Dask-CHTC

Release 0.1.0

Josh Karpel

Jul 13, 2020

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.

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.

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

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/JoshKarpel/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.

CHAPTER

TWO

RUNNING JUPYTER THROUGH DASK-CHTC

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

Attention: You will need to learn how to *forward ports over SSH* to actually connect to the Jupyter notebook servers that you will learn how to run here. We recommend skimming this page, then reading about port forwarding, then coming back here to try out the commands in full.

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. For example, if you normally start up a Jupyter Lab instance by running

```
$ jupyter lab
[... Jupyter startup logs cut ...]
[I 10:31:41.908 LabApp] Use Control-C to stop this server and shut down all kernels␣
↪(twice to skip confirmation).
[W 10:31:41.966 LabApp] No web browser found: could not locate runnable browser.
[C 10:31:41.967 LabApp]

    To access the notebook, open this file in a browser:
        file:///home/karpel/.local/share/jupyter/runtime/nbserver-2176065-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=9af0f23248bc26014727d1dc85b5fcc07f4803caf777f87a
     or http://127.0.0.1:8888/?token=9af0f23248bc26014727d1dc85b5fcc07f4803caf777f87a
```

then the equivalent `dask-chtc jupyter` command would be

```
$ 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>
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
```

Note the HTCondor job events are mixed into the output stream. This is purely for diagnostic purposes; you may (for example) find them 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]

    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
```

(continues on next page)

(continued from previous page)

```
        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)
```

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>
```

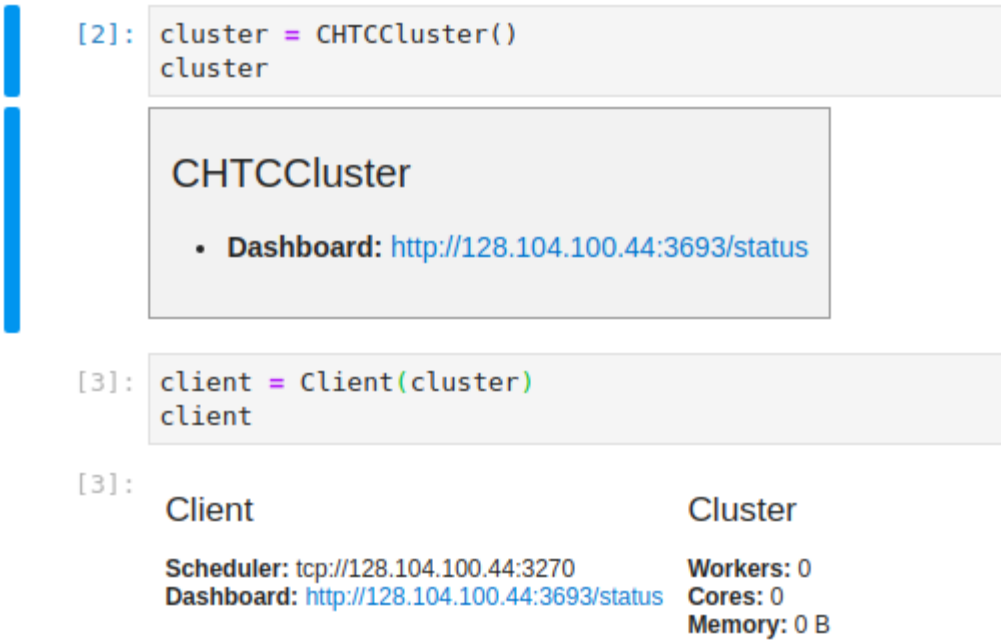
In this prompt, enter a `-L` argument like above and press enter:

```
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`:



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.

CHAPTER

FOUR

EXAMPLE NOTEBOOK

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.

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.

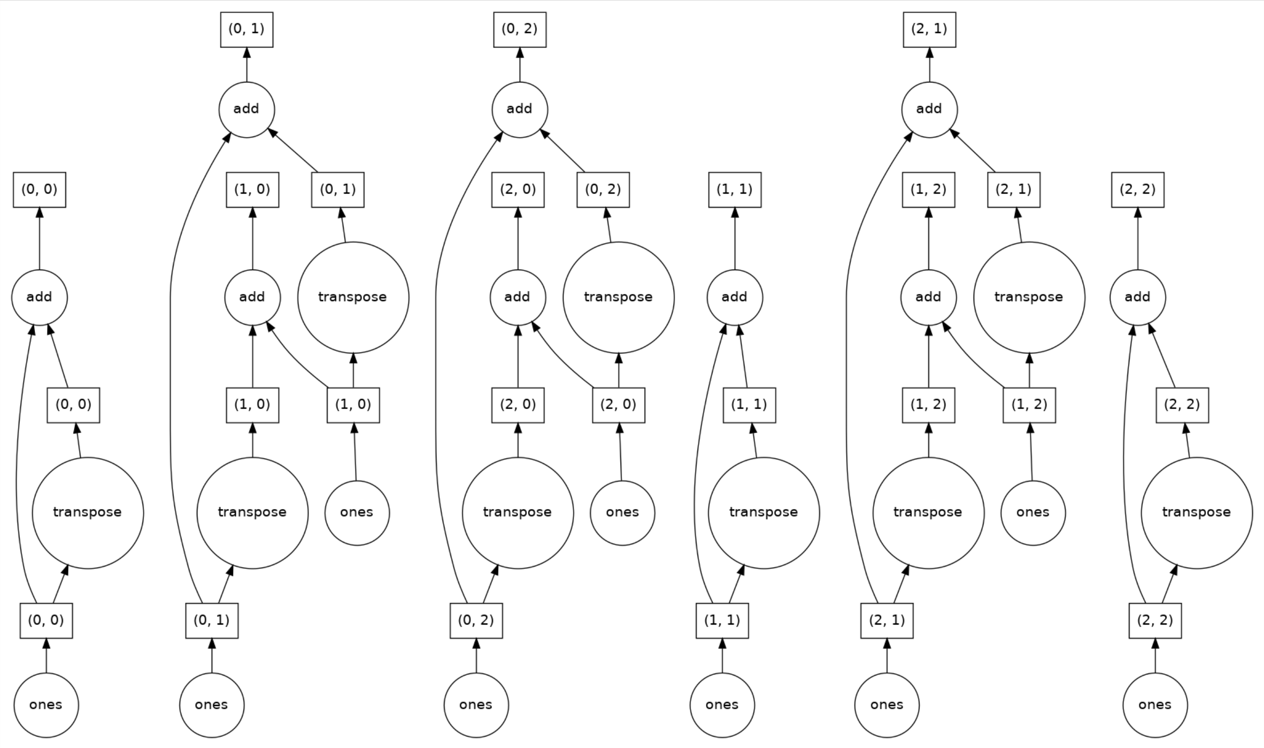
[7]:
```
y = x + x.T
y
```

[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.

CHAPTER

FIVE

API REFERENCE

class dask_chtc.**CHTCCluster**(**, worker_image=None, gpu_lab=False, gpus=None, scheduler_port=range(3100, 3500), dashboard_port=range(3500, 3900), batch_name=None, python='./entrypoint.sh python3', **kwargs*)

Bases: `dask_jobqueue.htcondor.HTCondorCluster`

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).
- **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`.
- **scheduler_port** (`Union[int, Iterable[int]]`) – The port (or range of ports) to use for the Dask scheduler to communicate with the workers. If you want to customize this, keep in mind that only certain ports are usable due to CHTC's infrastructure (the default is a reasonable range) and that you must provide a large enough range to find an unused port, or the scheduler will not be able to start up. You do not need to forward the scheduler port via SSH. We do not recommend changing the default!
- **dashboard_port** (`Union[int, Iterable[int]]`) – The port (or range of ports) to use for the Dask scheduler's dashboard. You will need to use *SSH port forwarding* to forward this port to your own computer.
- **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`.

CHAPTER

SIX

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:

6.1 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.

6.1.1 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.

6.1.2 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 "started" 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 run 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.

```
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".

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

INDEX

Symbols

C

D

J