mpiplus Documentation

mpiplus

Apr 27, 2023

CONTENTS

A GPU-accelerated Python framework for exploring algorithms for alchemical free energy calculations

Ported from the *mpi* module in Yank.

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Installing via *conda*

1.1.2 Development Build

The development build of *mpiplus* is available on `github`

```
$ git clone git@github.com:choderalab/mpiplus.git
$ cd mpiplus
$ python setup.py install
```

Warning: Development builds may be unstable and are generally subjected to less testing than releases. Use at your own risk!

1.2 mpi

1.2.1 mpiplus

MPI

Utilities to run on MPI.

Provide functions and decorators that simplify running the same code on multiple nodes. One benefit is that serial and parallel code is exactly the same.

Global variables

disable_mpi
[bool] Set this to True to force running serially.

Routines

get_mpicomm()
Automatically detect and configure MPI execution and return an MPI communicator.

run_single_node()
Run a task on a single node.

on_single_node()
Decorator version of *run_single_node()*.

distribute()
Map a task on a sequence of arguments on all the nodes.

delay_termination()
A context manager to delay the response to termination signals.

delayed_termination()
A decorator version of *delay_termination()*.

mpiplus.mpiplus.**delay_termination**()

Context manager to delay handling of termination signals.

This allows to avoid interrupting tasks such as writing to the file system, which could result in the corruption of the file.

mpiplus.mpiplus.**delayed_termination**(*func*)

Decorator that runs the function with *delay_termination()*.

mpiplus.mpiplus.**distribute**(*task*, *distributed_args*, **other_args*, *send_results_to='all'*, *propagate_exceptions_to='all'*, *sync_nodes=False*, *group_size=None*, ***kwargs*)

Map the task on a sequence of arguments to be executed on different nodes.

If MPI is not activated, this simply runs serially on this node. The algorithm guarantees that each node will be assigned to the same job_id (i.e. the index of the argument in `distributed_args`) every time.

task
[callable] The task to be distributed among nodes. The task will be called as `task(distributed_args[job_id], *other_args, **kwargs)`, so the parameter to be distributed must the the first one.

distributed_args
[iterable] The sequence of the parameters to distribute among nodes.

send_results_to
[int, 'all', or None, optional, default='all'] If the string 'all', the result will be sent to all nodes. If an int, the result will be send only to the node with rank `send_results_to`. If None no result will be sent to any other node (This is useful if the distributed function is a routine). The return value of distribute depends on the value of this parameter.

propagate_exceptions_to
['all', 'group', or None, optional] When one of the processes raise an exception during the task execution, this controls which other processes raise it (default is 'all'). This can be 'group' or None only if `send_results_to` is None.

sync_nodes
[bool, optional] If True, the nodes will be synchronized at the end of the execution (i.e. the task will be blocking) even if the result is not shared (default is False).

group_size
[None, int or list of int, optional, default is None] If not None, the `distributed_args` are distributed among groups of nodes that are isolated from each other. This is particularly useful if `task` also calls *distribute()*, since normally that would result in unexpected behavior.

If an integer, the nodes are split into equal groups of `group_size` nodes. If `n_nodes % group_size != 0`, the first jobs are allocated more nodes than the latest. If a list of integers, the nodes are split in possibly unequal groups (see example below).

other_args
Other parameters to pass to task beside the assigned distributed parameters.

kwargs
Keyword arguments to pass to task beside the assigned distributed parameters.

all_results
[list] All the return values for all the arguments if the results where sent to the node, or only the return values of the arguments processed by this node otherwise.

arg_indices
[list of int, optional] This is returned as part of a tuple `(all_results, job_indices)` only if `send_results_to` is set to an int or None. In this case `all_results[i]` is the return value of `task(all_args[arg_indices[i]])`.

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

When send_results_to is not set to *all*, the return value include also the indices of the arguments associated to the result.

```
>>> distribute(square, [1, 2, 3, 4], send_results_to=0)
([1, 4, 9, 16], [0, 1, 2, 3])
```

Divide the nodes in two groups of 2. The task, in turn, can distribute another task among the nodes in its own group.

```
>>> def supertask(list_of_bases):
...     return distribute(square, list_of_bases, send_results_to='all')
>>> list_of_supertask_args = [[1, 2, 3], [4], [5, 6]]
>>> distribute(supertask, distributed_args=list_of_supertask_args,
...            send_results_to='all', group_size=2)
[[1, 4, 9], [16], [25, 36]]
```

mpiplus.mpiplus.**get_mpicomm**()

Retrieve the MPI communicator for this execution.

The function automatically detects if the program runs on MPI by checking specific environment variables set by various MPI implementations. On first execution, it modifies sys.excepthook and register a handler for SIGINT, SIGTERM, SIGABRT to call MPI's `Abort()` to correctly terminate all processes.

mpicomm
[mpi4py communicator or None] The communicator for this node, None if the program doesn't run with MPI.

mpiplus.mpiplus.**on_single_node**(*rank*, *broadcast_result=False*, *sync_nodes=False*)

A decorator version of run_single_node.

Decorates a function to be always executed with *run_single_node()*.

rank
[int] The rank of the MPI communicator that must execute the task.

broadcast_result
[bool, optional] If True the result is broadcasted to all nodes. If False, only the node executing the function will receive its return value, and all other nodes will receive None (default is False).

sync_nodes
[bool, optional] If True, the nodes will be synchronized at the end of the execution (i.e. the task will be blocking) even if the result is not broadcasted (default is False).

run_single_node

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

mpiplus.mpiplus.**run_single_node**(*rank*, *task*, **args*, ***kwargs*)

Run task on a single node.

If MPI is not activated, this simply runs locally.

task
[callable] The task to run on node rank.

rank
[int] The rank of the MPI communicator that must execute the task.

broadcast_result
[bool, optional] If True, the result is broadcasted to all nodes. If False, only the node executing the task will receive the return value of the task, and all other nodes will receive None (default is False).

sync_nodes
[bool, optional] If True, the nodes will be synchronized at the end of the execution (i.e. the task will be blocking) even if the result is not broadcasted (default is False).

args
The ordered arguments to pass to task.

kwargs
The keyword arguments to pass to task.

result
The return value of the task. This will be None on all nodes that is not the rank unless `broadcast_result` is set to True.

```
>>> def add(a, b):
...     return a + b
>>> # Run 3+4 on node 0.
>>> run_single_node(0, task=add, a=3, b=4, broadcast_result=True)
7
```

CHAPTER

TWO

LICENSE

mpiplus is licensed under the MIT License. See the *LICENSE* file distributed with mpiplus for more details.

PYTHON MODULE INDEX

m

INDEX