distributed Documentation

Release 1.12.2

Matthew Rocklin

October 01, 2016

Getting Started

Dask.distributed is a lightweight library for distributed computing in Python. It extends both the `concurrent.futures` and `dask` APIs to moderate sized clusters.

See the quickstart to get started.

Motivation

Why build yet-another-distributed-system?

`Distributed` serves to complement the existing PyData analysis stack. In particular it meets the following needs:

- **Low latency:** Each task suffers about 1ms of overhead. A small computation and network roundtrip can complete in less than 10ms.
- **Peer-to-peer data sharing:** Workers communicate with each other to share data. This removes central bottlenecks for data transfer.
- **Complex Scheduling:** Supports complex workflows (not just map/filter/reduce) which are necessary for sophisticated algorithms used in nd-arrays, machine learning, image processing, and statistics.
- **Pure Python:** Built in Python using well-known technologies. This eases installation, improves efficiency (for Python users), and simplifies debugging.
- **Data Locality:** Scheduling algorithms cleverly execute computations where data lives. This minimizes network traffic and improves efficiency.
- **Familiar APIs:** Compatible with the concurrent.futures API in the Python standard library. Compatible with dask API for parallel algorithms
- **Easy Setup:** As a Pure Python package distributed is `pip` installable and easy to set up on your own cluster.

CHAPTER 2

Architecture

Dask.distributed is a centrally managed, distributed, dynamic task scheduler. The central `dask-scheduler` process coordinates the actions of several `dask-worker` processes spread across multiple machines and the concurrent requests of several clients.

The scheduler is asynchronous and event driven, simultaneously responding to requests for computation from multiple clients and tracking the progress of multiple workers. The event-driven and asynchronous nature makes it flexible to concurrently handle a variety of workloads coming from multiple users at the same time while also handling a fluid worker population with failures and additions. Workers communicate amongst each other for bulk data transfer over sockets.

Internally the scheduler tracks all work as a constantly changing directed acyclic graph of tasks. A task is a Python function operating on Python objects, which can be the results of other tasks. This graph of tasks grows as users submit more computations, fills out as workers complete tasks, and shrinks as users leave or become disinterested in previous results.

Users interact by connecting a local Python session to the scheduler and submitting work, either by individual calls to the simple interface `e.submit(function, *args, **kwargs)` or by using the large data collections and parallel algorithms of the parent `dask` library. The collections in the `dask` library like `dask.array` and `dask.dataframe` provide easy access to sophisticated algorithms and familiar APIs like NumPy and Pandas, while the simple `e.submit` interface provides users with custom control when they want to break out of canned "big data" abstractions and submit fully custom workloads.

CHAPTER 3

Contents

3.1 Install Distributed

You can install distributed with `conda`, with `pip`, or by installing from source.

3.1.1 Conda

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

```
conda install distributed -c conda-forge
```

3.1.2 Pip

Or install distributed with `pip`:

```
pip install distributed --upgrade
```

3.1.3 Source

To install distributed from source, clone the repository from github:

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

3.1.4 Notes

Note for Macports users: There is a known issue. with python from macports that makes executables be placed in a location that is not available by default. A simple solution is to extend the *PATH* environment variable to the location where python from macports install the binaries:

```
$ export PATH=/opt/local/Library/Frameworks/Python.framework/Versions/3.5/bin:$PATH

or

$ export PATH=/opt/local/Library/Frameworks/Python.framework/Versions/2.7/bin:$PATH
```

3.2 Quickstart

3.2.1 Install

```
$ pip install distributed --upgrade
```

See installation document for more information.

3.2.2 Setup Dask.distributed the Hard Way

Set up scheduler and worker processes on your local computer:

```
$ dask-scheduler
Scheduler started at 127.0.0.1:8786

$ dask-worker 127.0.0.1:8786
$ dask-worker 127.0.0.1:8786
$ dask-worker 127.0.0.1:8786
```

Note: At least one `dask-worker` must be running after launching a scheduler.

Launch an Executor and point it to the IP/port of the scheduler.

```
>>> from distributed import Executor
>>> executor = Executor('127.0.0.1:8786')
```

See setup for advanced use.

3.2.3 Setup Dask.distributed the Easy Way

If you create an executor without providing an address it will start up a local scheduler and worker for you.

```
>>> from distributed import Executor
>>> executor = Executor()
>>> executor
<Executor: scheduler="127.0.0.1:8786" processes=8 cores=8>
```

Map and Submit Functions

Use the `map` and `submit` methods to launch computations on the cluster. The `map/submit` functions send the function and arguments to the remote workers for processing. They return `Future` objects that refer to remote data on the cluster. The `Future` returns immediately while the computations run remotely in the background.

```
>>> def square(x):
        return x ** 2

>>> def neg(x):
        return -x

>>> A = executor.map(square, range(10))
>>> B = executor.map(neg, A)
>>> total = executor.submit(sum, B)
>>> total.result()
-285
```

Gather

The `map/submit` functions return `Future` objects, lightweight tokens that refer to results on the cluster. By default the results of computations *stay on the cluster*.

```
>>> total  # Function hasn't yet completed
<Future: status: waiting, key: sum-58999c52e0fa35c7d7346c098f5085c7>

>>> total  # Function completed, result ready on remote worker
<Future: status: finished, key: sum-58999c52e0fa35c7d7346c098f5085c7>
```

Gather results to your local machine either with the `Future.result` method for a single future, or with the `Executor.gather` method for many futures at once.

```
>>> total.result()     # result for single future
-285
>>> executor.gather(A) # gather for many futures
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Restart

When things go wrong, or when you want to reset the cluster state, call the `restart` method.

```
>>> executor.restart()
```

See executor for advanced use.

3.3 Setup Network

A `distributed` network consists of one `Scheduler` node and several `Worker` nodes. One can set these up in a variety of ways

3.3.1 Using the Command Line

We launch the `dask-scheduler` executable in one process and the `dask-worker` executable in several processes, possibly on different machines.

Launch `dask-scheduler` on one node:

```
$ dask-scheduler
Start scheduler at 192.168.0.1:8786
```

Then launch `dask-worker` on the rest of the nodes, providing the address to the node that hosts `dask-scheduler`:

```
$ dask-worker 192.168.0.1:8786
Start worker at:            192.168.0.2:12345
Registered with center at:  192.168.0.1:8786

$ dask-worker 192.168.0.1:8786
```

```
Start worker at:            192.168.0.3:12346
Registered with center at:  192.168.0.1:8786

$ dask-worker 192.168.0.1:8786
Start worker at:            192.168.0.4:12347
Registered with center at:  192.168.0.1:8786
```

There are various mechanisms to deploy these executables on a cluster, ranging from manualy SSH-ing into all of the nodes to more automated systems like SGE/SLURM/Torque or Yarn/Mesos.

3.3.2 Using SSH

For this functionality, *paramiko* library must be installed (e.g. by running *pip install paramiko*).

The convenience script `dask-ssh` opens several SSH connections to your target computers and initializes the network accordingly. You can give it a list of hostnames or IP addresses:

```
$ dask-ssh 192.168.0.1 192.168.0.2 192.168.0.3 192.168.0.4
```

Or you can use normal UNIX grouping:

```
$ dask-ssh 192.168.0.{1,2,3,4}
```

Or you can specify a hostfile that includes a list of hosts:

```
$ cat hostfile.txt
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.4

$ dask-ssh --hostfile hostfile.txt
```

3.3.3 Using the Python API

Alternatively you can start up the `distributed.scheduler.Scheduler` and `distributed.worker.Worker` objects within a Python session manually. Both are `torando.tcpserver.TCPServer` objects.

Start the Scheduler, provide the listening port (defaults to 8786) and Tornado IOLoop (defaults to `IOLoop.current()`)

```python
from distributed import Scheduler
from tornado.ioloop import IOLoop
from threading import Thread

loop = IOLoop.current()
t = Thread(target=loop.start, daemon=True)
t.start()

s = Scheduler(loop=loop)
s.start(8786)
```

On other nodes start worker processes that point to the IP address and port of the scheduler.

```
from distributed import Worker
from tornado.ioloop import IOLoop
from threading import Thread

loop = IOLoop.current()
t = Thread(target=loop.start, daemon=True)
t.start()

w = Worker('127.0.0.1', 8786, loop=loop)
w.start(0)  # choose randomly assigned port
```

Alternatively, replace `Worker` with `Nanny` if you want your workers to be managed in a separate process by a local nanny process.

3.3.4 Using LocalCluster

You can do the work above easily using LocalCluster.

```
from distributed import LocalCluster
c = LocalCluster(nanny=False)
```

A scheduler will be available under `c.scheduler` and a list of workers under `c.workers`. There is an IOLoop running in a background thread.

3.3.5 Using Amazon EC2

See the EC2 quickstart for information on the `dask-ec2` easy setup script to launch a canned cluster on EC2.

3.3.6 Cleanup

It is common and safe to terminate the cluster by just killing the processes. The workers and scheduler have no persistent state.

Programmatically you can use the client interface (`rpc`) to call the `terminate` methods on the workers and schedulers.

3.3.7 Software Environment

The workers and clients should all share the same software environment. That means that they should all have access to the same libraries and that those libraries should be the same version. Dask generally assumes that it can call a function on any worker with the same outcome (unless explicitly told otherwise.)

This is typically enforced through external means, such as by having a network file system (NFS) mount for libraries, by starting the `dask-worker` processes in equivalent docker containers, or through any of the other means typically employed by cluster administrators.

3.4 EC2 Startup Script

First, add your AWS credentials to `~/.aws/credentials` like this:

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY
```

For other ways to manage or troubleshoot credentials, see the boto3 docs.

Now, you can quickly deploy a scheduler and workers on EC2 using the `dask-ec2` quickstart application:

```
pip install dask-ec2
dask-ec2 up --keyname YOUR-AWS-KEY --keypair ~/.ssh/YOUR-AWS-SSH-KEY.pem
```

This provisions a cluster on Amazon's EC2 cloud service, installs Anaconda, and sets up a scheduler and workers. In then prints out instructions on how to connect to the cluster.

3.4.1 Options

The `dask-ec2` startup script comes with the following options for creating a cluster:

```
$ dask-ec2 up --help
Usage: dask-ec2 up [OPTIONS]

Options:
  --keyname TEXT                Keyname on EC2 console  [required]
  --keypair PATH                Path to the keypair that matches the keyname [required]
  --name TEXT                   Tag name on EC2
  --region-name TEXT            AWS region  [default: us-east-1]
  --ami TEXT                    EC2 AMI  [default: ami-d05e75b8]
  --username TEXT               User to SSH to the AMI  [default: ubuntu]
  --type TEXT                   EC2 Instance Type  [default: m3.2xlarge]
  --count INTEGER               Number of nodes  [default: 4]
  --security-group TEXT         Security Group Name  [default: dask-ec2-default]
  --volume-type TEXT            Root volume type  [default: gp2]
  --volume-size INTEGER         Root volume size (GB)  [default: 500]
  --file PATH                   File to save the metadata  [default: cluster.yaml]
  --provision / --no-provision  Provision salt on the nodes  [default: True]
  --dask / --no-dask            Install Dask.Distributed in the cluster [default: True]
  --nprocs INTEGER              Number of processes per worker  [default: 1]
  -h, --help                    Show this message and exit.
```

3.4.2 Connect

Connection instructions follow successful completion of the `dask-ec2 up` command. The involve the following:

```
dask-ec2 ssh          # SSH into head node
ipython               # Start IPython console on head node
```

```
>>> from distributed import Executor, s3, progress
>>> e = Executor('127.0.0.1:8786')
```

This executor now has access to all the cores of your cluster.

3.4.3 Destroy

You can destroy your cluster from your local machine with the destroy command:

```
dask-ec2 destroy
```

3.5 Web Interface

Information about the current state of the network helps to track progress, identify performance issues, and debug failures.

Dask.distributed includes a web interface to help deliver this information over a normal web page in real time. This web interface is launched by default wherever the scheduler is launched if the scheduler machine has Bokeh installed (`conda install bokeh -c bokeh`). The web interface is normally available at `http://scheduler-address:8787/status/` and can be viewed any normal web browser.

3.5.1 Plots

Example Computation

The following plots show a trace of the following computation:

```
from distributed import Executor
from time import sleep
import random

def inc(x):
    sleep(random.random() / 10)
    return x + 1

def dec(x):
    sleep(random.random() / 10)
    return x - 1

def add(x, y):
    sleep(random.random() / 10)
    return x + y


e = Executor('127.0.0.1:8786')

incs = e.map(inc, range(100))
decs = e.map(dec, range(100))
adds = e.map(add, incs, decs)
total = e.submit(sum, adds)

del incs, decs, adds
total.result()
```

Progress

The interface shows the progress of the various computations as well as the exact number completed.

Each bar is assigned a color according to the function being run. Each bar has a few components. On the left the lighter shade is the number of tasks that have both completed and have been released from memory. The darker shade to the right corresponds to the tasks that are completed and whose data still reside in memory. If errors occur then they appear as a black colored block to the right.

Typical computations may involve dozens of kinds of functions. We handle this visually with the following approaches:

1. Functions are ordered by the number of total tasks
2. The colors are assigned in a round-robin fashion from a standard palette
3. The progress bars shrink horizontally to make space for more functions
4. Only the largest functions (in terms of number of tasks) are displayed

Counts of tasks processing, waiting for dependencies, processing, etc.. are displayed in the title bar.

Memory Use

The interface shows the relative memory use of each function with a horizontal bar sorted by function name.

The title shows the number of total bytes in use. Hovering over any bar tells you the specific function and how many bytes its results are actively taking up in memory. This does not count data that has been released.

Task Stream

The task stream plot shows when tasks complete on which workers. Worker cores are on the y-axis and time is on the x-axis. As a worker completes a task its start and end times are recorded and a rectangle is added to this plot accordingly.

If data transfer occurs between workers a *red* bar appears preceding the task bar showing the duration of the transfer. If an error occurs than a *black* bar replaces the normal color. This plot show the last 1000 tasks. It resets if there is a delay greater than 10 seconds.

For a full history of the last 100,000 tasks see the `tasks/` page.

Resources

The resources plot show the average CPU and Memory use over time as well as average network traffic. More detailed information on a per-worker basis is available in the `workers/` page.

3.5.2 Connecting to Web Interface

Default

By default, `dask-scheduler` prints out the address of the web interface:

```
INFO -  Bokeh UI at:  http://10.129.39.91:8787/status
...
INFO - Starting Bokeh server on port 8787 with applications at paths ['/status', '/tasks']
```

The machine hosting the scheduler runs an HTTP server serving at that address.

3.5.3 Troubleshooting

Some clusters restrict the ports that are visible to the outside world. These ports may include the default port for the web interface, `8787`. There are a few ways to handle this:

1. Open port `8787` to the outside world. Often this involves asking your cluster administrator.

2. Use a different port that is publicly accessible using the `--bokeh-port PORT` option on the `dask-scheduler` command.
3. Use fancier techniques, like Port Forwarding

Running distributed on a remote machine can cause issues with viewing the web UI – this depends on the remote machines network configuration.

Port Forwarding

If you have SSH access then one way to gain access to a blocked port is through SSH port forwarding. A typical use case looks like the following:

```
local$ ssh -L 8000:localhost:8787 user@remote
remote$ dask-scheduler  # now, the web UI is visible at localhost:8000
remote$ # continue to set up dask if needed -- add workers, etc
```

It is then possible to go to `localhost:8000` and see Dask Web UI. This same approach is not specific to dask.distributed, but can be used by any service that operates over a network, such as Jupyter notebooks. For example, if we chose to do this we could forward port 8888 (the default Jupyter port) to port 8001 with `ssh -L 8001:localhost:8888 user@remote`.

3.6 Examples

3.6.1 Word count in HDFS

Setup

In this example, we'll use `distributed` with the `hdfs3` library to count the number of words in text files (Enron email dataset, 6.4 GB) stored in HDFS.

Copy the text data from Amazon S3 into HDFS on the cluster:

```
$ hadoop distcp s3n://AWS_SECRET_ID:AWS_SECRET_KEY@blaze-data/enron-email hdfs:///tmp/enron
```

where `AWS_SECRET_ID` and `AWS_SECRET_KEY` are valid AWS credentials.

Start the `distributed` scheduler and workers on the cluster.

Code example

Import `distributed`, `hdfs3`, and other standard libraries used in this example:

```
>>> import hdfs3
>>> from collections import defaultdict, Counter
>>> from distributed import Executor, progress
```

Initalize a connection to HDFS, replacing `NAMENODE_HOSTNAME` and `NAMENODE_PORT` with the hostname and port (default: 8020) of the HDFS namenode.

```
>>> hdfs = hdfs3.HDFileSystem('NAMENODE_HOSTNAME', port=NAMENODE_PORT)
```

Initalize a connection to the `distributed` executor, replacing `EXECUTOR_IP` and `EXECUTOR_PORT` with the IP address and port of the `distributed` scheduler.

```
>>> e = Executor('EXECUTOR_IP:EXECUTOR_PORT')
```

Generate a list of filenames from the text data in HDFS:

```
>>> filenames = hdfs.glob('/tmp/enron/*/*')
>>> print(filenames[:5])

['/tmp/enron/edrm-enron-v2_nemec-g_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_ring-r_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_bailey-s_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_fischer-m_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_geaccone-t_xml.zip/merged.txt']
```

Print the first 1024 bytes of the first text file:

```
>>> print(hdfs.head(filenames[0]))

b'Date: Wed, 29 Nov 2000 09:33:00 -0800 (PST)\r\nFrom: Xochitl-Alexis Velasc
o\r\nTo: Mark Knippa, Mike D Smith, Gerald Nemec, Dave S Laipple, Bo Barnwel
l\r\nCc: Melissa Jones, Iris Waser, Pat Radford, Bonnie Shumaker\r\nSubject:
 Finalize ECS/EES Master Agreement\r\nX-SDOC: 161476\r\nX-ZLID: zl-edrm-enro
n-v2-nemec-g-2802.eml\r\n\r\n\r\nPlease plan to attend a meeting to finalize the
 ECS/EES  Master Agreement \r\ntomorrow 11/30/00 at 1:30 pm CST.\r\n\r\nI wi
ll email everyone tomorrow with location.\r\n\r\nDave-I will also email you
the call in number tomorrow.\r\n\r\nThanks\r\nXochitl\r\n\r\n***********\r\n
EDRM Enron Email Data Set has been produced in EML, PST and NSF format by ZL
 Technologies, Inc. This Data Set is licensed under a Creative Commons Attri
bution 3.0 United States License <http://creativecommons.org/licenses/by/3.0
/us/> . To provide attribution, please cite to "ZL Technologies, Inc. (http:
//www.zlti.com)."\r\n***********\r\nDate: Wed, 29 Nov 2000 09:40:00 -0800 (P
ST)\r\nFrom: Jill T Zivley\r\nTo: Robert Cook, Robert Crockett, John Handley
, Shawna'
```

Create a function to count words in each file:

```
>>> def count_words(fn):
...     word_counts = defaultdict(int)
...     with hdfs.open(fn) as f:
...         for line in f:
...             for word in line.split():
...                 word_counts[word] += 1
...     return word_counts
```

Before we process all of the text files using the distributed workers, let's test our function locally by counting the number of words in the first text file:

```
>>> counts = count_words(filenames[0])
>>> print(sorted(counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])

[(b'the', 144873),
 (b'of', 98122),
 (b'to', 97202),
 (b'and', 90575),
 (b'or', 60305),
 (b'in', 53869),
 (b'a', 43300),
 (b'any', 31632),
 (b'by', 31515),
 (b'is', 30055)]
```

We can perform the same operation of counting the words in the first text file, except we will use `e.submit` to execute the computation on a `distributed` worker:

```
>>> future = e.submit(count_words, filenames[0])
>>> counts = future.result()
>>> print(sorted(counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])

[(b'the', 144873),
 (b'of', 98122),
 (b'to', 97202),
 (b'and', 90575),
 (b'or', 60305),
 (b'in', 53869),
 (b'a', 43300),
 (b'any', 31632),
 (b'by', 31515),
 (b'is', 30055)]
```

We are ready to count the number of words in all of the text files using `distributed` workers. Note that the `map` operation is non-blocking, and you can continue to work in the Python shell/notebook while the computations are running.

```
>>> futures = e.map(count_words, filenames)
```

We can check the status of some `futures` while all of the text files are being processed:

```
>>> len(futures)

161

>>> futures[:5]

[<Future: status: finished, key: count_words-5114ab5911de1b071295999c9049e941>,
 <Future: status: pending, key: count_words-d9e0d9daf6a1eab4ca1f26033d2714e7>,
 <Future: status: pending, key: count_words-d2f365a2360a075519713e9380af45c5>,
 <Future: status: pending, key: count_words-bae65a245042325b4c77fc8dde1acf1e>,
 <Future: status: pending, key: count_words-03e82a9b707c7e36eab95f4feec1b173>]

>>> progress(futures)

[########################################] | 100% Completed |  3min  0.2s
```

When the `futures` finish reading in all of the text files and counting words, the results will exist on each worker. This operation required about 3 minutes to run on a cluster with three worker machines, each with 4 cores and 16 GB RAM.

Note that because the previous computation is bound by the GIL in Python, we can speed it up by starting the `distributed` workers with the `--nprocs 4` option.

To sum the word counts for all of the text files, we need to gather some information from the `distributed` workers. To reduce the amount of data that we gather from the workers, we can define a function that only returns the top 10,000 words from each text file.

```
>>> def top_items(d):
...     items = sorted(d.items(), key=lambda kv: kv[1], reverse=True)[:10000]
...     return dict(items)
```

We can then `map` the futures from the previous step to this culling function. This is a convenient way to construct a pipeline of computations using futures:

```
>>> futures2 = e.map(top_items, futures)
```

We can `gather` the resulting culled word count data for each text file to the local process:

```
>>> results = e.gather(iter(futures2))
```

To sum the word counts for all of the text files, we can iterate over the results in `futures2` and update a local dictionary that contains all of the word counts.

```
>>> all_counts = Counter()
>>> for result in results:
...     all_counts.update(result)
```

Finally, we print the total number of words in the results and the words with the highest frequency from all of the text files:

```
>>> print(len(all_counts))

8797842

>>> print(sorted(all_counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])

[(b'0', 67218380),
 (b'the', 19586868),
 (b'-', 14123768),
 (b'to', 11893464),
 (b'N/A', 11814665),
 (b'of', 11724827),
 (b'and', 10253753),
 (b'in', 6684937),
 (b'a', 5470371),
 (b'or', 5227805)]
```

The complete Python script for this example is shown below:

```
# word-count.py

import hdfs3
from collections import defaultdict, Counter
from distributed import Executor, progress

hdfs = hdfs3.HDFileSystem('NAMENODE_HOSTNAME', port=NAMENODE_PORT)
e = Executor('EXECUTOR_IP:EXECUTOR_PORT')

filenames = hdfs.glob('/tmp/enron/*/*')
print(filenames[:5])
print(hdfs.head(filenames[0]))


def count_words(fn):
    word_counts = defaultdict(int)
    with hdfs.open(fn) as f:
        for line in f:
            for word in line.split():
                word_counts[word] += 1
    return word_counts

counts = count_words(filenames[0])
print(sorted(counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])
```

```
future = e.submit(count_words, filenames[0])
counts = future.result()
print(sorted(counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])

futures = e.map(count_words, filenames)
len(futures)
futures[:5]
progress(futures)


def top_items(d):
    items = sorted(d.items(), key=lambda kv: kv[1], reverse=True)[:10000]
    return dict(items)

futures2 = e.map(top_items, futures)
results = e.gather(iter(futures2))

all_counts = Counter()
for result in results:
    all_counts.update(result)

print(len(all_counts))

print(sorted(all_counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])
```

3.7 Executor

The Executor is the primary entry point for users of `distributed`.

After you setup a cluster, initialize an `Executor` by pointing it to the address of a `Scheduler`:

```
>>> from distributed import Executor
>>> executor = Executor('127.0.0.1:8786')
```

3.7.1 Usage

submit

You can submit individual function calls with the `executor.submit` method

```
>>> def inc(x):
        return x + 1

>>> x = executor.submit(inc, 10)
>>> x
<Future - key: inc-e4853cffcc2f51909cdb69d16dacd1a5>
```

The result is on one of the distributed workers. We can continue using `x` in further calls to `submit`:

```
>>> type(x)
Future
>>> y = executor.submit(inc, x)
```

Gather results

We can collect results in a variety of ways. First, we can use the `.result()` method on futures

```
>>> x.result()
2
```

Second, we can use the gather method on the executor

```
>>> executor.gather([x, y])
(2, 3)
```

Third, we can use the `as_completed` function to iterate over results as soon as they become available.

```
>>> from distributed import as_completed
>>> seq = as_completed([x, y])
>>> next(seq).result()
2
>>> next(seq).result()
3
```

But, as always, we want to minimize communicating results back to the local process. It's often best to leave data on the cluster and operate on it remotely with functions like `submit`, `map`, `get` and `compute`. See efficiency for more information on efficient use of distributed.

`map`

We can map a function over many inputs at once

```
>>> L = executor.map(inc, range(10))
```

The `map` method returns a list of futures. This is a break with the `concurrent.futures` API, which returns the results directly. We keep the results as futures so that they can stay on the distributed cluster.

Additionally, we don't do any kind of batching so every function application will be a new task which will have a couple milliseconds of overhead. It is unwise to use `executor.map` for small, fast functions where scheduling overhead is likely to be more expensive than the cost of the function itself. For example, our function `inc` is actually a *terrible* function to parallelize in practice.

`dask`

Distributed provides a dask compliant task scheduling interface. It provides this through two methods, `get` (synchronous) and `compute` (asynchronous).

get

We provide dask graph dictionaries to the scheduler:

```
>>> dsk = {'x': 1, 'y': (inc, 'x')}
>>> executor.get(dsk, 'y')
2
```

This function pulls results back by default. This is so that it can integrate with existing dask code.

```
>>> import dask.array as da
>>> x = da.random.random(1000000000, chunks=(1000000,))
>>> x.sum().compute()  # use local threads
499999359.23511785
```

```
>>> x.sum().compute(get=executor.get)  # use distributed cluster
499999359.23511785
```

compute

We can also provide dask collections (arrays, bags, dataframes, delayed values) to the executor with the `compute` method.

```
>>> type(x)
dask.array.Array
>>> type(df)
dask.dataframe.DataFrame

>>> x_future, df_future = executor.compute(x, df)
```

This immediately returns standard `Future` objects as would be returned by `submit` or `map`.

`restart`

When things go wrong, restart the cluster with the `.restart()` method.

```
>>> executor.restart()
```

This both resets the scheduler state and all of the worker processes. All current data and computations will be lost. All existing futures set their status to `'cancelled'`.

See resilience for more information.

3.7.2 Internals

Data Locality

By default the executor does not bring results back to your local computer but leaves them on the distributed network. As a result, computations on returned results like the following don't require any data transfer.

```
>>> y = executor.submit(inc, x)  # no data transfer required
```

In addition, the internal scheduler endeavors to run functions on worker nodes that already have the necessary input data. It avoids worker-to-worker communication when convenient.

Pure Functions by Default

By default we assume that all functions are pure. If this is not the case you should use the `pure=False` keyword argument.

The executor associates a key to all computations. This key is accessible on the Future object.

```
>>> from operator import add
>>> x = executor.submit(add, 1, 2)
>>> x.key
'add-ebf39f96ad7174656f97097d658f3fa2'
```

This key should be the same accross all computations with the same inputs and across all machines. If you run the computation above on any computer with the same environment then you should get the exact same key.

The scheduler avoids redundant computations. If the result is already in memory from a previous call then that old result will be used rather than recomputing it. Calls to submit or map are idempotent in the common case.

While convenient, this feature may be undesired for impure functions, like `random`. In these cases two calls to the same function with the same inputs should produce different results. We accomplish this with the `pure=False` keyword argument. In this case keys are randomly generated (by `uuid4`.)

```
>>> import numpy as np
>>> executor.submit(np.random.random, 1000, pure=False).key
'random_sample-fc814a39-ee00-42f3-8b6f-cac65bcb5556'
>>> executor.submit(np.random.random, 1000, pure=False).key
'random_sample-a24e7220-a113-47f2-a030-72209439f093'
```

Garbage Collection

Prolonged use of `distributed` may allocate a lot of remote data. The executor can clean up unused results by reference counting.

The executor reference counts `Future` objects. When a particular key no longer has any Future objects pointing to it it will be released from distributed memory if no active computations still require it.

In this way garbage collection in the distributed memory space of your cluster mirrors garbage collection within your local Python session.

Known future keys and reference counts can be found in the following dictionaries:

```
>>> executor.futures
>>> executor.refcount
```

The scheduler also cleans up intermediate results when provided full dask graphs. You can always use the lower level `delete` or `clear` functions in `distributed.client` to manage data manually.

Coroutines

If you are operating in an asynchronous environment then all blocking functions listed above have asynchronous equivalents. Currently these have the exact same name but are prepended with an underscore (_) so, `.result` is synchronous while `._result` is asynchronous. If a function has no asynchronous counterpart then that means it does not significantly block. The `.submit` and `.map` functions are examples of this; they return immediately in either case.

3.8 Local Cluster

For convenience you can start a local cluster from your Python session.

```
>>> from distributed import Executor, LocalCluster
>>> c = LocalCluster()
LocalCluster("127.0.0.1:8786", workers=8, ncores=8)
>>> e = Executor(c)
<Executor: scheduler=127.0.0.1:8786 processes=8 cores=8>
```

Alternatively, a `LocalCluster` is made for you automatically if you create an `Executor` with no arguments.

```
>>> from distributed import Executor
>>> e = Executor()
>>> e
<Executor: scheduler=127.0.0.1:8786 processes=8 cores=8>
```

class `distributed.deploy.local.`**`LocalCluster`**(*n_workers=None*, *threads_per_worker=None*, *nanny=True*, *loop=None*, *start=True*, *scheduler_port=8786*, *silence_logs=50*, *diagnostics_port=8787*, *services={'http': <function HTTPScheduler at 0x7fbe34170048>}*, ***kwargs*)

Create local Scheduler and Workers

This creates a "cluster" of a scheduler and workers running on the local machine.

Parameters **n_workers: int**

Number of workers to start

threads_per_worker: int

Number of threads per each worker

nanny: boolean

If true start the workers in separate processes managed by a nanny. If False keep the workers in the main calling process

scheduler_port: int

Port of the scheduler. 8786 by default, use 0 to choose a random port

Examples

```
>>> c = LocalCluster()  # Create a local cluster with as many workers as cores
>>> c
LocalCluster("127.0.0.1:8786", workers=8, ncores=8)
```

```
>>> e = Executor(c)  # connect to local cluster
```

Add a new worker to the cluster >>> w = c.start_worker(ncores=2) # doctest: +SKIP

Shut down the extra worker >>> c.remove_worker(w) # doctest: +SKIP

Start a diagnostic web server and open a new browser tab >>> c.start_diagnostics_server(show=True) # doctest: +SKIP

`close`()

Close the cluster

`start_diagnostics_server`(*port=8787*, *show=False*, *silence=50*)

Start Diagnostics Web Server

This starts a web application to show diagnostics of what is happening on the cluster. This application runs in a separate process and is generally available at the following location:

http://localhost:8787/status/

`start_worker`(*port=0*, *ncores=0*, ***kwargs*)

Add a new worker to the running cluster

Parameters **port: int (optional)**

Port on which to serve the worker, defaults to 0 or random

ncores: int (optional)

Number of threads to use. Defaults to number of logical cores

nanny: boolean

If true start worker in separate process managed by a nanny

Returns The created Worker or Nanny object. Can be discarded.

Examples

```
>>> c = LocalCluster()
>>> c.start_worker(ncores=2)
```

stop_worker(*w*)

Stop a running worker

Examples

```
>>> c = LocalCluster()
>>> w = c.start_worker(ncores=2)
>>> c.stop_worker(w)
```

3.9 Efficiency

Parallel computing done well is responsive and rewarding. However, several speed-bumps can get in the way. This section describes common ways to ensure performance.

3.9.1 Leave data on the cluster

Wait as long as possible to gather data locally. If you want to ask a question of a large piece of data on the cluster it is often faster to submit a function onto that data then to bring the data down to your local computer.

For example if we have a numpy array on the cluster and we want to know its shape we might choose one of the following options:

1. **Slow:** Gather the numpy array to the local process, access the `.shape` attribute
2. **Fast:** Send a lambda function up to the cluster to compute the shape

```
>>> x = executor.submit(np.random.random, (1000, 1000))
>>> type(x)
Future
```

Slow

```
>>> x.result().shape()  # Slow from lots of data transfer
(1000, 1000)
```

Fast

```
>>> executor.submit(lambda a: a.shape, x).result()  # fast
(1000, 1000)
```

3.9.2 Use larger tasks

The scheduler adds about *one millisecond* of overhead per task or Future object. While this may sound fast it's quite slow if you run a billion tasks. If your functions run faster than 100ms or so then you might not see any speedup from using distributed computing.

A common solution is to batch your input into larger chunks.

Slow

```
>>> futures = executor.map(f, seq)
>>> len(futures)  # avoid large numbers of futures
1000000000
```

Fast

```
>>> def f_many(chunk):
...     return [f(x) for x in chunk]

>>> from toolz import partition_all
>>> chunks = partition_all(1000000, seq)  # Collect into groups of size 1000

>>> futures = executor.map(f_many, chunks)
>>> len(futures)  # Compute on larger pieces of your data at once
1000
```

3.9.3 Adjust between Threads and Processes

By default a single `Worker` runs many computations in parallel using as many threads as your compute node has cores. When using pure Python functions this may not be optimal and you may instead want to run several separate worker processes on each node, each using one thread. When configuring your cluster you may want to use the options to the `dask-worker` executable as follows:

```
$ dask-worker ip:port --nprocs 8 --nthreads 1
```

Note that if you're primarily using NumPy, Pandas, SciPy, Scikit Learn, Numba, or other C/Fortran/LLVM/Cython-accelerated libraries then this is not an issue for you. Your code is likely optimal for use with multi-threading.

3.9.4 Don't go distributed

Consider the dask and concurrent.futures modules, which have similar APIs to distributed but operate on a single machine. It may be that your problem performs well enough on a laptop or large workstation.

Consider accelerating your code through other means than parallelism. Better algorithms, data structures, storage formats, or just a little bit of C/Fortran/Numba code might be enough to give you the 10x speed boost that you're looking for. Parallelism and distributed computing are expensive ways to accelerate your application.

3.10 Data Locality

Data movement often needlessly limits performance.

This is especially true for analytic computations. `Distributed` minimizes data movement when possible and enables the user to take control when necessary. This document describes current scheduling policies and user API around data locality.

3.10.1 Current Policies

Task Submission

In the common case distributed runs tasks on workers that already hold dependent data. If you have a task `f(x)` that requires some data `x` then that task will very likely be run on the worker that already holds `x`.

If a task requires data split among multiple workers, then the scheduler chooses to run the task on the worker that requires the least data transfer to it. The size of each data element is measured by the workers using the `sys.getsizeof` function, which depends on the `__sizeof__` protocol generally available on most relevant Python objects.

Data Scatter

When a user scatters data from their local process to the distributed network this data is distributed in a round-robin fashion grouping by number of cores. So for example If we have two workers `Alice` and `Bob`, each with two cores and we scatter out the list `range(10)` as follows:

```
futures = e.scatter(range(10))
```

Then Alice and Bob receive the following data

- Alice: `[0, 1, 4, 5, 8, 9]`
- Bob: `[2, 3, 6, 7]`

3.10.2 User Control

Complex algorithms may require more user control.

For example the existence of specialized hardware such as GPUs or database connections may restrict the set of valid workers for a particular task.

In these cases use the `workers=` keyword argument to the `submit`, `map`, or `scatter` functions, providing a hostname, IP address, or alias as follows:

```
future = e.submit(func, *args, workers=['Alice'])
```

- Alice: `[0, 1, 4, 5, 8, 9, new_result]`
- Bob: `[2, 3, 6, 7]`

Required data will always be moved to these workers, even if the volume of that data is significant. If this restriction is only a preference and not a strict requirement, then add the `allow_other_workers` keyword argument to signal that in extreme cases such as when no valid worker is present, another may be used.

```
future = e.submit(func, *args, workers=['Alice'],
                  allow_other_workers=True)
```

Additionally the `scatter` function supports a `broadcast=` keyword argument to enforce that the all data is sent to all workers rather than round-robined. If new workers arrive they will not automatically receive this data.

```
futures = e.scatter([1, 2, 3], broadcast=True)  # send data to all workers
```

- Alice: `[1, 2, 3]`
- Bob: `[1, 2, 3]`

Valid arguments for `workers=` include the following:

- A single IP addresses, IP/Port pair, or hostname like the following:

```
192.168.1.100, 192.168.1.100:8989, alice, alice:8989
```

- A list or set of the above:

```
['alice'], ['192.168.1.100', '192.168.1.101:9999']
```

If only a hostname or IP is given then any worker on that machine will be considered valid. Additionally, you can provide aliases to workers upon creation.:

```
$ dask-worker scheduler_address:8786 --name worker_1
```

And then use this name when specifying workers instead.

```
e.map(func, sequence, workers='worker_1')
```

See the efficiency page to learn about best practices.

3.11 Managing Computation

Data and Computation in Dask.distributed are always in one of three states

1. Concrete values in local memory. Example include the integer `1` or a numpy array.
2. Lazy computations in a dask graph, perhaps stored in a `dask.delayed` or `dask.dataframe` object.
3. Running computations or remote data, pointed to by `Future` objects pointing to computations currently in flight.

All three of these forms are important and there are functions the convert between all three states.

3.11.1 Dask Collections to Concrete Values

You can turn any dask collection into a concrete value by calling the `.compute()` method or `dask.compute(...)` function. This function will block until the computation is finished, going straight from a lazy dask collection to a concrete value in local memory.

This approach is the most familiar and straightforward, especially for people coming from the standard single-machine Dask experience or from just normal programming. It is great when you have data already in memory and want to get small fast results right to your local process.

```
>>> df = dd.read_csv('s3://...')
>>> df.value.sum().compute()
100000000
```

However, this approach often breaks down if you try to bring the entire dataset back to local RAM

```
>>> df.compute()
MemoryError(...)
```

It also forces you to wait until the computation finishes before handing back control of the interpreter.

3.11.2 Dask Collections to Futures

You can asynchronously submit lazy dask graphs to run on the cluster with the `e.compute` and `e.persist` methods. These functions return Future objects immediately. These futures can then be queried to determine the state of the computation.

e.compute

The `.compute` method takes a collection and returns a single future.

```
>>> df = dd.read_csv('s3://...')
>>> total = e.compute(df.sum())  # Return a single future
>>> total
Future(..., status='pending')

>>> total.result()               # Block until finished
100000000
```

Because this is a single future the result must fit on a single worker machine. Like `dask.compute` above, the `e.compute` method is only appropriate when results are small and should fit in memory. The following would likely fail:

```
>>> future = e.compute(df)       # Blows up memory
```

Instead, you should use `e.persist`

e.persist

The `.persist` method submits the task graph behind the Dask collection to the scheduler, obtaining Futures for all of the top-most tasks (for example one Future for each Pandas DataFrame in a Dask DataFrame). It then returns a copy of the collection pointing to these futures instead of the previous graph. This new collection is semantically equivalent but now points to actively running data rather than a lazy graph. If you look at the dask graph within the collection you will see the Future objects directly:

```
>>> df = dd.read_csv('s3://...')
>>> df.dask                          # Recipe to compute df in chunks
{('read', 0): (load_s3_bytes, ...),
 ('parse', 0): (pd.read_csv, ('read', 0)),
 ('read', 1): (load_s3_bytes, ...),
 ('parse', 1): (pd.read_csv, ('read', 1)),
 ...
}

>>> df = e.persist(df)               # Start computation
>>> df.dask                          # Now points to running futures
{('parse', 0): Future(..., status='finished'),
 ('parse', 1): Future(..., status='pending'),
 ...
}
```

The collection is returned immediately and the computation happens in the background on the cluster. Eventually all of the futures of this collection will be completed at which point further queries on this collection will likely be very fast.

Typically the workflow is to define a computation with a tool like `dask.dataframe` or `dask.delayed` until a point where you have a nice dataset to work from, then persist that collection to the cluster and then perform many fast queries off of the resulting collection.

3.11.3 Concrete Values to Futures

We obtain futures through a few different ways. One is the mechanism above, by wrapping Futures within Dask collections. Another is by submitting data or tasks directly to the cluster with `e.scatter`, `e.submit` or `e.map`.

```
futures = e.scatter(args)                        # Send data
future = e.submit(function, *args, **kwrags)     # Send single task
futures = e.map(function, sequence, **kwargs)    # Send many tasks
```

In this case `*args` or `**kwargs` can be normal Python objects, like `1` or `'hello'`, or they can be other `Future` objects if you want to link tasks together with dependencies.

Unlike Dask collections like dask.delayed these task submissions happen immediately. The concurrent.futures interface is very similar to dask.delayed except that execution is immediate rather than lazy.

3.11.4 Futures to Concrete Values

You can turn an individual `Future` into a concrete value in the local process by calling the `Future.result()` method. You can convert a collection of futures into concrete values by calling the `e.gather` method.

```
>>> future.result()
1

>>> e.gather(futures)
[1, 2, 3, 4, ...]
```

3.11.5 Futures to Dask Collections

As seen in the Collection to futures section it is common to have currently computing `Future` objects within Dask graphs. This lets us build further computations on top of currently running computations. This is most often done with dask.delayed workflows on custom computations:

```
>>> x = delayed(sum)(futures)
>>> y = delayed(product)(futures)
>>> future = e.compute(x + y)
```

Mixing the two forms allow you to build and submit a computation in stages like `sum(...) + product(...)`. This is often valuable if you want to wait to see the values of certain parts of the computation before determining how to proceed. Submitting many computations at once allows the scheduler to be slightly more intelligent when determining what gets run.

If this page interests you then you may also want to check out the doc page on Managing Memory

3.12 Managing Memory

Dask.distributed stores the results of tasks in the distributed memory of the worker nodes. The central scheduler tracks all data on the cluster and determines when data should be freed. Completed results are usually cleared from memory as quickly as possible in order to make room for more computation. The result of a task is kept in memory if either of the following conditions hold:

1. A client holds a future pointing to this task. The data should stay in RAM so that the client can gather the data on demand.
2. The task is necessary for ongoing computations that are working to produce the final results pointed to by futures. These tasks will be removed once no ongoing tasks require them.

When users hold Future objects or persisted collections (which contain many such Futures inside their `.dask` attribute) they pin those results to active memory. When the user deletes futures or collections from their local Python process the scheduler removes the associated data from distributed RAM. Because of this relationship, distributed memory reflects the state of local memory. A user may free distributed memory on the cluster by deleting persisted collections in the local session.

3.12.1 Creating Futures

The following functions produce Futures

`Executor.submit`(func, *args, **kwargs)	Submit a function application to the scheduler
`Executor.map`(func, *iterables, **kwargs)	Map a function on a sequence of arguments
`Executor.compute`(args[, sync, optimize_graph])	Compute dask collections on cluster
`Executor.persist`(collections[, optimize_graph])	Persist dask collections on cluster
`Executor.scatter`(data[, workers, broadcast, ...])	Scatter data into distributed memory

Submit and map handle raw Python functions. Compute and persist handle Dask collections like arrays, bags, delayed values, and dataframes. Scatter sends data directly from the local process.

3.12.2 Persisting Collections

Calls to `Executor.compute` or `Executor.persist` submit task graphs to the cluster and return `Future` objects that point to particular output tasks.

Compute returns a single future per input, persist returns a copy of the collection with each block or partition replaced by a single future. In short, use `persist` to keep full collection on the cluster and use `compute` when you want a small result as a single future.

Persist is more common and is often used as follows with collections:

```
>>> # Construct dataframe, no work happens
>>> df = dd.read_csv(...)
>>> df = df[df.x > 0]
>>> df = df.assign(z = df.x + df.y)

>>> # Pin data in distributed ram, this triggers computation
>>> df = e.persist(df)

>>> # continue operating on df
```

Note for Spark users: this differs from what you're accustomed to. Persist is an immediate action. However, you'll get control back immediately as computation occurs in the background.

In this example we build a computation by parsing CSV data, filtering rows, and then adding a new column. Up until this point all work is lazy; we've just built up a recipe to perform the work as a graph in the `df` object.

When we call `df = e.persist(df)` we cut this graph off of the `df` object, send it up to the scheduler, receive `Future` objects in return and create a new dataframe with a very shallow graph that points directly to these futures. This happens more or less immediately (as long as it takes to serialize and send the graph) and we can continue working on our new `df` object while the cluster works to evaluate the graph in the background.

3.12.3 Difference with dask.compute

The operations `e.persist(df)` and `e.compute(df)` are asynchronous and so differ from the traditional `df.compute()` method or `dask.compute` function, which blocks until a result is available. The `.compute()` method does not persist any data on the cluster. The `.compute()` method also brings the entire result back to the local machine, so it is unwise to use it on large datasets. However, `.compute()` is very convenient for smaller results particularly because it does return concrete results in a way that most other tools expect.

Typically we use asynchronous methods like `e.persist` to set up large collections and then use `df.compute()` for fast analyses.

```
>>> # df.compute()   # This is bad and would likely flood local memory
>>> df = e.persist(df)     # This is good and asynchronously pins df
>>> df.x.sum().compute()   # This is good because the result is small
>>> future = e.compute(df.x.sum())  # This is also good but less intuitive
```

3.12.4 Clearing data

We remove data from distributed ram by removing the collection from our local process. Remote data is removed once all Futures pointing to that data are removed from all client machines.

```
>>> del df  # Deleting local data often deletes remote data
```

If this is the only copy then this will likely trigger the cluster to delete the data as well.

However if we have multiple copies or other collections based on this one then we'll have to delete them all.

```
>>> df2 = df[df.x < 10]
>>> del df  # would not delete data, because df2 still tracks the futures
```

3.12.5 Aggressively Clearing Data

To definitely remove a computation and all computations that depend on it you can always `cancel` the futures/collection.

```
>>> e.cancel(df)  # kills df, df2, and every other dependent computation
```

Alternatively, if you want a clean slate, you can restart the cluster. This clears all state and does a hard restart of all worker processes. It generally completes in around a second.

```
>>> e.restart()
```

3.12.6 Resilience

Results are not intentionally copied unless necessary for computations on other worker nodes. Resilience is achieved through recomputation by maintaining the provenance of any result. If a worker node goes down the scheduler is able to recompute all of its results. The complete graph for any desired Future is maintained until no references to that future exist.

3.12.7 Advanced techniques

At first the result of a task is not intentionally copied, but only persists on the node where it was originally computed or scattered. However result may be copied to another worker node in the course of normal computation if that result is required by another task that is intended to by run by a different worker. This occurs if a task requires two pieces of data on different machines (at least one must move) or through work stealing. In these cases it is the policy for the second machine to maintain its redundant copy of the data. This helps to organically spread around data that is in high demand.

However, advanced users may want to control the location, replication, and balancing of data more directly throughout the cluster. They may know ahead of time that certain data should be broadcast throughout the network or that their data has become particularly imbalanced, or that they want certain pieces of data to live on certain parts of their network. These considerations are not usually necessary.

Executor.rebalance([futures, workers])	Rebalance data within network
Executor.replicate(futures[, n, workers, ...])	Set replication of futures within network
Executor.scatter(data[, workers, broadcast, ...])	Scatter data into distributed memory

3.13 Joblib Frontend

Joblib is a library for simple parallel programming primarily developed and used by the Scikit Learn community. As of version 0.10.0 it contains a plugin mechanism to allow Joblib code to use other parallel frameworks to execute computations. The `distributed` scheduler implements such a plugin in the `distributed.joblib` module and registers it appropriately with Joblib. As a result, any joblib code (including many scikit-learn algorithms) will run on the distributed scheduler if you enclose it in a context manager as follows:

```python
import distributed.joblib
from joblib import Parallel, parallel_backend

with parallel_backend('distributed', scheduler_host='HOST:PORT'):
    # normal Joblib code
```

Note that scikit-learn bundles joblib internally, so if you want to specify the joblib backend you'll need to import `parallel_backend` from scikit-learn instead of `joblib`. As an example you might distributed a randomized cross validated parameter search as follows.

```python
import distributed.joblib
# Scikit-learn bundles joblib, so you need to import from
# `sklearn.externals.joblib` instead of `joblib` directly
from sklearn.externals.joblib import parallel_backend
from sklearn.datasets import load_digits
from sklearn.grid_search import RandomizedSearchCV
from sklearn.svm import SVC
import numpy as np

digits = load_digits()

param_space = {
    'C': np.logspace(-6, 6, 13),
    'gamma': np.logspace(-8, 8, 17),
    'tol': np.logspace(-4, -1, 4),
    'class_weight': [None, 'balanced'],
}

model = SVC(kernel='rbf')
search = RandomizedSearchCV(model, param_space, cv=3, n_iter=50, verbose=10)

with parallel_backend('distributed', scheduler_host='localhost:8786'):
    search.fit(digits.data, digits.target)
```

3.14 IPython Integration

Dask.distributed integrates with IPython in three ways:

1. You can launch a Dask.distributed cluster from an IPyParallel cluster
2. You can launch IPython kernels from Dask Workers and Schedulers to assist with debugging
3. They both support the common concurrent.futures interface

3.14.1 Launch Dask from IPyParallel

IPyParallel is IPython's distributed computing framework that allows you to easily manage many IPython engines on different computers.

An IPyParallel `Client` can launch a `dask.distributed` Scheduler and Workers on those IPython engines, effectively launching a full dask.distributed system.

This is possible with the Client.become_distributed method:

```
$ ipcluster start
```

```
>>> from ipyparallel import Client
>>> c = Client()  # connect to IPyParallel cluster

>>> e = c.become_distributed()  # start dask on top of IPyParallel
>>> e
<Executor: scheduler="127.0.0.1:59683" processes=8 cores=8>
```

3.14.2 Launch IPython within Dask Workers

It is sometimes convenient to inspect the `Worker` or `Scheduler` process interactively. Fortunately IPython gives us a way to launch interactive sessions within Python processes. This is available through the following methods:

Executor.start_ipython_workers([workers, ...])	Start IPython kernels on workers
Executor.start_ipython_scheduler([...])	Start IPython kernel on the scheduler

These methods start IPython kernels running in a separate thread within the specified Worker or Schedulers. These kernels are accessible either through IPython magics or a QT-Console.

Example with IPython Magics

```
>>> e.start_ipython_scheduler()
>>> %scheduler scheduler.processing
{'127.0.0.1:3595': ['inc-1', 'inc-2'],
 '127.0.0.1:53589': ['inc-2', 'add-5']}

>>> info = e.start_ipython_workers()
>>> %remote info['127.0.0.1:3595'] worker.active
{'inc-1', 'inc-2'}
```

Example with qt-console

You can also open up a full interactive IPython qt-console on the scheduler or each of the workers:

```
>>> e.start_ipython_scheduler(qtconsole=True)
>>> e.start_ipython_workers(qtconsole=True)
```

3.15 Publish Datasets

A *dataset* is a named reference to a Dask collection or list of futures that has been published to the cluster. It is available for any client to see and persists beyond the scope of an individual session.

3.15.1 Motivating Example

We load CSV data from S3, manipulate it, and then persist the data to the cluster.

```
from dask.distributed import Executor
e = Executor('scheduler-address:8786')

import dask.dataframe as dd
df = dd.read_csv('s3://my-bucket/*.csv')
df2 = df[df.balance < 0]
df2 = e.persist(df2)

>>> df2.head()
      name  balance
0    Alice     -100
1      Bob     -200
2  Charlie     -300
3   Dennis     -400
4    Edith     -500
```

To share this collection with a colleague we publish it under the name `'negative_accounts'`

```
e.publish_dataset(negative_accounts=df2)
```

Now any other client can connect to the scheduler and retrieve this published dataset.

```
>>> from dask.distributed import Executor
>>> e = Executor('scheduler-address:8786')

>>> e.list_datasets()
['negative_accounts']

>>> df = e.get_dataset('negative_accounts')
>>> df.head()
      name  balance
0    Alice     -100
1      Bob     -200
2  Charlie     -300
3   Dennis     -400
4    Edith     -500
```

This allows users to easily share results. It also allows for the persistence of important and commonly used datasets beyond a single session. Published datasets continue to reside in distributed memory even after all clients requesting them have disconnected.

3.15.2 Notes

Published collections are not automatically persisted. If you publish an un-persisted collection then others will still be able to get the collection from the scheduler, but operations on that collection will start from scratch. This allows you to publish views on data that do not permanently take up cluster memory but can be surprising if you expect "publishing" to automatically make a computed dataset rapidly available.

Any client can publish or unpublish a dataset.

Publishing too many large datasets can quickly consume a cluster's RAM.

3.15.3 API

Executor.publish_dataset(**kwargs)	Publish named datasets to scheduler
Executor.list_datasets()	List named datasets available on the scheduler
Executor.get_dataset(name)	Get named dataset from the scheduler
Executor.unpublish_dataset(name)	Remove named datasets from scheduler

3.16 Data Streams with Queues

The `Executor` methods `scatter`, `map`, and `gather` can consume and produce standard Python `Queue` objects. This is useful for processing continuous streams of data. However, it does not constitute a full streaming data processing pipeline like Storm.

3.16.1 Example

We connect to a local Executor.

```
>>> from distributed import Executor
>>> e = Executor('127.0.0.1:8786')
>>> e
<Executor: scheduler=127.0.0.1:8786 workers=1 threads=4>
```

We build a couple of toy data processing functions:

```
from time import sleep
from random import random

def inc(x):
    from random import random
    sleep(random() * 2)
    return x + 1

def double(x):
    from random import random
    sleep(random())
    return 2 * x
```

And we set up an input Queue and map our functions across it.

```
>>> from queue import Queue
>>> input_q = Queue()
>>> remote_q = e.scatter(input_q)
>>> inc_q = e.map(inc, remote_q)
>>> double_q = e.map(double, inc_q)
```

We will fill the input_q with local data from some stream, and then remote_q, inc_q and double_q will fill with Future objects as data gets moved around.

We gather the futures from the double_q back to a queue holding local data in the local process.

```
>>> result_q = e.gather(double_q)
```

Insert Data Manually

Because we haven't placed any data into any of the queues everything is empty, including the final output, result_q.

```
>>> result_q.qsize()
0
```

But when we insert an entry into the input_q, it starts to make its way through the pipeline and ends up in the result_q.

```
>>> input_q.put(10)
>>> result_q.get()
22
```

Insert data in a separate thread

We simulate a slightly more realistic situation by dumping data into the input_q in a separate thread. This simulates what you might get if you were to read from an active data source.

```
def load_data(q):
    i = 0
    while True:
        q.put(i)
        sleep(random())
        i += 1

 >>> from threading import Thread
 >>> load_thread = Thread(target=load_data, args=(input_q,))
 >>> load_thread.start()

 >>> result_q.qsize()
 4
 >>> result_q.qsize()
 9
```

We consume data from the result_q and print results to the screen.

```
>>> while True:
...     item = result_q.get()
...     print(item)
2
4
6
8
10
12
...
```

3.16.2 Limitations

- This doesn't do any sort of auto-batching of computations, so ideally you batch your data to take significantly longer than 1ms to run.
- This isn't a proper streaming system. There is no support outside of what you see here. In particular there are no policies for dropping data, joining over time windows, etc..

3.16.3 Extensions

We can extend this small example to more complex systems that have buffers, split queues, merge queues, etc. all by manipulating normal Python Queues.

Here are a couple of useful function to multiplex and merge queues:

```
from queue import Queue
from threading import Thread

def multiplex(n, q, **kwargs):
    """ Convert one queue into several equivalent Queues

    >>> q1, q2, q3 = multiplex(3, in_q)
    """
    out_queues = [Queue(**kwargs) for i in range(n)]
    def f():
        while True:
            x = q.get()
            for out_q in out_queues:
                out_q.put(x)
    t = Thread(target=f)
    t.daemon = True
    t.start()
    return out_queues

def push(in_q, out_q):
    while True:
        x = in_q.get()
        out_q.put(x)

def merge(*in_qs, **kwargs):
    """ Merge multiple queues together

    >>> out_q = merge(q1, q2, q3)
    """
    out_q = Queue(**kwargs)
    threads = [Thread(target=push, args=(q, out_q)) for q in in_qs]
    for t in threads:
        t.daemon = True
        t.start()
    return out_q
```

With useful functions like these we can build out more sophisticated data processing pipelines that split off and join back together. By creating queues with `maxsize=` we can control buffering and apply back pressure.

3.17 Launch Tasks from Tasks

Sometimes it is convenient to launch tasks from other tasks. For example you may not know what computations to run until you have the results of some initial computations.

3.17.1 Motivating example

We want to download one piece of data and turn it into a list. Then we want to submit one task for every element of that list. We don't know how long the list will be until we have the data.

So we send off our original `download_and_convert_to_list` function, which downloads the data and converts it to a list on one of our worker machines:

```
future = e.submit(download_and_convert_to_list, uri)
```

But now we need to submit new tasks for individual parts of this data. We have three options.

1. Gather the data back to the local process and then submit new jobs from the local process
2. Gather only enough information about the data back to the local process and submit jobs from the local process
3. Submit a task to the cluster that will submit other tasks directly from that worker

3.17.2 Gather the data locally

If the data is not large then we can bring it back to the client to perform the necessary logic on our local machine:

```
>>> data = future.result()                  # gather data to local process
>>> data                                    # data is a list
[...]

>>> futures = e.map(process_element, data)  # submit new tasks on data
>>> analysis = e.submit(aggregate, futures) # submit final aggregation task
```

This is straightforward and, if `data` is small then it is probably the simplest, and therefore correct choice. However, if `data` is large then we have to choose another option.

3.17.3 Submit tasks from client

We can run small functions on our remote data to determine enough to submit the right kinds of tasks. In the following example we compute the `len` function on `data` remotely and then break up data into its various elements.

```
>>> n = e.submit(len, data)                 # compute number of elements
>>> n = n.result()                          # gather n (small) locally

>>> from operator import getitem
>>> elements = [e.submit(getitem, data, i) for i in range(n)]  # split data

>>> futures = e.map(process_element, elements)
>>> analysis = e.submit(aggregate, futures)
```

We compute the length remotely, gather back this very small result, and then use it to submit more tasks to break up the data and process on the cluster. This is more complex because we had to go back and forth a couple of times between the cluster and the local process, but the data moved was very small, and so this only added a few milliseconds to our total processing time.

3.17.4 Submit tasks from worker

Note: this interface is new and experimental. It may be changed without warning in future versions.

Alternatively we submit tasks from other tasks. This allows us to make decisions while on worker nodes. To do this we will make a new client object on the worker itself. There is a convenience function for this that will connect you to the correct scheduler that the worker is connected to.

```
from distributed import local_executor

def process_all(data):
    with local_executor() as e:
        elements = e.scatter(data)
        futures = e.map(process_element, elements)
        analysis = e.submit(aggregate, futures)
        result = analysis.result()
    return result

 analysis = e.submit(process_all, data)  # spawns many tasks
```

This approach is more complex, but very powerful. It allows you to spawn tasks that themselves act as potentially long-running clients, managing their own independent workloads.

Extended Example

This example computing the fibonacci numbers creates tasks that submit tasks that submit tasks that submit other tasks, etc..

‘‘‘python In [1]: from distributed import Executor, local_executor

In [2]: e = Executor()

In [3]: def fib(n): ...: if n < 2: ...: return n ...: else: ...: with local_executor() as ee: ...: a = ee.submit(fib, n - 1) ...: b = ee.submit(fib, n - 2) ...: a, b = ee.gather([a, b]) ...: return a + b ...:

In [4]: future = e.submit(fib, 100)

In [5]: future Out[5]: <Future: status: finished, type: int, key: fib-7890e9f06d5f4e0a8fc7ec5c77590ace>

In [6]: future.result() Out[6]: 354224848179261915075 ‘‘‘

Technical details

Tasks that invoke `local_executor` are conservatively assumed to be *long running*. They can take a long time blocking, waiting for other tasks to finish. In order to avoid having them take up processing slots the following actions occur whenever a task invokes `local_executor`.

1. The thread on the worker that runs this functions *secedes* from the thread pool and goes off on its own. This allows the thread pool to populate that slot with a new thread and continue processing tasks without counting this long running task against its normal quota.
2. The Worker sends a message back to the scheduler temporarily increasing its allowed number of tasks by one. This likewise lets the scheduler allocate more tasks to this worker, not counting this long running task against it.

Because of this behavior you can happily launch long running control tasks that manage worker-side clients happily, without fear of deadlocking the cluster.

3.18 Submitting Applications

The dask-submit cli can be used to submit an application to the dask cluster running remotely. If your code depends on resources that can only be access from cluster running dask, dask-submit provides a mechanism to send the script to the cluster for execution from a different machine.

For example, S3 buckets could not be visible from your local machine and hence any attempt to create a dask graph from local machine may not be work.

3.18.1 Submitting dask Applications with *dask-submit*

In order to remotely submit scripts to the cluster from a local machine or a CI/CD environment, we need to run a remote client on the same machine as the scheduler:

```
#scheduler machine
dask-remote --port 8788
```

After making sure the *dask-remote* is running, you can submit a script by:

```
#local machine
dask-submit <dask-remote-address>:<port> <script.py>
```

Some of the commonly used arguments are:

- REMOTE_CLIENT_ADDRESS: host name where dask-remote client is running
- FILEPATH: Local path to file containing dask application

For example, given the following dask application saved in a file called script.py:

```python
from distributed import Executor

def inc(x):
     return x + 1

if __name__=='__main__':
     executor = Executor('127.0.0.1:8786')
     x = executor.submit(inc, 10)
     print(x.result())
```

We can submit this application from a local machine by running:

```
dask-submit <remote-client-address>:<port> script.py
```

3.19 API

Executor

Executor([address, start, loop, timeout, ...])	Drive computations on a distributed cluster
Executor.cancel(futures)	Cancel running futures
Executor.compute(args[, sync, optimize_graph])	Compute dask collections on cluster
Executor.gather(futures[, errors, maxsize])	Gather futures from distributed memory
Executor.get(dsk, keys[, restrictions, ...])	Compute dask graph
Executor.get_dataset(name)	Get named dataset from the scheduler

Continued on next page

Table 3.5 – continued from previous page

`Executor.has_what`([workers])	Which keys are held by which workers
`Executor.list_datasets`()	List named datasets available on the scheduler
`Executor.map`(func, *iterables, **kwargs)	Map a function on a sequence of arguments
`Executor.ncores`([workers])	The number of threads/cores available on each worker node
`Executor.persist`(collections[, optimize_graph])	Persist dask collections on cluster
`Executor.publish_dataset`(**kwargs)	Publish named datasets to scheduler
`Executor.rebalance`([futures, workers])	Rebalance data within network
`Executor.replicate`(futures[, n, workers, ...])	Set replication of futures within network
`Executor.restart`()	Restart the distributed network
`Executor.run`(function, *args, **kwargs)	Run a function on all workers outside of task scheduling system
`Executor.scatter`(data[, workers, broadcast, ...])	Scatter data into distributed memory
`Executor.shutdown`([timeout])	Send shutdown signal and wait until scheduler terminates
`Executor.start_ipython_workers`([workers, ...])	Start IPython kernels on workers
`Executor.start_ipython_scheduler`([...])	Start IPython kernel on the scheduler
`Executor.submit`(func, *args, **kwargs)	Submit a function application to the scheduler
`Executor.unpublish_dataset`(name)	Remove named datasets from scheduler
`Executor.upload_file`(filename)	Upload local package to workers
`Executor.who_has`([futures])	The workers storing each future's data

Future

`Future`(key, executor)	A remotely running computation
`Future.cancel`()	Returns True if the future has been cancelled
`Future.cancelled`()	Returns True if the future has been cancelled
`Future.done`()	Is the computation complete?
`Future.exception`()	Return the exception of a failed task
`Future.result`()	Wait until computation completes.
`Future.traceback`()	Return the traceback of a failed task

Other

`as_completed`(fs)	Return futures in the order in which they complete
`distributed.diagnostics.progress`(*futures, ...)	Track progress of futures
`wait`(fs[, timeout, return_when])	Wait until all futures are complete

3.19.1 Asynchronous methods

If you desire Tornado coroutines rather than typical functions these can commonly be found as underscore-prefixed versions of the functions above. For example the `e.restart()` method can be replaced in an asynchronous workflow with `yield e._restart()`. Many methods like `e.compute` are non-blocking regardless; these do not have a coroutine-equivalent.

```
e.restart()  # synchronous
yield e._restart()  # non-blocking
```

3.19.2 Executor

class `distributed.executor.`**`Executor`**(*address=None*, *start=True*, *loop=None*, *timeout=3*, *set_as_default=True*)

Drive computations on a distributed cluster

The Executor connects users to a distributed compute cluster. It provides an asynchronous user interface around functions and futures. This class resembles executors in `concurrent.futures` but also allows `Future` objects within `submit/map` calls.

Parameters address: string, tuple, or ‘‘Scheduler‘‘

This can be the address of a `Scheduler` server, either as a string `'127.0.0.1:8787'` or tuple `('127.0.0.1', 8787)` or it can be a local `Scheduler` object.

See also:

`distributed.scheduler.Scheduler` Internal scheduler

Examples

Provide cluster's head node address on initialization:

```
>>> executor = Executor('127.0.0.1:8787')
```

Use `submit` method to send individual computations to the cluster

```
>>> a = executor.submit(add, 1, 2)
>>> b = executor.submit(add, 10, 20)
```

Continue using submit or map on results to build up larger computations

```
>>> c = executor.submit(add, a, b)
```

Gather results with the `gather` method.

```
>>> executor.gather([c])
33
```

`cancel`(*futures*)

Cancel running futures

This stops future tasks from being scheduled if they have not yet run and deletes them if they have already run. After calling, this result and all dependent results will no longer be accessible

Parameters futures: list of Futures

`compute`(*args*, *sync=False*, *optimize_graph=True*, ***kwargs*)

Compute dask collections on cluster

Parameters args: iterable of dask objects or single dask object

Collections like dask.array or dataframe or dask.value objects

sync: bool (optional)

Returns Futures if False (default) or concrete values if True

optimize_graph: bool

Whether or not to optimize the underlying graphs

kwargs:

Options to pass to the graph optimize calls

Returns List of Futures if input is a sequence, or a single future otherwise

See also:

Executor.get Normal synchronous dask.get function

Examples

```
>>> from dask import do, value
>>> from operator import add
>>> x = dask.do(add)(1, 2)
>>> y = dask.do(add)(x, x)
>>> xx, yy = executor.compute([x, y])
>>> xx
<Future: status: finished, key: add-8f6e709446674bad78ea8aeecfee188e>
>>> xx.result()
3
>>> yy.result()
6
```

Also support single arguments

```
>>> xx = executor.compute(x)
```

`gather` (*futures*, *errors='raise'*, *maxsize=0*)
Gather futures from distributed memory

Accepts a future, nested container of futures, iterator, or queue. The return type will match the input type.

Returns Future results

See also:

Executor.scatter Send data out to cluster

Examples

```
>>> from operator import add
>>> e = Executor('127.0.0.1:8787')
>>> x = e.submit(add, 1, 2)
>>> e.gather(x)
3
>>> e.gather([x, [x], x])  # support lists and dicts
[3, [3], 3]
```

```
>>> seq = e.gather(iter([x, x]))  # support iterators
>>> next(seq)
3
```

`get` (*dsk*, *keys*, *restrictions=None*, *loose_restrictions=None*, ***kwargs*)
Compute dask graph

Parameters dsk: dict

keys: object, or nested lists of objects

restrictions: dict (optional)

A mapping of {key: {set of worker hostnames}} that restricts where jobs can take place

See also:

Executor.compute Compute asynchronous collections

Examples

```
>>> from operator import add
>>> e = Executor('127.0.0.1:8787')
>>> e.get({'x': (add, 1, 2)}, 'x')
3
```

get_dataset(*name*)

Get named dataset from the scheduler

See also:

Executor.publish_dataset, *Executor.list_datasets*

has_what(*workers=None*)

Which keys are held by which workers

Parameters workers: list (optional)

A list of worker addresses, defaults to all

See also:

Executor.who_has, *Executor.ncores*

Examples

```
>>> x, y, z = e.map(inc, [1, 2, 3])
>>> wait([x, y, z])
>>> e.has_what()
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
                         'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
                         'inc-1e297fc27658d7b67b3a758f16bcf47a']}
```

list_datasets()

List named datasets available on the scheduler

See also:

Executor.publish_dataset, *Executor.get_dataset*

map(*func*, **iterables*, ***kwargs*)

Map a function on a sequence of arguments

Arguments can be normal objects or Futures

Parameters func: callable

iterables: Iterables, Iterators, or Queues

pure: bool (defaults to True)

Whether or not the function is pure. Set `pure=False` for impure functions like `np.random.random`.

workers: set, iterable of sets

A set of worker hostnames on which computations may be performed. Leave empty to default to all workers (common case)

Returns List, iterator, or Queue of futures, depending on the type of the

inputs.

See also:

Executor.submit Submit a single function

Examples

```
>>> L = executor.map(func, sequence)
```

nbytes (*keys=None*, *summary=True*)

The bytes taken up by each key on the cluster

This is as measured by `sys.getsizeof` which may not accurately reflect the true cost.

Parameters keys: list (optional)

A list of keys, defaults to all keys

summary: boolean, (optional)

Summarize keys into key types

See also:

Executor.who_has

Examples

```
>>> x, y, z = e.map(inc, [1, 2, 3])
>>> e.nbytes(summary=False)
{'inc-1c8dd6be1c21646c71f76c16d09304ea': 28,
 'inc-1e297fc27658d7b67b3a758f16bcf47a': 28,
 'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b': 28}
```

```
>>> e.nbytes(summary=True)
{'inc': 84}
```

ncores (*workers=None*)

The number of threads/cores available on each worker node

Parameters workers: list (optional)

A list of workers that we care about specifically. Leave empty to receive information about all workers.

See also:

Executor.who_has, *Executor.has_what*

Examples

```
>>> e.ncores()
{'192.168.1.141:46784': 8,
 '192.167.1.142:47548': 8,
 '192.167.1.143:47329': 8,
 '192.167.1.144:37297': 8}
```

persist(*collections*, *optimize_graph=True*, ***kwargs*)
Persist dask collections on cluster

Starts computation of the collection on the cluster in the background. Provides a new dask collection that is semantically identical to the previous one, but now based off of futures currently in execution.

Parameters collections: sequence or single dask object

Collections like dask.array or dataframe or dask.value objects

optimize_graph: bool

Whether or not to optimize the underlying graphs

kwargs:

Options to pass to the graph optimize calls

Returns List of collections, or single collection, depending on type of input.

See also:

Executor.compute

Examples

```
>>> xx = executor.persist(x)
>>> xx, yy = executor.persist([x, y])
```

processing(*workers=None*)
The tasks currently running on each worker

Parameters workers: list (optional)

A list of worker addresses, defaults to all

See also:

Executor.stacks, *Executor.who_has*, *Executor.has_what*, *Executor.ncores*

Examples

```
>>> x, y, z = e.map(inc, [1, 2, 3])
>>> e.processing()
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
                         'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
                         'inc-1e297fc27658d7b67b3a758f16bcf47a']}
```

publish_dataset(***kwargs*)
Publish named datasets to scheduler

This stores a named reference to a dask collection or list of futures on the scheduler. These references are available to other executors which can download the collection or futures with `get_dataset`.

Datasets are not immediately computed. You may wish to call `Executor.persist` prior to publishing a dataset.

Parameters **kwargs: dict**

named collections to publish on the scheduler

Returns None

See also:

`Executor.list_datasets`, *`Executor.get_dataset`*, *`Executor.unpublish_dataset`*, *`Executor.persist`*

Examples

Publishing client: >>> df = dd.read_csv('s3://...') # doctest: +SKIP >>> df = e.persist(df) # doctest: +SKIP >>> e.publish_dataset(my_dataset=df) # doctest: +SKIP

Receiving client: >>> e.list_datasets() # doctest: +SKIP ['my_dataset'] >>> df2 = e.get_dataset('my_dataset') # doctest: +SKIP

`rebalance` (*futures=None*, *workers=None*)
Rebalance data within network

Move data between workers to roughly balance memory burden. This either affects a subset of the keys/workers or the entire network, depending on keyword arguments.

This operation is generally not well tested against normal operation of the scheduler. It it not recommended to use it while waiting on computations.

Parameters **futures: list, optional**

A list of futures to balance, defaults all data

workers: list, optional

A list of workers on which to balance, defaults to all workers

`replicate` (*futures*, *n=None*, *workers=None*, *branching_factor=2*)
Set replication of futures within network

This performs a tree copy of the data throughout the network individually on each piece of data.

This operation blocks until complete. It does not guarantee replication of data to future workers.

Parameters **futures: list of futures**

Futures we wish to replicate

n: int, optional

Number of processes on the cluster on which to replicate the data. Defaults to all.

workers: list of worker addresses

Workers on which we want to restrict the replication. Defaults to all.

branching_factor: int, optional

The number of workers that can copy data in each generation

See also:

`Executor.rebalance`

Examples

```
>>> x = e.submit(func, *args)
>>> e.replicate([x])  # send to all workers
>>> e.replicate([x], n=3)  # send to three workers
>>> e.replicate([x], workers=['alice', 'bob'])  # send to specific
>>> e.replicate([x], n=1, workers=['alice', 'bob'])  # send to one of specific workers
>>> e.replicate([x], n=1)  # reduce replications
```

restart()
Restart the distributed network

This kills all active work, deletes all data on the network, and restarts the worker processes.

run(*function*, **args*, ***kwargs*)
Run a function on all workers outside of task scheduling system

This calls a function on all currently known workers immediately, blocks until those results come back, and returns the results asynchronously as a dictionary keyed by worker address. This method if generally used for side effects, such and collecting diagnostic information or installing libraries.

Parameters function: callable

***args: arguments for remote function**

****kwargs: keyword arguments for remote function**

workers: list

Workers on which to run the function. Defaults to all known workers.

Examples

```
>>> e.run(os.getpid)
{'192.168.0.100:9000': 1234,
 '192.168.0.101:9000': 4321,
 '192.168.0.102:9000': 5555}
```

Restrict computation to particular workers with the `workers=` keyword argument.

```
>>> e.run(os.getpid, workers=['192.168.0.100:9000',
...                           '192.168.0.101:9000'])
{'192.168.0.100:9000': 1234,
 '192.168.0.101:9000': 4321}
```

scatter(*data*, *workers=None*, *broadcast=False*, *maxsize=0*)
Scatter data into distributed memory

Parameters data: list, iterator, dict, or Queue

Data to scatter out to workers. Output type matches input type.

workers: list of tuples (optional)

Optionally constrain locations of data. Specify workers as hostname/port pairs, e.g. `('127.0.0.1', 8787)`.

broadcast: bool (defaults to False)

Whether to send each data element to all workers. By default we round-robin based on number of cores.

maxsize: int (optional)

Maximum size of queue if using queues, 0 implies infinite

Returns List, dict, iterator, or queue of futures matching the type of input.

See also:

Executor.gather Gather data back to local process

Examples

```
>>> e = Executor('127.0.0.1:8787')
>>> e.scatter([1, 2, 3])
[<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>,
 <Future: status: finished, key: 58e78e1b34eb49a68c65b54815d1b158>,
 <Future: status: finished, key: d3395e15f605bc35ab1bac6341a285e2>]
```

```
>>> e.scatter({'x': 1, 'y': 2, 'z': 3})
{'x': <Future: status: finished, key: x>,
 'y': <Future: status: finished, key: y>,
 'z': <Future: status: finished, key: z>}
```

Constrain location of data to subset of workers >>> e.scatter([1, 2, 3], workers=[('hostname', 8788)]) # doctest: +SKIP

Handle streaming sequences of data with iterators or queues >>> seq = e.scatter(iter([1, 2, 3])) # doctest: +SKIP >>> next(seq) # doctest: +SKIP <Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>,

Broadcast data to all workers >>> [future] = e.scatter([element], broadcast=True) # doctest: +SKIP

scheduler_info()

Basic information about the workers in the cluster

Examples

```
>>> e.scheduler_info()
{'id': '2de2b6da-69ee-11e6-ab6a-e82aea155996',
 'services': {},
 'type': 'Scheduler',
 'workers': {'127.0.0.1:40575': {'active': 0,
                                 'last-seen': 1472038237.4845693,
                                 'name': '127.0.0.1:40575',
                                 'services': {},
                                 'stored': 0,
                                 'time-delay': 0.0061032772064208984}}}
```

shutdown(*timeout=10*)

Send shutdown signal and wait until scheduler terminates

stacks(*workers=None*)

The task queues on each worker

Parameters workers: list (optional)

A list of worker addresses, defaults to all

See also:

Executor.processing, *Executor.who_has*, *Executor.has_what*, *Executor.ncores*

Examples

```
>>> x, y, z = e.map(inc, [1, 2, 3])
>>> e.stacks()
{'192.168.1.141:46784': ['inc-1c8dd6be1c21646c71f76c16d09304ea',
                         'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b',
                         'inc-1e297fc27658d7b67b3a758f16bcf47a']}
```

start(***kwargs*)

Start scheduler running in separate thread

start_ipython_scheduler(*magic_name='scheduler_if_ipython', qtconsole=False, qtconsole_args=None*)

Start IPython kernel on the scheduler

Parameters magic_name: str or None (optional)

If defined, register IPython magic with this name for executing code on the scheduler. If not defined, register %scheduler magic if IPython is running.

qtconsole: bool (optional)

If True, launch a Jupyter QtConsole connected to the worker(s).

qtconsole_args: list(str) (optional)

Additional arguments to pass to the qtconsole on startup.

Returns connection_info: dict

connection_info dict containing info necessary to connect Jupyter clients to the scheduler.

See also:

Executor.start_ipython_workers Start IPython on the workers

Examples

```
>>> e.start_ipython_scheduler()
>>> %scheduler scheduler.processing
{'127.0.0.1:3595': {'inc-1', 'inc-2'},
 '127.0.0.1:53589': {'inc-2', 'add-5'}}
```

```
>>> e.start_ipython_scheduler(qtconsole=True)
```

start_ipython_workers(*workers=None, magic_names=False, qtconsole=False, qtconsole_args=None*)

Start IPython kernels on workers

Parameters workers: list (optional)

A list of worker addresses, defaults to all

magic_names: str or list(str) (optional)

If defined, register IPython magics with these names for executing code on the workers.

qtconsole: bool (optional)

If True, launch a Jupyter QtConsole connected to the worker(s).

qtconsole_args: list(str) (optional)

Additional arguments to pass to the qtconsole on startup.

Returns iter_connection_info: list

List of connection_info dicts containing info necessary to connect Jupyter clients to the workers.

See also:

`Executor.start_ipython_scheduler` start ipython on the scheduler

Examples

```
>>> info = e.start_ipython_workers()
>>> %remote info['192.168.1.101:5752'] worker.data
{'x': 1, 'y': 100}
```

```
>>> e.start_ipython_workers('192.168.1.101:5752', magic_names='w')
>>> %w worker.data
{'x': 1, 'y': 100}
```

```
>>> e.start_ipython_workers('192.168.1.101:5752', qtconsole=True)
```

`submit` (*func*, **args*, ***kwargs*)
Submit a function application to the scheduler

Parameters **func: callable**

***args:**

****kwargs:**

pure: bool (defaults to True)

Whether or not the function is pure. Set `pure=False` for impure functions like `np.random.random`.

workers: set, iterable of sets

A set of worker hostnames on which computations may be performed. Leave empty to default to all workers (common case)

allow_other_workers: bool (defaults to False)

Used with *workers*. Inidicates whether or not the computations may be performed on workers that are not in the *workers* set(s).

Returns Future

See also:

`Executor.map` Submit on many arguments at once

Examples

```
>>> c = executor.submit(add, a, b)
```

unpublish_dataset (*name*)

Remove named datasets from scheduler

See also:

Executor.publish_dataset

Examples

```
>>> e.list_datasets()
['my_dataset']
>>> e.unpublish_datasets('my_dataset')
>>> e.list_datasets()
[]
```

upload_file (*filename*)

Upload local package to workers

This sends a local file up to all worker nodes. This file is placed into a temporary directory on Python's system path so any .py or .egg files will be importable.

Parameters filename: string

Filename of .py or .egg file to send to workers

Examples

```
>>> executor.upload_file('mylibrary.egg')
>>> from mylibrary import myfunc
>>> L = e.map(myfunc, seq)
```

who_has (*futures=None*)

The workers storing each future's data

Parameters futures: list (optional)

A list of futures, defaults to all data

See also:

Executor.has_what, *Executor.ncores*

Examples

```
>>> x, y, z = e.map(inc, [1, 2, 3])
>>> wait([x, y, z])
>>> e.who_has()
{'inc-1c8dd6be1c21646c71f76c16d09304ea': ['192.168.1.141:46784'],
 'inc-1e297fc27658d7b67b3a758f16bcf47a': ['192.168.1.141:46784'],
 'inc-fd65c238a7ea60f6a01bf4c8a5fcf44b': ['192.168.1.141:46784']}
```

```
>>> e.who_has([x, y])
{'inc-1c8dd6be1c21646c71f76c16d09304ea': ['192.168.1.141:46784'],
 'inc-1e297fc27658d7b67b3a758f16bcf47a': ['192.168.1.141:46784']}
```

3.19.3 CompatibleExecutor

class `distributed.executor.`**`CompatibleExecutor`**(*address=None*, *start=True*, *loop=None*, *timeout=3*, *set_as_default=True*)

A concurrent.futures-compatible Executor

A subclass of Executor that conforms to concurrent.futures API, allowing swapping in for other Executors.

`map`(*func*, **iterables*, ***kwargs*)

Map a function on a sequence of arguments

Returns iter_results: iterable

Iterable yielding results of the map.

See also:

`Executor.map` for more info

3.19.4 Future

class `distributed.executor.`**`Future`**(*key*, *executor*)

A remotely running computation

A Future is a local proxy to a result running on a remote worker. A user manages future objects in the local Python process to determine what happens in the larger cluster.

See also:

`Executor` Creates futures

Examples

Futures typically emerge from Executor computations

```
>>> my_future = executor.submit(add, 1, 2)
```

We can track the progress and results of a future

```
>>> my_future
<Future: status: finished, key: add-8f6e709446674bad78ea8aeecfee188e>
```

We can get the result or the exception and traceback from the future

```
>>> my_future.result()
```

`cancel`()

Returns True if the future has been cancelled

`cancelled`()

Returns True if the future has been cancelled

done()
Is the computation complete?

exception()
Return the exception of a failed task

See also:

Future.traceback

result()
Wait until computation completes. Gather result to local process

traceback()
Return the traceback of a failed task

This returns a traceback object. You can inspect this object using the `traceback` module. Alternatively if you call `future.result()` this traceback will accompany the raised exception.

See also:

Future.exception

Examples

```
>>> import traceback
>>> tb = future.traceback()
>>> traceback.export_tb(tb)
[...]
```

3.19.5 Other

distributed.executor.**as_completed**(*fs*)
Return futures in the order in which they complete

This returns an iterator that yields the input future objects in the order in which they complete. Calling `next` on the iterator will block until the next future completes, irrespective of order.

This function does not return futures in the order in which they are input.

distributed.diagnostics.**progress**(**futures*, ***kwargs*)
Track progress of futures

This operates differently in the notebook and the console

- Notebook: This returns immediately, leaving an IPython widget on screen
- Console: This blocks until the computation completes

Parameters futures: Futures

A list of futures or keys to track

notebook: bool (optional)

Running in the notebook or not (defaults to guess)

multi: bool (optional)

Track different functions independently (defaults to True)

complete: bool (optional)

Track all keys (True) or only keys that have not yet run (False) (defaults to True)

Examples

```
>>> progress(futures)
[########################################] | 100% Completed |  1.7s
```

`distributed.executor.`**`wait`** (*fs*, *timeout=None*, *return_when='ALL_COMPLETED'*)
Wait until all futures are complete

Parameters fs: list of futures

Returns Named tuple of completed, not completed

3.20 Frequently Asked Questions

3.20.1 Too many open file descriptors?

Your operating system imposes a limit to how many open files or open network connections any user can have at once. Depending on the scale of your cluster the `dask-scheduler` may run into this limit.

By default most Linux distributions set this limit at 1024 open files/connections and OS-X at 128 or 256. Each worker adds a few open connections to a running scheduler (somewhere between one and ten, depending on how contentious things get.)

If you are on a managed cluster you can usually ask whoever manages your cluster to increase this limit. If you have root access and know what you are doing you can change the limits on Linux by editing `/etc/security/limits.conf`. Instructions are here under the heading "User Level FD Limits": http://www.cyberciti.biz/faq/linux-increase-the-maximum-number-of-open-files/

3.21 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 avaialble at the Dask developer guidelines in the main documentation.

3.21.1 Install

After setting up an environment as described in the Dask developer guidelines you can clone this repository with git:

```
git clone git@github.com:dask/distributed.git
```

and install it from source:

```
cd distributed
python setup.py install
```

3.21.2 Test

Test using `py.test`:

```
py.test distributed --verbose
```

3.21.3 Tornado

Dask.distributed is a Tornado TCP application. Tornado provides us with both a communication layer on top of sockets, as well as a syntax for writing asynchronous coroutines, similar to asyncio. You can make modest changes to the policies within this library without understanding much about Tornado, however moderate changes will probably require you to understand Tornado IOLoops, coroutines, and a little about non-blocking communication.. The Tornado API docuemntation is quite good and we recommend that you read the following resources:

- http://www.tornadoweb.org/en/stable/gen.html
- http://www.tornadoweb.org/en/stable/ioloop.html

Additionally, if you want to interact at a low level with the communication between workers and scheduler then you should understand the Tornado `TCPServer` and `IOStream` available here:

- http://www.tornadoweb.org/en/stable/networking.html

Dask.distributed wraps a bit of logic around Tornado. See Foundations for more information.

3.21.4 Writing Tests

Testing distributed systems is normally quite difficult because it is difficult to inspect the state of all components when something goes wrong. Fortunately, the non-blocking asynchronous model within Tornado allows us to run a scheduler, multiple workers, and multiple clients all within a single thread. This gives us predictable performance, clean shutdowns, and the ability to drop into any point of the code during execution. At the same time, sometimes we want everything to run in different processes in order to simulate a more realistic setting.

The test suite contains three kinds of tests

1. `@gen_cluster`: Fully asynchronous tests where all components live in the same event loop in the main thread. These are good for testing complex logic and inspecting the state of the system directly. They are also easier to debug and cause the fewest problems with shutdowns.
2. `with cluster()`: Tests with multiple processes forked from the master process. These are good for testing the synchronous (normal user) API and when triggering hard failures for resilience tests.
3. `popen`: Tests that call out to the command line to start the system. These are rare and mostly for testing the command line interface.

If you are comfortable with the Tornado interface then you will be happiest using the `@gen_cluster` style of test

```
@gen_cluster(executor=True)
def test_submit(e, s, a, b):
    assert isinstance(e, Executor)
    assert isinstance(e, Scheduler)
    assert isinstance(a, Worker)
    assert isinstance(b, Worker)

    future = e.submit(inc, 1)
    assert future.key in e.futures

    # result = future.result()  # This synchronous API call would block
    result = yield future._result()
    assert result == 2

    assert future.key in s.tasks
    assert future.key in a.data or future.key in b.data
```

The `@gen_cluster` decorator sets up a scheduler, executor, and workers for you and cleans them up after the test. It also allows you to directly inspect the state of every element of the cluster directly. However, you can not use the normal synchronous API (doing so will cause the test to wait forever) and instead you need to use the coroutine API, where all blocking functions are prepended with an underscore (_). Beware, it is a common mistake to use the blocking interface within these tests.

If you want to test the normal synchronous API you can use a `with cluster` style test, which sets up a scheduler and workers for you in different forked processes:

```
def test_submit_sync(loop):
    with cluster() as (s, [a, b]):
        with Executor(('127.0.0.1', s['port']), loop=loop) as e:
            future = e.submit(inc, 1)
            assert future.key in e.futures

            result = future.result()  # use the synchronous/blocking API here
            assert result == 2

            a['proc'].terminate()  # kill one of the workers

            result = future.result()  # test that future remains valid
            assert result == 2
```

In this style of test you do not have access to the scheduler or workers. The variables `s, a, b` are now dictionaries holding a `multiprocessing.Process` object and a port integer. However, you can now use the normal synchronous API (never use yield in this style of test) and you can close processes easily by terminating them.

Typically for most user-facing functions you will find both kinds of tests. The `@gen_cluster` tests test particular logic while the `with cluster` tests test basic interface and resilience.

You should avoid `popen` style tests unless absolutely necessary, such as if you need to test the command line interface.

3.22 Foundations

You should read through the quickstart before reading this document.

Distributed computing is hard for two reasons:

1. Consistent coordination of distributed systems requires sophistication
2. Concurrent network programming is tricky and error prone

The foundations of `distributed` provide abstractions to hide some complexity of concurrent network programming (#2). These abstractions ease the construction of sophisticated parallel systems (#1) in a safer envirotnment. However, as with all layered abstractions, ours has flaws. Critical feedback is welcome.

3.22.1 Concurrency with Tornado Coroutines

Worker and Scheduler nodes operate concurrently. They serve several overlapping requests and perform several overlapping computations at the same time without blocking. There are several approaches for concurrent programming, we've chosen to use Tornado for the following reasons:

1. Developing and debugging is more comfortable without threads
2. Tornado's documentation is excellent
3. Stackoverflow coverage is excellent
4. Performance is satisfactory

3.22.2 Communication with Tornado Streams (raw sockets)

Workers, the Scheduler, and clients communicate with each other over the network. They use *raw sockets* as mediated by tornado streams. We separate messages by a sentinel value.

`distributed.core.`**`read`**`(`*stream*`)`
    Read a message from a stream

`distributed.core.`**`write`**`(`*stream*, *msg*`)`
    Write a message to a stream

3.22.3 Servers

Worker and Scheduler nodes serve requests over TCP. Both Worker and Scheduler objects inherit from a `Server` class. This Server class thinly wraps `tornado.tcpserver.TCPServer`. These servers expect requests of a particular form.

class `distributed.core.`**`Server`**`(`*handlers*, *max_buffer_size=2069891072.0*, ***kwargs*`)`
    Distributed TCP Server

Superclass for both Worker and Scheduler objects. Inherits from `tornado.tcpserver.TCPServer`, adding a protocol for RPC.

Handlers

Servers define operations with a `handlers` dict mapping operation names to functions. The first argument of a handler function must be a stream for the connection to the client. Other arguments will receive inputs from the keys of the incoming message which will always be a dictionary.

```
>>> def pingpong(stream):
...     return b'pong'
```

```
>>> def add(stream, x, y):
...     return x + y
```

```
>>> handlers = {'ping': pingpong, 'add': add}
>>> server = Server(handlers)
>>> server.listen(8000)
```

Message Format

The server expects messages to be dictionaries with a special key, *'op'* that corresponds to the name of the operation, and other key-value pairs as required by the function.

So in the example above the following would be good messages.

- `{'op': 'ping'}`
- `{'op': 'add': 'x': 10, 'y': 20}`

3.22.4 RPC

To interact with remote servers we typically use `rpc` objects.

class `distributed.core.`**`rpc`**(*arg=None*, *stream=None*, *ip=None*, *port=None*, *addr=None*, *timeout=3*)
Conveniently interact with a remote server

Normally we construct messages as dictionaries and send them with read/write

```
>>> stream = yield connect(ip, port)
>>> msg = {'op': 'add', 'x': 10, 'y': 20}
>>> yield write(stream, msg)
>>> response = yield read(stream)
```

To reduce verbosity we use an `rpc` object.

```
>>> remote = rpc(ip=ip, port=port)
>>> response = yield remote.add(x=10, y=20)
```

One rpc object can be reused for several interactions. Additionally, this object creates and destroys many streams as necessary and so is safe to use in multiple overlapping communications.

When done, close streams explicitly.

```
>>> remote.close_streams()
```

3.22.5 Example

Here is a small example using distributed.core to create and interact with a custom server.

Server Side

```
from tornado import gen
from tornado.ioloop import IOLoop
from distributed.core import write, Server

def add(stream, x=None, y=None):  # simple handler, just a function
    return x + y

@gen.coroutine
def stream_data(stream, interval=1):  # complex handler, multiple responses
    data = 0
    while True:
        yield gen.sleep(interval)
        data += 1
        yield write(stream, data)

s = Server({'add': add, 'stream': stream_data})
s.listen(8888)

IOLoop.current().start()
```

Client Side

```
from tornado import gen
from tornado.ioloop import IOLoop
from distributed.core import connect, read, write

@gen.coroutine
def f():
    stream = yield connect('127.0.0.1', 8888)
    yield write(stream, {'op': 'add', 'x': 1, 'y': 2})
    result = yield read(stream)
    print(result)

>>> IOLoop().run_sync(f)
3

@gen.coroutine
def g():
    stream = yield connect('127.0.0.1', 8888)
    yield write(stream, {'op': 'stream', 'interval': 1})
    while True:
        result = yield read(stream)
        print(result)

>>> IOLoop().run_sync(g)
1
2
3
...
```

Client Side with rpc

RPC provides a more pythonic interface. It also provides other benefits, such as using multiple streams in concurrent cases. Most distributed code uses rpc. The exception is when we need to perform multiple reads or writes, as with the stream data case above.

```
from tornado import gen
from tornado.ioloop import IOLoop
from distributed.core import rpc

@gen.coroutine
def f():
    # stream = yield connect('127.0.0.1', 8888)
    # yield write(stream, {'op': 'add', 'x': 1, 'y': 2})
    # result = yield read(stream)
    r = rpc(ip='127.0.0.1', 8888)
    result = yield r.add(x=1, y=2)

    print(result)

>>> IOLoop().run_sync(f)
3
```

3.22.6 Everything is a Server

Workers, Scheduler, and Nanny objects all inherit from Server. Each maintains separate state and serves separate functions but all communicate in the way shown above. They talk to each other by opening connections, writing messages that trigger remote functions, and then collect the results with read.

3.23 Client Interaction

As discussed in the quickstart users can interact with the worker-center network with the Executor abstraction.

This is built with lower level functions described below.

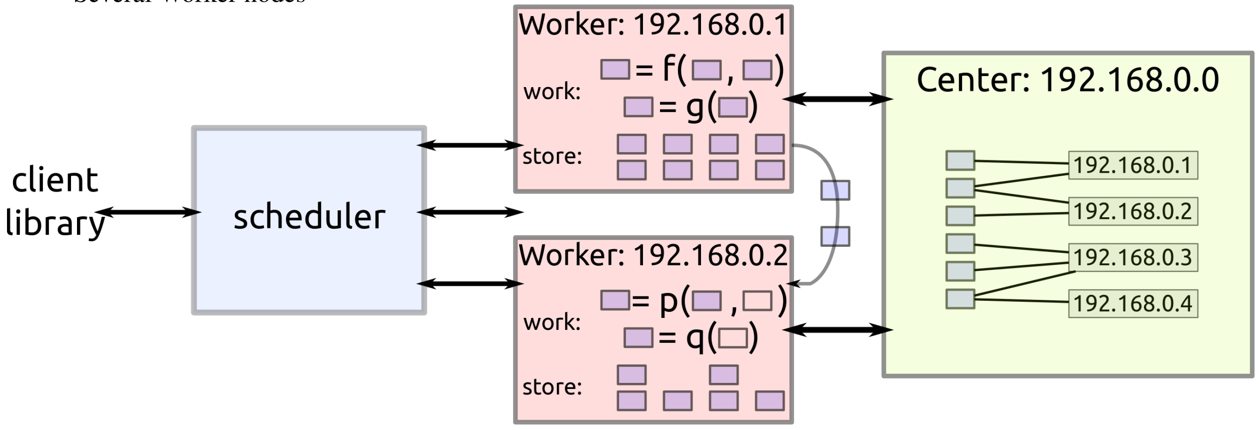
3.23.1 Scatter/Gather

Users rarely create RemoteData objects by hand. They are created by other client libraries or functions like `gather` and `scatter`.

3.24 Worker

We build a distributed network from two kinds of nodes.

- A single scheduler node
- Several Worker nodes



This page describes the worker nodes.

3.24.1 Serve Data

Workers serve data from a local dictionary of data:

```
{'x': np.array(...),
 'y': pd.DataFrame(...)}
```

Operations include normal dictionary operations, like get, set, and delete key-value pairs. In the following example we connect to two workers, collect data from one worker and send it to another.

```
alice = rpc(ip='192.168.0.101', port=8788)
d = yield alice.get_data(keys=['x', 'y'])

bob = rpc(ip='192.168.0.102', port=8788)
yield bob.update_data(data=d)
```

However, this is only an example, typically one does not manually manage data transfer between workers. They handle that as necessary on their own.

3.24.2 Compute

Workers evaluate functions provided by the user on their data. They evaluate functions either on their data or can automatically collect data from peers (as shown above) if they don't have the necessary data but their peers do:

```
z <- add(x, y)  # can be done with only local data
z <- add(x, a)  # need to find out where we can get 'a'
```

The result of such a computation on our end is just a response `b'OK'`. The actual result stays on the remote worker.

```
>>> response, metadata = yield alice.compute(function=add, keys=['x', 'a'])
>>> response
b'OK'
>>> metadata
{'nbytes': 1024}
```

The worker also reports back to the center/scheduler whenever it completes a computation. Metadata storage is centralized but all data transfer is peer-to-peer. Here is a quick example of what happens during a call to `compute`:

```
client:  Hey Alice!   Compute ``z <- add(x, a)``

Alice:   Hey Center!  Who has a?
Center:  Hey Alice!   Bob has a.
Alice:   Hey Bob!     Send me a!
Bob:     Hey Alice!   Here's a!

Alice:   Hey Client!  I've computed z and am holding on to it!
Alice:   Hey Center!  I have z!
```

class `distributed.worker.`**`Worker`**(*scheduler_ip, scheduler_port, ip=None, ncores=None, loop=None, local_dir=None, services=None, service_ports=None, name=None, heartbeat_interval=1000, **kwargs*)

Worker Node

Workers perform two functions:

1. **Serve data** from a local dictionary
2. **Perform computation** on that data and on data from peers

Additionally workers keep a scheduler informed of their data and use that scheduler to gather data from other workers when necessary to perform a computation.

You can start a worker with the `dworker` command line application:

```
    $ dworker scheduler-ip:port
```

State

- **data: `{key: object}`:** Dictionary mapping keys to actual values
- **active: `{key}`:** Set of keys currently under computation
- **ncores: `int`:** Number of cores used by this worker process
- **executor: `concurrent.futures.ThreadPoolExecutor`:** Executor used to perform computation
- **local_dir: `path`:** Path on local machine to store temporary files
- **scheduler: `rpc`:** Location of scheduler. See `.ip/.port` attributes.
- **name: `string`:** Alias

- **services: `{str: Server}`:** Auxiliary web servers running on this worker
- **service_ports:** `{str: port}`:

See also:

distributed.scheduler.Scheduler

Examples

Create schedulers and workers in Python:

```
>>> from distributed import Scheduler, Worker
>>> c = Scheduler('192.168.0.100', 8786)
>>> w = Worker(c.ip, c.port)
>>> yield w._start(port=8786)
```

Or use the command line:

```
$ dask-scheduler
Start scheduler at 127.0.0.1:8786

$ dask-worker 127.0.0.1:8786
Start worker at:               127.0.0.1:8786
Registered with scheduler at:  127.0.0.1:8787
```

3.25 Scheduler

The scheduler orchestrates all work across the cluster. It tracks all active workers and clients and manages the workers to perform the tasks requested by the clients. It tracks the current state of the entire cluster, determining which tasks execute on which workers in what order. It is a Tornado TCPServer consisting of several concurrent coroutines running in a single event loop.

3.25.1 Design

The scheduler tracks the state of many tasks among many workers. It updates the state of tasks in response to stimuli from workers and from clients. After the update of any new information it ensures that the entire system is on track to complete the desired tasks.

For performance, all updates happen in constant time, regardless of the number of known tasks. To achieve constant-time performance the scheduler state is heavily indexed, with several data structures (around 20) indexing each other to achieve fast lookup. All state is a mixture of interwoven dictionaries, lists, and sets.

This interwoven collection of dictionaries, sets, and lists is difficult to update consistently without error. The introduction of small errors and race conditions leads to infrequent but deadlocking errors that erode confidence in the scheduler. To mitigate this complexity we introduce coarser scale task transitions that are easier to reason about and are themselves heavily tested. Most code that responds to external stimuli/events is then written as a sequence of task transitions.

3.25.2 Transitions

A task is a single Python function call on data intended to be run within a single worker. Tasks fall into the following states with the following allowed transitions

- Released: known but not actively computing or in memory
- Waiting: On track to be computed, waiting on dependencies to arrive in memory
- Queue (ready): Ready to be computed by any worker
- Stacks (ready): Ready to be computed by a particular preferred worker
- No-worker (ready, rare): Ready to be computed, but no appropriate worker exists
- Processing: Actively being computed by one or more workers
- Memory: In memory on one or more workers
- Erred: Task has computed and erred
- Removed (not actually a state): Task is no longer needed by any client and so it removed from state

Every transition between states is a separate method in the scheduler. These task transition functions are prefixed with `transition` and then have the name of the start and finish task state like the following.

```
def transition_released_waiting(self, key):

def transition_processing_memory(self, key):

def transition_processing_erred(self, key):
```

These functions each have three effects.

1. They perform the necessary transformations on the scheduler state (the 20 dicts/lists/sets) to move one key between states.
2. They return a dictionary of recommended `{key:  state}` transitions to enact directly afterwards. For example after we transition a key into memory we may find that many waiting keys are now ready to transition from waiting to a ready state.
3. Optionally they include a set of validation checks that can be turned on for testing.

Rather than call these functions directly we call the central function `transition`:

```
def transition(self, key, final_state):
    """ Transition key to the suggested state """
```

This transition function finds the appropriate path from the current to the final state. Italso serves as a central point for logging and diagnostics.

Often we want to enact several transitions at once or want to continually respond to new transitions recommended by initial transitions until we reach a steady state. For that we use the `transitions` function (note the plural `s`).

```
def transitions(self, recommendations):
    recommendations = recommendations.copy()
    while recommendations:
        key, finish = recommendations.popitem()
        new = self.transition(key, finish)
        recommendations.update(new)
```

This function runs `transition`, takes the recommendations and runs them as well, repeating until no further task-transitions are recommended.

3.25.3 Stimuli

Transitions occur from stimuli, which are state-changing messages to the scheduler from workers or clients. The scheduler responds to the following stimuli:

- **Workers**
  - Task finished: A task has completed on a worker and is now in memory
  - Task erred: A task ran and erred on a worker
  - Task missing data: A task tried to run but was unable to find necessary data on other workers
  - Worker added: A new worker was added to the network
  - Worker removed: An existing worker left the network
- **Clients**
  - Update graph: The client sends more tasks to the scheduler
  - Release keys: The client no longer desires the result of certain keys

Stimuli functions are prepended with the text `stimulus`, and take a variety of keyword arguments from the message as in the following examples:

```python
def stimulus_task_finished(self, key=None, worker=None, nbytes=None,
                           type=None, compute_start=None, compute_stop=None,
                           transfer_start=None, transfer_stop=None):

def stimulus_task_erred(self, key=None, worker=None,
                        exception=None, traceback=None)
```

These functions change some non-essential administrative state and then call transition functions.

Note that there are several other non-state-changing messages that we receive from the workers and clients, such as messages requesting information about the current state of the scheduler. These are not considered stimuli.

3.25.4 API

class `distributed.scheduler.`**`Scheduler`**(*center=None, loop=None, max_buffer_size=2069891072.0, delete_interval=500, synchronize_worker_interval=5000, ip=None, services=None, allowed_failures=3, validate=False, **kwargs*)

Dynamic distributed task scheduler

The scheduler tracks the current state of workers, data, and computations. The scheduler listens for events and responds by controlling workers appropriately. It continuously tries to use the workers to execute an ever growing dask graph.

All events are handled quickly, in linear time with respect to their input (which is often of constant size) and generally within a millisecond. To accomplish this the scheduler tracks a lot of state. Every operation maintains the consistency of this state.

The scheduler communicates with the outside world through Tornado IOStreams It maintains a consistent and valid view of the world even when listening to several clients at once.

A Scheduler is typically started either with the `dask-scheduler` executable:

```
$ dask-scheduler
Scheduler started at 127.0.0.1:8786
```

Or within a LocalCluster a Executor starts up without connection information:

```
>>> e = Executor()
>>> e.cluster.scheduler
Scheduler(...)
```

Users typically do not interact with the scheduler directly but rather with the client object `Executor`.

State

The scheduler contains the following state variables. Each variable is listed along with what it stores and a brief description.

- **tasks: `{key: task}`:** Dictionary mapping key to a serialized task like the following: `{'function': b'...', 'args': b'...'}` or `{'task': b'...'}`
- **dependencies: `{key: {keys}}`:** Dictionary showing which keys depend on which others
- **dependents: `{key: {keys}}`:** Dictionary showing which keys are dependent on which others
- **task_state: `{key: string}`:** Dictionary listing the current state of every task among the following: released, waiting, stacks, queue, no-worker, processing, memory, erred
- **priority: `{key: tuple}`:** A score per key that determines its priority
- **waiting: `{key: {key}}`:** Dictionary like dependencies but excludes keys already computed
- **waiting_data: `{key: {key}}`:** Dictionary like dependents but excludes keys already computed
- **ready: `deque(key)`** Keys that are ready to run, but not yet assigned to a worker
- **stacks: `{worker: [keys]}`:** List of keys waiting to be sent to each worker
- **processing: `{worker: {key: cost}}`:** Set of keys currently in execution on each worker and their expected duration
- **rprocessing: `{key: {worker}}`:** Set of workers currently executing a particular task
- **who_has: `{key: {worker}}`:** Where each key lives. The current state of distributed memory.
- **has_what: `{worker: {key}}`:** What worker has what keys. The transpose of who_has.
- **released: `{keys}`** Set of keys that are known, but released from memory
- **unrunnable: `{key}`** Keys that we are unable to run
- **retrictions: `{key: {hostnames}}`:** A set of hostnames per key of where that key can be run. Usually this is empty unless a key has been specifically restricted to only run on certain hosts. These restrictions don't include a worker port. Any worker on that hostname is deemed valid.
- **loose_retrictions: `{key}`:** Set of keys for which we are allow to violate restrictions (see above) if not valid workers are present.
- **exceptions: `{key: Exception}`:** A dict mapping keys to remote exceptions
- **tracebacks: `{key: list}`:** A dict mapping keys to remote tracebacks stored as a list of strings
- **exceptions_blame: `{key: key}`:** A dict mapping a key to another key on which it depends that has failed
- **suspicious_tasks: `{key: int}`** Number of times a task has been involved in a worker failure
- **deleted_keys: `{key: {workers}}`** Locations of workers that have keys that should be deleted
- **wants_what: `{client: {key}}`:** What keys are wanted by each client.. The transpose of who_wants.
- **who_wants: `{key: {client}}`:** Which clients want each key. The active targets of computation.

- **nbytes: `{key: int}`:** Number of bytes for a key as reported by workers holding that key.
- **stealable: `[[key]]`** A list of stacks of stealable keys, ordered by stealability
- **ncores: `{worker: int}`:** Number of cores owned by each worker
- **idle: `{worker}`:** Set of workers that are not fully utilized
- **worker_info: `{worker: {str: data}}`:** Information about each worker
- **host_info: `{hostname: dict}`:** Information about each worker host
- **occupancy: `{worker: time}`** Expected runtime for all tasks currently processing on a worker
- **services: `{str: port}`:** Other services running on this scheduler, like HTTP
- **loop: `IOLoop`:** The running Torando IOLoop
- **streams: `[IOStreams]`:** A list of Tornado IOStreams from which we both accept stimuli and report results
- **task_duration: `{key-prefix: time}`** Time we expect certain functions to take, e.g. `{'sum': 0.25}`
- **coroutines: `[Futures]`:** A list of active futures that control operation
- **scheduler_queues: `[Queues]`:** A list of Tornado Queues from which we accept stimuli
- **report_queues: `[Queues]`:** A list of Tornado Queues on which we report results

`add_client`(*stream*, *client=None*)
Add client to network

We listen to all future messages from this IOStream.

`add_keys`(*stream=None*, *address=None*, *keys=()*)
Learn that a worker has certain keys

This should not be used in practice and is mostly here for legacy reasons.

`add_plugin`(*plugin*)
Add external plugin to scheduler

See http://distributed.readthedocs.io/en/latest/plugins.html

`add_worker`(*stream=None*, *address=None*, *keys=()*, *ncores=None*, *name=None*, *coerce_address=True*, *nbytes=None*, *now=None*, *host_info=None*, ***info*)
Add a new worker to the cluster

`broadcast`(*stream=None*, *msg=None*, *workers=None*, *hosts=None*)
Broadcast message to workers, return all results

`cancel_key`(*key*, *client*, *retries=5*)
Cancel a particular key and all dependents

`change_worker_cores`(*stream=None*, *worker=None*, *diff=0*)
Add or remove cores from a worker

This is used when a worker wants to spin off a long-running task

`cleanup`()
Clean up queues and coroutines, prepare to stop

`clear_data_from_workers`()
Send delete signals to clear unused data from workers

This watches the `.deleted_keys` attribute, which stores a set of keys to be deleted from each worker. This function is run periodically by the `._delete_periodic_callback` to actually remove tha data.

This runs every `self.delete_interval` milliseconds.

`client_releases_keys`(*keys=None*, *client=None*)
Remove keys from client desired list

`close`(*stream=None*, *fast=False*)
Send cleanup signal to all coroutines then wait until finished

See also:

`Scheduler.cleanup`

`close_streams`()
Close all active IOStreams

`coerce_address`(*addr*)
Coerce possible input addresses to canonical form

Handles lists, strings, bytes, tuples, or aliases

`correct_time_delay`(*worker*, *msg*)
Apply offset time delay in message times.

Clocks on different workers differ. We keep track of a relative "now" through periodic heartbeats. We use this known delay to align message times to Scheduler local time. In particular this helps with diagnostics.

Operates in place

`ensure_occupied`()
Run ready tasks on idle workers

Work stealing policy

If some workers are idle but not others, if there are no globally ready tasks, and if there are tasks in worker stacks, then we start to pull preferred tasks from overburdened workers and deploy them back into the global pool in the following manner.

We determine the number of tasks to reclaim as the number of all tasks in all stacks times the fraction of idle workers to all workers. We sort the stacks by size and walk through them, reclaiming half of each stack until we have enough task to fill the global pool. We are careful not to reclaim tasks that are restricted to run on certain workers.

See also:

`Scheduler.ensure_occupied_queue`, *`Scheduler.ensure_occupied_stacks`*, *`Scheduler.work_steal`*

`ensure_occupied_queue`(*worker*, *count*)
Send at most count tasks from the ready queue to the specified worker

See also:

`Scheduler.ensure_occupied`, *`Scheduler.ensure_occupied_stacks`*

`ensure_occupied_stacks`(*worker*)
Send tasks to worker while it has tasks and free cores

These tasks may come from the worker's own stacks or from the global ready deque.

We update the idle workers set appropriately.

See also:

`Scheduler.ensure_occupied`, *`Scheduler.ensure_occupied_queue`*

feed(*stream*, *function=None*, *setup=None*, *teardown=None*, *interval=1*, ***kwargs*)

Provides a data stream to external requester

Caution: this runs arbitrary Python code on the scheduler. This should eventually be phased out. It is mostly used by diagnostics.

finished()

Wait until all coroutines have ceased

gather(*stream=None*, *keys=None*)

Collect data in from workers

handle_messages(*in_queue*, *report*, *client=None*)

The master client coroutine. Handles all inbound messages from clients.

This runs once per Client IOStream or Queue.

See also:

`Scheduler.worker_stream` The equivalent function for workers

handle_queues(*scheduler_queue*, *report_queue*)

Register new control and report queues to the Scheduler

Queues are not in common use. This may be deprecated in the future.

identity(*stream*)

Basic information about ourselves and our cluster

issaturated(*worker*, *latency=0.005*)

Determine if a worker has enough work to avoid being idle

A worker is saturated if the following criteria are met

1. It is working on at least as many tasks as it has cores
2. The expected time it will take to complete all of its currently assigned tasks is at least a full round-trip time. This is relevant when it has many small tasks

rebalance(*stream=None*, *keys=None*, *workers=None*)

Rebalance keys so that each worker stores roughly equal bytes

remove_client(*client=None*)

Remove client from network

remove_plugin(*plugin*)

Remove external plugin from scheduler

remove_worker(*stream=None*, *address=None*)

Remove worker from cluster

We do this when a worker reports that it plans to leave or when it appears to be unresponsive. This may send its tasks back to a released state.

replicate(*stream=None*, *keys=None*, *n=None*, *workers=None*, *branching_factor=2*)

Replicate data throughout cluster

This performs a tree copy of the data throughout the network individually on each piece of data.

Parameters keys: Iterable

list of keys to replicate

n: int

Number of replications we expect to see within the cluster

branching_factor: int, optional

The number of workers that can copy data in each generation

See also:

`Scheduler.rebalance`

report (*msg*)
Publish updates to all listening Queues and Streams

If the message contains a key then we only send the message to those streams that care about the key.

restart ()
Restart all workers. Reset local state.

scatter (*stream=None*, *data=None*, *workers=None*, *client=None*, *broadcast=False*)
Send data out to workers

See also:

`Scheduler.broadcast`

send_task_to_worker (*worker*, *key*)
Send a single computational task to a worker

start (*port=8786*, *start_queues=True*)
Clear out old state and restart all running coroutines

start_ipython (*stream=None*)
Start an IPython kernel

Returns Jupyter connection info dictionary.

steal_time_ratio (*key*, *bandwidth=100000000*)
The compute to communication time ratio of a key

Returns ratio: The compute/communication time ratio of the task

loc: The self.stealable bin into which this key should go

stimulus_cancel (*stream*, *keys=None*, *client=None*)
Stop execution on a list of keys

stimulus_missing_data (*keys=None*, *key=None*, *worker=None*, *ensure=True*, ***kwargs*)
Mark that certain keys have gone missing. Recover.

stimulus_task_erred (*key=None*, *worker=None*, *exception=None*, *traceback=None*, ***kwargs*)
Mark that a task has erred on a particular worker

stimulus_task_finished (*key=None*, *worker=None*, ***kwargs*)
Mark that a task has finished execution on a particular worker

transition (*key*, *finish*, **args*, ***kwargs*)
Transition a key from its current state to the finish state

Returns Dictionary of recommendations for future transitions

See also:

`Scheduler.transitions` transitive version of this function

Examples

```
>>> self.transition('x', 'waiting')
{'x': 'ready'}
```

transition_story(**keys*)
Get all transitions that touch one of the input keys

transitions(*recommendations*)
Process transitions until none are left

This includes feedback from previous transitions and continues until we reach a steady state

update_data(*stream=None*, *who_has=None*, *nbytes=None*, *client=None*)
Learn that new data has entered the network from an external source

See also:

`Scheduler.mark_key_in_memory`

update_graph(*client=None*, *tasks=None*, *keys=None*, *dependencies=None*, *restrictions=None*, *priority=None*, *loose_restrictions=None*)
Add new computations to the internal dask graph

This happens whenever the Executor calls submit, map, get, or compute.

work_steal()
Steal tasks from saturated workers to idle workers

This moves tasks from the bottom of the stacks of over-occupied workers to the stacks of idling workers.

See also:

`Scheduler.ensure_occupied`

worker_stream(*worker*)
Listen to responses from a single worker

This is the main loop for scheduler-worker interaction

See also:

`Scheduler.handle_messages` Equivalent coroutine for clients

workers_list(*workers*)
List of qualifying workers

Takes a list of worker addresses or hostnames. Returns a list of all worker addresses that match

`distributed.scheduler.`**decide_worker**(*dependencies*, *stacks*, *processing*, *who_has*, *has_what*, *restrictions*, *loose_restrictions*, *nbytes*, *key*)
Decide which worker should take task

```
>>> dependencies = {'c': {'b'}, 'b': {'a'}}
>>> stacks = {'alice:8000': ['z'], 'bob:8000': []}
>>> processing = {'alice:8000': set(), 'bob:8000': set()}
>>> who_has = {'a': {'alice:8000'}}
>>> has_what = {'alice:8000': {'a'}}
>>> nbytes = {'a': 100}
>>> restrictions = {}
>>> loose_restrictions = set()
```

We choose the worker that has the data on which 'b' depends (alice has 'a')

```
>>> decide_worker(dependencies, stacks, processing, who_has, has_what,
...               restrictions, loose_restrictions, nbytes, 'b')
'alice:8000'
```

If both Alice and Bob have dependencies then we choose the less-busy worker

```
>>> who_has = {'a': {'alice:8000', 'bob:8000'}}
>>> has_what = {'alice:8000': {'a'}, 'bob:8000': {'a'}}
>>> decide_worker(dependencies, stacks, processing, who_has, has_what,
...               restrictions, loose_restrictions, nbytes, 'b')
'bob:8000'
```

Optionally provide restrictions of where jobs are allowed to occur

```
>>> restrictions = {'b': {'alice', 'charlie'}}
>>> decide_worker(dependencies, stacks, processing, who_has, has_what,
...               restrictions, loose_restrictions, nbytes, 'b')
'alice:8000'
```

If the task requires data communication, then we choose to minimize the number of bytes sent between workers. This takes precedence over worker occupancy.

```
>>> dependencies = {'c': {'a', 'b'}}
>>> who_has = {'a': {'alice:8000'}, 'b': {'bob:8000'}}
>>> has_what = {'alice:8000': {'a'}, 'bob:8000': {'b'}}
>>> nbytes = {'a': 1, 'b': 1000}
>>> stacks = {'alice:8000': [], 'bob:8000': []}
```

```
>>> decide_worker(dependencies, stacks, processing, who_has, has_what,
...               {}, set(), nbytes, 'c')
'bob:8000'
```

3.26 Resilience

Software fails, Hardware fails, network connections fail, user code fails. This document describes how `distributed` responds in the face of these failures and other known bugs.

3.26.1 User code failures

When a function raises an error that error is kept and transmitted to the executor on request. Any attempt to gather that result *or any dependent result* will raise that exception.

```
>>> def div(a, b):
...     return a / b

>>> x = executor.submit(div, 1, 0)
>>> x.result()
ZeroDivisionError: division by zero

>>> y = executor.submit(add, x, 10)
>>> y.result()  # same error as above
ZeroDivisionError: division by zero
```

This does not affect the smooth operation of the scheduler or worker in any way.

3.26.2 Closed Network Connections

If the connection to a remote worker unexpectedly closes and the local process appropriately raises an `IOError` then the scheduler will reroute all pending computations to other workers.

If the lost worker was the only worker to hold vital results necessary for future computations then those results will be recomputed by surviving workers. The scheduler maintains a full history of how each result was produced and so is able to reproduce those same computations on other workers.

This has some fail cases.

1. If results depend on impure functions then you may get a different (although still entirely accurate) result
2. If the worker failed due to a bad function, for example a function that causes a segmentation fault, then that bad function will repeatedly be called on other workers, and proceed to kill the distributed system, one worker at a time.
3. Data ‘‘scatter‘‘ed out to the workers is not kept in the scheduler (it is often quite large) and so the loss of this data is irreparable.

3.26.3 Hardware Failures

It is not clear under which circumstances the local process will know that the remote worker has closed the connection. If the socket does not close cleanly then the system will wait for a timeout, roughly three seconds, before marking the worker as failed and resuming smooth operation.

3.26.4 Scheduler Failure

The process containing the scheduler might die. There is currently no persistence mechanism to record and recover the scheduler state. The data will remain on the cluster until cleared.

3.26.5 Restart and Nanny Processes

The executor provides a mechanism to restart all of the workers in the cluster. This is convenient if, during the course of experimentation, you find your workers in an inconvenient state that makes them unresponsive. The `Executor.restart` method does the following process:

1. Sends a soft shutdown signal to all of the coroutines watching workers
2. Sends a hard kill signal to each worker’s Nanny process, which oversees that worker. This Nanny process terminates the worker process ungracefully and unregisters that worker from the Scheduler.
3. Clears out all scheduler state and sets all Future’s status to `'cancelled'`
4. Sends a restart signal to all Nanny processes, which in turn restart clean Worker processes and register these workers with the Scheduler. New workers may not have the same port as their previous iterations. The `.nannies` dictionary on the Executor serves as an accurate set of aliases if necessary.
5. Restarts the scheduler, with clean and empty state

This effectively removes all data and clears out all computations from the scheduler. Any data or computations not saved to persistent storage are lost. This process is very robust to a number of failure modes, including non-responsive or swamped workers but not including full hardware failures.

Currently the user may experience a few error logging messages from Tornado upon closing their session. These can safely be ignored.

3.27 Journey of a Task

We follow a single task through the user interface, scheduler, worker nodes, and back. Hopefully this helps to illustrate the inner workings of the system.

3.27.1 User code

A user computes the addition of two variables already on the cluster, then pulls the result back to the local process.

```
e = Executor('host:port')
x = e.submit(...)
y = e.submit(...)

z = e.submit(add, x, y)  # we follow z

print(z.result())
```

3.27.2 Step 1: Executor

`z` begins its life when the `Executor.submit` function sends the following message to the `Scheduler`:

```
{'op': 'update-graph',
 'tasks': {'z': (add, x, y)},
 'keys': ['z']}
```

The executor then creates a `Future` object with the key `'z'` and returns that object back to the user. This happens even before the message has been received by the scheduler. The status of the future says `'pending'`.

3.27.3 Step 2: Arrive in the Scheduler

A few milliseconds later, the scheduler receives this message on an open socket.

The scheduler updates its state with this little graph that shows how to compute `z`.:

```
scheduler.tasks.update[msg['tasks']]
```

The scheduler also updates *a lot* of other state. Notably, it has to identify that `x` and `y` are themselves variables, and connect all of those dependencies. This is a long and detail oriented process that involves updating roughly 10 sets and dictionaries. Interested readers should investigate `distributed/scheduler.py::update_state()`. While this is fairly complex and tedious to describe rest assured that it all happens in constant time and in about a millisecond.

3.27.4 Step 3: Select a Worker

Once the latter of `x` and `y` finishes, the scheduler notices that all of `z`'s dependencies are in memory and that `z` itself may now run. Which worker should `z` select? We consider a sequence of criteria:

1. First, we quickly downselect to only those workers that have either `x` or `y` in local memory.
2. Then, we select the worker that would have to gather the least number of bytes in order to get both `x` and `y` locally. E.g. if two different workers have `x` and `y` and if `y` takes up more bytes than `x` then we select the machine that holds `y` so that we don't have to communicate as much.
3. If there are multiple workers that require the minimum number of communication bytes then we select the worker that is the least busy

z considers the workers and chooses one based on the above criteria. In the common case the choice is pretty obvious after step 1. z waits on a stack associated with the chosen worker. The worker may still be busy though, so z may wait a while.

Note: This policy is under flux and this part of this document is quite possibly out of date.

3.27.5 Step 4: Transmit to the Worker

Eventually the worker finishes a task, has a spare core, and z finds itself at the top of the stack (note, that this may be some time after the last section if other tasks placed themselves on top of the worker's stack in the meantime.)

We place z into a worker_queue associated with that worker and a worker_core coroutine pulls it out. z's function, the keys associated to its arguments, and the locations of workers that hold those keys are packed up into a message that looks like this:

```
{'op': 'compute',
 'function': execute_task,
 'args': ((add, 'x', 'y'),),
 'who_has': {'x': {(worker_host, port)},
             'y': {(worker_host, port), (worker_host, port)}},
 'key': 'z'}
```

This message is serialized and sent across a TCP socket to the worker.

3.27.6 Step 5: Execute on the Worker

The worker unpacks the message, and notices that it needs to have both x and y. If the worker does not already have both of these then it gathers them from the workers listed in the who_has dictionary also in the message. For each key that it doesn't have it selects a valid worker from who_has at random and gathers data from it.

After this exchange, the worker has both the value for x and the value for y. So it launches the computation add(x, y) in a local ThreadPoolExecutor and waits on the result.

In the mean time the worker repeats this process concurrently for other tasks. Nothing blocks.

Eventually the computation completes. The Worker stores this result in its local memory:

```
data['x'] = ...
```

And transmits back a success, and the number of bytes of the result:

```
Worker: Hey Scheduler, 'z' worked great.
        I'm holding onto it.
        It takes up 64 bytes.
```

The worker does not transmit back the actual value for z.

3.27.7 Step 6: Scheduler Aftermath

The scheduler receives this message and does a few things:

1. It notes that the worker has a free core, and sends up another task if available
2. If x or y are no longer needed then it sends a message out to relevant workers to delete them from local memory.
3. It sends a message to all of the clients that z is ready and so all client Future objects that are currently waiting should, wake up. In particular, this wakes up the z.result() command executed by the user originally.

3.27.8 Step 7: Gather

When the user calls `z.result()` they wait both on the completion of the computation and for the computation to be sent back over the wire to the local process. Usually this isn't necessary, usually you don't want to move data back to the local process but instead want to keep in on the cluster.

But perhaps the user really wanted to actually know this value, so they called `z.result()`.

The scheduler checks who has `z` and sends them a message asking for the result. This message doesn't wait in a queue or for other jobs to complete, it starts instantly. The value gets serialized, sent over TCP, and then deserialized and returned to the user (passing through a queue or two on the way.)

3.27.9 Step 8: Garbage Collection

The user leaves this part of their code and the local variable `z` goes out of scope. The Python garbage collector cleans it up. This triggers a decremented reference on the executor (we didn't mention this, but when we created the `Future` we also started a reference count.) If this is the only instance of a Future pointing to `z` then we send a message up to the scheduler that it is OK to release `z`. The user no longer requires it to persist.

The scheduler receives this message and, if there are no computations that might depend on `z` in the immediate future, it removes elements of this key from local scheduler state and adds the key to a list of keys to be deleted periodically. Every 500 ms a message goes out to relevant workers telling them which keys they can delete from their local memory. The graph/recipe to create the result of `z` persists in the scheduler for all time.

3.27.10 Overhead

The user experiences this in about 10 milliseconds, depending on network latency.

3.28 Protocol

The scheduler, workers, and clients pass messages between each other. Semantically these messages encode commands, status updates, and data, like the following:

- Please compute the function `sum` on the data `x` and store in `y`
- The computation `y` has been completed
- Be advised that a new worker named `alice` is available for use
- Here is the data for the keys `'x'`, and `'y'`

In practice we represent these messages with dictionaries/mappings:

```
{'op': 'compute',
 'function': ...
 'args': ['x']}

{'op': 'task-complete',
 'key': 'y',
 'nbytes': 26}

{'op': 'register-worker',
 'address': '192.168.1.42',
 'name': 'alice',
 'ncores': 4}
```

```
{'x': b'...',
 'y': b'...'}
```

When we communicate these messages between nodes we need to serialize these messages down to a string of bytes that can then be deserialized on the other end to their in-memory dictionary form. For simple cases several options exist like JSON, MsgPack, Protobuffers, and Thrift. The situation is made more complex by concerns like serializing Python functions and Python objects, optional compression, cross-language support, large messages, and efficiency.

This document describes the protocol used by `dask.distributed` today. Be advised that this protocol changes rapidly as we continue to optimize for performance.

3.28.1 Overview

We may split a single message into multiple message-part to suit different protocols. Generally small bits of data are encoded with MsgPack while large bytestrings are handled specially by a custom format. Each message-part gets its own header, which is always encoded as msgpack. After serializing all message parts we have a sequence of bytestrings or *frames* which we send along the wire, prepended with length information.

The application doesn't know any of this, it just sends us Python dictionaries with various datatypes and we produce a list of bytestrings that get written to a socket. This format is fast both for many frequent messages and for large messages.

3.28.2 MsgPack for Messages

Most messages are encoded with MsgPack, a self describing semi-structured serialization format that is very similar to JSON, but smaller, faster, not human-readable, and supporting of bytestrings and (soon) timestamps. We chose MsgPack as a base serialization format for the following reasons:

- It does not require separate headers, and so is easy and flexible to use which is particularly important in an early stage project like `dask.distributed`
- It is very fast, much faster than JSON, and there are nicely optimized implementations, particularly within the `pandas.msgpack` module. With few exceptions (described later) MsgPack does not come anywhere near being a bottleneck, even under heavy use.
- Unlike JSON it supports bytestrings
- It covers the standard set of types necessary to encode most information
- It is widely implemented in a number of languages (see cross language section below)

However, MsgPack fails (correctly) in the following ways:

- It does not provide any way for us to encode Python functions or user defined data types
- It does not support bytestrings greater than 4GB and is generally inefficient for very large messages.

Because of these failings we supplement it with a language-specific protocol and a special case for large bytestrings.

3.28.3 CloudPickle for Functions and Data

Pickle and CloudPickle are Python libraries to serialize almost any Python object, including functions. We use these libraries to transform the users' functions and data into bytes before we include them in the dictionary/map that we pass off to msgpack. In the introductory example you may have noticed that we skipped providing an example for the function argument:

```
{'op': 'compute',
 'function': ...
 'args': ['x']}
```

That is because this value `...` will actually be the result of calling `cloudpickle.dumps(myfunction)`. Those bytes will then be included in the dictionary that we send off to msgpack, which will only have to deal with bytes rather than obscure Python functions.

3.28.4 Cross Language Specialization

The Client and Workers must agree on a language-specific serialization format. In the standard `dask.distributed` client and worker objects this ends up being the following:

```
bytes = cloudpickle.dumps(obj, protocol=pickle.HIGHEST_PROTOCOL)
obj = cloudpickle.loads(bytes)
```

This varies between Python 2 and 3 and so your client and workers must match their Python versions and software environments.

However, the Scheduler never uses the language-specific serialization and instead only deals with MsgPack. If the client sends a pickled function up to the scheduler the scheduler will not unpack function but will instead keep it as bytes. Eventually those bytes will be sent to a worker, which will then unpack the bytes into a proper Python function. Because the Scheduler never unpacks language-specific serialized bytes it may be in a different language.

The client and workers must share the same language and software environment, the scheduler may differ.

This has a few advantages:

1. The Scheduler is protected from unpickling unsafe code
2. The Scheduler can be run under `pypy` for improved performance. This is only useful for larger clusters.
3. We could conceivably implement workers and clients for other languages (like R or Julia) and reuse the Python scheduler. The worker and client code is fairly simple and much easier to reimplement than the scheduler, which is complex.
4. The scheduler might some day be rewritten in more heavily optimized C or Go

3.28.5 Compression

Fast compression libraries like LZ4 or Snappy may increase effective bandwidth by compressing data before sending and decompressing it after reception. This is especially valuable on lower-bandwidth networks.

If either of these libraries is available (we prefer LZ4 to Snappy) then for every message greater than 1kB we try to compress the message and, if the compression is at least a 10% improvement, we send the compressed bytes rather than the original payload. We record the compression used within the header as a string like `'lz4'` or `'snappy'`.

To avoid compressing large amounts of uncompressable data we first try to compress a sample. We take 10kB chunks from five locations in the dataset, arrange them together, and try compressing the result. If this doesn't result in significant compression then we don't try to compress the full result.

3.28.6 Header

The header is a small dictionary encoded in msgpack that includes some metadata about the message, such as compression.

3.28.7 Large Bytestrings

Whenever a message comes in with very large byte values like the following:

```
{'key': 'x',
 'address': 'alice',
 'data-1': b'...'  # very long bytestring
 'data-2': b'...'  # very long bytestring
 }
```

We separate the message into two messages, one encoding all of the large bytestrings, and one encoding everything else:

```
{'key': 'x', 'addresss': 'alice'}
{'data-1': b'...', 'data-2': b'...'}
```

The first message we pass normally with msgpack, the second we pass in multiple parts, including a header that contains the keys and compression used for each value:

```
{'keys': ['data-1', 'data-2'],
 'compression': ['lz4', None]}
b'...'
b'...'
```

3.28.8 Frames

At the end of the pipeline we have a sequence of bytestrings or frames. We need to tell the receiving end how many frames there are and how long each these frames are. We order the frames and lengths of frames as follows:

1. The number of frames, stored as an 8 byte unsigned integer
2. The length of each frame, each stored as an 8 byte unsigned integer
3. Each of the frames

In the following sections we describe how we create these frames.

3.28.9 Performance

For large numpy arrays we currently suffer three memory copies. On a nice machine this ends up being a 1-1.5 GB/s bottleneck, which is almost always faster than the network bandwidth. These copies come from NumPy (two memcopies) and Tornado (one memcopy).

For small messages we generally serialize in around 5 microseconds.

3.29 Work Stealing

Some tasks prefer to run on certain workers. This may be because that worker holds data dependencies of the task or because the user has expressed a loose desire that the task run in a particular place. Occasionally this results in a few very busy workers and several idle workers. In this situation the idle workers may choose to steal work from busy workers, even if stealing work requires the costly movement of data.

This is a performance optimization and not required for correctness. Work stealing provides robustness in many ad-hoc cases, but can also backfire when we steal the wrong tasks and reduce performance.

3.29.1 Task criteria for stealing

If a task has been specifically restricted to run on particular workers (such as is the case when special hardware is required) then we do not steal. Barring this case, stealing usually occurs for tasks that have been assigned to a particular worker because that worker holds the data necessary to compute the task.

Stealing is profitable when the computation time for a task is much longer than the communication time of the task's dependencies. It is also good long term if stealing causes highly-sought-after data to be replicated on more workers.

Bad example

We do not want to steal tasks that require moving a large dependent piece of data across a wire from the victim to the thief if the computation is fast. We end up spending far more time in communication than just waiting a bit longer and giving up on parallelism.

```
[data] = e.scatter([np.arange(1000000000)])
x = e.submit(np.sum, data)
```

Good example

We do want to steal task tasks that only need to move dependent pieces of data, especially when the computation time is expensive (here 100 seconds.)

```
[data] = e.scatter([100])
x = e.submit(sleep, data)
```

Fortunately we often know both the number of bytes of dependencies (as reported by calling `sys.getsizeof` on the workers) and the runtime cost of previously seen functions.

3.29.2 When do we worksteal

The scheduler maintains a set of idle workers and a set of saturated workers. At various events, such as when new tasks arrive from the client, when new workers arrive, or when we learn that workers have completed a set of tasks, we play these two sets of idle and saturated workers against each other.

3.29.3 Choosing tasks to steal

Occupied workers maintain a stack of excess work. The tasks at the top of this stack are prioritized to be run by that worker before the tasks at the bottom.

Ideally we choose the worker with the *largest* stack of excess work and then select the task at the *bottom* of this stack, hopefully starting a new sequence of computations that are somewhat unrelated to what the busy worker is currently working on.

All operations in the scheduler endeavor to be computed in constant time (or linear time relative to the number of processed tasks.) We can pull from the bottom of the stack in constant time by implementing each worker's stack as a `collections.deque`. However, we currently do not maintain the data structures necessary to efficiently find the most occupied workers. Common solutions, like maintaining priority queue of workers by stack length add a `log(n)` cost to the common case.

Instead we just call `next(iter(saturated_workers))` and allow Python to iterate through the set of saturated workers however it prefers.

3.30 Scheduler Plugins

class `distributed.diagnostics.plugin.`**`SchedulerPlugin`**

Interface to extend the Scheduler

The scheduler operates by triggering and responding to events like `task_finished`, `update_graph`, `task_erred`, etc..

A plugin enables custom code to run at each of those same events. The scheduler will run the analagous methods on this class when each event is triggered. This runs user code within the scheduler thread that can perform arbitrary operations in synchrony with the scheduler itself.

Plugins are often used for diagnostics and measurement, but have full access to the scheduler and could in principle affect core scheduling.

To implement a plugin implement some of the methods of this class and add the plugin to the scheduler with `Scheduler.add_plugin(myplugin)`.

Examples

```
>>> class Counter(SchedulerPlugin):
...     def __init__(self):
...         self.counter = 0
...
...     def transition(self, key, start, finish, *args, **kwargs):
...         if start == 'processing' and finish == 'memory':
...             self.counter += 1
...
...     def restart(self, scheduler):
...         self.counter = 0
```

```
>>> c = Counter()
>>> scheduler.add_plugin(c)
```

`restart`(*scheduler*, ***kwargs*)

Run when the scheduler restarts itself

`update_graph`(*scheduler*, *dsk=None*, *keys=None*, *restrictions=None*, ***kwargs*)

Run when a new graph / tasks enter the scheduler

3.31 Related Work

Writing the “related work” for a project called “distributed”, is a Sisyphean task. We’ll list a few notable projects that you’ve probably already heard of down below.

You may also find the dask comparison with spark of interest.

3.31.1 Big Data World

- The venerable Hadoop provides batch processing with the MapReduce programming paradigm. Python users typically use Hadoop Streaming or MRJob.
- Spark builds on top of HDFS systems with a nicer API and in-memory processing. Python users typically use PySpark.

- Storm provides streaming computation. Python users typically use streamparse.

This is a woefully inadequate representation of the excellent work blossoming in this space. A variety of projects have come into this space and rival or complement the projects above. Still, most “Big Data” processing hype probably centers around the three projects above, or their derivatives.

3.31.2 Python Projects

There are dozens of Python projects for distributed computing. Here we list a few of the more prominent projects that we see in active use today.

Task scheduling

- Celery: An asynchronous task scheduler, focusing on real-time processing.
- Luigi: A bulk big-data/batch task scheduler, with hooks to a variety of interesting data sources.

Ad hoc computation

- IPython Parallel: Allows for stateful remote control of several running ipython sessions.
- Scoop: Implements the concurrent.futures API on distributed workers. Notably allows tasks to spawn more tasks.

Direct Communication

- MPI4Py: Wraps the Message Passing Interface popular in high performance computing.
- PyZMQ: Wraps ZeroMQ, the gentleman’s socket.

Venerable

There are a couple of older projects that often get mentioned

- Dispy: Embarrassingly parallel function evaluation
- Pyro: Remote objects / RPC

3.31.3 Relationship

In relation to these projects `distributed`...

- Supports data-local computation like Hadoop and Spark
- Uses a task graph with data dependencies abstraction like Luigi
- In support of ad-hoc applications, like IPython Parallel and Scoop

3.31.4 In depth comparison to particular projects

IPython Parallel

Short Description

IPython Parallel is a distributed computing framework from the IPython project. It uses a centralized hub to farm out jobs to several `ipengine` processes running on remote workers. It communicates over ZeroMQ sockets and centralizes communication through the central hub.

IPython parallel has been around for a while and, while not particularly fancy, is quite stable and robust.

IPython Parallel offers parallel `map` and remote `apply` functions that route computations to remote workers

```
>>> view = Client(...)[:]
>>> results = view.map(func, sequence)
>>> result = view.apply(func, *args, **kwargs)
>>> future = view.apply_async(func, *args, **kwargs)
```

It also provides direct execution of code in the remote process and collection of data from the remote namespace.

```
>>> view.execute('x = 1 + 2')
>>> view['x']
[3, 3, 3, 3, 3, 3]
```

Brief Comparison

Distributed and IPython Parallel are similar in that they provide `map` and `apply/submit` abstractions over distributed worker processes running Python. Both manage the remote namespaces of those worker processes.

They are dissimilar in terms of their maturity, how worker nodes communicate to each other, and in the complexity of algorithms that they enable.

Distributed Advantages

The primary advantages of `distributed` over IPython Parallel include

1. Peer-to-peer communication between workers
2. Dynamic task scheduling

`Distributed` workers share data in a peer-to-peer fashion, without having to send intermediate results through a central bottleneck. This allows `distributed` to be more effective for more complex algorithms and to manage larger datasets in a more natural manner. IPython parallel does not provide a mechanism for workers to communicate with each other, except by using the central node as an intermediary for data transfer or by relying on some other medium, like a shared file system. Data transfer through the central node can easily become a bottleneck and so IPython parallel has been mostly helpful in embarrassingly parallel work (the bulk of applications) but has not been used extensively for more sophisticated algorithms that require non-trivial communication patterns.

The distributed executor includes a dynamic task scheduler capable of managing deep data dependencies between tasks. The IPython parallel docs include a recipe for executing task graphs with data dependencies. This same idea is core to all of `distributed`, which uses a dynamic task scheduler for all operations. Notably, `distributed.Future` objects can be used within `submit/map/get` calls before they have completed.

```
>>> x = executor.submit(f, 1)   # returns a future
>>> y = executor.submit(f, 2)   # returns a future
>>> z = executor.submit(add, x, y)   # consumes futures
```

The ability to use futures cheaply within `submit` and `map` methods enables the construction of very sophisticated data pipelines with simple code. Additionally, distributed can serve as a full dask task scheduler, enabling support for distributed arrays, dataframes, machine learning pipelines, and any other application build on dask graphs. The dynamic task schedulers within `distributed` are adapted from the dask task schedulers and so are fairly sophisticated/efficient.

IPython Parallel Advantages

IPython Parallel has the following advantages over `distributed`

1. Maturity: IPython Parallel has been around for a while.
2. Explicit control over the worker processes: IPython parallel allows you to execute arbitrary statements on the workers, allowing it to serve in system administration tasks.
3. Deployment help: IPython Parallel has mechanisms built-in to aid deployment on SGE, MPI, etc.. Distributed does not have any such sugar, though is fairly simple to set up by hand.
4. Various other advantages: Over the years IPython parallel has accrued a variety of helpful features like IPython interaction magics, `@parallel` decorators, etc..

concurrent.futures

The `distributed.Executor` API is modeled after `concurrent.futures` and PEP-3184. It has a few notable differences:

- `distributed` accepts `Future` objects within calls to `submit/map`. It is preferable to submit Future objects directly rather than wait on them before submission.
- The `map` function returns `Future` objects, not concrete results. The `map` function returns immediately.
- It is not yet possible to cancel a `Future` (though this is theoretically possible please raise an issue if this is of concrete importance to you.)
- Distributed generally does not support timeouts or callbacks

`distributed.CompatibleExecutor` is a subclass of `distributed.Executor` that does conform to the `concurrent.futures` API, allowing it to be used as a drop-in replacement for other Executors using the common API.