radiuss

Welcome to the RADIUSS AWS Tutorial Series!

Go to: https://software.llnl.gov/radiuss/event/2023/07/11/radiuss-on-aws/ to learn more about our other tutorials and documentation!

Date	Time (Pacific)	Project
August 3, 2023	9:00a.m.–11:00a.m.	Build, link, and test large-scale applications with **BLT**
August 8–9 2023	8:00a.m.–11:30a.m. both days	Learn to install your software quickly with **Spack**
August 10, 2023	9:00a.m.–11:00a.m.	Use **MFEM** for scalable finite element discretization application development
August 14, 2023	9:00a.m.–12:00p.m.	Integrate performance profiling capabilities into your applications with **Caliper** Analyze hierarchical performance data with **Hatchet** Optimize application performance on supercomputers with **Thicket**
August 17, 2023	9:00a.m.–11:00a.m.	Use **RAJA** to run and port codes quickly across NVIDIA, AMD, and Intel GPUs Discover, provision, and manage HPC memory with **Umpire**
August 22, 2023	9:00a.m.–11:00a.m.	Visualize and analyze your simulations in situ with **Ascent**
August 24, 2023	9:00a.m.–11:00a.m.	Leverage robust, flexible software components for scientific applications with **Axom**
August 29, 2023	9:00a.m.–11:00a.m.	Analyze runs of your code with **WEAVE**
August 31, 2023	9:00a.m.–11:00a.m.	Learn to run thousands of jobs in a workflow with **Flux**

Caliper: A Performance Profiling Library

2023 RADIUSS Tutorial Series

August 14, 2023

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

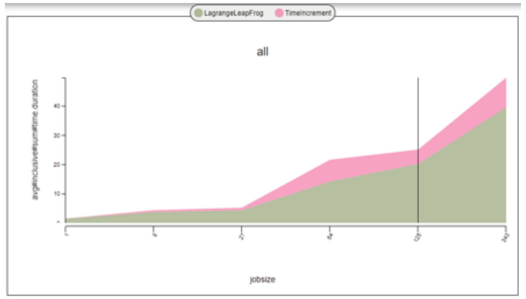
- Integrates a performance profiler into your program
  - Profiling is always available
  - Simplifies performance profiling for application end users
- Common instrumentation interface
  - Provides program context information for other tools
- Designed for HPC
  - MPI, OpenMP, CUDA, HIP, Kokkos support;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



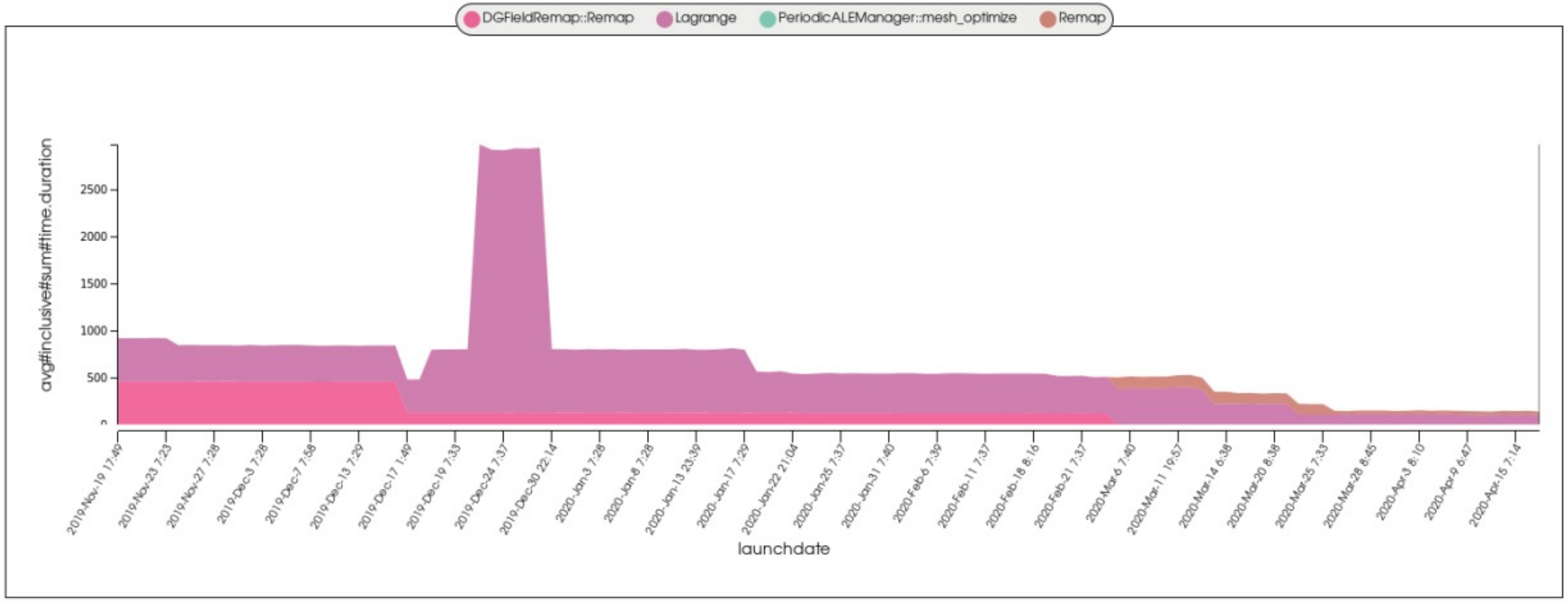
Comparing runs

```
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

Building Automated Performance Analysis Workflows

Enabling performance analysis as a routine activity
for HPC software development

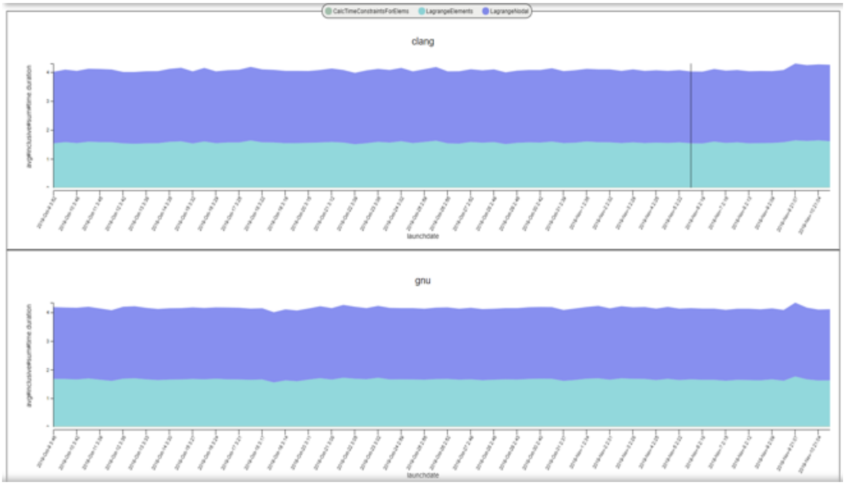


Nightly test performance of a large physics code over 5 months

Performance Analysis with Caliper, SPOT, Hatchet, and Thicket

"spot" config

Pre-populated Jupyter notebooks



SPOT and Thicket:
Analysis of
large collections of runs

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

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

Caliper:
Instrumentation and Profiling

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.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

```
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 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”);
}
```

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 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 the SPOT web visualization framework or Thicket

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

Sample Profiling

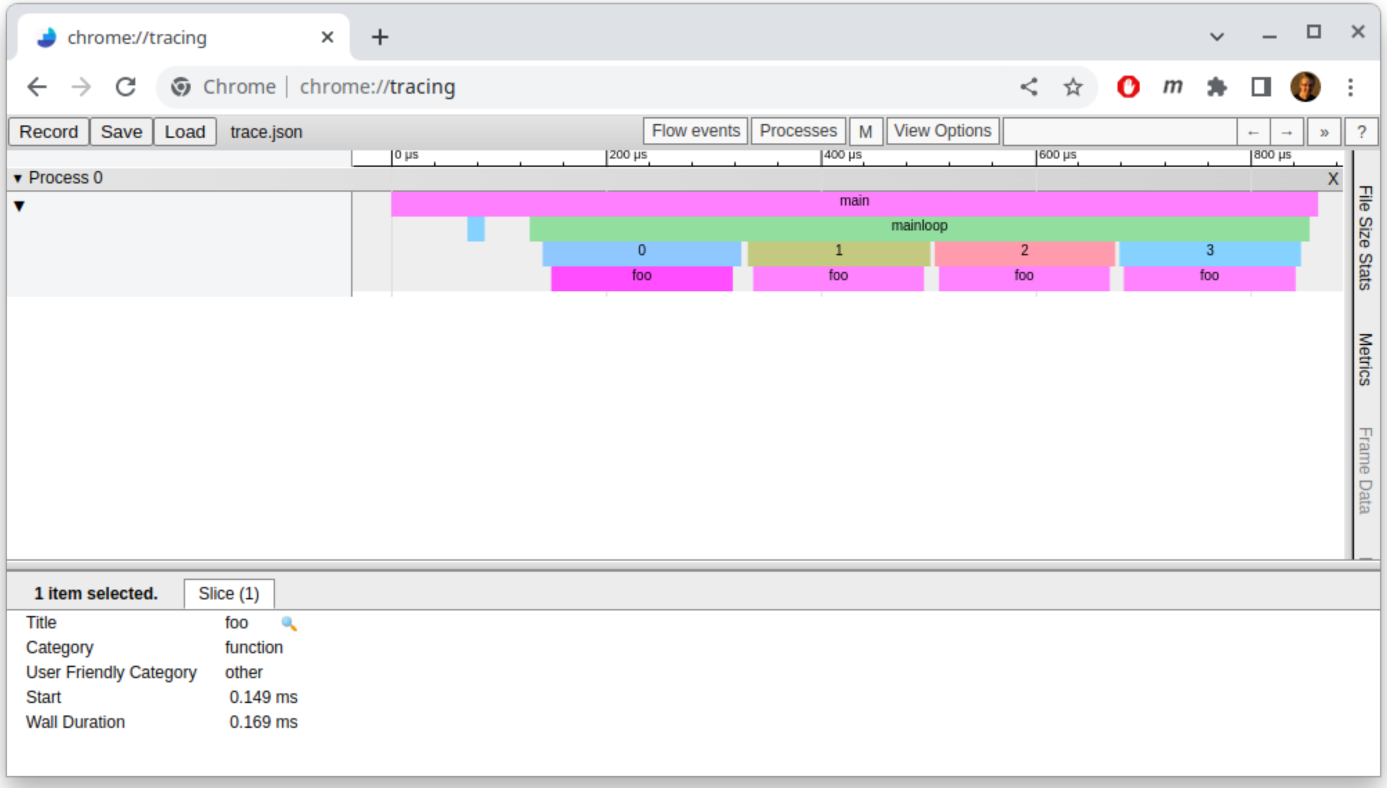
```
$ CALI_CONFIG=sample-report ./lulesh2.0
```

```
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

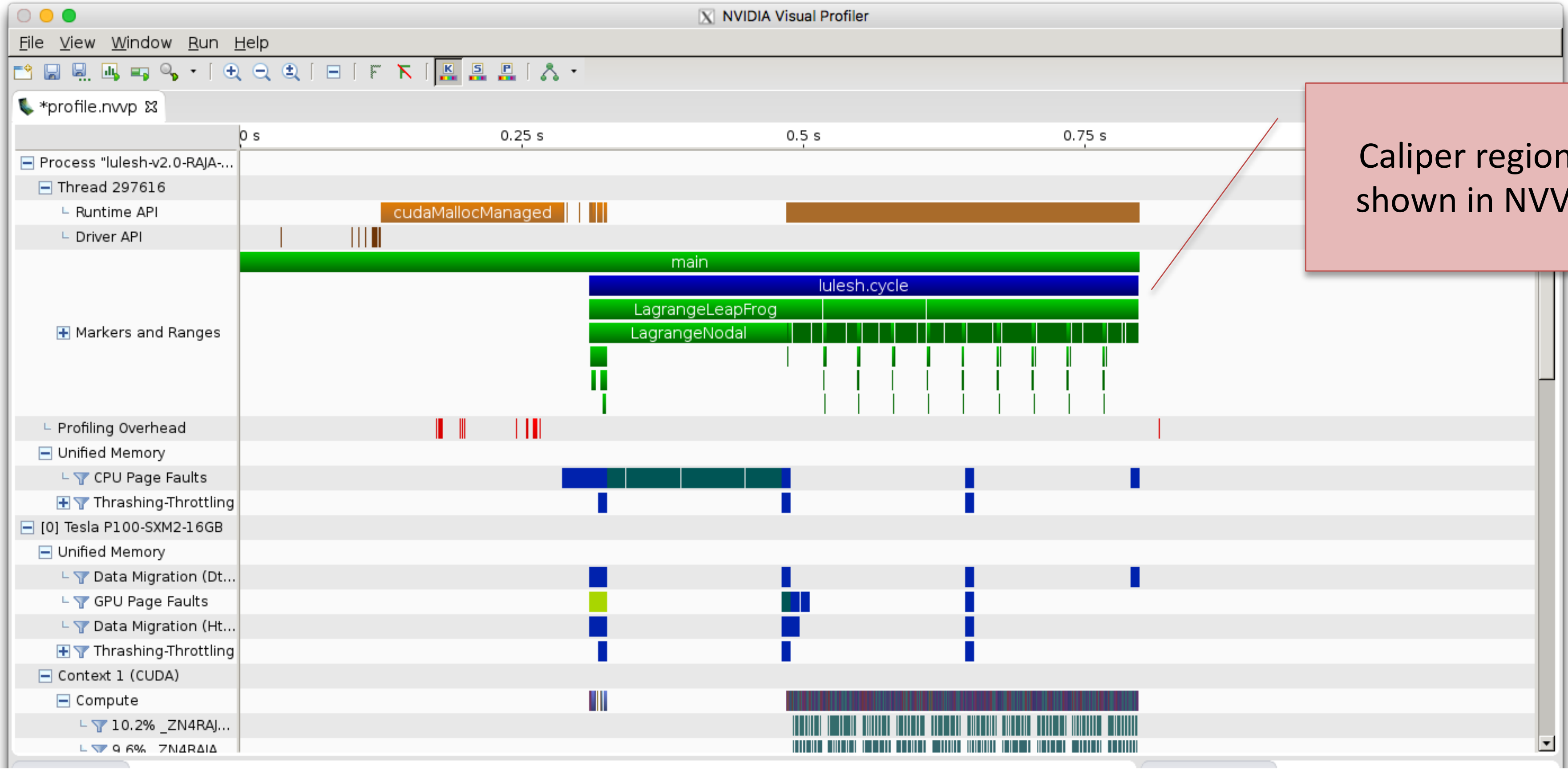
```
$ CALI_CONFIG=event-trace,output=trace.cali ./lulesh2.0
$ cali2traceevent.py trace.cali trace.json
```



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

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

```
$ ./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

Ensemble Performance Data Collection for Thicket

```
#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();
```

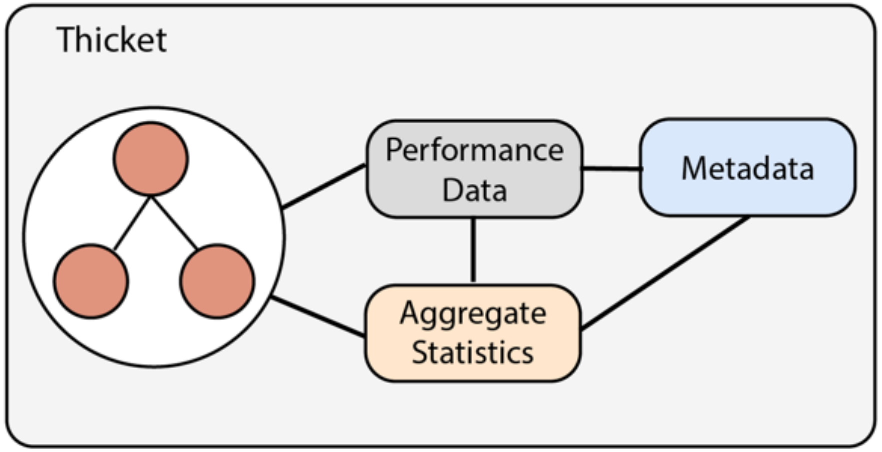
Caliper configuration

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

Run program with the “spot” profiling config

Profile data (.cali)

Experiment directory



Thicket analysis framework

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 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

```
#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

CASC

Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

ECP

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.

Disclaimer

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