Towards Collaborative Continuous Benchmarking for HPC

Olga Pearce
Lawrence Livermore National Lab

Alec Scott
Lawrence Livermore National Lab

Gregory Becker
Lawrence Livermore National Lab

Riyaz Haque
Lawrence Livermore National Lab

Nathan Hanford
Lawrence Livermore National Lab

Stephanie Brink
Lawrence Livermore National Lab

Doug Jacobsen
Google

Heidi Poxon
Amazon Web Services

Jens Domke
RIKEN R-CCS

Todd Gamblin
Lawrence Livermore National Lab

ABSTRACT

Benchmarking is integral to procurement of HPC systems, communicating HPC center workloads to HPC vendors, and verifying performance of the delivered HPC systems. Currently, HPC benchmarking is manual and challenging at every step, posing a high barrier to entry, and hampering reproducibility of the benchmarks across different HPC systems. In this paper, we propose *collaborative continuous benchmarking* to enable functional reproducibility, automation, and community collaboration in HPC benchmarking. Recent progress in HPC automation allows us to consider previously unimaginable large-scale improvements to the HPC ecosystem. We define the minimal requirements for collaborative continuous benchmarking and develop a common language to streamline the interactions between HPC centers, vendors, and researchers. We demonstrate the initial implementation of collaborative continuous benchmarking, and introduce an open source continuous benchmarking repository, Benchpark, for community collaboration. We believe collaborative continuous benchmarking will help overcome the human bottleneck in HPC benchmarking, enabling better evaluation of our systems and enabling a more productive collaboration within the HPC community.

ACM Reference Format:
Olga Pearce, Alec Scott, Gregory Becker, Riyaz Haque, Nathan Hanford, Stephanie Brink, Doug Jacobsen, Heidi Poxon, Jens Domke, and Todd Gamblin. 2023. Towards Collaborative Continuous Benchmarking for HPC. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3624062.3624135

1 INTRODUCTION

The system lifecycle for an HPC system spans several years and includes stages related to procurement, acceptance, and service. **HPC system benchmarking** is an integral component throughout this process. During the procurement of a system, benchmarking is used to communicate HPC center workloads with HPC vendors and other organizations. Benchmarking enables performance modeling across different hardware. It also helps evaluate which of the proposed HPC systems will result in the best performance for a particular HPC center workload, and is useful for co-designing future HPC system procurements. During acceptance of an HPC system, benchmarking can be used to assess early access systems and the changing software stacks. Benchmarking is also critical for determining if the delivered system reaches the expected performance. And finally, once the system has been accepted and is in service, benchmarking is a useful tool for tracking system performance over time and diagnosing hardware failures.

HPC ecosystem players, such as application developers and HPC centers use benchmarking in their various roles. HPC application developers write benchmarks to communicate performance characteristics, as well as hardware and software requirements of their applications. HPC centers incorporate benchmarks into their system procurement and acceptance processes. HPC vendors use benchmarks to understand hardware and software requirements of the systems to propose or deliver. HPC researchers use benchmarks to study and model performance and propose new hardware, software, and middleware designs. Productive involvement of the many HPC ecosystem players is only possible if they are able to functionally reproduce the benchmarks.

The state-of-the-practice of HPC system benchmarking is manual, high maintenance, and challenging at every step. Beyond deciding the benchmark problem definition and writing the benchmark source code, various challenges arise:

- Build environments on each HPC system are different, and porting build scripts to new systems is manual;
- Execution environments on each HPC system are different, and porting execution scripts to new systems is manual;
- Hardware and software stacks on each HPC system are evolving (e.g., hardware upgrades, firmware upgrades, software stack upgrades), and therefore require manual updates to the build and execution scripts for the benchmark;
- Triggering builds and runs of the benchmark is manual, so benchmark results do not stay up-to-date; and
- Performance analysis of the benchmark results is manual, limiting how much of the performance is tracked.

olga@llnl.gov.

SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624135

The state-of-the-practice in benchmarking is fully manual and presents a high barrier to entry for the HPC players to engage in benchmarking and make their contributions to HPC. Additionally, when these players invest effort and resources into getting benchmarks to work, there is no mechanism for them to reproducibly share their work with others. As a result, benchmark maintenance falls behind any such efforts by invested players. We identify several **key observations** which underpin our work:

- The many HPC players in benchmarking need to be able to run the benchmarks in a functionally reproducible manner;
- Reproducibility is only possible when the process is sufficiently described to enable automation;
- Automation of benchmark building, running, and evaluation will enable more frequent testing of benchmarks, while lowering the barrier to entry for HPC players;
- Enabling collaborative maintenance of orthogonal concerns in benchmarking (from algorithms to systems) will enable the many players in HPC benchmarking to contribute to the maintenance, and enable broader testing and thus more value from benchmarking efforts.

We believe automated benchmarking in HPC is now possible because of improvements in automation across the industry. In recent years, HPC centers have developed automated systems for building and running software and have worked to bring continuous integration capabilities into the HPC center, which allows us to consider the large-scale improvements to the HPC ecosystem made possible by applying these methods which are new to HPC but have been in production within other computing ecosystems for years.

We propose *continuous benchmarking*, an approach to defining benchmarking in a reproducible way, thus enabling both automation and reproducibility. We leverage continuous integration (CI) to test each step of benchmarking, including source code, inputs, building, running, and evaluation. Our approach enables maintenance of these components in an orthogonal way, thus distributing the workload of benchmark maintenance among the HPC players and enabling community contribution to the benchmarking effort. Our continuous benchmarking approach will lead to better software practices, easier maintenance, better benchmarks, and software sustainability, leading to better understanding and thus better co-design of HPC systems.

This paper makes the following contributions:

- Determination of the minimal requirements for *continuous benchmarking*;
- Definition of the *continuous benchmark suite* and its open source components;
- Design of a portable benchpark repository, Benchpark [22], for community collaboration on benchmarking with minimized human effort required via maximizing reuse by orthogonalizing components;
- Demonstration of the initial implementation.

Section 2 introduces **Benchpark**, our proposed architecture for collaborative continuous benchmarking. Section 3 describes the automation components we leverage in Benchpark (Spack 3.1, Ramble 3.2, CI 3.3). Section 4 demonstrates the Benchpark workflow for running experiments, and for adding new benchmarks and systems to Benchpark, and Section 5 discusses our next steps. Section 6 describes related efforts in the area. Section 7 discusses benchmark sustainability, collaborative maintenance, and the role of cloud.

2 BENCHPARK: AN ARCHITECTURE FOR CONTINUOUS BENCHMARKING

We define *continuous benchmarking* as a fully automated mechanism for evaluating HPC benchmark performance on specified HPC systems. Continuous benchmarking will enable testing the impact of software changes to a wide variety of components, including benchmark source code and input parameters, build instructions, execution instructions, and CI testing. Being able to automate the benchmarking process and store the results of the evaluation before and after any changes to hardware, firmware, drivers, or software will provide a deeper understanding of the impact of these changes.

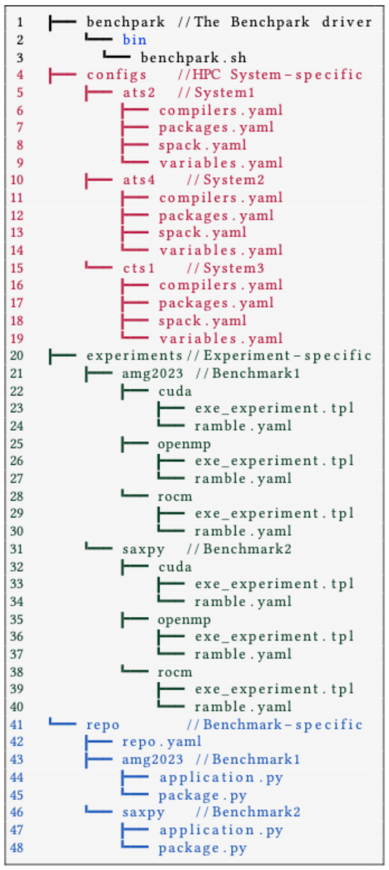
A continuous benchmarking system must be able to track the location of versioned benchmark source code. It needs to track benchmark inputs and run parameters, which may be system dependent. It needs reproducible methods for building and running benchmarks. Those methods need to be integrated with the inputs and run parameters. Build and run methods need to be easily portable across systems, or the continuous benchmarking framework of one architecture will be of limited value to any other system. Finally, it needs to automate the characterization of performance results from the benchmarks. All of these systems all must follow data integrity standards to avoid deterioration of the working code with minimal human oversight and intervention.

We introduce Benchpark [22], a continuous benchmarking framework designed for automation of testing and a vehicle for collaboration of HPC players from across the world where human input and validation is currently necessary. Benchpark is an infrastructure-as-code project combining a variety of open source tools into a fully specified system for tracking benchmark performance across a variety of systems, across multiple HPC centers, and across arbitrary choices of benchmarks. Table 1 presents the components of Benchpark, and implementation choices we made to enable orthogonalization of benchmarks, systems, and experiments. In Table 1 and the remainder of the paper, we adopt the following color scheme:

- **Benchmark-specific:** Specification for each benchmark, without any system-specific information. Benchpark requires exactly one such specification per *benchmark*, including where to find benchmark source and build specs (package.py), benchmark input and run specification (application.py).
- **HPC System-specific:** Specification for each system, without any benchmark-specific information. Benchpark requires exactly one such specification per *system*, including system software (compilers.yaml,packages.yaml), build config files (spack.yaml), scheduler and launcher (variables.yaml), and (optional) performance counters to collect.
- **Experiment-specific:** We define experiments as the instances of the benchmark the user wants to run on a given system. Examples include a strong-scaling study of a benchmark (a set of experiments with the same problem size, scaled on a different number of resources) on a CPU+GPU heterogeneous system using the GPU for the main computation. The complete experiment specification is specified in spack, experiments, and success_criteria sections of ramble.yaml.

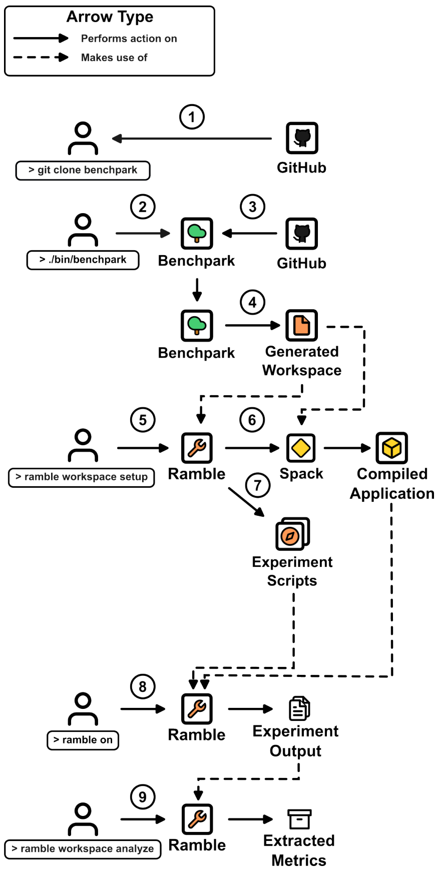
Table 1: Components of *Benchpark*, a collaborative continuous benchmark suite

	Component	Benchmark-specific	HPC System-specific	Experiment-specific
1	Source code	package.py	archspec (Sec. 3.1.3)	ramble.yaml: spack
2	Build instructions	package.py	Spack config. files, spack.yaml	ramble.yaml: spack
3	Benchmark input	application.py, (optional) data	variables.yaml	ramble.yaml: experiments
4	Run instructions	application.py	variables.yaml: scheduler, launcher	ramble.yaml: experiments
5	Experiment evaluation	(optional) application.py	(optional) hardware counters, etc.	ramble.yaml: success_criteria
6	CI testing	.gitlab-ci.yml	Hubcast@LLNL/RIKEN/AWS/...	Benchpark executable



```
├── benchpark  //The Benchpark driver
    └── bin
        └── benchpark.sh
├── configs      //HPC System-specific
    ├── ats2     //System1
        ├── compilers.yaml
        ├── packages.yaml
        ├── spack.yaml
        └── variables.yaml
    ├── ats4      //System2
        ├── compilers.yaml
        ├── packages.yaml
        ├── spack.yaml
        └── variables.yaml
    └── cts1      //System3
        ├── compilers.yaml
        ├── packages.yaml
        ├── spack.yaml
        └── variables.yaml
├── experiments //Experiment-specific
    ├── amg2023  //Benchmark1
        ├── cuda
            ├── exe_experiment.tpl
            └── ramble.yaml
        ├── openmp
            ├── exe_experiment.tpl
            └── ramble.yaml
        └── rocm
            ├── exe_experiment.tpl
            └── ramble.yaml
    └── saxpy    //Benchmark2
        ├── cuda
            ├── exe_experiment.tpl
            └── ramble.yaml
        ├── openmp
            ├── exe_experiment.tpl
            └── ramble.yaml
        └── rocm
            ├── exe_experiment.tpl
            └── ramble.yaml
└── repo          //Benchmark-specific
    ├── repo.yaml
    ├── amg2023  //Benchmark1
        ├── application.py
        └── package.py
    └── saxpy      //Benchmark2
        ├── application.py
        └── package.py
```

(a) Benchpark directory structure



(b) Benchpark component interaction

1: User clones Benchpark repository
> git clone benchpark

2: User runs Benchpark with a system profile and benchmark suite template
> ./bin/benchpark $experiment
> $system $workspace_dir

3: Benchpark clones Spack and Ramble

4: Benchpark generates workspace config

5: User calls Ramble within workspace
> ramble workspace setup

6: Ramble uses Spack to build each benchmark

7: Ramble renders batch experiment scripts

8: User calls Ramble to submit batch experiment scripts, Ramble executes each benchmark with experiment scripts
> ramble on

9: User calls Ramble to analyze the benchmark output and extract metrics
> ramble workspace analyze

(c) Workflow: Run Benchpark experiments

Figure 1: Benchpark components and user workflow for running experiments using Benchpark

An overview of Benchpark is shown in Figure 1, including the directory structure (Figure 1a), component interaction (Figure 1b), and the workflow of running experiments using Benchpark (Figure 1c). Benchpark heavily leverages Spack (see Section 3.1) and Ramble (see Section 3.2) for its functionality.

The Benchpark repository contains four subdirectories (Figure 1a): 1) `benchpark` directory contains the driver script, 2) `configs` contains the specific HPC system details, 3) `experiments` contains the test descriptions for each of the benchmarks, 4) `repo` directory is a construct from Spack and Ramble and is for overlay information not contained in the upstream Spack or Ramble repositories.

Figure 1b and Figure 1c illustrate how the different Benchpark components interact. First, the user clones the Benchpark repository from GitHub. Then, the user runs Benchpark with a system profile and benchmark suite template file. Internally, Benchpark clones Spack and Ramble from GitHub, then creates a workspace. Next, the user calls Ramble within the workspace, and Ramble builds each benchmark through Spack and generates batch experiment scripts. Lastly, the user calls Ramble again to perform benchmark analysis and extract performance metrics.

3 HPC AUTOMATION ENABLES BENCHPARK

Recent progress in HPC automation allows us to consider large-scale improvements in HPC benchmarking. We next describe the technologies we leverage in Benchpark: Spack (for reproducible build instructions), Ramble (for reproducible run instructions), and the CI technologies for automation of Benchpark.

3.1 Reproducible Build Instructions

Spack [10, 18] is a package manager for HPC. Spack runs entirely in user-space, and allows for combinatorial versioning of installations across all facets of the build configuration space. Spack consists of four primary components:

(1) The Spec syntax, to specify the user constraints on a build, called *abstract specs*;
(2) The *concretizer*, an algorithm that takes abstract specs and fills in remaining choice points for the build space, producing *concrete specs*;
(3) Package files, which define the build space for the package and provide package installation recipes templatized by the concrete spec output from the concretizer;
(4) The installation engine, which handles installing packages from source or binary cache.

Spack adds an *environment* behavior similar to Python's `virtual-env` [28], following a manifest-and-lock model similar to Bundler [17] and other package managers [13, 14, 27]. In Spack, environment manifests are treated as user input, and the output of the concretizer is written to a lockfile. Algorithm 2 shows an example workflow.

```
spack env create --dir .
spack env activate --dir .
spack add amg2023+caliper
spack --config-scope /path/to/configs concretize
spack install
```

Figure 2: Spack environment workflow

3.1.1 *Creating a Spack environment.*

A Spack environment manifest provides a list of abstract specs, and can be combined with configuration. In Benchpark, we list the packages needed to run our benchmarks in the environment manifest as abstract specs. Spack environments are written in YAML. Spack environments can be managed by Spack or stored independently from Spack, as defined in the manifest in spack.yaml (Figure 3). In Benchpark, the list of abstract specs for the environment is configured in Ramble 3.2. The environment is instantiated with Spack, and concretized with configuration to target the current system.

```
spack:
  specs: [amg2023+caliper]
  concretizer:
    unify: true
  view: true
```

Figure 3: spack.yaml: A simple Spack environment manifest combining an abstract spec `amg2023+caliper` with configuration for concretizer and view

3.1.2 *Importing local configuration.*

Spack provides a highly customizable configuration to tailor the behavior of Spack to the local system. Configurable parameters in Spack include the location of available compilers, externally installed packages to use for dependencies, and detailed preferences for concretizer decisions for build options. Benchpark specifies per-system directories with known Spack configurations, which can be tailored for both system architecture and HPC site policy.

Benchpark's Spack configurations specify the locations of system software and compilers (which are determined by HPC facility staff), as well as preferred versions of dependencies, which are determined by specialists with knowledge of particular benchmarks. An example Spack configuration is shown in Figure 4.

```
packages:
  blas:
    externals:
    - spec: intel-oneapi-mkl@2022.1.0
      prefix: /path/to/intel-oneapi-mkl
    buildable: false
  mpi:
    externals:
    - spec: mvapich2@2.3.7-gcc12.1.1-magic
      prefix: /path/to/mvapich2
    buildable: false
```

Figure 4: A system-specific spack.yaml configures Spack to use BLAS and MPI libraries installed on the system

3.1.3 *Archspec.*

Spack uses Archspec [7] to build executables for Benchpark. Archspec is a library for detecting and determining compiler flags for various CPU architectures. Originally developed in Spack, Archspec is used by Spack 1) to tailor build recipes to the target architecture, and 2) to determine the system architecture, and to determine which source to fetch for packages with alternate source code for different architectures.

3.2 Reproducible Run Instructions

Ramble [12, 15] is a Python experimentation framework enabling the creation of large sets of experiments with concise YAML files. It is heavily based off of Spack's infrastructure, and provides domain-specific languages for describing how experiments can be created for applications as well as abstract modifiers for changing the behavior of the experiments in repeatable ways.

One of the goals of Ramble is to improve replicability of experimental results, and increase productivity when performing large sets of experiments. The primary entry point for users is a `ramble workspace`, which is a self contained directory representing a set of experiments. A workspace is configured with a YAML file, and at least one template execution script.

Figure 5 shows the Ramble workflow for creating reproducible experiments; the steps are described in subsequent sections.

3.2.1 *Workspace Create.*

The `workspace_create` constructs a self-contained directory for a set of experiments, with a directory structure for experiment directories, required input files, and software environment definitions.

3.2.2 *Workspace Edit.*

The `ramble workspace edit` command opens `ramble.yaml`, the primary configuration file for the workspace,

```
ramble workspace create
ramble workspace edit
ramble workspace setup
ramble on
ramble workspace analyze
```

Figure 5: Ramble experiments workflow

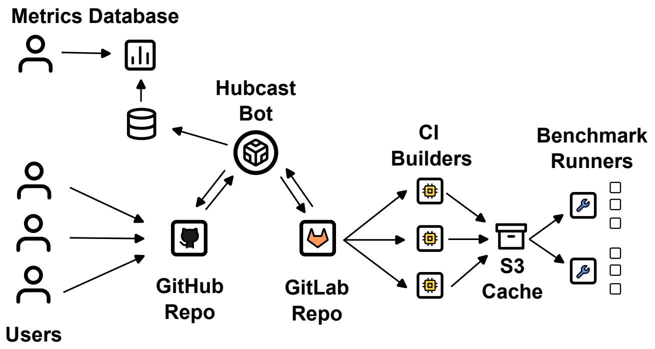


Figure 6: Benchpark automation workflow

in an editor for the user to manipulate. The user defines what experiments to run, and whether the experiments should be executed sequentially or submitted to a batch scheduler (or a mix of the two).

3.2.3 Workspace Setup. The `ramble workspace setup` command sets up the workspace via the following actions:

- Ensuring any required compilers are installed / accessible;
- Downloading source and input files;
- Installing any required software with Spack (see Section 3.1);
- Creating execution directories for every experiment;
- Generating files from every template file in the `configs`.

Once setup is complete, all experiments are ready for execution.

3.2.4 Ramble On. The `ramble on` command is used to execute all of the experiments. Its exact behavior depends on the configuration within the workspace, but it supports many execution methods.

3.2.5 Workspace Analyze. Once all of the jobs within the workspace are complete, the `ramble workspace analyze` command can be used to extract all of the figures of merit for the experiments.

3.3 Benchpark Automation

The automation workflow in Benchpark is shown in Figure 6. Benchpark relies on GitLab CI through Hubcast and Jacamar to manage the continuous integration task of continuous benchmarking, including scalability of runs and delegating user-login information to individual HPC centers for security. We leverage GitLab CI to test each component of Benchpark, including source code, inputs, builds, run scripts, and evaluation on systems both in the cloud and hosted locally. GitLab was chosen over native GitHub runners due to GitLab's popularity at HPC centers (because of compatibility with Jacamar) and because it can be used in private HPC environments for smaller communities.

3.3.1 Hubcast. Hubcast [23] is a secure mirroring application for GitHub and GitLab repositories. Unlike GitLab's built-in mirroring functionality, Hubcast allows untrusted pull requests from forks to be mirrored to a GitLab once they pass a configured set of security criteria. Once mirrored, these pull request branches may then be used for GitLab CI and the status of any workflows will be reported back to GitHub via Hubcast.

We use Hubcast in Benchpark to securely run user contributions via GitLab CI on both local and cloud resources while maintaining the canonical repository on GitHub. To prevent untrusted code from running on HPC resources, a pull request must be reviewed and approved by a site and system administrator, before Hubcast will mirror the commit to GitLab, GitLab CI will begin executing, and the status will be streamed back through Hubcast to show as a native status check on the pull request on GitHub.

3.3.2 Jacamar CI. Jacamar [8] is a custom executor for GitLab CI runners in HPC environments. Instead of running multiple CI jobs all under a single service user, Jacamar uses `setuid` to execute jobs as the user who triggered them. This allows the actions of a job to be more easily tied back to the user who initiated it and prevent the creation of additional service accounts.

We use Jacamar within Benchpark to improve system security and limit the actions possible by CI jobs. If a job is submitted by a user without an account at a participating site, the job will be run as the user who approved the pull request further improving logging and audit checks.

4 ADDING BENCHMARKS TO BENCHPARK

To add a benchmark to Benchpark, a full specification of the benchmark, its build, and its run instructions for at least one platform is required. Similarly, one must give a full specification of the system to add the system to Benchpark. To demonstrate the addition of benchmarks and systems to Benchpark, we will detail our specification of the following two benchmarks in Benchpark:

(1) A `saxpy` micro-benchmark;
(2) AMG2023 [21];

These Benchpark benchmarks currently build & run on 3 systems:

(1) **cts1:** a CPU-only system (Intel Xeon),
(2) **ats2:** IBM Power9+NVIDIA V100 CPU/GPU hybrid system;
(3) **ats4 EAS:** AMD Trento+MI-250X CPU/GPU hybrid system.

We demonstrate our implementation of the Benchpark components required for continuous benchmarking (Table 1) using the `saxpy` micro-benchmark as an example.

4.1 Benchmark Source

Figure 7 illustrates a snippet of the `saxpy` source code: a single kernel ported to the target architecture.

```
void saxpy_kernel(float* r, float* x, float* y, int size){
  for (int i = 0; i < size; ++i) {
    r[i] = A * x[i] + y[i];
  }
}
```

Figure 7: Source code snippet of saxpy_kernel

4.2 Benchmark Input

Our implementation of `saxpy` takes in one input parameter (size) and generates arrays of the input size in the `main` function prior to calling the kernel; more complex benchmarks may take in more parameters and/or input files. Benchmark input must be specified in Ramble's application.py (lines 6-8 in Figure 8), with further parameterization specified in the experiments section of ramble.yaml (line 20 in Figure 10) to specify the specific experiment of interest and meet any further target machine constraints, e.g., the problem size must fit into the system memory. Our full saxpy implementation also contains references to a variety of APIs, such as MPI, so those parameters are specified as well.

4.3 Build Instructions

Spack package.py (Figure 11) contains application-specific Spack instructions required to build the benchmark (see Section 3.1). The specification in package.py does not contain any system-specific information, and works across systems. To build an executable on a given system, we also need system-specific Spack configuration files, i.e., compilers.yaml (compiler definitions) and packages.yaml (package definitions). The system-specific spack.yaml (Figure 9) specifies compiler and package versions to use on the given system, which feeds into the experiment-specific ramble.yaml (line 3 and lines 25-34 in Figure 10) to couple the application and the system configuration and produce a concrete package definition.

```
1   class Saxpy(SpackApplication):
2     name = "saxpy"
3     ...
4     executable('p', 'saxpy -n {n}', use_mpi=True)
5     workload('problem', executables=['p'])
6     workload_variable('n', default='1',
7           description='problem size',
8           workloads=['problem'])
9     figure_of_merit("success",
10          fom_regex=r'(?P<done>Kernel done)',
11          group_name='done', units='')
12    success_criteria('pass',
13          mode='string', match=r'Kernel done',
14          file='{experiment_run_dir}/{experiment_name}.out')
```

Figure 8: Ramble application.py

```
spack:
  packages:
    default-compiler:
      spack_spec: gcc@12.1.1
    default-mpi:
      spack_spec: mvapich2@2.3.7-gcc12.1.1
    gcc1211:
      spack_spec: gcc@12.1.1
    lapack:
      spack_spec: intel-oneapi-mkl@2022.1.0
    mpi-compilers:
      spack_spec: mvapich2@2.3.7-compilers
```

Figure 9: Ramble spack.yaml

4.4 Run Instructions

To fully specify an experiment, defined as a given application running on a given system, in a configuration desired by the user (e.g., scaling study), we need the following specifications:

(1) application-specific application.py;
(2) system-specific variables.py;
(3) experiment-specific execute_experiments.tpl.

```
1   ramble:
2     include:
3     - ./configs/spack.yaml
4     - ./configs/variables.yaml
5     config:
6       deprecated: true
7       spack_flags:
8         install: '--add --keep-stage'
9         concretize: '-U -f'
10    applications:
11      saxpy:
12        workloads:
13          problem:
14            env_vars:
15              set:
16                OMP_NUM_THREADS: '{n_threads}'
17            variables:
18              n_ranks: '8'
19              batch_time: '120'
20            experiments:
21              saxpy_{n}_{n_nodes}_{n_ranks}_{n_threads}:
22                variables:
23                  processes_per_node: ['8', '4']
24                  n_nodes: ['1', '2']
25                  n_threads: ['2', '4']
26                  n: ['512', '1024']
27                matrices:
28                  - size_threads:
29                    - n
30                    - n_threads
31    spack:
32      packages:
33        saxpy:
34          spack_spec: saxpy@1.0.0 +openmp ^cmake@3.23.1
35          compiler: default-compiler
36      environments:
37        saxpy:
38          packages:
39          - default-mpi
40          - saxpy
```

Figure 10: Ramble ramble.yaml. Compiler and packages sections point at the definitions in spack.yaml (Figure 9).

```
class Saxpy(CMakePackage, CudaPackage, ROCmPackage):
  """Test saxpy problem."""
  ...
  version('1.0.0')

  variant("openmp", default=True, description="OpenMP")

  def cmake_args(self):
    spec = self.spec
    args = []

    if '+openmp' in spec:
      args.append('-DUSE_OPENMP=ON')
      ...

    if '+cuda' in spec:
      args.append('-DUSE_CUDA=ON')
      ...

    if '+rocm' in spec:
      args.append('-DUSE_HIP=ON')
      ...

   return args
```

Figure 11: Spack package.py

Ramble's application.py provides Ramble instructions for executing the experiment (lines 4-8 in Figure 8) and evaluating the outcome of the experiment (lines 9-14 in Figure 8). Similarly to Spack's package.py, Ramble's application.py is application-specific, and does not contain any system-specific information.

```
variables:
  mpi_command: 'srun -N {n_nodes} -n {n_ranks}'
  batch_submit: 'sbatch {execute_experiment}'
  batch_nodes: '#SBATCH -N {n_nodes}'
  batch_ranks: '#SBATCH -n {n_ranks}'
  batch_timeout: '#SBATCH -t {batch_time}:00'
  compilers: [gcc1211, intel202160classic]
```

Figure 12: Ramble variables.yaml

```
#!/bin/bash
{batch_nodes}
{batch_ranks}
cd {experiment_run_dir}
{spack_setup}
{command}
```

Figure 13: Ramble execute_experiment.tpl template

The necessary system-specific scheduler and launcher commands are defined in variables.yaml (Figure 12).

The necessary experiment-specific input parameters and configuration options are defined in ramble.yaml (lines 5-24 in Figure 10), and are used to generate a set of concrete experiments. The "matrices" section is the syntactic sugar that allows a user to define a cross-product of the variables to generate a set of experiments (https://googlecloudplatform.github.io/ramble/workspace_config.html#variable-matrices).

Lastly, we use a template file, execute_experiment.tpl (Figure 13), to generate the runtime script for each individual experiment with the correct options instantiated from ramble.yaml and variables.yaml.

4.5 Performance Evaluation and FOM

After performing a set of experiments, we use `ramble workspace analyze` to evaluate the outcomes. Ramble provides several ways to define success/failure and figures of merit (FOM); application-specific criteria can be defined in application.py (lines 9-14 in Figure 8 in our `saxpy` example), or for individual experiments in ramble.yaml (Figure 10). Ramble also provides the `modifier` construct to capture architecture-specific FOMs (e.g., hardware counters); we are currently working on the implementation of these more advanced evaluation techniques for our benchmarks.

5 FUTURE WORK

One of the goals of this work is to enable performance analysis and modeling of our benchmarks across many systems. To that end, we would like to enable our collaborators to contribute the performance results of the benchmarks as they execute them on their systems. Benchpark produces an exact specification of the experiments, including application-specific, system-specific, and experiment-specific manifests that enable functional reproducibility of these experiments. Storing the Benchpark manifest with the performance results will enable introspection into benchmark performance across systems and time. We are also looking into creating a dashboard for the Benchpark results, which would provide a quick glance of the multi-dimensional performance data for our benchmarks. The interactive dashboard could be designed with some pre-built plots and visualizations, and the user could toggle data on/off as needed. We are working to understand the needs of the HPC benchmarking community, along with dashboards of other efforts (e.g., MLCommons [26]), to propose a dashboard design.

For performance measurements such as function-level timings and GPU performance counters, we plan to annotate the benchmarks with Caliper [2, 3, 19], a portable performance profiling library for HPC applications. After we gain experience with the automated generation of benchmark performance metrics, we will look into enabling optional use of different performance analysis tools. Caliper can be configured to use always-on profiling, enabling collection of performance profiles for each run of our benchmark under different build settings or execution contexts. We will use Adiak [20] to collect metadata related to the build settings and execution contexts, enabling filtering and sorting of collected profiles.

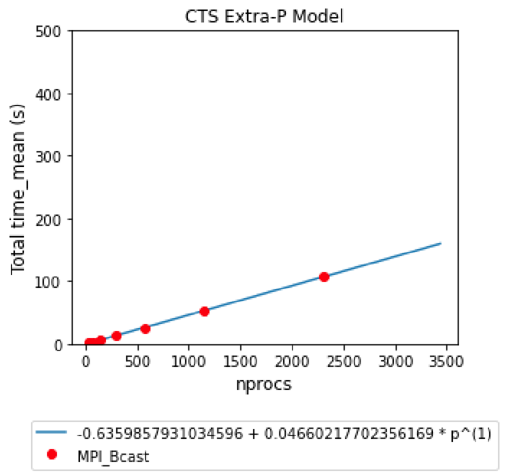


Figure 14: Extra-P model for performance of a function in one of our applications. Red dots represent performance measurements of an MPI_Bcast function on the CTS architecture. The blue line is a scaling function computed by Extra-P from the performance measurements.

Thicket [5, 24] is a python-based toolkit for exploratory data analysis (EDA) of multi-dimensional performance data, and can be used to analyze the Caliper-generated performance profiles. Thicket composes performance data from multiple performance profiles potentially generated at different scales, on different architectures, using different versions of dependencies, and by different tools. With Thicket, we can programmatically analyze the multi-dimensional performance data with scripts. An example of an EDA analysis is shown in Figure 14, which shows an analytical performance model computed by Extra-P [6].

In addition to collecting benchmark performance results from collaborators, we will look at collecting metrics on benchmark usage (which codes in Benchpark are accessed most heavily, which have been contributed to most recently, etc.). The need or value for a specific benchmark can change as science, algorithms, or computing needs change, and understanding which benchmarks are most relevant to the community can also improve procurement, vendor, and system monitoring productivity.

6 RELATED WORK

Continuous benchmarking of backbone network performance between HPC centers has been available for decades [29]. However, continuously benchmarking HPC systems themselves has proven difficult for a variety of reasons. For example, HPC applications may have vastly different characteristics, such as strong-scaling vs. weak-scaling applications, and nearest-neighbor exchanges vs. collective operations. Furthermore, the underlying hardware can vary widely between HPC systems, and many of these differences cannot easily be abstracted away by an operating system providing protocol abstraction layers above the hardware, as is the case in inter-center networking.

Nevertheless, application-specific and domain-specific continuous benchmarking systems have been deployed. Ginkgo library [1] has an automated performance evaluation framework. While such an approach could be adapted for different code-bases on a case-by-case basis, our approach seeks to synthesize the installation, dependency management, and results of multiple benchmarks, mini-applications, and proxy applications in a single framework with a single interface while representing all dependencies, from the application to the hardware.

There are also repeatable benchmark frameworks such as Pavilion [16] and JUBE [4], which are targeted at HPC systems. Both are capable of storing and organizing benchmark output, and critically, they are capable of interfacing with HPC system schedulers. However, neither of these has the ability to manage benchmarks across many systems with heterogenous dependencies as they don't directly leverage a package manager.

The Machine Learning Commons (MLCommons) [25, 26] has standardized a set of benchmarks specific to machine learning applications in their MLPerf benchmark suite. In contrast, our continuous benchmarking system is designed to run a much larger set of benchmarks across a large variety of systems with vastly different dependencies. Most of MLPerf is aimed at benchmarking cloud offerings, similarly to PerfKit [11].

All of these approaches share some fundamental commonalities with our approach. They are designed to be reproducible, they can be run continuously as the underlying software changes, and they offer rich introspection into how the benchmarks were run, along with rich performance data. In contrast, our approach to continuous benchmarking focuses more on leveraging package management and federated CI technologies that have only recently proliferated in order to address a much larger set of benchmarking needs from developers, HPC centers, and hardware vendors.

7 BENCHMARK SUSTAINABILITY

Community-driven approaches for software sustainability, such as Spack, show that many can benefit by contributing their knowledge to a single repository, assembling the collective knowledge—there in the form of build scripts and dependencies—of the entire HPC community. Our Benchpark approach is the logical next step for benchmarking, because it allows broad community contribution, limits the divergence between real science code and benchmark code, and lowers the barrier of entry for non-experts by providing build and run recipes. These features should lower the human labor involved in designing, maintaining, and running benchmarks. In the past, (procurement) benchmarks have been very much a one-off or fairly static code base. Every few years, when a new system is in planing, a few computer scientists try to find a number of representative applications from a sea of domain scientists and code bases running on their system. This information is hardly shared across centers and nations, with notable exceptions of the ECP Proxy Applications [9] which represents the interest of multiple US centers, simultaneously. Benchpark allows the individual benchmarks to be closer to their Spack counterparts and being updates over time, but also being "frozen" in time for procurement purposes. Therefore, the computer scientists can focus more on analyzing the benchmarks and track system performance over time, while the domain scientists do not need to periodically (re-)create special-purpose copies of their code (or fragments thereof) which do not serve them any immediate purpose.

7.1 Benchpark for Collaboration

Besides our indications for software sustainability—of benchmarks in particular—we also see a clear role of Benchpark in collaborations between supercomputing centers, HPC/cloud vendors, and/or nations. For example, during our collaboration on Benchpark we moved a few simple benchmark kernels between an on-premise supercomputer and cloud instances of similar architecture for competitive performance benchmarking. To our surprise, the microbenchmark was executing correctly on one system but crashing on the other. Even after deploying a near identical operating system in the cloud and moving the exact same binary and dependencies between the systems, the faulty behavior persisted. Only due to the compactness of the microbenchmark and because of the limited number of people involved in benchmarking both systems, the root cause, i.e., a bug in the underlying math library related to a specific hardware feature (which was missing in the cloud), was identified within days by the software vendor. While Benchpark was not ready by the time this occurred, it demonstrated to us that quickly deploying complex benchmarks (e.g., proxy-apps and upwards) in a reproducible way across multiple different architectures for functionality or performance comparisons is very labor-intensive and communication-heavy. Potentially, every involved person in the collaboration between on-premise, cloud, and software vendor, will require the identical source code, build environment and instructions, runtime environment, inputs, and/or hardware to identify and debug problems. Benchpark will alleviate the inter-person (mis-)communication by providing a reproducible and easily track-/shareable environment, especially when cross-site access for individuals is impractical.

7.2 The Role of Cloud

As we work to streamline the interactions between HPC centers, vendors, and researchers by developing a collaborative continuous benchmarking solution, using cloud technologies can strengthen the solution. As a test of benchmark portability, cloud resources can be treated like another platform. Cloud infrastructure provides a variety of node architectures that can be used to test and evaluate benchmark behavior. Configuring a cluster of desired or locally unavailable processors without the need to wait in queues for access can quickly provide answers to functional or performance questions. Making benchmark results easily accessible to the HPC community is another way to incorporate cloud resources, including use of services that are designed to analyze data. As an example of community access, the Spack build pipeline and rolling binary cache makes packages available to all Spack users around the globe through Amazon CloudFront, focusing the time to build applications on only the dependencies with special requirements.

8 CONCLUSIONS

Benchmarking is an important component to co-designing, procuring, and evaluating performance on HPC systems. Currently, the process for system benchmarking is a manual and tedious effort, leading to a high barrier to entry, and hampering productivity and reproducibility of benchmarks across different HPC systems.

In this paper, we introduce *collaborative continuous benchmarking* to enable functional reproducibility, automation, and community collaboration in HPC benchmarking. We introduced Benchpark, an open source repository for continuous benchmarking. We believe this will lead to better evaluation of our systems and more productive collaborations within the HPC community.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-853344).

REFERENCES

[1] Hartwig Anzt, Yen-Chen Chen, Terry Cojean, Jack Dongarra, Goran Flegar, Pratik Nayak, Enrique S. Quintana-Ortí, Yuhsiang M. Tsai, and Weichung Wang. 2019. Towards Continuous Benchmarking: An Automated Performance Evaluation Framework for High Performance Software. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (Zurich, Switzerland) *(PASC '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 11 pages. https://doi.org/10.1145/3324989.3325719

[2] David Boehme, Pascal Aschwanden, Olga Pearce, Kenneth Weiss, and Matthew LeGendre. 2021. Ubiquitous Performance Analysis. In *High Performance Computing*, Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.). Springer International Publishing, Cham, 431–449.

[3] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 47, 11 pages.

[4] Thomas Breuer, Sebastian LÃ¼hrs, Andreas Smolenko, and Julia Wellmann. 2022. JUBE (Version 2.5.1); 2.5.1. https://doi.org/10.5281/ZENODO.7534373

[5] Stephanie Brink, Michael McKinsey, David Boehme, W. Daryl Hawkins, Connor Scully-Allison, Ian Lumsden, Treece Burgess, Vanessa Lama, Katherine E. Isaacs, Jakob LÃ¼ttgau, Michela Taufer, and Olga Pearce. 2023. Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, Orlando, FL, USA. https://doi.org/10.1145/3588195.3592989

[6] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. 2013. Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*. ACM, 1–12. https://doi.org/10.1145/2503210.2503277

[7] Massimiliano Culpo, Gregory Becker, Carlos Eduardo Arango Gutierrez, Kenneth Hoste, and Todd Gamblin. 2020. archspec: A library for detecting, labeling, and reasoning about microarchitectures. In *In 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC'20)* (12).

[8] ECP. 2010. Jacamar-CI. https://gitlab.com/ecp-ci/jacamar-ci.

[9] Exascale Computing Project. 2018. ECP Proxy Apps Suite. https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/.

[10] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack Package Manager: Bringing Order to HPC Software Chaos *(Supercomputing 2015 (SCâ€™15))*. Austin, Texas, USA. https://doi.org/10.1145/2807591.2807623 LLNL-CONF-669890.

[11] Google. 2016. PerfKit Benchmarker. https://github.com/GoogleCloudPlatform/PerfKitBenchmarker.

[12] Google. 2023. Ramble. https://github.com/GoogleCloudPlatform/ramble.

[13] I. Bicking. 2011. pip: Package Install tool for Python. https://github.com/pypa/pip.

[14] I. Z. Schlueter. 2009. NPM. https://github.com/npm/npm.

[15] Doug Jacobsen and Bob Bird. 2023. Ramble: A flexible, extensible, and composable experimentation framework. In *HPC Tests Workshop at the ACM/IEEE International Conference on High Performance Computing, Network, Storage, and Analysis (SC|23)*. ACM, Denver, CO, USA.

[16] LANL. 2014. Pavilion Framework. https://github.com/hpc/pavilion2.

[17] Carl Lerche, Yehuda Katz, and André Arko. 2010. Bundler. https://github.com/rubygems/bundler/blob/master/LICENSE.md.

[18] LLNL. 2015. Spack. https://github.com/spack/spack.

[19] LLNL. 2017. Caliper. https://github.com/llnl/caliper.

[20] LLNL. 2019. Adiak. http://github.com/LLNL/adiak.

[21] LLNL. 2023. AMG2023. https://github.com/LLNL/amg2023.

[22] LLNL. 2023. Benchpark. https://github.com/LLNL/benchpark.

[23] LLNL. 2023. Hubcast. https://github.com/LLNL/hubcast.

[24] LLNL. 2023. Thicket. https://github.com/llnl/thicket.

[25] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. 2019. MLPerf Training Benchmark. arXiv:1910.01500 [cs.LG]

[26] ML Commons. 2023. MLPerf. https://mlcommons.org/en/.

[27] Rust. 2014. Cargo: The Rust package manager. https://github.com/rust-lang/cargo.

[28] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA.

[29] J. Zurawski, M. Swany, and D. Gunter. 2006. A Scalable Framework for Representation and Exchange of Network Measurements. In *2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006.* 9 pp.–417. https://doi.org/10.1109/TRIDNT.2006.1649176