Automated Application Performance Analysis with Caliper, SPOT, and Hatchet

2022 ECP Annual Meeting Tutorial

May 2, 2022

David Boehme, Stephanie Brink, Olga Pearce

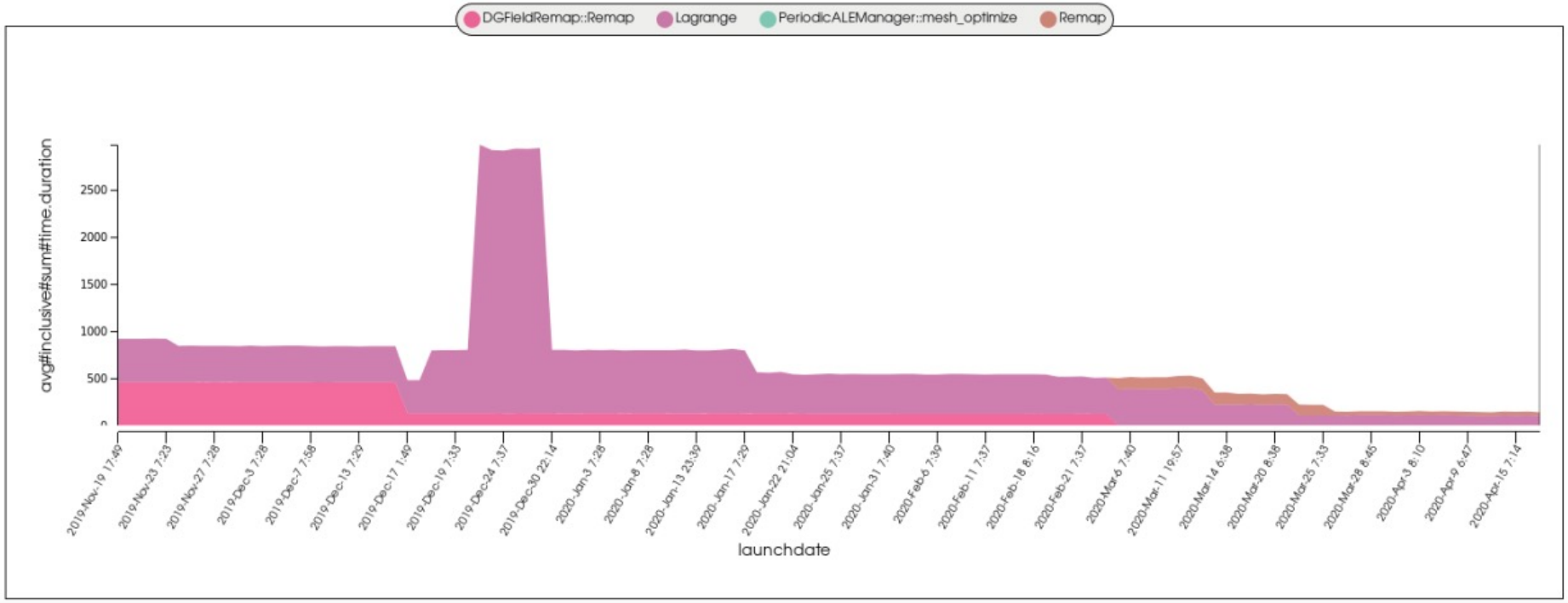
LLNL-PRES-821032

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

Building Automated Performance Analysis Workflows

Enabling performance analysis as a routine activity
for HPC software development



Nightly test performance of a large physics code over 5 months

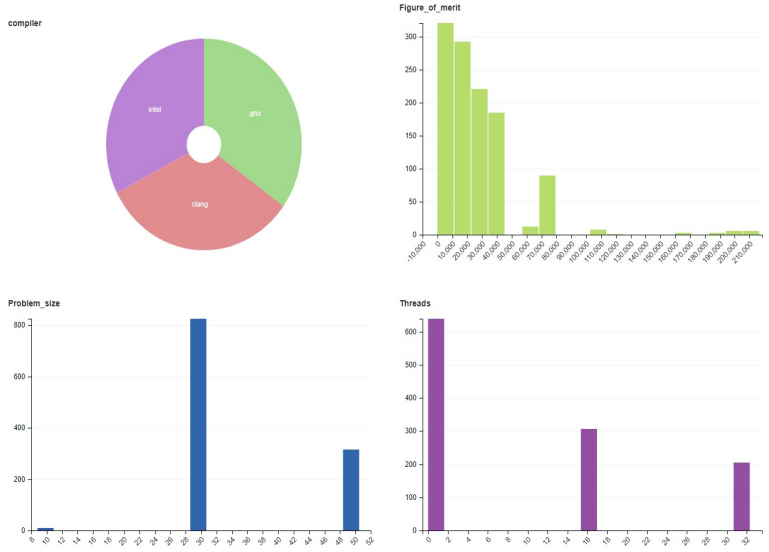
Building Automated Performance Analysis Workflows

Caliper

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

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

Caliper:
Instrumentation and Profiling



SPOT: Analysis of
large collections of 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
```

Hatchet:
Call graph analysis in Python

Caliper: A Performance Profiling Library

2022 ECP Annual Meeting: Tutorial

May 2, 2022

David Boehme
Computer Scientist

LLNL-PRES-821032
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
- Advanced profiling features
  - MPI, CUDA, ROCm, Kokkos support; call-stack sampling; hardware counters; memory profiling

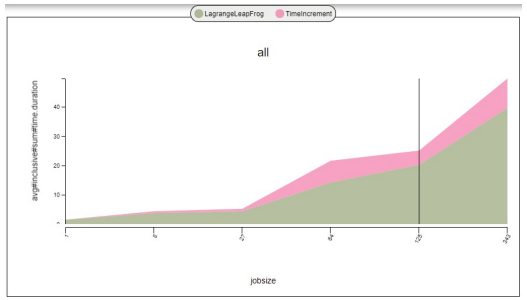
Caliper Use Cases

- 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



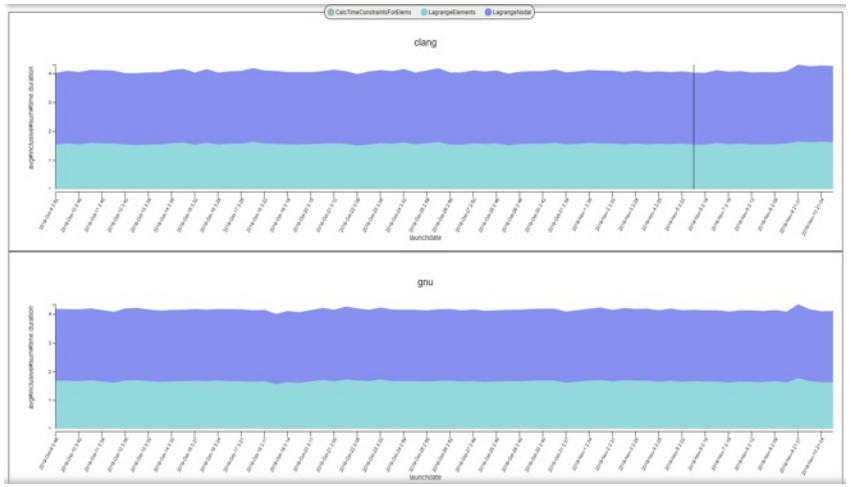
Comparing runs

```
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

Performance Analysis with Caliper, SPOT and Hatchet

"spot" config



Pre-populated Jupyter notebooks

Caliper

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

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

Caliper:
Instrumentation and Profiling

SPOT web frontend:
Analysis of
large collections of runs

hatchet-region-profile,
hatchet-sample-profile

```
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
```

Hatchet:
Call graph analysis in Python

Materials, Contact & Links

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

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


```
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
```

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

Environment variables

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

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”);
}
```

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

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 Configurations

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
callpath-sample-report	Print time spent in functions 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 the SPOT web visualization framework

Use `mpi-caliquery --help=configs` to list all built-in configs and their options

Built-In Profiling Configurations: 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 configurations 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

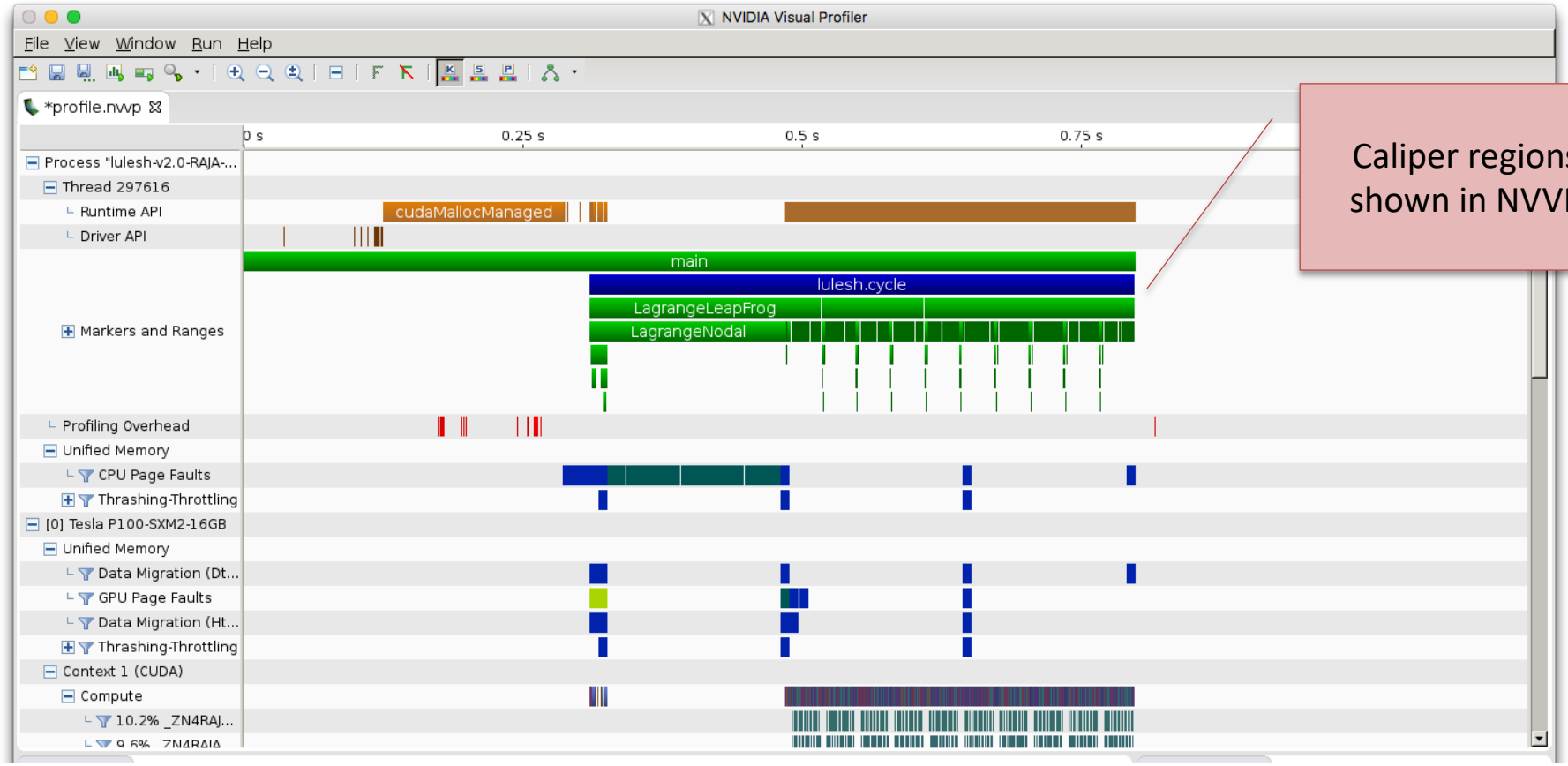
```
$ lrun -n 4 ./tea_leaf runtime-report,profile.cuda
```

```
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

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

Call Graph Analysis with the Hatchet Python Library

- Caliper records data for hatchet with `hatchet-region-profile` or `hatchet-sample-profile`

```
$ CALI_CONFIG=hatchet-sample-profile srun -n 8 ./lulesh2.0
```

Hatchet allows manipulation, computation, comparison, and visualization of call graph data

```
>>> gf = hat.GraphFrame.from_caliper_json('/Users/boehme3/Documents/Data/lulesh_8x4_callpath-sample-profile.json')
>>> gf.subgraph_sum(['time'])
>>> gf = gf.filter(lambda x: x['name'] != '__restore_rt')
>>> gf = gf.filter(lambda x: x['name'].find('_omp_fn') == -1).squash()
>>> print(gf.tree())
  __          __          __          __
 / /_  ____ _/ /_____ _/ /_  ___  / /_
/ __ \/ __ `/ __/ ___/ __ \/ _ \/ __/
/ / / / /_/ / /_/ /__/ / / /  __/ /_
/_/ /_/\__,_/\__/\___/_/ /_/\___/\__/  v1.3.0

5.850 __clone
└─ 5.850 start_thread
   └─ 5.850 gomp_thread_start
      ├─ 0.070 CalcElemVolume(dou...t*, double const*)
      ├─ 0.005 UNKNOWN 4
      ├─ 0.075 cbrt
      │  ├─ 0.000 frexp
      │  └─ 0.020 ldexp
      │     └─ 0.010 scalbn
      ├─ 0.005 gomp_barrier_wait
      ├─ 2.545 gomp_barrier_wait_end
      ├─ 0.605 gomp_team_barrier_wait_end
```

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

```
$ ./examples/apps/cxx-example -P 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
```

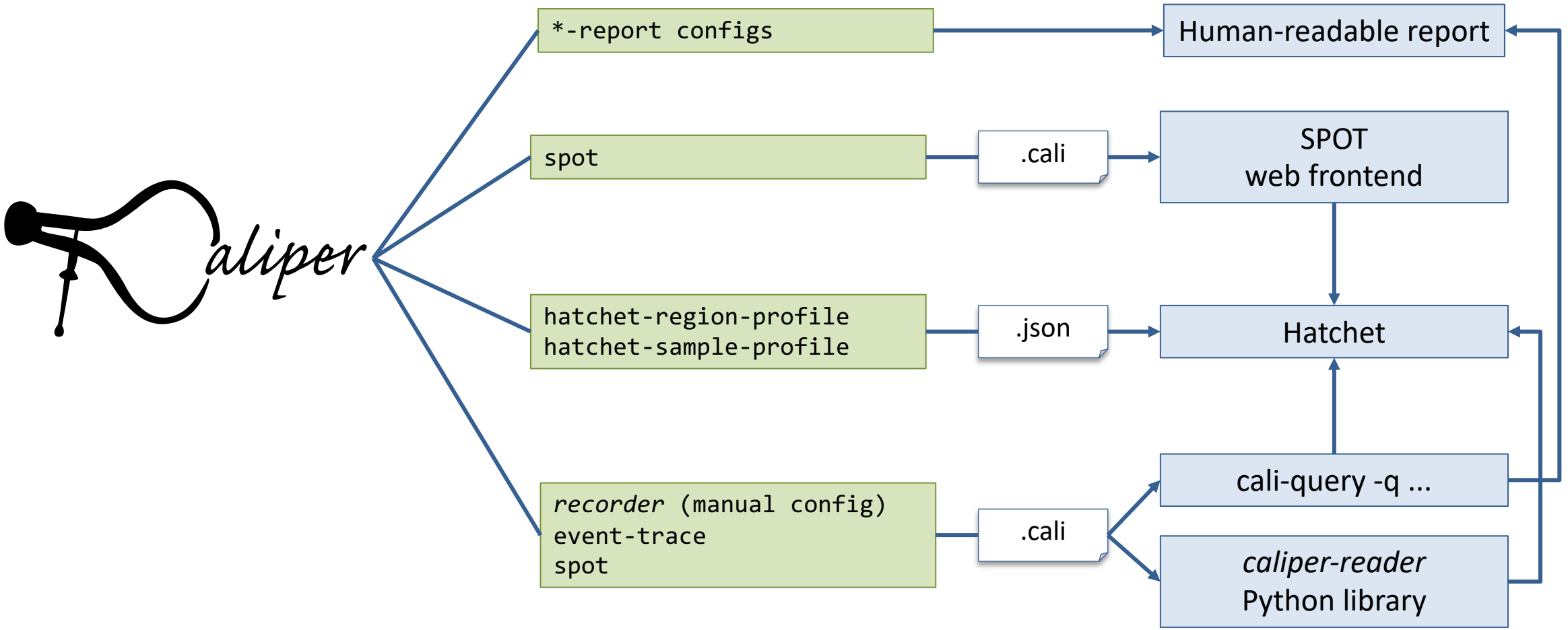
caliper.config

```
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

Caliper Output Formats and Processing Workflows



Recording Data for SPOT

Recording Data for SPOT with Caliper and Adiak

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

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

Region instrumentation

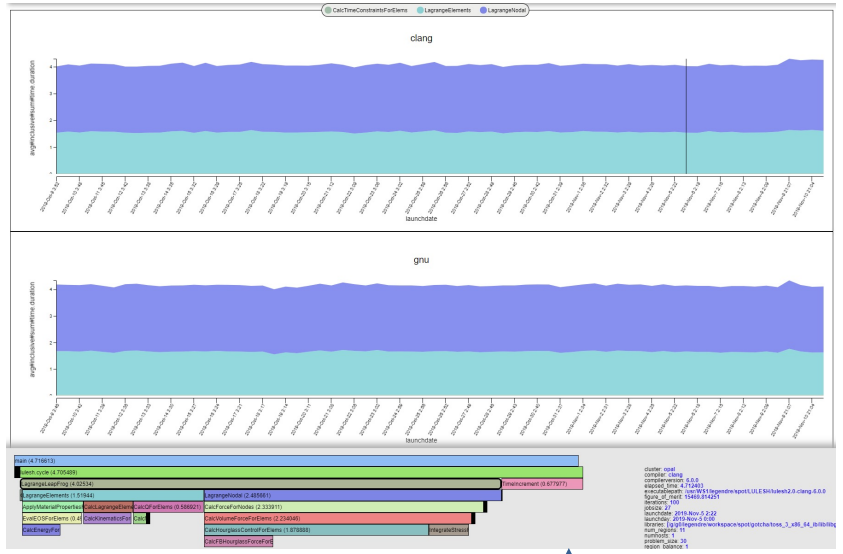
```
adiak::clustername();
adiak::jobsize();

adiak::value("iterations", opts.its);
adiak::value("problem_size", opts.nx);
adiak::value("num_regions", opts.numReg);
```

Metadata collection [adiak]

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

Caliper configuration



SPOT Web GUI

Profile data (.cali)

Experiment directory

```
$ ./app -P spot
```

Run program with the “spot” profiling config

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.

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 comes with built-in functions to collect common environment metadata
- SPOT requires at least `launchdate`

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_nameval("myvec",    adiak_general, NULL, "[%d]", ints, 4);

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

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

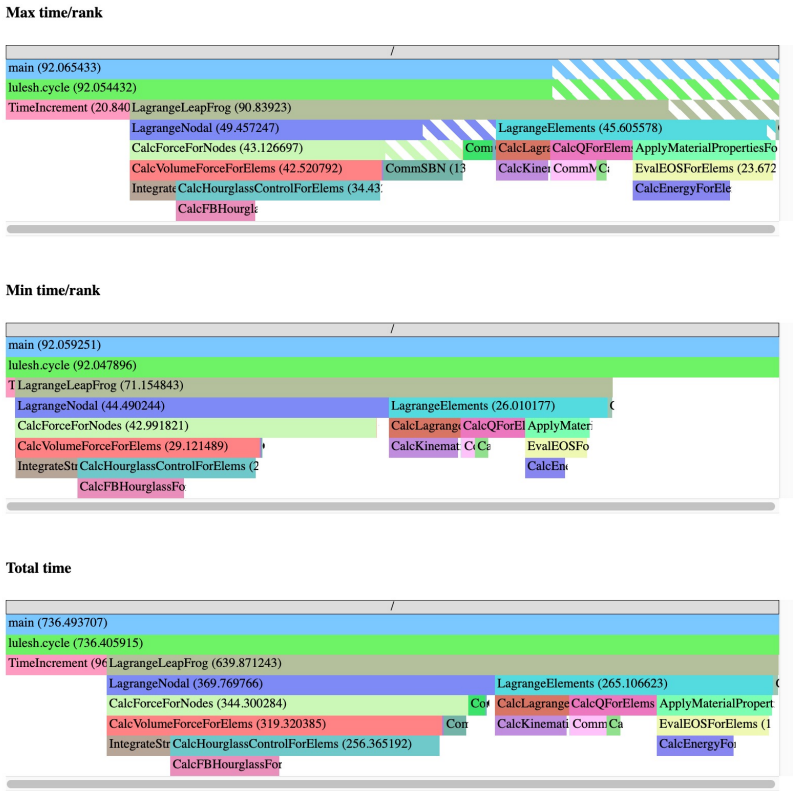
- In C, `adiak_nameval()` uses printf()-style descriptors to determine data types

The spot config: Region Profiling

```
$ CALI_CONFIG=spot,profile.mpi ./lulesh2.0
```

```
$ ls *.cali
210304-17175150010.cali
```

- “spot” records and aggregates time spent in instrumented regions, like runtime-report
- Supports many profiling options (e.g., MPI function profiling)
- Collect profiling output (.cali files) in a directory for analysis in SPOT



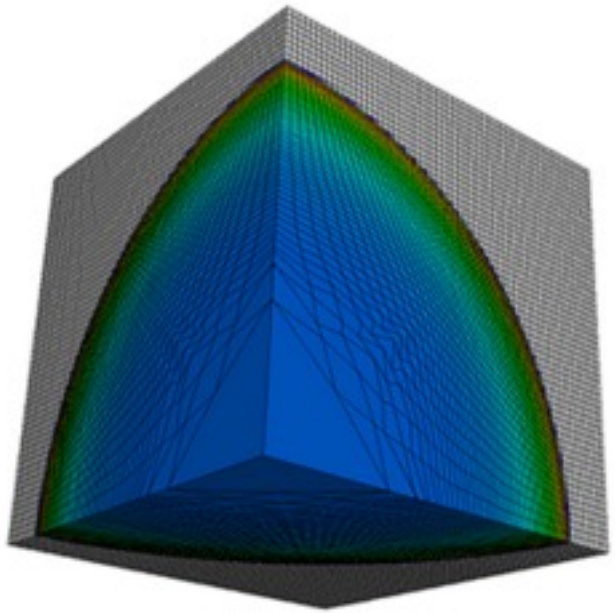
SPOT region profile flame graphs

Example:
Caliper and Adiak in LULESH

Modified LULESH Proxy App with Caliper and Adiak Support

https://github.com/daboehme/LULESH/tree/adiak-caliper-support

```
$ mpirun -n 8 ./lulesh2.0 -P runtime-report,profile.mpi
```



```
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
[...]
```

LULESH Example: Region Annotations

```
void CalcLagrangeElements(Domain& domain)
{
  CALI_CXX_MARK_FUNCTION;
  ...
```

Function annotation in LULESH

- Top-level functions provide meaningful basis for performance analysis in LULESH
- Annotated 17 out of 39 computational functions and 5 communication functions

LULESH Example: Main Loop Annotation

```
CALI_CXX_MARK_LOOP_BEGIN(cycleloop, "lulesh.cycle");

while((locDom->time() < locDom->stoptime()) && (locDom->cycle() < opts.its)) {
  CALI_CXX_MARK_LOOP_ITERATION(cycleloop, locDom->cycle());

  // ...
}

CALI_CXX_MARK_LOOP_END(cycleloop);
```

Main loop annotation in LULESH

- Annotation of the main time-stepping loop and iterations for loop profiling

LULESH Example: Initialization and ConfigManager

```
adiak::init(adiak_comm_p);

cali::ConfigManager mgr;
if (!opts.caliperConfig.empty())
   mgr.add(opts.caliperConfig.c_str());

if (mgr.error())
   std::cerr << "Caliper config parse error: " << mgr.error_msg() << std::endl;

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

ConfigManager setup in LULESH

- Profiling control via ConfigManager API
- Modified LULESH command-line parsing code to read Caliper config string (not shown)

LULESH Example: Recording Metadata With Adiak

```
void RecordGlobals(const cmdLineOpts& opts, int num_threads)
{
  adiak::user();
  adiak::launchdate();
  adiak::executablepath();
  adiak::libraries();
  adiak::cmdline();
  adiak::clustername();
  adiak::jobsize();

  adiak::value("threads", num_threads);
  adiak::value("iterations", opts.its);
  adiak::value("problem_size", opts.nx);
  adiak::value("num_regions", opts.numReg);
  adiak::value("region_cost", opts.cost);
  adiak::value("region_balance", opts.balance);
}
```

Recording environment and LULESH config

```
void VerifyAndWriteFinalOutput(...)
{
  // ...
  adiak::value("elapsed_time", elapsed_time);
  adiak::value("figure_of_merit", 1000.0/grindTime2);
}
```

Recording global performance metrics at program end

- Adiak calls record environment info, LULESH configuration options, and global performance metrics

LULESH Example: Build System Modifications

```
find_package(caliper REQUIRED)
find_package(adiak REQUIRED)

# ...

add_executable(${LULESH_EXEC} ${LULESH_SOURCES})

target_include_directories(${LULESH_EXEC} PRIVATE ${caliper_INCLUDE_DIR} ${adiak_INCLUDE_DIRS})
target_link_libraries(${LULESH_EXEC} caliper adiak)
```

CMakeLists.txt

- Using caliper and adiak `find_package()` support in LULESH CMake script

SPOT Tutorial Slides

2022 ECP Annual Meeting

May 2, 2022

Matthew LeGendre

LLNL-PRES-821032

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

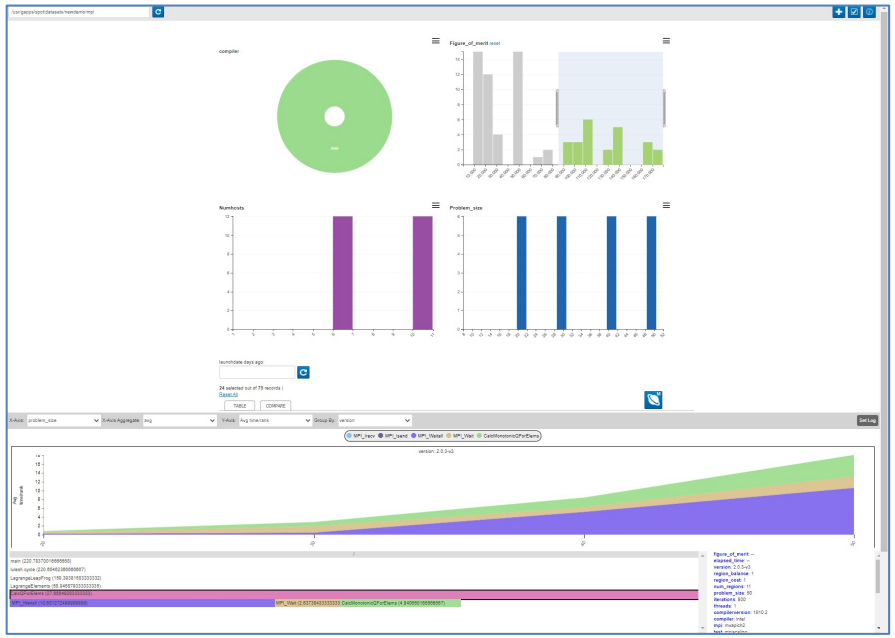
Getting SPOT

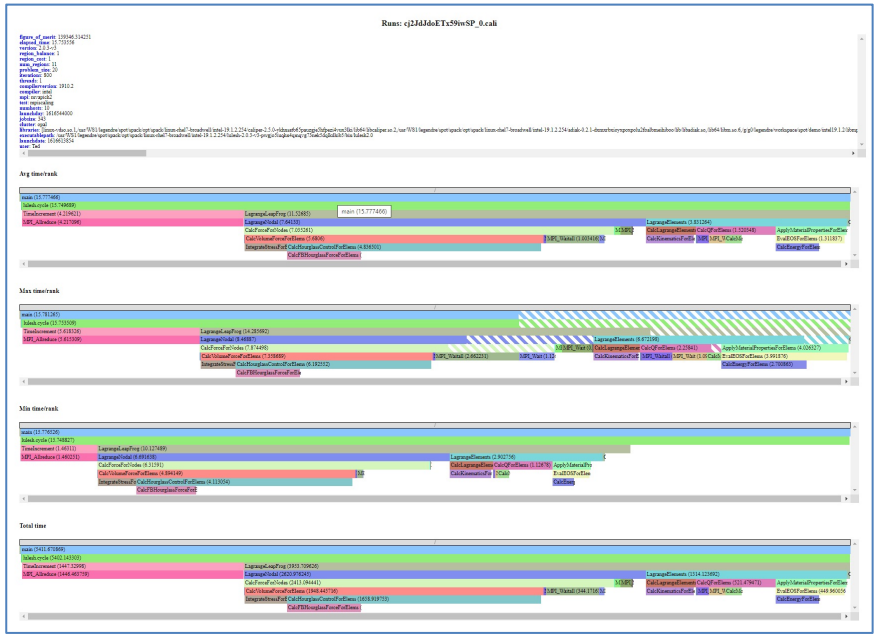
- LLNL installation: https://lc.llnl.gov/spot2

- Docker container: https://github.com/LLNL/spot2_container

SPOT: Software Performance and Analysis Tracking

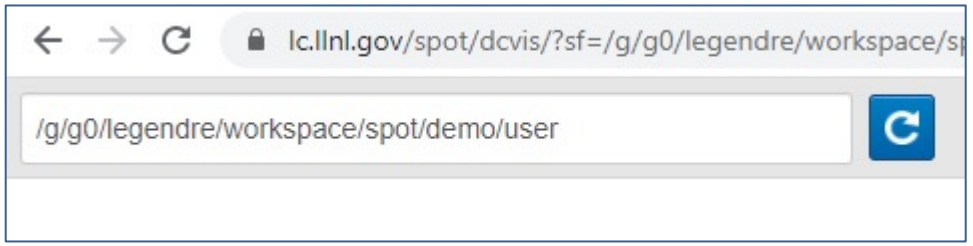
- SPOT is visualization for **collections of performance profiles**. Common uses cases are:
  - Performance comparison of nightly tests. Look for performance regressions.
  - Performance tracking of developer changes. Run an MPI scaling study.
  - Collect performance profiles from users. Understand how users run and hit performance issues.





SPOT's Workflow

1. Point SPOT webpage at directory containing *.cali files.

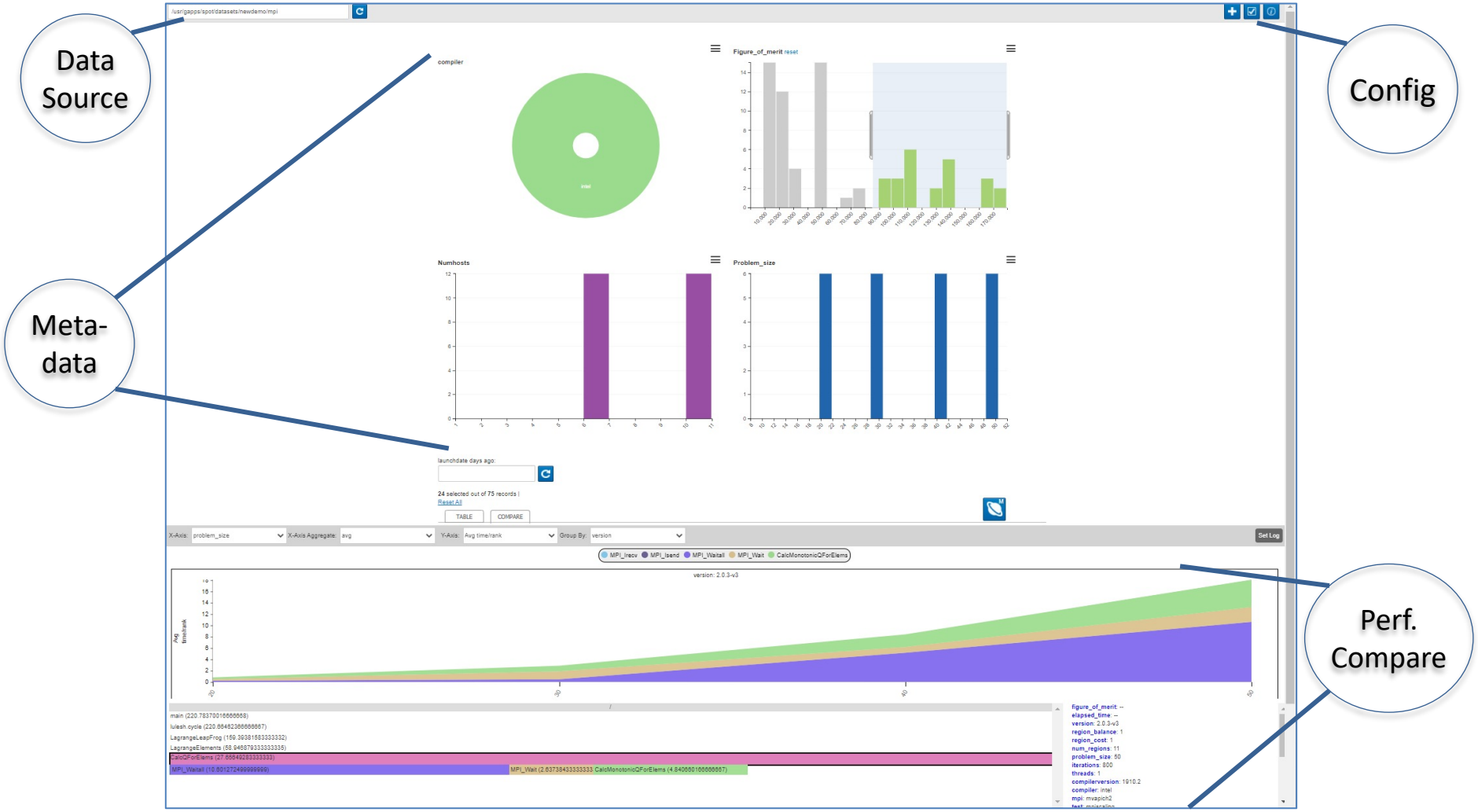


2. Select "interesting" sets of runs to analyze
   - Don't compare performance of 1d test run. vs 3d multi-physics run.

3. Run common analysis with SPOT.
   - i.e., Graph history of performance changes
   - i.e., View performance context trees

4. Run advanced analysis with Hatchet.

SPOT's Landing Page



SPOT's Landing Page

Table View

SPOT is Based on CrossFilter

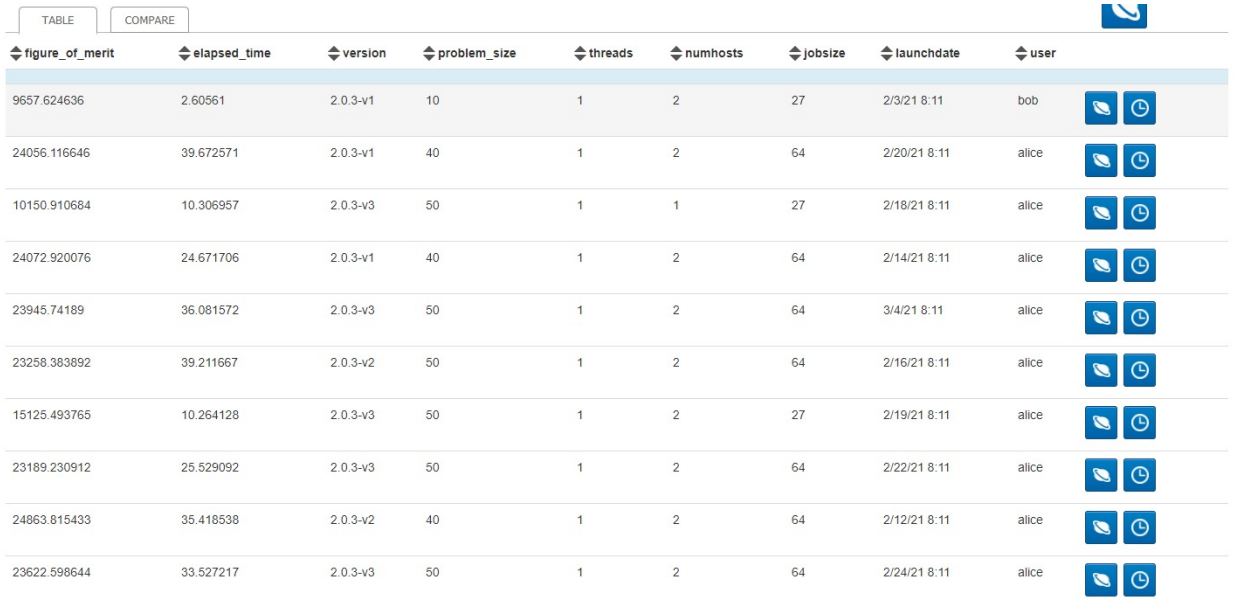


TABLE COMPARE

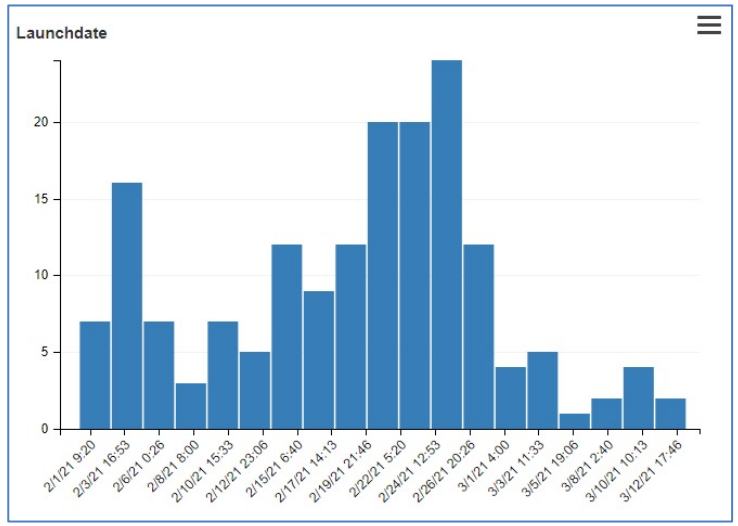
figure_of_merit	elapsed_time	version	problem_size	threads	numhosts	jobsize	launchdate	user
9657.624636	2.60561	2.0.3-v1	10	1	2	27	2/3/21 8:11	bob
24056.116646	39.672571	2.0.3-v1	40	1	2	64	2/20/21 8:11	alice
10150.910684	10.306957	2.0.3-v3	50	1	1	27	2/18/21 8:11	alice
24072.920076	24.671706	2.0.3-v1	40	1	2	64	2/14/21 8:11	alice
23945.74189	36.081572	2.0.3-v3	50	1	2	64	3/4/21 8:11	alice
23258.383892	39.211667	2.0.3-v2	50	1	2	64	2/16/21 8:11	alice
15125.493765	10.264128	2.0.3-v3	50	1	2	27	2/19/21 8:11	alice
23189.230912	25.529092	2.0.3-v3	50	1	2	64	2/22/21 8:11	alice
24863.815433	35.418538	2.0.3-v2	40	1	2	64	2/12/21 8:11	alice
23622.598644	33.527217	2.0.3-v3	50	1	2	64	2/24/21 8:11	alice

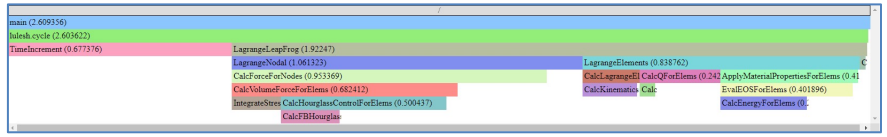
SPOT's Crossfilter: https://dc-js.github.io/dc.js/
D3-Based Crossfilter: https://square.github.io/crossfilter/

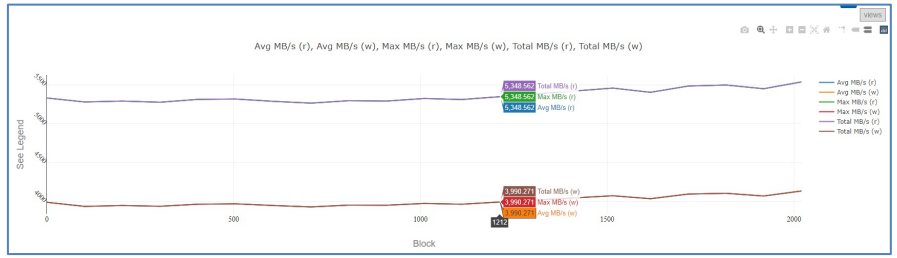
- CrossFilter is good at finding correlations in data.
- Crossfilter helps you manage/slice your data to make interesting comparisons.

Each App Run has Two Types of Data

- **Metadata**
  - Name/value data about job.
    - User, host, FOM, input parameters, …
  - Displayed as histograms
- **Performance Data**
  - Can be different metrics.
    - Avg Time/Rank, Max Time/Rank, bandwidth, …
  - Displayed as a flame graph or timeseries.

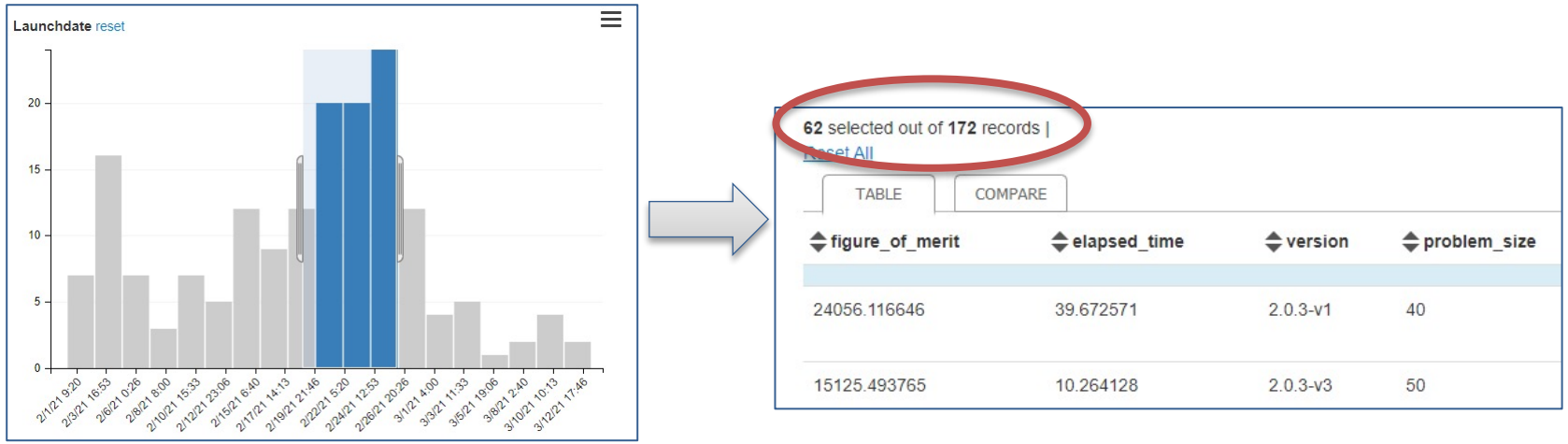






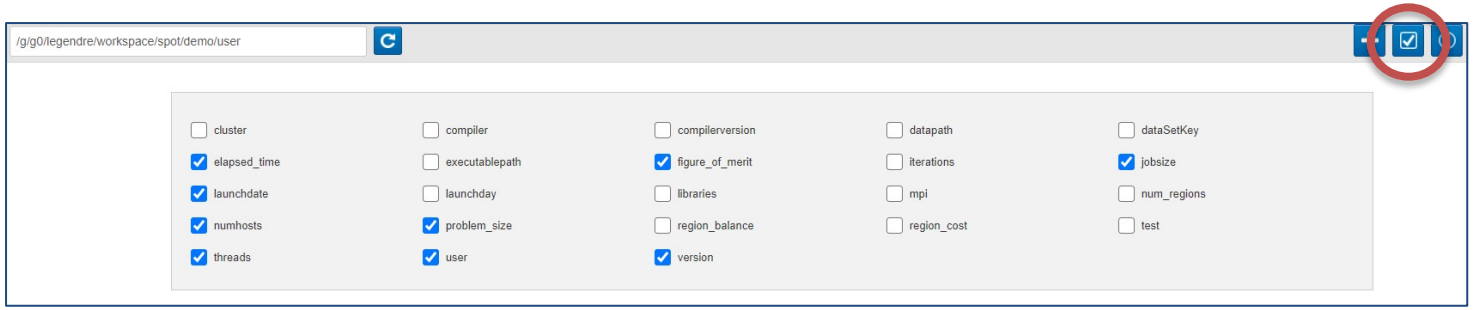
Metadata Can be Filtered

- Select regions of metadata histograms to filter.
  - Filtering in one histogram updates all histograms range.
    - E.g., Filtering Launchdate to ‘Feb 19-Feb26’ will update the ‘Users’ histogram to only show users who ran between those dates.
  - While filtered, comparisons and Jupyter notebooks will operate on the filtered data.

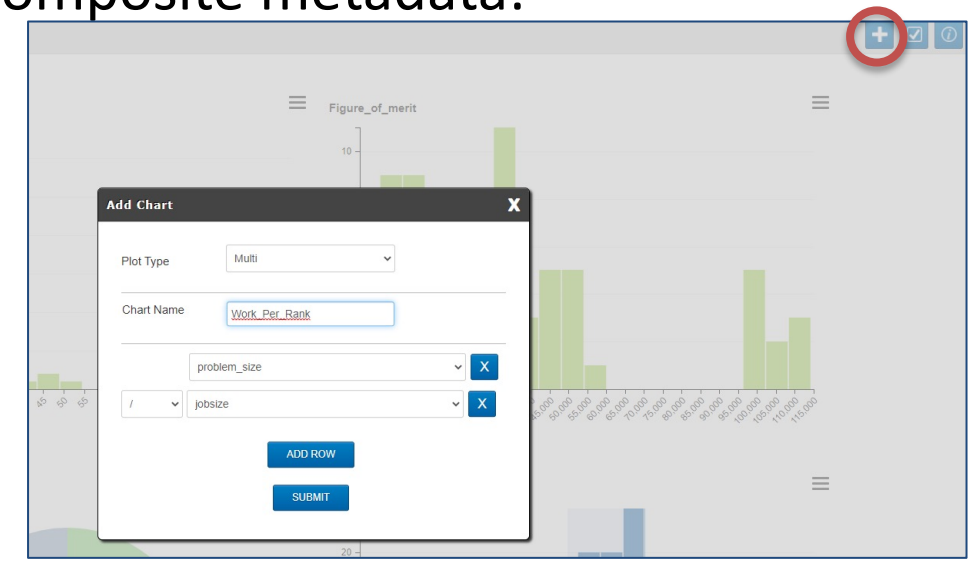


Metadata is Configurable

- Can show/hide metadata:

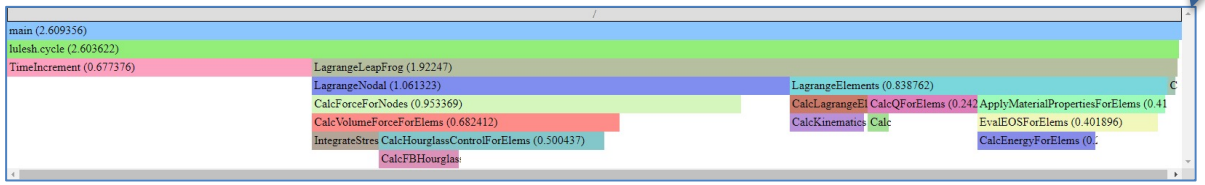


- Can create composite metadata:

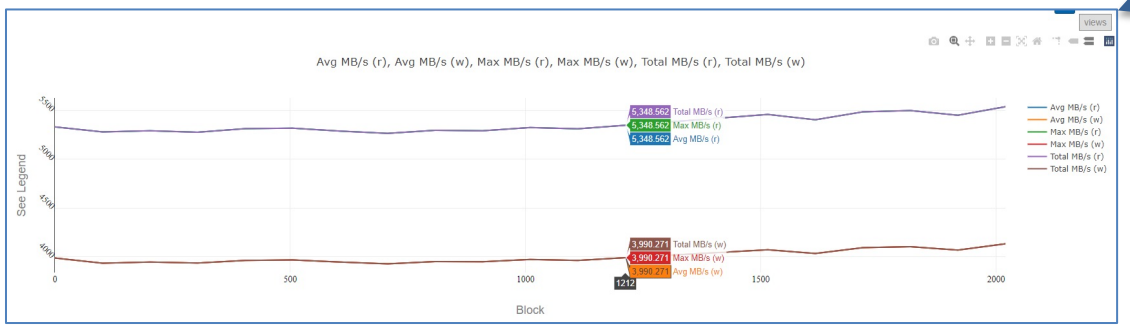


Performance Data can be Visualized Per-Run

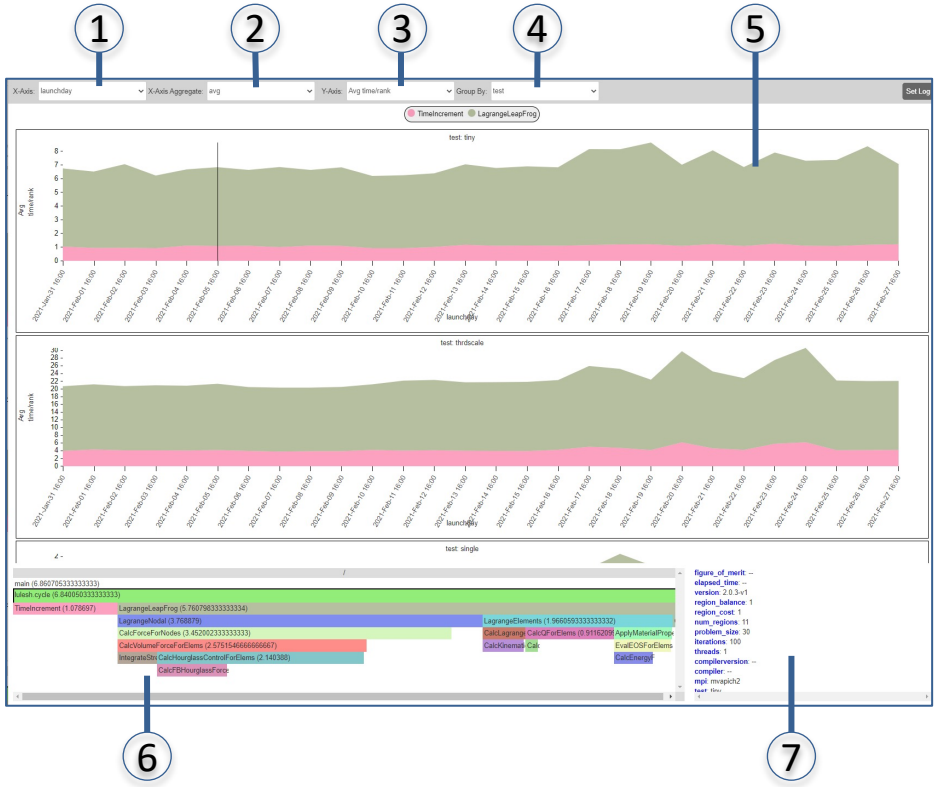
- FlameGraph: performance against code



- Timeseries: performance against time



Performance Data can be Compared Across Runs



1. Metadata value on x-axis
2. Aggregation (min/max/avg…) if multiple runs at same point.
3. Metric to graph on y-axis.
4. Grouping metadata. Each unique value is a graph.
5. Stacked performance graph. Click to drill-down.
6. Flame graph for runs currently under cursor.
7. Metadata for runs currently under cursor.

Live Demo

CASC

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.