The Caliper Profiling Library

41st VI-HPS Tuning Workshop

CASC
Center for Applied Scientific Computing

Feb 7, 2022

David Boehme

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 Instrumentation and 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

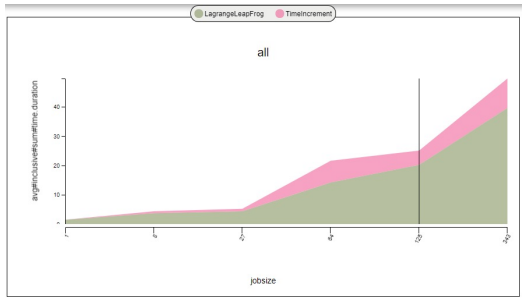
Caliper Use Cases

- 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

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

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();
}
```

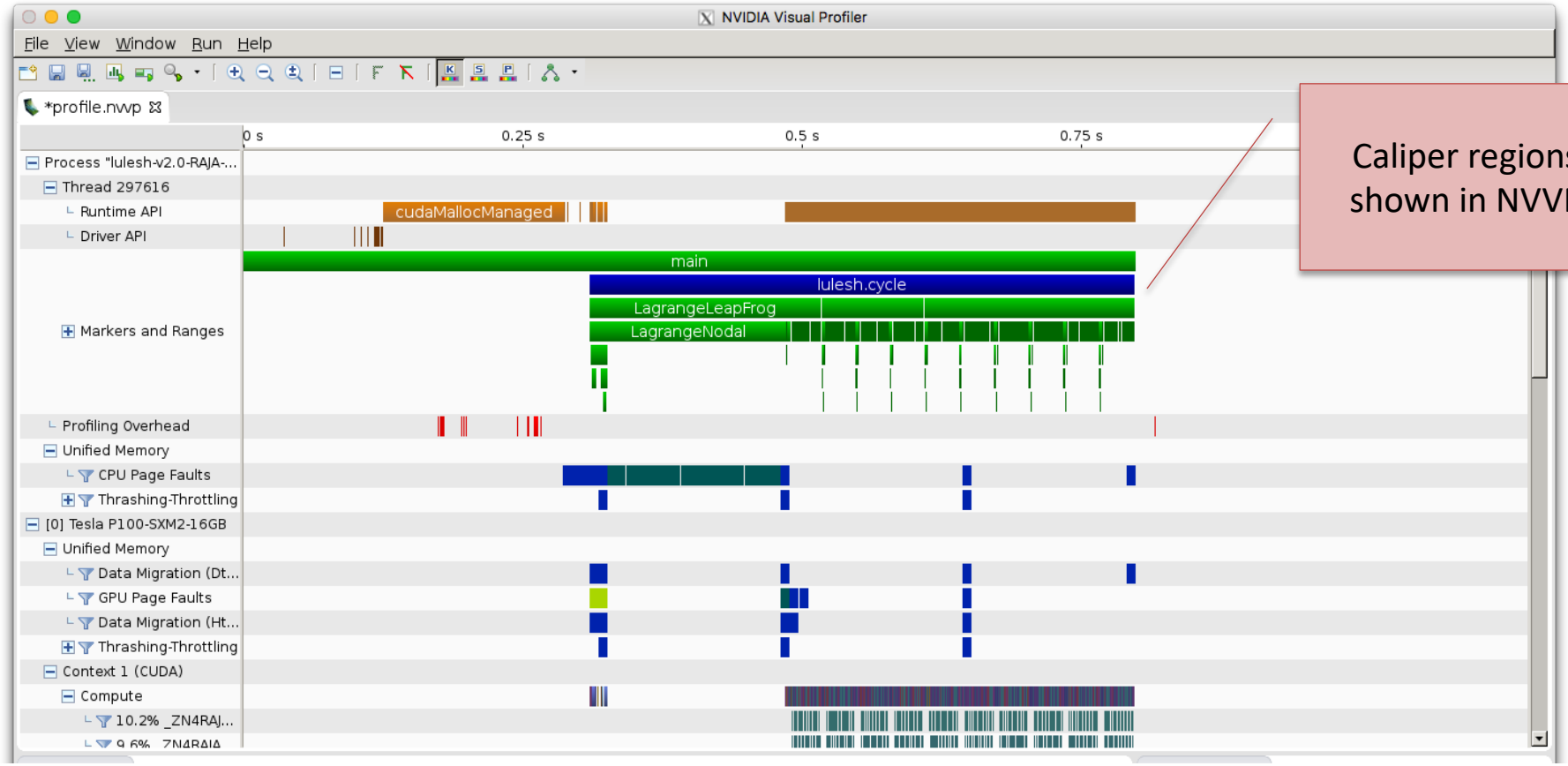
- Use ConfigManager to access Caliper's built-in profiling configurations
- Use add() to add profiling configurations (same config strings as CALI_CONFIG)
- Use start() to start profiling
- Use flush() to collect and write output

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

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

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 machine-readable JSON data 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

Recording Program Metadata: 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

Contact & Links

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

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

Tutorial Environment on JUWELS

- Tutorial materials and example apps:
  https://github.com/daboehme/caliper-tutorial/tree/vihps-training-41
- Build and setup on JUWELS:

```
$ module load CMake GCC CUDA ParaStationMPI
$ git clone --recursive https://github.com/daboehme/caliper-tutorial.git -b vihps-training-41
$ cd caliper-tutorial
$ . setup-env.sh juwels
```

- A pre-built installation with Caliper, Adiak, and the example apps is available:

```
$ export PATH=$PATH:/p/project/training2123/tools/caliper/bin
$ CALI_CONFIG=runtime-report lulesh2.0 -i 10
```

Using the training installation in your own Codes

- Note: Installation is built for GCC / ParaStationMPI / CUDA 11.3
- Classic build/make:

```
$ export CALI_DIR=/p/project/training2123/tools/caliper
$ g++ -o app -L$(CALI_DIR)/lib -L$(CALI_DIR)/lib64 -ladiak -lcaliper -I$(CALI_DIR)/include *.c
```

- In CMake:

```
find_package(adiak)
find_package(caliper)

add_executable(myapp ${SOURCES})
target_link_libraries(myapp PRIVATE adiak::adiak caliper)
```

CASC

Center for Applied
Scientific Computing

Lawrence Livermore
National Laboratory