EXASCALE COMPUTING PROJECT
Preparing PETSc/TAO for Exascale

libEnsemble + PETSc/TAO: Sustaining a Library for Dynamic Ensemble-Based Computations

David Bindel[1,2] Stephen Hudson[1] Jeffrey Larson[1] Stefan M. Wild[1]
[1]Argonne National Laboratory [2]Cornell University

Overview

libEnsemble is a Python library to coordinate the concurrent evaluation of ensembles of computations. Designed with flexibility in mind, libEnsemble can utilize massively parallel resources to accelerate the solution of design, decision, and inference problems.

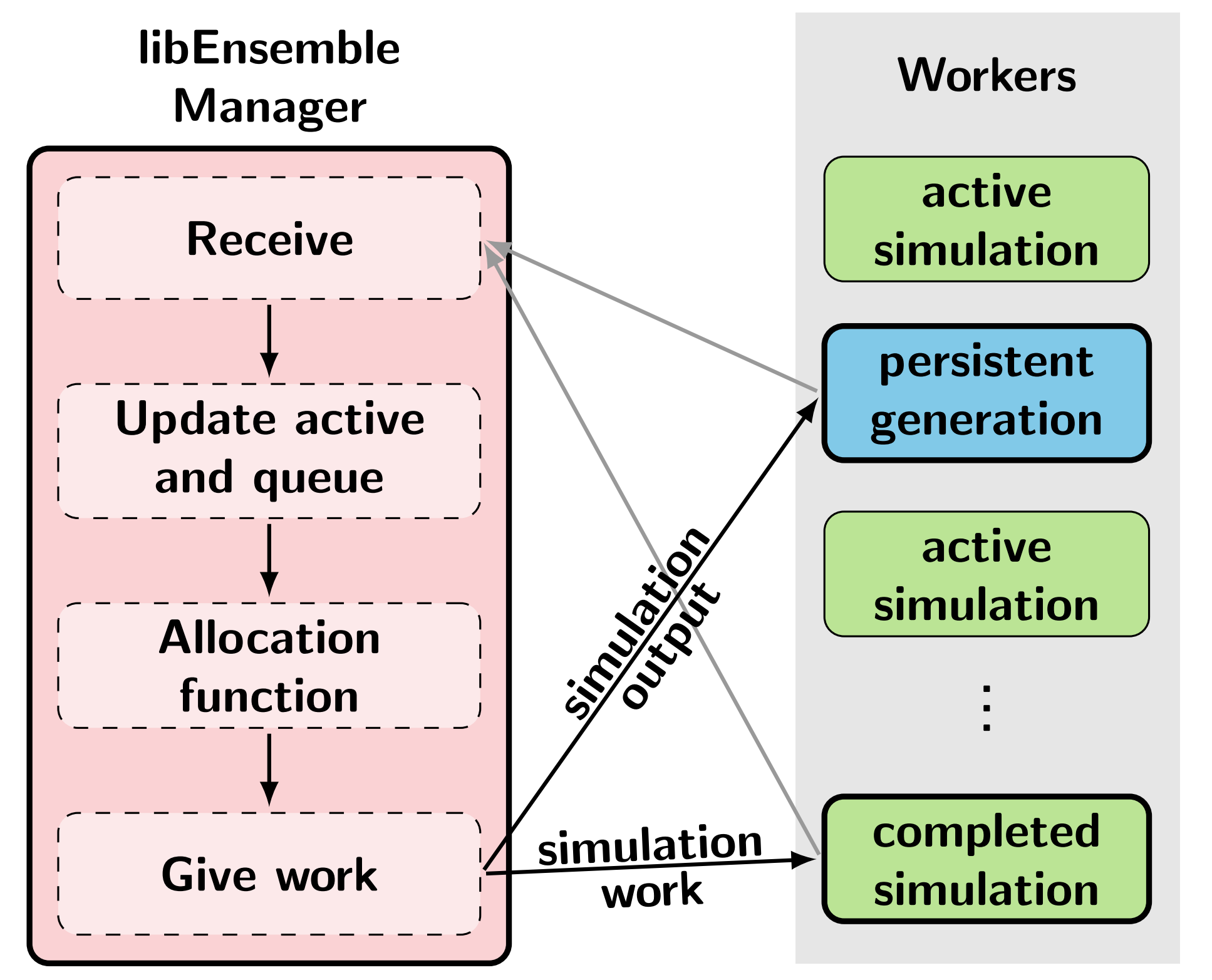
libEnsemble aims for:

- Extreme scaling;
- Fault tolerance;
- Monitoring/killing jobs (recovers resources);
- Portability and flexibility; and
- Exploitation of persistent data/control flow.

Manager and Workers

libEnsemble employs a manager-worker scheme that can run on various communication mediums (including MPI, multiprocessing, and TCP).
Workers can run simulation functions or generator functions (which create new parameters/inputs for simulations).

- Ex.- Random Sample → Simulations → Optimization → Simulations



Composable Software

libEnsemble encapsulates the job control and worker communication layers making it highly adaptable to future systems and easy to experiment.

Applications & Users Wanted!

libEnsemble is an open-source PETSc project:

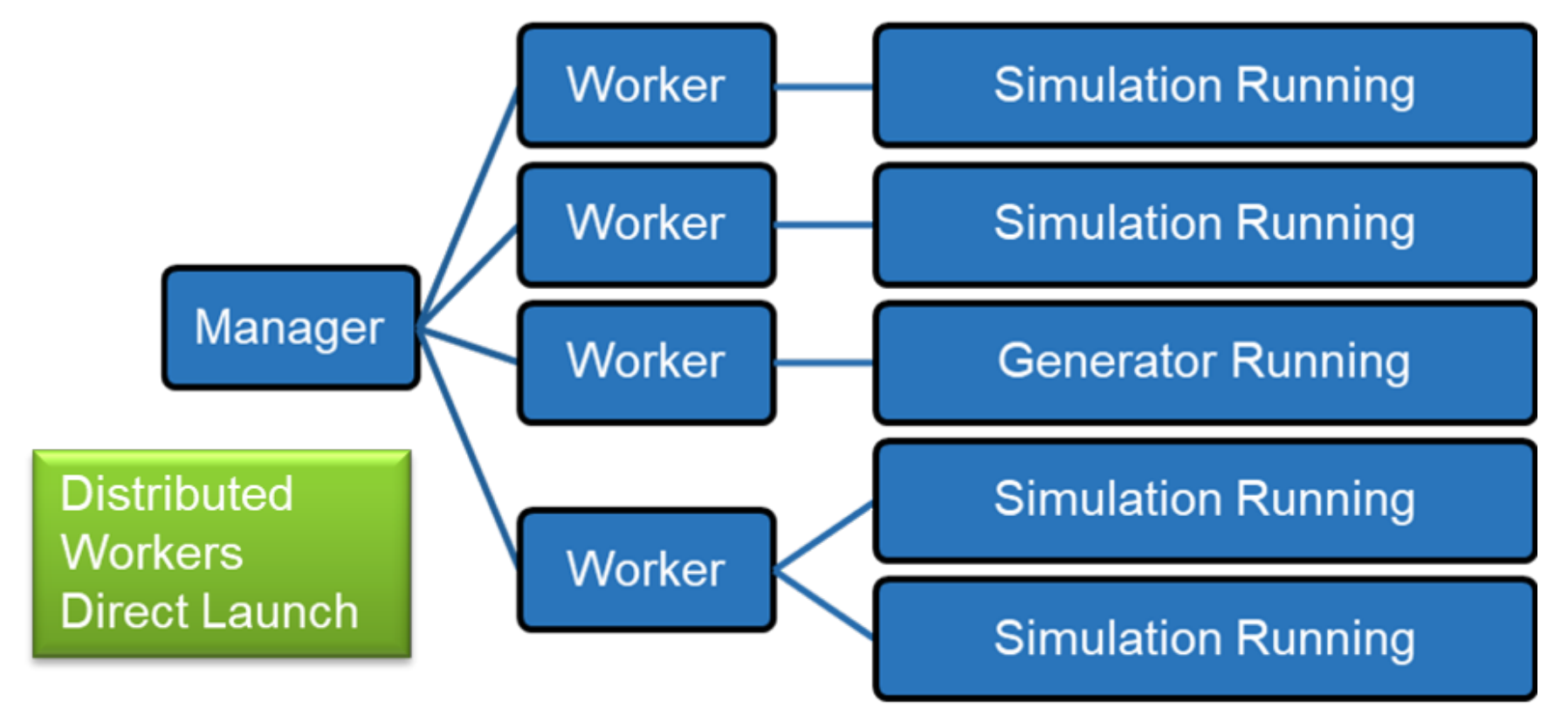
https://github.com/libensemble/libensemble
https://libensemble.readthedocs.io

libEnsemble

Flexible Run Mechanisms

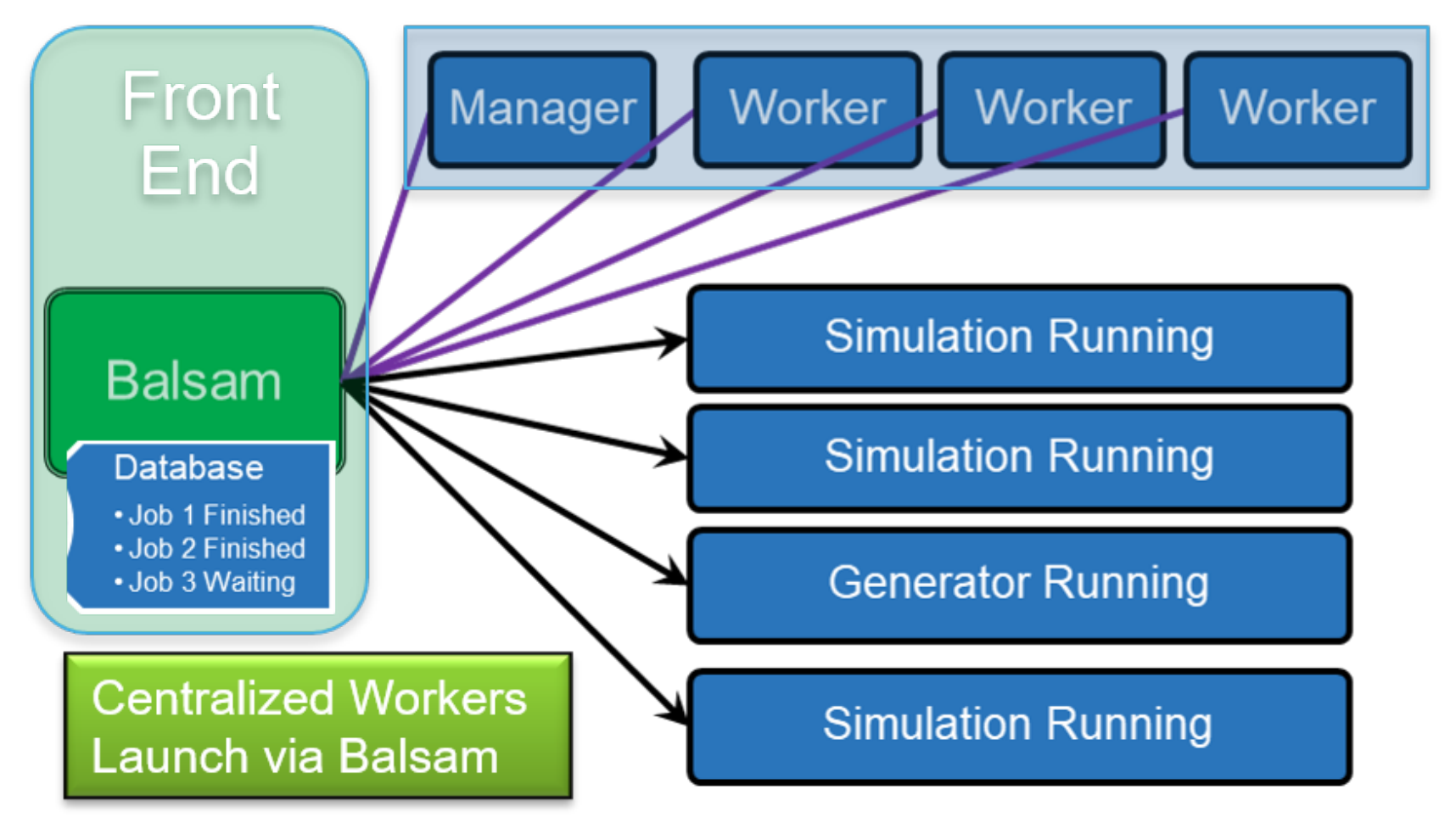
Workers can run in various configurations.

Distributed: Workers can run on compute nodes and launch jobs directly in-place.



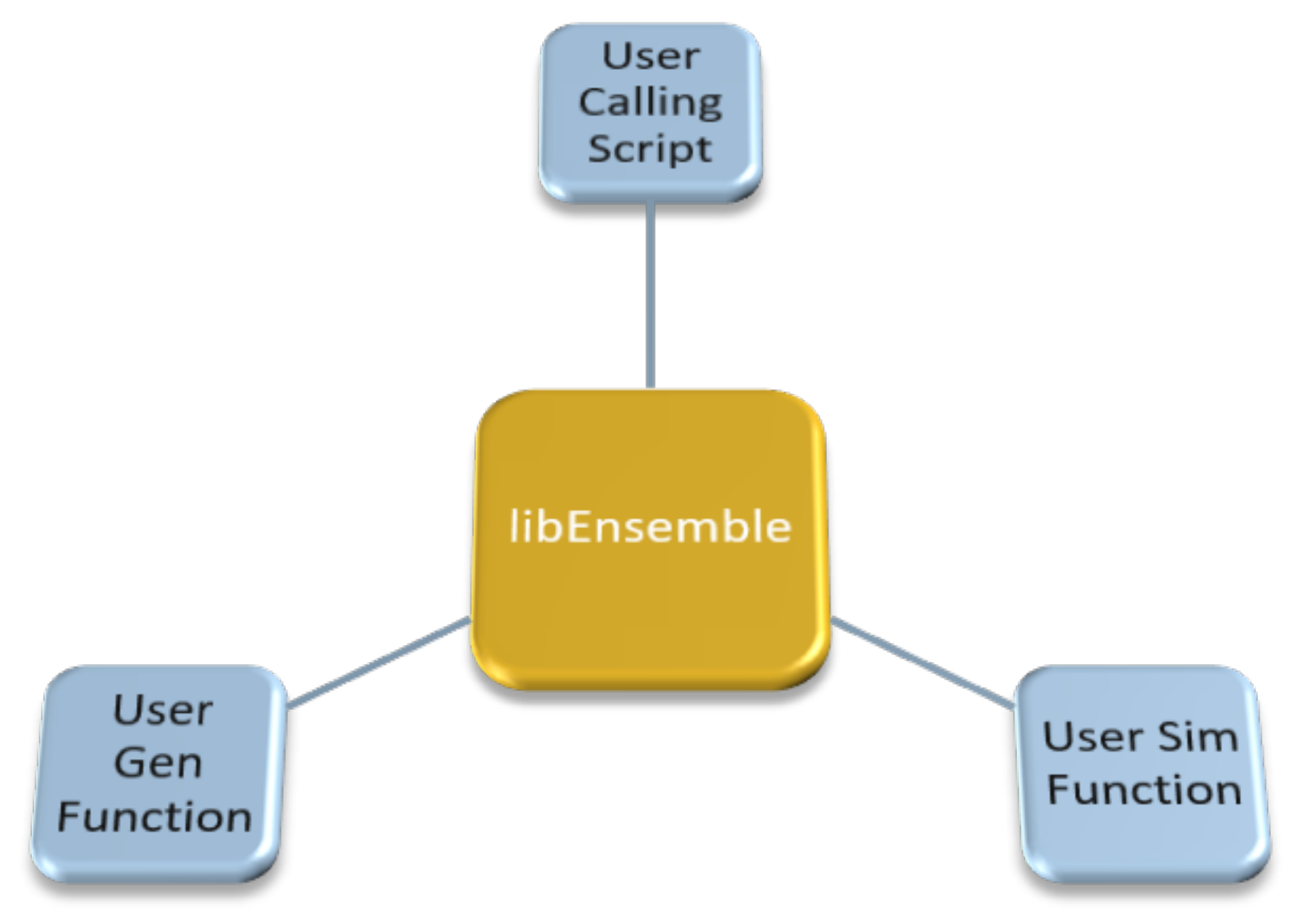
Centralized: Workers run on dedicated nodes and launch jobs to another set of nodes.

Balsam: Used as a proxy job launcher; Balsam runs on front end and maintains a database of jobs.



Using libEnsemble

User selects or supplies a generation function that produces simulation input as well as a simulation function that performs and monitors the simulations. Examples and templates of these functions are included in the library.



There are many potential use cases including:

Example gen. funcs

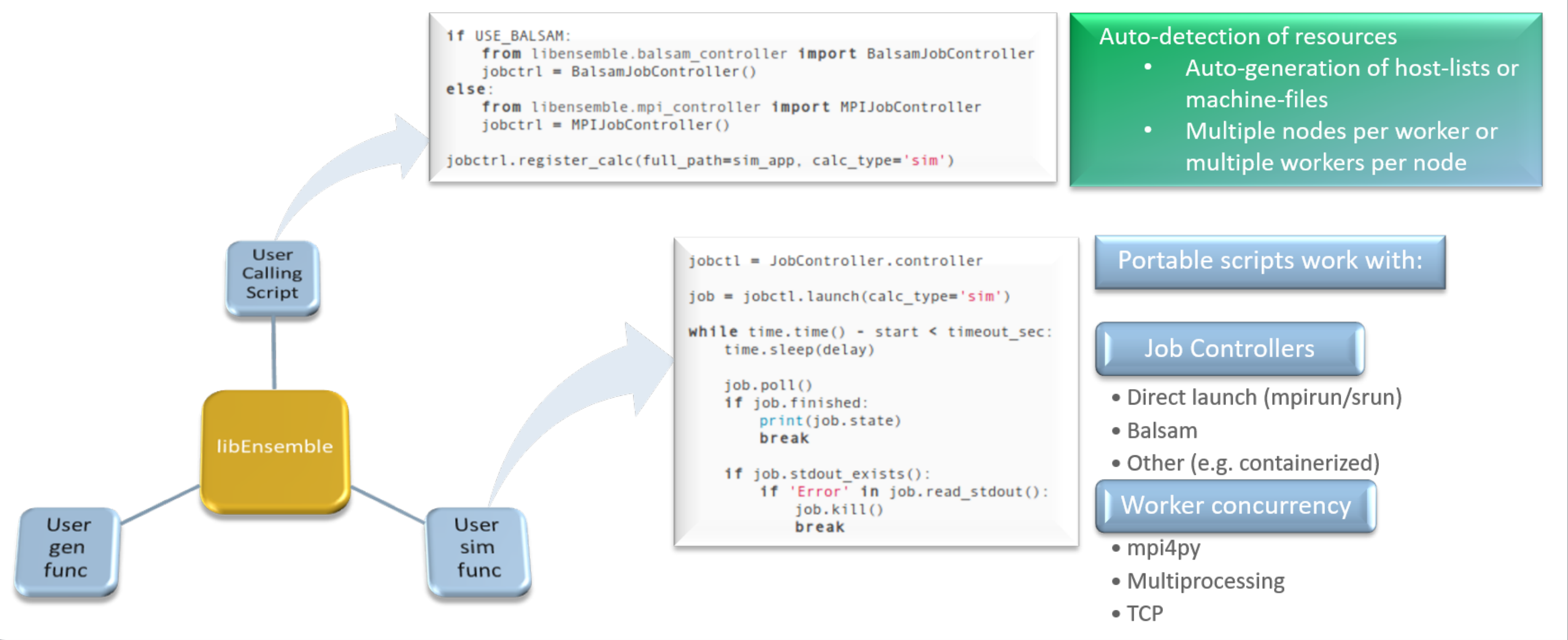
- Bayesian parameter estimation
- Surrogate models
- Sensitivity analysis
- Design optimization
- Supervised learning

Example sim. funcs

- Particle accelerator simulation
- Subsurface flow
- PETSc simulations
- DFT calculations
- Quantum chemistry

Job Controller

A job controller interface is provided and allows users to write portable simulation/generator functions in Python. These are agnostic of both the job launch/management system and worker concurrency. The main job controller functions are *launch*, *poll*, and *kill*.



```
if USE_BALSAM:
    from libensemble.balsam_controller import BalsamJobController
    jobctrl = BalsamJobController()
else:
    from libensemble.mpi_controller import MPIJobController
    jobctrl = MPIJobController()

jobctrl.register_calc(full_path=sim_app, calc_type='sim')
```

Auto-detection of resources
- Auto-generation of host-lists or machine-files
- Multiple nodes per worker or multiple workers per node

```
jobctl = JobController.controller

job = jobctl.launch(calc_type='sim')

while time.time() - start < timeout_sec:
    time.sleep(delay)

    job.poll()
    if job.finished:
        print(job.state)
        break

    if job.stdout_exists():
        if 'Error' in job.read_stdout():
            job.kill()
            break
```

Portable scripts work with:

Job Controllers
- Direct launch (mpirun/srun)
- Balsam
- Other (e.g. containerized)

Worker concurrency
- mpi4py
- Multiprocessing
- TCP

Collaborations

We are interested in collaborations with applications and math/CS libraries:

- Efficient application level scheduling and resource partitioning.
- Minimizing Python overheads at scale.
- Fault tolerance approaches.
- Workflows.

Future

Experimental and aspirational features:

- TCP communicator could allow workers to be dynamically added from cloud resources.
- Job controllers can capitalize on novel features such as containerized launches to optimize resource partitioning and startup costs.
- Persistent gen./sim. functions and distributed workers can make use of on-node resources (e.g., SSDs, GPUs).

Project Workflow

Project management centered around GitHub

- Developers and users create GitHub issues.
- Project administrators add these to GitHub kanban.
- Pull requests can reference GitHub issues.
- Web tools run on pushes for testing, coverage, and building documentation.
- Documentation uses Sphinx (with autodoc) to extract docstrings and automatically update.
- Release on GitHub/PyPI/Spack.

Testing

There are four components to libEnsemble testing as listed below.

Unit tests: Using pytest test framework (includes mocking of objects). Runs on Travis CI.

Regression tests: Full tests that can be run at varying scales. Runs on Travis CI (up to four processes).

Scaling tests: Highly configurable tests with minimal or no dependencies.

- Testing features at scale including job scheduling/launching/killing, resource partitioning, on-node/hierarchical storage, and hardware fault tolerance.
- Useful for testing functionality and for profiling.
- Potential for system vendors to use at pre-production stage.

Standalone tests: Isolated tests that can be run on platforms to check what libEnsemble components/options of work on a given platform: e.g., job launches and node partitioning, communications, killing jobs, and recovering resources.

Small scale testing is automated and runs in Travis CI with multiple Python versions and MPI libraries. Larger tests performed manually on HPC platforms as some options currently cannot be tested on Travis.

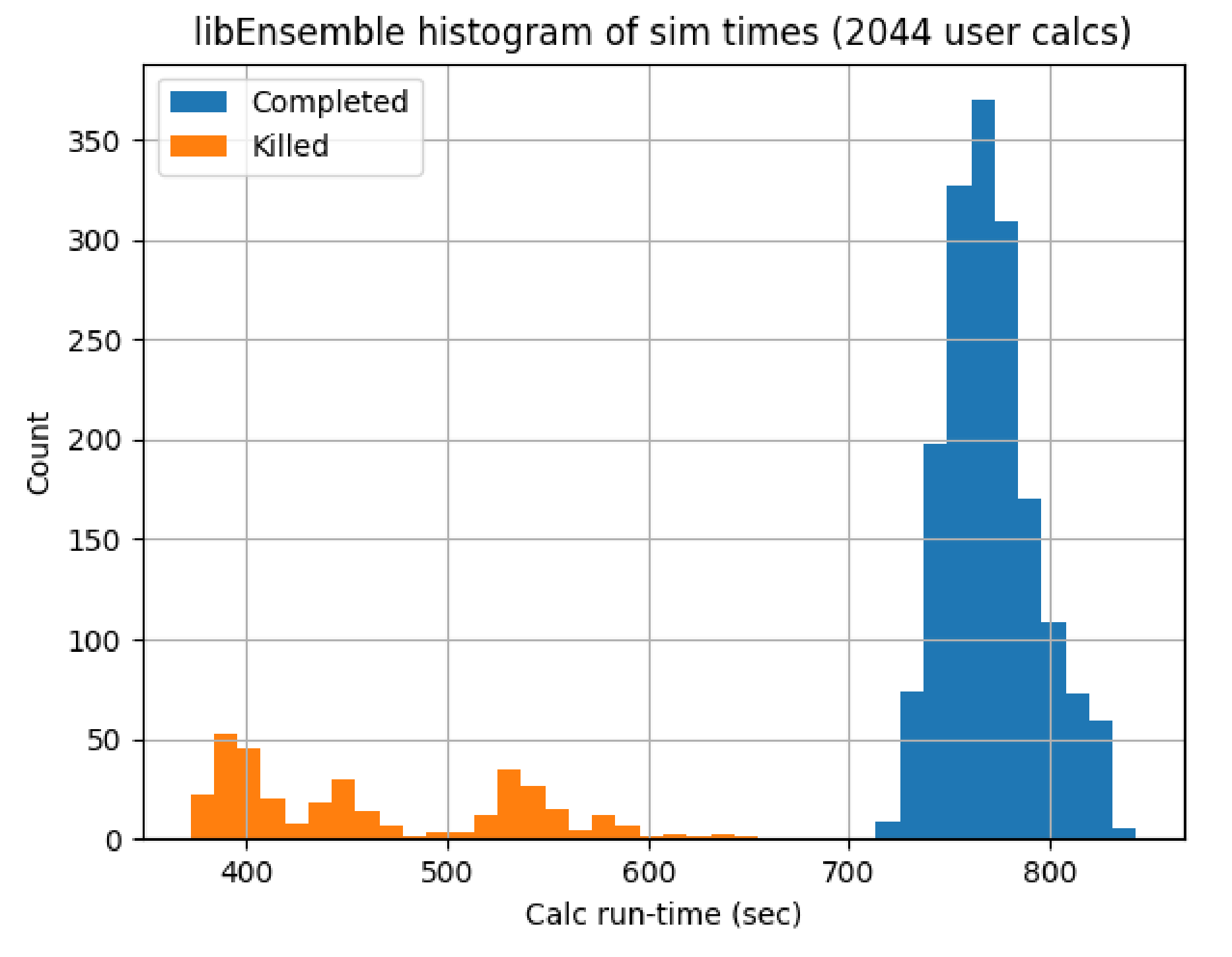
Code Coverage:

- Coverage for unit/regression tests is approximately 90% (Branch coverage included).
- Coveralls (web tool) is linked with Travis CI.
- Coverage is worked out for unit and regression tests and merged to provide a combined score.

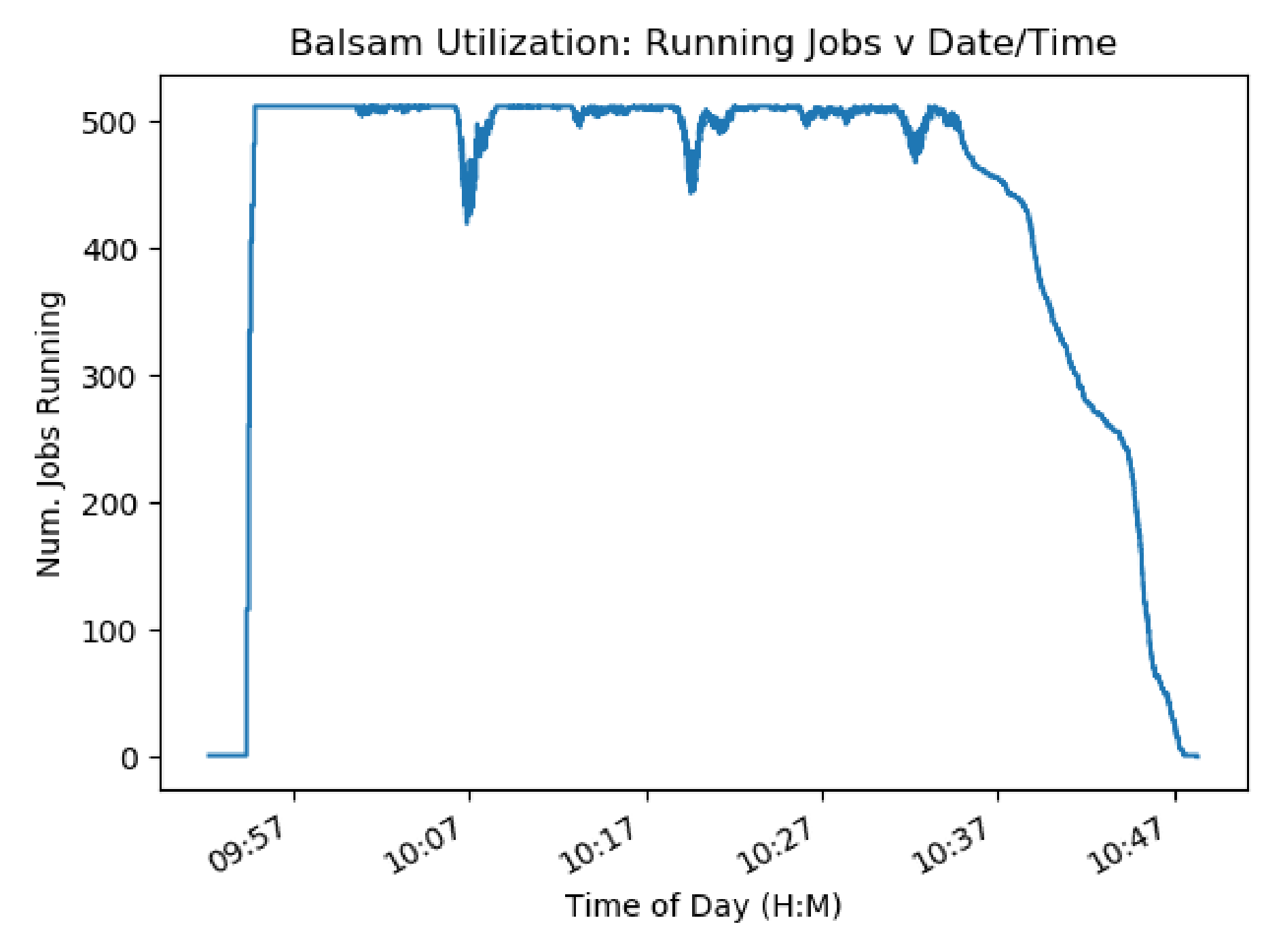
Running at Scale

Ex.- libEnsemble scaling: 1,030 node ensemble on ALCF/Theta (Cray XC40) with Balsam

- 511 workers (2,044 2-node simulations) using MPI.
- OPAL (Object Oriented Parallel Accelerator Library) simulation functions.



Histogram of completed and killed simulations (binned by run time). Killing jobs once they are identified as redundant improves efficiency of ensembles.



Total number of Balsam-launched applications running over time. The startup delay is due to parallel imports of Python libraries.