libEnsemble User Manual

Release 1.1.0+dev

**Stephen Hudson, Jeffrey Larson, Stefan M. Wild,
David Bindel, John-Luke Navarro**

December 13, 2023

Contents

Chapter 1

libEnsemble: A complete toolkit for dynamic ensembles of calculations

Adaptive, portable, and scalable software for connecting "deciders" to experiments or simulations.

- **Dynamic ensembles**: Generate parallel tasks on-the-fly based on previous computations.
- **Extreme portability and scaling**: Run on or across laptops, clusters, and leadership-class machines.
- **Heterogeneous computing**: Dynamically and portably assign CPUs, GPUs, or multiple nodes.
- **Application monitoring**: Ensemble members can run, monitor, and cancel apps.
- **Data-flow between tasks**: Running ensemble members can send and receive data.
- **Low start-up cost**: No additional background services or processes required.

libEnsemble is effective at solving design, decision, and inference problems on parallel resources.

Quickstart

1.1 Installation

Install libEnsemble and its dependencies from PyPI using pip:

```
pip install libensemble
```

Other install methods are described in the docs.

1.2 Basic Usage

Create an `Ensemble`, then customize it with general settings, simulation and generator parameters, and an exit condition. Run the following via `python this_file.py --comms local --nworkers 4`:

```
import numpy as np

from libensemble import Ensemble
from libensemble.gen_funcs.sampling import uniform_random_sample
from libensemble.sim_funcs.six_hump_camel import six_hump_camel
from libensemble.specs import ExitCriteria, GenSpecs, SimSpecs
from libensemble.tools import add_unique_random_streams

if __name__ == "__main__":
    sampling = Ensemble(parse_args=True)
    sampling.sim_specs = SimSpecs(
        sim_f=six_hump_camel,
        inputs=["x"],
        outputs=[("f", float)],
    )
    sampling.gen_specs = GenSpecs(
        gen_f=uniform_random_sample,
        outputs=[("x", float, (2,))],
        user={
            "gen_batch_size": 500,
            "lb": np.array([-3, -2]),
            "ub": np.array([3, 2]),
        },
    )

    sampling.persis_info = add_unique_random_streams({}, sampling.nworkers + 1)
    sampling.exit_criteria = ExitCriteria(sim_max=101)
    sampling.run()
    sampling.save_output(__file__)

    if sampling.is_manager:
        print("Some output data:\n", sampling.H[["x", "f"]][:10])
```

1.3 Resources

Support:

- Ask questions or report issues on GitHub.
- Email `libEnsemble@lists.mcs.anl.gov` to request libEnsemble Slack page.
- Join the libEnsemble mailing list for updates about new releases.

Further Information:

- Documentation is provided by ReadtheDocs.
- Contributions to libEnsemble are welcome.
- Browse production functions and workflows in the Community Examples repository.

Cite libEnsemble:

```
@article{Hudson2022,
  title   = {{libEnsemble}: A Library to Coordinate the Concurrent
             Evaluation of Dynamic Ensembles of Calculations},
  author  = {Stephen Hudson and Jeffrey Larson and John-Luke Navarro and Stefan M. Wild},
  journal = {{IEEE} Transactions on Parallel and Distributed Systems},
  volume  = {33},
  number  = {4},
  pages   = {977--988},
  year    = {2022},
  doi     = {10.1109/tpds.2021.3082815}
}
```

See the `tutorial` for a step-by-step beginners guide.

See the user guide for more information.

Example Compatible Packages

libEnsemble and the Community Examples repository include example generator functions for the following libraries:

- APOSMM Asynchronously parallel optimization solver for finding multiple minima. Supported local optimization routines include:
  - DFO-LS Derivative-free solver for (bound constrained) nonlinear least-squares minimization
  - NLopt Library for nonlinear optimization, providing a common interface for various methods
  - scipy.optimize Open-source solvers for nonlinear problems, linear programming, constrained and nonlinear least-squares, root finding, and curve fitting.
  - PETSc/TAO Routines for the scalable (parallel) solution of scientific applications
- DEAP Distributed evolutionary algorithms
- Distributed optimization methods for minimizing sums of convex functions. Methods include:
  - Primal-dual sliding (https://arxiv.org/pdf/2101.00143).
  - Distributed gradient descent with gradient tracking (https://arxiv.org/abs/1908.11444).
  - Proximal sliding (https://arxiv.org/abs/1406.0919).
- ECNoise Estimating Computational Noise in Numerical Simulations
- Surmise Modular Bayesian calibration/inference framework
- Tasmanian Toolkit for Adaptive Stochastic Modeling and Non-Intrusive ApproximatioN
- VTMOP Fortran package for large-scale multiobjective multidisciplinary design optimization

libEnsemble has also been used to coordinate many computationally expensive simulations. Select examples include:

- OPAL Object Oriented Parallel Accelerator Library. (See this IPAC manuscript.)
- WarpX Advanced electromagnetic particle-in-cell code. (See example WarpX + libE scripts.)

Chapter 2

Constructing Workflows

We now give greater detail in programming with libEnsemble.

2.1 Running an Ensemble

libEnsemble features two approaches to run an ensemble. We recommend the newer `Ensemble` class, but will continue to support `libE()` for backward compatibility.

Ensemble Class

class libensemble.ensemble.**Ensemble**

The primary object for a libEnsemble workflow. Parses and validates settings, sets up logging, and maintains output.

Example

```
import numpy as np

from libensemble import Ensemble
from libensemble.gen_funcs.sampling import latin_hypercube_sample
from libensemble.sim_funcs.one_d_func import one_d_example
from libensemble.specs import ExitCriteria, GenSpecs, SimSpecs

sampling = Ensemble(parse_args=True)
sampling.sim_specs = SimSpecs(
    sim_f=one_d_example,
    inputs=["x"],
    outputs=[("f", float)],
)
sampling.gen_specs = GenSpecs(
    gen_f=latin_hypercube_sample,
    outputs=[("x", float, (1,))],
    user={
```

(continues on next page)

(continued from previous page)

```
        "gen_batch_size": 500,
        "lb": np.array([-3]),
        "ub": np.array([3]),
    },
)

sampling.add_random_streams()
sampling.exit_criteria = ExitCriteria(sim_max=101)

if __name__ == "__main__":
    sampling.run()
    sampling.save_output(__file__)
```

Run the above example via `python this_file.py --comms local --nworkers 4`. The `parse_args=True` parameter instructs the Ensemble class to read command-line arguments.

Configure by:

Option 1: Providing parameters on instantiation

```
from libensemble import Ensemble
from my_simulator import sim_find_energy

sim_specs = {
    "sim_f": sim_find_energy,
    "in": ["x"],
    "out": [("y", float)],
}

experiment = Ensemble(sim_specs=sim_specs)
```

Option 2: Assigning parameters to an instance

```
from libensemble import Ensemble, SimSpecs
from my_simulator import sim_find_energy

sim_specs = SimSpecs(
    sim_f=sim_find_energy,
    inputs=["x"],
    outputs=[("y", float)],
)

experiment = Ensemble()
experiment.sim_specs = sim_specs
```

Option 3: Loading parameters from files

```python
from libensemble import Ensemble

experiment = Ensemble()

my_experiment.from_yaml("my_parameters.yaml")
# or...
my_experiment.from_toml("my_parameters.toml")
# or...
my_experiment.from_json("my_parameters.json")
```

my_parameters.yaml

```yaml
libE_specs:
    save_every_k_gens: 20

exit_criteria:
    sim_max: 80

gen_specs:
    gen_f: generator.gen_random_sample
    outputs:
        x:
            type: float
            size: 1
    user:
        gen_batch_size: 5

sim_specs:
    sim_f: simulator.sim_find_sine
    inputs:
        - x
    outputs:
        y:
            type: float
```

my_parameters.toml

```toml
[libE_specs]
    save_every_k_gens = 300

[exit_criteria]
    sim_max = 80

[gen_specs]
    gen_f = "generator.gen_random_sample"
    [gen_specs.outputs]
        [gen_specs.outputs.x]
            type = "float"
```

(continues on next page)

(continued from previous page)

```
            size = 1
    [gen_specs.user]
        gen_batch_size = 5

[sim_specs]
    sim_f = "simulator.sim_find_sine"
    inputs = ["x"]
    [sim_specs.outputs]
        [sim_specs.outputs.y]
            type = "float"
```

my_parameters.json

```
{
    "libE_specs": {
        "save_every_k_gens": 300,
    },
    "exit_criteria": {
        "sim_max": 80
    },
    "gen_specs": {
        "gen_f": "generator.gen_random_sample",
        "outputs": {
            "x": {
                "type": "float",
                "size": 1
            }
        },
        "user": {
            "gen_batch_size": 5
        }
    },
    "sim_specs": {
        "sim_f": "simulator.sim_find_sine",
        "inputs": ["x"],
        "outputs": {
            "f": {"type": "float"}
        }
    }
}
```

Parameters

- **sim_specs** (dict or SimSpecs) – Specifications for the simulation function
- **gen_specs** (dict or GenSpecs, Optional) – Specifications for the generator function
- **exit_criteria** (dict or ExitCriteria, Optional) – Tell libEnsemble when to stop a run
- **libE_specs** (dict or LibeSpecs, Optional) – Specifications for libEnsemble
- **alloc_specs** (dict or AllocSpecs, Optional) – Specifications for the allocation function

- **persis_info** (dict, Optional) – Persistent information to be passed between user function instances (example)
- **executor** (Executor, Optional) – libEnsemble Executor instance for use within simulation or generator functions
- **H0** (NumPy structured array, Optional) – A libEnsemble history to be prepended to this run's history (example)
- **parse_args** (*bool, Optional*) – Read nworkers, comms, and other arguments from the command-line. For MPI, calculate nworkers and set the is_manager Boolean attribute on MPI rank 0. See the parse_args docs for more information.

ready()

Quickly verify that all necessary data has been provided

Return type
bool

run()

Initializes libEnsemble.

MPI/comms Notes

Manager–worker intercommunications are parsed from the comms key of libE_specs. An MPI runtime is assumed by default if --comms local wasn't specified on the command-line or in libE_specs.

If a MPI communicator was provided in libE_specs, then each .run() call will initiate intercommunications on a **duplicate** of that communicator. Otherwise, a duplicate of COMM_WORLD will be used.

Returns

- **H** (*NumPy structured array*) – History array storing rows for each point. (example)
- **persis_info** (dict) – Final state of persistent information (example)
- **exit_flag** (int) – Flag containing final task status

```
0 = No errors
1 = Exception occurred
2 = Manager timed out and ended simulation
3 = Current process is not in libEnsemble MPI communicator
```

Return type
(numpy.ndarray[*Any*, numpy.dtype[+_ScalarType_co]], <class 'dict'>, <class 'int'>)

from_yaml(*file_path*)

Parameterizes libEnsemble from yaml file

Parameters
file_path (*str*) –

from_toml(*file_path*)

Parameterizes libEnsemble from toml file

Parameters
file_path (*str*) –

from_json(*file_path*)

Parameterizes libEnsemble from `json` file

Parameters
file_path (*str*) –

add_random_streams(*num_streams=0*, *seed=''*)

Adds `np.random` generators for each worker ID to `self.persis_info`.

Parameters

- **num_streams** (*int, Optional*) – Number of matching worker ID and random stream entries to create. Defaults to `self.nworkers`.
- **seed** (*str, Optional*) – Seed for NumPy's RNG.

save_output(*file*)

Writes out History array and persis_info to files. If using a workflow_dir, will place with specified filename in that directory.

Format: `<calling_script>_results_History_length=<length>_evals=<Completed evals>_ranks=<nworkers>`

Parameters
file (*str*) –

libE()

The libE module is the outer libEnsemble routine.

This module sets up the manager and the team of workers, configured according to the contents of `libE_specs`. The manager/worker communications scheme used in libEnsemble is parsed from the `comms` key if present, with valid values being `mpi`, `local` (for multiprocessing), or `tcp`. MPI is the default; if a communicator is specified, each call to this module will initiate manager/worker communications on a duplicate of that communicator. Otherwise, a duplicate of `COMM_WORLD` will be used.

In the vast majority of cases, programming with libEnsemble involves the creation of a *calling script*, a Python file where libEnsemble is parameterized via the various specification dictionaries (e.g. `libE_specs`, `sim_specs`, and `gen_specs`). The outer libEnsemble routine `libE()` is imported and called with such dictionaries to initiate libEnsemble. A simple calling script (from `the first tutorial`) may resemble:

```
import numpy as np
from libensemble.libE import libE
from generator import gen_random_sample
from simulator import sim_find_sine
from libensemble.tools import add_unique_random_streams

nworkers, is_manager, libE_specs, _ = parse_args()

libE_specs["save_every_k_gens"] = 20

gen_specs = {
    "gen_f": gen_random_sample,
    "out": [("x", float, (1,))],
    "user": {"lower": np.array([-3]), "upper": np.array([3]), "gen_batch_size": 5},
}

```

(continues on next page)

(continued from previous page)

```
sim_specs = {"sim_f": sim_find_sine, "in": ["x"], "out": [("y", float)]}

persis_info = add_unique_random_streams({}, nworkers + 1)

exit_criteria = {"sim_max": 80}

H, persis_info, flag = libE(sim_specs, gen_specs, exit_criteria, persis_info, libE_
↪specs=libE_specs)
```

This will initiate libEnsemble with a Manager and `nworkers` workers (parsed from the command line), and runs on laptops or supercomputers. If an exception is encountered by the manager or workers, the history array is dumped to file, and MPI abort is called.

On macOS (since Python 3.8) and Windows, the default multiprocessing start method is `"spawn"` and you must place most calling script code (or just `libE()` / `Ensemble().run()` at a minimum) in an `if __name__ == "__main__:"` block.

Therefore a calling script that is universal across all platforms and comms-types may resemble:

```
import numpy as np
from libensemble.libE import libE
from generator import gen_random_sample
from simulator import sim_find_sine
from libensemble.tools import add_unique_random_streams

if __name__ == "__main__":
    nworkers, is_manager, libE_specs, _ = parse_args()

    libE_specs["save_every_k_gens"] = 20

    gen_specs = {
        "gen_f": gen_random_sample,
        "out": [("x", float, (1,))],
        "user": {
            "lower": np.array([-3]),
            "upper": np.array([3]),
            "gen_batch_size": 5,
        },
    }

    sim_specs = {
        "sim_f": sim_find_sine,
        "in": ["x"],
        "out": [("y", float)],
    }

    persis_info = add_unique_random_streams({}, nworkers + 1)

    exit_criteria = {"sim_max": 80}

    H, persis_info, flag = libE(sim_specs, gen_specs, exit_criteria, persis_info, libE_
↪specs=libE_specs)
```

Alternatively, you may set the multiprocessing start method to `"fork"` via the following:

```
from multiprocessing import set_start_method

set_start_method("fork")
```

But note that this is incompatible with some libraries.

See below for the complete traditional API.

libensemble.libE.**libE**(*sim_specs*, *gen_specs*, *exit_criteria*, *persis_info={}*, *alloc_specs=AllocSpecs(alloc_f=<function give_sim_work_first>*, *user={'num_active_gens': 1}*, *outputs=[])*, *libE_specs={}*, *H0=None*)

Parameters

- **sim_specs** (dict or SimSpecs) – Specifications for the simulation function (example)
- **gen_specs** (dict or GenSpecs, Optional) – Specifications for the generator function (example)
- **exit_criteria** (dict or ExitCriteria, Optional) – Tell libEnsemble when to stop a run (example)
- **persis_info** (dict, Optional) – Persistent information to be passed between user functions (example)
- **alloc_specs** (dict or AllocSpecs, Optional) – Specifications for the allocation function (example)
- **libE_specs** (dict or LibeSpecs, Optional) – Specifications for libEnsemble (example)
- **H0** – A libEnsemble history to be prepended to this run's history (example)

Returns

- **H** (NumPy structured array) – History array storing rows for each point. (example)
- **persis_info** (dict) – Final state of persistent information (example)
- **exit_flag** (int) – Flag containing final task status

```
0 = No errors
1 = Exception occurred
2 = Manager timed out and ended simulation
3 = Current process is not in libEnsemble MPI communicator
```

Return type

(<class 'numpy.ndarray'>, *Dict*, <class 'int'>)

2.2 Configuring libEnsemble

libEnsemble workflows are configured via the following data structures. See here for instruction on constructing a complete workflow.

2.2.1 Simulation Specs

Used to specify the simulation function, its inputs and outputs, and user data.

Can be constructed and passed to libEnsemble as a Python class or a dictionary.

class

```
...
from libensemble import SimSpecs
from simulator import sim_find_sine

...

sim_specs = SimSpecs(
    sim_f=sim_find_sine,
    inputs=["x"],
    out=[("y", float)],
    user={"batch": 1234},
)
...
```

pydantic model libensemble.specs.**SimSpecs**

Fields

dict

```
...
from simulator import six_hump_camel

...

sim_specs = {
    "sim_f": six_hump_camel,
    "in": ["x"],
    "out": [("y", float)],
    "user": {"batch": 1234},
}
...
```

- test_uniform_sampling.py has a `sim_specs` that declares the name of the `"in"` field variable, `"x"` (as specified by the corresponding generator `"out"` field `"x"` from the `gen_specs example`). Only the field name is required in `sim_specs["in"]`.

```
    sim_specs = {
        "sim_f": six_hump_camel,  # Function whose output is being minimized
        "in": ["x"],  # Keys to be given to sim_f
        "out": [("f", float)],  # Name of the outputs from sim_f
    }
```

- run_libe_forces.py has a longer `sim_specs` declaration with a number of user-specific fields. These are given to the corresponding sim_f, which can be found at forces_simf.py.

```
    "sim_f": run_forces,  # Function whose output is being minimized
    "in": ["x"],  # Name of input for sim_f
    "out": [("energy", float)],  # Name, type of output from sim_f
    "user": {
        "keys": ["seed"],
        "cores": 2,
        "sim_particles": 1e3,
        "sim_timesteps": 5,
        "sim_kill_minutes": 10.0,
        "particle_variance": 0.2,
        "kill_rate": 0.5,
        "fail_on_sim": False,
        "fail_on_submit": False,  # Won't occur if 'fail_on_sim' True
    },
}
```

2.2.2 Generator Specs

Used to specify the generator function, its inputs and outputs, and user data.

Can be constructed and passed to libEnsemble as a Python class or a dictionary.

class

```
...
import numpy as np
from libensemble import GenSpecs
from generator import gen_random_sample

...

gen_specs = GenSpecs(
    gen_f=gen_random_sample,
    out=[("x", float, (1,))],
    user={
        "lower": np.array([-3]),
        "upper": np.array([3]),
        "gen_batch_size": 5,
    },
)
...
```

pydantic model libensemble.specs.**GenSpecs**

Fields

dict

```
...
import numpy as np
from generator import gen_random_sample

...

gen_specs = {
    "gen_f": gen_random_sample,
    "out": [("x", float, (1,))],
    "user": {
        "lower": np.array([-3]),
        "upper": np.array([3]),
        "gen_batch_size": 5,
    },
}
```

See also:

- test_uniform_sampling.py: the generator function `uniform_random_sample` in sampling.py will generate 500 random points uniformly over the 2D domain defined by `gen_specs["ub"]` and `gen_specs["lb"]`.

```
    gen_specs = {
        "gen_f": uniform_random_sample,  # Function generating sim_f input
        "out": [("x", float, (2,))],  # Tell libE gen_f output, type, size
        "user": {
            "gen_batch_size": 500,  # Used by this specific gen_f
            "lb": np.array([-3, -2]),  # Used by this specific gen_f
            "ub": np.array([3, 2]),  # Used by this specific gen_f
        },
    }
```

See also:

- test_persistent_aposmm_nlopt.py shows an example where `gen_specs["in"]` is empty, but `gen_specs["persis_in"]` specifies values to return to the persistent generator.
- test_persistent_aposmm_with_grad.py shows a similar example where an `H0` is used to provide points from a previous run. In this case, `gen_specs["in"]` is populated to provide the generator with data for the initial points.
- In some cases you might be able to give different (perhaps fewer) fields in `"persis_in"` than `"in"`; you may not need to give `x` for example, as the persistent generator already has `x` for those points. See more example uses of `persis_in`.

Note:

- In all interfaces, custom fields should only be placed in `"user"`
- Generator `"out"` fields typically match Simulation `"in"` fields, and vice-versa.

2.2.3 General Specs

libEnsemble is primarily customized by setting options within a `LibeSpecs` class or dictionary.

```
from libensemble.specs import LibeSpecs

specs = LibeSpecs(
    comm=MPI.COMM_WORLD,
    comms="mpi",
    save_every_k_gens=1000,
    sim_dirs_make=True,
    ensemble_dir_path="/scratch/ensemble",
)
```

Settings by Category

General

comms [str] = "mpi":
Manager/Worker communications mode: `'mpi'`, `'local'`, or `'tcp'`.

nworkers [int]:
Number of worker processes in `"local"` or `"tcp"`.

mpi_comm [MPI communicator] = MPI.COMM_WORLD:
libEnsemble MPI communicator.

dry_run [bool] = False:
Whether libEnsemble should immediately exit after validating all inputs.

abort_on_exception [bool] = True:
In MPI mode, whether to call `MPI_ABORT` on an exception. If `False`, an exception will be raised by the manager.

worker_timeout [int] = 1:
On libEnsemble shutdown, number of seconds after which workers considered timed out, then terminated.

kill_canceled_sims [bool] = False:
Try to kill sims with `cancel_requested` set to `True`. If `False`, the manager avoids this moderate overhead.

disable_log_files [bool] = False:
Disable `ensemble.log` and `libE_stats.txt` log files.

Directories

General

use_workflow_dir [bool] = False:
Whether to place *all* log files, dumped arrays, and default ensemble-directories in a separate `workflow` directory. Each run is suffixed with a hash. If copying back an ensemble directory from another location, the copy is placed here.

workflow_dir_path [str]:
Optional path to the workflow directory.

ensemble_dir_path [str] = "./ensemble":
Path to main ensemble directory. Can serve as single working directory for workers, or contain calculation directories.

```
LibeSpecs.ensemble_dir_path = "/scratch/my_ensemble"
```

ensemble_copy_back [bool] = `False`:
Whether to copy back contents of `ensemble_dir_path` to launch location. Useful if `ensemble_dir_path` is located on node-local storage.

reuse_output_dir [bool] = `False`:
Whether to allow overwrites and access to previous ensemble and workflow directories in subsequent runs. `False` by default to protect results.

calc_dir_id_width [int] = 4:
The width of the numerical ID component of a calculation directory name. Leading zeros are padded to the sim/gen ID.

use_worker_dirs [bool] = `False`:
Whether to organize calculation directories under worker-specific directories:

False

```
- /ensemble_dir
    - /sim0000
    - /gen0001
    - /sim0001
    ...
```

True

```
- /ensemble_dir
    - /worker1
        - /sim0000
        - /gen0001
        - /sim0004
        ...
    - /worker2
    ...
```

Sims

sim_dirs_make [bool] = `False`:
Whether to make calculation directories for each simulation function call.

sim_dir_copy_files [list]:
Paths to files or directories to copy into each sim directory, or ensemble directory. List of strings or `pathlib.Path` objects.

sim_dir_symlink_files [list]:
Paths to files or directories to symlink into each sim directory, or ensemble directory. List of strings or `pathlib.Path` objects.

sim_input_dir [str]:
Copy this directory's contents into the working directory upon calling the simulation function.

Gens

gen_dirs_make [bool] = `False`:
Whether to make generator-specific calculation directories for each generator function call. *Each persistent generator creates a single directory*.

gen_dir_copy_files [list]:
Paths to copy into the working directory upon calling the generator function. List of strings or `pathlib.Path` objects

gen_dir_symlink_files [list]:
Paths to files or directories to symlink into each gen directory. List of strings or `pathlib.Path` objects

gen_input_dir [str]:
Copy this directory's contents into the working directory upon calling the generator function.

Profiling

profile [bool] = `False`:
Profile manager and worker logic using `cProfile`.

safe_mode [bool] = `True`:
Prevents user functions from overwriting internal fields, but requires moderate overhead.

stats_fmt [dict]:
A dictionary of options for formatting `"libE_stats.txt"`. See "Formatting Options for libE_stats.txt".

TCP

workers [list]:
TCP Only: A list of worker hostnames.

ip [str]:
TCP Only: IP address for Manager's system.

port [int]:
TCP Only: Port number for Manager's system.

authkey [str]:
TCP Only: Authkey for Manager's system.

workerID [int]:
TCP Only: Worker ID number assigned to the new process.

worker_cmd [list]:
TCP Only: Split string corresponding to worker/client Python process invocation. Contains a local Python path, calling script, and manager/server format-fields for `manager_ip`, `manager_port`, `authkey`, and `workerID`. `nworkers` is specified normally.

History

save_every_k_sims [int]:
Save history array to file after every k simulated points.

save_every_k_gens [int]:
Save history array to file after every k generated points.

save_H_and_persis_on_abort [bool] = `True`:
Save states of `H` and `persis_info` to file on aborting after an exception.

save_H_on_completion Optional[bool] = `False`
Save state of `H` to file upon completing a workflow. Also enabled when either `save_every_k_sims` or `save_every_k_gens` is set.

save_H_with_date Optional[bool] = `False`
Save `H` filename contains date and timestamp.

H_file_prefix Optional[str] = `"libE_history"`
Prefix for `H` filename.

use_persis_return_gen [bool] = `False`:
Adds persistent generator output fields to the History array on return.

use_persis_return_sim [bool] = `False`:
Adds persistent simulator output fields to the History array on return.

final_gen_send [bool] = `False`:
Send final simulation results to persistent generators before shutdown. The results will be sent along with the `PERSIS_STOP` tag.

Resources

disable_resource_manager [bool] = `False`:
Disable the built-in resource manager, including automatic resource detection and/or assignment of resources to workers. `"resource_info"` will be ignored.

platform [str]:
Name of a known platform, e.g., `LibeSpecs.platform = "perlmutter_g"` Alternatively set the `LIBE_PLATFORM` environment variable.

platform_specs [Platform|dict]:
A `Platform` object (or dictionary) specifying settings for a platform.. Fields not provided will be auto-detected. Can be set to a known platform object.

num_resource_sets [int]:
The total number of resource sets into which resources will be divided. By default resources will be divided by workers (excluding `zero_resource_workers`).

gen_num_procs [int] = `0`:
The default number of processors (MPI ranks) required by generators. Unless overridden by equivalent `persis_info` settings, generators will be allocated this many processors for applications launched via the MPIExecutor.

gen_num_gpus [int] = `0`:
The default number of GPUs required by generators. Unless overridden by the equivalent `persis_info` settings, generators will be allocated this many GPUs.

enforce_worker_core_bounds [bool] = `False`:
Permit submission of tasks with a higher processor count than the CPUs available to the worker. Larger node counts are not allowed. Ignored when `disable_resource_manager` is set.

dedicated_mode [bool] = `False`:
Disallow any resources running libEnsemble processes (manager and workers) from being valid targets for app submissions.

zero_resource_workers [list of ints]:
List of workers (by IDs) that require no resources. For when a fixed mapping of workers to resources is required. Otherwise, use `num_resource_sets`. For use with supported allocation functions.

resource_info [dict]:
Provide resource information that will override automatically detected resources. The allowable fields are given below in "Overriding Resource Auto-Detection" Ignored if `disable_resource_manager` is set.

scheduler_opts [dict]:
Options for the resource scheduler. See "Scheduler Options" for more options.

Complete Class API

pydantic model libensemble.specs.**LibeSpecs**

Scheduler Options

See options for `built-in scheduler`.

Overriding Resource Auto-Detection

Note that `"cores_on_node"` and `"gpus_on_node"` are supported for backward compatibility, but use of `Platform specification` is recommended for these settings.

Resource Info Fields

The allowable `libE_specs["resource_info"]` fields are:

```
"cores_on_node" [tuple (int, int)]:
    Tuple (physical cores, logical cores) on nodes.

"gpus_on_node" [int]:
    Number of GPUs on each node.

"node_file" [str]:
    Name of file containing a node-list. Default is "node_list".

"nodelist_env_slurm" [str]:
    The environment variable giving a node list in Slurm format
    (Default: Uses ``SLURM_NODELIST``).  Queried only if
    a ``node_list`` file is not provided and the resource manager is
    enabled.

"nodelist_env_cobalt" [str]:
```

(continues on next page)

(continued from previous page)

```
    The environment variable giving a node list in Cobalt format
    (Default: Uses ``COBALT_PARTNAME``) Queried only
    if a ``node_list`` file is not provided and the resource manager
    is enabled.

"nodelist_env_lsf" [str]:
    The environment variable giving a node list in LSF format
    (Default: Uses ``LSB_HOSTS``) Queried only
    if a ``node_list`` file is not provided and the resource manager
    is enabled.

"nodelist_env_lsf_shortform" [str]:
    The environment variable giving a node list in LSF short-form
    format (Default: Uses ``LSB_MCPU_HOSTS``) Queried only
    if a ``node_list`` file is not provided and the resource manager is
    enabled.
```

For example:

```
customizer = {cores_on_node": (16, 64),
              "node_file": "libe_nodes"}

libE_specs["resource_info"] = customizer
```

Formatting Options for libE_stats File

The allowable `libE_specs["stats_fmt"]` fields are:

```
"task_timing" [bool] = ``False``:
    Outputs elapsed time for each task launched by the executor.

"task_datetime" [bool] = ``False``:
    Outputs the elapsed time and start and end time for each task launched by the␣
↪executor.
    Can be used with the ``"plot_libe_tasks_util_v_time.py"`` to give task utilization␣
↪plots.

"show_resource_sets" [bool] = ``False``:
    Shows the resource set IDs assigned to each worker for each call of the user␣
↪function.
```

2.2.4 Allocation Specs

Allocation function specifications to be set in the user calling script. *Optional*.

Can be constructed and passed to libEnsemble as a Python class or a dictionary.

pydantic model libensemble.specs.**AllocSpecs**

Specifications for configuring an Allocation Function.

Fields

- alloc_f (Callable)
- user (dict | None)
- outputs (List[Tuple[str, Any] | Tuple[str, Any, int | Tuple]])

field alloc_f: Callable = <function give_sim_work_first>

Python function matching the `alloc_f` interface. Decides when simulator and generator functions should be called, and with what resources and parameters.

field outputs: List[Tuple[str, Any] | Tuple[str, Any, int | Tuple]] = [] (alias 'out')

List of 2- or 3-tuples corresponding to NumPy dtypes. e.g. `("dim", int, (3,))`, or `("path", str)`. Allocation functions that modify libEnsemble's History array with additional fields should list those fields here. Also used to construct libEnsemble's history array.

field user: dict | None = {'num_active_gens': 1}

A user-data dictionary to place bounds, constants, settings, or other parameters for customizing the allocation function.

Note:

- libEnsemble uses the following defaults if the user doesn't provide their own `alloc_specs`:

Listing 1: Default settings for alloc_specs

```
    alloc_f: Callable = give_sim_work_first
    """
    Python function matching the ``alloc_f`` interface. Decides when simulator and␣
↪generator functions
    should be called, and with what resources and parameters.
    """

    user: Optional[dict] = {"num_active_gens": 1}
    """
    A user-data dictionary to place bounds, constants, settings, or other parameters
    for customizing the allocation function.
    """

    outputs: List[Union[Tuple[str, Any], Tuple[str, Any, Union[int, Tuple]]]] = []
    """
    List of 2- or 3-tuples corresponding to NumPy dtypes. e.g. ``("dim", int, (3,))``, or␣
↪``("path", str)``.
    Allocation functions that modify libEnsemble's History array with additional fields␣
↪should list those
    fields here. Also used to construct libEnsemble's history array.
    """
```

- Users can import and adjust these defaults using:

```
from libensemble.specs import AllocSpecs
my_new_alloc = AllocSpecs()
my_new_alloc.alloc_f = another_function
```

See also:

- test_uniform_sampling_one_residual_at_a_time.py specifies fields to be used by the allocation function `give_sim_work_first` from fast_alloc_and_pausing.py.

```
    alloc_specs = {
        "alloc_f": give_sim_work_first,  # Allocation function
        "user": {
            "stop_on_NaNs": True,  # Should alloc preempt evals
            "batch_mode": True,  # Wait until all sim evals are done
            "num_active_gens": 1,  # Only allow one active generator
            "stop_partial_fvec_eval": True,  # Should alloc preempt evals
        },
    }
```

2.2.5 Platform Specs

libEnsemble detects platform specifications including MPI runners and resources. Usually this will result in the correct settings. However, users can configure platform specifications via the `platform_specs` option or indicate a known platform via the `platform` option.

platform_specs

A `Platform` object or dictionary specifying settings for a platform.

To define a platform (in calling script):

Platform Object

```
from libensemble.resources.platforms import Platform

libE_specs["platform_specs"] = Platform(
    mpi_runner="srun",
    cores_per_node=64,
    logical_cores_per_node=128,
    gpus_per_node=8,
    gpu_setting_type="runner_default",
    gpu_env_fallback="ROCR_VISIBLE_DEVICES",
    scheduler_match_slots=False,
)
```

Dictionary

```
libE_specs["platform_specs"] = {
    "mpi_runner": "srun",
    "cores_per_node": 64,
    "logical_cores_per_node": 128,
    "gpus_per_node": 8,
    "gpu_setting_type": "runner_default",
    "gpu_env_fallback": "ROCR_VISIBLE_DEVICES",
    "scheduler_match_slots": False,
}
```

The list of platform fields is given below. Any fields not given will be auto-detected by libEnsemble.

Platform Fields

pydantic model libensemble.resources.platforms.**Platform**

Config

- **arbitrary_types_allowed**: *bool = True*
- **populate_by_name**: *bool = True*
- **extra**: *str = forbid*
- **validate_assignment**: *bool = True*

To use an existing platform:

```
from libensemble.resources.platforms import PerlmutterGPU

libE_specs["platform_specs"] = PerlmutterGPU()
```

See known platforms.

platform

A string giving the name of a known platform defined in the platforms module.

```
libE_specs["platform"] = "perlmutter_g"
```

Note: the environment variable LIBE_PLATFORM is an alternative way of setting.

E.g., in the command line or batch submission script:

```
export LIBE_PLATFORM="perlmutter_g"
```

Known Platforms List

Known_platforms

pydantic model libensemble.resources.platforms.**Known_platforms**

A list of platforms with known configurations.

There are three ways to specify a known system:

["platform_specs"]

```
from libensemble.resources.platforms import PerlmutterGPU

libE_specs["platform_specs"] = PerlmutterGPU()
```

["platform"]

```
libE_specs["platform"] = "perlmutter_g"
```

export LIBE_PLATFORM

On command-line or batch submission script:

```
export LIBE_PLATFORM="perlmutter_g"
```

If the platform is not specified, libEnsemble will attempt to detect known platforms (this is not guaranteed).

Note: libEnsemble should work on any platform, and detects most system configurations correctly. These options are helpful for optimization and where auto-detection encounters ambiguity or an unknown feature.

field generic_rocm: GenericROCm

field crusher: Crusher

field frontier: Frontier

field perlmutter_c: PerlmutterCPU

field perlmutter_g: PerlmutterGPU

field polaris: Polaris

field spock: Spock

field summit: Summit

field sunspot: Sunspot

2.2.6 persis_info

Holds persistent information that can be updated during the ensemble.

An initialized `persis_info` dictionary can be provided to the `libE()` call or as an attribute of the `Ensemble` class.

Dictionary keys that have an integer value contain entries that are passed to and from the corresponding workers. These are received in the `persis_info` argument of user functions, and returned as the optional second return value.

A typical example is a random number generator stream to be used in consecutive calls to a generator (see `add_unique_random_streams()`)

All other entries persist on the manager and can be updated in the calling script between ensemble invocations, or in the allocation function.

Examples:

RNG or reusable structures

Listing 2: libensemble/libensemble/gen_funcs/sampling.py

```
def uniform_random_sample(_, persis_info, gen_specs):
    """
    Generates ``gen_specs["user"]["gen_batch_size"]`` points uniformly over the domain
    defined by ``gen_specs["user"]["ub"]`` and ``gen_specs["user"]["lb"]``.

    .. seealso::
        `test_uniform_sampling.py <https://github.com/Libensemble/libensemble/blob/
↪develop/libensemble/tests/functionality_tests/test_uniform_sampling.py>`_ # noqa
    """
    ub = gen_specs["user"]["ub"]
    lb = gen_specs["user"]["lb"]

    n = len(lb)
    b = gen_specs["user"]["gen_batch_size"]

    H_o = np.zeros(b, dtype=gen_specs["out"])

    H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))

    return H_o, persis_info


```

Incrementing indexes or process counts

Listing 3: libensemble/alloc_funcs/fast_alloc.py

```
    for wid in support.avail_worker_ids():
        # Skip any cancelled points
        while persis_info["next_to_give"] < len(H) and H[persis_info["next_to_give"]][
↪"cancel_requested"]:
            persis_info["next_to_give"] += 1

```

Tracking running generators

Listing 4: libensemble/alloc_funcs/start_only_persistent.py

```
        avail_workers = support.avail_worker_ids(persistent=False, zero_resource_
↪workers=True)

        for wid in avail_workers:
            if gen_count < user.get("num_active_gens", 1):
                # Finally, start a persistent generator as there is nothing else to do.
                try:
                    Work[wid] = support.gen_work(
```

(continues on next page)

(continued from previous page)

```
                        wid,
                        gen_specs.get("in", []),
                        range(len(H)),
                        persis_info.get(wid),
                        persistent=True,
                        active_recv=active_recv_gen,
                    )
                except InsufficientFreeResources:
                    break

                persis_info["num_gens_started"] = persis_info.get("num_gens_started", 0) + 1
                gen_count += 1

```

Allocation function triggers shutdown

Listing 5: libensemble/alloc_funcs/start_only_persistent.py

```
    if gen_count < persis_info.get("num_gens_started", 0):
        # When a persistent worker is done, trigger a shutdown (returning exit condition of 1)
        return Work, persis_info, 1

```

When there are repeated calls to `libE()` or `ensemble.run()`, users may need to modify or reset the contents of `persis_info` in some cases.

See also:

From: support.py

```
persis_info_1 = {
    "total_gen_calls": 0,  # Counts gen calls in alloc_f
    "last_worker": 0,  # Remembers last gen worker in alloc_f
    "next_to_give": 0,  # Remembers next H row to give in alloc_f
}

persis_info_1[0] = {
    "run_order": {},  # Used by manager to remember run order
    "total_runs": 0,  # Used by manager to count total runs
    "rand_stream": np.random.default_rng(1),
}
```

2.2.7 Exit Criteria

The following criteria (or termination tests) can be used to configure when to stop a workflow.

Can be constructed and passed to libEnsemble as a Python class or a dictionary.

pydantic model libensemble.specs.**ExitCriteria**

Specifications for configuring when libEnsemble should stop a given run.

Fields

- sim_max (int | None)
- gen_max (int | None)
- wallclock_max (float | None)
- stop_val (Tuple[str, float] | None)

field gen_max: int | None = None

Stop when this many new points have been generated by generator functions.

field sim_max: int | None = None

Stop when this many new points have been evaluated by simulation functions.

field stop_val: Tuple[str, float] | None = None

Stop when `H[str] < float` for the given `(str, float)` pair.

field wallclock_max: float | None = None

Stop when this many seconds has elapsed since the manager initialized.

See also:

From test_persistent_aposmm_dfols.py.

```
exit_criteria = {
    "sim_max": 1000,
    "wallclock_max": 100,
    "stop_val": ("f", 3000),
}
```

2.3 Output Management

2.3.1 Default Log Files

The history array `H` and `persis_info` dictionary are returned to the user by libEnsemble. If libEnsemble aborts on an exception, these structures are dumped automatically to these files:

- `libE_history_at_abort_<sim_count>.npy`
- `libE_persis_info_at_abort_<sim_count>.pickle`

Two other libEnsemble files produced by default:

- `libE_stats.txt`: One-line summaries for each user calculation.
- `ensemble.log`: Logging output. Multiple runs will append output if this file isn't removed. See below for config info.

Global options:

`libE_specs["disable_log_files"] = True`: Disable output files

`libE_specs["use_workflow_dir"] = True`: Place output files in workflow-instance directories

`libE_specs["save_H_and_persis_on_abort"] = False`: Disable dumping the History array and `persis_info` to files

```
from libensemble.specs import LibeSpecs

specs = LibeSpecs(disable_log_files=True, save_H_and_persis_on_abort=False)
```

2.3.2 Logger Configuration

The libEnsemble logger uses the standard Python logging levels (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`) plus one additional custom level (`MANAGER_WARNING`) between `WARNING` and `ERROR`.

The default level is `INFO`, which includes information about how tasks are submitted and when tasks are killed. To gain additional diagnostics, set the logging level to `DEBUG`. libEnsemble writes to `ensemble.log` by default. A log file name can also be supplied.

To change the logging level to `DEBUG`:

```
from libensemble import logger
logger.set_level("DEBUG")
```

Logger messages of `MANAGER_WARNING` level or higher are also displayed through stderr by default. This boundary can be adjusted:

```
from libensemble import logger

# Only display messages with level >= ERROR
logger.set_stderr_level("ERROR")
```

stderr displaying can be effectively disabled by setting the stderr level to `CRITICAL`.

Logger Module

`logger.`**`set_level`**(*level*)

Sets libEnsemble logging level

Parameters
level (*int*) –

Return type
None

`logger.`**`get_level`**()

Returns libEnsemble logging level

Return type
int

`logger.`**`set_filename`**(*filename*)

Sets logger filename if loggers not yet created, else None

Parameters
filename (*str*) –

Return type
None

`logger.set_directory(dirname)`

Sets target directory to contain logfiles if loggers not yet created

Parameters
dirname (*str*) –

Return type
None

`logger.set_stderr_level(level)`

Sets logger to mirror certain messages to stderr

Parameters
level (*int*) –

Return type
None

`logger.get_stderr_level()`

Returns libEnsemble stderr logging level

Return type
int

Note: The `scripts` directory, in the libEnsemble project root directory, contains scripts to compare outputs and create plots based on the ensemble output.

2.3.3 Analysis Utilities

Analysis Utilities

Timing analysis scripts

Note that all plotting scripts produce a file rather than opening a plot interactively.

The following scripts must be run in the directory with the `libE_stats.txt` file. They extract and plot information from that file.

- `plot_libe_calcs_util_v_time.py`: Extracts worker utilization vs. time plot (with one-second sampling). Shows the number of workers running user sim or gen functions over time.
- `plot_libe_tasks_util_v_time.py`: Extracts launched task utilization v time plot (with one-second sampling). Shows the number of workers with active tasks, launched via the executor, over time.
- `plot_libe_histogram.py`: Creates histogram showing the number of completed/killed/failed user calculations binned by run time.

Results analysis scripts

- `print_npy.py`: Prints to screen from a given `*.npy` file containing a NumPy structured array. Use `done` to print only the lines containing `"sim_ended"` points. Example:

```
./print_npy.py run_libe_forces_results_History_length=1000_evals=8.npy done
```

- `print_fields.py`: Prints to screen from a given `*.npy` file containing a NumPy structured array. This is a more versatile version of `print_npy.py` that allows the user to select fields to print and boolean conditions determining which rows are printed (see `./print_fields.py -h` for usage).
- `compare_npy.py`: Compares either two provided `*.npy` files or one provided `*.npy` file with an expected results file (by default located at ../expected.npy). A tolerance is given on floating-point results, and NANs are compared as equal. Variable fields (such as those containing a time) are ignored. These fields may need to be modified depending on the user's history array.
- `plot_pareto_2d.py`: Loop through objective points in f and extract the Pareto front. Arguments are an `*.npy` file and a budget.
- `plot_pareto_3d.py`: Loop through objective points in f and extract the Pareto front. Arguments are an `*.npy` file and a budget.
- `print_pickle.py`: Prints to screen from a given `*.pickle` file. Example:

```
./print_pickle.py persis_info_length=1000_evals=1000_workers=2.pickle
```

2.4 History Array

```
H: numpy structured array
    A record of runtime attributes and output data for all ensemble members.
```

2.4.1 Overview

libEnsemble uses a NumPy structured array to store information about each point (ensemble member) generated and processed in the ensemble.

The manager maintains a global copy. Each row contains:

1. Data generated by the `gen_f`
2. Resultant output from the `sim_f`
3. `Reserved fields` containing metadata

When the history array is initialized, it creates fields for each `gen_specs["out"]` and `sim_specs["out"]` entry. These entries may resemble:

```
gen_specs["out"] = [("x", float, 2), ("theta", int)]
sim_specs["out"] = [("f", float)]
```

Therefore, the `gen_f` and `sim_f` must return output as NumPy structured arrays for slotting into these fields.

Ensure input/output field names for a function match each other or a `reserved field`:

```
gen_specs["out"] = [("x", float, 2), ("theta", int)]  # produces "x" and "theta"
sim_specs["in"] = ["x", "theta", "sim_id"]  # accepts "x", "theta" and "sim_id", a␣
↪reserved field
```

2.4.2 Reserved Fields

User fields and reserved fields are combined together in the final History array returned by libEnsemble.

These reserved fields can be modified to adjust how/when a point is evaluated:

- `sim_id` [int]: Each unit of work must have a `sim_id`. This can be set by the generator or by the manager by default. Users should ensure these IDs are sequential and unique when running multiple generators.
- `cancel_requested` [bool]: Can be set `True` in a generator to request attempted cancellation of the corresponding simulation.

The following fields are automatically populated by libEnsemble:

`gen_worker` [int]: Worker that generated this entry

`gen_started_time` [float]: Time gen_worker was initiated that produced this entry

`gen_ended_time` [float]: Time gen_worker requested this entry

`sim_worker` [int]: Worker that did (or is doing) the sim evaluation for this entry

`sim_started` [bool]: `True` if entry was given to sim_worker for sim evaluation

`sim_started_time` [float]: Time entry was given to sim_worker for a sim evaluation

`sim_ended` [bool]: `True` if entry's sim evaluation completed

`sim_ended_time` [float]: Time entry's sim evaluation completed

`gen_informed` [bool]: `True` if gen_worker was informed about the sim evaluation of this entry

`gen_informed_time` [float]: Time gen_worker was informed about the sim evaluation of this entry

`kill_sent` [bool]: `True` if a kill signal was sent to worker for this entry

Other than `"sim_id"` and `cancel_requested`, these fields cannot be overwritten by user functions unless `libE_specs["safe_mode"]` is set to `False`.

> **Warning:** Adjusting values in protected fields may crash libEnsemble.

2.4.3 Example Workflow updating History

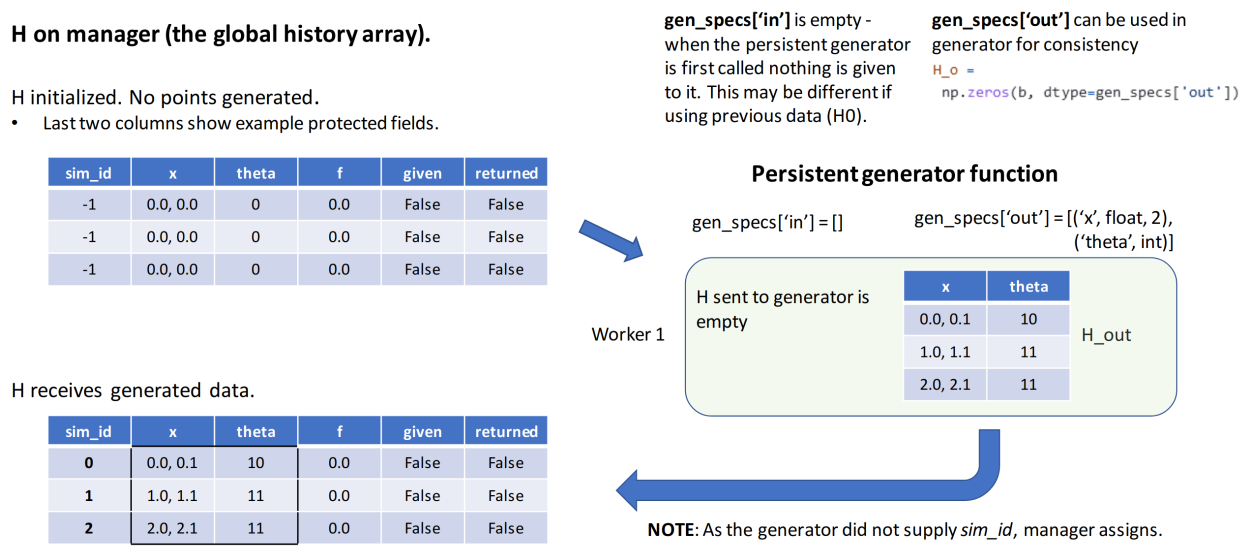
Step 1: The history array is initialized on the manager

The history array is initialized using the libEnsemble reserved field and the user-provided `gen_specs["out"]` and `sim_specs["out"]` entries. In the figure below, only the reserved fields: `sim_id`, `sim_started`, and `sim_ended` are shown for brevity.

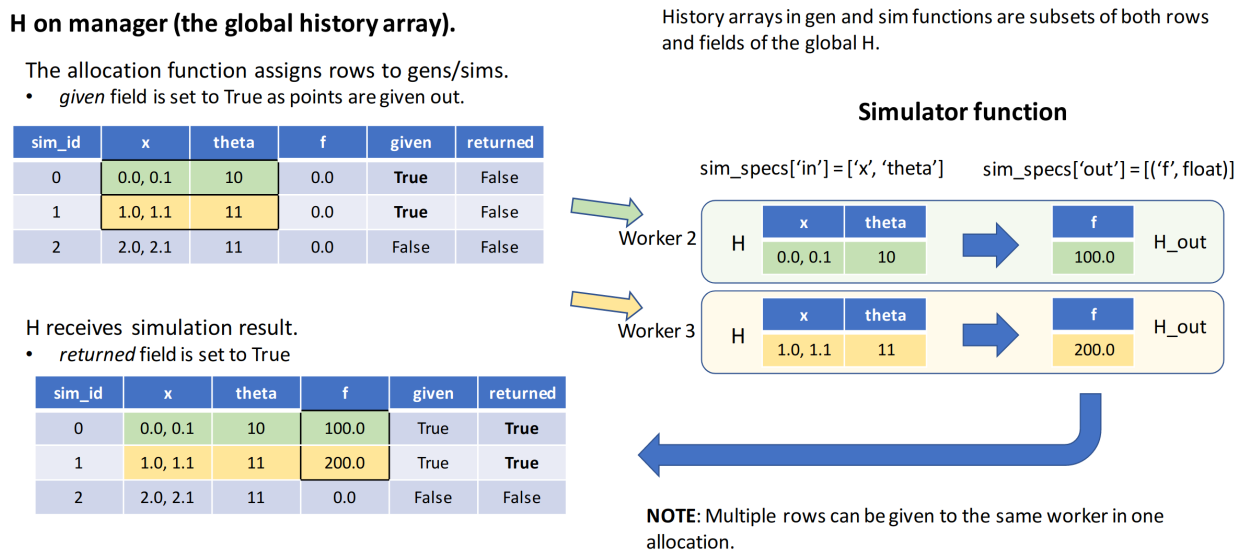
sim_id	x	theta	f	given	returned
-1	0.0, 0.0	0	0.0	False	False
-1	0.0, 0.0	0	0.0	False	False
-1	0.0, 0.0	0	0.0	False	False

`gen_f` and `sim_f` functions accept a local history array as the first argument that contains only the rows and fields specified. For new function calls these will be specified by either `gen_specs["in"]` or `sim_specs["in"]`. For generators this may be empty.

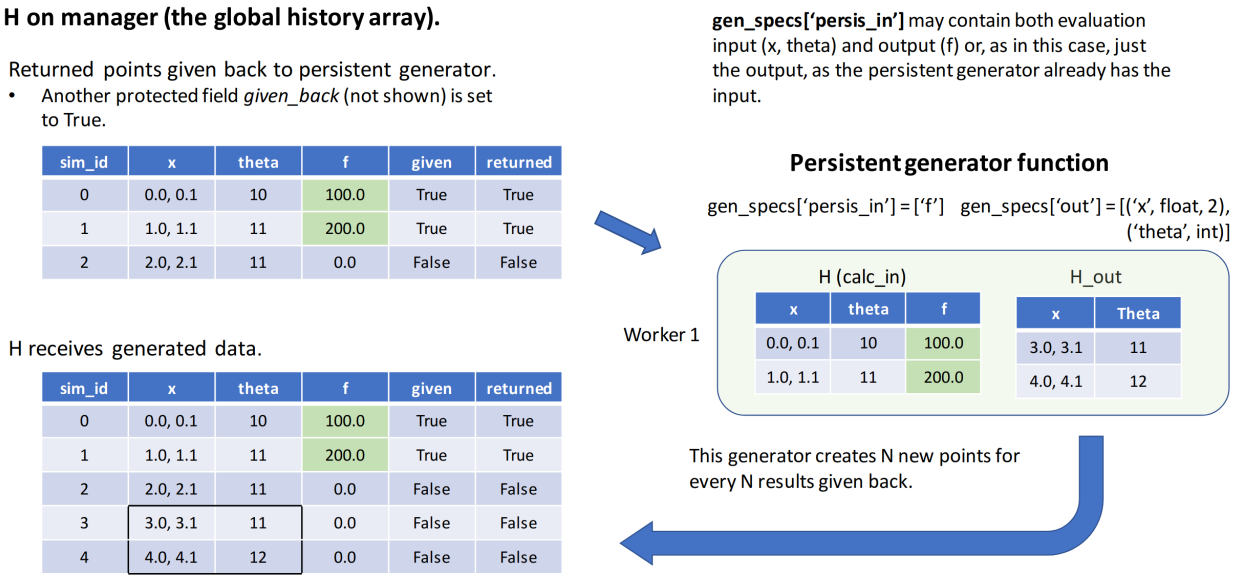
Step 2: Persistent generator `gen_f` is called



Step 3: Points are given out for `sim_f` to evaluate



Step 4: Results returned to persistent generator `gen_f`



2.5 Resource Manager

libEnsemble comes with built-in resource management. This entails the detection of available resources (e.g., nodelists, core counts, and GPUs), and the allocation of resources to workers.

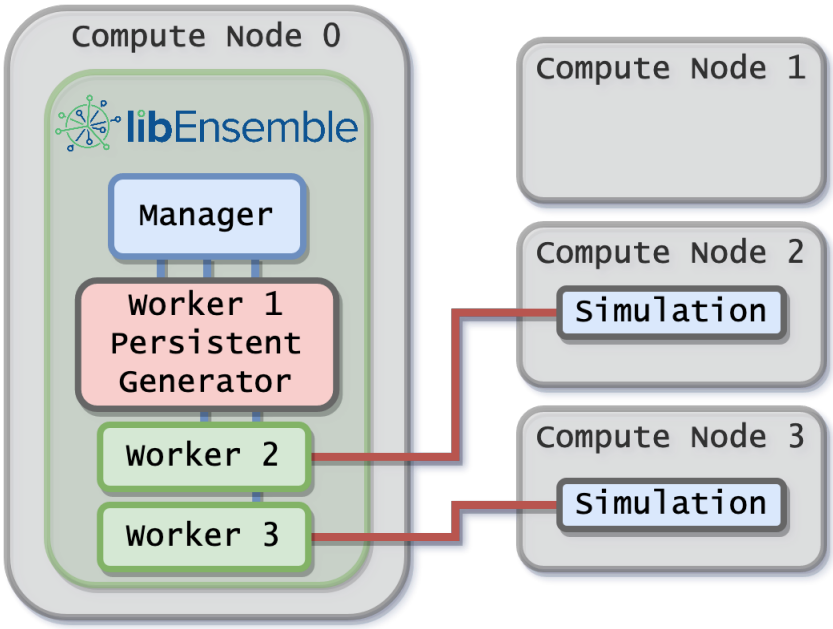
Resource management can be disabled by setting `libE_specs["disable_resource_manager"] = True`. This will prevent libEnsemble from doing any resource detection or management.

2.5.1 Zero-resource workers

Users with persistent `gen_f` functions may notice that the persistent workers are still automatically assigned resources. This can be wasteful if those workers only run `gen_f` functions in-place (i.e., they do not use the Executor to submit applications to allocated nodes). Suppose the user is using the `parse_args()` function and runs:

```
python run_ensemble_persistent_gen.py --comms local --nworkers 3
```

If three nodes are available in the node allocation, the result may look like the following.



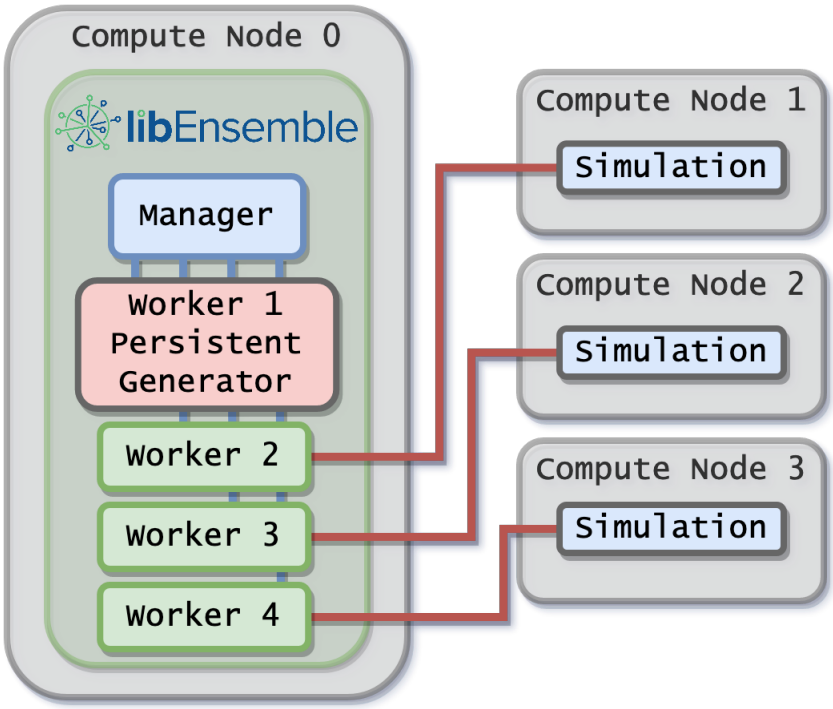
To avoid the the wasted node above, add an extra worker:

```
python run_ensemble_persistent_gen.py --comms local --nworkers 4
```

and in the calling script (*run_ensemble_persistent_gen.py*), explicitly set the number of resource sets to the number of workers that will be running simulations.

```
nworkers, is_manager, libE_specs, _ = parse_args()
libE_specs["num_resource_sets"] = nworkers - 1
```

When the `num_resource_sets` option is used, libEnsemble will use the dynamic resource scheduler, and any worker may assign work to any node. This works well for most users.



Optional: An alternative way to express the above would be to use the command line:

```
python run_ensemble_persistent_gen.py --comms local --nsim_workers 3
```

This would automatically set the `num_resource_sets` option and add a single worker for the persistent generator - a common use-case.

In general, the number of resource sets should be set to enable the maximum concurrency desired by the ensemble, taking into account generators and simulators.

Users can set generator resources using the *libE_specs* options `gen_num_procs` and/or `gen_num_gpus`, which take integer values. If only `gen_num_gpus` is set, then the number of processors is set to match.

To vary generator resources, `persis_info` settings can be used in allocation functions before calling the `gen_work` support function. This takes the same options (`gen_num_procs` and `gen_num_gpus`).

Alternatively, the setting `persis_info["gen_resources"]` can also be set to a number of resource sets.

The available nodes are always divided by the number of resource sets, and there may be multiple nodes or a partition of a node in each resource set. If the split is uneven, resource sets are not split between nodes. For example, if there are two nodes and five resource sets, one node will have three resource sets, and the other will have two.

Placing zero-resource functions on a fixed worker

If the generator must always be on worker one, then instead of using `num_resource_sets`, use the `zero_resource_workers` *libE_specs* option:

```
libE_specs["zero_resource_workers"] = [1]
```

in the calling script and worker one will not be allocated resources. In general, set the parameter `zero_resource_workers` to a list of worker IDs that should not have resources assigned.

This approach can be useful if running in `distributed mode`.

The use of the `zero_resource_workers` *libE_specs* option must be supported by the allocation function, see `start_only_persistent`)

2.5.2 Dynamic Assignment of Resources

Overview

libEnsemble comes with built-in resource management. This entails the `detection of available resources` (e.g., nodelists, core counts, and GPUs), and the allocation of resources to workers.

By default, the provisioned resources are divided by the number of workers. libEnsemble's `MPI Executor` is aware of these supplied resources, and if not given any of `num_nodes`, `num_procs`, or `procs_per_node` in the submit function, it will try to use all nodes and CPU cores available to the worker.

Detected resources can be overridden using the `libE_specs` option `resource_info`.

Variable resource assignment

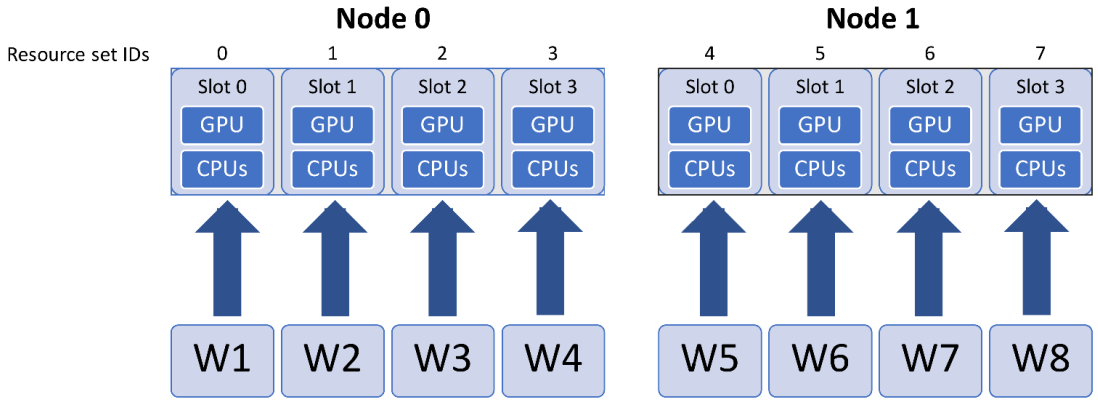
Note: As of **version 0.10.0**, the concept of resource sets is not needed. The generator can use special `gen_specs["out"]` fields of `num_procs` and `num_gpus` for each simulation generated. These will be used to assign resources and will be automatically passed through and used by the executor (if no other run configuration is given in the submit line). Furthermore, GPUs will be automatically assigned in the correct way for the given system (including Nvidia, AMD, and Intel GPUs); you do not need to set `CUDA_VISIBLE_DEVICES` or equivalent. Example: test_GPU_variable_resources.py

In slightly more detail, the resource manager divides resources into **resource sets**. One resource set is the smallest unit of resources that can be assigned (and dynamically reassigned) to workers. By default, the provisioned resources are divided by the number of workers (excluding any workers given in the `zero_resource_workers` `libE_specs` option). However, it can also be set directly by the `num_resource_sets` `libE_specs` option. If the latter is set, the dynamic resource assignment algorithm will always be used.

If there are more resource sets than nodes, then the resource sets on each node will be given a slot number, enumerated from zero. For example, if there are three slots on a node, they will have slot numbers 0, 1, and 2.

The resource manager will not split a resource set over nodes, rather the resource sets on each node will be the integer division of resource sets over nodes, with the remainder dealt out from the first node. Even breakdowns are generally preferable, however.

For example, say a given system has four GPUs per node, and the user has run libEnsemble on two nodes, with eight workers. The default division of resources would be:



Variable Size simulations

A dynamic assignment of resources to simulation workers can be achieved by the convention of using a field in the history array called `resource_sets`. While this is technically a user space field, the allocation functions are set up to read this field, check available resources, and assign resource sets to workers, along with the work request (simulation).

In the calling script, use a `gen_specs["out"]` field called `resource_sets`:

```
gen_specs = {
    "gen_f": gen_f,
    "in": ["sim_id"],
    "out": [
        ("priority", float),
        ("resource_sets", int),
        ("x", float, n),
    ],
}
```

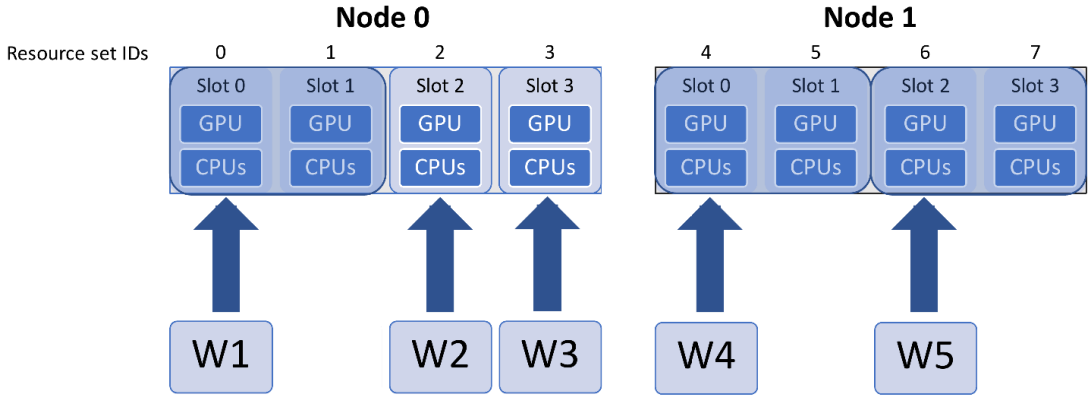
For an example calling script, see the regression test test_persistent_sampling_CUDA_variable_resources.py

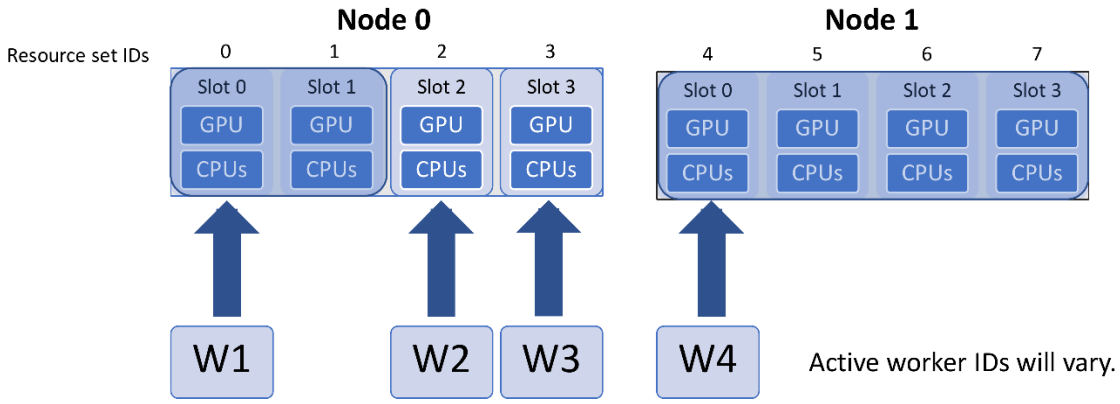
In the generator, the `resource_sets` field must be set to a value for each point (simulation) generated (if it is not set, it will have the initialized value of zero, and supply zero resources).

```
H_o = np.zeros(b, dtype=gen_specs["out"])
for i in range(0, b):
    H_o["x"][i] = x[b]
    H_o["resource_sets"][i] = sim_size[b]
```

For an example generator, see the *uniform_sample* function in persistent_sampling_var_resources.py

When the allocation function assigns the points to workers for evaluation, it will check if the requested number of resource sets are available for each point to evaluate. If they are not available, then the evaluation will not be given to a worker until enough resources become available. This functionality is built into the supplied allocation functions and generally requires no modification from the user.





The particular nodes and slots assigned to each worker will be determined by the libEnsenble `built-in scheduler`, although users can provide an alternative scheduler via the `allocation function`. In short, the scheduler will prefer fitting simulations onto a node, and using even splits across nodes, if necessary.

Accessing resources from the simulation function

In the user's simulation function, the resources supplied to the worker can be `interrogated directly via the resources class attribute`. libEnsemble's executors (e.g., the `MPI Executor`) are aware of these supplied resources, and if not given any of `num_nodes`, `num_procs`, or `procs_per_node` in the submit function, it will try to use all nodes and CPU cores available.

var_resources.py has two examples of how resource information for the worker may be accessed in the sim function (*multi_points_with_variable_resources* and *CUDA_variable_resources*).

For example, in *CUDA_variable_resources*, the environment variable `CUDA_VISIBLE_DEVICES` is set to slots:

```
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")  # Use convenience function.
num_nodes = resources.local_node_count
cores_per_node = resources.slot_count  # One CPU per GPU
```

In the figure above, this would result in worker one setting:

```
export CUDA_VISIBLE_DEVICES=0,1
```

while worker five would set:

```
export CUDA_VISIBLE_DEVICES=2,3
```

Note: If the user sets the number of resource sets directly using the `num_resource_sets` `libE_specs` option, then the dynamic resource assignment algorithm will always be used. If `resource_sets` is not a field in `H`, then each worker will use one resource set.

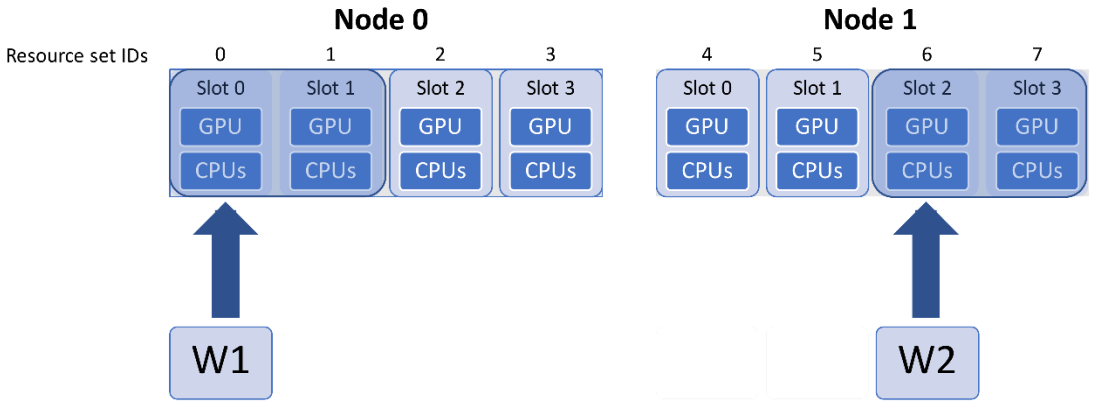
Resource Scheduler Options

The following options are available for the `built-in scheduler` and can be set by a dictionary supplied via `libE_specs["scheduler_opts"]`

split2fit [boolean]
: Try to split resource sets across more nodes if space is not currently available on the minimum node count required. Allows more efficient scheduling. Default: True

match_slots [boolean]:
: When splitting resource sets across multiple nodes, slot IDs must match. Useful if setting an environment variable such as `CUDA_VISIBLE_DEVICES` to specific slot counts, which should match over multiple nodes. Default: True

In the following example, assume the next simulation requires **four** resource sets. This could fit on one node if all slots were free, but only two are free on each node.



`split2fit` allows the two resource sets to be used on each node. However, the task will not be scheduled unless `match_slots` is set to *False*:

```
libE_specs["scheduler_opts"] = {"match_slots": False}
```

This is only recommended if not enumerating resources to slot IDs (e.g., via `CUDA_VISIBLE_DEVICES`).

Note that if six resource sets were requested, then they would be split three per node, even if `split2fit` is *False*, as this could otherwise never be scheduled.

Varying generator resources

By default, generators are not allocated resources in dynamic mode. Fixed resources for the generator can be set using the *libE_specs* options `gen_num_procs` and `gen_num_gpus`, which take integer values. If only `gen_num_gpus` is set, then the number of processors will be set to match.

To vary generator resources, `persis_info` settings can be used in allocation functions before calling the `gen_work` support function. This takes the same options (`gen_num_procs` and `gen_num_gpus`).

Alternatively, the setting `persis_info["gen_resources"]` can also be set to a number of resource sets.

Note that persistent workers maintain their resources until they come out of a persistent state.

Example scenarios

Persistent generator

You have *one* persistent generator and want *eight* workers to run concurrent simulations. In this case you can run with *nine* workers.

Either explicitly set eight resource sets (recommended):

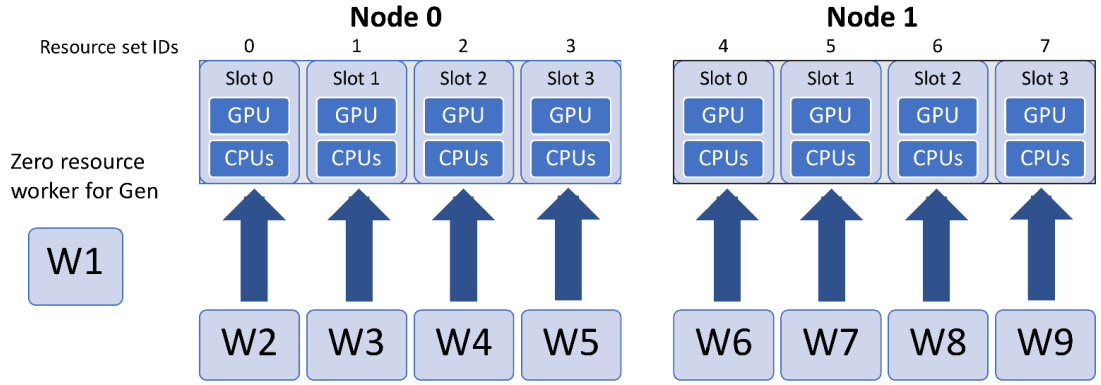
```
libE_specs["num_resource_sets"] = 8
```

Or if the generator should always be the same worker, use one zero-resource worker:

```
libE_specs["zero_resource_workers"] = [1]
```

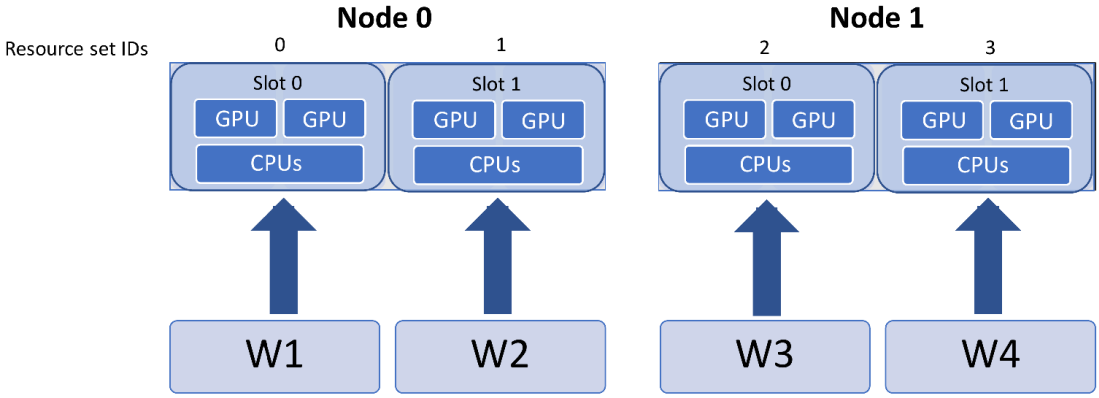
For the second option, an allocation function supporting zero-resource workers must be used.

Using the two-node example above, the initial worker mapping in this example will be:



Using large resource sets

Note that resource_sets and slot numbers are based on workers by default. If you halved the workers in this example you would have the following (each resource set has twice the CPUs and GPUs).

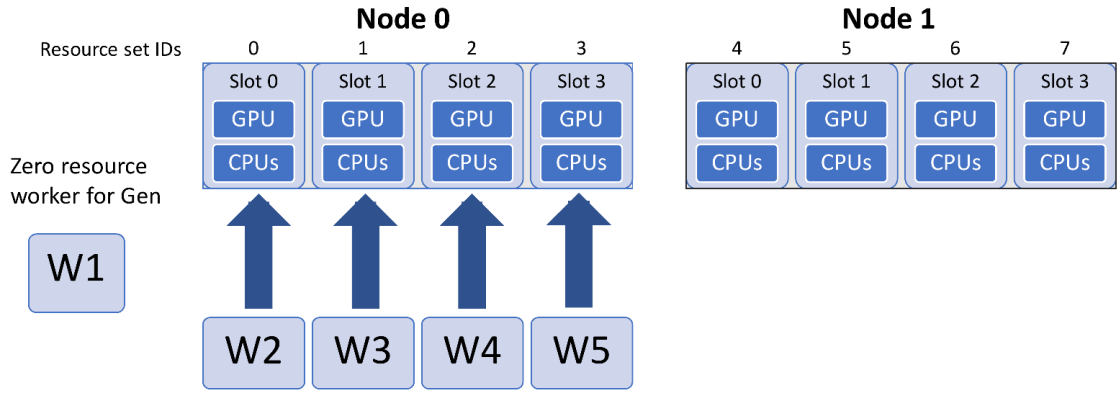


To set CUDA_VISIBLE_DEVICES to slots in this case, use the `multiplier` argument in the `set_env_to_slots` function:

```
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES", multiplier=2)
```

Setting more resource sets than workers

Resource sets can be set to more than the number of corresponding workers. In this example there are 5 workers (one for the generator) and 8 resource sets. The additional resources will be used for larger simulations.



This could be achieved by setting:

```
libE_specs["num_resource_sets"] = 8
```

and running on 5 workers.

Also, this can be set on the command line as a convenience.

```
python run_ensemble.py --comms local --nworkers 5 --nresource_sets 8
```

2.5.3 Resource Detection

The resource manager can detect system resources, and partition these to workers. The `MPI Executor` accesses the resources available to the current worker when launching tasks.

Node-lists are detected by an environment variable on the following systems:

Scheduler	Nodelist Env. variable
SLURM	SLURM_NODELIST
COBALT	COBALT_PARTNAME
LSF	LSB_HOSTS/LSB_MCPU_HOSTS
PBS	PBS_NODEFILE

These environment variable names can be modified via the `resource_info` `libE_specs` option.

On other systems you may have to supply a node list in a file called **node_list** in your run directory. For example, on ALCF system Cooley, the session node list can be obtained as follows:

```
cat $COBALT_NODEFILE > node_list
```

Resource detection can be disabled by setting `libE_specs["disable_resource_manager"] = True`, and users can simply supply run configuration options on the Executor submit line.

This will usually work sufficiently on systems that have application-level scheduling and queuing (e.g., `jsrun` on Summit). However, on many cluster and multi-node systems, if the built-in resource manager is disabled, then runs without a hostlist or machinefile supplied may be undesirably scheduled to the same nodes.

System detection for resources can be overridden using the `resource_info` `libE_specs` option.

2.5.4 Scheduler Module

The scheduler is called within the scope of the allocation function, usually via the `alloc_support` module function `assign_resources()` (either called directly or via `sim_work()` or `gen_work()`), which is a wrapper for the main scheduler function `assign_resources()`.

The alloc_support module allows users to supply an alternative scheduler that fits this interface. This could be achieved, for example, by inheriting the built-in scheduler and making modifications.

Options can also be provided to the scheduler though the `libE_specs["scheduler_opts"]` dictionary.

class resources.scheduler.**ResourceScheduler**(*user_resources=None*, *sched_opts={}*)

Calculates and returns resource set ids from a dictionary of resource sets by group. The available resource sets are read initially from the resources module or from a resources object passed in.

Resource sets are locally provisioned to work items by a call to the `assign_resources` function, and a cache of available resource sets is maintained for the life of the object (usually corresponding to one call of the allocation function). Note that work item resources are formally assigned to workers only when a work item is sent to the worker.

__init__(*user_resources=None*, *sched_opts={}*)

Initiate a ResourceScheduler object

Parameters

- **user_resources** (*Resources, optional*) – A resources object. If present overrides the class variable.
- **sched_opts** (*dict, optional*) – A dictionary of scheduler options. Passed via `libE_specs["scheduler_opts"]`

The supported fields for sched_opts are:

```
"split2fit" [Boolean]:
    Try to split resource sets across more nodes if space is not currently
    available on the minimum node count required. Allows more efficient
    scheduling.
    Default: True

"match_slots" [Boolean]:
    When splitting resource sets across multiple nodes, slot IDs must match.
    Useful if setting an environment variable such as ``CUDA_VISIBLE_DEVICES``
    to specific slots, which should match over multiple nodes.
    Default: True
```

assign_resources(*rsets_req*, *use_gpus=None*, *user_params=[]*)

Schedule resource sets to a work item if possible.

If the resources required are less than one node, they will be allocated to the smallest available sufficient slot.

If the resources required are more than one node, then the scheduler will attempt to find an even split. If no even split is possible, then enough additional resource sets will be assigned to enable an even split.

Returns a list of resource set IDs or raises an exception (either InsufficientResourcesError or InsufficientFreeResources).

2.5.5 Worker Resources Module

The worker resources module can be interrogated by the user function on a worker to obtain the following attributes. The convenience functions below can also be used.

class resources.worker_resources.**WorkerResources**(*num_workers*, *resources*, *workerID*)

Bases: RSetResources

Provide system resources per worker to libEnsemble and executor.

Object Attributes:

Some of these attributes may be updated as the ensemble progresses.

`rsets` below is used to abbreviate `resource sets`.

Variables

- **workerID** (*int*) – workerID for this worker.
- **local_nodelist** (*list*) – A list of all nodes assigned to this worker.
- **rset_team** (*list*) – List of rset IDs currently assigned to this worker.
- **num_rsets** (*int*) – The number of resource sets assigned to this worker.
- **slots** (*dict*) – A dictionary with a list of slot IDs for each node.
- **even_slots** (*bool*) – True if each node has the same number of slots.
- **matching_slots** (*bool*) – True if each node has matching slot IDs.
- **slot_count** (*int*) – The number of slots per node if even_slots is True, else None.
- **slots_on_node** (*list*) – A list of slots IDs if matching_slots is True, else None.
- **local_node_count** (*int*) – The number of nodes available to this worker (rounded up to whole number).
- **rsets_per_node** (*int*) – The number of rsets per node (if a rset > 1 node, will be 1).

The worker_resources attributes can be queried, and convenience functions called, via the resources class attribute. For example:

With resources imported:

```
from libensemble.resources.resources import Resources
```

A user function (sim/gen) may do:

```
resources = Resources.resources.worker_resources
num_nodes = resources.local_node_count
cores_per_node = resources.slot_count  # One CPU per GPU
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")  # Use convenience function.
```

Note that **slots** are resource sets enumerated on a node (starting with zero). If a resource set has more than one node, then each node is considered to have slot zero.

If `even_slots` is True, then the attributes `slot_count` will give the number of slots on each node. If `matching_slots` is True, then `slots_on_node` will give the slot IDs for all nodes. These can be used for simplicity; otherwise, the `slots` dictionary can be used to get information for each node.

get_slots_as_string(*multiplier=1*, *delimiter=','*, *limit=None*)

Returns list of slots as a string

Parameters

- **multiplier** – Optional int. Assume this many items per slot.
- **delimiter** – Optional int. Delimiter for output string.
- **limit** – Optional int. Maximum slots (truncate list after this many slots).

set_env_to_slots(*env_var*, *multiplier=1*, *delimiter=','*)

Sets the given environment variable to slots

Parameters

- **env_var** – String. Name of environment variable to set.
- **multiplier** – Optional int. Assume this many items per slot.
- **delimiter** – Optional int. Delimiter for output string.

Example usage in a sim function:

With resources imported:

```
from libensemble.resources.resources import Resources
```

Obtain worker resources:

```
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")
```

set_env_to_gpus(*env_var=None*, *delimiter=','*)

Sets the given environment variable to GPUs

Parameters

- **env_var** – String. Name of environment variable to set.
- **delimiter** – Optional int. Delimiter for output string.

Example usage in a sim function:

With resources imported:

```
from libensemble.resources.resources import Resources
```

Obtain worker resources:

```
resources = Resources.resources.worker_resources
resources.set_env_to_gpus("CUDA_VISIBLE_DEVICES")
```

2.6 Writing User Functions

User functions typically require only some familiarity with NumPy, but if they conform to the user function APIs, they can incorporate methods from machine-learning, mathematics, resource management, or other libraries/applications.

These guides describe common development patterns and optional components:

2.6.1 Generator Functions

Generator and Simulator functions have relatively similar interfaces.

```
def my_generator(Input, persis_info, gen_specs, libE_info):
    batch_size = gen_specs["user"]["batch_size"]

    Output = np.zeros(batch_size, gen_specs["out"])
    ...
    Output["x"], persis_info = generate_next_simulation_inputs(Input["f"], persis_info)

    return Output, persis_info
```

Most gen_f function definitions written by users resemble:

```
def my_generator(Input, persis_info, gen_specs, libE_info):
```

where:

- Input is a selection of the History array
- persis_info is a dictionary containing state information
- gen_specs is a dictionary of generator parameters, including which fields from the History array got sent
- libE_info is a dictionary containing libEnsemble-specific entries

Valid generator functions can accept a subset of the above parameters. So a very simple generator can start:

```
def my_generator(Input):
```

If gen_specs was initially defined:

```
gen_specs = {
    "gen_f": some_function,
    "in": ["f"],
    "out:" ["x", float, (1,)],
    "user": {
        "batch_size": 128
    }
}
```

Then user parameters and a *local* array of outputs may be obtained/initialized like:

```
batch_size = gen_specs["user"]["batch_size"]
Output = np.zeros(batch_size, dtype=gen_specs["out"])
```

This array should be populated by whatever values are generated within the function:

```
Output["x"], persis_info = generate_next_simulation_inputs(Input["f"], persis_info)
```

Then return the array and `persis_info` to libEnsemble:

```
return Output, persis_info
```

Between the `Output` definition and the `return`, any level and complexity of computation can be performed. Users are encouraged to use the `executor` to submit applications to parallel resources if necessary, or plug in components from other libraries to serve their needs.

Note: State `gen_f` information like checkpointing should be appended to `persis_info`.

Persistent Generators

While non-persistent generators return after completing their calculation, persistent generators do the following in a loop:

1. Receive simulation results and metadata; exit if metadata instructs
2. Perform analysis
3. Send subsequent simulation parameters

Persistent generators don't need to be re-initialized on each call, but are typically more complicated. The `APOSMM` optimization generator function included with libEnsemble is persistent so it can maintain multiple local optimization subprocesses based on results from complete simulations.

Use `gen_specs["persis_in"]` to specify fields to send back to the generator throughout the run. `gen_specs["in"]` only describes the input fields when the function is **first called**.

Functions for a persistent generator to communicate directly with the manager are available in the `libensemble.tools.persistent_support` class.

Sending/receiving data is supported by the `PersistentSupport` class:

```
from libensemble.tools import PersistentSupport
from libensemble.message_numbers import STOP_TAG, PERSIS_STOP, EVAL_GEN_TAG, FINISHED_
↪PERSISTENT_GEN_TAG

my_support = PersistentSupport(libE_info, EVAL_GEN_TAG)
```

Implementing functions from the above class is relatively simple:

send

libensemble.tools.persistent_support.PersistentSupport.**send**(*self*, *output*, *calc_status=0*, *keep_state=False*)

Send message from worker to manager.

Parameters

- **output** (*ndarray[Any, dtype[_ScalarType_co]]*) – Output array to be sent to manager.
- **calc_status** (*int*) – (Optional) Provides a task status.

- **keep_state** – (Optional) If True the manager will not modify its record of the workers state (usually the manager changes the worker's state to inactive, indicating the worker is ready to receive more work, unless using active receive mode).

Return type
None

This function call typically resembles:

```
my_support.send(local_H_out[selected_IDs])
```

Note that this function has no return.

recv

libensemble.tools.persistent_support.PersistentSupport.**recv**(*self*, *blocking=True*)

Receive message to worker from manager.

Parameters
blocking (*bool*) – (Optional) If True (default), will block until a message is received.

Returns
message tag, Work dictionary, calc_in array

Return type
(<class 'int'>, <class 'dict'>, numpy.ndarray[*Any*, numpy.dtype[+_ScalarType_co]])

This function call typically resembles:

```
tag, Work, calc_in = my_support.recv()

if tag in [STOP_TAG, PERSIS_STOP]:
    cleanup()
    break
```

The logic following the function call is typically used to break the persistent generator's main loop and return.

send_recv

libensemble.tools.persistent_support.PersistentSupport.**send_recv**(*self*, *output*, *calc_status=0*)

Send message from worker to manager and receive response.

Parameters

- **output** (*ndarray[Any, dtype[_ScalarType_co]]*) – Output array to be sent to manager.
- **calc_status** (*int*) – (Optional) Provides a task status.

Returns
message tag, Work dictionary, calc_in array

Return type
(<class 'int'>, <class 'dict'>, numpy.ndarray[*Any*, numpy.dtype[+_ScalarType_co]])

This function performs both of the previous functions in a single statement. Its usage typically resembles:

```
tag, Work, calc_in = my_support.send_recv(local_H_out[selected_IDs])
if tag in [STOP_TAG, PERSIS_STOP]:
    cleanup()
    break
```

Once the persistent generator's loop has been broken because of the tag from the manager, it should return with an additional tag:

```
return local_H_out, persis_info, FINISHED_PERSISTENT_GEN_TAG
```

See `calc_status` for more information about the message tags.

Active receive mode

By default, a persistent worker is expected to receive and send data in a *ping pong* fashion. Alternatively, a worker can be initiated in *active receive* mode by the allocation function (see `start_only_persistent`). The persistent worker can then send and receive from the manager at any time.

Ensure there are no communication deadlocks in this mode. In manager–worker message exchanges, only the worker-side receive is blocking by default (a non-blocking option is available).

Cancelling Simulations

Previously submitted simulations can be cancelled by sending a message to the manager:

libensemble.tools.persistent_support.PersistentSupport.**request_cancel_sim_ids**(*self*, *sim_ids*)

Request cancellation of sim_ids.

Parameters
sim_ids (*List[int]*) – A list of sim_ids to cancel.

A message is sent to the manager to mark requested sim_ids as cancel_requested.

- If a generated point is cancelled by the generator **before sending** to another worker for simulation, then it won't be sent.
- If that point has **already been evaluated** by a simulation, the `cancel_requested` field will remain `True`.
- If that point is **currently being evaluated**, a kill signal will be sent to the corresponding worker; it must be manually processed in the simulation function.

The `Borehole Calibration tutorial` gives an example of the capability to cancel pending simulations.

Modification of existing points

To change existing fields of the History array, create a NumPy array where the `dtype` contains the `sim_id` and the fields to be modified. Send this array with `keep_state=True` to the manager. This will overwrite the manager's History array.

For example, the cancellation function `request_cancel_sim_ids` could be replicated by the following (where `sim_ids_to_cancel` is a list of integers):

```
# Send only these fields to existing H rows and libEnsemble will slot in the change.
H_o = np.zeros(len(sim_ids_to_cancel), dtype=[("sim_id", int), ("cancel_requested",␣
↪bool)])
```

(continues on next page)

(continued from previous page)

```
H_o["sim_id"] = sim_ids_to_cancel
H_o["cancel_requested"] = True
ps.send(H_o, keep_state=True)
```

Generator initiated shutdown

If using a supporting allocation function, the generator can prompt the ensemble to shutdown by simply exiting the function (e.g., on a test for a converged value). For example, the allocation function `start_only_persistent` closes down the ensemble as soon as a persistent generator returns. The usual return values should be given.

Examples

Examples of non-persistent and persistent generator functions can be found here.

2.6.2 Simulator Functions

Simulator and `Generator functions` have relatively similar interfaces.

```
def my_simulation(Input, persis_info, sim_specs, libE_info):
    batch_size = sim_specs["user"]["batch_size"]

    Output = np.zeros(batch_size, sim_specs["out"])
    ...
    Output["f"], persis_info = do_a_simulation(Input["x"], persis_info)

    return Output, persis_info
```

Most `sim_f` function definitions written by users resemble:

```
def my_simulation(Input, persis_info, sim_specs, libE_info):
```

where:

- `Input` is a selection of the `History array`
- `persis_info` is a dictionary containing state information
- `sim_specs` is a dictionary of simulation parameters, including which fields from the History array got sent
- `libE_info` is a dictionary containing libEnsemble-specific entries

Valid simulator functions can accept a subset of the above parameters. So a very simple simulator function can start:

```
def my_simulation(Input):
```

If sim_specs was initially defined:

```
sim_specs = {
    "sim_f": some_function,
    "in": ["x"],
    "out:" ["f", float, (1,)],
    "user": {
```

(continues on next page)

(continued from previous page)

```
        "batch_size": 128
    }
}
```

Then user parameters and a *local* array of outputs may be obtained/initialized like:

```
batch_size = sim_specs["user"]["batch_size"]
Output = np.zeros(batch_size, dtype=sim_specs["out"])
```

This array should be populated with output values from the simulation:

```
Output["f"], persis_info = do_a_simulation(Input["x"], persis_info)
```

Then return the array and `persis_info` to libEnsemble:

```
return Output, persis_info
```

Between the `Output` definition and the `return`, any level and complexity of computation can be performed. Users are encouraged to use the `executor` to submit applications to parallel resources if necessary, or plug in components from other libraries to serve their needs.

Executor

libEnsemble's Executors are commonly used within simulator functions to launch and monitor applications. An excellent overview is already available `here`.

See the `Executor with Electrostatic Forces tutorial` for an additional example to try out.

Persistent Simulators

Although comparatively uncommon, simulator functions can also be written in a persistent fashion. See the `here` for a general API overview of writing persistent generators, since the interface is largely identical. The only differences are to pass `EVAL_SIM_TAG` when instantiating a `PersistentSupport` class instance and to return `FINISHED_PERSISTENT_SIM_TAG` when the simulator function returns.

Note: An example routine using a persistent simulator can be found in test_persistent_sim_uniform_sampling.

2.6.3 Allocation Functions

Although the included allocation functions are sufficient for most users, those who want to fine-tune how data or resources are allocated to their generator or simulator can write their own.

The `alloc_f` is unique since it is called by libEnsemble's manager instead of a worker.

For allocation functions, as with the other user functions, the level of complexity can vary widely. We encourage experimenting with:

1. Prioritization of simulations
2. Sending results immediately or in batch
3. Assigning varying resources to evaluations

Example

Listing 6: libensemble.alloc_funcs.fast_alloc.give_sim_work_first

```
from libensemble.tools.alloc_support import AllocSupport, InsufficientFreeResources


def give_sim_work_first(W, H, sim_specs, gen_specs, alloc_specs, persis_info, libE_info):
    """
    This allocation function gives (in order) entries in ``H`` to idle workers
    to evaluate in the simulation function. The fields in ``sim_specs["in"]``
    are given. If all entries in `H` have been given a be evaluated, a worker
    is told to call the generator function, provided this wouldn't result in
    more than ``alloc_specs["user"]["num_active_gen"]`` active generators.

    This fast_alloc variation of give_sim_work_first is useful for cases that
    simply iterate through H, issuing evaluations in order and, in particular,
    is likely to be faster if there will be many short simulation evaluations,
    given that this function contains fewer column length operations.

    tags: alloc, simple, fast

    .. seealso::
        `test_fast_alloc.py <https://github.com/Libensemble/libensemble/blob/develop/
↪libensemble/tests/functionality_tests/test_fast_alloc.py>`_ # noqa
    """

    if libE_info["sim_max_given"] or not libE_info["any_idle_workers"]:
        return {}, persis_info

    user = alloc_specs.get("user", {})
    manage_resources = libE_info["use_resource_sets"]

    support = AllocSupport(W, manage_resources, persis_info, libE_info)

    gen_count = support.count_gens()
    Work = {}
    gen_in = gen_specs.get("in", [])

    for wid in support.avail_worker_ids():
        # Skip any cancelled points
        while persis_info["next_to_give"] < len(H) and H[persis_info["next_to_give"]][
↪"cancel_requested"]:
            persis_info["next_to_give"] += 1

        # Give sim work if possible
        if persis_info["next_to_give"] < len(H):
            try:
                Work[wid] = support.sim_work(wid, H, sim_specs["in"], [persis_info["next_
↪to_give"]], [])
            except InsufficientFreeResources:
                break
            persis_info["next_to_give"] += 1
```

(continues on next page)

(continued from previous page)

```
        elif gen_count < user.get("num_active_gens", gen_count + 1):
            # Give gen work
            return_rows = range(len(H)) if gen_in else []
            try:
                Work[wid] = support.gen_work(wid, gen_in, return_rows, persis_info.
↪get(wid))
            except InsufficientFreeResources:
                break
            gen_count += 1
            persis_info["total_gen_calls"] += 1

    return Work, persis_info
```

Most `alloc_f` function definitions written by users resemble:

```
def my_allocator(W, H, sim_specs, gen_specs, alloc_specs, persis_info, libE_info):
```

where:

- `W` is an array containing worker state info
- `H` is the *trimmed* History array, containing rows from the generator
- `libE_info` is a set of statistics to determine the progress of work or exit conditions

Most users first check that it is appropriate to allocate work:

```
if libE_info["sim_max_given"] or not libE_info["any_idle_workers"]:
    return {}, persis_info
```

If the allocation is to continue, a support class is instantiated and a `Work dictionary` is initialized:

```
manage_resources = "resource_sets" in H.dtype.names or libE_info["use_resource_sets"]
support = AllocSupport(W, manage_resources, persis_info, libE_info)
Work = {}
```

This Work dictionary is populated with integer keys `wid` for each worker and dictionary values to give to those workers:

Example `Work`

```
{
    1: {
        "H_fields": ["x"],
        "persis_info": {"rand_stream": RandomState(...) at ..., "worker_num": 1},
        "tag": 1,
        "libE_info": {"H_rows": array([368])}
    },

    2: {
        "H_fields": ["x"],
        "persis_info": {"rand_stream": RandomState(...) at ..., "worker_num": 2},
        "tag": 1,
        "libE_info": {"H_rows": array([369])}
```

(continues on next page)

(continued from previous page)

```
    },

    3: {
        "H_fields": ["x"],
        "persis_info": {"rand_stream": RandomState(...) at ..., "worker_num": 3},
        "tag": 1,
        "libE_info": {"H_rows": array([370])}
    },
    ...

}
```

This Work dictionary instructs each worker to call the `sim_f` (`tag: 1`) with data from `"x"` and a given `"H_row"` from the History array. A worker-specific `persis_info` is also given.

Constructing these arrays and determining which workers are available for receiving data is simplified by the `AllocSupport` class available within the `libensemble.tools.alloc_support` module:

AllocSupport

class libensemble.tools.alloc_support.**AllocSupport**(*W*, *manage_resources=False*, *persis_info={}*, *libE_info={}*, *user_resources=None*, *user_scheduler=None*)

A helper class to assist with writing allocation functions.

This class contains methods for common operations like populating work units, determining which workers are available, evaluating what values need to be distributed to workers, and others.

Note that since the `alloc_f` is called periodically by the Manager, this class instance (if used) will be recreated/destroyed on each loop.

__init__(*W*, *manage_resources=False*, *persis_info={}*, *libE_info={}*, *user_resources=None*, *user_scheduler=None*)

Instantiate a new AllocSupport instance

`W` is passed in for convenience on init; it is referenced by the various methods, but never modified.

By default, an `AllocSupport` instance uses any initiated libEnsemble resource module and the built-in libEnsemble scheduler.

Parameters

- **W** – A `Worker array`
- **manage_resources** – (Optional) Boolean for if to assign resource sets when creating work units.
- **persis_info** – (Optional) A `dictionary of persistent information.`.
- **scheduler_opts** – (Optional) A dictionary of options to pass to the resource scheduler.
- **user_resources** – (Optional) A user supplied `resources` object.
- **user_scheduler** – (Optional) A user supplied `user_scheduler` object.

assign_resources(*rsets_req*, *use_gpus=None*, *user_params=[]*)

Schedule resource sets to a work record if possible.

For default scheduler, if more than one group (node) is required, will try to find even split, otherwise allocates whole nodes.

Raises `InsufficientFreeResources` if the required resources are not currently available, or `InsufficientResourcesError` if the required resources do not exist.

Parameters

- **rsets_req** – Int. Number of resource sets to request.
- **use_gpus** – Bool. Whether to use GPU resource sets.
- **user_params** – List of Integers. User parameters num_procs, num_gpus.

Returns
List of Integers. Resource set indices assigned.

avail_worker_ids(*persistent=None*, *active_recv=False*, *zero_resource_workers=None*)

Returns available workers as a list of IDs, filtered by the given options.

Parameters

- **persistent** – (Optional) Int. Only return workers with given `persis_state` (1=sim, 2=gen).
- **active_recv** – (Optional) Boolean. Only return workers with given active_recv state.
- **zero_resource_workers** – (Optional) Boolean. Only return workers that require no resources.

Returns
List of worker IDs.

If there are no zero resource workers defined, then the `zero_resource_workers` argument will be ignored.

count_gens()

Returns the number of active generators.

test_any_gen()

Returns `True` if a generator worker is active.

count_persis_gens()

Return the number of active persistent generators.

sim_work(*wid*, *H*, *H_fields*, *H_rows*, *persis_info*, ***libE_info*)

Add sim work record to given `Work` dictionary.

Includes evaluation of required resources if the worker is not in a persistent state.

Parameters

- **wid** – Int. Worker ID.
- **H** – `History array`. For parsing out requested resource sets.
- **H_fields** – Which fields from `H` to send.
- **H_rows** – Which rows of `H` to send.
- **persis_info** – Worker specific `persis_info` dictionary.

Returns
a Work entry.

Additional passed parameters are inserted into `libE_info` in the resulting work record.

If `rset_team` is passed as an additional parameter, it will be honored, assuming that any resource checking has already been done.

gen_work(*wid*, *H_fields*, *H_rows*, *persis_info*, ***libE_info*)

Add gen work record to given `Work` dictionary.

Includes evaluation of required resources if the worker is not in a persistent state.

Parameters

- **Work** – `Work dictionary`.
- **wid** – Worker ID.
- **H_fields** – Which fields from `H` to send.
- **H_rows** – Which rows of `H` to send.
- **persis_info** – Worker specific `persis_info` dictionary.

Returns
A Work entry.

Additional passed parameters are inserted into `libE_info` in the resulting work record.

If `rset_team` is passed as an additional parameter, it will be honored, and assume that any resource checking has already been done. For example, passing `rset_team=[]`, would ensure that no resources are assigned.

all_sim_started(*H*, *pt_filter=None*, *low_bound=None*)

Returns `True` if all expected points have started their sim.

Excludes cancelled points.

Parameters

- **pt_filter** – (Optional) Boolean array filtering expected returned points in `H`.
- **low_bound** – (Optional) Lower bound for testing all returned.

Returns
True if all expected points have started their sim.

all_sim_ended(*H*, *pt_filter=None*, *low_bound=None*)

Returns `True` if all expected points have had their sim_end.

Excludes cancelled points that were not already sim_started.

Parameters

- **pt_filter** – (Optional) Boolean array filtering expected returned points in `H`.
- **low_bound** – (Optional) Lower bound for testing all returned.

Returns
True if all expected points have had their sim_end.

all_gen_informed(*H*, *pt_filter=None*, *low_bound=None*)

Returns `True` if gen has been informed of all expected points.

Excludes cancelled points that were not already given out.

Parameters

- **pt_filter** – (Optional) Boolean array filtering expected sim_end points in H.
- **low_bound** – (Optional) Lower bound for testing all returned.

Returns

True if gen have been informed of all expected points.

points_by_priority(*H*, *points_avail*, *batch=False*)

Returns indices of points to give by priority.

Parameters

- **points_avail** – Indices of points that are available to give.
- **batch** – (Optional) Boolean. Should batches of points with the same priority be given simultaneously.

Returns

An array of point indices to give.

The Work dictionary is returned to the manager alongside `persis_info`. If `1` is returned as the third value, this instructs the ensemble to stop.

Note: An error occurs when the `alloc_f` returns nothing while all workers are idle

Information from the manager describing the progress of the current libEnsemble routine can be found in `libE_info`:

```
libE_info =  {"exit_criteria": dict,              # Criteria for ending routine
              "elapsed_time": float,              # Time elapsed since start of routine
              "manager_kill_canceled_sims": bool,  # True if manager is to send kills to␣
↪cancelled simulations
              "sim_started_count": int,           # Total number of points given for␣
↪simulation function evaluation
              "sim_ended_count": int,             # Total number of points returned␣
↪from simulation function evaluations
              "gen_informed_count": int,          # Total number of evaluated points␣
↪given back to a generator function
              "sim_max_given": bool,              # True if `sim_max` simulations have␣
↪been given out to workers
              "use_resource_sets": bool}          # True if num_resource_sets has been␣
↪explicitly set.
```

Most often, the allocation function will just return once `sim_max_given` is `True`, but the user could choose to do something different, such as cancel points or keep returning completed points to the generator.

Generators that construct models based on *all evaluated points*, for example, may need simulation work units at the end of an ensemble to be returned to the generator anyway.

Alternatively, users can use `elapsed_time` to track runtime inside their allocation function and detect impending timeouts, then pack up cleanup work requests, or mark points for cancellation.

The remaining values above are useful for efficient filtering of H values (e.g., `sim_ended_count` saves filtering by an entire column of H.)

Descriptions of included allocation functions can be found here. The default allocation function is `give_sim_work_first`. During its worker ID loop, it checks if there's unallocated work and assigns simulations for that work. Otherwise, it initializes generators for up to `"num_active_gens"` instances. Other settings like `batch_mode` are also supported. See here for more information about `give_sim_work_first`.

2.6.4 User Function API

libEnsemble requires functions for generation, simulation, and allocation.

While libEnsemble provides a default allocation function, the simulator and generator functions must be specified. The required API and example arguments are given here. `Example sim and gen functions` are provided in the libEnsemble package.

`See here for more in-depth guides to writing user functions`

As of v0.10.0, valid simulator and generator functions can *accept and return a smaller subset of the listed parameters and return values*. For instance, a `def my_simulation(one_Input) -> one_Output` function is now accepted, as is `def my_generator(Input, persis_info) -> Output, persis_info`.

sim_f API

The simulator function will be called by libEnsemble's workers with *up to* the following arguments and returns:

```
Out, persis_info, calc_status = sim_f(H[sim_specs["in"]][sim_ids_from_allocf], persis_
↪info, sim_specs, libE_info)
```

Parameters:

H: `numpy structured array` `(example)`

persis_info: `dict` `(example)`

sim_specs: `dict` `(example)`

libE_info: `dict` `(example)`

Returns:

H: `numpy structured array` with keys/value-sizes matching those in sim_specs["out"] `(example)`

persis_info: `dict` `(example)`

calc_status: `int`, optional Provides a task status to the manager and the libE_stats.txt file `(example)`

gen_f API

The generator function will be called by libEnsemble's workers with *up to* the following arguments and returns:

```
Out, persis_info, calc_status = gen_f(H[gen_specs["in"]][sim_ids_from_allocf], persis_
↪info, gen_specs, libE_info)
```

Parameters:

H: numpy structured array (example)

persis_info: dict (example)

gen_specs: dict (example)

libE_info: dict (example)

Returns:

H: numpy structured array with keys/value-sizes matching those in gen_specs["out"] (example)

persis_info: dict (example)

calc_status: int, optional Provides a task status to the manager and the libE_stats.txt file (example)

alloc_f API

The allocation function will be called by libEnsemble's manager with the following API:

```
Work, persis_info, stop_flag = alloc_f(W, H, sim_specs, gen_specs, alloc_specs, persis_
↪info, libE_info)
```

Parameters:

W: numpy structured array (example)

H: numpy structured array (example)

sim_specs: dict (example)

gen_specs: dict (example)

alloc_specs: dict (example)

persis_info: dict (example)

libE_info: dict Various statistics useful to the allocation function for determining how much work has been evaluated, or if the routine should prepare to complete. See the allocation function guide for more information.

Returns:

Work: dict Dictionary with integer keys i for work to be sent to worker i. (example)

persis_info: dict (example)

stop_flag: int, optional Set to 1 to request libEnsemble manager to stop giving additional work after receiving existing work

2.6.5 calc_status

```
Worker     1: Calc     0: gen Time: 0.00 Start: 2019-11-19 18:53:43 End: 2019-11-19␣
↪18:53:43 Status: Not set
Worker     1: Calc     1: sim Time: 4.41 Start: 2019-11-19 18:53:43 End: 2019-11-19␣
↪18:53:48 Status: Worker killed
Worker     2: Calc     0: sim Time: 5.42 Start: 2019-11-19 18:53:43 End: 2019-11-19␣
↪18:53:49 Status: Completed
Worker     1: Calc     2: sim Time: 2.41 Start: 2019-11-19 18:53:48 End: 2019-11-19␣
↪18:53:50 Status: Worker killed
Worker     2: Calc     1: sim Time: 2.41 Start: 2019-11-19 18:53:49 End: 2019-11-19␣
↪18:53:51 Status: Worker killed
Worker     1: Calc     3: sim Time: 4.41 Start: 2019-11-19 18:53:50 End: 2019-11-19␣
↪18:53:55 Status: Completed
Worker     2: Calc     2: sim Time: 4.41 Start: 2019-11-19 18:53:51 End: 2019-11-19␣
↪18:53:56 Status: Completed
```

`calc_status` is similar to an exit code, and is either an integer attribute with a corresponding description, or a user-specified string. They are the third optional return value from a user function, and are printed to `libE_stats.txt`.

Built-in codes are available in the `libensemble.message_numbers` module, but users are also free to return any custom string.

calc_status with Executor

```
from libensemble.message_numbers import WORKER_DONE, WORKER_KILL, TASK_FAILED

task = exctr.submit(calc_type="sim", num_procs=cores, wait_on_start=True)
calc_status = UNSET_TAG
poll_interval = 1  # secs
while not task.finished:
    if task.runtime > time_limit:
        task.kill()  # Timeout
    else:
        time.sleep(poll_interval)
        task.poll()

if task.finished:
    if task.state == "FINISHED":
        print("Task {} completed".format(task.name))
        calc_status = WORKER_DONE
    elif task.state == "FAILED":
        print("Warning: Task {} failed: Error code {}".format(task.name, task.errcode))
        calc_status = TASK_FAILED
    elif task.state == "USER_KILLED":
        print("Warning: Task {} has been killed".format(task.name))
        calc_status = WORKER_KILL
    else:
        print("Warning: Task {} in unknown state {}. Error code {}".format(task.name,␣
↪task.state, task.errcode))

outspecs = sim_specs["out"]
```

(continues on next page)

(continued from previous page)

```
output = np.zeros(1, dtype=outspecs)
output["energy"][0] = final_energy

return output, persis_info, calc_status
```

Custom calc_status

```
from libensemble.message_numbers import WORKER_DONE, TASK_FAILED

task = exctr.submit(calc_type="sim", num_procs=cores, wait_on_start=True)

task.wait(timeout=60)

file_output = read_task_output(task)
if task.errcode == 0:
    if "fail" in file_output:
        calc_status = "Task failed successfully?"
    else:
        calc_status = WORKER_DONE
else:
    calc_status = TASK_FAILED

outspecs = sim_specs["out"]
output = np.zeros(1, dtype=outspecs)
output["energy"][0] = final_energy

return output, persis_info, calc_status
```

Available values

```
FINISHED_PERSISTENT_SIM_TAG = 11  # tells manager sim_f done persistent mode
FINISHED_PERSISTENT_GEN_TAG = 12  # tells manager gen_f done persistent mode
MAN_SIGNAL_FINISH = 20  # Kill tasks and shutdown worker
MAN_SIGNAL_KILL = 21  # Kill running task - but don't stop worker
WORKER_KILL = 30  # Worker kills not covered by a more specific case
WORKER_KILL_ON_ERR = 31  # Worker killed due to an error in results
WORKER_KILL_ON_TIMEOUT = 32  # Worker killed on timeout
TASK_FAILED = 33  # Calc had tasks that failed
WORKER_DONE = 34  # Calculation was successful
```

Corresponding messages

```
calc_status_strings = {
    UNSET_TAG: "Not set",
    FINISHED_PERSISTENT_SIM_TAG: "Persis sim finished",
    FINISHED_PERSISTENT_GEN_TAG: "Persis gen finished",
    MAN_SIGNAL_FINISH: "Manager killed on finish",
    MAN_SIGNAL_KILL: "Manager killed task",
    WORKER_KILL_ON_ERR: "Worker killed task on Error",
    WORKER_KILL_ON_TIMEOUT: "Worker killed task on Timeout",
    WORKER_KILL: "Worker killed",
    TASK_FAILED: "Task Failed",
    WORKER_DONE: "Completed",
    CALC_EXCEPTION: "Exception occurred",
    None: "Unknown Status",
}
```

2.6.6 Work Dictionary

The work dictionary contains metadata that is used by the manager to send a packet of work to a worker. The dictionary uses integer keys `i` and values that determine the data given to worker `i`. Populated in the allocation function. `Work[i]` has the following form:

```
Work[i]: [dict]:

    Required keys:
    "H_fields" [list]: The field names of the history H to be sent to worker i
    "persis_info" [dict]: Any persistent info to be sent to worker i
    "tag" [int]: "EVAL_SIM_TAG"/"EVAL_GEN_TAG" if worker i is to call sim/gen_func
    "libE_info" [dict]: Info sent to/from worker to help manager update the H array

    libE_info contains the following:
    "H_rows" [list of ints]: History rows to send to worker i
    "rset_team" [list of ints]: The resource sets to be assigned (if dynamic scheduling␣
↪is used)
    "persistent" [bool]: True if worker i will enter persistent mode (Default: False)
```

The work dictionary is typically set using the `gen_work` or `sim_work` `helper functions` in the allocation function. `H_fields`, for example, is usually packed from either `sim_specs["in"]`, `gen_specs["in"]` or the equivalent "persis_in" variants.

See also:

For allocation functions giving work dictionaries using persistent workers, see start_only_persistent.py or start_persistent_local_opt_gens.py. For a use case where the allocation and generator functions combine to do simulation evaluations with different resources, see test_uniform_sampling_with_variable_resources.py.

2.6.7 Worker Array

The worker array `W` contains information about each worker's state. Used within allocation functions to determine which workers are eligible to receive work.

```
W: numpy structured array
    "worker_id" [int]:
        The worker ID
    "active" [int]:
        Is the worker active or not
    "persis_state" [int]:
        Is the worker in a persis_state
    "active_recv" [int]:
        Is the worker in an active receive state
    "blocked" [int]:
        Is the worker's resources blocked by another calculation
```

We use the following convention:

Worker state	active	persis_state	blocked
idle worker	0	0	0
active, nonpersistent sim	1	0	0
active, nonpersistent gen	2	0	0
active, persistent sim	1	1	0
active, persistent gen	2	2	0
waiting, persistent sim	0	1	0
waiting, persistent gen	0	2	0
worker blocked by some other calculation	1	0	1

Note:

- libEnsemble's manager receives only from workers with a nonzero `"active"` state
- libEnsemble's manager calls the `alloc_f` only if some worker has an `"active"` state of zero, or is in an *active receive* state.

See also:

For an example allocation function that queries the worker array, see persistent_aposmm_alloc.

2.7 Executors

libEnsemble's Executors can be used within user functions to provide a simple, portable interface for running and managing user applications.

2.7.1 Executor Overview

Most computationally expensive libEnsemble workflows involve launching applications from a `sim_f` or `gen_f` running on a worker to the compute nodes of a supercomputer, cluster, or other compute resource.

The **Executor** provides a portable interface for running applications on any system.

Detailed description

An **Executor** interface is provided by libEnsemble to remove the burden of system interaction from the user and improve workflow portability. Users first register their applications to Executor instances, which then return corresponding `Task` objects upon submission within user functions.

Task attributes and retrieval functions can be queried to determine the status of running application instances. Functions are also provided to access and interrogate files in the task's working directory.

libEnsemble's Executors and Tasks contain many familiar features and methods to Python's native concurrent futures interface. Executors feature the `submit()` function for launching apps (detailed below), but currently do not support `map()` or `shutdown()`. Tasks are much like `futures`. They feature the `cancel()`, `cancelled()`, `running()`, `done()`, `result()`, and `exception()` functions from the standard.

The main `Executor` class can subprocess serial applications in place, while the `MPIExecutor` is used for running MPI applications, and the `BalsamExecutor` for submitting MPI run requests from a worker running on a compute node to the Balsam service. This second approach is suitable for systems that don't allow submitting MPI applications from compute nodes.

Typically, users choose and parameterize their `Executor` objects in their calling scripts, where each executable generator or simulation application is registered to it. If an alternative Executor like Balsam is used, then the applications can be registered as in the example below. Once in the user-side worker code (sim/gen func), the Executor can be retrieved without any need to specify the type.

Once the Executor is retrieved, tasks can be submitted by specifying the `app_name` from registration in the calling script alongside other optional parameters described in the API.

Basic usage

In calling script

To set up an MPI executor, register an MPI application, and add to the ensemble object.

```
from libensemble import Ensemble
from libensemble.executors import MPIExecutor

exctr = MPIExecutor()
exctr.register_app(full_path="/path/to/my/exe", app_name="sim1")
ensemble = Ensemble(executor=exctr)
```

If using the `libE()` call, the Executor in the calling script does **not** have to be passed to the `libE()` function. It is transferred via the `Executor.executor` class variable.

In user simulation function:

```
def sim_func(H, persis_info, sim_specs, libE_info):

    input_param = str(int(H["x"][0][0]))
    exctr = libE_info["executor"]

    task = exctr.submit(
        app_name="sim1",
        num_procs=8,
        app_args=input_param,
        stdout="out.txt",
        stderr="err.txt",
    )

    # Wait for task to complete
    task.wait()
```

Example use-cases:

- `Electrostatic Forces example`: Launches the `forces.x` MPI application.
- `Forces example with GPUs`: Auto-assigns GPUs via executor.

See the `Executor` or `MPIExecutor` interface for the complete API.

See `Running on HPC Systems` for illustrations of how common options such as `libE_specs["dedicated_mode"]` affect the run configuration on clusters and supercomputers.

Advanced Features

Example of polling output and killing application:

In simulation function (sim_f).

```
import time


def sim_func(H, persis_info, sim_specs, libE_info):
    input_param = str(int(H["x"][0][0]))
    exctr = libE_info["executor"]

    task = exctr.submit(
        app_name="sim1",
        num_procs=8,
        app_args=input_param,
        stdout="out.txt",
        stderr="err.txt",
    )

    timeout_sec = 600
    poll_delay_sec = 1

    while not task.finished:
        # Has manager sent a finish signal
```

(continues on next page)

(continued from previous page)

```
        if exctr.manager_kill_received():
            task.kill()
            my_cleanup()

        # Check output file for error and kill task
        elif task.stdout_exists():
            if "Error" in task.read_stdout():
                task.kill()

        elif task.runtime > timeout_sec:
            task.kill()  # Timeout

        else:
            time.sleep(poll_delay_sec)
            task.poll()

    print(task.state)  # state may be finished/failed/killed
```

Users who wish to poll only for manager kill signals and timeouts don't necessarily need to construct a polling loop like above, but can instead use the `Executor` built-in `polling_loop()` method. An alternative to the above simulation function may resemble:

```
def sim_func(H, persis_info, sim_specs, libE_info):
    input_param = str(int(H["x"][0][0]))
    exctr = libE_info["executor"]

    task = exctr.submit(
        app_name="sim1",
        num_procs=8,
        app_args=input_param,
        stdout="out.txt",
        stderr="err.txt",
    )

    timeout_sec = 600
    poll_delay_sec = 1

    exctr.polling_loop(task, timeout=timeout_sec, delay=poll_delay_sec)

    print(task.state)  # state may be finished/failed/killed
```

Note: Applications or tasks submitted via the Balsam Executor are referred to as **"jobs"** within Balsam, including within Balsam's database and when describing the state of a completed submission.

The `MPIExecutor` autodetects system criteria such as the appropriate MPI launcher and mechanisms to poll and kill tasks. It also has access to the resource manager, which partitions resources amongst workers, ensuring that runs utilize different resources (e.g., nodes). Furthermore, the `MPIExecutor` offers resilience via the feature of re-launching tasks that fail to start because of system factors.

Various back-end mechanisms may be used by the Executor to best interact with each system, including proxy launchers or task management systems such as Balsam. Currently, these Executors launch at the application level within an existing resource pool. However, submissions to a batch scheduler may be supported in future Executors.

2.7.2 Base Executor - Local apps

This module contains the classes `Executor` and `Task`. An `executor` can create and manage multiple `tasks`. `Task` attributes are queried to determine status.

See the Executor APIs for optional arguments.

Base Executor

Only for running local serial-launched applications. To run MPI applications and use detected resources, use the `MPIExecutor`

class libensemble.executors.executor.**Executor**

The executor can create, poll and kill runnable tasks

Class Attributes:

Variables
Executor – executor: The executor object is stored here and can be retrieved in user functions.

__init__()

Instantiate a new Executor instance.

Returns
A new Executor object is created. This is typically created in the user calling script.

Return type
Executor

register_app(*full_path*, *app_name=None*, *calc_type=None*, *desc=None*, *precedent=''*)

Registers a user application to libEnsemble.

The `full_path` of the application must be supplied. Either `app_name` or `calc_type` can be used to identify the application in user scripts (in the **submit** function). `app_name` is recommended.

Parameters

- **full_path** (*str*) – The full path of the user application to be registered
- **app_name** (*str, Optional*) – Name to identify this application.
- **calc_type** (*str, Optional*) – Calculation type: Set this application as the default 'sim' or 'gen' function.
- **desc** (*str, Optional*) – Description of this application
- **precedent** (*str, Optional*) – Any str that should directly precede the application full path.

Return type
None

manager_poll()

Polls for a manager signal

The executor manager_signal attribute will be updated.

Return type
int

manager_kill_received()

Return True if received kill signal from the manager

Return type
bool

polling_loop(*task*, *timeout=None*, *delay=0.1*, *poll_manager=False*)

Optional, blocking, generic task status polling loop. Operates until the task finishes, times out, or is Optionally killed via a manager signal. On completion, returns a presumptive `calc_status` integer. Potentially useful for running an application via the Executor until it stops without monitoring its intermediate output.

Parameters

- **task** (*object*) – a Task object returned by the executor on submission
- **timeout** (*int, Optional*) – Maximum number of seconds for the polling loop to run. Tasks that run longer than this limit are killed. Default: No timeout
- **delay** (*int, Optional*) – Sleep duration between polling loop iterations. Default: 0.1 seconds
- **poll_manager** (*bool, Optional*) – Whether to also poll the manager for 'finish' or 'kill' signals. If detected, the task is killed. Default: False.

Returns
calc_status – presumptive integer attribute describing the final status of a launched task

Return type
int

submit(*calc_type=None*, *app_name=None*, *app_args=None*, *stdout=None*, *stderr=None*, *dry_run=False*, *wait_on_start=False*, *env_script=None*)

Create a new task and run as a local serial subprocess.

The created `task` object is returned.

Parameters

- **calc_type** (*str, Optional*) – The calculation type: 'sim' or 'gen' Only used if app_name is not supplied. Uses default sim or gen application.
- **app_name** (*str, Optional*) – The application name. Must be supplied if calc_type is not.
- **app_args** (*str, Optional*) – A str of the application arguments to be added to task submit command line
- **stdout** (*str, Optional*) – A standard output filename
- **stderr** (*str, Optional*) – A standard error filename
- **dry_run** (*bool, Optional*) – Whether this is a dry_run - no task will be launched; instead runline is printed to logger (at INFO level)
- **wait_on_start** (*bool, Optional*) – Whether to wait for task to be polled as RUNNING (or other active/end state) before continuing
- **env_script** (*str, Optional*) – The full path of a shell script to set up the environment for the launched task. This will be run in the subprocess, and not affect the worker environment. The script should start with a shebang.

Returns
task – The launched task object

Return type
Task

Task

Tasks are created and returned by the Executor's `submit()`. Tasks can be polled, killed, and waited on with the respective `poll`, `kill`, and `wait` functions. Task information can be queried through instance attributes and query functions.

class libensemble.executors.executor.**Task**(*app=None*, *app_args=None*, *workdir=None*, *stdout=None*, *stderr=None*, *workerid=None*, *dry_run=False*)

Manages the creation, configuration and status of a launchable task

workdir_exists()

Returns true if the task's workdir exists

Return type
bool | None

file_exists_in_workdir(*filename*)

Returns true if the named file exists in the task's workdir

Parameters
filename (*str*) –

Return type
bool

read_file_in_workdir(*filename*)

Opens and reads the named file in the task's workdir

Parameters
filename (*str*) –

Return type
str

stdout_exists()

Returns true if the task's stdout file exists in the workdir

Return type
bool

read_stdout()

Opens and reads the task's stdout file in the task's workdir

Return type
str

stderr_exists()

Returns true if the task's stderr file exists in the workdir

Return type
bool

`read_stderr()`

Opens and reads the task's stderr file in the task's workdir

Return type
str

`poll()`

Polls and updates the status attributes of the task

Return type
None

`wait(timeout=None)`

Waits on completion of the task or raises TimeoutExpired exception

Status attributes of task are updated on completion.

Parameters
timeout (*int or float, Optional*) – Time in seconds after which a TimeoutExpired exception is raised. If not set, then simply waits until completion. Note that the task is not automatically killed on timeout.

Return type
None

`result(timeout=None)`

Wrapper for task.wait() that also returns the task's status on completion.

Parameters
timeout (*int or float, Optional*) – Time in seconds after which a TimeoutExpired exception is raised. If not set, then simply waits until completion. Note that the task is not automatically killed on timeout.

Return type
str

`exception(timeout=None)`

Wrapper for task.wait() that instead returns the task's error code on completion.

Parameters
timeout (*int or float, Optional*) – Time in seconds after which a TimeoutExpired exception is raised. If not set, then simply waits until completion. Note that the task is not automatically killed on timeout.

`running()`

Return `True` if task is currently running.

Return type
bool

`done()`

Return `True` if task is finished.

Return type
bool

`kill(wait_time=60)`

Kills or cancels the supplied task

Parameters
wait_time (*int, Optional*) – Time in seconds to wait for termination between sending SIGTERM and a SIGKILL signals.

Return type
None

Sends SIGTERM, waits for a period of <wait_time> for graceful termination, then sends a hard kill with SIGKILL. If <wait_time> is 0, we go immediately to SIGKILL; if <wait_time> is none, we never do a SIGKILL.

`cancel()`

Wrapper for task.kill() without waiting

Return type
None

`cancelled()`

Return `True if task successfully cancelled.

Return type
bool

Task Attributes

Note: These should not be set directly. Tasks are launched by the Executor, and task information can be queried through the task attributes below and the query functions.

task.state
(string) The task status. One of (“UNKNOWN”|”CREATED”|”WAITING”|”RUNNING”|”FINISHED”|”USER_KILLED”|”

task.process
(process obj) The process object used by the underlying process manager (e.g., return value of subprocess.Popen).

task.errcode
(int) The error code (or return code) used by the underlying process manager.

task.finished
(boolean) True means task has finished running - not whether it was successful.

task.success
(boolean) Did task complete successfully (e.g., the return code is zero)?

task.runtime
(int) Time in seconds that task has been running.

task.submit_time
(int) Time since epoch that task was submitted.

task.total_time
(int) Total time from task submission to completion (only available when task is finished).

Run configuration attributes - some will be autogenerated:

task.workdir
(string) Work directory for the task

task.name
(string) Name of task - autogenerated

task.app
(app obj) Use application/executable, registered using exctr.register_app

task.app_args
(string) Application arguments as a string

task.stdout
(string) Name of file where the standard output of the task is written (in task.workdir)

task.stderr
(string) Name of file where the standard error of the task is written (in task.workdir)

task.dry_run
(boolean) True if task corresponds to dry run (no actual submission)

task.runline
(string) Complete, parameterized command to be subprocessed to launch app

2.7.3 MPI Executor - MPI apps

This module launches and controls the running of MPI applications.

In order to create an MPI executor, the calling script should contain:

```
exctr = MPIExecutor()
```

The MPIExecutor will use system resource information supplied by the libEsnemble resource manager when submitting tasks.

See this example for usage.

class libensemble.executors.mpi_executor.**MPIExecutor**(*custom_info={}*)

Bases: Executor

The MPI executor can create, poll and kill runnable MPI tasks

Parameters
custom_info (*dict, Optional*) – Provide custom overrides to selected variables that are usually auto-detected. See below.

custom_info usage

The MPIExecutor automatically detects MPI runners and launch mechanisms. However it is possible to override the detected information using the `custom_info` argument. This takes a dictionary of values.

The allowable fields are:

```
'mpi_runner' [string]:
    Select runner: 'mpich', 'openmpi', 'aprun', 'srun', 'jsrun', 'custom'
    All except 'custom' relate to runner classes in libEnsemble.
    Custom allows user to define their own run-lines but without parsing
    arguments or making use of auto-resources.
'runner_name' [string]:
    Runner name: Replaces run command if present. All runners have a default
    except for 'custom'.
'subgroup_launch' [bool]:
    Whether MPI runs should be initiated in a new process group. This needs
```

(continues on next page)

(continued from previous page)

```
    to be correct for kills to work correctly. Use the standalone test at
    libensemble/tests/standalone_tests/kill_test to determine correct value
    for a system.
```

For example:

```
customizer = {'mpi_runner': 'mpich',
              'runner_name': 'wrapper -x mpich'}

from libensemble.executors.mpi_executor import MPIExecutor
exctr = MPIExecutor(custom_info=customizer)
```

submit(*calc_type=None*, *app_name=None*, *num_procs=None*, *num_nodes=None*, *procs_per_node=None*, *num_gpus=None*, *machinefile=None*, *app_args=None*, *stdout=None*, *stderr=None*, *stage_inout=None*, *hyperthreads=False*, *dry_run=False*, *wait_on_start=False*, *extra_args=None*, *auto_assign_gpus=False*, *match_procs_to_gpus=False*, *env_script=None*, *mpi_runner_type=None*)

Creates a new task, and either executes or schedules execution.

The created `task` object is returned.

The user must supply either the app_name or calc_type arguments (app_name is recommended). All other arguments are optional.

Parameters

- **calc_type** (*str, Optional*) – The calculation type: 'sim' or 'gen' Only used if app_name is not supplied. Uses default sim or gen application.
- **app_name** (*str, Optional*) – The application name. Must be supplied if calc_type is not.
- **num_procs** (*int, Optional*) – The total number of processes (MPI ranks)
- **num_nodes** (*int, Optional*) – The number of nodes
- **procs_per_node** (*int, Optional*) – The processes per node
- **num_gpus** (*int, Optional*) – The total number of GPUs
- **machinefile** (*str, Optional*) – Name of a machinefile
- **app_args** (*str, Optional*) – A string of the application arguments to be added to task submit command line
- **stdout** (*str, Optional*) – A standard output filename
- **stderr** (*str, Optional*) – A standard error filename
- **stage_inout** (*str, Optional*) – A directory to copy files from; default will take from current directory
- **hyperthreads** (*bool, Optional*) – Whether to submit MPI tasks to hyperthreads
- **dry_run** (*bool, Optional*) – Whether this is a dry_run - no task will be launched; instead runline is printed to logger (at INFO level)
- **wait_on_start** (*bool, Optional*) – Whether to wait for task to be polled as RUNNING (or other active/end state) before continuing
- **extra_args** (*str, Optional*) – Additional command line arguments to supply to MPI runner. If arguments are recognised as MPI resource configuration (num_procs,

num_nodes, procs_per_node) they will be used in resources determination unless also supplied in the direct options.

- **auto_assign_gpus** (*bool, Optional*) – Auto-assign GPUs available to this worker using either the method supplied in configuration or determined by detected environment. Default: False
- **match_procs_to_gpus** (*bool, Optional*) – For use with auto_assign_gpus. Auto-assigns MPI processors to match the assigned GPUs. Default: False unless auto_assign_gpus is True and no other CPU configuration is supplied.
- **env_script** (*str, Optional*) – The full path of a shell script to set up the environment for the launched task. This will be run in the subprocess, and not affect the worker environment. The script should start with a shebang.
- **mpi_runner_type** (*(str|dict), Optional*) – An MPI runner to be used for this submit only. Supply either a string for the MPI runner type or a dictionary for detailed configuration (see custom_info on MPIExecutor constructor). This will not change the default MPI runner for the executor. Example string inputs are "mpich", "openmpi", "srun", "jsrun", "aprun".

Returns
task – The launched task object

Return type
Task

Note that if some combination of num_procs, num_nodes, and procs_per_node is provided, these will be honored if possible. If resource detection is on and these are omitted, then the available resources will be divided among workers.

manager_kill_received()

Return True if received kill signal from the manager

Return type
bool

manager_poll()

Polls for a manager signal

The executor manager_signal attribute will be updated.

Return type
int

polling_loop(*task*, *timeout=None*, *delay=0.1*, *poll_manager=False*)

Optional, blocking, generic task status polling loop. Operates until the task finishes, times out, or is Optionally killed via a manager signal. On completion, returns a presumptive calc_status integer. Potentially useful for running an application via the Executor until it stops without monitoring its intermediate output.

Parameters

- **task** (*object*) – a Task object returned by the executor on submission
- **timeout** (*int, Optional*) – Maximum number of seconds for the polling loop to run. Tasks that run longer than this limit are killed. Default: No timeout
- **delay** (*int, Optional*) – Sleep duration between polling loop iterations. Default: 0.1 seconds
- **poll_manager** (*bool, Optional*) – Whether to also poll the manager for 'finish' or 'kill' signals. If detected, the task is killed. Default: False.

Returns
calc_status – presumptive integer attribute describing the final status of a launched task

Return type
int

`register_app`(*full_path*, *app_name=None*, *calc_type=None*, *desc=None*, *precedent=''*)

Registers a user application to libEnsemble.

The `full_path` of the application must be supplied. Either `app_name` or `calc_type` can be used to identify the application in user scripts (in the **submit** function). `app_name` is recommended.

Parameters

- **full_path** (*str*) – The full path of the user application to be registered
- **app_name** (*str, Optional*) – Name to identify this application.
- **calc_type** (*str, Optional*) – Calculation type: Set this application as the default 'sim' or 'gen' function.
- **desc** (*str, Optional*) – Description of this application
- **precedent** (*str, Optional*) – Any str that should directly precede the application full path.

Return type
None

Class-specific Attributes

Class-specific attributes can be set directly to alter the behavior of the MPI Executor. However, they should be used with caution, because they may not be implemented in other executors.

max_submit_attempts
(int) Maximum number of launch attempts for a given task. *Default: 5*.

fail_time
(int or float) *Only if wait_on_start is set.* Maximum run time to failure in seconds that results in relaunch. *Default: 2*.

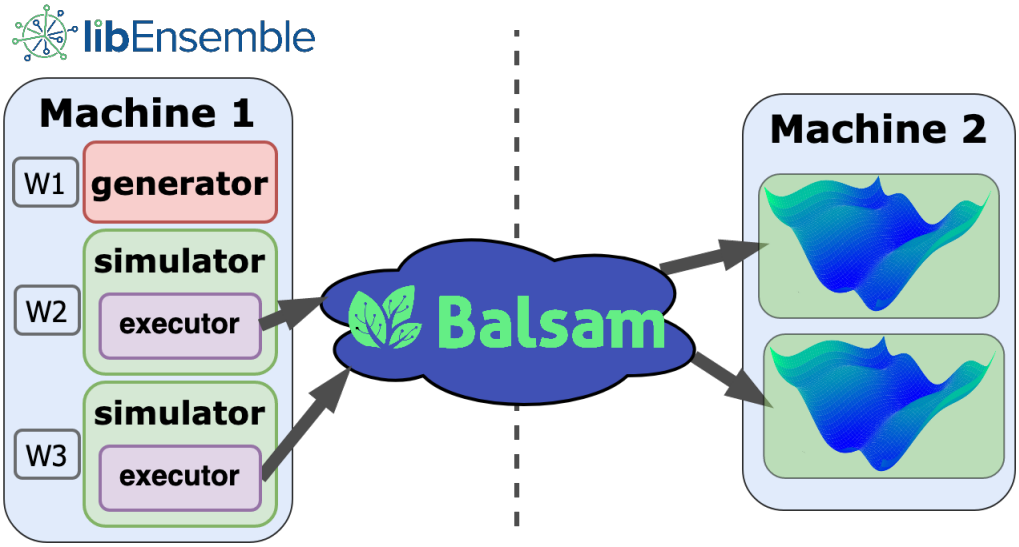
retry_delay_incr
(int or float) Delay increment between launch attempts in seconds. *Default: 5*. (i.e., First retry after 5 seconds, then 10 seconds, then 15, etc...)

Example. To increase resilience against submission failures:

```
taskctrl = MPIExecutor()
taskctrl.max_launch_attempts = 8
taskctrl.fail_time = 5
taskctrl.retry_delay_incr = 10
```

2.7.4 Balsam Executor - Remote apps

This module launches and controls tasks via Balsam, and can submit tasks from any machine, to any machine running a Balsam site.



At this time, access to Balsam is limited to those with valid organizational logins authenticated through Globus.

Initialization

To initialize a Balsam executor:

```
from libensemble.executors.balsam_executors import BalsamExecutor
exctr = BalsamExecutor()
```

App and Resource registration

Note that Balsam `ApplicationDefinition` instances are registered instead of paths and task submissions will not run until Balsam reserves compute resources at a site:

```
from libensemble.executors.balsam_executors import BalsamExecutor
from balsam.api import ApplicationDefinition

class HelloApp(ApplicationDefinition):
    site = "my-balsam-site"
    command_template = "/path/to/hello.app {{ my_name }}"

exctr = BalsamExecutor()
exctr.register_app(HelloApp, app_name="hello")

exctr.submit_allocation(
    site_id=999,  # corresponds to "my-balsam-site", found via ``balsam site ls``
    num_nodes=4,  # Total number of nodes requested for *all jobs*
    wall_time_min=30,
    queue="debug-queue",
```

(continues on next page)

(continued from previous page)

```
    project="my-project",
)
```

Task Submission

Task submissions of registered apps aren't too different from the other executors, except Balsam expects application arguments in dictionary form. Note that these fields must match the templating syntax in each `ApplicationDefinition`'s `command_template` field:

```
args = {"my_name": "World"}

task = exctr.submit(
    app_name="hello",
    app_args=args,
    num_procs=4,
    num_nodes=1,
    procs_per_node=4,
)
```

Application instances submitted by the executor to the Balsam service will get scheduled within the reserved resource allocation. **Each Balsam app can only be submitted to the site specified in its class definition.** Output files will appear in the Balsam site's `data` directory, but can be automatically transferred back via Globus.

Reading Balsam's documentation is highly recommended.

class libensemble.executors.balsam_executor.**BalsamExecutor**

Bases: Executor

Wraps the Balsam service. Via this Executor, Balsam `Jobs` can be submitted to Balsam sites, either local or on remote machines.

Note: Task kills are not configurable in the Balsam executor.

__init__()

Instantiate a new `BalsamExecutor` instance.

Return type
None

register_app(*BalsamApp*, *app_name=None*, *calc_type=None*, *desc=None*, *precedent=None*)

Registers a Balsam `ApplicationDefinition` to libEnsemble. This class instance *must* have a `site` and `command_template` specified. See the Balsam docs for information on other optional fields.

Parameters

- **BalsamApp** (`ApplicationDefinition` object) – A Balsam `ApplicationDefinition` instance.
- **app_name** (*str, Optional*) – Name to identify this application.
- **calc_type** (*str, Optional*) – Calculation type: Set this application as the default `'sim'` or `'gen'` function.
- **desc** (*str, Optional*) – Description of this application

- **precedent** (*str | None*) –

Return type
None

submit_allocation(*site_id*, *num_nodes*, *wall_time_min*, *job_mode='mpi'*, *queue='local'*, *project='local'*, *optional_params={}*, *filter_tags={}*, *partitions=[]*)

Submits a Balsam `BatchJob` machine allocation request to Balsam. Corresponding Balsam applications with a matching site can be submitted to this allocation. Effectively a wrapper for `BatchJob.objects.create()`.

Parameters

- **site_id** (*int*) – The corresponding `site_id` for a Balsam site. Retrieve via `balsam site ls`
- **num_nodes** (*int*) – The number of nodes to request from a machine with a running Balsam site
- **wall_time_min** (*int*) – The number of walltime minutes to request for the `BatchJob` allocation
- **job_mode** (*str, Optional*) – Either `"serial"` or `"mpi"`. Default: `"mpi"`
- **queue** (*str, Optional*) – Specifies the queue from which the `BatchJob` should request nodes. Default: `"local"`
- **project** (*str, Optional*) – Specifies the project that should be charged for the requested machine time. Default: `"local"`
- **optional_params** (*dict, Optional*) – Additional system-specific parameters to set, based on fields in Balsam's `job-template.sh`
- **filter_tags** (*dict, Optional*) – Directs the resultant `BatchJob` to only run Jobs with matching tags.
- **partitions** (*List[dict], Optional*) – Divides the allocation into multiple launcher partitions, with differing `job_mode`, `num_nodes`. `filter_tags`, etc. See the Balsam docs.

Return type
The corresponding `BatchJob` object.

revoke_allocation(*allocation*, *timeout=60*)

Terminates a Balsam `BatchJob` machine allocation remotely. Balsam apps should no longer be submitted to this allocation. Best to run after libEnsemble completes, or after this `BatchJob` is no longer needed. Helps save machine time.

Parameters

- **allocation** (`BatchJob` object) – a `BatchJob` with a corresponding machine allocation that should be cancelled.
- **timeout** (*int, Optional*) – Timeout and warn user after this many seconds of attempting to revoke an allocation.

Return type
bool

submit(*calc_type=None*, *app_name=None*, *app_args=None*, *num_procs=None*, *num_nodes=None*, *procs_per_node=None*, *max_tasks_per_node=None*, *machinefile=None*, *gpus_per_rank=0*, *transfers={}*, *workdir=''*, *dry_run=False*, *wait_on_start=False*, *extra_args={}*, *tags={}*)

Initializes and submits a Balsam `Job` based on a registered `ApplicationDefinition` and requested resources. A corresponding libEnsemble `Task` object is returned.

Parameters

- **calc_type** (*str, Optional*) – The calculation type: `'sim'` or `'gen'` Only used if `app_name` is not supplied. Uses default sim or gen application.
- **app_name** (*str, Optional*) – The application name. Must be supplied if `calc_type` is not.
- **app_args** (*dict*) – A dictionary of options that correspond to fields to template in the ApplicationDefinition's `command_template` field.
- **num_procs** (*int, Optional*) – The total number of MPI ranks on which to submit the task
- **num_nodes** (*int, Optional*) – The number of nodes on which to submit the task
- **procs_per_node** (*int, Optional*) – The processes per node for this task
- **max_tasks_per_node** (*int*) – Instructs Balsam to schedule at most this many Jobs per node.
- **machinefile** (*str, Optional*) – Name of a machinefile for this task to use. Unused by Balsam
- **gpus_per_rank** (*int, Optional*) – Number of GPUs to reserve for each MPI rank
- **transfers** (*dict, Optional*) – A Job-specific Balsam transfers dictionary that corresponds with an `ApplicationDefinition` `transfers` field. See the Balsam docs for more information.
- **workdir** (*str*) – Specifies as name for the Job's output directory within the Balsam site's data directory. Default: `libe_workflow`
- **dry_run** (*bool, Optional*) – Whether this is a dry run - no task will be launched; instead runline is printed to logger (at `INFO` level)
- **wait_on_start** (*bool, Optional*) – Whether to block, and wait for task to be polled as `RUNNING` (or other active/end state) before continuing
- **extra_args** (*dict, Optional*) – Additional arguments to supply to MPI runner.
- **tags** (*dict, Optional*) – Additional tags to organize the `Job` or restrict which `BatchJobs` run it.

Returns

task – The launched task object

Return type

`BalsamTask`

Note that since Balsam Jobs are often sent to entirely different machines than where libEnsemble is running, how libEnsemble's resource manager has divided local resources among workers doesn't impact what resources can be requested for a Balsam `Job` running on an entirely different machine.

class libensemble.executors.balsam_executor.**BalsamTask**(*app=None*, *app_args=None*, *workdir=None*, *stdout=None*, *stderr=None*, *workerid=None*)

Bases: `Task`

Wraps a Balsam `Job` from the Balsam service.

The same attributes and query routines are implemented. Use `task.process` to refer to the matching Balsam `Job` initialized by the `BalsamExecutor`, with every Balsam `Job` method invocable on it. Otherwise, libEnsemble task methods like `poll()` can be used directly.

Parameters

- **app** (*Application | None*) –
- **app_args** (*dict*) –
- **workdir** (*str | None*) –
- **stdout** (*str*) –
- **stderr** (*str*) –
- **workerid** (*int*) –

`poll()`

Polls and updates the status attributes of the supplied task. Requests Job information from Balsam service.

Return type
None

`wait(timeout=None)`

Waits on completion of the task or raises `TimeoutExpired`.

Status attributes of task are updated on completion.

Parameters
timeout (*int or float, Optional*) – Time in seconds after which a TimeoutExpired exception is raised. If not set, then simply waits until completion. Note that the task is not automatically killed on timeout.

Return type
None

`kill()`

Cancels the task. Killing a running task is unsupported by Balsam at this time.

Return type
None

2.8 Convenience Tools and Functions

Setup Helpers

`tools.add_unique_random_streams(persis_info, nstreams, seed='')`

Creates nstreams random number streams for the libE manager and workers when nstreams is num_workers + 1. Stream i is initialized with seed i by default. Otherwise the streams can be initialized with a provided seed.

The entries are appended to the provided persis_info dictionary.

```
persis_info = add_unique_random_streams(old_persis_info, nworkers + 1)
```

Parameters

- **persis_info** (`dict`) – Persistent information dictionary. `(example)`
- **nstreams** (`int`) – Number of independent random number streams to produce.

- **seed** (int) – (Optional) Seed for identical random number streams for each worker. If explicitly set to None, random number streams are unique and seed via other pseudorandom mechanisms.

tools.**eprint**(**args*, ***kwargs*)

Prints a user message to standard error

class tools.**ForkablePdb**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*)

A Pdb subclass that may be used from a forked multiprocessing child

Usage:

```
from libensemble.tools import ForkablePdb

ForkablePdb().set_trace()
```

tools.**parse_args**()

Parses command-line arguments. Use in calling script.

```
from libensemble.tools import parse_args

nworkers, is_manager, libE_specs, misc_args = parse_args()
```

Or for object interface, when creating the ensemble object.

```
from libensemble import Ensemble

ensemble = Ensemble(parse_args=True)
```

From the shell:

```
$ python calling_script --comms local --nworkers 4
```

Usage:

```
usage: test_... [-h] [--comms [{local, tcp, ssh, client, mpi}]]
                [--nworkers [NWORKERS]] [--workers WORKERS [WORKERS ...]]
                [--nsim_workers [NSIM_WORKERS]]
                [--nresource_sets [NRESOURCE_SETS]]
                [--workerID [WORKERID]] [--server SERVER SERVER SERVER]
                [--pwd [PWD]] [--worker_pwd [WORKER_PWD]]
                [--worker_python [WORKER_PYTHON]]
                [--tester_args [TESTER_ARGS [TESTER_ARGS ...]]]

Note that running via an MPI runner uses the default 'mpi' comms, and '--nworkers'
will be ignored. The number of processes are supplied via the MPI run line. One␣
↪being
the manager, and the rest are workers.

--comms,          Communications medium for manager and workers. Default is 'mpi'.
--nworkers,       (For 'local' or 'tcp' comms) Set number of workers.
--nresource_sets, Explicitly set the number of resource sets. This sets
                  libE_specs["num_resource_sets"]. By default, resources will be
```

(continues on next page)

(continued from previous page)

```
                    divided by workers (excluding zero_resource_workers).
--nsim_workers,     (For 'local" or 'mpi' comms) A convenience option for cases with
                    persistent generators - sets the number of simulation workers.
                    If used with no other criteria, one additional worker for running␣
↪a
                    generator will be added, and the number of resource sets will be␣
↪assigned
                    the given value. If '--nworkers' has also been specified, will␣
↪generate
                    enough additional workers to match the other criteria. If '--
↪nresource_sets'
                    is also specified, will not override resource sets.

Example command lines:

Run with 'local' comms and 4 workers
$ python calling_script --comms local --nworkers 4

Run with 'local' comms and 5 workers - one gen worker (no resources), and 4 sim␣
↪workers.
$ python calling_script --comms local --nsim_workers 4

Run with 'local' comms with 4 workers and 8 resource sets. The extra resource sets␣
↪will
be used for larger simulations (using variable resource assignment).
$ python calling_script --comms local --nresource_sets 8

Previous example with 'mpi' comms.
$ mpirun -np 5 python calling_script --nresource_sets 8
```

Returns

- **nworkers** (int) – Number of workers libEnsemble will initiate
- **is_manager** (bool) – Indicates whether the current process is the manager process
- **libE_specs** (dict) – Settings and specifications for libEnsemble (example)

tools.**save_libE_output**(*H*, *persis_info*, *calling_file*, *nworkers*, *dest_path='/home/docs/checkouts/readthedocs.org/user_builds/libensemble/checkouts/docs-forces_tutorial/docs'*, *mess='Run completed'*)

Writes out history array and persis_info to files.

Format: <calling_script>_results_History_length=<length>_evals=<Completed evals>_ranks=<nworkers>

```
save_libE_output(H, persis_info, __file__, nworkers)
```

Parameters

- **H** (NumPy structured array) – History array storing rows for each point. (example)
- **persis_info** (dict) – Persistent information dictionary. (example)
- **calling_file** (str) – Name of user-calling script (or user chosen name) to prefix output files. The convention is to send __file__ from user calling script.

- **nworkers** (int) – The number of workers in this ensemble. Added to output file names.
- **mess** (str) – A message to print/log when saving the file.

Persistent Helpers

These routines are commonly used within persistent generator functions such as `persistent_aposmm` in `libensemble/gen_funcs/` for intermediate communication with the manager. Persistent simulator functions are also supported.

class persistent_support.**PersistentSupport**(*libE_info*, *calc_type*)

A helper class to assist with writing persistent user functions.

Parameters

- **libE_info** (*Dict[str, Dict[Any, Any]]*) –
- **calc_type** (*int*) –

send(*output*, *calc_status=0*, *keep_state=False*)

Send message from worker to manager.

Parameters

- **output** (*ndarray[Any, dtype[_ScalarType_co]]*) – Output array to be sent to manager.
- **calc_status** (*int*) – (Optional) Provides a task status.
- **keep_state** – (Optional) If True the manager will not modify its record of the workers state (usually the manager changes the worker's state to inactive, indicating the worker is ready to receive more work, unless using active receive mode).

Return type
None

recv(*blocking=True*)

Receive message to worker from manager.

Parameters
blocking (*bool*) – (Optional) If True (default), will block until a message is received.

Returns
message tag, Work dictionary, calc_in array

Return type
(<class 'int'>, <class 'dict'>, numpy.ndarray[*Any*, numpy.dtype[+_ScalarType_co]])

send_recv(*output*, *calc_status=0*)

Send message from worker to manager and receive response.

Parameters

- **output** (*ndarray[Any, dtype[_ScalarType_co]]*) – Output array to be sent to manager.
- **calc_status** (*int*) – (Optional) Provides a task status.

Returns
message tag, Work dictionary, calc_in array

Return type
(<class 'int'>, <class 'dict'>, numpy.ndarray[*Any*, numpy.dtype[+_ScalarType_co]])

request_cancel_sim_ids(*sim_ids*)

Request cancellation of sim_ids.

Parameters
sim_ids (*List[int]*) – A list of sim_ids to cancel.

A message is sent to the manager to mark requested sim_ids as cancel_requested.

Allocation Helpers

These routines are used within custom allocation functions to help prepare `Work` structures for workers. See the routines within `libensemble/alloc_funcs/` for examples.

exception alloc_support.**AllocException**

Raised for any exception in the alloc support

class alloc_support.**AllocSupport**(*W*, *manage_resources=False*, *persis_info={}*, *libE_info={}*, *user_resources=None*, *user_scheduler=None*)

A helper class to assist with writing allocation functions.

This class contains methods for common operations like populating work units, determining which workers are available, evaluating what values need to be distributed to workers, and others.

Note that since the `alloc_f` is called periodically by the Manager, this class instance (if used) will be recreated/destroyed on each loop.

assign_resources(*rsets_req*, *use_gpus=None*, *user_params=[]*)

Schedule resource sets to a work record if possible.

For default scheduler, if more than one group (node) is required, will try to find even split, otherwise allocates whole nodes.

Raises `InsufficientFreeResources` if the required resources are not currently available, or `InsufficientResourcesError` if the required resources do not exist.

Parameters

- **rsets_req** – Int. Number of resource sets to request.
- **use_gpus** – Bool. Whether to use GPU resource sets.
- **user_params** – List of Integers. User parameters num_procs, num_gpus.

Returns
List of Integers. Resource set indices assigned.

avail_worker_ids(*persistent=None*, *active_recv=False*, *zero_resource_workers=None*)

Returns available workers as a list of IDs, filtered by the given options.

Parameters

- **persistent** – (Optional) Int. Only return workers with given `persis_state` (1=sim, 2=gen).
- **active_recv** – (Optional) Boolean. Only return workers with given active_recv state.
- **zero_resource_workers** – (Optional) Boolean. Only return workers that require no resources.

Returns
List of worker IDs.

If there are no zero resource workers defined, then the `zero_resource_workers` argument will be ignored.

count_gens()

Returns the number of active generators.

test_any_gen()

Returns `True` if a generator worker is active.

count_persis_gens()

Return the number of active persistent generators.

sim_work(*wid*, *H*, *H_fields*, *H_rows*, *persis_info*, ***libE_info*)

Add sim work record to given `Work` dictionary.

Includes evaluation of required resources if the worker is not in a persistent state.

Parameters

- **wid** – Int. Worker ID.
- **H** – `History array`. For parsing out requested resource sets.
- **H_fields** – Which fields from `H` to send.
- **H_rows** – Which rows of `H` to send.
- **persis_info** – Worker specific `persis_info` dictionary.

Returns

a Work entry.

Additional passed parameters are inserted into `libE_info` in the resulting work record.

If `rset_team` is passed as an additional parameter, it will be honored, assuming that any resource checking has already been done.

gen_work(*wid*, *H_fields*, *H_rows*, *persis_info*, ***libE_info*)

Add gen work record to given `Work` dictionary.

Includes evaluation of required resources if the worker is not in a persistent state.

Parameters

- **Work** – `Work dictionary`.
- **wid** – Worker ID.
- **H_fields** – Which fields from `H` to send.
- **H_rows** – Which rows of `H` to send.
- **persis_info** – Worker specific `persis_info` dictionary.

Returns

A Work entry.

Additional passed parameters are inserted into `libE_info` in the resulting work record.

If `rset_team` is passed as an additional parameter, it will be honored, and assume that any resource checking has already been done. For example, passing `rset_team=[]`, would ensure that no resources are assigned.

all_sim_started(*H*, *pt_filter=None*, *low_bound=None*)

Returns `True` if all expected points have started their sim.

Excludes cancelled points.

Parameters

- **pt_filter** – (Optional) Boolean array filtering expected returned points in `H`.
- **low_bound** – (Optional) Lower bound for testing all returned.

Returns

True if all expected points have started their sim.

all_sim_ended(*H*, *pt_filter=None*, *low_bound=None*)

Returns `True` if all expected points have had their sim_end.

Excludes cancelled points that were not already sim_started.

Parameters

- **pt_filter** – (Optional) Boolean array filtering expected returned points in `H`.
- **low_bound** – (Optional) Lower bound for testing all returned.

Returns

True if all expected points have had their sim_end.

all_gen_informed(*H*, *pt_filter=None*, *low_bound=None*)

Returns `True` if gen has been informed of all expected points.

Excludes cancelled points that were not already given out.

Parameters

- **pt_filter** – (Optional) Boolean array filtering expected sim_end points in `H`.
- **low_bound** – (Optional) Lower bound for testing all returned.

Returns

True if gen have been informed of all expected points.

points_by_priority(*H*, *points_avail*, *batch=False*)

Returns indices of points to give by priority.

Parameters

- **points_avail** – Indices of points that are available to give.
- **batch** – (Optional) Boolean. Should batches of points with the same priority be given simultaneously.

Returns

An array of point indices to give.

Chapter 3

Running libEnsemble

libEnsemble runs with one manager and multiple workers. Each worker may run either a generator or simulator function (both are Python scripts). Generators determine the parameters/inputs for simulations. Simulator functions run and manage simulations, which often involve running a user application (see `Executor`).

To use libEnsemble, you will need a calling script, which in turn will specify generator and simulator functions. Many `examples` are available.

There are currently three communication options for libEnsemble (determining how the Manager and Workers communicate). These are `mpi`, `local`, `tcp`. The default is `mpi`.

Note: You do not need the `mpi` communication mode to use the `MPI Executor`. The communication modes described here only refer to how the libEnsemble manager and workers communicate.

MPI Comms

This option uses mpi4py for the Manager/Worker communication. It is used automatically if you run your libEnsemble calling script with an MPI runner such as:

```
mpirun -np N python myscript.py
```

where `N` is the number of processes. This will launch one manager and `N-1` workers.

This option requires `mpi4py` to be installed to interface with the MPI on your system. It works on a standalone system, and with both `central and distributed modes` of running libEnsemble on multi-node systems.

It also potentially scales the best when running with many workers on HPC systems.

Limitations of MPI mode

If launching MPI applications from workers, then MPI is nested. **This is not supported with Open MPI**. This can be overcome by using a proxy launcher (see `Balsam`). This nesting does work with MPICH and its derivative MPI implementations.

It is also unsuitable to use this mode when running on the **launch** nodes of three-tier systems (e.g., Theta/Summit). In that case `local` mode is recommended.

Local Comms

Uses Python's built-in multiprocessing module. The `comms` type `local` and number of workers `nworkers` may be provided in `libE_specs`. Then run:

```
python myscript.py
```

Or, if the script uses the `parse_args()` function or an `Ensemble` object with `Ensemble(parse_args=True)`, you can specify these on the command line:

```
python myscript.py --comms local --nworkers N
```

This will launch one manager and `N` workers.

libEnsemble will run on **one node** in this scenario. To `disallow this node` from app-launches (if running libEnsemble on a compute node), set `libE_specs["dedicated_mode"] = True`.

This mode is often used to run on a **launch** node of a three-tier system (e.g., Theta/Summit), ensuring the whole compute-node allocation is available for launching apps. Make sure there are no imports of `mpi4py` in your Python scripts.

Note that on macOS (since Python 3.8) and Windows, the default multiprocessing method is `"spawn"` instead of `"fork"`; to resolve many related issues, we recommend placing calling script code in an `if __name__ == "__main__":` block.

Limitations of local mode

- Workers cannot be `distributed` across nodes.
- In some scenarios, any import of `mpi4py` will cause this to break.
- Does not have the potential scaling of MPI mode, but is sufficient for most users.

TCP Comms

Run the Manager on one system and launch workers to remote systems or nodes over TCP. Configure through `libE_specs`, or on the command line if using an `Ensemble` object with `Ensemble(parse_args=True)`,

Reverse-ssh interface

Set `comms` to `ssh` to launch workers on remote ssh-accessible systems. This colocates workers, functions, and any applications. User functions can also be persistent, unlike when launching remote functions via `Globus Compute`.

The remote working directory and Python need to be specified. This may resemble:

```
python myscript.py --comms ssh --workers machine1 machine2 --worker_pwd /home/workers --
↪worker_python /home/.conda/.../python
```

Limitations of TCP mode

- There cannot be two calls to `libE()` or `Ensemble.run()` in the same script.

3.1 Further Command Line Options

See the **parse_args()** function in `Convenience Tools` for further command line options.

3.2 liberegister / libesubmit

Command-line utilities for preparing and launching libEnsemble workflows onto almost any machine and any scheduler, using a PSI/J Python implementation.

liberegister

Creates an initial, platform-independent PSI/J serialization of a libEnsemble submission. Run this utility on a script:

```
liberegister my_calling_script.py --comms local --nworkers 4
```

This produces an initial `my_calling_script.json` serialization conforming to PSI/J's specification:

my_calling_script.json

```
{
    "version": 0.1,
    "type": "JobSpec",
    "data": {
        "name": "libe-job",
        "executable": "python",
        "arguments": [
            "my_calling_script.py",
            "--comms",
            "local",
            "--nworkers",
            "4"
        ],
        "directory": null,
        "inherit_environment": true,
        "environment": {
            "PYTHONNOUSERSITE": "1"
        },
        "stdin_path": null,
        "stdout_path": null,
        "stderr_path": null,
        "resources": {
            "node_count": 1,
            "process_count": null,
            "process_per_node": null,
            "cpu_cores_per_process": null,
            "gpu_cores_per_process": null,
            "exclusive_node_use": true
        },
        "attributes": {
```

(continues on next page)

(continued from previous page)

```
            "duration": "30",
            "queue_name": null,
            "project_name": null,
            "reservation_id": null,
            "custom_attributes": {}
        },
        "launcher": null
    }
}
```

libesubmit

Further parameterizes a serialization, and submits a corresponding Job to the specified scheduler:

```
libesubmit my_calling_script.json -q debug -A project -s slurm --nnodes 8
```

Results in:

```
*** libEnsemble 0.9.3 ***
Imported PSI/J serialization: my_calling_script.json. Preparing submission...
Calling script: my_calling_script.py
...found! Proceeding.
Submitting Job!: Job[id=ce4ead75-a3a4-42a3-94ff-c44b3b2c7e61, native_id=None,␣
↪executor=None, status=JobStatus[NEW, time=1658167808.5125017]]

$ squeue --long --users=user
Mon Jul 18 13:10:15 2022
        JOBID PARTITION     NAME     USER    STATE       TIME TIME_LIMI  NODES␣
↪NODELIST(REASON)
    2508936    debug  ce4ead75     user  PENDING       0:00     30:00      8 (Priority)
```

This also produces a Job-specific representation, e.g:

8ba9de56.my_calling_script.json

```
{
    "version": 0.1,
    "type": "JobSpec",
    "data": {
        "name": "libe-job",
        "executable": "/Users/jnavarro/miniconda3/envs/libe/bin/python3.9",
        "arguments": [
            "my_calling_script.py",
            "--comms",
            "local",
            "--nworkers",
            "4"
        ],
        "directory": "/home/user/libensemble/scratch",
        "inherit_environment": true,
```

(continues on next page)

(continued from previous page)

```
        "environment": {
            "PYTHONNOUSERSITE": "1"
        },
        "stdin_path": null,
        "stdout_path": "8ba9de56.my_calling_script.out",
        "stderr_path": "8ba9de56.my_calling_script.err",
        "resources": {
            "node_count": 8,
            "process_count": null,
            "process_per_node": null,
            "cpu_cores_per_process": null,
            "gpu_cores_per_process": null,
            "exclusive_node_use": true
        },
        "attributes": {
            "duration": "30",
            "queue_name": "debug",
            "project_name": "project",
            "reservation_id": null,
            "custom_attributes": {}
        },
        "launcher": null
    }
}
```

If libesubmit is run on a `.json` serialization from liberegister and can't find the specified calling script, it'll help search for matching candidate scripts.

3.3 Persistent Workers

In a regular (non-persistent) worker, the user's generator or simulation function is called whenever the worker receives work. A persistent worker is one that continues to run the generator or simulation function between work units, maintaining the local data environment.

A common use-case consists of a persistent generator (such as `persistent_aposmm`) that maintains optimization data while generating new simulation inputs. The persistent generator runs on a dedicated worker while in persistent mode. This requires an appropriate `allocation function` that will run the generator as persistent.

When running with a persistent generator, it is important to remember that a worker will be dedicated to the generator and cannot run simulations. For example, the following run:

```
mpirun -np 3 python my_script.py
```

starts one manager, one worker with a persistent generator, and one worker for running simulations.

If this example was run as:

```
mpirun -np 2 python my_script.py
```

No simulations will be able to run.

3.4 Environment Variables

Environment variables required in your run environment can be set in your Python sim or gen function. For example:

```
os.environ["OMP_NUM_THREADS"] = 4
```

set in your simulation script before the Executor *submit* command will export the setting to your run. For running a bash script in a sub environment when using the Executor, see the `env_script` option to the `MPI Executor`.

3.5 Further Run Information

For running on multi-node platforms and supercomputers, there are alternative ways to configure libEnsemble to resources. See the `Running on HPC Systems` guide for more information, including some examples for specific systems.

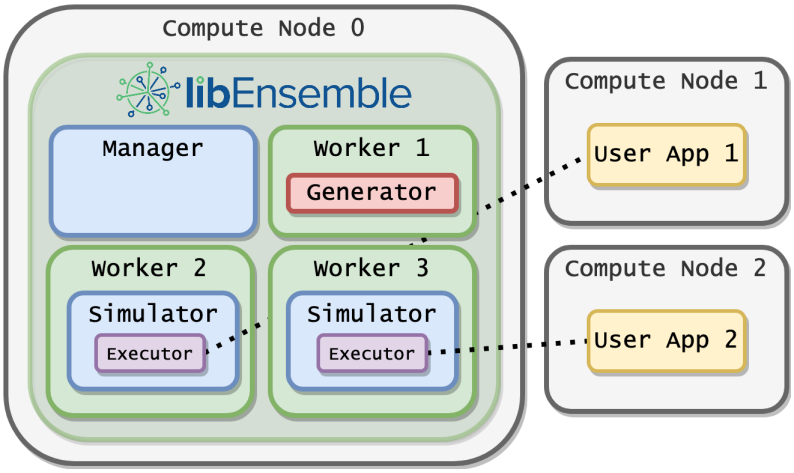
Chapter 4

Running on HPC Systems

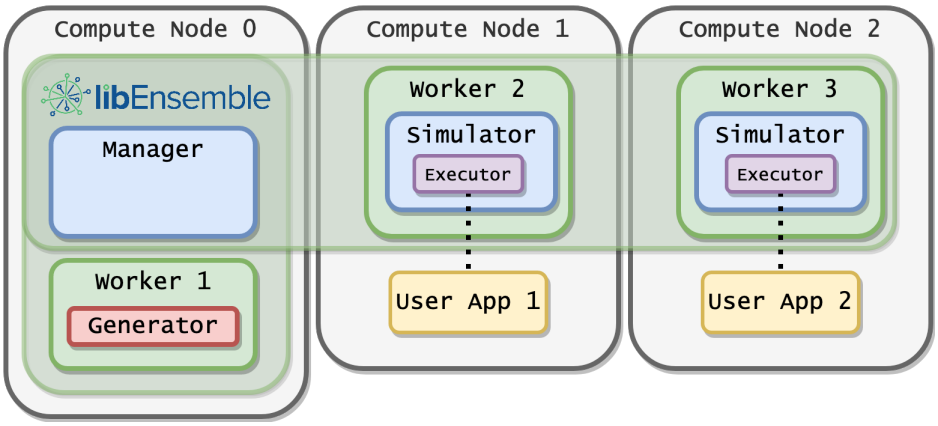
4.1 Central vs. Distributed

libEnsemble has been developed, supported, and tested on systems of highly varying scales, from laptops to thousands of compute nodes. On multi-node systems, there are two basic modes of configuring libEnsemble to run and launch tasks (user applications) on the available nodes.

The first mode we refer to as **central** mode, where the libEnsemble manager and worker processes are grouped onto one or more dedicated nodes. Workers launch applications onto the remaining allocated nodes:



Alternatively, in **distributed** mode, the libEnsemble (manager/worker) processes will share nodes with submitted tasks. This enables libEnsemble, using the *mpi4py* communicator, to be run with the workers spread across nodes so as to be co-located with their tasks.



Configurations with multiple nodes per worker or multiple workers per node are both common use cases. The distributed approach allows the libEnsemble worker to read files produced by the application on local node storage. HPC systems that allow only one application to be launched to a node at any one time prevent distributed configuration.

4.2 Configuring the Run

On systems with a job scheduler, libEnsemble is typically run within a single `job submission`. All user simulations will run on the nodes within that allocation.

How does libensemble know where to run tasks (user applications)?

The libEnsemble `Executor` can be initialized from the user calling script, and then used by workers to run tasks. The Executor will automatically detect the nodes available on most systems. Alternatively, the user can provide a file called **node_list** in the run directory. By default, the Executor will divide up the nodes evenly to each worker. If the argument `libE_specs["dedicated_mode"]=True` is used when initializing libEnsemble, then any node that is running a libEnsemble manager or worker will be removed from the node-list available to the workers, ensuring libEnsemble has dedicated nodes.

To run in central mode using a 5-node allocation with 4 workers: From the head node of the allocation:

```
mpirun -np 5 python myscript.py
```

or:

```
python myscript.py --comms local --nworkers 4
```

Either of these will run libEnsemble (inc. manager and 4 workers) on the first node. The remaining 4 nodes will be divided amongst the workers for submitted applications. If the same run was performed without `libE_specs["dedicated_mode"]=True`, runs could be submitted to all 5 nodes. The number of workers can be modified to allow either multiple workers to map to each node or multiple nodes per worker.

To launch libEnsemble distributed requires a less trivial libEnsemble run script. For example:

```
mpirun -np 5 -ppn 1 python myscript.py
```

would launch libEnsemble with 5 processes across 5 nodes. However, the manager would have its own node, which is likely wasteful. More often, a `machinefile` is used to add the manager to the first node. In the `examples` directory, you can find an example submission script, configured to run libensemble distributed, with multiple workers per node or multiple nodes per worker, and adding the manager onto the first node.

HPC systems that only allow one application to be launched to a node at any one time, will not allow a distributed configuration.

4.3 Systems with Launch/MOM Nodes

Some large systems have a 3-tier node setup. That is, they have a separate set of launch nodes (known as MOM nodes on Cray Systems). User batch jobs or interactive sessions run on a launch node. Most such systems supply a special MPI runner that has some application-level scheduling capability (e.g., `aprun`, `jsrun`). MPI applications can only be submitted from these nodes. Examples of these systems include: Summit, Sierra, and Theta.

There are two ways of running libEnsemble on these kinds of systems. The first, and simplest, is to run libEnsemble on the launch nodes. This is often sufficient if the worker's simulation or generation functions are not doing much work (other than launching applications). This approach is inherently centralized. The entire node allocation is available for the worker-launched tasks.

However, running libEnsemble on the compute nodes is potentially more scalable and will better manage simulation and generation functions that contain considerable computational work or I/O. Therefore the second option is to use proxy task-execution services like Balsam.

4.4 Balsam - Externally Managed Applications

Running libEnsemble on the compute nodes while still submitting additional applications requires alternative Executors that connect to external services like Balsam. Balsam can take tasks submitted by workers and execute them on the remaining compute nodes, or *to entirely different systems*.

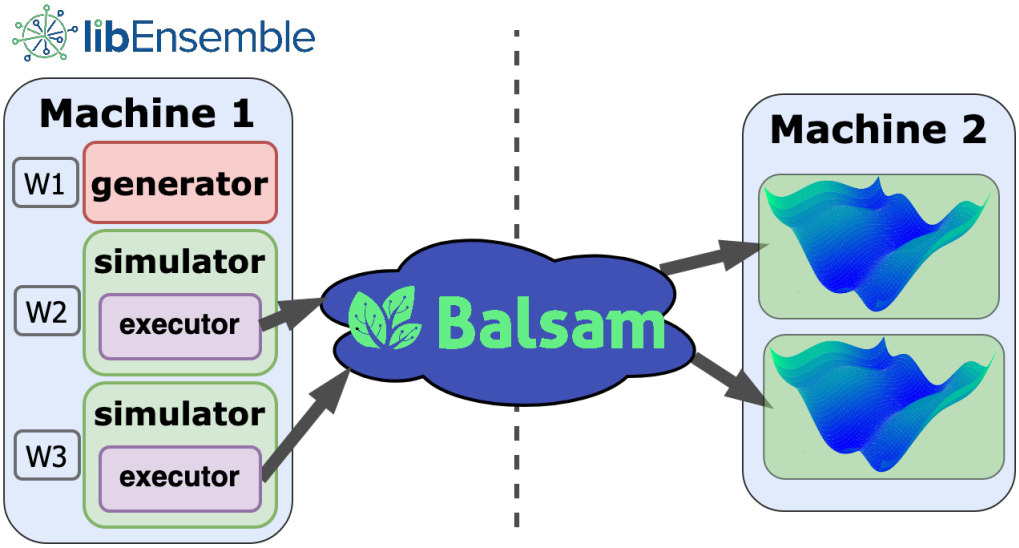


Fig. 1: (New) Multi-System: libEnsemble + BalsamExecutor

Submission scripts for running on launch/MOM nodes and for using Balsam, can be found in the `examples`.

4.5 Mapping Tasks to Resources

The `resource manager` can `detect system resources`, and partition these to workers. The `MPI Executor` accesses the resources available to the current worker when launching tasks.

4.5.1 Zero-resource workers

Users with persistent `gen_f` functions may notice that the persistent workers are still automatically assigned system resources. This can be resolved by `fixing the number of resource sets`.

4.6 Overriding Auto-Detection

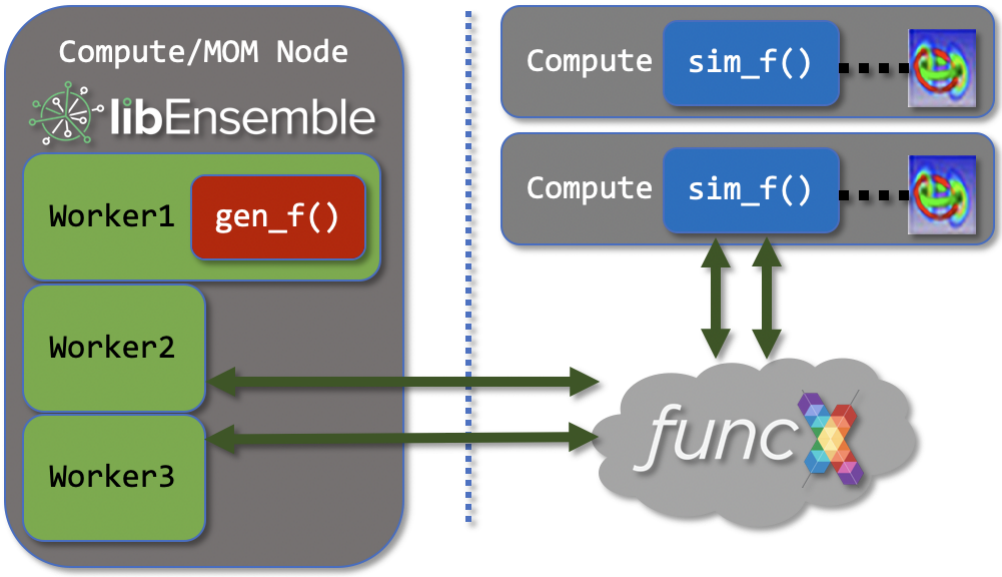
libEnsemble can automatically detect system information. This includes resource information, such as available nodes and the number of cores on the node, and information about available MPI runners.

System detection for resources can be overridden using the `resource_info` libE_specs option.

When using the MPI Executor, it is possible to override the detected information using the *custom_info* argument. See the `MPI Executor` for more.

4.7 Globus Compute - Remote User Functions

Alternatively to much of the above, if libEnsemble is running on some resource with internet access (laptops, login nodes, other servers, etc.), workers can be instructed to launch generator or simulator user function instances to separate resources from themselves via Globus Compute (formerly funcX), a distributed, high-performance function-as-a-service platform:



This is useful for running ensembles across machines and heterogeneous resources, but comes with several caveats:

1. User functions registered with Globus Compute must be *non-persistent*, since manager-worker communicators can't be serialized or used by a remote resource.

2. Likewise, the `Executor.manager_poll()` capability is disabled. The only available control over remote functions by workers is processing return values or exceptions when they complete.
3. Globus Compute imposes a handful of task-rate and data limits on submitted functions.
4. Users are responsible for authenticating via Globus and maintaining their Globus Compute endpoints on their target systems.

Users can still define Executor instances within their user functions and submit MPI applications normally, as long as libEnsemble and the target application are accessible on the remote system:

```
# Within remote user function
from libensemble.executors import MPIExecutor
exctr = MPIExecutor()
exctr.register_app(full_path="/home/user/forces.x", app_name="forces")
task = exctr.submit(app_name="forces", num_procs=64)
```

Specify a Globus Compute endpoint in either `sim_specs` or `gen_specs` via the `globus_compute_endpoint` argument. For example:

```
from libensemble.specs import SimSpecs

sim_specs = SimSpecs(
    sim_f = sim_f,
    inputs = ["x"],
    out = [("f", float)],
    globus_compute_endpoint = "3af6dc24-3f27-4c49-8d11-e301ade15353",
)
```

See the `libensemble/tests/scaling_tests/globus_compute_forces` directory for a complete remote-simulation example.

4.8 Instructions for Specific Platforms

The following subsections have more information about configuring and launching libEnsemble on specific HPC systems.

4.8.1 Bebop

Bebop is a Cray CS400 cluster with Intel Broadwell and Knights Landing compute nodes available in the Laboratory Computing Resources Center (LCRC) at Argonne National Laboratory.

Configuring Python

Begin by loading the Python 3 Anaconda module:

```
module load anaconda3
```

Create a conda virtual environment in which to install libEnsemble and all dependencies:

```
conda config --add channels intel
conda create --name my_env intelpython3_core python=3
source activate my_env
```

Installing libEnsemble and Dependencies

You should have an indication that the virtual environment is activated. Start by installing mpi4py in this environment, making sure to reference the preinstalled Intel MPI compiler. Your prompt should be similar to the following block:

```
(my_env) user@login:~$ CC=mpiicc MPICC=mpiicc pip install mpi4py --no-binary mpi4py
```

libEnsemble can then be installed via `pip` or `conda`. To install via `pip`:

```
(my_env) user@login:~$ pip install libensemble
```

To install via `conda`:

```
(my_env) user@login:~$ conda config --add channels conda-forge
(my_env) user@login:~$ conda install -c conda-forge libensemble
```

See here for more information on advanced options for installing libEnsemble.

Job Submission

Bebop uses Slurm for job submission and management. The two commands you'll likely use the most to run jobs are `srun` and `sbatch` for running interactively and batch, respectively.

libEnsemble node-worker affinity is especially flexible on Bebop. By adjusting `srun` runtime options users may assign multiple libEnsemble workers to each allocated node(oversubscription) or assign multiple nodes per worker.

Interactive Runs

You can allocate four Knights Landing nodes for thirty minutes through the following:

```
salloc -N 4 -p knl -A [username OR project] -t 00:30:00
```

With your nodes allocated, queue your job to start with four MPI ranks:

```
srun -n 4 python calling.py
```

`mpirun` should also work. This line launches libEnsemble with a manager and **three** workers to one allocated compute node, with three nodes available for the workers to launch calculations with the Executor or a launch command. This is an example of running in `centralized` mode, and, if using the `Executor`, libEnsemble should be initiated with `libE_specs["dedicated_mode"]=True`

Note: When performing a `distributed` MPI libEnsemble run and not oversubscribing, specify one more MPI process than the number of allocated nodes. The manager and first worker run together on a node.

If you would like to interact directly with the compute nodes via a shell, the following starts a bash session on a Knights Landing node for thirty minutes:

```
srun --pty -A [username OR project] -p knl -t 00:30:00 /bin/bash
```

Note: You will need to reactivate your conda virtual environment and reload your modules! Configuring this routine to occur automatically is recommended.

Batch Runs

Batch scripts specify run settings using `#SBATCH` statements. A simple example for a libEnsemble use case running in `distributed` MPI mode on Broadwell nodes resembles the following:

```
#!/bin/bash
#SBATCH -J myjob
#SBATCH -N 4
#SBATCH -p bdwall
#SBATCH -A myproject
#SBATCH -o myjob.out
#SBATCH -e myjob.error
#SBATCH -t 00:15:00

# These four lines construct a machinefile for the executor and slurm
srun hostname | sort -u > node_list
head -n 1 node_list > machinefile.$SLURM_JOBID
cat node_list >> machinefile.$SLURM_JOBID
export SLURM_HOSTFILE=machinefile.$SLURM_JOBID

srun --ntasks 5 python calling_script.py
```

With this saved as `myscript.sh`, allocating, configuring, and running libEnsemble on Bebop is achieved by running

```
sbatch myscript.sh
```

Example submission scripts for running on Bebop in distributed and centralized mode are also given in the `examples`.

Debugging Strategies

View the status of your submitted jobs with `squeue`, and cancel jobs with `scancel <Job ID>`.

Additional Information

See the LCRC Bebop docs here for more information about Bebop.

4.8.2 Frontier

Frontier is an HPE Cray EX exascale system located at Oak Ridge Leadership Computing Facility (OLCF).

Each Frontier compute node contains one 64-core AMD EPYC and four AMD MI250X GPUs (eight logical GPUs).

Frontier uses the SLURM scheduler to submit jobs from login nodes to run on the compute nodes.

Installing libEnsemble

Begin by loading the `python` module:

```
module load cray-python
```

You may wish to create a virtual environment to install packages in (see python_on_frontier).

Example of using virtual environment

Having created a dir `/ccs/proj/<project_id>/libensemble`:

```
python -m venv /ccs/proj/<project_id>/libensemble/libe_env
source /ccs/proj/<project_id>/libensemble/libe_env/bin/activate
```

libEnsemble can be installed via pip:

```
pip install libensemble
```

See `advanced installation` for other installation options.

Example

Note that a video demonstration of this example is also available.

To run the `forces_gpu` tutorial on Frontier.

To obtain the example you can git clone libEnsemble - although only the forces sub-directory is needed:

```
git clone https://github.com/Libensemble/libensemble
cd libensemble/libensemble/tests/scaling_tests/forces/forces_app
```

To compile forces:

```
module load rocm
module load craype-accel-amd-gfx90a
cc -DGPU -I${ROCM_PATH}/include -L${ROCM_PATH}/lib -lamdhip64 -fopenmp -O3 -o forces.x␣
↪forces.c
```

Now go to forces_gpu directory:

```
cd ../forces_gpu
```

Now grab an interactive session on one node:

```
salloc --nodes=1 -A <project_id> --time=00:10:00
```

Then in the session run:

```
python run_libe_forces.py --comms local --nworkers 8
```

To see GPU usage, ssh into the node you are on in another window and run:

```
module load rocm
watch -n 0.1 rocm-smi
```

4.8.3 Perlmutter

Perlmutter is an HPE Cray "Shasta" system located at NERSC. Its compute nodes are equipped with four A100 NVIDIA GPUs.

It uses the SLURM scheduler to submit jobs from login nodes to run on the compute nodes.

Configuring Python and Installation

Begin by loading the `python` module. The following modules are recommended:

```
module load python
```

Create a conda environment

You can create a conda environment in which to install libEnsemble and all dependencies. For example:

```
conda create -n libe-pm python=3.9 -y
```

As Perlmutter has a shared HOME filesystem with other clusters, using the `-pm` suffix (for Perlmutter) is good practice.

Activate your virtual environment with:

```
export PYTHONNOUSERSITE=1
conda activate libe-pm
```

Installing libEnsemble and dependencies

Having loaded the Anaconda Python module, libEnsemble can be installed by one of the following ways.

1. Install via **pip** into the environment.

```
(my_env) user@perlmutter07:~$ pip install libensemble
```

2. Install via **conda**:

```
(my_env) user@perlmutter07:~$ conda config --add channels conda-forge
(my_env) user@perlmutter07:~$ conda install -c conda-forge libensemble
```

See `advanced installation` for other installation options.

Job Submission

Perlmutter uses Slurm for job submission and management. The two most common commands for initiating jobs are `salloc` and `sbatch` for running in interactive and batch modes, respectively. libEnsemble runs on the compute nodes on Perlmutter using either `multi-processing` (recommended) or `mpi4py`.

Example

To run the `forces_gpu` tutorial on Perlmutter.

To obtain the example you can git clone libEnsemble - although only the forces sub-directory is needed:

```
git clone https://github.com/Libensemble/libensemble
cd libensemble/libensemble/tests/scaling_tests/forces/forces_app
```

To compile forces:

```
module load PrgEnv-nvidia cudatoolkit craype-accel-nvidia80
cc -DGPU -O3 -fopenmp -mp=gpu -target-accel=nvidia80 -o forces.x forces.c
```

Now go to forces_gpu directory:

```
cd ../forces_gpu
```

Now grab an interactive session on one node:

```
salloc -N 1 -t 20 -C gpu -q interactive -A <project_id>
```

Then in the session run:

```
python run_libe_forces.py --comms local --nworkers 4
```

To see GPU usage, ssh into the node you are on in another window and run:

```
watch -n 0.1 nvidia-smi
```

To watch video

There is a video demonstration of the forces example on Perlmutter.

Note: The video uses libEnsemble version 0.9.3, where some adjustments of the scripts are needed to run on Perlmutter. These adjustments are no longer necessary. libEnsemble now correctly detects MPI runner and GPU setting on Perlmutter and the GPU code runs with many more particles than the CPU version (forces_simple).

Example submission scripts are also given in the `examples`.

Running libEnsemble with mpi4py

Running libEnsemble with `local` comms is usually sufficient on Perlmutter. However, if you need to use `mpi4py`, you should install and run as follows:

```
module load PrgEnv-gnu cudatoolkit
MPICC="cc -target-accel=nvidia80 -shared" pip install --force --no-cache-dir --no-
↪binary=mpi4py mpi4py
```

This line will build `mpi4py` on top of a CUDA-aware Cray MPICH.

To run using 4 workers (one manager):

```
export SLURM_EXACT=1
srun -n 5 python my_script.py
```

More information on using Python and `mpi4py` on Perlmutter can be found in the Python on Perlmutter documentation.

Perlmutter FAQ

Some FAQs specific to Perlmutter. See more on the FAQ page.

srun: Job ****** step creation temporarily disabled, retrying (Requested nodes are busy)

Having created a dir `/ccs/proj/<project_id>/libensemble`:

You may also see: `srun: Job ****** step creation still disabled, retrying (Requested nodes are busy)`

This error has been encountered on Perlmutter. It is recommended to add these lines to submission scripts:

```
export SLURM_EXACT=1
export SLURM_MEM_PER_NODE=0
```

and to **avoid** using `#SBATCH` commands that may limit resources to srun job steps such as:

```
#SBATCH --ntasks-per-node=4
#SBATCH --gpus-per-task=1
```

Instead provide these to sub-tasks via the `extra_args` option to the `MPIExecutor` `submit` function.

GTL_DEBUG: [0] cudaHostRegister: no CUDA-capable device is detected

If using the environment variable `MPICH_GPU_SUPPORT_ENABLED`, then `srun` commands, at time of writing, expect an option for allocating GPUs (e.g.~ `--gpus-per-task=1` would allocate one GPU to each MPI task of the MPI run). It is recommended that tasks submitted via the `MPIExecutor` specify this in the `extra_args` option to the `submit` function (rather than using an `#SBATCH` command). This is needed even when using setting `CUDA_VISIBLE_DEVICES` or other options.

If running the libEnsemble user calling script with `srun`, then it is recommended that `MPICH_GPU_SUPPORT_ENABLED` is set in the user `sim_f` or `gen_f` function where GPU runs will be submitted, instead of in the batch script. E.g:

```
os.environ["MPICH_GPU_SUPPORT_ENABLED"] = "1"
```

warning: /tmp/pgcudafatYDO6wtSva6K2.o: missing .note.GNU-stack section implies executable stack

This warning has been recently encountered when compiling the forces example on Perlmutter. This does not affect the run, but can be suppressed by adding `-Wl,-znoexecstack` to the build line.

Additional Information

See the NERSC Perlmutter docs for more information about Perlmutter.

4.8.4 Polaris

Polaris is a 560-node HPE system located in the ALCF at Argonne National Laboratory. The compute nodes are equipped with one AMD EPYC Milan processor and four A100 NVIDIA GPUs. It uses the PBS scheduler to submit jobs from login nodes to run on the compute nodes.

Configuring Python and Installation

Python and libEnsemble are available on Polaris with the *conda* module. Load the `conda` module and activate the base environment:

```
module load conda
conda activate base
```

This also gives you access to machine-optimized packages such as mpi4py.

To install further packages, including updating libEnsemble, you may either create a virtual environment on top of this (if just using `pip install`) or clone the base environment (if you need `conda install`). More details at Python for Polaris.

Example of Conda + virtual environment

To create a virtual environment that allows installation of further packages:

```
python -m venv /path/to-venv --system-site-packages
. /path/to-venv/bin/activate
```

where `/path/to-venv` can be anywhere you have write access. For future sessions, just load the `conda` module and run the activate line.

You can now pip install libEnsemble:

```
pip install libensemble
```

See here for more information on advanced options for installing libEnsemble, including using Spack.

Ensuring use of mpiexec

Prior to libE v 0.10.0, when using the `MPIExecutor` it is necessary to manually tell libEnsemble to use``mpiexec`` instead of `aprun`. When setting up the executor use:

```
from libensemble.executors.mpi_executor import MPIExecutor
exctr = MPIExecutor(custom_info={'mpi_runner':'mpich', 'runner_name':'mpiexec'})
```

From version 0.10.0, this is not necessary.

Job Submission

Polaris uses the PBS scheduler to submit jobs from login nodes to run on the compute nodes. libEnsemble runs on the compute nodes using either `multi-processing` or `mpi4py`

A simple example batch script for a libEnsemble use case that runs 5 workers (e.g., one persistent generator and four for simulations) on one node:

```
#!/bin/bash
#PBS -A <myproject>
#PBS -lwalltime=00:15:00
#PBS -lselect=1
#PBS -q debug
#PBS -lsystem=polaris
#PBS -lfilesystems=home:grand

export MPICH_GPU_SUPPORT_ENABLED=1

cd $PBS_O_WORKDIR

python run_libe_forces.py --comms local --nworkers 5
```

The script can be run with:

```
qsub submit_libe.sh
```

Or you can run an interactive session with:

```
qsub -A <myproject> -l select=1 -l walltime=15:00 -lfilesystems=home:grand -qdebug -I
```

You may need to reload your `conda` module and reactivate `venv` environment again after starting the interactive session.

Demonstration

For an example that runs a small ensemble using a C application (offloading work to the GPU), see the `forces_gpu` tutorial. A video demonstration of this example is also available.

4.8.5 Spock/Crusher

Spock and Crusher are early-access testbed systems located at Oak Ridge Leadership Computing Facility (OLCF).

Each Spock compute node consists of one 64-core AMD EPYC "Rome" CPU and four AMD MI100 GPUs.

Each Crusher compute node contains a 64-core AMD EPYC and 4 AMD MI250X GPUs (8 Graphics Compute Dies).

These systems use the SLURM scheduler to submit jobs from login nodes to run on the compute nodes.

Configuring Python and Installation

Begin by loading the `python` module:

```
module load cray-python
```

Job Submission

Slurm is used for job submission and management. libEnsemble runs on the compute nodes using either `multi-processing` or `mpi4py`.

If running more than one worker per node, the following is recommended to prevent resource conflicts:

```
export SLURM_EXACT=1
export SLURM_MEM_PER_NODE=0
```

Installing libEnsemble and dependencies

libEnsemble can be installed via pip:

```
pip install libensemble
```

Example

To run the `forces_gpu` tutorial on Spock or Crusher.

To obtain the example you can git clone libEnsemble - although only the forces sub-directory is needed:

```
git clone https://github.com/Libensemble/libensemble
cd libensemble/libensemble/tests/scaling_tests/forces/forces_app
```

To compile forces (in addition to cray-python module):

```
module load rocm
module load craype-accel-amd-gfx90a # (craype-accel-amd-gfx908 on Spock)
cc -DGPU -I${ROCM_PATH}/include -L${ROCM_PATH}/lib -lamdhip64 -fopenmp -O3 -o forces.x␣
↪forces.c
```

Now go to forces_gpu directory:

```
cd ../forces_gpu
```

Now grab an interactive session on one node:

```
salloc --nodes=1 -A <project_id> --time=00:10:00
```

Then in the session run:

```
python run_libe_forces.py --comms local --nworkers 4
```

To see GPU usage, ssh into the node you are on in another window and run:

```
module load rocm
watch -n 0.1 rocm-smi
```

4.8.6 Summit

Summit is an IBM AC922 system located at the Oak Ridge Leadership Computing Facility (OLCF). Each of the approximately 4,600 compute nodes on Summit contains two IBM POWER9 processors and six NVIDIA Volta V100 accelerators.

Summit features three tiers of nodes: login, launch, and compute nodes.

Users on login nodes submit batch runs to the launch nodes. Batch scripts and interactive sessions run on the launch nodes. Only the launch nodes can submit MPI runs to the compute nodes via `jsrun`.

Configuring Python

Begin by loading the Python 3 Anaconda module:

```
$ module load python
```

You can now create and activate your own custom conda environment:

```
conda create --name myenv python=3.9
export PYTHONNOUSERSITE=1 # Make sure get python from conda env
. activate myenv
```

If you are installing any packages with extensions, ensure that the correct compiler module is loaded. If using mpi4py, this must be installed from source, referencing the compiler. Currently, mpi4py must be built with gcc:

```
module load gcc
```

With your environment activated, run

```
CC=mpicc MPICC=mpicc pip install mpi4py --no-binary mpi4py
```

Installing libEnsemble

Obtaining libEnsemble is now as simple as `pip install libensemble`. Your prompt should be similar to the following line:

```
(my_env) user@login5:~$ pip install libensemble
```

Note: If you encounter pip errors, run `python -m pip install --upgrade pip` first

Or, you can install via `conda`:

```
(my_env) user@login5:~$ conda config --add channels conda-forge
(my_env) user@login5:~$ conda install -c conda-forge libensemble
```

See here for more information on advanced options for installing libEnsemble.

Special note on resource sets and Executor submit options

When using the portable MPI run configuration options (e.g., num_nodes) to the `MPIExecutor` `submit` function, it is important to note that, due to the resource sets used on Summit, the options refer to resource sets as follows:

- num_procs (int, optional) – The total number resource sets for this run.
- num_nodes (int, optional) – The number of nodes on which to submit the run.
- procs_per_node (int, optional) – The number of resource sets per node.

It is recommended that the user defines a resource set as the minimal configuration of CPU cores/processes and GPUs. These can be added to the `extra_args` option of the *submit* function. Alternatively, the portable options can be ignored and everything expressed in `extra_args`.

For example, the following *jsrun* line would run three resource sets, each having one core (with one process), and one GPU, along with some extra options:

```
jsrun -n 3 -a 1 -g 1 -c 1 --bind=packed:1 --smpiargs="-gpu"
```

To express this line in the `submit` function may look something like the following:

```
exctr = Executor.executor
task = exctr.submit(app_name="mycode",
                    num_procs=3,
                    extra_args="-a 1 -g 1 -c 1 --bind=packed:1 --smpiargs="-gpu""
                    app_args="-i input")
```

This would be equivalent to:

```
exctr = Executor.executor
task = exctr.submit(app_name="mycode",
                    extra_args="-n 3 -a 1 -g 1 -c 1 --bind=packed:1 --smpiargs="-gpu""
                    app_args="-i input")
```

The libEnsemble resource manager works out the resources available to each worker, but unlike some other systems, `jsrun` on Summit dynamically schedules runs to available slots across and within nodes. It can also queue tasks. This allows variable size runs to easily be handled on Summit. If oversubsciption to the *jsrun* system is desired, then libEnsemble's resource manager can be disabled in the calling script via:

```
libE_specs["disable_resource_manager"] = True
```

In the above example, the task being submitted used three GPUs, which is half those available on a Summit node, and thus two such tasks may be allocated to each node (from different workers), if they were running at the same time.

Job Submission

Summit uses LSF for job management and submission. For libEnsemble, the most important command is `bsub` for submitting batch scripts from the login nodes to execute on the launch nodes.

It is recommended to run libEnsemble on the launch nodes (assuming workers are submitting MPI applications) using the `local` communications mode (multiprocessing). In the future, Balsam may be used to run libEnsemble on compute nodes.

Interactive Runs

You can run interactively with `bsub` by specifying the `-Is` flag, similarly to the following:

```
$ bsub -W 30 -P [project] -nnodes 8 -Is
```

This will place you on a launch node.

Note: You will need to reactivate your conda virtual environment.

Batch Runs

Batch scripts specify run settings using `#BSUB` statements. The following simple example depicts configuring and launching libEnsemble to a launch node with multiprocessing. This script also assumes the user is using the `parse_args()` convenience function from libEnsemble's `tools module`.

```
#!/bin/bash -x
#BSUB -P <project code>
#BSUB -J libe_mproc
#BSUB -W 60
#BSUB -nnodes 128
#BSUB -alloc_flags "smt1"

# --- Prepare Python ---

# Load conda module and gcc.
module load python
module load gcc

# Name of conda environment
export CONDA_ENV_NAME=my_env

# Activate conda environment
export PYTHONNOUSERSITE=1
source activate $CONDA_ENV_NAME

# --- Prepare libEnsemble ---

# Name of calling script
export EXE=calling_script.py

# Communication Method
export COMMS="--comms local"

# Number of workers.
export NWORKERS="--nworkers 128"

hash -r # Check no commands hashed (pip/python...)

# Launch libE
python $EXE $COMMS $NWORKERS > out.txt 2>&1
```

With this saved as `myscript.sh`, allocating, configuring, and queueing libEnsemble on Summit is achieved by running

```
$ bsub myscript.sh
```

Example submission scripts are also given in the `examples`.

Launching User Applications from libEnsemble Workers

Only the launch nodes can submit MPI runs to the compute nodes via `jsrun`. This can be accomplished in user `sim_f` functions directly. However, it is highly recommended that the `Executor` interface be used inside the `sim_f` or `gen_f`, because this provides a portable interface with many advantages including automatic resource detection, portability, launch failure resilience, and ease of use.

Additional Information

See the OLCF guides for more information about Summit.

4.8.7 Theta

Theta is a Cray XC40 system based on the second-generation Intel Xeon Phi processor, available in the ALCF at Argonne National Laboratory.

Theta features three tiers of nodes: login, MOM, and compute nodes. Users on login nodes submit batch jobs to the MOM nodes. MOM nodes execute user batch scripts to run on the compute nodes via `aprun`.

Theta will not schedule more than one MPI application per compute node.

Configuring Python

Begin by loading the Python 3 Miniconda module:

```
$ module load miniconda-3/latest
```

Create a conda virtual environment. We recommend cloning the base environment. This environment will contain mpi4py and many other packages that are configured correctly for Theta:

```
$ conda create --name my_env --clone $CONDA_PREFIX
```

Note: The "executing transaction" step of creating your new environment may take a while!

Following a successful environment creation, the prompt will suggest activating your new environment immediately. A conda error may result; follow the on-screen instructions to configure your shell with conda.

Activate your virtual environment with

```
$ export PYTHONNOUSERSITE=1
$ conda activate my_env
```

Alternative

If you do not wish to clone the miniconda environment and instead create your own, and you are using `mpi4py` make sure the install picks up Cray's compiler drivers. E.g:

```
$ conda create --name my_env python=3.9
$ export PYTHONNOUSERSITE=1
$ conda activate my_env
$ CC=cc MPICC=cc pip install mpi4py --no-binary mpi4py
```

More information on using conda on Theta is also available.

Installing libEnsemble and Balsam

libEnsemble

You should get an indication that your virtual environment is activated. Obtaining libEnsemble is now as simple as `pip install libensemble`. Your prompt should be similar to the following line:

```
(my_env) user@thetalogin6:~$ pip install libensemble
```

Note: If you encounter pip errors, run `python -m pip install --upgrade pip` first.

Or, you can install via `conda` (which comes with some common dependencies):

```
(my_env) user@thetalogin6:~$ conda config --add channels conda-forge
(my_env) user@thetalogin6:~$ conda install -c conda-forge libensemble
```

See here for more information on advanced options for installing libEnsemble.

Balsam (Optional)

Balsam allows libEnsemble to be run on compute nodes, and still submit tasks from workers (see Job Submission below). The Balsam Executor can submit tasks to the Balsam Service, which will submit these tasks dynamically to a corresponding Balsam Site.

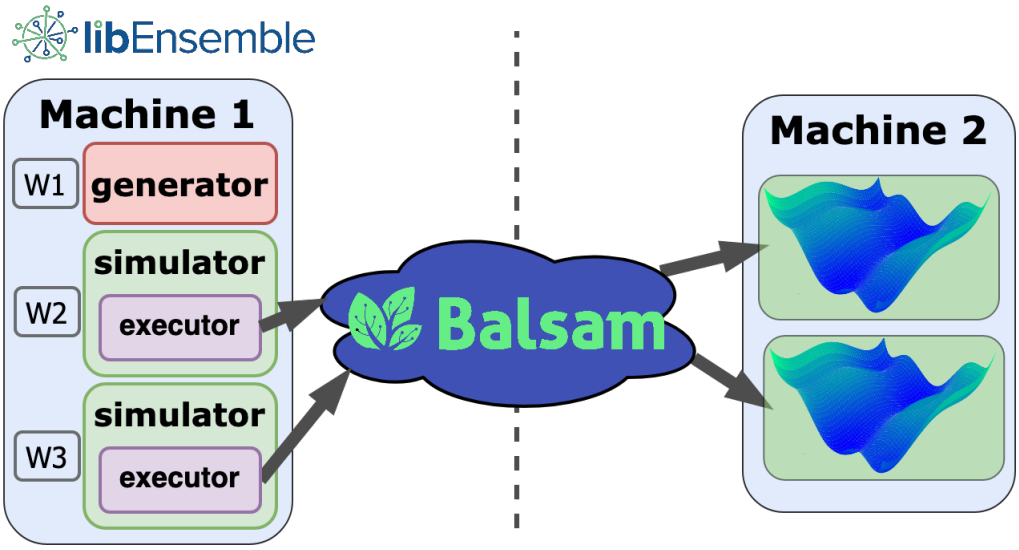
See the `Balsam Executor` docs for more information.

Job Submission

On Theta, libEnsemble can be launched to two locations:

1. **A MOM Node**: All of libEnsemble's manager and worker processes run centrally on a front-end MOM node. libEnsemble's MPI Executor takes responsibility for direct user-application submission to allocated compute nodes. libEnsemble must be configured to run with *multiprocessing* communications, since mpi4py isn't configured for use on the MOM nodes.

1. **The Compute Nodes**: libEnsemble is submitted to Balsam, and all manager and worker processes are tasked to a back-end compute node and run centrally. libEnsemble's Balsam Executor interfaces with the Balsam service for dynamic user-application submission to the compute nodes.



When considering on which nodes to run libEnsemble, consider whether your `sim_f` or `gen_f` user functions (not applications) execute computationally expensive code, or code built specifically for the compute node architecture. Recall also that only the MOM nodes can launch MPI applications.

Although libEnsemble workers on the MOM nodes can technically submit user applications to the compute nodes directly via `aprun` within user functions, it is highly recommended that the aforementioned `executor` interface be used instead. The libEnsemble Executor features advantages such as automatic resource detection, portability, launch failure resilience, and ease of use.

Theta features one default production queue, `default`, and two debug queues, `debug-cache-quad` and `debug-flat-quad`.

Note: For the default queue, the minimum number of nodes to allocate at once is 128.

Module and environment variables

In order to ensure proper functioning of libEnsemble, including the ability to kill running tasks, the following environment variable should be set:

```
export PMI_NO_FORK=1
```

It is also recommended that the following environment modules be unloaded, if present:

```
module unload trackdeps
module unload darshan
module unload xalt
```

Interactive Runs

You can run interactively with `qsub` by specifying the `-I` flag, similarly to the following:

```
$ qsub -A [project] -n 8 -q debug-cache-quad -t 60 -I
```

This will place you on a MOM node. Then, to launch jobs to the compute nodes, use `aprun` where you would use `mpirun`.

Note: You will need to reactivate your conda virtual environment. Configuring this routine to occur automatically is recommended.

Batch Runs

Batch scripts specify run settings using `#COBALT` statements. The following simple example depicts configuring and launching libEnsemble to a MOM node with multiprocessing. This script also assumes the user is using the `parse_args()` convenience function from libEnsemble's `tools module`.

```
#!/bin/bash -x
#COBALT -t 02:00:00
#COBALT -n 128
#COBALT -q default
#COBALT -A [project]
#COBALT -O libE-project

# --- Prepare Python ---

# Obtain Conda PATH from miniconda-3/latest module
CONDA_DIR=/soft/datascience/conda/miniconda3/latest/bin

# Name of conda environment
export CONDA_ENV_NAME=my_env

# Activate conda environment
export PYTHONNOUSERSITE=1
source $CONDA_DIR/activate $CONDA_ENV_NAME

# --- Prepare libEnsemble ---

# Name of calling script
export EXE=calling_script.py

# Communication Method
export COMMS="--comms local"

# Number of workers.
export NWORKERS="--nworkers 128"

# Required for killing tasks from workers on Theta
export PMI_NO_FORK=1
```

(continues on next page)

(continued from previous page)

```
# Unload Theta modules that may interfere with task monitoring/kills
module unload trackdeps
module unload darshan
module unload xalt

python $EXE $COMMS $NWORKERS > out.txt 2>&1
```

With this saved as `myscript.sh`, allocating, configuring, and queueing libEnsemble on Theta is achieved by running

```
$ qsub --mode script myscript.sh
```

Debugging Strategies

View the status of your submitted jobs with `qstat -fu [user]`.

Theta features two debug queues each with sixteen nodes. Each user can allocate up to eight nodes at once for a maximum of one hour. To allocate nodes on a debug queue interactively, use

```
$ qsub -A [project] -n 4 -q debug-flat-quad -t 60 -I
```

Additional Information

See the ALCF Support Center for more information about Theta.

Read the documentation for Balsam here.

4.8.8 libEnsemble with SLURM

SLURM is a popular open-source workload manager.

libEnsemble can read SLURM node lists and partition these to workers. By default this is done by `reading an environment variable`.

Example SLURM submission scripts for various systems are given in the `examples`. Further examples are given in some of the specific platform guides (e.g., `Perlmutter guide`)

By default, the `MPIExecutor` uses `mpirun` as a priority over `srun` as it works better in some cases. If `mpirun` does not work well, then try telling the MPIExecutor to use `srun` when it is initiated in the calling script:

```
from libensemble.executors.mpi_executor import MPIExecutor
exctr = MPIExecutor(custom_info={"mpi_runner":"srun"})
```

Common Errors

SLURM systems can have various configurations which may affect what is required when assigning more than one worker to any given node.

srun: Job ****** step creation temporarily disabled, retrying (Requested nodes are busy)

You may also see: `srun: Job ****** step creation still disabled, retrying (Requested nodes are busy)`

It is recommended to add these to submission scripts to prevent resource conflicts:

```
export SLURM_EXACT=1
export SLURM_MEM_PER_NODE=0
```

Alternatively, the `--exact` option to srun, along with other relevant options can be given on any `srun` lines, including the `MPIExecutor` submission lines via the `extra_args` option (from version 0.10.0, these are added automatically).

Secondly, while many configurations are possible, it is recommended to **avoid** using `#SBATCH` commands that may limit resources to srun job steps such as:

```
#SBATCH --ntasks-per-node=4
#SBATCH --gpus-per-task=1
```

Instead provide these to sub-tasks via the `extra_args` option to the `MPIExecutor` `submit` function.

GTL_DEBUG: [0] cudaHostRegister: no CUDA-capable device is detected

If using the environment variable `MPICH_GPU_SUPPORT_ENABLED`, then `srun` commands may expect an option for allocating GPUs (e.g., `--gpus-per-task=1` would allocate one GPU to each MPI task of the MPI run). It is recommended that tasks submitted via the `MPIExecutor` specify this in the `extra_args` option to the `submit` function (rather than using an `#SBATCH` command).

If running the libEnsemble calling script with `srun`, then it is recommended that `MPICH_GPU_SUPPORT_ENABLED` is set in the user `sim_f` or `gen_f` function where GPU runs will be submitted, instead of in the batch script. For example:

```
os.environ["MPICH_GPU_SUPPORT_ENABLED"] = "1"
```

Note on Resource Binding

Note: Update: From version 0.10.0, it is recommended that GPUs are assigned automatically by libEnsemble. See the `forces_gpu` tutorial as an example.

Note that the use of `CUDA_VISIBLE_DEVICES` and other environment variables is often a highly portable way of assigning specific GPUs to workers, and has been known to work on some systems when other methods do not. See the libEnsemble regression test test_persistent_sampling_CUDA_variable_resources.py for an example of setting CUDA_VISIBLE_DEVICES in the imported simulator function (`CUDA_variable_resources`).

On other systems, like Perlmutter, using an option such as `--gpus-per-task=1` or `-gres=gpu:1` in `extra_args` is sufficient to allow SLURM to find the free GPUs.

Note that the `srun` options such as:

```
--gpu-bind=map_gpu:2,3
```

do not necessarily provide absolute GPU slots when there are more than one concurrent job steps (`sruns`) running on a node. If desired, such options could be set using the `worker resources` module in a similar manner to how `CUDA_VISIBLE_DEVICES` is set in the example.

Some useful commands

Find SLURM version:

```
scontrol --version
```

Find SLURM system configuration:

```
scontrol show config
```

Find SLURM partition configuration for a partition called "gpu":

```
scontrol show partition gpu
```

4.8.9 Example Scheduler Submission Scripts

Below are example submission scripts used to configure and launch libEnsemble on a variety of high-powered systems. See `here` for more information about the respective systems and configuration.

Alternatively to interacting with the scheduler or configuring submission scripts, libEnsemble now features a portable set of `command-line utilities` for submitting workflows to almost any system or scheduler.

Slurm - Basic

Listing 1: /examples/libE_submission_scripts/submit_slurm_simple.sh

```
#!/bin/bash
#SBATCH -J libE_simple
#SBATCH -A <myproject>
#SBATCH -p <partition_name>
#SBATCH -C <constraint_name>
#SBATCH --time 10
#SBATCH --nodes 2

# Usually either -p or -C above is used.

# On some SLURM configurations, these ensure runs can share nodes
export SLURM_EXACT=1
export SLURM_MEM_PER_NODE=0

python libe_calling_script.py --comms local --nworkers 8
```

Bridges - Central Mode

Listing 2: /examples/libE_submission_scripts/bridges_submit_slurm_central.sh

```
#!/bin/bash
#SBATCH -J libE_test_central
#SBATCH -N 5
#SBATCH -p RM
#SBATCH -A <my_project>
#SBATCH -o tlib.%j.%N.out
#SBATCH -e tlib.%j.%N.error
#SBATCH -t 00:30:00

# Launch script for running in central mode with mpi4py.
#   libEnsemble will run on a dedicated node (or nodes).
#   The remaining nodes in the allocation will be dedicated to worker launched apps.
#   Initialize Executor with auto-resources=True and central_mode=True.

# User to edit these variables
export EXE=libE_calling_script.py
export NUM_WORKERS=4

mpirun -np $(($NUM_WORKERS+1)) -ppn $(($NUM_WORKERS+1)) python $EXE

# To use local mode instead of mpi4py (with parse_args())
# python $EXE --comms local --nworkers $NUM_WORKERS
```

Bebop - Central Mode

Listing 3: /examples/libE_submission_scripts/bebop_submit_slurm_central.sh

```
#!/bin/bash
#SBATCH -J libE_test_central
#SBATCH -N 5
#SBATCH -p knlall
#SBATCH -A <my_project>
#SBATCH -o tlib.%j.%N.out
#SBATCH -e tlib.%j.%N.error
#SBATCH -t 01:00:00

# Launch script for running in central mode with mpi4py.
#   libEnsemble will run on a dedicated node (or nodes).
#   The remaining nodes in the allocation will be dedicated to worker launched apps.
#   Use executor with auto-resources=True and central_mode=True.

# User to edit these variables
export EXE=libE_calling_script.py
export NUM_WORKERS=4
export I_MPI_FABRICS=shm:tmi

# Overcommit will allow ntasks up to the no. of contexts on one node (eg. 320 on Bebop)
srun --overcommit --ntasks=$(($NUM_WORKERS+1)) --nodes=1 python $EXE
```

(continues on next page)

(continued from previous page)

```
# To use local mode instead of mpi4py (with parse_args())
# python calling_script.py --comms local --nworkers $NUM_WORKERS
```

Bebop - Distributed Mode

Listing 4: /examples/libE_submission_scripts/bebop_submit_slurm_distrib.sh

```
#!/bin/bash
#SBATCH -J libE_test
#SBATCH -N 4
#SBATCH -p knlall
#SBATCH -A <my_project>
#SBATCH -o tlib.%j.%N.out
#SBATCH -e tlib.%j.%N.error
#SBATCH -t 01:00:00

# Launch script that runs in distributed mode with mpi4py.
#   Workers are evenly spread over nodes and manager added to the first node.
#   Requires even distribution - either multiple workers per node or nodes per worker
#   Option for manager to have a dedicated node.
#   Use of MPI Executor will ensure workers co-locate tasks with workers
#   If node_list file is kept, this informs libe of resources. Else, libe auto-detects.

# User to edit these variables
export EXE=libE_calling_script.py
export NUM_WORKERS=4
export MANAGER_NODE=false # true = Manager has a dedicated node (assign one extra)
export USE_NODE_LIST=true # If false, allow libE to determine node_list from environment.

# As libE shares nodes with user applications allow fallback if contexts overrun.
unset I_MPI_FABRICS
export I_MPI_FABRICS_LIST=tmi,tcp
export I_MPI_FALLBACK=1

# If using in calling script (After N mins manager kills workers and exits cleanly)
export LIBE_WALLCLOCK=55

#-------------------------------------------------------------------------------
# Work out distribution
if [[ $MANAGER_NODE = "true" ]]; then
  WORKER_NODES=$(($SLURM_NNODES-1))
else
  WORKER_NODES=$SLURM_NNODES
fi

if [[ $NUM_WORKERS -ge $WORKER_NODES ]]; then
  SUB_NODE_WORKERS=true
  WORKERS_PER_NODE=$(($NUM_WORKERS/$WORKER_NODES))
else
  SUB_NODE_WORKERS=false
```

(continues on next page)

(continued from previous page)

```
  NODES_PER_WORKER=$(($WORKER_NODES/$NUM_WORKERS))
fi;
#-------------------------------------------------------------------------------

# A little useful information
echo -e "Manager process running on: $HOSTNAME"
echo -e "Directory is:  $PWD"

# Generate a node list with 1 node per line:
srun hostname | sort -u > node_list

# Add manager node to machinefile
head -n 1 node_list > machinefile.$SLURM_JOBID

# Add worker nodes to machinefile
if [[ $SUB_NODE_WORKERS = "true" ]]; then
  awk -v repeat=$WORKERS_PER_NODE '{for(i=0; i<repeat; i++)print}' node_list \
  >>machinefile.$SLURM_JOBID
else
  awk -v patt="$NODES_PER_WORKER" 'NR % patt == 1' node_list \
  >> machinefile.$SLURM_JOBID
fi;

if [[ $USE_NODE_LIST = "false" ]]; then
  rm node_list
  wait
fi;

# Put in a timestamp
echo Starting execution at: `date`

# To use srun
export SLURM_HOSTFILE=machinefile.$SLURM_JOBID

# The "arbitrary" flag should ensure SLURM_HOSTFILE is picked up
# cmd="srun --ntasks $(($NUM_WORKERS+1)) -m arbitrary python $EXE"
cmd="srun --ntasks $(($NUM_WORKERS+1)) -m arbitrary python $EXE $LIBE_WALLCLOCK"

echo The command is: $cmd
echo End PBS script information.
echo All further output is from the process being run and not the script.\n\n $cmd

$cmd

# Print the date again -- when finished
echo Finished at: `date`
```

Theta - On MOM Node with Multiprocessing

Listing 5: /examples/libE_submission_scripts/theta_submit_mproc.sh

```
#!/bin/bash -x
#COBALT -t 00:30:00
#COBALT -O libE_mproc_MOM
#COBALT -n 4
#COBALT -q debug-flat-quad # Up to 8 nodes only
##COBALT -q default # For large jobs >=128 nodes
##COBALT -A <project code>

# Script to run libEnsemble using multiprocessing on launch nodes.
# Assumes Conda environment is set up.

# To be run with central job management
# - Manager and workers run on launch node.
# - Workers submit tasks to the compute nodes in the allocation.

# Name of calling script
export EXE=libE_calling_script.py

# Communication Method
export COMMS="--comms local"

# Number of workers.
export NWORKERS="--nworkers 4"

# Wallclock for libE (allow clean shutdown)
export LIBE_WALLCLOCK=25 # Optional if pass to script

# Name of Conda environment
export CONDA_ENV_NAME=<conda_env_name>

# Conda location - theta specific
export PATH=/opt/intel/python/2017.0.035/intelpython35/bin:$PATH
export LD_LIBRARY_PATH=~/.conda/envs/$CONDA_ENV_NAME/lib:$LD_LIBRARY_PATH
export PMI_NO_FORK=1 # Required for python kills on Theta

# Unload Theta modules that may interfere with job monitoring/kills
module unload trackdeps
module unload darshan
module unload xalt

# Activate conda environment
export PYTHONNOUSERSITE=1
. activate $CONDA_ENV_NAME

# Launch libE
# python $EXE $NUM_WORKERS > out.txt 2>&1  # No args. All defined in calling script
# python $EXE $COMMS $NWORKERS > out.txt 2>&1  # If calling script is using parse_args()
python $EXE $LIBE_WALLCLOCK $COMMS $NWORKERS > out.txt 2>&1 # If calling script takes␣
↪wall-clock as positional arg.
```

Summit - On Launch Nodes with Multiprocessing

Listing 6: /examples/libE_submission_scripts/summit_submit_mproc.sh

```
#!/bin/bash -x
#BSUB -P <project code>
#BSUB -J libe_mproc
#BSUB -W 30
#BSUB -nnodes 4
#BSUB -alloc_flags "smt1"

# Script to run libEnsemble using multiprocessing on launch nodes.
# Assumes Conda environment is set up.

# To be run with central job management
# - Manager and workers run on launch node.
# - Workers submit tasks to the compute nodes in the allocation.

# Name of calling script-
export EXE=libE_calling_script.py

# Communication Method
export COMMS="--comms local"

# Number of workers.
export NWORKERS="--nworkers 4"

# Wallclock for libE.  (allow clean shutdown)
export LIBE_WALLCLOCK=25 # Optional if pass to script

# Name of Conda environment
export CONDA_ENV_NAME=<conda_env_name>

# Need these if not already loaded
# module load python
# module load gcc/4.8.5

# Activate conda environment
export PYTHONNOUSERSITE=1
. activate $CONDA_ENV_NAME

# hash -d python # Check pick up python in conda env
hash -r # Check no commands hashed (pip/python...)

# Launch libE
# python $EXE $NUM_WORKERS > out.txt 2>&1  # No args. All defined in calling script
# python $EXE $COMMS $NWORKERS > out.txt 2>&1  # If calling script is using parse_args()
python $EXE $LIBE_WALLCLOCK $COMMS $NWORKERS > out.txt 2>&1 # If calling script takes␣
↪wall-clock as positional arg.
```

Chapter 5

Developer's Guide

This section contains further information for those interested in adding features to libEnsemble.

5.1 Contributing to libEnsemble

Contributions may be made via a GitHub pull request to

> https://github.com/Libensemble/libensemble

libEnsemble uses the Gitflow model. Contributors should branch from, and make pull requests to, the develop branch. The main branch is used only for releases. Pull requests may be made from a fork, for those without repository write access.

Code should pass flake8 tests, allowing for the exceptions given in the flake8 file in the project directory. Python code should be formatted using the latest version of black by running the following in the base libensemble directory:

```
black --config=.black .
```

Issues can be raised at

> https://github.com/Libensemble/libensemble/issues

Issues may include reporting bugs or suggested features. Administrators will add issues, as appropriate, to the project board at

> https://github.com/Libensemble/libensemble/projects

By convention, user branch names should have a <type>/<name> format, where example types are feature, bugfix, testing, docs, and experimental. Administrators may take a hotfix branch from the main, which will be merged into main (as a patch) and develop. Administrators may also take a release branch off develop and then merge this branch into main and develop for a release. Most branches should relate to an issue or feature.

When a branch closes a related issue, the pull request message should include the phrase "Closes #N," where N is the issue number. This will automatically close out the issues when they are pulled into the default branch (currently main).

libEnsemble is distributed under a 3-clause BSD license (see LICENSE). The act of submitting a pull request (with or without an explicit Signed-off-by tag) will be understood as an affirmation of the following:

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

5.2 Internal Modules

This section documents the internal modules of libEnsemble.

5.2.1 Manager Module

libEnsemble manager routines

manager.**manager_main**(*hist*, *libE_specs*, *alloc_specs*, *sim_specs*, *gen_specs*, *exit_criteria*, *persis_info*, *wcomms=[]*)

Manager routine to coordinate the generation and simulation evaluations

Parameters

- **hist** (libensemble.history.History) – A libEnsemble history type object.
- **libE_specs** (dict) – Specifications for libEnsemble
- **alloc_specs** (dict) – Specifications for the allocation function
- **sim_specs** (dict) – Specifications for the simulation function
- **gen_specs** (dict) – Specifications for the generator function
- **exit_criteria** (dict) – Criteria for libEnsemble to stop a run
- **persis_info** (dict) – Persistent information to be passed between user functions

- **wcomms** (list, Optional) – A list of comm type objects for each worker. Default is an empty list.

Return type
(<class 'dict'>, <class 'int'>, <class 'int'>)

class manager.**Manager**(*hist*, *libE_specs*, *alloc_specs*, *sim_specs*, *gen_specs*, *exit_criteria*, *wcomms=[]*)

Manager class for libensemble.

Parameters

- **hist** (*ndarray[Any, dtype[_ScalarType_co]]*) –
- **libE_specs** (*dict*) –
- **alloc_specs** (*dict*) –
- **sim_specs** (*dict*) –
- **gen_specs** (*dict*) –
- **exit_criteria** (*dict*) –
- **wcomms** (*list*) –

__init__(*hist*, *libE_specs*, *alloc_specs*, *sim_specs*, *gen_specs*, *exit_criteria*, *wcomms=[]*)

Initializes the manager

Parameters

- **hist** (*ndarray[Any, dtype[_ScalarType_co]]*) –
- **libE_specs** (*dict*) –
- **alloc_specs** (*dict*) –
- **sim_specs** (*dict*) –
- **gen_specs** (*dict*) –
- **exit_criteria** (*dict*) –
- **wcomms** (*list*) –

term_test_wallclock(*max_elapsed*)

Checks against wallclock timeout

Parameters
max_elapsed (*int*) –

Return type
bool

term_test_sim_max(*sim_max*)

Checks against max simulations

Parameters
sim_max (*int*) –

Return type
bool

term_test_gen_max(*gen_max*)

Checks against max generator calls

Parameters
gen_max (*int*) –

Return type
bool

term_test_stop_val(*stop_val*)

Checks against stop value criterion

Parameters
stop_val (*Any*) –

Return type
bool

term_test(*logged=True*)

Checks termination criteria

Parameters
logged (*bool*) –

Return type
bool | int

run(*persis_info*)

Runs the manager

Parameters
persis_info (*dict*) –

Return type
(<class 'dict'>, <class 'int'>, <class 'int'>)

5.2.2 Worker Module

libEnsemble worker class

worker.**worker_main**(*comm*, *sim_specs*, *gen_specs*, *libE_specs*, *workerID=None*, *log_comm=True*, *resources=None*, *executor=None*)

Evaluates calculations given to it by the manager.

Creates a worker object, receives work from manager, runs worker, and communicates results. This routine also creates and writes to the workers summary file.

Parameters

- **comm** (*communicator*) – Comm object for manager communications
- **sim_specs** (*dict*) – Parameters/information for simulation calculations
- **gen_specs** (*dict*) – Parameters/information for generation calculations
- **libE_specs** (*dict*) – Parameters/information for libE operations
- **workerID** (*int*) – Manager assigned worker ID (if None, default is comm.rank)
- **log_comm** (*bool*) – Whether to send logging over comm
- **resources** (*Resources*) –
- **executor** (*Executor*) –

Return type
None

class worker.**Worker**(*comm*, *dtypes*, *workerID*, *sim_specs*, *gen_specs*, *libE_specs*)

The worker class provides methods for controlling sim and gen funcs

Object Attributes:

These are public object attributes.

Variables

- **communicator** (*comm*) – Comm object for manager communications
- **dtypes** (*dict*) – Dictionary containing type information for sim and gen inputs
- **workerID** (*int*) – The libensemble Worker ID
- **sim_specs** (*dict*) – Parameters/information for simulation calculations
- **calc_iter** (*dict*) – Dictionary containing counts for each type of calc (e.g. sim or gen)

Parameters

- **comm** (*communicator*) –
- **dtypes** (*dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*) –
- **workerID** (*int*) –
- **sim_specs** (*dict*) –
- **gen_specs** (*dict*) –
- **libE_specs** (*dict*) –

__init__(*comm*, *dtypes*, *workerID*, *sim_specs*, *gen_specs*, *libE_specs*)

Initializes new worker object

Parameters

- **comm** (*communicator*) –
- **dtypes** (*dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*) –
- **workerID** (*int*) –
- **sim_specs** (*dict*) –
- **gen_specs** (*dict*) –
- **libE_specs** (*dict*) –

Return type
None

run()

Runs the main worker loop.

Return type
None

5.2.3 History Module

Note that this is the developer API reference for the internal history module. See history array for the user reference.

class libensemble.history.**History**(*alloc_specs*, *sim_specs*, *gen_specs*, *exit_criteria*, *H0*)

The History class provides methods for managing the history array.

Object Attributes:

These are set on initialization.

Variables

- **H** (*numpy.ndarray*) – History array storing rows for each point. Field names are in libensemble/tools/fields_keys.py. Numpy structured array.
- **offset** (*int*) – Starting index for this ensemble (after H0 read in)
- **index** (*int*) – Index where libEnsemble should start filling in H
- **sim_started_count** (*int*) – Number of points given to sim functions (according to H)
- **sim_ended_count** (*int*) – Number of points evaluated (according to H)

Parameters

- **alloc_specs** (*dict*) –
- **sim_specs** (*dict*) –
- **gen_specs** (*dict*) –
- **exit_criteria** (*dict*) –
- **H0** (*ndarray[Any, dtype[_ScalarType_co]]*) –

Note that index, sim_started_count and sim_ended_count reflect the total number of points in H and therefore include those prepended to H in addition to the current run.

__init__(*alloc_specs*, *sim_specs*, *gen_specs*, *exit_criteria*, *H0*)

Forms the numpy structured array that records everything from the libEnsemble run

Parameters

- **alloc_specs** (*dict*) –
- **sim_specs** (*dict*) –
- **gen_specs** (*dict*) –
- **exit_criteria** (*dict*) –
- **H0** (*ndarray[Any, dtype[_ScalarType_co]]*) –

Return type
None

update_history_f(*D*, *safe_mode*, *kill_canceled_sims=False*)

Updates the history after points have been evaluated

Parameters

- **D** (*dict*) –
- **safe_mode** (*bool*) –
- **kill_canceled_sims** (*bool*) –

Return type
None

update_history_x_out(*q_inds*, *sim_worker*, *kill_canceled_sims=False*)

Updates the history (in place) when new points have been given out to be evaluated

Parameters

- **q_inds** (*numpy.typing.NDArray*) – Row IDs for history array H
- **sim_worker** (*int*) – Worker ID
- **kill_canceled_sims** (*bool*) –

Return type
None

update_history_to_gen(*q_inds*)

Updates the history (in place) when points are given back to the gen

Parameters
q_inds (*ndarray[Any, dtype[_ScalarType_co]]*) –

update_history_x_in(*gen_worker*, *D*, *safe_mode*, *gen_started_time*)

Updates the history (in place) when new points have been returned from a gen

Parameters

- **gen_worker** (*int*) – The worker who generated these points
- **D** (*numpy.typing.NDArray*) – Output from gen_func
- **safe_mode** (*bool*) –
- **gen_started_time** (*int*) –

Return type
None

grow_H(*k*)

Adds k rows to H in response to gen_f producing more points than available rows in H.

Parameters
k (*int*) – Number of rows to add to H

Return type
None

trim_H()

Returns truncated array

Return type
ndarray[*Any*, *dtype*[*_ScalarType_co*]]

5.2.4 Resources Module

This module detects and returns system resources

class resources.resources.**Resources**(*libE_specs*, *platform_info={}*, *top_level_dir=None*)

Provides system resources to libEnsemble and executor.

A resources instance is always initialized unless `libE_specs["disable_resource_manager"]` is `True`.

Class Attributes:

Variables
Resources – resources: The resources object is stored here and can be retrieved in user functions.

Parameters

- **libE_specs** (*dict*) –
- **platform_info** (*dict*) –
- **top_level_dir** (*str*) –

Object Attributes:

These are set on initialization.

Variables

- **top_level_dir** (*string*) – Directory where searches for node_list file.
- **glob_resources** (*GlobalResources*) – Maintains resources available to libEnsemble.

Parameters

- **libE_specs** (*dict*) –
- **platform_info** (*dict*) –
- **top_level_dir** (*str*) –

The following are set up after manager/worker fork.

The resource manager is set up only on the manager, while the worker resources object is set up on workers.

Variables

- **resource_manager** (*ResourceManager*) – An object that manages resource set assignment to workers.
- **worker_resources** (*WorkerResources*) – An object that contains worker-specific resources.

Parameters

- **libE_specs** (*dict*) –
- **platform_info** (*dict*) –
- **top_level_dir** (*str*) –

__init__(*libE_specs*, *platform_info={}*, *top_level_dir=None*)

Initiate a new resources object

Parameters

- **libE_specs** (*dict*) –

- **platform_info** (*dict*) –
- **top_level_dir** (*str | None*) –

Return type
None

classmethod init_resources(*libE_specs*, *platform_info={}*)

Initiate resource management

Parameters

- **libE_specs** (*dict*) –
- **platform_info** (*dict*) –

Return type
None

set_worker_resources(*num_workers*, *workerid*)

Initiate the worker resources component of resources

Parameters

- **num_workers** (*int*) –
- **workerid** (*int*) –

Return type
None

set_resource_manager(*num_workers*)

Initiate the resource manager component of resources

Parameters
num_workers (*int*) –

Return type
None

add_comm_info(*libE_nodes*)

Adds comms-specific information to resources

Removes libEnsemble nodes from nodelist if in dedicated_mode.

Return type
None

class resources.resources.**GlobalResources**(*libE_specs*, *platform_info={}*, *top_level_dir=None*)

Object Attributes:

These are set on initialization. :ivar str top_level_dir: Directory where searches for node_list file :ivar EnvResources env_resources: Object storing environment variables used by resources :ivar list global_nodelist: list of all nodes available for running user applications :ivar int logical_cores_avail_per_node: Logical cores (including SMT threads) available on a node :ivar int physical_cores_avail_per_node: Physical cores available on a node :ivar list zero_resource_workers: List of workerIDs to have no resources. :ivar bool dedicated_mode: Whether to remove libE nodes from global nodelist. :ivar int num_resource_sets: Number of resource sets, if supplied by the user.

Parameters

- **libE_specs** (*dict*) –
- **platform_info** (*dict*) –

- **top_level_dir** (*str*) –

__init__(*libE_specs*, *platform_info={}*, *top_level_dir=None*)

Initializes a new Resources instance

Determines the compute resources available for current allocation, including node list and cores/hardware threads available within nodes.

The following parameters may be extracted from `libE_specs`

Parameters

- **top_level_dir** (*str, Optional*) – Directory libEnsemble runs in (default is current working directory)
- **dedicated_mode** (*bool, Optional*) – If true, then dedicate nodes to running libEnsemble. Dedicated mode means that any nodes running libE processes (manager and workers), will not be available to worker-launched tasks (user applications). They will be removed from the nodelist (if present), before dividing into resource sets.
- **zero_resource_workers** (*List[int], Optional*) – List of workers that require no resources.
- **num_resource_sets** (*int, Optional*) – The total number of resource sets. Resources will be divided into this number. Default: None. If None, resources will be divided by workers (excluding zero_resource_workers).
- **cores_on_node** (*tuple (int, int), Optional*) – If supplied gives (physical cores, logical cores) for the nodes. If not supplied, this will be auto-detected.
- **gpus_on_node** (*int, Optional*) – If supplied gives number of GPUs for the nodes. If not supplied, this will be auto-detected.
- **enforce_worker_core_bounds** (*bool, Optional*) – If True, then libEnsemble's executor will raise an exception if it detects that a worker has been instructed to launch tasks with the number of requested processes being excessive to the number of cores allocated to that worker, or not enough processes were requested to satisfy allocated cores.
- **node_file** (*str, Optional*) – If supplied, give the name of a file in the run directory to use as a node-list for use by libEnsemble. Defaults to a file named "node_list". If the file does not exist, then the node-list will be auto-detected.
- **nodelist_env_slurm** (*str, Optional*) – The environment variable giving a node list in Slurm format (Default: uses SLURM_NODELIST). Note: This is queried only if a node_list file is not provided.
- **nodelist_env_cobalt** (*str, Optional*) – The environment variable giving a node list in Cobalt format (Default: uses COBALT_PARTNAME). Note: This is queried only if a node_list file is not provided.
- **nodelist_env_lsf** (*str, Optional*) – The environment variable giving a node list in LSF format (Default: uses LSB_HOSTS). Note: This is queried only if a node_list file is not provided.
- **nodelist_env_lsf_shortform** (*str, Optional*) – The environment variable giving a node list in LSF short-form format (Default: uses LSB_MCPU_HOSTS) Note: This is only queried if a node_list file is not provided.
- **libE_specs** (*dict*) –
- **platform_info** (*dict*) –

Return type
None

add_comm_info(*libE_nodes*)

Adds comms-specific information to resources

Removes libEnsemble nodes from nodelist if in dedicated_mode.

update_scheduler_opts(*scheduler_opts*)

Add scheduler options from platform_info, if not present

static is_nodelist_shortnames(*nodelist*)

Returns False if any entry contains a '.', else True

static remove_nodes(*global_nodelist_in*, *remove_list*)

Removes any nodes in remove_list from the global nodelist

static get_global_nodelist(*node_file='node_list'*, *rundir=None*, *env_resources=None*)

Returns the list of nodes available to all libEnsemble workers.

If a node_file exists this is used, otherwise the environment is interrogated for a node list. If a dedicated manager node is used, then a node_file is recommended.

In dedicated mode, any node with a libE worker is removed from the list.

5.2.5 RSET Resources Module

class rset_resources.**RSetResources**(*num_workers*, *resources*)

A class that creates a fixed mapping of resource sets to the available resources.

Object Attributes:

These are set on initialisation and include inherited. `rsets` below is used to abbreviate `resource sets`.

Variables

- **num_workers** (*int*) – Total number of workers
- **num_workers_2assign2** (*int*) – The number of workers that will be assigned resource sets.
- **total_num_rsets** (*int*) – The total number of resource sets.
- **split_list** (*list*) – A list of lists, where each element is the list of nodes for a given rset.
- **local_rsets_list** (*list*) – A list over rsets, where each element is the number of rsets that share the node.
- **rsets_per_node** (*int*) – The number of rsets per node (if an rset > 1 node, this will be 1)

__init__(*num_workers*, *resources*)

Initializes a new RSetResources instance

Determines the compute resources available for each resource set.

Unless resource sets is set explicitly, the number of resource sets is the number of workers, excluding any workers defined as zero resource workers.

Parameters

- **num_workers** (*int*) – The total number of workers

- **resources** (*Resources*) – A Resources object containing global nodelist and intranode information

static get_group_list(*split_list*, *gpus_per_node=0*)

Return lists of group ids and slot IDs by resource set

static best_split(*a*, *n*)

Creates the most even split of list a into n parts and return list of lists

static get_rsets_on_a_node(*num_rsets*, *resources*)

Returns the number of resource sets that can be placed on each node

If there are more nodes than resource sets, returns 1.

static get_workers2assign2(*num_workers*, *resources*)

Returns workers to assign resources to

static even_assignment(*nnodes*, *nworkers*)

Returns True if workers are evenly distributied to nodes, else False

static expand_list(*nnodes*, *nworkers*, *nodelist*)

Duplicates each element of `nodelist` to best map workers to nodes.

Returns node list with duplicates, and a list of local (on-node) worker counts, both indexed by worker.

static get_split_list(*num_rsets*, *resources*)

Returns a list of lists for each worker

Assumes that self.global_nodelist has been calculated (in __init__).

static get_partitioned_nodelist(*num_rsets*, *resources*)

Returns lists of nodes available to all resource sets

Assumes that self.global_nodelist has been calculated (in __init__). Also self.global_nodelist will have already removed non-application nodes

5.2.6 Worker Resources Module

class resources.worker_resources.**ResourceManager**(*num_workers*, *resources*)

Bases: `RSetResources`

Provides methods for managing the assignment of resource sets to workers.

Parameters

- **num_workers** (*int*) –
- **resources** (*GlobalResources*) –

__init__(*num_workers*, *resources*)

Initializes a new ResourceManager instance

Instantiates the numpy structured array that holds information for each resource set.

Parameters

- **num_workers** (*int*) – The number of workers
- **resources** (*Resources*) – A Resources object containing global nodelist and intranode information

Return type
None

assign_rsets(*rset_team*, *worker_id*)

Mark the resource sets given by rset_team as assigned to worker_id

free_rsets(*worker=None*)

Free up assigned resource sets

static get_index_list(*num_workers*, *num_rsets*, *zero_resource_list*)

Map WorkerID to index into a nodelist

Parameters

- **num_workers** (*int*) –
- **num_rsets** (*int*) –
- **zero_resource_list** (*List[int | Any]*) –

Return type
List[int | None]

class resources.worker_resources.**WorkerResources**(*num_workers*, *resources*, *workerID*)

Bases: RSetResources

Provide system resources per worker to libEnsemble and executor.

Object Attributes:

Some of these attributes may be updated as the ensemble progresses.

rsets below is used to abbreviate resource sets.

Variables

- **workerID** (*int*) – workerID for this worker.
- **local_nodelist** (*list*) – A list of all nodes assigned to this worker.
- **rset_team** (*list*) – List of rset IDs currently assigned to this worker.
- **num_rsets** (*int*) – The number of resource sets assigned to this worker.
- **slots** (*dict*) – A dictionary with a list of slot IDs for each node.
- **even_slots** (*bool*) – True if each node has the same number of slots.
- **matching_slots** (*bool*) – True if each node has matching slot IDs.
- **slot_count** (*int*) – The number of slots per node if even_slots is True, else None.
- **slots_on_node** (*list*) – A list of slots IDs if matching_slots is True, else None.
- **local_node_count** (*int*) – The number of nodes available to this worker (rounded up to whole number).
- **rsets_per_node** (*int*) – The number of rsets per node (if a rset > 1 node, will be 1).

The worker_resources attributes can be queried, and convenience functions called, via the resources class attribute. For example:

With resources imported:

```
from libensemble.resources.resources import Resources
```

A user function (sim/gen) may do:

```
resources = Resources.resources.worker_resources
num_nodes = resources.local_node_count
cores_per_node = resources.slot_count  # One CPU per GPU
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")  # Use convenience function.
```

Note that **slots** are resource sets enumerated on a node (starting with zero). If a resource set has more than one node, then each node is considered to have slot zero.

If `even_slots` is True, then the attributes `slot_count` will give the number of slots on each node. If `matching_slots` is True, then `slots_on_node` will give the slot IDs for all nodes. These can be used for simplicity; otherwise, the `slots` dictionary can be used to get information for each node.

__init__(*num_workers*, *resources*, *workerID*)

Initializes a new WorkerResources instance

Determines the compute resources available for current worker, including node list and cores/hardware threads available within nodes.

Parameters

- **num_workers** (*int*) – The number of workers
- **resources** (*Resources*) – A Resources object containing global nodelist and intranode information
- **workerID** (*int*) – workerID of current process

get_slots_as_string(*multiplier=1*, *delimiter=','*, *limit=None*)

Returns list of slots as a string

Parameters

- **multiplier** – Optional int. Assume this many items per slot.
- **delimiter** – Optional int. Delimiter for output string.
- **limit** – Optional int. Maximum slots (truncate list after this many slots).

set_env_to_slots(*env_var*, *multiplier=1*, *delimiter=','*)

Sets the given environment variable to slots

Parameters

- **env_var** – String. Name of environment variable to set.
- **multiplier** – Optional int. Assume this many items per slot.
- **delimiter** – Optional int. Delimiter for output string.

Example usage in a sim function:

With resources imported:

```
from libensemble.resources.resources import Resources
```

Obtain worker resources:

```
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")
```

set_env_to_gpus(*env_var=None*, *delimiter=','*)

Sets the given environment variable to GPUs

Parameters

- **env_var** – String. Name of environment variable to set.
- **delimiter** – Optional int. Delimiter for output string.

Example usage in a sim function:

With resources imported:

```
from libensemble.resources.resources import Resources
```

Obtain worker resources:

```
resources = Resources.resources.worker_resources
resources.set_env_to_gpus("CUDA_VISIBLE_DEVICES")
```

doihave_gpus()

Are this workers current resource sets GPU rsets

set_rset_team(*rset_team*)

Update worker team and local attributes

Updates: rset_team
local_nodelist slots (dictionary with list of partitions for each node) slot_count - number of slots on each node local_node_count

Parameters
rset_team (*List[int]*) –

Return type
None

set_gen_procs_gpus(*libE_info*)

Add gen supplied procs and gpus

set_slot_count()

Sets attributes even_slots and matching_slots.

Also sets slot_count if even_slots (else None) and sets slots_on_node if matching_slots (else None).

Return type
None

static get_local_nodelist(*workerID*, *rset_team*, *split_list*, *rsets_per_node*)

Returns the list of nodes available to the given worker and the slot dictionary

Parameters

- **workerID** (*int*) –
- **rset_team** (*List[int]*) –
- **split_list** (*List[List[str]]*) –
- **rsets_per_node** (*int*) –

Return type
Tuple[*List*[str], *Dict*[str, *List*[int]]]

5.2.7 Environment Resources Module

This module stores environment variables for use in resource detection

class env_resources.**EnvResources**(*nodelist_env_slurm=None*, *nodelist_env_cobalt=None*, *nodelist_env_pbs=None*, *nodelist_env_lsf=None*, *nodelist_env_lsf_shortform=None*)

Stores environment variables to query for system resource information

Class Attributes:

Variables

- **default_nodelist_env_slurm** (*string*) – Default SLRUM nodelist environment variable
- **default_nodelist_env_cobalt** (*string*) – Default Cobalt nodelist environment variable
- **default_nodelist_env_pbs** (*string*) – Default PBS nodelist environment variable (points to nodefile)
- **default_nodelist_env_lsf** (*string*) – Default LSF nodelist environment variable
- **default_nodelist_env_lsf_shortform** (*string*) – Default LSF short-form nodelist environment variable

Parameters

- **nodelist_env_slurm** (*str | None*) –
- **nodelist_env_cobalt** (*str | None*) –
- **nodelist_env_pbs** (*str | None*) –
- **nodelist_env_lsf** (*str | None*) –
- **nodelist_env_lsf_shortform** (*str | None*) –

Object Attributes:

These are set on initialization.

Variables

- **nodelists** (*dict*) – Environment variable names to query for nodelists by scheduler
- **ndlist_funcs** (*dict*) – Functions to extract nodelists from environment by scheduler

Parameters

- **nodelist_env_slurm** (*str | None*) –
- **nodelist_env_cobalt** (*str | None*) –
- **nodelist_env_pbs** (*str | None*) –
- **nodelist_env_lsf** (*str | None*) –
- **nodelist_env_lsf_shortform** (*str | None*) –

__init__(*nodelist_env_slurm=None*, *nodelist_env_cobalt=None*, *nodelist_env_pbs=None*, *nodelist_env_lsf=None*, *nodelist_env_lsf_shortform=None*)

Initializes a new EnvResources instance

Determines the environment variables to query for resource information. These are either provided or given defaults.

Parameters

- **nodelist_env_slurm** (*String, optional*) – The environment variable giving a node list in Slurm format (Default: uses SLURM_NODELIST). Note: This is queried only if a node_list file is not provided.
- **nodelist_env_cobalt** (*String, optional*) – The environment variable giving a node list in Cobalt format (Default: uses COBALT_PARTNAME). Note: This is queried only if a node_list file is not provided and disable_resource_manager=True.
- **nodelist_env_pbs** (*String, optional*) – The environment variable giving a path to a nodefile in PBS format (Default: uses PBS_NODEFILE). This is queried if a node_list file is not provided.
- **nodelist_env_lsf** (*String, optional*) – The environment variable giving a node list in LSF format (Default: uses LSB_HOSTS). Note: This is queried only if a node_list file is not provided.
- **nodelist_env_lsf_shortform** (*String, optional*) – The environment variable giving a node list in LSF short-form format (Default: uses LSB_MCPU_HOSTS). Note: This is queried only if a node_list file is not provided.

Return type
None

get_nodelist()

Returns nodelist from environment or an empty list

Return type
List[str | *Any*]

static abbrev_nodenames(*node_list*, *prefix=None*)

Returns nodelist with only string up to first dot

Parameters

- **node_list** (*List[str]*) –
- **prefix** (*str | None*) –

Return type
List[str]

static cobalt_abbrev_nodenames(*node_list*, *prefix='nid'*)

Returns nodelist with prefix and leading zeros stripped

Parameters

- **node_list** (*List[str]*) –
- **prefix** (*str*) –

Return type
List[str]

shortnames(*node_list*)

Returns nodelist with entries in abbreviated form

Parameters
node_list (*List[str]*) –

Return type
List[str]

static get_slurm_nodelist(*node_list_env*)

Gets global libEnsemble nodelist from the Slurm environment

Parameters
node_list_env (*str*) –

Return type
List[str | *Any*]

static get_cobalt_nodelist(*node_list_env*)

Gets global libEnsemble nodelist from the Cobalt environment

Parameters
node_list_env (*str*) –

Return type
List[str | *Any*]

static get_pbs_nodelist(*node_list_env*)

Gets global libEnsemble nodelist path from PBS environment

Parameters
node_list_env (*str*) –

Return type
List[str | *Any*]

static get_lsf_nodelist(*node_list_env*)

Gets global libEnsemble nodelist from the LSF environment

Parameters
node_list_env (*str*) –

Return type
List[str | *Any*]

static get_lsf_nodelist_frm_shortform(*node_list_env*)

Gets global libEnsemble nodelist from the LSF environment from short-form version

Parameters
node_list_env (*str*) –

Return type
List[str | *Any*]

5.2.8 Node Resources Module

This module for detects and returns intranode resources

node_resources.**get_cpu_cores**(*hyperthreads=False*)

Returns the number of cores on the node.

If hyperthreads is true, this is the logical CPU cores; else the physical cores are returned.

Note: This returns cores available on the current node. It will not work for systems of multiple node types

Parameters
hyperthreads (*bool*) –

Return type
int

node_resources.**get_sub_node_resources**(*launcher=None*, *remote_mode=False*, *env_resources=None*)

Returns logical and physical cores and GPUs per node as a tuple

First checks for known system values, then for environment values, and finally for detected values. If remote_mode is True, then detection launches a job via the MPI launcher.

Any value that is already valid, is not overwritten by successive stages.

Parameters

- **launcher** (*str | None*) –
- **remote_mode** (*bool*) –
- **env_resources** (*EnvResources | None*) –

Return type
Tuple[int, int, int]

5.2.9 MPI Resources Module

Manages libensemble resources related to MPI tasks launched from nodes.

exception mpi_resources.**MPIResourcesException**

Resources module exception

mpi_resources.**get_MPI_variant**()

Returns MPI base implementation

Returns
mpi_variant – MPI variant 'aprun' or 'jsrun' or 'msmpi' or 'mpich' or 'openmpi' or 'srun'

Return type
str

mpi_resources.**get_MPI_runner**(*mpi_runner=None*)

Return whether `mpirun` is openmpi or mpich

Return type
str

mpi_resources.**task_partition**(*num_procs*, *num_nodes*, *procs_per_node*, *machinefile=None*)

Takes provided nprocs/nodes/ranks and outputs working configuration of procs/nodes/ranks or error

Parameters

- **num_procs** (*int | None*) –
- **num_nodes** (*int | None*) –
- **procs_per_node** (*int | None*) –
- **machinefile** (*str | None*) –

Return type
Tuple[None, None, None] | *Tuple*[int, int, int]

mpi_resources.**get_resources**(*resources*, *num_procs=None*, *num_nodes=None*, *procs_per_node=None*, *hyperthreads=False*)

Reconciles user-supplied options with available worker resources to produce run configuration.

Detects resources available to worker, checks whether an existing user-supplied config is valid, and fills in any missing config information (i.e., num_procs/num_nodes/procs_per_node)

User-supplied config options are honored, and an exception is raised if these are infeasible.

mpi_resources.**create_machinefile**(*resources*, *machinefile=None*, *num_procs=None*, *num_nodes=None*, *procs_per_node=None*, *hyperthreads=False*)

Creates a machinefile based on user-supplied config options, completed by detected machine resources

Parameters

- **resources** (*resources.Resources*) –
- **machinefile** (*str | None*) –
- **num_procs** (*int*) –
- **num_nodes** (*int | None*) –
- **procs_per_node** (*int | None*) –
- **hyperthreads** (*bool*) –

Return type
Tuple[bool, None, int, int]

mpi_resources.**get_hostlist**(*resources*, *num_nodes=None*)

Creates a hostlist based on user-supplied config options.

completed by detected machine resources

5.2.10 Scheduler Module

class resources.scheduler.**ResourceScheduler**(*user_resources=None*, *sched_opts={}*)

Calculates and returns resource set ids from a dictionary of resource sets by group. The available resource sets are read initially from the resources module or from a resources object passed in.

Resource sets are locally provisioned to work items by a call to the `assign_resources` function, and a cache of available resource sets is maintained for the life of the object (usually corresponding to one call of the allocation function). Note that work item resources are formally assigned to workers only when a work item is sent to the worker.

assign_resources(*rsets_req*, *use_gpus=None*, *user_params=[]*)

Schedule resource sets to a work item if possible.

If the resources required are less than one node, they will be allocated to the smallest available sufficient slot.

If the resources required are more than one node, then the scheduler will attempt to find an even split. If no even split is possible, then enough additional resource sets will be assigned to enable an even split.

Returns a list of resource set IDs or raises an exception (either InsufficientResourcesError or InsufficientFreeResources).

find_rsets_any_slots(*valid_rsets_by_group*, *max_grpsize*, *rsets_req*, *ngroups*, *rsets_per_group*)

Find optimal non-matching slots across groups

find_candidate(*rsets_by_group*, *group_list*, *rsets_per_group*, *max_upper_bound*)

Find a candidate slot in a group

get_avail_rsets_by_group()

Return a dictionary of resource set IDs for each group (e.g. node)

If groups are not set they will all be in one group (group 0)

E.g: Say 8 resource sets / 2 nodes GROUP 1: [1,2,3,4] GROUP 2: [5,6,7,8]

filter_for_rset_type(*avail_rsets_by_group*, *use_gpus*)

Return avail_rsets_by_group filtered by rset type (gpus/non-gpus/all)

filter_out_rset_team(*avail_rsets_by_group*, *rset_team*)

Return avail_rsets_by_group filtered by rset type (gpus/non-gpus/all)

static get_slots_of_len(*d*, *n*)

Filter dictionary to values >= n

get_avail_slots_by_group(*rsets_by_group*)

Return a dictionary of free slot IDS for each group (e.g. node)

calc_rsets_even_grps(*rsets_req*, *max_grpsize*, *max_groups*, *extend*)

Calculate an even breakdown to best fit rsets_req input

check_params(*user_params*, *ngroups*)

Return True if all user params divide by number of groups, else False

calc_even_split_uneven_groups(*rsets_per_grp*, *ngroups*, *rsets_req*, *sorted_lens*, *max_grps*, *user_params*)

Calculate an even breakdown to best fit rsets_req with uneven groups

assign_team_from_slots(*slots_avail_by_group*, *cand_groups*, *cand_slots*, *rsets_per_group*)

Assign resource set team from slots

static get_sorted_lens(*avail_rsets*)

Get max length of a list value in a dictionary

get_matching_slots(*slots_avail_by_group*, *num_groups_req*, *rsets_per_group*)

Get first N matching slots across groups

Assumes num_groups_req > 0.

check_total_rsets(*rsets_req*, *use_gpus*)

Raise exceptions if rsets requested is more than total that exist or available

5.3 Release Management

This section documents the steps to be followed for each libEnsemble release.

5.3.1 Release Process

A release can be undertaken only by a project administrator. A project administrator should have an administrator role on the libEnsemble GitHub, PyPI, and readthedocs pages.

Before release

- A GitHub issue is created with a checklist for the release.
- A release branch should be taken off develop (or develop pulls controlled).
- Release notes for this version are added to the documentation with release date, including a list of supported (tested) platforms.
- Version number is updated wherever it appears (and `+dev` suffix is removed) (in `libensemble/version.py`).
- Year in `README.rst` under *Citing libEnsemble* is checked for correctness. (Note: The year generated in docs by `docs/conf.py` should be automatic).
- `setup.py` and `libensemble/__init__.py` are checked to ensure all information is up to date.
- Update `.wci.yml` in root directory (version, date and any other information).
- `MANIFEST.in` is checked. Locally, try out `python setup.py sdist` and check created tarball. contains correct files and directories for PyPI package.
- Tests are run with source to be released (this may iterate):
  - On-line CI (GitHub Actions) tests must pass.
  - Scaling tests must be run on HPC platforms listed as supported in release notes. Test variants by platform, launch mechanism, scale, and other factors can be configured and exported by the libE-Templater.
  - Coverage must not have decreased unless there is a justifiable reason.
  - Documentation must build and display correctly wherever hosted (currently readthedocs.com).
- Pull request from either the develop or release branch to main requesting one or more reviewers (including at least one other administrator).
- Reviewer will check that all tests have passed and will then approve merge.

During release

An administrator will take the following steps.

- Merge the pull request into main.
- Once CI tests have passed on main:
  - A GitHub release will be taken from the main (`github release`).
  - A tarball (source distribution) will be uploaded to PyPI (`PyPI release`).
  - The Conda package will be updated (`Conda release`).
  - Spack package will be updated (`Spack release`).
- If the merge was made from a release branch (instead of develop), merge this branch into develop.
- Create a new commit on develop that appends `+dev` to the version number (wherever is appears).

After release

- Ensure all relevant GitHub issues are closed and moved to the *Done* column on the kanban project board (inc. the release checklist).
- Email libEnsemble mailing list, and notify the *everyone* channel in the libEnsemble Slack workspace.

5.3.2 Release Platforms

GitHub release

The administrator should follow the GitHub instructions to draft a new release. These can currently be found at https://help.github.com/en/articles/creating-releases.

Both the version and title will be of the form vX.Y.Z, for example, v0.5.0.

From version 1.0, these should follow semantic versioning, where X/Y/Z are major, minor, and patch revisions.

Prior to version 1.0, the second number may include breaking API changes, and the third number may include minor additions.

The release notes should be included in the description. These should already be in `docs/release_notes.rst`. The release notes should be copied only for the current release, starting from the date. Hint: To see example of raw input, click *edit* next to one of the previous releases.

Note that unlike some platforms (e.g., PyPI), GitHub releases can be edited or deleted once created.

PyPI release

libEnsemble is released on the Python Package Index (commonly known as PyPI). This enables users to `pip install` the package.

The package is stored on PyPI in the form of a source distribution (commonly known as a tarball). The tarball should be created as detailed below (which creates the distribution package using the MANIFEST.in file in the git root directory. Do not use the tarball on GitHub, which does not follow MANIFEST.in and does not contain the required PKG-INFO file.

You will need logon credentials for the libEnsemble PyPI. You will also need twine (which can be pip or Conda installed).

In the package directory on the main branch (the one containing setup.py) do the following:

Create distribution:

```
python setup.py sdist
```

Upload (you will need username/password here):

```
twine upload dist/*
```

If you now run

```
pip install libensemble
```

it should find the new version.

It should also be visible here:

https://pypi.org/project/libensemble/

For more details on creating PyPI packages see

https://betterscientificsoftware.github.io/python-for-hpc/tutorials/python-pypi-packaging/

Conda release

libEnsemble is released as part of the conda-forge distribution. This enables users to `conda install` the package.

The Conda package is created from the conda-forge feedstock repository. This repository comes with some common dependencies and automatically creates three variants (no-mpi, mpich, Open MPI).

Automatic PR

Note that once libEnsemble has been released on PYPI a conda-forge bot will usually detect the new release and automatically create a pull request with the changes below. It may take a few hours for this to happen. If no other changes are required (e.g., new dependencies), then you can simply wait for the tests to pass and merge.

Manual PR

If necessary, a manual PR can be created as follows.

Create a fork of the repository (not a branch). In the file `recipe/meta.yaml` bump the `version number`, set the `build number` to zero, and update the `sha256`. The latter can be obtained by running `sha256sum` on the github tarball. For example, for v0.6.0:

```
sha256sum libensemble-0.6.0.tar.gz
```

Then, use the phrase *@conda-forge-admin, please rerender* in a comment in the pull request for automated rerendering. The github-actions bot will reply with a message when ready to merge.

Release

Approvals from other libEnsemble administrators will be required. Once the pull request is merged, the new package should become available to Conda, in the *conda-forge* channel, after a processing delay.

You can then check the three versions:

- *conda install libensemble*
- *conda install libensemble=*=mpi_mpich**.
- *conda install libensemble=*=mpi_openmpi**

A workflow for updating libEnsemble on Spack

This assumes you have already:

- made a PyPI package for new version of libEnsemble and
- made a GitHub fork of Spack and cloned it to your local system.

Details on how to create forks can be found at https://help.github.com/articles/fork-a-repo.

You now have a configuration like that shown at (but without the upstream/local connection). https://stackoverflow.com/questions/6286571/are-git-forks-actually-git-clones.

Upstream, in this case, is the official Spack repository on GitHub. Origin is your fork on GitHub, and Local Machine is your local clone (from your fork).

Make sure SPACK_ROOT is set and Spack binary is in your path:

```
export SPACK_ROOT=<PATH/TO/LOCAL/SPACK/REPO>
export PATH=$SPACK_ROOT/bin:$PATH
```

Do ONCE in your local checkout:

To set upstream repo:

```
git remote add upstream https://github.com/spack/spack.git
git remote -v # check added
```

(Optional) To prevent accidental pushes to upstream:

```
git remote set-url --push upstream no_push
git remote -v # Check for line: `upstream no_push (push)`
```

Updating (the main develop branch)

You will now update your local machine from the upstream repo (if in doubt, make a copy of the local repo in your file system before doing the following).

Check that the upstream remote is present:

```
git remote -v
```

Ensure that you are on the develop branch:

```
git checkout develop
```

Fetch from the upstream repo:

```
git fetch upstream
```

To update your local machine, you may wish to rebase or overwrite your local files. Select from the following:

If you have local changes to go "on top" of latest code:

```
git rebase upstream/develop
```

Or to make your local machine identical to upstream repo (**WARNING:** Any local changes will be lost!):

```
git reset --hard upstream/develop
```

(Optional) You may want to update your forked (origin) repo on GitHub at this point. This may requires a forced push:

```
git push origin develop --force
```

Making changes

The instructions below assume you make changes on the default develop branch. You can optionally create a branch to make changes on. Doing so may be a good idea, especially if you have multiple packages, to make separate branches for each package.

See the Spack [packaging](https://spack.readthedocs.io/en/latest/packaging_guide.html) and [contribution](https://spack.readthedocs.io/en/latest/contribution_guide.html) guides for more info.

Quick example to update libEnsemble

This will open the libEnsemble `package.py` file in your editor (given by environment variable `EDITOR`):

```
spack edit py-libensemble  # SPACK_ROOT must be set (see above) (python packages use "py-
↪" prefix)
```

Or just open it manually: `var/spack/repos/builtin/packages/py-libensemble/package.py`.

Now get checksum for new lines:

Get the tarball (see PyPI instructions), for the new release and use:

```
sha256sum libensemble-*.tar.gz
```

Update the `package.py` file by pasting in the new checksum lines (and make sure the URL line points to the latest version). Also update any dependencies for the new version.

Check package:

```
spack style
```

This will install a few python spack packages and run style checks on just your changes. Make adjustments if needed, until this passes.

If okay - add, commit, and push to origin (forked repo). For example, if your version number is 0.9.1:

```
git commit -am "libEnsemble: add v0.9.1"
git push origin develop --force
```

Once the branch is pushed to the forked repo, go to GitHub and do a pull request from this branch on the fork to the develop branch on the upstream.

Express Summary: Make Fork Identical to Upstream

Quick summary for bringing develop branch on forked repo up to speed with upstream (YOU WILL LOSE ANY CHANGES):

```
git remote add upstream https://github.com/spack/spack.git
git fetch upstream
git checkout develop
git reset --hard upstream/develop
git push origin develop --force
```

Reference: <https://stackoverflow.com/questions/9646167/clean-up-a-fork-and-restart-it-from-the-upstream/39628366>

Chapter 6

Appendices

This section contains tutorials, frequently asked questions, examples, and other libEnsemble information.

6.1 Advanced Installation

libEnsemble can be installed from `pip`, `Conda`, or `Spack`.

libEnsemble requires the following dependencies, which are typically automatically installed alongside libEnsemble:

- Python >= `3.9`
- NumPy >= `1.21`
- psutil >= `5.9.4`
- pydantic <= `1.10.12`
- pyyaml >= `v6.0`
- tomli >= `1.2.1`

Given libEnsemble's compiled dependencies, the following installation methods each offer a trade-off between convenience and the ability to customize builds, including platform-specific optimizations.

We always recommend installing in a virtual environment from Conda or another source.

Further recommendations for selected HPC systems are given in the `HPC platform guides`.

pip

To install the latest PyPI release:

```
pip install libensemble
```

To pip install libEnsemble from the latest develop branch:

```
python -m pip install --upgrade git+https://github.com/Libensemble/libensemble.
↪git@develop
```

Installing with mpi4py

If you wish to use `mpi4py` with libEnsemble (choosing MPI out of the three `communications options`), then this should be installed to work with the existing MPI on your system. For example, the following line:

```
pip install mpi4py
```

will use the `mpicc` compiler wrapper on your PATH to identify the MPI library. To specify a different compiler wrapper, add the `MPICC` option. You also may wish to avoid existing binary builds; for example,:

```
MPICC=mpiicc pip install mpi4py --no-binary mpi4py
```

On Summit, the following line is recommended (with gcc compilers):

```
CC=mpicc MPICC=mpicc pip install mpi4py --no-binary mpi4py
```

conda

Install libEnsemble with Conda from the conda-forge channel:

```
conda config --add channels conda-forge
conda install -c conda-forge libensemble
```

This package comes with some useful optional dependencies, including optimizers and will install quickly as ready binary packages.

Installing with mpi4py with Conda

If you wish to use `mpi4py` with libEnsemble (choosing MPI out of the three `communications options`), you can use the following.

Note: For clusters and HPC systems, always install `mpi4py` to use the system MPI library (see pip instructions above).

For a standalone build that comes with an MPI implementation, you can install libEnsemble using one of the following variants.

To install libEnsemble with MPICH:

```
conda install -c conda-forge libensemble=*=mpi_mpich*
```

To install libEnsemble with Open MPI:

```
conda install -c conda-forge libensemble=*=mpi_openmpi*
```

The asterisks will pick up the latest version and build.

Note: This syntax may not work without adjustments on macOS or any non-bash shell. In these cases, try:

```
conda install -c conda-forge libensemble='*'=mpi_mpich'*'
```

For a complete list of builds for libEnsemble on Conda:

```
conda search libensemble --channel conda-forge
```

Spack

Install libEnsemble using the Spack distribution:

```
spack install py-libensemble
```

The above command will install the latest release of libEnsemble with the required dependencies only. Other optional dependencies can be specified through variants. The following line installs libEnsemble version 0.7.2 with some common variants (e.g., using APOSMM):

```
spack install py-libensemble @0.7.2 +mpi +scipy +mpmath +petsc4py +nlopt
```

The list of variants can be found by running:

```
spack info py-libensemble
```

On some platforms you may wish to run libEnsemble without `mpi4py`, using a serial PETSc build. This is often preferable if running on the launch nodes of a three-tier system (e.g., Theta/Summit):

```
spack install py-libensemble +scipy +mpmath +petsc4py ^py-petsc4py~mpi ^petsc~mpi~hdf5~
↪hypre~superlu-dist
```

The installation will create modules for libEnsemble and the dependent packages. These can be loaded by running:

```
spack load -r py-libensemble
```

Any Python packages will be added to the PYTHONPATH when the modules are loaded. If you do not have modules on your system you may need to install `lmod` (also available in Spack):

```
spack install lmod
. $(spack location -i lmod)/lmod/lmod/init/bash
spack load lmod
```

Alternatively, Spack could be used to build the serial `petsc4py`, and Conda could use this by loading the `py-petsc4py` module thus created.

Hint: When combining Spack and Conda, you can access your Conda Python and packages in your `~/.spack/packages.yaml` while your Conda environment is activated, using `CONDA_PREFIX` For example, if you have an activated Conda environment with Python 3.9 and SciPy installed:

```
packages:
python:
    externals:
    - spec: "python"
    prefix: $CONDA_PREFIX
    buildable: False
py-numpy:
    externals:
    - spec: "py-numpy"
    prefix: $CONDA_PREFIX/lib/python3.9/site-packages/numpy
    buildable: False
py-scipy:
    externals:
    - spec: "py-scipy"
    prefix: $CONDA_PREFIX/lib/python3.9/site-packages/scipy
    buildable: True
```

For more information on Spack builds and any particular considerations for specific systems, see the spack_libe repository. In particular, this includes some example `packages.yaml` files (which go in `~/.spack/`). These files are used to specify dependencies that Spack must obtain from the given system (rather than building from scratch). This may include `Python` and the packages distributed with it (e.g., `numpy`), and will often include the system MPI library.

6.1.1 Optional Dependencies for Additional Features

The following packages may be installed separately to enable additional features:

- Balsam - Manage and submit applications to the Balsam service with our `BalsamExecutor`
- pyyaml and tomli - Parameterize libEnsemble via yaml or toml
- Globus Compute - Submit simulation or generator function instances to remote Globus Compute endpoints
- psi-j-python and tqdm - Use *liberegister* and *libesubmit* to submit libEnsemble jobs to any scheduler

6.2 Tutorials

6.2.1 Simple Introduction

This tutorial demonstrates the capability to perform ensembles of calculations in parallel using libEnsemble.

We recommend reading this brief Overview.

For this tutorial, our generator will produce uniform randomly sampled values, and our simulator will calculate the sine of each. By default we don't need to write a new allocation function.

1. Getting started

libEnsemble is written entirely in Python. Let's make sure the correct version is installed.

```
python --version  # This should be >= 3.9
```

For this tutorial, you need NumPy and (optionally) Matplotlib to visualize your results. Install libEnsemble and these other libraries with

```
pip install libensemble
pip install matplotlib # Optional
```

If your system doesn't allow you to perform these installations, try adding `--user` to the end of each command.

2. Generator

Let's begin the coding portion of this tutorial by writing our generator function, or `gen_f`.

An available libEnsemble worker will call this generator function with the following parameters:

- `InputArray`: A selection of the `History array` (H), passed to the generator function in case the user wants to generate new values based on simulation outputs. Since our generator produces random numbers, it'll be ignored this time.
- `persis_info`: Dictionary with worker-specific information. In our case, this dictionary contains NumPy Random Stream objects for generating random numbers.

- gen_specs: Dictionary with user-defined static fields and parameters. Customizable parameters such as lower and upper bounds and batch sizes are placed within the gen_specs["user"] dictionary.

Later on, we'll populate gen_specs and persis_info when we initialize libEnsemble.

For now, create a new Python file named sine_gen.py. Write the following:

Listing 1: examples/tutorials/simple_sine/sine_gen.py

```
import numpy as np


def gen_random_sample(InputArray, persis_info, gen_specs):
    # Pull out user parameters
    user_specs = gen_specs["user"]

    # Get lower and upper bounds
    lower = user_specs["lower"]
    upper = user_specs["upper"]

    # Determine how many values to generate
    num = len(lower)
    batch_size = user_specs["gen_batch_size"]

    # Create empty array of "batch_size" zeros. Array dtype should match "out" fields
    OutputArray = np.zeros(batch_size, dtype=gen_specs["out"])

    # Set the "x" output field to contain random numbers, using random stream
    OutputArray["x"] = persis_info["rand_stream"].uniform(lower, upper, (batch_size,␣
↪num))

    # Send back our output and persis_info
    return OutputArray, persis_info
```

Our function creates batch_size random numbers uniformly distributed between the lower and upper bounds. A random stream from persis_info is used to generate these values, which are then placed into an output NumPy array that matches the dtype from gen_specs["out"].

3. Simulator

Next, we'll write our simulator function or sim_f. Simulator functions perform calculations based on values from the generator function. The only new parameter here is sim_specs, which serves a purpose similar to the gen_specs dictionary.

Create a new Python file named sine_sim.py. Write the following:

Listing 2: examples/tutorials/simple_sine/sine_sim.py

```
import numpy as np


def sim_find_sine(InputArray, _, sim_specs):
    # Create an output array of a single zero
    OutputArray = np.zeros(1, dtype=sim_specs["out"])

```

(continues on next page)

(continued from previous page)

```
    # Set the zero to the sine of the InputArray value
    OutputArray["y"] = np.sin(InputArray["x"])

    # Send back our output
    return OutputArray
```

Our simulator function is called by a worker for every work item produced by the generator function. This function calculates the sine of the passed value, and then returns it so the worker can store the result.

4. Script

Now lets write the script that configures our generator and simulator functions and starts libEnsemble.

Create an empty Python file named `calling.py`. In this file, we'll start by importing NumPy, libEnsemble's setup classes, and the generator and simulator functions we just created.

In a class called `LibeSpecs` we'll specify the number of workers and the manager/worker intercommunication method. `"local"`, refers to Python's multiprocessing.

```
import numpy as np
from sine_gen import gen_random_sample
from sine_sim import sim_find_sine

from libensemble import Ensemble
from libensemble.specs import ExitCriteria, GenSpecs, LibeSpecs, SimSpecs

if __name__ == "__main__":  # Python-quirk required on macOS and windows
    libE_specs = LibeSpecs(nworkers=4, comms="local")
```

We configure the settings and specifications for our `sim_f` and `gen_f` functions in the `GenSpecs` and `SimSpecs` classes, which we saw previously being passed to our functions *as dictionaries*. These classes also describe to libEnsemble what inputs and outputs from those functions to expect.

```
    gen_specs = GenSpecs(
        gen_f=gen_random_sample,  # Our generator function
        out=[("x", float, (1,))],  # gen_f output (name, type, size)
        user={
            "lower": np.array([-3]),  # lower boundary for random sampling
            "upper": np.array([3]),  # upper boundary for random sampling
            "gen_batch_size": 5,  # number of x's gen_f generates per call
        },
    )

    sim_specs = SimSpecs(
        sim_f=sim_find_sine,  # Our simulator function
        inputs=["x"],  # InputArray field names. "x" from gen_f output
        out=[("y", float)],  # sim_f output. "y" = sine("x")
    )  # sim_specs_end_tag
```

We then specify the circumstances where libEnsemble should stop execution in `ExitCriteria`.

```
    exit_criteria = ExitCriteria(sim_max=80)  # Stop libEnsemble after 80 simulations
```

Now we're ready to write our libEnsemble `libE` function call. `ensemble.H` is the final version of the history array. `ensemble.flag` should be zero if no errors occur.

```
    ensemble = Ensemble(sim_specs, gen_specs, exit_criteria, libE_specs)
    ensemble.add_random_streams()  # setup the random streams unique to each worker
    ensemble.run()  # start the ensemble. Blocks until completion.

    history = ensemble.H  # start visualizing our results

    print([i for i in history.dtype.fields])  # (optional) to visualize our history array
    print(history)
```

That's it! Now that these files are complete, we can run our simulation.

```
python calling.py
```

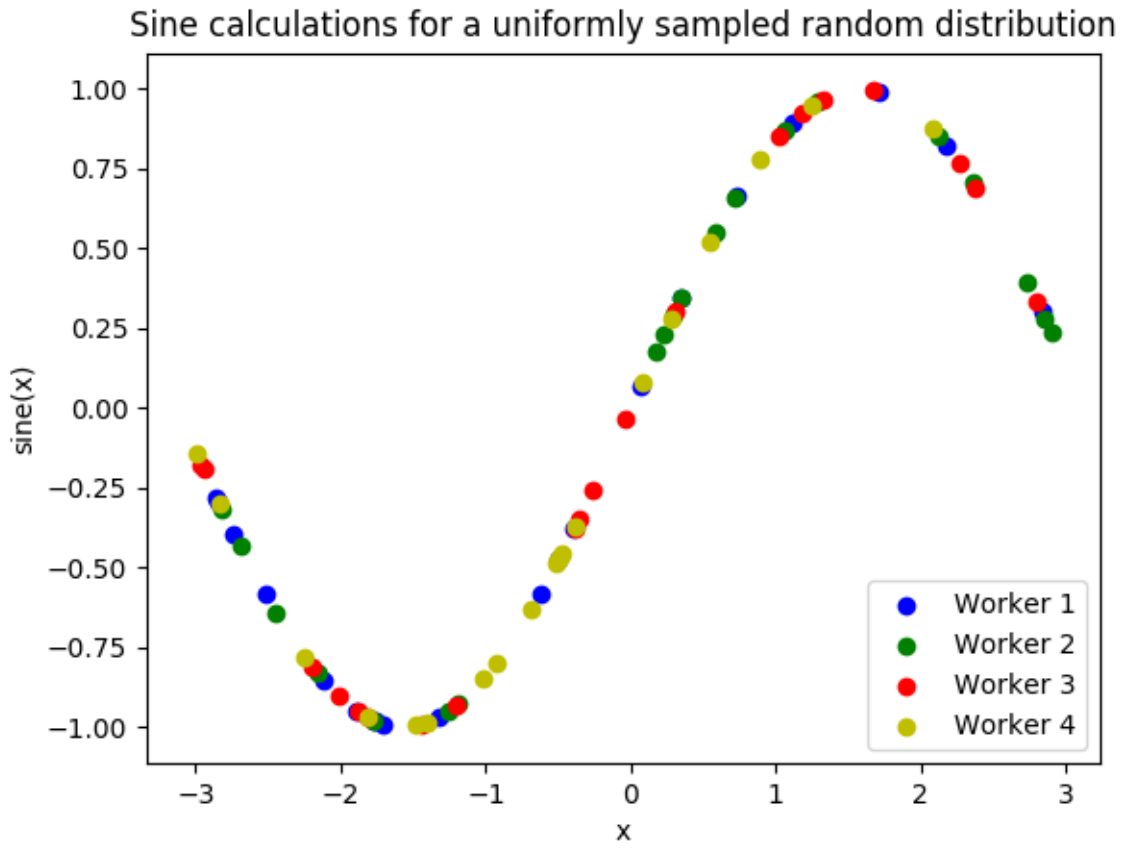
If everything ran perfectly and you included the above print statements, you should get something similar to the following output (although the columns might be rearranged).

```
["y", "sim_started_time", "gen_worker", "sim_worker", "sim_started", "sim_ended", "x",
↪"allocated", "sim_id", "gen_ended_time"]
[(-0.37466051, 1.559+09, 2, 2,  True,  True, [-0.38403059],  True,  0, 1.559+09)
(-0.29279634, 1.559+09, 2, 3,  True,  True, [-2.84444261],  True,  1, 1.559+09)
( 0.29358492, 1.559+09, 2, 4,  True,  True, [ 0.29797487],  True,  2, 1.559+09)
(-0.3783986, 1.559+09, 2, 1,  True,  True, [-0.38806564],  True,  3, 1.559+09)
(-0.45982062, 1.559+09, 2, 2,  True,  True, [-0.47779319],  True,  4, 1.559+09)
...
```

In this arrangement, our output values are listed on the far left with the generated values being the fourth column from the right.

Two additional log files should also have been created. `ensemble.log` contains debugging or informational logging output from libEnsemble, while `libE_stats.txt` contains a quick summary of all calculations performed.

Here is graphed output using `Matplotlib`, with entries colored by which worker performed the simulation:



If you want to verify your results through plotting and installed Matplotlib earlier, copy and paste the following code into the bottom of your calling script and run `python calling.py` again

```
import matplotlib.pyplot as plt

colors = ["b", "g", "r", "y", "m", "c", "k", "w"]

for i in range(1, libE_specs.nworkers + 1):
    worker_xy = np.extract(history["sim_worker"] == i, history)
    x = [entry.tolist()[0] for entry in worker_xy["x"]]
    y = [entry for entry in worker_xy["y"]]
    plt.scatter(x, y, label="Worker {}".format(i), c=colors[i - 1])

plt.title("Sine calculations for a uniformly sampled random distribution")
plt.xlabel("x")
plt.ylabel("sine(x)")
plt.legend(loc="lower right")
plt.savefig("tutorial_sines.png")
```

Each of these example files can be found in the repository in examples/tutorials/simple_sine.

Exercise

Write a Calling Script with the following specifications:

1. Set the generator function's lower and upper bounds to -6 and 6, respectively

2. Increase the generator batch size to 10
3. Set libEnsemble to stop execution after 160 *generations* using the `gen_max` option
4. Print an error message if any errors occurred while libEnsemble was running

Click Here for Solution

```python
import numpy as np
from sine_gen import gen_random_sample
from sine_sim import sim_find_sine

from libensemble import Ensemble
from libensemble.specs import ExitCriteria, GenSpecs, LibeSpecs, SimSpecs

if __name__ == "__main__":
    libE_specs = LibeSpecs(nworkers=4, comms="local")

    gen_specs = GenSpecs(
        gen_f=gen_random_sample,  # Our generator function
        out=[("x", float, (1,))],  # gen_f output (name, type, size)
        user={
            "lower": np.array([-6]),  # lower boundary for random sampling
            "upper": np.array([6]),  # upper boundary for random sampling
            "gen_batch_size": 10,  # number of x's gen_f generates per call
        },
    )

    sim_specs = SimSpecs(
        sim_f=sim_find_sine,  # Our simulator function
        inputs=["x"],  # InputArray field names. "x" from gen_f output
        out=[("y", float)],  # sim_f output. "y" = sine("x")
    )

    exit_criteria = ExitCriteria(gen_max=160)

    ensemble = Ensemble(sim_specs, gen_specs, exit_criteria, libE_specs)
    ensemble.add_random_streams()
    ensemble.run()

    if ensemble.flag != 0:
        print("Oh no! An error occurred!")
```

5. Next steps

libEnsemble with MPI

MPI is a standard interface for parallel computing, implemented in libraries such as MPICH and used at extreme scales. MPI potentially allows libEnsemble's processes to be distributed over multiple nodes and works in some circumstances where Python's multiprocessing does not. In this section, we'll explore modifying the above code to use MPI instead of multiprocessing.

We recommend the MPI distribution MPICH for this tutorial, which can be found for a variety of systems here. You also need mpi4py, which can be installed with `pip install mpi4py`. If you'd like to use a specific version or distribution

of MPI instead of MPICH, configure mpi4py with that MPI at installation with `MPICC=<path/to/MPI_C_compiler> pip install mpi4py` If this doesn't work, try appending `--user` to the end of the command. See the mpi4py docs for more information.

Verify that MPI has been installed correctly with `mpirun --version`.

Modifying the script

Only a few changes are necessary to make our code MPI-compatible. For starters, comment out the `libE_specs` definition:

```
    # libE_specs = LibeSpecs(nworkers=4, comms="local")
```

We'll be parameterizing our MPI runtime with a `parse_args=True` argument to the `Ensemble` class instead of `libE_specs`. We'll also use an `ensemble.is_manager` attribute so only the first MPI rank runs the data-processing code.

The bottom of your calling script should now resemble:

```
    # replace libE_specs with parse_args=True. Detects MPI runtime
    ensemble = Ensemble(sim_specs, gen_specs, exit_criteria, parse_args=True)

    ensemble.add_random_streams()
    ensemble.run()  # start the ensemble. Blocks until completion.

    if ensemble.is_manager:  # only True on rank 0
        history = ensemble.H  # start visualizing our results
        print([i for i in history.dtype.fields])
        print(history)

        import matplotlib.pyplot as plt

        colors = ["b", "g", "r", "y", "m", "c", "k", "w"]

        for i in range(1, ensemble.nworkers + 1):
            worker_xy = np.extract(history["sim_worker"] == i, history)
            x = [entry.tolist()[0] for entry in worker_xy["x"]]
            y = [entry for entry in worker_xy["y"]]
            plt.scatter(x, y, label="Worker {}".format(i), c=colors[i - 1])

        plt.title("Sine calculations for a uniformly sampled random distribution")
        plt.xlabel("x")
        plt.ylabel("sine(x)")
        plt.legend(loc="lower right")
        plt.savefig("tutorial_sines.png")
```

With these changes in place, our libEnsemble code can be run with MPI by

```
mpirun -n 5 python calling.py
```

where `-n 5` tells `mpirun` to produce five processes, one of which will be the manager process with the libEnsemble manager and the other four will run libEnsemble workers.

This tutorial is only a tiny demonstration of the parallelism capabilities of libEnsemble. libEnsemble has been developed primarily to support research on High-Performance computers, with potentially hundreds of workers performing calculations simultaneously. Please read our `platform guides` for introductions to using libEnsemble on many such machines.

libEnsemble's Executors can launch non-Python user applications and simulations across allocated compute resources. Try out this feature with a more-complicated libEnsemble use-case within our `Electrostatic Forces tutorial`.

6.2.2 Executor with Electrostatic Forces

This tutorial highlights libEnsemble's capability to portably execute and monitor external scripts or user applications within simulation or generator functions using the `executor`.

This tutorial's calling script registers a compiled executable that simulates electrostatic forces between a collection of particles. The simulator function launches instances of this executable and reads output files to determine the result.

This tutorial uses libEnsemble's `MPI Executor`, which automatically detects available MPI runners and resources.

This example also uses a persistent generator. This generator runs on a worker throughout the ensemble, producing new simulation parameters as requested.

Getting Started

The simulation source code `forces.c` can be obtained directly from the libEnsemble repository in the forces_app directory.

Assuming MPI and its C compiler `mpicc` are available, compile `forces.c` into an executable (`forces.x`) with:

```
mpicc -O3 -o forces.x forces.c -lm
```

Alternative build lines for different platforms can be found in the `build_forces.sh` file in the same directory.

Calling Script

Complete scripts for this example can be found in the forces_simple directory.

Let's begin by writing our calling script to specify our simulation and generation functions and call libEnsemble. Create a Python file called *run_libe_forces.py* containing:

```
import os
import sys

import numpy as np
from forces_simf import run_forces  # Sim func from current dir

from libensemble import Ensemble
from libensemble.alloc_funcs.start_only_persistent import only_persistent_gens as alloc_f
from libensemble.executors import MPIExecutor
from libensemble.gen_funcs.persistent_sampling import persistent_uniform as gen_f
from libensemble.specs import AllocSpecs, ExitCriteria, GenSpecs, LibeSpecs, SimSpecs

if __name__ == "__main__":
    # Initialize MPI Executor
    exctr = MPIExecutor()

    # Register simulation executable with executor
    sim_app = os.path.join(os.getcwd(), "../forces_app/forces.x")

    if not os.path.isfile(sim_app):
```

(continues on next page)

(continued from previous page)

```
        sys.exit("forces.x not found - please build first in ../forces_app dir")

    exctr.register_app(full_path=sim_app, app_name="forces")

    # Parse number of workers, comms type, etc. from arguments
    ensemble = Ensemble(parse_args=True, executor=exctr)
```

We first instantiate our `MPI Executor`. Registering an application is as easy as providing the full file-path and giving it a memorable name. This Executor will later be used within our simulation function to launch the registered app.

The last line above initializes the ensemble. The `parse_args` parameter is used to read *comms* and *nworkers* from the command line. This sets the respective *libE_specs* options.

Next, we will add basic configuration for the ensemble. As one worker will run a persistent generator, we calculate the number of workers that need resources to run simulations. We also set *sim_dirs_make* so that a directory is created for each simulation. This helps organize output and also helps prevent workers from overwriting previous results.

```
    nsim_workers = ensemble.nworkers - 1  # One worker is for persistent generator

    # Persistent gen does not need resources
    ensemble.libE_specs = LibeSpecs(
        num_resource_sets=nsim_workers, sim_dirs_make=True, ensemble_dir_path="./test_
↪executor_forces_tutorial"
    )
```

Next we define the `sim_specs` and `gen_specs`. Recall that these are used to specify to libEnsemble what user functions and input/output fields to expect, and also to parameterize user functions:

```
    ensemble.sim_specs = SimSpecs(
        sim_f=run_forces,
        inputs=["x"],
        outputs=[("energy", float)],
    )

    ensemble.gen_specs = GenSpecs(
        gen_f=gen_f,
        inputs=[],  # No input when starting persistent generator
        persis_in=["sim_id"],  # Return sim_ids of evaluated points to generator
        outputs=[("x", float, (1,))],
        user={
            "initial_batch_size": nsim_workers,
            "lb": np.array([1000]),  # min particles
            "ub": np.array([3000]),  # max particles
        },
    )  # gen_specs_end_tag
```

Next, configure an allocation function, which starts the one persistent generator and farms out the simulations. We also tell it to wait for all simulations to return their results, before generating more parameters.

```
    ensemble.alloc_specs = AllocSpecs(
        alloc_f=alloc_f,
        user={
            "async_return": False,  # False causes batch returns
```

(continues on next page)

(continued from previous page)

```
        },
    )
```

Now we set `exit_criteria` to exit after running eight simulations.

We also give each worker a seeded random stream, via the `persis_info` option. These can be used for random number generation if required.

Finally we `run` the ensemble.

```
    # Instruct libEnsemble to exit after this many simulations
    ensemble.exit_criteria = ExitCriteria(sim_max=8)

    # Seed random streams for each worker, particularly for gen_f
    ensemble.add_random_streams()

    # Run ensemble
    ensemble.run()
```

Exercise

This may take some additional browsing of the docs to complete.

Write an alternative Calling Script similar to above, but with the following differences:

1. Set `libEnsemble's logger` to print debug messages.
2. Override the MPIExecutor's detected MPI runner with `"openmpi"`.
3. Tell the allocation function to return results to the generator asynchronously.
4. Use the ensemble function `save_output()` to save the History array and `persis_info` to files after libEnsemble completes.

Click Here for Solutions

Soln 1. Debug logging gives lots of information.

```
from libensemble import Ensemble, logger
from libensemble.alloc_funcs.start_only_persistent import only_persistent_gens␣
↪as alloc_f
from libensemble.executors import MPIExecutor
from libensemble.gen_funcs.persistent_sampling import persistent_uniform as␣
↪gen_f
from libensemble.specs import AllocSpecs, ExitCriteria, GenSpecs, LibeSpecs,␣
↪SimSpecs

logger.set_level("DEBUG")
```

Soln 2. This can also be specified via `platform_specs` option.

```
    # Initialize MPI Executor
    exctr = MPIExecutor(custom_info={"mpi_runner": "openmpi"})
```

Soln 3. Set `async_return` to *True* in the allocation .

```
    # Starts one persistent generator. Simulated values are returned in batch.
    ensemble.alloc_specs = AllocSpecs(
        alloc_f=alloc_f,
        user={
            "async_return": True,
        },
    )
```

Soln 4. End your script in the following manner to save the output based on the name of the calling script. You can give any string in place of `__file__`.

```
    # Run ensemble
    ensemble.run()

    ensemble.save_output(__file__)
```

Simulation Function

Our simulation function is where we'll use libEnsemble's executor to configure and submit our application for execution. We'll poll this task's state while it runs, and once we've detected it has finished we'll send any results or exit statuses back to the manager.

Create another Python file named `forces_simf.py` containing the following for starters:

```
import numpy as np

# Optional status codes to display in libE_stats.txt for each gen or sim
from libensemble.message_numbers import TASK_FAILED, WORKER_DONE


def run_forces(H, persis_info, sim_specs, libE_info):
    """Runs the forces MPI application"""

    calc_status = 0

    # Parse out num particles, from generator function
    particles = str(int(H["x"][0][0]))

    # app arguments: num particles, timesteps, also using num particles as seed
    args = particles + " " + str(10) + " " + particles

    # Retrieve our MPI Executor
    exctr = libE_info["executor"]

    # Submit our forces app for execution.
    task = exctr.submit(app_name="forces", app_args=args)

    # Block until the task finishes
    task.wait()
```

We retrieve the generated number of particles from `H` and construct an argument string for our launched application. The particle count doubles up as a random number seed here.

We then retrieve our previously instantiated Executor. libEnsemble will use the MPI runner detected (or provided by `platform options`). As *num_procs* (or similar) is not specified, libEnsemble will assign the processors available to this worker.

After submitting the "forces" app for execution, a `Task` object is returned that correlates with the launched app. This object is roughly equivalent to a Python future and can be polled, killed, and evaluated in a variety of helpful ways. For now, we're satisfied with waiting for the task to complete via `task.wait()`.

We can assume that afterward, any results are now available to parse. Our application produces a `forces.stat` file that contains either energy computations for every timestep or a "kill" message if particles were lost, which indicates a bad run - this can be ignored for now.

To complete our simulation function, parse the last energy value from the output file into a local output `History array`, and if successful, set the simulation function's exit status `calc_status` to `WORKER_DONE`. Otherwise, send back `NAN` and a `TASK_FAILED` status:

```
    # Try loading final energy reading, set the sim's status
    statfile = "forces.stat"
    try:
        data = np.loadtxt(statfile)
        final_energy = data[-1]
        calc_status = WORKER_DONE
    except Exception:
        final_energy = np.nan
        calc_status = TASK_FAILED

    # Define our output array, populate with energy reading
    output = np.zeros(1, dtype=sim_specs["out"])
    output["energy"] = final_energy

    # Return final information to worker, for reporting to manager
    return output, persis_info, calc_status
```

`calc_status` will be displayed in the `libE_stats.txt` log file.

That's it! As can be seen, with libEnsemble, it's relatively easy to get started with launching applications.

Running the example

This completes our calling script and simulation function. Run libEnsemble with:

```
python run_libe_forces.py --comms local --nworkers [nworkers]
```

where `nworkers` is one more than the number of concurrent simulations.

Output files (including `forces.stat` and files containing `stdout` and `stderr` content for each task) should appear in the current working directory. Overall workflow information should appear in `libE_stats.txt` and `ensemble.log` as usual.

Example run / output

For example, after running:

```
python run_libe_forces.py --comms local --nworkers 3
```

my `libE_stats.txt` resembled:

```
Manager     : Starting ensemble at: 2023-09-12 18:12:08.517
Worker     2: sim_id     0: sim Time: 0.205 Start: ... End: ... Status: Completed
Worker     3: sim_id     1: sim Time: 0.284 Start: ... End: ... Status: Completed
Worker     2: sim_id     2: sim Time: 0.117 Start: ... End: ... Status: Completed
Worker     3: sim_id     3: sim Time: 0.294 Start: ... End: ... Status: Completed
Worker     2: sim_id     4: sim Time: 0.124 Start: ... End: ... Status: Completed
Worker     3: sim_id     5: sim Time: 0.174 Start: ... End: ... Status: Completed
Worker     3: sim_id     7: sim Time: 0.135 Start: ... End: ... Status: Completed
Worker     2: sim_id     6: sim Time: 0.275 Start: ... End: ... Status: Completed
Worker     1: Gen no     1: gen Time: 1.038 Start: ... End: ... Status: Persis gen␣
↪finished
Manager     : Exiting ensemble at: 2023-09-12 18:12:09.565 Time Taken: 1.048
```

where `status` is set based on the simulation function's returned `calc_status`.

My `ensemble.log` (on a four-core laptop) resembled:

```
[0]  ... libensemble.libE (INFO): Logger initializing: [workerID] precedes each line.␣
↪[0] = Manager
[0]  ... libensemble.libE (INFO): libE version v0.10.2+dev
[0]  ... libensemble.manager (INFO): Manager initiated on node shuds
[0]  ... libensemble.manager (INFO): Manager exit_criteria: {'sim_max': 8}
[2]  ... libensemble.worker (INFO): Worker 2 initiated on node shuds
[3]  ... libensemble.worker (INFO): Worker 3 initiated on node shuds
[1]  ... libensemble.worker (INFO): Worker 1 initiated on node shuds
[2]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker2_0: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 2023 10 2023
[3]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker3_0: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 2900 10 2900
[2]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker2_0 finished␣
↪with errcode 0 (FINISHED)
[3]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker3_0 finished␣
↪with errcode 0 (FINISHED)
[2]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker2_1: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 1288 10 1288
[3]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker3_1: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 2897 10 2897
[2]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker2_1 finished␣
↪with errcode 0 (FINISHED)
[3]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker3_1 finished␣
↪with errcode 0 (FINISHED)
[2]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker2_2: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 1623 10 1623
[3]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker3_2: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 1846 10 1846
[2]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker2_2 finished␣
```

(continues on next page)

(continued from previous page)

```
↪with errcode 0 (FINISHED)
[3]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker3_2 finished␣
↪with errcode 0 (FINISHED)
[2]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker2_3: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 2655 10 2655
[3]  ... libensemble.executors.mpi_executor (INFO): Launching task libe_task_forces_
↪worker3_3: mpirun -hosts shuds -np 2 --ppn 2 /home/.../forces_app/forces.x 1818 10 1818
[3]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker3_3 finished␣
↪with errcode 0 (FINISHED)
[2]  ... libensemble.executors.executor (INFO): Task libe_task_forces_worker2_3 finished␣
↪with errcode 0 (FINISHED)
[0]  ... libensemble.manager (INFO): Term test tripped: sim_max
[0]  ... libensemble.manager (INFO): Term test tripped: sim_max
[0]  ... libensemble.libE (INFO): Manager total time: 1.043
```

Note again that the four cores were divided equally among two workers that run simulations.

That concludes this tutorial. Each of these example files can be found in the repository in examples/tutorials/forces_with_executor.

For further experimentation, we recommend trying out this libEnsemble tutorial workflow on a cluster or multi-node system, since libEnsemble can also manage those resources and is developed to coordinate computations at huge scales. See `HPC platform guides` for more information.

See the `forces_gpu tutorial` for a similar workflow including GPUs. That tutorial also shows how to dynamically assign resources to each simulation.

Please feel free to contact us or open an issue on GitHub if this tutorial workflow doesn't work properly on your cluster or other compute resource.

Exercises

These may require additional browsing of the documentation to complete.

1. Adjust `submit()` to launch with four processes.
2. Adjust `submit()` again so the app's `stdout` and `stderr` are written to `stdout.txt` and `stderr.txt` respectively.
3. Add a fourth argument to the args line to make 20% of simulations go bad.
4. Construct a `while not task.finished:` loop that periodically sleeps for a tenth of a second, calls `task.poll()`, then reads the output `.stat` file, and calls `task.kill()` if the output file contains `"kill\n"` or if `task.runtime` exceeds sixty seconds.

Click Here for Solution

Showing updated sections only (`---` refers to snips where code is unchanged).

```
import time

...
args = particles + " " + str(10) + " " + particles + " " + str(0.2)
...
```

(continues on next page)

(continued from previous page)

```
statfile = "forces.stat"
task = exctr.submit(
    app_name="forces",
    app_args=args,
    num_procs=4,
    stdout="stdout.txt",
    stderr="stderr.txt",
)

while not task.finished:
    time.sleep(0.1)
    task.poll()

    if task.file_exists_in_workdir(statfile):
        with open(statfile, "r") as f:
            if "kill\n" in f.readlines():
                task.kill()

    if task.runtime > 60:
        task.kill()

...
```

6.2.3 Executor - Assign GPUs

This tutorial shows the most portable way to assign tasks (user applications) to the GPU. The libEnsemble scripts in this example are available under forces_gpu in the libEnsemble repository.

This example is based on the `simple forces tutorial` with a slightly modified simulation function (to assign GPUs) and a greatly increased number of particles (allows live GPU usage to be viewed).

In the first example, each worker will be using one GPU. The code will assign the GPUs available to each worker, using the appropriate method. This works on systems using **Nvidia**, **AMD**, and **Intel** GPUs without modifying the scripts.

A video demonstrates running this example on Frontier.

Simulation function

The `sim_f` (`forces_simf.py`) is as follows. The lines that are different from the simple forces example are highlighted:

```
import numpy as np

# Optional status codes to display in libE_stats.txt for each gen or sim
from libensemble.message_numbers import TASK_FAILED, WORKER_DONE

# Optional - to print GPU settings
from libensemble.tools.test_support import check_gpu_setting


def run_forces(H, persis_info, sim_specs, libE_info):
    """Launches the forces MPI app and auto-assigns ranks and GPU resources.
```

(continues on next page)

(continued from previous page)

```
12
13      Assigns one MPI rank to each GPU assigned to the worker.
14      """
15
16      calc_status = 0
17
18      # Parse out num particles, from generator function
19      particles = str(int(H["x"][0][0]))
20
21      # app arguments: num particles, timesteps, also using num particles as seed
22      args = particles + " " + str(10) + " " + particles
23
24      # Retrieve our MPI Executor
25      exctr = libE_info["executor"]
26
27      # Submit our forces app for execution.
28      task = exctr.submit(
29          app_name="forces",
30          app_args=args,
31          auto_assign_gpus=True,
32          match_procs_to_gpus=True,
33      )
34
35      # Block until the task finishes
36      task.wait()
37
38      # Optional - prints GPU assignment (method and numbers)
39      check_gpu_setting(task, assert_setting=False, print_setting=True)
40
41      # Try loading final energy reading, set the sim's status
42      statfile = "forces.stat"
43      try:
44          data = np.loadtxt(statfile)
45          final_energy = data[-1]
46          calc_status = WORKER_DONE
47      except Exception:
48          final_energy = np.nan
49          calc_status = TASK_FAILED
50
51      # Define our output array, populate with energy reading
52      output = np.zeros(1, dtype=sim_specs["out"])
53      output["energy"] = final_energy
54
55      # Return final information to worker, for reporting to manager
56      return output, persis_info, calc_status
```

Lines 31-32 tell the executor to use the GPUs assigned to this worker, and to match processors (MPI ranks) to GPUs.

The user can also set `num_procs` and `num_gpus` in the generator as in the forces_gpu_var_resources example, and skip lines 31-32.

Line 37 simply prints out how the GPUs were assigned. If this is not as expected, `platform configuration` can be provided.

While this is sufficient for most users, note that it is possible to query the resources assigned to *this* worker (nodes and partitions of nodes), and use this information however you want.

How to query this worker's resources

The example shown below implements a similar, but less portable, version of the above (excluding output lines).

```
import numpy as np

# To retrieve our MPI Executor and resources instances
from libensemble.executors.executor import Executor
from libensemble.resources.resources import Resources

# Optional status codes to display in libE_stats.txt for each gen or sim
from libensemble.message_numbers import WORKER_DONE, TASK_FAILED


def run_forces(H, _, sim_specs):
    calc_status = 0

    # Parse out num particles, from generator function
    particles = str(int(H["x"][0][0]))

    # app arguments: num particles, timesteps, also using num particles as seed
    args = particles + " " + str(10) + " " + particles

    # Retrieve our MPI Executor instance and resources
    exctr = Executor.executor
    resources = Resources.resources.worker_resources

    resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")

    # Submit our forces app for execution. Block until the task starts.
    task = exctr.submit(
        app_name="forces",
        app_args=args,
        num_nodes=resources.local_node_count,
        procs_per_node=resources.slot_count,
        wait_on_start=True,
    )

    # Block until the task finishes
    task.wait()

    # Stat file to check for bad runs
    statfile = "forces.stat"

    # Read final energy
    data = np.loadtxt(statfile)
    final_energy = data[-1]

    # Define our output array,  populate with energy reading
    output = np.zeros(1, dtype=sim_specs["out"])
```

(continues on next page)

(continued from previous page)

```
47      output["energy"][0] = final_energy
48
49
50  return output
```

The above code will assign a GPU to each worker on CUDA-capable systems, so long as the number of workers is chosen to fit the resources.

If you want to have one rank with multiple GPUs, then change source lines 30/31 accordingly.

The `resource` attributes used are:

- **local_node_count**: The number of nodes available to this worker
- **slot_count**: The number of slots per node for this worker

and the line:

```
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")
```

will set the environment variable `CUDA_VISIBLE_DEVICES` to match the assigned slots (partitions on the node).

Note: slots refers to the `resource sets` enumerated on a node (starting with zero). If a resource set has more than one node, then each node is considered to have slot zero. [diagram]

Note that if you are on a system that automatically assigns free GPUs on the node, then setting `CUDA_VISIBLE_DEVICES` is not necessary unless you want to ensure workers are strictly bound to GPUs. For example, on many **SLURM** systems, you can use `--gpus-per-task=1` (e.g., Perlmutter). Such options can be added to the *exctr.submit* call as `extra_args`:

```
task = exctr.submit(
...
    extra_args="--gpus-per-task=1"
)
```

Alternative environment variables can be simply substituted in `set_env_to_slots`. (e.g., `HIP_VISIBLE_DEVICES`, `ROCR_VISIBLE_DEVICES`).

Note: On some systems `CUDA_VISIBLE_DEVICES` may be overridden by other assignments such as `--gpus-per-task=1`

Compiling the Forces application

First, compile the forces application under the `forces_app` directory.

Compile **forces.x** using one of the GPU build lines in build_forces.sh or similar for your platform.

Running the example

As an example, if you have been allocated two nodes, each with four GPUs, then assign nine workers (the extra worker runs the persistent generator).

For example:

```
python run_libe_forces.py --comms local --nworkers 9
```

See `zero-resource workers` for more ways to express this.

Changing the number of GPUs per worker

If you want to have two GPUs per worker on the same system (with four GPUs per node), you could assign only four workers. You will see that two GPUs are used for each forces run.

Varying resources

A variant of this example where you may specify any number of processors and GPUs for each simulation is given in the forces_gpu_var_resources example.

In this example, when simulations are parameterized in the generator function, the `gen_specs["out"]` field `num_gpus` is set for each simulation (based on the number of particles). These values will automatically be used for each simulation (they do not need to be passed as a `sim_specs["in"]`).

Further guidance on varying the resources assigned to workers can be found under the `resource manager` section.

Multiple applications

Another variant of this example, forces_multi_app, has two applications, one that uses GPUs, and another that only uses CPUs. Dynamic resource management can manage both types of resources and assign these to the same nodes concurrently, for maximum efficiency.

Checking GPU usage

The output of *forces.x* will say if it has run on the host or device. When running libEnsemble, this can be found in the simulation directories (under the `ensemble` directory).

You can check you are running forces on the GPUs as expected by using profiling tools and/or by using a monitoring utility. For NVIDIA GPUs, for example, the **Nsight** profiler is generally available and can be run from the command line. To simply run *forces.x* stand-alone you could run:

```
nsys profile --stats=true mpirun -n 2 ./forces.x
```

To use the *nvidia-smi* monitoring tool while running, open another shell where your code is running (this may entail using *ssh* to get on to the node), and run:

```
watch -n 0.1 nvidia-smi
```

This will update GPU usage information every 0.1 seconds. You would need to ensure the code runs for long enough to register on the monitor, so let's try 100,000 particles:

```
mpirun -n 2 ./forces.x 100000
```

It is also recommended that you run without the profiler when using the *nvidia-smi* utility.

This can also be used when running via libEnsemble, so long as you are on the node where the forces applications are being run.

Alternative monitoring devices include `rocm-smi` (AMD) and `intel_gpu_top` (Intel). The latter does not need the *watch* command.

Example submission script

A simple example batch script for `Perlmutter` that runs 8 workers on 2 nodes:

```
#!/bin/bash
#SBATCH -J libE_small_test
#SBATCH -A <myproject>
#SBATCH -C gpu
#SBATCH --time 10
#SBATCH --nodes 2

export MPICH_GPU_SUPPORT_ENABLED=1
export SLURM_EXACT=1

python run_libe_forces.py --comms local --nworkers 9
```

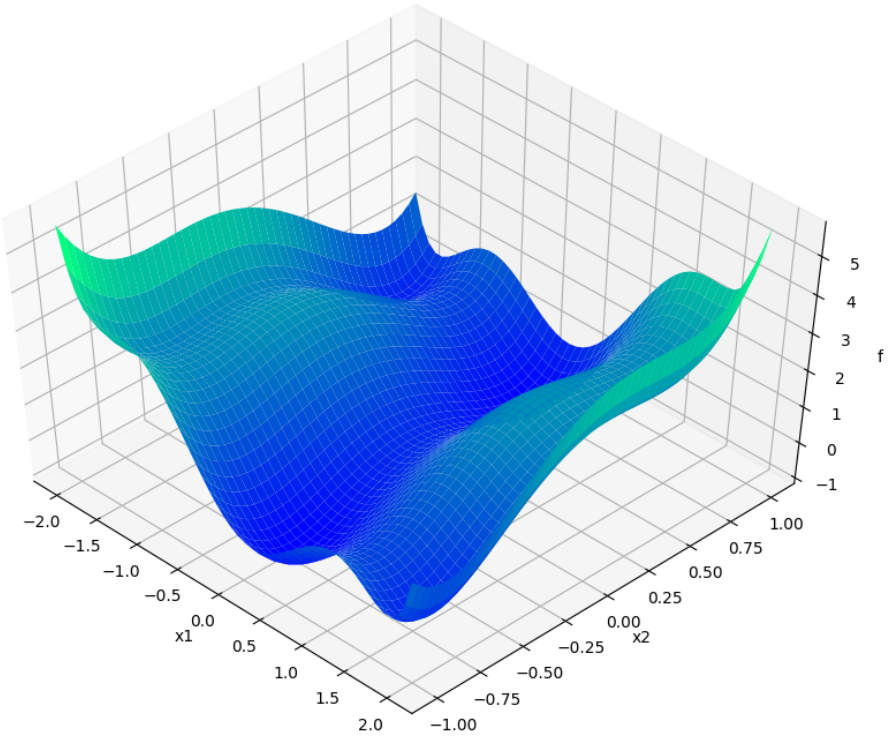
where `SLURM_EXACT` is set to help prevent resource conflicts on each node.

6.2.4 Optimization with APOSMM

This tutorial demonstrates libEnsemble's capability to identify multiple minima of simulation output using the built-in `APOSMM` (Asynchronously Parallel Optimization Solver for finding Multiple Minima) `gen_f`. In this tutorial, we'll create a simple simulation `sim_f` that defines a function with multiple minima, then write a libEnsemble calling script that imports APOSMM and parameterizes it to check for minima over a domain of outputs from our `sim_f`.

Six-Hump Camel Simulation Function

Describing APOSMM's operations is simpler with a given function on which to depict evaluations. We'll use the Six-Hump Camel function, known to have six global minima. A sample space of this function, containing all minima, appears below:



Create a new Python file named `six_hump_camel.py`. This will be our `sim_f`, incorporating the above function. Write the following:

```
import numpy as np


def six_hump_camel(H, _, sim_specs):
    """Six-Hump Camel sim_f."""

    batch = len(H["x"])  # Num evaluations each sim_f call.
    H_o = np.zeros(batch, dtype=sim_specs["out"])  # Define output array H

    for i, x in enumerate(H["x"]):
        H_o["f"][i] = three_hump_camel_func(x)  # Function evaluations placed into H

    return H_o


def six_hump_camel_func(x):
    """Six-Hump Camel function definition"""
    x1 = x[0]
    x2 = x[1]
    term1 = (4 - 2.1 * x1**2 + (x1**4) / 3) * x1**2
```

(continues on next page)

(continued from previous page)

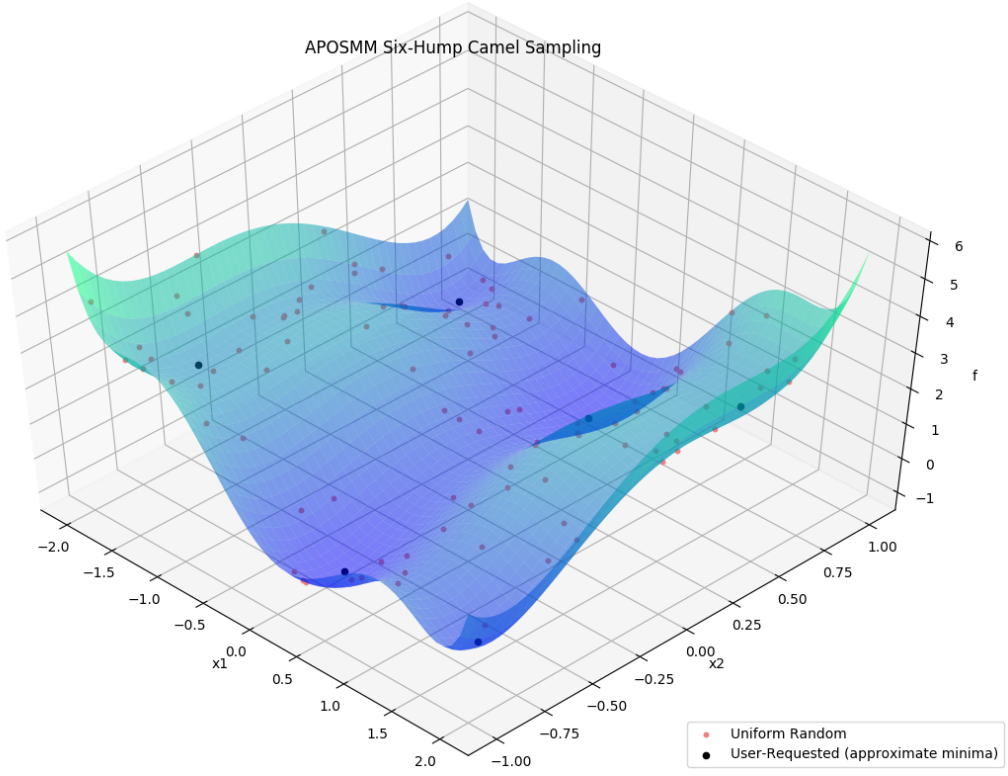
```
    term2 = x1 * x2
    term3 = (-4 + 4 * x2**2) * x2**2

    return term1 + term2 + term3
```

APOSMM Operations

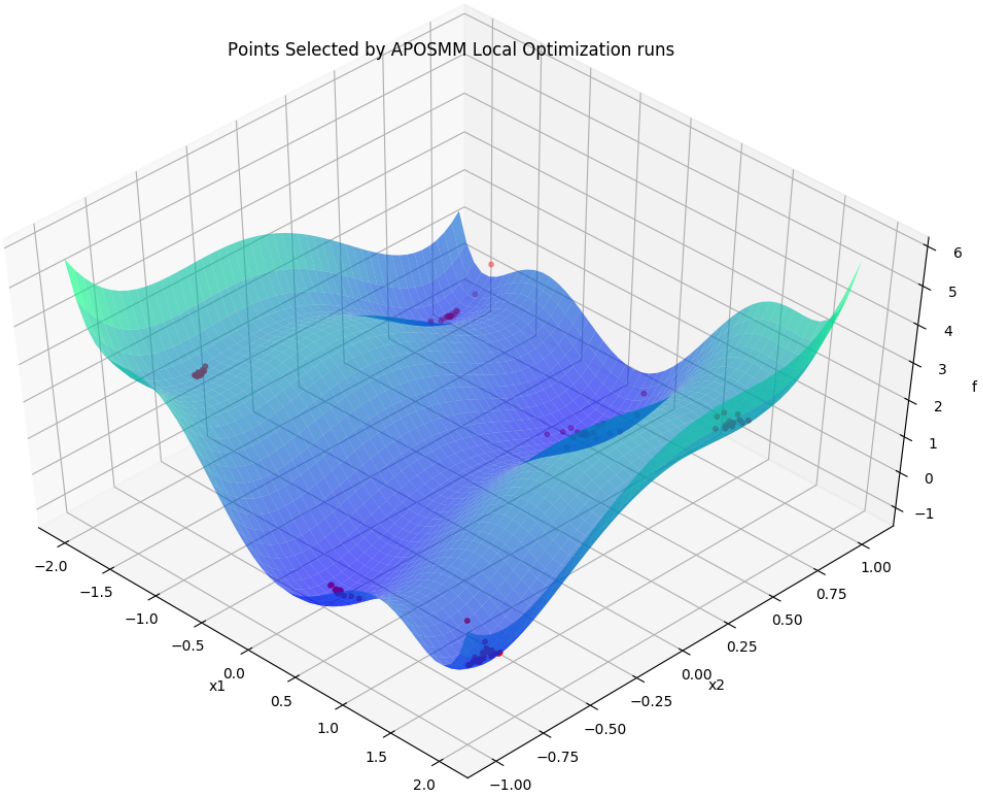
APOSMM coordinates multiple local optimization runs starting from a collection of sample points. These local optimization runs occur concurrently, and can incorporate a variety of optimization methods, including from NLopt, PETSc/TAO, SciPy, or other external scripts.

Before APOSMM can start local optimization runs, some number of uniformly sampled points must be evaluated (if no prior simulation evaluations are provided). User-requested sample points can also be provided to APOSMM:



Specifically, APOSMM will begin local optimization runs from evaluated points that don't have points with smaller function values nearby (within a threshold `r_k`). For the above example, after APOSMM receives the evaluations of the uniformly sampled points, it will begin at most `max_active_runs` local optimization runs.

As function values are returned to APOSMM, APOSMM gives them to each local optimization run in order to generate the next point(s); these are returned to the manager to be evaluated by the simulation routine. As runs complete (a minimum is found, or some termination criteria for the local optimization run is satisfied), additional local optimization runs may be started or additional uniformly sampled points may be evaluated. This continues until a `STOP_TAG` is sent by the manager, for example when the budget of simulation evaluations has been exhausted, or when a sufficiently "good" simulation output has been observed.



Throughout, generated and evaluated points are appended to the `History` array, with the field `"local_pt"` being `True` if the point is part of a local optimization run, and `"local_min"` being `True` if the point has been ruled a local minimum.

APOSMM Persistence

The most recent version of APOSMM included with libEnsemble is referred to as Persistent APOSMM. Unlike most other user functions that are initiated and completed by workers multiple times based on allocation, a single worker process initiates APOSMM so that it "persists" and keeps running over the course of the entire libEnsemble routine. APOSMM begins its own parallel evaluations and communicates points back and forth with the manager, which are then given to workers and evaluated by simulation routines.

In practice, since a single worker becomes "persistent" for APOSMM, users must ensure that enough workers or MPI ranks are initiated to support libEnsemble's manager, a persistent worker to run APOSMM, and simulation routines. The following:

```
mpiexec -n 3 python my_aposmm_routine.py
```

results in only one worker process available to perform simulation routines.

Calling Script

Create a new Python file named `my_first_aposmm.py`. Start by importing NumPy, libEnsemble routines, APOSMM, our `sim_f`, and a specialized allocation function:

```
import numpy as np

from six_hump_camel import six_hump_camel

from libensemble.libE import libE
from libensemble.gen_funcs.persistent_aposmm import aposmm
from libensemble.alloc_funcs.persistent_aposmm_alloc import persistent_aposmm_alloc
from libensemble.tools import parse_args, add_unique_random_streams
```

This allocation function starts a single Persistent APOSMM routine and provides `sim_f` output for points requested by APOSMM. Points can be sampled points or points from local optimization runs.

APOSMM supports a wide variety of external optimizers. The following statements set optimizer settings to `"scipy"` to indicate to APOSMM which optimization method to use, and help prevent unnecessary imports or package installations:

```
import libensemble.gen_funcs

libensemble.gen_funcs.rc.aposmm_optimizers = "scipy"
```

Set up `parse_args()`, our `sim_specs`, `gen_specs`, and `alloc_specs`:

```
nworkers, is_manager, libE_specs, _ = parse_args()

sim_specs = {
    "sim_f": six_hump_camel,  # Simulation function
    "in": ["x"],  # Accepts "x" values
    "out": [("f", float)],  # Returns f(x) values
}

gen_out = [
    ("x", float, 2),  # Produces "x" values
    ("x_on_cube", float, 2),  # "x" values scaled to unit cube
    ("sim_id", int),  # Produces sim_id's for History array indexing
    ("local_min", bool),  # Is a point a local minimum?
    ("local_pt", bool),  # Is a point from a local opt run?
]

gen_specs = {
    "gen_f": aposmm,  # APOSMM generator function
    "persis_in": ["f"] + [n[0] for n in gen_out],
    "out": gen_out,  # Output defined like above dict
    "user": {
        "initial_sample_size": 100,  # Random sample 100 points to start
        "localopt_method": "scipy_Nelder-Mead",
        "opt_return_codes": [0],  # Status integers specific to localopt_method
        "max_active_runs": 6,  # Occur in parallel
        "lb": np.array([-2, -1]),  # Lower bound of search domain
        "ub": np.array([2, 1]),  # Upper bound of search domain
```

(continues on next page)

(continued from previous page)

```
    },
}

alloc_specs = {"alloc_f": persistent_aposmm_alloc}
```

`gen_specs["user"]` fields above that are required for APOSMM are:

- `"lb"` - Search domain lower bound
- `"ub"` - Search domain upper bound
- `"localopt_method"` - Chosen local optimization method
- `"initial_sample_size"` - Number of uniformly sampled points generated before local optimization runs.
- `"opt_return_codes"` - A list of integers that local optimization methods return when a minimum is detected. SciPy's Nelder-Mead returns 0, but other methods (not used in this tutorial) return 1.

Also note the following:

- `gen_specs["in"]` is empty. For other `gen_f`'s this defines what fields to give to the `gen_f` when called, but here APOSMM's `alloc_f` defines those fields.
- `"x_on_cube"` in `gen_specs["out"]`. APOSMM works internally on `"x"` values scaled to the unit cube. To avoid back-and-forth scaling issues, both types of `"x"`'s are communicated back, even though the simulation will likely use `"x"` values. (APOSMM performs handshake to ensure that the `x_on_cube` that was given to be evaluated is the same the one that is given back.)
- `"sim_id"` in `gen_specs["out"]`. APOSMM produces points in its local History array that it will need to update later, and can best reference those points (and avoid a search) if APOSMM produces the IDs itself, instead of libEnsemble.

Other options and configurations for APOSMM can be found in the APOSMM `API reference`.

Set `exit_criteria` so libEnsemble knows when to complete, and `persis_info` for random sampling seeding:

```
exit_criteria = {"sim_max": 2000}
persis_info = add_unique_random_streams({}, nworkers + 1)
```

Finally, add statements to `initiate libEnsemble`, and quickly check calculated minima:

```
if __name__ == "__main__":  # required by multiprocessing on macOS and windows
    H, persis_info, flag = libE(sim_specs, gen_specs, exit_criteria, persis_info, alloc_
↪specs, libE_specs)

if is_manager:
    print("Minima:", H[np.where(H["local_min"])]["x"])
```

Final Setup, Run, and Output

If you haven't already, install SciPy so APOSMM can access the required optimization method:

```
pip install scipy
```

Finally, run this libEnsemble / APOSMM optimization routine with the following:

```
python my_first_aposmm.py --comms local --nworkers 4
```

Please note that one worker will be "persistent" for APOSMM for the duration of the routine.

After a couple seconds, the output should resemble the following:

```
[0] libensemble.libE (MANAGER_WARNING):
*********************************************************************************
User generator script will be creating sim_id.
Take care to do this sequentially.
Also, any information given back for existing sim_id values will be overwritten!
So everything in gen_specs["out"] should be in gen_specs["in"]!
*********************************************************************************

Minima: [[ 0.08993295 -0.71265804]
 [ 1.70360676 -0.79614982]
 [-1.70368421  0.79606073]
 [-0.08988064  0.71270945]
 [-1.60699361 -0.56859108]
 [ 1.60713962  0.56869567]]
```

The first section labeled `MANAGER_WARNING` is a default libEnsemble warning for generator functions that create `sim_id`'s, like APOSMM. It does not indicate a failure.

The local minima for the Six-Hump Camel simulation function as evaluated by APOSMM with libEnsemble should be listed directly below the warning.

Please see the API reference here for more APOSMM configuration options and other information.

Each of these example files can be found in the repository in examples/tutorials/aposmm.

Applications

APOSMM is not limited to evaluating minima from pure Python simulation functions. Many common libEnsemble use-cases involve using libEnsemble's `MPI Executor` to launch user applications with parameters requested by APOSMM, then evaluate their output using APOSMM, and repeat until minima are identified. A currently supported example can be found in libEnsemble's WarpX Scaling Test.

6.2.5 Calibration with Simulation Cancellation

Introduction - Calibration with libEnsemble and a Regression Model

This tutorial demonstrates libEnsemble's capability to selectively cancel pending simulations based on instructions from a calibration Generator Function. This capability is desirable, especially when evaluations are expensive, since compute resources may then be more effectively applied toward critical evaluations.

For a somewhat different approach than libEnsemble's `other tutorials`, we'll emphasize the settings, functions, and data fields within the calling script, `persistent generator`, manager, and `sim_f` that make this capability possible, rather than outlining a step-by-step process.

The libEnsemble regression test `test_persistent_surmise_calib.py` demonstrates cancellation of pending simulations, while the `test_persistent_surmise_killsims.py` test demonstrates libEnsemble's capability to also kill running simulations that have been marked as cancelled.

Overview of the Calibration Problem

The generator function featured in this tutorial can be found in `gen_funcs/persistent_surmise_calib.py` and uses the surmise library for its calibration surrogate model interface. The surmise library uses the "PCGPwM" emulation method in this example.

Say there is a computer model $f(\theta, x)$ to be calibrated. To calibrate is to find some parameter θ_0 such that $f(\theta_0, x)$ closely resembles data collected from a physical experiment. For example, a (simple) physical experiment may involve dropping a ball at different heights to study the gravitational constant, and the corresponding computer model could be the set of differential equations that govern the drop. In a case where the computation of the computer model is relatively expensive, we employ a fast surrogate model to approximate the model and to inform good parameters to test next. Here the computer model $f(\theta, x)$ is accessible only through performing `sim_f` evaluations.

As a convenience for testing, the `observed` data values are modelled by calling the `sim_f` for the known true theta, which in this case is the center of a unit hypercube. These values are therefore stored at the start of libEnsemble's main `History array` array, and have associated `sim_id`'s.

The generator function `gen_f` then samples an initial batch of parameters $(\theta_1, \ldots, \theta_n)$ and constructs a surrogate model.

For illustration, the initial batch of evaluations are arranged in the following sense:

$$\mathbf{f} = \begin{pmatrix} f(\theta_1)^\mathsf{T} \\ \vdots \\ f(\theta_n)^\mathsf{T} \end{pmatrix} = \begin{pmatrix} f(\theta_1, x_1) & \ldots & f(\theta_1, x_m) \\ \vdots & \ddots & \vdots \\ f(\theta_n, x_1) & \ldots & f(\theta_n, x_m) \end{pmatrix}.$$

The surrogate then generates (suggests) new parameters for `sim_f` evaluations, so the number of parameters n grows as more evaluations are scheduled and performed. As more evaluations are performed and received by `gen_f`, the surrogate evolves and suggests parameters closer to θ_0 with uncertainty estimates. The calibration can be terminated when either `gen_f` determines it has found θ_0 with some tolerance in the surrounding uncertainty, or computational resource runs out. At termination, the generator exits and returns, initiating the shutdown of the libEnsemble routine.

The following is a pseudocode overview of the generator. Functions directly from the calibration library used within the generator function have the `calib:` prefix. Helper functions defined to improve the data received by the calibration library by interfacing with libEnsemble have the `libE:` prefix. All other statements are workflow logic or persistent generator helper functions like `send` or `receive`:

```
libE: calculate observation values and first batch
while STOP_signal not received:
    receive: evaluated points
    unpack points into 2D Theta x Point structures
    if new model condition:
        calib: construct new model
    else:
        wait to receive more points
    if some condition:
        calib: generate new thetas from model
        calib: if error threshold reached:
            exit loop - done
        send: new points to be evaluated
    if any sent points must be obviated:
        libE: mark points with cancel request
            send: points with cancel request
```

Point Cancellation Requests and Dedicated Fields

While the generator loops and updates the model based on returned points from simulations, it detects conditionally if any new Thetas should be generated from the model, simultaneously evaluating if any *pending* simulations ought to be cancelled ("obviated"). If so, the generator then calls `cancel_columns()`:

```
if select_condition(pending):
    new_theta, info = select_next_theta(step_add_theta, cal, emu, pending, n_explore_
↪theta)
    ...
    c_obviate = info["obviatesugg"]  # suggested
    if len(c_obviate) > 0:
        cancel_columns(obs_offset, c_obviate, n_x, pending, ps)
```

`obs_offset` is an offset that excludes the observations when mapping points in surmise data structures to `sim_id`'s, `c_obviate` is a selection of columns to cancel, `n_x` is the number of `x` values, and `pending` is used to check that points marked for cancellation have not already returned. `ps` is the instantiation of the *PersistentSupport* class that is set up for persistent generators, and provides an interface for communication with the manager.

Within `cancel_columns()`, each column in `c_obviate` is iterated over, and if a point is `pending` and thus has not yet been evaluated by a simulation, its `sim_id` is appended to a list to be sent to the Manager for cancellation. Cancellation is requested using the helper function `request_cancel_sim_ids` provided by the *PersistentSupport* class. Each of these helper functions is described here. The entire `cancel_columns()` routine is listed below:

```
def cancel_columns(obs_offset, c, n_x, pending, ps):
    """Cancel columns"""
    sim_ids_to_cancel = []
    columns = np.unique(c)
    for c in columns:
        col_offset = c * n_x
        for i in range(n_x):
```

(continues on next page)

(continued from previous page)

```
            sim_id_cancel = obs_offset + col_offset + i
            if pending[i, c]:
                sim_ids_to_cancel.append(sim_id_cancel)
                pending[i, c] = 0

    ps.request_cancel_sim_ids(sim_ids_to_cancel)
```

In future calls to the allocation function by the manager, points that would have been distributed for simulation work but are now marked with "cancel_requested" will not be processed. The manager will send kill signals to workers that are already processing cancelled points. These signals can be caught and acted on by the user `sim_f`; otherwise they will be ignored.

Allocation Function and Cancellation Configuration

The allocation function used in this example is the *only_persistent_gens* function in the *start_only_persistent* module. The calling script passes the following specifications:

```
libE_specs["kill_canceled_sims"] = True

alloc_specs = {
    "alloc_f": alloc_f,
    "user": {
        "init_sample_size": init_sample_size,
        "async_return": True,
        "active_recv_gen": True,
    },
}
```

async_return tells the allocation function to return results to the generator as soon as they come back from evaluation (once the initial sample is complete).

init_sample_size gives the size of the initial sample that is batch returned to the gen. This is calculated from other parameters in the calling script.

active_recv_gen allows the persistent generator to handle irregular communications (see below).

By default, workers (including persistent workers), are only allocated work when they're in an `idle or non-active state`. However, since this generator must asynchronously update its model, the worker running this generator remains in an *active receive* state, until it becomes non-persistent. This means both the manager and persistent worker (generator in this case) must be prepared for irregular sending/receiving of data.

Calling Script - Reading Results

Within the libEnsemble calling script, once the main `libE()` function call has returned, it's a simple enough process to view the History rows that were marked as cancelled:

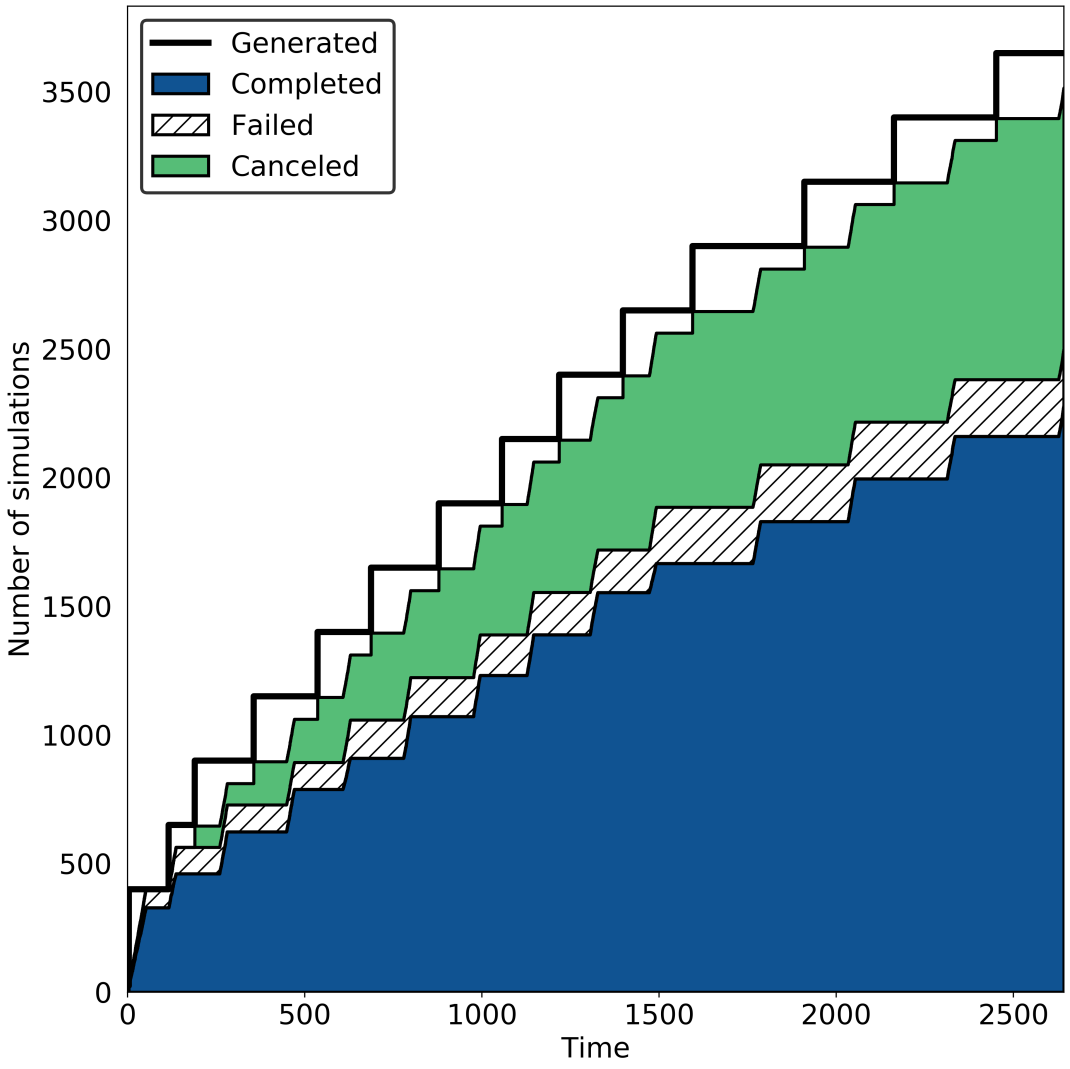
```
if __name__ == "__main__":  # required by multiprocessing on macOS and windows
    H, persis_info, flag = libE(sim_specs, gen_specs,
                                exit_criteria, persis_info,
                                alloc_specs=alloc_specs,
                                libE_specs=libE_specs)
```

(continues on next page)

(continued from previous page)

```
if is_manager:
    print("Cancelled sims", H["cancel_requested"])
```

Here's an example graph showing the relationship between scheduled, cancelled (obviated), failed, and completed simulations requested by the `gen_f`. Notice that for each batch of scheduled simulations, most either complete or fail but the rest are successfully obviated:



Please see the `test_persistent_surmise_calib.py` regression test for an example routine using the surmise calibration generator. The associated simulation function and allocation function are included in `sim_funcs/surmise_test_function.py` and `alloc_funcs/start_only_persistent.py` respectively.

Using cancellations to kill running simulations

If a generated point is cancelled by the generator before it has been given to a worker for evaluation, then it will never be given. If it has already returned from the simulation, then results can be returned, but the `cancel_requested` field remains as True. However, if the simulation is running when the manager receives the cancellation request, a kill signal will be sent to the worker. This can be caught and acted upon by a user function, otherwise it will be ignored. To demonstrate this, the test `test_persistent_surmise_killsims.py` captures and processes this signal from the manager.

In order to do this, a compiled version of the borehole function is launched by `sim_funcs/borehole_kills.py` via the `Executor`. As the borehole application used here is serial, we use the `Executor base class` rather than the commonly used `MPIExecutor` class. The base Executor submit routine simply sub-processes a serial application in-place. After the initial sample batch of evaluations has been processed, an artificial delay is added to the sub-processed borehole to allow time to receive the kill signal and terminate the application. Killed simulations will be reported at the end of the test. As this is dependent on timing, the number of killed simulations will vary between runs. This test is added simply to demonstrate the killing of running simulations and thus uses a reduced number of evaluations.

6.3 Frequently Asked Questions

If you have any additional questions, feel free to contact us through Support.

6.3.1 Debugging

We recommend using the following options to help debug workflows:

```
from libensemble import logger
logger.set_level("DEBUG")
libE_specs["safe_mode"] = True
```

6.3.2 Common Errors

"Manager only - must be at least one worker (2 MPI tasks)" when running with multiprocessing and multiple workers specified.

If your code was recently switched from MPI to multiprocessing, make sure that `libE_specs` is populated with `"comms": "local"` and `"nworkers": [int]`.

"AssertionError: alloc_f did not return any work, although all workers are idle."

This error occurs when the manager is waiting although all workers are idle. Note that a worker can be in a persistent state but is marked as idle when it has returned data to the manager and is ready to receive work.

Some possible causes of this error are:

- An MPI libEnsemble run was initiated with only one process, resulting in one manager but no workers. Similarly, the error may arise when running with only two processes when using a persistent generator. The generator will occupy one worker, leaving none to run simulation functions.
- An error in the allocation function. For example, perhaps the allocation waiting for all requested evaluations to be returned (e.g., before starting a new generator), but this condition is not returning True even though all scheduled evaluations have returned. This can be due to incorrect implementation (e.g., it has not considered

points that are cancelled or paused or in some other state that prevents the allocation function from sending them out to workers).

- A persistent worker (usually a generator) has sent a message back to the manager but is still performing work and may return further points. In this case, consider starting the generator in `active_recv` mode. This can be specified in the allocation function and will cause the worker to maintain its active status.
- A persistent worker has requested resources that prevents any simulations from taking place. By default, persistent workers hold onto resources even when not active. This may require the worker to return from persistent mode.
- When returning points to a persistent generator (often the top code block in allocation functions). For example, `support.avail_worker_ids(persistent=EVAL_GEN_TAG)` Make sure that the `EVAL_GEN_TAG` is specified and not just `persistent=True`.

libensemble.history (MANAGER_WARNING): Giving entries in H0 back to gen. Marking entries in H0 as 'gen_informed' if 'sim_ended'.

This warning is harmless. It's saying that as the provided History array is being "reloaded" into the generator, the copy is being slightly modified.

I keep getting: "Not enough processors per worker to honor arguments." when using the Executor. Can I submit tasks to allocated processors anyway?

You may have set *enforce_worker_core_bounds* to True when setting up the Executor. Also, the resource manager can be completely disabled with:

```
libE_specs["disable_resource_manager"] = True
```

Note that the Executor `submit()` method has a parameter `hyperthreads` which will attempt to use all hyperthreads/SMT threads available if set to `True`.

FileExistsError: [Errno 17] File exists: "./ensemble"

This can happen when libEnsemble tries to create ensemble or simulation directories that already exist from previous runs. To avoid this, ensure the ensemble directory paths are unique by appending some unique value to `libE_specs["ensemble_dir_path"]`, or automatically instruct runs to operate in unique directories via `libE_specs["use_workflow_dir"] = True`.

PETSc and MPI errors with "[unset]: write_line error; fd=-1 buf=:cmd=abort exitcode=59"

with `python [test with PETSc].py --comms local --nworkers 4`

This error occurs on some platforms when using PETSc with libEnsemble in `local` (multiprocessing) mode. We believe this is due to PETSc initializing MPI before libEnsemble forks processes using multiprocessing. The recommended solution is running libEnsemble in MPI mode. An alternative solution may be using a serial build of PETSc.

Note: This error may depend on how multiprocessing handles an existing MPI communicator in a particular platform.

6.3.3 HPC Errors and Questions

Why does libEnsemble hang on certain systems when running with MPI?

Another symptom may be the manager only communicating with Worker 1. This issue may occur if matching probes, which mpi4py uses by default, are not supported by the communications fabric, like Intel's Truescale (TMI) fabric. This can be solved by switching fabrics or disabling matching probes before the MPI module is first imported.

Add these two lines BEFORE `from mpi4py import MPI`:

```
import mpi4py
mpi4py.rc.recv_mprobe = False
```

Also see https://software.intel.com/en-us/articles/python-mpi4py-on-intel-true-scale-and-omni-path-clusters.

can't open hfi unit: -1 (err=23) - [13] MPI startup(): tmi fabric is not available and fallback fabric is not enabled

This may occur on TMI when libEnsemble Python processes have been launched to a node and these, in turn, execute tasks on the node; creating too many processes for the available contexts. Note that while processes can share contexts, the system is confused by the fact that there are two phases: first libEnsemble processes and then subprocesses to run user tasks. The solution is to either reduce the number of processes running or to specify a fallback fabric through environment variables:

```
unset I_MPI_FABRICS
export I_MPI_FABRICS_LIST=tmi,tcp
export I_MPI_FALLBACK=1
```

Alternatively, libEnsemble can be run in central mode where all workers run on dedicated nodes while launching all tasks onto other nodes. To do this add a node for libEnsemble, and add `libE_specs["dedicated_mode"] = True` to your calling script.

What does "_pickle.UnpicklingError: invalid load key, "x00"." indicate?

This has been observed with the OFA fabric when using mpi4py and usually indicates MPI messages aren't being received correctly. The solution is to either switch fabric or turn off matching probes. See the answer to "Why does libEnsemble hang on certain systems when running with MPI?"

For more information see https://bitbucket.org/mpi4py/mpi4py/issues/102/unpicklingerror-on-commrecv-after-iprobe.

srun: Job ****** step creation temporarily disabled, retrying (Requested nodes are busy)

Note that this message has been observed on Perlmutter when none of the problems below are present, and is likely caused by interference with system processes that run between tasks. In this case, it may cause overhead but does not prevent correct functioning.

When running on a SLURM system, this implies that you are trying to run on a resource that is already dedicated to another task. The reason can vary, some reasons are:

- All the contexts are in use. This has occurred when using TMI fabric on clusters. See question **can't open hfi unit: -1 (err=23)** for more info.
- All the memory is assigned to the first job-step (srun application), due to a default exclusive mode scheduling policy. This has been observed on Perlmutter and SDF.

In some cases using these environment variables will stop the issue:

```
export SLURM_EXACT=1
export SLURM_MEM_PER_NODE=0
```

Alternatively, this can be resolved by limiting the memory and other resources given to each task using the `--exact` option to srun along with other relevant options. For example:

```
srun --exact -n 4 -c 1 --mem-per-cpu=4G
```

would ensure that one CPU and 4 Gigabytes of memory are assigned to each MPI process. The amount of memory should be determined by the memory on the node divided by the number of CPUs. In the executor, this can be expressed via the `extra_args` option.

If libEnsemble is sharing nodes with submitted tasks (user applications launched by workers), then you may need to do this for your launch of libEnsemble also, ensuring there are enough resources for both the libEnsemble manager and workers and the launched tasks. If this is complicated, we recommended using a `dedicated node for libEnsemble`.

6.3.4 libEnsemble Help

How can I debug specific libEnsemble processes?

This is most easily addressed when running libEnsemble locally. Try

`mpiexec -np [num processes] xterm -e "python [calling script].py"`

to launch an xterm terminal window specific to each process. Mac users will need to install xQuartz.

If running in `local` mode, try using one of the `ForkablePdb` routines in `libensemble.tools` to set breakpoints and debug similarly to `pdb`. How well this works varies by system.

```
from libensemble.tools import ForkablePdb
ForkablePdb().set_trace()
```

Can I use the MPI Executor when running libEnsemble with multiprocessing?

Yes. The Executor type determines only how libEnsemble workers execute and interact with user applications and is *independent* of `comms` chosen for manager/worker communications.

How can I disable libEnsemble's output files?

Set `libE_specs["disable_log_files"]` to `True`.

If libEnsemble aborts on an exception, the History array and `persis_info` dictionaries will be dumped. This can be suppressed by setting `libE_specs["save_H_and_persis_on_abort"]` to `False`.

See `here` for more information about these files.

How can I silence libEnsemble or prevent printed warnings?

Some logger messages at or above the `MANAGER_WARNING` level are mirrored to stderr automatically. To disable this, set the minimum stderr displaying level to `CRITICAL` via the following:

```
from libensemble import logger
logger.set_stderr_level("CRITICAL")
```

This effectively puts libEnsemble in silent mode.

See the `Logger Configuration` docs for more information.

6.3.5 macOS and Windows Errors

Can I run libEnsemble on Windows?

Although we have run many libEnsemble workflows successfully on Windows using both MPI and local comms, Windows is not rigorously supported. We highly recommend Unix-like systems. Windows tends to produce more platform-specific issues that are difficult to reproduce and troubleshoot.

Windows - How can I run libEnsemble with MPI comms?

We have run Windows workflows with MPI comms. However, as most MPI distributions have either dropped Windows support (MPICH and Open MPI) or are no longer being maintained (`msmpi`), we cannot guarantee success.

We recommend experimenting with the many Unix-like emulators, containers, virtual machines, and other such systems. The Installing PETSc On Microsoft Windows documentation contains valuable information.

Otherwise, install `msmpi` and `mpi4py` from conda and experiment, or use `local` comms.

Windows - "A required privilege is not held by the client"

Assuming you were trying to use the `sim_dir_symlink_files` or `gen_dir_symlink_files` options, this indicates that to allow libEnsemble to create symlinks, you need to run your current `cmd` shell as administrator.

"RuntimeError: An attempt has been made to start a new process... this probably means that you are not using fork... " if __name__ == "__main__": freeze_support() ...

You need to place your main entry point code underneath an `if __name__ == "__main__":` block.

Explanation: Python chooses one of three methods to start new processes when using multiprocessing (`--comms local` with libEnsemble). These are `"fork"`, `"spawn"`, and `"forkserver"`. `"fork"` is the default on Unix, and in our experience is quicker and more reliable, but `"spawn"` is the default on Windows and macOS (See the Python multiprocessing docs).

Prior to libEnsemble v0.9.2, if libEnsemble detected macOS, it would automatically switch the multiprocessing method to `"fork"`. We decided to stop doing this to avoid overriding defaults and compatibility issues with some libraries.

If you'd prefer to use `"fork"` or not reformat your code, you can set the multiprocessing start method by placing the following near the top of your calling script:

```
import multiprocessing
multiprocessing.set_start_method("fork", force=True)
```

"macOS - Fatal error in MPI_Init_thread: Other MPI error, error stack: ... gethostbyname failed"

Resolve this by appending `127.0.0.1 [your hostname]` to /etc/hosts. Unfortunately, `127.0.0.1 localhost` isn't satisfactory for preventing this.

macOS - How do I stop the Firewall Security popups when running with the Executor?

There are several ways to address this nuisance, but all involve trial and error. An easy (but insecure) solution is temporarily disabling the firewall through System Preferences -> Security & Privacy -> Firewall -> Turn Off Firewall. Alternatively, adding a firewall "Allow incoming connections" rule can be attempted for the offending executable. We've had limited success running `sudo codesign --force --deep --sign - /path/to/application.app` on our executables, then confirming the next alerts for the executable and `mpiexec.hydra`.

6.4 Known Issues

The following selection describes known bugs, errors, or other difficulties that may occur when using libEnsemble.

- Platforms using SLURM version 23.02 experience a pickle error when using `mpi4py` comms. Disabling matching probes via the environment variable `export MPI4PY_RC_RECV_MPROBE=0` or adding `mpi4py.rc.recv_mprobe = False` at the top of the calling script should resolve this error. If using the MPI executor and multiple workers per node, some users may experience failed applications with the message `srun: error: CPU binding outside of job step allocation, allocated` in the application's standard error. This is being investigated. If this happens we recommend using `local` comms in place of `mpi4py`.
- When using the Executor: OpenMPI does not work with direct MPI task submissions in mpi4py comms mode, since OpenMPI does not support nested MPI executions. Use either `local` mode or the Balsam Executor instead.
- Local comms mode (multiprocessing) may fail if MPI is initialized before forking processors. This is thought to be responsible for issues combining multiprocessing with PETSc on some platforms.
- Remote detection of logical cores via `LSB_HOSTS` (e.g., Summit) returns the number of physical cores as SMT info not available.
- TCP mode does not support (1) more than one libEnsemble call in a given script or (2) the auto-resources option to the Executor.
- libEnsemble may hang on systems with matching probes not enabled on the native fabric, like on Intel's Truescale (TMI) fabric for instance. See the FAQ for more information.
- We currently recommended running in Central mode on Bridges as distributed runs are experiencing hangs.

6.5 Example User Functions and Calling Scripts

Here we give example generation, simulation, and allocation functions for libEnsemble, as well as example calling scripts.

Additional examples from libEnsemble's users are available in the libEnsemble Community Repository, with corresponding generator documentation available here.

6.5.1 Generator Functions

Example Compatible Packages

libEnsemble and the Community Examples repository include example generator functions for the following libraries:

- APOSMM Asynchronously parallel optimization solver for finding multiple minima. Supported local optimization routines include:
  - DFO-LS Derivative-free solver for (bound constrained) nonlinear least-squares minimization
  - NLopt Library for nonlinear optimization, providing a common interface for various methods
  - scipy.optimize Open-source solvers for nonlinear problems, linear programming, constrained and nonlinear least-squares, root finding, and curve fitting.
  - PETSc/TAO Routines for the scalable (parallel) solution of scientific applications
- DEAP Distributed evolutionary algorithms
- Distributed optimization methods for minimizing sums of convex functions. Methods include:
  - Primal-dual sliding (https://arxiv.org/pdf/2101.00143).
  - Distributed gradient descent with gradient tracking (https://arxiv.org/abs/1908.11444).
  - Proximal sliding (https://arxiv.org/abs/1406.0919).
- ECNoise Estimating Computational Noise in Numerical Simulations
- Surmise Modular Bayesian calibration/inference framework
- Tasmanian Toolkit for Adaptive Stochastic Modeling and Non-Intrusive ApproximatioN
- VTMOP Fortran package for large-scale multiobjective multidisciplinary design optimization

libEnsemble has also been used to coordinate many computationally expensive simulations. Select examples include:

- OPAL Object Oriented Parallel Accelerator Library. (See this IPAC manuscript.)
- WarpX Advanced electromagnetic particle-in-cell code. (See example WarpX + libE scripts.)

Important: See the API for generation functions `here`.

---

sampling

This module contains multiple generation functions for sampling a domain. All use (and return) a random stream in `persis_info`, given by the allocation function.

`sampling.`**`uniform_random_sample`**`(_, `*`persis_info`*`, `*`gen_specs`*`)`

Generates `gen_specs["user"]["gen_batch_size"]` points uniformly over the domain defined by `gen_specs["user"]["ub"]` and `gen_specs["user"]["lb"]`.

See also:

test_uniform_sampling.py # noqa

`sampling.`**`uniform_random_sample_with_variable_resources`**`(_, `*`persis_info`*`, `*`gen_specs`*`)`

Generates `gen_specs["user"]["gen_batch_size"]` points uniformly over the domain defined by `gen_specs["user"]["ub"]` and `gen_specs["user"]["lb"]`.

Also randomly requests a different number of resource sets to be used in each evaluation.

This generator is used to test/demonstrate setting of resource sets.

#.. seealso::

#`test_uniform_sampling_with_variable_resources.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_uniform_sampling_with_variable_resources.py>`_ # noqa

`sampling.`**`uniform_random_sample_with_var_priorities_and_resources`**(*H*, *persis_info*, *gen_specs*)

Generates points uniformly over the domain defined by `gen_specs["user"]["ub"]` and `gen_specs["user"]["lb"]`. Also, randomly requests a different priority and number of resource sets to be used in the evaluation of the generated points, after the initial batch.

This generator is used to test/demonstrate setting of priorities and resource sets.

`sampling.`**`uniform_random_sample_obj_components`**(*H*, *persis_info*, *gen_specs*)

Generates points uniformly over the domain defined by `gen_specs["user"]["ub"]` and `gen_specs["user"]["lb"]` but requests each `obj_component` be evaluated separately.

See also:

test_uniform_sampling_one_residual_at_a_time.py # noqa

`sampling.`**`latin_hypercube_sample`**(*_*, *persis_info*, *gen_specs*)

Generates `gen_specs["user"]["gen_batch_size"]` points in a Latin hypercube sample over the domain defined by `gen_specs["user"]["ub"]` and `gen_specs["user"]["lb"]`.

See also:

test_1d_sampling.py # noqa

`sampling.`**`uniform_random_sample_cancel`**(*_*, *persis_info*, *gen_specs*)

Similar to uniform_random_sample but with immediate cancellation of selected points for testing.

sampling.py

```
"""
This module contains multiple generation functions for sampling a domain. All
use (and return) a random stream in ``persis_info``, given by the allocation
function.
"""
import numpy as np

__all__ = [
    "uniform_random_sample",
    "uniform_random_sample_with_variable_resources",
    "uniform_random_sample_with_var_priorities_and_resources",
    "uniform_random_sample_obj_components",
    "latin_hypercube_sample",
    "uniform_random_sample_cancel",
]


def uniform_random_sample(_, persis_info, gen_specs):
    """
    Generates ``gen_specs["user"]["gen_batch_size"]`` points uniformly over the domain
    defined by ``gen_specs["user"]["ub"]`` and ``gen_specs["user"]["lb"]``.
```

(continues on next page)

(continued from previous page)

```
    .. seealso::
        `test_uniform_sampling.py <https://github.com/Libensemble/libensemble/blob/
↪develop/libensemble/tests/functionality_tests/test_uniform_sampling.py>`_ # noqa
    """
    ub = gen_specs["user"]["ub"]
    lb = gen_specs["user"]["lb"]

    n = len(lb)
    b = gen_specs["user"]["gen_batch_size"]

    H_o = np.zeros(b, dtype=gen_specs["out"])

    H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))

    return H_o, persis_info


def uniform_random_sample_with_variable_resources(_, persis_info, gen_specs):
    """
    Generates ``gen_specs["user"]["gen_batch_size"]`` points uniformly over the domain
    defined by ``gen_specs["user"]["ub"]`` and ``gen_specs["user"]["lb"]``.

    Also randomly requests a different number of resource sets to be used in each␣
↪evaluation.

    This generator is used to test/demonstrate setting of resource sets.

    #.. seealso::
        #`test_uniform_sampling_with_variable_resources.py <https://github.com/
↪Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_
↪uniform_sampling_with_variable_resources.py>`_ # noqa
    """

    ub = gen_specs["user"]["ub"]
    lb = gen_specs["user"]["lb"]
    max_rsets = gen_specs["user"]["max_resource_sets"]

    n = len(lb)
    b = gen_specs["user"]["gen_batch_size"]

    H_o = np.zeros(b, dtype=gen_specs["out"])

    H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))
    H_o["resource_sets"] = persis_info["rand_stream"].integers(1, max_rsets + 1, b)

    print(f'GEN: H rsets requested: {H_o["resource_sets"]}')

    return H_o, persis_info


def uniform_random_sample_with_var_priorities_and_resources(H, persis_info, gen_specs):
```

(continues on next page)

(continued from previous page)

```
    """
    Generates points uniformly over the domain defined by ``gen_specs["user"]["ub"]`` and
    ``gen_specs["user"]["lb"]``. Also, randomly requests a different priority and number of
    resource sets to be used in the evaluation of the generated points, after the initial batch.

    This generator is used to test/demonstrate setting of priorities and resource sets.

    """
    ub = gen_specs["user"]["ub"]
    lb = gen_specs["user"]["lb"]
    max_rsets = gen_specs["user"]["max_resource_sets"]

    n = len(lb)

    if len(H) == 0:
        b = gen_specs["user"]["initial_batch_size"]

        H_o = np.zeros(b, dtype=gen_specs["out"])
        for i in range(0, b):
            # x= i*np.ones(n)
            x = persis_info["rand_stream"].uniform(lb, ub, (1, n))
            H_o["x"][i] = x
            H_o["resource_sets"][i] = 1
            H_o["priority"] = 1

    else:
        H_o = np.zeros(1, dtype=gen_specs["out"])
        # H_o["x"] = len(H)*np.ones(n)  # Can use a simple count for testing.
        H_o["x"] = persis_info["rand_stream"].uniform(lb, ub)
        H_o["resource_sets"] = persis_info["rand_stream"].integers(1, max_rsets + 1)
        H_o["priority"] = 10 * H_o["resource_sets"]
        # print("Created sim for {} resource sets".format(H_o["resource_sets"]), flush=True)

    return H_o, persis_info


def uniform_random_sample_obj_components(H, persis_info, gen_specs):
    """
    Generates points uniformly over the domain defined by ``gen_specs["user"]["ub"]``
    and ``gen_specs["user"]["lb"]`` but requests each ``obj_component`` be evaluated
    separately.

    .. seealso::
        `test_uniform_sampling_one_residual_at_a_time.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_uniform_sampling_one_residual_at_a_time.py>`_ # noqa
    """
    ub = gen_specs["user"]["ub"]
    lb = gen_specs["user"]["lb"]
```

(continues on next page)

(continued from previous page)

```
    n = len(lb)
    m = gen_specs["user"]["components"]
    b = gen_specs["user"]["gen_batch_size"]

    H_o = np.zeros(b * m, dtype=gen_specs["out"])
    for i in range(0, b):
        x = persis_info["rand_stream"].uniform(lb, ub, (1, n))
        H_o["x"][i * m : (i + 1) * m, :] = np.tile(x, (m, 1))
        H_o["priority"][i * m : (i + 1) * m] = persis_info["rand_stream"].uniform(0, 1,␣
↪m)
        H_o["obj_component"][i * m : (i + 1) * m] = np.arange(0, m)

        H_o["pt_id"][i * m : (i + 1) * m] = len(H) // m + i

    return H_o, persis_info


def uniform_random_sample_cancel(_, persis_info, gen_specs):
    """
    Similar to uniform_random_sample but with immediate cancellation of
    selected points for testing.

    """
    ub = gen_specs["user"]["ub"]
    lb = gen_specs["user"]["lb"]

    n = len(lb)
    b = gen_specs["user"]["gen_batch_size"]

    H_o = np.zeros(b, dtype=gen_specs["out"])
    for i in range(b):
        if i % 10 == 0:
            H_o[i]["cancel_requested"] = True

    H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))

    return H_o, persis_info


def latin_hypercube_sample(_, persis_info, gen_specs):
    """
    Generates ``gen_specs["user"]["gen_batch_size"]`` points in a Latin
    hypercube sample over the domain defined by ``gen_specs["user"]["ub"]`` and
    ``gen_specs["user"]["lb"]``.

    .. seealso::
        `test_1d_sampling.py <https://github.com/Libensemble/libensemble/blob/develop/
↪libensemble/tests/regression_tests/test_1d_sampling.py>`_ # noqa
    """

    ub = gen_specs["user"]["ub"]
```

(continues on next page)

(continued from previous page)

```
    lb = gen_specs["user"]["lb"]

    n = len(lb)
    b = gen_specs["user"]["gen_batch_size"]

    H_o = np.zeros(b, dtype=gen_specs["out"])

    A = lhs_sample(n, b, persis_info["rand_stream"])

    H_o["x"] = A * (ub - lb) + lb

    return H_o, persis_info


def lhs_sample(n, k, stream):
    # Generate the intervals and random values
    intervals = np.linspace(0, 1, k + 1)
    rand_source = stream.uniform(0, 1, (k, n))
    rand_pts = np.zeros((k, n))
    sample = np.zeros((k, n))

    # Add a point uniformly in each interval
    a = intervals[:k]
    b = intervals[1:]
    for j in range(n):
        rand_pts[:, j] = rand_source[:, j] * (b - a) + a

    # Randomly perturb
    for j in range(n):
        sample[:, j] = rand_pts[stream.permutation(k), j]

    return sample
```

persistent_sampling

Persistent generator providing points using sampling

`persistent_sampling.`**`persistent_uniform`**`(_, persis_info, gen_specs, libE_info)`

This generation function always enters into persistent mode and returns `gen_specs["initial_batch_size"]` uniformly sampled points the first time it is called. Afterwards, it returns the number of points given. This can be used in either a batch or asynchronous mode by adjusting the allocation function.

See also:

test_persistent_uniform_sampling.py test_persistent_uniform_sampling_async.py

`persistent_sampling.`**`persistent_uniform_final_update`**`(_, persis_info, gen_specs, libE_info)`

Assuming the value `"f"` returned from sim_f is stochastic, this generation is updating an estimated mean `"f_est"` of the sim_f output at each of the corners of the domain.

See also:

test_persistent_uniform_sampling_running_mean.py

persistent_sampling.**persistent_request_shutdown**(*_*, *persis_info*, *gen_specs*, *libE_info*)

This generation function is similar in structure to persistent_uniform, but uses a count to test exiting on a threshold value. This principle can be used with a supporting allocation function (e.g. start_only_persistent) to shutdown an ensemble when a condition is met.

See also:

test_persistent_uniform_gen_decides_stop.py

persistent_sampling.**uniform_nonblocking**(*_*, *persis_info*, *gen_specs*, *libE_info*)

This generation function is designed to test non-blocking receives.

See also:

test_persistent_uniform_sampling.py

persistent_sampling.**batched_history_matching**(*_*, *persis_info*, *gen_specs*, *libE_info*)

Given - sim_f with an input of x with len(x)=n - b, the batch size of points to generate - q<b, the number of best samples to use in the following iteration

Pseudocode: Let (mu, Sigma) denote a mean and covariance matrix initialized to the origin and the identity, respectively.

While true (batch synchronous for now):

> Draw b samples x_1, … , x_b from MVN(mu, Sigma) Evaluate f(x_1), … , f(x_b) and determine the set of q x_i whose f(x_i) values are smallest (breaking ties lexicographically) Update (mu, Sigma) based on the sample mean and sample covariance of these q x values.

See also:

test_persistent_uniform_sampling.py

persistent_sampling.**persistent_uniform_with_cancellations**(*_*, *persis_info*, *gen_specs*, *libE_info*)

persistent_sampling.py

```
"""Persistent generator providing points using sampling"""

import numpy as np

from libensemble.message_numbers import EVAL_GEN_TAG, FINISHED_PERSISTENT_GEN_TAG,␣
↪PERSIS_STOP, STOP_TAG
from libensemble.tools.persistent_support import PersistentSupport

__all__ = [
    "persistent_uniform",
    "persistent_uniform_final_update",
    "persistent_request_shutdown",
    "uniform_nonblocking",
    "batched_history_matching",
    "persistent_uniform_with_cancellations",
]


def _get_user_params(user_specs):
    """Extract user params"""
```

(continues on next page)

(continued from previous page)

```
    b = user_specs["initial_batch_size"]
    ub = user_specs["ub"]
    lb = user_specs["lb"]
    n = len(lb)  # dimension
    assert isinstance(b, int), "Batch size must be an integer"
    assert isinstance(n, int), "Dimension must be an integer"
    assert isinstance(lb, np.ndarray), "lb must be a numpy array"
    assert isinstance(ub, np.ndarray), "ub must be a numpy array"
    return b, n, lb, ub


def persistent_uniform(_, persis_info, gen_specs, libE_info):
    """
    This generation function always enters into persistent mode and returns
    ``gen_specs["initial_batch_size"]`` uniformly sampled points the first time it
    is called. Afterwards, it returns the number of points given. This can be
    used in either a batch or asynchronous mode by adjusting the allocation
    function.

    .. seealso::
        `test_persistent_uniform_sampling.py <https://github.com/Libensemble/libensemble/
↪blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_sampling.py>
↪`_
        `test_persistent_uniform_sampling_async.py <https://github.com/Libensemble/
↪libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_
↪sampling_async.py>`_
    """  # noqa

    b, n, lb, ub = _get_user_params(gen_specs["user"])
    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)

    # Send batches until manager sends stop tag
    tag = None
    while tag not in [STOP_TAG, PERSIS_STOP]:
        H_o = np.zeros(b, dtype=gen_specs["out"])
        H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))
        if "obj_component" in H_o.dtype.fields:
            H_o["obj_component"] = persis_info["rand_stream"].integers(
                low=0, high=gen_specs["user"]["num_components"], size=b
            )
        tag, Work, calc_in = ps.send_recv(H_o)
        if hasattr(calc_in, "__len__"):
            b = len(calc_in)

    return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG


def persistent_uniform_final_update(_, persis_info, gen_specs, libE_info):
    """
    Assuming the value ``"f"`` returned from sim_f is stochastic, this
    generation is updating an estimated mean ``"f_est"`` of the sim_f output at
    each of the corners of the domain.
```

(continues on next page)

(continued from previous page)

```
    .. seealso::
        `test_persistent_uniform_sampling_running_mean.py <https://github.com/
↪Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_
↪persistent_uniform_sampling_running_mean.py>`_
    """  # noqa

    b, n, lb, ub = _get_user_params(gen_specs["user"])
    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)

    def generate_corners(x, y):
        n = len(x)
        corner_indices = np.arange(2**n)
        corners = []
        for index in corner_indices:
            corner = [x[i] if index & (1 << i) else y[i] for i in range(n)]
            corners.append(corner)
        return corners

    def sample_corners_with_probability(corners, p, b):
        selected_corners = np.random.choice(len(corners), size=b, p=p)
        sampled_corners = [corners[i] for i in selected_corners]
        return sampled_corners, selected_corners

    corners = generate_corners(lb, ub)

    # Start with equal probabilies
    p = np.ones(2**n) / 2**n

    running_total = np.nan * np.ones(2**n)
    number_of_samples = np.zeros(2**n)
    sent = np.array([], dtype=int)

    # Send batches of `b` points until manager sends stop tag
    tag = None
    next_id = 0
    while tag not in [STOP_TAG, PERSIS_STOP]:
        H_o = np.zeros(b, dtype=gen_specs["out"])
        H_o["sim_id"] = range(next_id, next_id + b)
        next_id += b

        sampled_corners, corner_ids = sample_corners_with_probability(corners, p, b)

        H_o["corner_id"] = corner_ids
        H_o["x"] = sampled_corners
        sent = np.append(sent, corner_ids)

        tag, Work, calc_in = ps.send_recv(H_o)
        if hasattr(calc_in, "__len__"):
            b = len(calc_in)
            for row in calc_in:
                number_of_samples[row["corner_id"]] += 1
```

(continues on next page)

(continued from previous page)

```
                if np.isnan(running_total[row["corner_id"]]):
                    running_total[row["corner_id"]] = row["f"]
                else:
                    running_total[row["corner_id"]] += row["f"]

    # Having received a PERSIS_STOP, update f_est field for all points and return
    # For manager to honor final H_o return, must have set libE_specs["use_persis_return_
↪gen"] = True
    f_est = running_total / number_of_samples
    H_o = np.zeros(len(sent), dtype=[("sim_id", int), ("corner_id", int), ("f_est",␣
↪float)])
    for count, i in enumerate(sent):
        H_o["sim_id"][count] = count
        H_o["corner_id"][count] = i
        H_o["f_est"][count] = f_est[i]

    return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG


def persistent_request_shutdown(_, persis_info, gen_specs, libE_info):
    """
    This generation function is similar in structure to persistent_uniform,
    but uses a count to test exiting on a threshold value. This principle can
    be used with a supporting allocation function (e.g. start_only_persistent)
    to shutdown an ensemble when a condition is met.

    .. seealso::
        `test_persistent_uniform_gen_decides_stop.py <https://github.com/Libensemble/
↪libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_
↪gen_decides_stop.py>`_
    """  # noqa
    b, n, lb, ub = _get_user_params(gen_specs["user"])
    shutdown_limit = gen_specs["user"]["shutdown_limit"]
    f_count = 0
    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)

    # Send batches until manager sends stop tag
    tag = None
    while tag not in [STOP_TAG, PERSIS_STOP]:
        H_o = np.zeros(b, dtype=gen_specs["out"])
        H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))
        tag, Work, calc_in = ps.send_recv(H_o)
        if hasattr(calc_in, "__len__"):
            b = len(calc_in)
        f_count += b
        if f_count >= shutdown_limit:
            print("Reached threshold.", f_count, flush=True)
            break  # End the persistent gen

    return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG


```

(continues on next page)

(continued from previous page)

```python
def uniform_nonblocking(_, persis_info, gen_specs, libE_info):
    """
    This generation function is designed to test non-blocking receives.

    .. seealso::
        `test_persistent_uniform_sampling.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_sampling.py>`_
    """  # noqa
    b, n, lb, ub = _get_user_params(gen_specs["user"])
    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)

    # Send batches until manager sends stop tag
    tag = None
    while tag not in [STOP_TAG, PERSIS_STOP]:
        H_o = np.zeros(b, dtype=gen_specs["out"])
        H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))
        ps.send(H_o)

        received = False
        spin_count = 0
        while not received:
            tag, Work, calc_in = ps.recv(blocking=False)
            if tag is not None:
                received = True
            else:
                spin_count += 1

        persis_info["spin_count"] = spin_count

        if hasattr(calc_in, "__len__"):
            b = len(calc_in)

    return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG


def batched_history_matching(_, persis_info, gen_specs, libE_info):
    """
    Given
    - sim_f with an input of x with len(x)=n
    - b, the batch size of points to generate
    - q<b, the number of best samples to use in the following iteration

    Pseudocode:
    Let (mu, Sigma) denote a mean and covariance matrix initialized to the
    origin and the identity, respectively.

    While true (batch synchronous for now):

        Draw b samples x_1, ... , x_b from MVN( mu, Sigma)
        Evaluate f(x_1), ... , f(x_b) and determine the set of q x_i whose f(x_i) values␣
↪are smallest (breaking ties lexicographically)
```

(continues on next page)

(continued from previous page)

```
        Update (mu, Sigma) based on the sample mean and sample covariance of these q x values.

    .. seealso::
        `test_persistent_uniform_sampling.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_sampling.py>`_
    """  # noqa
    lb = gen_specs["user"]["lb"]

    n = len(lb)
    b = gen_specs["user"]["initial_batch_size"]
    q = gen_specs["user"]["num_best_vals"]
    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)

    mu = np.zeros(n)
    Sigma = np.eye(n)
    tag = None

    while tag not in [STOP_TAG, PERSIS_STOP]:
        H_o = np.zeros(b, dtype=gen_specs["out"])
        H_o["x"] = persis_info["rand_stream"].multivariate_normal(mu, Sigma, b)

        # Send data and get next assignment
        tag, Work, calc_in = ps.send_recv(H_o)
        if calc_in is not None:
            all_inds = np.argsort(calc_in["f"])
            best_inds = all_inds[:q]
            mu = np.mean(H_o["x"][best_inds], axis=0)
            Sigma = np.cov(H_o["x"][best_inds].T)

    return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG


def persistent_uniform_with_cancellations(_, persis_info, gen_specs, libE_info):
    ub = gen_specs["user"]["ub"]
    lb = gen_specs["user"]["lb"]
    n = len(lb)
    b = gen_specs["user"]["initial_batch_size"]

    # Start cancelling points from half initial batch onward
    cancel_from = b // 2  # Should get at least this many points back

    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)

    # Send batches until manager sends stop tag
    tag = None
    while tag not in [STOP_TAG, PERSIS_STOP]:
        H_o = np.zeros(b, dtype=gen_specs["out"])
        H_o["x"] = persis_info["rand_stream"].uniform(lb, ub, (b, n))
        tag, Work, calc_in = ps.send_recv(H_o)

```

(continues on next page)

(continued from previous page)

```
        if hasattr(calc_in, "__len__"):
            b = len(calc_in)

            # Cancel as many points as got back
            cancel_ids = list(range(cancel_from, cancel_from + b))
            cancel_from += b
            ps.request_cancel_sim_ids(cancel_ids)

    return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG
```

persistent_sampling_var_resources

Persistent random sampling using various methods of dynamic resource assignment

Each function generates points uniformly over the domain defined by `gen_specs["user"]["ub"]` and `gen_specs["user"]["lb"]`.

Most functions use a random request of resources over a range, setting num_procs, num_gpus, or resource sets. The function `uniform_sample_with_var_gpus` uses the `x` value to determine the number of GPUs requested.

persistent_sampling_var_resources.**uniform_sample**(*_*, *persis_info*, *gen_specs*, *libE_info*)

Randomly requests a different number of resource sets to be used in the evaluation of the generated points.

See also:

test_uniform_sampling_with_variable_resources.py

persistent_sampling_var_resources.**uniform_sample_with_procs_gpus**(*_*, *persis_info*, *gen_specs*, *libE_info*)

Randomly requests a different number of processors and gpus to be used in the evaluation of the generated points.

See also:

test_GPU_variable_resources.py

persistent_sampling_var_resources.**uniform_sample_with_var_priorities**(*_*, *persis_info*, *gen_specs*, *libE_info*)

Initial batch has matching priorities, after which a different number of resource sets and priorities are requested for each point.

persistent_sampling_var_resources.**uniform_sample_diff_simulations**(*_*, *persis_info*, *gen_specs*, *libE_info*)

Randomly requests a different number of processors for each simulation. One simulation type also uses GPUs.

See also:

test_GPU_variable_resources_multi_task.py

persistent_sampling_var_resources.**uniform_sample_with_sim_gen_resources**(*_*, *persis_info*, *gen_specs*, *libE_info*)

Randomly requests a different number of processors and gpus to be used in the evaluation of the generated points.

See also:

test_GPU_variable_resources.py

APOSMM

Asynchronously Parallel Optimization Solver for finding Multiple Minima (APOSMM) coordinates concurrent local optimization runs in order to identify many local minima.

Required: mpmath, SciPy

Optional (see below): petsc4py, nlopt, DFO-LS

Configuring APOSMM

APOSMM works with a choice of optimizers, some requiring external packages. To import the optimization packages (and their dependencies) at a global level (recommended), add the following lines in the calling script before importing APOSMM:

```
import libensemble.gen_funcs
libensemble.gen_funcs.rc.aposmm_optimizers = <optimizers>
```

where `optimizers` is a string (or list of strings) from the available options:

`"petsc"`, `"nlopt"`, `"dfols"`, `"scipy"`, `"external"`

Issues with ensemble hanging or failed simulations?

Note that if using **mpi4py** comms, PETSc must be imported at the global level or the ensemble may hang.

Exception: In the case that you are using the MPIExecutor or other MPI inside a user function and you are using Open MPI, then you must:

- Use `local` comms for libEnsemble (not `mpi4py`)
- Must **NOT** include the *rc* line above

This is because PETSc imports MPI, and a global import of PETSc would result in nested MPI (which is not supported by Open MPI). When the above line is not used, an import local to the optimization function will happen.

To see the optimization algorithms supported, see `LocalOptInterfacer`.

See also:

`Persistent APOSMM Tutorial`

Persistent APOSMM

This module contains methods used our implementation of the Asynchronously Parallel Optimization Solver for finding Multiple Minima (APOSMM) method. https://doi.org/10.1007/s12532-017-0131-4

This implementation of APOSMM was developed by Kaushik Kulkarni and Jeffrey Larson in the summer of 2019.

`persistent_aposmm.aposmm`(*H*, *persis_info*, *gen_specs*, *libE_info*)

APOSMM coordinates multiple local optimization runs, starting from points which do not have a better point nearby (within a distance `r_k`). This generation function uses a `local_H` (serving a similar purpose as `H` in libEnsemble) containing the fields:

- `'x' [n floats]`: Parameters being optimized over
- `'x_on_cube' [n floats]`: Parameters scaled to the unit cube
- `'f' [float]`: Objective function being minimized

- 'local_pt' [bool]: True if point from a local optimization run
- 'dist_to_unit_bounds' [float]: Distance to domain boundary
- 'dist_to_better_l' [float]: Dist to closest better local opt point
- 'dist_to_better_s' [float]: Dist to closest better sample point
- 'ind_of_better_l' [int]: Index of point 'dist_to_better_l' away
- 'ind_of_better_s' [int]: Index of point 'dist_to_better_s' away
- 'started_run' [bool]: True if point has started a local opt run
- 'num_active_runs' [int]: Number of active local runs point is in
- 'local_min' [float]: True if point has been ruled a local minima
- 'sim_id' [int]: Row number of entry in history

and optionally

- 'fvec' [m floats]: All objective components (if performing a least-squares calculation)
- 'grad' [n floats]: The gradient (if available) of the objective with respect to x.

Note:

- If any of the above fields are desired after a libEnsemble run, name them in gen_specs['out'].
- If intitializing APOSMM with past function values, make sure to include 'x', 'x_on_cube', 'f', 'local_pt', etc. in gen_specs['in'] (and, of course, include them in the H0 array given to libensemble).

Necessary quantities in gen_specs['user'] are:

- 'lb' [n floats]: Lower bound on search domain
- 'ub' [n floats]: Upper bound on search domain
- 'localopt_method' [str]: Name of an NLopt, PETSc/TAO, or SciPy method (see 'advance_local_run' below for supported methods). When using a SciPy method, must supply 'opt_return_codes', a list of integers that will be used to determine if the x produced by the localopt method should be ruled a local minimum. (For example, SciPy's COBYLA has a 'status' of 1 if at an optimum, but SciPy's Nelder-Mead and BFGS have a 'status' of 0 if at an optimum.)
- 'initial_sample_size' [int]: Number of uniformly sampled points must be returned (non-nan value) before a local opt run is started. Can be zero if no additional sampling is desired, but if zero there must be past sim_f values given to libEnsemble in H0.

Optional gen_specs['user'] entries are:

- 'sample_points' [numpy array]: Points to be sampled (original domain). If more sample points are needed by APOSMM during the course of the optimization, points will be drawn uniformly over the domain
- 'components' [int]: Number of objective components
- 'dist_to_bound_multiple' [float in (0, 1]]: What fraction of the distance to the nearest boundary should the initial step size be in localopt runs
- 'lhs_divisions' [int]: Number of Latin hypercube sampling partitions (0 or 1 results in uniform sampling)
- 'mu' [float]: Distance from the boundary that all localopt starting points must satisfy
- 'nu' [float]: Distance from identified minima that all starting points must satisfy
- 'rk_const' [float]: Multiplier in front of the r_k value

- `'max_active_runs' [int]`: Bound on number of runs APOSMM is advancing
- `'stop_after_k_minima' [int]`: Tell APOSMM to stop after this many local minima have been identified by a local optimization run.
- `'stop_after_k_runs' [int]`: Tell APOSMM to stop after this many runs have ended. (The number of ended runs may be less than the number of minima if, for example, a local optimization run ends due to a evaluation constraint, but not convergence criteria.)

If the rules in `decide_where_to_start_localopt` produces more than `'max_active_runs'` in some iteration, then existing runs are prioritized.

And `gen_specs['user']` must also contain fields for the given localopt_method's convergence tolerances (e.g., gatol/grtol for PETSC/TAO or ftol_rel for NLopt)

See also:

test_persistent_aposmm_scipy for basic APOSMM usage.

See also:

test_persistent_aposmm_with_grad for an example where past function values are given to libEnsemble/APOSMM.

`persistent_aposmm.`**`initialize_APOSMM`**(*H*, *user_specs*, *libE_info*)

Computes common values every time that APOSMM is reinvoked

See also:

start_persistent_local_opt_gens.py

`persistent_aposmm.`**`decide_where_to_start_localopt`**(*H*, *n*, *n_s*, *rk_const*, *ld=0*, *mu=0*, *nu=0*)

APOSMM starts a local optimization runs from a point that:

- is not in an active local optimization run,
- is more than `mu` from the boundary (in the unit-cube domain),
- is more than `nu` from identified minima (in the unit-cube domain),
- does not have a better point within a distance `r_k` of it.

For further details, see the conditions (S1-S5 and L1-L8) in Table 1 of the APOSMM paper This method first identifies sample points satisfying S2-S5, and then identifies all localopt points that satisfy L1-L7. We then start from any sample point also satisfying S1. We do not check condition L8 currently.

We don't consider points in the history that have not returned from computation, or that have a `nan` value. As APOSMM works on the unit cube, note that `mu` and `nu` implicitly depend on the scaling of the original domain: adjusting the initial domain can make a run start (or not start) at a point that didn't (or did) previously.

Parameters

- **`H`** (*`numpy.ndarray`*) – History array storing rows for each point. Numpy structured array.
- **`n`** (*`int`*) – Problem dimension
- **`n_s`** (*`int`*) – Number of sample points in H
- **`r_k_const`** (*`float`*) – Radius for deciding when to start runs
- **`ld`** (*`int`*) – Number of Latin hypercube sampling divisions (0 or 1 means uniform random sampling over the domain)
- **`mu`** (*`float`*) – Nonnegative distance from the boundary that all starting points must satisfy

- **nu** (*float*) – Nonnegative distance from identified minima that all starting points must satisfy

Returns

start_inds – Indices where a local opt run should be started, sorted by increasing function value.

Return type

list

See also:

start_persistent_local_opt_gens.py

persistent_aposmm.**update_history_dist**(*H*, *n*)

Updates distances/indices after new points that have been evaluated.

See also:

start_persistent_local_opt_gens.py

LocalOptInterfacer

This module contains methods for APOSMM to interface with various local optimization routines.

class aposmm_localopt_support.**LocalOptInterfacer**(*user_specs*, *x0*, *f0*, *grad0=None*)

This class defines the APOSMM interface to various local optimization routines.

Currently supported routines are

- NLopt routines ['LN_SBPLX', 'LN_BOBYQA', 'LN_COBYLA', 'LN_NEWUOA', 'LN_NELDERMEAD', 'LD_MMA']
- PETSc/TAO routines ['pounders', 'blmvm', 'nm']
- SciPy routines ['scipy_Nelder-Mead', 'scipy_COBYLA', 'scipy_BFGS']
- DFOLS ['dfols']
- External local optimizer ['external_localopt'] (which use files to pass/receive x/f values)

iterate(*data*)

Returns an instance of either `numpy.ndarray` corresponding to the next iterative guess or `ConvergedMsg` when the solver has completed its run.

Parameters

- **x_on_cube** – A numpy array of the point being evaluated (for a handshake)
- **f** – A numpy array of the function evaluation.
- **grad** – A numpy array of the function's gradient.
- **fvec** – A numpy array of the function's component values.

destroy()

Recursively kill any optimizer processes still running

close()

Join process and close queue

aposmm_localopt_support.**run_local_nlopt**(*user_specs*, *comm_queue*, *x0*, *f0*, *child_can_read*, *parent_can_read*)

Runs an NLopt local optimization run starting at `x0`, governed by the parameters in `user_specs`.

aposmm_localopt_support.**run_local_tao**(*user_specs*, *comm_queue*, *x0*, *f0*, *child_can_read*, *parent_can_read*)

Runs a PETSc/TAO local optimization run starting at `x0`, governed by the parameters in `user_specs`.

aposmm_localopt_support.**run_local_dfols**(*user_specs*, *comm_queue*, *x0*, *f0*, *child_can_read*, *parent_can_read*)

Runs a DFOLS local optimization run starting at `x0`, governed by the parameters in `user_specs`.

aposmm_localopt_support.**run_local_ibcdfo_pounders**(*user_specs*, *comm_queue*, *x0*, *f0*, *child_can_read*, *parent_can_read*)

Runs a IBCDFO local optimization run starting at `x0`, governed by the parameters in `user_specs`.

Although IBCDFO methods can receive previous evaluations, few other methods support that, so APOSMM assumes the first point will be re-evaluated (but not be sent back to the manager).

aposmm_localopt_support.**run_local_scipy_opt**(*user_specs*, *comm_queue*, *x0*, *f0*, *child_can_read*, *parent_can_read*)

Runs a SciPy local optimization run starting at `x0`, governed by the parameters in `user_specs`.

aposmm_localopt_support.**run_external_localopt**(*user_specs*, *comm_queue*, *x0*, *f0*, *child_can_read*, *parent_can_read*)

Runs an external local optimization run starting at `x0`, governed by the parameters in `user_specs`.

uniform_or_localopt

Required: nlopt This module is a persistent generation function that performs a uniform random sample when `libE_info["persistent"]` isn't `True`, or performs a single persistent persistent nlopt local optimization run.

uniform_or_localopt.**uniform_or_localopt**(*H*, *persis_info*, *gen_specs*, *libE_info*)

This generation function returns `gen_specs["user"]["gen_batch_size"]` uniformly sampled points when called in nonpersistent mode (i.e., when `libE_info["persistent"]` isn't `True`). Otherwise, the generation function starts a persistent nlopt local optimization run.

See also:

test_uniform_sampling_then_persistent_localopt_runs.py # noqa

persistent_tasmanian

Required: Tasmanian, pypackaging, scikit-build

Note that Tasmanian can be pip installed, but currently must use either *venv* or *–user* install.

`E.g: pip install scikit-build packaging Tasmanian --user` A persistent generator using the uncertainty quantification capabilities in Tasmanian.

persistent_tasmanian.**lex_le**(*x*, *y*, *tol=1e-12*)

Returns True if x <= y lexicographically up to some tolerance.

persistent_tasmanian.**get_2D_insert_indices**(*x*, *y*, *x_ord=array([], dtype=int64)*, *y_ord=array([], dtype=int64)*, *tol=1e-12*)

Finds the row indices in a 2D numpy array *x* for which the sorted values of *y* can be inserted into. If *x_ord* (resp. *y_ord*) is empty, then *x* (resp. *y*) must be lexicographically sorted. Otherwise, *x[x_ord]* (resp. *y[y_ord]*) must be lexicographically sorted. Complexity is O(x.shape[0] + y.shape[0]).

persistent_tasmanian.**get_2D_duplicate_indices**(*x*, *y*, *x_ord=array([], dtype=int64)*, *y_ord=array([], dtype=int64)*, *tol=1e-12*)

Finds the row indices of a 2D numpy array *x* which overlap with *y*. If *x_ord* (resp. *y_ord*) is empty, then *x* (resp. *y*) must be lexicographically sorted. Otherwise, *x[x_ord]* (resp. *y[y_ord]*) must be lexicographically sorted.Complexity is O(x.shape[0] + y.shape[0]).

persistent_tasmanian.**get_state**(*queued_pts*, *queued_ids*, *id_offset*, *new_points=array([], dtype=float64)*, *completed_points=array([], dtype=float64)*, *tol=1e-12*)

Creates the data to be sent and updates the state arrays and scalars if new information (new_points or compeleted_points) arrives. Ensures that the output state arrays remain sorted if the input state arrays are already sorted.

persistent_tasmanian.**get_H0**(*gen_specs*, *refined_pts*, *refined_ord*, *queued_pts*, *queued_ids*, *tol=1e-12*)

For runs following the first one, get the history array H0 based on the ordering in *refined_pts*

persistent_tasmanian.**sparse_grid_batched**(*H*, *persis_info*, *gen_specs*, *libE_info*)

Implements batched construction for a Tasmanian sparse grid, using the loop described in Tasmanian Example 09: sparse grid example

persistent_tasmanian.**sparse_grid_async**(*H*, *persis_info*, *gen_specs*, *libE_info*)

Implements asynchronous construction for a Tasmanian sparse grid, using the logic in the dynamic Tasmanian model construction function: sparse grid dynamic example

persistent_tasmanian.**get_sparse_grid_specs**(*user_specs*, *sim_f*, *num_dims*, *num_outputs=1*, *mode='batched'*)

Helper function that generates the simulator, generator, and allocator specs as well as the persis_info dictionary to ensure that they are compatible with the custom generators in this script. The outputs should be used in the main libE() call.

INPUTS:

user_specs (dict)
[a dictionary of user specs that is needed in the generator specs;] expects the key "tasmanian_init" whose value is a 0-argument lambda that initializes an appropriate Tasmanian sparse grid object.

sim_f (func)
[a lambda function that takes in generator outputs (simulator inputs)] and returns simulator outputs.

num_dims (int) : number of model inputs.

num_outputs (int) : number of model outputs.

mode (string) : can either be "batched" or "async".

OUTPUTS:

sim_specs (dict) : a dictionary of simulation specs and also one of the inputs of libE().

gen_specs (dict) : a dictionary of generator specs and also one of the inputs of libE().

alloc_specs (dict) : a dictionary of allocation specs and also one of the inputs of libE().

persis_info (dict)
[a dictionary containing common information that is passed to all] workers and also one of the inputs of libE().

persistent_tasmanian.py

```
"""
A persistent generator using the uncertainty quantification capabilities in
`Tasmanian <https://tasmanian.ornl.gov/>`_.
"""

import numpy as np

from libensemble.alloc_funcs.start_only_persistent import only_persistent_gens as allocf
from libensemble.message_numbers import EVAL_GEN_TAG, FINISHED_PERSISTENT_GEN_TAG,␣
↪PERSIS_STOP, STOP_TAG
from libensemble.tools import parse_args
from libensemble.tools.persistent_support import PersistentSupport


def lex_le(x, y, tol=1e-12):
    """
    Returns True if x <= y lexicographically up to some tolerance.
    """
    cmp = np.fabs(x - y) > tol
    ind = np.argmax(cmp)
    if not cmp[ind]:
        return True
    return x[ind] <= y[ind]


def get_2D_insert_indices(x, y, x_ord=np.empty(0, dtype="int"), y_ord=np.empty(0, dtype=
↪"int"), tol=1e-12):
    """
    Finds the row indices in a 2D numpy array `x` for which the sorted values of `y` can␣
↪be inserted
    into. If `x_ord` (resp. `y_ord`) is empty, then `x` (resp. `y`) must be␣
↪lexicographically
    sorted. Otherwise, `x[x_ord]` (resp. `y[y_ord]`) must be lexicographically sorted.␣
↪Complexity is
    O(x.shape[0] + y.shape[0]).
    """
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    if x.size == 0:
        return np.zeros(y.shape[0], dtype="int")
    else:
        if x_ord.size == 0:
            x_ord = np.arange(x.shape[0], dtype="int")
        if y_ord.size == 0:
            y_ord = np.arange(y.shape[0], dtype="int")
        x_ptr = 0
        y_ptr = 0
        out_ord = np.empty(0, dtype="int")
        while y_ptr < y.shape[0]:
            # The case where y[k] <= max of x[k:end, :]
            xk = x[x_ord[x_ptr], :]
```

(continues on next page)

(continued from previous page)

```
            yk = y[y_ord[y_ptr], :]
            if lex_le(yk, xk, tol=tol):
                out_ord = np.append(out_ord, x_ord[x_ptr])
                y_ptr += 1
            else:
                x_ptr += 1
                # The edge case where y[k] is the largest of all elements of x.
                if x_ptr >= x_ord.shape[0]:
                    for i in range(y_ptr, y_ord.shape[0], 1):
                        out_ord = np.append(out_ord, x_ord.shape[0])
                        y_ptr += 1
                    break
        return out_ord


def get_2D_duplicate_indices(x, y, x_ord=np.empty(0, dtype="int"), y_ord=np.empty(0,␣
↪dtype="int"), tol=1e-12):
    """
    Finds the row indices of a 2D numpy array `x` which overlap with `y`. If `x_ord` (resp.
↪ `y_ord`)
    is empty, then `x` (resp. `y`) must be lexicographically sorted. Otherwise, `x[x_ord]`␣
↪(resp.
    `y[y_ord]`) must be lexicographically sorted.Complexity is O(x.shape[0] + y.shape[0]).
    """
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    if x.size == 0:
        return np.empty(0, dtype="int")
    else:
        if x_ord.size == 0:
            x_ord = np.arange(x.shape[0], dtype="int")
        if y_ord.size == 0:
            y_ord = np.arange(y.shape[0], dtype="int")
        x_ptr = 0
        y_ptr = 0
        out_ord = np.empty(0, dtype="int")
        while y_ptr < y.shape[0] and x_ptr < x.shape[0]:
            # The case where y[k] <= max of x[k:end, :]
            xk = x[x_ord[x_ptr], :]
            yk = y[y_ord[y_ptr], :]
            if all(np.fabs(yk - xk) <= tol):
                out_ord = np.append(out_ord, x_ord[x_ptr])
                x_ptr += 1
            elif lex_le(xk, yk, tol=tol):
                x_ptr += 1
            else:
                y_ptr += 1
        return out_ord


def get_state(queued_pts, queued_ids, id_offset, new_points=np.array([]), completed_
↪points=np.array([]), tol=1e-12):
```

(continues on next page)

(continued from previous page)

```
    """
    Creates the data to be sent and updates the state arrays and scalars if new information
    (new_points or compeleted_points) arrives. Ensures that the output state arrays remain sorted if
    the input state arrays are already sorted.
    """
    if new_points.size > 0:
        new_points_ord = np.lexsort(np.rot90(new_points))
        new_points_ids = id_offset + np.arange(new_points.shape[0])
        id_offset += new_points.shape[0]
        insert_idx = get_2D_insert_indices(queued_pts, new_points, y_ord=new_points_ord, tol=tol)
        queued_pts = np.insert(queued_pts, insert_idx, new_points[new_points_ord], axis=0)
        queued_ids = np.insert(queued_ids, insert_idx, new_points_ids[new_points_ord], axis=0)

    if completed_points.size > 0:
        completed_ord = np.lexsort(np.rot90(completed_points))
        delete_ind = get_2D_duplicate_indices(queued_pts, completed_points, y_ord=completed_ord, tol=tol)
        queued_pts = np.delete(queued_pts, delete_ind, axis=0)
        queued_ids = np.delete(queued_ids, delete_ind, axis=0)

    return queued_pts, queued_ids, id_offset


def get_H0(gen_specs, refined_pts, refined_ord, queued_pts, queued_ids, tol=1e-12):
    """
    For runs following the first one, get the history array H0 based on the ordering in `refined_pts`
    """

    def approx_eq(x, y):
        return np.argmax(np.fabs(x - y)) <= tol

    num_ids = queued_ids.shape[0]
    H0 = np.zeros(num_ids, dtype=gen_specs["out"])
    refined_priority = np.flip(np.arange(refined_pts.shape[0], dtype="int"))
    rptr = 0
    for qptr in range(num_ids):
        while not approx_eq(refined_pts[refined_ord[rptr]], queued_pts[qptr]):
            rptr += 1
        assert rptr <= refined_pts.shape[0]
        H0["x"][qptr] = queued_pts[qptr]
        H0["sim_id"][qptr] = queued_ids[qptr]
        H0["priority"][qptr] = refined_priority[refined_ord[rptr]]
    return H0


# =======================
```

(continues on next page)

(continued from previous page)

```
# Main generator functions
# ========================


def sparse_grid_batched(H, persis_info, gen_specs, libE_info):
    """
    Implements batched construction for a Tasmanian sparse grid,
    using the loop described in Tasmanian Example 09:
    `sparse grid example <https://github.com/ORNL/TASMANIAN/blob/master/InterfacePython/
↪example_sparse_grids_09.py>`_

    """
    U = gen_specs["user"]
    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)
    grid = U["tasmanian_init"]()  # initialize the grid
    allowed_refinements = [
        "setAnisotropicRefinement",
        "getAnisotropicRefinement",
        "setSurplusRefinement",
        "getSurplusRefinement",
        "none",
    ]
    assert (
        "refinement" in U and U["refinement"] in allowed_refinements
    ), f"Must provide a gen_specs['user']['refinement'] in: {allowed_refinements}"

    while grid.getNumNeeded() > 0:
        aPoints = grid.getNeededPoints()

        H0 = np.zeros(len(aPoints), dtype=gen_specs["out"])
        H0["x"] = aPoints

        # Receive values from manager
        tag, Work, calc_in = ps.send_recv(H0)
        if tag in [STOP_TAG, PERSIS_STOP]:
            break
        aModelValues = calc_in["f"]

        # Update surrogate on grid
        t = aModelValues.reshape((aModelValues.shape[0], grid.getNumOutputs()))
        t = t.flatten()
        t = np.atleast_2d(t).T
        grid.loadNeededPoints(t)

        if "tasmanian_checkpoint_file" in U:
            grid.write(U["tasmanian_checkpoint_file"])

        # set refinement, using user["refinement"] to pick the refinement strategy
        if U["refinement"] in ["setAnisotropicRefinement", "getAnisotropicRefinement"]:
            assert "sType" in U
            assert "iMinGrowth" in U
            assert "iOutput" in U
```

(continues on next page)

(continued from previous page)

```
            grid.setAnisotropicRefinement(U["sType"], U["iMinGrowth"], U["iOutput"])
        elif U["refinement"] in ["setSurplusRefinement", "getSurplusRefinement"]:
            assert "fTolerance" in U
            assert "iOutput" in U
            assert "sCriteria" in U
            grid.setSurplusRefinement(U["fTolerance"], U["iOutput"], U["sCriteria"])

    return H0, persis_info, FINISHED_PERSISTENT_GEN_TAG


def sparse_grid_async(H, persis_info, gen_specs, libE_info):
    """
    Implements asynchronous construction for a Tasmanian sparse grid,
    using the logic in the dynamic Tasmanian model construction function:
    `sparse grid dynamic example <https://github.com/ORNL/TASMANIAN/blob/master/Addons/tsgConstructSurrogate.hpp>`_

    """
    U = gen_specs["user"]
    ps = PersistentSupport(libE_info, EVAL_GEN_TAG)
    grid = U["tasmanian_init"]()  # initialize the grid
    allowed_refinements = ["getCandidateConstructionPoints", "getCandidateConstructionPointsSurplus"]
    assert (
        "refinement" in U and U["refinement"] in allowed_refinements
    ), f"Must provide a gen_specs['user']['refinement'] in: {allowed_refinements}"
    tol = U["_match_tolerance"] if "_match_tolerance" in U else 1.0e-12

    # Choose the refinement function based on U["refinement"].
    if U["refinement"] == "getCandidateConstructionPoints":
        assert "sType" in U
        assert "liAnisotropicWeightsOrOutput" in U
    if U["refinement"] == "getCandidateConstructionPointsSurplus":
        assert "fTolerance" in U
        assert "sRefinementType" in U

    def get_refined_points(g, U):
        if U["refinement"] == "getCandidateConstructionPoints":
            return g.getCandidateConstructionPoints(U["sType"], U["liAnisotropicWeightsOrOutput"])
        else:
            assert U["refinement"] == "getCandidateConstructionPointsSurplus"
            return g.getCandidateConstructionPointsSurplus(U["fTolerance"], U["sRefinementType"])
        # else:
        #     raise ValueError("Unknown refinement string")

    # Asynchronous helper and state variables.
    num_dims = grid.getNumDimensions()
    num_completed = 0
    offset = 0
    queued_pts = np.empty((0, num_dims), dtype="float")
```

(continues on next page)

(continued from previous page)

```
    queued_ids = np.empty(0, dtype="int")

    # First run.
    grid.beginConstruction()
    init_pts = get_refined_points(grid, U)
    queued_pts, queued_ids, offset = get_state(queued_pts, queued_ids, offset, new_
↪points=init_pts, tol=tol)
    H0 = np.zeros(init_pts.shape[0], dtype=gen_specs["out"])
    H0["x"] = init_pts
    H0["sim_id"] = np.arange(init_pts.shape[0], dtype="int")
    H0["priority"] = np.flip(H0["sim_id"])
    tag, Work, calc_in = ps.send_recv(H0)

    # Subsequent runs.
    while tag not in [STOP_TAG, PERSIS_STOP]:
        # Parse the points returned by the allocator.
        num_completed += calc_in["x"].shape[0]
        queued_pts, queued_ids, offset = get_state(
            queued_pts, queued_ids, offset, completed_points=calc_in["x"], tol=tol
        )

        # Compute the next batch of points (if they exist).
        new_pts = np.empty((0, num_dims), dtype="float")
        refined_pts = np.empty((0, num_dims), dtype="float")
        refined_ord = np.empty(0, dtype="int")
        if grid.getNumLoaded() < 1000 or num_completed > 0.2 * grid.getNumLoaded():
            # A copy is needed because the data in the calc_in arrays are not contiguous.
            grid.loadConstructedPoint(np.copy(calc_in["x"]), np.copy(calc_in["f"]))
            if "tasmanian_checkpoint_file" in U:
                grid.write(U["tasmanian_checkpoint_file"])
            refined_pts = get_refined_points(grid, U)
            # If the refined points are empty, then there is a stopping condition␣
↪internal to the
            # Tasmanian sparse grid that is being triggered by the loaded points.
            if refined_pts.size == 0:
                break
            refined_ord = np.lexsort(np.rot90(refined_pts))
            delete_ind = get_2D_duplicate_indices(refined_pts, queued_pts, x_ord=refined_
↪ord, tol=tol)
            new_pts = np.delete(refined_pts, delete_ind, axis=0)

        if new_pts.shape[0] > 0:
            # Update the state variables with the refined points and update the queue in␣
↪the allocator.
            num_completed = 0
            queued_pts, queued_ids, offset = get_state(queued_pts, queued_ids, offset,␣
↪new_points=new_pts, tol=tol)
            H0 = get_H0(gen_specs, refined_pts, refined_ord, queued_pts, queued_ids,␣
↪tol=tol)
            tag, Work, calc_in = ps.send_recv(H0)
        else:
            tag, Work, calc_in = ps.recv()
```

(continues on next page)

(continued from previous page)

```
    return [], persis_info, FINISHED_PERSISTENT_GEN_TAG


def get_sparse_grid_specs(user_specs, sim_f, num_dims, num_outputs=1, mode="batched"):
    """
    Helper function that generates the simulator, generator, and allocator specs as well as the
    persis_info dictionary to ensure that they are compatible with the custom generators in this
    script. The outputs should be used in the main libE() call.

    INPUTS:
        user_specs  (dict)   : a dictionary of user specs that is needed in the generator specs;
                               expects the key "tasmanian_init" whose value is a 0-argument lambda
                               that initializes an appropriate Tasmanian sparse grid object.

        sim_f       (func)   : a lambda function that takes in generator outputs (simulator inputs)
                               and returns simulator outputs.

        num_dims    (int)    : number of model inputs.

        num_outputs (int)    : number of model outputs.

        mode        (string) : can either be "batched" or "async".

    OUTPUTS:
        sim_specs   (dict) : a dictionary of simulation specs and also one of the inputs of libE().

        gen_specs   (dict) : a dictionary of generator specs and also one of the inputs of libE().

        alloc_specs (dict) : a dictionary of allocation specs and also one of the inputs of libE().

        persis_info (dict) : a dictionary containing common information that is passed to all
                             workers and also one of the inputs of libE().

    """

    assert "tasmanian_init" in user_specs
    assert mode in ["batched", "async"]

    sim_specs = {
        "sim_f": sim_f,
        "in": ["x"],
```

(continues on next page)

(continued from previous page)

```
    }
    gen_out = [
        ("x", float, (num_dims,)),
        ("sim_id", int),
        ("priority", int),
    ]
    gen_specs = {
        "persis_in": [t[0] for t in gen_out] + ["f"],
        "out": gen_out,
        "user": user_specs,
    }
    alloc_specs = {
        "alloc_f": allocf,
        "user": {},
    }

    if mode == "batched":
        gen_specs["gen_f"] = sparse_grid_batched
        sim_specs["out"] = [("f", float, (num_outputs,))]
    if mode == "async":
        gen_specs["gen_f"] = sparse_grid_async
        sim_specs["out"] = [("x", float, (num_dims,)), ("f", float, (num_outputs,))]
        alloc_specs["user"]["active_recv_gen"] = True
        alloc_specs["user"]["async_return"] = True

    nworkers, _, _, _ = parse_args()
    persis_info = {}
    for i in range(nworkers + 1):
        persis_info[i] = {"worker_num": i}

    return sim_specs, gen_specs, alloc_specs, persis_info
```

persistent_fd_param_finder

persistent_fd_param_finder.**fd_param_finder**(*H*, *persis_info*, *gen_specs*, *libE_info*)

This generation function loops through a set of suitable finite difference parameters for a mapping F from R^n to R^m.

See also:

test_persistent_fd_param_finder.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/regression_tests/test_persistent_fd_param_finder.py>`_ # noqa

persistent_surmise

Required: Surmise

Note that currently the github fork https://github.com/mosesyhc/surmise should be used:

```
pip install --upgrade git+https://github.com/bandframework/surmise.git@develop
```

The `Borehole Calibration tutorial` uses this generator as an example of the capability to cancel pending simulations. This module contains a simple calibration example using the Surmise package.

`persistent_surmise_calib.`**`surmise_calib`**(*H*, *persis_info*, *gen_specs*, *libE_info*)

Generator to select and obviate parameters for calibration.

6.5.2 Simulation Functions

Below are example simulation functions available in libEnsemble. Most of these demonstrate an inexpensive algorithm and do not launch tasks (user applications). To see an example of a simulation function launching tasks, see the `Electrostatic Forces tutorial`.

Important: See the API for simulation functions `here`.

---

six_hump_camel

This module contains various versions that evaluate the six-hump camel function.

Six-hump camel function is documented here:
https://www.sfu.ca/~ssurjano/camel6.html

`six_hump_camel.`**`six_hump_camel`**(*H*, *persis_info*, *sim_specs*, *libE_info*)

Evaluates the six hump camel function for a collection of points given in `H["x"]`. Additionally evaluates the gradient if `"grad"` is a field in `sim_specs["out"]` and pauses for `sim_specs["user"]["pause_time"]` if defined.

See also:

test_uniform_sampling.py # noqa

`six_hump_camel.`**`six_hump_camel_simple`**(*x*, *_*, *sim_specs*)

Evaluates the six hump camel function for a single point `x`.

See also:

test_fast_alloc.py # noqa

`six_hump_camel.`**`persistent_six_hump_camel`**(*H*, *persis_info*, *sim_specs*, *libE_info*)

Similar to `six_hump_camel`, but runs in persistent mode.

six_hump_camel.py

```
"""
This module contains various versions that evaluate the six-hump camel function.

Six-hump camel function is documented here:
  https://www.sfu.ca/~ssurjano/camel6.html

"""
__all__ = [
    "six_hump_camel",
    "six_hump_camel_simple",
    "persistent_six_hump_camel",
]

import sys
import time

import numpy as np

from libensemble.message_numbers import EVAL_SIM_TAG, FINISHED_PERSISTENT_SIM_TAG,␣
↪PERSIS_STOP, STOP_TAG
from libensemble.tools.persistent_support import PersistentSupport


def six_hump_camel(H, persis_info, sim_specs, libE_info):
    """
    Evaluates the six hump camel function for a collection of points given in ``H["x"]``.
    Additionally evaluates the gradient if ``"grad"`` is a field in
    ``sim_specs["out"]`` and pauses for ``sim_specs["user"]["pause_time"]]`` if
    defined.

    .. seealso::
        `test_uniform_sampling.py  <https://github.com/Libensemble/libensemble/blob/
↪develop/libensemble/tests/functionality_tests/test_uniform_sampling.py>`_ # noqa
    """

    batch = len(H["x"])
    H_o = np.zeros(batch, dtype=sim_specs["out"])

    for i, x in enumerate(H["x"]):
        H_o["f"][i] = six_hump_camel_func(x)

        if "grad" in H_o.dtype.names:
            H_o["grad"][i] = six_hump_camel_grad(x)

        if "user" in sim_specs and "pause_time" in sim_specs["user"]:
            time.sleep(sim_specs["user"]["pause_time"])

    return H_o, persis_info


def six_hump_camel_simple(x, _, sim_specs):
```

(continues on next page)

(continued from previous page)

```
    """
    Evaluates the six hump camel function for a single point ``x``.

    .. seealso::
        `test_fast_alloc.py <https://github.com/Libensemble/libensemble/blob/develop/
↪libensemble/tests/functionality_tests/test_fast_alloc.py>`_ # noqa
    """

    H_o = np.zeros(1, dtype=sim_specs["out"])

    H_o["f"] = six_hump_camel_func(x[0][0][:2])  # Ignore more than 2 entries of x

    if sim_specs["user"].get("pause_time"):
        time.sleep(sim_specs["user"]["pause_time"])

    if sim_specs["user"].get("rand"):
        H_o["f"] += np.random.normal(0, 1)

    return H_o


def persistent_six_hump_camel(H, persis_info, sim_specs, libE_info):
    """
    Similar to ``six_hump_camel``, but runs in persistent mode.
    """

    ps = PersistentSupport(libE_info, EVAL_SIM_TAG)

    # Either start with a work item to process - or just start and wait for data
    if H.size > 0:
        tag = None
        Work = None
        calc_in = H
    else:
        tag, Work, calc_in = ps.recv()

    while tag not in [STOP_TAG, PERSIS_STOP]:
        # calc_in: This should either be a function (unpack_work ?) or included/unpacked␣
↪in ps.recv/ps.send_recv.
        if Work is not None:
            persis_info = Work.get("persis_info", persis_info)
            libE_info = Work.get("libE_info", libE_info)

        # Call standard six_hump_camel sim
        H_o, persis_info = six_hump_camel(calc_in, persis_info, sim_specs, libE_info)

        tag, Work, calc_in = ps.send_recv(H_o)

    final_return = None

    # Overwrite final point - for testing only
    if sim_specs["user"].get("replace_final_fields", 0):
```

(continues on next page)

(continued from previous page)

```
        calc_in = np.ones(1, dtype=[("x", float, (2,))])
        H_o, persis_info = six_hump_camel(calc_in, persis_info, sim_specs, libE_info)
        final_return = H_o

    return final_return, persis_info, FINISHED_PERSISTENT_SIM_TAG


def six_hump_camel_func(x):
    """
    Definition of the six-hump camel
    """
    x1 = x[0]
    x2 = x[1]
    term1 = (4 - 2.1 * x1**2 + (x1**4) / 3) * x1**2
    term2 = x1 * x2
    term3 = (-4 + 4 * x2**2) * x2**2

    return term1 + term2 + term3


def six_hump_camel_grad(x):
    """
    Definition of the six-hump camel gradient
    """

    x1 = x[0]
    x2 = x[1]
    grad = np.zeros(2)

    grad[0] = 2.0 * (x1**5 - 4.2 * x1**3 + 4.0 * x1 + 0.5 * x2)
    grad[1] = x1 + 16 * x2**3 - 8 * x2

    return grad


if __name__ == "__main__":
    x = (float(sys.argv[1]), float(sys.argv[2]))
    result = six_hump_camel_func(x)
    print(result)
```

chwirut

chwirut1.**chwirut_eval**(*H*, *_*, *sim_specs*)

Evaluates the chwirut objective function at a given set of points in `H["x"]`. If `"obj_component"` is a field in `sim_specs["out"]`, only that component of the objective will be evaluated. Otherwise, all 214 components are evaluated and returned in the `"fvec"` field.

See also:

test_persistent_aposmm_pounders.py # noqa for an example where the entire fvec is computed each call.

See also:

test_uniform_sampling_one_residual_at_a_time.py # noqa for an example where one component of fvec is computed per call

noisy_vector_mapping

This module contains a test noisy function

noisy_vector_mapping.**func_wrapper**(*H*, *persis_info*, *sim_specs*, *libE_info*)

Wraps an objective function

See also:

test_persistent_fd_param_finder.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/regression_tests/test_persistent_fd_param_finder.py>`_ # noqa

noisy_vector_mapping.**noisy_function**(*x*)

noisy_vector_mapping.py

```
"""
This module contains a test noisy function
"""

import numpy as np
from numpy import cos, sin
from numpy.linalg import norm


def func_wrapper(H, persis_info, sim_specs, libE_info):
    """
    Wraps an objective function

    .. seealso::
        `test_persistent_fd_param_finder.py` <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/regression_tests/test_persistent_fd_param_finder.py>`_ # noqa
    """

    batch = len(H["x"])
    H0 = np.zeros(batch, dtype=sim_specs["out"])

    for i, x in enumerate(H["x"]):
        H0["f_val"][i] = noisy_function(x)[H["f_ind"][i]]

    return H0, persis_info


def noisy_function(x):
    """ """
    x1 = x[0]
    x2 = x[1]
    term1 = (4 - 2.1 * x1**2 + (x1**4) / 3) * x1**2
```

(continues on next page)

(continued from previous page)

```
    term2 = x1 * x2
    term3 = (-4 + 4 * x2**2) * x2**2

    phi1 = 0.9 * sin(100 * norm(x, 1)) * cos(100 * norm(x, np.inf)) + 0.1 * cos(norm(x,␣
↪2))
    phi1 = phi1 * (4 * phi1**2 - 3)

    phi2 = 0.8 * sin(100 * norm(x, 1)) * cos(100 * norm(x, np.inf)) + 0.2 * cos(norm(x,␣
↪2))
    phi2 = phi2 * (4 * phi2**2 - 3)

    phi3 = 0.7 * sin(100 * norm(x, 1)) * cos(100 * norm(x, np.inf)) + 0.3 * cos(norm(x,␣
↪2))
    phi3 = phi3 * (4 * phi3**2 - 3)

    F = np.zeros(3)
    F[0] = (1 + 1e-1 * phi1) * term1
    F[1] = (1 + 1e-2 * phi2) * term2
    F[2] = (1 + 1e-3 * phi3) * term3

    return F
```

periodic_func

This module contains a periodic test function

periodic_func.**func_wrapper**(*H*, *persis_info*, *sim_specs*, *libE_info*)

Wraps an objective function

periodic_func.**periodic_func**(*x*)

This function is periodic

borehole

borehole.**borehole**(*H*, *persis_info*, *sim_specs*, _)

Wraps the borehole function

borehole.**borehole_func**(*x*)

This evaluates the Borehole function for n-by-8 input matrix x, and returns the flow rate through the Borehole. (Harper and Gupta, 1983) input:

Parameters

x (*numpy.typing.NDArray*) –

```
x[:,0]: Tu, transmissivity of upper aquifer (m^2/year)
x[:,1]: Tl, transmissivity of lower aquifer (m^2/year)
x[:,2]: Hu, potentiometric head of upper aquifer (m)
x[:,3]: Hl, potentiometric head of lower aquifer (m)
x[:,4]: r, radius of influence (m)
x[:,5]: rw, radius of borehole (m)
x[:,6]: Kw, hydraulic conductivity of borehole (m/year)
x[:,7]: L, length of borehole (m)
```

Returns
f – vector of dimension (n, 1): flow rate through the Borehole (m^3/year)

Return type
numpy.ndarray

borehole.**gen_borehole_input**(*n*)

Generates and returns n inputs for the Borehole function, according to distributions outlined in Harper and Gupta (1983).

input:
n: number of input to generate

output:
matrix of (n, 8), input to borehole_func(x) function

executor_hworld

executor_hworld.**executor_hworld**(*H*, *_*, *sim_specs*, *info*)

Tests launching and polling task and exiting on task finish

executor_hworld.py

```
import numpy as np

from libensemble.message_numbers import (
    MAN_SIGNAL_FINISH,
    TASK_FAILED,
    UNSET_TAG,
    WORKER_DONE,
    WORKER_KILL_ON_ERR,
    WORKER_KILL_ON_TIMEOUT,
)

__all__ = ["executor_hworld"]

# Alt send values through X
sim_ended_count = 0


def custom_polling_loop(exctr, task, timeout_sec=5.0, delay=0.3):
    import time

    calc_status = UNSET_TAG  # Sim func determines status of libensemble calc - returned␣
↪to worker

    while task.runtime < timeout_sec:
        time.sleep(delay)

        if exctr.manager_kill_received():
            exctr.kill(task)
            calc_status = MAN_SIGNAL_FINISH  # Worker will pick this up and close down
            print(f"Task {task.id} killed by manager on worker {exctr.workerID}")
```

(continues on next page)

(continued from previous page)

```
            break

        task.poll()
        if task.finished:
            break
        elif task.state == "RUNNING":
            print(f"Task {task.id} still running on worker {exctr.workerID} ....")

        if task.stdout_exists():
            if "Error" in task.read_stdout():
                print(
                    "Found (deliberate) Error in output file - cancelling " f"task {task.id} on worker {exctr.workerID}"
                )
                exctr.kill(task)
                calc_status = WORKER_KILL_ON_ERR
                break

    # After exiting loop
    if task.finished:
        print(f"Task {task.id} done on worker {exctr.workerID}")
        # Fill in calc_status if not already
        if calc_status == UNSET_TAG:
            if task.state == "FINISHED":  # Means finished successfully
                calc_status = WORKER_DONE
            elif task.state == "FAILED":
                calc_status = TASK_FAILED

    else:
        # assert task.state == 'RUNNING', "task.state expected to be RUNNING. Returned: " + str(task.state)
        print(f"Task {task.id} timed out - killing on worker {exctr.workerID}")
        exctr.kill(task)
        if task.finished:
            print(f"Task {task.id} done on worker {exctr.workerID}")
        calc_status = WORKER_KILL_ON_TIMEOUT

    return task, calc_status


def executor_hworld(H, _, sim_specs, info):
    """Tests launching and polling task and exiting on task finish"""
    exctr = info["executor"]
    cores = sim_specs["user"]["cores"]
    ELAPSED_TIMEOUT = "elapsed_timeout" in sim_specs["user"]

    wait = False
    args_for_sim = "sleep 1"
    calc_status = UNSET_TAG

    batch = len(H["x"])
    H_o = np.zeros(batch, dtype=sim_specs["out"])
```

(continues on next page)

(continued from previous page)

```
    if "six_hump_camel" not in exctr.default_app("sim").full_path:
        global sim_ended_count
        sim_ended_count += 1
        print("sim_ended_count", sim_ended_count, flush=True)

        if ELAPSED_TIMEOUT:
            args_for_sim = "sleep 60"  # Manager kill - if signal received else completes
            timeout = 65.0

        else:
            timeout = 6.0
            launch_shc = False

            if sim_ended_count == 1:
                args_for_sim = "sleep 1"  # Should finish
            elif sim_ended_count == 2:
                args_for_sim = "sleep 1 Error"  # Worker kill on error
            elif sim_ended_count == 3:
                wait = True
                args_for_sim = "sleep 1"  # Should finish
                launch_shc = True
            elif sim_ended_count == 4:
                args_for_sim = "sleep 8"  # Worker kill on timeout
                timeout = 1.0
            elif sim_ended_count == 5:
                args_for_sim = "sleep 2 Fail"  # Manager kill - if signal received else␣
↪completes

        task = exctr.submit(calc_type="sim", num_procs=cores, app_args=args_for_sim,␣
↪hyperthreads=True)

        if wait:
            task.wait()
            if not task.finished:
                calc_status = UNSET_TAG
            if task.state == "FINISHED":
                calc_status = WORKER_DONE
            elif task.state == "FAILED":
                calc_status = TASK_FAILED

        else:
            if sim_ended_count >= 2:
                calc_status = exctr.polling_loop(task, timeout=timeout, delay=0.3, poll_
↪manager=True)
                if sim_ended_count == 2 and task.stdout_exists() and "Error" in task.
↪read_stdout():
                    calc_status = WORKER_KILL_ON_ERR
            else:
                task, calc_status = custom_polling_loop(exctr, task, timeout)

    else:
```

(continues on next page)

(continued from previous page)

```
        launch_shc = True
        calc_status = UNSET_TAG

        # Comparing six_hump_camel output, directly called vs. submitted as app
        for i, x in enumerate(H["x"]):
            H_o["f"][i] = six_hump_camel_func(x)
            if launch_shc:
                # Test launching a named app.
                app_args = " ".join(str(val) for val in list(x[:]))
                task = exctr.submit(app_name="six_hump_camel", num_procs=1, app_args=app_
↪args)
                task.wait()
                output = np.float64(task.read_stdout())
                assert np.isclose(H_o["f"][i], output)
                calc_status = WORKER_DONE

    # This is just for testing at calling script level - status of each task
    H_o["cstat"] = calc_status

    return H_o, calc_status


def six_hump_camel_func(x):
    """
    Definition of the six-hump camel
    """
    x1 = x[0]
    x2 = x[1]
    term1 = (4 - 2.1 * x1**2 + (x1**4) / 3) * x1**2
    term2 = x1 * x2
    term3 = (-4 + 4 * x2**2) * x2**2

    return term1 + term2 + term3
```

6.5.3 Allocation Functions

Below are example allocation functions available in libEnsemble.

Important: See the API for allocation functions here.

Note: The default allocation function is `give_sim_work_first`.

give_sim_work_first

give_sim_work_first.**give_sim_work_first**(*W*, *H*, *sim_specs*, *gen_specs*, *alloc_specs*, *persis_info*, *libE_info*)

Decide what should be given to workers. This allocation function gives any available simulation work first, and only when all simulations are completed or running does it start (at most `alloc_specs["user"]["num_active_gens"]`) generator instances.

Allows for a `alloc_specs["user"]["batch_mode"]` where no generation work is given out unless all entries in `H` are returned.

Can give points in highest priority, if `"priority"` is a field in `H`. If alloc_specs[“user”][“give_all_with_same_priority”] is set to True, then all points with the same priority value are given as a batch to the sim.

Workers performing sims will be assigned resources given in H[“resource_sets”] this field exists, else defaulting to one. Workers performing gens are assigned resource_sets given by persis_info[“gen_resources”] or zero.

This is the default allocation function if one is not defined.

tags: alloc, default, batch, priority

See also:

test_uniform_sampling.py # noqa

Parameters

- **W** (*ndarray[Any, dtype[_ScalarType_co]]*) –
- **H** (*ndarray[Any, dtype[_ScalarType_co]]*) –
- **sim_specs** (*dict*) –
- **gen_specs** (*dict*) –
- **alloc_specs** (*dict*) –
- **persis_info** (*dict*) –
- **libE_info** (*dict*) –

Return type
Tuple[dict]

give_sim_work_first.py

```
import time
from typing import Tuple

import numpy as np
import numpy.typing as npt

from libensemble.tools.alloc_support import AllocSupport, InsufficientFreeResources


def give_sim_work_first(
    W: npt.NDArray,
    H: npt.NDArray,
```

(continues on next page)

(continued from previous page)

```
    sim_specs: dict,
    gen_specs: dict,
    alloc_specs: dict,
    persis_info: dict,
    libE_info: dict,
) -> Tuple[dict]:
    """
    Decide what should be given to workers. This allocation function gives any
    available simulation work first, and only when all simulations are
    completed or running does it start (at most ``alloc_specs["user"]["num_active_gens
↪"]``)
    generator instances.

    Allows for a ``alloc_specs["user"]["batch_mode"]`` where no generation
    work is given out unless all entries in ``H`` are returned.

    Can give points in highest priority, if ``"priority"`` is a field in ``H``.
    If alloc_specs["user"]["give_all_with_same_priority"] is set to True, then
    all points with the same priority value are given as a batch to the sim.

    Workers performing sims will be assigned resources given in H["resource_sets"]
    this field exists, else defaulting to one. Workers performing gens are
    assigned resource_sets given by persis_info["gen_resources"] or zero.

    This is the default allocation function if one is not defined.

    tags: alloc, default, batch, priority

    .. seealso::
        `test_uniform_sampling.py <https://github.com/Libensemble/libensemble/blob/
↪develop/libensemble/tests/functionality_tests/test_uniform_sampling.py>`_ # noqa
    """

    user = alloc_specs.get("user", {})

    if "cancel_sims_time" in user:
        # Cancel simulations that are taking too long
        rows = np.where(np.logical_and.reduce((H["sim_started"], ~H["sim_ended"], ~H[
↪"cancel_requested"])))[0]
        inds = time.time() - H["sim_started_time"][rows] > user["cancel_sims_time"]
        to_request_cancel = rows[inds]
        for row in to_request_cancel:
            H[row]["cancel_requested"] = True

    if libE_info["sim_max_given"] or not libE_info["any_idle_workers"]:
        return {}, persis_info

    # Initialize alloc_specs["user"] as user.
    batch_give = user.get("give_all_with_same_priority", False)
    gen_in = gen_specs.get("in", [])

    manage_resources = libE_info["use_resource_sets"]
```

(continues on next page)

(continued from previous page)

```
    support = AllocSupport(W, manage_resources, persis_info, libE_info)
    gen_count = support.count_gens()
    Work = {}

    points_to_evaluate = ~H["sim_started"] & ~H["cancel_requested"]
    for wid in support.avail_worker_ids():
        if np.any(points_to_evaluate):
            sim_ids_to_send = support.points_by_priority(H, points_avail=points_to_evaluate, batch=batch_give)
            try:
                Work[wid] = support.sim_work(wid, H, sim_specs["in"], sim_ids_to_send, persis_info.get(wid))
            except InsufficientFreeResources:
                break
            points_to_evaluate[sim_ids_to_send] = False
        else:
            # Allow at most num_active_gens active generator instances
            if gen_count >= user.get("num_active_gens", gen_count + 1):
                break

            # Do not start gen instances in batch mode if workers still working
            if user.get("batch_mode") and not support.all_sim_ended(H):
                break

            # Give gen work
            return_rows = range(len(H)) if gen_in else []
            try:
                Work[wid] = support.gen_work(wid, gen_in, return_rows, persis_info.get(wid))
            except InsufficientFreeResources:
                break
            gen_count += 1

    return Work, persis_info
```

fast_alloc

`fast_alloc.`**`give_sim_work_first`**(*W*, *H*, *sim_specs*, *gen_specs*, *alloc_specs*, *persis_info*, *libE_info*)

This allocation function gives (in order) entries in `H` to idle workers to evaluate in the simulation function. The fields in `sim_specs["in"]` are given. If all entries in *H* have been given a be evaluated, a worker is told to call the generator function, provided this wouldn't result in more than `alloc_specs["user"]["num_active_gen"]` active generators.

This fast_alloc variation of give_sim_work_first is useful for cases that simply iterate through H, issuing evaluations in order and, in particular, is likely to be faster if there will be many short simulation evaluations, given that this function contains fewer column length operations.

tags: alloc, simple, fast

See also:

test_fast_alloc.py # noqa

fast_alloc.py

```python
from libensemble.tools.alloc_support import AllocSupport, InsufficientFreeResources


def give_sim_work_first(W, H, sim_specs, gen_specs, alloc_specs, persis_info, libE_info):
    """
    This allocation function gives (in order) entries in ``H`` to idle workers
    to evaluate in the simulation function. The fields in ``sim_specs["in"]``
    are given. If all entries in `H` have been given a be evaluated, a worker
    is told to call the generator function, provided this wouldn't result in
    more than ``alloc_specs["user"]["num_active_gen"]`` active generators.

    This fast_alloc variation of give_sim_work_first is useful for cases that
    simply iterate through H, issuing evaluations in order and, in particular,
    is likely to be faster if there will be many short simulation evaluations,
    given that this function contains fewer column length operations.

    tags: alloc, simple, fast

    .. seealso::
        `test_fast_alloc.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_fast_alloc.py>`_ # noqa
    """

    if libE_info["sim_max_given"] or not libE_info["any_idle_workers"]:
        return {}, persis_info

    user = alloc_specs.get("user", {})
    manage_resources = libE_info["use_resource_sets"]

    support = AllocSupport(W, manage_resources, persis_info, libE_info)

    gen_count = support.count_gens()
    Work = {}
    gen_in = gen_specs.get("in", [])

    for wid in support.avail_worker_ids():
        # Skip any cancelled points
        while persis_info["next_to_give"] < len(H) and H[persis_info["next_to_give"]]["cancel_requested"]:
            persis_info["next_to_give"] += 1

        # Give sim work if possible
        if persis_info["next_to_give"] < len(H):
            try:
                Work[wid] = support.sim_work(wid, H, sim_specs["in"], [persis_info["next_to_give"]], [])
            except InsufficientFreeResources:
                break
            persis_info["next_to_give"] += 1

        elif gen_count < user.get("num_active_gens", gen_count + 1):
```

(continues on next page)

(continued from previous page)

```
            # Give gen work
            return_rows = range(len(H)) if gen_in else []
            try:
                Work[wid] = support.gen_work(wid, gen_in, return_rows, persis_info.
↪get(wid))
            except InsufficientFreeResources:
                break
            gen_count += 1
            persis_info["total_gen_calls"] += 1

    return Work, persis_info
```

start_only_persistent

start_only_persistent.**only_persistent_gens**(*W*, *H*, *sim_specs*, *gen_specs*, *alloc_specs*, *persis_info*, *libE_info*)

This allocation function will give simulation work if possible, but otherwise start up to `alloc_specs["user"]["num_active_gens"]` persistent generators (defaulting to one).

By default, evaluation results are given back to the generator once all generated points have been returned from the simulation evaluation. If `alloc_specs["user"]["async_return"]` is set to True, then any returned points are given back to the generator.

If any workers are marked as zero_resource_workers, then these will only be used for generators.

If any of the persistent generators has exited, then ensemble shutdown is triggered.

User options:

To be provided in calling script: E.g., `alloc_specs["user"]["async_return"] = True`

init_sample_size: int, optional
Initial sample size - always return in batch. Default: 0

num_active_gens: int, optional
Maximum number of persistent generators to start. Default: 1

async_return: Boolean, optional
Return results to gen as they come in (after sample). Default: False (batch return).

active_recv_gen: Boolean, optional
Create gen in active receive mode. If True, the manager does not need to wait for a return from the generator before sending further returned points. Default: False

tags: alloc, batch, async, persistent, priority

See also:

test_persistent_uniform_sampling.py # noqa test_persistent_uniform_sampling_async.py # noqa test_persistent_surmise_calib.py # noqa test_persistent_uniform_gen_decides_stop.py # noqa

start_only_persistent.**only_persistent_workers**(*W*, *H*, *sim_specs*, *gen_specs*, *alloc_specs*, *persis_info*, *libE_info*)

This allocation function will give simulation work if possible to any worker not listed as a zero_resource_worker. On the first call, the worker will be placed into a persistent state that will be maintained until libE is exited.

Otherwise, zero resource workers will be given up to a maximum of `alloc_specs["user"]["num_active_gens"]` persistent generators (defaulting to one).

By default, evaluation results are given back to the generator once all generated points have been returned from the simulation evaluation. If `alloc_specs["user"]["async_return"]` is set to True, then any returned points are given back to the generator.

If any of the persistent generators has exited, then ensemble shutdown is triggered.

Note, that an alternative to using zero resource workers would be to set a fixed number of simulation workers in persistent state at the start, allowing at least one worker for the generator - a minor alteration.

User options:

To be provided in calling script: E.g., `alloc_specs["user"]["async_return"] = True`

init_sample_size: int, optional
Initial sample size - always return in batch. Default: 0

num_active_gens: int, optional
Maximum number of persistent generators to start. Default: 1

async_return: Boolean, optional
Return results to gen as they come in (after sample). Default: False (batch return).

active_recv_gen: Boolean, optional
Create gen in active receive mode. If True, the manager does not need to wait for a return from the generator before sending further returned points. Default: False

See also:

test_persistent_gensim_uniform_sampling.py # noqa

start_only_persistent.py

```
import numpy as np

from libensemble.message_numbers import EVAL_GEN_TAG, EVAL_SIM_TAG
from libensemble.tools.alloc_support import AllocSupport, InsufficientFreeResources


def only_persistent_gens(W, H, sim_specs, gen_specs, alloc_specs, persis_info, libE_
↪info):
    """
    This allocation function will give simulation work if possible, but
    otherwise start up to ``alloc_specs["user"]["num_active_gens"]``
    persistent generators (defaulting to one).

    By default, evaluation results are given back to the generator once
    all generated points have been returned from the simulation evaluation.
    If ``alloc_specs["user"]["async_return"]`` is set to True, then any
    returned points are given back to the generator.

    If any workers are marked as zero_resource_workers, then these will only
    be used for generators.

    If any of the persistent generators has exited, then ensemble shutdown
    is triggered.

    **User options**:
```

(continues on next page)

(continued from previous page)

```
    To be provided in calling script: E.g., ``alloc_specs["user"]["async_return"] = True``

    init_sample_size: int, optional
        Initial sample size - always return in batch. Default: 0

    num_active_gens: int, optional
        Maximum number of persistent generators to start. Default: 1

    async_return: Boolean, optional
        Return results to gen as they come in (after sample). Default: False (batch return).

    active_recv_gen: Boolean, optional
        Create gen in active receive mode. If True, the manager does not need to wait
        for a return from the generator before sending further returned points.
        Default: False

    tags: alloc, batch, async, persistent, priority

    .. seealso::
        `test_persistent_uniform_sampling.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_sampling.py>`_ # noqa
        `test_persistent_uniform_sampling_async.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_sampling_async.py>`_ # noqa
        `test_persistent_surmise_calib.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/regression_tests/test_persistent_surmise_calib.py>`_ # noqa
        `test_persistent_uniform_gen_decides_stop.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_uniform_gen_decides_stop.py>`_ # noqa
    """

    if libE_info["sim_max_given"] or not libE_info["any_idle_workers"]:
        return {}, persis_info

    # Initialize alloc_specs["user"] as user.
    user = alloc_specs.get("user", {})
    manage_resources = libE_info["use_resource_sets"]

    active_recv_gen = user.get("active_recv_gen", False)  # Persistent gen can handle irregular communications
    init_sample_size = user.get("init_sample_size", 0)  # Always batch return until this many evals complete
    batch_give = user.get("give_all_with_same_priority", False)

    support = AllocSupport(W, manage_resources, persis_info, libE_info)
    gen_count = support.count_persis_gens()
    Work = {}

```

(continues on next page)

(continued from previous page)

```
    # Asynchronous return to generator
    async_return = user.get("async_return", False) and sum(H["sim_ended"]) >= init_sample_size

    if gen_count < persis_info.get("num_gens_started", 0):
        # When a persistent worker is done, trigger a shutdown (returning exit condition of 1)
        return Work, persis_info, 1

    # Give evaluated results back to a running persistent gen
    for wid in support.avail_worker_ids(persistent=EVAL_GEN_TAG, active_recv=active_recv_gen):
        gen_inds = H["gen_worker"] == wid
        returned_but_not_given = np.logical_and.reduce((H["sim_ended"], ~H["gen_informed"], gen_inds))
        if np.any(returned_but_not_given):
            if async_return or support.all_sim_ended(H, gen_inds):
                point_ids = np.where(returned_but_not_given)[0]
                Work[wid] = support.gen_work(
                    wid,
                    gen_specs["persis_in"],
                    point_ids,
                    persis_info.get(wid),
                    persistent=True,
                    active_recv=active_recv_gen,
                )
                returned_but_not_given[point_ids] = False

    # Now the give_sim_work_first part
    points_to_evaluate = ~H["sim_started"] & ~H["cancel_requested"]
    avail_workers = support.avail_worker_ids(persistent=False, zero_resource_workers=False)
    for wid in avail_workers:
        if not np.any(points_to_evaluate):
            break

        sim_ids_to_send = support.points_by_priority(H, points_avail=points_to_evaluate, batch=batch_give)

        try:
            Work[wid] = support.sim_work(wid, H, sim_specs["in"], sim_ids_to_send, persis_info.get(wid))
        except InsufficientFreeResources:
            break

        points_to_evaluate[sim_ids_to_send] = False

    # Start persistent gens if no worker to give out. Uses zero_resource_workers if defined.
    if not np.any(points_to_evaluate):
        avail_workers = support.avail_worker_ids(persistent=False, zero_resource_workers=True)
```

(continues on next page)

(continued from previous page)

```
        for wid in avail_workers:
            if gen_count < user.get("num_active_gens", 1):
                # Finally, start a persistent generator as there is nothing else to do.
                try:
                    Work[wid] = support.gen_work(
                        wid,
                        gen_specs.get("in", []),
                        range(len(H)),
                        persis_info.get(wid),
                        persistent=True,
                        active_recv=active_recv_gen,
                    )
                except InsufficientFreeResources:
                    break

                persis_info["num_gens_started"] = persis_info.get("num_gens_started", 0) + 1
                gen_count += 1

    return Work, persis_info, 0


def only_persistent_workers(W, H, sim_specs, gen_specs, alloc_specs, persis_info, libE_info):
    """
    This allocation function will give simulation work if possible to any worker
    not listed as a zero_resource_worker. On the first call, the worker will be
    placed into a persistent state that will be maintained until libE is exited.

    Otherwise, zero resource workers will be given up to a maximum of
    ``alloc_specs["user"]["num_active_gens"]`` persistent generators (defaulting to one).

    By default, evaluation results are given back to the generator once
    all generated points have been returned from the simulation evaluation.
    If ``alloc_specs["user"]["async_return"]`` is set to True, then any
    returned points are given back to the generator.

    If any of the persistent generators has exited, then ensemble shutdown
    is triggered.

    Note, that an alternative to using zero resource workers would be to set
    a fixed number of simulation workers in persistent state at the start, allowing
    at least one worker for the generator - a minor alteration.

    **User options**:

    To be provided in calling script: E.g., ``alloc_specs["user"]["async_return"] = True``

    init_sample_size: int, optional
        Initial sample size - always return in batch. Default: 0

```

(continues on next page)

(continued from previous page)

```
    num_active_gens: int, optional
        Maximum number of persistent generators to start. Default: 1

    async_return: Boolean, optional
        Return results to gen as they come in (after sample). Default: False (batch return).

    active_recv_gen: Boolean, optional
        Create gen in active receive mode. If True, the manager does not need to wait
        for a return from the generator before sending further returned points.
        Default: False


    .. seealso::
        `test_persistent_gensim_uniform_sampling.py <https://github.com/Libensemble/libensemble/blob/develop/libensemble/tests/functionality_tests/test_persistent_sim_uniform_sampling.py>`_ # noqa
    """

    if libE_info["sim_max_given"] or not libE_info["any_idle_workers"]:
        return {}, persis_info

    # Initialize alloc_specs["user"] as user.
    user = alloc_specs.get("user", {})
    manage_resources = libE_info["use_resource_sets"]
    active_recv_gen = user.get("active_recv_gen", False)  # Persistent gen can handle irregular communications
    init_sample_size = user.get("init_sample_size", 0)  # Always batch return until this many evals complete
    batch_give = user.get("give_all_with_same_priority", False)

    support = AllocSupport(W, manage_resources, persis_info, libE_info)
    gen_count = support.count_persis_gens()
    Work = {}

    # Asynchronous return to generator
    async_return = user.get("async_return", False) and sum(H["sim_ended"]) >= init_sample_size

    if gen_count < persis_info.get("num_gens_started", 0):
        # When a persistent gen worker is done, trigger a shutdown (returning exit condition of 1)
        return Work, persis_info, 1

    # Give evaluated results back to a running persistent gen
    for wid in support.avail_worker_ids(persistent=EVAL_GEN_TAG, active_recv=active_recv_gen):
        gen_inds = H["gen_worker"] == wid
        returned_but_not_given = np.logical_and.reduce((H["sim_ended"], ~H["gen_informed"], gen_inds))
        if np.any(returned_but_not_given):
            if async_return or support.all_sim_ended(H, gen_inds):
```

(continues on next page)

(continued from previous page)

```
                point_ids = np.where(returned_but_not_given)[0]
                Work[wid] = support.gen_work(
                    wid,
                    gen_specs["persis_in"],
                    point_ids,
                    persis_info.get(wid),
                    persistent=True,
                    active_recv=active_recv_gen,
                )
                returned_but_not_given[point_ids] = False

    # Now the give_sim_work_first part
    points_to_evaluate = ~H["sim_started"] & ~H["cancel_requested"]
    avail_workers = list(
        set(support.avail_worker_ids(persistent=False, zero_resource_workers=False))
        | set(support.avail_worker_ids(persistent=EVAL_SIM_TAG, zero_resource_
↪workers=False))
    )
    for wid in avail_workers:
        if not np.any(points_to_evaluate):
            break

        sim_ids_to_send = support.points_by_priority(H, points_avail=points_to_evaluate,␣
↪batch=batch_give)
        try:
            # Note that resources will not change if worker is already persistent.
            Work[wid] = support.sim_work(
                wid, H, sim_specs["in"], sim_ids_to_send, persis_info.get(wid),␣
↪persistent=True
            )
        except InsufficientFreeResources:
            break

        points_to_evaluate[sim_ids_to_send] = False

    # Start persistent gens if no sim work to give out. Uses zero_resource_workers if␣
↪defined.
    if not np.any(points_to_evaluate):
        avail_workers = support.avail_worker_ids(persistent=False, zero_resource_
↪workers=True)

        for wid in avail_workers:
            if gen_count < user.get("num_active_gens", 1):
                # Finally, start a persistent generator as there is nothing else to do.
                try:
                    Work[wid] = support.gen_work(
                        wid,
                        gen_specs.get("in", []),
                        range(len(H)),
                        persis_info.get(wid),
                        persistent=True,
                        active_recv=active_recv_gen,
```

(continues on next page)

(continued from previous page)

```
                    )
                except InsufficientFreeResources:
                    break
                persis_info["num_gens_started"] = persis_info.get("num_gens_started", 0)␣
↪+ 1
                gen_count += 1
    del support
    return Work, persis_info, 0
```

start_persistent_local_opt_gens

libensemble.alloc_funcs.start_persistent_local_opt_gens.**start_persistent_local_opt_gens**(*W*, *H*, *sim_specs*, *gen_specs*, *alloc_specs*, *persis_info*, *libE_info*)

This allocation function will do the following:

- Start up a persistent generator that is a local opt run at the first point identified by APOSMM's decide_where_to_start_localopt. Note, it will do this only if at least one worker will be left to perform simulation evaluations.
- If multiple starting points are available, the one with smallest function value is chosen.
- If no candidate starting points exist, points from existing runs will be evaluated (oldest first).
- If no points are left, call the generation function.

tags: alloc, persistent, aposmm

See also:

test_uniform_sampling_then_persistent_localopt_runs.py # noqa

6.5.4 Calling Scripts

Below are example calling scripts used to populate specifications for each user function and libEnsemble before initiating libEnsemble via the primary `libE()` call. The primary libEnsemble-relevant portions have been highlighted in each example. Non-highlighted portions may include setup routines, compilation steps for user applications, or output processing. The first two scripts correspond to random sampling calculations, while the third corresponds to an optimization routine.

Many other examples of calling scripts can be found in libEnsemble's regression tests.

Local Sine Tutorial

This example is from the Local Sine `Tutorial`, meant to run with Python's multiprocessing as the primary `comms` method.

Listing 3: examples/tutorials/simple_sine/calling.py

```
import matplotlib.pyplot as plt
import numpy as np
from gen import gen_random_sample
from sim import sim_find_sine

from libensemble import Ensemble
from libensemble.specs import ExitCriteria, GenSpecs, LibeSpecs, SimSpecs

if __name__ == "__main__":  # Python-quirk required on macOS and windows
    libE_specs = LibeSpecs(nworkers=4, comms="local")

    gen_specs = GenSpecs(
        gen_f=gen_random_sample,  # Our generator function
        out=[("x", float, (1,))],  # gen_f output (name, type, size)
        user={
            "lower": np.array([-3]),  # lower boundary for random sampling
            "upper": np.array([3]),  # upper boundary for random sampling
            "gen_batch_size": 5,  # number of x's gen_f generates per call
        },
    )

    sim_specs = SimSpecs(
        sim_f=sim_find_sine,  # Our simulator function
        inputs=["x"],  # Input field names. "x" from gen_f output
        out=[("y", float)],  # sim_f output. "y" = sine("x")
    )

    exit_criteria = ExitCriteria(sim_max=80)  # Stop libEnsemble after 80 simulations

    ensemble = Ensemble(sim_specs, gen_specs, exit_criteria, libE_specs)
    ensemble.add_random_streams()  # setup the random streams unique to each worker
    ensemble.run()  # start the ensemble. Blocks until completion.

    history = ensemble.H  # start visualizing our results

    colors = ["b", "g", "r", "y", "m", "c", "k", "w"]

    for i in range(1, libE_specs.nworkers + 1):
        worker_xy = np.extract(history["sim_worker"] == i, history)
        x = [entry.tolist()[0] for entry in worker_xy["x"]]
        y = [entry for entry in worker_xy["y"]]
        plt.scatter(x, y, label="Worker {}".format(i), c=colors[i - 1])

    plt.title("Sine calculations for a uniformly sampled random distribution")
    plt.xlabel("x")
    plt.ylabel("sine(x)")
```

(continues on next page)

(continued from previous page)

```
    plt.legend(loc="lower right")
    plt.savefig("tutorial_sines.png")
```

Electrostatic Forces with Executor

These examples are from a test for evaluating the scaling capabilities of libEnsemble by calculating particle electrostatic forces through a user application. This application is registered with either the MPI or Balsam Executor, then submitted for execution in the `sim_f`. Note the use of the `parse_args()` and `save_libE_output()` convenience functions from the `tools` module in the first calling script.

Traditional Version

Listing 4: tests/scaling_tests/forces/forces_adv/run_libe_forces.py

```
#!/usr/bin/env python
import os
import sys

import numpy as np
from forces_simf import run_forces  # Sim func from current dir
from forces_support import check_log_exception, test_ensemble_dir, test_libe_stats

from libensemble import logger
from libensemble.executors.mpi_executor import MPIExecutor

# Import libEnsemble modules
from libensemble.libE import libE
from libensemble.manager import ManagerException
from libensemble.tools import add_unique_random_streams, parse_args, save_libE_output

PERSIS_GEN = False

if PERSIS_GEN:
    from libensemble.alloc_funcs.start_only_persistent import only_persistent_gens as␣
↪alloc_f
    from libensemble.gen_funcs.persistent_sampling import persistent_uniform as gen_f
else:
    from libensemble.alloc_funcs.give_sim_work_first import give_sim_work_first as alloc_
↪f
    from libensemble.gen_funcs.sampling import uniform_random_sample as gen_f


logger.set_level("INFO")  # INFO is now default

nworkers, is_manager, libE_specs, _ = parse_args()

sim_app = os.path.join(os.getcwd(), "../forces_app/forces.x")

if not os.path.isfile(sim_app):
    sys.exit("forces.x not found - please build first in ../forces_app dir")
```

(continues on next page)

(continued from previous page)

```

if is_manager:
    print(f"\nRunning with {nworkers} workers\n")

exctr = MPIExecutor()
exctr.register_app(full_path=sim_app, app_name="forces")

# Note: Attributes such as kill_rate are to control forces tests, this would not be a typical parameter.

# State the objective function, its arguments, output, and necessary parameters (and their sizes)
sim_specs = {
    "sim_f": run_forces,  # Function whose output is being minimized
    "in": ["x"],  # Name of input for sim_f
    "out": [("energy", float)],  # Name, type of output from sim_f
    "user": {
        "keys": ["seed"],
        "cores": 2,
        "sim_particles": 1e3,
        "sim_timesteps": 5,
        "sim_kill_minutes": 10.0,
        "particle_variance": 0.2,
        "kill_rate": 0.5,
        "fail_on_sim": False,
        "fail_on_submit": False,  # Won't occur if 'fail_on_sim' True
    },
}
# end_sim_specs_rst_tag

# State the generating function, its arguments, output, and necessary parameters.
gen_specs = {
    "gen_f": gen_f,  # Generator function
    "in": [],  # Generator input
    "out": [("x", float, (1,))],  # Name, type and size of data produced (must match sim_specs 'in')
    "user": {
        "lb": np.array([0]),  # Lower bound for random sample array (1D)
        "ub": np.array([32767]),  # Upper bound for random sample array (1D)
        "gen_batch_size": 1000,  # How many random samples to generate in one call
    },
}

if PERSIS_GEN:
    alloc_specs = {"alloc_f": alloc_f}
else:
    alloc_specs = {
        "alloc_f": alloc_f,
        "user": {
            "batch_mode": True,  # If true wait for all sims to process before generate more
            "num_active_gens": 1,  # Only one active generator at a time
```

(continues on next page)

(continued from previous page)

```
        },
    }

libE_specs["save_every_k_gens"] = 1000  # Save every K steps
libE_specs["sim_dirs_make"] = True  # Separate each sim into a separate directory
libE_specs["profile"] = False  # Whether to have libE profile on (default False)

# Maximum number of simulations
sim_max = 8
exit_criteria = {"sim_max": sim_max}

# Create a different random number stream for each worker and the manager
persis_info = {}
persis_info = add_unique_random_streams(persis_info, nworkers + 1)

try:
    H, persis_info, flag = libE(
        sim_specs,
        gen_specs,
        exit_criteria,
        persis_info=persis_info,
        alloc_specs=alloc_specs,
        libE_specs=libE_specs,
    )

except ManagerException:
    if is_manager and sim_specs["user"]["fail_on_sim"]:
        check_log_exception()
        test_libe_stats("Exception occurred\n")
else:
    if is_manager:
        save_libE_output(H, persis_info, __file__, nworkers)
        if sim_specs["user"]["fail_on_submit"]:
            test_libe_stats("Task Failed\n")
        test_ensemble_dir(libE_specs, "./ensemble", nworkers, sim_max)
```

Object + yaml Version

Listing 5: tests/scaling_tests/forces/forces_adv/run_libe_forces_from_yaml.py

```
#!/usr/bin/env python
import os
import sys

import numpy as np

from libensemble.ensemble import Ensemble
from libensemble.executors.mpi_executor import MPIExecutor
from libensemble.tools import add_unique_random_streams

```

(continues on next page)

(continued from previous page)

```
####################

sim_app = os.path.join(os.getcwd(), "../forces_app/forces.x")

if not os.path.isfile(sim_app):
    sys.exit("forces.x not found - please build first in ../forces_app dir")


####################

forces = Ensemble(parse_args=True)
forces.from_yaml("forces.yaml")

forces.logger.set_level("INFO")

if forces.is_manager:
    print(f"\nRunning with {forces.nworkers} workers\n")

exctr = MPIExecutor()
exctr.register_app(full_path=sim_app, app_name="forces")

forces.libE_specs["ensemble_dir_path"] = "./ensemble"
forces.gen_specs.user.update(
    {
        "lb": np.array([0]),
        "ub": np.array([32767]),
    }
)

forces.persis_info = add_unique_random_streams({}, forces.nworkers + 1)

forces.run()
forces.save_output(__file__)
```

Listing 6: tests/scaling_tests/forces/forces_adv/forces.yaml

```
libE_specs:
    save_every_k_gens: 1000
    sim_dirs_make: True
    profile: False

exit_criteria:
    sim_max: 8

sim_specs:
    sim_f: forces_simf.run_forces
    inputs:
        - x
    outputs:
        energy:
            type: float

```

(continues on next page)

(continued from previous page)

```
    user:
        keys:
            - seed
        cores: 1
        sim_particles: 1.e+3
        sim_timesteps: 5
        sim_kill_minutes: 10.0
        particle_variance: 0.2
        kill_rate: 0.5
        fail_on_sim: False
        fail_on_submit: False

gen_specs:
    gen_f: libensemble.gen_funcs.sampling.uniform_random_sample
    outputs:
        x:
            type: float
            size: 1
    user:
        gen_batch_size: 1000

alloc_specs:
    alloc_f: libensemble.alloc_funcs.give_sim_work_first.give_sim_work_first
    outputs:
        allocated:
            type: bool
    user:
        batch_mode: True
        num_active_gens: 1
```

Persistent APOSMM with Gradients

This example is also from the regression tests and demonstrates configuring a persistent run via a custom allocation function.

Listing 7: tests/regression_tests/test_persistent_aposmm_with_grad.py

```
"""
Runs libEnsemble with APOSMM with an NLopt local optimizer that uses gradient
information from the sim_f

Execute via one of the following commands (e.g. 3 workers):
   mpiexec -np 4 python test_persistent_aposmm_with_grad.py
   python test_persistent_aposmm_with_grad.py --nworkers 3 --comms local
   python test_persistent_aposmm_with_grad.py --nworkers 3 --comms tcp

When running with the above commands, the number of concurrent evaluations of
the objective function will be 2, as one of the three workers will be the
persistent generator.
"""

```

(continues on next page)

(continued from previous page)

```
# Do not change these lines - they are parsed by run-tests.sh
# TESTSUITE_COMMS: local mpi tcp
# TESTSUITE_NPROCS: 4
# TESTSUITE_EXTRA: true

import multiprocessing
import sys
from math import gamma, pi, sqrt

import numpy as np

import libensemble.gen_funcs

# Import libEnsemble items for this test
from libensemble.libE import libE
from libensemble.sim_funcs.six_hump_camel import six_hump_camel as sim_f
from libensemble.sim_funcs.six_hump_camel import six_hump_camel_func, six_hump_camel_grad

libensemble.gen_funcs.rc.aposmm_optimizers = "nlopt"
from time import time

from libensemble.alloc_funcs.persistent_aposmm_alloc import persistent_aposmm_alloc as␣
↪alloc_f
from libensemble.gen_funcs.persistent_aposmm import aposmm as gen_f
from libensemble.tests.regression_tests.support import six_hump_camel_minima as minima
from libensemble.tools import add_unique_random_streams, parse_args, save_libE_output

# Main block is necessary only when using local comms with spawn start method (default␣
↪on macOS and Windows).
if __name__ == "__main__":
    multiprocessing.set_start_method("fork", force=True)

    nworkers, is_manager, libE_specs, _ = parse_args()

    if is_manager:
        start_time = time()

    if nworkers < 2:
        sys.exit("Cannot run with a persistent worker if only one worker -- aborting...")

    n = 2
    sim_specs = {
        "sim_f": sim_f,
        "in": ["x"],
        "out": [("f", float), ("grad", float, n)],
    }

    gen_out = [
        ("x", float, n),
        ("x_on_cube", float, n),
        ("sim_id", int),
        ("local_min", bool),
```

(continues on next page)

(continued from previous page)

```
        ("local_pt", bool),
    ]

    gen_in = ["x", "f", "grad", "local_pt", "sim_id", "sim_ended", "x_on_cube", "local_
↪min"]

    gen_specs = {
        "gen_f": gen_f,
        "in": gen_in,
        "persis_in": gen_in,
        "out": gen_out,
        "user": {
            "initial_sample_size": 0,  # Don't need to do evaluations because the␣
↪sampling already done below
            "localopt_method": "LD_MMA",
            "rk_const": 0.5 * ((gamma(1 + (n / 2)) * 5) ** (1 / n)) / sqrt(pi),
            "stop_after_k_minima": 25,
            "xtol_rel": 1e-6,
            "ftol_rel": 1e-6,
            "max_active_runs": 6,
            "lb": np.array([-3, -2]),
            "ub": np.array([3, 2]),
        },
    }

    alloc_specs = {"alloc_f": alloc_f}

    persis_info = add_unique_random_streams({}, nworkers + 1)

    exit_criteria = {"sim_max": 1000}

    # Load in "already completed" set of 'x','f','grad' values to give to libE/persistent_
↪aposmm
    sample_size = len(minima)

    H0_dtype = [
        ("x", float, n),
        ("grad", float, n),
        ("sim_id", int),
        ("x_on_cube", float, n),
        ("sim_ended", bool),
        ("f", float),
        ("gen_informed", bool),
        ("sim_started", bool),
    ]
    H0 = np.zeros(sample_size, dtype=H0_dtype)

    # Two points in the following sample have the same best function value, which
    # tests the corner case for some APOSMM logic
    H0["x"] = np.round(minima, 1)
    H0["x_on_cube"] = (H0["x"] - gen_specs["user"]["lb"]) / (gen_specs["user"]["ub"] -␣
↪gen_specs["user"]["lb"])
```

(continues on next page)

(continued from previous page)

```
    H0["sim_id"] = range(sample_size)
    H0[["sim_started", "gen_informed", "sim_ended"]] = True

    for i in range(sample_size):
        H0["f"][i] = six_hump_camel_func(H0["x"][i])
        H0["grad"][i] = six_hump_camel_grad(H0["x"][i])

    # Perform the run
    H, persis_info, flag = libE(sim_specs, gen_specs, exit_criteria, persis_info, alloc_
↪specs, libE_specs, H0=H0)

    if is_manager:
        assert persis_info[1].get("run_order"), "Run_order should have been given back"
        assert (
            len(persis_info[1]["run_order"]) >= gen_specs["user"]["stop_after_k_minima"]
        ), "This test should have many runs started."
        assert len(H) < exit_criteria["sim_max"], "Test should have stopped early due to
↪'stop_after_k_minima'"

        print("[Manager]:", H[np.where(H["local_min"])]["x"])
        print("[Manager]: Time taken =", time() - start_time, flush=True)

        tol = 1e-5
        for m in minima:
            # The minima are known on this test problem.
            # We use their values to test APOSMM has identified all minima
            print(np.min(np.sum((H[H["local_min"]]["x"] - m) ** 2, 1)), flush=True)
            assert np.min(np.sum((H[H["local_min"]]["x"] - m) ** 2, 1)) < tol

        save_libE_output(H, persis_info, __file__, nworkers)
```

6.6 Release Notes

Below are the notes from all libEnsemble releases.

GitHub issues are referenced, and can be viewed with hyperlinks on the github releases page.

Date
November 8, 2023

New capabilities:

- New history array save options in libE_specs. #1103/#1139/#1141 * *save_H_on_completion* saves history before exiting main libE function. * *save_H_with_date* includes date and timestamp in the save. * *H_file_prefix* provides prefix for saved file. * *save_H_on_completion* defaults to True when *save_every_k_gens/sims* is set.

Support for Python versions:

- Adds support/testing for Python 3.12
- Removes testing of Python 3.8

Note

- Tests were run on Linux and MacOS with Python versions 3.9, 3.10, 3.11, 3.12

- Heterogeneous workflows tested on Frontier (OLCF), Polaris, and Perlmutter (NERSC).
- Tests were also run on Bebop and Improv LCRC systems.

Known Issues

- See known issues section in the documentation.

6.6.1 Release 1.0.0

Date
September 25, 2023

New capabilities:

- *libE_specs* option *final_gen_send* returns last results to the generator (replaces *final_fields*). #1086
- *libE_specs* option *reuse_output_dir* allows reuse of workflow and ensemble directories. #1028 #1041
- *libE_specs* option *calc_dir_id_width* no. of digits for calc ID in output sim/gen directories. #1052 / #1066
- Added *gen_num_procs* and *gen_num_gpus libE_specs* (and *persis_info*) options for resourcing a generator. #1068
- Added *gpu_env_fallback* option to platform fields - specifies a GPU environment variable (for non-MPI usage). #1050
- New MPIExecutor *submit()* argument *mpi_runner_type* specifies an MPI runner for current call only. #1054
- Allow oversubscription when using the *num_procs gen_specs["out"]* option. #1058
- sim/gen_specs can use *outputs* in place of *out* to be consistent with *inputs*. #1075
- Executor can be obtained from *libE_info* (4th parameter) in user functions. #1078

Breaking changes:

- *libE_specs* option *final_fields* is removed in favor of *final_gen_send*. #1086
- *libE_specs* option *kill_canceled_sims* now defaults to **False**. #1062
- *parse_args* is not run automatically by *Ensemble* constructor.

Updates to **Object Oriented** Ensemble interface:

- Added *parse_args* as option to *Ensemble* constructor. #1065
- The *executor* can be passed as an option to the *Ensemble* constructor. #1078
- Better handling of *Ensemble.add_random_streams* and *ensemble.persis_info*. #1074

Output changes:

- The worker ID suffix is removed from sim/gen output directories. #1041
- Separate *ensemble.log* and *libE_stats.txt* for different workflows directories. #1027 #1041
- Defaults to four digits for sim/gen ID in output directories (adds digits on overflow). #1052 / #1066

Bug fixes:

- Resolved PETSc/OpenMPI issue (when using the Executor). #1064
- Prevent *mpi4py* validation running during local comms (when using OO interface). #1065

Performance changes:

- Optimize *kill_cancelled_sims* function. #1043 / #1063

- *safe_mode* defaults to **False** (for performance). #1053

Updates to example functions:

- Multiple regression tests and examples ported to use OO ensemble interface. #1014

Update forces examples:

- Make persistent generator the default for both simple and GPU examples (inc. updated tutorials).
- Update to object oriented interface.
- Added separate variable resources example for forces GPU.
- Rename *multi_task* example to *multi_app*.

Documentation:

- General overhaul and simplification of documentation. #992

Note

- Tested platforms include Linux, MacOS, Windows, and major systems such as Frontier (OLCF), Polaris, and Perlmutter (NERSC). The major system tests ran heterogeneous workflows.
- Tested Python versions: (Cpython) 3.7, 3.8, 3.9, 3.10, 3.11.

Known Issues

- See known issues section in the documentation.

6.6.2 Release 0.10.2

Date
July 24, 2023

- Fixes issues with workflow directories: * Ensure relative paths are interpreted from where libEnsemble is run. #1020 * Create intermediate directories for workflow paths. #1017
- Fixes issue where libEnsemble pre-initialized a shared multiprocssing queue. #1026

Note

- Tested platforms include Linux, MacOS, Windows and major systems including Frontier (OLCF), Polaris (ALCF), Perlmutter (NERSC), Theta (ALCF) and Bebop. The major system tests ran heterogeneous workflows.

Known issues

- On systems using SLURM 23.02, some issues have been experienced when using `mpi4py` comms.
- See the known issues section in the documentation for more information (https://libensemble.readthedocs.io/en/main/known_issues.html).

6.6.3 Release 0.10.1

Date
July 10, 2023

Hotfix for breaking changes in Pydantic.

- Pin Pydantic to version < 2.
- Minor fixes for NumPy 1.25 deprecations.

Note

- Tested platforms include Linux, MacOS, Windows and major systems including Frontier (OLCF) and Perlmutter (NERSC). The major system tests ran heterogeneous workflows.
- Tested Python versions: (Cpython) 3.7, 3.8, 3.9, 3.10, 3.11.

Known issues

- See known issues section in the documentation.

6.6.4 Release 0.10.0

Date
May 26, 2023

New capabilities:

- Enhance portability and simplify the assignment of procs/GPUs to worker resources #928 / #983 * Auto-detect GPUs across systems (inc. Nvidia, AMD, and Intel GPUs). * Auto-determination of GPU assignment method by MPI runner or provided platform. * Portable *auto_assign_gpus* / *match_procs_to_gpus* and *num_gpus* arguments added to the MPI executor submit. * Add *set_to_gpus* function (similar to *set_to_slots*). * Allow users to specify known systems via option or environment variable. * Allow users to specify their own system configurations. * These changes remove a number of tweaks that were needed for particular platforms.
- Resource management supports GPU and non-GPU simulations in the same ensemble. #993 * User's can specify *num_procs* and *num_gpus* in the generator for each evaluation.
- Pydantic models are used for validating major libE input (input can be provided as classes or dictionaries). #878
- Added option to store output and ensemble directories in a workflow directory. #982
- Simplify user function interface. Valid user functions can accept <4 parameters and return <3 values. #971
- New option to parse settings from **TOML**. #745
- New *dry_run* option to *libE()* that checks scripts are valid and returns. #987
- Added an option to the executor submit function to pre-execute a script in the task environment. #996

Breaking changes:

- Removed old Balsam Executor. #921
- Ensemble class moved from *libensemble.api* to *libensemble.ensemble*. #1003
- Default to one resource set per simulation in dynamic scheduling mode. #996

Documentation:

- Added type hints/annotations for major modules/functions. #823
- Added Polaris Guide. #930

- Added Frontier Guide. #909
- Added PBS example scripts. #956 #930
- Streamlined and improved the readability of docs. #1004

Tests and Examples:

- Updated forces_gpu tutorial example. #956 * Source code edit is not required for the GPU version. * Reports whether running on device or host. * Increases problem size. * Added versions with persistent generator and multi-task (GPU v non-GPU).
- Moved multiple tests, generators, and simulators to the community repo.
- Added ytopt example. And updated heFFTe example. #943
- Support Python 3.11 #922

  Note

- Tested platforms include Linux, MacOS, Windows and major systems: Frontier (OLCF), Polaris (ALCF), and Perlmutter (NERSC). The major system tests ran heterogeneous workflows.
- Recent testing was also carried out on Summit (IBM Power9/LSF), but this was not possible at time of release.
- Tested Python versions: (Cpython) 3.7, 3.8, 3.9, 3.10, 3.11.

  Known issues

- See known issues section in the documentation.

6.6.5 Release 0.9.3

Date
October 13, 2022

New capabilities:

- New pair of utilities, *liberegister* and *libesubmit* (based on *PSI/J*), for easily preparing and launching libEnsemble workflows with local comms onto most machines and schedulers. #807
- New persistent support function to cancel sim_ids (*request_cancel_sim_ids*). #880
- *keep_state* option for persistent workers: this lets the manager know that the information being sent is intermediate. #880

Other enhancements:

- The Executor *manager_poll()* interface now sets consistent flags instead of literal strings. #877
- Some internal modules and the test suite now work on Windows. #869 #888
- Specifying the *num_resource_sets libE_specs* option instead of *zero_resource_workers* is now recommended except when using a fixed worker/resource mapping. Use `persis_info["gen_resources"]` to assign persistent generator resources (default is zero). #905
- An extraneous warning removed. #903

  Note

- Tested platforms include Linux, MacOS, Windows, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), Swing (A100 GPU system), Perlmutter (HPE Cray EX with A100 NVIDIA GPUs). For Perlmutter, see "Known issues" below.
- Tested Python versions: (Cpython) 3.7, 3.8, 3.9, 3.10.

Known issues

- At time of testing on Perlmutter there was an issue running concurrent applications on a node, following a recent system update. This also affects previous versions of libEnsemble, and is being investigated.
- See known issues section in the documentation.

6.6.6 Release 0.9.2

Date
July 06, 2022

New capabilities:

- Support auto-detection of PBS node lists. #602
- Added configuration options for *libE_stats.txt* file. #743
- Support for *spawn* and *forkserver* multiprocessing start methods. #797
- Note that macOS no longer switches to using *fork*. macOS (since Python 3.8) and Windows default to using *spawn*. When using *spawn*, we recommend placing calling script code in an `if __name__ == "__main__":` block. The multiprocessing interface can be used to switch methods (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.set_start_method).

Updates to example functions:

Added simple dynamic sampling example. #833 Added heFFTe example. #844 Regression tests separated into problem examples and functionality tests. #839

Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), Swing (A100 GPU system), Perlmutter (HPE Cray EX with A100 NVIDIA GPUs).
- Tested Python versions: (Cpython) 3.7, 3.8, 3.9, 3.10.

Known issues

- The APOSMM generator function has been noted to operate slower than expected with the *spawn* multiprocessing start method. For this reason we recommend using *fork* with APOSMM, when using *local* comms (*fork* is the default method on Linux systems).
- See known issues section in the documentation.

6.6.7 Release 0.9.1

Date
May 11, 2022

This release has minimal changes, but a large number of touched lines.

- Reformatted code for **black** compliance, including string normalization. #811, #814, #821
- Added Spock and Crusher guides. #802
- User can now set `calc_status` to any string (for output in libE_stats). #808
- Added a workflows community initiative file. #817

Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), Swing (A100 GPU system), Perlmutter (HPE Cray EX with A100 NVIDIA GPUs).
- Tested Python versions: (Cpython) 3.7, 3.8, 3.9, 3.10.

  Known issues

- See known issues section in the documentation.

6.6.8 Release 0.9.0

Date
Apr 29, 2022

Featured new capabilities:

- New *Balsam* Executor with multi-site capability (run user applications on remote systems). #631, #729
- Support for *funcX* (place user functions on remote systems). #712 / #713
- Added partial support for concurrent/futures interface. (cancel(), cancelled(), done(), running(), result(), exception() and context manager) #719

Breaking API / helper function changes:

See "Updating for libEnsemble v0.9.0" wiki for details: https://github.com/Libensemble/libensemble/wiki/Updating-for-libEnsemble-v0.9.0

- Scheduler options moved from *alloc_specs['user']* to *libE_specs*. #790
- *BalsamMPIExecutor* is now *LegacyBalsamMPIExecutor*. #729
- The exit_criteria *elapsed_wallclock_time* has been renamed *wallclock_max*. #750 (with a deprecation warning)
- Clearer and consistent naming of libE-protected fields in history array. #760

Updates to example functions:

- Moved some examples to new repository - [libe-community-examples](https://github.com/Libensemble/libe-community-examples) (VTMOP, DEAP, DeepDriveMD). #716, #721, #726
- Updates to Tasmanian examples to include asynchronous generator example. #727 / #732
- Added multi-task, multi-fidelity optimization regression tests using *ax*. #717 / #720

Other functionality enhancements:

- Non-blocking option added for persistent user function receives. #752
- Added *match_slots* option to resource scheduler. #746

Documentation:

- Added tutorial on assigning tasks to GPUs. #768
- Refactored Executor tutorial for simplicity. #749
- Added Perlmutter guide. #728
- Added Slurm guide. #728
- Refactored examples and tutorials - added exercises. #736 / #737
- Updated history array documentation with visual workflow example. #723

  Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), Swing (A100 GPU system), Perlmutter (HPE Cray EX with A100 NVIDIA GPUs).
- Tested Python versions: (Cpython) 3.7, 3.8, 3.9, 3.10.

Known issues

- OpenMPI does not work with direct MPI job launches in `mpi4py` comms mode, since it does not support nested MPI launches. (Either use local mode or the Balsam Executor.)
- See known issues section in the documentation for more issues.

6.6.9 Release 0.8.0

Date
Oct 20, 2021

Featured new capabilities:

- Variable resource workers (dynamic reassignment of resources to workers). #643
- Alternative libE interface. An Ensemble object is created and can be parameterized by a YAML file. #645
- Improved support classes/functions for alloc/gen/sims and executors.
- Many new example generator/simulators and workflows.

Breaking API / helper function changes:

See "Updating for libEnsemble v0.8.0" wiki for details: https://github.com/Libensemble/libensemble/wiki/Updating-for-libEnsemble-v0.8.0

- Resources management is now independent of the executor. #345
- The `'persis_in'` field has been added to gen_specs/sim_specs (instead of being hard-coded in alloc funcs). #626 / #670
- `alloc support` module is now a class. #643 / #656
- `gen_support` module is replaced by Persistent Worker support module (now a class). #609 / #671
- Remove `libE_` prefix from the logger. #608
- `avail_worker_ids` function should specify `EVAL_GEN_TAG` or `EVAL_SIM_TAG` instead of `True`. #615 #643
- Pass `libE_info` to allocation functions (allows more flexibility for user and efficiency improvements). #672
- `'given_back'` is now a protected libEnsemble field in the manager's history array. #651
- Several name changes to functions and parameters (See the wiki above for details). #529 / #659

Updates to example functions:

- Suite of distributed optimization methods for minimizing sums of convex functions. #647 / #649. Methods include:
- primal-dual sliding (https://arxiv.org/pdf/2101.00143).
- N-agent, or distributed gradient descent w/ gradient tracking (https://arxiv.org/abs/1908.11444).
- proximal sliding (https://arxiv.org/abs/1406.0919).
- Added batched construction for Tasmanian example. #644
- Added Tasmanian dependency to Spack package. spack/spack#25762
- Added VTMOP source code and example usage. #676

- Added a multi-fidelity persistent_gp regression test. #683 / #684
- Added a DeepDriveMD inspired workflow. #630
- Created a persistent sim example. #614 / #615
- Added an example where cancellations are given from the alloc func. #677

Other functionality changes:

- A helper function for generic task polling loop has been added. #572 / #612
- Break main loop now happens when sim_max is returned rather than given out. #624
- Enable a final communication with gen. #620 / #628
- Logging updates - includes timestamps, enhanced debug logging, and libEnsemble version. #629 / #674

Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), Swing (A100 GPU system).
- Tested Python versions: (Cpython) 3.6, 3.7, 3.8, 3.9, 3.10[1].

Known issues

- OpenMPI does not work with direct MPI job launches in `mpi4py` comms mode, since it does not support nested MPI launches. (Either use local mode or the Balsam Executor.)
- See known issues section in the documentation for more issues.

6.6.10 Release 0.7.2

Date
May 03, 2021

API additions:

- Active receive option added that allows irregular manager/worker communication patterns. (#527 / #595)
- A mechanism is added for the cancellation/killing of previously issued evaluations. (#528 / #595 / #596)
- A submit function is added in the base `Executor` class that runs a serial application locally. (#531 / #595)
- Added libEnsemble history array protected fields: *returned_time*, *last_given_time*, and *last_gen_time*. (#590)
- Updated libE_specs options (`mpi_comm` and `profile`). (#547 / #548)
- Explicit seeding of random streams in `add_unique_random_streams()` is now possible. (#542 / #545)

Updates to example functions:

- Added Surmise calibration generator function and two examples (regression tests). (#595)

Other changes:

- Better support for uneven worker to node distribution (including at sub-node level). (#591 / #600)
- Fixed crash when running on Windows. (#534)
- Fixed crash when running with empty *persis_info*. (#571 / #578)
- Error handling has been made more robust. (#592)
- Improve `H0` processing (esp. for pre-generated, but not evaluated points). (#536 / #537)

[1] A reduced set of tests were run for python 3.10 due to some unavailable test dependencies at time of release.

- A global `sim_id` is now given, rather than a local count, in _libE_stats.txt_. Also a global gen count is given. (#587, #588)
- Added support for Python 3.9. (#532 / Removed support for Python 3.5. (#562)
- Improve SLURM nodelist detection (more robust). (#560)
- Add check that user does not change protected history fields (Disable via `libE_specs['safe_mode'] = False`). (#541)
- Added `print_fields.py` script for better interrogating the output history files. (#558)
- In examples, `is_master` changed to `is_manager` to be consistent with manager/worker nomenclature. (#524)

Documentation:

- Added tutorial **Borehole Calibration with Selective Simulation Cancellation**. (#581 / #595)

  Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm).
- Tested Python versions: (Cpython) 3.6, 3.7, 3.8, 3.9.

  Known issues

- OpenMPI does not work with direct MPI job launches in `mpi4py` comms mode, since it does not support nested MPI launches (Either use local mode or Balsam Executor).
- See known issues section in the documentation for more issues.

6.6.11 Release 0.7.1

Date
Oct 15, 2020

Dependencies:

- `psutils` is now a required dependency. (#478 #491)

API additions:

- Executor updates:
  - Addition of a zero-resource worker option for persistent gens (does not allocate nodes to gen). (#500)
  - Multiple applications can be registered to the Executor (and submitted) by name. (#498)
  - Wait function added to Tasks. (#499)
- Gen directories can now be created with options analogous to those for sim dirs. (#349 / #489)

Other changes:

- Improve comms efficiency (Repack fields when NumPy version 1.15+). (#511)
- Fix multiprocessing error on macOS/Python3.8 (Use 'fork' instead of 'spawn'). (#502 / #503)

Updates to example functions:

- Allow APOSMM to trigger ensemble exit when condition reached. (#507)
- Improvement in how persistent APOSMM shuts down subprocesses (preventing PETSc MPI-abort). (#478)

Documentation:

- APOSMM Tutorial added. (#468)
- Writing guide for user functions added to docs (e.g., creating sim_f, gen_f, alloc_f). (#510)
- Addition of posters and presentations section to docs (inc. Jupyter notebooks/binder links). (#492 #497)

Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), and Bridges (HPE system at PSC).
- Cori (Cray XC40/Slurm) was not tested with release code due to system issues.
- Tested Python versions: (Cpython) 3.5, 3.6, 3.7, 3.8.

Known issues

- We currently recommend running in Central mode on Bridges, as distributed runs are experiencing hangs.
- OpenMPI does not work with direct MPI job launches in mpi4py comms mode, since it does not support nested MPI launches (Either use local mode or Balsam Executor).
- See known issues section in the documentation for more issues.

6.6.12 Release 0.7.0

Date
May 22, 2020

Breaking API changes:

- *Job_controller*/*Job* renamed to *Executor*/*Task* and `launch` function to `submit`. (#285)
- Executors/Resources/Utils moved into sub-packages. `utils` now in package `tools`. (#285)
- sim/gen/alloc support functions moved into `tools` sub-package. (#285)
- Restructuring of *sim* directory creation with `libE_specs` configuration options. E.g: When `sim_input_dir` is given, directories for each *sim* are created. (#267)
- User can supply a file called `node_list` (replaces `worker_list`). (#455)

API additions:

- Added gen_funcs.rc configuration framework with option to select APOSMM Optimizers for import. (#444)
- Provide `alloc_specs` defaults via *alloc_funcs.defaults* module. (#325)
- Added `extra_args` option to the Executor submit function to allow addition of arbitrary MPI runner options. (#445)
- Added `custom_info` argument to MPI Executor to allow overriding of detected settings. (#448)
- Added `libE_specs` option to disable log files. (#368)

Other changes:

- Added libEnsemble Conda package, hosted on conda-forge.
- Bugfix: Intermittent failures with repeated libE calls under *mpi4py* comms. Every libE call now uses its own duplicate of provided communicator and closes out. (#373/#387)
- More accurate timing in *libE_stats.txt*. (#318)
- Addition of new post-processing scripts.

Updates to example functions:

- Persistent APOSMM is now the recommended APOSMM (*aposmm.py* renamed to *old_aposmm.py*). (#435)
- New alloc/gen func: Finite difference parameters with noise estimation. (#350)
- New example gen func: Tasmanian UQ generator. (#351)
- New example gen func: Deap/NSGA2 generator. (#407)
- New example gen func to interface with VTMOP.
- New example sim func: Borehole. (#367)
- New example use-case: WarpX/APOSMM. (#425)

Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), Cori (Cray XC40/Slurm), and Bridges (HPE system at PSC).
- Tested Python versions: (Cpython) 3.5, 3.6, 3.7, 3.8.

Known issues

- We currently recommended running in Central mode on Bridges as distributed runs are experiencing hangs.
- See known issues section in the documentation for more issues.

6.6.13 Release 0.6.0

Date
December 4, 2019

API changes:

- sim/gen/alloc_specs options that do not directly involve these routines are moved to libE_specs (see docs). (#266, #269)
- sim/gen/alloc_specs now require user-defined attributes to be added under the `'user'` field (see docs and examples). (#266, #269)
- Addition of a utils module to help users create calling scripts. Includes an argument parser and utility functions. (#308)
- check_inputs() function is moved to the utils module. (#308)
- The libE_specs option `nprocesses` has been changed to `nworkers`. (#235)

New example functions:

- Addition of a persistent APOSMM generator function. (#217)

Other changes:

- Overhaul of documentation, including HPC platform guides and a new pdf structure. (inc. #232, #282)
- Addition of OpenMP threading and GPU support to forces test. (#250)
- Balsam job_controller now tested on Travis. (#47)

Note

- Tested platforms include Linux, MacOS, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), Bebop (Cray CS400/Slurm), and Cori (Cray XC40/Slurm).
- Tested Python versions: (Cpython) 3.5, 3.6, 3.7

Known issues

- These are unchanged from v0.5.0.
- A known issues section has now been added to the documentation.

6.6.14 Release 0.5.2

Date
August 19, 2019

- Code has been restructured to meet xSDK package policies for interoperable ECP software (version 0.5.0). #208
- The use of MPI.COMM_WORLD has been removed. Uses a duplicate of COMM_WORLD if no communicator passed (any process not in communicator returns with an exit code of 3). #108
- All output from libEnsemble goes via logger. MANAGER_WARNING level added. This level and above are echoed to stderr by default. API option to change echo level.
- Simulation directories are created only during sim_f calls are suffixed by _worker. #146
- New user function libE.check_inputs() can be used to check valid configuration of inputs. Can be called in serial or under MPI (see libE API). #65
- Installation option has been added to install dependencies used in tests `pip install libensemble[extras]`.
- A profiling option has been added to sim_specs. #170
- Results comparison scripts have been included for convenience.

Note

- Tested platforms include Linux, MacOS (**New**), Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), and Bebop (Cray CS400/Slurm).
- Tested Python versions: (Cpython) 3.5, 3.6, 3.7
- **Note** Support has been removed for Python 3.4 since it is officially retired. Also NumPy has removed support.

Known issues

- These are unchanged from v0.5.0.

6.6.15 Release 0.5.1

Date
July 11, 2019

- Fixed LSF resource detection for large jobs on LSF systems (e.g., Summit). #184
- Added support for macOS. #182
- Improved the documentation (including addition of beginner's tutorial and FAQ).

Note

- Tested platforms include Local Linux, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), and Bebop (Cray CS400/Slurm).
- Tested Python versions: (Cpython) 3.4, 3.5, 3.6, 3.7.

Known issues

- These are unchanged from v0.5.0.

6.6.16 Release 0.5.0

Date
May 22, 2019

- Added local (multiprocessing) and TCP options for manager/worker communications, in addition to mpi4py. (#42).
- Example: libEnsemble can be run on MOM/launch nodes (e.g., those of ALCF/Theta & OLCF/Summit) and can remotely detect compute resources.
- Example: libEnsemble can be run on a system without MPI.
- Example: libEnsemble can be run with a local manager and remote TCP workers.
- Added support for Summit/LSF scheduler in job controller.
- MPI job controller detects and retries launches on failure; adding resilience. (#143)
- Job controller supports option to extract/print job times in libE_stats.txt. (#136)
- Default logging level changed to INFO. (#164)
- Logging interface added, which allows user to change logging level and file. (#110)
- All worker logging and calculation stats are routed through manager.
- libEnsemble can be run without a gen_func, for example, when using a previously computed random sample. (#122)
- Aborts dump persis_info with the history.

Note

- **This version no longer supports Python 2.**
- Tested platforms include Local Linux, Theta (Cray XC40/Cobalt), Summit (IBM Power9/LSF), and Bebop (Cray CS400/Slurm).

Known issues

- OpenMPI does not work with direct MPI job launches in mpi4py comms mode, since it does not support nested MPI launches (Either use local mode or Balsam job controller).
- Local comms mode (multiprocessing) may fail if MPI is initialized before forking processors. This is thought to be responsible for issues combining with PETSc.
- Remote detection of logical cores via LSB_HOSTS (e.g., Summit) returns number of physical cores since SMT info not available.
- TCP mode does not support (1) more than one libEnsemble call in a given script or (2) the auto-resources option to the job controller.

6.6.17 Release 0.4.1

Date
February 20, 2019

- Logging no longer uses root logger (also added option to change libEnsemble log level). (#105)
- Added wait_on_run option for job controller launch to block until jobs have started. (#111)
- persis_info can be passed to sim as well as gen functions. (#112)
- Postprocessing scripts added to create performance/utilization graphs. (#102)

- New scaling test added (not part of current CI test suite). (#114)

6.6.18 Release 0.4.0

Date
November 7, 2018

- Separated job controller classes into different modules including a base class (API change).
- Added central_mode run option to distributed type (MPI) job_controllers (API addition). (#93)
- Made poll and kill job methods (API change).
- In job_controller, set_kill_mode is removed and replaced by a wait argument for a hard kill (API change).
- Removed register module - incorporated into job_controller (API change).
- APOSMM has improved asynchronicity when batch mode is false (with new example). (#96)
- Manager errors (instead of hangs) when alloc_f or gen_f don't return work when all workers are idle. (#95)

Known issues

- OpenMPI is not supported with direct MPI launches since nested MPI launches are not supported.

6.6.19 Release 0.3.0

Date
September 7, 2018

- Issues with killing jobs have been fixed. (#21)
- Fixed job_controller manager_poll to work with multiple jobs. (#62)
- API change: persis_info now included as an argument to libE and is returned from libE instead of gen_info
- Gen funcs: aposmm_logic module renamed to aposmm.
- New example gen and allocation functions.
- Updated Balsam launch script (with new Balsam workflow).
- History is dumped to file on manager or worker exception and MPI aborted (with exit code 1). (#46)
- Default logging level changed to DEBUG and redirected to file ensemble.log.
- Added directory of standalone tests (comms, job kills, and nested MPI launches).
- Improved and speeded up unit tests. (#68)
- Considerable documentation enhancements.

Known issues

- OpenMPI is not supported with direct MPI launches since nested MPI launches are not supported.

6.6.20 Release 0.2.0

Date
June 29, 2018

- Added job_controller interface (for portable user scripts).
- Added support for using the Balsam job manager. Enables portability and dynamic scheduling.
- Added autodetection of system resources.
- Scalability testing: Ensemble performed with 1023 workers on Theta (Cray XC40) using Balsam.
- Tested MPI libraries: MPICH and Intel MPI.

Known issues

- Killing MPI jobs does not work correctly on some systems (including Cray XC40 and CS400). In these cases, libEnsemble continues, but processes remain running.
- OpenMPI does not work correctly with direct launches (and has not been tested with Balsam).

6.6.21 Release 0.1.0

Date
November 30, 2017

- Initial release.

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

R

S

T

U

W