libEnsemble

libEnsemble Tutorial 2022

Stephen Hudson
Mathematics and Computer Science Division
Argonne National Laboratory

Argonne, IL
July 07, 2022

libEnsemble

History array

Stephen Hudson (June 2022)

H array - Introduction

H on manager (the global history array).

H is a ***numpy structured array***. This is a numpy array with named fields. Each row represents a simulation to evaluate.

Fields can hold different data types. The fields and types in H are defined by gen/sim_specs['out'] given by tuples.

In addition, there are protected fields (*sim_started, sim_ended,* and *gen_informed* are shown as examples).

User fields

sim_id	x	theta	f	sim_ started	sim_ ended	gen_ informed
-1	0.0, 0.0	0	0.0	False	False	False
-1	0.0, 0.0	0	0.0	False	False	False
-1	0.0, 0.0	0	0.0	False	False	False

The *sim_id* field in the manager's H array is usually the same as the index for generated points.

Example: Each simulation has two inputs (**x** and **theta**) and one output (**f**).

```
gen_specs['out'] = [('x', float, 2),
                    ('theta', int)]

sim_specs['out'] = [('f', float)]
```

H arrays – Generator is called

H on manager (the global history array).

H initialized. No points generated.

sim_id	x	theta	f	sim_started	sim_ended
-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

H receives generated data.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	0.0	False	False
1	1.0, 1.1	11	0.0	False	False
2	2.0, 2.1	11	0.0	False	False

gen_specs['in'] brings in inputs and outputs from previous simulations.

gen_specs['out'] can be used in generator for consistency

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

Generator function

```
def my_gen(H, persis_info, gen_specs, libE_info):
```

gen_specs['in'] = ['x', 'theta', 'f']

gen_specs['out'] = [('x', float, 2), ('theta', int)]

Worker 1

H sent to generator has no rows on first call – requesting points.

x	theta
0.0, 0.1	10
1.0, 1.1	11
2.0, 2.1	11

H_out

NOTE: As the generator did not supply ***sim_id***, manager assigns.

H arrays – Points are given out for evaluation

H on manager (the global history array).

The allocation function assigns rows to gens/sims.

- *sim_started* field is set to True as points are given out.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	0.0	**True**	False
1	1.0, 1.1	11	0.0	**True**	False
2	2.0, 2.1	11	0.0	False	False

H receives simulation result.

- *sim_ended* field is set to True

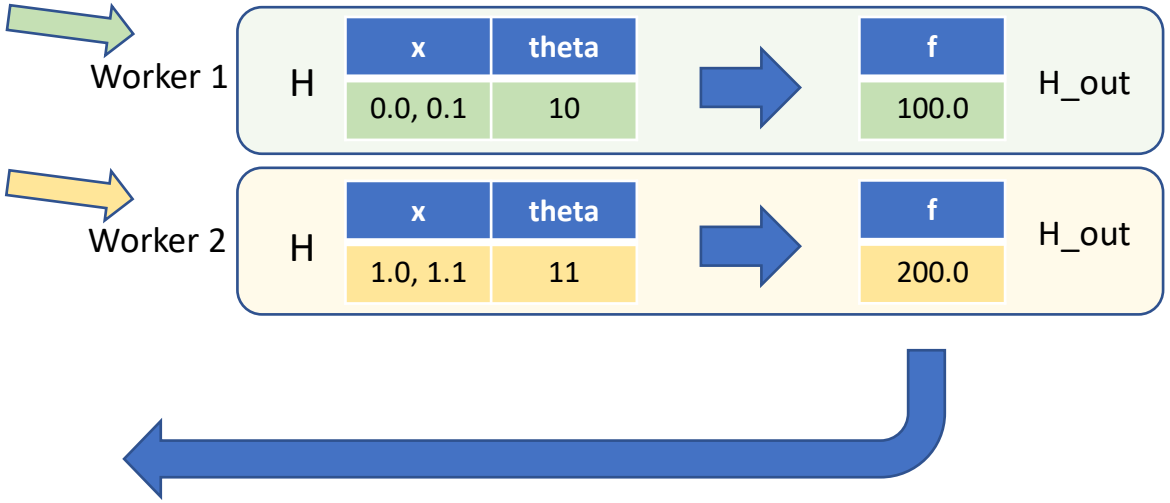
sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	100.0	True	**True**
1	1.0, 1.1	11	200.0	True	**True**
2	2.0, 2.1	11	0.0	False	False

History arrays in gen and sim functions are subsets of both rows and fields of the global H.

Simulator function

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

sim_specs['in'] = ['x', 'theta'] sim_specs['out'] = [('f', float)]



NOTE: Multiple rows can be given to the same worker in one allocation.

H arrays – Results returned to generator

H on manager (the global history array).

Returned points given back to the generator.

- Another protected field *gen_informed* (not shown) is set to True.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	100.0	True	True
1	1.0, 1.1	11	200.0	True	True
2	2.0, 2.1	11	0.0	False	False

H receives generated data.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	100.0	True	True
1	1.0, 1.1	11	200.0	True	True
2	2.0, 2.1	11	0.0	False	False
3	3.0, 3.1	11	0.0	False	False
4	4.0, 4.1	12	0.0	False	False

gen_specs['in'] may contain both evaluation input (x, theta) and output (f).

Generator function

gen_specs['in'] = ['x', 'theta', 'f']

gen_specs['out'] = [('x', float, 2), ('theta', int)]

Worker 1

H (calc_in)

x	theta	f
0.0, 0.1	10	100.0
1.0, 1.1	11	200.0

H_out

x	Theta
3.0, 3.1	11
4.0, 4.1	12

This generator creates N new points for every N results given back.

Persistent Generator

- Now a persistent gen (spot the difference)
- A persistent generator continues to run on a worker and communicates with the manager via send/recv functions.
- These are provided by the **PersistentSupport** module.
- Remember to add a worker for the persistent generator.

Non-persistent Generator	Persistent Generator

```python
def uniform_random_sample(H, persis_info, gen_specs, _):
    """Generate batch of random floats in [0,1)"""
    b = gen_specs["user"]["gen_batch_size"]

    H_o = np.zeros(b, dtype=gen_specs["out"])
    H_o["x"] = persis_info["rand_stream"].uniform(b)

    return H_o, persis_info
```

```python
def persistent_uniform(H, persis_info, gen_specs, libE_info):
    """Generate batches of random floats in [0,1)"""
    b = gen_specs["user"]["initial_batch_size"]

    ps = PersistentSupport(libE_info, EVAL_GEN_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(b)
        tag, _, H = ps.send_recv(H_o)

    return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG
```

Non-persistent Generator	Persistent Generator

`def uniform_random_sample(H, persis_info, gen_specs, _):`
`  """Generate batch of random floats in [0,1)"""`
`  b = gen_specs["user"]["gen_batch_size"]`
 	
`  H_o = np.zeros(b, dtype=gen_specs["out"])`
`  H_o["x"] = persis_info["rand_stream"].uniform(b)`
 	
`  return H_o, persis_info`
	`def persistent_uniform(H, persis_info, gen_specs, libE_info):`
	`  """Generate batches of random floats in [0,1)"""`
	`  b = gen_specs["user"]["initial_batch_size"]`
	
	`  ps = PersistentSupport(libE_info, EVAL_GEN_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(b)`
	`  tag, _, H = ps.send_recv(H_o)`
	
	`  return H_o, persis_info, FINISHED_PERSISTENT_GEN_TAG`

H arrays – Persistent generator is called

H on manager (the global history array).

H initialized. No points generated.

sim_id	x	theta	f	sim_started	sim_ended
-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

H receives generated data.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	0.0	False	False
1	1.0, 1.1	11	0.0	False	False
2	2.0, 2.1	11	0.0	False	False

gen_specs[‘in’] is empty - when the persistent generator is first called nothing is given to it. This may be different if using previous data (H0).

gen_specs[‘out’] can be used in generator for consistency

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

Persistent generator function

gen_specs[‘in’] = []

gen_specs[‘out’] = [(‘x’, float, 2), (‘theta’, int)]

Worker 1

H sent to generator is empty

x	theta
0.0, 0.1	10
1.0, 1.1	11
2.0, 2.1	11

H_out

NOTE: As the generator did not supply *sim_id*, manager assigns.

H arrays – Points are given out for evaluation

H on manager (the global history array).

The allocation function assigns rows to gens/sims.

- *sim_started* field is set to True as points are given out.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	0.0	**True**	False
1	1.0, 1.1	11	0.0	**True**	False
2	2.0, 2.1	11	0.0	False	False

H receives simulation result.

- *sim_ended* field is set to True

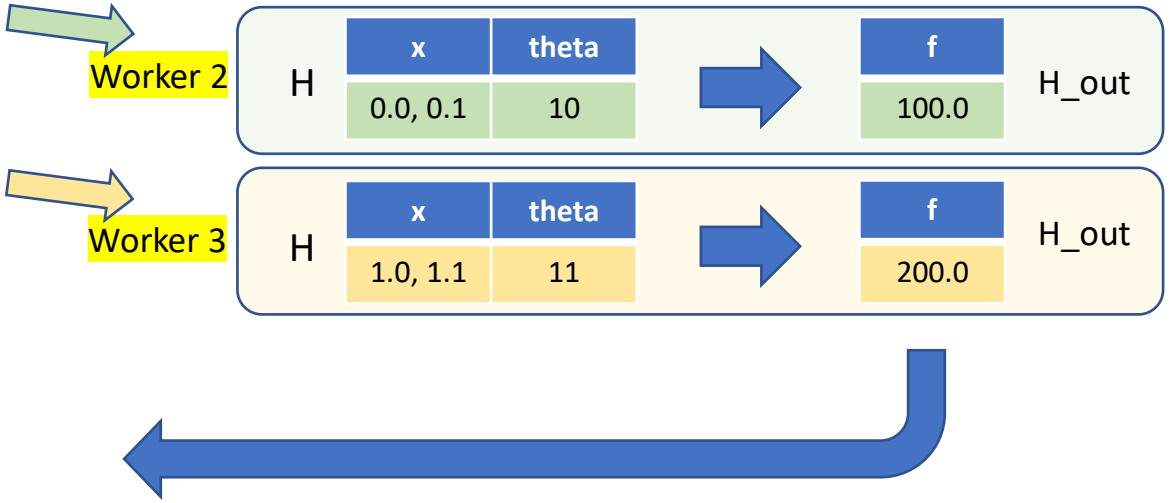
sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	100.0	True	**True**
1	1.0, 1.1	11	200.0	True	**True**
2	2.0, 2.1	11	0.0	False	False

History arrays in gen and sim functions are subsets of both rows and fields of the global H.

Simulator function

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

sim_specs['in'] = ['x', 'theta'] sim_specs['out'] = [('f', float)]



NOTE: Multiple rows can be given to the same worker in one allocation.

H arrays – Results returned to persistent generator

H on manager (the global history array).

Returned points given back to persistent generator.

- Another protected field *gen_informed* (not shown) is set to True.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	100.0	True	True
1	1.0, 1.1	11	200.0	True	True
2	2.0, 2.1	11	0.0	False	False

H receives generated data.

sim_id	x	theta	f	sim_started	sim_ended
0	0.0, 0.1	10	100.0	True	True
1	1.0, 1.1	11	200.0	True	True
2	2.0, 2.1	11	0.0	False	False
3	3.0, 3.1	11	0.0	False	False
4	4.0, 4.1	12	0.0	False	False

gen_specs['persis_in'] may contain both evaluation input (x, theta) and output (f) or, as in this case, just the output, as the persistent generator already has the input.

Persistent generator function

gen_specs['persis_in'] = ['f'] gen_specs['out'] = [('x', float, 2), ('theta', int)]

Worker 1

H (calc_in)

x	theta	f
0.0, 0.1	10	100.0
1.0, 1.1	11	200.0

H_out

x	Theta
3.0, 3.1	11
4.0, 4.1	12

This generator creates N new points for every N results given back.

Storing the history array

- The history array is usually written to file at the end of the ensemble – via the user's calling script.

- **Error handling -** libEnsemble captures exceptions from the manager and workers, and will write history array to file before closing down.

- You can also write the array after every N evaluations.

libEnsemble

Running User Applications

Stephen Hudson (June 2022)

Portable/flexible workflows

- How can user's reuse python scripts across various platforms?

- An executor is a portable interface that can ***execute*** applications on on various platforms.

- libEnsemble's executors include:
  - Base Executor – subprocess application in-place (e.g. serial/multi-threaded)
  - MPI Executor – Launch an MPI run (via detected runner).
  - Balsam Executor – Launch MPI or serial runs via Balsam (inc. remote systems).

Executor Interface

- Allows portable user-side functions.

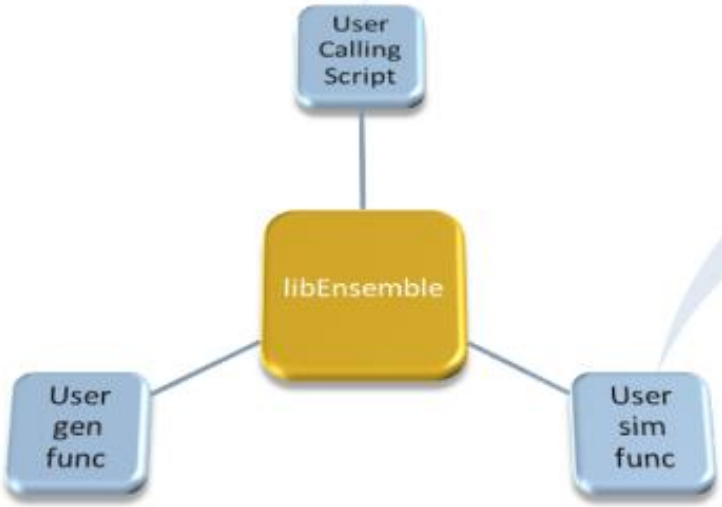
```
from libensemble.executors import MPIExecutor
exctr = MPIExecutor()  # Or BalsamExecutor
exctr.register_app(full_path=sim_app, app_name="forces")
```

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

```
exctr = Executor.executor
  task = exctr.submit(app_name="forces")
  while not task.finished:
    if 'Error' in task.read_stdout():
      task.kill()
    else:
      time.sleep(poll_interval)
      task.poll()
```

The main **Executor** functions are ***submit***, ***poll***, **wait** and ***kill***.

Retries launches on failure



Portable scripts Work with:

Job control systems	MPI runners	Worker concurrency
• Direct launch • Balsam • Other	• mpirun • srun • aprun • jsrun	• mpi4py • Multiprocessing • TCP

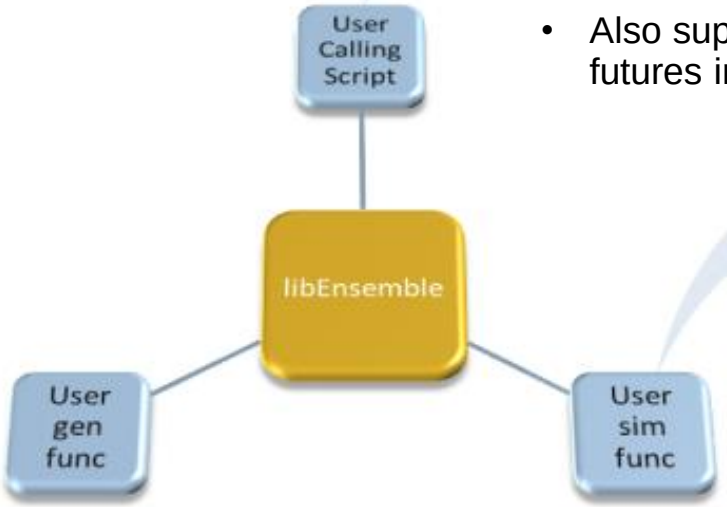
EXASCALE COMPUTING PROJECT

Executor Interface

- Allows portable user-side functions.

```
from libensemble.executors import MPIExecutor
exctr = MPIExecutor()  # Or BalsamExecutor
exctr.register_app(full_path=sim_app, app_name="forces")
```

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



- Also supports futures interface

```
exctr = Executor.executor
  future = exctr.submit(app_name="forces")
  while not future.done():
    if 'Error' in task.read_stdout():
      future.cancel()
    else:
      time.sleep(poll_interval)
      future.poll()
```

The main **Executor** functions are ***submit***, ***poll***, **wait** and ***kill***.

Retries launches on failure

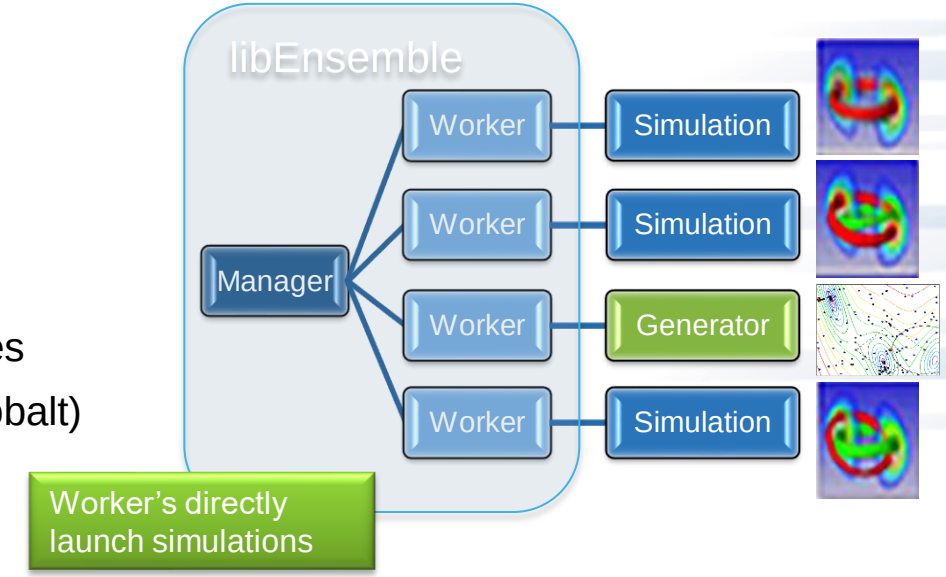
Portable scripts Work with:

Job control systems	MPI runners	Worker concurrency
• Direct launch • Balsam • Other	• mpirun • srun • aprun • jsrun	• mpi4py • Multiprocessing • TCP

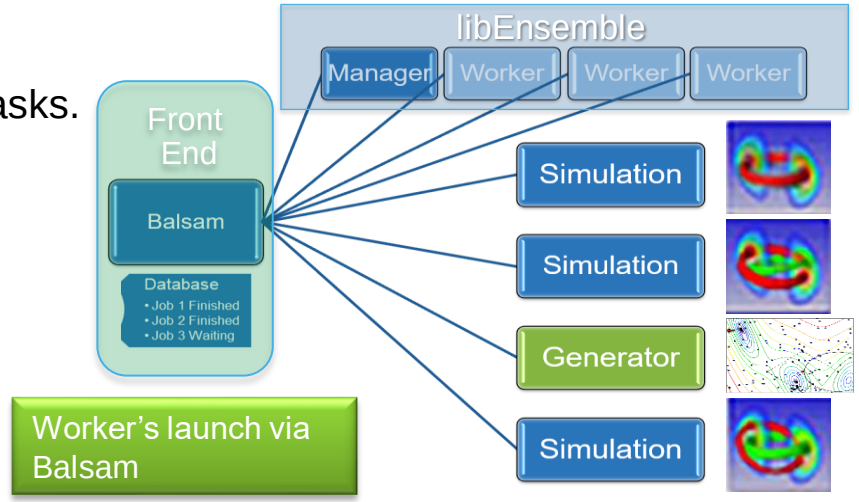
EXASCALE COMPUTING PROJECT

libEnsemble Manager/Workers

- Workers Launch MPI Applications
  - Use MPIExecutor to launch tasks (applications)
  - Possible on clusters → Launch from compute nodes
  - Supercomputers (inc. Theta) launch from MOM/Launch nodes
  - libEnsemble manages/schedules resources (slurm/lsf/pbs/cobalt)

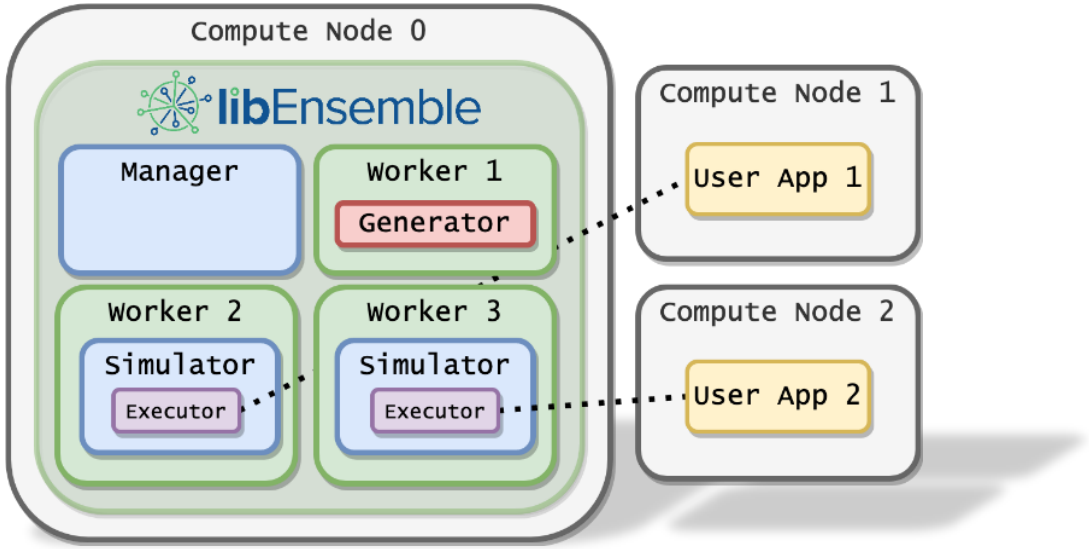


- Use Balsam as proxy job launcher.
  - Argonne (*Data Science Group*) Project
  - Balsam runs on front-end and maintains database of tasks.
  - Direct launches are replaced by creating Balsam tasks
  - Balsam dynamically schedules and manages tasks
  - Swap in BalsamExecutor to launch



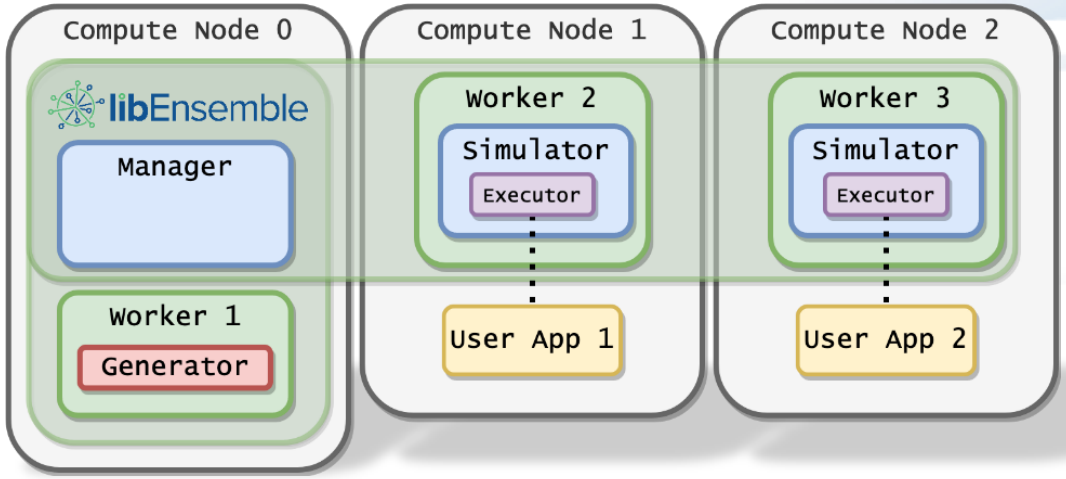
Flexible run configurations

- On compute nodes libEnsemble can be run in **dedicated** mode - does share nodes with the worker launched applications.



- `libE_specs['dedicated_mode'] = True`

- It is possible to **distribute workers over nodes**. User function can use local memory to interact with tasks.



- See example submission scripts

Example SLURM submission script

```
#!/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 systems, 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
```

- Examples in repo:
  - ***examples/libE_submission_scripts***
- Docs contain guides for multiple systems inc:
  - Summit
  - Perlmutter
  - Theta
  - Spock/Crusher
  - more…
  - https://libensemble.readthedocs.io/en/main/platforms/platforms_index.html

Executor Tutorial: Electrostatic Forces

- In the repo, navigate to: ***libensemble/tests/scaling_tests/forces***
- Now go into ***forces_app*** to build the application.
  - $ mpicc -O3 -o forces.x forces.c -lm
- To run with libEnsemble:
  - $ cd ../forces_simple
  - $ python run_libe_forces.py --comms local --nworkers 4

- Note this example produces an ***ensemble*** directory with output for each run of forces.
- See ***libE_stats.out*** for a summary of each simulation with timing.

Tutorial online: https://libensemble.readthedocs.io/en/main/tutorials/executor_forces_tutorial.html

Binder: https://mybinder.org/v2/gh/Libensemble/libensemble/develop?filepath=examples%2Ftutorials%2Fforces_tutorial_notebook.ipynb

Set up and run via Executor

Calling Script

```
from forces_simf import run_forces  # Sim from current dir
from libensemble.executors import MPIExecutor

# Initialize MPI Executor instance
exctr = MPIExecutor()

# Register simulation executable with executor
sim_app = os.path.join(os.getcwd(), "../forces_app/forces.x")
exctr.register_app(full_path=sim_app, app_name="forces")
```

Simulation function

```
import numpy as np
from libensemble.executors import Executor

def run_forces(H, persis_info, sim_specs, libE_info):
# Retrieve our MPI Executor instance
    exctr = Executor.executor
    task = exctr.submit(app_name="forces",  app_args=args)
    task.wait()

    data = np.loadtxt("forces.stat")
    final_energy = data[-1]
```

Tutorial online: https://libensemble.readthedocs.io/en/main/tutorials/executor_forces_tutorial.html

Binder: https://mybinder.org/v2/gh/Libensemble/libensemble/develop?filepath=examples%2Ftutorials%2Fforces_tutorial_notebook.ipynb

Exercises

- Use https://libensemble.readthedocs.io/en/main/executor/mpi_executor.html to modify the sim functions as follows:
  - Adjust the executor submit method to launch forces with four processes.
  - Adjust submit() again so the app's stdout and stderr are written to ``stdout.txt`` and ``stderr.txt`` respectively.
  - Set a timeout for the task, and kill if taking too long (see how long runs take in libE_stats.txt).

libEnsemble

Resource Manager

Stephen Hudson (June 2022)

Resource Management

- On HPC systems, libEnsemble is *usually* run within one job submission / node allocation.

- libEnsemble has its own resource manager
  - Resources are divided amongst workers.
  - Node-lists detected via standard env. variables on SLURM, PBS, COBALT, LSF.
  - Or supply a node-list in a file called 'nodelist' in the run directory.
  - Can disable with `libE_specs['disable_resource_manager'] == True`
  - Executor is aware of resources, and will use all cpu resources assigned to it, if not specified in submit function.

- Allows libEnsemble workflows to run consistently across various systems, irrespective of systems application level resource scheduling.

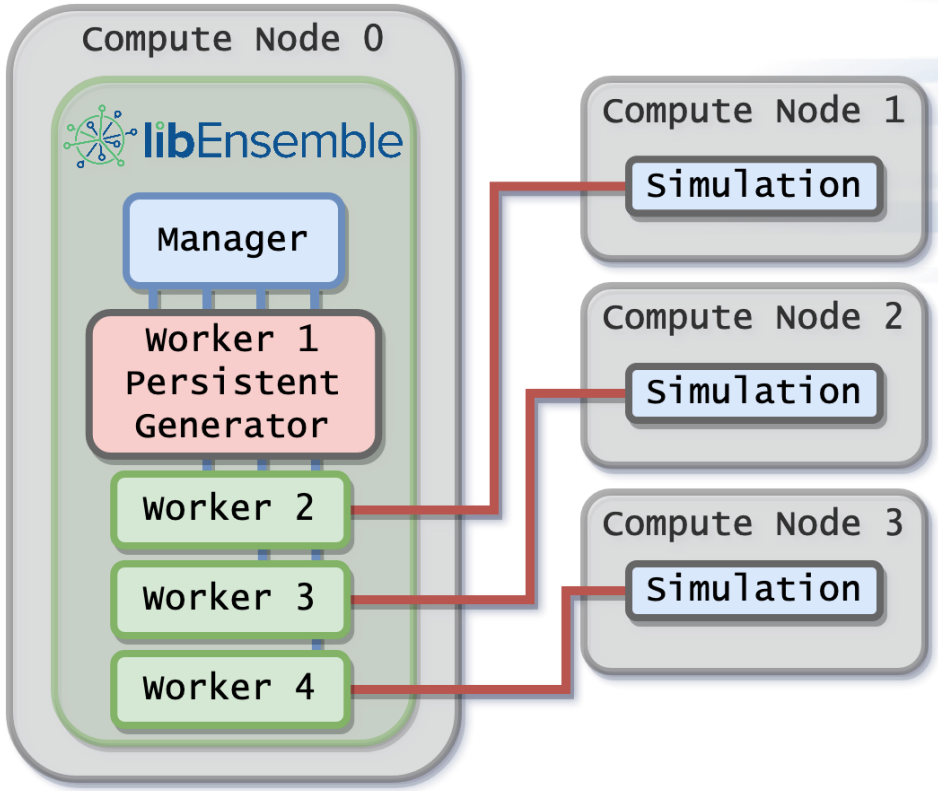
Zero-resource workers

- Most common case is that a persistent generator does not require resources.
  - Supply a list of zero-resource worker IDs and add an extra worker.

- In this example, run with 4 workers and set:
  - `libE_specs['zero_resource_workers'] == [1]`

Hint: The parse_args command line reader also has a **`nsim_workers`** option that will add the gen worker and set this option for you.

$ python run_libe_forces.py --comms local --nsim_workers 3



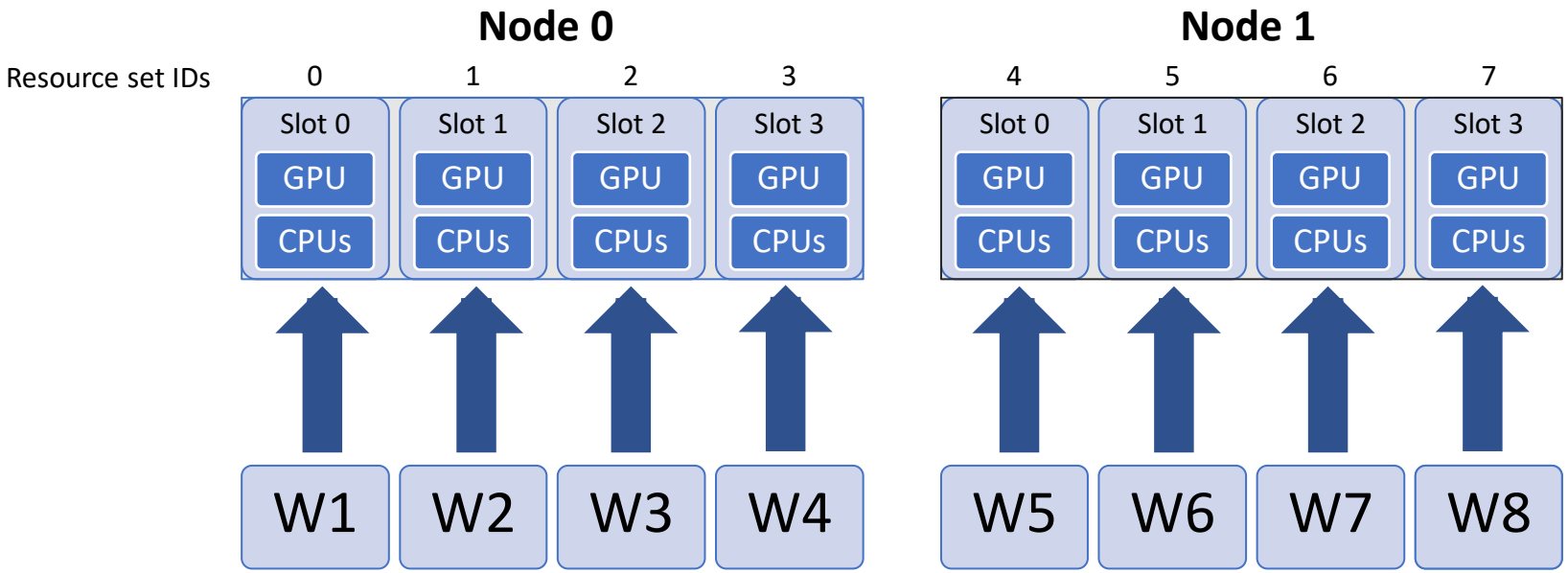
libEnsemble

Dynamic resource assignment

Stephen Hudson (June 2022)

EXASCALE COMPUTING PROJECT

- Run with as many workers as needed for smallest size simulations
  - One ***worker*** points to one ***resource set***.
  - If at sub-node level, slots are enumerated on a node.

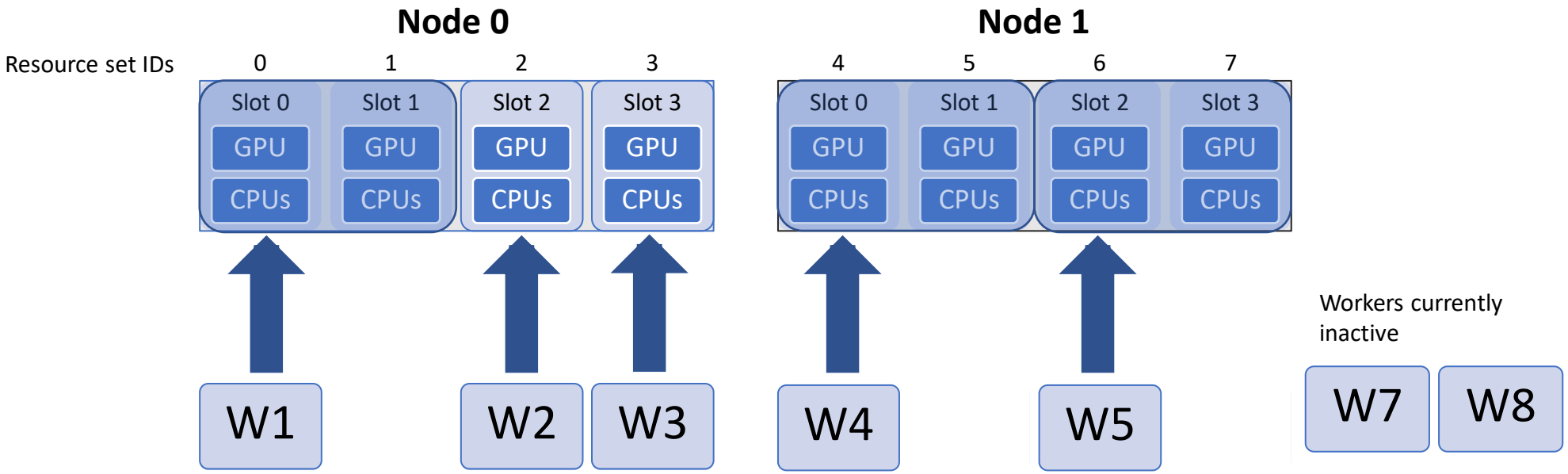


sim_id	x	f	Resource_sets
0	******	***	1
1	******	***	1
2	******	***	1
3	******	***	1
4	******	***	1
5	******	***	1
6	******	***	1
7	******	***	1

In calling script ...

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

- Generator provides a ***resource_sets*** field in H. Giving no. of resource sets required for each sim.
  - Allocation functions find smallest space on a node that fits required resources.
  - The next available worker is given work and resources.

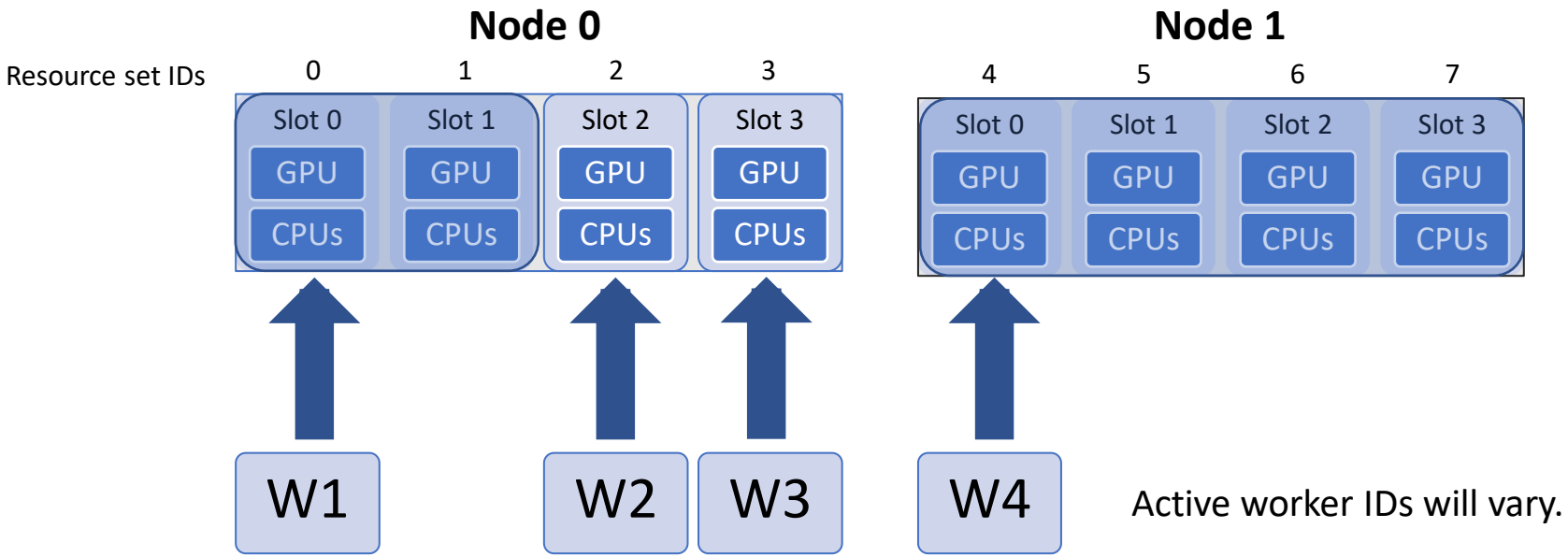


sim_id	x	f	Resource_sets
8	******	***	2
9	******	***	1
10	******	***	1
11	******	***	2
12	******	***	2 ⌛ Waiting for resources

In generator ...

```
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]
```

- The simulator can obtain resources available to **this** worker via Resources module.
  - MPIExecutor is aware of resources module, and will automate **num_nodes** etc.. if not supplied.



Active worker IDs will vary.

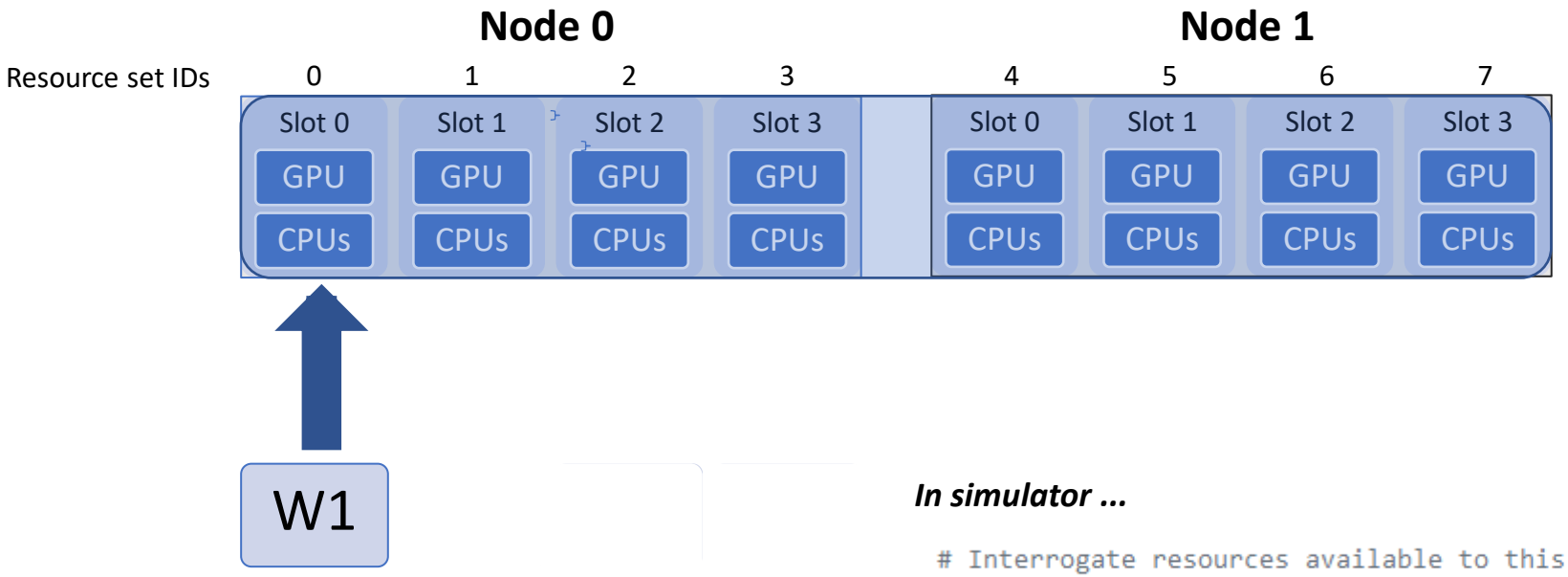
sim_id	x	f	Resource_sets
12	******	***	2
13	******	***	1
14	******	***	1
15	******	***	4

In simulator ...

```
# Interrogate resources available to this worker
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")
```

Sim:12
CUDA_VISIBLE_DEVICES = 0,1

- Multi-node scenarios
  - Currently, if ***resource_sets*** takes up more than one node, it will split evenly if possible or round up to full nodes.



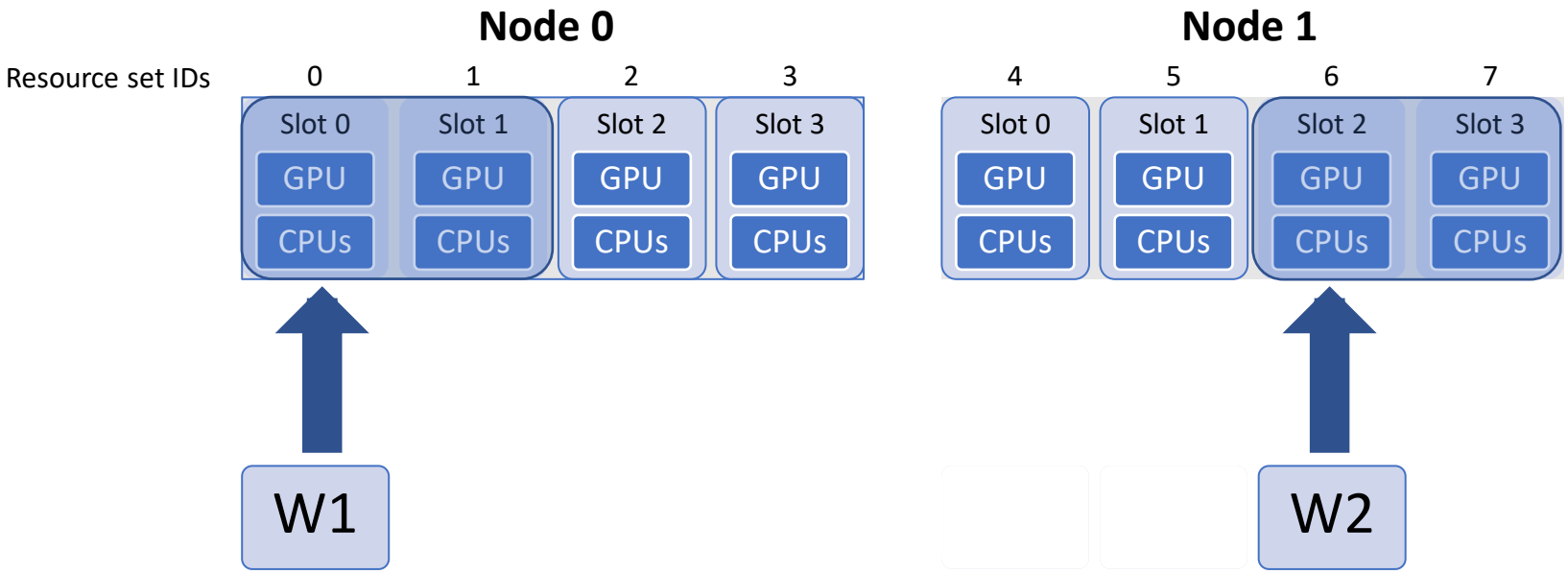
sim_id	x	f	Resource_sets
16	******	***	7
17	******	***	1
18	******	***	1
19	******	***	2

In simulator ...

```
# Interrogate resources available to this worker
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")
```

Sim:16
CUDA_VISIBLE_DEVICES = 0,1,2,3

- Scheduler Options
  - Sim 22 could fit on one node if all slots were free – but only 2 are free on each node.
  - **split2fit** (default: True) If True will split across nodes if an even split exists.
  - **match_slots** (default: True) If True slots much match between nodes.



Note: If 6 resource sets were requested, would split 3 per node, even if **split2fit** is False, as this can never fit on one node.

sim_id	x	f	Resource_sets
20	******	***	2
21	******	***	2
22	******	***	4

Waiting for resources

In calling script ...

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

This would allow sim 22 to be scheduled. But may be an issue if setting CUDA_VISIBLE_DEVICES.

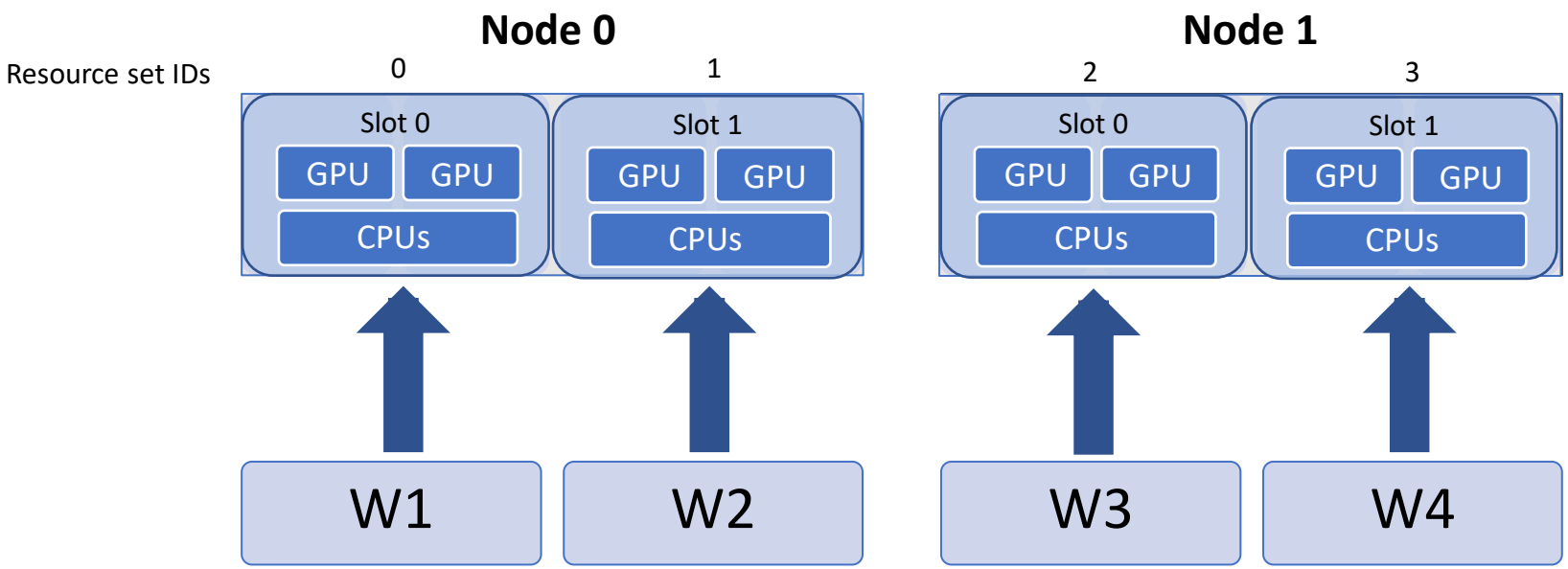
- **Example**: Sim function wants one CPU and one GPU per *resource_set* (where MPI tasks will be the number of resource sets).

```python
from libensemble.resources.resources import Resources
from libensemble.executors.executor import Executor

# Sim function
def my_sim(H, persis_info, sim_specs, libE_info):
    ...
    resources = Resources.resources.worker_resources
    # Convert Python list to comma delimited string
    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

    # Launch application via system MPI runner, using assigned resources.
    task = exctr.submit(app_name='my_application',
                                    app_args='input_file',
                                    num_nodes=num_nodes,
                                    ranks_per_node=cores_per_node,
                                    stdout='out.txt',
                                    stderr='err.txt')
```

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

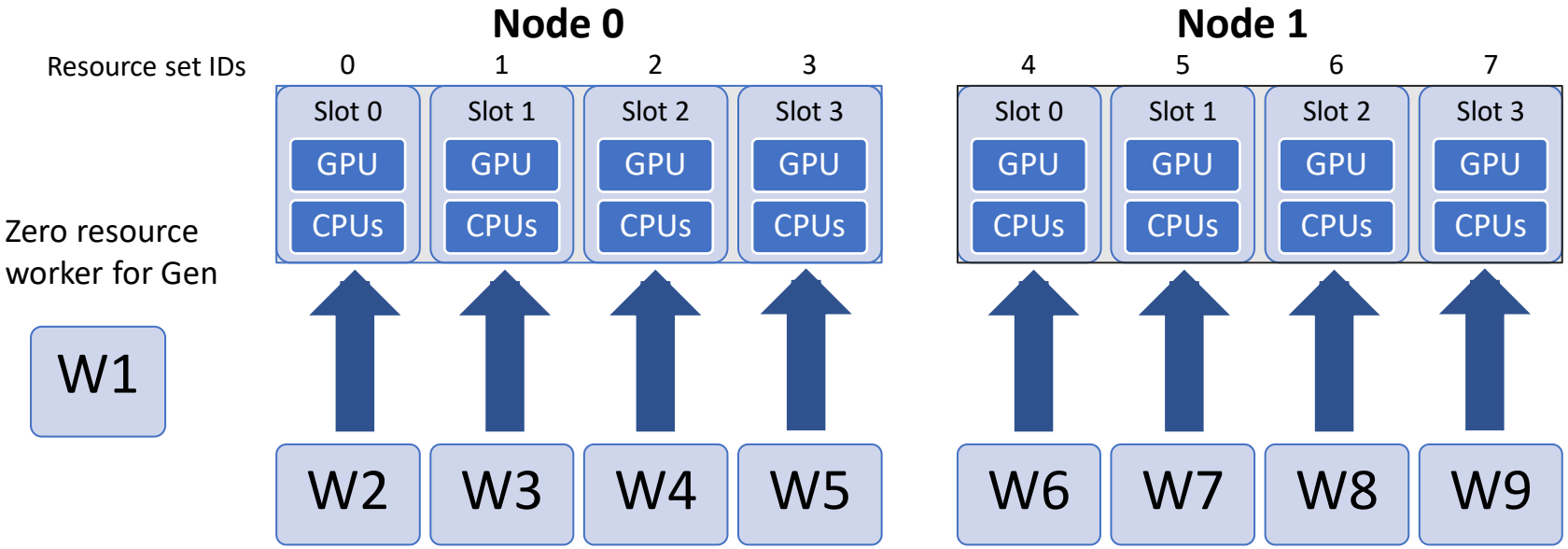


sim_id	x	f	Resource_sets
0	******	***	1
1	******	***	1
2	******	***	1
3	******	***	1

```
# Interrogate resources available to this worker
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")
```

```
# Interrogate resources available to this worker
resources = Resources.resources.worker_resources
resources.set_env_to_slots("CUDA_VISIBLE_DEVICES", multiplier=2)
```

- For a persistent generator. Run this example with 9 workers:
  - Either use a zero resource worker (if gen should always be same worker)
  - Or set **num_resources_sets** to 8 explicitly.



sim_id	x	f	Resource_sets
0	******	***	1
1	******	***	1
2	******	***	1
3	******	***	1
4	******	***	1
5	******	***	1
6	******	***	1
7	******	***	1

In calling script ...

Either:

```
libE_specs['zero_resource_workers'] = [1]
```

OR:

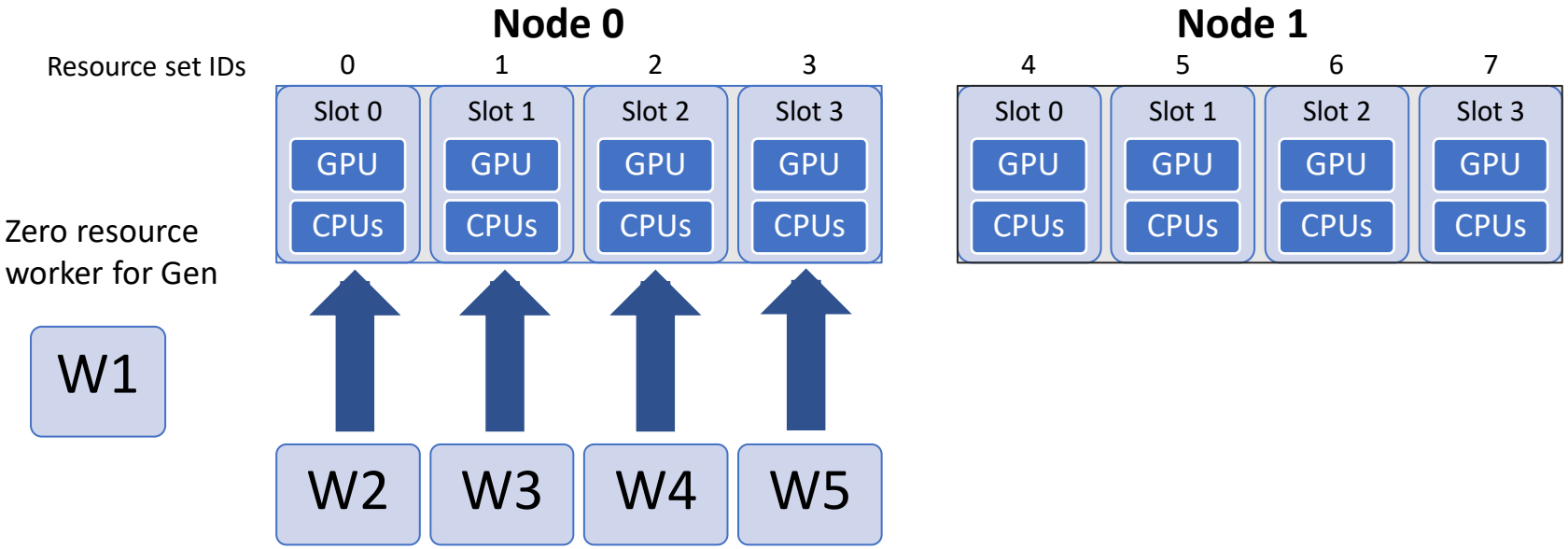
```
libE_specs['num_resource_sets'] = 8
```

To give generator resources set:

```
persis_info['gen_resources'] = 0
```

Default is zero (persistent workers keep their resources).

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



sim_id	x	f	Resource_sets
0	******	***	1
1	******	***	1
2	******	***	1
3	******	***	1
4	******	***	1
5	******	***	1
6	******	***	1
7	******	***	1

In calling script ...

```
libE_specs['num_resource_sets'] = 8
```

Can also specify on the command line ...

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

Where to find

- Tutorial:
  - Assign GPUs – Basic GPU example.
  - (in repository at ***libensemble/tests/scaling_tests/forces/forces_gpu***)

- Example regression tests:
  - ***test_persistent_gp.py***
  - ***test_persistent_sampling_CUDA_variable_resources.py*** (Demo – runs on CPU)

- Docs:
  - https://libensemble.readthedocs.io/en/main/resource_manager/overview.html

Executor Tutorial 2: Electrostatic Forces on GPU

- In the repo, navigate to: ***libensemble/tests/scaling_tests/forces***

- If running on a GPU go into ***forces_app*** to build the application with OMP TARGET line enabled.
  - Find correct build line from ***build_forces.sh***

- To run with libEnsemble:
  - $ cd ../forces_gpu
  - $ python run_libe_forces.py --comms local --nworkers 4

- On SLURM systems use env. variable `export SLURM_EXACT=True` when multiple user applications share a node.

Tutorial online: https://libensemble.readthedocs.io/en/main/tutorials/forces_gpu_tutorial.html
libEnsemble with GPUs demo: **https://www.youtube.com/watch?v=Av8ctYph7-Y**

Extract resources for this worker

Simulation function

Use worker resources information to configure run:

- Assign environment variables
- Set MPI command line options.

Find more options in docs:

https://libensemble.readthedocs.io/en/main/resource_manager/worker_resources.html

```
from libensemble.resources.resources import Resources


def run_forces(H, persis_info, sim_specs, libE_info):
    # Only showing new changed lines for varying resources
    resources = Resources.resources.worker_resources
    resources.set_env_to_slots("CUDA_VISIBLE_DEVICES")


    task = exctr.submit(
        app_name="forces", app_args=args,
        num_nodes=resources.local_node_count,
        procs_per_node=resources.slot_count,
        # extra_args="--gpus-per-task=1"  # Let slurm assign GPUs
    )
```

Tutorial online: https://libensemble.readthedocs.io/en/main/tutorials/forces_gpu_tutorial.html

Variable resources

Modify lines in calling script

Simulation function is unchanged.

You can uncomment prints in gen and sim to show resources.

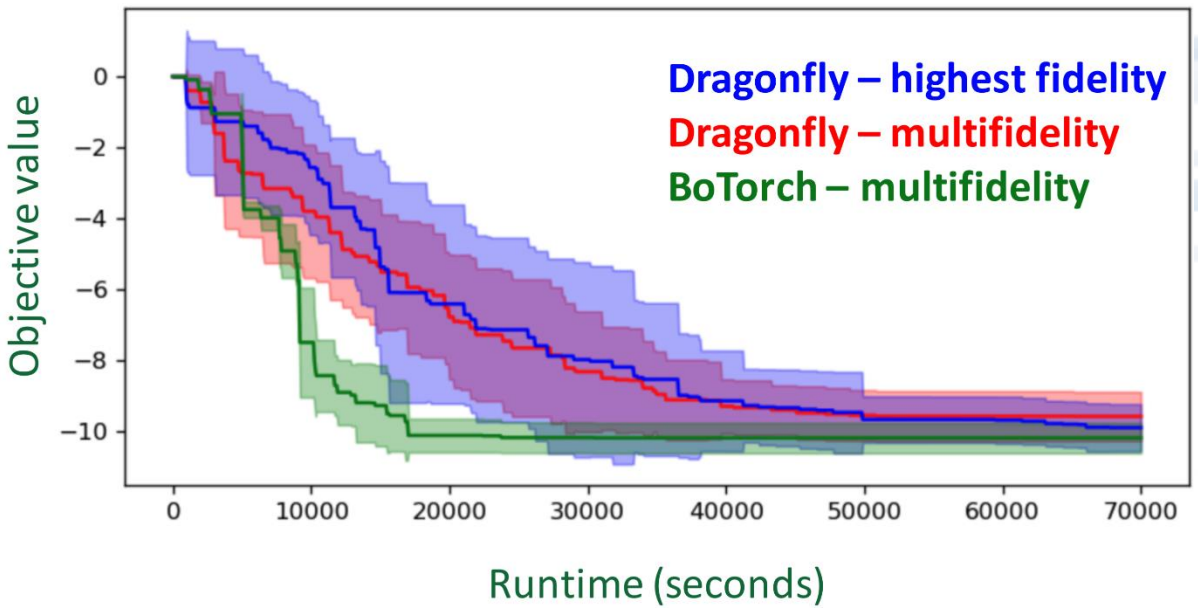
This simple example uses a random number of resource sets for each simulation.

```
from libensemble.gen_funcs.sampling import (
    uniform_random_sample_with_variable_resources as gen_f
)
gen_specs = {
    "gen_f":  gen_f , "in": [],
    "out": [
        ("x", float, (1,)),
        ("resource_sets", int)
    ],
    "user": {
        # ....
        "gen_batch_size": 8,
        "max_resource_sets": nworkers
    }
}
```

Tutorial online: https://libensemble.readthedocs.io/en/main/tutorials/forces_gpu_tutorial.html

Using libEnsemble for multi-fidelity simulations

- libEnsemble used for multi-fidelity ensembles with WarpX and FBPIC.
  - libEnsemble now coupled with Dragonfly and BoTorch optimization methods.
  - Methods observe simulation output and request subsequent simulations at various fidelity levels
  - libEnsemble dynamically allocates CPU/GPU resources as requested by the methods
  - Increased computational efficiency as less-expensive, lower fidelity simulations can guide numerical optimization methods



Progress from ten replications of libEnsemble+FBPIC with two multifidelity methods and a single (highest) fidelity method. Objective value is computed from the highest fidelity simulation.

- **libEnsemble docs:**
  https://libensemble.readthedocs.io

  libEnsemble with GPUs demo
  https://www.youtube.com/watch?v=Av8ctYph7-Y

  GitHub:
  https://github.com/Libensemble/libensemble

Thank you!