Dask-jobqueue Documentation

Release 0.6.2+2.g383c368

['Dask-jobqueue Development Team']

Aug 15, 2019

GETTING STARTED

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

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

ONE

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

TWO

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

THREE

MORE DETAILS

A good entry point to know more about how to use `dask-jobqueue` is *Talks & Tutorials*.

3.1 Installing

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

3.1.1 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 --upgrade   # Install everything from last released version
```

3.1.2 Conda

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

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

3.1.3 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
```

3.1.4 Test

Test dask-jobqueue with `pytest`:

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

3.2 Interactive Use

Dask-jobqueue is most often used for interactive processing using tools like IPython or Jupyter notebooks. This page provides instructions on how to launch an interactive Jupyter notebook server and Dask dashboard on your HPC system.

3.2.1 Using Jupyter

It is convenient to run a Jupyter notebook server on the HPC for use with dask-jobqueue. You may already have a Jupyterhub instance available on your system, which can be used as is. Otherwise, documentation for starting your own Jupyter notebook server is available at Pangeo documentation.

Once Jupyter is installed and configured, using a Jupyter notebook is done by:

- Starting a Jupyter notebook server on the HPC (it is often good practice to run/submit this as a job to an interactive queue, see Pangeo docs for more details).

```
$ jupyter notebook --no-browser --ip=`hostname` --port=8888
```

- Reading the output of the command above to get the ip or hostname of your notebook, and use SSH tunneling on your local machine to access the notebook. This must only be done in the probable case where you don't have direct access to the notebook URL from your computer browser.

```
$ ssh -N -L 8888:x.x.x.x:8888 username@hpc_domain
```

Now you can go to `http://localhost:8888` on your browser to access the notebook server.

3.2.2 Viewing the Dask Dashboard

Whether or not you are using dask-jobqueue in Jupyter, IPython or other tools, at one point you will want to have access to Dask dashboard. Once you've started a cluster and connected a client to it using commands described in *Example*), inspecting `client` object will give you the Dashboard URL, for example `http://172.16.23.102:8787/status`. The Dask Dashboard may be accessible by clicking the link displayed, otherwise, you'll have to use SSH tunneling:

```
# General syntax
$ ssh -fN your-login@scheduler-ip-address -L port-number:localhost:port-number
# As applied to this example:
$ ssh -fN username@172.16.23.102 -L 8787:localhost:8787
```

Now, you can go to `http://localhost:8787` on your browser to view the dashboard. Note that you can do SSH tunneling for both Jupyter and Dashboard in one command.

A good example of using Jupyter along with dask-jobqueue and the Dashboard is available below:

3.2.3 Dask Dashboard with Jupyter

If you are using dask-jobqueue within Jupyter, one user friendly solution to see the Dashboard is to use nbserverproxy. As the dashboard HTTP end point is launched inside the same node as Jupyter, this is a great solution for viewing it without having to do SSH tunneling. You just need to install `nbserverproxy` in the Python environment you use for launching the notebook, and activate it as indicated in the docs:

```
pip install nbserverproxy
jupyter serverextension enable --py nbserverproxy
```

Then, once started, the dashboard will be accessible from your notebook URL by adding the path `/proxy/8787/status`, replacing 8787 by any other port you use or the dashboard is bind to if needed. Sor for example:

- `http://localhost:8888/proxy/8787/status` with the example above
- `http://myjupyterhub.org/user/username/proxy/8787/status` if using JupyterHub

Note that if using Jupyterhub, the service admin should deploy nbserverproxy on the environment used for starting singleuser notebook, but each user may have to activate the nbserverproxy extension.

Finally, you may want to update the dashboard link that is displayed in the notebook, shown from Cluster and Client objects. In order to do this, edit dask config file, either `~/.config/dask/jobqueue.yaml` or `~/.config/dask/distributed.yaml`, and add the following:

```
distributed.dashboard.link: "/proxy/{port}/status" # for user launched notebook
distributed.dashboard.link: "/user/{JUPYTERHUB_USER}/proxy/{port}/status" # for␣
↪jupyterhub launched notebook
```

3.3 Talks & Tutorials

- A 4 hour `dask` and `dask-jobqueue` tutorial was presented in July 2019 by @willirath: Jupyter notebooks, videos: part 1 and part 2.
- A 30 minute presentation by andersy005 at Scipy 2019 that features how `dask-jobqueue` is used on the NCAR HPC cluster: slides and video.

3.4 How this works

3.4.1 Scheduler and jobs

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',
     shebang='#!/usr/bin/env zsh',  # default is bash
     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())
```

```
#!/usr/bin/env zsh

#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/username/path/to/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.

3.4.2 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`.

3.5 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.

3.5.1 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
```

(continues on next page)

(continued from previous page)

```
    shebang='#!/usr/bin/env zsh',    # Interpreter for your batch script (default is␣
↪bash)
    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
```

3.5.2 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
    shebang: "#!/usr/bin/env zsh"

    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 configuration 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

3.6 Configure Dask-Jobqueue

To properly use Dask and Dask-Jobqueue on an HPC system you need to provide a bit of information about that system and how you plan to use it.

You provide this information either as keyword arguments to the constructor:

```
cluster = PBSCluster(cores=36, memory='100GB', queue='regular', ...)
```

Or as part of a configuration file:

```
jobqueue:
  pbs:
    cores: 36
    memory: 100GB
    queue: regular
    ...
```

```
cluster = PBSCluster()
```

For more information on handling configuration files see Dask configuration documentation.

This page explains what these parameters mean and how to find out information about them.

3.6.1 Cores and Memory

These numbers correspond to the size of a single job, which is typically the size of a single node on your cluster. It does not mean the total amount of cores or memory that you want for your full deployment. Recall that dask-jobqueue will launch several jobs in normal operation.

Cores should be provided as an integer, while memory is typically provided as a string, like "100 GB".

```
cores: 36
memory: 100GB
```

Gigabyte vs Gibibyte

It is important to note that Dask makes the difference between power of 2 and power of 10 when specifying memory. This means that: - 1GB = 10^9 bytes - 1GiB = 2^30 bytes

`memory` configuration is interpreted by Dask memory parser, and for most JobQueueCluster implementation translated as a resource requirement for job submission. But most job schedulers (this is the case with PBS and Slurm at least) uses KB or GB, but mean KiB or GiB. Dask jobqueue takes that into account, so you may not find the amount of memory you were expecting when querying your job queuing system. To give an example, with PBSCluster, if you specify '20GB' for the `memory` kwarg, you will end up with a request for 19GB on PBS side. This is because 20GB ~= 18.6GiB, which is rounded up.

This can be avoided by always using 'GiB' in dask-jobqueue configuration.

3.6.2 Processes

By default Dask will run one Python process per job. However, you can optionally choose to cut up that job into multiple processes using the `processes` configuration value. This can be advantageous if your computations are bound by the GIL, but disadvantageous if you plan to communicate a lot between processes. Typically we find that for pure Numpy workloads a low number of processes (like one) is best, while for pure Python workloads a high number of processes (like one process per two cores) is best. If you are unsure then you might want to experiment a bit, or just choose a moderate number, like one process per four cores.

```
cores: 36
memory: 100GB
processes: 9
```

3.6.3 Queue

Many HPC systems have a variety of different queues to which you can submit jobs. These typically have names like "regular", "debug", and "priority". These are set up by your cluster administrators to help direct certain jobs based on their size and urgency.

```
queue: regular
```

If you are unfamiliar with using queues on your system you should leave this blank, or ask your IT administrator.

3.6.4 Project

You may have an allocation on your HPC system that is referenced by a *project*. This is typically a short bit of text that references your group or a particular project. This is typically given to you by your IT administrator when they give you an allocation of hours on the HPC system.

```
project: XYZW-1234
```

If this sounds foreign to you or if you don't use project codes then you should leave this blank, or ask your IT administrator.

3.6.5 Local Storage

When Dask workers run out of memory they typically start writing data to disk. This is often a wise choice on personal computers or analysis clusters, but can be unwise on HPC systems if they lack local storage. When Dask workers try to write excess data to disk on systems that lack local storage this can cause the Dask process to die in unexpected ways.

If your nodes have fast locally attached storage mounted somewhere then you should direct dask-jobqueue to use that location.

```
local-directory: /scratch
```

Sometimes your job scheduler will give this location to you as an environment variable. If so you should include that environment variable, prepended with the `$` sign and it will be expanded appropriately after the jobs start.

```
local-directory: $LOCAL_STORAGE
```

3.6.6 No Local Storage

If your nodes do not have locally attached storage then we recommend that you turn off Dask's policy to write excess data to disk. This must be done in a configuration file and must be separate from the `jobqueue` configuration section (though it is fine to include it in the same file).

```
jobqueue:
  pbs:
    cores: 36
    memory: 100GB
    ...

distributed:
  worker:
    memory:
      target: False    # Avoid spilling to disk
      spill: False     # Avoid spilling to disk
      pause: .80       # Pause worker threads at 80% use
      terminate: 0.95  # Restart workers at 95% use
```

3.6.7 Network Interface

HPC systems often have advanced networking hardware like Infiniband. Dask workers can take use of this network using TCP-over-Infiniband, this can yield improved bandwidth during data transfers. To get this increased speed you often have to specify the network interface of your accelerated hardware. If you have sufficient permissions then you can find a list of all network interfaces using the `ifconfig` UNIX command

```
$ ifconfig
lo          Link encap:Local Loopback                       # Localhost
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
eth0        Link encap:Ethernet  HWaddr XX:XX:XX:XX:XX:XX   # Ethernet
            inet addr:192.168.0.101
            ...
ib0         Link encap:Infiniband                           # Fast InfiniBand
            inet addr:172.42.0.101
```

Note: on some clusters `ifconfig` may need root access. You can use this python code to list all the network interfaces instead:

```
import psutil
psutil.net_if_addrs()
```

Alternatively, your IT administrators will have this information.

3.6.8 Managing Configuration files

By default when dask-jobqueue is first imported it places a file at `~/.config/dask/jobqueue.yaml` with a commented out version of many different job schedulers. You may want to do a few things to clean this up:

1. Remove all of the commented out portions that don't apply to you. For example if you use only PBS, then consider removing the entries under SGE, SLURM, etc..
2. Feel free to rename the file or to include other configuration options in the file for other parts of Dask. The `jobqueue.yaml` filename is not special, nor is it special that each component of Dask has its own configuration file. It is ok to combine or split up configuration files as suits your group.

3. Ask your IT administrator to place a generic file in `/etc/dask` for global use. Dask will look first in `/etc/dask` and then in `~/.config/dask` for any `.yaml` files preferring those in the user's home directory to those in the `/etc/dask`. By providing a global file IT should be able to provide sane settings for everyone on the same system

3.7 Example Deployments

Deploying dask-jobqueue on different clusters requires a bit of customization. Below, we provide a few examples from real deployments in the wild:

Additional examples from other cluster welcome here.

3.7.1 PBS Deployments

```
from dask_jobqueue import PBSCluster

cluster = PBSCluster(queue='regular',
                     project='DaskOnPBS',
                     local_directory='$TMPDIR',
                     cores=24,
                     processes=6,
                     memory='16GB',
                     resource_spec='select=1:ncpus=24:mem=100GB')

cluster = PBSCluster(cores=24,
                     processes=6,
                     shebang='#!/usr/bin/env zsh',
                     memory="6GB",
                     project='P48500028',
                     queue='premium',
                     resource_spec='select=1:ncpus=36:mem=109G',
                     walltime='02:00:00',
                     interface='ib0')
```

Moab Deployments

On systems which use the Moab Workload Manager, a subclass of `PBSCluster` can be used, called `MoabCluster`:

```
import os
from dask_jobqueue import MoabCluster

cluster = MoabCluster(cores=6,
                      processes=6,
                      project='gfdl_m',
                      memory='16G',
                      resource_spec='pmem=96G',
                      job_extra=['-d /home/First.Last', '-M none'],
                      local_directory=os.getenv('TMPDIR', '/tmp'))
```

3.7.2 SGE Deployments

On systems which use SGE as the scheduler, `SGECluster` can be used. Note that Grid Engine has a slightly involved history , so there are a variety of Grid Engine derivatives. `SGECluster` can be used for any derivative of Grid Engine, for example: SGE (Son of Grid Engine), Oracle Grid Engine, Univa Grid Engine.

Because the variety of Grid Engine derivatives and configuration deployments, it is not possible to use the `memory` keyword argument to automatically specify the amount of RAM requested. Instead, you specify the resources desired according to how your system is configured, using the `resource_spec` keyword argument, in addition to the `memory` keyword argument (which is used by Dask internally for memory management, see this for more details).

In the example below, our system administrator has used the `m_mem_free` keyword argument to let us request for RAM. Other known keywords may include `mem_req` and `mem_free`. We had to check with our cluster documentation and/or system administrator for this. At the same time, we must also correctly specify the `memory` keyword argument, to enable Dask's memory management to do its work correctly.

```
from dask_jobqueue import SGECluster

cluster = SGECluster(queue='default.q',
                     walltime="1500000",
                     processes=10,    # we request 10 processes per worker
                     memory='20GB',   # for memory requests, this must be specified
                     resource_spec='m_mem_free=20G',   # for memory requests, this␣
↪also needs to be specified
                     )
```

3.7.3 LSF Deployments

```
from dask_jobqueue import LSFCluster

cluster = LSFCluster(queue='general',
                     project='cpp',
                     walltime='00:30',
                     cores=15,
                     memory='25GB')
```

3.7.4 SLURM Deployments

```
from dask_jobqueue import SLURMCluster

cluster = SLURMCluster(cores=8,
                       processes=4,
                       memory="16GB",
                       project="woodshole",
                       walltime="01:00:00",
                       queue="normal")
```

3.7.5 SLURM Deployment: Low-priority node usage

```
from dask_jobqueue import SLURMCluster

cluster = SLURMCluster(cores=24,
```

(continues on next page)

(continued from previous page)

```
                        processes=6,
                        memory="16GB",
                        project="co_laika",
                        queue='savio2_bigmem',
                        env_extra=['export LANG="en_US.utf8"',
                                   'export LANGUAGE="en_US.utf8"',
                                   'export LC_ALL="en_US.utf8"'],
                        job_extra=['--qos="savio_lowprio"'])
```

3.7.6 SLURM Deployment: Providing additional arguments to the dask-workers

Keyword arguments can be passed through to dask-workers. An example of such an argument is for the specification of abstract resources, described here. This could be used to specify special hardware availability that the scheduler is not aware of, for example GPUs. Below, the arbitrary resources "ssdGB" and "GPU" are specified. Notice that the `extra` keyword is used to pass through arguments to the dask-workers.

```
from dask_jobqueue import SLURMCluster
from distributed import Client
from dask import delayed

cluster = SLURMCluster(memory='8g',
                       processes=1,
                       cores=2,
                       extra=['--resources ssdGB=200,GPU=2'])

cluster.start_workers(2)
client = Client(cluster)
```

The client can then be used as normal. Additionally, required resources can be specified for certain steps in the processing. For example:

```
def step_1_w_single_GPU(data):
    return "Step 1 done for: %s" % data


def step_2_w_local_IO(data):
    return "Step 2 done for: %s" % data


stage_1 = [delayed(step_1_w_single_GPU)(i) for i in range(10)]
stage_2 = [delayed(step_2_w_local_IO)(s2) for s2 in stage_1]

result_stage_2 = client.compute(stage_2,
                                resources={tuple(stage_1): {'GPU': 1},
                                           tuple(stage_2): {'ssdGB': 100}})
```

3.8 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.

3.8.1 Cheyenne

NCAR's Cheyenne Supercomputer uses both PBS (for Cheyenne itself) and Slurm (for the attached DAV clusters Geyser/Caldera).

```
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:
    name: dask-worker
    cores: 36                  # Total number of cores per job
    memory: '109 GB'           # Total amount of memory per job
    processes: 9               # Number of Python processes per job
    interface: ib0             # Network interface to use like eth0 or ib0

    queue: regular
    walltime: '00:30:00'
    resource-spec: select=1:ncpus=36:mem=109GB

  slurm:
    name: dask-worker

    # Dask worker options
    cores: 1                   # Total number of cores per job
    memory: '25 GB'            # Total amount of memory per job
    processes: 1               # Number of Python processes per job

    interface: ib0

    project: PXYZ123
    walltime: '00:30:00'
    job-extra: {-C geyser}
```

3.8.2 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.

Alternatively you may use the NERSC jupyterhub which will launch a notebook server on a reserved large memory

node of Cori. In this case no special interactive session is needed and dask jobqueue will perform as expected. You can also access the Dask dashboard directly. See an example notebook

```
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: 115GB
        processes: 4
        queue: debug
        walltime: '00:10:00'
        job-extra: ['-C haswell', '-L project, SCRATCH, cscratch1']
```

3.8.3 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
    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']
```

3.8.4 SDSC Comet

San Diego Supercomputer Center's Comet cluster, available to US scientists via XSEDE. Also, note that port 8787 is open both on login and computing nodes, so you can directly access Dask's dashboard.

```
jobqueue:
  slurm:
    name: dask-worker

    # Dask worker options
    cores: 24                  # Total number of cores per job
    memory: 120GB              # Total amount of memory per job (total 128GB per␣
↪node)
    processes: 1               # Number of Python processes per job

    interface: ib0             # Network interface to use like eth0 or ib0
    death-timeout: 60          # Number of seconds to wait if a worker can not find␣
↪a scheduler
```

(continues on next page)

(continued from previous page)

```
    local-directory: /scratch/$USER/$SLURM_JOB_ID # local SSD

    # SLURM resource manager options
    queue: compute
    # project: xxxxxxx # choose project other than default
    walltime: '00:30:00'
    job-mem: 120GB              # Max memory that can be requested to SLURM
```

3.8.5 Ifremer DATARMOR

See this (French) or this (English through Google Translate) for more details about the Ifremer DATARMOR cluster.

See this for more details about this `dask-jobqueue` config.

```
jobqueue:
  pbs:
    name: dask-worker

    # Dask worker options
    # number of processes and core have to be equal to avoid using multiple
    # threads in a single dask worker. Using threads can generate netcdf file
    # access errors.
    cores: 28
    processes: 28
    # this is using all the memory of a single node and corresponds to about
    # 4GB / dask worker. If you need more memory than this you have to decrease
    # cores and processes above
    memory: 120GB
    interface: ib0
    # This should be a local disk attach to your worker node and not a network
    # mounted disk. See
    # https://jobqueue.dask.org/en/latest/configuration-setup.html#local-storage
    # for more details.
    local-directory: $TMPDIR

    # PBS resource manager options
    queue: mpi_1
    project: myPROJ
    walltime: '48:00:00'
    resource-spec: select=1:ncpus=28:mem=120GB
    # disable email
    job-extra: ['-m n']
```

3.9 API

HTCondorCluster([disk, job_extra, config_name])	Launch Dask on an HTCondor cluster with a shared file system
LSFCluster([queue, project, ncpus, mem, ...])	Launch Dask on a LSF cluster
MoabCluster([queue, project, resource_spec, ...])	Launch Dask on a Moab cluster
OARCluster([queue, project, resource_spec, ...])	Launch Dask on a OAR cluster
PBSCluster([queue, project, resource_spec, ...])	Launch Dask on a PBS cluster

Continued on next page

Table 1 – continued from previous page

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

3.9.1 dask_jobqueue.HTCondorCluster

class dask_jobqueue.**HTCondorCluster**(*disk=None*, *job_extra=None*, *config_name='htcondor'*, ***kwargs*)

Launch Dask on an HTCondor cluster with a shared file system

Parameters

disk [str] Total amount of disk per job

job_extra [dict] Extra submit file attributes for the 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 [list] Additional arguments to pass to *dask-worker*

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

log_directory [str] Directory to use for job scheduler logs.

shebang [str] Path to desired interpreter for your batch submission script.

python [str] Python executable used to launch Dask workers.

config_name [str] Section to use from jobqueue.yaml configuration file.

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

Examples

```
>>> from dask_jobqueue.htcondor import HTCondorCluster
>>> cluster = HTCondorCluster(cores=24, memory="4GB", disk="4GB")
>>> cluster.scale(10)
```

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

This also works with adaptive clusters. This automatically launches and kill workers based on load. HTCondor can take longer to start jobs than other batch systems - tune Adaptive parameters accordingly.

```
>>> cluster.adapt(minimum=5, startup_cost='60s')
```

__init__(*self*, *disk=None*, *job_extra=None*, *config_name='htcondor'*, ***kwargs*)

Methods

__init__(self[, disk, job_extra, config_name])	
adapt(self[, minimum_cores, maximum_cores, ...])	Turn on adaptivity For keyword arguments see dask.distributed.Adaptive Instead of minimum and maximum parameters which apply to the number of worker, If Cluster object implements jobqueue_worker_spec attribute, one can use the following parameters: Parameters ———- minimum_cores: int Minimum number of cores for the cluster maximum_cores: int Maximum number of cores for the cluster minimum_memory: str Minimum amount of memory for the cluster maximum_memory: str Maximum amount of memory for the cluster Examples ——— >>> cluster.adapt(minimum=0, maximum=10, interval='500ms') >>> cluster.adapt(minimum_cores=24, maximum_cores=96) >>> cluster.adapt(minimum_memory='60 GB', maximum_memory= '1 TB')
close(self, **kwargs)	Stops all running and pending jobs and stops scheduler
env_lines_to_dict(self, env_lines)	Convert an array of export statements (what we get from env-extra in the config) into a dict
job_file(self)	Write job submission script to temporary file
job_script(self)	Construct a job submission script
scale(self[, n, cores, memory])	Scale cluster to n workers or to the given number of cores or memory number of cores and memory are converted into number of workers using jobqueue_worker_spec attribute.
scale_down(self, workers[, n])	Close the workers with the given addresses
scale_up(self, n, **kwargs)	Brings total worker count up to n
start_workers(self[, n])	Start workers and point them to our local scheduler
stop_all_jobs(self)	Stops all running and pending jobs
stop_jobs(self, jobs)	Stop a list of jobs
stop_workers(self, workers)	Stop a list of workers
worker_key(worker_state)	Callable mapping a WorkerState object to a group, see Scheduler.workers_to_close

Attributes

cancel_command	
dashboard_link	
executable	
finished_jobs	Jobs that have finished
job_id_regexp	
jobqueue_worker_spec	single worker process info needed for scaling on cores or memory
loop	
pending_jobs	Jobs pending in the queue

Continued on next page

Table 3 – continued from previous page

running_jobs	Jobs with currently active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_comm	
scheduler_info	
submit_command	
worker_process_memory	
worker_process_threads	
workers	workers currently connected to the scheduler

3.9.2 dask_jobqueue.LSFCluster

class dask_jobqueue.**LSFCluster**(*queue=None*, *project=None*, *ncpus=None*, *mem=None*, *walltime=None*, *job_extra=None*, *lsf_units=None*, *config_name='lsf'*, ***kwargs*)

Launch Dask on a LSF cluster

Parameters

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

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

ncpus [int] Number of cpus. Passed to *#BSUB -n* option.

mem [int] Request memory in bytes. Passed to *#BSUB -M* option.

walltime [str] Walltime for each worker job in HH:MM. Passed to *#BSUB -W* option.

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

lsf_units [str] Unit system for large units in resource usage set by the LSF_UNIT_FOR_LIMITS in the lsf.conf file of a cluster.

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 [list] Additional arguments to pass to *dask-worker*

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

log_directory [str] Directory to use for job scheduler logs.

shebang [str] Path to desired interpreter for your batch submission script.

python [str] Python executable used to launch Dask workers.

config_name [str] Section to use from jobqueue.yaml configuration file.

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

Examples

```
>>> from dask_jobqueue import LSFCluster
>>> cluster = LSFcluster(queue='general', project='DaskonLSF',
...                      cores=15, memory='25GB')
>>> 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__(*self*, *queue=None*, *project=None*, *ncpus=None*, *mem=None*, *walltime=None*, *job_extra=None*, *lsf_units=None*, *config_name='lsf'*, ***kwargs*)

Methods

__init__(self[, queue, project, ncpus, mem, ...])	
adapt(self[, minimum_cores, maximum_cores, ...])	Turn on adaptivity For keyword arguments see dask.distributed.Adaptive Instead of minimum and maximum parameters which apply to the number of worker, If Cluster object implements jobqueue_worker_spec attribute, one can use the following parameters: Parameters ———- minimum_cores: int Minimum number of cores for the cluster maximum_cores: int Maximum number of cores for the cluster minimum_memory: str Minimum amount of memory for the cluster maximum_memory: str Maximum amount of memory for the cluster Examples ——— >>> cluster.adapt(minimum=0, maximum=10, interval='500ms') >>> cluster.adapt(minimum_cores=24, maximum_cores=96) >>> cluster.adapt(minimum_memory='60 GB', maximum_memory= '1 TB')
close(self, **kwargs)	Stops all running and pending jobs and stops scheduler
job_file(self)	Write job submission script to temporary file
job_script(self)	Construct a job submission script
scale(self[, n, cores, memory])	Scale cluster to n workers or to the given number of cores or memory number of cores and memory are converted into number of workers using jobqueue_worker_spec attribute.
scale_down(self, workers[, n])	Close the workers with the given addresses
scale_up(self, n, **kwargs)	Brings total worker count up to n
start_workers(self[, n])	Start workers and point them to our local scheduler
stop_all_jobs(self)	Stops all running and pending jobs
stop_jobs(self, jobs)	Stop a list of jobs
stop_workers(self, workers)	Stop a list of workers

Continued on next page

Table 4 – continued from previous page

worker_key(worker_state)	Callable mapping a WorkerState object to a group, see Scheduler.workers_to_close

Attributes

cancel_command	
dashboard_link	
finished_jobs	Jobs that have finished
job_id_regexp	
jobqueue_worker_spec	single worker process info needed for scaling on cores or memory
loop	
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currently active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_comm	
scheduler_info	
submit_command	
worker_process_memory	
worker_process_threads	
workers	workers currently connected to the scheduler

3.9.3 dask_jobqueue.MoabCluster

class dask_jobqueue.**MoabCluster**(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='pbs'*, ***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 [list] Additional arguments to pass to *dask-worker*

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

log_directory [str] Directory to use for job scheduler logs.

shebang [str] Path to desired interpreter for your batch submission script.

python [str] Python executable used to launch Dask workers.

config_name [str] Section to use from jobqueue.yaml configuration file.

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

Examples

```
>>> import os
>>> from dask_jobqueue import MoabCluster
>>> cluster = MoabCluster(processes=6, cores=6, project='gfdl_m',
                          memory='96G', resource_spec='96G',
                          job_extra=['-d /home/First.Last', '-M none'],
                          local_directory=os.getenv('TMPDIR', '/tmp'))
>>> cluster.scale(60)  # 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__(*self*, *queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='pbs'*, ***kwargs*)

Methods

__init__(self[, queue, project, ...])	
adapt(self[, minimum_cores, maximum_cores, ...])	Turn on adaptivity For keyword arguments see dask.distributed.Adaptive Instead of minimum and maximum parameters which apply to the number of worker, If Cluster object implements jobqueue_worker_spec attribute, one can use the following parameters: Parameters ———- minimum_cores: int Minimum number of cores for the cluster maximum_cores: int Maximum number of cores for the cluster minimum_memory: str Minimum amount of memory for the cluster maximum_memory: str Maximum amount of memory for the cluster Examples ——— >>> cluster.adapt(minimum=0, maximum=10, interval='500ms') >>> cluster.adapt(minimum_cores=24, maximum_cores=96) >>> cluster.adapt(minimum_memory='60 GB', maximum_memory= '1 TB')
close(self, **kwargs)	Stops all running and pending jobs and stops scheduler

Continued on next page

Table 6 – continued from previous page

job_file(self)	Write job submission script to temporary file
job_script(self)	Construct a job submission script
scale(self[, n, cores, memory])	Scale cluster to n workers or to the given number of cores or memory number of cores and memory are converted into number of workers using jobqueue_worker_spec attribute.
scale_down(self, workers[, n])	Close the workers with the given addresses
scale_up(self, n, **kwargs)	Brings total worker count up to n
start_workers(self[, n])	Start workers and point them to our local scheduler
stop_all_jobs(self)	Stops all running and pending jobs
stop_jobs(self, jobs)	Stop a list of jobs
stop_workers(self, workers)	Stop a list of workers
worker_key(worker_state)	Callable mapping a WorkerState object to a group, see Scheduler.workers_to_close

Attributes

cancel_command	
dashboard_link	
finished_jobs	Jobs that have finished
job_id_regexp	
jobqueue_worker_spec	single worker process info needed for scaling on cores or memory
loop	
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currently active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_comm	
scheduler_info	
scheduler_name	
submit_command	
worker_process_memory	
worker_process_threads	
workers	workers currently connected to the scheduler

3.9.4 dask_jobqueue.OARCluster

class dask_jobqueue.**OARCluster**(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='oar'*, ***kwargs*)

Launch Dask on a OAR cluster

Parameters

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

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

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

walltime [str] Walltime for each worker job.

job_extra [list] List of other OAR options, for example *-t besteffort*. Each option will be prepended with the #OAR 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 [list] Additional arguments to pass to *dask-worker*

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

log_directory [str] Directory to use for job scheduler logs.

shebang [str] Path to desired interpreter for your batch submission script.

python [str] Python executable used to launch Dask workers.

config_name [str] Section to use from jobqueue.yaml configuration file.

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

Examples

```
>>> from dask_jobqueue import OARCluster
>>> cluster = OARCluster(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__(*self*, *queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='oar'*, ***kwargs*)

Methods

__init__(self[, queue, project, ...])	

Continued on next page

Table 8 – continued from previous page

adapt(self[, minimum_cores, maximum_cores, ...])	Turn on adaptivity For keyword arguments see dask.distributed.Adaptive Instead of minimum and maximum parameters which apply to the number of worker, If Cluster object implements jobqueue_worker_spec attribute, one can use the following parameters: Parameters ———- minimum_cores: int Minimum number of cores for the cluster maximum_cores: int Maximum number of cores for the cluster minimum_memory: str Minimum amount of memory for the cluster maximum_memory: str Maximum amount of memory for the cluster Examples ——— >>> cluster.adapt(minimum=0, maximum=10, interval='500ms') >>> cluster.adapt(minimum_cores=24, maximum_cores=96) >>> cluster.adapt(minimum_memory='60 GB', maximum_memory= '1 TB')
close(self, **kwargs)	Stops all running and pending jobs and stops scheduler
job_file(self)	Write job submission script to temporary file
job_script(self)	Construct a job submission script
scale(self[, n, cores, memory])	Scale cluster to n workers or to the given number of cores or memory number of cores and memory are converted into number of workers using jobqueue_worker_spec attribute.
scale_down(self, workers[, n])	Close the workers with the given addresses
scale_up(self, n, **kwargs)	Brings total worker count up to n
start_workers(self[, n])	Start workers and point them to our local scheduler
stop_all_jobs(self)	Stops all running and pending jobs
stop_jobs(self, jobs)	Stop a list of jobs
stop_workers(self, workers)	Stop a list of workers
worker_key(worker_state)	Callable mapping a WorkerState object to a group, see Scheduler.workers_to_close

Attributes

cancel_command	
dashboard_link	
finished_jobs	Jobs that have finished
job_id_regexp	
jobqueue_worker_spec	single worker process info needed for scaling on cores or memory
loop	
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currently active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_comm	
scheduler_info	
submit_command	

Continued on next page

Table 9 – continued from previous page

worker_process_memory	
worker_process_threads	
workers	workers currently connected to the scheduler

3.9.5 dask_jobqueue.PBSCluster

class dask_jobqueue.**PBSCluster**(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='pbs'*, ***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 [list] Additional arguments to pass to *dask-worker*

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

log_directory [str] Directory to use for job scheduler logs.

shebang [str] Path to desired interpreter for your batch submission script.

python [str] Python executable used to launch Dask workers.

config_name [str] Section to use from jobqueue.yaml configuration file.

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

Examples

```
>>> from dask_jobqueue import PBSCluster
>>> cluster = PBSCluster(queue='regular', project='DaskOnPBS', cores=12)
>>> 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:

```
>>> cluster = PBSCluster(queue='regular', project='DaskOnPBS',
...                      local_directory='$TMPDIR',
...                      cores=24, processes=6, memory='100GB')
```

__init__(*self*, *queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='pbs'*, ***kwargs*)

Methods

__init__(self[, queue, project, ...])	
adapt(self[, minimum_cores, maximum_cores, ...])	Turn on adaptivity For keyword arguments see dask.distributed.Adaptive Instead of minimum and maximum parameters which apply to the number of worker, If Cluster object implements jobqueue_worker_spec attribute, one can use the following parameters: Parameters ———- minimum_cores: int Minimum number of cores for the cluster maximum_cores: int Maximum number of cores for the cluster minimum_memory: str Minimum amount of memory for the cluster maximum_memory: str Maximum amount of memory for the cluster Examples ——— >>> cluster.adapt(minimum=0, maximum=10, interval='500ms') >>> cluster.adapt(minimum_cores=24, maximum_cores=96) >>> cluster.adapt(minimum_memory='60 GB', maximum_memory= '1 TB')
close(self, **kwargs)	Stops all running and pending jobs and stops scheduler
job_file(self)	Write job submission script to temporary file
job_script(self)	Construct a job submission script
scale(self[, n, cores, memory])	Scale cluster to n workers or to the given number of cores or memory number of cores and memory are converted into number of workers using jobqueue_worker_spec attribute.
scale_down(self, workers[, n])	Close the workers with the given addresses
scale_up(self, n, **kwargs)	Brings total worker count up to n
start_workers(self[, n])	Start workers and point them to our local scheduler
stop_all_jobs(self)	Stops all running and pending jobs
stop_jobs(self, jobs)	Stop a list of jobs
stop_workers(self, workers)	Stop a list of workers
worker_key(worker_state)	Callable mapping a WorkerState object to a group, see Scheduler.workers_to_close

Attributes

cancel_command	
dashboard_link	
finished_jobs	Jobs that have finished
job_id_regexp	
jobqueue_worker_spec	single worker process info needed for scaling on cores or memory
loop	
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currently active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_comm	
scheduler_info	
submit_command	
worker_process_memory	
worker_process_threads	
workers	workers currently connected to the scheduler

3.9.6 dask_jobqueue.SGECluster

class dask_jobqueue.**SGECluster**(*queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='sge'*, ***kwargs*)

Launch Dask on a SGE cluster

Note: If you want a specific amount of RAM, both `memory` and `resource_spec` must be specified. The exact syntax of `resource_spec` is defined by your GridEngine system administrator. The amount of `memory` requested should match the `resource_spec`, so that Dask's memory management system can perform accurately.

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.

job_extra [list] List of other SGE options, for example -w e. Each option will be prepended with the #$ 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 [list] Additional arguments to pass to *dask-worker*

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

log_directory [str] Directory to use for job scheduler logs.

shebang [str] Path to desired interpreter for your batch submission script.

python [str] Python executable used to launch Dask workers.

config_name [str] Section to use from jobqueue.yaml configuration file.

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__(*self*, *queue=None*, *project=None*, *resource_spec=None*, *walltime=None*, *job_extra=None*, *config_name='sge'*, ***kwargs*)

Methods

__init__(self[, queue, project, ...])	
adapt(self[, minimum_cores, maximum_cores, ...])	Turn on adaptivity For keyword arguments see dask.distributed.Adaptive Instead of minimum and maximum parameters which apply to the number of worker, If Cluster object implements jobqueue_worker_spec attribute, one can use the following parameters: Parameters ———- minimum_cores: int Minimum number of cores for the cluster maximum_cores: int Maximum number of cores for the cluster minimum_memory: str Minimum amount of memory for the cluster maximum_memory: str Maximum amount of memory for the cluster Examples ——— >>> cluster.adapt(minimum=0, maximum=10, interval='500ms') >>> cluster.adapt(minimum_cores=24, maximum_cores=96) >>> cluster.adapt(minimum_memory='60 GB', maximum_memory= '1 TB')
close(self, **kwargs)	Stops all running and pending jobs and stops scheduler
job_file(self)	Write job submission script to temporary file
job_script(self)	Construct a job submission script

Continued on next page

Table 12 – continued from previous page

scale(self[, n, cores, memory])	Scale cluster to n workers or to the given number of cores or memory number of cores and memory are converted into number of workers using jobqueue_worker_spec attribute.
scale_down(self, workers[, n])	Close the workers with the given addresses
scale_up(self, n, **kwargs)	Brings total worker count up to n
start_workers(self[, n])	Start workers and point them to our local scheduler
stop_all_jobs(self)	Stops all running and pending jobs
stop_jobs(self, jobs)	Stop a list of jobs
stop_workers(self, workers)	Stop a list of workers
worker_key(worker_state)	Callable mapping a WorkerState object to a group, see Scheduler.workers_to_close

Attributes

cancel_command	
dashboard_link	
finished_jobs	Jobs that have finished
job_id_regexp	
jobqueue_worker_spec	single worker process info needed for scaling on cores or memory
loop	
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currently active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_comm	
scheduler_info	
submit_command	
worker_process_memory	
worker_process_threads	
workers	workers currently connected to the scheduler

3.9.7 dask_jobqueue.SLURMCluster

class dask_jobqueue.**SLURMCluster**(*queue=None*, *project=None*, *walltime=None*, *job_cpu=None*, *job_mem=None*, *job_extra=None*, *config_name='slurm'*, ***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 [list] Additional arguments to pass to *dask-worker*

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

log_directory [str] Directory to use for job scheduler logs.

shebang [str] Path to desired interpreter for your batch submission script.

python [str] Python executable used to launch Dask workers.

config_name [str] Section to use from jobqueue.yaml configuration file.

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

Examples

```
>>> from dask_jobqueue import SLURMCluster
>>> cluster = SLURMCluster(processes=6, cores=24, memory="120GB",
                           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__(*self*, *queue=None*, *project=None*, *walltime=None*, *job_cpu=None*, *job_mem=None*, *job_extra=None*, *config_name='slurm'*, ***kwargs*)

Methods

__init__(self[, queue, project, walltime, ...])	

Continued on next page

Table 14 – continued from previous page

adapt(self[, minimum_cores, maximum_cores, ...])	Turn on adaptivity For keyword arguments see dask.distributed.Adaptive Instead of minimum and maximum parameters which apply to the number of worker, If Cluster object implements jobqueue_worker_spec attribute, one can use the following parameters: Parameters ———- minimum_cores: int Minimum number of cores for the cluster maximum_cores: int Maximum number of cores for the cluster minimum_memory: str Minimum amount of memory for the cluster maximum_memory: str Maximum amount of memory for the cluster Examples ——— >>> cluster.adapt(minimum=0, maximum=10, interval='500ms') >>> cluster.adapt(minimum_cores=24, maximum_cores=96) >>> cluster.adapt(minimum_memory='60 GB', maximum_memory= '1 TB')
close(self, **kwargs)	Stops all running and pending jobs and stops scheduler
job_file(self)	Write job submission script to temporary file
job_script(self)	Construct a job submission script
scale(self[, n, cores, memory])	Scale cluster to n workers or to the given number of cores or memory number of cores and memory are converted into number of workers using jobqueue_worker_spec attribute.
scale_down(self, workers[, n])	Close the workers with the given addresses
scale_up(self, n, **kwargs)	Brings total worker count up to n
start_workers(self[, n])	Start workers and point them to our local scheduler
stop_all_jobs(self)	Stops all running and pending jobs
stop_jobs(self, jobs)	Stop a list of jobs
stop_workers(self, workers)	Stop a list of workers
worker_key(worker_state)	Callable mapping a WorkerState object to a group, see Scheduler.workers_to_close

Attributes

cancel_command	
dashboard_link	
finished_jobs	Jobs that have finished
job_id_regexp	
jobqueue_worker_spec	single worker process info needed for scaling on cores or memory
loop	
pending_jobs	Jobs pending in the queue
running_jobs	Jobs with currently active workers
scheduler	The scheduler of this cluster
scheduler_address	
scheduler_comm	
scheduler_info	
submit_command	

Continued on next page

Table 15 – continued from previous page

worker_process_memory	
worker_process_threads	
workers	workers currently connected to the scheduler

3.10 How to debug

Dask jobqueue has been developed and tested by several contributors, each having a given HPC system setup to work on: a job scheduler in a given version running on a given OS. Thus, in some specific cases, it might not work out of the box on your system. This section provides some hints to help you determine what may be going wrong.

3.10.1 Checking job script

Dask-jobqueue submits "job scripts" to your queueing system (see *How this works*). Inspecting these scripts often reveals errors in the configuration of your Cluster object or maybe directives unexpected by your job scheduler, in particular the header containing `#PBS`, `#SBATCH` or equivalent lines. This can be done easily once you've created a cluster object:

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

If everything in job script appears correct, the next step is to try to submit a test job using the script. You can simply copy and paste printed content to a real job script file, and submit it using `qsub`, `sbatch`, `bsub` or what is appropriate for you job queuing system.

To correct any problem detected at this point, you could try to use `job_extra` or `env_extra` kwargs when initializing your cluster object.

3.10.2 Activate debug mode

Dask-jobqueue uses the Python logging module. To understand better what is happening under the hood, you may want to activate logging display. This can be done by running this line of python code in your script or notebook:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
```

3.10.3 Interact with your job queuing system

Every worker is launched inside a batch job, as explained above. It can be very helpful to query your job queuing system. Some things you might want to check:

- are there running jobs related to dask-jobqueue?
- are there finished jobs, error jobs?
- what is the stdout or stderr of dask-jobqueue jobs?

3.10.4 Other things you might look at

From here it gets a little more complicated. A couple of other already seen problems are the following:

- The submit command used in dask-jobqueue (`qsub` or equivalent) doesn't correspond to the one that you use. Check in the given `JobQueueCluster` implementation that job submission command and arguments look familiar to you, eventually try them.
- The submit command output is not the same as the one expected by dask-jobqueue. We use submit command stdout to parse the job_id corresponding to the launched group of worker. If the parsing fails, then dask-jobqueue won't work as expected and may throw exceptions. You can have a look at the parsing function `JobQueueCluster._job_id_from_submit_output`.

3.11 Changelog

3.11.1 0.6.2 / 2019-07-31

- Update documentation
- Ensure compatibility with Dask 2.2 (GH#303)

3.11.2 0.6.1 / 2019-07-25

- more fixes related to `distributed >= 2` changes (GH#278, GH#291)
- `distributed >= 2.1` is now required (GH#295)
- remove deprecated `threads` parameter from all the `Cluster` classes (GH#297)
- doc improvements (GH#290, GH#294, GH#296)

3.11.3 0.6.0 / 2019-07-06

- Drop Python 2 support (GH#284)
- Fix adaptive compatibility with SpecificationCluster in Distributed 2.0 (GH#282)

3.11.4 0.5.0 / 2019-06-20

- Keeping up to date with Dask and Distributed (GH#268)
- Formatting with Black (GH#256, GH#248)
- Improve some batch scheduler integration (GH#274, GH#256, GH#232)
- Add HTCondor compatibility (GH#245)
- Add the possibility to specify named configuration (:pr: *204*)
- Allow free configuration of Dask diagnostic_port (:pr: *192)*
- Start work on ClusterManager, see https://github.com/dask/distributed/issues/2235 (GH#187, GH#184, GH#183)
- A lot of other tiny fixes and improvements(GH#277, GH#261, GH#260, GH#250, GH#244, GH#200, GH#189)

3.11.5 0.4.1 / 2018-10-18

- Handle worker restart with clearer message (GH#138)
- Better error handling on job submission failure (GH#146)
- Fixed Python 2.7 error when starting workers (GH#155)
- Better handling of extra scheduler options (GH#160)
- Correct testing of Python 2.7 compatibility (GH#154)
- Add ability to override python used to start workers (GH#167)
- Internal improvements and edge cases handling (GH#97)
- Possibility to specify a folder to store every job logs file (GH#145)
- Require all cores on the same node for LSF (GH#177)

3.11.6 0.4.0 / 2018-09-06

- Use number of worker processes as an argument to `scale` instead of number of jobs.
- Bind scheduler bokeh UI to every network interfaces by default.
- Adds an OAR job queue system implementation.
- Adds an LSF job queue system implementation.
- Adds some convenient methods to JobQueueCluster objects: `__repr__`, `stop_jobs()`, `close()`.

3.12 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.

3.12.1 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-jobqueue.git
```

and install it from source:

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

3.12.2 Formatting

When you're done making changes, check that your changes pass flake8 checks and use black formatting:

```
flake8 dask_jobqueue
black dask_jobqueue
```

To get flake8 and black, just pip install them. You can also use pre-commit to add them as pre-commit hooks.

3.12.3 Test

Test using `pytest`:

```
pytest dask-jobqueue --verbose
```

3.12.4 Test with Job scheduler

Some tests require to have a fully functional job queue cluster running, this is done through Docker and Docker compose tools. You must thus have them installed on your system following their docs.

You can then use the same commands as Travis CI does for your local testing, for example with pbs:

```
source ci/pbs.sh
jobqueue_before_install
jobqueue_install
jobqueue_script
```

3.13 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

INDEX

Symbols

H

L

M

O

P

S