mpilock

Release 1.0.0

Robin De Schepper

Apr 10, 2021

CONTENTS:

CHAPTER

ONE

ABOUT

`mpilock` offers a *mpilock.WindowController* class with a high-level API for parallel access to resources. The *mpilock.WindowController* can be used to perform *read()*, *write()* or *single_write()*.

Read operations happen in parallel while write operations will lock the resource and prevent any new read or write operations and will wait for all existing read operations to finish. After the write operation completes the lock is released and other operations can resume.

The *mpilock.WindowController* does not contain any logic to control the resources, it only locks and synchronizes the MPI processes. Once the operation permission is obtained it's up to the user to perform the reading/writing to the resources.

The *sync()* function is a factory function for *WindowControllers* and can simplify creation of *WindowControllers*.

CHAPTER

TWO

EXAMPLE USAGE

```
from mpilock import sync
from h5py import File

# Create a default WindowController on `COMM_WORLD` with the master on rank 0
ctrl = sync()

# Fencing is the preferred idiom to fence anyone that isn't writing out of
# the writer's code block, and afterwards share a resource
with ctrl.single_write() as fence:
  # Makes anyone without access long jump to the end of the with statement
  fence.guard()
  resource = h5py.File("hello.world", "w")
  # Put a resource to be collected by other processes
  fence.share(resource)
resource = fence.collect()

try:
  # Acquire a parallel read lock, guarantees noone writes while you're reading.
  with ctrl.read():
    data = resource["/my_data"][()]
  # Acquire a write lock, will block all reading and writing.
  with ctrl.write():
    resource.create_dataset(ctrl.rank, data=data)
finally:
  with ctrl.single_write() as fence:
    fence.guard()
    resource.close()

 # The window controller itself needs to be closed as well (is done atexit)
 ctrl.close()
```

CHAPTER

THREE

REFERENCE

class mpilock.**Fence**(*master*, *access*, *comm*)

Bases: object

Can be used to fence off pieces of code from processes that shouldn't access it. Additionally it can be used to share a resource to all processes that was created within the fenced off code block using *Fence.share()* and *Fence.collect()*.

collect()

Collect the object that was put to share within the fenced off code block.

Returns Shared object

Return type any

guard()

Kicks out all MPI processes that do not have access to the fenced off code block. Works only within a `with` statement or a `try` statement that catches *FencedSignal* exceptions.

share(*obj*)

Put an object to share with all other MPI processes from within a fenced off code block.

exception mpilock.**FencedSignal**

Bases: Exception

class mpilock.**WindowController**(*comm=None*, *master=0*)

Bases: object

The `WindowController` manages the state of the MPI windows underlying the lock functionality. Instances can be created using the *sync()* factory function.

The controller can create read and write locks during which your MPI processes are aware of each other's operations and a write lock will never be granted if other read or write operations are ongoing, while read locks may be granted while other read operations are ongoing, but not if any write locks are acquired or being requested.

close()

Close the `WindowController` and its underlying MPI Windows.

property closed

Is this `WindowController` in a closed state? If so, further locks can not be requested.

property master

Return the MPI rank of the master process.

property rank

Return the MPI rank of this process.

read()

Acquire a read lock. Read locks can be granted while other read locks are held, but will not start as long as write locks are held or being requested (write operations have priority over read operations).

The preferred idiom for read locks is as follows:

```
controller = sync()
with controller.read():
    # Perform reading operation
    pass
```

Returns A read lock

single_write(*handle=None*, *rank=None*)

Perform a collective operation where only 1 node writes to the resource and the other processes wait for this operation to complete.

Python does not support any long jump patterns so the preferred idiom for collective write locks is the fencing pattern:

```
controller = sync()
with controller.single_write() as fence:
    # Kick out any processes that don't have to write
    fence.guard()
    # Perform writing operation on just 1 process
    pass
# All kicked out processes resume code together outside of the with block.
```

Returns A fenced write lock.

write()

Acquire a write lock. Will wait for all active read locks to be released and prevent any new read locks from being aqcuired.

The preferred idiom for write locks is as follows:

```
controller = sync()
with controller.write():
    # Perform writing operation
    pass
```

Keep in mind that if you run this code on multiple processes at the same time that they will write one by one, but they will still all write eventually. If only one of the nodes needs to perform the writing operation see *single_write()*

Returns An unfenced write lock

mpilock.**sync**(*comm=None*, *master=0*)

Create a *WindowController* that synchronizes read write operations across all MPI processes in the communicator.

Parameters

- **comm** (*int*) – MPI communicator
- **master** – Rank of the master of the communicator, will be picked whenever something needs to be organized or decided by a single node in the communicator.

Returns A controller

Return type *WindowController*

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX