Caliper: A Performance Profiling Library

2021 ECP Annual Meeting: Tutorial

ECP
EXASCALE COMPUTING PROJECT

CASC
Center for Applied Scientific Computing

April 12, 2021

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, Kokkos support; call-stack sampling; hardware counters; memory profiling

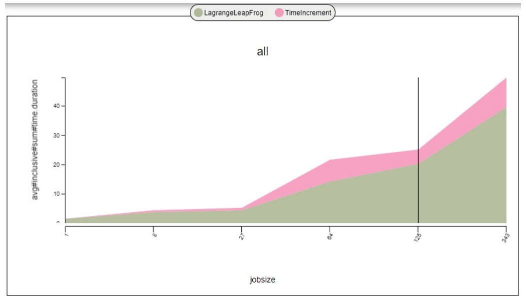
Caliper Use Cases

- 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

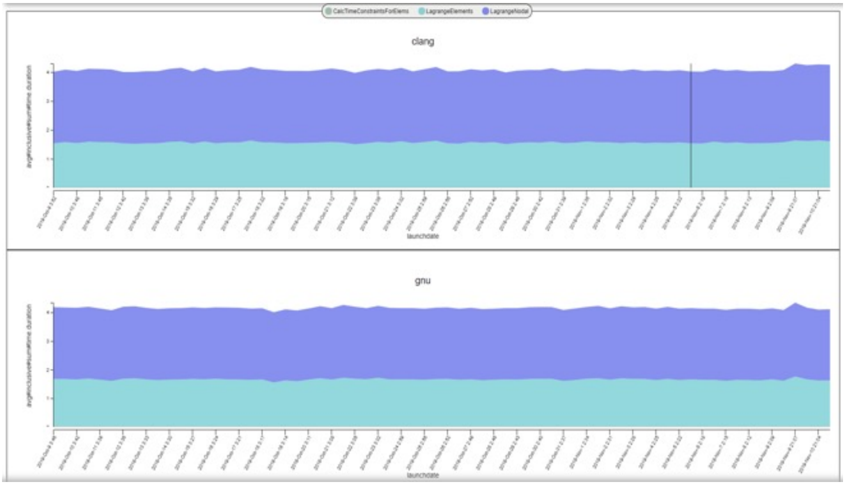
Performance Analysis with Caliper, SPOT and Hatchet

“spot” config

```
#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

Pre-populated Jupyter notebooks

```
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

Contact & Links

- 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

Caliper Step-by-Step

1. Install Caliper
2. Add Caliper to target code as library dependency
3. Instrument source-code regions
4. [optional] Add program metadata annotations
5. [optional] Add ConfigManager profiling control API
6. Run program with profiling configuration

Building and Linking the Caliper Library

- Install Caliper manually (CMake build system) or with the spack package manager

```
$ spack install caliper
```

- Link libcaliper.so

```
$ g++ -o app $(OBJECTS) –L$(CALIPER_DIR)/lib64 -lcaliper
```

- CMake `find_package()` support is available

```
find_package(caliper)
add_executable(myapp ${SOURCES})
target_include_directories(myapp ${caliper_INCLUDE_DIR})
target_link_libraries(myapp PRIVATE caliper)
```

```
$ cmake –Dcaliper_DIR=<caliper installation dir>/share/cmake/caliper
```

Recommended CMake Build Options

```
$ spack install caliper +adiak+cuda+papi+mpi+libdw+libunwind+sampler
```

CMake Flags	Effect
-DWITH_ADIAK=On -Dadiak_DIR=<adiak install location>/lib/cmake/adiak	Program metadata recording with the Adiak library. Required for SPOT.
-DWITH_MPI=On	Enables report aggregation and MPI function profiling. Required for SPOT and loop-report.
-DWITH_PAPI=On -DPAPI_PREFIX=<papi install location>	Enables PAPI hardware counter recording.
-DWITH_SAMPLER=On -DWITH_LIBDW=On -DWITH_LIBUNWIND=On	Enables call-path sampling.
-DWITH_NVTX=On -DWITH_CUPTI=On -DCUDA_TOOLKIT_ROOT_DIR=<cuda location>	Enables CUDA profiling and annotation forwarding for NVidia NVProf/NSight tools.

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: 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

Region Profiling: Best Practices

- Instrument high-level program subdivisions (kernels, phases, ...)
- Choose good names
- Add instrumentation incrementally

Built-In Profiling Configurations

```
$ CALI_CONFIG=runtime-report ./examples/apps/cxx-example
```

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

runtime-report measures and prints time in annotated regions

```
$ CALI_CONFIG=hatchet-region-profile ./examples/apps/cxx-example
$ ls *.json
$ region_profile.json
```

hatchet-region-profile records per-process time profile of annotated regions for analysis with hatchet

- Built-in profiling configurations cover common performance analysis use cases

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

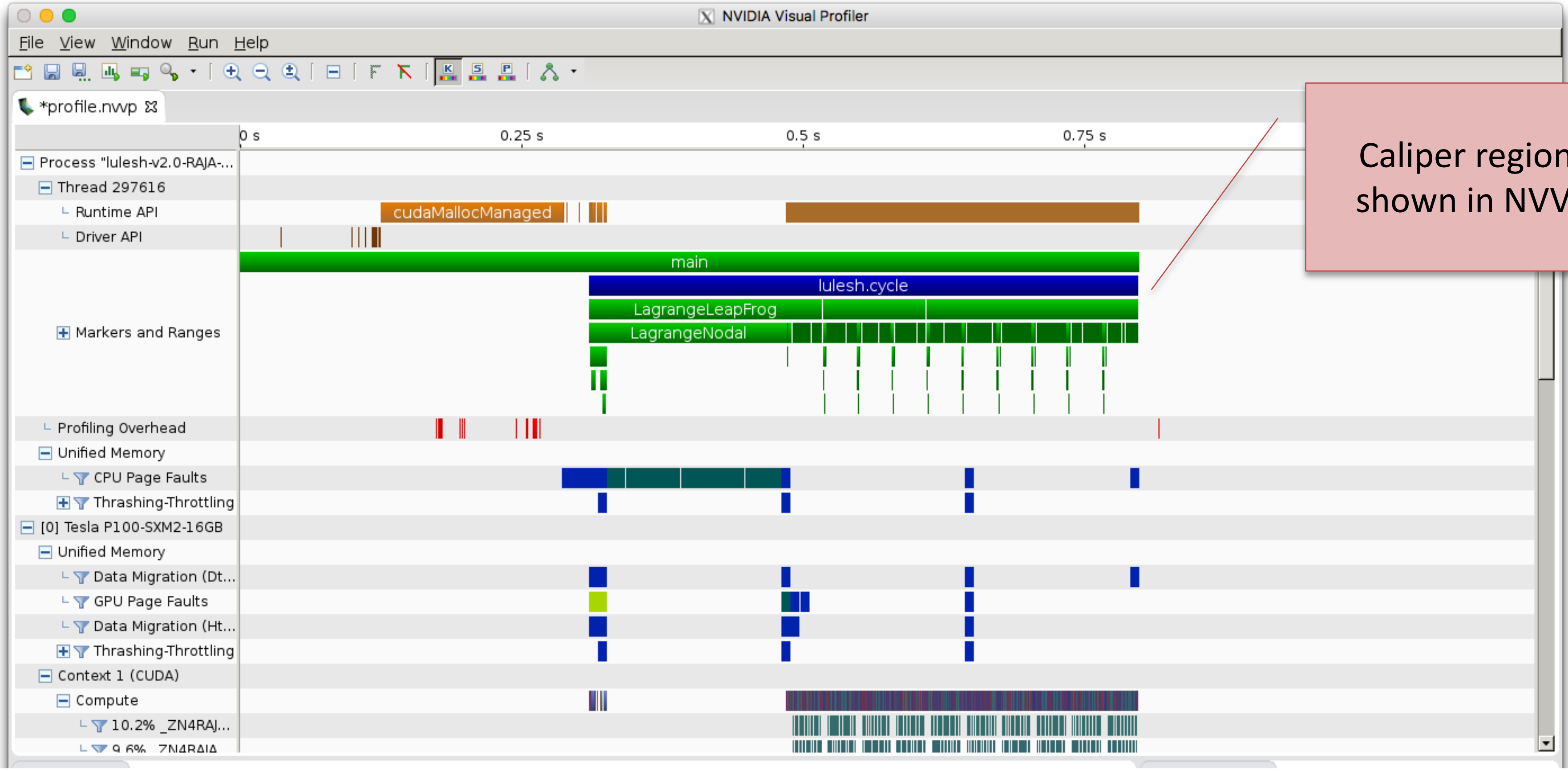
```
$ lrun -n 4 ./tea_leaf runtime-report,profile.cuda
```

```
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

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

ConfigManager vs. CALI_CONFIG vs. Manual Configuration

- Use ConfigManager or CALI_CONFIG for Caliper's built-in measurement configurations

```
$ CALI_CONFIG=runtime-report ./examples/apps/cxx-example
```

- ConfigManager allows use of program-specific inputs (e.g., command-line arguments)

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

- You can create custom measurement and report configurations manually

```
$ export CALI_SERVICES_ENABLE=aggregate,event,mpi,mpireport,timestamp
$ export CALI_MPIREPORT_CONFIG="SELECT
      min(sum#time.duration) as \"Min Time/rank\",
      max(sum#time.duration) as \"Max Time/rank\",
      avg(sum#time.duration) as \"Avg Time/rank\"
    GROUP BY prop:nested FORMAT tree"
$ ./examples/apps/cxx-example
```

Loop Profiling: Marking Loops and Loop Iterations

C++

```
CALI_MARK_CXX_MARK_LOOP_BEGIN(mainloop_id, ”mainloop”);

for (int i = 0; i < N; ++i) {
  CALI_CXX_MARK_LOOP_ITERATION(mainloop_id, i);
  // ...
}

CALI_CXX_MARK_LOOP_END(mainloop_id);
```

- Mark loops and iterations to support loop profiling options
- Generally, it’s best to only annotate outer loops (e.g., the main time step loop)

Loop Profiling: Loop and Iteration Summary

```
$ ./examples/apps/cxx-example 5000 -P loop-report
```

```
Loop summary:
------------

Loop     Iterations Time (s) Iter/s (min) Iter/s (max) Iter/s (avg)
mainloop       5000 6.815763   380.539973  2462.197671   723.821101

Iteration summary (mainloop):
-----------------

Block Iterations Time (s) Iter/s
    0       1232 0.500366 2462.197671
 1232        575 0.500723 1148.339501
 1807        447 0.500756  892.650313
 2254        377 0.501059  752.406403
 2631        333 0.501320  664.246390
 2964        301 0.501534  600.158713
 3265        277 0.500940  552.960434
 3542        256 0.502077  509.881950
[...]
```

loop-report config prints time in instrumented loops

Loop Profiling: Measurement Intervals

- Loop measurement intervals can be time or iteration based (“measure every *x* seconds” or “measure every *N* iterations”)

```
$ ./examples/apps/cxx-example 5000 -P loop-report(iteration_interval=500)
```

```
Block Iterations Time (s) Iter/s
    0        500 0.110812 4512.146699
  500        500 0.244453 2045.382957
 1000        500 0.378453 1321.168018
 1500        500 0.532856 938.339814
 2000        500 0.660435 757.076775
 2500        500 0.785368 636.644223
[...]
```

Measuring every 500 iterations

Loop Profiling: Iteration Blocks

- Output adapts to any loop length:
  Iterations are grouped into *blocks* so that only *N* blocks are shown (default: 20)

```
$ ./examples/apps/cxx-example 5000 -P loop-report(iteration_interval=500,timeseries.maxrows=3)
```

```
Block Iterations Time (s) Iter/s
               0 0.000034 0.000000
    0       2000 1.294308 1545.227257
 1666       1500 2.359132 635.827075
 3332       1500 3.484032 430.535655
```

Group iterations into three blocks

```
loop-report(iteration_interval=1,timeseries.maxrows=0)
```

Measure and show every iteration

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
```

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
```

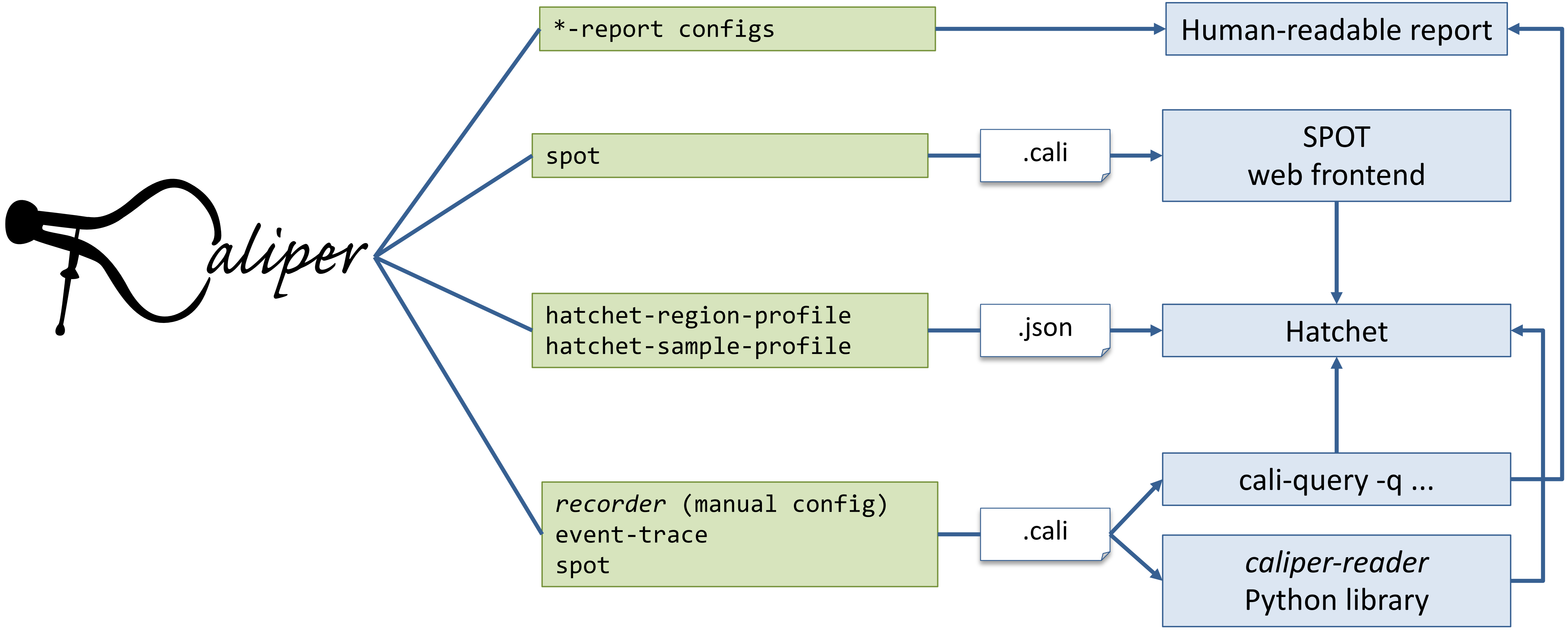
caliper.config

```
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();
```

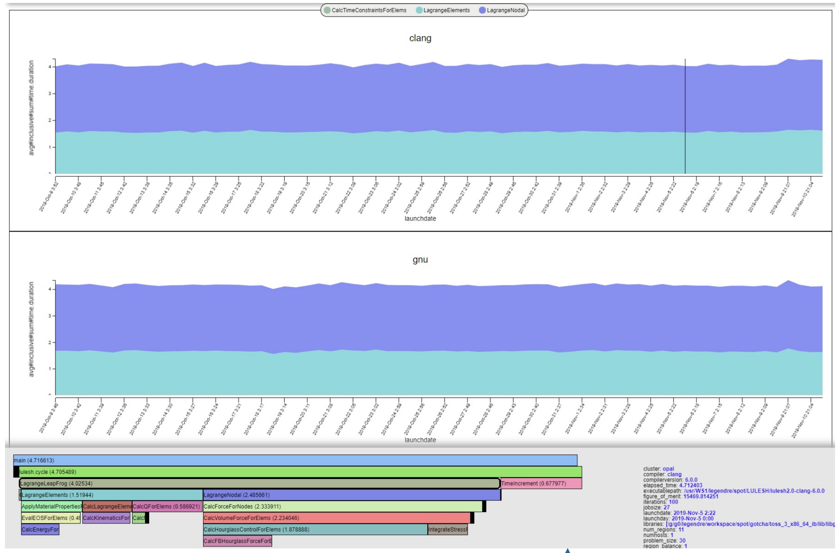
Caliper configuration

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

Run program with the “spot” profiling config

Profile data (.cali)

Experiment directory



SPOT Web GUI

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++

```cpp
#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

```
#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”);
```

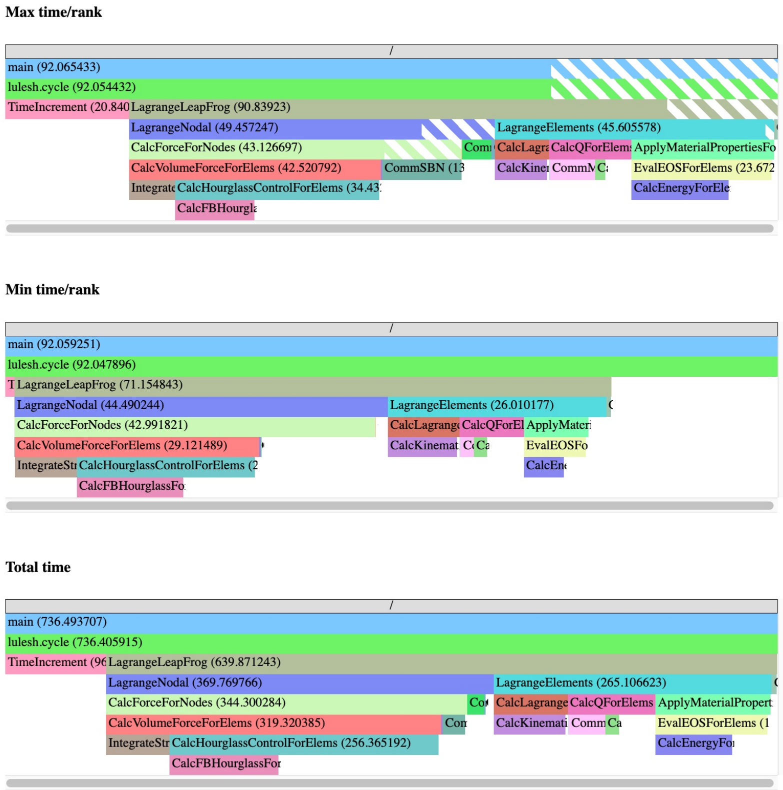
- In C, `adiak_nameval()` uses printf()-style descriptors to determine data types

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

The spot config: Loop Profiling

```
$ CALI_CONFIG=spot,timeseries=true,timeseries.metrics=mem.bandwidth ./app
```

- Enable the “timeseries” option to record loop profiles for SPOT
- Use “timeseries.metrics” to enable metric options for the loop profile

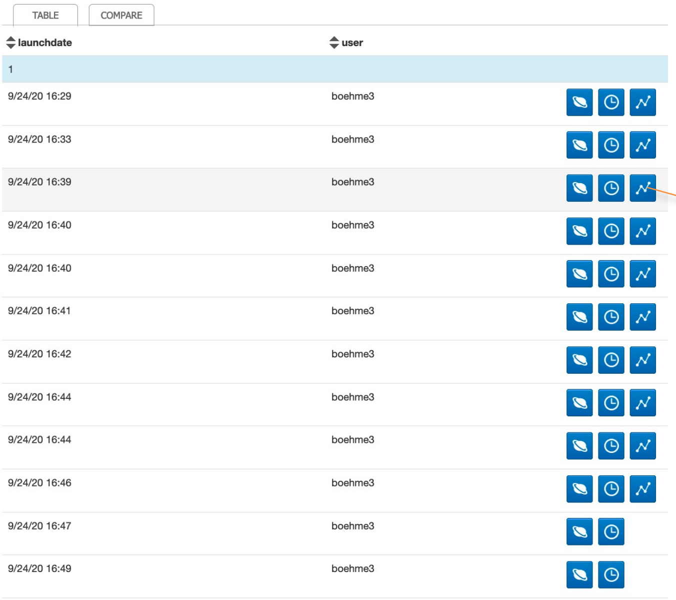


TABLE | COMPARE

launchdate	user
1	
9/24/20 16:29	boehme3
9/24/20 16:33	boehme3
9/24/20 16:39	boehme3
9/24/20 16:40	boehme3
9/24/20 16:40	boehme3
9/24/20 16:41	boehme3
9/24/20 16:42	boehme3
9/24/20 16:44	boehme3
9/24/20 16:44	boehme3
9/24/20 16:46	boehme3
9/24/20 16:47	boehme3
9/24/20 16:49	boehme3

Timeseries data available

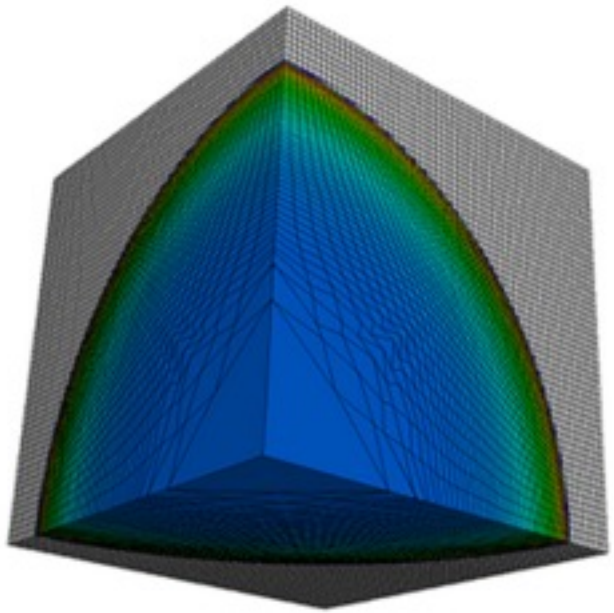
SPOT loop profile visualization

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

CASC

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

EXASCALE COMPUTING PROJECT

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.