Reproducible Benchmarking for High-Performance Computing

ISC

June 22, 2026

Ramble

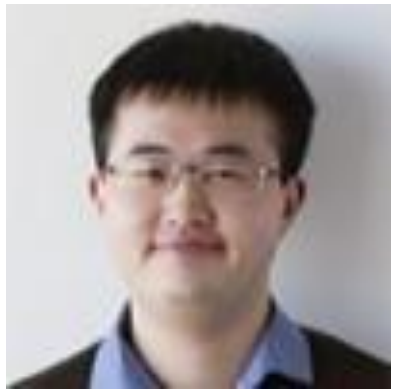
Stephanie Brink (LLNL), Robert Bird (Google), Lin Guo (Google),
Gregory Becker (LLNL), Olga Pearce (LLNL)

Prepared by LLNL under Contract DE-AC52-07NA27344.

Tutorial Presenters

Stephanie Brink
LLNL

Bob Bird
Google



Lin Guo
Google

Greg Becker
LLNL

Tutorial Agenda (approximate)

Welcome and Introduction	All	10 min
Benchpark Overview and Basics Hands on: Running an experiment, analyzing performance of a scaling study	Stephanie	75 min
Ramble Overview and Basics	Bob/Lin	25 min
Break	All	30 min
Benchpark System and Experiment Specifications Hands on: Writing an experiment in Benchpark	Stephanie	60 min
Q&A and Wrap-up	All	10 min

Tutorial Materials

Find these slides and associated scripts here:

software.llnl.gov/benchpark/tutorial-101.html

We also have a channel on Spack slack.
You can join here:

slack.spack.io

Join the **#benchpark-support** channel!

You can continue to ask questions here after the tutorial has ended.

Tutorial Instances: http://bit.ly/4kGQDlc

- We have an AWS instance for the hands-on component of this tutorial
- The instance provides:
  - Pre-installed Benchpark and required dependencies
  - Job scheduler

When logging in to the instance:

- Please use a unique username to avoid resource allocation conflicts
  - First initial followed by last name (e.g., John Doe would be jdoe)
- PW: hpctutorial26

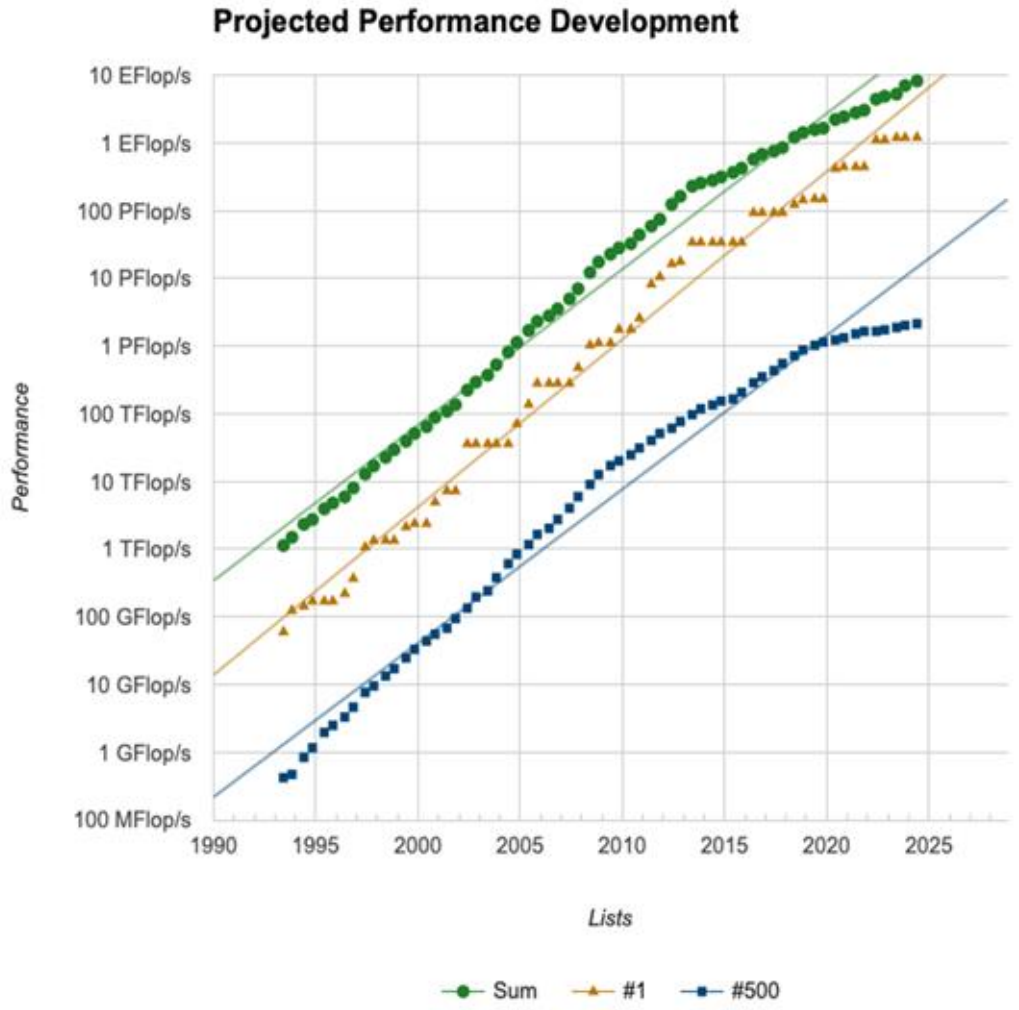
We benchmark HPC systems for many reasons

- Procurements
  - Communicate datacenter workload to vendors
  - Co-design systems, monitor progress
  - System acceptance (contractual specification)
- Validation of software stack, tools
  - Compilers
  - Debuggers
  - Correctness tools
  - Performance tools
- Research
  - Programming models
  - Computational methods
  - Performance across architectures

Benchmarking is challenging

- What are we trying to characterize?
- Are we capturing the best the system can do?
- Is something else impacting performance?
- Did we build and run the code in the optimal and reproducible way?

Source: https://www.top500.org/statistics/perfdevel/



HPC benchmarks run on diverse HPC hardware

Lawrence Berkeley Nat'l Lab
AMD Zen + **NVIDIA**

Fugaku
RIKEN **Fujitsu** ARM a64fx

Lawrence Livermore Nat'l Lab
IBM Power9 + **NVIDIA**

Lawrence Livermore Nat'l Lab
AMD Zen + **Radeon**

Oak Ridge National Lab
AMD Zen + **Radeon**

Argonne National Lab
Intel Xeon + Xe

- Benchmark source code
  - Abstraction (OpenMP, RAJA, Kokkos)
  - Hardware-specific (CUDA, ROCm)
- Optimized code for the CPU and GPU
  - Must make effective use of the hardware
  - Can make 10-100x performance difference
- Rely heavily on system packages
  - Need to use optimized communication and MPI libraries that come with machines

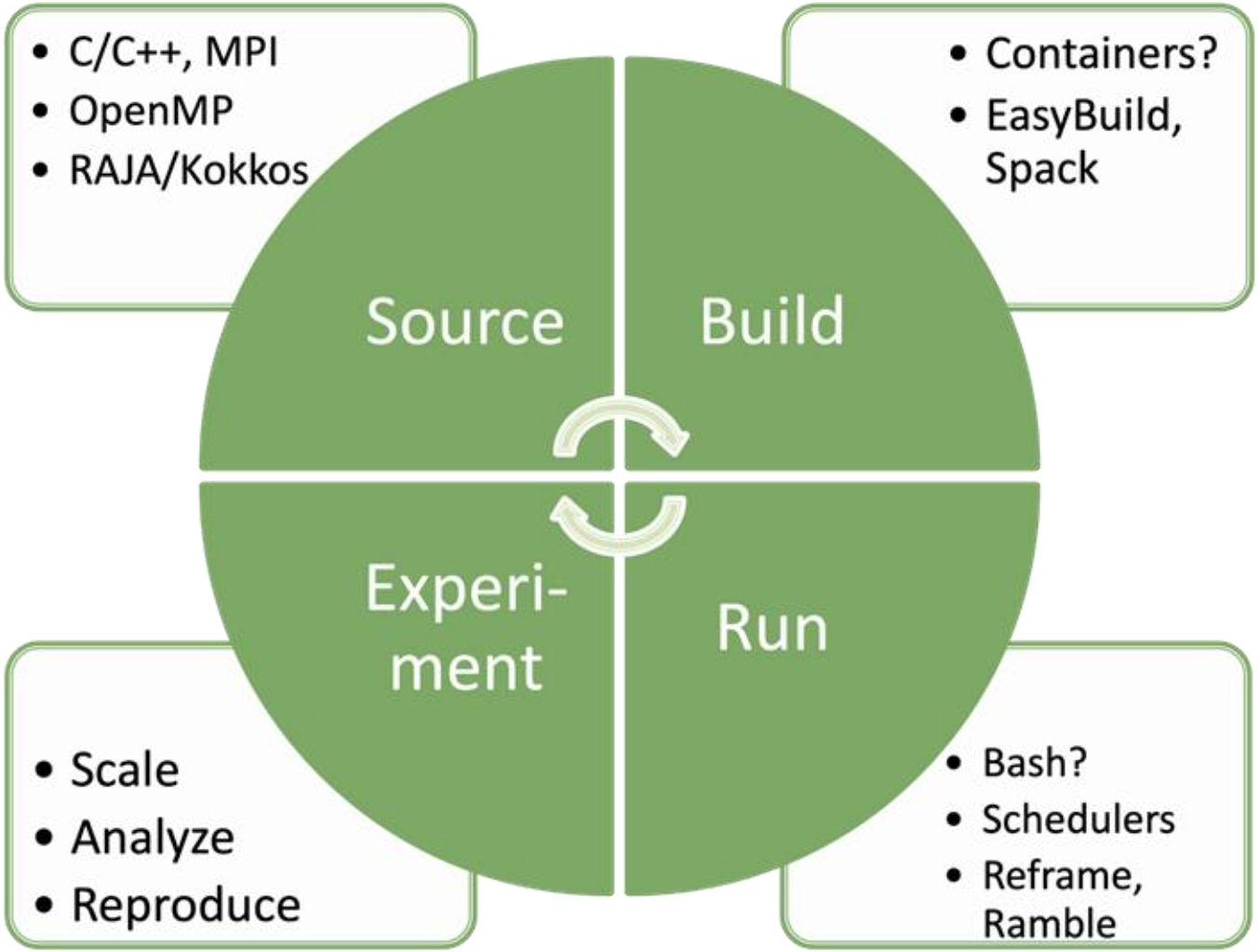
Writing benchmark source code is only the beginning

State-of-the-practice:
HPC system benchmarking is manual!

- Building on each system is different, porting the builds to new systems is manual
- Running on each system is different, porting run scripts to new systems is manual
- Systems keep changing, requiring updates to how we build and run benchmarks
- Triggering builds and runs is manual: benchmark results don't stay up to date
- Performance analysis of results is manual

HPC benchmarks are HPC software

- Portability
- Maintenance
- Testing/CI
- Verification
- Reproducibility



All components must work for your system, focus on explainable performance

Benchpark enables complete specification of HPC benchmarks

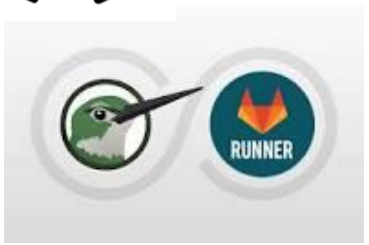
Infrastructure-as-code benchmark specification codifies:

- Benchmark build and run instructions
- HPC Systems
- HPC Experiments

github.com/llnl/benchpark

Leverage advances in HPC automation

- Source code
- Build specification
- Run specification
- CI

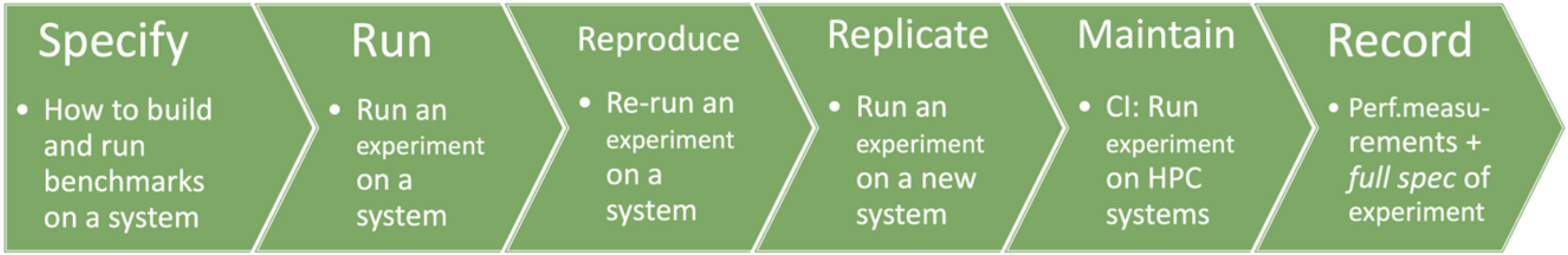


Ramble

Spack

Every part of the specification is codified: used to communicate, automate

Benchpark enables reproducible specifications of benchmarks

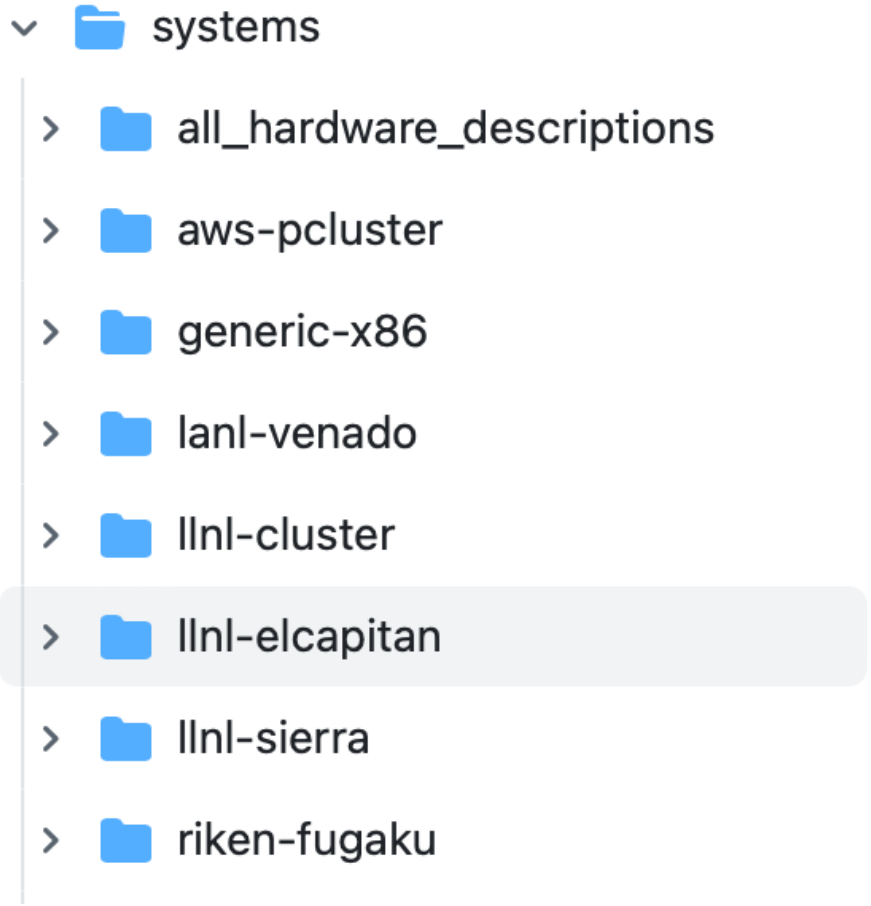


Full specification enables reproducibility, replicability, and automation

HPC System definition for *Performance*

- Resources (CPU cores, GPUs)
- Scheduler (Slurm, flux)
- Process mapping (cores, sockets, GPUs, NICs)
- On-node parallelism (CUDA, ROCM, OpenMP)
- Software stack
  - Compilers (which is best?)
  - MPI implementations (best?)
  - Math libraries

HPC System in Benchpark: *Specify Once*




```
class LlnlElcapitan(System):
    variant(
        "rocm",
        default="6.4.3",
        values=("6.4.3", "7.2.1"),
        description="ROCm version",
    )
    variant(
        "compiler",
        default="cce",
        values=("gcc", "cce", "rocmcc"),
        description="Which compiler to use",
    )
    def initialize(self):
        super().initialize()
        self.scheduler = "flux”
        self.sys_cores_per_node = "84”
        self.sys_gpus_per_node = "4”
```

benchpark system init --dest=elcap llnl-elcapitan cluster=elcapitan rocm=6.4.3 compiler=cce

System Specifications

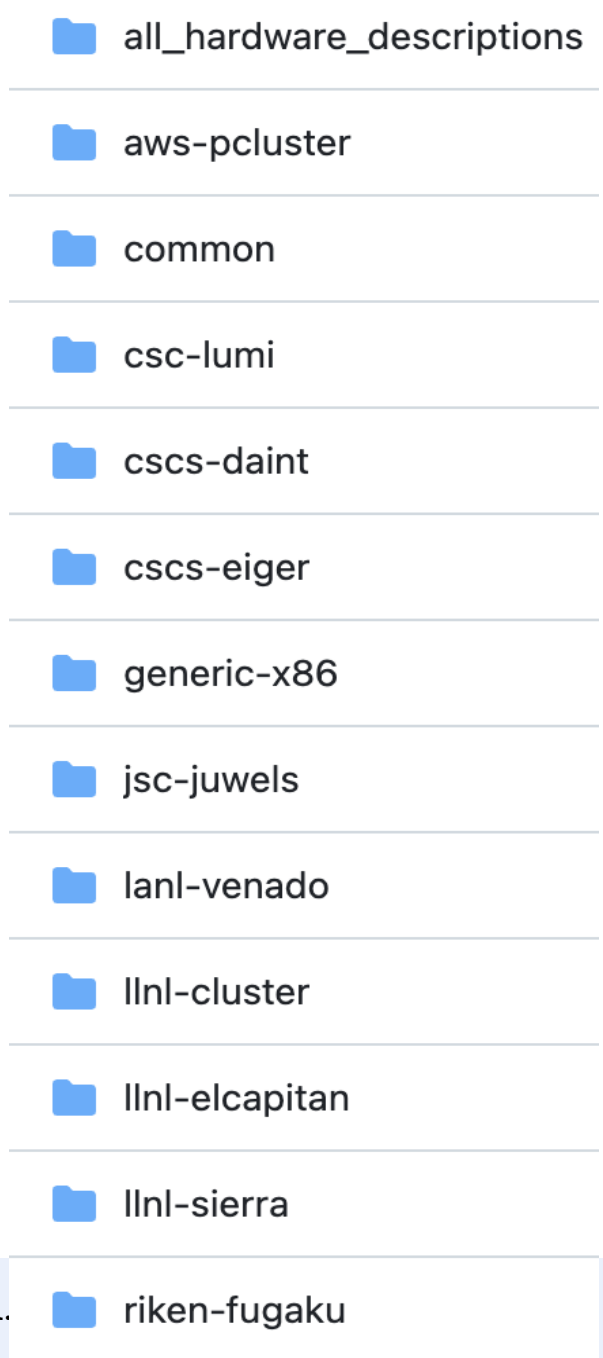
The table below provides a directory of information for systems that have been specified in Benchpark. The column headers in the table below are available for use as the `system` parameter in `benchpark setup`.

Search:

name	integrator.vendor	integrator.name	processor.vendor	processor.name	processor.ISA	processor.u
Atos-zen2-A100-Infiniband	Atos	XH2000	AMD	EPYC-Zen2	x86_64	zen2
AWS_PCluster-zen-EFA	AWS	ParallelCluster	AMD	EPYC-Zen	x86_64	zen
DELL-cascadelake-InfiniBand	DELL		Intel	Xeon6248R	x86_64	cascadelake
DELL-sapphirerapids-OmniPath	DELLEMC	PowerEdge	Intel	XeonPlatinum8480	x86_64	sapphirerapi
Fujitsu-A64FX-TofuD	Fujitsu	FX1000	Fujitsu	A64FX	Armv8.2-A-SVE	aarch64
HPECray-haswell-P100-Infiniband	HPECray		Intel	Xeon-E5-2650v3	x86_64	haswell
HPECray-neoverse-H100-Slingshot	HPECray	EX254n	NVIDIA	Grace	Armv9	neoverse
HPECray-zen2-Slingshot	HPECray		AMD	EPYC-7742	x86_64	zen2
HPECray-zen3-MI250X-Slingshot	HPECray	EX235a	AMD	EPYC-Zen3	x86_64	zen3
HPECray-zen4-MI300A-Slingshot	HPECray	EX255a	AMD	EPYC-Zen4	x86_64	zen4
IBM-power9-V100-Infiniband	IBM	AC922	IBM	POWER9	ppc64le	power9
Penguin-icelake-OmniPath	PenguinComputing	RelionCluster	Intel	XeonPlatinum924248C	x86_64	icelake
Supermicro-icelake-OmniPath	Supermicro		Intel	XeonPlatinum8276L	x86_64	icelake
x86_64					x86_64	

HPC systems in Benchpark: Jun 2026

- 4 in Europe
- 6 in US labs
- 1 in Japan
- 1 at a university
- 2 cloud systems



HPC Experiment definition for *Performance*

- On-node parallelism (CUDA, ROCM, OpenMP)
- Problem sizes
  - Overall problem size, or
  - Per node or per GPU
- Scaling studies
  - How to scale
  - How to decompose
- Resources (cores, GPUs)

Goal: Specify reproducible sets of experiments that map onto specific systems

HPC Experiment in Benchpark: Specify Once

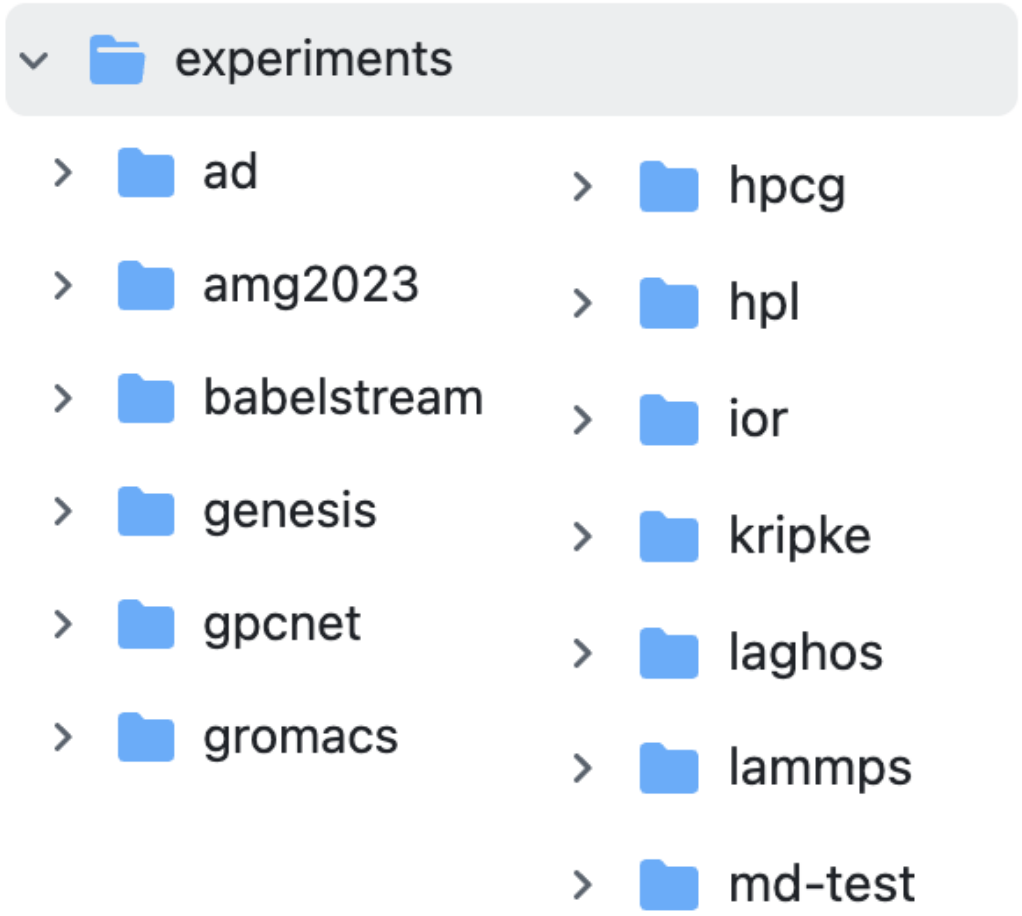
```
class Amg2023(
    Experiment,
    ProgrammingModel(
        ProgrammingModelType.Mpionly,
        ProgrammingModelType.Openmp,
        ProgrammingModelType.Cuda,
        ProgrammingModelType.Rocm,
    ),
    Scaling(ScalingMode.Strong, ScalingMode.Weak, ScalingMode.Throughput),
    Caliper,
):
    variant(
        "workload",
        default="problem1",
        values=("problem1", "problem2"),
        description="problem1 or problem2",
    )

    def compute_applications_section(self):
        self.register_scaling_config(
            {
                ScalingMode.Strong: {
```

benchpark experiment init --dest=amg elcap amg2023 +rocm +strong workload=problem2 caliper=mpi,time

Benchmarks and Experiments

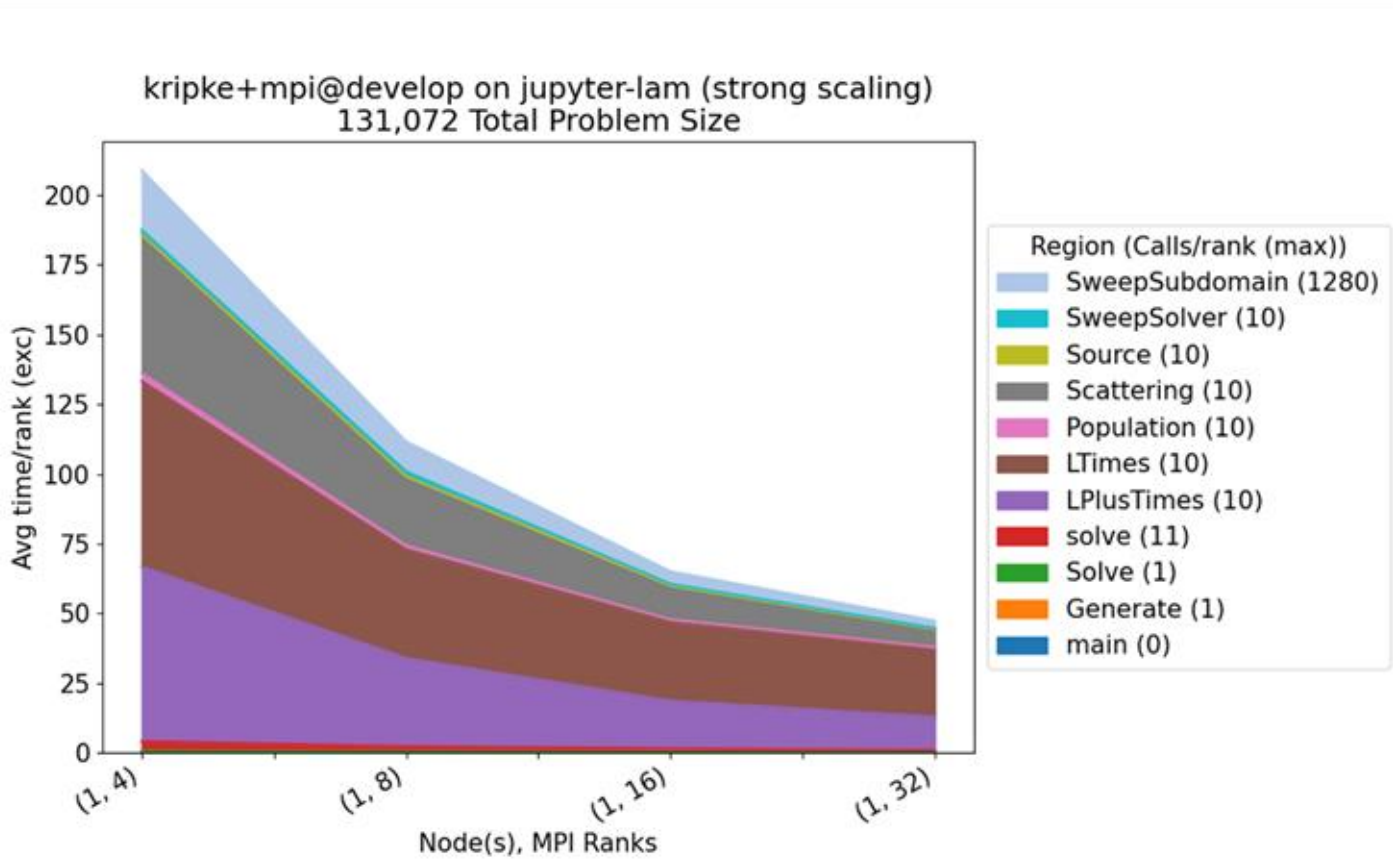
	hpl	hpcg	qws
application-domain	['synthetic']	['synthetic']	['qcd
benchmark-scale	['large-scale']	['large-scale']	['we
communication	['mpi']	['mpi']	['mp
communication-performance-characteristics	['network-collectives', 'network-point-to-point']	['network-point-to-point']	[]
compute-performance-characteristics	[]	[]	[]
math-libraries	['blas']	[]	[]
memory-access-characteristics	[]	[]	[]
mesh-representation	[]	[]	[]
method-type	['dense-linear-algebra', 'solver']	['conjugate-gradient', 'solver', 'sparse-linear-algebra']	[]
programming-language	['c']	['c++']	['c++
programming-model	[]	['openmp']	[]

Experiments in Benchpark: June 2026



- HPL, HPCG
- 4 microbenchmarks
- 3 MPI benchmarks
- 8 US, 1 Europe, 2 Japan

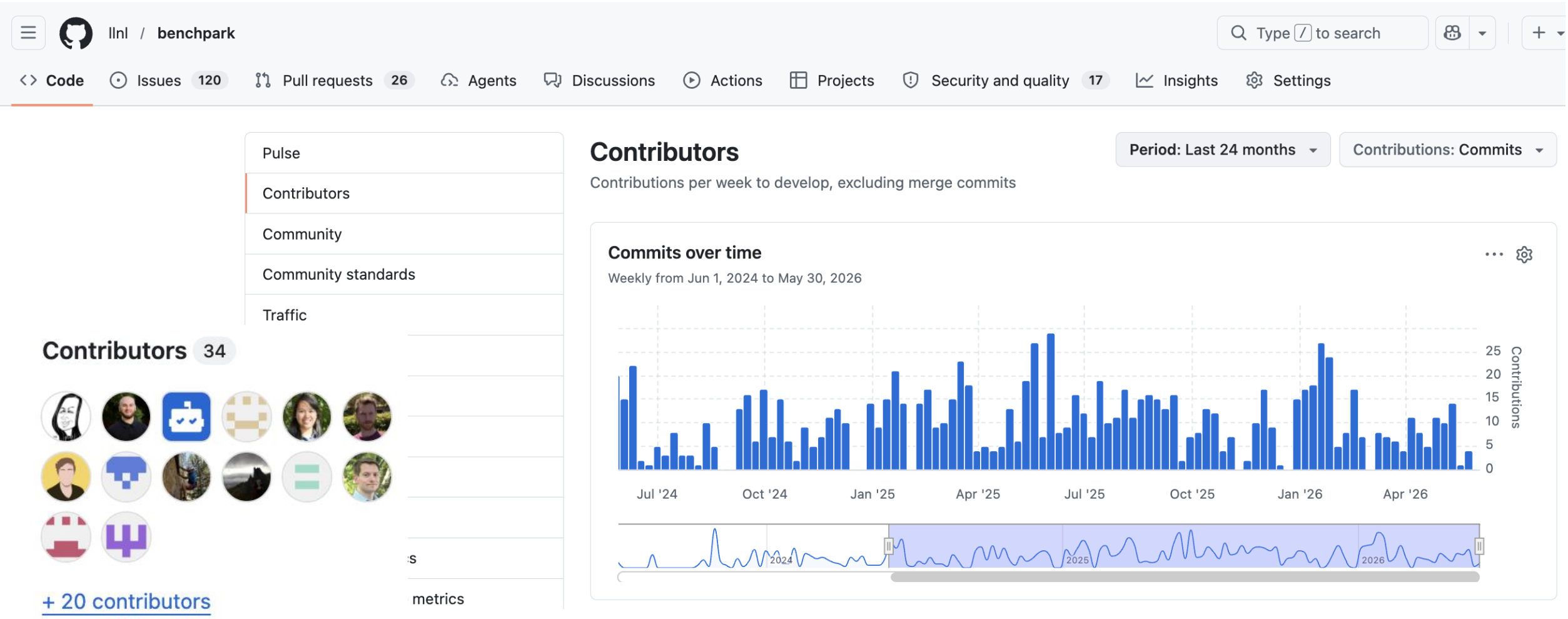
`benchpark analyze` for generating pre-defined analysis charts




```
main
├─ Generate
│  ├─ MPI_Allreduce
│  ├─ MPI_Comm_split
│  └─ MPI_Scan
├─ MPI_Allreduce
├─ MPI_Bcast
├─ MPI_Comm_dup
├─ MPI_Comm_free
├─ MPI_Comm_split
├─ MPI_Finalize
├─ MPI_Finalized
├─ MPI_Gather
├─ MPI_Get_library_version
├─ MPI_Initialized
└─ Solve
   └─ solve
      ├─ LPlusTimes
      ├─ LTimes
      ├─ Population
      │  └─ MPI_Allreduce
      ├─ Scattering
      ├─ Source
      └─ SweepSolver
         ├─ MPI_Irecv
         ├─ MPI_Isend
         ├─ MPI_Testany
         ├─ MPI_Waitall
         └─ SweepSubdomain
```

See https://software.llnl.gov/benchpark/benchpark-analyze.html

Contributions from 11 organizations (60% non-LLNL)



Nightly [benchpark-develop] *71 builds*

Bulk Select | View Timeline

		Update	Test			
Site	Build Name	Revision	Not Run	Fail	Pass	Start Time
dane4	ad	ae81b7	0	1	0	7 hours ago
dane7	amg2023	ae81b7	0	0_{-1}	1^{+1}	6 hours ago
matrix1	amg2023 +cuda	ae81b7	0	0_{-1}	1^{+1}	8 hours ago
matrix1	amg2023 +cuda workload=problem2	ae81b7	0	0_{-1}	1^{+1}	28 minutes ago
dane5	amg2023 +openmp	ae81b7	0	0	1	4 hours ago
tuolumne2151	amg2023 +rocm	ae81b7	0	0	1	8 hours ago
tuolumne2152	amg2023 +rocm gpumode=CPX	ae81b7	0	0	1	5 hours ago
tuolumne1003	amg2023 +rocm gpumode=TPX	ae81b7	0	0	1	5 hours ago
tuolumne2151	amg2023 +rocm workload=problem2	ae81b7	0	0	1	3 hours ago
dane5	amg2023 workload=problem2	ae81b7	0	0	1	2 hours ago
dane2	amg2023@develop	ae81b7	0	0	1	6 hours ago
matrix2	babelstream +cuda	ae81b7	0	0	1	1 hour ago
dane1	babelstream +openmp	ae81b7	0	0	1	4 hours ago
tuolumne1003	babelstream +rocm	ae81b7	0	0	1	7 hours ago
matrix2	branson +cuda	ae81b7	0	0_{-1}	1^{+1}	2 hours ago
dane4	branson +openmp	ae81b7	0	0	1	4 hours ago

Public CDash dashboard for CI tests

Benchpark codifies benchmarking steps

- Benchpark does **not** replace benchmark source, build system, or Spack package
- Benchpark manages benchmark **experiments** and how they map onto **systems** with specified (or default)
  - Compilers
  - MPI/comm libraries
  - Math libraries
- Start running benchmark experiments on your system with just a few commands

```
git clone git@github.com:LLNL/benchpark.git
```

```
benchpark list systems
benchpark system init --dest=elcap llnl-elcapitan
```

```
benchpark list experiments
benchpark experiment init --dest=amg llnl-elcapitan
                          amg2023 +rocm +strong
```

```
benchpark setup amg workspace
ramble workspace setup
```

```
ramble on
```

Who does Benchpark target

People who want to use or distribute benchmarks for HPC!

1. End Users of HPC Benchmarks

- Install, run, analyze performance of HPC benchmarks

2. Benchmark Developers

- People who want to share their benchmarks

3. Procurement teams at HPC Centers

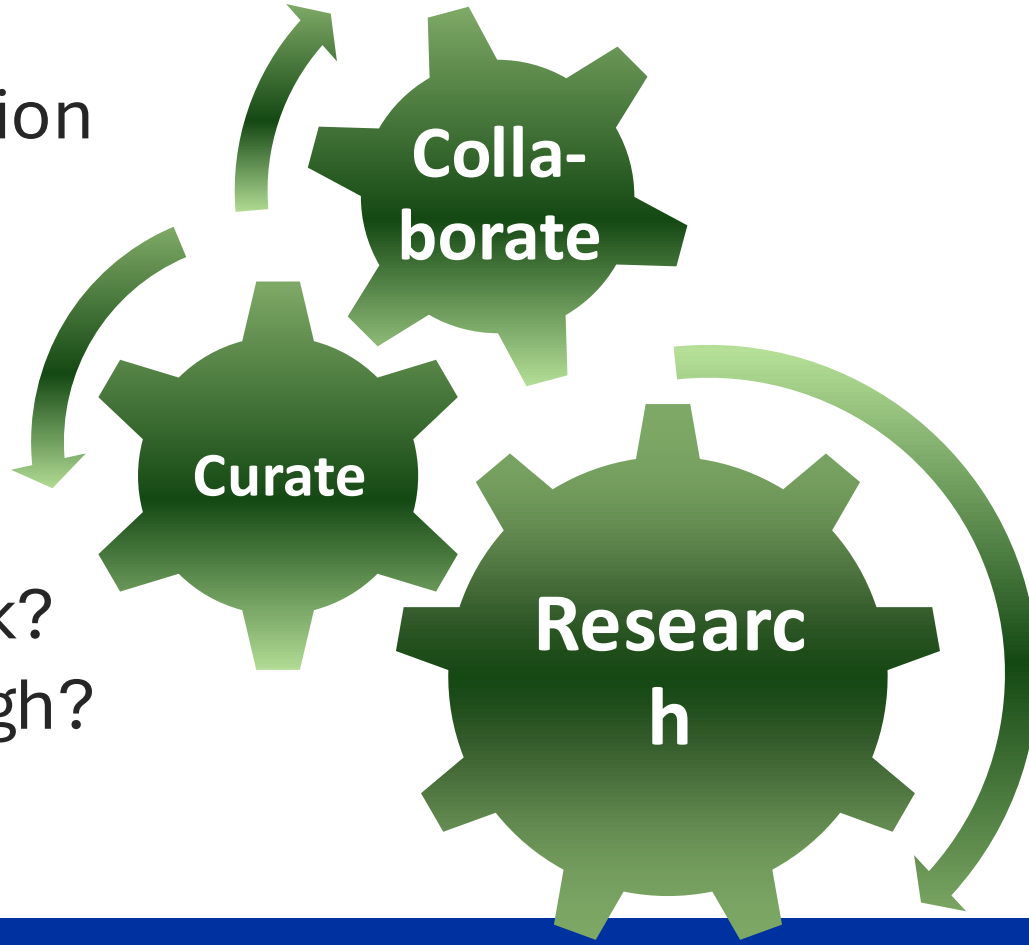
- Curate workload representation, evaluate and monitor system progress

4. HPC Vendors

- Understand the curated workload of HPC centers, propose systems

Catalogued library of *working* benchmarks

- Enables exploration of large configuration space
  - Architectural
  - Software stack
  - Temporal
- What architecture and system configuration is best for my benchmark?
- On my system, is OpenMPI good enough?
- What purpose will Benchpark help you address?

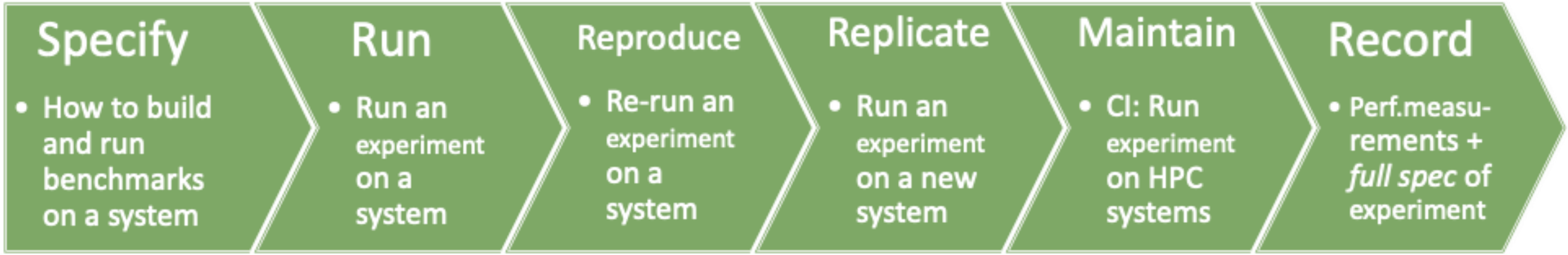


Building a community to contribute/benefit

Benchpark: Open collaborative repository for reproducible specifications of HPC benchmarks

Infrastructure-as-code benchmark specification enables **reproducibility, replicability, and automation**

- HPC Systems
- HPC Experiments



- Tagging, keywords for publications
- Performance metrics, metrics of usefulness
- Dashboards: Archive specs+results

- CI pipelines on PRs from GitHub at data centers across the world and in the cloud

*Olga Pearce et al, Continuous Benchmarking, HPCTests SC|23

*Olga Pearce et al, Repeat, Replicate, Reproduce, ACM REP 2025

Full specification enables reproducibility, replicability, and automation

Benchpark roadmap and community engagement

Future directions:

- Suite curation: Reproducible specification of an entire suite
- Tagging: Keywords for publications, finding benchmarks
- Metrics: Performance, usefulness
- Dashboards: Archive+share *specs*+results, Slices in configuration space
- CI pipelines at data centers across the world and in the cloud

Community Engagement:

- Co-design / vendors on board, but incentive for app teams? (carrot or stick?)
- Who owns which parts of the specification and approves changes?
- Who finances R&D and maintenance?
- ROI for the people working on it? → think about post docs, researchers, etc.

Collaboration, reproducibility, fully-specified public results

Hands On Session 1
Running an existing benchmark on an existing system

Follow script at http://software.llnl.gov/benchpark/tutorial-101.html

Tutorial Instances: http://bit.ly/4kGQDlc

- We have an AWS instance for the hands-on component of this tutorial
- The instance provides:
  - Pre-installed Benchpark and required dependencies
  - Job scheduler

When logging in to the instance:

- Please use a unique username to avoid resource allocation conflicts
  - First initial followed by last name (e.g., John Doe would be jdoe)
- PW: hpctutorial25

Tutorial Materials

Find these slides and associated scripts here:

software.llnl.gov/benchpark/tutorial-101.html

We also have a channel room on Spack slack.
You can join here:

slack.spack.io

Join the **#benchpark-support** channel!

You can continue to ask questions here after the tutorial has ended.

What is Ramble?

What is Ramble?

Ramble is an open-source experimentation framework written in python.

Ramble's primary goals are to:

- **Accelerate productivity**
- **Improve experiment portability**
- **Increase reproducibility information**
- **Encoding domain knowledge**

Ramble is particularly good at generating parameterized "experiments", but can be used for more than performance focused experiments.

What is Ramble?

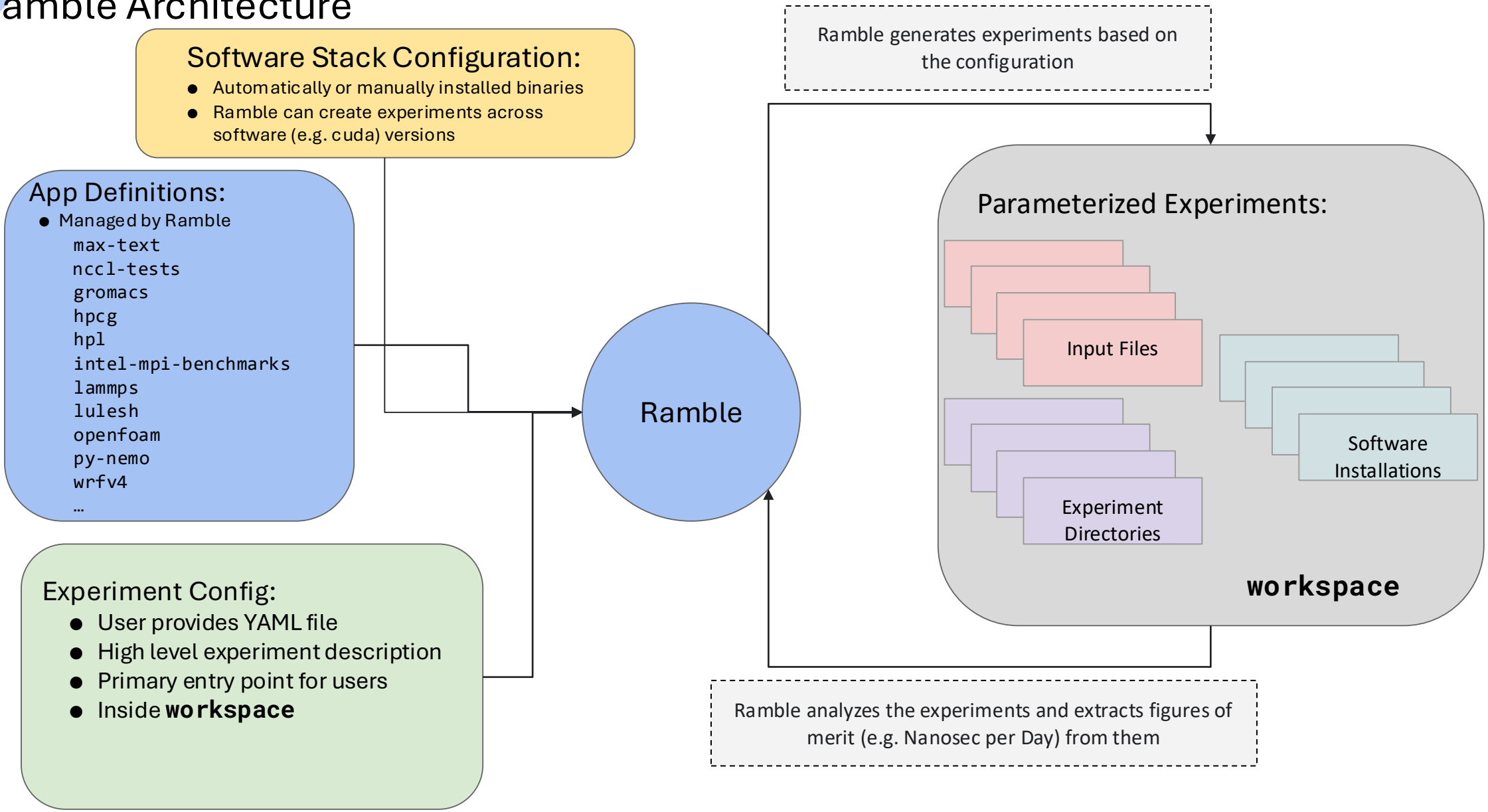
Ramble is:

- Written in python
- Multi-platform
- Heavily based on Spack (Written on top of Spack's infrastructure)
- An experimentation framework (Related to running experiments like Spack is to building software)
- Open Source: https://github.com/GoogleCloudPlatform/ramble

Ramble provides:

- A flexible templating engine
- A domain specific language for describing an application and its workloads
- Standardized definitions of how workloads are configured and executed

Ramble Architecture



Ramble Components

Object Definitions

Portable definitions written in python, that represent:

Applications:

- Executables (commands to run)
- Input files (things to download)
- Software (things to install)
- Workloads (executables + input files)
- Figures of merit (things to analyze)

Package Managers (spack, eessi)
Workflow Managers (slurm, batch, gke)
Modifiers (lscpu, intel-aps, etc..)

Ramble supports **public** and **private** object definitions through configurable repos.

Does not include experiment or system details.

Experiment Configuration (Workspace)

Self contained directory describing a set of **parameterized** experiments to execute.

Contains:

- YAML File, describes:
  - Experiments
  - Software stack
  - Execution workflow
- Template files

After setup, contains:

- Input files
- Experiment execution directories
- Software environments

Experiment Directory

Each experiment has **their own** execution directory.

These contain:

- Rendered versions of the template files (which can then be executed)

After executing the might contain:

- Links to the input files
- Output files from the experiment to be analyzed

Ramble Components

When Ramble manages the software stack, it can parameterize portions of the stack to generate various studies. This includes:

- Changing compilers
- Changing dependencies (e.g. MPI impl., CUDA vs. ROCM)
- Package versions (e.g. CUDA versions)
- Optimization Targets (e.g. Icelake vs. Zen 3)
- etc...

Software Stack

Ramble supports using a predefined software stack, or generating a new software stack to produce experiments.

Supported software stack workflows in Ramble include:

- Pre-installed software (Considered “Existing”)
  - e.g. ISV applications installed in a static location
  - Software installed manually through another mechanism (yum, apt, pip, etc..)

Allows very flexible descriptions of the software environment

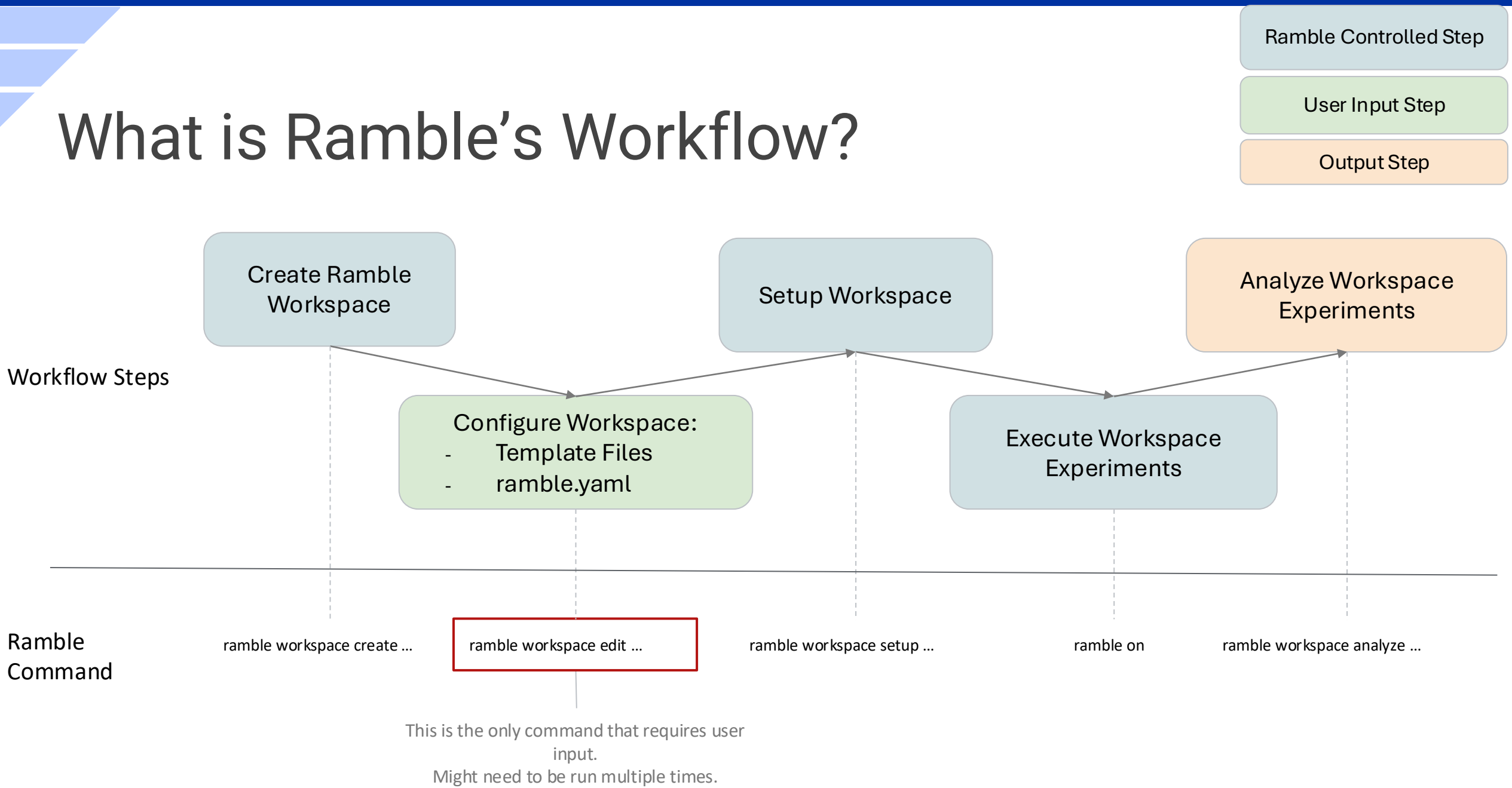
- Driving the creation of a software stack:
  - Using a 3P package manager (currently supports spack, pip, eessi, environment-modules)
  - Can parameterize various aspects of the stack
  - Can use a binary cache to accelerate deployments, and provide binary objects to customers

Spack is used to distribute well defined software stacks.

Various organizations provide public spack caches, which can make it faster to deploy a complex software stack.

How does Ramble work?

What is Ramble's Workflow?

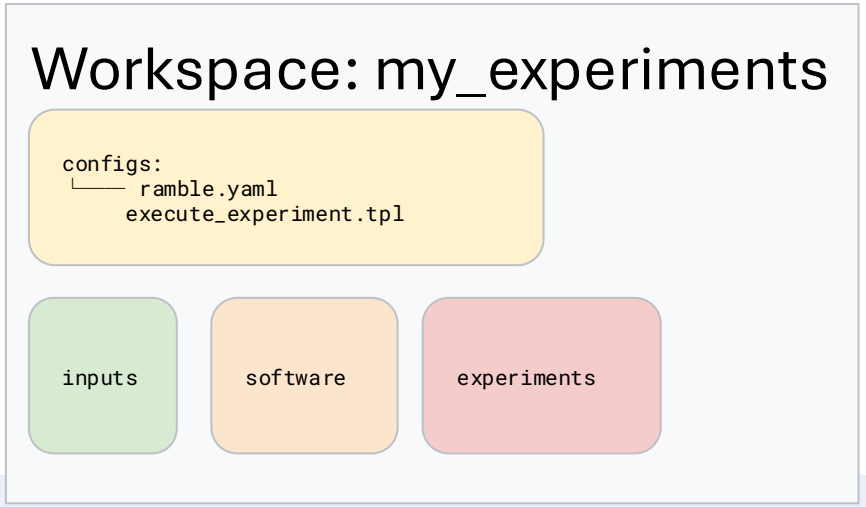


Ramble Workspace:

A workspace is a self contained directory, that contains:

- software environments
- input files
- experiments

Workspaces are independent, and should represent a set of independent experiments you want to execute.



Workspace Config:

Ramble's workspace config YAML can generate many experiments with only a little syntax.

```
applications:
  wrfv3:                                          <-- Application Definition Name
    n_repeats: 5                                  } Define 5 repeats per experiment
    variables:
      processes_per_node: [30, 56]                } Define two vectors, of length 2.
      partition: ['c2', 'c2d']
      n_ranks: '{processes_per_node}*{n_nodes}'
    workloads:
      CONUS_2p5km:                                <-- Workload Name (from Application Definition)
        experiments:
          scaling_study_{partition}_{nodes}nodes:
            variables:
              n_nodes: [1, 2, 4, 8, 16, 32]       } Define one vector, of length 6
            matrix:                               } Define one matrix, using the n_nodes vector. Shape is 1x6.
            - n_nodes
```

- Vectors that are not used by a matrix are zipped together (must be the same length)
- Zip of vectors are crossed with any matrices
- Result is: 1x6 (matrix) x 2 (vectors) x 5 (repeats) = 60, where each index is a 4-element tuple: (processes_per_node, n_ranks, n_nodes, repeat_idx)

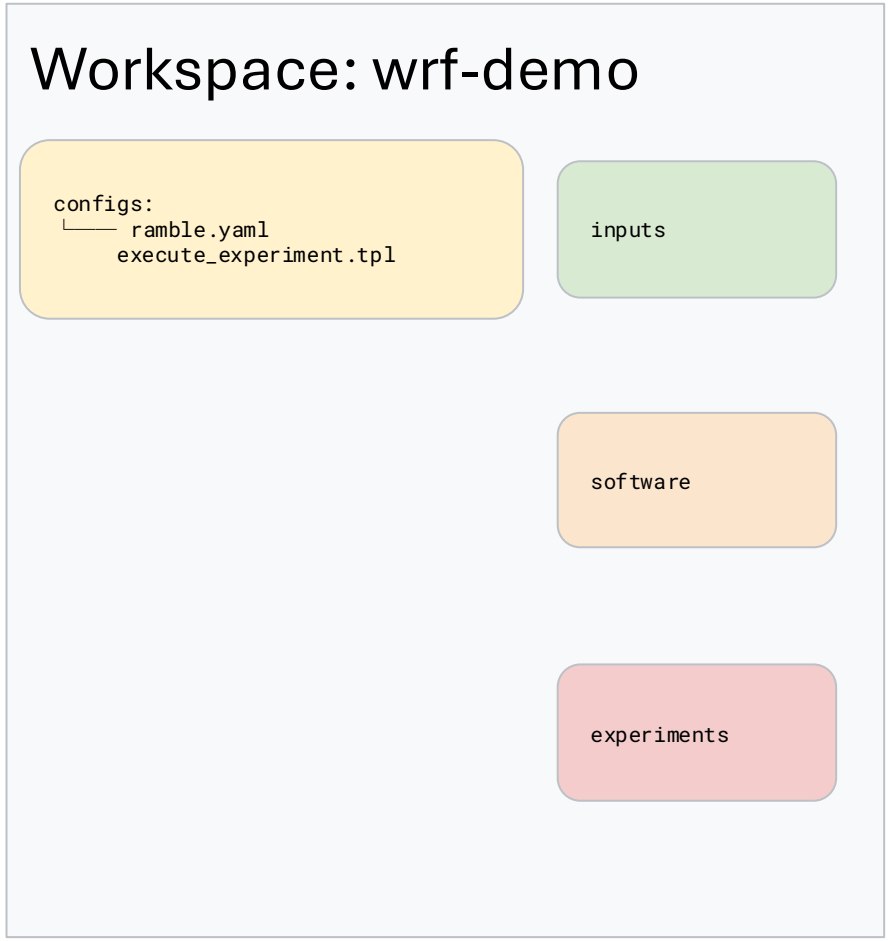
Workspace Config:

In addition to the YAML config, Ramble has a template engine to render scripts:

```
#!/bin/bash

{command}
```

```
#!/bin/bash

source <path/to/spack>
spack env activate <path/to/env>
cp <inputs>/* <experiment_dir>/.
mpirun -n <n_ranks> wrf.exe
```

Workspace Setup:

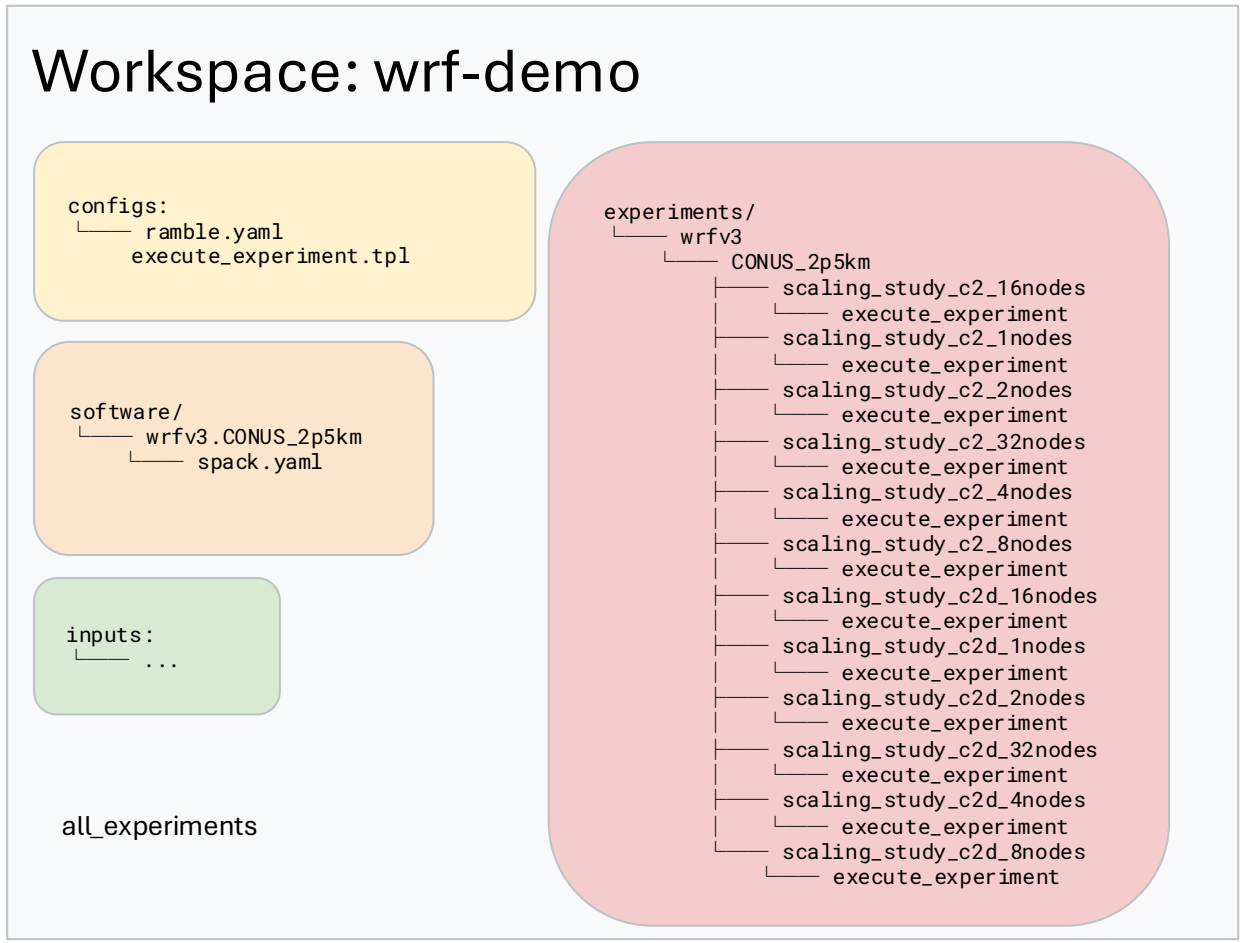


Execute input phases:
- Determine which inputs are necessary
- Download them
- Define variables for referring to these inputs

Execute software phases (Only used on SpackApplications currently):
- Install necessary compilers
- Create spack environments for the experiments
- Using spack, install the required software
- Define variables for referring to the spack environments

Execute experiment creation phases:
- Create experiment execution directories
- Define variables specific to the experiment
- Render any *.tpl files into the execution directories
- Append the experiment to $workspace/all_experiments

Execute Experiments:



After setting up a workspace, the experiments can be executed using:

- $workspace/all_experiments
- ramble on (with an activated workspace)

Depending on the ramble.yaml and execute_experiment.tpl this:

- Executes experiments sequentially
- Submits experiments to a workload manager

Analyze Workspace:

After experiments are executed, Ramble can extract their figures of merit.

This is done through:

```
ramble workspace analyze
```

Ramble will process the output files described in an experiment's application.py, and extract success criteria and figures of merit.

These are then written to a file (e.g. results.txt) in the provided format (can be text, yaml, or json).

What other things can you do with Ramble?

Parameterizing Software Definitions

A common workflow in optimizing software is exploring how modifications to a software stack impact performance. Ramble's support of these workflows can be seen below:

```
ramble:
  …
  applications:
    wrfv3:
      workloads:
        CONUS_2p5km:
          experiments:
            '{mpi}_{partition}_{n_nodes}nodes':
              variables:
                n_nodes: [1, 2, 4, 8, 16, 32]
                mpi: ['impi', 'ompi']
                env_name: 'wrf-{mpi}'
              matrix:
              - n_nodes
              - mpi
```

```
[ramble:]
  …
  software:
    packages:
      wrf:
        pkg_spec: wrf@3.9.1.1
      impi:
        pkg_spec: intel-oneapi-mpi@2021.9
      ompi:
        pkg_spec: openmpi@4.1.4
    environments:
      wrf-{mpi}:
        packages:
        - wrf
        - '{mpi}'
```

Parameterizing Software Definitions

A common workflow in optimizing software is exploring how modifications to a software stack impact performance. Ramble's support of these workflows can be seen below:

```
ramble:
  …
  applications:
    wrfv3:
      workloads:
        CONUS_2p5km:
          experiments:
            '{mpi}_{partition}_{n_nodes}nodes':
              variables:
                n_nodes: [1, 2, 4, 8, 16, 32]
                mpi: ['impi', 'ompi']
                env_name: 'wrf-{mpi}'
              matrix:
              - n_nodes
              - mpi
```

```
[ramble:]
  …
  software:
    packages:
      wrf:
        pkg_spec: wrf@3.9.1.1
      impi:
        pkg_spec: intel-oneapi-mpi@2021.9
      ompi:
        pkg_spec: openmpi@4.1.4
  environments:
    wrf-{mpi}:
      packages:
      - wrf
      - '{mpi}'
```

Defines spack environments

Variable expansion

MPI packages

Performance Analysis Tools

Modifiers all composable definitions to edit the experiment behavior

```
ramble:
  …
  modifiers:
  - name: intel-aps
  applications:
    wrfv3:
      workloads:
        CONUS_2p5km:
          experiments:
            '{mpi}_{partition}_{n_nodes}nodes':
              variables:
                n_nodes: [1, 2, 4, 8, 16, 32]
                mpi: ['impi', 'ompi']
                env_name: 'wrf-{mpi}'
              matrix:
              - n_nodes
              - mpi
```

```
[ramble:]
  …
  software:
    packages:
      wrf:
        pkg_spec: wrf@3.9.1.1
      impi:
        pkg_spec: intel-oneapi-mpi@2021.9
      ompi:
        pkg_spec: openmpi@4.1.4
    environments:
      wrf-{mpi}:
        packages:
        - wrf
        - '{mpi}'
```

Performance Analysis Tools

Modifiers all composable definitions to edit the experiment behavior

```
ramble:
  …
  modifiers:
  - name: intel-aps
  applications:
    wrfv3:
      workloads:
        CONUS_2p5km:
          experiments:
            '{mpi}_{partition}_{n_nodes}nodes':
              variables:
                n_nodes: [1, 2, 4, 8, 16, 32]
                mpi: ['impi', 'ompi']
                env_name: 'wrf-{mpi}'
              matrix:
              - n_nodes
              - mpi
```

Performance Analysis Tool

```
[ramble:]
  …
  software:
    packages:
      wrf:
        pkg_spec: wrf@3.9.1.1
      impi:
        pkg_spec: intel-oneapi-mpi@2021.9
      ompi:
        pkg_spec: openmpi@4.1.4
    environments:
      wrf-{mpi}:
        packages:
        - wrf
        - '{mpi}'
```

See you at 4:30pm!

Coffee Break

Three Rs in HPC Benchmarking

1. Repeat
2. Replicate
3. Reproduce

Three Rs in HPC Benchmarking

1. Repeat: run on same system
2. Replicate
3. Reproduce

	Component	Repeat
(1)	System H/W	✓
(2)	System S/W	✓
(3)	Application	✓

- Verifying benchmark still builds and runs
- Handing over benchmarking tasks from one user to another
- Evaluating benchmark performance changes as system ages
- Running multiple trials of a benchmark for noise variability

Three Rs in HPC Benchmarking

1. Repeat
2. Replicate: run with a different software stack
3. Reproduce

	Component	Repeat	Replicate
(1)	System H/W	✓	✓
(2)	System S/W	✓	≈
(3)	Application	✓	✓

- Verifying benchmark still builds and runs with update system software
- Validating performance of updated system software
- Evaluating performance differences of different software substitutes (e.g., gcc or icc compilers)

Three Rs in HPC Benchmarking

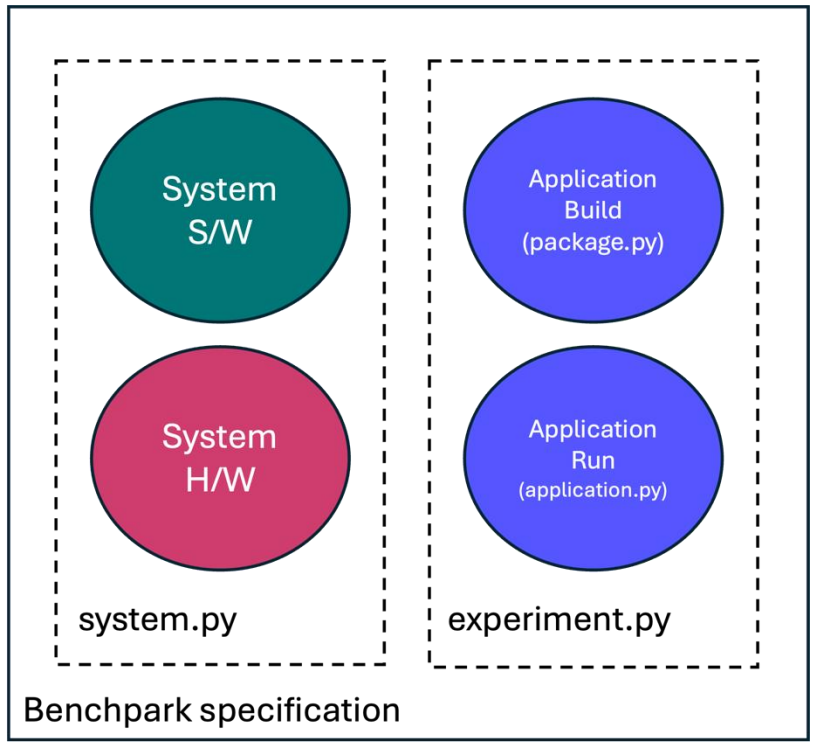
1. Repeat
2. Replicate
3. Reproduce: run on a different system

	Component	Repeat	Replicate	Reproduce
(1)	System H/W	✓	✓	X
(2)	System S/W	✓	≈	X
(3)	Application	✓	✓	≈

- Verifying benchmark builds and runs on diverse hardware
- Evaluating benchmark performance across different hardware
- Exploring potential benefits of hardware upgrades

Benchpark Specification for Three Rs

	Component	Repeat	Replicate	Reproduce
(1)	System H/W	✓	✓	X
(2)	System S/W	✓	≈	X
(3)	Application	✓	✓	≈



Benchpark Specification for the Three Rs

	Component	Repeat	Replicate	Reproduce
(1)	System H/W	✓	✓	X
(2)	System S/W	✓	≈	X
(3)	Application	✓	✓	≈

Repeat as a different user

```
sysa:~ user1$ benchpark system init --dest=sysa intel-cluster
       cluster=sysa
sysa:~ user1$ benchpark experiment init --dest=qs
       quicksilver +openmp +weak
sysa:~ user1$ benchpark setup ./qs ./sysa workspace/
```

```
sysa:~ user2$ benchpark system init --dest=sysa  intel-
       cluster cluster=sysa
sysa:~ user2$ benchpark experiment init --dest=qs
       quicksilver +openmp +weak
sysa:~ user2$ benchpark setup ./qs ./sysa workspace2/
```

Replicate with different SW stack

```
sysa:~ user1$ benchpark system init --dest=sysa1  intel-
       cluster cluster=sysa compiler=gcc1
sysa:~ user1$ benchpark experiment init --dest=qs
       quicksilver +openmp +weak
sysa:~ user1$ benchpark setup ./qs ./sysa1 workspace/
```

```
sysa:~ user2$ benchpark system init --dest=sysa2  intel-
       cluster cluster=sysa compiler=gcc2
sysa:~ user2$ benchpark experiment init --dest=qs
       quicksilver +openmp +weak
sysa:~ user2$ benchpark setup ./qs ./sysa2 workspace2/
```

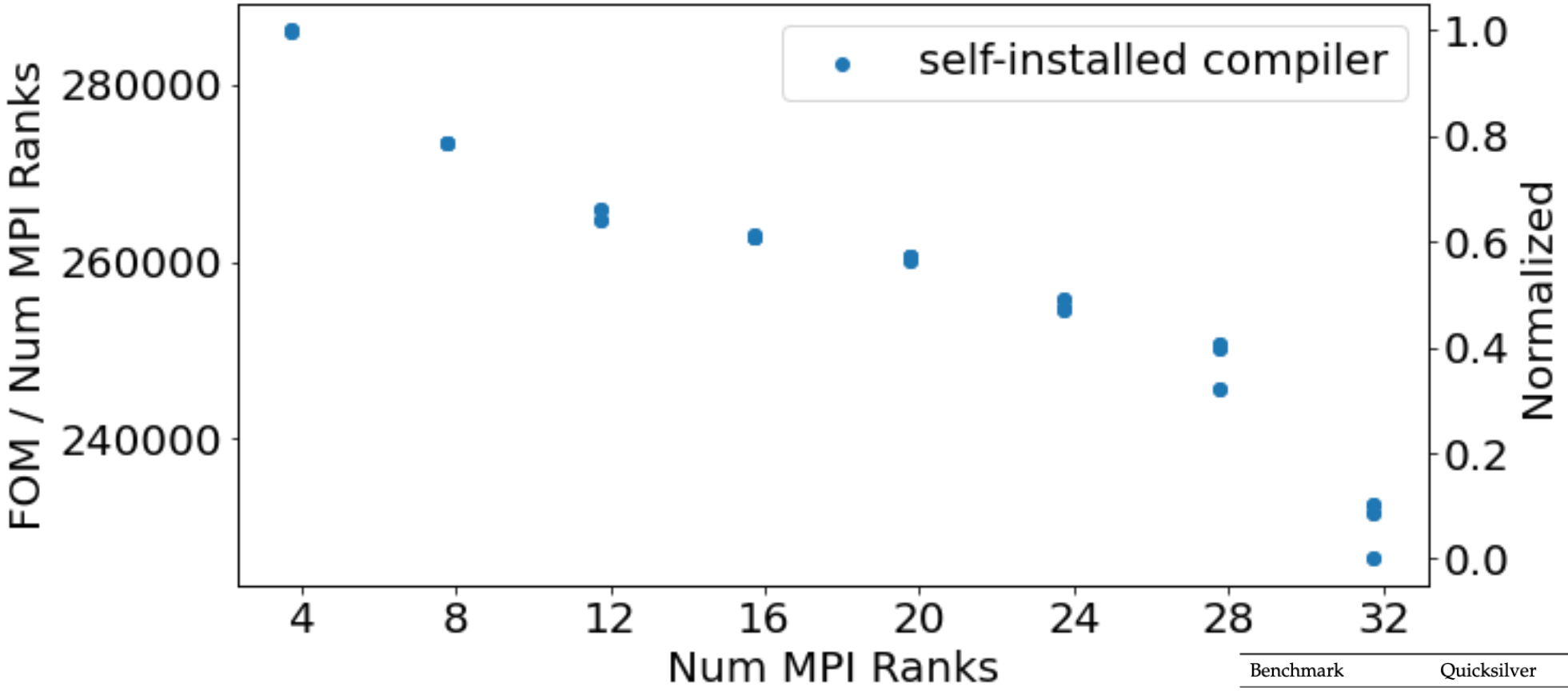
Reproduce on a different system

```
sysa:~ user1$ benchpark system init --dest=sysa  intel-
       cluster cluster=sysa
sysa:~ user1$ benchpark experiment init --dest=qs-openmp
       quicksilver +openmp +weak
sysa:~ user1$ benchpark setup ./qs-openmp ./sysa worksp/
```

```
sysb:~ user2$ benchpark system init --dest=sysb  ibm-cluster
sysb:~ user2$ benchpark experiment init --dest=qs-cuda
       quicksilver +cuda +weak
sysb:~ user2$ benchpark setup ./qs-cuda ./sysb work2/
```

Repeat runs of Quicksilver on Ruby using *same compiler*

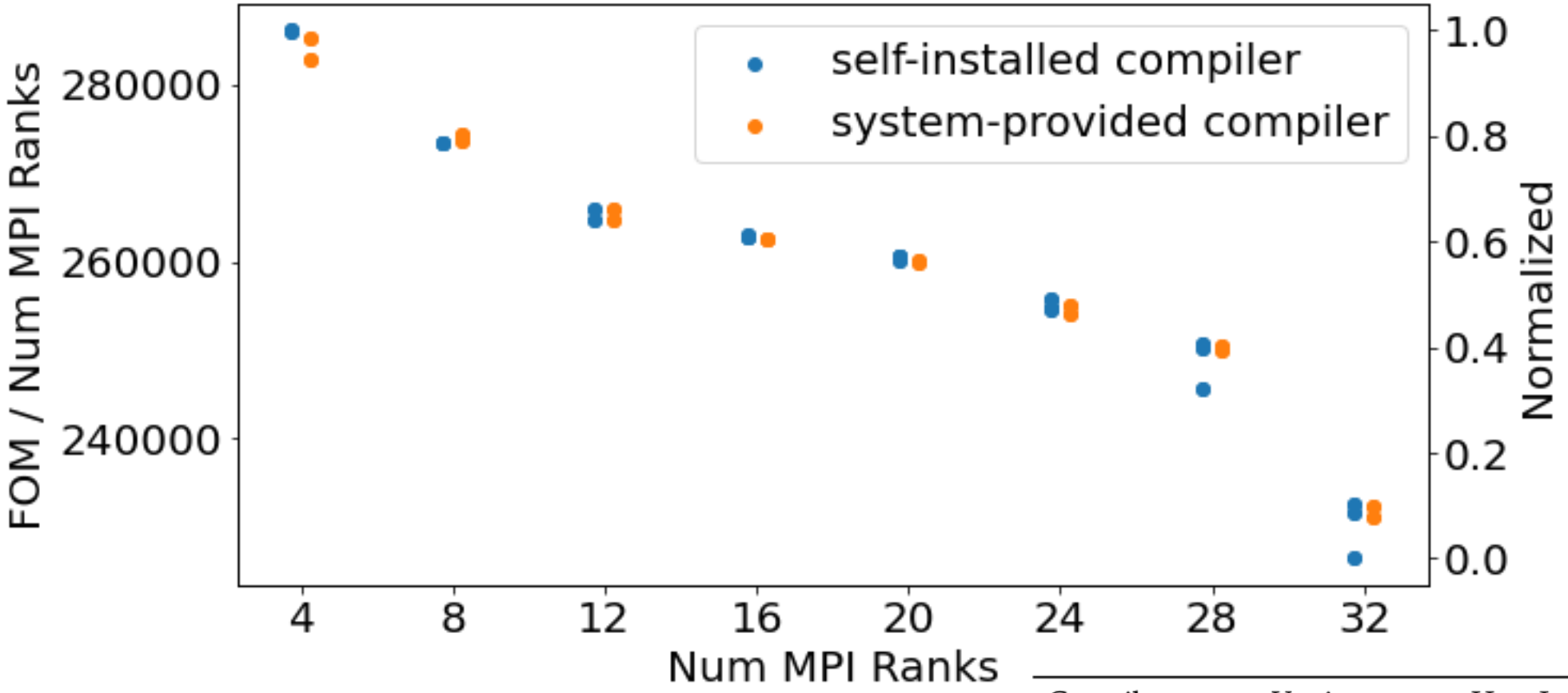
	Component	Repeat	Replicate	Reproduce
(1)	System H/W	✓	✓	X
(2)	System S/W	✓	≈	X
(3)	Application	✓	✓	≈



Benchmark	Quicksilver
Variants	+mpi, +openmp
Num Ranks	4, 8, 12, 16, 20, 24, 28, 32
Num Nodes	1
Scaling	weak
Repetitions	4 (Increase as appropriate for analysis)

Replicate runs of Quicksilver on Ruby with *different compilers*

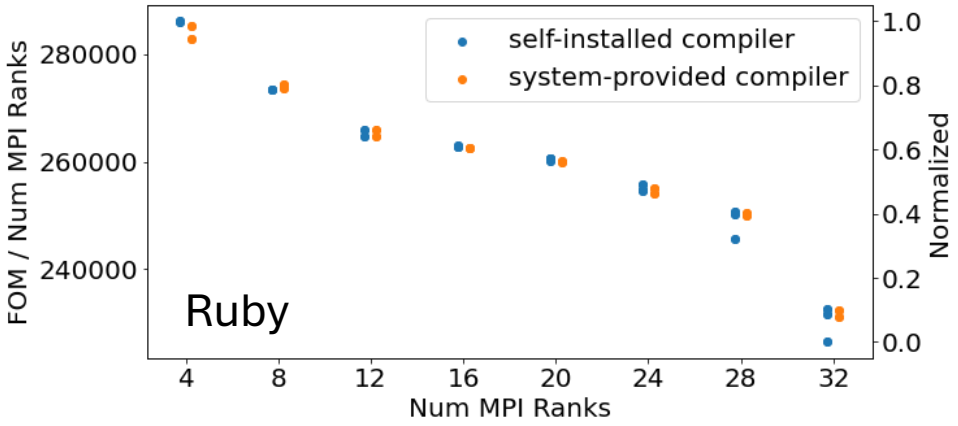
	Component	Repeat	Replicate	Reproduce
(1)	System H/W	✓	✓	X
(2)	System S/W	✓	≈	X
(3)	Application	✓	✓	≈

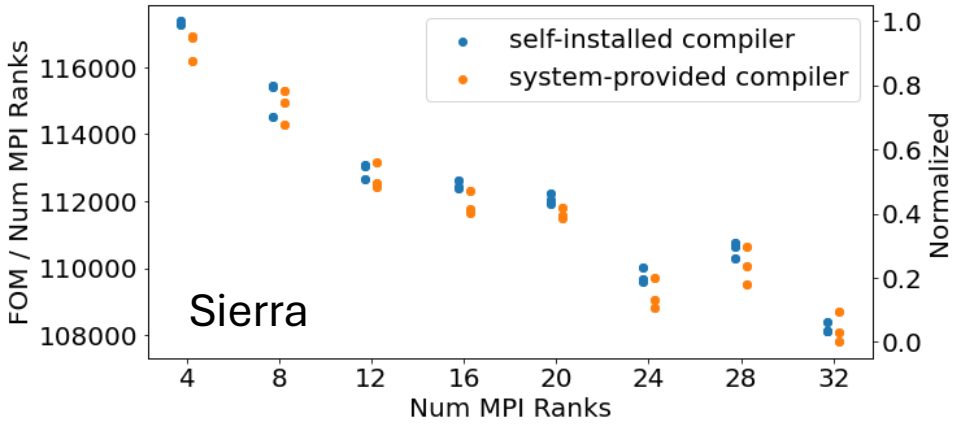


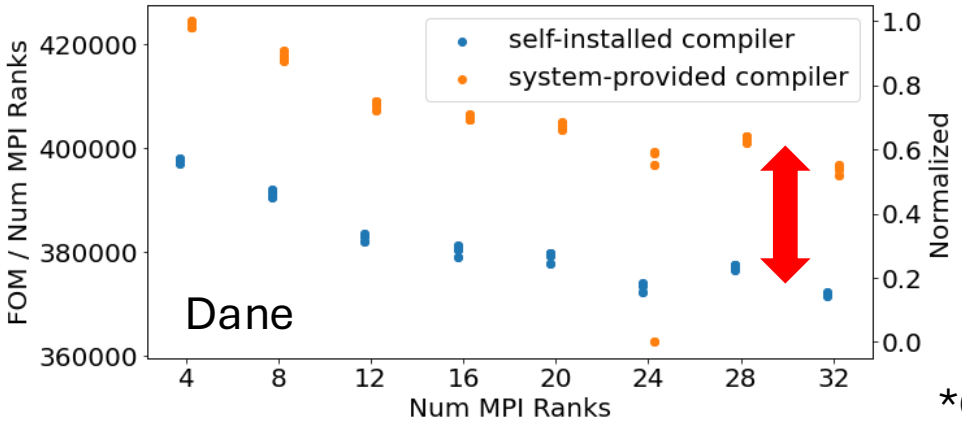
Compiler	Version	How Installed
gcc	12.1.1	system-provided
gcc	12.1.0	self-installed

Reproduce runs of Quicksilver across *different systems*

	Component	Repeat	Replicate	Reproduce
(1)	System H/W	✓	✓	X
(2)	System S/W	✓	≈	X
(3)	Application	✓	✓	≈







System	P-value	Statistically Significant?
Ruby	0.490	No
Sierra	0.349	No
Dane	0.00000105	Yes

System Name	CPU Hardware	Number of Cores
Ruby	Intel Cascade Lake	56
Sierra	IBM Power 9	44
Dane	Intel Sapphire Rapids	112

*Gap on Dane is statistically significant

Analysis of calltrees to understand performance difference using Thicket

```
0.000 main
└─ 0.134 qs.mainloop
   ├─ 0.019 MPI_Allreduce
   ├─ 0.228 cycleFinalize
   │  ├─ 0.003 MPI_Allreduce
   │  └─ 0.000 MPI_Wait
   ├─ 0.558 cycleInit
   │  ├─ 0.011 MPI_Allreduce
   │  └─ 0.022 MPI_Barrier
   └─ 0.022 cycleTracking
      ├─ 0.000 MPI_Cancel
      ├─ 0.001 MPI_Irecv
      ├─ 0.000 MPI_Wait
      ├─ 230.542 cycleTracking_Kernel
      ├─ 1.158 cycleTracking_MPI
      │  ├─ 0.018 MPI_Irecv
      │  ├─ 0.023 MPI_Isend
      │  ├─ 0.108 MPI_Test
      │  ├─ 0.046 MPI_Wait
      │  └─ 0.001 cycleTracking_Test_Done
      │     └─ 20.566 MPI_Allreduce
      └─ 0.000 cycleTracking_Test_Done
         └─ 0.000 MPI_Allreduce
```

Self-Installed Compiler

```
0.000 main
└─ 0.135 qs.mainloop
   ├─ 0.025 MPI_Allreduce
   ├─ 0.226 cycleFinalize
   │  ├─ 0.003 MPI_Allreduce
   │  └─ 0.000 MPI_Wait
   ├─ 0.559 cycleInit
   │  ├─ 0.014 MPI_Allreduce
   │  └─ 0.024 MPI_Barrier
   └─ 0.022 cycleTracking
      ├─ 0.000 MPI_Cancel
      ├─ 0.001 MPI_Irecv
      ├─ 0.000 MPI_Wait
      ├─ 219.612 cycleTracking_Kernel
      ├─ 1.165 cycleTracking_MPI
      │  ├─ 0.019 MPI_Irecv
      │  ├─ 0.023 MPI_Isend
      │  ├─ 0.106 MPI_Test
      │  ├─ 0.046 MPI_Wait
      │  └─ 0.001 cycleTracking_Test_Done
      │     └─ 19.478 MPI_Allreduce
      └─ 0.000 cycleTracking_Test_Done
         └─ 0.000 MPI_Allreduce
```

System-Provided Compiler

```
Legend (Metric: Avg time/rank (exc)
207.49 – 230.54
161.38 – 207.49
115.27 – 161.38
69.16 – 115.27
23.05 – 69.16
0.00 – 23.05
```

Takeaways

- Formalize how to repeat, replicate, and reproduce HPC benchmarking experiments
- Showcase how Benchpark enables reproducibility through specification of experiments and HPC systems
- Full record of the specification is preserved for introspection and reproducibility

Hands On Session 2
Adding an experiment for a benchmark

Follow script at http://software.llnl.gov/benchpark/add-an-experiment.html

Tutorial Instances: http://bit.ly/4kGQDlc

- We have an AWS instance for the hands-on component of this tutorial
- The instance provides:
  - Pre-installed Benchpark and required dependencies
  - Job scheduler

When logging in to the instance:

- Please use a unique username to avoid resource allocation conflicts
  - First initial followed by last name (e.g., John Doe would be jdoe)
- PW: hpctutorial25

s://software.llnl.gov/benchpark/tutorial-101.html

Performance Tools: Caliper

- Caliper is an 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
  - Supports MPI, OpenMP, CUDA, HIP, Kokkos, RAJA
- software.llnl.gov/Caliper

Performance Tools: Thicket

- Thicket is a toolkit for Exploratory Data Analysis
- Enables exploratory data analysis of multi-run data
- Compose data from diff. sources and types
  - Different scaling (e.g., strong, weak)
  - Different application parameters
  - Different compilers and optimization levels
  - Different hardware types (e.g., CPUs, GPUs)
  - Different performance tools
- Perform analysis on the thicket of runs
  - Manipulate the set of data
  - Visualize the dataset
  - Perform analysis on the data
  - Model data
  - Leverage third-party tools in the Python ecosystem

thicket.readthedocs.io

Ramble modifiers in Benchpark encapsulate reusable patterns to perform a specific configuration of an experiment

- Affinity modifier for pinning threads
- Allocation modifier to request resources
- Hwloc modifier to capture hardware topology
- Caliper modifier to profile an application

```
benchpark experiment init --dest=amg amg2023 --system=elcap –rocm workload=problem2 +strong caliper=mpi.time
```

Join us after the tutorial!

Tutorial material: http://software.llnl.gov/benchpark/tutorial-101.html

Connect with us on Spack slack

#benchpark-support

We want your feedback!

Contribute systems, benchmarks, experiments, and features on GitHub

slack.spack.io
#benchpark-support

★ Star us on GitHub!
github.com/llnl/benchpark