Dask-mpi Documentation

Release 2021.11.0+1.gd0cba37.dirty

['Dask-MPI Development Team']

Nov 12, 2021

GETTING STARTED

Easily deploy Dask using MPI

The Dask-MPI project makes it easy to deploy Dask from within an existing MPI environment, such as one created with the common MPI command-line launchers `mpirun` or `mpiexec`. Such environments are commonly found in high performance supercomputers, academic research institutions, and other clusters where MPI has already been installed.

Dask-MPI provides two convenient interfaces to launch Dask, either from within a batch script or directly from the command-line.

CHAPTER

ONE

BATCH SCRIPT EXAMPLE

You can turn your batch Python script into an MPI executable with the `dask_mpi.initialize` function.

```
from dask_mpi import initialize
initialize()

from dask.distributed import Client
client = Client()  # Connect this local process to remote workers
```

This makes your Python script launchable directly with `mpirun` or `mpiexec`.

```
mpirun -np 4 python my_client_script.py
```

This deploys the Dask scheduler and workers as well as the user's Client process within a single cohesive MPI computation.

CHAPTER

TWO

COMMAND LINE EXAMPLE

Alternatively you can launch a Dask cluster directly from the command-line using the `dask-mpi` command and specifying a scheduler file where Dask can write connection information.

```
mpirun -np 4 dask-mpi --scheduler-file ~/dask-scheduler.json
```

You can then access this cluster either from a separate batch script or from an interactive session (such as a Jupyter Notebook) by referencing the same scheduler file that `dask-mpi` created.

```
from dask.distributed import Client
client = Client(scheduler_file='~/dask-scheduler.json')
```

CHAPTER

THREE

USE JOB QUEUING SYSTEM DIRECTLY

You can also use Dask Jobqueue to deploy Dask directly on a job queuing system like SLURM, SGE, PBS, LSF, Torque, or others. This can be especially nice when you want to dynamically scale your cluster during your computation, or for interactive use.

3.1 Installing

You can install Dask-MPI with `pip`, `conda`, or by installing from source.

3.1.1 Pip

Pip can be used to install both Dask-MPI and its dependencies (e.g. dask, distributed, NumPy, Pandas, etc.) that are necessary for different workloads.:

```
pip install dask_mpi --upgrade   # Install everything from last released version
```

3.1.2 Conda

To install the latest version of Dask-MPI from the conda-forge repository using conda:

```
conda install dask-mpi -c conda-forge
```

3.1.3 Install from Source

To install Dask-MPI from source, clone the repository from github:

```
git clone https://github.com/dask/dask-mpi.git
cd dask-mpi
pip install .
```

You can also install directly from git main branch:

```
pip install git+https://github.com/dask/dask-mpi
```

3.1.4 Test

Test Dask-MPI with `pytest`:

```
git clone https://github.com/dask/dask-mpi.git
cd dask-mpi
pytest dask_mpi
```

3.2 Dask-MPI with Batch Jobs

Dask, with Dask Distributed, is an incredibly powerful engine behind interactive sessions (see *Dask-MPI with Interactive Jobs*). However, there are many scenarios where your work is pre-defined and you do not need an interactive session to execute your tasks. In these cases, running in *batch-mode* is best.

Dask-MPI makes running in batch-mode in an MPI environment easy by providing an API to the same functionality created for the `dask-mpi` *Command-Line Interface (CLI)*. However, in batch mode, you need the script running your Dask Client to run in the same environment in which your Dask cluster is constructed, and you want your Dask cluster to shut down after your Client script has executed.

To make this functionality possible, Dask-MPI provides the `initialize()` method as part of its *Application Program Interface (API)*. The `initialize()` function, when run from within an MPI environment (i.e., created by the use of `mpirun` or `mpiexec`), launches the Dask Scheduler on MPI rank 0 and the Dask Workers on MPI ranks 2 and above. On MPI rank 1, the `initialize()` function "passes through" to the Client script, running the Dask-based Client code the user wishes to execute.

For example, if you have a Dask-based script named `myscript.py`, you would be able to run this script in parallel, using Dask, with the following command.

```
mpirun -np 4 python myscript.py
```

This will run the Dask Scheduler on MPI rank 0, the user's Client code on MPI rank 1, and 2 workers on MPI rank 2 and MPI rank 3. To make this work, the `myscript.py` script must have (presumably near the top of the script) the following code in it.

```python
from dask_mpi import initialize
initialize()

from distributed import Client
client = Client()
```

The Dask Client will automatically detect the location of the Dask Scheduler running on MPI rank 0 and connect to it.

When the Client code is finished executing, the Dask Scheduler and Workers (and, possibly, Nannies) will be terminated.

Tip: Running Batch Jobs with Job Schedulers

It is common in High-Performance Computing (HPC) environments to request the necessary computing resources with a job scheduler, such LSF, PBS, or SLURM. In such environments, is is advised that the `mpirun ... python myscript.py` command be placed in a job submission script such that the resources requested from the job scheduler match the resources used by the `mpirun` command.

For more details on the `initialize()` method, see the *Application Program Interface (API)*.

3.2.1 Connecting to Dashboard

Due to the fact that Dask might be initialized on a node that isn't the login node a simple port forwarding can be insufficient to connect to a dashboard.

To find out which node is the one hosting the dashboard append initialization code with location logging:

```
from dask_mpi import initialize
initialize()

from dask.distributed import Client
from distributed.scheduler import logger
import socket

client = Client()

host = client.run_on_scheduler(socket.gethostname)
port = client.scheduler_info()['services']['dashboard']
login_node_address = "supercomputer.university.edu" # Provide address/domain of login␣
↪node

logger.info(f"ssh -N -L {port}:{host}:{port} {login_node_address}")
```

Then in batch job output file search for the logged line and use in your terminal:

```
ssh -N -L PORT_NUMBER:node03:PORT_NUMBER supercomputer.university.edu
```

The Bokeh Dashboard will be available at `localhost:PORT_NUMBER`.

3.3 Dask-MPI with Interactive Jobs

Dask-MPI can be used to easily launch an entire Dask cluster in an existing MPI environment, and attach a client to that cluster in an interactive session.

In this scenario, you would launch the Dask cluster using the Dask-MPI command-line interface (CLI) `dask-mpi`.

```
mpirun -np 4 dask-mpi --scheduler-file scheduler.json
```

In this example, the above code will use MPI to launch the Dask Scheduler on MPI rank 0 and Dask Workers (or Nannies) on all remaining MPI ranks.

It is advisable, as shown in the previous example, to use the `--scheduler-file` option when using the `dask-mpi` CLI. The `--scheduler-file` option saves the location of the Dask Scheduler to a file that can be referenced later in your interactive session. For example, the following code would create a Dask Client and connect it to the Scheduler using the scheduler JSON file.

```
from distributed import Client
client = Client(scheduler_file='/path/to/scheduler.json')
```

As long as your interactive session has access to the same filesystem where the scheduler JSON file is saved, this procedure will let you run your interactive session easily attach to your separate `dask-mpi` job.

After a Dask cluster has been created, the `dask-mpi` CLI can be used to add more workers to the cluster by using the `--no-scheduler` option.

```
mpirun -n 5 dask-mpi --scheduler-file scheduler.json --no-scheduler
```

In this example (above), 5 more workers will be created and they will be registered with the Scheduler (whose address is in the scheduler JSON file).

Tip: Running with a Job Scheduler

In High-Performance Computing environments, job schedulers, such as LSF, PBS, or SLURM, are commonly used to request the necessary resources needed for an MPI job, such as the number of CPU cores, the total memory needed, and/or the number of nodes over which to spread out the MPI job. In such a case, it is advisable that the user place the `mpirun ... dask-mpi ...` command in a job submission script, with the number of MPI ranks (e.g., `-np 4`) matches the number of cores requested from the job scheduler.

Warning: MPI Jobs and Dask Nannies

It is many times useful to launch your Dask-MPI cluster (using `dask-mpi`) with Dask Nannies (i.e., with the `--worker-class distributed.Nanny` option), rather than strictly with Dask Workers. This is because the Dask Nannies can relaunch a worker when a failure occurs. However, in some MPI environments, Dask Nannies will not be able to work as expected. This is because some installations of MPI may restrict the number of actual running processes from exceeding the number of MPI ranks requested. When using Dask Nannies, the Nanny process is executed and runs in the background after forking a Worker process. Hence, one Worker process will exist for each Nanny process. Some MPI installations will kill any forked process, and you will see many errors arising from the Worker processes being killed. If this happens, disable the use of Nannies with the `--worker-class distributed.Worker` option to `dask-mpi`.

For more details on how to use the `dask-mpi` command, see the *Command-Line Interface (CLI)*.

3.4 Dask-MPI with GPUs

When running *dask-mpi* on GPU enabled systems you will be provided with one or more GPUs per MPI rank.

Today Dask assumes one worker process per GPU with workers tied correctly to each GPU. To help with this the dask-cuda package exists which contains cluster and worker classes which are designed to correctly configure your GPU environment.

```
conda install -c rapidsai -c nvidia -c conda-forge dask-cuda
# or
python -m pip install dask-cuda
```

It is possible to leverage `dask-cuda` with `dask-mpi` by setting the worker class to use `dask_cuda.CUDAWorker`.

```
mpirun -np 4 dask-mpi --worker-class dask_cuda.CUDAWorker
```

```python
from dask_mpi import initialize

initialize(worker_class="dask_cuda.CUDAWorker")
```

Tip: If your cluster is configured so that each rank represents one node you may have multiple GPUs per node. Workers will be created per GPU, not per rank so `CUDAWorker` will create one worker per GPU with names following the pattern {rank}-{gpu_index}. So if you set -np 4 but you have four GPUs per node you will end up with sixteen workers in your cluster.

3.4.1 Additional configuration

You may also want to pass additional configuration options to dask_cuda.CUDAWorker in addition to the ones supported by dask-mpi. It is common to configure things like memory management and network protocols for GPU workers.

You can pass any additional options that are accepted by dask_cuda.CUDAWorker with the worker options paramater.

On the CLI this is expected to be a JSON serialised dictionary of values.

```
mpirun -np 4 dask-mpi --worker-class dask_cuda.CUDAWorker --worker-options '{"rmm_
↪managed_memory": true}'
```

In Python it is just a dictionary.

```
from dask_mpi import initialize

initialize(worker_class="dask_cuda.CUDAWorker", worker_options={"rmm_managed_memory":␣
↪True})
```

Tip: For more information on using GPUs with Dask check out the dask-cuda documentation.

3.5 Command-Line Interface (CLI)

3.5.1 dask-mpi

```
dask-mpi [OPTIONS] [SCHEDULER_ADDRESS]
```

Options

--scheduler-file <scheduler_file>
Filename to JSON encoded scheduler information.

--scheduler-port <scheduler_port>
Specify scheduler port number. Defaults to random.

--interface <interface>
Network interface like ‘eth0’ or ‘ib0’

--protocol <protocol>
Network protocol to use like TCP

--nthreads <nthreads>
Number of threads per worker.

--memory-limit <memory_limit>
Number of bytes before spilling data to disk. This can be an integer (nbytes) float (fraction of total memory) or ‘auto’

`--local-directory` `<local_directory>`
Directory to place worker files

`--scheduler, --no-scheduler`
Whether or not to include a scheduler. Use –no-scheduler to increase an existing dask cluster

`--nanny, --no-nanny`
Start workers in nanny process for management (deprecated use –worker-class instead)

`--worker-class` `<worker_class>`
Class to use when creating workers

`--worker-options` `<worker_options>`
JSON serialised dict of options to pass to workers

`--dashboard-address` `<dashboard_address>`
Address for visual diagnostics dashboard

`--name` `<name>`
Name prefix for each worker, to which dask-mpi appends `-<worker_rank>`.

Arguments

`SCHEDULER_ADDRESS`
Optional argument

3.6 Application Program Interface (API)

`initialize`([interface, nthreads, ...])	Initialize a Dask cluster using mpi4py

3.6.1 dask_mpi.core.initialize

`dask_mpi.core.`**`initialize`**(*interface=None*, *nthreads=1*, *local_directory=''*, *memory_limit='auto'*, *nanny=False*, *dashboard=True*, *dashboard_address=':8787'*, *protocol=None*, *worker_class='distributed.Worker'*, *worker_options=None*, *comm=None*, *exit=True*)

Initialize a Dask cluster using mpi4py

Using mpi4py, MPI rank 0 launches the Scheduler, MPI rank 1 passes through to the client script, and all other MPI ranks launch workers. All MPI ranks other than MPI rank 1 block while their event loops run.

In normal operation these ranks exit once rank 1 ends. If exit=False is set they instead return an bool indicating whether they are the client and should execute more client code, or a worker/scheduler who should not. In this case the user is responsible for the client calling send_close_signal when work is complete, and checking the returned value to choose further actions.

Parameters

interface [str] Network interface like ‘eth0’ or ‘ib0’

nthreads [int] Number of threads per worker

local_directory [str] Directory to place worker files

memory_limit [int, float, or ‘auto’] Number of bytes before spilling data to disk. This can be an integer (nbytes), float (fraction of total memory), or ‘auto’.

nanny [bool] Start workers in nanny process for management (deprecated, use worker_class instead)

dashboard [bool] Enable Bokeh visual diagnostics

dashboard_address [str] Bokeh port for visual diagnostics

protocol [str] Protocol like 'inproc' or 'tcp'

worker_class [str] Class to use when creating workers

worker_options [dict] Options to pass to workers

comm: mpi4py.MPI.Intracomm Optional MPI communicator to use instead of COMM_WORLD

exit: bool Whether to call sys.exit on the workers and schedulers when the event loop completes.

Returns

is_client: bool Only returned if exit=False. Inidcates whether this rank should continue to run client code (True), or if it acts as a scheduler or worker (False).

3.7 How Dask-MPI Works

Dask-MPI works by using the `mpi4py` package and using MPI to selectively run different code on different MPI ranks. Hence, like any other application of the `mpi4py` package, it requires creating the appropriate MPI environment through the running of the `mpirun` or `mpiexec` commands.

```
mpirun -np 8 dask-mpi --no-nanny --scheduler-file ~/scheduler.json
```

or

```
mpirun -np 8 python my_dask_script.py
```

3.7.1 Using the Dask-MPI CLI

By convention, Dask-MPI always launches the Scheduler on MPI rank 0. When using the CLI (`dask-mpi`), Dask-MPI launches the Workers (or Nannies and Workers) on the remaining MPI ranks (MPI ranks 1 and above). On each MPI rank, a `tornado` event loop is started after the Scheduler and Workers are created. These event loops continue until a kill signal is sent to one of the MPI processes, and then the entire Dask cluster (all MPI ranks) is shut down.

When using the `--no-scheduler` option of the Dask-MPI CLI, more workers can be added to an existing Dask cluster. Since these two runs will be in separate `mpirun` or `mpiexec` executions, they will only be tied to each other through the scheduler. If a worker in the new cluster crashes and takes down the entire MPI environment, it will not have anything to do with the first (original) Dask cluster. Similarly, if the first cluster is taken down, the new workers will wait for the Scheduler to reactivate so they can re-connect.

3.7.2 Using the Dask-MPI API

Again, Dask-MPI always launches the Scheduler on MPI rank 0. When using the `initialize()` method, Dask-MPI runs the Client script on MPI rank 1 and launches the Workers on the remaining MPI ranks (MPI ranks 2 and above). The Dask Scheduler and Workers start their `tornado` event loops once they are created on their given MPI ranks, and these event loops run until the Client process (MPI rank 1) sends the termination signal to the Scheduler. Once the Scheduler receives the termination signal, it will shut down the Workers, too.

3.8 Development Guidelines

This repository is part of the Dask projects. General development guidelines including where to ask for help, a layout of repositories, testing practices, and documentation and style standards are available at the Dask developer guidelines in the main documentation.

3.8.1 Install

After setting up an environment as described in the Dask developer guidelines you can clone this repository with git:

```
git clone git@github.com:dask/dask-mpi.git
```

and install it from source:

```
cd dask-mpi
python setup.py install
```

3.8.2 Test

Test using `pytest`:

```
py.test dask_mpi --verbose
```

3.8.3 Build docs

To build docs site after cloning and installing from sources use:

```
cd dask-mpi/docs
make html
```

Output will be placed in `build` directory. Required dependencies for building docs can be found in `dask-mpi/docs/environment.yml`.

3.9 History

This package came out of the Dask Distributed project with help from the Pangeo collaboration. The original code was contained in the `distributed.cli.dask_mpi` module and the original tests were contained in the `distributed.cli.tests.test_dask_mpi` module. The impetus for pulling Dask-MPI out of Dask-Distributed was provided by feedback on the Dask Distributted Issue 2402.

Development history for these original files was preserved.

INDEX