OpenMMTools MCMC framework

Andrea Rizzi
Chodera Lab, MSKCC

Group meeting
May 17th, 2017

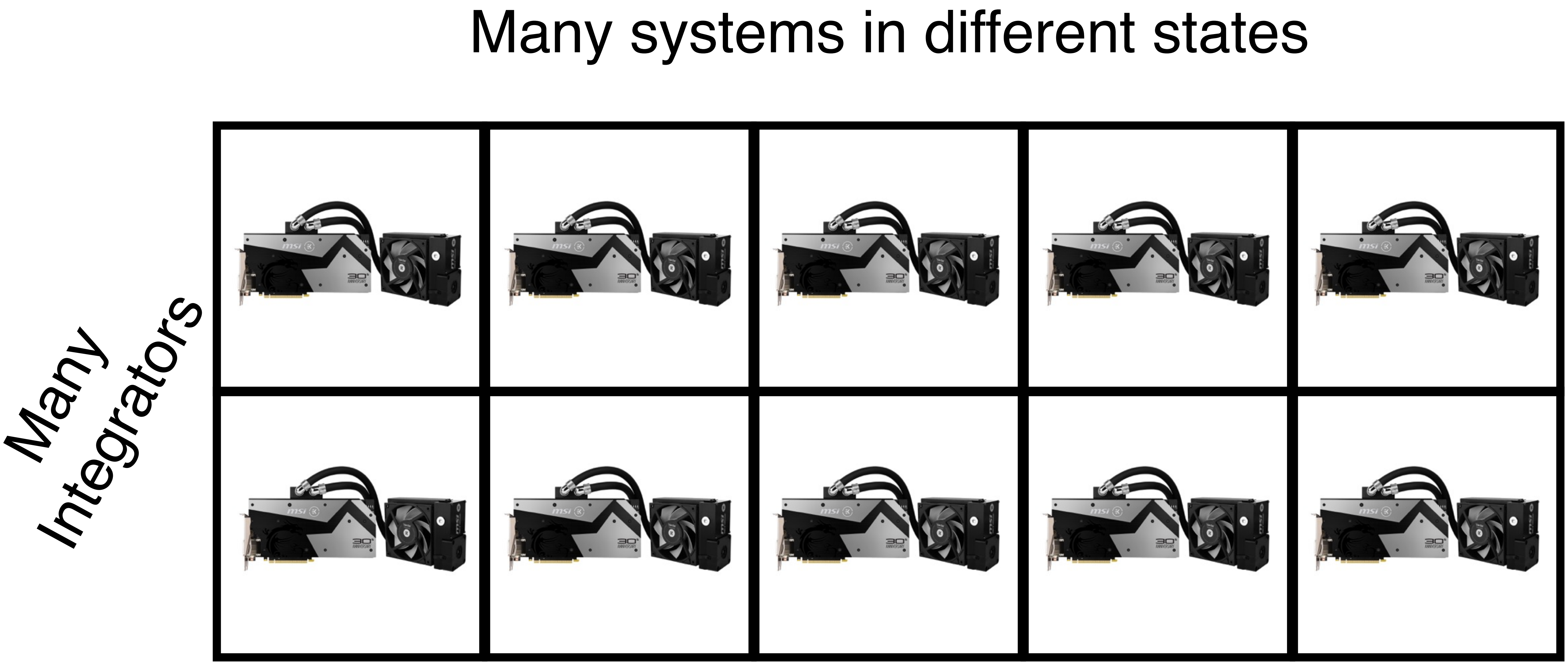
OpenMM at its best

1 System

1 Integrator

1 GPU Context

Our simulations



The general problem to solve

- Keep all Contexts in memory the whole time: not enough memory
- Create and destroy Contexts every time: not enough time.
- We often implement very similar code but optimized for our particular application.

Main classes

- `ThermodynamicState`

  Think of it as an fancy System.

- `SamplerState`

  Hold positions and box vectors.

- `ContextCache`

  Centralized Context creation/destruction/caching.

- `MCMCMove`

  Think of it as a fancy Integrator.

- `CompoundThermodynamicState`

  A class to extend the concept of thermodynamic state beyond temperature and pressure.

ThermodynamicState

- It has two jobs:
  - Easy and safe **editing** of Systems and Contexts.
  - Provides implementation of the concept of **compatible thermodynamic states** which is used throughout the framework to make things efficient.

ThermodynamicState: Basics

- NVT ensemble

```
>>> system = WaterBox(box_edge=10*angstrom).system
>>> state = ThermodynamicState(system=system, temperature=298.0*kelvin)
>>> state.volume
Quantity(value=1.0, unit=nanometer**3)
>>> state.pressure
None
```

- Switch to NPT ensemble

```
>>> state.pressure = 1.0*atmosphere  # Add MonteCarloBarostat
>>> state.volume
None
>>> state.pressure
Quantity(value=1.0, unit=atmosphere)
```

ThermodynamicState: More control on barostat/thermostat

- Convenience methods for reading/editing system.

```
>>> barostat = state.barostat
>>> barostat.setFrequency(100)
>>> barostat.setDefaultPressure(1.2*atmosphere)
>>> state.barostat = barostat
>>> state.pressure
Quantity(value=1.2, unit=atmosphere)
```

ThermodynamicState: More control on barostat/thermostat

- Let ThermodynamicState infer ensemble

```
>>> thermostat = openmm.AndersenThermostat(200.0*kelvin, 1.0/picosecond)
>>> barostat = openmm.MonteCarloBarostat(1.0*atmosphere, 200.0*kelvin)
>>> system.addForce(thermostat)
>>> system.addForce(barostat)

>>> state = ThermodynamicState(system=system)
>>> state.temperature
Quantity(value=200.0, unit=kelvin)
>>> state.pressure
Quantity(value=1.0, unit=atmosphere)
>>> state.volume
None
```

ThermodynamicState: Consistency checks

- You can't set pressure on a non-periodic system

```
>>> system = TolueneVacuum().system
>>> state = ThermodynamicState(system, temperature=300*kelvin,
...                                    pressure=1.0*atmosphere)
Traceback (most recent call last):
...
ThermodynamicsError: Non-periodic systems cannot have a barostat.
```

ThermodynamicState: Consistency checks

- Consistency checks

```
error_messages = {
    MULTIPLE_THERMOSTATS: "System has multiple thermostats.",
    NO_THERMOSTAT: "System does not have a thermostat specifying the temperature.",
    NONE_TEMPERATURE: "Cannot set temperature of the thermodynamic state to None.",
    INCONSISTENT_THERMOSTAT: "System thermostat is inconsistent with thermodynamic state.",
    MULTIPLE_BAROSTATS: "System has multiple barostats.",
    UNSUPPORTED_BAROSTAT: "Found unsupported barostat {} in system.",
    NO_BAROSTAT: "System does not have a barostat specifying the pressure.",
    INCONSISTENT_BAROSTAT: "System barostat is inconsistent with thermodynamic state.",
    BAROSTATED_NONPERIODIC: "Non-periodic systems cannot have a barostat.",
    INCONSISTENT_INTEGRATOR: "Integrator is coupled to a heat bath at a different temperature.",
    INCOMPATIBLE_SAMPLER_STATE: "The sampler state has a different number of particles.",
    INCOMPATIBLE_ENSEMBLE: "Cannot apply to a context in a different thermodynamic ensemble."
}
```

ThermodynamicState: Modifying Contexts and compatible states

- Create a Context

```
>>> state1 = ThermodynamicState(system, temperature=300*kelvin,
...                                    pressure=1.0*bar)
>>> integrator = openmm.LangevinIntegrator(300*kelvin, 5.0/picosecond,
...                                        1.5*femtosecond)
>>> context = state1.create_context(integrator)
```

- Modify the thermodynamic state of the Context

```
>>> state2 = ThermodynamicState(system, temperature=350*kelvin,
...                                    pressure=1.2*bar))
>>> if state2.is_state_compatible(state1):
...     state2.apply_to_context(context)

>>> context.getIntegrator().getTemperature()
Quantity(value=350.0, unit=kelvin)
>>> context.getParameter(state2.barostat.Pressure())
Quantity(value=1.2, unit=bar)
```

ThermodynamicState: Modifying Contexts and compatible states

- Two states x, y are **compatible** if a Context created by x can be converted to y.
- There's also a `is_context_compatible()` but it's much more expensive than `is_state_compatible()`.
- **Gotcha! #1** `apply_to_context()` doesn't check for compatibility before applying, you are responsible for it.
- `ContextCache` takes care of this for you (see later).

Ok, but practically which states are compatible?

- In short: same system, same ensemble.

- More details: We associate a unique hash to each state in such a way that compatible states have the exact same hash.

```
@classmethod
def _standardize_and_hash(cls, system):
    """Standardize the system and return its hash."""
    cls._standardize_system(system)
    system_serialization = openmm.XmlSerializer.serialize(system)
    return system_serialization.__hash__()
```

- **Gotcha! #2** two states result incompatible if you have same exact forces and particles but in different orders.

Ensure correct equilibrium distribution

- Temperature is trickier because it can be controlled by either a thermostat force or an integrator.

```
def create_context(self, integrator, platform=None):
    """Create a context in this ThermodynamicState."""
    # Check if integrator exposes getter/setter for temperature.
    is_thermostated = self._is_integrator_thermostated(integrator)

    # Include a thermostat if integrator is not thermostated.
    system = self.get_system(remove_thermostat=is_thermostated)

    # Create context.
    if platform is None:
        return openmm.Context(system, integrator)
    else:
        return openmm.Context(system, integrator, platform)
```

Ensure correct equilibrium distribution

- **Gotcha! #3** your integrator have to expose `get/setTemperature` or `get/set_temperature` methods to be considered thermostated.

- **Gotcha! #4** the system property returns a system with a thermostat, use `get_system` to remove it

```
>>> state.system  # Has thermostat and barostat (if in NPT).
>>> state.get_system(remove_thermostat=True, remove_barostat=True)
```

Small detour on thermostated integrators

- Copied integrators loose extra methods.

```
>>> from openmmtools.integrators import GeodesicBAOABIntegrator
>>> integrator = GeodesicBAOABIntegrator(temperature=300.0*kelvin)
>>> integrator.getTemperature()
Quantity(value=300.0, unit=kelvin)

# Copied integrator looses getter/setter.
>>> copied_integrator = copy.deepcopy(integrator)
>>> copied_integrator.getTemperature()
Traceback (most recent call last):
...
AttributeError: type object 'object' has no attribute '__getattr__'

# And Context integrators have the same problem.
>>> context = state.create_context(integrator)
>>> context.getIntegrator().getTemperature()
Traceback (most recent call last):
...
AttributeError: type object 'object' has no attribute '__getattr__'
```

Small detour on thermostated integrators

- Integrators inheriting from `utils.RestorableOpenMMObject` (most of them) can be restored.

```
>>> from openmmtools.utils import RestorableOpenMMObject
>>> RestorableOpenMMObject.restore_interface(copied_integrator)
>>> copied_integrator.getTemperature()
Quantity(value=300.0, unit=kelvin)
```

- `ThermodynamicState` always calls that method before checking the temperature.

- Current implementation raises warning if your integrator defines a global variable `kT` but no getter.

Question: What to do when propagation is done by multiple integrators?

```
>>> compound_integrator = CompoundIntegrator()
>>> compound_integrator.addIntegrator(VerletIntegrator(1*femtosecond))
>>> compound_integrator.addIntegrator(GHMCIntegrator(300*kelvin))

>>> context = state.create_context(compound_integrator)  # ?
```

Performance note: memory management and copying

- Internal state of a `ThermodynamicState` object:
  - `_standard_system`

    Single instance shared among all compatible states
  - `_temperature`
  - `_pressure`
- **Copy/deepcopy is quick.** When you copying a `ThermodynamicSystem` you are not copying any `System`.
- **Modifying things other than thermodynamic variables is slow.** This involves a system copy and serialization. However, it's easy to extend the set of thermodynamic variables.

ContextCache

- Builds on the concept of compatibility to decide whether to create a new Context or to modify an existing one.

```
>>> integrator = GeodesicBAOABIntegrator()
>>> context = state1.create_context(integrator)
>>> if state2.is_state_compatible(state1):
...     state2.apply_to_context(context)
... else:
...     integrator = GeodesicBAOABIntegrator()
...     context = state2.create(integrator)
```

```
>>> cache = openmmtools.cache.ContextCache()
>>> integrator = GeodesicBAOABIntegrator()
>>> context, context_integrator = cache.get_context(state1, integrator)
>>> context, context_integrator = cache.get_context(state2, integrator)
```

ContextCache

```
>>> cache = openmmtools.cache.ContextCache()
>>> integrator = GeodesicBAOABIntegrator()
>>> context, context_integrator = cache.get_context(state1, integrator)
>>> context, context_integrator = cache.get_context(state2, integrator)
```

- In general, `context_integrator == context.getIntegrator() != integrator`.
- `context_integrator` has already gone through `RestorableIntegrator.restore_interface()`.
- When you don't care about the integrator (e.g. compute point energies).

```
>>> context, context_integrator = cache.get_context(state)
```

ContextCache configuration

- Capacity and time to live.

```
>>> platform = openmm.Platform.getPlatformByName('CPU')
>>> cache = ContextCache(capacity=5, time_to_live=2, platform=platform)

>>> context, integrator = cache.get_context(nvt_state)
>>> context, integrator = cache.get_context(npt_state)  # First access.
>>> len(cache)
2
>>> cache.get_context(npt_state)  # Second access deletes NVT state.
>>> len(cache)
1
```

- There is also a separate LRUCache in same module in case you need it.

```
>>> cache = openmmtools.cache.LRUCache(capacity=None, time_to_live=100)
>>> cache['key1'] = 1
```

MCMCMove framework

- How does it look to use MCMC framework to run simulations?

```
>>> alanine = openmmtools.testsystems.AlanineDipeptideImplicit()
>>> thermodynamic_state = ThermodynamicState(system=alanine.system,
...                                          temperature=300*kelvin)
>>> sampler_state = SamplerState(positions=alanine.positions)
>>> mcmc_move = GHMCMove(timestep=1.5*femtosecond, n_steps=500)

>>> for i in range(n_iterations):
...     mcmc_move.apply(thermodynamic_state, sampler_state)
```

- Combining MCMC blocks into a protocol.

```
>>> SequenceMove(move_list=[
...     MCDisplacementMove(atom_subset=ligand_atoms),
...     MCRotationMove(atom_subset=ligand_atoms),
...     LangevinDynamicsMove(timestep=2.0*femtoseconds, n_steps=500,
...                          reassign_velocities=True, n_restart_attempts=6)
... ])
```

Converting Integrators into MCMCMoves

- Transform an integrator in an MCMCMove.

```
>>> integrator = GeodesicBAOABIntegrator()
>>> gBAOAB_move = IntegratorMove(integrator, n_steps=100, n_restart_attempts=5)
```

- Restart move if NaN encountered and save serialized `MCMCMove`, `System`, `State`, `Integrator` for debugging.

```
>>> try:
...     gBAOAB_move.apply(thermodynamic_state, sampler_state)
... except mmtools.mcmc.IntegratorMoveError as e:
...     e.serialize_error('path/to/debug/files_prefix')
```

- **Gotcha!** This doesn't work if your integrator changes the thermodynamic state, but you can inherit from `BaseIntegrator`.

Converting Integrators into MCMCMoves

- Create an MCMCMove based on an integrator

```
>>> class MyMove(BaseIntegratorMove):
...     def __init__(self, timestep, n_steps, **kwargs):
...         super(MyMove, self).__init__(n_steps, **kwargs)
...         self.timestep = timestep
...
...     def _get_integrator(self, thermodynamic_state):
...         return MyIntegrator(self.timestep)
...
...     def _before_integration(self, context, thermodynamic_state):
...         context.setVelocitiesToTemperature(thermodynamic_state.temperature)
...
...     def _after_integration(self, context, thermodynamic_state):
...         # Read statistics from context.getIntegrator() parameters.
...         # Update thermodynamic_state from context parameters.
```

ContextCache in MCMCMove

- By default MCMC moves use a global instance of `ContextCache`. It's global, so keep an eye on it.

```
openmmtools.cache.global_context_cache
```

- It's possible to use MCMC moves without a global cache or without any cache, if you prefer.

```
>>> platform = openmm.Platform.getPlatformByName('CPU')
>>> local_cache = ContextCache(platform=platform)
>>> dummy_cache = DummyContextCache()
>>> move = SequenceMove(move_list=[
...     MCDisplacementMove(atom_subset=ligand_atoms, context_cache=local_cache),
...     MCRotationMove(atom_subset=ligand_atoms, context_cache=dummy_cache)
... ])
```

Can we avoid creating a new context for each integrator?

- Similar concept of compatible integrator is already implemented, but very limited so far.

- Future: collapse several integrators often applied to same state in a single `CompoundIntegrator`?

Extending ThermodynamicState

- Our concept of thermodynamic state is not limited on temperature and pressure (alchemical state, protonation state, restraint strength).
- Composition over inheritance: define only the portion of the state you want to change (`AlchemicalState`, `RestraintState`).

Composable state example

- A composable state needs to implement a specific interface.

```
class AlchemicalState(object):

    def __init__(self):
        self.lambda_electrostatics = 1.0
        self.lambda_sterics = 1.0

    def apply_to_system(self, system):
        """Set lambda parameters in the system."""

    def check_system_consistency(self, system):
        """Raise AlchemicalStateError if system has different lambda parameters."""

    def apply_to_context(self, context):
        """Set lambda parameters in the context."""

    def _standardize_system(cls, system):
        """Set all lambda parameters of the system to 1.0"""
```

Composable state example

- Once you have, it you can combine it to a `ThermodynamicState`.

```
>>> thermodynamic_state = ThermodynamicState(system, temperature=300*kelvin)
>>> alchemical_state = AlchemicalState(lambda_sterics=0.5)
>>> restraint_state = RestraintState(lambda_restraint=1.0)
>>> compound_state = CompoundThermodynamicState(
...     thermodynamic_state,
...     composable_states=[alchemical_state, restraint_state]
... )
```

- This behave both as a `ThermodynamicState` and your composable state(s).

```
>>> context, integrator = global_context_cache.get_context(compound_state)
>>> compound_state.lambda_electrostatics = 0.0
>>> compound_state.apply_to_context(context)
>>> move.apply(compound_state, sampler_state)
```

Alchemical functions

```
>>> f_sterics = 'step_hm(lambda - 0.5) + 2*lambda * step_hm(0.5 - lambda)'
>>> f_electrostatics = '2*(lambda - 0.5) * step(lambda - 0.5)'

>>> compound_state.set_alchemical_variable('lambda', 1.0)
>>> compound_state.lambda_sterics = AlchemicalFunction(f_sterics)
>>> compound_state.lambda_electrostatics = AlchemicalFunction(f_electrostatics)

>>> for l in [1.0, 0.75, 0.5, 0.25, 0.0]:
...     compound_state.set_alchemical_variable('lambda', l)
...     print(compound_state.lambda_sterics, compound_state.lambda_electrostatics)
1.0 1.0
1.0 0.5
1.0 0.0
0.5 0.0
0.0 0.0
```

MPI module

- Run task on a single node a send result to all.

```
>>> def add(a, b):
...     return a + b
>>> result = mpi.run_single_node(rank=0, task=add, a=3, b=4,
...                              broadcast_result=True)
>>> result
7
```

- Or equivalently

```
>>> @on_single_node(rank=0, broadcast_result=True)
... def add(a, b):
...     return a + b
>>> add(3, 4)
7
```

- Do not broadcast result but place a barrier after task.

```
>>> mpi.run_single_node(rank=0, task=add, a=3, b=4, sync_nodes=True)
```

MPI module

- Distribute task across available nodes.

```
>>> def square(x):
...     return x**2
>>> result = mpi.distribute(square, distributed_args=[1, 2, 3, 4],
...                         send_results_to='all')
>>> result
[1, 4, 9, 16]
```

- Send all results only to a specific node (network optimization).

```
>>> results, args_indices = mpi.distribute(square, distributed_args=[1, 2, 3, 4],
...                                        send_results_to=0)
>>> results  # This is [4, 16] for node 1
[1, 4, 9, 16]
>>> args_indices  # This is [1, 3] for node 1
[0, 1, 2, 3]
```

- Do not broadcast result but place a barrier after task.

```
>>> mpi.distribute(square, distributed_args=[1, 2, 3, 4], sync_nodes=True)
```

Thanks!

John Chodera
Levi Naden
Patrick Grinaway
Gregory Ross
Josh Fass
Bas Rustenburg
Sonya Hanson
Steven Albanese
Mehtap Isik
Chaya Stern
Rafal Wiewiora