Reproducible Performance Measurement and Analysis for High-Performance Computing Applications

HPDC Tutorial

July 20, 2025

Stephanie Brink, David Boehme, Ian Lumsden,
Dewi Yokelson, Michela Taufer, and Olga Pearce

LLNL-CFPRES-2202755
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Lawrence Livermore National Laboratory

Tutorial Presenters

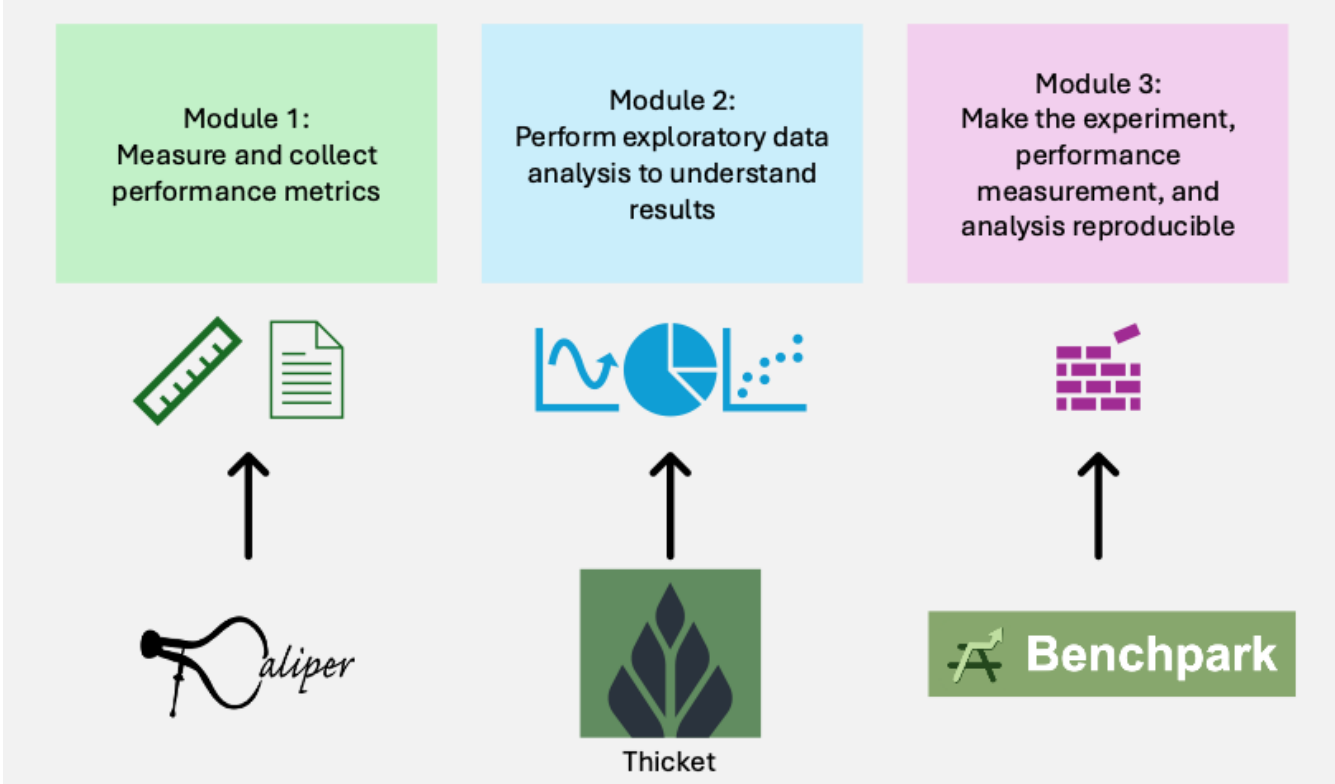
Stephanie Brink
LLNL

David Boehme
LLNL

Ian Lumsden
UTK

Overall Tutorial Agenda (approximate)

Welcome and Introduction	15 min
Module 1: Performance measurement with Caliper	45 min
Module 2: Performance analysis with Thicket	30 min
Break	30 min
Module 2 (cont'd): Performance analysis with Thicket	20 min
Module 3: Reproducible experiments with Benchpark	60 min
Q&A and Wrapup	10 min



*Hands on component to accompany each module

Overall Tutorial Materials

Find these slides and associated scripts here:

- Caliper: https://software.llnl.gov/Caliper/ under “Materials”
- Thicket: https://thicket.readthedocs.io/en/latest/tutorial_materials.html
- Benchpark: https://software.llnl.gov/benchpark/basic-tutorial.html

We also have a channel on Spack slack.

You can join here:

slack.spack.io

Join the **#benchpark-support** channel!

You can continue to ask questions here after the tutorial has ended.

Tutorial Instances: http://bit.ly/4kGQDlc

- We have an AWS instance for the hands-on component of this tutorial
- The instance provides:
  - Pre-installed Thicket, Caliper, and Benchpark and required dependencies
  - Caliper source code demos
  - Thicket Jupyter notebooks and datasets for performance analysis

When logging in to the instance:

- Please use a unique username to avoid resource allocation conflicts
  - First initial followed by last name (e.g., jdoe)
- PW: hpctutorial25

Performance Measurement with Caliper

David Boehme

Caliper: A Performance Profiling Library

2025 HPC Tutorial

July 20, 2025

David Boehme
Computer Scientist

LLNL-CFPRES-2202755
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Caliper: A Performance Profiling Library

- Integrates a performance profiler into your program
  - Profiling is always available
  - Simplifies performance profiling for application end users
- Common instrumentation interface
  - Provides program context information for other tools
- Designed for HPC
  - Supports MPI, OpenMP, CUDA, HIP, Kokkos, RAJA

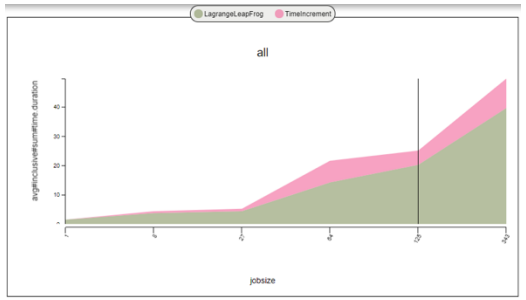
Caliper Use Cases

- Lightweight always-on profiling
  - Performance summary report for each run
- Performance debugging
- Performance introspection
- Comparison studies across runs
  - Performance regression testing
  - Configuration and scaling studies
- Automated workflows

Performance reports

```
Path       Min time/rank Max time/rank Avg time/rank Time %
main            0.000119      0.000119      0.000119  7.079120
  mainloop      0.000067      0.000067      0.000067  3.985723
    foo         0.000646      0.000646      0.000646 38.429506
  init          0.000017      0.000017      0.000017  1.011303
```

Caliper



Comparing runs

```
0.000 foo
├ 5.000 bar
│  ├ 5.000 baz
│  └ 10.000 grault
├ 0.000 qux
│  └ 5.000 quux
│     └ 10.000 corge
│        ├ 5.000 bar
│        │  ├ 5.000 baz
│        │  └ 10.000 grault
│        ├ 10.000 grault
│        └ 15.000 garply
└ 0.000 waldo
```

Debugging

The Caliper Performance Analysis Stack

Applications / libraries

```
CALI_MARK_BEGIN("foo");
CALI_MARK_END("foo");
```

Instrumentation markers

```
adiak::collect_all();
adiak::value("problem size", input.size);
```

Run metadata annotations

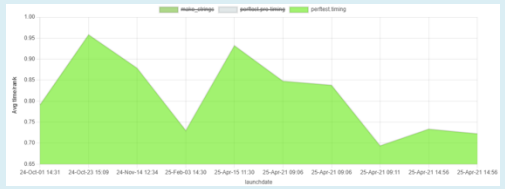
Caliper ← **Adiak**

Reports

```
Path          Time (E) Time (I) Time % (E)
main          0.000008 0.000379   1.953781
  setup       0.000366 0.000366  84.122052
  main loop   0.000003 0.000005   0.780592
    foo       0.000001 0.000001   0.267251
    bar       0.000001 0.000001   0.134545
```

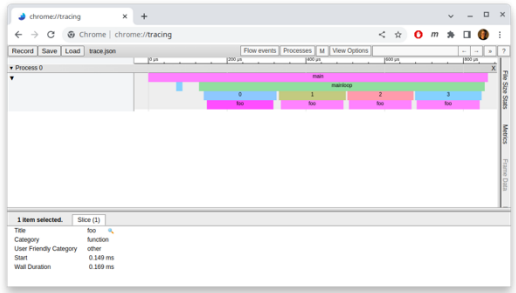
Thicket

```
592.462 main
├─ 572.913 setup
└─ 6.183 main loop
   ├─ 1.579 foo
   └─ 0.617 bar

Legend (Metric: Total time (usec) Min: 0.62 Max: 592.46 indices: {'pr
533.28 - 592.46
414.91 - 533.28
296.54 - 414.91
178.17 - 296.54
59.80 - 178.17
0.62 - 59.80

name User code   ◀ Only in left graph   ▶ Only in right graph
```



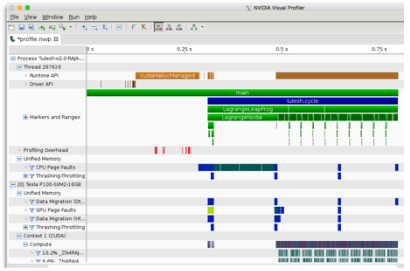
TreeScape

Python frameworks for analyzing large sets of runs



Event traces

Annotation bindings



3rd party/vendor tools

Materials, Contact & Links

- Tutorial materials: https://github.com/daboehme/caliper-tutorial

```
$ git clone --recursive https://github.com/daboehme/caliper-tutorial.git
$ . setup-env.sh
```

- GitHub repository: https://github.com/LLNL/Caliper
- Documentation: https://llnl.github.io/Caliper
- GitHub Discussions: https://github.com/LLNL/Caliper/discussions

- Contact: David Boehme (boehme3@llnl.gov)

Using Caliper

Using Caliper: Workflow

```
cali::ConfigManager mgr;
mgr.add(opts.caliperConfig.c_str());
mgr.start();
// ...
mgr.flush();
```

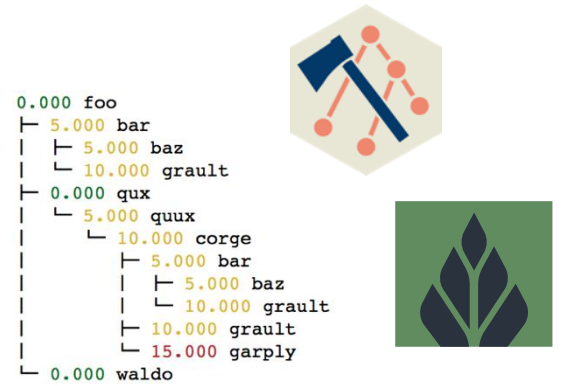
ConfigManager API

```
Path        Min time/rank Max time/rank
main            0.000119      0.000119
  mainloop      0.000067      0.000067
    foo         0.000646      0.000646
  init          0.000017      0.000017
```

Human-readable reports

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

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

Source-code annotation API




```
$ CALI_CONFIG=runtime-report ./app
```

Environment variables

Profile/trace data processing
(e.g., Hatchet or Thicket)

Code Instrumentation

Runtime Configuration

Data Analysis

Region Profiling: Marking Code Regions

C/C++

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

void main() {
  CALI_MARK_BEGIN(”init”);
  do_init();
  CALI_MARK_END(”init”);

  CALI_MARK_PHASE_BEGIN(”hydro”);
  do_hydro();
  CALI_MARK_PHASE_END(”hydro”);
}
```

Fortran

```
USE caliper_mod


CALL cali_begin_region(‘init’)

CALL do_init()

CALL cali_end_region(‘init’)
```

- Use annotation macros (C/C++) or functions to mark and name code regions

Region Profiling: Best Practices

- Be selective: Instrument high-level program subdivisions (kernels, phases, ...)
- Be clear: Choose meaningful names
- Start small: Add instrumentation incrementally

```
RAJA::ReduceSum<RAJA::omp_reduce, double> ompdot(0.0);

CALI_MARK_BEGIN(”dotproduct”);

RAJA::forall<RAJA::omp_parallel_for_exec>(RAJA::RangeSegment(0, N), [=] (int i) {
  ompdot += a[i] * b[i];
});
dot = ompdot.get();

CALI_MARK_END(”dotproduct”);
```

Caliper annotations give meaningful names to high-level program constructs

Instrumentation Support for RAJA and Kokkos

```
RAJA::dynamic_forall<policy_list>(pol, range,
    RAJA::expt::Reduce<RAJA::operators::plus>(&sum),
      RAJA::Name("RAJA dynamic forall reduction"),
      [=] RAJA_HOST_DEVICE (int i, VAL_INT_SUM &_sum) {

      c[i] = a[i] + b[i];
      _sum += 1;
  });
```

- RAJA now supports Caliper via optional RAJA::Name() parameter to label a kernel
- Caliper can also be used as a Kokkos profiling tool

Region Profiling: Printing a Runtime Report

```
$ cd Caliper/build
$ make cxx-example
$ CALI_CONFIG=runtime-report ./examples/apps/cxx-example
```

```
Path        Min time/rank Max time/rank Avg time/rank Time %
main            0.000119      0.000119      0.000119  7.079120
  mainloop      0.000067      0.000067      0.000067  3.985723
    foo         0.000646      0.000646      0.000646 38.429506
  init          0.000017      0.000017      0.000017  1.011303
```

- Set the CALI_CONFIG environment variable to access Caliper's built-in profiling configurations
- "runtime-report" measures, aggregates, and prints time in annotated code regions

List of Caliper's Built-in Profiling Recipes

Config name	Description
runtime-report	Print a time profile for annotated regions
loop-report	Print summary and time-series information for loops
mpi-report	Print time spent in MPI functions
sample-report	Print time spent in regions using call-path sampling
event-trace	Record a trace of region enter/exit events in .cali format
hatchet-region-profile	Record a region time profile for processing with hatchet or cali-query
hatchet-sample-profile	Record a sampling profile for processing with hatchet or cali-query
spot	Record a time profile for Thicket or TreeScape

Use `cali-query --help=configs` to list all built-in configs and their options

Built-In Profiling Recipes: Configuration String Syntax

Config name specifies the kind of performance measurement

Parameters enable additional features, metrics, or output options

```
$ CALI_CONFIG="runtime-report(mem.highwatermark,output=stdout)" ./examples/apps/cxx-example
```

```
Path        Min time/rank Max time/rank Avg time/rank   Time % Allocated MB
main            0.000179      0.000179      0.000179 2.054637     0.000047
  mainloop      0.000082      0.000082      0.000082 0.941230     0.000016
    foo         0.000778      0.000778      0.000778 8.930211     0.000016
  init          0.000020      0.000020      0.000020 0.229568     0.000000
```

- Most Caliper measurement recipes have optional parameters to enable additional features or configure output settings

Profiling Options: MPI Function Profiling

```
$ CALI_CONFIG=runtime-report,profile.mpi ./lulesh2.0
```

```
Path                                  Min time/rank  Max time/rank  Avg time/rank  Time %
MPI_Comm_dup                           0.000034       0.003876       0.001999   0.10089
main                                   0.009013       0.010797       0.010173   0.51335
  MPI_Reduce                           0.000031       0.000049       0.000037   0.001886
  lulesh.cycle                         0.002031       0.002258       0.002085   0.105220
    LagrangeLeapFrog                   0.002158       0.002511       0.002227   0.112366
      CalcTimeConstraintsForElems      0.015166       0.015443       0.015277   0.770922
      CalcQForElems                    0.058781       0.060196       0.059699   3.01254
          CalcMonotonicQForElems       0.035331       0.041057       0.038496   1.942601
          CommMonoQ                    0.005280       0.006152       0.005544   0.279781
            MPI_Wait                   0.004182       0.084533       0.035324   1.78249
          CommSend                     0.006893       0.009062       0.008071   0.407298
            MPI_Waitall                0.000986       0.001778       0.001343   0.067789
            MPI_Isend                  0.004564       0.005785       0.004930   0.248765
          CommRecv                     0.002265       0.002616       0.002341   0.118144
[...]
```

The profile.mpi option measures time spent in MPI functions

Profiling Options: CUDA Profiling

```
$ lrun -n 4 ./tea_leaf runtime-report,profile.cuda
```

```
Path                          Min time/rank  Max time/rank  Avg time/rank  Time %
timestep_loop                      0.000175       0.000791       0.000345   0.002076
[...]
  total_solve                      0.000105       0.000689       0.000252   0.001516
    solve                          0.583837       0.617376       0.594771   3.581811
      dot_product                  0.000936       0.001015       0.000969   0.005837
      cudaMalloc                   0.000060       0.000066       0.000063   0.000382
      internal_halo_update         0.077627       0.079476       0.078697   0.473925
      halo_update                  0.158597       0.161853       0.160023   0.963685
      halo_exchange                1.502106       1.572522       1.532860   9.231136
        cudaMemcpy                11.840890      11.871018      11.860343  71.424929
        cudaLaunchKernel           1.177454       1.230816       1.211668   7.296865
      cudaMemcpy                   0.470123       0.471485       0.470596   2.834008
      cudaLaunchKernel             0.658269       0.682566       0.673030   4.053100
[...]
```

The `profile.cuda` option measures time in CUDA runtime API calls

Use cali-query to get Help for Profiling Recipes

```
$ cali-query --help runtime-report
```

```
runtime-report
 Print a time profile for annotated regions
  Options:
   aggregate_across_ranks    Aggregate results across MPI ranks
   calc.inclusive            Report inclusive instead of exclusive times
   cuda.gputime              Report GPU time in CUDA activities
   exclude_regions           Do not take snapshots for the given region names/patterns.
   include_branches          Only take snapshots for branches with the given region names.
   include_regions           Only take snapshots for the given region names/patterns.
   io.bytes                  Report I/O bytes written and read
   io.bytes.read             Report I/O bytes read
   io.bytes.written          Report I/O bytes written
   io.read.bandwidth         Report I/O read bandwidth
   io.write.bandwidth        Report I/O write bandwidth
   level                     Minimum region level that triggers snapshots
   main_thread_only          Only include measurements from the main thread in results.
   max_column_width          Maximum column width in the tree display
...
```

Sample Profiling

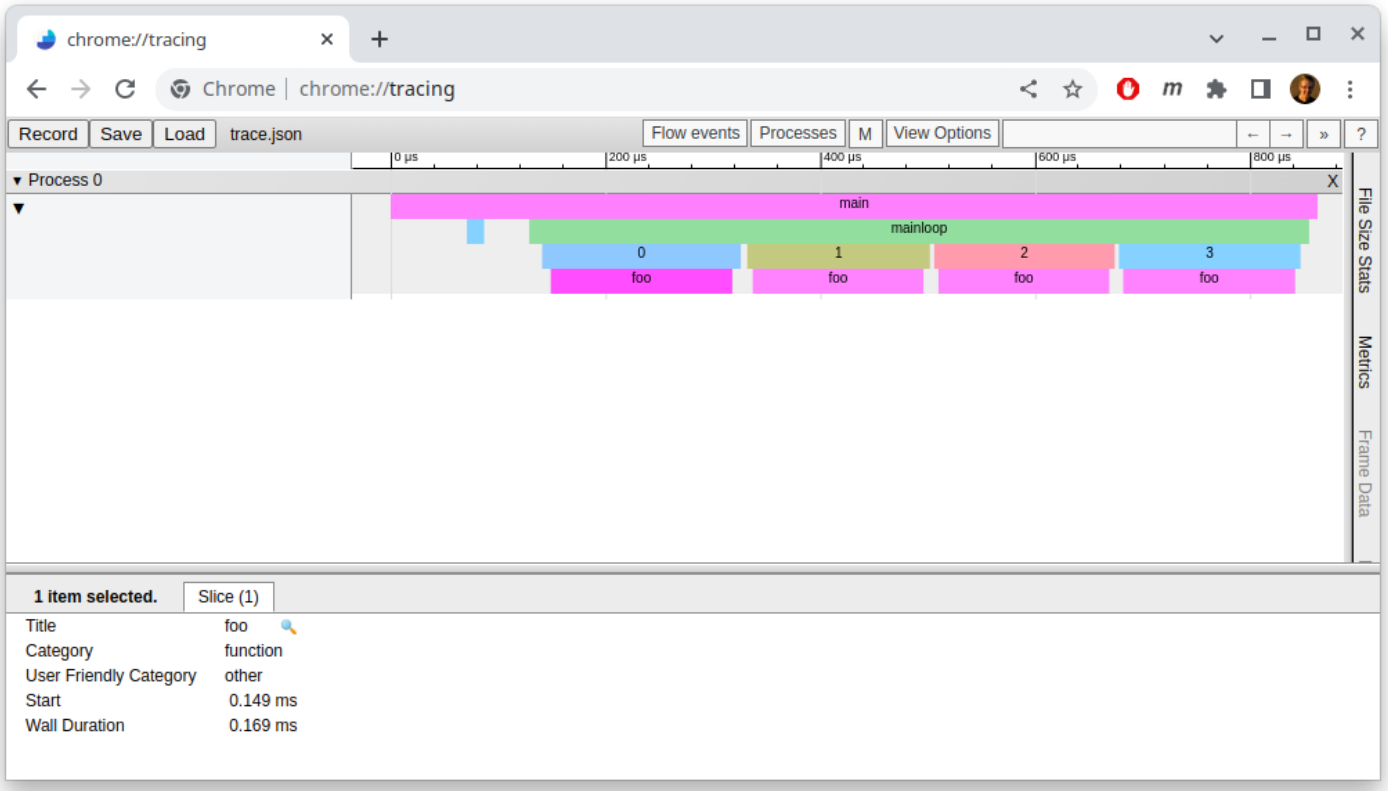
```
$ CALI_CONFIG=sample-report ./lulesh2.0
```

```
Path   Min time/rank Max time/rank Avg time/rank Total time   Time % Function
main
 |-       0.005000    0.005000    0.005000   0.035000 0.059691 Domain::AllocateElemPersistent
 |-       0.005000    0.005000    0.005000   0.035000 0.059691 Domain::SetupThreadSupportStru
 |-       0.005000    0.005000    0.005000   0.005000 0.008527 sysmalloc
 |-       0.005000    0.005000    0.005000   0.005000 0.008527 Domain::BuildMesh(int, int, in
lulesh.cycle
  TimeIncrement
   |-      0.075000    0.740000    0.355000   2.840000 4.843523 gomp_barrier_wait_end
   |-      0.005000    0.060000    0.027857   0.195000 0.332566 psm2_mq_ipeek2
   |-      0.005000    0.005000    0.005000   0.015000 0.025582 psm_no_lock
   |-      0.005000    0.060000    0.023571   0.165000 0.281402 psm_progress_wait
   |-      0.015000    0.030000    0.022143   0.155000 0.264347 mv2_shm_bcast
   |-      0.005000    0.025000    0.013750   0.055000 0.093801 amsh_poll
   |-      0.005000    0.010000    0.007500   0.030000 0.051164 psmi_poll_internal
...
```

The *sample-report* recipe samples source functions or file+line locations.

Event Tracing and Timeline Visualization

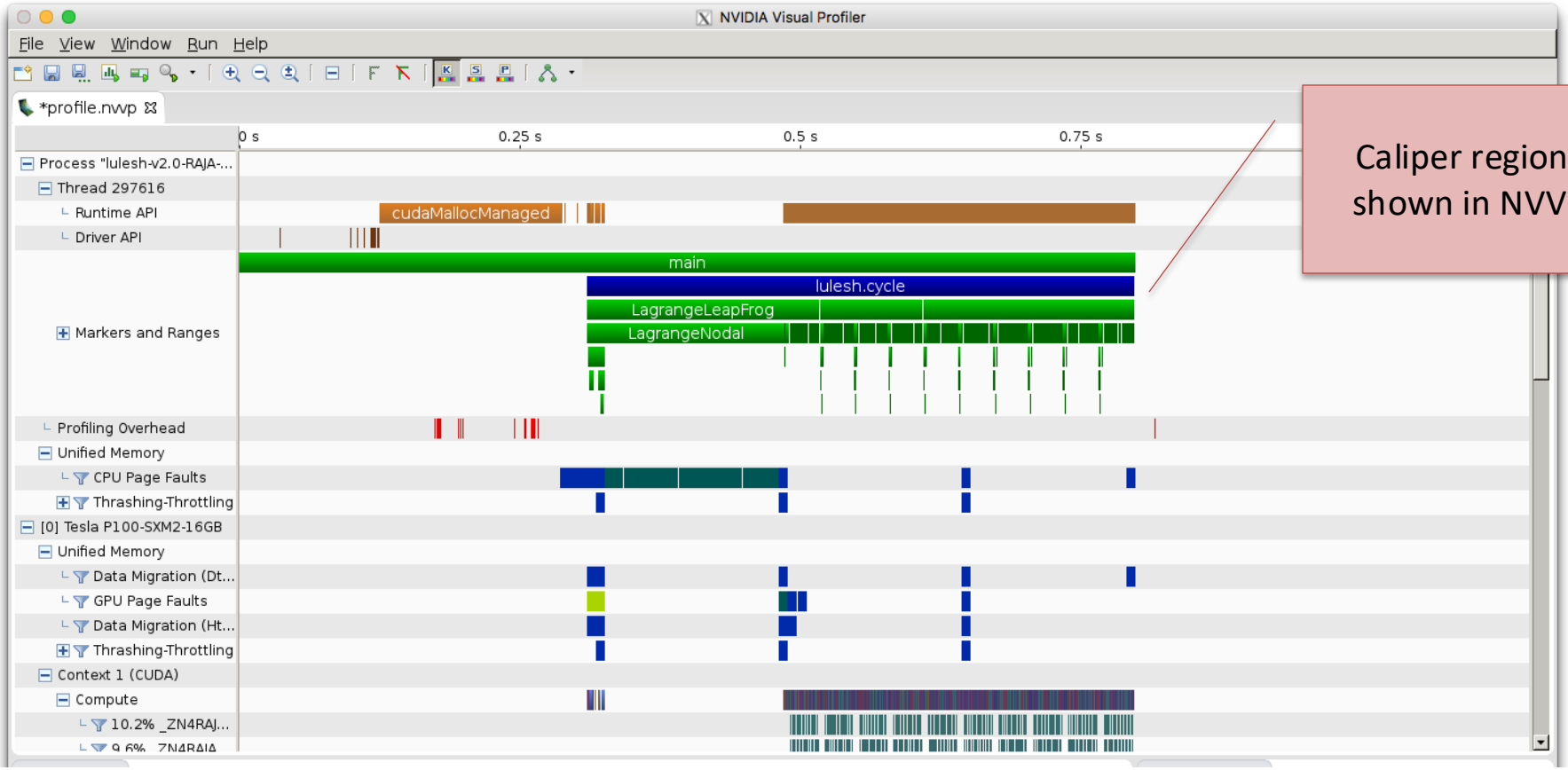
```
$ CALI_CONFIG=event-trace,output=trace.cali ./lulesh2.0
$ cali2traceevent trace.cali trace.json
```



Caliper event traces can be visualized in Chrome trace browser or Perfetto

Forwarding Annotations to Third-Party Tools

```
$ CALI_CONFIG=nvtx nvprof <nvprof-opts> ./app
```



Caliper regions shown in NVVP

The nvtx config forwards annotations to NVidia's NVTX API

Generating Reports with cali-query

```
$ mpirun -n 8 lulesh2.0 -P hatchet-region-profile,profile.mpi
$ cali-query -q “select mpi.rank,sum#sum#time.duration where mpi.function=MPI_Allreduce format tree”
    region_profile.cali
```

```
Path                mpi.rank time
main
  lulesh.cycle
    TimeIncrement
      MPI_Allreduce
        |-                 0 0.002174
        |-                 1 0.083150
        |-                 2 0.100689
        |-                 3 0.171060
        |-                 4 0.171347
        |-                 5 0.204563
        |-                 6 0.161819
        |-                 7 0.203488
```

- Run SQL-like queries on Caliper output with the cali-query tool

Control Profiling Programmatically: The ConfigManager API

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

int main(int argc, char* argv[])
{
  cali::ConfigManager mgr;
  mgr.add(argv[1]);
  if (mgr.error())
    std::cerr << mgr.error_msg() << "\n";

  mgr.start();
  // ...
  mgr.flush();
}
```

- Use ConfigManager to access Caliper's built-in profiling configurations

```
$ ./app runtime-report
```

- Now we can use command-line arguments or other program inputs to enable profiling

Manual Configuration Allows Custom Analyses

```
cali-query -q "select alloc.label#cupti.fault.addr as Pool,
  cupti.uvm.kind as UVM\ Event,
  scale(cupti.uvm.bytes,1e-6) as MB,
  scale(cupti.activity.duration,1e-9) as Time
group by
  prop:nested,alloc.label#cupti.fault.addr,cupti.uvm.kind
where cupti.uvm.kind format tree" trace.cali
```

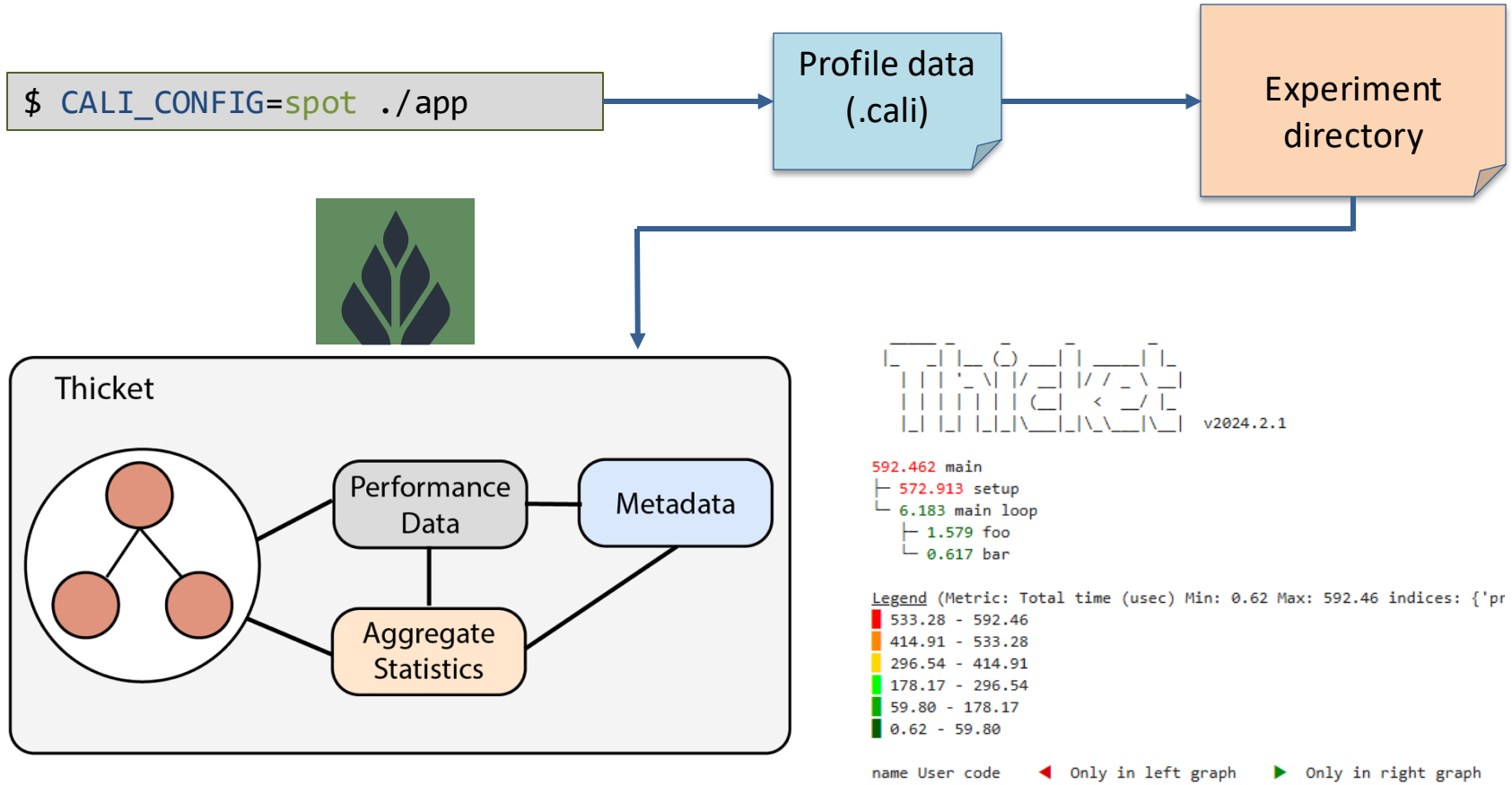
caliper.config

```
CALI_SERVICES_ENABLE=alloc,cupti,cuptitrace,mpi,trace,recorder
CALI_ALLOC_RESOLVE_ADDRESSES=true
CALI_CUPTI_CALLBACK_DOMAINS=sync
CALI_CUPTITRACE_ACTIVITIES=uvm
CALI_CUPTITRACE_CORRELATE_CONTEXT=false
CALI_CUPTITRACE_FLUSH_ON_SNAPSHOT=true
```

```
Path
main
  solve
    TIME_STEPPING
      enforceBC
        CURVI in EnforceBC
          CurviCartIC
            CurviCartIC::PART 3  Pool          UVM Event      MB          Time
              curvilinear4sgwind UM_pool       pagefaults.gpu             2.806946
              curvilinear4sgwind UM_pool       HtoD           7862.747136 0.232238
              curvilinear4sgwind UM_pool_temps pagefaults.gpu             0.130167
              curvilinear4sgwind UM_pool       DtoH           9986.441216 0.378583
              curvilinear4sgwind UM_pool       pagefaults.cpu
```

- Mapping CPU/GPU unified memory transfer events to Umpire memory pools in SW4

Performance Comparison Studies with Thicket



Thicket: Exploratory Data Analysis in Python for large sets of runs

Recording Program Metadata with the Adiak Library

TeaLeaf_CUDA example [C++]

```
#include <adiak.hpp>

adiak::user();
adiak::launchdate();
adiak::jobsize();

adiak::value("end_step", readInt(input, "end_step"));
adiak::value("halo_depth", readInt(input, "halo_depth"));

if (tl_use_ppcg) {
  adiak::value("solver", "PPCG");
// [...]
```

Use built-in Adiak functions to collect common metadata

Use key:value functions to collect program-specific data

- Use the Adiak C/C++ library to record program metadata
  - Environment info (user, launchdate, system name, ...)
  - Program configuration (input problem description, problem size, ...)
- Enables performance comparisons across runs. Required for SPOT and Thicket.

Adiak: Built-in Functions for Common Metadata

```
adiak_user();               /* user name */
adiak_uid();                /* user id */
adiak_launchdate();         /* program start time (UNIX timestamp) */
adiak_executable();         /* executable name */
adiak_executablepath();     /* full executable file path */
adiak_cmdline();            /* command line parameters */
adiak_hostname();           /* current host name */
adiak_clustername();        /* cluster name */

adiak_job_size();           /* MPI job size */
adiak_hostlist();           /* all host names in this MPI job */

adiak_walltime();           /* wall-clock job runtime */
adiak_cputime();            /* job cpu runtime */
adiak_systime();            /* job sys runtime */

adiak_collect_all();        /* everything */
```

- Adiak comes with built-in functions to collect common environment metadata

Adiak: Recording Custom Key-Value Data in C++

C++

```
#include <adiak.hpp>

vector<int> ints { 1, 2, 3, 4 };
adiak::value("myvec", ints);

adiak::value("myint", 42);
adiak::value("mydouble", 3.14);
adiak::value("mystring", "hi");

adiak::value("mypath", adiak::path("/dev/null"));
adiak::value("compiler", adiak::version("gcc@8.3.0"));
```

- Adiak supports many basic and structured data types
  - Strings, integers, floating point, lists, tuples, sets, …
- `adiak::value()` records key:value pairs with overloads for many data types

Adiak: Recording Custom Key-Value Data in C

C

```c
#include <adiak.h>

int ints[] = { 1, 2, 3, 4 };
adiak_namevalue("myvec",    adiak_general, NULL, "[%d]", ints, 4);

adiak_namevalue("myint",    adiak_general, NULL, "%d", 42);
adiak_namevalue("mydouble", adiak_general, NULL, "%f", 3.14);
adiak_namevalue("mystring", adiak_general, NULL, "%s", "hi");

adiak_namevalue("mypath",   adiak_general, NULL, "%p", "/dev/null");
adiak_namevalue("compiler", adiak_general, NULL, "%v", "gcc@8.3.0");
```

- In C, `adiak_namevalue()` uses printf()-style descriptors to determine data types

Live Demo

Tutorial Instances: http://bit.ly/4kGQDlc

- We have an AWS instance for the hands-on component of this tutorial
- The instance provides:
  - Pre-installed Thicket, Caliper, and Benchpark and required dependencies
  - Jupyter notebooks
  - Datasets for performance analysis

When logging in to the instance:

- Please use a unique username to avoid resource allocation conflicts
  - First initial followed by last name (e.g., jdoe)
- PW: hpctutorial2025

CASC

Center for Applied
Scientific Computing

Performance Analysis with Thicket I

Stephanie Brink

Performance Analysis using Thicket

HPDC Tutorial

20 July 2025

Stephanie Brink

LLNL-CFPRES-2202755

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

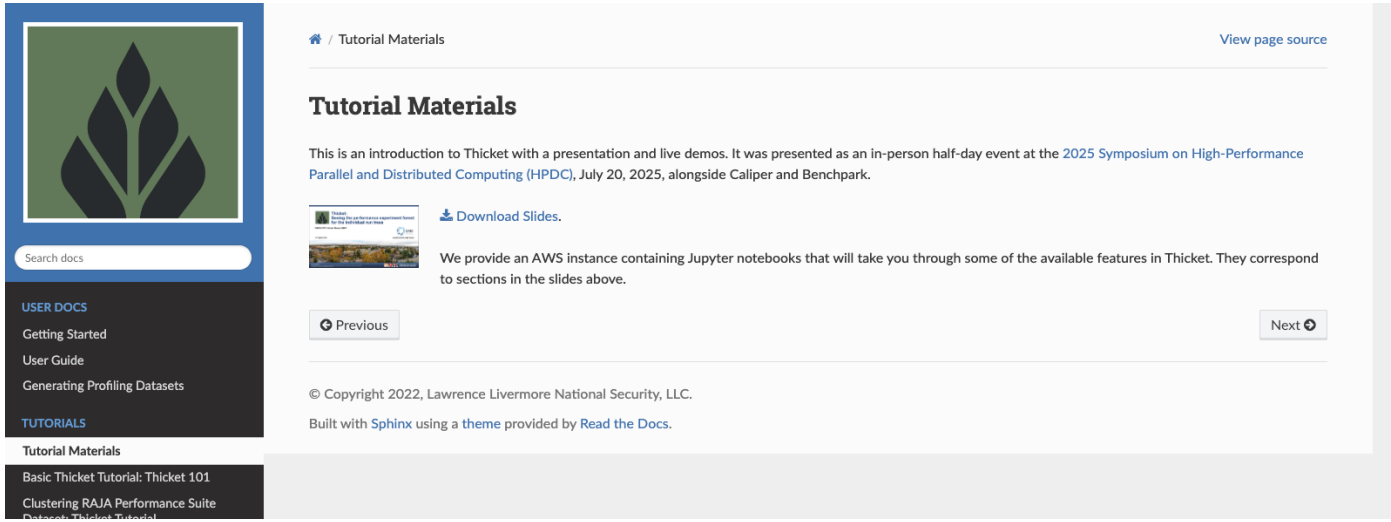
Lawrence Livermore National Laboratory

Tutorial Getting Started

Presentation slides: https://thicket.readthedocs.io/en/latest/tutorial_materials.html

Tutorial Agenda

- Welcome and overview
- Presentation (with slides)
- Hands-on session (AWS instance)

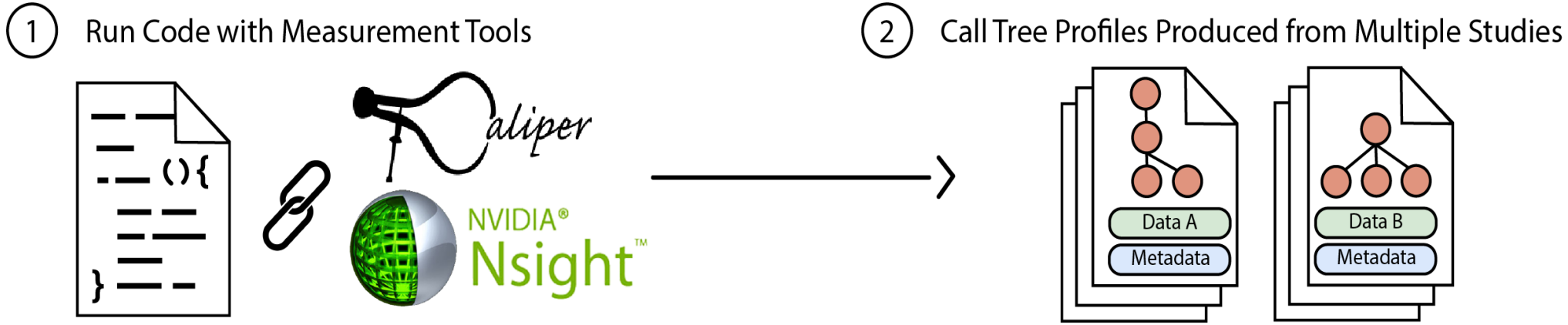


AWS instance included in the tutorial materials provides:

- Pre-installed Thicket and required dependencies
- Jupyter notebooks and associated datasets for analysis

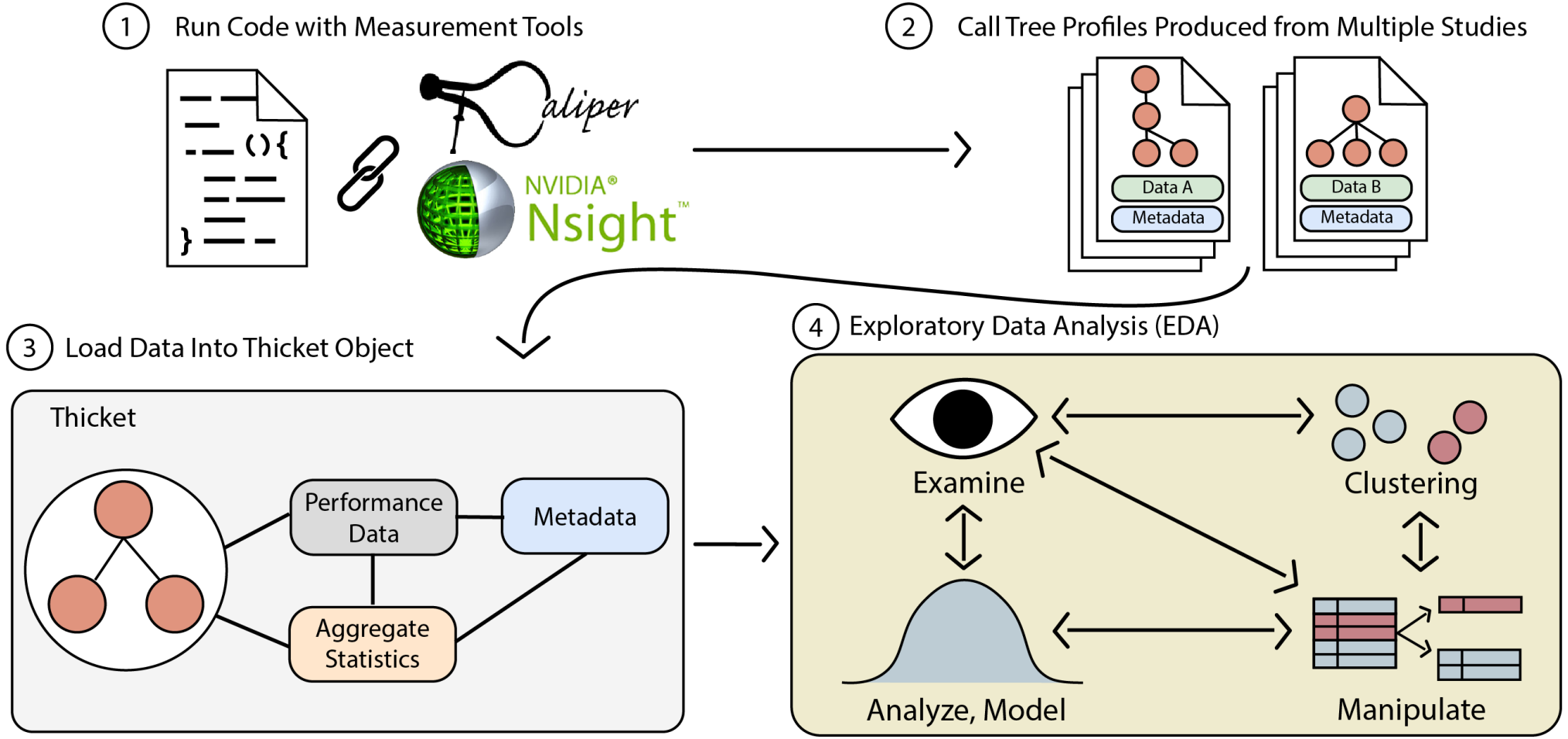
Challenge: Performance analysis in complex HPC ecosystem

- HPC software and hardware are increasingly complex. Need to understand:
  - Strong scaling and weak scaling of applications
  - Impact of application parameters on performance
  - Impact of choice of compilers and optimization levels
  - Performance on different hardware architectures (e.g., CPUs, GPUs)
  - Different tools to measure different aspects of application performance



Goal: Analyze and visualize performance data from different sources and types

Our big picture solution for analyzing and visualizing performance data from different sources and type

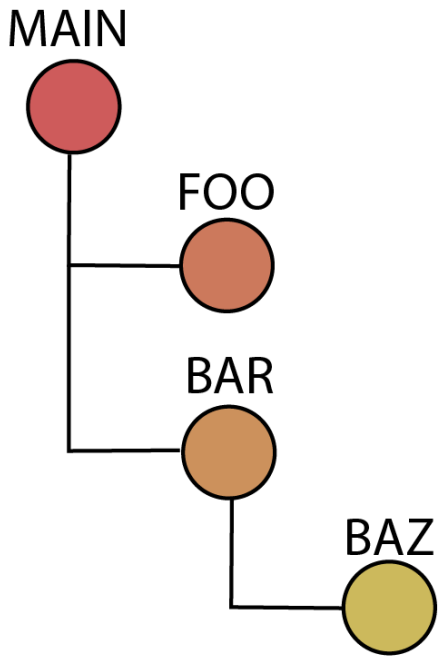


https://github.com/LLNL/thicket
https://github.com/LLNL/thicket-tutorial

What do profiling tools collect per run?

e.g., Caliper

1) Call Tree



2) Performance data

Node	Cache Misses
MAIN	
FOO	
BAR	
BAZ	

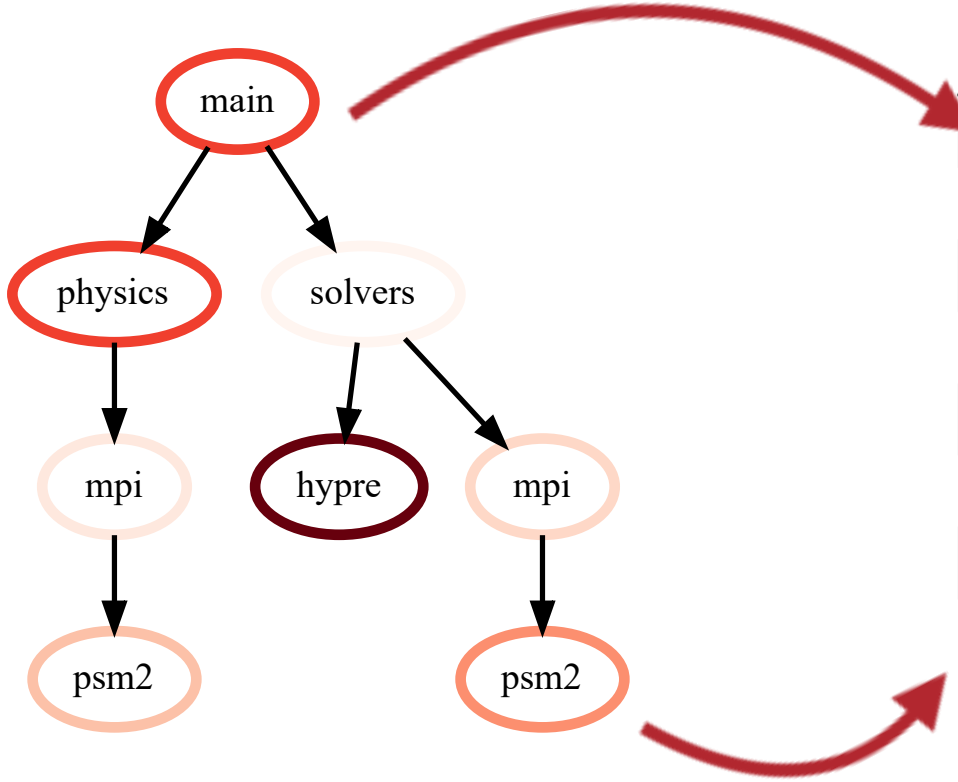
- Time, FLOPS
- Cache misses
- Memory accesses

3) Metadata per run

User	Platform

- Batch submission (user, launch date)
- Hardware info (platform)
- Build info (compiler versions/flags)
- Runtime info (problem parameters, number of MPI ranks used)

Thicket builds upon Hatchet's *GraphFrame*: a Graph and a Dataframe



node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
mpi	mpi	2	mpi	5.0	20.0
psm2	psm2	3	psm2	15.0	15.0
solvers	solvers	4	solvers	0.0	100.0
hypre	hypre	5	hypre	65.0	65.0
mpi	mpi	6	mpi	10.0	35.0
psm2	psm2	7	psm2	25.0	25.0

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)

https://github.com/LLNL/hatchet
https://github.com/LLNL/hatchet-tutorial

Visualizing Hatchet's GraphFrame components

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

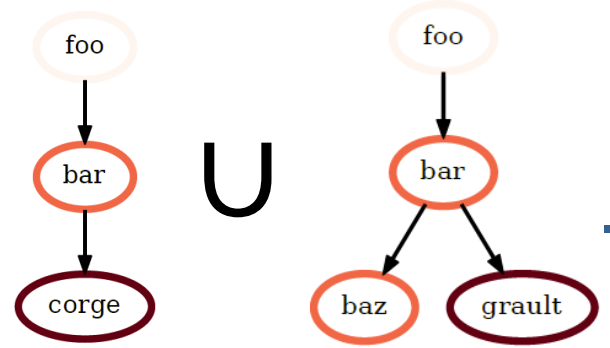
```
0.000 foo
├─ 6.000 bar
│  └─ 5.000 baz
├─ 0.000 qux
│  └─ 5.000 quux
│     ├─ 10.000 corge
│     ├─ 15.000 garply
│     └─ 1.000 grault
└─ 15.000 waldo
   ├─ 3.000 fred
   │  └─ 5.000 plugh
   └─ 15.000 garply
```

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

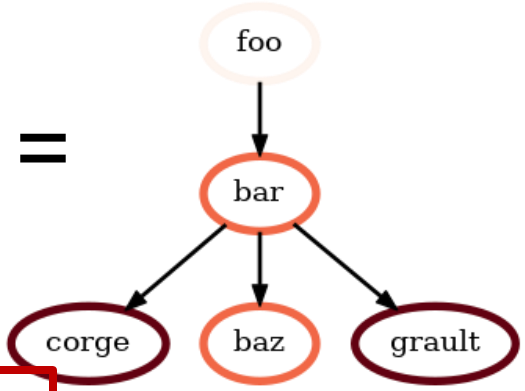
node	name	time	time (inc)
{'name': 'foo'}	foo	0.0	130.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name': 'grault'}	grault	10.0	10.0
{'name': 'qux'}	qux	0.0	60.0
{'name': 'quux'}	quux	5.0	60.0
{'name': 'corge'}	corge	10.0	55.0
{'name': 'bar'}	bar	5.0	20.0
{'name': 'baz'}	baz	5.0	5.0
{'name':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

```
0.000 foo
├─ 2.000 bar
│  └─ 5.000 baz
└─ inf qux
   └─ 4.000 quux
      ├─ 2.000 corge
      ├─ nan garply
      └─ nan grault
```

=

gf1

```
0.000 foo
├─ 6.000 bar
│  └─ 5.000 baz
└─ 3.000 qux
   └─ 2.000 quux
      └─ 8.000 corge
```

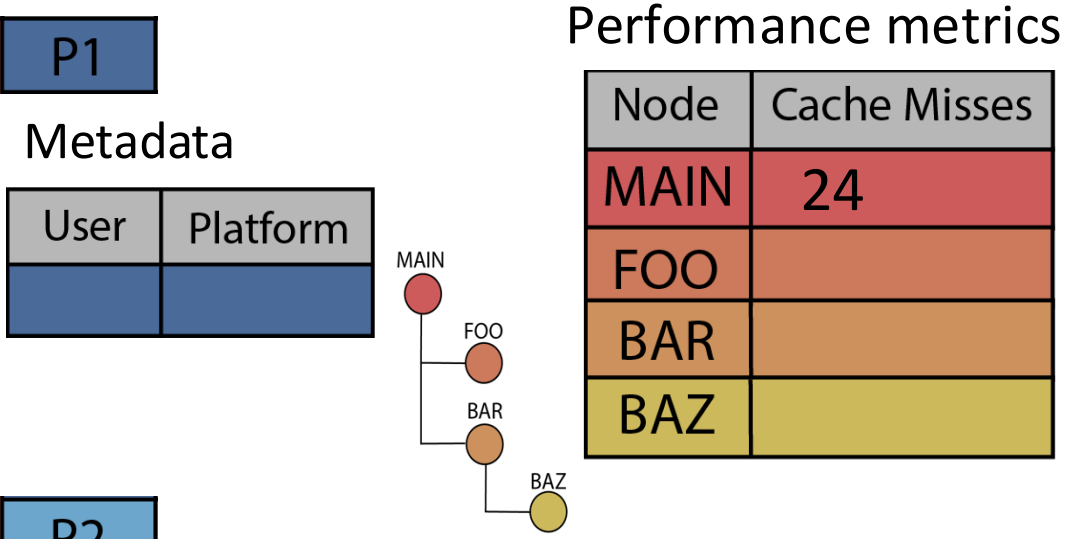
/

gf2

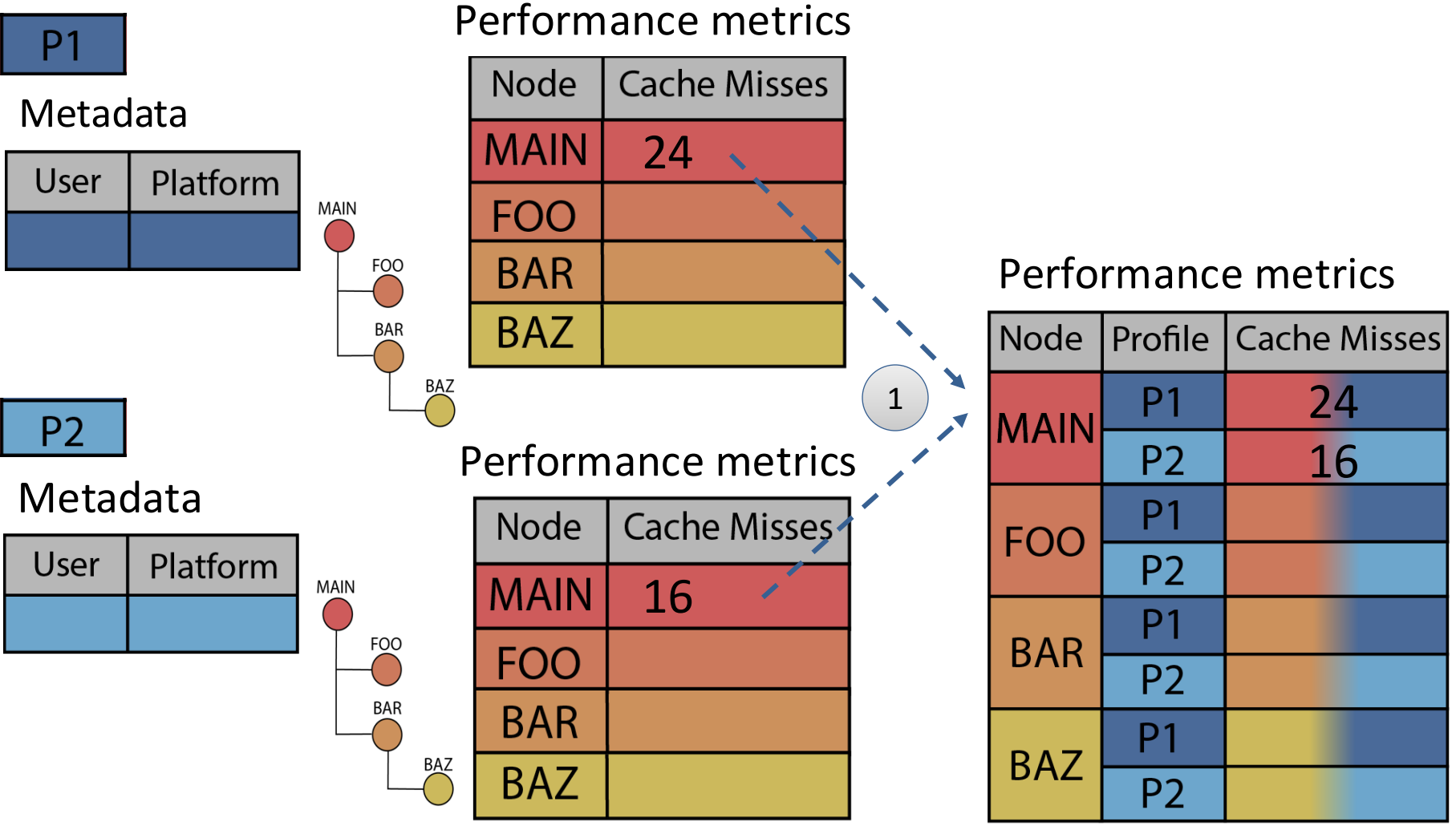
```
0.000 foo
├─ 3.000 bar
│  └─ 1.000 baz
└─ 0.000 qux
   └─ 0.500 quux
      ├─ 4.000 corge
      ├─ 15.000 garply
      └─ 0.250 grault
```

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

Use Thicket to *compose* performance profiles in Python

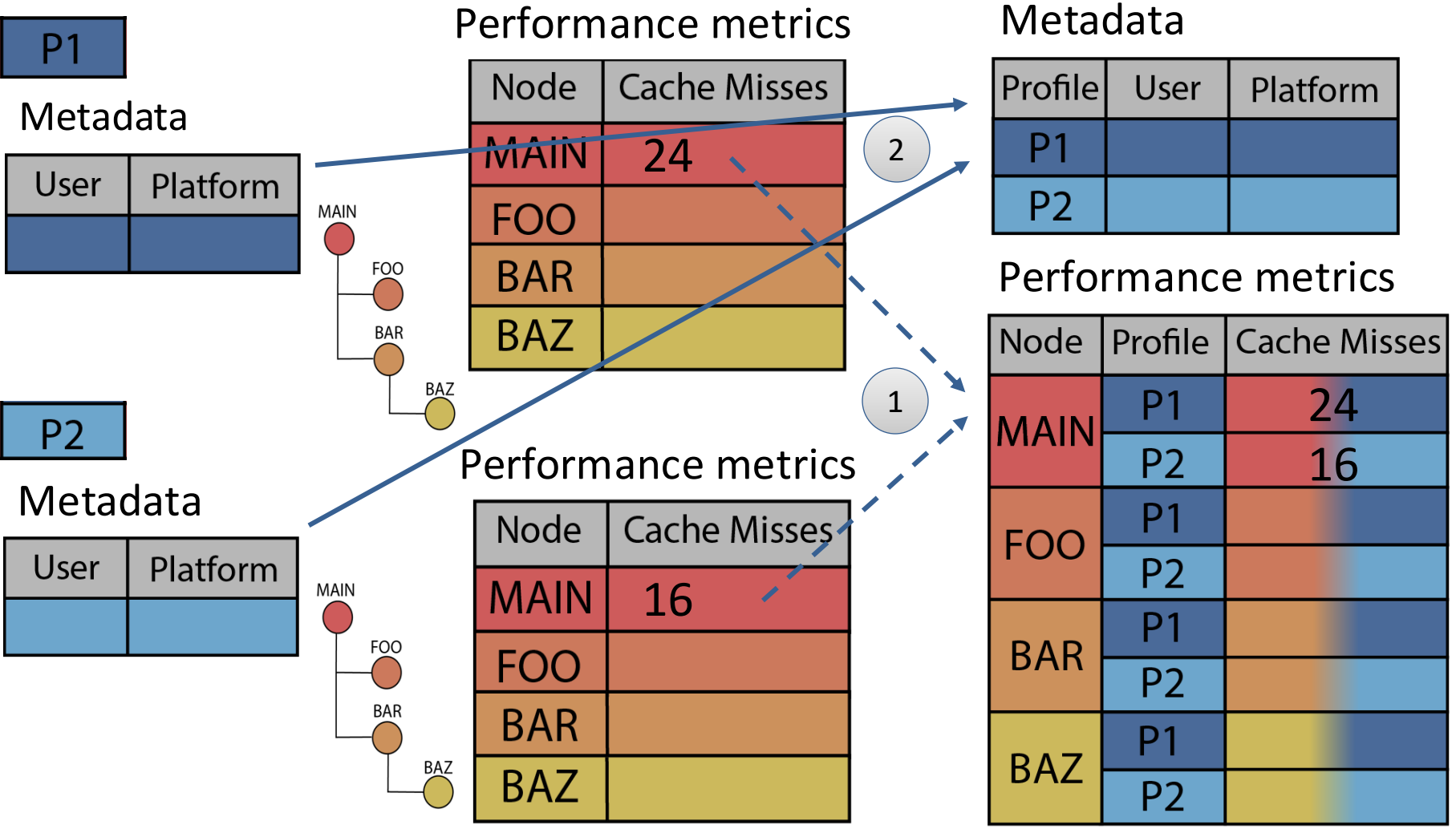


Use Thicket to *compose* performance profiles in Python



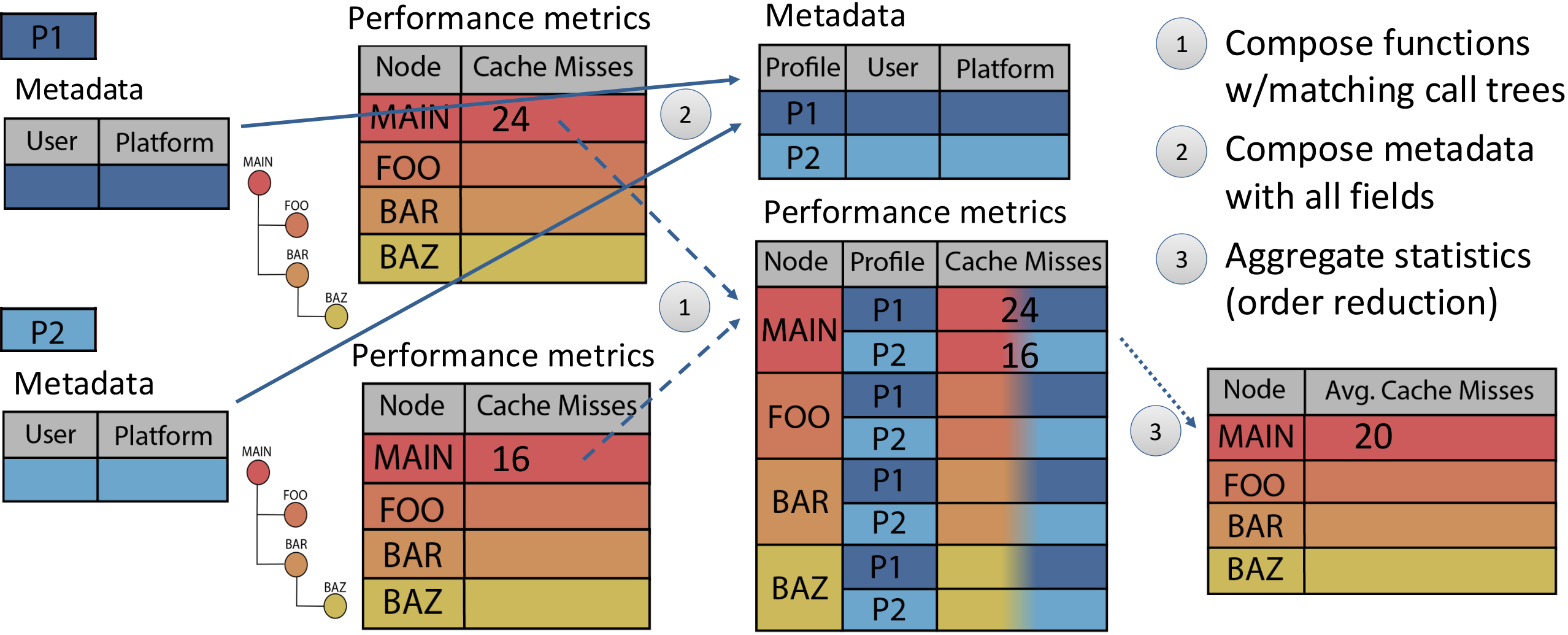
1 Compose functions w/matching call trees

Use Thicket to *compose* performance profiles in Python



1. Compose functions w/matching call trees
2. Compose metadata with all fields

Use Thicket to *compose* performance profiles in Python



Thicket components are *interconnected*

Metadata

Profile	User	Platform
P1	Jon	lassen
P2	Bob	lassen

Filtered Metadata

Profile	User	Platform
P2	Bob	lassen

Performance metrics

Node	Profile	Cache Misses
MAIN	P1	
	P2	
FOO	P1	
	P2	
BAR	P1	
	P2	
BAZ	P1	
	P2	

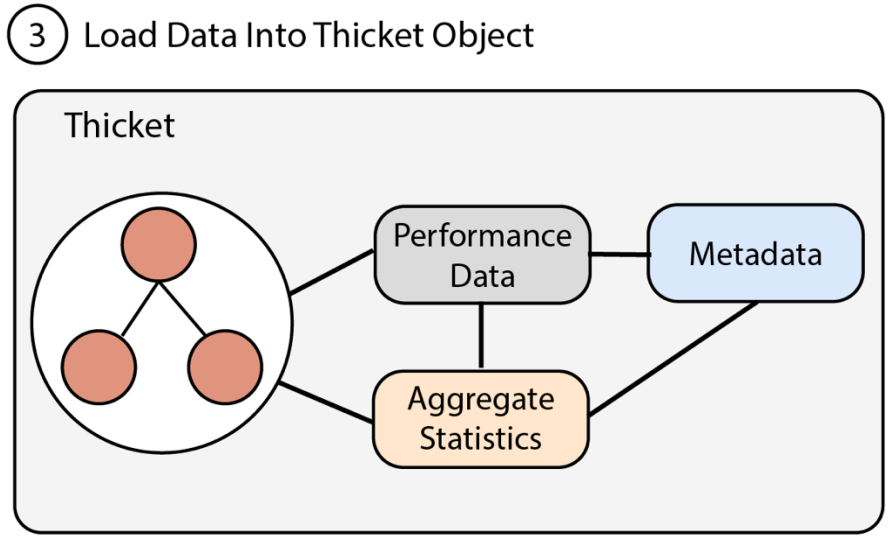
Filter on metadata:
platform=="lassen" &&
user=="Bob"

Filtered Performance metrics

Node	Profile	Cache Misses
MAIN	P2	
FOO	P2	
BAR	P2	
BAZ	P2	

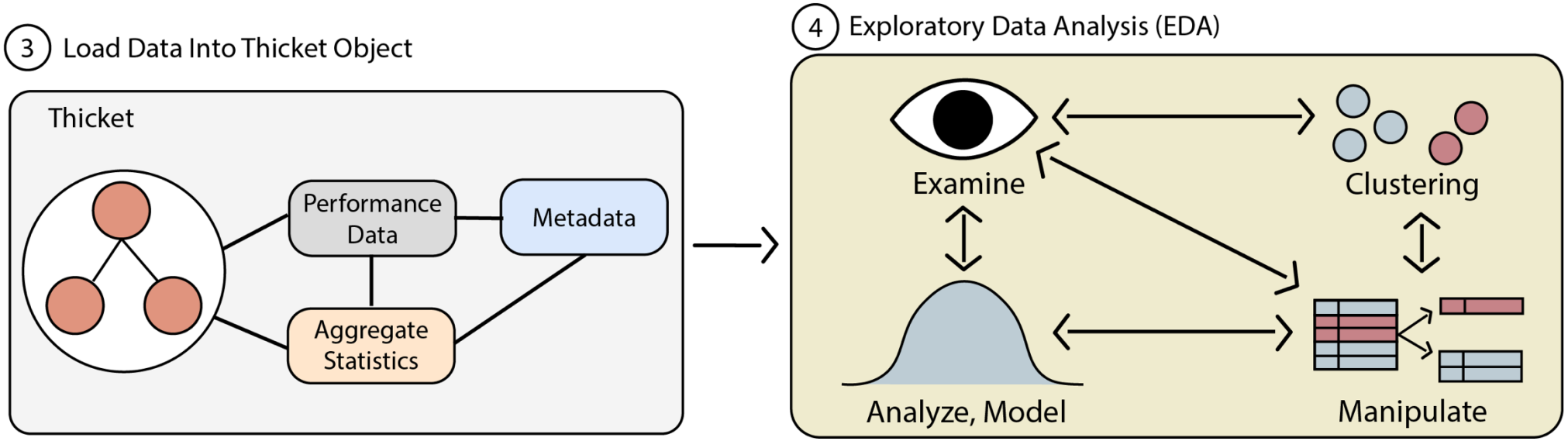
Metadata fields useful for understanding and manipulating thicket object!

Thicket enables exploratory data analysis of multi-run data



- Compose data from diff. sources and types
  - Different scaling (e.g., strong, weak)
  - Different application parameters
  - Different compilers and optimization levels
  - Different hardware types (e.g., CPUs, GPUs)
  - Different performance tools

Thicket enables exploratory data analysis of multi-run data



- Compose data from diff. sources and types
  - Different scaling (e.g., strong, weak)
  - Different application parameters
  - Different compilers and optimization levels
  - Different hardware types (e.g., CPUs, GPUs)
  - Different performance tools

- Perform analysis on the thicket of runs
  - Manipulate the set of data
  - Visualize the dataset
  - Perform analysis on the data
  - Model data
  - Leverage third-party tools in the Python ecosystem

Case Study 1: RAJA Performance Suite

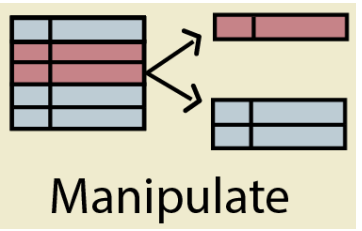
- Open-source suite of loop-based kernels commonly found in HPC applications showcasing performance of different programming models on different hardware

http://github.com/llnl/rajaperf

- 560 runs/profiles:
  - 2 clusters (CPU, CPU+GPU)
  - 4 problem sizes
  - 3 compilers, 4 optimizations
  - 3 programming models (sequential, OpenMP, CUDA)
  - 3 performance tools (Caliper, PAPI, Nsight Compute)

	cluster	systype build	problem size	compiler	compiler optimizations	omp num threads	cuda compiler	block sizes	RAJA variant	#profiles
0	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	clang++-9.0.0	[-O0, -O1, -O2, -O3]	1	N/A	N/A	Sequential	160
1	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	g++-8.3.1	[-O0, -O1, -O2, -O3]	1	N/A	N/A	Sequential	160
2	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	clang++-9.0.0	-O0	72	N/A	N/A	OpenMP	40
3	quartz	toss_3_x86_64_ib	[1M, 2M, 4M, 8M]	g++-8.3.1	-O0	72	N/A	N/A	OpenMP	40
4	lassen	blueos_3_ppc64le_ib_p9	[1M, 2M, 4M, 8M]	xlc++_r-16.1.1.12	-O0	1	nvcc-11.2.152	[128, 256, 512, 1024]	CUDA	160

Use Thicket to *compose* multi-platform, multi-tool data



Thicket object composed of 2 profiles run on CPU

node	problem_size	time (exc)	Reps	Retiring	Backend bound
Apps_NODAL_ACCUMULATION_3D	1M	0.204583	100	0.144928	0.783786
	4M	0.795511	100	0.139002	0.788017
Apps_VOL3D	1M	0.067061	100	0.402238	0.510525
	4M	0.241508	100	0.400775	0.515976

Thicket object composed of 2 profiles run on GPU

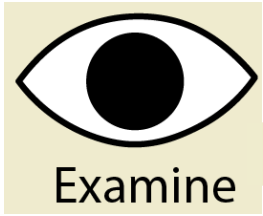
node	problem_size	time (gpu)	gpu__compute_memory_throughput	gpu__dram_throughput	sm__throughput
Apps_NODAL_ACCUMULATION_3D	1M	0.007478	70.689752	46.724767	7.330745
	4M	0.026951	74.275834	51.257993	7.688628
Apps_VOL3D	1M	0.006028	81.012826	67.751194	35.676942
	4M	0.021422	91.929933	70.122011	35.386470

CPU

GPU

node	problem_size	time (exc)	Reps	Retiring	Backend bound	time (gpu)	gpu__compute_memory_throughput	gpu__dram_throughput	sm__throughput
Apps_NODAL_ACCUMULATION_3D	1M	0.204583	100	0.144928	0.783786	0.007478	70.689752	46.724767	7.330745
	4M	0.795511	100	0.139002	0.788017	0.026951	74.275834	51.257993	7.688628
Apps_VOL3D	1M	0.067061	100	0.402238	0.510525	0.006028	81.012826	67.751194	35.676942
	4M	0.241508	100	0.400775	0.515976	0.021422	91.929933	70.122011	35.386470

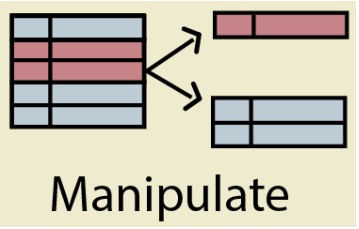
Analyze multi-architecture/multi-tool data



- Dataset: 4 types of profiles side-by-side to compare CPU to GPU performance
  1. Basic CPU metrics from Caliper
  2. Top-down metrics from Caliper/PAPI
  3. GPU runtime from Caliper
  4. GPU metrics from Nsight Compute
- Examples of analysis:
  - Compute CPU/GPU speedup
  - Correlate memory and compute usage on the CPU vs. GPU

		1 CPU			2 CPU top-down		3 GPU	4 GPU Nsight Compute				Derived
Node	Problem size	time (exc)	Bytes/Rep	Flops/Rep	Retiring	Backend bound	time (gpu)	gpu__compute_memory_throughput	gpu__dram_throughput	sm__throughput	sm__warps_active	speedup
Apps_VOL3D	8M	0.498815	282109496	632421288	0.377843	0.540604	0.040761	93.742058	72.140428	36.206767	54.459589	12.237556
Lcals_HYDRO_1D	8M	2.077556	201326600	41943040	0.032965	0.909545	0.242928	92.944968	92.944968	6.595714	95.266148	8.552147

Manipulate: Filter using call path query




```
0.001 Base_CUDA
└─ 0.000 Algorithm
   ├─ 0.000 Algorithm_MEMCPY
   │  ├─ 0.002 Algorithm_MEMCPY.block_128
   │  ├─ 0.009 Algorithm_MEMCPY.block_256
   │  └─ 0.006 Algorithm_MEMCPY.library
   ├─ 0.000 Algorithm_MEMSET
   │  ├─ 0.001 Algorithm_MEMSET.block_128
   │  ├─ 0.004 Algorithm_MEMSET.block_256
   │  └─ 0.003 Algorithm_MEMSET.library
   ├─ 0.000 Algorithm_REDUCE_SUM
   │  ├─ 0.003 Algorithm_REDUCE_SUM.block_128
   │  ├─ 0.004 Algorithm_REDUCE_SUM.block_256
   │  └─ 0.002 Algorithm_REDUCE_SUM.cub
   └─ 0.000 Algorithm_SCAN
      └─ 0.006 Algorithm_SCAN.default
```

Input call tree

Filter on call path:
(1) Node named
"Base_CUDA"

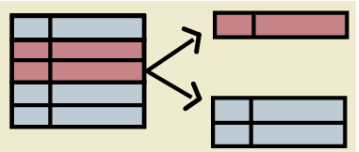
```
0.001 Base_CUDA
```

Output call tree

RAJA

I Lumsden et al. "Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications", e-Science 2022

Manipulate: Filter using call path query




```
0.001 Base_CUDA
└─ 0.000 Algorithm
   ├─ 0.000 Algorithm_MEMCPY
   │  ├─ 0.002 Algorithm_MEMCPY.block_128
   │  ├─ 0.009 Algorithm_MEMCPY.block_256
   │  └─ 0.006 Algorithm_MEMCPY.library
   ├─ 0.000 Algorithm_MEMSET
   │  ├─ 0.001 Algorithm_MEMSET.block_128
   │  ├─ 0.004 Algorithm_MEMSET.block_256
   │  └─ 0.003 Algorithm_MEMSET.library
   ├─ 0.000 Algorithm_REDUCE_SUM
   │  ├─ 0.003 Algorithm_REDUCE_SUM.block_128
   │  ├─ 0.004 Algorithm_REDUCE_SUM.block_256
   │  └─ 0.002 Algorithm_REDUCE_SUM.cub
   └─ 0.000 Algorithm_SCAN
      └─ 0.006 Algorithm_SCAN.default
```

Input call tree

Filter on call path:

(1) Node named "Base_CUDA"

(2) Node with "block_128" in name (and any nodes in between)

```
0.001 Base_CUDA
└─ 0.000 Algorithm
   ├─ 0.000 Algorithm_MEMCPY
   │  └─ 0.002 Algorithm_MEMCPY.block_128
   ├─ 0.000 Algorithm_MEMSET
   │  └─ 0.001 Algorithm_MEMSET.block_128
   └─ 0.000 Algorithm_REDUCE_SUM
      └─ 0.003 Algorithm_REDUCE_SUM.block_128
```

Output call tree

RAJA

I Lumsden et al. "Enabling Call Path Querying in Hatchet to Identify Performance Bottlenecks in Scientific Applications", e-Science 2022

30 Min Break

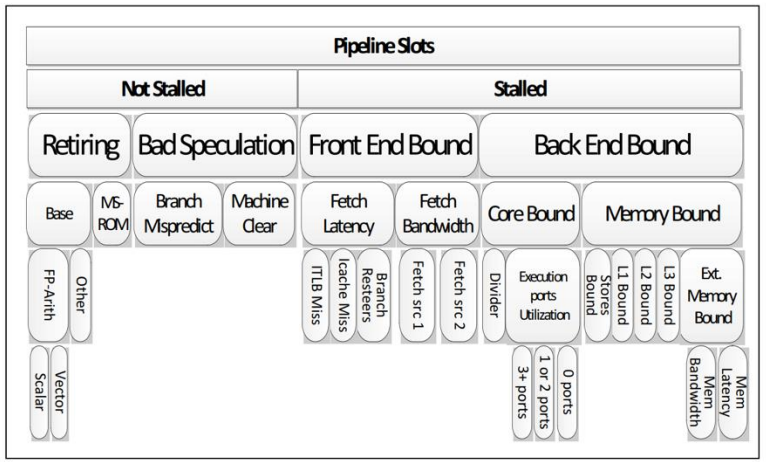
See you at 3:30pm

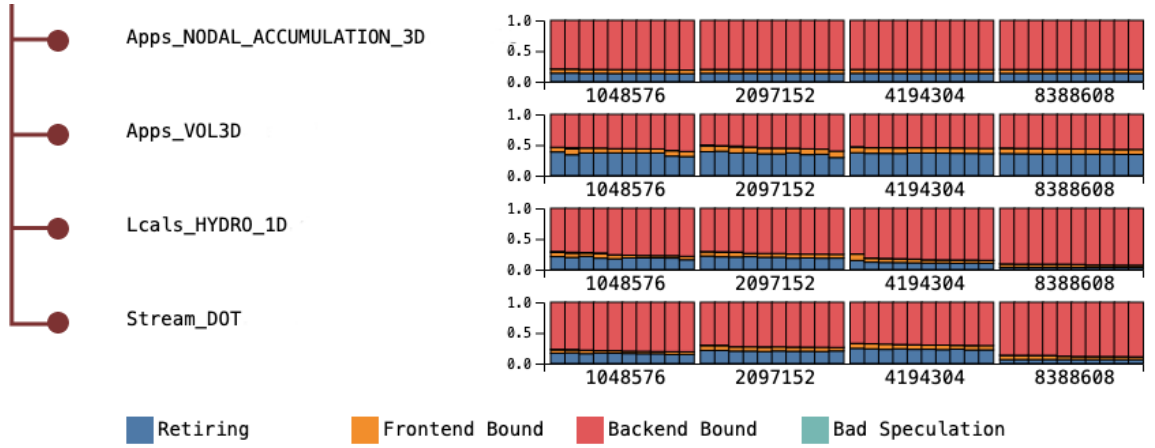
Tutorial: Reproducible Performance Measurement and Analysis for High-Performance Computing Applications

Performance Analysis with Thicket II

Stephanie Brink

Visualize: Intel CPU top-down analysis



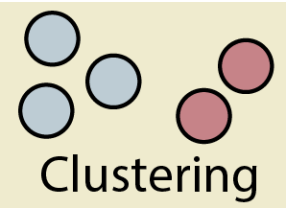


- *Top-down analysis* uses HW counters in a hierarchy to identify bottlenecks*
- Use Caliper's top-down module to derive top-down metrics for call-tree regions

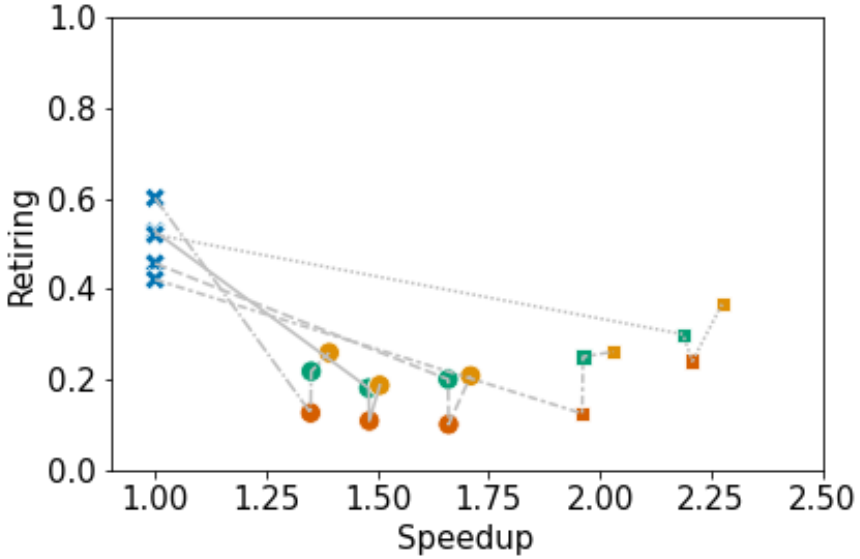
- Thicket's *tree+table* visualization shows top-down metrics as stacked bar charts, each bar is a profile
  - Apps_VOL3D has the highest retiring rates
  - Lcals_HYDRO and Stream_DOT become more backend bound as problem size grows

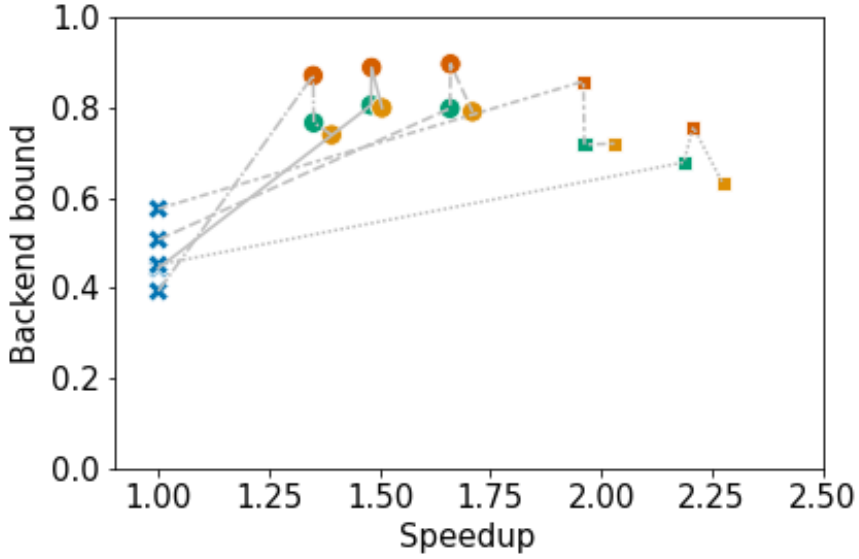
* Yasin, A.: A Top-Down Method for Performance Analysis and Counters Architecture. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 35–44. IEEE, CA, USA (Mar 2014).

Use third-party Python libraries, e.g., Scikit-learn clustering



1. Select data of interest
   - Filter 8M problem size
   - Use query language to extract all implementations of the Stream kernel
2. (optional) Normalize data
3. Apply scikit-learn clustering to top-down analysis metrics of runs with different compiler optimization levels



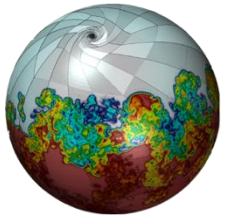


Optimization Level
- -O0
- -O1
- -O2
- -O3

K-Means Clusters
- 0
- 1
- 2

Kernels
- Stream_ADD
- Stream_COPY
- Stream_DOT
- Stream_MUL
- Stream_TRIAD

RAJA

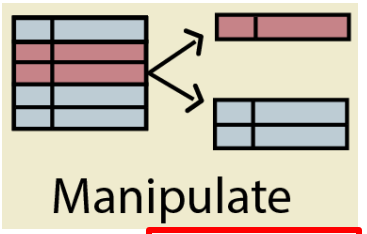


Case Study 2: MARBL multi-physics code

- MARBL is a next-generation multi-physics code developed at LLNL
- 60 runs/profiles:
  - 2 clusters (rztopaz, AWS ParallelCluster)
  - 2 MPI libraries (impi, openmpi)
  - 6 node/rank counts
  - 5 repeat runs per config

	cluster	ccompiler	mpi	version	numhosts	mpi.world.size	#profiles
0	ip----	/usr/tce/packages/clang/clang-9.0.0	impi	v1.1.0-203-gcb0efb3	[1, 2, 4, 8, 16, 32]	[36, 72, 144, 288, 576, 1152]	30
1	rztopaz	/usr/tce/packages/clang/clang-9.0.0	openmpi	v1.1.0-201-g891eaf1	[1, 2, 4, 8, 16, 32]	[36, 72, 144, 288, 576, 1152]	30

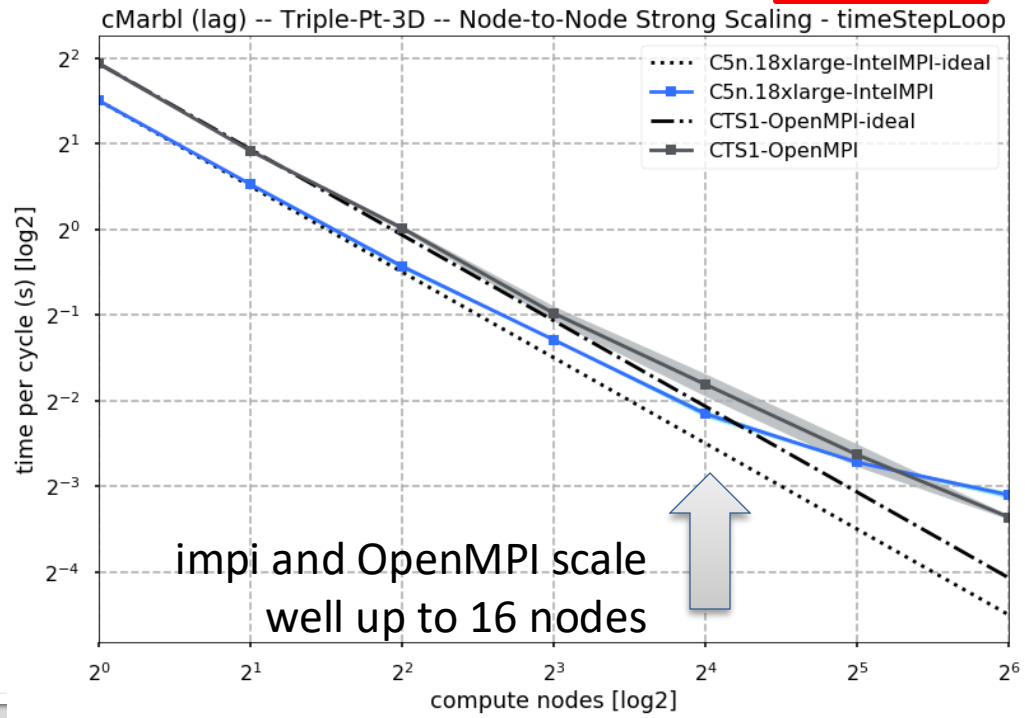
Manipulate: Compute noise and scaling

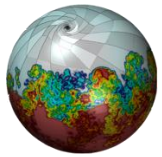


node	profile	Total time	name	mpi.world.size
{'name': 'main', 'type': 'function'}	-8554409769265002864	58036.664552	main	144
	-7335101512240609798	55318.808836	main	36
	-6029692086108825020	156984.246813	main	2304
	-5606382734792961361	64122.371533	main	288
	-4058809097109060732	155040.998627	main	2304
	-3193575964635936033	71010.504038	main	576
	-2978339073585311581	55910.708449	main	72
	-2939704488254773514	157934.204076	main	2304
	-2771797711381234985	56893.512948	main	144
	-2638513839856695106	97432.260966	main	1152

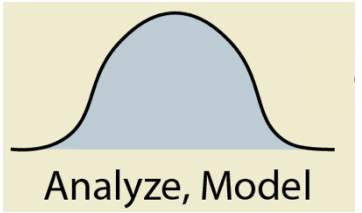
node	profile	Total time	name	mpi.world.size
{'name': 'main', 'type': 'function'}	-7335101512240609798	55318.808836	main	36
	-843517585394879415	55110.656885	main	36
	7720382918482619866	55155.581578	main	36
	8293335926964337960	55139.134916	main	36
	8335957980556391465	55013.682102	main	36

1. Use groupby(mpi.world.size) to generate unique subsets of data which are repeated runs; compute noise
2. Compose runs on different platforms and at different scales
3. Generate strong scaling plot with matplotlib
   - Deviation shown in shaded region, dots are average of 5 runs



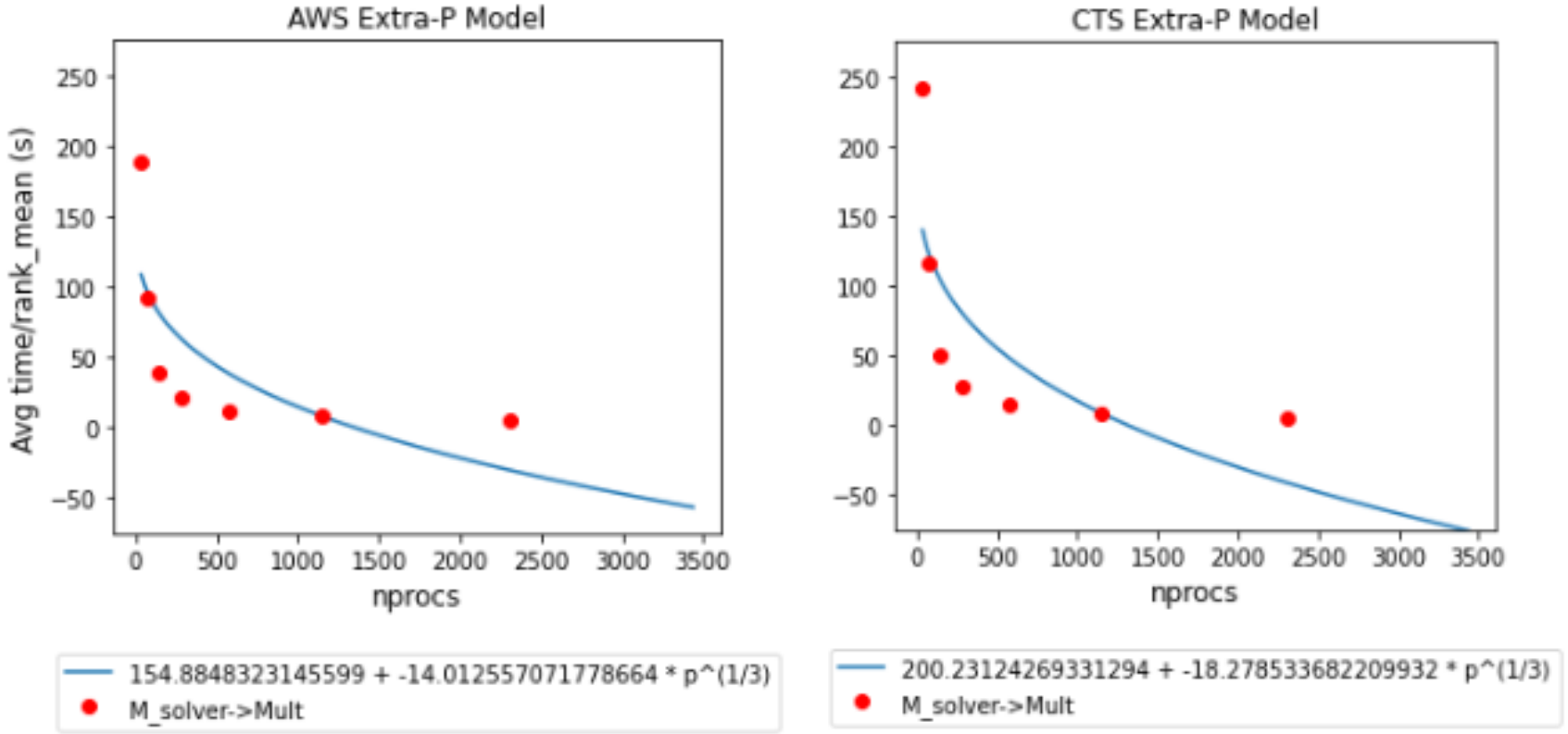


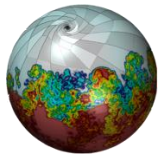
Model: Use third-party Python library, Extra-P



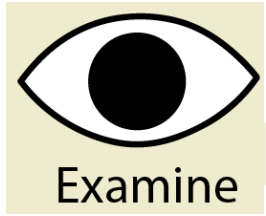
Extra-P derives an analytical performance model from an ensemble of profiles covering one or more modeling parameters http://github.com/extra-p/extrap

- Select functions of interest
- Call Extra-P to model scaling on different hardware types

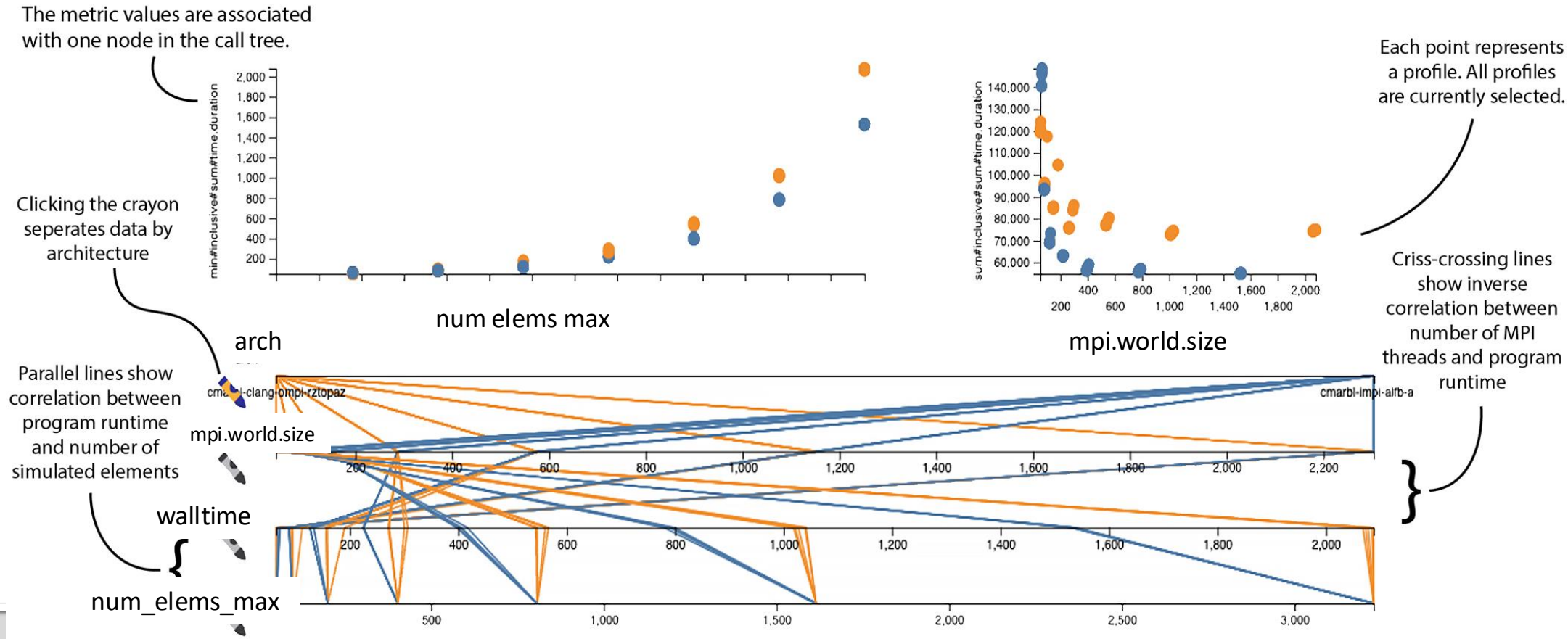




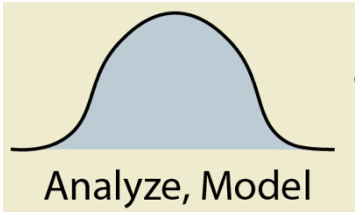
Visualize metadata with parallel coordinates plot



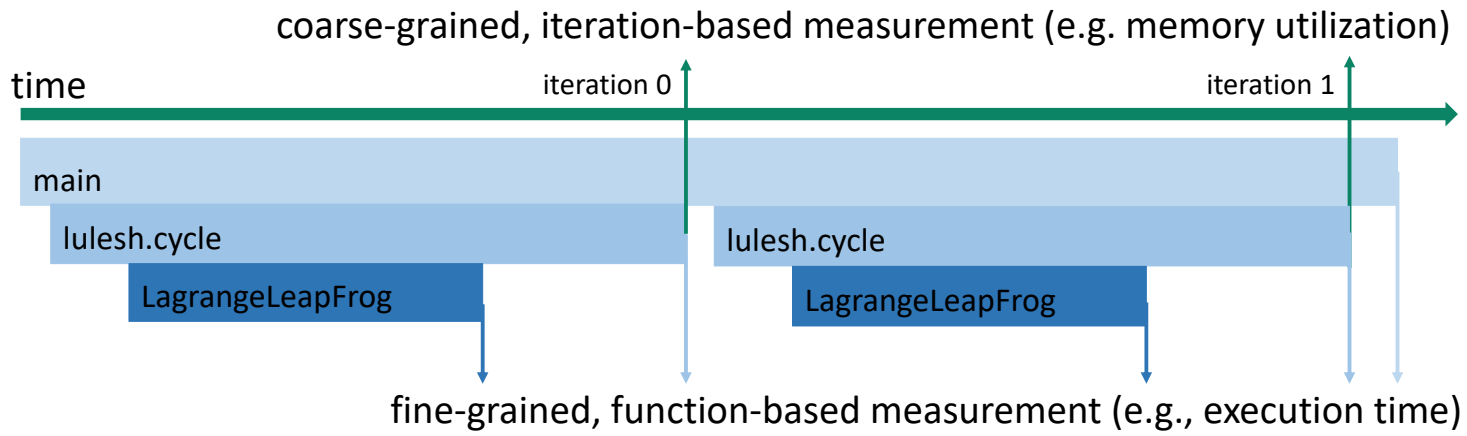
- Thicket's interactive parallel coordinates plot shows relationships between metadata variables, and between metadata and performance data



Can we observe performance fluctuations over time?



- Caliper collects metrics at set intervals
- Thicket can then categorize temporal patterns [1]:

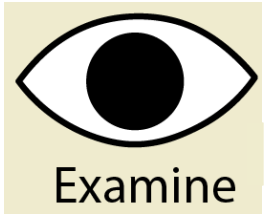


$$P_{temporal}(t) = 1 - \frac{\sum_{t=0}^{T} M_t}{\sum_{t=0}^{T} max_{0<t<T} M_t}$$

Pattern	Constant	Phased	Dynamic	Sporadic
Score	0.0-0.2	0.2-0.4	0.4-0.6	0.6-1.0
Symbol	→	⇝	⇝	⇝

[1] I.B. Peng, I. Karlin, M. B. Gokhale, K. Shoga, M. P. LeGendre, and T. Gamblin. A holistic view of memory utilization on hpc systems: Current and future trends. Proceedings of the International Symposium on Memory Systems, 2021.

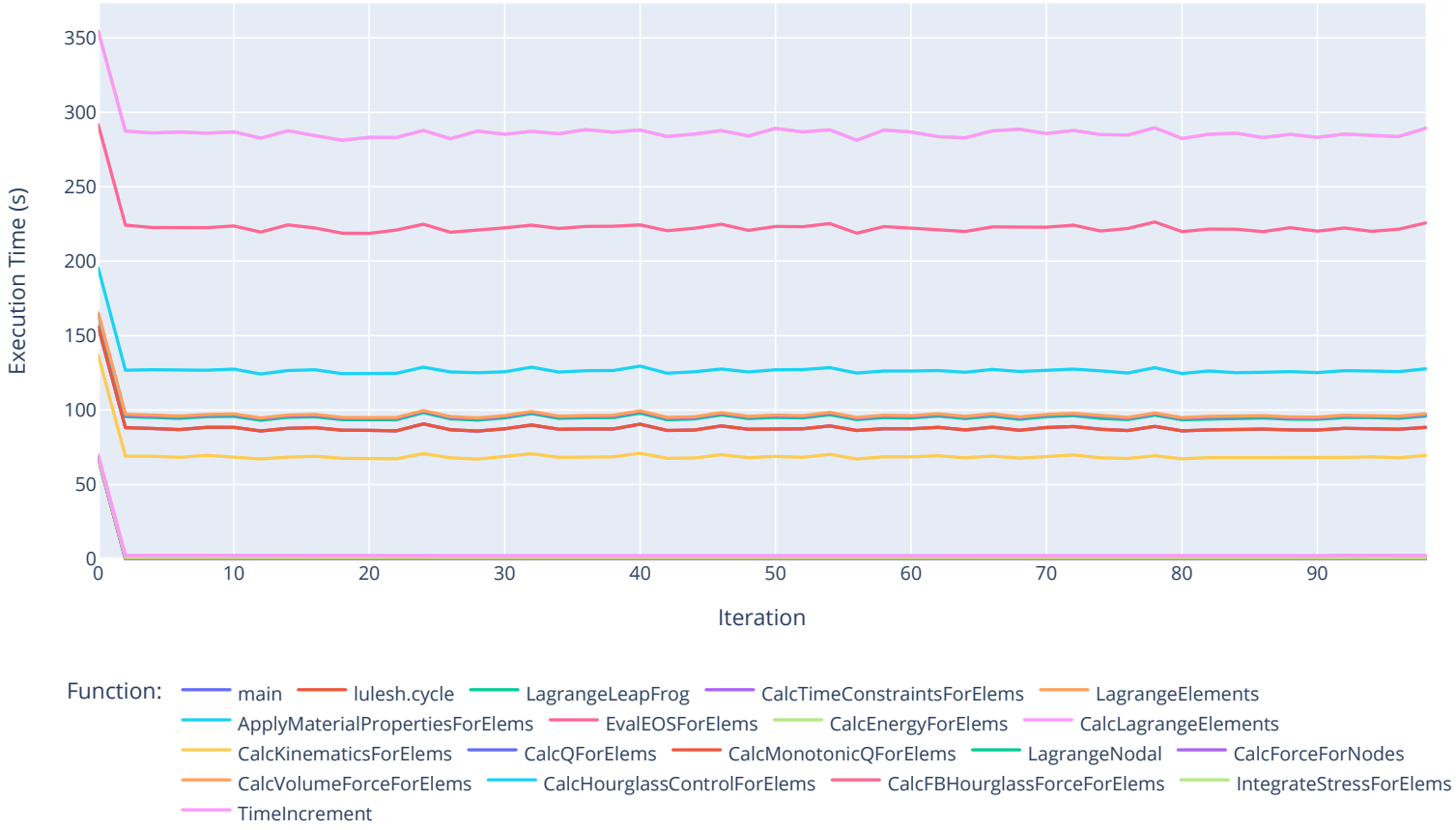
Timeseries metrics visualizations



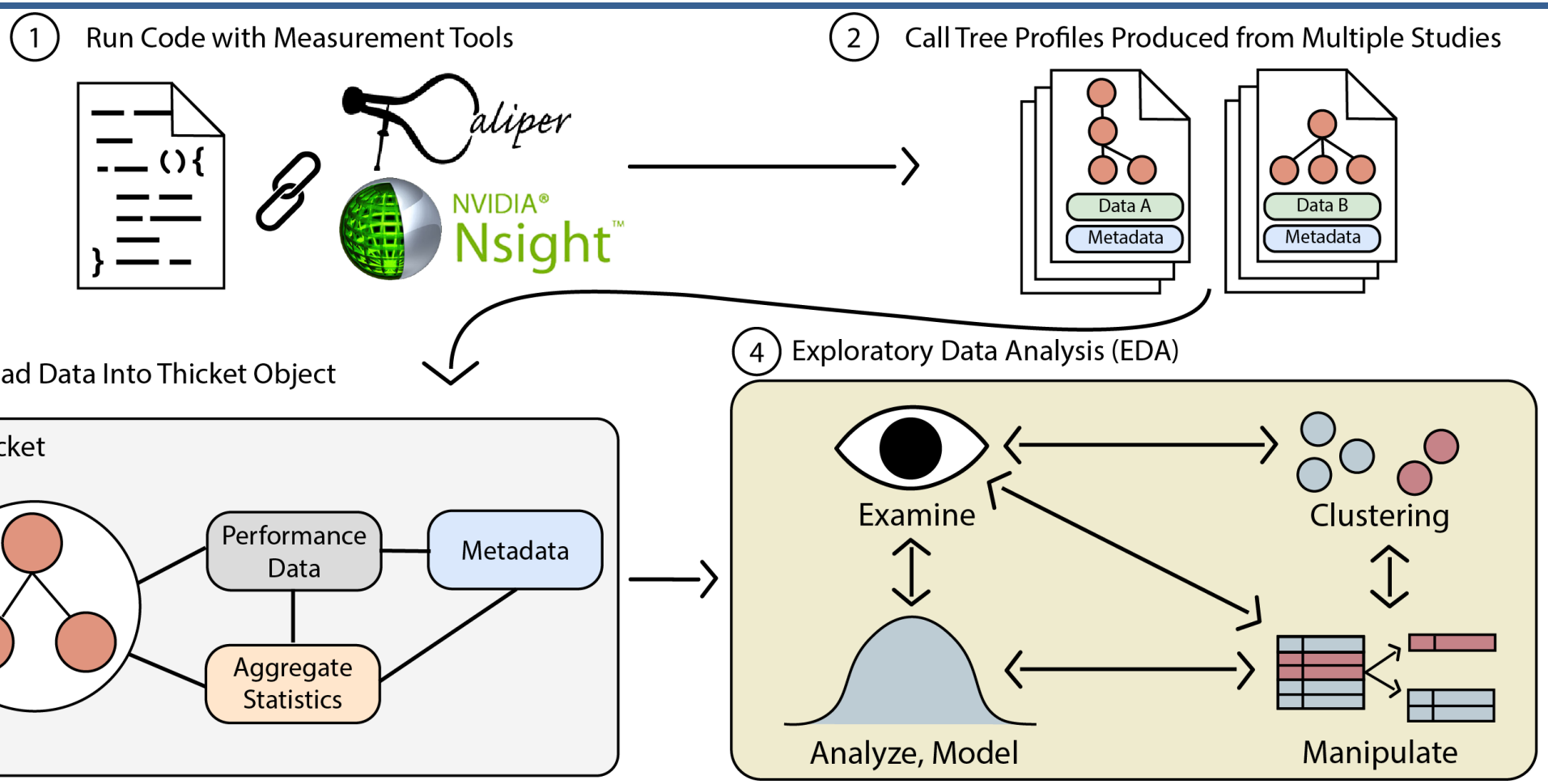
- Display pattern symbol and temporal score as part of the call tree
- Use python plotting libraries to create a more granular visualization

```
0.488 ⇝ lulesh.cycle
├─ 0.280 ⇝ LagrangeLeapFrog
│   ├─ 0.242 ⇝ CalcTimeConstraintsForElems
│   ├─ 0.151 ⇝ LagrangeElements
```

```
67788.343 → lulesh.cycle
├─ 8168.648  LagrangeLeapFrog
│   ├─ 8783.894  CalcTimeConstraintsForElem
│   ├─ 398433.500  LagrangeElements
```



Thicket is a toolkit for exploratory data analysis of multi-run data



https://github.com/LLNL/thicket
https://github.com/LLNL/thicket-tutorial

Tutorial Instances: http://bit.ly/4kGQDlc

- We have an AWS instance for the hands-on component of this tutorial
- The instance provides:
  - Pre-installed Thicket, Caliper, and Benchpark and required dependencies
  - Caliper source code demos
  - Thicket Jupyter notebooks and datasets for performance analysis

When logging in to the instance:

- Please use a unique username to avoid resource allocation conflicts
  - First initial followed by last name (e.g., jdoe)
- PW: hpctutorial25

CASC

Center for Applied
Scientific Computing

Reproducible Experiments with Benchpark

Stephanie Brink

We benchmark HPC systems for a variety of reasons

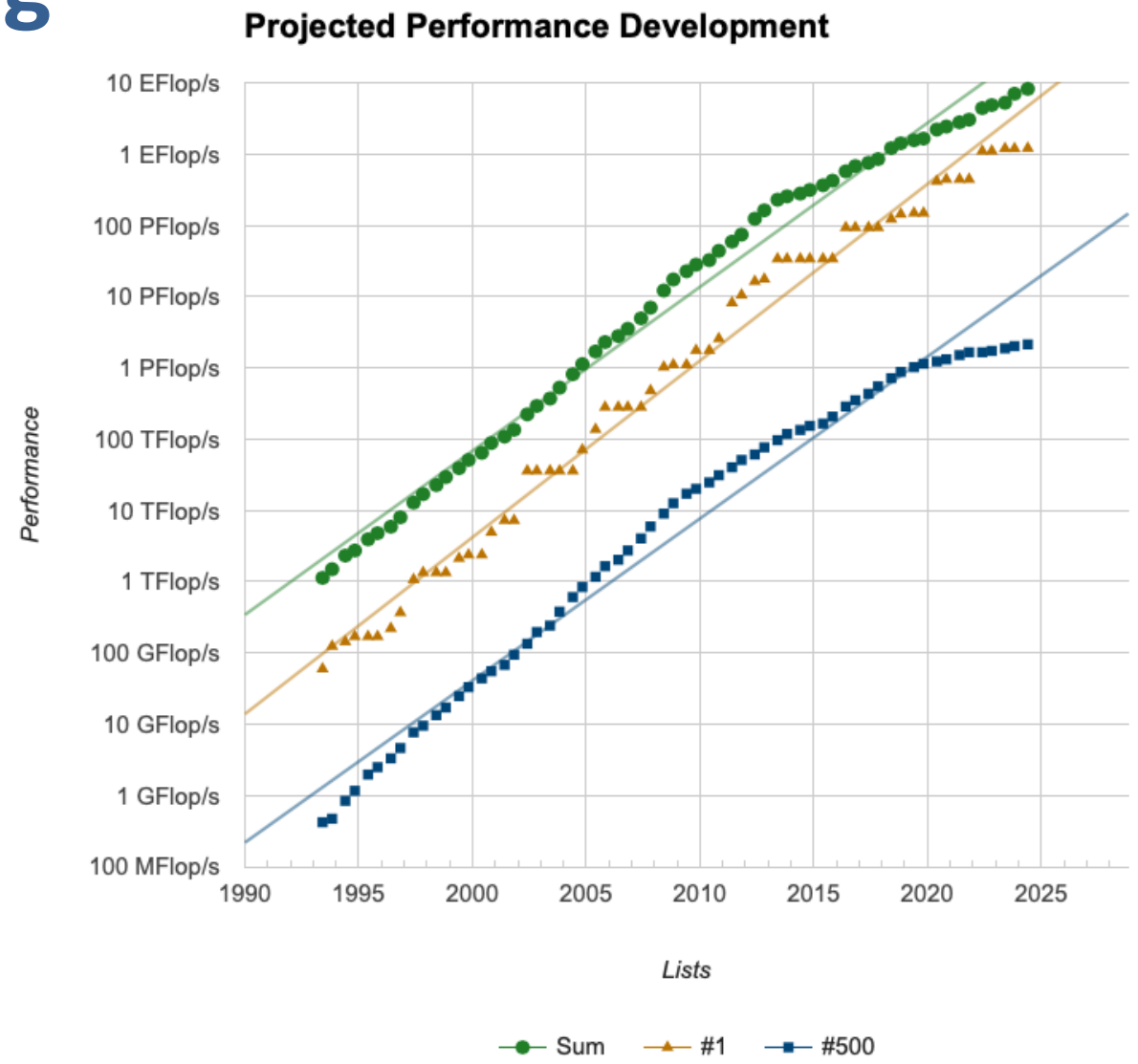
- Procurements
  - *Communicate* datacenter workload to vendors
  - Co-design systems, monitor progress
  - System acceptance (contractual specification)
- Validation of software stack, tools
  - Compilers
  - Debuggers
  - Correctness tools
  - Performance tools
- Research
  - Programming models
  - Computational methods
  - Performance across architectures

Many stakeholders, communication is key for collaboration

Source: https://www.top500.org/statistics/perfdevel/

Benchmarking is challenging

- What are we trying to characterize?
- Are we capturing the best the system can do?
- Is something else impacting performance?
- Did we build and run the code in the *optimal* and *reproducible* way?



HPC benchmarks run on diverse HPC systems

Lawrence Berkeley Nat'l Lab
AMD Zen + **NVIDIA**

F u g a k u
RIKEN **Fujitsu** ARM a64fx

Lawrence Livermore Nat'l Lab
IBM Power9 + **NVIDIA**

Lawrence Livermore Nat'l Lab
AMD Zen + **Radeon**

Oak Ridge National Lab
AMD Zen + **Radeon**

Argonne National Lab
Intel Xeon + Xe

- Benchmark source code
  - Abstraction (OpenMP, RAJA, Kokkos)
  - Hardware-specific (CUDA, ROCm)
- Optimized code for the CPU and GPU
  - Must make effective use of the hardware
  - Can make 10-100x performance difference
- Rely heavily on system packages
  - Need to use optimized communication and MPI libraries that come with machines

Writing benchmark source code is only the beginning

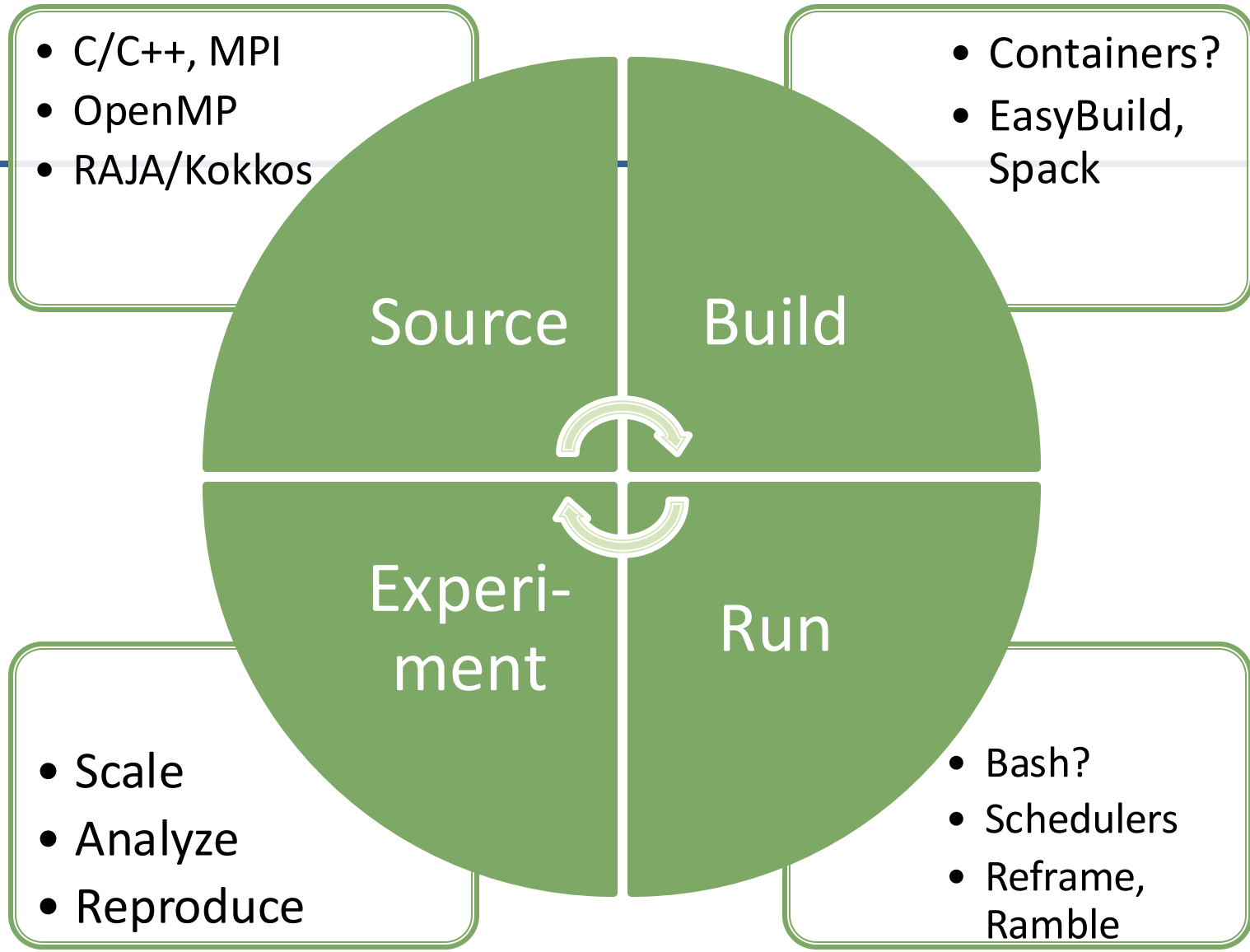
State of the practice:
HPC system benchmarking is manual!

- **Building** on each system is different, porting the builds to new systems is manual
- **Running** on each system is different, porting run scripts to new systems is manual
- Systems keep **changing**, requiring updates to how we build and run benchmarks
- **Triggering** builds and runs is manual: benchmark results don't stay up to date
- **Performance analysis** of results is manual

Communicate technical specification via code, documentation, white papers

HPC Benchmarks are HPC Software

- Portability
- Maintenance
- Testing/CI
- Verification
- Reproducibility



All components must work *for your system*, focus on explainable *performance*

Benchpark enables complete specification of HPC benchmarks

Infrastructure-as-code benchmark specification codifies:

- Benchmark build and run instructions
- **HPC Systems**
- **HPC Experiments**

github.com/llnl/benchpark

Leverage advances in HPC automation

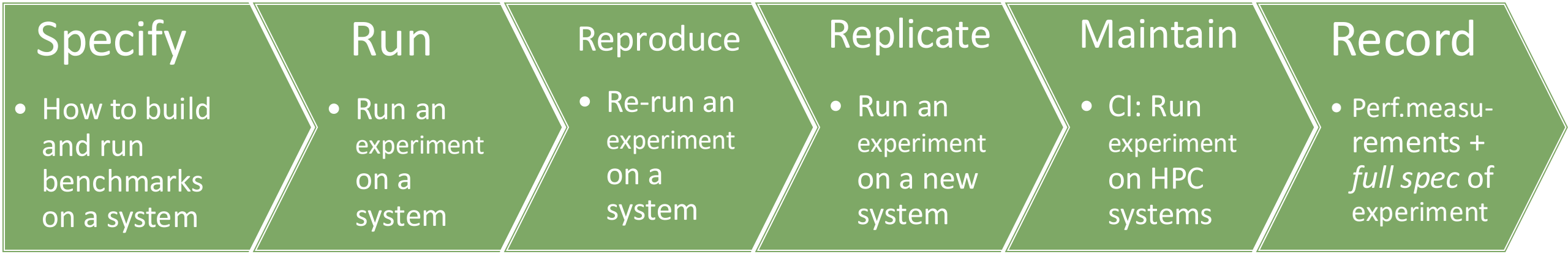
- Source code
- Build specification
- Run specification
- CI

Ramble

Spack

Every part of the specification codified: use to communicate, automate

Benchpark enables *reproducible* specifications of benchmarks



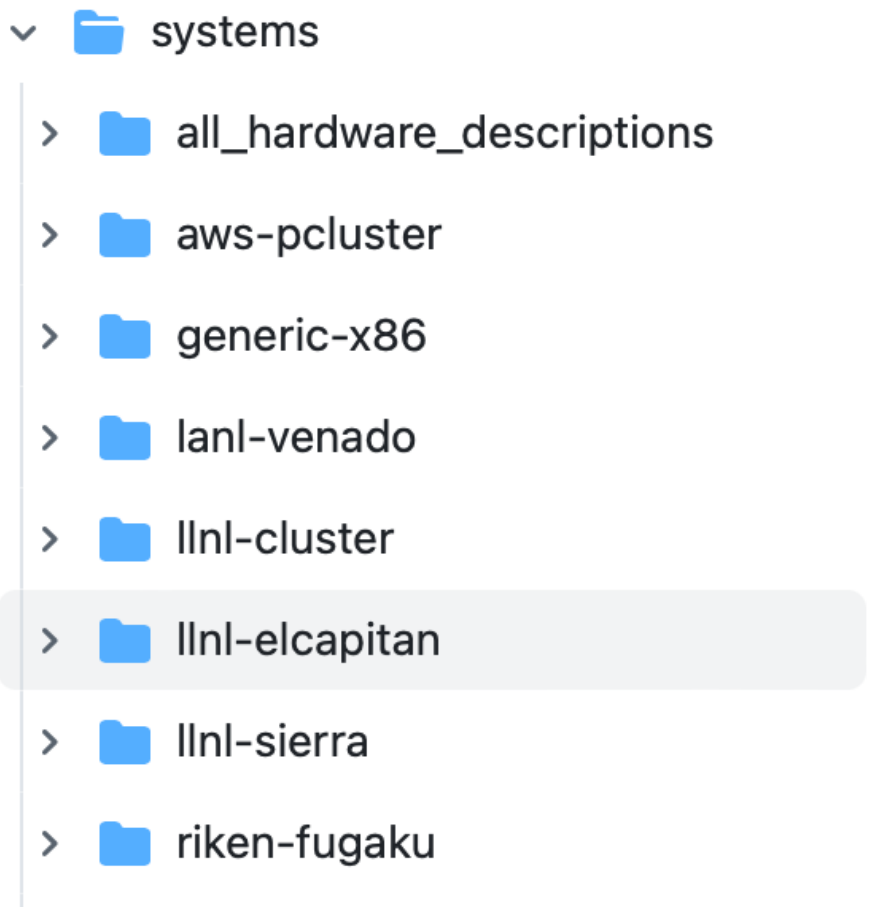
Full specification enables reproducibility, replicability, and automation

HPC System definition for *Performance*

- Resources (CPU cores, GPUs)
- Scheduler (Slurm, flux)
- Process mapping (cores, sockets, GPUs, NICs)
- On-node parallelism (CUDA, ROCM, OpenMP)
- Software stack
  - Compilers (which is best?)
  - MPI implementations (best?)
  - Math libraries

Goal: Use the system correctly

HPC System in Benchpark: *Specify Once*




```
class LlnlElcapitan(System):
    variant(
        "rocm",
        default="5.5.1",
        values=("5.4.3", "5.5.1", ”6.2.4"),
        description="ROCm version",
    )

    variant(
        "compiler",
        default="cce",
        values=("gcc", "cce", "rocmcc"),
        description="Which compiler to use",
    )

    def initialize(self):
        super().initialize()
        self.scheduler = "flux"
        self.sys_cores_per_node = "128"
```

benchpark system init --dest=elcap llnl-elcapitan rocm=6.2.4 compiler=cce

System Specifications

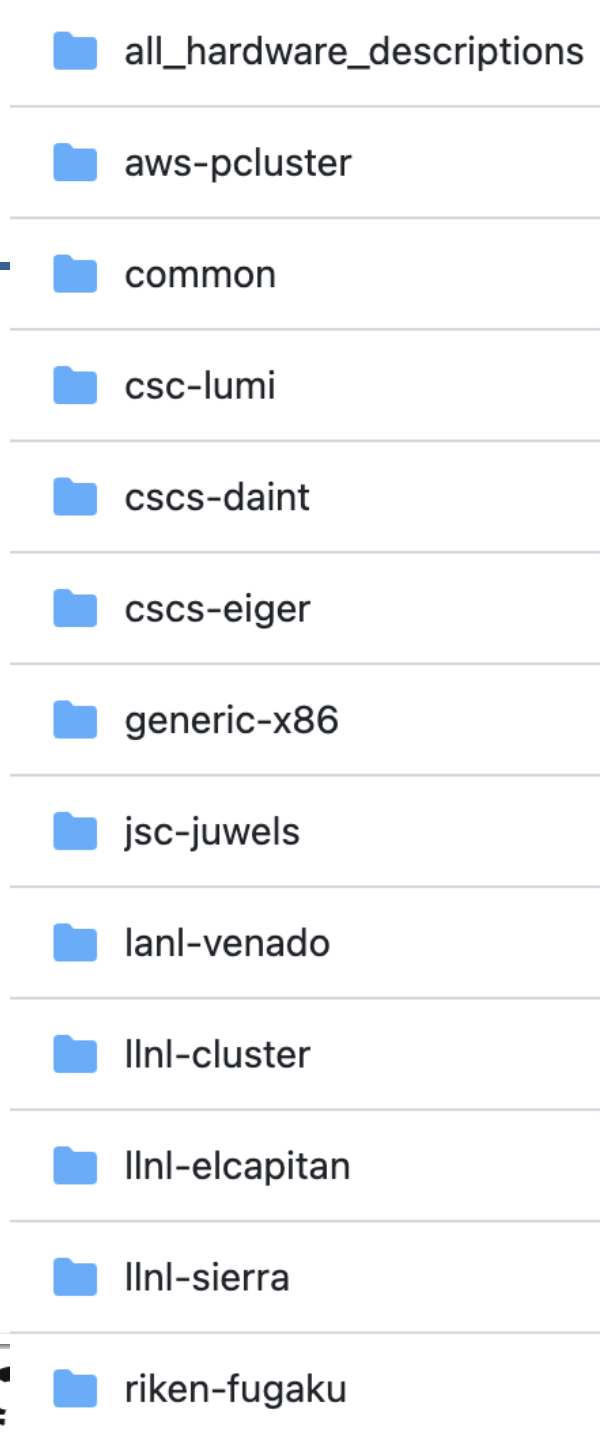
The table below provides a directory of information for systems that have been specified in Benchpark. The column headers in the table below are available for use as the `system` parameter in `benchpark setup`.

Search:

name	integrator.vendor	integrator.name	processor.vendor	processor.name	processor.ISA	processor.u
Atos-zen2-A100-Infiniband	Atos	XH2000	AMD	EPYC-Zen2	x86_64	zen2
AWS_PCluster-zen-EFA	AWS	ParallelCluster	AMD	EPYC-Zen	x86_64	zen
DELL-cascadelake-InfiniBand	DELL		Intel	Xeon6248R	x86_64	cascadelake
DELL-sapphirerapids-OmniPath	DELLEMC	PowerEdge	Intel	XeonPlatinum8480	x86_64	sapphirerapi
Fujitsu-A64FX-TofuD	Fujitsu	FX1000	Fujitsu	A64FX	Armv8.2-A-SVE	aarch64
HPECray-haswell-P100-Infiniband	HPECray		Intel	Xeon-E5-2650v3	x86_64	haswell
HPECray-neoverse-H100-Slingshot	HPECray	EX254n	NVIDIA	Grace	Armv9	neoverse
HPECray-zen2-Slingshot	HPECray		AMD	EPYC-7742	x86_64	zen2
HPECray-zen3-MI250X-Slingshot	HPECray	EX235a	AMD	EPYC-Zen3	x86_64	zen3
HPECray-zen4-MI300A-Slingshot	HPECray	EX255a	AMD	EPYC-Zen4	x86_64	zen4
IBM-power9-V100-Infiniband	IBM	AC922	IBM	POWER9	ppc64le	power9
Penguin-icelake-OmniPath	PenguinComputing	RelionCluster	Intel	XeonPlatinum924248C	x86_64	icelake
Supermicro-icelake-OmniPath	Supermicro		Intel	XeonPlatinum8276L	x86_64	icelake
x86_64					x86_64	

Systems in Benchpark: July 2025

- 4 in Europe
- 6 in US labs
- 1 in Japan
- 1 at a university
- 2 cloud systems



HPC Experiment definition for *Performance*

- On-node parallelism (CUDA, ROCM, OpenMP)
- Problem sizes
  - Overall problem size, or
  - Per node or per GPU
- Scaling studies
  - How to scale
  - How to decompose
- Resources (cores, GPUs)

Goal: Specify reproducible sets of experiments that map onto specific Systems

HPC Experiment in Benchpark: *Specify Once*

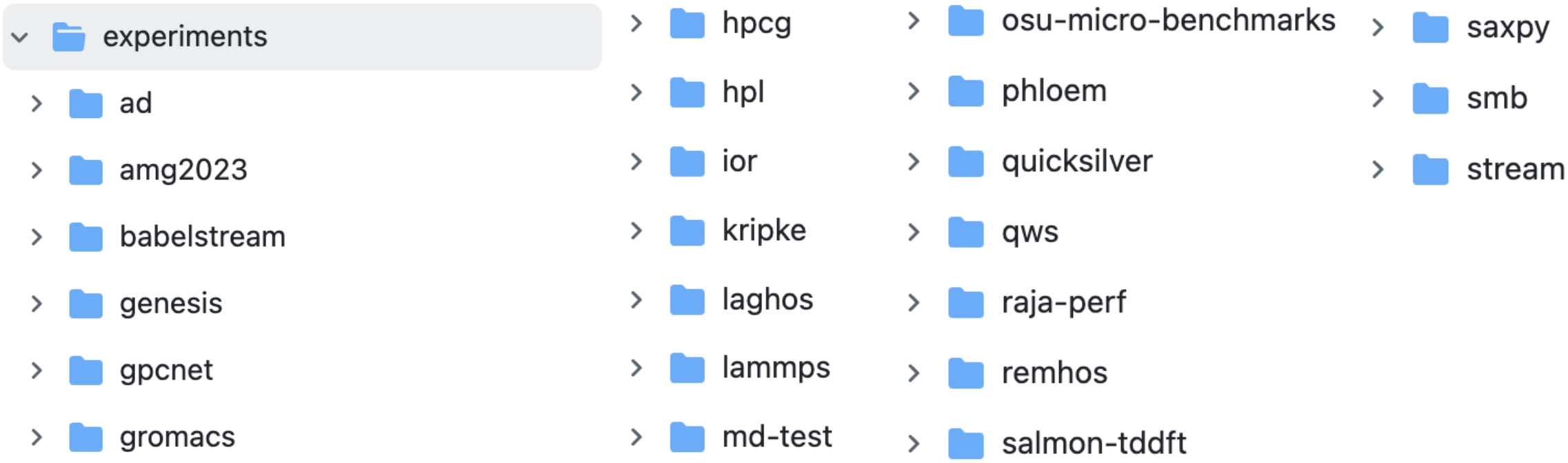
```
class Amg2023(Experiment, OpenMPExperiment, CudaExperiment, ROCmExperiment,
        Scaling(ScalingMode.Strong, ScalingMode.Weak, ScalingMode.Throughput,
                Caliper):
  variant(
    "workload",
    default="problem1",
    values=("problem1", "problem2"),
    description="problem1 or problem2",
  )

  def compute_applications_section(self):
        self.register_scaling_config({
            ScalingMode.Strong: {
                “n_resources”: lambda var, itr, dim, scaling_factor: var.val(dim) * scale,
                “problem_size”: lambda var, itr, dim, scaling_factor: var.val(dim) // scale,
            },
            ScalingMode.Weak: {
                “n_resources”: lambda var, itr, dim, scaling_factor: var.val(dim) * scale,
                “problem_size”: lambda var, itr, dim, scaling_factor: var.val(dim),
            },
            ScalingMode.Throughput: {
                “n_resources”: lambda var, itr, dim, scaling_factor: var.val(dim),
                “problem_size”: lambda var, itr, dim, scaling_factor: var.val(dim) * scale,
```

amg2023 +rocm scaling=strong workload=problem2 caliper=mpi,time

Benchmarks and Experiments

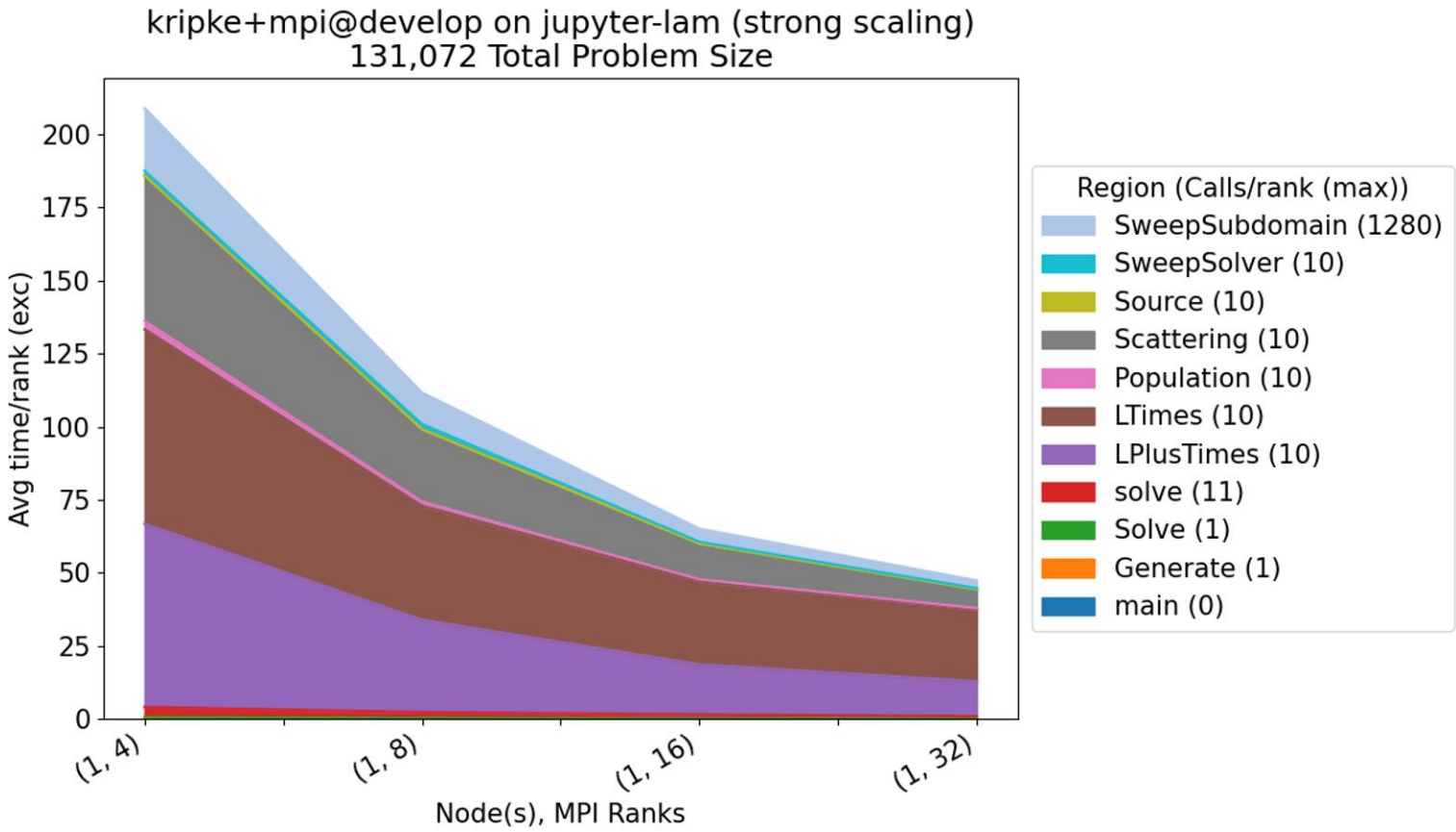
	ad	amg2023
application-domain	[]	['engineering']
benchmark-scale	[]	['large-scale', 'multi-node', 'single-node', 'sub-node']
communication	['mpi']	['mpi']
communication-performance-characteristics	[]	['network-collectives', 'network-latency-bound']
compute-performance-characteristics	[]	['high-branching', 'mixed-precision']
math-libraries	[]	['hypre']
memory-access-characteristics	[]	['high-memory-bandwidth', 'irregular-memory-access', 'large-memory-footprint', 'regular-m
mesh-representation	[]	['block-structured-grid']
method-type	['compiler-transformation']	['solver', 'sparse-linear-algebra']
programming-language	['c', 'c++']	['c']
programming-model	[]	['openmp', 'cuda', 'rocm']
instrumented-caliper	False	True
scaling-experiments	[]	['strong', 'weak']

Experiments in Benchpark: July 2025



- HPL, HPCG
- 4 microbenchmarks
- 3 MPI benchmarks
- 8 US
- 1 Europe
- 2 Japan

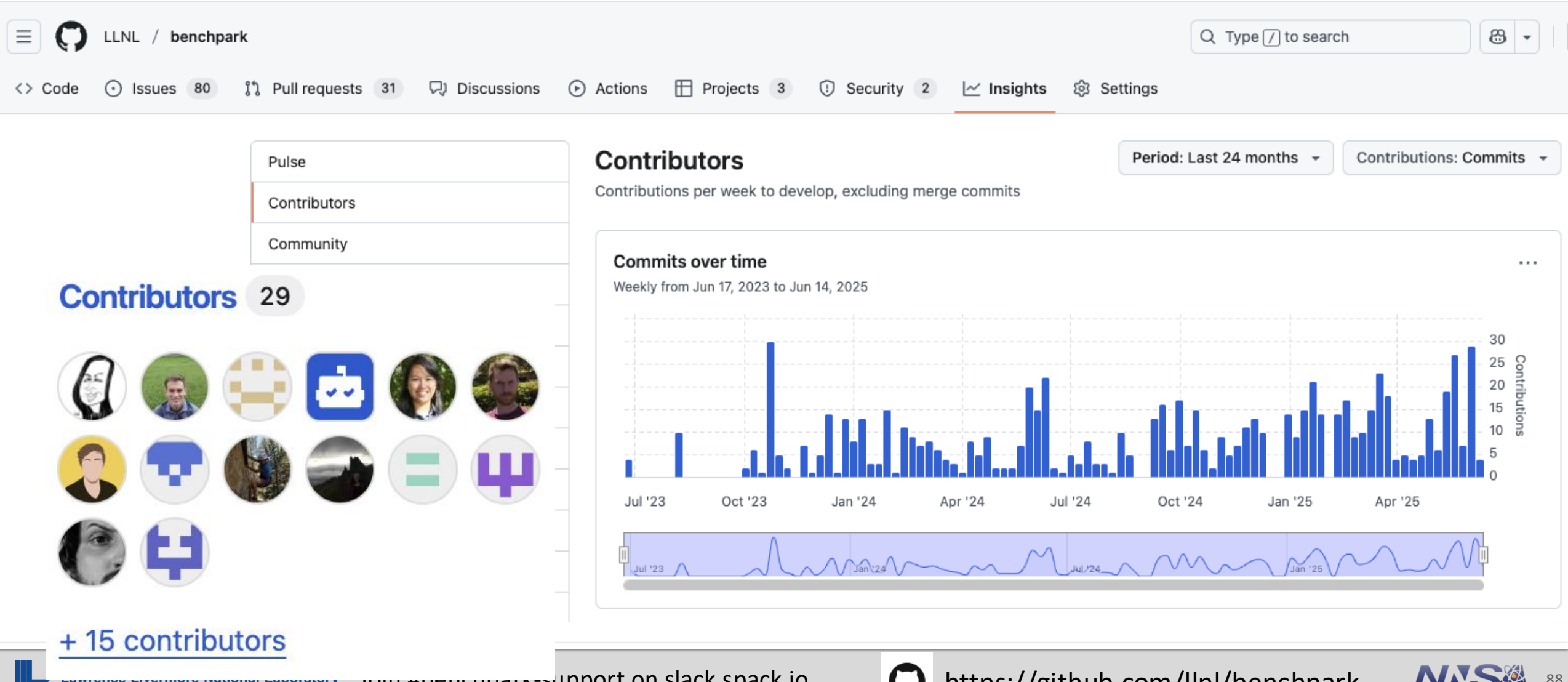
`benchpark analyze` for generating pre-defined analysis charts




```
main
├─ Generate
│  ├─ MPI_Allreduce
│  ├─ MPI_Comm_split
│  └─ MPI_Scan
├─ MPI_Allreduce
├─ MPI_Bcast
├─ MPI_Comm_dup
├─ MPI_Comm_free
├─ MPI_Comm_split
├─ MPI_Finalize
├─ MPI_Finalized
├─ MPI_Gather
├─ MPI_Get_library_version
├─ MPI_Initialized
└─ Solve
   └─ solve
      ├─ LPlusTimes
      ├─ LTimes
      ├─ Population
      │  └─ MPI_Allreduce
      ├─ Scattering
      ├─ Source
      └─ SweepSolver
         ├─ MPI_Irecv
         ├─ MPI_Isend
         ├─ MPI_Testany
         ├─ MPI_Waitall
         └─ SweepSubdomain
```

See https://software.llnl.gov/benchpark/benchpark-analyze.html

Contributions from 11 orgs (60% non-LLNL)



Benchpark codifies benchmarking steps

- Benchpark does **not** replace benchmark source, build system, or Spack package
- Benchpark manages benchmark **experiments** and how they map onto **systems** with specified (or default)
  - Compilers
  - MPI/comm libraries
  - Math libraries
- Start running benchmark experiments on your system with just a few commands

```
git clone git@github.com:LLNL/benchpark.git
```

```
benchpark list systems
benchpark system init --dest=elcap llnl-elcapitan
```

```
benchpark list experiments
benchpark experiment init dest=amg
                amg2023 +rocm scaling=strong
```

```
benchpark setup amg elcap workspace
ramble workspace setup
```

```
ramble on
```

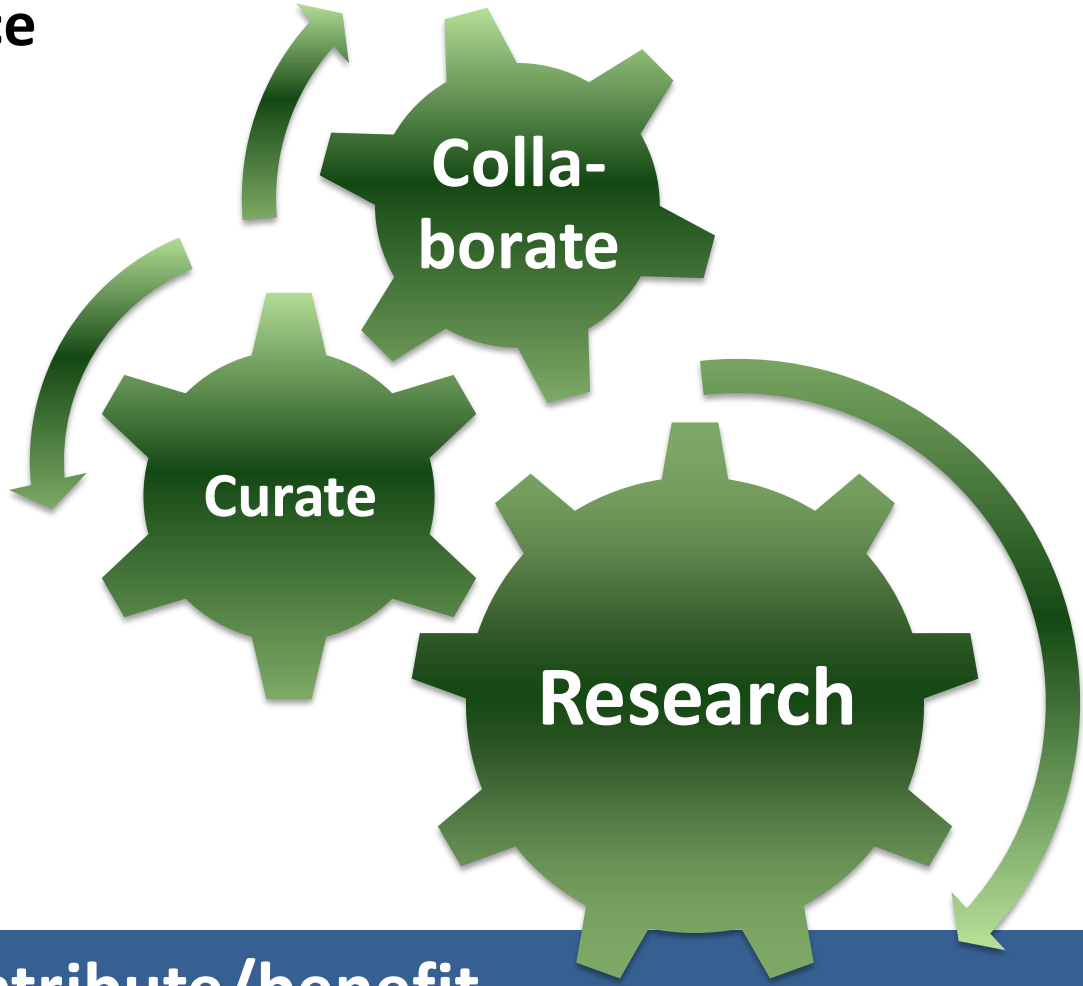
Who can use Benchpark

People who want to use or distribute benchmarks for HPC!

1. **End Users of HPC Benchmarks**
   - Install, run, analyze performance of HPC benchmarks
2. **Benchmark Developers**
   - People who want to share their benchmarks
3. **Procurement teams at HPC Centers**
   - Curate workload representation, evaluate and monitor system progress
4. **HPC Vendors**
   - Understand the curated workload of HPC centers, propose systems

Catalogued library of *working* benchmarks

- **Enables exploration of large configuration space**
  - —Architectural
  - —Software stack
  - —Temporal
- What architecture and system configuration is best for *my benchmark*?
- On *my system*, is OpenMPI good enough?
- What purpose will Benchpark help you address?



Building a community to contribute/benefit

Benchpark: Open collaborative repository for *reproducible* specifications of HPC benchmarks

Infrastructure-as-code benchmark specification enables **reproducibility, replicability, and automation**

- HPC Systems
- HPC Experiments

Specify	Run	Reproduce	Replicate	Maintain	Record
• How to build and run benchmarks on a system	• Run an experiment on a system	• Re-run an experiment on a system	• Run an experiment on a new system	• CI: Run experiment on HPC systems	• Perf.measu-rements + *full spec* of experiment

- Tagging, keywords for publications
- Performance metrics, metrics of usefulness
- Dashboards: Archive specs+results

- CI pipelines on PRs from GitHub at data centers across the world and in the cloud

*Olga Pearce et al, Continuous Benchmarking, HPCTests SC|23
*Olga Pearce et al, Repeat, Replicate, Reproduce, ACM REP 2025

Full specification enables reproducibility, replicability, and automation

Benchpark roadmap and community engagement

Future directions:

- Suite curation: Reproducible specification of an entire suite
- Tagging: Keywords for publications, finding benchmarks
- Metrics: Performance, usefulness
- Dashboards: Archive+share *specs*+results, Slices in configuration space
- CI pipelines at data centers across the world and in the cloud

Community Engagement:

- Co-design / vendors on board, but incentive for app teams? (carrot or stick?)
- Who owns which parts of the specification and approves changes?
- Who finances R&D and maintenance?
- ROI for the people working on it? → think about post docs, researchers, etc.

Collaboration, reproducibility, fully specified public results

Benchpark Tutorial Materials

Find these slides and associated scripts here:

software.llnl.gov/benchpark/basic-tutorial.html

We also have a channel on Spack slack.
You can join here:

slack.spack.io

Join the **#benchpark-support** channel!

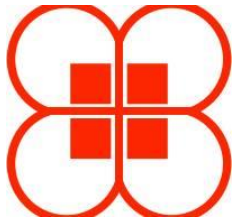
You can continue to ask questions here after the conference has ended.

Tutorial Instances: http://bit.ly/4kGQDlc

- We have an AWS instance for the hands-on component of this tutorial
- The instance provides:
  - Pre-installed Thicket, Caliper, and Benchpark and required dependencies
  - Caliper source code demos
  - Thicket Jupyter notebooks and datasets for performance analysis

When logging in to the instance:

- Please use a unique username to avoid resource allocation conflicts
  - First initial followed by last name (e.g., jdoe)
- PW: hpctutorial25



Join us after HPDC!

- All tutorial materials
  - software.llnl.gov/Caliper
  - thicket.readthedocs.io/en/latest/tutorial_materials.html
  - software.llnl.gov/benchpark/basic-tutorial.html
- Connect with us on Spack slack

  #benchpark-support

slack.spack.io
#benchpark-support

- We want your feedback!

Thank you!

★ Star us on GitHub!

github.com/llnl/caliper

github.com/llnl/thicket

github.com/llnl/benchpark

CASC

Center for Applied
Scientific Computing