Dask-CHTC

Release 0.1.0

Josh Karpel

Mar 25, 2021

CONTENTS

Dask-CHTC builds on top of Dask-Jobqueue to spawn Dask workers in the CHTC HTCondor pool. It also provides tools for *running Jupyter notebook servers in a controlled way on CHTC submit nodes*, which you may find helpful for providing an interactive development environment to use Dask in.

Note: If you're interested in using Dask at CHTC but have never used CHTC resources before, please fill out the CHTC contact form to get in touch with our Research Computing Facilitators. If you've already had an engagement meeting, send an email to chtc@cs.wisc.edu and let them know you're interested in using Dask.

Attention: We currently only support the Dask-CHTC workflow on the `submit3.chtc.wisc.edu` submit node. If you do not have an account on `submit3.chtc.wisc.edu`, you will need to request one.

Attention: Dask-CHTC is prototype software! If you notice any issues or have any suggestions for improvements, please write up a GitHub issue detailing the problem or proposal. We also recommend "watching" the GitHub repository to keep track of new releases, and upgrading promptly when they occur.

These pages will get you started with Dask-CHTC:

Installing Dask-CHTC How to install Python and Dask-CHTC on a CHTC submit node.

Running Jupyter through Dask-CHTC How to use Dask-CHTC to run a Jupyter notebook server on a CHTC submit node.

Networking and Port Forwarding Information on CHTC networking and how to forward ports over SSH, which will allow you to connect to Jupyter notebooks and Dask dashboards running on CHTC submit nodes.

Dask Cluster Creation A brief example Jupyter notebook, showing how to start up a `CHTCCluster` and use it to perform some calculations.

These pages have information for troubleshooting problems and handling specific use cases:

Troubleshooting Solutions and advice for tackling specific problems that might arise while using Dask-CHTC.

Configuration Information on how to configure Dask-CHTC and the Dask JupyterLab extension.

Building Docker Images for Dask-CHTC Information on how to build Docker images for use with Dask-CHTC.

Detailed information on the Python API and the associated command line tool can be found on these pages:

API Reference API documentation for `dask_chtc`.

CLI Reference Documentation for the `dask-chtc` CLI tool.

CHAPTER

ONE

INSTALLING DASK-CHTC

Dask-CHTC is a Python package that allows you to run Dask clusters and Jupyter notebook servers on CHTC submit nodes. To run Dask on CHTC, you will need to

1. Install a "personal" Python on a CHTC submit node.
2. Install whatever other packages you want, like `numpy`, `matplotlib`, `dask-ml`, or `jupyterlab`.
3. Install `dask-chtc` itself.

Attention: We currently only support the Dask-CHTC workflow on the `submit3.chtc.wisc.edu` submit node. If you do not have an account on `submit3.chtc.wisc.edu`, you will need to request one.

1.1 Install a Personal Python

You will need a Python installation on a CHTC submit node to run your code. You will be able to manage packages in this Python installation just like you would on your local machine, without needing to work with the CHTC system administrators.

Since you do not have permission to install packages in the "system" Python on CHTC machines (and since you should never do that anyway), you will need to make a "personal" Python installation. We **highly recommend** doing this using Miniconda, a minimal installation of Python and the `conda` package manager.

To create a Miniconda installation, first log in to a CHTC submit node (via `ssh`). Then, download the latest version of the Miniconda installer using `wget`:

```
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
```

Run the installer using `bash`:

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```

The installer will ask you to accept a license agreement, and then ask you several questions about how to install Miniconda itself. We recommend that you **do** "initialize Miniconda3 by running conda init" when prompted; this will cause Python you just installed to be your default Python in future shell sessions (instead of the system Python).

You may need to begin a new shell session for the commands in the next sections to work as expected. To check that everything is hooked up correctly, try running these commands:

```
$ which python
$ which conda
$ which pip
```

They should all resolve to paths inside the Miniconda installation you just created.

1.2 Install Packages

Your personal Python installation will be used to run all of your code, so you will need to install any packages that you depend on, either inside your code (like `numpy`) or to do things like run Jupyter notebooks (provided by the `jupyter` package).

To install packages in your new personal Python installation, use the `conda install` command. For example, to install `numpy`, run

```
$ conda install numpy
```

You may occasionally need a package that isn't in the standard Anaconda channel. Many more packages, and more up-to-date versions of packages, are often available in the community-created conda-forge channel channel. For example, to install Dask from `conda-forge` instead of the default channel, you would run

```
$ conda install -c conda-forge dask
```

where `-c` is short for `--channel`.

Some packages are provided by other channels. For example, PyTorch asks you to install from their own `conda` channel:

```
$ conda install -c pytorch pytorch
```

`conda` is mostly compatible with `pip`; if a package is not available via `conda` at all, you can install it with `pip` as usual.

1.3 Install Dask-CHTC

Attention: These instructions will change in the future as Dask-CHTC stabilizes.

To install Dask-CHTC itself, run

```
$ pip install --upgrade git+https://github.com/CHTC/dask-chtc.git
```

To check that installation worked correctly, try running

```
$ dask-chtc --version
Dask-CHTC version x.y.z
```

If you don't see the version message or some error occurs, try re-installing. If that fails, please let us know.

1.4 What's Next?

If you like working inside a Jupyter environment, you should read the next two pages: jupyter and *Networking and Port Forwarding*.

If you are going to run Dask non-interactively (i.e., through a normal Python script, not a notebook), then you're almost ready to go. Pull the `CHTCCluster` and Dask client creation code from *Dask Cluster Creation* and start computing! This pages assumes that Jupyter is available. For more details on connecting to Jupyter, see *the guide on port forwarding*. This page will only detail how to start Jupyter.

Warning: **Do not** run Jupyter notebook servers on CHTC submit nodes except through the process described on this page.

Launching Jupyter with this process allows the CHTC admins to effectively monitor resource usage in the Jupyter process and debug/info messages to be more easily displayed. If you persist the Jupyter session through other means like tmux, the CHTC admins may kill your entire tmux session if it consumes too much CPU and memory.

CHAPTER TWO

RUNNING JUPYTER THROUGH DASK-CHTC

Attention: You may want to interact with your Dask cluster through a Jupyter notebook. Dask-CHTC provides a way to run a Jupyter notebook server on a CHTC submit node.

Warning: Jupyter must be installed, which amounts to running `conda install jupyterlab` or adding *jupyter* to your `environment.yml` file. For more detail, see the Jupyter install documentation,

You can run a notebook server via the Dask-CHTC command line tool, via the `jupyter` subcommand. The command line tool will run the notebook server as an HTCondor job. To see the detailed documentation for this subcommand, run

```
$ dask-chtc jupyter --help
Usage: dask-chtc jupyter [OPTIONS] COMMAND [ARGS]...

    [... long description cut ...]

    Commands:
      start   Start a Jupyter notebook server as a persistent HTCondor job.
      run     Run a Jupyter notebook server as an HTCondor job.
      status  Get information about your running Jupyter notebook server.
      stop    Stop a Jupyter notebook server that was started via "start".
```

The four sub-sub-commands of `dask-chtc jupyter` (`run`, `start`, `status`, and `stop`) let us run and interact with a Jupyter notebook server. You can run `dask-chtc jupyter <subcommand> --help` to get detailed documentation on each of them, but for now, let's try out the `run` subcommand.

2.1 Using the `run` subcommand

The `run` subcommand is the simplest way to launch a Jupyter notebook server. It is designed to mimic the behavior of running a Jupyter notebook server on your local machine. Any command line arguments you pass to it will be passed to the actual `jupyter` command line tool.

Jupyter Lab instances are normally started with `jupyter lab`. The equivalent command for Dask-CHTC is `dask-chtc jupyter run lab`:

```
$ dask-chtc jupyter run lab
000 (7858010.000.000) 2020-07-13 10:38:46 Job submitted from host: <128.104.100.44:
↪9618?addrs=128.104.100.44-9618+[2607-f388-107c-501-92e2-baff-fe2c-2724]-9618&
↪alias=submit3.chtc.wisc.edu&noUDP&sock=schedd_4216_675f>
```

(continues on next page)

(continued from previous page)

```
001 (7858010.000.000) 2020-07-13 10:38:47 Job executing on host: <128.104.100.44:9618?
↪addrs=128.104.100.44-9618+[2607-f388-107c-501-92e2-baff-fe2c-2724]-9618&
↪alias=submit3.chtc.wisc.edu&noUDP&sock=starter_5948_a76b_2712469>
[... Jupyter startup logs cut ...]
[I 10:38:51.582 LabApp] Use Control-C to stop this server and shut down all kernels␣
↪(twice to skip confirmation).
[C 10:38:51.587 LabApp]

    To access the notebook, open this file in a browser:
        file:///home/karpel/.local/share/jupyter/runtime/nbserver-2187556-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=fedee94f539b0beea492bb358d549ed79025b714f3b308c4
     or http://127.0.0.1:8888/?token=fedee94f539b0beea492bb358d549ed79025b714f3b308c4
```

Dask-CHTC mixes HTCondor job diagnostic information into the normal Jupyter output stream. These messages may be helpful if your notebook server job is unexpectedly interrupted.

Just like running `jupyter lab`, if you press Control-C, the notebook server will be stopped:

```
^C
[C 10:40:35.962 LabApp] received signal 15, stopping
[I 10:40:35.963 LabApp] Shutting down 0 kernels
004 (7858010.000.000) 2020-07-13 10:40:36 Job was evicted.
    (0) CPU times
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
        Usr 0 00:00:01, Sys 0 00:00:00  -  Run Local Usage
    0  -  Run Bytes Sent By Job
    0  -  Run Bytes Received By Job
009 (7858010.000.000) 2020-07-13 10:40:36 Job was aborted.
    Shut down Jupyter notebook server (by user karpel)
```

You can think of this notebook server as being tied to your `ssh` session. If your `ssh` session disconnects (either because you quit manually, or because it timed out, or because you closed your laptop, or any number of other possible reasons) **your notebook server will also stop**. The next section will discuss how to run your notebook server in a more persistent manner.

2.2 Using the `start`, `status`, and `stop` subcommands

The `start` subcommand is similar to the `run` subcommand, except that if you end the command by Control-C or your terminal session ending, **the notebook server will not be stopped**. The command will still "take over" your terminal, echoing log messages just like the `run` subcommand did:

```
$ dask-chtc jupyter start lab
000 (7858021.000.000) 2020-07-13 10:52:51 Job submitted from host: <128.104.100.44:
↪9618?addrs=128.104.100.44-9618+[2607-f388-107c-501-92e2-baff-fe2c-2724]-9618&
↪alias=submit3.chtc.wisc.edu&noUDP&sock=schedd_4216_675f>
001 (7858021.000.000) 2020-07-13 10:52:51 Job executing on host: <128.104.100.44:9618?
↪addrs=128.104.100.44-9618+[2607-f388-107c-501-92e2-baff-fe2c-2724]-9618&
↪alias=submit3.chtc.wisc.edu&noUDP&sock=starter_5948_a76b_2713469>
[... Jupyter startup logs cut ...]
[I 10:52:56.060 LabApp] Use Control-C to stop this server and shut down all kernels␣
↪(twice to skip confirmation).
[C 10:52:56.066 LabApp]
```

(continues on next page)

(continued from previous page)

```
    To access the notebook, open this file in a browser:
        file:///home/karpel/.local/share/jupyter/runtime/nbserver-2209285-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3342f18a95d7d61c51a2b8cf80b836e932ac53f9ebdb3965
     or http://127.0.0.1:8888/?token=3342f18a95d7d61c51a2b8cf80b836e932ac53f9ebdb3965
^C
```

Even though we pressed Control-C, the notebook server will still be running. We can look at the status of our notebook server job using the `status` subcommand, which will show us various diagnostic information on both the Jupyter notebook server and the HTCondor job it is running inside:

```
$ dask-chtc jupyter status
 RUNNING  jupyter lab
├─ Contact Address: http://127.0.0.1:8888/?
↪token=3342f18a95d7d61c51a2b8cf80b836e932ac53f9ebdb3965
├─ Python Executable: /home/karpel/.python/envs/dask-chtc/bin/python3.7
├─ Working Directory:  /home/karpel/dask-chtc
├─ Job ID: 7858021.0
├─ Last status change at:  2020-07-13 15:52:51+00:00 UTC (4 minutes ago)
├─ Originally started at: 2020-07-13 15:52:51+00:00 UTC (4 minutes ago)
├─ Output: /home/karpel/.dask-chtc/jupyter-logs/current.out
├─ Error:  /home/karpel/.dask-chtc/jupyter-logs/current.err
└─ Events: /home/karpel/.dask-chtc/jupyter-logs/current.events
```

This may be particularly useful for recovering the contact address of a notebook server that you started running in a previous `ssh` session.

To stop your notebook server, run

```
$ dask-chtc jupyter stop
[C 11:02:57.820 LabApp] received signal 15, stopping
[I 11:02:57.821 LabApp] Shutting down 0 kernels
004 (7858021.000.000) 2020-07-13 11:02:58 Job was evicted.
    (0) CPU times
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
        Usr 0 00:00:01, Sys 0 00:00:00  -  Run Local Usage
    0  -  Run Bytes Sent By Job
    0  -  Run Bytes Received By Job
009 (7858021.000.000) 2020-07-13 11:02:58 Job was aborted.
    Shut down Jupyter notebook server (by user karpel)
```

2.3 What's Next?

Once you're able to *connect to your Jupyter notebook server*, you should move on to *Dask Cluster Creation* to learn how to create a `CHTCCluster`.

CHAPTER

THREE

NETWORKING AND PORT FORWARDING

For security reasons, most ports on CHTC submit and execute nodes are not open for traffic. This means that programs that need to communicate over ports, like the Dask distributed scheduler or a Jupyter notebook server, will not be able to communicate with your computer, or possibly even with other computers inside the CHTC pool, over any given set of ports.

3.1 Port Forwarding for Jupyter Notebook Servers on Submit Nodes

A Jupyter notebook server is, essentially, a web application. For example, when you run `jupyter lab` on your local machine, you are starting up a web server that listens for internet connections on a particular port. You may recall seeing a message that looks like this during startup:

```
[I 10:52:56.060 LabApp] The Jupyter Notebook is running at:
[I 10:52:56.060 LabApp] http://localhost:8888/?
↪token=3342f18a95d7d61c51a2b8cf80b836e932ac53f9ebdb3965
[I 10:52:56.060 LabApp]  or http://127.0.0.1:8888/?
↪token=3342f18a95d7d61c51a2b8cf80b836e932ac53f9ebdb3965
```

You typically visit one of those addresses using your web browser to connect the JavaScript-based "frontend" interface to the notebook server "backend". The `/?token=...` part of each address is an authorization token; it prevents anyone who doesn't have it from actually running any code on your notebook server. The actual addresses are `http://localhost:8888` and `http://127.0.0.1:8888` The part before the second `:` is the address of the machine (like in a normal website address), except that in this case they are both special addresses which "loopback" on the machine itself. The number after the second `:` is the port number to talk to on the machine. So both of these addresses are variations on "talk to port 8888 on myself".

When you *run a Jupyter notebook server on a CHTC submit node*, you'll get the same kinds of addresses, but you won't be able to connect to them from the web browser on your local machine: the addresses mean "talk to myself", but "myself" is the submit node, not your local machine.

To work around this issue, you can "forward" a port from the submit machine back to your local machine using `ssh`. A port on your machine and a port on the submit machine will be "tied together" over your existing SSH connection. Connecting to that port on your local machine (the "local" port) will effectively connect to the target port of the submit machine (the "remote" port).

There are two ways to forward ports using `ssh`, depending on when you know which ports you want to forward. If you know the local and remote port numbers ahead of time, you can specify port forwarding using the `-L` argument of `ssh`:

```
$ ssh -L localhost:3000:localhost:4000 <user@hostname>
```

That command would connect local port 3000 to remote port 4000. A Jupyter notebook running on port 4000 on the remote machine, like so:

```
$ dask-chtc jupyter run lab --port 4000
[... Jupyter startup logs cut ...]
[I 13:06:41.784 LabApp] The Jupyter Notebook is running at:
[I 13:06:41.784 LabApp] http://localhost:4000/?
↪token=1186ba8ed4248f58338c48e3c016e192eb43f9c8d470e37d
[I 13:06:41.784 LabApp]  or http://127.0.0.1:4000/?
↪token=1186ba8ed4248f58338c48e3c016e192eb43f9c8d470e37d
```

Could be reached from a web browser running on your computer by going to `http://localhost:3000`. For simplicity, we recommend using the same port number for the local and remote ports – then you can just copy-paste the address from the Juypter logs!

If you don't know the port number ahead of time (perhaps the remote port you wanted to use is already in use, and the Jupyter notebook server actually starts up on port 4001), you can forward a port from an existing SSH session by opening the "SSH command line". From the terminal, inside the SSH session, type `~C` (i.e., hold shift and press the ~ key, release shift, then hold shift again and press the `C` key). Your prompt should change to

```
ssh>
```

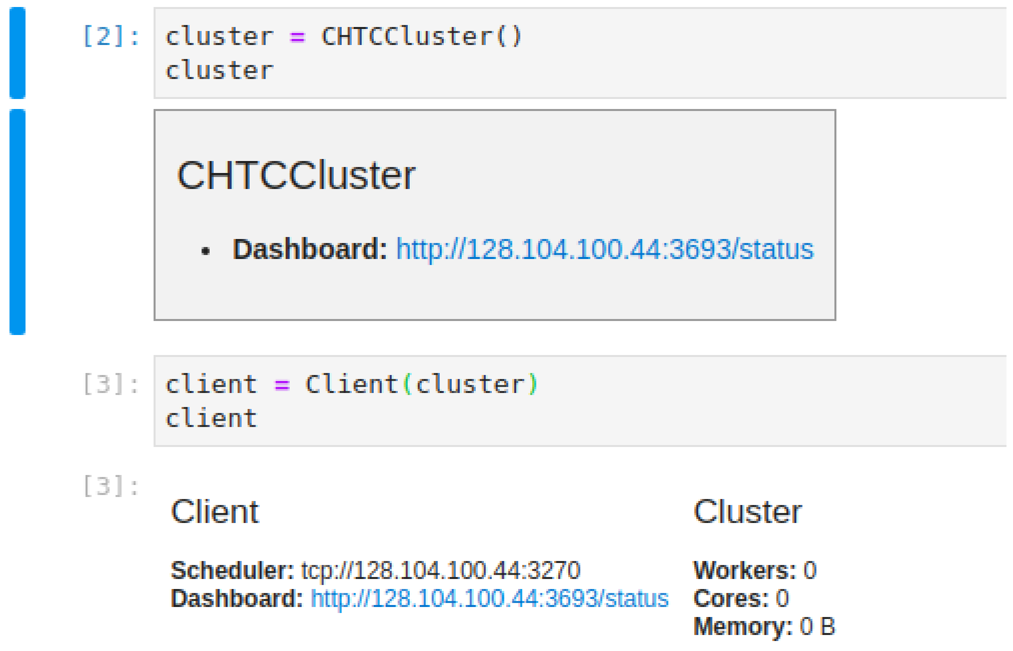
In this prompt, enter a `-L` argument like above and press enter:

```
ssh> -L localhost:3001:localhost:4001
Forwarding port.
```

Press enter again to return to your normal terminal prompt. The port is now forwarded, as if you had added the `-L` argument to your original `ssh` command.

3.2 Forwarding a Port for the Dask Dashboard

The Dask scheduler exposes a dashboard as a web application. If you are using Dask through Jupyter, the dashboard address will be shown in the representations of both the `Cluster` and `Client`:



```
[2]: cluster = CHTCCluster()
     cluster
```

CHTCCluster

- **Dashboard:** http://128.104.100.44:3693/status

```
[3]: client = Client(cluster)
     client
```

[3]:

Client

Scheduler: tcp://128.104.100.44:3270
Dashboard: http://128.104.100.44:3693/status

Cluster

Workers: 0
Cores: 0
Memory: 0 B

Programmatically, the address is available in `client.scheduler_info()['services']`.

Be wary: Dask is showing an "external" address that would be appropriate for a setup without security firewalls. Instead of connecting to that address, you should point your web browser (or the Dask Jupyterlab extension, for example) to something like `localhost:<port>/status`, after forwarding the remote port that the dashboard is hosted on to some local port.

3.3 Dask Scheduler and Worker Internal Networking

The Dask scheduler and workers all need to talk to each bidirectionally. This is handled internally by Dask-CHTC, and you shouldn't have to do anything about it. Please let us know if you run into any issues you believe are caused by internal networking failures.

3.4 What's Next?

Now that you can connect to your Jupyter notebook server, you should move on to *Dask Cluster Creation* to learn how to create a *`CHTCCluster`*.

CHAPTER

FOUR

DASK CLUSTER CREATION

Dask-CHTC's primary job is to provide `CHTCCluster`, an object that manages a pool of Dask workers on the CHTC pool on your behalf. To start computing with Dask itself, all we need to do is connect our `CHTCCluster` to a standard `dask.distributed Client`. This notebook shows you how to do that.

If you are reading the notebook in the documentation, you can download a copy of this notebook to run on the CHTC pool yourself by clicking on "View page source" link in the top right, then saving the raw notebook as `example.ipynb` and uploading it via the Jupyter interface.

If you are running this notebook live using JupyterLab, we recommend installing the Dask JupyterLab Extension. We'll point out what dashboard address to point it to later in the notebook.

4.1 Create Cluster and Client

```
[1]: from dask_chtc import CHTCCluster
     from dask.distributed import Client
```

```
[2]: cluster = CHTCCluster()
     cluster
```

```
CHTCCluster('tcp://128.104.100.44:3388', workers=0, threads=0, memory=0 B)
```

```
[3]: client = Client(cluster)
     client
```

```
[3]: <Client: 'tcp://128.104.100.44:3388' processes=0 threads=0, memory=0 B>
```

(If you are running this notebook live, you now have enough information to hook up the Dask JupyterLab extension – make sure you forward the dashboard port over SSH, then point the extension at `locahost:<local port>`.)

4.2 Scale the Cluster

There are two ways to ask the cluster to get workers: `Cluster.scale` and `Cluster.adapt`. `Cluster.scale` submits requests for a certain number of workers exactly one time. This is not sufficiently flexible for use in the CHTC pool, where your workers may come-and-go unexpectedly, for reasons outside of your control. **We recommend always using** `Cluster.adapt`, which lets you set a minimum and maximum number of workers (or amount of cores/memory/etc.) to keep in your Dask worker pool. It will dynamically submit extra worker requests as necessary to meet the minimum, or to meet increased demand by your workflow.

Warning: a bug in `Cluster.adapt` may cause work to be lost when scaling down workers because of reduced demand. Until the relevant issue is fixed, either use `Cluster.scale` or use `Cluster.adapt` with the minimum the same as the maximum.

Let's ask for at least 10, and up to 20 workers:

```
[4]: cluster.adapt(minimum=10, maximum=20);
```

It will likely take a few minutes for the workers to show up in your pool. They must first find some available resource, then download the Docker image they will run inside.

4.3 Do Some Calculations

Let's flex our cluster by doing a small calculation: we'll create a Dask array of `1`s and add it to its own transpose.

```
[5]: import dask.array as da
```

```
[6]: x = da.ones((15, 15), chunks = 5)
     x
```

```
[6]: dask.array<ones, shape=(15, 15), dtype=float64, chunksize=(5, 5), chunktype=numpy.
     ↪ndarray>
```

Now we'll add `x` to its own transpose. Note that Dask doesn't actually do anything yet; it is just building up a task graph representing the computation.

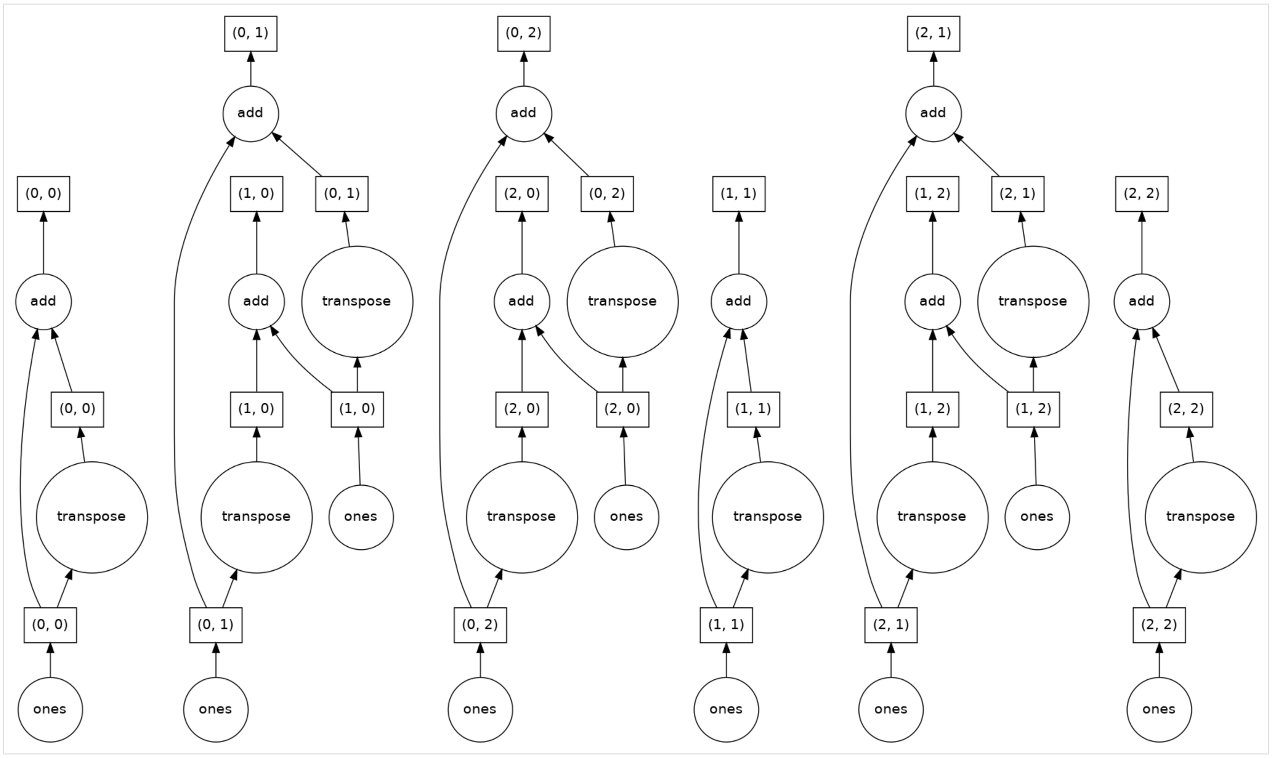
```
[7]: y = x + x.T
     y
```

```
[7]: dask.array<add, shape=(15, 15), dtype=float64, chunksize=(5, 5), chunktype=numpy.
     ↪ndarray>
```

We can visualize the task graph that Dask will execute to compute `y` by calling the `visualize` method. If this doesn't work for you, you may need to install Graphviz and the Python wrapper for it: `conda install python-graphviz`.

```
[8]: y.visualize()
```

[8]:



We can execute the computation by calling the `compute` method on `y`:

[9]:
```
z = y.compute()
z
```

[9]:
```
array([[2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.]])
```

4.4 What's Next?

1. Learn more about how to use Dask for parallel computing. For example, you could go through the Dask tutorial, or read about how to use Dask-ML for hyperparameter search.
2. Check out the various *configuration options* available to you on the `CHTCCluster`. You can pass arguments to it to define what kind of workers to request. For example, you can set how much memory they should have, which Docker image they should run in, or set them to request GPUs.
3. If you have extra requirements beyond what is in the default Docker image, consider *building your own Docker image*.

CHAPTER

FIVE

TROUBLESHOOTING

5.1 Dask

5.1.1 unsupported pickle protocol: 5

If you get an error with the reason `unsupported pickle protocol: 5`, like

```
distributed.protocol.core - CRITICAL - Failed to deserialize
Traceback (most recent call last):
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/core.
↪py", line 130, in loads
    value = _deserialize(head, fs, deserializers=deserializers)
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/
↪serialize.py", line 302, in deserialize
    return loads(header, frames)
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/
↪serialize.py", line 64, in pickle_loads
    return pickle.loads(x, buffers=buffers)
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/
↪pickle.py", line 75, in loads
    return pickle.loads(x)
ValueError: unsupported pickle protocol: 5
distributed.utils - ERROR - unsupported pickle protocol: 5
Traceback (most recent call last):
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/utils.py",␣
↪line 656, in log_errors
    yield
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/client.py",␣
↪line 1221, in _handle_report
    msgs = await self.scheduler_comm.comm.read()
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/comm/tcp.py",␣
↪line 206, in read
    allow_offload=self.allow_offload,
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/comm/utils.py
↪", line 87, in from_frames
    res = _from_frames()
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/comm/utils.py
↪", line 66, in _from_frames
    frames, deserialize=deserialize, deserializers=deserializers
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/core.
↪py", line 130, in loads
    value = _deserialize(head, fs, deserializers=deserializers)
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/
↪serialize.py", line 302, in deserialize
```

(continues on next page)

(continued from previous page)

```
    return loads(header, frames)
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/
↪serialize.py", line 64, in pickle_loads
    return pickle.loads(x, buffers=buffers)
  File "/home/karpel/miniconda3/lib/python3.7/site-packages/distributed/protocol/
↪pickle.py", line 75, in loads
    return pickle.loads(x)
ValueError: unsupported pickle protocol: 5
```

You are encountering an issue with mismatched Python versions between your Dask client and the workers. Python 3.8 introduced a new default protocol for Python's `pickle` module, which Dask uses to move some kinds of data around. In general, **you should always make sure that your Python versions match**. For this specific issue, you just need to make sure that you are using Python 3.7 or less (or Python 3.8 or greater) for both the Dask client and the workers.

5.2 Jupyter

5.2.1 Jupyter notebook server is stuck in the `REMOVED` state

If something goes wrong during a normal `dask-chtc jupyter stop`, you may find that your notebook server will refuse to shut down. The notebook server status will get stuck in `REMOVED`, like this:

```
$ dask-chtc jupyter status
 REMOVED  jupyter lab
├─ Contact Address: http://127.0.0.1:8888/?
↪token=d1717bce73ebc0e54ebeb16eeeef70811ead8eaae23e213c
├─ Python Executable: /home/karpel/miniconda3/bin/python
├─ Working Directory:  /home/karpel
├─ Job ID: 8138911.0
├─ Last status change at:  2020-07-19 21:34:02+00:00 UTC (23 minutes ago)
├─ Originally started at: 2020-07-19 18:57:07+00:00 UTC (3 hours ago)
├─ Output: /home/karpel/.dask-chtc/jupyter-logs/current.out
├─ Error:  /home/karpel/.dask-chtc/jupyter-logs/current.err
└─ Events: /home/karpel/.dask-chtc/jupyter-logs/current.events
```

Because you can only run one notebook server at a time, this will prevent you from launching a new notebook server. To resolve this issue, you should run `dask-chtc jupyter stop --force`:

```
$ dask-chtc jupyter stop --force
000 (16453.000.000) 2020-07-21 11:58:25 Job submitted from host: <10.0.1.43:40415?
↪addrs=10.0.1.43-40415+[2600-6c44-1180-1661-99fa-fc04-10e3-fd8d]-40415&alias=JKARPEL&
↪noUDP&sock=schedd_20423_5f31>
001 (16453.000.000) 2020-07-21 11:58:27 Job executing on host: <10.0.1.43:40415?
↪addrs=10.0.1.43-40415+[2600-6c44-1180-1661-99fa-fc04-10e3-fd8d]-40415&alias=JKARPEL&
↪noUDP&sock=starter_20464_7d39_11>
005 (16453.000.000) 2020-07-21 11:58:30 Job terminated.
    (0) Abnormal termination (signal 9)
    (0) No core file
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Total Remote Usage
        Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
    0  -  Run Bytes Sent By Job
```

(continues on next page)

(continued from previous page)

```
    0  -  Run Bytes Received By Job
    0  -  Total Bytes Sent By Job
    0  -  Total Bytes Received By Job
```

Always try stopping your notebook server with a plain `stop` command before trying `stop --force`; `--force` does not give the notebook server a chance to shut down cleanly, so your Jupyter kernels may be interrupted while in the middle of an operation.

CHAPTER SIX

CONFIGURATION

Dask-CHTC uses Dask's configuration system for most configuration needs. Dask stores configuration files in YAML format in the directory `~/.config/dask` (where ~ means "your home directory"). Any YAML files in this directory will be read by Dask when it starts up and integrated into its runtime configuration.

6.1 Configuring Dask-CHTC

Dask-CHTC's *`CHTCCluster`* is a type of Dask-Jobqueue cluster, so it is configured through Dask-Jobqueue's configuration system.

This is the default configuration file included with Dask-CHTC:

```
jobqueue:
  chtc:
    # The internal name prefix for the Dask workers
    name: dask-worker

    # The HTCondor JobBatchName for the worker jobs.
    batch-name: dask-worker

    # Worker job resource requests and other options.
    cores: 1                 # Number of cores per worker job
    gpus: null               # Number of GPUs per worker job
    memory: "2 GiB"          # Amount of memory per worker job
    disk: "10 GiB"           # Amount of disk per worker job
    processes: null          # Number of Python processes per worker (null lets Dask␣
↪decide)

    # Whether to use GPULab machines.
    gpu-lab: false

    # What Docker image to use for the Dask worker jobs.
    worker-image: "daskdev/dask:latest"

    # Send HTCondor job log files to this directory
    log-directory: null

    # Extra command line arguments for the Dask worker.
    extra: []

    # Extra environment variables for the Dask worker.
    env-extra: []
```

(continues on next page)

(continued from previous page)

```
    # Extra submit descriptors; not all are available because some are used␣
↪internally.
    job-extra: {}

    # Extra options for the Dask scheduler
    scheduler-options: {}

    # Number of seconds to die after if the worker can not find a scheduler.
    death-timeout: 60

    # INTERNAL OPTIONS BELOW
    # You probably don't need to change these!

    # Directory to spill extra worker memory to (null lets Dask decide)
    local-directory: null

    # Controls the shebang of the job submit file that jobqueue will generate.
    shebang: "#!/usr/bin/env condor_submit"

    # Networking options.
    interface: null
```

A copy of this file (with everything commented out) will be placed in `~/.config/dask/jobqueue-chtc.yaml` the first time you run Dask-CHTC. Options found in that file are used as defaults for the runtime arguments of *`CHTCCluster`* and its parent classes in Dask-Jobqueue, starting with `dask_jobqueue.HTCondorCluster`. You can override any of them at runtime by passing different arguments to the *`CHTCCluster`* constructor.

Dask-CHTC provides a command line tool to help inspect and edit its configuration file. For full details, run `dask-chtc config --help`. The subcommands of `dask-chtc config` will (among other things) let you show the contents of the configuration file, open it in your editor, and reset it to the package defaults.

Warning: Dask-CHTC is prototype software, and the names and meanings of configuration options are not necessarily stable. Be prepared to reset your configuration to track changes in Dask-CHTC!

6.2 Configuring the Dask JupyterLab Extension

The Dask JupyterLab extension lets you view the Dask scheduler's dashboard as part of your JupyterLab. It can also be used to launch a Dask cluster. To configure the cluster that it launches, you write a Dask configuration file, typically stored at `~/.config/dask/labextension.yaml`. Here is an minimal configuration file for launching a *`CHTCCluster`*:

```
labextension:
  factory:
    module: 'dask_chtc'
    class: 'CHTCCluster'
    kwargs: {}
  default:
    workers: null
    adapt: null
```

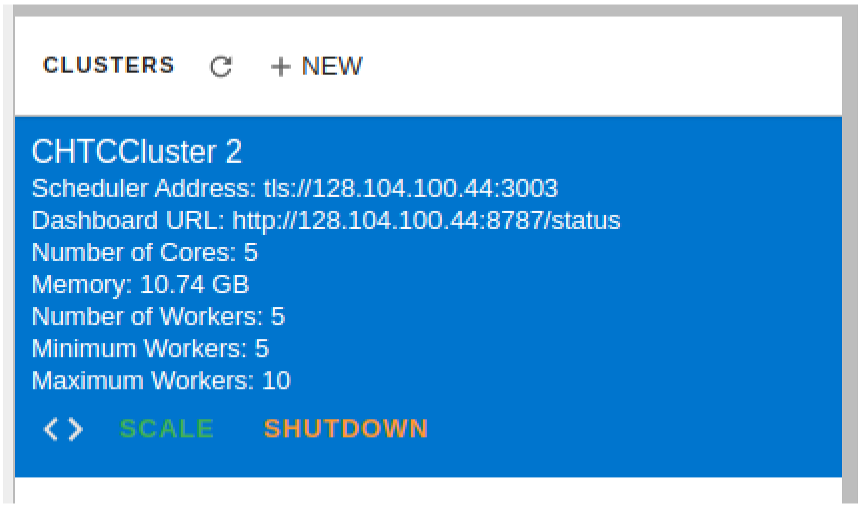
Configuration options set via `~/.config/dask/jobqueue-chtc.yaml` will be honored by the JupyterLab extension; note that you are specifying arguments in the extension configuration file as if you were calling the *`CHTCCluster`* constructor directly.

To connect to the cluster created by the lab extension, you must pass the appropriate security options through. First, get the security options:

```
from dask_chtc import CHTCCluster

sec = CHTCCluster.security()
```

Then, (after creating a new cluster by clicking `+NEW`), click the <> button to insert a cell with the right cluster address:



CHTCCluster 2
Scheduler Address: tls://128.104.100.44:3003
Dashboard URL: http://128.104.100.44:8787/status
Number of Cores: 5
Memory: 10.74 GB
Number of Workers: 5
Minimum Workers: 5
Maximum Workers: 10

And modify it to use the security options by adding the `security` keyword argument:

```
from dask.distributed import Client

client = Client("tls://128.104.100.44:3003", security=sec)
client
```

CHAPTER

SEVEN

BUILDING DOCKER IMAGES FOR DASK-CHTC

Dask-CHTC runs all Dask workers inside Docker containers, which are built from Docker images. This guide won't cover how to build Docker images; innumerable tutorials are available on the CHTC website and the wider internet, and the actual Docker docs are usually useful. Our focus will be on the specific requirements for images for use with Dask-CHTC.

The main requirements are:

1. `dask` needs to be installed in your image so that it can run a Dask worker. You'll also want to make sure various associated libraries like `lz4` and `blosc` (named `python-blosc` in `conda`) are installed. You'll get warnings when the workers start for missing libraries or version mismatches in these associated libraries; we recommend making sure they are all resolved.
2. Any library you use in your application must also be available on the workers. For example, if you `import foobar` in your code, the `foobar` package must be available for import for the client as well as all of the workers. (You can be a little more minimal than this, but it's not worth it – just make sure everything is installed.)
3. Any image you use **must** have a `tini` entrypoint (see their README for details on what `tini` does). This ensures that the HTCondor job that your Dask worker is running in is able to shut down cleanly when the Dask client orders it to stop. If you don't do this, you may notice "zombie" workers that remain alive even after being told to stop, either by the Dask cluster itself or by "brute force" stopping them with `condor_rm`.

A few other considerations to keep in mind:

- Images must be pushed to Docker Hub for HTCondor to use them. If you push your image to `yourname/repository:tag`, then you should set `worker_image = "yourname/repository:tag"` in your *`CHTCCluster`*.
- Always use an explicit tag; do not rely on `latest`. HTCondor caches Docker images by tag, not by SHA, so if you change your image without changing its tag, you may get an older version of your image if your worker lands on an HTCondor slot that you have used in the recent past.
- Minimize the size of your Docker images. Although workers with small resource requests will likely find a slot in under a minute, it may take several minutes to download a large Docker image. Most useful base images are already a few GB, so try to keep the final image size under 5 GB.

7.1 Images for CPU Workers

Docker images for Dask workers that don't need to use GPUs are mostly the same as normal Docker images. Dask provides a nice image which you can use directly or build off of (the default image for Dask-CHTC is `daskdev/dask:latest`).

Here's an example `Dockerfile` that installs some extra `conda` packages on top of `daskdev/dask`:

```
# Inherit from a Dask image. Make sure to use a specific tag, but not
# necessarily this one - it's good to keep up to date!
FROM daskdev/dask:2.20.0

# Install various extra Python packages.
RUN : \
 && conda install --yes \
    xarray \
    dask-ml \
 && conda clean --yes --all \
 && :
```

Note: The trick used in the long `RUN` statement:

```
RUN : \
 && ... \
 && :
```

is to help keep your diffs clean. `:` is a no-op command in `bash`. Try it out!

7.2 Images for GPU Workers

If you want your workers to use GPUs, **you must use a Docker image that inherits from an NVIDIA CUDA image** (their Docker Hub page). If you don't inherit from this image, your Dask worker will not be able to use GPUs even if it lands on a HTCondor slot that has one (the image works in concert with a special distribution of the Docker daemon itself published by NVIDIA that CHTC runs on its GPU nodes).

You could inherit from one of those images yourself, or inherit from an image that itself inherits from `nvidia/cuda`. For example, the PyTorch Docker images inherit from the NVIDIA images, so you could use them as your base image.

Here's an example `Dockerfile` that builds off the PyTorch image by installing Dask-ML and Skorch:

```
# Inherit from a PyTorch image. Make sure to use a specific tag, but not
# necessarily this one - it's good to keep up to date!
FROM pytorch/pytorch:1.5.1-cuda10.1-cudnn7-runtime

# Install various extra Python packages.
RUN : \
 && conda install --yes \
    dask \
    dask-ml \
    lz4 \
    python-blosc \
    tini \
 && conda install --yes \
```

(continues on next page)

(continued from previous page)

```
    -c conda-forge \
    skorch \
 && conda clean --yes --all \
 && :

# Always run under tini!
# See https://github.com/krallin/tini if you want to know why.
# (The daskdev/dask image used above already does this.)
ENTRYPOINT ["tini", "--"]
```

CHAPTER

EIGHT

API REFERENCE

class dask_chtc.**CHTCCluster**(**, worker_image=None, gpu_lab=False, gpus=None, batch_name=None, python='./entrypoint.sh python3', **kwargs*)

A customized `dask_jobqueue.HTCondorCluster` subclass for spawning Dask workers in the CHTC HTCondor pool.

It provides a variety of custom arguments designed around the CHTC pool, and forwards any remaining arguments to `dask_jobqueue.HTCondorCluster`.

Parameters

- **worker_image** (`Optional[str]`) – The Docker image to run the Dask workers inside. Defaults to `daskdev/dask:latest` (Dockerfile). See *this page* for advice on building Docker images for use with Dask-CHTC.
- **gpu_lab** (`bool`) – If `True`, workers will be allowed to run on GPULab nodes. If this is `True`, the default value of `gpus` becomes `1`. Defaults to `False`.
- **gpus** (`Optional[int]`) – The number of GPUs to request. Defaults to `0` unless `gpu_lab = True`, in which case the default is `1`.
- **batch_name** (`Optional[str]`) – The HTCondor JobBatchName to assign to the worker jobs. This can be helpful for more sensible output for *condor_q*. Defaults to `"dask-worker"`.
- **python** (`str`) – The command to execute to start Python inside the worker job. Only modify this if you know what you're doing!
- **kwargs** (`Any`) – Additional keyword arguments, like `cores` or `memory`, are passed to `dask_jobqueue.HTCondorCluster`.

adapt(**args, minimum_jobs=None, maximum_jobs=None, **kwargs*)

Scale Dask cluster automatically based on scheduler activity.

Parameters

- **minimum** (*int*) – Minimum number of workers to keep around for the cluster
- **maximum** (*int*) – Maximum number of workers to keep around for the cluster
- **minimum_memory** (*str*) – Minimum amount of memory for the cluster
- **maximum_memory** (*str*) – Maximum amount of memory for the cluster
- **minimum_jobs** (*int*) – Minimum number of jobs
- **maximum_jobs** (*int*) – Maximum number of jobs
- ****kwargs** – Extra parameters to pass to dask.distributed.Adaptive

See also:

`dask.distributed.Adaptive` for more keyword arguments

`scale`(*n=None*, *jobs=0*, *memory=None*, *cores=None*)

Scale cluster to specified configurations.

Parameters

- **`n`** (*`int`*) – Target number of workers
- **`jobs`** (*`int`*) – Target number of jobs
- **`memory`** (*`str`*) – Target amount of memory
- **`cores`** (*`int`*) – Target number of cores

`classmethod security`()

Return the Dask `Security` object used by Dask-CHTC. Can also be used to create a new Dask `Client` with the correct security settings for connecting to your workers, e.g. if you started your *`CHTCCluster`* via the Dask JupyterLab extension.

CHAPTER NINE

CLI REFERENCE

Dask-CHTC provides a command line tool called `dask-chtc`.

View the available sub-commands by running:

```
dask-chtc --help  # View available commands
```

Here's the full documentation on all of the available commands:

9.1 dask-chtc

Command line tools for Dask-CHTC.

```
dask-chtc [OPTIONS] COMMAND [ARGS]...
```

Options

-v, --verbose
Show log messages as the CLI runs.

--version
Show the version and exit.

9.1.1 config

Inspect and edit Dask-CHTC's configuration.

Dask-CHTC provides a Dask/Dask-Jobqueue configuration file which provides default values for the arguments of CHTCCluster. You can use the subcommands in this group to show, edit, or reset the contents of this configuration file.

See https://docs.dask.org/en/latest/configuration.html#yaml-files for more information on Dask configuration files.

```
dask-chtc config [OPTIONS] COMMAND [ARGS]...
```

edit

Opens your preferred editor on the configuration file.

Set the EDITOR environment variable to change your preferred editor.

```
dask-chtc config edit [OPTIONS]
```

path

Echo the path to the configuration file.

```
dask-chtc config path [OPTIONS]
```

reset

Reset the configuration file's contents.

```
dask-chtc config reset [OPTIONS]
```

Options

`--yes`
Confirm the action without prompting.

show

Show the contents of the configuration file.

To show what Dask actually parsed from the configuration file, add the –parsed option.

```
dask-chtc config show [OPTIONS]
```

Options

`--parsed`
Show the parsed Dask config instead of the contents of the configuration file.

9.1.2 jupyter

Run a Jupyter notebook server as an HTCondor job.

Do not run Jupyter notebook servers on CHTC submit nodes except by using these commands!

Only one Jupyter notebook server can be created by this tool at a time. The subcommands let you create and interact with that server in various ways.

The "run" subcommand runs the notebook server as if you had started it yourself. If your terminal session ends, the notebook server will also stop.

The "start" subcommand runs the notebook server as a persistent HTCondor job: it will not be removed if your terminal session ends. The "status" subcommand can then be used to get information about your notebook server (like its contact address, to put into your web browser). The "stop" subcommand can be used to stop your started notebook server.

```
dask-chtc jupyter [OPTIONS] COMMAND [ARGS]...
```

run

Run a Jupyter notebook server as an HTCondor job.

The Jupyter notebook server will be connected to your terminal session: if you press Ctrl-c or disconnect from the server, your notebook server will end.

To start a notebook server that is not connected to your terminal session, use the "start" subcommand.

Extra arguments will be forwarded to Jupyter. For example, to start Jupyter Lab on some known port, you could run:

> dask-chtc jupyter run lab –port 3456

```
dask-chtc jupyter run [OPTIONS] [JUPYTER_ARGS]...
```

Arguments

JUPYTER_ARGS
: Optional argument(s)

start

Start a Jupyter notebook server as a persistent HTCondor job.

Just like the "run" subcommand, this will start a Jupyter notebook server and show you any output from it. Unlike the "run" subcommand, the Jupyter notebook server will not be connected to your terminal session: if you press Ctrl-c or disconnect from the server, your notebook server will continue running (though you will stop seeing output from it).

You can see the status of a persistent notebook server started by this command by using the "status" subcommand.

To start a notebook server that is connected to your terminal session, use the "run" subcommand.

Extra arguments will be forwarded to Jupyter. For example, to start Jupyter Lab on some known port, you could run

> dask-chtc jupyter start lab –port 3456

```
dask-chtc jupyter start [OPTIONS] [JUPYTER_ARGS]...
```

Arguments

JUPYTER_ARGS
: Optional argument(s)

status

Get information about your running Jupyter notebook server.

If you have started a Jupyter notebook server in the past and need to find it's address again, use this command.

If you are trying to shut down your notebook server job and it is stuck in the REMOVED state, try running "dask-chtc jupyter stop –force".

```
dask-chtc jupyter status [OPTIONS]
```

Options

--raw
Print the raw HTCondor job ad instead of the formatted output.

stop

Stop a Jupyter notebook server that was started via "start".

If the –force option is given, the notebook server will be killed without giving it time to shutdown cleanly. We recommend always trying a normal stop first, then stopping it again with –force only if it is stuck in the REMOVED state for more than a few minutes (use the "status" subcommand to see its current state).

```
dask-chtc jupyter stop [OPTIONS]
```

Options

-f, --force
Stop your notebook server without giving it a chance to clean up.

INDEX