



Performance Analysis with Hatchet

2 May 2022

Stephanie Brink, Olga Pearce



Getting Hatchet Tutorial Materials

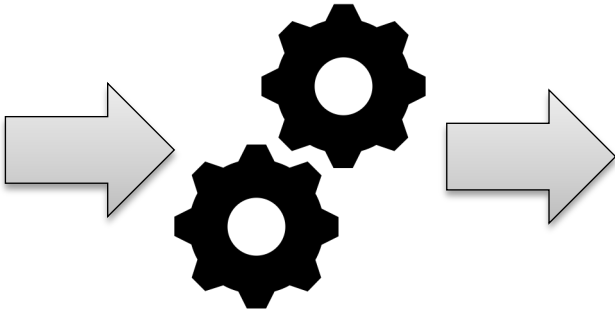
- The SPOT container includes a sample Jupyter notebook, Hatchet 2022.1.0 install, and Lulesh datasets.
 - Alternatively, the sample Jupyter notebook and the Lulesh datasets are available directly at <https://github.com/llnl/hatchet-tutorial>. This repository is integrated with BinderHub, which will create a local interactive environment for you to run the notebook.
- Following this tutorial, you can substitute your own SPOT/Caliper data files into the example notebook.
- We'll use this material in the live demo portion of the tutorial.

Automated Application Performance Analysis: Caliper → SPOT → Hatchet

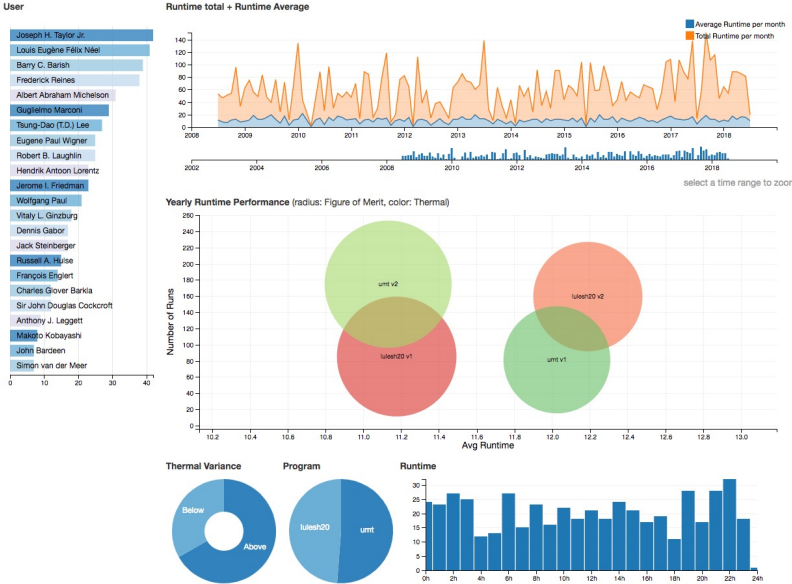
```
#include <caliper/cali.h>

static inline
void LagrangeElements(Domain&
domain, Index_t numElem)
{
    CALI_CXX_MARK_FUNCTION;
    // ...
}
```

Caliper instrumentation
in the application



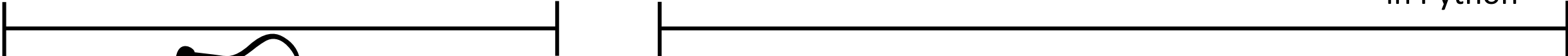
At runtime: Performance
and Metadata Collection



Web-based Visualization and
Analysis Tools



Analyze
caliper datasets
in Python



SPOT and Hatchet

c/o D Boehme

*Hatchet can analyze other datasets (HPCToolkit, gprof, TAU, Ascent (WIP))

SPOT Web Interface: Run Table and Jupyter Notebooks

All records selected. Please click on the graph to apply filters.

The screenshot displays a Jupyter Notebook interface. On the left, a table lists merit figures and their associated parameters:

Figure_of_merit	Problem_size	Threads	Jobsize	launchdate
1				
5491.526855	60	36	1	2019-Jul-16 14:03
12412.877512	30	4	8	2019-Jul-16 14:04
5525.153912	60	36	1	2019-Jul-16 14:05
11339.841229	30			
5100.923858	60			
11197.780437	30			
5071.148951	60			
10247.931707	30			
18468.799766	60			
18666.324557	60			

The main notebook area shows a Jupyter Notebook interface with the following content:

```

In [4]: # Print the tree representation using the inclusive time metric
print(gf.tree(metric_column="time (inc)"))

```

The output is a tree representation of the computational graph, showing the time taken for each node. The root node is 'main' with a time of 79.951. The tree structure is as follows:

```

79.951 main
├── 79.886 lulesh.cycle
│   ├── 79.840 LagrangeLeapFrog
│   │   ├── 0.871 CalcTimeConstraintsForElems
│   │   ├── 21.696 LagrangeElements
│   │   │   ├── 16.367 ApplyMaterialPropertiesForElems
│   │   │   │   ├── 16.248 EvalEOSForElems
│   │   │   │   └── 12.419 CalcEnergyForElems
│   │   ├── 2.395 CalcLagrangeElements
│   │   ├── 2.260 CalcKinematicsForElems
│   │   ├── 2.870 CalcQForElems
│   │   └── 1.389 CalcMonotonicQForElems
│   └── 57.258 LagrangeNodal
│       ├── 56.424 CalcForceForNodes
│       │   ├── 56.273 CalcVolumeForceForElems
│       │   │   ├── 44.650 CalcHourglassControlForElems
│       │   │   │   ├── 14.380 CalcFBHourglassForceForElems
│       │   │   │   └── 11.210 IntegrateStressForElems
│       └── 0.032 TimeIncrement

```

A legend is provided for the time metric:

```

Legend (Metric: time (inc))
71.96 - 79.95
55.98 - 71.96
39.99 - 55.98
24.01 - 39.99
8.02 - 24.01
0.03 - 8.02

```

The bottom of the notebook shows the user code and the graph visualization options:

```

name User code
Only in left graph
Only in right graph

```

Buttons bring up Jupyter notebook or specialized analysis views

Jupyter notebook contains
Hatchet functions

c/o D Boehme

Hatchet is a performance analysis tool for parallel profiles



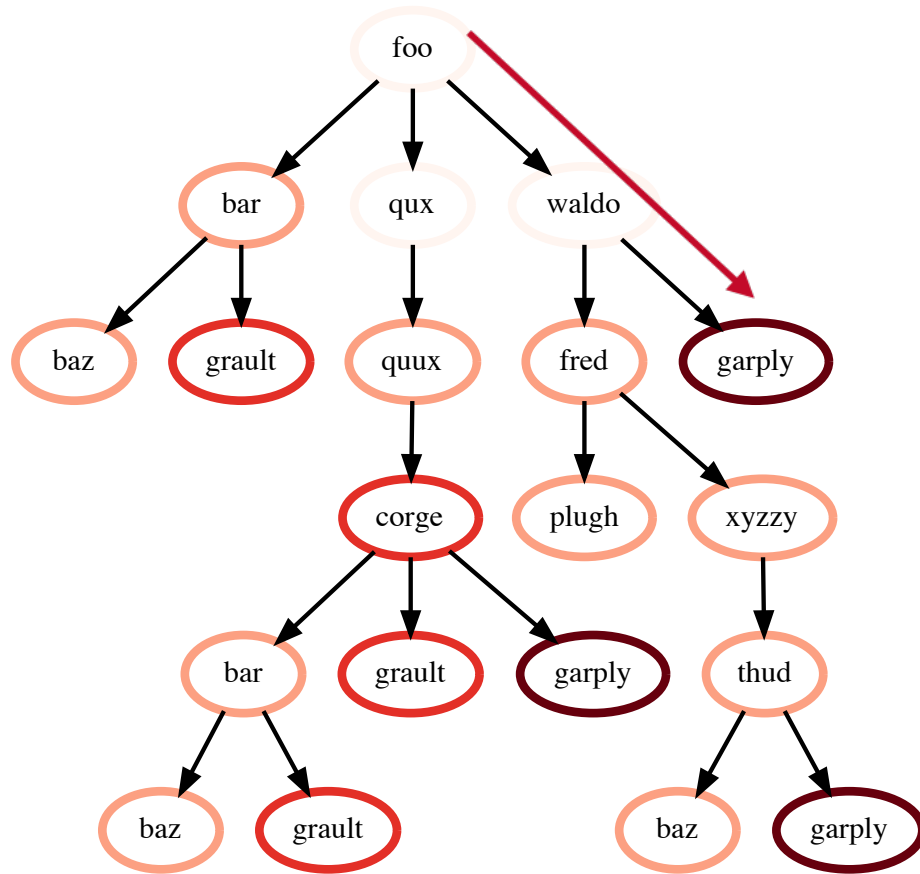
- Identify performance bottlenecks to enhance application development
 - Profiling and tracing tools (*e.g.*, Caliper, HPCToolkit, TAU, Score-P, Gprof, Callgrind) provide insights into parts of the code that consume the most time
- Hatchet is an open-source python-based tool for enabling programmatic analysis of structured (or hierarchical) data
- Hatchet can be used to sub-select and focus on a specific region of the data, compare multiple execution profiles, and automate analysis in python scripts



<https://github.com/llnl/hatchet/>



What do profiling/tracing tools collect?



Calling Context Tree (CCT)

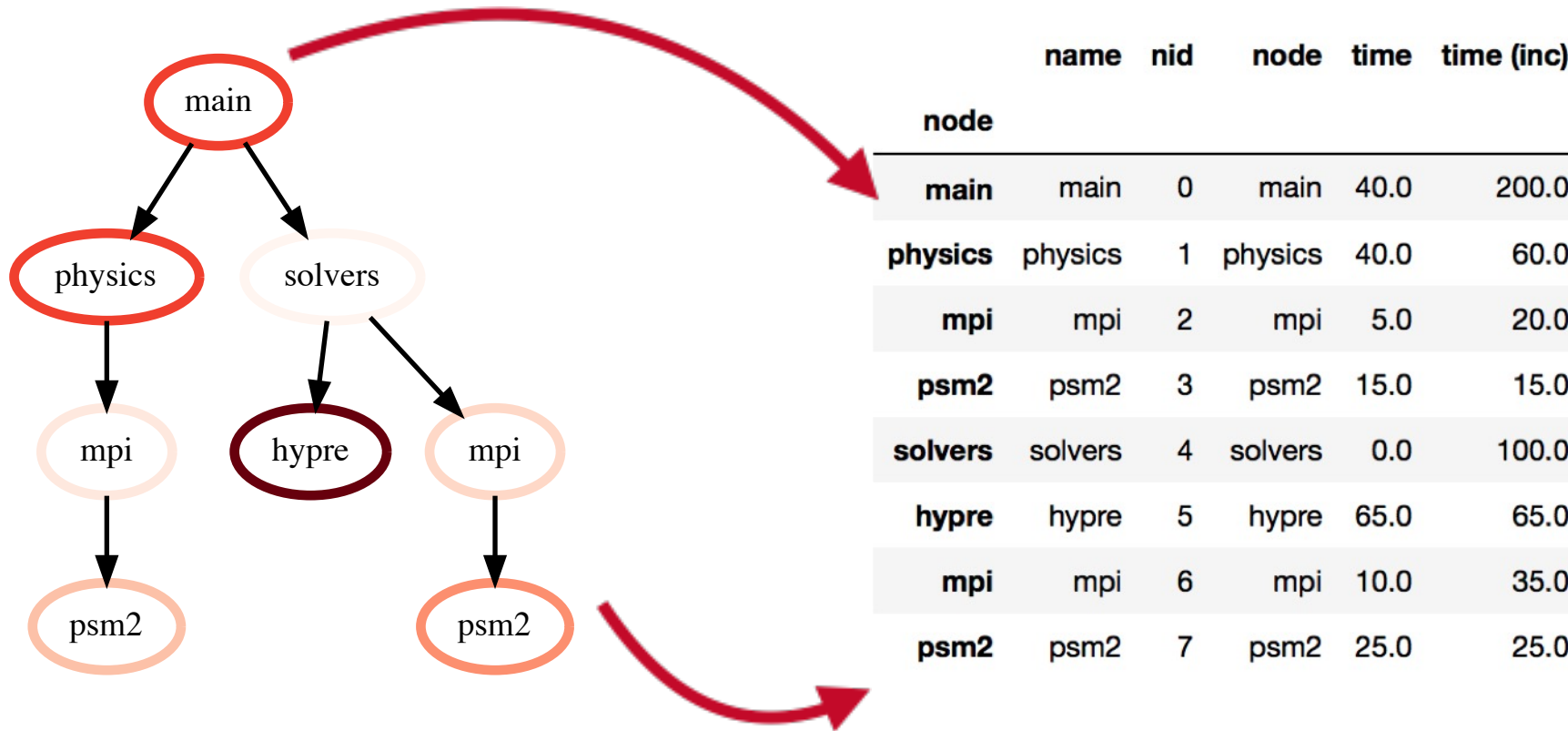
Each node may contain:

- Contextual Info
 - File
 - Line number
 - Function name
 - Callpath
 - Load module
 - Rank ID
 - Thread ID
- Performance Metrics
 - Time
 - Flops
 - Cache misses

Hatchet can read profiles from:

- Caliper
- HPCToolkit
- Gprof
- TAU
- Ascent (WIP)

Hatchet's *GraphFrame*: a Graph and a Dataframe



Graph: Stores relationships between parents and children

Pandas Dataframe: 2D table storing numerical data associated with each node (may be unique per rank, per thread)

Visualizing Hatchet's GraphFrame components

```
>>> print(gf.tree()) # print graph
>>> print(gf.dataframe) # print dataframe
```

```
0.000 foo
├─ 6.000 bar
│   └─ 5.000 baz
├─ 0.000 qux
│   └─ 5.000 quux
│       ├── 10.000 corge
│       ├── 15.000 garply
│       └─ 1.000 grault
└─ 15.000 waldo
    ├── 3.000 fred
    │   └─ 5.000 plugh
    └─ 15.000 garply
```

Legend (Metric: time)

■ 13.50 - 15.00
■ 10.50 - 13.50
■ 7.50 - 10.50
■ 4.50 - 7.50
■ 1.50 - 4.50
■ 0.00 - 1.50

name User code



Only in left graph

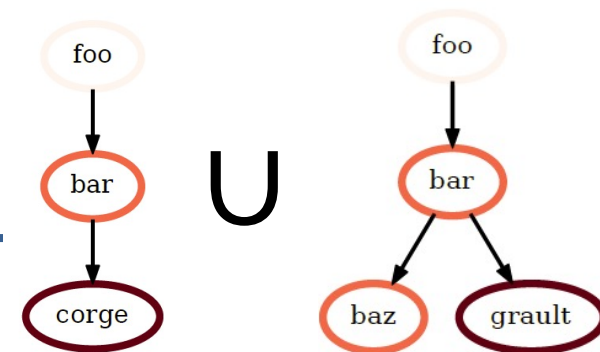


Only in right graph

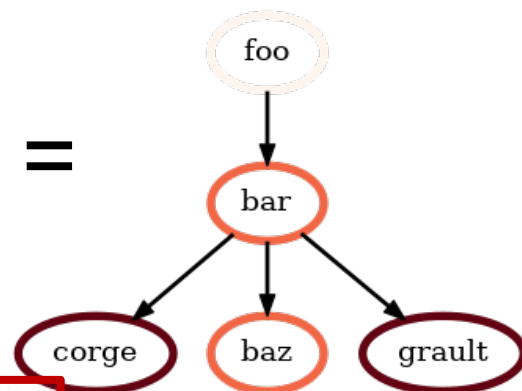
	name	time	time (inc)
node			
{'name': 'foo'}	foo	0.0	130.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name': 'grault'}	grault	10.0	10.0
{'name': 'quux'}	quux	0.0	60.0
{'name': 'quux'}	quux	5.0	60.0
{'name': 'corge'}	corge	10.0	55.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name': 'grault'}	grault	10.0	10.0
{'name': 'garply'}	garply	15.0	15.0
{'name': 'grault'}	grault	10.0	10.0

Compare GraphFrames using division (or add, subtract, multiply)

```
>>> gf3 = gf1 / gf2 # divide graphframes
```



*First, unify two trees since structure is different



gf3		gf1		gf2
0.000 foo		0.000 foo		0.000 foo
├ 2.000 bar		├ 6.000 bar		├ 3.000 bar
├ ─ 5.000 baz		├ ─ 5.000 baz		├ ─ 1.000 baz
├ inf qux	=	├ 3.000 qux	/	├ 0.000 qux
├ ─ 4.000 quux		├ 2.000 quux		├ 0.500 quux
├ ─ 2.000 corge		├ 8.000 corge		├ 4.000 corge
├ nan garply				├ 15.000 garply
├ nan grault				├ 0.250 grault

```
>>> gf3 = gf1 + gf2 # add graphframes
>>> gf3 = gf1 - gf2 # subtract graphframes
>>> gf3 = gf1 * gf2 # multiply graphframes
```

Filter the GraphFrame *by node metrics in the dataframe*

```
>>> filter_func = lambda x: x["time"] > 1          # filter function
>>> filt_gf = gf.filter(filter_func, squash=True)  # apply filter and rewire graph
```

```
0.000 foo
├─ 6.000 bar
│   └─ 5.000 baz
├─ 0.000 qux
│   └─ 5.000 quux
│       ├── 10.000 corge
│       ├── 15.000 garply
│       └─ 1.000 grault
└─ 15.000 waldo
    ├── 3.000 fred
    │   └─ 5.000 plugh
    └─ 15.000 garply
```



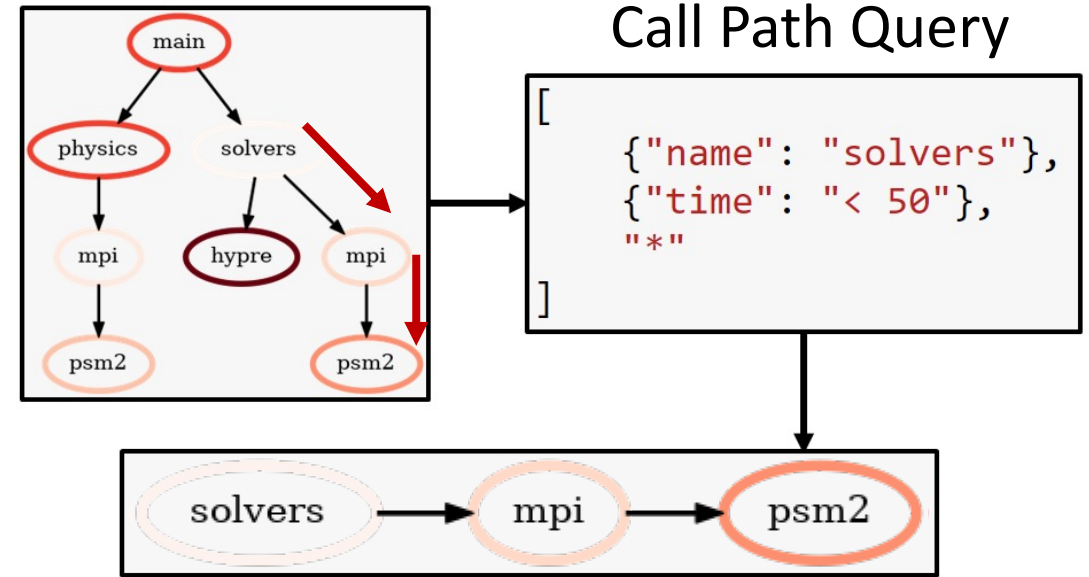
Keep only those
nodes with a value
greater than 1

```
6.000 bar
└─ 5.000 baz
5.000 quux
├─ 10.000 corge
└─ 15.000 garply
15.000 waldo
├─ 3.000 fred
│   └─ 5.000 plugh
└─ 15.000 garply
```

Filter the GraphFrame using Hatchet's call path query language

- Data reduction using *call path* pattern matching

```
# filter using call path query language
query = [
    { "name": "solvers" },
    { "time": "< 50" },
    "*"
]
filt_gf = gf.filter(query, squash=True)
```

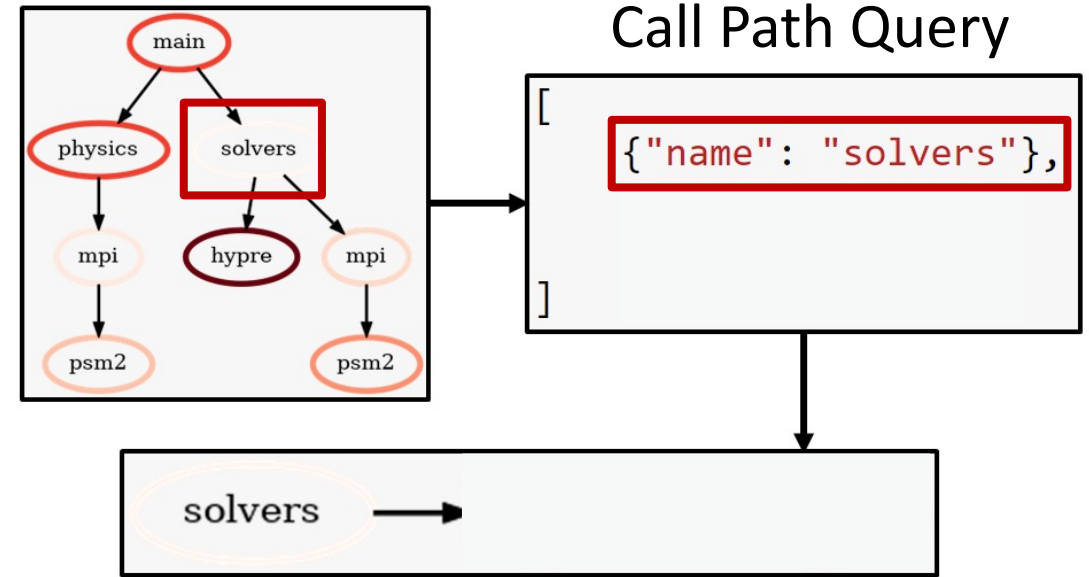


Matches a call path (1) rooted at a node with name “solvers”, (2) followed by a node with a time metric value less than 50, and (3) followed by any number of children nodes.

Filter the GraphFrame using Hatchet's call path query language

- Data reduction using *call path* pattern matching

```
# filter using call path query language
query = [
    { "name": "solvers" },
    { "time": "< 50" },
    "*"
]
filt_gf = gf.filter(query, squash=True)
```

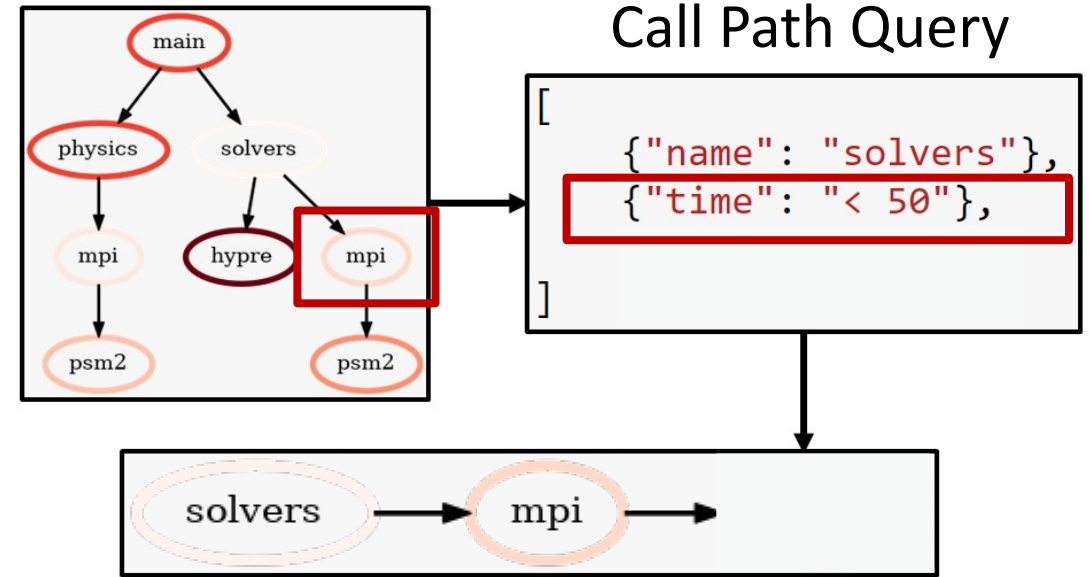


Matches a call path (1) rooted at a node with name “solvers”, (2) followed by a node with a time metric value less than 50, and (3) followed by any number of children nodes.

Filter the GraphFrame using Hatchet's call path query language

- Data reduction using *call path* pattern matching

```
# filter using call path query language
query = [
    { "name": "solvers" },
    { "time": "< 50" },
    "*"
]
filt_gf = gf.filter(query, squash=True)
```

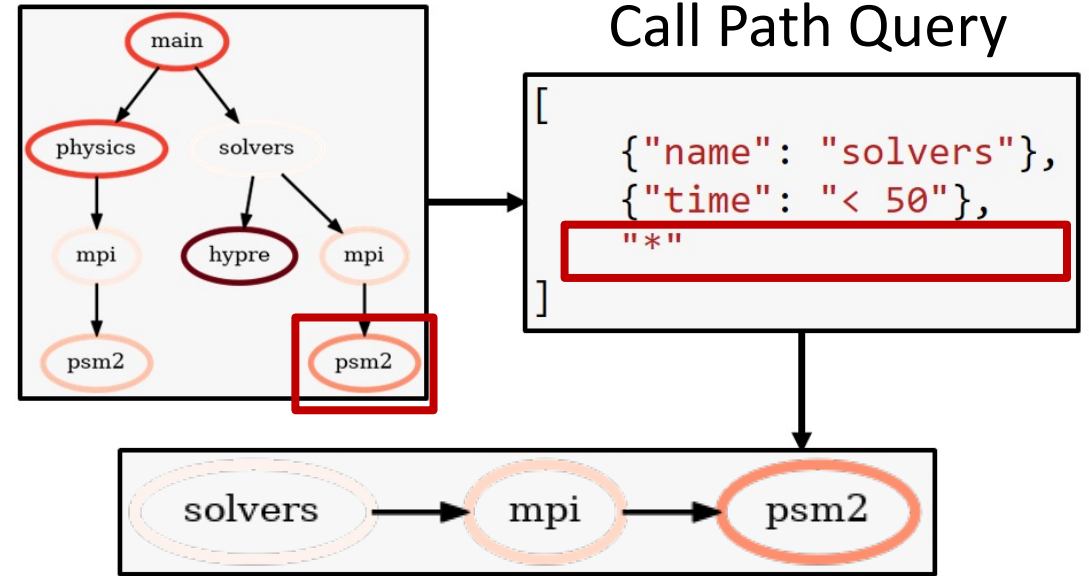


Matches a call path (1) rooted at a node with name “solvers”, (2) followed by a node with a time metric value less than 50, and (3) followed by any number of children nodes.

Filter the GraphFrame *using Hatchet's call path query language*

- Data reduction using *call path* pattern matching

```
# filter using call path query language
query = [
    { "name": "solvers" },
    { "time": "< 50" },
    "*"
]
filt_gf = gf.filter(query, squash=True)
```



Matches a call path (1) rooted at a node with name “solvers”, (2) followed by a node with a time metric value less than 50, and (3) followed by any number of children nodes.

How do I load SPOT/Caliper data into Hatchet?

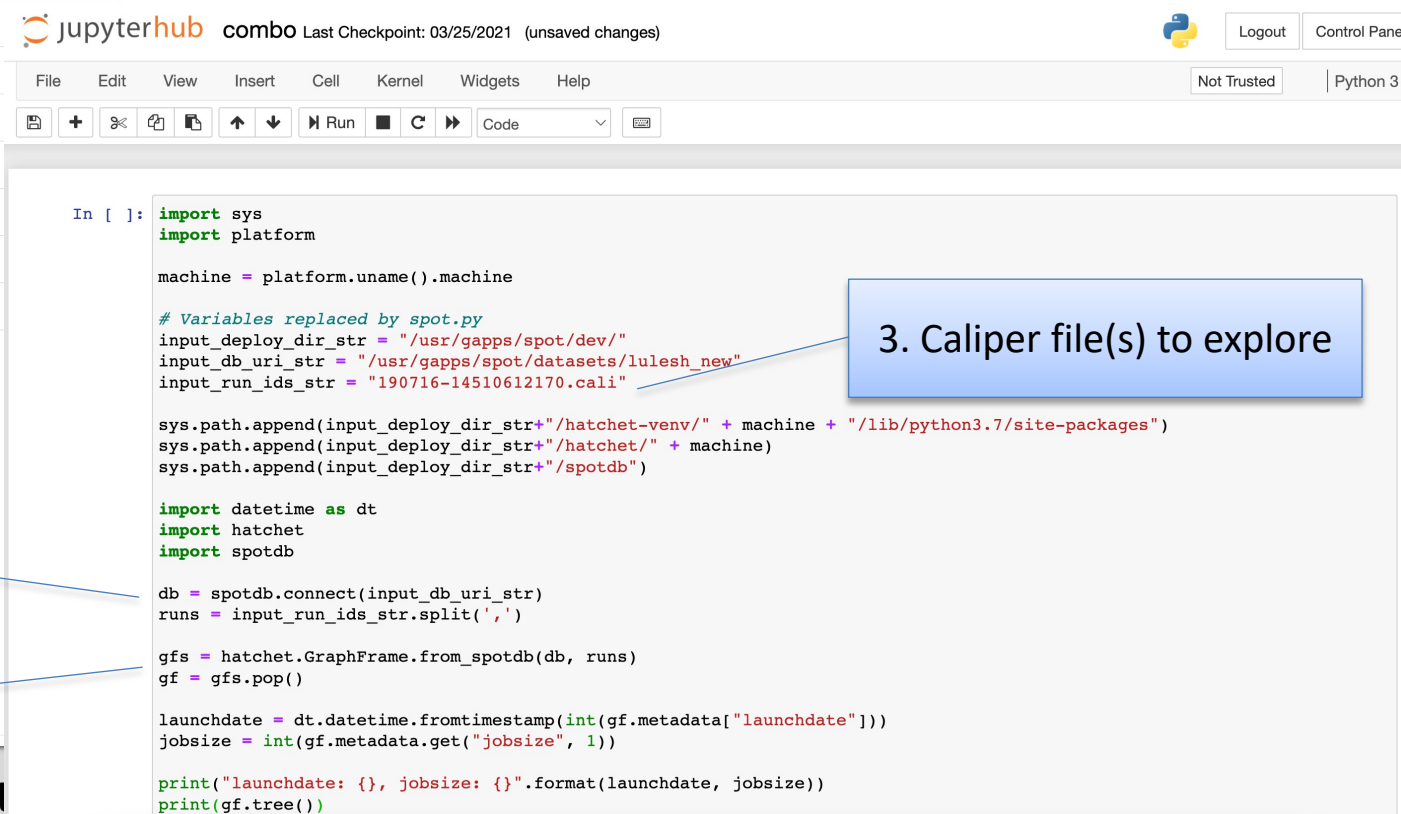
1. Directory of SPOT/Caliper files

2. Buttons bring up filled-in Jupyter notebook loading 1 or many SPOT/Caliper files

3. Caliper file(s) to explore

4. Connect to SPOT database

5. Hatchet's SPOT database reader loads into Hatchet's GraphFrame object



The screenshot shows a Jupyter notebook interface with the following components:

- Header:** "jupyterhub combo Last Checkpoint: 03/25/2021 (unsaved changes)" with "Logout" and "Control Panel" buttons.
- Menu Bar:** File, Edit, View, Insert, Cell, Kernel, Widgets, Help.
- Toolbar:** Includes icons for saving, adding, deleting, and running code.
- Code Cell:** Contains the following Python code:

```
In [ ]: import sys
import platform

machine = platform.uname().machine

# Variables replaced by spot.py
input_deploy_dir_str = "/usr/gapps/spot/dev/"
input_db_uri_str = "/usr/gapps/spot/datasets/lulesh_new"
input_run_ids_str = "190716-14510612170.cali"

sys.path.append(input_deploy_dir_str+"/hatchet-venv/" + machine + "/lib/python3.7/site-packages")
sys.path.append(input_deploy_dir_str+"/hatchet/" + machine)
sys.path.append(input_deploy_dir_str+"/spotdb")

import datetime as dt
import hatchet
import spotdb

db = spotdb.connect(input_db_uri_str)
runs = input_run_ids_str.split(',')





















gfs = hatchet.GraphFrame.from_spotdb(db, runs)
gf = gfs.pop()

launchdate = dt.datetime.fromtimestamp(int(gf.metadata["launchdate"]))
jobsize = int(gf.metadata.get("jobsize", 1))

print("launchdate: {}, jobsize: {}".format(launchdate, jobsize))
print(gf.tree())
```

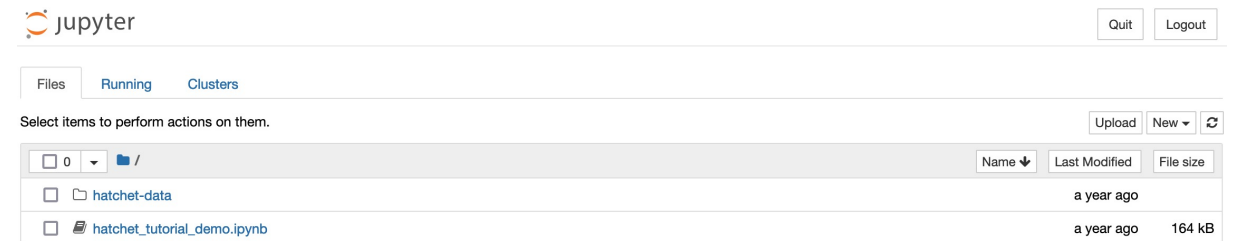
All records selected. Please click on the graph to apply filters.

TABLE COMPARE

Figure_of_merit	Problem_size	Threads	Jobsizes	launchdate	
1					
5491.526855	60	36	1	2019-Jul-16 14:03	 
12412.877512	30	4	8	2019-Jul-16 14:04	 
5525.153912	60	36	1	2019-Jul-16 14:05	 
11339.841229	30	4	8	2019-Jul-16 14:06	 
5100.923858	60	36	1	2019-Jul-16 14:07	 
11197.780437	30	4	8	2019-Jul-16 14:08	 
5071.148951	60	36	1	2019-Jul-16 14:09	 
10247.931707	30	4	8	2019-Jul-16 14:10	 
18468.799766	60	8	8	2019-Jul-16 14:06	 
18666.324557	60	8	8	2019-Jul-16 14:13	 

Hands-On Time!

- The SPOT container includes a sample Jupyter notebook, Hatchet 2022.1.0 install, and Lulesh datasets.
 - Alternatively, the sample Jupyter notebook and the Lulesh datasets are available directly at <https://github.com/llnl/hatchet-tutorial>. This repository is integrated with BinderHub, which will create a local interactive environment for you to run the notebook.
- Following this tutorial, you can substitute your own SPOT/Caliper data files into the example notebook.
- Hop over to Jupyter to run the notebook
- We'll be walking through `hatchet_tutorial_demo.ipynb`



Review: Topics covered in today's tutorial

- Single graph:

- **Load** SPOT/Caliper data file
- **Visualize** tree and dataframe
- **Filter and squash** tree

```
# Read in a SPOT/Caliper file
gf = ht.GraphFrame.from_spotdb(
    <spot-database>,
    <list-of-runs>,
)
```

```
# Print tree visualization
print(gf.tree(
    metric_column="Total time (inc)"))
```

- Subtract two trees:

- Load two SPOT/Caliper data files
- Compute **percent change** of two nightly test runs (two different times)
- **Update** existing **column** in dataframe
- **Added** new **column** to dataframe
- Visualize resulting tree

```
# Print dataframe
print(gf.dataframe)
```

- Speedup of two trees:

- Load two SPOT/Caliper data files
- **Divide** two graphs for speedup comparison
- Visualize resulting tree
- **Generate speedup plot** for interesting functions

```
# Divide two trees
gf3 = gf2 / gf1
```

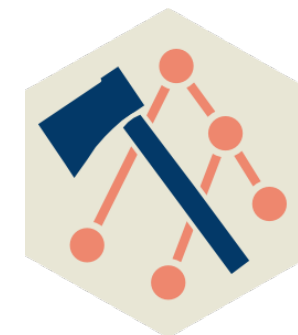
```
# Diff two trees
gf3 = (gf2 - gf1) / gf1
```

Readily available features not covered in today's tutorial

- **Add or multiply** two graphframes
- **Insert new column** to dataframe of metrics
 - Scale and offset “time” column by some factor: https://llnl-hatchet.readthedocs.io/en/latest/advanced_examples.html#applying-scalar-operations-to-attributes
 - Compute imbalance across MPI ranks within a single application execution: https://llnl-hatchet.readthedocs.io/en/latest/advanced_examples.html#applying-scalar-operations-to-attributes
- **Groupby-and-aggregate** nodes by other columns (e.g., function name, file name)
 - `res = gf.groupby_aggregate(["file"], {"time": np.sum})`
- For more details, please visit our User Guide: https://llnl-hatchet.readthedocs.io/en/latest/user_guide.html

Summary

- Hatchet is a performance analysis tool for parallel profiles
- It enables programmatic analysis of hierarchical data from one or multiple execution profiles
- Future Work:
 - Support other profile formats, add a format for outputting GraphFrames to disk
 - Implement a higher-level API for automating performance analysis
- Hatchet <https://github.com/LLNL/hatchet>
- Hatchet Tutorial <https://github.com/LLNL/hatchet-tutorial>
- Caliper <https://github.com/LLNL/Caliper>
- SPOT https://github.com/LLNL/spot2_container



Please contact us or submit GitHub issues for Hatchet questions, issues, or feature requests!



Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.