIA Volta V100 (though only the CPUs were used in this work). For the multi-threaded version of Fido, we use 40 OpenMP threads. We profile both versions of Fido using HPCToolkit [5].

After profiling the two versions of the application, we first use pandas [10], [11], matplotlib [20], and the original Hatchet (without our Query Language) to measure the percentage of total execution time spent in each function call. The results of this analysis are shown in Figure 3. All functions that take less than one second are summed into a single value called "Remaining Time". We observe that `malloc` is the largest contributor to the execution time of both the sequential and multi-threaded versions of Fido. This suggests that there is some type of memory inefficiency in Fido that needs to be addressed.

Because the `malloc` function takes such a large percentage of the execution time in both versions, other potential bottlenecks directly linked to the source code of Fido are not explicitly revealed when using the original Hatchet. The Query Language can play an important role in revealing such bottlenecks. To this end, we use the Query Language to remove

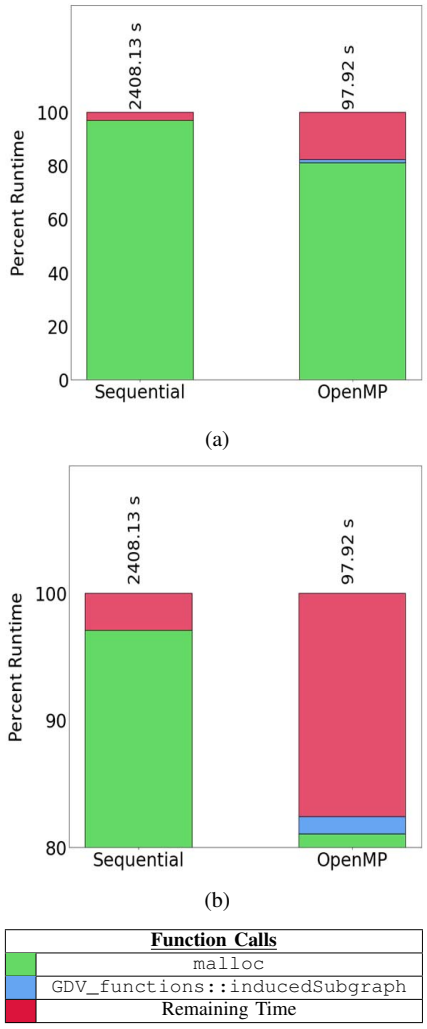


Fig. 3: Fido's percent of overall runtime spent executing functions that take longer than one second. All functions that take less than one second are merged into "Remaining Time". For readability purposes, Figure 3b zooms in on the upper range of Figure 3a (note: y-axis range is now 80-100%).

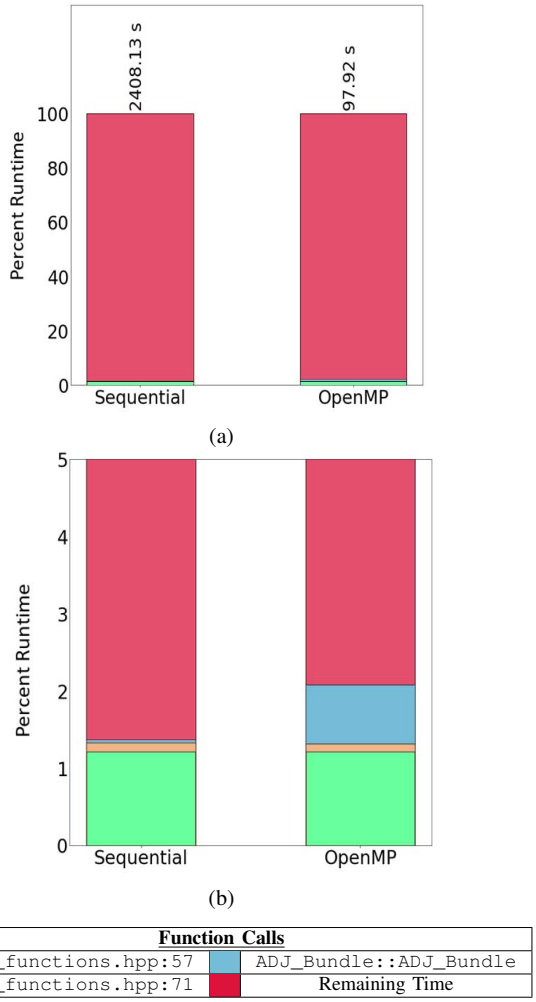
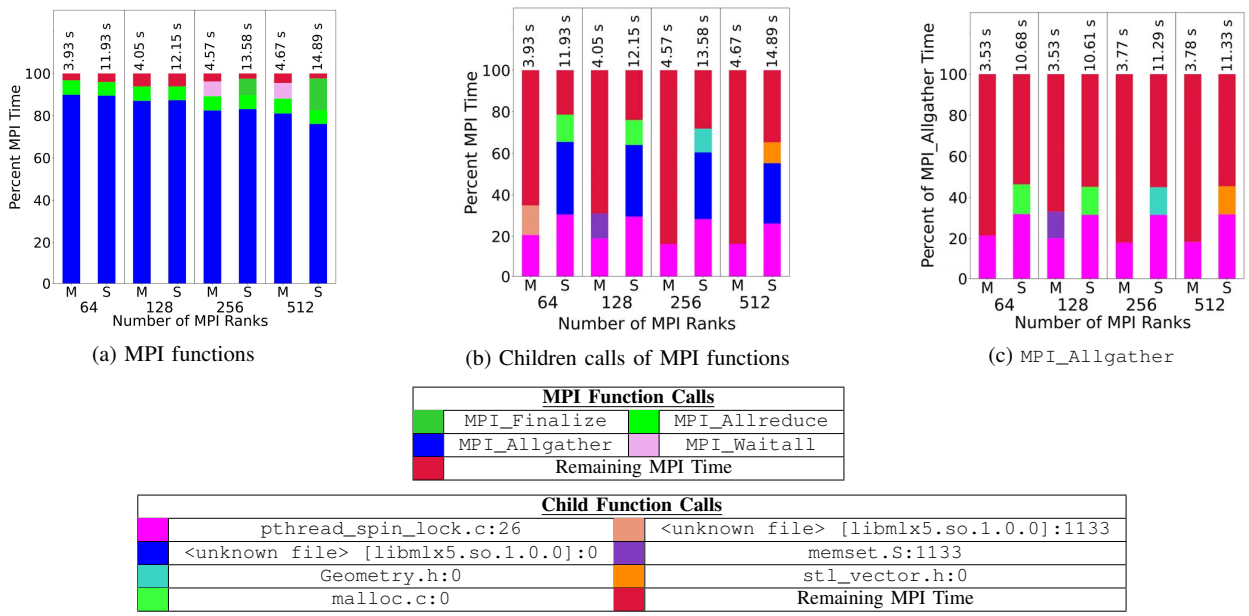


Fig. 4: Fido's percent of overall runtime spent in functions that take longer than one second after using the Query Language to hide `malloc`, standard language functions, and compiler functions. All functions that take less than one second are merged into "Remaining Time". For readability purposes, Figure 4b zooms in on the lower range of Figure 4a (note: y-axis range is now 0-5%). GDV_Functions.hpp represents loops in the `GDV_functions::inducedSubgraph` function from Figure 3.

all instances of `malloc` from the profiling data. Furthermore, we add other predicates to our query to remove standard language and compiler functions such as those with names starting with C++'s `std::` namespace or those with names containing libc and gcc. The resulting paths highlight the impact of functions related directly to Fido's source code and algorithm.

We repeat the analysis on the filtered profiling data without `malloc` as well as the removed standard language and compiler functions. Figure 4 shows the results for the filtered profiling data. Again, all functions that take less than one second are summed into "Remaining Time". The two items identified by GDV_Functions.hpp represent loops in the `GDV_functions::inducedSubgraph` function from Figure 3. We observe that the `ADJ_Bundle::ADJ_Bundle` function takes a longer percentage of total runtime in the OpenMP version of Fido than in the sequential version. Although the difference in overall runtime (as indicated by the numbers above the bars in Figure 4a) causes the OpenMP version of `ADJ_Bundle::ADJ_Bundle` to be slightly faster than the sequential version, the difference in percentage suggests that this function could be a target for further performance improvements. Furthermore, we observe that most Fido functions take less than one second and thus are summed into the "Remaining Time" value. This suggests that there are no other key functions that impact performance in isolation and can be individually optimized. On the other hand,



(a) MPI functions (b) Children calls of MPI functions (c) `MPI_Allgather`

Fig. 5: AMG2013's percent of (a) total MPI time spent in MPI functions, (b) total MPI time spent in the children calls of MPI functions, and (c) total `MPI_Allgather` time spent in children calls. We denote the MVAPICH and Spectrum-MPI libraries by *M* and *S*. Results in Figures 5b and 5c are generated from profiling data using the Object-based Dialect to obtain the child calls of all the MPI functions.

all the functions identified by our Query Language should be considered in concert for any further optimization.

B. Case Study 2: MVAPICH vs. Spectrum-MPI in AMG2013

In this case study, we evaluate the effectiveness of Hatchet when using the Object-based Dialect to identify sources of performance losses associated with MPI calls in the AMG2013 [16] mini-application. 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 runs using HPCToolkit [5].

First, we use pandas [10], [11], matplotlib [20], and the original Hatchet (without our Query Language) to determine the amount of time spent in functions defined by the MPI standard. Figure 5a shows the results of this initial analysis. In the figure, we use M and S for the MVAPICH and Spectrum-MPI libraries, respectively. A first insight from the figure is that MVAPICH outperforms Spectrum-MPI. To identify possible causes of the performance differences, we use the augmented Hatchet and its Object-based Dialect to generate filtered profiling data containing the children calls of all the MPI functions. We use the Object-based Dialect rather than the Query Language or the String-based Dialect because of its simplicity in providing us with the call paths containing all the children calls. Figure 5b shows the total MPI time spent in each children call. We observe that Spectrum-MPI spends a large amount of time in `libmlx5.so` (i.e., the Mellanox InfiniBand user-space driver). Note that MVAPICH is also using the same function. Thus, the worse performance of Spectrum-MPI may be linked to differences in its use of the driver.

Figure 5a also shows that only four MPI calls have time greater than 5% of the total MPI runtime with `MPI_Allgather` taking more than 75% of the total MPI time across the executions. This suggests that another possible reason for the performance differences is in the `MPI_Allgather` function. Thus, we re-examine the profiling data for the two AMG2013 versions using the Object-based Dialect to extract call paths containing only functions called by `MPI_Allgather`. Figure 5c shows the total time spent in each of `MPI_Allgather`'s children calls. We observe that the time spent in the `pthread_spin_lock` function is consistently larger in Spectrum-MPI than in MVAPICH. Thus, the worse performance of Spectrum-MPI may be linked to differences in its use of `pthread_spin_lock`.

The conclusions shown in Figures 5b and 5c are made possible only with the support of the Object-based Dialect.

C. Case Study 3: Interactive Visualization of Call Paths

In this case study, we demonstrate Hatchet's interactive call path visualization [17] and its use of the String-based Dialect. The interactive call path visualization presented in Figure 6 addresses the users' need for more robust visualization options when dealing with Hatchet GraphFrames. To augment users' data analysis workflows, this visualization is designed for use inside of Jupyter notebooks. Hatchet uses a library called Roundtrip [21] to manage the passing of code and data between the runtime context of the Jupyter notebook and the JavaScript which powers the visualization. Note that Hatchet assumes that the visualized call paths take the form of a tree-based data structure. This tree-based representation of call paths is referred to as a calling context tree.

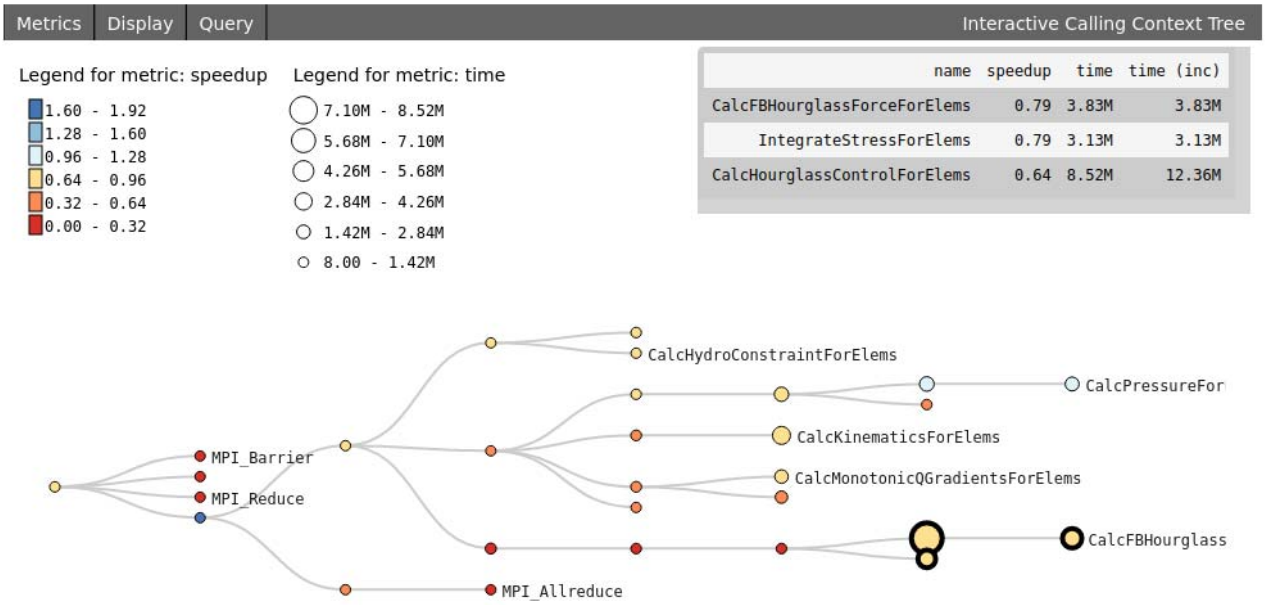


Fig. 6: An example of Hatchet's calling context tree visualization. The String-based Dialect enables users to collapse tree-based representation of call paths in this visualization and return a description of the representation in the form of an automatically generated query.

Most relevant to the work in this paper, Hatchet's call path visualization supports the exporting of changes made to a tree through point and click interactions. A user can collapse the tree-based representation of call paths in the visualization and return a description of the modified representation in the form of an automatically generated query written in the String-based Dialect. This query can, in turn, be used to synchronize the visualization with the original profiling data.

We use the String-based Dialect rather than the Query Language because of the Roundtrip interface which manages the transfer of data. Since the query is constructed on the JavaScript side of the visualization, it must be transferred back to the Jupyter side as a string. While it is possible to pass a string-encoded Python program back to the Jupyter notebook, doing so introduces security risks and undesirable room for error. Furthermore, we use the String-based Dialect rather than the Object-based Dialect because the latter does not support the creation of complex predicates using Boolean logic beyond conjunction. In other words, the Object-based Dialect supports the *Predicate Combination through Conjunction* capability category, but not the other logical operator categories. As a result, divergent tree-based representations of call paths cannot be described by the Object-based Dialect even though it is very common to produce such representations with point and click manipulation.

Using the String-based Dialect allows us to generate queries to pass back to Jupyter as a simple string and generate complex queries with predicates combining options beyond conjunction on the JavaScript side of the calling context tree visualization. Furthermore, the String-based Dialect enables us to easily store a query into a single variable that users can pass directly into a filter function. This simplifies and abstracts the process of applying changes made in the visualization to a Hatchet GraphFrame. Finally, by representing queries as strings, the String-based Dialect allows queries produced by the calling context tree visualization to be easily saved to file. This allows changes made with the visualization to be easily shared for the purposes of reproducibility and replicability [22].

To evaluate the call path visualization supported by the String-based Dialect, we use HPCToolkit-generated profiling data collected from two KRIPKE [23] executions. KRIPKE is a mini-application developed at LLNL to serve as a proxy for a fully functional discrete-ordinates transport code. KRIPKE is designed to support different in-memory data layouts, and allows work to be grouped into sets in order to expose more on-node parallelism. The profiling data is from runs on 64 and 128 cores, resulting in trees of 1500 and 2700 nodes, respectively. The visualization allows us to significantly reduce the tree-based representation of call paths down from 1500 mostly irrelevant nodes to just over 100 nodes. This massive data reduction results in significantly faster runtimes in subsequent executions of the visualization on the reduced dataset. Furthermore, the data reduction results in more comprehensible visualizations as identified "noisy" call paths can be eliminated altogether.

VII. Related Work

In our work, we borrow key ideas of graph query languages used in graph databases and adapt it to analyze profiling data. The novelty of our work is in the use of the concepts for a new domain such as the effective and scalable analysis of large profiling datasets in scientific applications. Other examples of successful use of graph processing can be found in numerous areas of computer science (e.g., machine learning, computational sciences, medicine, and social media) [24] but not in HPC performance analysis. For example, graph databases have been developed to enable storing, manipulating, and analyzing large, dynamic graph datasets [24]. Graph databases are a type of NoSQL database that use some representation of a graph (e.g., adjacency matrix, adjacency list) to store data rather than a fixed, table-based schema [24]. Some examples of graph databases are Neo4j [25] and Amazon Neptune [26]. Additionally, these systems also provide some form of language (sometimes called a graph query language) to enable creation, modification, access, and traversal of the dataset [27]. These languages are usually based on some form of pattern matching, often involving finding all matches to some abstract path or subgraph within the dataset [27]. Examples for graph query languages are Cypher [12], Gremlin [28], and the in-development ISO standard GQL [29].

VIII. Conclusions

In this work, we present a novel Call Path Query Language and its two dialects for the in-depth analysis of profiling data from scientific applications. We augmented Hatchet with our Query Language and its dialects to provide new analysis capabilities. The augmented Hatchet enables users to discover insights into applications that would not otherwise be observable with the original Hatchet library or traditional performance analysis tools. In the case studies covered in this paper, our Query Language and its dialects identify specific functions that can be further optimized, attribute poor performance to specific functions, and reduce the size of call paths from 1500 mostly irrelevant nodes to just 100 relevant ones. Additionally, the String-based Dialect of our Query Language enables easy and safe interaction between Hatchet and other tools (e.g., JavaScript-based visualizations).

In future work, we plan to use our Query Language and its dialects to examine performance of additional HPC applications and their underlying software stacks, including in-situ scientific workflows for studying protein structure changes associated with phenomena such as protein-protein and protein-ligand interaction as well as membrane material properties.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory (LLNL) under Contract DE-AC52-07NA27344 (LLNL-CONF-835691). The UTK group acknowledges the support of NSF (#1841758, #1900888, and #2138811). The authors thank Dr. Abhinav Bhatele (University of Maryland), Dr. Todd Gamblin (LLNL), and Sanjukta Bhowmick (University of Northern Texas) for their feedback during the preparation of this work.

Available Jupyter Notebooks

A suite of Jupyter notebooks containing use cases of queries using our Query Language and the two dialects can be found at: https://github.com/LLNL/hatchet-tutorial.

References

[1] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a Call Graph Execution Profiler," pp. 120–126, 1982. [Online]. Available: https://doi.org/10.1145/800230.806987

[2] D. Böhme, T. Gamblin, D. Beckingsale, P. Bremer, A. Giménez, M. P. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance Introspection for HPC Software Stacks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, J. West and C. M. Pancake, Eds. IEEE Computer Society, 2016, pp. 550–560. [Online]. Available: https://doi.org/10.1109/SC.2016.46

[3] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011 - Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, ZIH, Dresden, September 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Springer, 2011, pp. 79–91. [Online]. Available: https://doi.org/10.1007/978-3-642-31476-6_7

[4] S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006. [Online]. Available: https://doi.org/10.1177/1094342006064482

[5] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs," *Concurr. Comput. Pract. Exp.*, vol. 22, no. 6, pp. 685–701, 2010. [Online]. Available: https://doi.org/10.1002/cpe.1553

[6] D. Chapp, N. Tan, S. Bhowmick, and M. Taufer, "Identifying Degree and Sources of Non-Determinism in MPI Applications Via Graph Kernels," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 12, pp. 2936–2952, 2021. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3081530

[7] P. Bell, K. Suarez, D. Chapp, N. Tan, S. Bhowmick, and M. Taufer, "ANACIN-X: A Software Framework for Studying Non-Determinism in MPI Applications," *Softw. Impacts*, vol. 10, p. 100151, 2021. [Online]. Available: https://doi.org/10.1016/j.simpa.2021.100151

[8] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the Overgrowth in Parallel Profiles," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, M. Taufer, P. Balaji, and A. J. Peña, Eds. ACM, 2019, pp. 20:1–20:21. [Online]. Available: https://doi.org/10.1145/3295500.3356219

[9] 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 *IEEE/ACM International Workshop on HPC User Support Tools and Workshop on Programming and Performance Visualization Tools, HUST/ProTools@SC 2020, Atlanta, GA, USA, November 18, 2020*. IEEE, 2020, pp. 49–58. [Online]. Available: https://doi.org/10.1109/HUSTProtools51951.2020.00013

[10] The pandas Development Team, "pandas-dev/pandas: Pandas," Aug. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7018966

[11] W. McKinney, "Data Structures for Statistical Computing in Python ," in *Proc. of the 9th Python in Science Conference*, 2010, pp. 56 – 61. [Online]. Available: https://doi.org/10.25080/Majora-92bf1922-00a

[12] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An Evolving Query Language for Property Graphs," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 1433–1445. [Online]. Available: https://doi.org/10.1145/3183713.3190657

[13] J. R. Ullmann, "An Algorithm for Subgraph Isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976. [Online]. Available: http://doi.acm.org/10.1145/321921.321925

[14] D. Terpstra, H. Jagode, H. You, and J. J. Dongarra, "Collecting Performance Data with PAPI-C," in *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Springer, 2009, pp. 157–173. [Online]. Available: https://doi.org/10.1007/978-3-642-11261-4_11

[15] S. Brink, I. Lumsden, J. Luettgau, V. Lama, C. Scully-Allison, K. E. Isaacs, M. Taufer, and O. Pearce, "Hatchet Tutorial: A Repository Containing Materials for Hatchet's Hands-On Tutorials," https://github.com/LLNL/hatchet-tutorial, 2017-2021.

[16] U. Yang, R. Falgout, and J. Park, "Algebraic Multigrid Benchmark, Version 00," Aug. 2017. [Online]. Available: https://www.osti.gov/biblio/1389816

[17] C. Scully-Allison, I. Lumsden, K. Williams, J. Bartels, M. Taufer, S. Brink, A. Bhatele, O. Pearce, and K. E. Isaacs, "Designing an Interactive, Notebook-Embedded, Tree Visualization to Support Exploratory Performance Analysis," May 2022. [Online]. Available: https://arxiv.org/abs/2205.04557

[18] O. Kuchaiev, T. Milenković, V. Memišević, W. Hayes, and N. Pržulj, "Topological Network Alignment Uncovers Biological Function and Phylogeny," *J. R. Soc. Interface.*, vol. 7, pp. 1341–1354, Sep. 2010. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsif.2010.0063

[19] D. Beckingsale, T. R. W. Scogland, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, and B. S. Ryujin, "RAJA: Portable Performance for Large-Scale Scientific Applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC 2019, Denver, CO, USA, November 22, 2019*. IEEE, 2019, pp. 71–81. [Online]. Available: https://doi.org/10.1109/P3HPC49587.2019.00012

[20] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://doi.org/10.1109/MCSE.2007.55

[21] C. Scully-Allison and J. Bartels, "Roundtrip," https://github.com/hdc-arizona/roundtrip, 2019.

[22] M. A. Heroux, L. Barba, M. Parashar, V. Stodden, and M. Taufer, "Toward a Compatible Reproducibility Taxonomy for Computational and Computing Sciences," Oct. 2018. [Online]. Available: http://www.osti.gov/servlets/purl/1481626

[23] A. Kunen, T. Bailey, and P. Brown, "KRIPKE - A Massively Parallel Transport Mini-App," Jun. 2015. [Online]. Available: https://www.osti.gov/biblio/1229802

[24] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying Graph Databases: Analysis and Taxonomy of Data Organization System Designs, and Graph Queries," 2019. [Online]. Available: http://arxiv.org/abs/1910.09017

[25] Neo4j, Inc, "The Neo4j Graph Data Platform," San Mateo, CA, 2021, (accessed 2022-08-25). [Online]. Available: https://neo4j.com/product/

[26] B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, S. Rondelli, A. Ryazanov, M. Schmidt, K. Sengupta, B. B. Thompson, D. Vaidya, and S. Wang, "Amazon Neptune: Graph Data Management in the Cloud," in *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*, ser. CEUR Workshop Proceedings, M. van Erp, M. Atre, V. López, K. Srinivas, and C. Fortuna, Eds., vol. 2180. CEUR-WS.org, 2018. [Online]. Available: http://ceur-ws.org/Vol-2180/paper-79.pdf

[27] R. Angles and C. Gutiérrez, "Survey of Graph Database Models," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1:1–1:39, 2008. [Online]. Available: https://doi.org/10.1145/1322432.1322433

[28] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, J. Cheney and T. Neumann, Eds. ACM, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1145/2815072.2815073

[29] A. Green, P. Furniss, P. Lindaaker, P. Selmer, H. Voigt, and S. Plantikow, "GQL Scope and Features," ISO, Tech. Rep., 2019. [Online]. Available: https://s3.amazonaws.com/artifacts.opencypher.org/website/materials/sql-pg-2018-0046r3-GQL-Scope-and-Features.pdf