Automatic Performance Data Collection with Caliper and Adiak

SIAM PP22 - Advances in Performance Modeling of Parallel Code

CASC
Center for Applied Scientific Computing

Feb 25, 2022

David Boehme

LLNL-PRES-XXXXXX

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

Performance Data Collection can be Tedious

- Custom setup and configuration steps
- No context (program inputs, execution environment) recorded with performance data
- Custom data formats for tool-specific visualization & analysis tools

Caliper and Adiak Automate Performance Data Collection

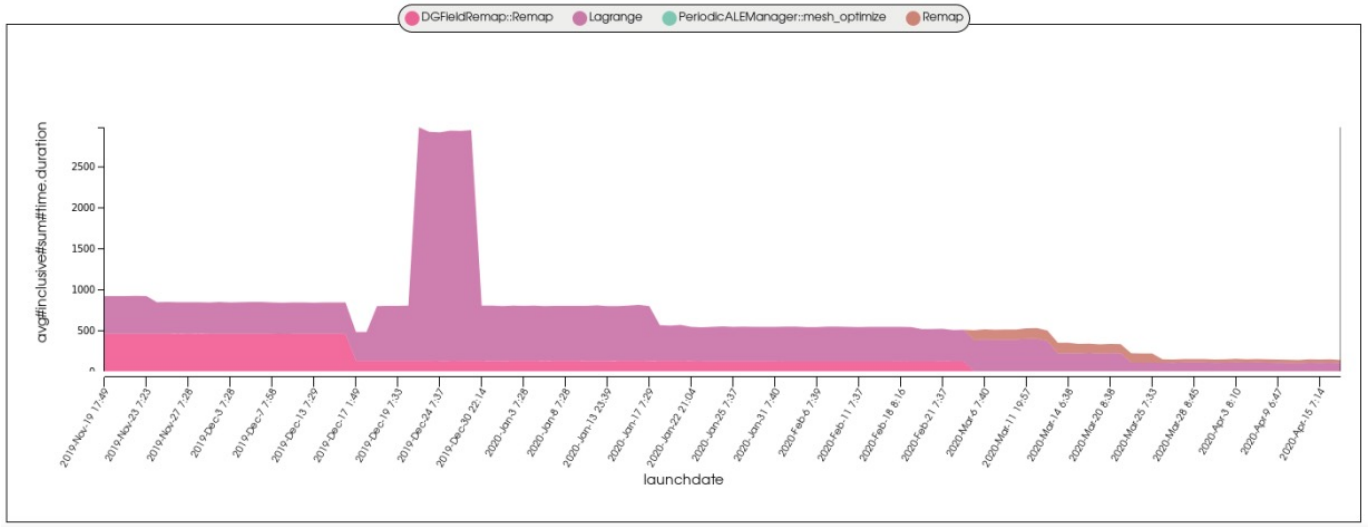
```
#include <caliper/cali.h>

void foo(int arg)
{
   CALI_CXX_MARK_FUNCTION;
// ...
```

Caliper: Embedded instrumentation and profiling library

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

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

Adiak: Program metadata collection

Automated performance regression testing



Nightly test performance of a large physics code over 5 months

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

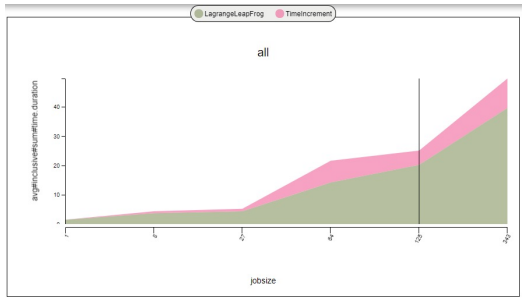
Caliper Use Cases

- 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

Performance Data Collection in a Ubiquitous Performance Analysis Workflow

Program Adaptation

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

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

Add code region instrumentation with Caliper

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

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

Add program metadata annotations with Adiak

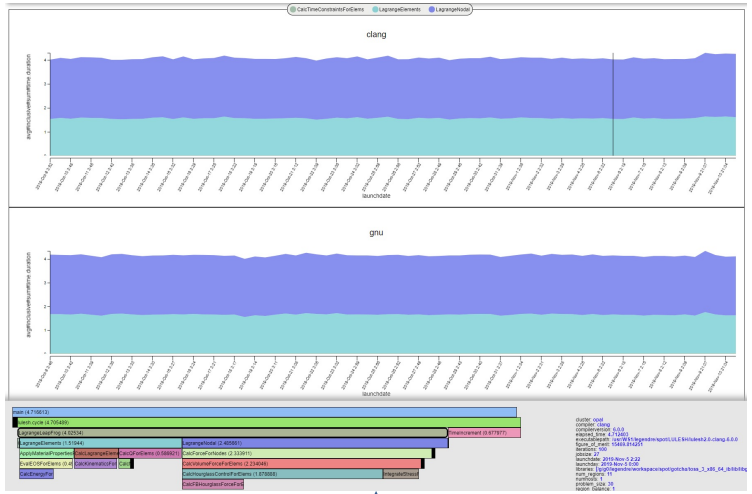
```
cali::ConfigManager mgr;
mgr.add(opts.caliperConfig.c_str());
mgr.start();
// ...
mgr.flush();
```

Measurement control with Caliper ConfigManager API

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

Program execution with profiling config

Visualization and Analysis



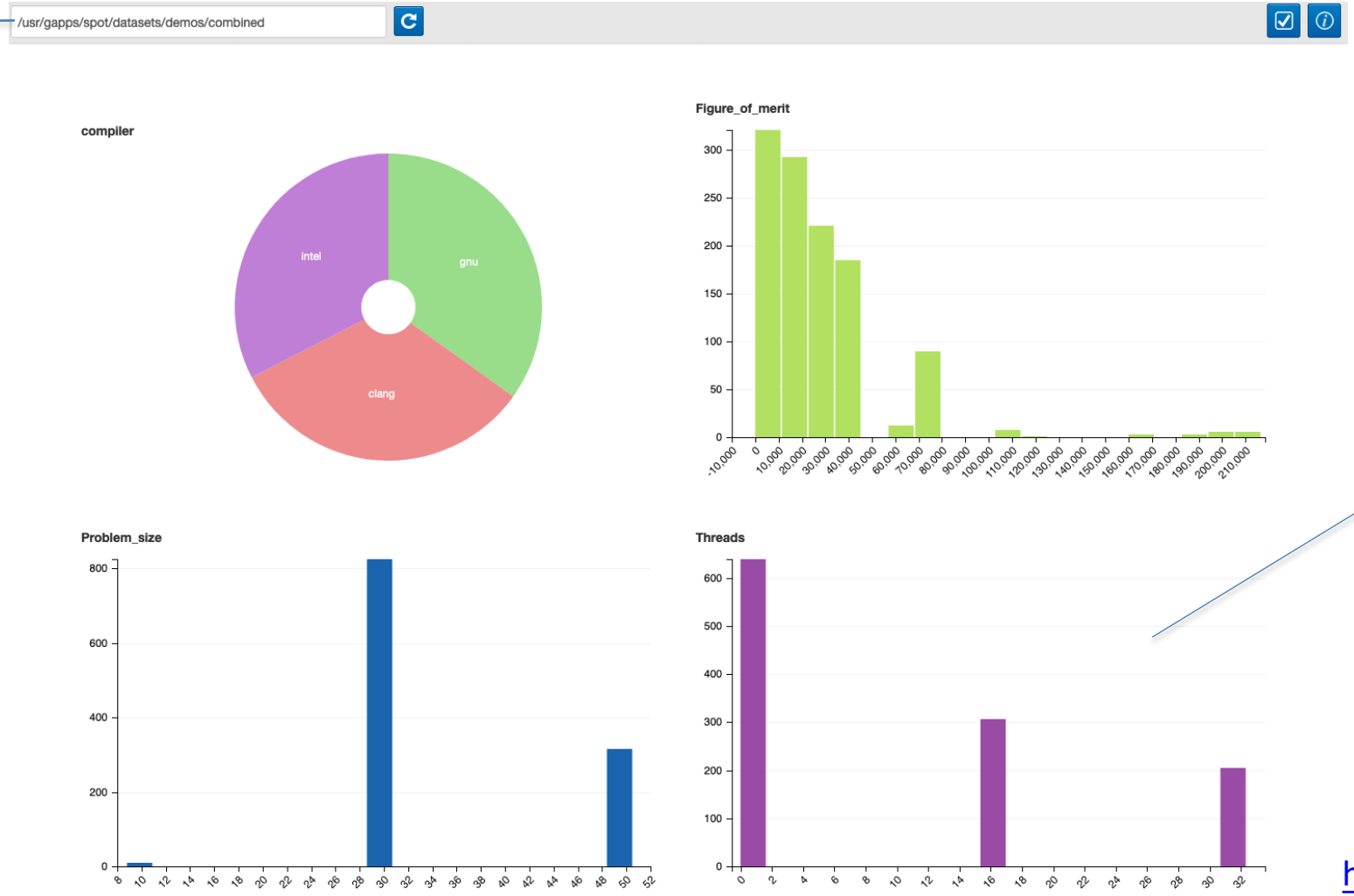
SPOT Web GUI

Profile data (.cali)

Experiment directory

Spot Web Interface: Performance Data Overview

Data directory

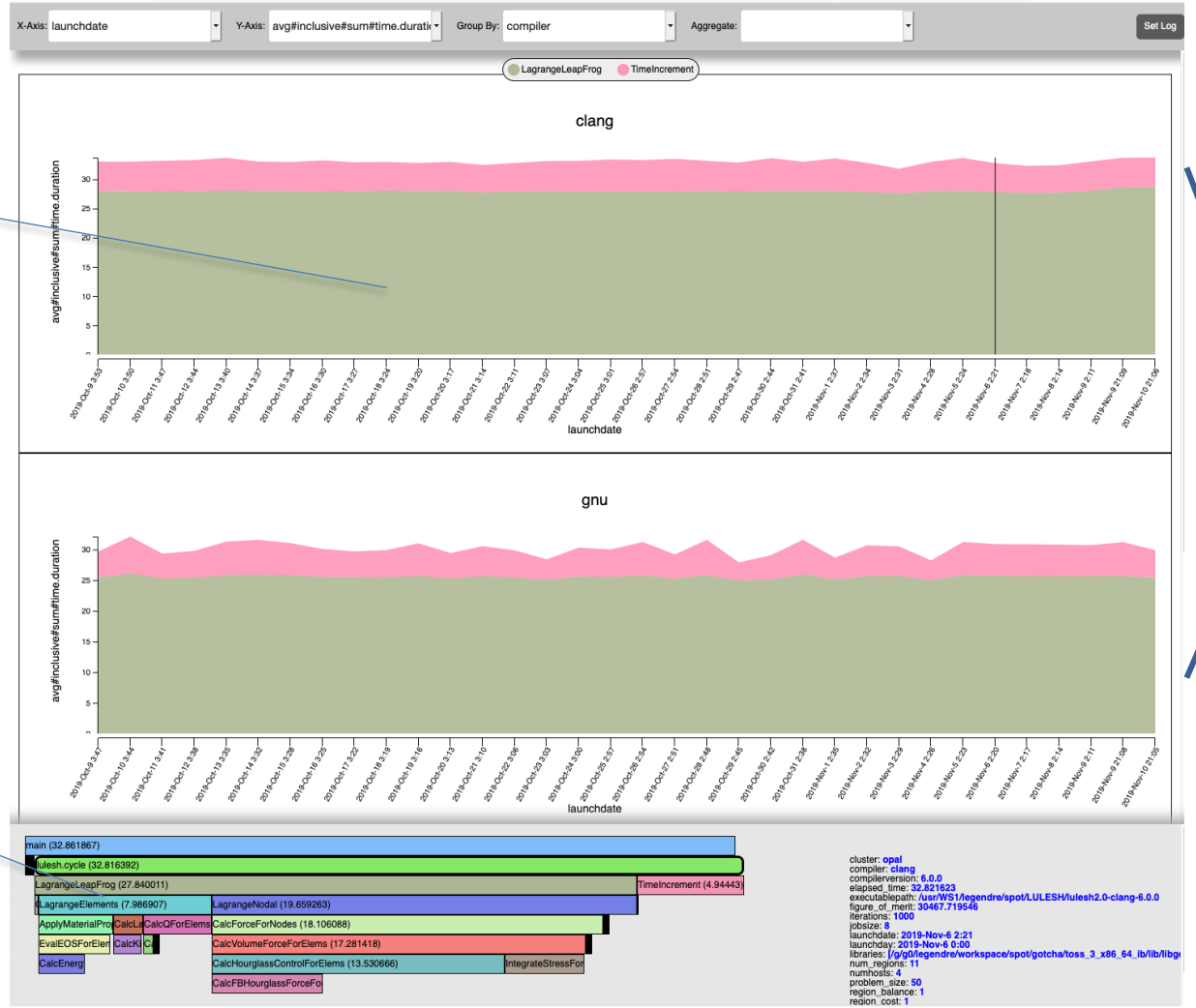


Run metadata widgets

https://lc.llnl.gov/spot/dcvis

Spot Web Interface: Performance Comparison View

Performance (runtime) for many runs, ordered by launch date



Comparing performance between compilers

Details for selected run (function hierarchy)

Code Instrumentation: User-Defined Regions

```
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 region annotations around a RAJA kernel in C++

User instrumentation offers:

Control

- Limit overhead and clutter

Interpretability

- Familiar terminology instead of compiler-generated identifiers

Long-term consistency

- Regions remain stable across program versions

Reliability

- No need for debug info or fragile lookup mechanisms

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

- The Adiak C/C++ library records program metadata
  - Environment info (user, launchdate, system name, ...)
  - Program configuration (input problem description, problem size, ...)
- Enables performance comparisons across runs

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();
}
```

- Caliper embeds profiling functionality in the target code
- ConfigManager API accesses Caliper's built-in profiling configurations

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

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

Using Caliper Profiling Configurations

Users enable a profiling configuration with a short configuration string

Parameters enable additional features, metrics, or output options

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

```
Path                                Min time/rank Max time/rank Avg time/rank  Time %
main                                     0.025360      0.026737      0.025952  4.788115
  MPI_Reduce                             0.000038      0.009638      0.002282  0.420984
  lulesh.cycle                           0.000222      0.000264      0.000233  0.042919
    LagrangeLeapFrog                     0.000217      0.000258      0.000226  0.041766
      CalcTimeConstraintsForElems        0.004274      0.004410      0.004317  0.796535
      LagrangeElements                   0.001346      0.001434      0.001389  0.256364
        ApplyMaterialPropertiesForElems  0.002386      0.002527      0.002428  0.447898
          EvalEOSForElems                0.017306      0.042594      0.026665  4.919779
            CalcEnergyForElems           0.031614      0.088280      0.051809  9.558748
        CalcQForElems                    0.035310      0.035903      0.035681  6.583168
          CalcMonotonicQForElems         0.014510      0.016097      0.015285  2.820023
          MPI_Wait                       0.000199      0.057949      0.022132  4.083461
...
```

The *runtime-report* config prints time in annotated regions. The *profile.mpi* option enables MPI function profiling.

Caliper Profiling Options

Option	
profile.mpi	Time in MPI calls
profile.cuda	Time in CUDA API calls
profile.hip	Time in HIP API calls
region.count	Number of region invocations
mem.highwatermark	Heap allocation (malloc/free) high-water mark
io.bytes	POSIX I/O bytes written/read
mpi.message.count	Number of MPI communications
mpi.message.size	Min/max/avg size of MPI messages

Output Suited for General-Purpose Data Science Tools

```
$ ./lulesh2.0 -P hatchet-region-profile,profile.mpi=true
```

Machine-readable output formats for use with general-purpose data science tools

```
import hatchet

gf = hatchet.GraphFrame.from_caliper('file.json')
print(gf.tree())
```

```
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 library

```
import caliperreader as cr

(records, globals) =
    cr.read_caliper_contents('file.cali')
```

caliper-reader Python library for reading native .cali files

Example: Data Collection for Model Parameterization

Problem: Extrapolate maximum MPI message size for given problem size in Lulesh

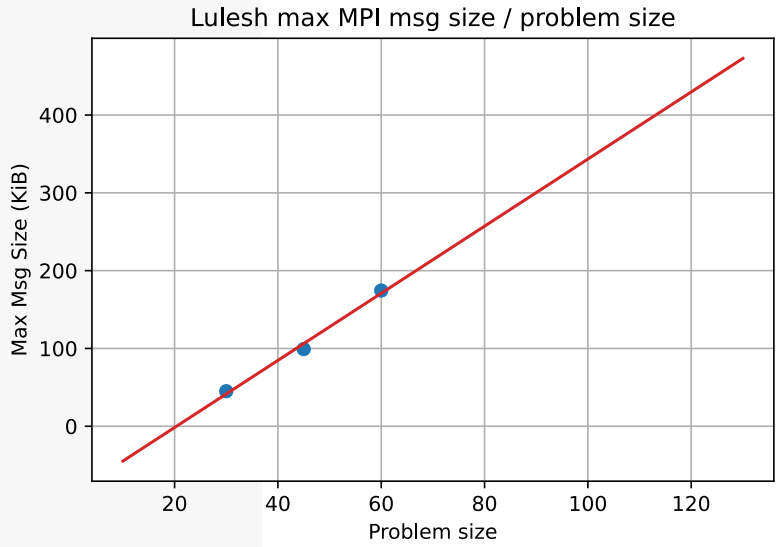
```
for size in 30 45 60
do
    srun -n 8 lulesh2.0 -s ${size} -P spot,mpi.message.size
done
```

```
$ ls *.cali
0.cali 1.cali 2.cali
```

Collecting message size data for three input problem sizes

Extrapolating MPI Message Sizes in Lulesh

In [53]:
```
import hatchet
import matplotlib as mpl
import matplotlib.pyplot as plt
from numpy.polynomial import Polynomial

files = [ '0.cali', '1.cali', '2.cali' ]

x = []
y = []

for f in files:
    gf = hatchet.GraphFrame.from_caliperreader('spot_lulesh_problemsizes/'+f)
    x.append(int(gf.metadata['problem_size']))
    y.append(gf.dataframe['max#max#mpimsg.max'].max() / 1024.0)

fit = Polynomial.fit(x, y, 1, domain=[10,130])
xx, yy = fit.linspace()

fig, ax = plt.subplots()
ax.set_xlabel('Problem size')
ax.set_ylabel('Max Msg Size (KiB)')
ax.set_title('Lulesh max MPI msg size / problem size')
ax.grid(True)

ax.plot(x, y, 'o', color='tab:blue')
ax.plot(xx, yy, color='tab:red')
```



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)

CASC

Center for Applied
Scientific Computing

Lawrence Livermore
National Laboratory