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.
from dask_jobqueue import PBSCluster
cluster = PBSCluster()
cluster.scale(jobs=10)  # Deploy ten single-node jobs

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
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.

```python
cluster.adapt(minimum_jobs=10, maximum_jobs=100)  # auto-scale between 10 and 100 jobs
cluster.adapt(maximum_memory="10 TB")  # or use core/memory limits
```
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):

```bash
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:

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

#### 3.1.3 Install from Source

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

```bash
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:

```bash
pip install -e .
```

You can also install directly from git master branch:

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

Test dask-jobqueue with pytest:

```bash
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.

We recommend first doing these steps from a login node (nothing will be computationally intensive) but at some point you may want to shift to a compute or interactive node.

_Note: We also recommend the `JupyterHub <https://jupyter.org/hub>`_ project, which allows HPC administrators to offer and control the process described in this document automatically. If you find this process valuable but tedious, then you may want to ask your system administrators to support it with JupyterHub._

3.2.1 Install JupyterLab

We recommend using JupyterLab, and the Dask JupyterLab Extension. This will make it easy to get Dask’s dashboard through your Jupyter session.

These can be installed with the following steps:

```bash
# Install JupyterLab and NodeJS (which we'll need to integrate Dask into JLab)
conda install jupyterlab nodejs -c conda-forge -y

# Install server-side pieces of the Dask-JupyterLab extension
pip install dask_labextension

# Integrate Dask-Labextension with Jupyter (requires NodeJS)
jupyter labextension install dask-labextension
```

You can also use `pip` rather than `conda`, but you will have to find some other way to install NodeJS.
3.2.2 Add A Password

For security, we recommend adding a password to your Jupyter notebook configuration.

```
jupyter notebook password
```

This is good both for security, and also means that you won’t have to copy around Jupyter tokens.

3.2.3 Start Jupyter

When you use Jupyter on your laptop you often just write `jupyter notebook` or `jupyter lab`. However, things are a bit different when starting a notebook server on a separate machine. As a first step, the following will work:

```
jupyter lab --no-browser --ip="*" --port 8888
```

Later, once we get SSH tunneling set up, you may want to come back and specify a specific IP address or hostname for added security.

3.2.4 SSH Tunneling

If your personal machine is on the same network as your cluster, then you can ignore this step.

If you are on a different network (like your home network), and have to SSH in, then it can be difficult to have your local web browser connect to the Jupyter server running on the HPC machine. If your institution doesn’t have something like JupyterHub set up, then the easiest way to accomplish this is to use SSH tunneling.

Often a command like the following works:

```
ssh -L 8888:login-node-hostname:8888 username@hpc.agency.gov
```

Where `login-node-hostname` and `username@hpc.agency.gov` are placeholders that you need to fill in:

- `login-node-hostname` is the name of the node from which you are running your Jupyter server, designated hostname below.

  ````
  username@hostname$ jupyter lab --no-browser --ip="*" --port 8888
  ````

  You might also run `echo $HOSTNAME` on that machine as well to see the host name.

- `hpc.agency.gov` is the address that you usually use to ssh into the cluster.

So in a real example this might look like the following:

```
alice@login2.summit $ jupyter lab --no-browser --ip="login2" --port 8888
alice@laptop $ ssh -L 8888:login2:8888 alice@summit.olcf.ornl.gov
```

Additionally, if port 8888 is busy then you may want to choose a different port, like 9999. Someone else may be using this port, particularly if they are setting up their own Jupyter server on this machine.

You can now visit `http://localhost:8888` on your local browser to access the Jupyter server.
3.2.5 Viewing the Dask Dashboard

When you start a Dask Jobqueue cluster you also start a Dask dashboard. This dashboard is valuable to help you understand the state of your computation and cluster.

Typically, the dashboard is served on a separate port from Jupyter, and so can be used whether you choose to use Jupyter or not. If you want to open up a connection to see the dashboard you can do so with SSH Tunneling as described above. The dashboard’s default port is at 8787, and is configurable with the `dashboard_address` keyword to the Dask Jobqueue cluster objects.

However, Jupyter is also able to proxy the dashboard connection through the Jupyter server, allowing you to access the dashboard at `http://localhost:8888/proxy/8787/status`. This requires no additional SSH tunneling. Additionally, if you place this address into the Dask Labextension search bar (click the Dask logo icon on the left side of your Jupyter session) then you can access the plots directly within Jupyter Lab, rather than open up another tab.

3.2.6 Configuration

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:

```yaml
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:

```python
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',
)
```
These parameters specify the characteristics of a *single job* or a *single compute node*, rather than the characteristics of your computation as a whole. It hasn’t actually launched any jobs yet. For the full computation, you will then ask for a number of jobs using the `scale` command:

```python
cluster.scale(jobs=2)  # launch 2 jobs, each of which starts 6 worker processes
cluster.scale(cores=48)  # Or specify cores or memory directly
cluster.scale(memory="200 GB")  # Or specify cores or memory directly
```

You can either specify the number of jobs, or the total number of cores or memory that you want.

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:

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

```bash
#!/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:

```python
cluster = PBSCluster(
    # Dask-worker specific keywords
    cores=24,  # Number of cores per job
    memory='100GB',  # Amount of memory per job
    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. You can either specify the number of workers, or the total number of cores or memory that you want.

```python
cluster.scale(jobs=2)  # launch 2 workers, each of which starts 6 worker processes
cluster.scale(cores=48)  # Or specify cores or memory directly
cluster.scale(memory="200 GB")  # Or specify cores or memory directly
```

These all accomplish the same thing. You can chose whichever makes the most sense to you.

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:

```yaml
# 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.
You can still override configuration values with keyword arguments

```python
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:

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

Or as part of a configuration file:

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

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”.

```yaml
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 queueing 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 $\approx 18.6$GiB, 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  Bcast:255.0.0.0  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  Bcast:192.168.0.255  Mask:255.255.255.0
  inet6 addr: ::1/128 Scope:Link
  inet6 addr: fe80::201:8c32:6959/64 Scope:Link
  UP BROADCAST RUNNING MTU:1500  Metric:1
  RX packets:22883153 errors:0 drop:0 overruns:0 frame:0
  TX packets:1410823  errors:0 dropped:0 overruns:0 carrier:0
  collisions:0 txqueuelen:1000
  RX bytes:2231868428 (2.1 GiB)  TX bytes:280834784 (274 MiB)
```

```
ib0        Link encap:Infiniband                          # Fast InfiniBand
  inet addr:172.42.0.101 Scope:Link
```

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

```python
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

```python
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`:

```python
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.

```python
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
                      )
```

### 3.7.3 LSF Deployments

```python
from dask_jobqueue import LSFCluster

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

3.7. Example Deployments
3.7.4 SLURM Deployments

```python
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

```python
from dask_jobqueue import SLURMCluster

cluster = SLURMCluster(cores=24,
        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.

```python
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:

```python
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
```

(continues on next page)
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 ma-
chines 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

(continues on next page)
3.8.2 NERSC Cori

NERSC Cori Supercomputer

It should be noted that 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:

```bash
$s alloc -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.

```yaml
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.

```yaml
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
```

(continues on next page)
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.

```yaml
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
    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.

```yaml
jobqueue:
  pbs:
    name: dask-worker
    # Dask worker options
    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
```

(continues on next page)
local-directory: $TMPDIR

# PBS resource manager options
queue: mpi_1
project: myPROJ
walltime: '48:00:00'
resource-spec: select=1:ncpus=28:mem=120GB

job-extra: ['-m n']

3.9 API

**HTCondorCluster**([n_workers, job_cls, loop, . . .]) Launch Dask on an HTCondor cluster with a shared file system

**LSFCluster**([n_workers, job_cls, loop, . . .]) Launch Dask on a LSF cluster

**MoabCluster**([n_workers, job_cls, loop, . . .]) Launch Dask on a PBS cluster

**OARCluster**([n_workers, job_cls, loop, . . .]) Launch Dask on an OAR cluster

**PBSCluster**([n_workers, job_cls, loop, . . .]) Launch Dask on a PBS cluster

**SGECluster**([n_workers, job_cls, loop, . . .]) Launch Dask on an SGE cluster

**SLURMCluster**([n_workers, job_cls, loop, . . .]) Launch Dask on a SLURM cluster

3.9.1 **dask_jobqueue.HTCondorCluster**

```python
dask_jobqueue.HTCondorCluster(n_workers=0, job_cls: dask_jobqueue.core.Job = None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **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
- **cores** [int] Total number of cores per job
- **memory** [str] Total amount of memory per job
- **processes** [int] Cut the job up into this many processes. Good for GIL workloads or for nodes with many cores. By default, processes = sqrt(cores) so that the number of processes and the number of threads per process is roughly the same.
- **interface** [str] Network interface like ‘eth0’ or ‘ib0’.
- **nanny** [bool] Whether or not to start a nanny process
- **local_directory** [str] Dask worker local directory for file spilling.
- **death_timeout** [float] Seconds to wait for a scheduler before closing workers
- **extra** [list] Additional arguments to pass to `dask-worker`
env_extra  [list] Other commands to add to script before launching worker.
header_skip  [list] Lines to skip in the header. Header lines matching this text will be removed
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. Defaults to the Python that is submitting these jobs
config_name  [str] Section to use from jobqueue.yaml configuration file.
name  [str] Name of Dask worker. This is typically set by the Cluster
n_workers  [int] Number of workers to start by default. Defaults to 0. See the scale method
silence_logs  [str] Log level like “debug”, “info”, or “error” to emit here if the scheduler is started locally
asynchronous  [bool] Whether or not to run this cluster object with the async/await syntax
security  [Security] A dask.distributed security object if you’re using TLS/SSL
dashboard_address  [str or int] An address like “:8787” on which to host the Scheduler’s dashboard

Examples

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

```python
>>> 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(maximum_jobs=20)
```

__init__

```
__init__ (self, n_workers=0, job_cls:dask_jobqueue.core.Job=None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)
Initialize self. See help(type(self)) for accurate signature.
```

Methods

```
__init__ (self[, n_workers, loop, security, ...])  Initialize self.
adapt(self, \*\*args, minimum_jobs, ...)  Scale Dask cluster automatically based on scheduler activity.
close(self[, timeout])
job_script(self)
logs(self[, scheduler, workers])  Return logs for the scheduler and workers
new_worker_spec(self)  Return name and spec for the next worker
scale(self[, n, jobs, memory, cores])  Scale cluster to specified configurations.
scale_down(self, workers)
```

Continued on next page
Table 2 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale_up(self[, n, memory, cores])</td>
<td>Scale cluster to n workers</td>
</tr>
<tr>
<td>sync(self, func, *args[, asynchronous, ...])</td>
<td></td>
</tr>
</tbody>
</table>

Attributes

- asynchronous
- config_name
- dashboard_link
- job_header
- job_name
- observed
- plan
- requested
- scheduler_address

3.9.2 dask_jobqueue.LSFCluster

```python
class dask_jobqueue.LSFCluster(n_workers=0, job_cls: dask_jobqueue.core.Job = None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **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.
- `cores` [int] Total number of cores per job
- `memory` [str] Total amount of memory per job
- `processes` [int] Cut the job up into this many processes. Good for GIL workloads or for nodes with many cores. By default, $process \approx \sqrt{\text{cores}}$ so that the number of processes and the number of threads per process is roughly the same.
- `interface` [str] Network interface like ‘eth0’ or ‘ib0’.
- `nanny` [bool] Whether or not to start a nanny process
- `local_directory` [str] Dask worker local directory for file spilling.
- `death_timeout` [float] Seconds to wait for a scheduler before closing workers
- `extra` [list] Additional arguments to pass to `dask-worker`
- `env_extra` [list] Other commands to add to script before launching worker.
- `header_skip` [list] Lines to skip in the header. Header lines matching this text will be removed
- `shebang` [str] Path to desired interpreter for your batch submission script.
- `python` [str] Python executable used to launch Dask workers. Defaults to the Python that is submitting these jobs
config_name  [str]  Section to use from jobqueue.yaml configuration file.

name  [str]  Name of Dask worker. This is typically set by the Cluster

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.

n_workers  [int]  Number of workers to start by default. Defaults to 0. See the scale method

silence_logs  [str]  Log level like “debug”, “info”, or “error” to emit here if the scheduler is

started locally

asynchronous  [bool]  Whether or not to run this cluster object with the async/await syntax

security  [Security]  A dask.distributed security object if you’re using TLS/SSL

dashboard_address  [str or int]  An address like “:8787” on which to host the Scheduler’s dash-

board

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.

use_stdin  [bool]  LSF’s bsub command allows us to launch a job by passing it as an argument

(bsub /tmp/jobscript.sh) or feeding it to stdin (bsub < /tmp/jobscript. sh). Depending on your cluster’s configuration and/or shared filesystem setup, one of those

methods may not work, forcing you to use the other one. This option controls which method
dask-jobqueue will use to submit jobs via bsub.

In particular, if your cluster fails to launch and the LSF log contains an error message similar
to the following:

```
/home/someuser/.lsbatch/1571869562.66512066: line 8: /tmp/va_˓
→yau8m.sh: No such file or directory
```

... then try passing use_stdin=True here or setting use-stdin: true in your

jobqueue.lsf config section.

Examples

```python
>>> from dask_jobqueue import LSFCluster
>>> cluster = LSFCluster(queue='general', project='DaskonLSF', ...
  cores=15, memory='25GB', use_stdin=True)
>>> cluster.scale(jobs=10)  # ask for 10 jobs
```

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

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

```python
>>> cluster.adapt(maximum_jobs=20)
```

___init___(self, n_workers=0, job_cls:dask_jobqueue.core.Job=None, loop=None, security=None, si-

lence_logs='error', name=None, asynchronous=False, interface=None, host=None, proto-

col='tcp://', dashboard_address=':8787', config_name=None, **kwargs)

Initialize self. See help(type(self)) for accurate signature.
## Dask-jobqueue Documentation, Release 0.7.0+12.g3f4cd0c

### Methods

**__init__** (self[, n_workers, loop, security, ...]) Initialize self.

**adapt**(self, *args, minimum_jobs, ... ) Scale Dask cluster automatically based on scheduler activity.

**close**(self[, timeout])

**job_script**(self)

**logs**(self[, scheduler, workers]) Return logs for the scheduler and workers

**new_worker_spec**(self) Return name and spec for the next worker

**scale**(self[, n, jobs, memory, cores]) Scale cluster to specified configurations.

**scale_down**(self, workers)

**scale_up**(self[, n, memory, cores]) Scale cluster to n workers

**sync**(self, func, *args[, asynchronous, ...])

### Attributes

**asynchronous**

**config_name**

**dashboard_link**

**job_header**

**job_name**

**observed**

**plan**

**requested**

**scheduler_address**

### 3.9.3 dask_jobqueue.MoabCluster

```python
class dask_jobqueue.MoabCluster(n_workers=0, job_cls: dask_jobqueue.core.Job = None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=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.

cores [int] Total number of cores per job

memory: str Total amount of memory per job

processes [int] Cut the job up into this many processes. Good for GIL workloads or for nodes with many cores. By default, `process == sqrt(cores)` so that the number of processes and the number of threads per process is roughly the same.

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

nanny [bool] Whether or not to start a nanny process

local_directory [str] Dask worker local directory for file spilling.
death_timeout [float] Seconds to wait for a scheduler before closing workers

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

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

header_skip [list] Lines to skip in the header. Header lines matching this text will be removed

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. Defaults to the Python that is submitting these jobs

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

name [str] Name of Dask worker. This is typically set by the Cluster

n_workers [int] Number of workers to start by default. Defaults to 0. See the scale method

silence_logs [str] Log level like “debug”, “info”, or “error” to emit here if the scheduler is started locally

asynchronous [bool] Whether or not to run this cluster object with the async/await syntax

security [Security] A dask.distributed security object if you’re using TLS/SSL

dashboard_address [str or int] An address like “:8787” on which to host the Scheduler’s dashboard

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. Each option will be prepended with the #PBS prefix.

Examples

```python
>>> from dask_jobqueue import MoabCluster
>>> cluster = MoabCluster(queue='regular', project="myproj", cores=24,
... memory="500 GB")
>>> cluster.scale(jobs=10) # ask for 10 jobs
```

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

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

```python
>>> cluster.adapt(maximum_jobs=20)
```

```
init__(self, n_workers=0, job_cls:dask_jobqueue.core.Job=None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)
Initialize self. See help(type(self)) for accurate signature.
```
Methods

```python
__init__(self[, n_workers, loop, security, ...]) Initialize self.
adapt(self, *args, minimum_jobs, ...) Scale Dask cluster automatically based on scheduler activity.
close(self[, timeout])
job_script(self)
logs(self[, scheduler, workers]) Return logs for the scheduler and workers
new_worker_spec(self) Return name and spec for the next worker
scale(self[, n, jobs, memory, cores]) Scale cluster to specified configurations.
scale_down(self, workers)
scale_up(self[, n, memory, cores]) Scale cluster to n workers
sync(self, func, *args[, asynchronous, ...])
```

Attributes

```python
asynchronous
config_name
dashboard_link
job_header
job_name
observed
plan
requested
scheduler_address
```

3.9.4 dask_jobqueue.OARCluster

```python
class dask_jobqueue.OARCluster(n_workers=0, job_cls: dask_jobqueue.core.Job = None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)
```

Launch Dask on an 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.
- `cores` [int] Total number of cores per job
- `memory` [str] Total amount of memory per job
- `processes` [int] Cut the job up into this many processes. Good for GIL workloads or for nodes with many cores. By default, `process = sqrt(cores)` so that the number of processes and the number of threads per process is roughly the same.
- `interface` [str] Network interface like ‘eth0’ or ‘ib0’.
- `nanny` [bool] Whether or not to start a nanny process
- `local_directory` [str] Dask worker local directory for file spilling.
**death_timeout** [float] Seconds to wait for a scheduler before closing workers

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

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

**header_skip** [list] Lines to skip in the header. Header lines matching this text will be removed

**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. Defaults to the Python that is submitting these jobs

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

**name** [str] Name of Dask worker. This is typically set by the Cluster

**n_workers** [int] Number of workers to start by default. Defaults to 0. See the scale method

**silence_logs** [str] Log level like “debug”, “info”, or “error” to emit here if the scheduler is started locally

**asynchronous** [bool] Whether or not to run this cluster object with the async/await syntax

**security** [Security] A dask.distributed security object if you’re using TLS/SSL

**dashboard_address** [str or int] An address like “:8787” on which to host the Scheduler’s dashboard

**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.

### Examples

```python
>>> from dask_jobqueue import OARCluster
>>> cluster = OARCluster(queue='regular')
>>> cluster.scale(jobs=10)  # ask for 10 jobs
```

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

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

```python
>>> cluster.adapt(maximum_jobs=20)
```

```python
__init__(self, n_workers=0, job_cls=dask_jobqueue.core.Job=None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=..., host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)
```

Initialize self. See help(type(self)) for accurate signature.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong> (self, n_workers, loop, security, ...)</td>
<td>Initialize self.</td>
</tr>
<tr>
<td>adapt(self, *args, minimum_jobs, ...)</td>
<td>Scale Dask cluster automatically based on scheduler activity.</td>
</tr>
<tr>
<td>close(self[, timeout])</td>
<td></td>
</tr>
<tr>
<td>job_script(self)</td>
<td></td>
</tr>
<tr>
<td>logs(self[, scheduler, workers])</td>
<td>Return logs for the scheduler and workers</td>
</tr>
<tr>
<td>new_worker_spec(self)</td>
<td>Return name and spec for the next worker</td>
</tr>
<tr>
<td>scale(self[, n, jobs, memory, cores])</td>
<td>Scale cluster to specified configurations.</td>
</tr>
<tr>
<td>scale_down(self, workers)</td>
<td></td>
</tr>
<tr>
<td>scale_up(self[, n, memory, cores])</td>
<td>Scale cluster to n workers</td>
</tr>
<tr>
<td>sync(self, func, *args[, asynchronous, ...])</td>
<td></td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous</td>
<td></td>
</tr>
<tr>
<td>config_name</td>
<td></td>
</tr>
<tr>
<td>dashboard_link</td>
<td></td>
</tr>
<tr>
<td>job_header</td>
<td></td>
</tr>
<tr>
<td>job_name</td>
<td></td>
</tr>
<tr>
<td>observed</td>
<td></td>
</tr>
<tr>
<td>plan</td>
<td></td>
</tr>
<tr>
<td>requested</td>
<td></td>
</tr>
<tr>
<td>scheduler_address</td>
<td></td>
</tr>
</tbody>
</table>

3.9.5 dask_jobqueue.PBSCluster

class dask_jobqueue.PBSCluster (n_workers=0, job_cls: dask_jobqueue.core.Job = None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=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.
- **cores** [int] Total number of cores per job
- **memory** [str] Total amount of memory per job
- **processes** [int] Cut the job up into this many processes. Good for GIL workloads or for nodes with many cores. By default, `process = sqrt(cores)` so that the number of processes and the number of threads per process is roughly the same.
- **interface** [str] Network interface like ‘eth0’ or ‘ib0’.
- **nanny** [bool] Whether or not to start a nanny process
- **local_directory** [str] Dask worker local directory for file spilling.
death_timeout [float] Seconds to wait for a scheduler before closing workers
extra [list] Additional arguments to pass to dask-worker
env_extra [list] Other commands to add to script before launching worker.
header_skip [list] Lines to skip in the header. Header lines matching this text will be removed
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. Defaults to the Python that is submitting these jobs
config_name [str] Section to use from jobqueue.yaml configuration file.
name [str] Name of Dask worker. This is typically set by the Cluster
n_workers [int] Number of workers to start by default. Defaults to 0. See the scale method
silence_logs [str] Log level like "debug", "info", or "error" to emit here if the scheduler is started locally
asynchronous [bool] Whether or not to run this cluster object with the async/await syntax
security [Security] A dask.distributed security object if you’re using TLS/SSL
dashboard_address [str or int] An address like "::8787" on which to host the Scheduler’s dashboard
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. Each option will be prepended with the #PBS prefix.

Examples

```python
>>> from dask_jobqueue import PBSCluster
>>> cluster = PBSCluster(queue='regular', project="myproj", cores=24,
... memory="500 GB")
>>> cluster.scale(jobs=10)  # ask for 10 jobs
```

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

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

```python
>>> cluster.adapt(maximum_jobs=20)
```

```python
@init__(self, n_workers=0, job_cls:dask_jobqueue.core.Job=None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)
Initialize self. See help(type(self)) for accurate signature.
```
Methods

```python
__init__(self[, n_workers, loop, security, ...]) Initialize self.
adapt(self, *args, minimum_jobs, ...) Scale Dask cluster automatically based on scheduler activity.
close(self[, timeout])
job_script(self)
logs(self[, scheduler, workers]) Return logs for the scheduler and workers
new_worker_spec(self) Return name and spec for the next worker
scale(self[, n, jobs, memory, cores]) Scale cluster to specified configurations.
scale_down(self, workers)
scale_up(self[, n, memory, cores]) Scale cluster to n workers
sync(self, func, *args[, asynchronous, ...])
```

Attributes

```python
asynchronous
config_name
dashboard_link
job_header
job_name
observed
plan
requested
scheduler_address
```

3.9.6 dask_jobqueue.SGECluster

class dask_jobqueue.SGECluster(n_workers=0, job_cls: dask_jobqueue.core.Job = None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)

Launch Dask on an 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.
- **cores** [int] Total number of cores per job
- **memory** str Total amount of memory per job
processes [int] Cut the job up into this many processes. Good for GIL workloads or for nodes with many cores. By default, \( \text{process} \approx \sqrt{\text{cores}} \) so that the number of processes and the number of threads per process is roughly the same.

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

nanny [bool] Whether or not to start a nanny process

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

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

extra [list] Additional arguments to pass to \textit{dask-worker}

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

header_skip [list] Lines to skip in the header. Header lines matching this text will be removed

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. Defaults to the Python that is submitting these jobs

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

name [str] Name of Dask worker. This is typically set by the Cluster

n_workers [int] Number of workers to start by default. Defaults to 0. See the scale method

silence_logs [str] Log level like “debug”, “info”, or “error” to emit here if the scheduler is started locally

asynchronous [bool] Whether or not to run this cluster object with the async/await syntax

security [Security] A dask.distributed security object if you’re using TLS/SSL

dashboard_address [str or int] An address like “:8787” on which to host the Scheduler’s dashboard

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.

Examples

```python
>>> from dask_jobqueue import SGECluster
>>> cluster = SGECluster(
...     queue='regular',
...     project="myproj",
...     cores=24,
...     memory="500 GB"
... )
>>> cluster.scale(jobs=10)  # ask for 10 jobs

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

This also works with adaptive clusters. This automatically launches and kill workers based on load.
```python
>>> cluster.adapt(maximum_jobs=20)
```

```python
__init__(self, n_workers=0, job_cls=dask_jobqueue.core.Job=None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)
```

Initialize self. See help(type(self)) for accurate signature.

Methods

```python
__init__(self[, n_workers, loop, security, ...]) Initialize self.

adapt(self, *args, minimum_jobs, ...) Scale Dask cluster automatically based on scheduler activity.

close(self[, timeout])

job_script(self)

logs(self[, scheduler, workers]) Return logs for the scheduler and workers

new_worker_spec(self) Return name and spec for the next worker

scale(self[, n, jobs, memory, cores]) Scale cluster to specified configurations.

scale_down(self[, workers])

scale_up(self[, n, memory, cores]) Scale cluster to n workers

sync(self, func, *args[, asynchronous, ...])
```

Attributes

```python
asynchronous
dashboard_link
job_header
job_name
observed
plan
requested
scheduler_address
```

3.9.7 dask_jobqueue.SLURMCluster

```python
class dask_jobqueue.SLURMCluster(n_workers=0, job_cls: dask_jobqueue.core.Job =
None, loop=None, security=None, silence_logs='error',
name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787',
config_name=None, **kwargs)
```

Launch Dask on a SLURM cluster

Parameters

```python
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.

cores [int] Total number of cores per job

memory: str Total amount of memory per job
```
processes [int] Cut the job up into this many processes. Good for GIL workloads or for nodes with many cores. By default, process ~= sqrt(cores) so that the number of processes and the number of threads per process is roughly the same.

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

nanny [bool] Whether or not to start a nanny process

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

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

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

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

header_skip [list] Lines to skip in the header. Header lines matching this text will be removed

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. Defaults to the Python that is submitting these jobs

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

name [str] Name of Dask worker. This is typically set by the Cluster

n_workers [int] Number of workers to start by default. Defaults to 0. See the scale method

silence_logs [str] Log level like “debug”, “info”, or “error” to emit here if the scheduler is started locally

asynchronous [bool] Whether or not to run this cluster object with the async/await syntax

security [Security] A dask.distributed security object if you’re using TLS/SSL

dashboard_address [str or int] An address like “:8787” on which to host the Scheduler’s dashboard

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.

Examples

```python
>>> from dask_jobqueue import SLURMCluster
>>> cluster = SLURMCluster(
...    queue='regular',
...    project='myproj',
...    cores=24,
...    memory="500 GB"
...)

>>> cluster.scale(jobs=10)    # ask for 10 jobs
```
```python
>>> from dask.distributed import Client
>>> client = Client(cluster)
```

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

```python
>>> cluster.adapt(maximum_jobs=20)
```

```python
__init__(self, n_workers=0, job_cls:dask_jobqueue.core.Job=None, loop=None, security=None, silence_logs='error', name=None, asynchronous=False, interface=None, host=None, protocol='tcp://', dashboard_address=':8787', config_name=None, **kwargs)
```

Initialize self. See help(type(self)) for accurate signature.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(self[, n_workers, loop, security, ...])</code></td>
<td>Initialize self.</td>
</tr>
<tr>
<td><code>adapt(self, *args, minimum_jobs,...)</code></td>
<td>Scale Dask cluster automatically based on scheduler activity.</td>
</tr>
<tr>
<td><code>close(self[, timeout])</code></td>
<td></td>
</tr>
<tr>
<td><code>job_script(self)</code></td>
<td></td>
</tr>
<tr>
<td><code>logs(self[, scheduler, workers])</code></td>
<td>Return logs for the scheduler and workers</td>
</tr>
<tr>
<td><code>new_worker_spec(self)</code></td>
<td>Return name and spec for the next worker</td>
</tr>
<tr>
<td><code>scale(self[, n, jobs, memory, cores])</code></td>
<td>Scale cluster to specified configurations.</td>
</tr>
<tr>
<td><code>scale_down(self, workers)</code></td>
<td></td>
</tr>
<tr>
<td><code>scale_up(self[, n, memory, cores])</code></td>
<td>Scale cluster to n workers</td>
</tr>
<tr>
<td><code>sync(self, func, *args[, asynchronous, ...])</code></td>
<td></td>
</tr>
</tbody>
</table>

### Attributes

- `asynchronous`
- `config_name`
- `dashboard_link`
- `job_header`
- `job_name`
- `observed`
- `plan`
- `requested`
- `scheduler_address`
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:

```python
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:

```python
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`. 

3.10. How to debug 37
3.11 Changelog

3.11.1 Development version

- **LSFCluster**: add `use_stdin` to `LSFCluster`. This switches between `bsub < job_script` and `bsub job_script` to launch a LSF job (GH#360).
- **HTCondorCluster**: support older HTCondor versions without `-file` argument (GH#351).
- all cluster classes: processes parameter default has changed. By default, processes \(\approx\) \(\sqrt{\text{cores}}\) so that the number of processes and the number of threads per process is roughly the same. Old default was to use one process and only threads, i.e. `processes=1, threads_per_process=\text{cores}`.
- fix bug (forgotten async def) in `OARCluster._submit_job` (GH#380).

3.11.2 0.7.0 / 2019-10-09

- Base Dask-Jobqueue on top of the core `dask.distributed.SpecCluster` class (GH#307)
  This is nearly complete reimplementation of the dask-jobqueue logic on top of more centralized logic. This improves standardization and adds new features, but does include the following **breaking changes**:
  - The `cluster.start_workers` method has been removed. Use `cluster.scale` instead.
  - The `cluster.stop_all_jobs()` method has been removed. Please use `cluster.scale(0)` instead.
  - The attributes `running_jobs`, `pending_jobs`, and `cancelled_jobs` have been removed. These have been moved upstream to the `dask.distributed.SpecCluster` class instead as `workers` and `worker_spec`, as well as `.plan`, `.requested`, and `.observed`.
  - The name attribute has been moved to `job_name`.
- You can now specify jobs in `.scale` and `.adapt`: for example `cluster.scale(jobs=2)` and `cluster.adapt(minimum_jobs=0, maximum_jobs=10)`. Specifying scaling in terms of jobs is generally more intuitive than in terms of Dask workers. This was part of GH#307.
- Update `.scale()` and `.adapt()` docstrings (GH#346)
- Update interactive docs (GH#340)
- Improve error message when cores or memory is not specified (GH#331)
- Fix Python 3.5.0 support in setup.py (GH#317)

3.11.3 0.6.3 / 2019-08-18

- Compatibility with Dask 2.3.0: add `scheduler_info` from `local_cluster` (GH#313)
- Remove lingering Python 2 specific code (GH#308)
- Remove `__future__` imports since we depend on Python >3.5 (GH#311)
- Remove Python 3 check for black in CI (GH#315)
3.11.4 0.6.2 / 2019-07-31

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

3.11.5 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)

3.11.6 0.6.0 / 2019-07-06

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

3.11.7 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)

3.11.8 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)
3.11.9 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:

```bash
git clone git@github.com:dask/dask-jobqueue.git
```

and install it from source:

```bash
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:

```bash
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:

```bash
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
Symbols

__init__() (dask_jobqueue.HTCondorCluster method), 23
__init__() (dask_jobqueue.LSFCluster method), 25
__init__() (dask_jobqueue.MoabCluster method), 27
__init__() (dask_jobqueue.OARCluster method), 29
__init__() (dask_jobqueue.PBSCluster method), 31
__init__() (dask_jobqueue.SGECluster method), 34
__init__() (dask_jobqueue.SLURMCluster method), 36

H
HTCondorCluster (class in dask_jobqueue), 22

L
LSFCluster (class in dask_jobqueue), 24

M
MoabCluster (class in dask_jobqueue), 26

O
OARCluster (class in dask_jobqueue), 28

P
PBSCluster (class in dask_jobqueue), 30

S
SGECluster (class in dask_jobqueue), 32
SLURMCluster (class in dask_jobqueue), 34