Dask-jobqueue Documentation

Release 0.3.0+6.gf7c565a

['Dask-jobqueue Development Team']

Jul 16, 2018

Contents

Easily deploy Dask on job queuing systems like PBS, Slurm, MOAB, and SGE.

The Dask-jobqueue project makes it easy to deploy Dask on common job queuing systems typically found in high performance supercomputers, academic research institutions, and other clusters. It provides a convenient interface that is accessible from interactive systems like Jupyter notebooks, or batch jobs.

CHAPTER 1

Example

```
from dask_jobqueue import PBSCluster
cluster = PBSCluster()
cluster.scale(10)          # Ask for ten workers

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

# wait for jobs to arrive, depending on the queue, this may take some time

import dask.array as da
x = ...                    # Dask commands now use these distributed resources
```

CHAPTER 2

Adaptivity

Dask jobqueue can also adapt the cluster size dynamically based on current load. This helps to scale up the cluster when necessary but scale it down and save resources when not actively computing.

```
cluster.adapt(minimum=6, maximum=90)  # auto-scale between 6 and 90 workers
```

CHAPTER 3

Configuration

Dask-jobqueue should be configured for your cluster so that it knows how many resources to request of each job and how to break up those resources. You can specify configuration either with keyword arguments when creating a `Cluster` object, or with a configuration file.

3.1 Keyword Arguments

You can pass keywords to the Cluster objects to define how Dask-jobqueue should define a single job:

```
cluster = PBSCluster(
     # Dask-worker specific keywords
     cores=24,              # Number of cores per job
     memory='100GB',        # Amount of memory per job
     processes=6,           # Number of Python processes to cut up each job
     local_directory='$TMPDIR',  # Location to put temporary data if necessary
     # Job scheduler specific keywords
     resource_spec='select=1:ncpus=24:mem=100GB',
     queue='regular',
     project='my-project',
     walltime='02:00:00',
)
```

Note that the `cores` and `memory` keywords above correspond not to your full desired deployment, but rather to the size of a *single job* which should be no larger than the size of a single machine in your cluster. Separately you will specify how many jobs to deploy using the scale method.

```
cluster.scale(12)  # launch 12 workers (2 jobs of 6 workers each) of the␣
↪specification provided above
```

3.2 Configuration Files

Specifying all parameters to the Cluster constructor every time can be error prone, especially when sharing this workflow with new users. Instead, we recommend using a configuration file like the following:

```
# jobqueue.yaml file
jobqueue:
  pbs:
    cores: 24
    memory: 100GB
    processes: 6

    interface: ib0
    local-directory: $TMPDIR

    resource-spec: "select=1:ncpus=24:mem=100GB"
    queue: regular
    project: my-project
    walltime: 00:30:00
```

See *Configuration Examples* for real-world examples.

If you place this in your `~/.config/dask/` directory then Dask-jobqueue will use these values by default. You can then construct a cluster object without keyword arguments and these parameters will be used by default.

```
cluster = PBSCluster()
```

You can still override configuration values with keyword arguments

```
cluster = PBSCluster(processes=12)
```

If you have imported `dask_jobqueue` then a blank `jobqueue.yaml` will be added automatically to `~/.config/dask/jobqueue.yaml`. You should use the section of that configuation file that corresponds to your job scheduler. Above we used PBS, but other job schedulers operate the same way. You should be able to share these with colleagues. If you can convince your IT staff you can also place such a file in `/etc/dask/` and it will affect all people on the cluster automatically.

For more information about configuring Dask, see the Dask configuration documentation

3.2.1 Installing

You can install dask-jobqueue with `pip`, `conda`, or by installing from source.

Pip

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

```
pip install "dask_jobqueue"    # Install everything from last released version
```

Conda

To install the latest version of dask-jobqueue from the conda-forge repository using conda:

```
conda install dask-jobqueue -c conda-forge
```

Install from Source

To install dask-jobqueue from source, clone the repository from github:

```
git clone https://github.com/dask/dask-jobqueue.git
cd dask-jobqueue
python setup.py install
```

or use `pip` locally if you want to install all dependencies as well:

```
pip install -e .
```

You can also install directly from git master branch:

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

Test

Test dask-jobqueue with `py.test`:

```
git clone https://github.com/dask/dask-jobqueue.git
cd dask-jobqueue
py.test dask_jobqueue
```

3.2.2 Configuration Examples

We include configuration files for known supercomputers. Hopefully these help both other users that use those machines and new users who want to see examples for similar clusters.

Additional examples from other cluster welcome here.

Cheyenne

Cheyenne Supercomputer

```
distributed:
  scheduler:
    bandwidth: 1000000000     # GB MB/s estimated worker-worker bandwidth
  worker:
    memory:
      target: 0.90  # Avoid spilling to disk
      spill: False  # Avoid spilling to disk
      pause: 0.80  # fraction at which we pause worker threads
      terminate: 0.95  # fraction at which we terminate the worker
  comm:
    compression: null

jobqueue:
  pbs:
    cores: 36
```

(continues on next page)

(continued from previous page)

```
    memory: 108GB
    processes: 4

    interface: ib0
    local-directory: $TMPDIR

    queue: regular
    project: null  # TODO, change me
    walltime: '00:30:00'

    resource-spec: select=1:ncpus=36:mem=109G
```

NERSC Cori

NERSC Cori Supercomputer

It should be noted that the the following config file assumes you are running the scheduler on a worker node. Currently the login node appears unable to talk to the worker nodes bidirectionally. As such you need to request an interactive node with the following:

```
$ salloc -N 1 -C haswell --qos=interactive -t 04:00:00
```

Then you will run dask jobqueue directly on that interactive node. Note the distributed section that is set up to avoid having dask write to disk. This was due to some weird behavior with the local filesystem.

```
distributed:
  worker:
    memory:
      target: False  # Avoid spilling to disk
      spill: False  # Avoid spilling to disk
      pause: 0.80  # fraction at which we pause worker threads
      terminate: 0.95  # fraction at which we terminate the worker

jobqueue:
    slurm:
        cores: 64
        memory: 128GB
        processes: 4
        queue: debug
        walltime: '00:10:00'
        job-extra: ['-C haswell', '-L project, SCRATCH, cscratch1']
```

ARM Stratus

Department of Energy Atmospheric Radiation Measurement (DOE-ARM) Stratus Supercomputer.

```
jobqueue:
  pbs:
    name: dask-worker
    cores: 36
    memory: 270GB
    processes: 6
    interface: ib0
    local-directory: $localscratch
```

(continues on next page)

(continued from previous page)

```
    queue: high_mem # Can also select batch or gpu_ssd
    project: arm
    walltime: 00:30:00 #Adjust this to job size
    job-extra: ['-W group_list=cades-arm']
```

3.2.3 History

This package came out of the Pangeo collaboration and was copy-pasted from a live repository at this commit. Unfortunately, development history was not preserved.

Original developers from that repository include the following:

- Jim Edwards
- Joe Hamman
- Matthew Rocklin

3.2.4 API

MoabCluster([queue, project, resource_spec, ...])	Launch Dask on a Moab cluster
PBSCluster([queue, project, resource_spec, ...])	Launch Dask on a PBS cluster
SLURMCluster([queue, project, walltime, ...])	Launch Dask on a SLURM cluster
SGECluster([queue, project, resource_spec, ...])	Launch Dask on a SGE cluster

dask_jobqueue.MoabCluster

class dask_jobqueue.**MoabCluster**(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, ***kwargs*)

Launch Dask on a Moab cluster

Parameters

queue [str] Destination queue for each worker job. Passed to *#PBS -q* option.

project [str] Accounting string associated with each worker job. Passed to *#PBS -A* option.

resource_spec [str] Request resources and specify job placement. Passed to *#PBS -l* option.

walltime [str] Walltime for each worker job.

job_extra [list] List of other PBS options, for example -j oe. Each option will be prepended with the #PBS prefix.

name [str] Name of Dask workers.

cores [int] Total number of cores per job

memory: str Total amount of memory per job

processes [int] Number of processes per job

interface [str] Network interface like 'eth0' or 'ib0'.

death_timeout [float] Seconds to wait for a scheduler before closing workers

local_directory [str] Dask worker local directory for file spilling.

extra [str] Additional arguments to pass to *dask-worker*

env_extra [list] Other commands to add to script before launching worker.

kwargs [dict] Additional keyword arguments to pass to *LocalCluster*

Examples

```
>>> import os
>>> from dask_jobqueue import MoabCluster
>>> cluster = MoabCluster(processes=6, threads=1, project='gfdl_m',
                          memory='16G', resource_spec='96G',
                          job_extra=['-d /home/First.Last', '-M none'],
                          local_directory=os.getenv('TMPDIR', '/tmp'))
>>> cluster.start_workers(10)  # submit enough jobs to deploy 10 workers
```

```
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

This also works with adaptive clusters. This automatically launches and kill workers based on load.

```
>>> cluster.adapt()
```

__init__(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, ***kwargs*)

Methods

__init__([queue, project, resource_spec, ...])	
adapt(**kwargs)	Turn on adaptivity
job_file()	Write job submission script to temporary file
job_script()	Construct a job submission script
scale(n)	Scale cluster to n workers
scale_down(workers)	Close the workers with the given addresses
scale_up(n, **kwargs)	Brings total worker count up to n
start_workers([n])	Start workers and point them to our local scheduler
stop_jobs(jobs)	Stop a list of jobs
stop_workers(workers)	Stop a list of workers

Attributes

cancel_command	
finished_jobs	Jobs that have finished
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currenly active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_name	
submit_command	
worker_threads	

dask_jobqueue.PBSCluster

class dask_jobqueue.**PBSCluster**(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, ***kwargs*)

Launch Dask on a PBS cluster

Parameters

queue [str] Destination queue for each worker job. Passed to *#PBS -q* option.

project [str] Accounting string associated with each worker job. Passed to *#PBS -A* option.

resource_spec [str] Request resources and specify job placement. Passed to *#PBS -l* option.

walltime [str] Walltime for each worker job.

job_extra [list] List of other PBS options, for example -j oe. Each option will be prepended with the #PBS prefix.

name [str] Name of Dask workers.

cores [int] Total number of cores per job

memory: str Total amount of memory per job

processes [int] Number of processes per job

interface [str] Network interface like 'eth0' or 'ib0'.

death_timeout [float] Seconds to wait for a scheduler before closing workers

local_directory [str] Dask worker local directory for file spilling.

extra [str] Additional arguments to pass to *dask-worker*

env_extra [list] Other commands to add to script before launching worker.

kwargs [dict] Additional keyword arguments to pass to *LocalCluster*

Examples

```
>>> from dask_jobqueue import PBSCluster
>>> cluster = PBSCluster(queue='regular', project='DaskOnPBS')
>>> cluster.scale(10)  # this may take a few seconds to launch
```

```
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

This also works with adaptive clusters. This automatically launches and kill workers based on load.

```
>>> cluster.adapt()
```

It is a good practice to define local_directory to your PBS system scratch directory, and you should specify resource_spec according to the processes and threads asked:

```
>>> cluster = PBSCluster(queue='regular', project='DaskOnPBS',
...                      local_directory=os.getenv('TMPDIR', '/tmp'),
...                      threads=4, processes=6, memory='16GB',
...                      resource_spec='select=1:ncpus=24:mem=100GB')
```

__init__(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, ***kwargs*)

Methods

`__init__`([queue, project, resource_spec, ...])	
`adapt`(**kwargs)	Turn on adaptivity
`job_file`()	Write job submission script to temporary file
`job_script`()	Construct a job submission script
`scale`(n)	Scale cluster to n workers
`scale_down`(workers)	Close the workers with the given addresses
`scale_up`(n, **kwargs)	Brings total worker count up to `n`
`start_workers`([n])	Start workers and point them to our local scheduler
`stop_jobs`(jobs)	Stop a list of jobs
`stop_workers`(workers)	Stop a list of workers

Attributes

`cancel_command`	
`finished_jobs`	Jobs that have finished
`pending_jobs`	Jobs pending in the queue
`running_jobs`	Jobs with currenly active workers
`scheduler`	The scheduler of this cluster
`scheduler_address`	
`scheduler_name`	
`submit_command`	
`worker_threads`	

dask_jobqueue.SLURMCluster

class dask_jobqueue.**SLURMCluster**(*queue=None*, *project=None*, *walltime=None*, *job_cpu=None*, *job_mem=None*, *job_extra=None*, ***kwargs*)

Launch Dask on a SLURM cluster

Parameters

queue [str] Destination queue for each worker job. Passed to *#SBATCH -p* option.

project [str] Accounting string associated with each worker job. Passed to *#SBATCH -A* option.

walltime [str] Walltime for each worker job.

job_cpu [int] Number of cpu to book in SLURM, if None, defaults to worker threads * processes

job_mem [str] Amount of memory to request in SLURM. If None, defaults to worker processes * memory

job_extra [list] List of other Slurm options, for example -j oe. Each option will be prepended with the #SBATCH prefix.

name [str] Name of Dask workers.

cores [int] Total number of cores per job

memory: str Total amount of memory per job

processes [int] Number of processes per job

interface [str] Network interface like 'eth0' or 'ib0'.

death_timeout [float] Seconds to wait for a scheduler before closing workers

local_directory [str] Dask worker local directory for file spilling.

extra [str] Additional arguments to pass to *dask-worker*

env_extra [list] Other commands to add to script before launching worker.

kwargs [dict] Additional keyword arguments to pass to *LocalCluster*

Examples

```
>>> from dask_jobqueue import SLURMCluster
>>> cluster = SLURMCluster(processes=6, threads=4, memory="16GB",
                           env_extra=['export LANG="en_US.utf8"',
                                      'export LANGUAGE="en_US.utf8"',
                                      'export LC_ALL="en_US.utf8"'])
>>> cluster.scale(10)  # this may take a few seconds to launch
```

```
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

This also works with adaptive clusters. This automatically launches and kill workers based on load.

```
>>> cluster.adapt()
```

__init__(*queue=None*, *project=None*, *walltime=None*, *job_cpu=None*, *job_mem=None*, *job_extra=None*, ***kwargs*)

Methods

__init__([queue, project, walltime, ...])	
adapt(**kwargs)	Turn on adaptivity
job_file()	Write job submission script to temporary file
job_script()	Construct a job submission script
scale(n)	Scale cluster to n workers
scale_down(workers)	Close the workers with the given addresses
scale_up(n, **kwargs)	Brings total worker count up to n
start_workers([n])	Start workers and point them to our local scheduler
stop_jobs(jobs)	Stop a list of jobs
stop_workers(workers)	Stop a list of workers

Attributes

cancel_command	
finished_jobs	Jobs that have finished
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currenly active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_name	

Continued on next page

Table 7 – continued from previous page

submit_command
worker_threads

dask_jobqueue.SGECluster

class dask_jobqueue.**SGECluster**(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, ***kwargs*)

Launch Dask on a SGE cluster

Parameters

queue [str] Destination queue for each worker job. Passed to *#$ -q* option.

project [str] Accounting string associated with each worker job. Passed to *#$ -A* option.

resource_spec [str] Request resources and specify job placement. Passed to *#$ -l* option.

walltime [str] Walltime for each worker job.

name [str] Name of Dask workers.

cores [int] Total number of cores per job

memory: str Total amount of memory per job

processes [int] Number of processes per job

interface [str] Network interface like 'eth0' or 'ib0'.

death_timeout [float] Seconds to wait for a scheduler before closing workers

local_directory [str] Dask worker local directory for file spilling.

extra [str] Additional arguments to pass to *dask-worker*

env_extra [list] Other commands to add to script before launching worker.

kwargs [dict] Additional keyword arguments to pass to *LocalCluster*

Examples

```
>>> from dask_jobqueue import SGECluster
>>> cluster = SGECluster(queue='regular')
>>> cluster.scale(10)  # this may take a few seconds to launch
```

```
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

This also works with adaptive clusters. This automatically launches and kill workers based on load.

```
>>> cluster.adapt()
```

__init__(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, ***kwargs*)

Methods

`__init__`([queue, project, resource_spec, ...])	
`adapt`(**kwargs)	Turn on adaptivity
`job_file`()	Write job submission script to temporary file
`job_script`()	Construct a job submission script
`scale`(n)	Scale cluster to n workers
`scale_down`(workers)	Close the workers with the given addresses
`scale_up`(n, **kwargs)	Brings total worker count up to `n`
`start_workers`([n])	Start workers and point them to our local scheduler
`stop_jobs`(jobs)	Stop a list of jobs
`stop_workers`(workers)	Stop a list of workers

Attributes

`cancel_command`	
`finished_jobs`	Jobs that have finished
`pending_jobs`	Jobs pending in the queue
`running_jobs`	Jobs with currenly active workers
`scheduler`	The scheduler of this cluster
`scheduler_address`	
`scheduler_name`	
`submit_command`	
`worker_threads`	

CHAPTER 4

How this works

Dask-jobqueue creates a Dask Scheduler in the Python process where the cluster object is instantiated:

```
cluster = PBSCluster(  # <-- scheduler started here
     cores=24,
     memory='100GB',
     processes=6,
     local_directory='$TMPDIR',
     resource_spec='select=1:ncpus=24:mem=100GB',
     queue='regular',
     project='my-project',
     walltime='02:00:00',
)
```

You then ask for more workers using the `scale` command:

```
cluster.scale(36)
```

The cluster generates a traditional job script and submits that an appropriate number of times to the job queue. You can see the job script that it will generate as follows:

```
>>> print(cluster.job_script())
```

```
#!/bin/bash

#PBS -N dask-worker
#PBS -q regular
#PBS -A P48500028
#PBS -l select=1:ncpus=24:mem=100G
#PBS -l walltime=02:00:00

/home/mrocklin/Software/anaconda/bin/dask-worker tcp://127.0.1.1:43745
--nthreads 4 --nprocs 6 --memory-limit 18.66GB --name dask-worker-3
--death-timeout 60
```

Each of these jobs are sent to the job queue independently and, once that job starts, a dask-worker process will start up and connect back to the scheduler running within this process.

If the job queue is busy then it's possible that the workers will take a while to get through or that not all of them arrive. In practice we find that because dask-jobqueue submits many small jobs rather than a single large one workers are often able to start relatively quickly. This will depend on the state of your cluster's job queue though.

When the cluster object goes away, either because you delete it or because you close your Python program, it will send a signal to the workers to shut down. If for some reason this signal does not get through then workers will kill themselves after 60 seconds of waiting for a non-existent scheduler.

CHAPTER 5

Workers vs Jobs

In dask-distributed, a `Worker` is a Python object and node in a dask `Cluster` that serves two purposes, 1) serve data, and 2) perform computations. `Jobs` are resources submitted to, and managed by, the job queueing system (e.g. PBS, SGE, etc.). In dask-jobqueue, a single `Job` may include one or more `Workers`.

Index

Symbols

M

P

S