Dask.distributed Documentation

Release 1.18.0+44.ga7adb35

Matthew Rocklin

Aug 16, 2017

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.

CHAPTER 1

Motivation

`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 TCP.

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 `client.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 `client.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

Install Dask.Distributed

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

Conda

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

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

Pip

Or install distributed with `pip`:

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

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
```

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
```

Quickstart

Install

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

See *installation* document for more information.

Setup Dask.distributed the Easy Way

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

```
>>> from dask.distributed import Client
>>> client = Client()  # set up local cluster on your laptop
>>> client
<Client: scheduler="127.0.0.1:8786" processes=8 cores=8>
```

Setup Dask.distributed the Hard Way

This allows dask.distributed to use multiple machines as workers.

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 Client and point it to the IP/port of the scheduler.

```
>>> from dask.distributed import Client
>>> client = Client('127.0.0.1:8786')
```

See *setup* for advanced use.

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 = client.map(square, range(10))
>>> B = client.map(neg, A)
>>> total = client.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 `Client.gather` method for many futures at once.

```
>>> total.result()    # result for single future
-285
>>> client.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.

```
>>> client.restart()
```

See *client* for advanced use.

Setup Network

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

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. Additionally, cluster SSH tools exist to send the same commands to many machines. One example is tmux-cssh.

Note:

- The scheduler and worker both need to accept TCP connections. By default the scheduler uses port 8786 and the worker binds to a random open port. If you are behind a firewall then you may have to open particular ports or tell Dask to use particular ports with the `--port` and `-worker-port` keywords. Other ports like 8787, 8788, and 8789 are also useful to keep open for the diagnostic web interfaces.
- More information about relevant ports is available by looking at the help pages with `dask-scheduler --help` and `dask-worker --help`

Using SSH

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
```

The `dask-ssh` utility depends on the `paramiko`:

```
pip install paramiko
```

Using a Shared Network File System and a Job Scheduler

Some clusters benefit from a shared network file system (NFS) and can use this to communicate the scheduler location to the workers:

```
dask-scheduler --scheduler-file /path/to/scheduler.json

dask-worker --scheduler-file /path/to/scheduler.json
dask-worker --scheduler-file /path/to/scheduler.json
```

```
>>> client = Client(scheduler_file='/path/to/scheduler.json')
```

This can be particularly useful when deploying `dask-scheduler` and `dask-worker` processes using a job scheduler like `SGE/SLURM/Torque/etc..` Here is an example using SGE's `qsub` command:

```
# Start a dask-scheduler somewhere and write connection information to file
qsub -b y /path/to/dask-scheduler --scheduler-file /path/to/scheduler.json

# Start 100 dask-worker processes in an array job pointing to the same file
qsub -b y -t 1-100 /path/to/dask-worker --scheduler-file /path/to/scheduler.json
```

Note, the `--scheduler-file` option is *only* valuable if your scheduler and workers share a standard POSIX file system.

Using the Python API

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

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

```
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('tcp://:8786')   # Listen on TCP port 8786
```

On other nodes start worker processes that point to the URL 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('tcp://127.0.0.1:8786', loop=loop)
w.start()  # 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. This allows workers to restart themselves in case of failure, provides some additional monitoring, and is useful when coordinating many workers that should live in different processes to avoid the GIL.

Using LocalCluster

You can do the work above easily using *LocalCluster*.

```
from distributed import LocalCluster
c = LocalCluster(processes=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.

Using Amazon EC2

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

Cluster Resource Managers

Dask.distributed has been deployed on dozens of different cluster resource managers. This section contains links to some external projects, scripts, and instructions that may serve as useful starting points.

Kubernetes

- https://github.com/martindurant/dask-kubernetes
- https://github.com/ogrisel/docker-distributed
- https://github.com/hammerlab/dask-distributed-on-kubernetes/

Marathon

- https://github.com/mrocklin/dask-marathon

DRMAA (SGE, SLURM, Torque, etc..)

- https://github.com/dask/dask-drmaa
- https://github.com/mfouesneau/dasksge

YARN

- https://github.com/dask/dask-yarn
- https://knit.readthedocs.io/en/latest/

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, using Conda environments, or through any of the other means typically employed by cluster administrators.

Windows

Note:

- Running a `dask-scheduler` on Windows architectures is supported for only a limited number of workers (roughly 100). This is a detail of the underlying tcp server implementation and is discussed here.
- Running `dask-worker` processes on Windows is well supported, performant, and without limit.

If you wish to run in a primarily Windows environment, it is recommneded to run a `dask-scheduler` on a linux or MacOSX environment, with `dask-worker` workers on the Windows boxes. This works because the scheduler environment is de-coupled from that of the workers.

Customizing initialization

Both `dask-scheduler` and `dask-worker` support a `--preload` option that allows custom initialization of each scheduler/worker respectively. A module or python file passed as a `--preload` value is guaranteed to be imported before establishing any connection. A `dask_setup(service)` function is called if found, with a `Scheduler` or `Worker` instance as the argument. As the service stops, `dask_teardown(service)` is called if present.

As an example, consider the following file that creates a *scheduler plugin* and registers it with the scheduler

```
# scheduler-setup.py
from distributed.diagnostics.plugin import SchedulerPlugin

class MyPlugin(SchedulerPlugin):
    def add_worker(self, scheduler=None, worker=None, **kwargs):
        print("Added a new worker at", worker)

def dask_setup(scheduler):
    plugin = MyPlugin()
    scheduler.add_plugin(plugin)
```

We can then run this preload script by referring to its filename (or module name if it is on the path) when we start the scheduler:

```
dask-scheduler --preload scheduler-setup.py
```

Client

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

After we *setup a cluster*, we initialize a `Client` by pointing it to the address of a `Scheduler`:

```
>>> from distributed import Client
>>> client = Client('127.0.0.1:8786')
```

There are a few different ways to interact with the cluster through the client:

1. The Client satisfies most of the standard concurrent.futures - PEP-3148 interface with `.submit`, `.map` functions and `Future` objects, allowing the immediate and direct submission of tasks.
2. The Client registers itself as the default Dask scheduler, and so runs all dask collections like dask.array, dask.bag, dask.dataframe and dask.delayed
3. The Client has additional methods for manipulating data remotely. See the full *API* for a thorough list.

Concurrent.futures

We can submit individual function calls with the `client.submit` method or many function calls with the `client.map` method

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

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

>>> L = client.map(inc, range(1000))
>>> L
[<Future - key: inc-e4853cffcc2f51909cdb69d16dacd1a5>,
 <Future - key: inc-...>,
 <Future - key: inc-...>,
 <Future - key: inc-...>, ...]
```

These results live on distributed workers.

We can submit tasks on futures. The function will go to the machine where the futures are stored and run on the result once it has completed.

```
>>> y = client.submit(inc, x)       # Submit on x, a Future
>>> total = client.submit(sum, L)   # Map on L, a list of Futures
```

We gather back the results using either the `Future.result` method for single futures or `client.gather` method for many futures at once.

```
>>> x.result()
11

>>> client.gather(L)
[1, 2, 3, 4, 5, ...]
```

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.

Dask

The parent library Dask contains objects like dask.array, dask.dataframe, dask.bag, and dask.delayed, which automatically produce parallel algorithms on larger datasets. All dask collections work smoothly with the distributed scheduler.

When we create a `Client` object it registers itself as the default Dask scheduler. All `.compute()` methods will automatically start using the distributed system.

```
client = Client('scheduler:8786')

my_dataframe.sum().compute()  # Now uses the distributed system by default
```

We can stop this behavior by using the `set_as_default=False` keyword argument when starting the Client.

Dask's normal `.compute()` methods are *synchronous*, meaning that they block the interpreter until they complete. Dask.distributed allows the new ability of *asynchronous* computing, we can trigger computations to occur in the background and persist in memory while we continue doing other work. This is typically handled with the `Client.persist` and `Client.compute` methods which are used for larger and smaller result sets respectively.

```
>>> df = client.persist(df)  # trigger all computations, keep df in memory
>>> type(df)
dask.DataFrame
```

For more information see the page on *Managing Computation*.

Pure Functions by Default

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

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

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

This key should be the same across all computations with the same inputs and across all machines. If we run the computation above on any computer with the same environment then we 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
>>> client.submit(np.random.random, 1000, pure=False).key
'random_sample-fc814a39-ee00-42f3-8b6f-cac65bcb5556'
>>> client.submit(np.random.random, 1000, pure=False).key
'random_sample-a24e7220-a113-47f2-a030-72209439f093'
```

Tornado Coroutines

If we are operating in an asynchronous environment then the blocking functions listed above become have asynchronous equivalents. You must start your client with the `asynchronous=True` keyword and `yield` or `await` blocking functions.

```
@gen.coroutine
def f():
    client = yield Client(asynchronous=True)
    future = client.submit(func, *args)
    result = yield future
    return result
```

If you want to reuse the same client in asynchronous and synchronous environments you can apply the `asynchronous=True` keyword at each method call.

```
client = Client()  # normal blocking client

@gen.corotuine
def f():
    futures = client.map(func, L)
    results = yield client.gather(futures, asynchronous=True)
    return results
```

See the *Asynchronous* documentation for more information.

Additional Links

For more information on how to use dask.distributed you may want to look at the following pages:

- *Managing Memory*
- *Managing Computation*
- *Data Locality*
- *API*

API

Client

`Client`([address, loop, timeout, ...])	Connect to and drive computation on a distributed Dask cluster
`Client.cancel`(futures[, asynchronous])	Cancel running futures
`Client.close`([timeout])	Close this client
`Client.compute`(collections[, sync, ...])	Compute dask collections on cluster
`Client.gather`(futures[, errors, maxsize, ...])	Gather futures from distributed memory
`Client.get`(dsk, keys[, restrictions, ...])	Compute dask graph
`Client.get_dataset`(name, **kwargs)	Get named dataset from the scheduler
`Client.get_executor`(**kwargs)	Return a concurrent.futures Executor for submitting tasks on this Client.
`Client.has_what`([workers])	Which keys are held by which workers

Continued on next page

Table 3.1 – continued from previous page

Client.list_datasets(**kwargs)	List named datasets available on the scheduler
Client.map(func, *iterables, **kwargs)	Map a function on a sequence of arguments
Client.ncores([workers])	The number of threads/cores available on each worker node
Client.persist(collections[, ...])	Persist dask collections on cluster
Client.publish_dataset(**kwargs)	Publish named datasets to scheduler
Client.rebalance([futures, workers])	Rebalance data within network
Client.replicate(futures[, n, workers, ...])	Set replication of futures within network
Client.restart(**kwargs)	Restart the distributed network
Client.run(function, *args, **kwargs)	Run a function on all workers outside of task scheduling system
Client.run_on_scheduler(function, *args, ...)	Run a function on the scheduler process
Client.scatter(data[, workers, broadcast, ...])	Scatter data into distributed memory
Client.scheduler_info(**kwargs)	Basic information about the workers in the cluster
Client.start_ipython_workers([workers, ...])	Start IPython kernels on workers
Client.start_ipython_scheduler([magic_name, ...])	Start IPython kernel on the scheduler
Client.submit(func, *args, **kwargs)	Submit a function application to the scheduler
Client.unpublish_dataset(name, **kwargs)	Remove named datasets from scheduler
Client.upload_file(filename, **kwargs)	Upload local package to workers
Client.who_has([futures])	The workers storing each future's data

worker_client(*args, **kwds)	Get client for this thread
get_worker()	Get the worker currently running this task
get_client([address, timeout])	Get a client while within a task
secede()	Have this task secede from the worker's thread pool

ReplayExceptionClient.get_futures_error(future)	Ask the scheduler details of the sub-task of the given failed future
ReplayExceptionClient.recreate_error_locally(future)	For a failed calculation, perform the blamed task locally for debugging.

Future

Future(key, client[, inform, state])	A remotely running computation
Future.add_done_callback(fn)	Call callback on future when callback has finished
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([timeout])	Return the exception of a failed task
Future.result([timeout])	Wait until computation completes, gather result to local process.
Future.traceback([timeout])	Return the traceback of a failed task

Client Coordination

Queue([name, client, maxsize])	Distributed Queue

Variable([name, client, maxsize])	Distributed Global Variable

Other

as_completed([futures, loop, with_results])	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
fire_and_forget(obj)	Run tasks at least once, even if we release the futures

Asynchronous methods

Most methods and functions can be used equally well within a blocking or asynchronous environment using Tornado coroutines. If used within a Tornado IOLoop then you should yield or await otherwise blocking operations appropriately.

You must tell the client that you intend to use it within an asynchronous environment by passing the `asynchronous=True` keyword

```
# blocking
client = Client()
future = client.submit(func, *args)  # immediate, no blocking/async difference
result = client.gather(future)  # blocking

# asynchronous Python 2/3
client = yield Client(asynchronous=True)
future = client.submit(func, *args)  # immediate, no blocking/async difference
result = yield client.gather(future)  # non-blocking/asynchronous

# asynchronous Python 3
client = await Client(asynchronous=True)
future = client.submit(func, *args)  # immediate, no blocking/async difference
result = await client.gather(future)  # non-blocking/asynchronous
```

The asynchronous variants must be run within a Tornado coroutine. See the *Asynchronous* documentation for more information.

Client

class distributed.**Client**(*address=None*, *loop=None*, *timeout=5*, *set_as_default=True*, *scheduler_file=None*, *security=None*, *start=None*, *asynchronous=False*, ***kwargs*)

Connect to and drive computation on a distributed Dask cluster

The Client connects users to a dask.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, or Cluster

This can be the address of a `Scheduler` server like a string `'127.0.0.1:8786'` or a cluster object like `LocalCluster()`

timeout: int

Timeout duration for initial connection to the scheduler

set_as_default: bool (True)

Claim this scheduler as the global dask scheduler

scheduler_file: string (optional)

Path to a file with scheduler information if available

security: (optional)

Optional security information

asynchronous: bool (False by default)

Set to True if this client will be used within a Tornado event loop

See also:

distributed.scheduler.Scheduler Internal scheduler

Examples

Provide cluster's scheduler node address on initialization:

```
>>> client = Client('127.0.0.1:8786')
```

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

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

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

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

Gather results with the `gather` method.

```
>>> client.gather(c)
33
```

asynchronous

Are we running in the event loop?

This is true if the user signaled that we might be when creating the client as in the following:

```
client = Client(asynchronous=True)
```

However, we override this expectation if we can definitively tell that we are running from a thread that is not the event loop. This is common when calling get_client() from within a worker task. Even though the client was originally created in asynchronous mode we may find ourselves in contexts when it is better to operate synchronously.

cancel (*futures*, *asynchronous=None*)

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

channel (**args*, ***kwargs*)
Deprecated: see dask.distributed.Queue instead

close (*timeout=10*)
Close this client

Clients will also close automatically when your Python session ends

If you started a client without arguments like `Client()` then this will also close the local cluster that was started at the same time.

See also:

Client.restart

compute (*collections*, *sync=False*, *optimize_graph=True*, *workers=None*, *allow_other_workers=False*, *resources=None*, ***kwargs*)
Compute dask collections on cluster

Parameters **collections: 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

workers: str, list, dict

Which workers can run which parts of the computation If a string a list then the output collections will run on the listed

workers, but other sub-computations can run anywhere

If a dict then keys should be (tuples of) collections and values should be addresses or lists.

allow_other_workers: bool, list

If True then all restrictions in workers= are considered loose If a list then only the keys for the listed collections are loose

****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:

Client.get Normal synchronous dask.get function

Examples

```
>>> from dask import delayed
>>> from operator import add
>>> x = delayed(add)(1, 2)
>>> y = delayed(add)(x, x)
>>> xx, yy = client.compute([x, y])
```

```
>>> xx
<Future: status: finished, key: add-8f6e709446674bad78ea8aeecfee188e>
>>> xx.result()
3
>>> yy.result()
6
```

Also support single arguments

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

gather (*futures*, *errors='raise'*, *maxsize=0*, *direct=None*, *asynchronous=None*)
Gather futures from distributed memory

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

Parameters **futures: Collection of futures**

This can be a possibly nested collection of Future objects. Collections can be lists, sets, iterators, queues or dictionaries

errors: string

Either 'raise' or 'skip' if we should raise if a future has erred or skip its inclusion in the output collection

maxsize: int

If the input is a queue then this produces an output queue with a maximum size.

Returns results: a collection of the same type as the input, but now with

gathered results rather than futures

See also:

Client.scatter Send data out to cluster

Examples

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

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

get (*dsk*, *keys*, *restrictions=None*, *loose_restrictions=None*, *resources=None*, *sync=True*, *asynchronous=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

sync: bool (optional)

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

See also:

Client.compute Compute asynchronous collections

Examples

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

get_dataset (*name*, ***kwargs*)
Get named dataset from the scheduler

See also:

Client.publish_dataset, *Client.list_datasets*

get_executor (***kwargs*)
Return a concurrent.futures Executor for submitting tasks on this Client.

Parameters ****kwargs:**

Any submit()- or map()- compatible arguments, such as *workers* or *resources*.

Returns An Executor object that's fully compatible with the concurrent.futures

API.

static get_restrictions (*collections*, *workers*, *allow_other_workers*)
Get restrictions from inputs to compute/persist

get_versions (*check=False*)
Return version info for the scheduler, all workers and myself

Parameters **check** : boolean, default False

raise ValueError if all required & optional packages do not match

Examples

```
>>> c.get_versions()
```

has_what (*workers=None*, ***kwargs*)
Which keys are held by which workers

Parameters **workers: list (optional)**

A list of worker addresses, defaults to all

See also:

Client.who_has, *Client.ncores*

Examples

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

list_datasets (***kwargs*)
List named datasets available on the scheduler

See also:

Client.publish_dataset, *Client.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

key: str, list

Prefix for task names if string. Explicit names if list.

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:

Client.submit Submit a single function

Examples

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

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

Client.who_has

Examples

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

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

ncores(*workers=None*, ***kwargs*)

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:

Client.who_has, *Client.has_what*

Examples

```
>>> c.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}
```

normalize_collection(*collection*)

Replace collection's tasks by already existing futures if they exist

This normalizes the tasks within a collections task graph against the known futures within the scheduler. It returns a copy of the collection with a task graph that includes the overlapping futures.

See also:

Client.persist trigger computation of collection's tasks

Examples

```
>>> len(x.dask)  # x is a dask collection with 100 tasks
100
>>> set(client.futures).intersection(x.dask)  # some overlap exists
10
```

```
>>> x = client.normalize_collection(x)
>>> len(x.dask)  # smaller computational graph
20
```

persist(*collections*, *optimize_graph=True*, *workers=None*, *allow_other_workers=None*, *resources=None*, ***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

workers: str, list, dict

Which workers can run which parts of the computation If a string a list then the output collections will run on the listed

workers, but other sub-computations can run anywhere

If a dict then keys should be (tuples of) collections and values should be addresses or lists.

allow_other_workers: bool, list

If True then all restrictions in workers= are considered loose If a list then only the keys for the listed collections are loose

kwargs:

Options to pass to the graph optimize calls

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

See also:

Client.compute

Examples

```
>>> xx = client.persist(x)
>>> xx, yy = client.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:

Client.stacks, *Client.who_has*, *Client.has_what*, *Client.ncores*

Examples

```
>>> x, y, z = c.map(inc, [1, 2, 3])
>>> c.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 Clients which can download the collection or futures with `get_dataset`.

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

Parameters **kwargs: dict**

named collections to publish on the scheduler

Returns None

See also:

Client.list_datasets, *Client.get_dataset*, *Client.unpublish_dataset*, *Client.persist*

Examples

Publishing client:

```
>>> df = dd.read_csv('s3://...')
>>> df = c.persist(df)
>>> c.publish_dataset(my_dataset=df)
```

Receiving client:

```
>>> c.list_datasets()
['my_dataset']
>>> df2 = c.get_dataset('my_dataset')
```

rebalance(*futures=None*, *workers=None*, ***kwargs*)

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*, ***kwargs*)
Set replication of futures within network

Copy data onto many workers. This helps to broadcast frequently accessed data and it helps to improve resilience.

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:

Client.rebalance

Examples

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

restart (***kwargs*)
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.

If your function takes an input argument named `dask_worker` then that variable will be populated with the worker itself.

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

```
>>> c.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.

```
>>> c.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}
```

```
>>> def get_status(dask_worker):
...     return dask_worker.status
```

```
>>> c.run(get_hostname)
{'192.168.0.100:9000': 'running',
 '192.168.0.101:9000': 'running}
```

run_coroutine (*function*, **args*, ***kwargs*)
Spawn a coroutine on all workers.

This spaws a coroutine on all currently known workers and then waits for the coroutine on each worker. The coroutines' results are returned as a dictionary keyed by worker address.

Parameters **function: a coroutine function**

(typically a function wrapped in gen.coroutine or a Python 3.5+ async function)

***args: arguments for remote function**

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

wait: boolean (default True)

Whether to wait for coroutines to end.

workers: list

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

run_on_scheduler (*function*, **args*, ***kwargs*)
Run a function on the scheduler process

This is typically used for live debugging. The function should take a keyword argument `dask_scheduler=`, which will be given the scheduler object itself.

See also:

Client.run Run a function on all workers

Client.start_ipython_scheduler Start an IPython session on scheduler

Examples

```
>>> def get_number_of_tasks(dask_scheduler=None):
...     return len(dask_scheduler.task_state)
```

```
>>> client.run_on_scheduler(get_number_of_tasks)
100
```

scatter(*data*, *workers=None*, *broadcast=False*, *direct=None*, *hash=True*, *maxsize=0*, *timeout=3*, *asynchronous=None*)

Scatter data into distributed memory

This moves data from the local client process into the workers of the distributed scheduler. Note that it is often better to submit jobs to your workers to have them load the data rather than loading data locally and then scattering it out to them.

Parameters **data: list, iterator, dict, Queue, or object**

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.

direct: bool (defaults to automatically check)

Send data directly to workers, bypassing the central scheduler This avoids burdening the scheduler but assumes that the client is able to talk directly with the workers.

maxsize: int (optional)

Maximum size of queue if using queues, 0 implies infinite

hash: bool (optional)

Whether or not to hash data to determine key. If False then this uses a random key

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

See also:

Client.gather Gather data back to local process

Examples

```
>>> c = Client('127.0.0.1:8787')
>>> c.scatter(1)
<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>
```

```
>>> c.scatter([1, 2, 3])
[<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>,
 <Future: status: finished, key: 58e78e1b34eb49a68c65b54815d1b158>,
 <Future: status: finished, key: d3395e15f605bc35ab1bac6341a285e2>]
```

```
>>> c.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

```
>>> c.scatter([1, 2, 3], workers=[('hostname', 8788)])
```

Handle streaming sequences of data with iterators or queues

```
>>> seq = c.scatter(iter([1, 2, 3]))
>>> next(seq)
<Future: status: finished, key: c0a8a20f903a4915b94db8de3ea63195>,
```

Broadcast data to all workers

```
>>> [future] = c.scatter([element], broadcast=True)
```

scheduler_info(***kwargs*)

Basic information about the workers in the cluster

Examples

```
>>> c.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*)

Close this client

Clients will also close automatically when your Python session ends

If you started a client without arguments like `Client()` then this will also close the local cluster that was started at the same time.

See also:

Client.restart

stacks(*workers=None*)

The task queues on each worker

Parameters workers: list (optional)

A list of worker addresses, defaults to all

See also:

Client.processing, *Client.who_has*, *Client.has_what*, *Client.ncores*

Examples

```
>>> x, y, z = c.map(inc, [1, 2, 3])
>>> c.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:

Client.start_ipython_workers Start IPython on the workers

Examples

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

```
>>> c.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. If string has asterix then expand asterix into 0, 1, ..., n for n 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:

Client.start_ipython_scheduler start ipython on the scheduler

Examples

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

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

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

Add asterix * in magic names to add one magic per worker

```
>>> c.start_ipython_workers(magic_names='w_*')
>>> %w_0 worker.data
{'x': 1, 'y': 100}
>>> %w_1 worker.data
{'z': 5}
```

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:

Client.map Submit on many arguments at once

Examples

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

unpublish_dataset (*name*, ***kwargs*)
Remove named datasets from scheduler

See also:

Client.publish_dataset

Examples

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

upload_file (*filename*, ***kwargs*)
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, .pyc, .egg or .zip files will be importable.

Parameters filename: string

Filename of .py, .pyc, .egg or .zip file to send to workers

Examples

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

who_has (*futures=None*, ***kwargs*)
The workers storing each future's data

Parameters futures: list (optional)

A list of futures, defaults to all data

See also:

Client.has_what, *Client.ncores*

Examples

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

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

class distributed.recreate_exceptions.**ReplayExceptionClient**(*client*)

A plugin for the client allowing replay of remote exceptions locally

Adds the following methods (and their async variants)to the given client:

- recreate_error_locally: main user method
- **get_futures_error: gets the task, its details and dependencies,** responsible for failure of the given future.

get_futures_error(*future*)

Ask the scheduler details of the sub-task of the given failed future

When a future evaluates to a status of "error", i.e., an exception was raised in a task within its graph, we an get information from the scheduler. This function gets the details of the specific task that raised the exception and led to the error, but does not fetch data from the cluster or execute the function.

Parameters **future** : future that failed, having status=="error", typically

after an attempt to gather() shows a stack-stace.

Returns Tuple:

- the function that raised an exception
- argument list (a tuple), may include values and keys
- keyword arguments (a dictionary), may include values and keys
- list of keys that the function requires to be fetched to run

See also:

ReplayExceptionClient.recreate_error_locally

recreate_error_locally(*future*)

For a failed calculation, perform the blamed task locally for debugging.

This operation should be performed after a future (result of gather, compute, etc) comes back with a status of "error", if the stack- trace is not informative enough to diagnose the problem. The specific task (part of the graph pointing to the future) responsible for the error will be fetched from the scheduler, together with the values of its inputs. The function will then be executed, so that pdb can be used for debugging.

Parameters **future** : future or collection that failed

The same thing as was given to gather, but came back with an exception/stack-trace. Can also be a (persisted) dask collection containing any errored futures.

Returns Nothing; the function runs and should raise an exception, allowing

the debugger to run.

Examples

```
>>> future = c.submit(div, 1, 0)
>>> future.status
'error'
```

```
>>> c.recreate_error_locally(future)
ZeroDivisionError: division by zero
```

If you're in IPython you might take this opportunity to use pdb

```
>>> %pdb
Automatic pdb calling has been turned ON
```

```
>>> c.recreate_error_locally(future)
ZeroDivisionError: division by zero
      1 def div(x, y):
----> 2     return x / y
ipdb>
```

Future

class distributed.**Future**(*key*, *client*, *inform=False*, *state=None*)

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:

Client Creates futures

Examples

Futures typically emerge from Client computations

```
>>> my_future = client.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()
```

add_done_callback(*fn*)

Call callback on future when callback has finished

The callback `fn` should take the future as its only argument. This will be called regardless of if the future completes successfully, errs, or is cancelled

The callback is executed in a separate thread.

cancel()

Returns True if the future has been cancelled

cancelled()

Returns True if the future has been cancelled

done()
Is the computation complete?

exception(*timeout=None*, ***kwargs*)
Return the exception of a failed task

If *timeout* seconds are elapsed before returning, a TimeoutError is raised.

See also:

Future.traceback

result(*timeout=None*)
Wait until computation completes, gather result to local process.

If *timeout* seconds are elapsed before returning, a TimeoutError is raised.

traceback(*timeout=None*, ***kwargs*)
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.

If *timeout* seconds are elapsed before returning, a TimeoutError is raised.

See also:

Future.exception

Examples

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

Other

distributed.**as_completed**(*futures=None*, *loop=None*, *with_results=False*)
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.

Additionally, you can also add more futures to this object during computation with the `.add` method

Examples

```
>>> x, y, z = client.map(inc, [1, 2, 3])
>>> for future in as_completed([x, y, z]):
...     print(future.result())
3
2
4
```

Add more futures during computation

```
>>> x, y, z = client.map(inc, [1, 2, 3])
>>> ac = as_completed([x, y, z])
>>> for future in ac:
...     print(future.result())
...     if random.random() < 0.5:
...         ac.add(c.submit(double, future))
4
2
8
3
6
12
24
```

Optionally wait until the result has been gathered as well

```
>>> ac = as_completed([x, y, z], results=True)
>>> for future, result in ac:
...     print(result)
2
4
3
```

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)

Notes

In the notebook, the output of *progress* must be the last statement in the cell. Typically, this means calling *progress* at the end of a cell.

Examples

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

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

Parameters **fs: list of futures**

timeout: number, optional

Time in seconds after which to raise a gen.TimeoutError

Returns Named tuple of completed, not completed

distributed.**worker_client**(**args*, ***kwds*)
Get client for this thread

This context manager is intended to be called within functions that we run on workers. When run as a context manager it delivers a client `Client` object that can submit other tasks directly from that worker.

Parameters **timeout: Number**

Timeout after which to err

separate_thread: bool, optional

Whether to run this function outside of the normal thread pool defaults to True

See also:

get_worker, *get_client*, *secede*

Examples

```
>>> def func(x):
...     with worker_client() as c:  # connect from worker back to scheduler
...         a = c.submit(inc, x)     # this task can submit more tasks
...         b = c.submit(dec, x)
...         result = c.gather([a, b])  # and gather results
...     return result
```

```
>>> future = client.submit(func, 1)  # submit func(1) on cluster
```

distributed.**get_worker**()
Get the worker currently running this task

See also:

get_client, *worker_client*

Examples

```
>>> def f():
...     worker = get_worker()  # The worker on which this task is running
...     return worker.address
```

```
>>> future = client.submit(f)
>>> future.result()
'tcp://127.0.0.1:47373'
```

distributed.**get_client**(*address=None*, *timeout=3*)
Get a client while within a task

This client connects to the same scheduler to which the worker is connected

See also:

get_worker, *worker_client*, *secede*

Examples

```
>>> def f():
...     client = get_client()
...     futures = client.map(lambda x: x + 1, range(10))  # spawn many tasks
...     results = client.gather(futures)
...     return sum(results)
```

```
>>> future = client.submit(f)
>>> future.result()
55
```

distributed.**secede**()
Have this task secede from the worker's thread pool

This opens up a new scheduling slot and a new thread for a new task.

See also:

get_client, *get_worker*

Examples

```
>>> def mytask(x):
...     # do some work
...     client = get_client()
...     futures = client.map(...)  # do some remote work
...     secede()  # while that work happens, remove ourself from the pool
...     return client.gather(futures)  # return gathered results
```

class distributed.**Queue**(*name=None*, *client=None*, *maxsize=0*)
Distributed Queue

This allows multiple clients to share futures or small bits of data between each other with a multi-producer/multi-consumer queue. All metadata is sequentialized through the scheduler.

Elements of the Queue must be either Futures or msgpack-encodable data (ints, strings, lists, dicts). All data is sent through the scheduler so it is wise not to send large objects. To share large objects scatter the data and share the future instead.

> **Warning:** This object is experimental and has known issues in Python 2

See also:

Variable shared variable between clients

Examples

```
>>> from dask.distributed import Client, Queue
>>> client = Client()
>>> queue = Queue('x')
>>> future = client.submit(f, x)
>>> queue.put(future)
```

get(*timeout=None*, *batch=False*, ***kwargs*)
Get data from the queue

Parameters **timeout: Number (optional)**

Time in seconds to wait before timing out

batch: boolean, int (optional)

If True then return all elements currently waiting in the queue. If an integer than return that many elements from the queue If False (default) then return one item at a time

put(*value*, *timeout=None*, ***kwargs*)
Put data into the queue

qsize(***kwargs*)
Current number of elements in the queue

class distributed.**Variable**(*name=None*, *client=None*, *maxsize=0*)
Distributed Global Variable

This allows multiple clients to share futures and data between each other with a single mutable variable. All metadata is sequentialized through the scheduler. Race conditions can occur.

Values must be either Futures or msgpack-encodable data (ints, lists, strings, etc..) All data will be kept and sent through the scheduler, so it is wise not to send too much. If you want to share a large amount of data then `scatter` it and share the future instead.

Warning: This object is experimental and has known issues in Python 2

See also:

Queue

Examples

```
>>> from dask.distributed import Client, Variable
>>> client = Client()
>>> x = Variable('x')
>>> x.set(123)  # docttest: +SKIP
>>> x.get()  # docttest: +SKIP
123
>>> future = client.submit(f, x)
>>> x.set(future)
```

delete()
Delete this variable

Caution, this affects all clients currently pointing to this variable.

get(*timeout=None*, ***kwargs*)
Get the value of this variable

set(*value*, ***kwargs*)
Set the value of this variable

Parameters value: Future or object

Must be either a Future or a msgpack-encodable value

Asyncio Client

Frequently Asked Questions

More questions can be found on StackOverflow at http://stackoverflow.com/search?tab=votes&q=dask%20distributed

How do I use external modules?

Use `client.upload_file`. For more detail, see the API docs and a StackOverflow question "Can I use functions imported from .py files in Dask/Distributed?" This function supports both standalone file and setuptools's `.egg` files for larger modules.

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/

Error when running dask-worker about `OMP_NUM_THREADS`

For more problems with OMP_NUM_THREADS, see http://stackoverflow.com/questions/39422092/error-with-omp-num-threads-when-using-dask-distributed

Does Dask handle Data Locality?

Yes, both data locality in memory and data locality on disk.

Often it's *much* cheaper to move computations to where data lives. If one of your tasks creates a large array and a future task computes the sum of that array, you want to be sure that the sum runs on the same worker that has the array in the first place, otherwise you'll wait for a long while as the data moves between workers. Needless communication can easily dominate costs if we're sloppy.

The Dask Scheduler tracks the location and size of every intermediate value produced by every worker and uses this information when assigning future tasks to workers. Dask tries to make computations more efficient by minimizing data movement.

Sometimes your data is on a hard drive or other remote storage that isn't controlled by Dask. In this case the scheduler is unaware of exactly where your data lives, so you have to do a bit more work. You can tell Dask to preferentially run a task on a particular worker or set of workers.

For example Dask developers use this ability to build in data locality when we communicate to data-local storage systems like the Hadoop File System. When users use high-level functions like `dask.dataframe.read_csv('hdfs:///path/to/files.*.csv'` Dask talks to the HDFS name node, finds the locations of all of the blocks of data, and sends that information to the scheduler so that it can make smarter decisions and improve load times for users.

PermissionError [Errno 13] Permission Denied: '/root/.dask'

This error can be seen when starting distributed through the standard process control tool `supervisor` and running as a non-root user. This is caused by `supervisor` not passing the shell environment variables through to the subprocess, head to this section of the supervisor documentation to see how to pass the `$HOME` and `$USER` variables through.

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.

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

Slow

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

Fast

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

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

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.

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.

Limitations

Dask.distributed has limitations. Understanding these can help you to reliably create efficient distributed computations.

Performance

- The central scheduler spends a few hundred microseconds on every task. For optimal performance, task durations should be greater than 10-100ms.
- Dask can not parallelize within individual tasks. Individual tasks should be a comfortable size so as not to overwhelm any particular worker.
- Dask assigns tasks to workers heuristically. It *usually* makes the right decision, but non-optimal situations do occur.
- The workers are just Python processes, and inherit all capabilities and limitations of Python. They do not bound or limit themselves in any way. In production you may wish to run dask-workers within containers.

Assumptions on Functions and Data

Dask assumes the following about your functions and your data:

- All functions must be serializable either with pickle or cloudpickle. This is *usually* the case except in fairly exotic situations. The following should work:

```
from cloudpickle import dumps, loads
loads(dumps(my_object))
```

- All data must be serializable either with pickle, cloudpickle, or using Dask's custom serialization system.
- Dask may run your functions multiple times, such as if a worker holding an intermediate result dies. Any side effects should be idempotent.
-

Security

As a distributed computing framework, Dask enables the remote execution of arbitrary code. You should only host dask-workers within networks that you trust. This is standard among distributed computing frameworks, but is worth repeating.

Data Locality

Data movement often needlessly limits performance.

This is especially true for analytic computations. Dask.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.

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 = client.scatter(range(10))
```

Then Alice and Bob receive the following data

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

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 = client.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 = client.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 = client.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.

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

Specify workers with Compute/Persist

The `workers=` keyword in `scatter`, `submit`, and `map` is fairly straightforward, taking either a worker hostname, host:port pair or a sequence of those as valid inputs:

```
client.submit(f, x, workers='127.0.0.1')
client.submit(f, x, workers='127.0.0.1:55852')
client.submit(f, x, workers=['192.168.1.101', '192.168.1.100'])
```

For more complex computations, such as occur with dask collections like dask.dataframe or dask.delayed, we sometimes want to specify that certain parts of the computation run on certain workers while other parts run on other workers.

```
x = delayed(f)(1)
y = delayed(f)(2)
z = delayed(g)(x, y)

future = client.compute(z, workers={z: '127.0.0.1',
                                    x: '192.168.0.1'})
```

Here the values of the dictionary are of the same form as before, a host, a host:port pair, or a list of these. The keys in this case are either dask collections or tuples of dask collections. All of the *final* keys of these collections will run on the specified machines; dependencies can run anywhere unless they are also listed in `workers=`. We explore this through a set of examples:

The computation `z = f(x, y)` runs on the host `127.0.0.1`. The other two computations for `x` and `y` can run anywhere.

```
future = client.compute(z, workers={z: '127.0.0.1'})
```

The computations for both `z` and `x` must run on `127.0.0.1`

```
future = client.compute(z, workers={z: '127.0.0.1',
                                    x: '127.0.0.1'})
```

Use a tuple to group collections. This is shorthand for the above.

```
future = client.compute(z, workers={(x, y): '127.0.0.1'})
```

Recall that all options for `workers=` in `scatter/submit/map` hold here as well.

```
future = client.compute(z, workers={(x, y): ['192.168.1.100', '192.168.1.101:9999']})
```

Set `allow_other_workers=True` to make these loose restrictions rather than hard requirements.

```
future = client.compute(z, workers={(x, y): '127.0.0.1'},
                        allow_other_workers=True)
```

Provide a collection to `allow_other_workers=[...]` to say that the keys for only some of the collections are loose. In the case below `z` *must* run on `127.0.0.1` while `x` *should* run on `127.0.0.1` but can run elsewhere if necessary:

```
future = client.compute(z, workers={(x, y): '127.0.0.1'},
                        allow_other_workers=[x])
```

This works fine with `persist` and with any dask collection (any object with a `._keys()` method):

```
df = dd.read_csv('s3://...')
df = client.persist(df, workers={df: ...})
```

See the *efficiency* page to learn about best practices.

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 in the local process.
2. Lazy computations in a dask graph, perhaps stored in a `dask.delayed` or `dask.dataframe` object.
3. Running computations or remote data, represented by `Future` objects pointing to computations currently in flight.

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

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.

Dask Collections to Futures

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

client.compute

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

```
>>> df = dd.read_csv('s3://...')
>>> total = client.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 `client.compute` method is only appropriate when results are small and should fit in memory. The following would likely fail:

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

Instead, you should use `client.persist`

client.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 = client.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.

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 `client.scatter`, `client.submit` or `client.map`.

```
futures = client.scatter(args)                          # Send data
future = client.submit(function, *args, **kwrags)       # Send single task
futures = client.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.

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 `client.gather` method.

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

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

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 = client.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

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.

Creating Futures

The following functions produce Futures

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

The `submit` and `map` methods handle raw Python functions. The `compute` and `persist` methods handle Dask collections like arrays, bags, delayed values, and dataframes. The `scatter` method sends data directly from the local process.

Persisting Collections

Calls to `Client.compute` or `Client.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 = client.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 = client.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.

Difference with dask.compute

The operations `client.persist(df)` and `client.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 `client.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 = client.persist(df)     # This is good and asynchronously pins df
>>> df.x.sum().compute()  # This is good because the result is small
>>> future = client.compute(df.x.sum())  # This is also good but less intuitive
```

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
```

Aggressively Clearing Data

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

```
>>> client.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.

```
>>> client.restart()
```

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.

For more information see *Resilience*.

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.

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

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.

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.

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

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

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 client 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 = client.submit(f, 1)  # returns a future
>>> y = client.submit(f, 2)  # returns a future
>>> z = client.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.Client* API is modeled after `concurrent.futures` and **PEP 3184**. It has a few notable differences:

- `distributed` accepts `Future` objects within calls to `submit/map`. When chaining computations, it is preferable to submit Future objects directly rather than wait on them before submission.
- The `map()` method returns `Future` objects, not concrete results. The `map()` method returns immediately.
- Despite sharing a similar API, `distributed Future` objects cannot always be substituted for `concurrent.futures.Future` objects, especially when using `wait()` or `as_completed()`.
- Distributed generally does not support callbacks.

If you need full compatibility with the `concurrent.futures.Executor` API, use the object returned by the `get_executor()` method.

Resilience

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

User code failures

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

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

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

>>> y = client.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.

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. This function will be marked as "bad" after it kills a fixed number of workers (defaults to three).
3. Data scattered out to the workers is not kept in the scheduler (it is often quite large) and so the loss of this data is irreparable. You may wish to call `Client.replicate` on the data with a suitable replication factor to ensure that it remains long-lived or else back the data off of some resilient store, like a file system.

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.

Scheduler Failure

The process containing the scheduler might die. There is currently no persistence mechanism to record and recover the scheduler state.

The workers and clients will all reconnect to the scheduler after it comes back online but records of ongoing computations will be lost.

Restart and Nanny Processes

The client 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 `Client.restart` method kills all workers, flushes all scheduler state, and then brings all workers back online, resulting in a clean cluster.

Scheduling Policies

This document describes the policies used to select the preference of tasks and to select the preference of workers used by Dask's distributed scheduler. For more information on how this these policies are enacted efficiently see *Scheduling State*.

Choosing Workers

When a task transitions from waiting to a processing state we decide a suitable worker for that task. If the task has significant data dependencies or if the workers are under heavy load then this choice of worker can strongly impact global performance. Currently workers for tasks are determined as follows:

1. If the task has no major dependencies and no restrictions then we find the least occupied worker.
2. Otherwise, if a task has user-provided restrictions (for example it must run on a machine with a GPU) then we restrict the available pool of workers to just that set, otherwise we consider all workers
3. From among this pool of workers we determine the workers to whom the least amount of data would need to be transferred.
4. We break ties by choosing the worker that currently has the fewest tasks, counting both those tasks in memory and those tasks processing currently.

This process is easy to change (and indeed this document may be outdated). We encourage readers to inspect the `decide_worker` function in scheduler.py

decide_worker(dependencies, occupancy, ...)	Decide which worker should take task

Choosing Tasks

We often have a choice between running many valid tasks. There are a few competing interests that might motivate our choice:

1. Run tasks on a first-come-first-served basis for fairness between multiple clients
2. Run tasks that are part of the critical path in an effort to reduce total running time and minimize straggler workloads
3. Run tasks that allow us to release many dependencies in an effort to keep the memory footprint small
4. Run tasks that are related so that large chunks of work can be completely eliminated before running new chunks of work

Accomplishing all of these objectives simultaneously is impossible. Optimizing for any of these objectives perfectly can result in costly overhead. The heuristics with the scheduler do a decent but imperfect job of optimizing for all of these (they all come up in important workloads) quickly.

Last in, first out

When a worker finishes a task the immediate dependencies of that task get top priority. This encourages a behavior of finishing ongoing work immediately before starting new work. This often conflicts with the first-come-first-served objective but often results in shorter total runtimes and significantly reduced memory footprints.

Break ties with children and depth

Often a task has multiple dependencies and we need to break ties between them with some other objective. Breaking these ties has a surprisingly strong impact on performance and memory footprint.

When a client submits a graph we perform a few linear scans over the graph to determine something like the number of descendants of each node (not quite, because it's a DAG rather than a tree, but this is a close proxy). This number can be used to break ties and helps us to prioritize nodes with longer critical paths and nodes with many children. The actual algorithms used are somewhat more complex and are described in detail in dask/order.py

Initial Task Placement

When a new large batch of tasks come in and there are many idle workers then we want to give each worker a set of tasks that are close together/related and unrelated from the tasks given to other workers. This usually avoids inter-worker communication down the line. The same depth-first-with-child-weights priority given to workers described above can usually be used to properly segment the leaves of a graph into decently well separated sub-graphs with relatively low inter-sub-graph connectedness.

First-Come-First-Served, Coarsely

The last-in-first-out behavior used by the workers to minimize memory footprint can distort the task order provided by the clients. Tasks submitted recently may run sooner than tasks submitted long ago because they happen to be more

convenient given the current data in memory. This behavior can be *unfair* but improves global runtimes and system efficiency, sometimes quite significantly.

However, workers inevitably run out of tasks that were related to tasks they were just working on and the last-in-first-out policy eventually exhausts itself. In these cases workers often pull tasks from the common task pool. The tasks in this pool *are* ordered in a first-come-first-served basis and so workers do behave in a fair scheduling manner at a *coarse* level if not a fine grained one.

Dask's scheduling policies are short-term-efficient and long-term-fair.

Where these decisions are made

The objectives above are mostly followed by small decisions made by the client, scheduler, and workers at various points in the computation.

1. As we submit a graph from the client to the scheduler we assign a numeric priority to each task of that graph. This priority focuses on computing deeply before broadly, preferring critical paths, preferring nodes with many dependencies, etc.. This is the same logic used by the single-machine scheduler and lives in dask/order.py.
2. When the graph reaches the scheduler the scheduler changes each of these numeric priorities into a tuple of two numbers, the first of which is an increasing counter, the second of which is the client-generated priority described above. This per-graph counter encourages a first-in-first-out policy between computations. All tasks from a previous call to compute have a higher priority than all tasks from a subsequent call to compute (or submit, persist, map, or any operation that generates futures).
3. Whenever a task is ready to run the scheduler assigns it to a worker. The scheduler does not wait based on priority.
4. However when the worker receives these tasks it considers their priorities when determining which tasks to prioritize for communication or for computation. The worker maintains a heap of all ready-to-run tasks ordered by this priority.

Scheduling State

Overview

The life of a computation with Dask can be described in the following stages:

1. The user authors a graph using some library, perhaps Dask.delayed or dask.dataframe or the `submit/map` functions on the client. They submit these tasks to the scheduler.
2. The schedulers assimilates these tasks into its graph of all tasks to track and as their dependencies become available it asks workers to run each of these tasks.
3. The worker receives information about how to run the task, communicates with its peer workers to collect dependencies, and then runs the relevant function on the appropriate data. It reports back to the scheduler that it has finished.
4. The scheduler reports back to the user that the task has completed. If the user desires, it then fetches the data from the worker through the scheduler.

Most relevant logic is in tracking tasks as they evolve from newly submitted, to waiting for dependencies, to actively running on some worker, to finished in memory, to garbage collected. Tracking this process, and tracking all effects that this task has on other tasks that might depend on it, is the majority of the complexity of the dynamic task scheduler. This section describes the system used to perform this tracking.

For more abstract information about the policies used by the scheduler, see *Scheduling Policies*.

State Variables

We start with a description of the state that the scheduler keeps on each task. Each of the following is a dictionary keyed by task name (described below):

- **tasks:** `{key: task}`:

  Dictionary mapping key to a serialized task.

  A key is the name of a task, generally formed from the name of the function, followed by a hash of the function and arguments, like `'inc-ab31c010444977004d656610d2d421ec'`.

  The value of this dictionary is the task, which is an unevaluated function and arguments. This is stored in one of two forms:

  - `{'function': inc, 'args': (1,), 'kwargs': {}}`; a dictionary with the function, arguments, and keyword arguments (kwargs). However in the scheduler these are stored serialized, as they were sent from the client, so it looks more like `{'function': b'\x80\x04\x95\xcb\...', 'args': b'...', }`
  - `{'task': (inc, 1)}`: a tuple satisfying the dask graph protocol. This again is stored serialized.

  These are the values that will eventually be sent to a worker when the task is ready to run.

- **dependencies and dependents:** `{key: {keys}}`:

  These are dictionaries which show which tasks depend on which others. They contain redundant information. If `dependencies[a] == {b, c}` then the task with the name of `a` depends on the results of the two tasks with the names of `b` and `c`. There will be complimentary entries in dependents such that `a in dependents[b]` and `a in dependents[c]` such as `dependents[b] == {a, d}`. Keeping the information around twice allows for constant-time access for either direction of query, so we can both look up a task's out-edges or in-edges efficiently.

- **waiting and waiting_data:** `{key: {keys}}`:

  These are dictionaries very similar to dependencies and dependents, but they only track keys that are still in play. For example `waiting` looks like `dependencies`, tracking all of the tasks that a certain task requires before it can run. However as tasks are completed and arrive in memory they are removed from their dependents sets in `waiting`, so that when a set becomes empty we know that a key is ready to run and ready to be allocated to a worker.

  The `waiting_data` dictionary on the other hand holds all of the dependents of a key that have yet to run and still require that this task stay in memory in services of tasks that may depend on it (its `dependents`). When a value set in this dictionary becomes empty its task may be garbage collected (unless some client actively desires that this task stay in memory).

- **task_state:** `{key: string}`:

  The `task_state` dictionary holds the current state of every key. Current valid states include released, waiting, no-worker, processing, memory, and erred. These states are explained further below.

- **priority:** `{key: tuple}`:

  The `priority` dictionary provides each key with a relative ranking. This ranking is generally a tuple of two parts. The first (and dominant) part corresponds to when it was submitted. Generally earlier tasks take precedence. The second part is determined by the client, and is a way to prioritize tasks within a large graph that may be important, such as if they are on the critical path, or good to run in order to release many dependencies. This is explained further in *Scheduling Policy*

A key's priority is only used to break ties, when many keys are being considered for execution. The priority does *not* determine running order, but does exert some subtle influence that does significantly shape the long term performance of the cluster.

- **processing:** `{worker: {key: cost}}`:

  Keys that are currently allocated to a worker. This is keyed by worker address and contains the expected cost in seconds of running that task.

- **rprocessing:** `{key: worker}`:

  The reverse of the `processing` dictionary. This is all keys that are currently running with the workers that is currently running them. This is redundant with `processing` and just here for faster indexed querying.

- **who_has:** `{key: {worker}}`:

  For keys that are in memory this shows on which workers they currently reside.

- **has_what:** `{worker: {key}}`:

  This is the transpose of `who_has`, showing all keys that currently reside on each worker.

- **released:** `{keys}`

  The set of keys that are known, but released from memory. These have typically run to completion and are no longer necessary.

- **unrunnable:** `{key}`

  The set `unrunnable` contains keys that are not currently able to run, probably because they have a user defined restriction (described below) that is not met by any available worker. These keys are waiting for an appropriate worker to join the network before computing.

- **host_restrictions:** `{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.

- **worker_restrictions:** `{key: {worker addresses}}`:

  A set of complete host:port worker addresses 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 workers.

- **loose_restrictions:** `{key}`:

  Set of keys for which we are allowed to violate restrictions (see above) if not valid workers are present and the task would otherwise go into the `unrunnable` set.

- **resource_restrictions:** `{key: {resource: quantity}}`:

  Resources required by a task, such as `{'GPU': 1}` or `{'memory': 1e9}`. These names must match resources specified when creating workers.

- **worker_resources:** `{worker: {str: Number}}`:

  The available resources on each worker like `{'gpu': 2, 'mem': 1e9}`. These are abstract quantities that constrain certain tasks from running at the same time.

- **used_resources:** `{worker: {str: Number}}`:

  The sum of each resource used by all tasks allocated to a particular worker.

- **exceptions and tracebacks:** `{key: Exception/Traceback}`:

Dictionaries mapping keys to remote exceptions and tracebacks. When tasks fail we store their exceptions and tracebacks (serialized from the worker) here so that users may gather the exceptions to see the error.

- **exceptions_blame:** `{key: key}`:

  If a task fails then we mark all of its dependent tasks as failed as well. This dictionary lets any failed task see which task was the origin of its failure.

- **suspicious_tasks:** `{key: int}`

  Number of times a task has been involved in a worker failure. Some tasks may cause workers to fail (such as `sys.exit(0)`). When a worker fails all of the tasks on that worker are reassigned to others. This combination of behaviors can cause a bad task to catastrophically destroy all workers on the cluster, one after another. Whenever a worker fails we mark each task currently running on that worker as suspicious. If a task is involved in three failures (or some other fixed constant) then we mark the task as failed.

- **who_wants:** `{key: {client}}`:

  When a client submits a graph to the scheduler it also specifies which output keys it desires. Those keys are tracked here where each desired key knows which clients want it. These keys will not be released from memory and, when they complete, messages will be sent to all of these clients that the task is ready.

- **wants_what:** `{client: {key}}`:

  The transpose of `who_wants`.

- **nbytes:** `{key: int}`:

  The number of bytes, as determined by `sizeof`, of the result of each finished task. This number is used for diagnostics and to help prioritize work.

Example Event and Response

Whenever an event happens, like when a client sends up more tasks, or when a worker finishes a task, the scheduler changes the state above. For example when a worker reports that a task has finished we perform actions like the following:

Task 'key' finished by 'worker':

```
task_state[key] = 'memory'

who_has[key].add(worker)
has_what[worker].add(key)

nbytes[key] = nbytes

processing[worker].remove(key)
del rprocessing[key]

if key in who_wants:
    send_done_message_to_clients(who_wants[key])

for dep in dependencies[key]:
   waiting_data[dep].remove(key)

for dep in dependents[key]:
   waiting[dep].remove(key)
```

```
for task in ready_tasks():
    worker = best_wrker(task):
    send_task_to_worker(task, worker)
```

State Transitions

The code presented in the section above is just for demonstration. In practice writing this code for every possible event is highly error prone, resulting in hard-to-track-down bugs. Instead the scheduler moves tasks between a fixed set of states, notably `'released'`, `'waiting'`, `'no-worker'`, `'processing'`, `'memory'`, `'error'`. Transitions between common pairs of states are well defined and, if no path exists between a pair, the graph of transitions can be traversed to find a valid sequence of transitions. Along with these transitions come consistent logging and optional runtime checks that are useful in testing.

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
- 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
- Forgotten (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 on other keys. 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. It also 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.

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:

```
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.

API

class `distributed.scheduler.`**`Scheduler`**(*center=None*, *loop=None*, *delete_interval=500*, *synchronize_worker_interval=60000*, *services=None*, *allowed_failures=3*, *extensions=None*, *validate=False*, *scheduler_file=None*, *security=None*, ***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 Comm objects. 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 Client starts up without connection information:

```
>>> c = Client()
>>> c.cluster.scheduler
Scheduler(...)
```

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

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, 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
- **processing: `{worker: {key: cost}}`:** Set of keys currently in execution on each worker and their expected duration
- **rprocessing: `{key: worker}`:** The worker 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
- **host_restrictions: `{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.
- **worker_restrictions: `{key: {workers}}`:** Like host_restrictions except that these include specific host:port worker names
- **loose_restrictions: `{key}`:** Set of keys for which we are allow to violate restrictions (see above) if not valid workers are present.

- **resource_restrictions: {key: {str: Number}}:** Resources required by a task, such as `{'GPU': 1}` or `{'memory': 1e9}`. These names must match resources specified when creating workers.
- **worker_resources: {worker: {str: Number}}:** The available resources on each worker like `{'gpu': 2, 'mem': 1e9}`. These are abstract quantities that constrain certain tasks from running at the same time.
- **used_resources: {worker: {str: Number}}:** The sum of each resource used by all tasks allocated to a particular worker.
- **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.
- **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
- **worker_bytes: {worker: int}:** Number of bytes in memory on each worker
- **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 Tornado IOLoop
- **comms: [Comm]:** A list of Comms 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

add_client (**args*, ***kwargs*)
Add client to network

We listen to all future messages from this Comm.

add_keys (*comm=None*, *worker=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 https://distributed.readthedocs.io/en/latest/plugins.html

add_worker(*comm=None*, *address=None*, *keys=()*, *ncores=None*, *name=None*, *resolve_address=True*, *nbytes=None*, *now=None*, *resources=None*, *host_info=None*, ***info*)
Add a new worker to the cluster

broadcast(**args*, ***kwargs*)
Broadcast message to workers, return all results

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

cleanup(**args*, ***kwargs*)
Clean up queues and coroutines, prepare to stop

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

close(**args*, ***kwargs*)
Send cleanup signal to all coroutines then wait until finished

See also:

Scheduler.cleanup

close_comms()
Close all active Comms.

close_worker(**args*, ***kwargs*)
Remove a worker from the cluster

This both removes the worker from our local state and also sends a signal to the worker to shut down. This works regardless of whether or not the worker has a nanny process restarting it

coerce_address(*addr*, *resolve=True*)
Coerce possible input addresses to canonical form. *resolve* can be disabled for testing with fake hostnames.

Handles strings, tuples, or aliases.

coerce_hostname(*host*)
Coerce the hostname of a worker.

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

feed(**args*, ***kwargs*)
Provides a data Comm 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(**args*, ***kwargs*)
Wait until all coroutines have ceased

gather(**args*, ***kwargs*)
Collect data in from workers

get_comm_cost(*key*, *worker*)
Get the estimated communication cost (in s.) to compute key on the given worker.

get_task_duration(*key*, *default=0.5*)
Get the estimated computation cost of the given key (not including any communication cost).

get_versions(*comm*)
Basic information about ourselves and our cluster

get_worker_service_addr(*worker*, *service_name*)
Get the (host, port) address of the named service on the *worker*. Returns None if the service doesn't exist.

handle_client(**args*, ***kwargs*)
Listen and respond to messages from clients

This runs once per Client Comm or Queue.

See also:

Scheduler.worker_stream The equivalent function for workers

handle_long_running(*key=None*, *worker=None*, *compute_duration=None*)
A task has seceded from the thread pool

We stop the task from being stolen in the future, and change task duration accounting as if the task has stopped.

handle_worker(**args*, ***kwargs*)
Listen to responses from a single worker

This is the main loop for scheduler-worker interaction

See also:

Scheduler.handle_client Equivalent coroutine for clients

identity(*comm=None*)
Basic information about ourselves and our cluster

rebalance(**args*, ***kwargs*)
Rebalance keys so that each worker stores roughly equal bytes

Policy

This orders the workers by what fraction of bytes of the existing keys they have. It walks down this list from most-to-least. At each worker it sends the largest results it can find and sends them to the least occupied worker until either the sender or the recipient are at the average expected load.

reevaluate_occupancy(**args*, ***kwargs*)
Periodically reassess task duration time

The expected duration of a task can change over time. Unfortunately we don't have a good constant-time way to propagate the effects of these changes out to the summaries that they affect, like the total expected runtime of each of the workers, or what tasks are stealable.

In this coroutine we walk through all of the workers and re-align their estimates with the current state of tasks. We do this periodically rather than at every transition, and we only do it if the scheduler process isn't under load (using psutil.Process.cpu_percent()). This lets us avoid this fringe optimization when we have better things to think about.

remove_client(*client=None*)
Remove client from network

remove_plugin(*plugin*)
Remove external plugin from scheduler

remove_worker(*comm=None*, *address=None*, *safe=False*, *close=True*)
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(**args*, ***kwargs*)
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*, *client=None*)
Publish updates to all listening Queues and Comms

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

restart(**args*, ***kwargs*)
Restart all workers. Reset local state.

run_function(*stream*, *function*, *args=()*, *kwargs={}*)
Run a function within this process

See also:

`Client.run_on_scheduler`

scatter(**args*, ***kwargs*)
Send data out to workers

See also:

Scheduler.broadcast

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

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

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

Returns Jupyter connection info dictionary.

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

stimulus_missing_data(*cause=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': 'processing'}
```

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(*comm=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*, *resources=None*, *submitting_task=None*)
Add new computations to the internal dask graph

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

valid_workers(*key*)
Return set of currently valid worker addresses for key

If all workers are valid then this returns `True`. This checks tracks the following state:

- worker_restrictions
- host_restrictions
- resource_restrictions

worker_objective(*key*, *worker*)
Objective function to determine which worker should get the key

Minimize expected start time. If a tie then break with data storate.

workers_list(*workers*)
List of qualifying workers

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

workers_to_close(*memory_ratio=2*)

Find workers that we can close with low cost

This returns a list of workers that are good candidates to retire. These workers are idle (not running anything) and are storing relatively little data relative to their peers. If all workers are idle then we still maintain enough workers to have enough RAM to store our data, with a comfortable buffer.

This is for use with systems like `distributed.deploy.adaptive`.

Parameters **memory_factor: Number**

Amount of extra space we want to have for our stored data. Defaults two 2, or that we want to have twice as much memory as we currently have data.

Returns to_close: list of workers that are OK to close

distributed.scheduler.**decide_worker**(*dependencies*, *occupancy*, *who_has*, *valid_workers*, *loose_restrictions*, *objective*, *key*)

Decide which worker should take task

```
>>> dependencies = {'c': {'b'}, 'b': {'a'}}
>>> occupancy = {'alice:8000': 0, 'bob:8000': 0}
>>> who_has = {'a': {'alice:8000'}}
>>> nbytes = {'a': 100}
>>> ncores = {'alice:8000': 1, 'bob:8000': 1}
>>> valid_workers = True
>>> loose_restrictions = set()
```

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

```
>>> decide_worker(dependencies, occupancy, who_has, has_what,
...               valid_workers, loose_restrictions, nbytes, ncores, '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, who_has, has_what,
...               valid_workers, loose_restrictions, nbytes, ncores, 'b')
'bob:8000'
```

Optionally provide valid workers of where jobs are allowed to occur

```
>>> valid_workers = {'alice:8000', 'charlie:8000'}
>>> decide_worker(dependencies, who_has, has_what,
...               valid_workers, loose_restrictions, nbytes, ncores, '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}
```

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

Worker

Overview

Workers provide two functions:

1. Compute tasks as directed by the scheduler
2. Store and serve computed results to other workers or clients

Each worker contains a ThreadPool that it uses to evaluate tasks as requested by the scheduler. It stores the results of these tasks locally and serves them to other workers or clients on demand. If the worker is asked to evaluate a task for which it does not have all of the necessary data then it will reach out to its peer workers to gather the necessary dependencies.

A typical conversation between a scheduler and two workers Alice and Bob may look like the following:

```
Scheduler -> Alice:  Compute ``x <- add(1, 2)``!
Alice -> Scheduler:  I've computed x and am holding on to it!

Scheduler -> Bob:    Compute ``y <- add(x, 10)``!
                     You will need x.  Alice has x.
Bob -> Alice:        Please send me x.
Alice -> Bob:        Sure.  x is 3!
Bob -> Scheduler:    I've computed y and am holding on to it!
```

Storing Data

Data is stored locally in a dictionary in the `.data` attribute that maps keys to the results of function calls.

```
>>> worker.data
{'x': 3,
 'y': 13,
 ...
 '(df, 0)': pd.DataFrame(...),
 ...
 }
```

This `.data` attribute is a `MutableMapping` that is typically a combination of in-memory and on-disk storage with an LRU policy to move data between them.

Spill Excess Data to Disk

Short version: To enable workers to spill excess data to disk start `dask-worker` with the `--memory-limit` option. Either giving `auto` to have it guess how many bytes to keep in memory or an integer, if you know the number of bytes it should use:

```
$ dask-worker scheduler:port --memory-limit=auto  # 75% of available RAM
$ dask-worker scheduler:port --memory-limit=2e9  # two gigabytes
```

Some workloads may produce more data at one time than there is available RAM on the cluster. In these cases Workers may choose to write excess values to disk. This causes some performance degradation because writing to and reading from disk is generally slower than accessing memory, but is better than running out of memory entirely, which can cause the system to halt.

If the `dask-worker --memory-limit=NBYTES` keyword is set during initialization then the worker will store at most NBYTES of data (as measured with `sizeof`) in memory. After that it will start storing least recently used (LRU) data in a temporary directory. Workers serialize data for writing to disk with the same system used to write data on the wire, a combination of `pickle` and the default compressor.

Now whenever new data comes in it will push out old data until at most NBYTES of data is in RAM. If an old value is requested it will be read from disk, possibly pushing other values down.

It is still possible to run out of RAM on a worker. Here are a few possible issues:

1. The objects being stored take up more RAM than is stated with the __sizeof__ protocol. If you use custom classes then we encourage adding a faithful `__sizeof__` method to your class that returns an accurate accounting of the bytes used.
2. Computations and communications may take up additional RAM not accounted for. It is wise to have a suitable buffer of memory that can handle your most expensive function RAM-wise running as many times as there are active threads on the machine.
3. It is possible to misjudge the amount of RAM on the machine. Using the `--memory-limit=auto` heuristic sets the value to 75% of the return value of `psutil.virtual_memory().total`.

Thread Pool

Each worker sends computations to a thread in a concurrent.futures.ThreadPoolExecutor for computation. These computations occur in the same process as the Worker communication server so that they can access and share data efficiently between each other. For the purposes of data locality all threads within a worker are considered the same worker.

If your computations are mostly numeric in nature (for example NumPy and Pandas computations) and release the GIL entirely then it is advisable to run `dask-worker` processes with many threads and one process. This reduces communication costs and generally simplifies deployment.

If your computations are mostly Python code and don't release the GIL then it is advisable to run `dask-worker` processes with many processes and one thread per core:

```
$ dask-worker scheduler:8786 --nprocs 8
```

If your computations are external to Python and long-running and don't release the GIL then beware that while the computation is running the worker process will not be able to communicate to other workers or to the scheduler. This situation should be avoided. If you don't link in your own custom C/Fortran code then this topic probably doesn't apply to you.

Command Line tool

Use the `dask-worker` command line tool to start an individual worker. Here are the available options:

```
$ dask-worker --help
Usage: dask-worker [OPTIONS] SCHEDULER

Options:
  --worker-port INTEGER  Serving worker port, defaults to randomly assigned
  --http-port INTEGER    Serving http port, defaults to randomly assigned
  --nanny-port INTEGER   Serving nanny port, defaults to randomly assigned
  --port INTEGER         Deprecated, see --nanny-port
  --host TEXT            Serving host. Defaults to an ip address that can
                         hopefully be visible from the scheduler network.
  --nthreads INTEGER     Number of threads per process. Defaults to number of
```

```
                        cores
  --nprocs INTEGER      Number of worker processes.  Defaults to one.
  --name TEXT           Alias
  --memory-limit TEXT    Number of bytes before spilling data to disk
  --no-nanny
  --help                Show this message and exit.
```

Internal Scheduling

Internally tasks that come to the scheduler proceed through the following pipeline:

The worker also tracks data dependencies that are required to run the tasks above. These follow through a simpler pipeline:

As tasks arrive they are prioritized and put into a heap. They are then taken from this heap in turn to have any remote dependencies collected. For each dependency we select a worker at random that has that data and collect the dependency from that worker. To improve bandwidth we opportunistically gather other dependencies of other tasks that are known to be on that worker, up to a maximum of 200MB of data (too little data and bandwidth suffers, too much data and responsiveness suffers). We use a fixed number of connections (around 10-50) so as to avoid overly-fragmenting our network bandwidth. After all dependencies for a task are in memory we transition the task to the ready state and put the task again into a heap of tasks that are ready to run.

We collect from this heap and put the task into a thread from a local thread pool to execute.

Optionally, this task may identify itself as a long-running task (see *Tasks launching tasks*), at which point it secedes from the thread pool.

A task either errs or its result is put into memory. In either case a response is sent back to the scheduler.

API Documentation

class distributed.worker.**Worker**(**args*, ***kwargs*)

Worker node in a Dask distributed cluster

Workers perform two functions:

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

Workers keep the 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 `dask-worker` command line application:

```
$ dask-worker scheduler-ip:port
```

Use the `--help` flag to see more options

$ dask-worker –help

The rest of this docstring is about the internal state the the worker uses to manage and track internal computations.

State

Informational State

These attributes don't change significantly during execution.

- **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}`:
- **total_connections: `int`** The maximum number of concurrent connections we want to see
- **total_comm_nbytes**: `int`
- **batched_stream: `BatchedSend`** A batched stream along which we communicate to the scheduler
- **log: `[(message)]`** A structured and queryable log. See `Worker.story`

Volatile State

This attributes track the progress of tasks that this worker is trying to complete. In the descriptions below a `key` is the name of a task that we want to compute and `dep` is the name of a piece of dependent data that we want to collect from others.

- **data: `{key: object}`:** Dictionary mapping keys to actual values
- **task_state: `{key: string}`:** The state of all tasks that the scheduler has asked us to compute. Valid states include waiting, constrained, exeucuting, memory, erred
- **tasks: `{key: dict}`** The function, args, kwargs of a task. We run this when appropriate
- **dependencies: `{key: {deps}}`** The data needed by this key to run
- **dependents: `{dep: {keys}}`** The keys that use this dependency
- **data_needed: deque(keys)** The keys whose data we still lack, arranged in a deque
- **waiting_for_data: `{kep: {deps}}`** A dynamic verion of dependencies. All dependencies that we still don't have for a particular key.
- **ready: [keys]** Keys that are ready to run. Stored in a LIFO stack
- **constrained: [keys]** Keys for which we have the data to run, but are waiting on abstract resources like GPUs. Stored in a FIFO deque
- **executing: {keys}** Keys that are currently executing
- **executed_count: int** A number of tasks that this worker has run in its lifetime
- **long_running: {keys}** A set of keys of tasks that are running and have started their own long-running clients.
- **dep_state: `{dep: string}`:** The state of all dependencies required by our tasks Valid states include waiting, flight, and memory
- **who_has: `{dep: {worker}}`** Workers that we believe have this data
- **has_what: `{worker: {deps}}`** The data that we care about that we think a worker has

- **pending_data_per_worker: {worker: [dep]}** The data on each worker that we still want, prioritized as a deque
- **in_flight_tasks: {task: worker}** All dependencies that are coming to us in current peer-to-peer connections and the workers from which they are coming.
- **in_flight_workers: {worker: {task}}** The workers from which we are currently gathering data and the dependencies we expect from those connections
- **comm_bytes: int** The total number of bytes in flight
- **suspicious_deps: {dep: int}** The number of times a dependency has not been where we expected it
- **nbytes: {key: int}** The size of a particular piece of data
- **types: {key: type}** The type of a particular piece of data
- **threads: {key: int}** The ID of the thread on which the task ran
- **exceptions: {key: exception}** The exception caused by running a task if it erred
- **tracebacks: {key: traceback}** The exception caused by running a task if it erred
- **startstops: {key: [(str, float, float)]}** Log of transfer, load, and compute times for a task
- **priorities: {key: tuple}** The priority of a key given by the scheduler. Determines run order.
- **durations: {key: float}** Expected duration of a task
- **resource_restrictions: {key: {str: number}}** Abstract resources required to run a task

Parameters **scheduler_ip: str**

scheduler_port: int

ip: str, optional

ncores: int, optional

loop: tornado.ioloop.IOLoop

local_dir: str, optional

Directory where we place local resources

name: str, optional

heartbeat_interval: int

Milliseconds between heartbeats to scheduler

memory_limit: int

Number of bytes of data to keep in memory before using disk

executor: concurrent.futures.Executor

resources: dict

Resources that thiw worker has like `{'GPU': 2}`

See also:

distributed.scheduler.Scheduler, distributed.nanny.Nanny

Examples

Use the command line to start a worker:

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

$ dask-worker 127.0.0.1:8786
Start worker at:               127.0.0.1:1234
Registered with scheduler at:  127.0.0.1:8786
```

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 the 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.

Criteria for stealing

Computation to Communication Ratio

Stealing is profitable when the computation time for a task is much longer than the communication time of the task's dependencies.

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] = client.scatter([np.arange(1000000000)])
x = client.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] = client.scatter([100])
x = client.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, which is maintained as an exponentially weighted moving average.

Saturated Worker Burden

Stealing may be profitable even when the computation-time to communication-time ratio is poor. This occurs when the saturated workers have a very long backlog of tasks and there are a large number of idle workers. We determine if

it acceptable to steal a task if the last task to be run by the saturated workers would finish more quickly if stolen or if it remains on the original/victim worker.

The longer the backlog of stealable tasks, and the smaller the number of active workers we have both increase our willingness to steal. This is balanced against the compute-to-communicate cost ratio.

Replicate Popular Data

It is also good long term if stealing causes highly-sought-after data to be replicated on more workers.

Steal from the Rich

We would like to steal tasks from particularly over-burdened workers rather than workers with just a few excess tasks.

Restrictions

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.

Choosing tasks to steal

We maintain a list of sets of stealable tasks, ordered into bins by computation-to-communication time ratio. The first bin contains all tasks with a compute-to-communicate ratio greater than or equal to 8 (considered high enough to always steal), the next bin with a ratio of 4, the next bin with a ratio of 2, etc.., all the way down to a ratio of 1/256, which we will never steal.

This data structure provides a somewhat-ordered view of all stealable tasks that we can add to and remove from in constant time, rather than `log(n)` as with more traditional data structures, like a heap.

During any stage when we submit tasks to workers we check if there are both idle and saturated workers and if so we quickly run through this list of sets, selecting tasks from the best buckets first, working our way down to the buckets of less desirable stealable tasks. We stop either when there are no more stealable tasks, no more idle workers, or when the quality of the task-to-be-stolen is not high enough given the current backlog.

This approach is fast, optimizes to steal the tasks with the best computation-to-communication cost ratio (up to a factor of two) and tends to steal from the workers that have the largest backlogs, just by nature that random selection tends to draw from the largest population.

Adaptive Deployments

It is possible to grow and shrink Dask clusters based on current use. This allows you to run Dask permanently on your cluster and have it only take up resources when necessary. Dask contains the logic about when to grow and shrink but relies on external cluster managers to launch and kill `dask-worker` jobs. This page describes the policies about adaptively resizing Dask clusters based on load, how to connect these policies to a particular job scheduler, and an example implementation.

Dynamically scaling a Dask cluster up and down requires tight integration with an external cluster management system that can deploy `dask-worker` jobs throughout the cluster. Several systems are in wide use today, including common examples like SGE, SLURM, Torque, Condor, LSF, Yarn, Mesos, Marathon, Kubernetes, etc.. These systems can be quite different from each other, but all are used to manage distributed services throughout different kinds of clusters.

The large number of relevant systems, the challenges of rigorously testing each, and finite development time precludes the systematic inclusion of all solutions within the dask/distributed repository. Instead, we include a generic interface that can be extended by someone with basic understanding of their cluster management tool. We encourage these as third party modules.

Policies

We control the number of workers based on current load and memory use. The scheduler checks itself periodically to determine if more or fewer workers are needed.

If there are excess unclaimed tasks, or if the memory of the current workers is more nearing full then the scheduler tries to increase the number of workers by a fixed factor, defaulting to 2. This causes exponential growth while growth is useful.

If there are idle workers and if the memory of the current workers is nearing empty then we gracefully retire the idle workers with the least amount of data in memory. We first move these results to the surviving workers and then remove the idle workers from the cluster. This shrinks the cluster while gracefully preserving intermediate results, shrinking the cluster when excess size is not useful.

Adaptive class interface

The `distributed.deploy.Adaptive` class contains the logic about when to ask for new workers, and when to close idle ones. This class requires both a scheduler and a cluster object.

The cluster object must support two methods, `scale_up(n)`, which takes in a target number of total workers for the cluster and `scale_down(workers)`, which takes in a list of addresses to remove from the cluster. The Adaptive class will call these methods with the correct values at the correct times.

```
class MyCluster(object):
    @gen.coroutine
    def scale_up(self, n):
        """
        Bring the total count of workers up to ``n``

        This function/coroutine should bring the total number of workers up to
        the number ``n``.

        This can be implemented either as a function or as a Tornado coroutine.
        """
        raise NotImplementedError()

    @gen.coroutine
    def scale_down(self, workers):
        """
        Remove ``workers`` from the cluster

        Given a list of worker addresses this function should remove those
        workers from the cluster.  This may require tracking which jobs are
        associated to which worker address.

        This can be implemented either as a function or as a Tornado coroutine.
        """

from distributed.deploy import Adaptive

scheduler = Scheduler()
```

```
cluster = MyCluster()
adapative_cluster = Adaptive(scheduler, cluster)
scheduler.start()
```

Implementing these `scale_up` and `scale_down` functions depends strongly on the cluster management system. See *LocalCluster* for an example.

Marathon: an example

We now present an example project that implements this cluster interface backed by the Marathon cluster management tool on Mesos. Full source code and testing apparatus is available here: http://github.com/mrocklin/dask-marathon

The implementation is small. It uses the Marathon HTTP API through the marathon Python client library. We reproduce the full body of the implementation below as an example:

```
from marathon import MarathonClient, MarathonApp
from marathon.models.container import MarathonContainer

class MarathonCluster(object):
    def __init__(self, scheduler,
                 executable='dask-worker',
                 docker_image='mrocklin/dask-distributed',
                 marathon_address='http://localhost:8080',
                 name=None, **kwargs):
        self.scheduler = scheduler

        # Create Marathon App to run dask-worker
        args = [executable, scheduler.address,
                '--name', '$MESOS_TASK_ID']  # use Mesos task ID as worker name
        if 'mem' in kwargs:
            args.extend(['--memory-limit',
                         str(int(kwargs['mem'] * 0.6 * 1e6))])
        kwargs['cmd'] = ' '.join(args)
        container = MarathonContainer({'image': docker_image})

        app = MarathonApp(instances=0, container=container, **kwargs)

        # Connect and register app
        self.client = MarathonClient(marathon_address)
        self.app = self.client.create_app(name or 'dask-%s' % uuid.uuid4(), app)

    def scale_up(self, instances):
        self.marathon_client.scale_app(self.app.id, instances=instances)

    def scale_down(self, workers):
        for w in workers:
            self.marathon_client.kill_task(self.app.id,
                                           self.scheduler.worker_info[w]['name'],
                                           scale=True)
```

Asynchronous Operation

Dask.distributed can operate as a fully asynchronous framework and so interoperate with other highly concurrent applications. Internally Dask is built on top of Tornado coroutines but also has a compatibility layer for asyncio (see

below).

Basic Operation

When starting a client provide the `asynchronous=True` keyword to tell Dask that you intend to use this client within an asynchronous context.

```
client = await Client(asynchronous=True)
```

Operations that used to block now provide Tornado coroutines on which you can `await`.

Fast functions that only submit work remain fast and don't need to be awaited. This includes all functions that submit work to the cluster, like `submit`, `map`, `compute`, and `persist`.

```
future = client.submit(lambda x: x + 1, 10)
```

You can await futures directly

```
result = await future

>>> print(result)
11
```

Or you can use the normal client methods. Any operation that waited until it received information from the scheduler should now be `await`'ed.

```
result = await client.gather(future)
```

If you want to reuse the same client in asynchronous and synchronous environments you can apply the `asynchronous=True` keyword at each method call.

```
client = Client()  # normal blocking client

async def f():
    futures = client.map(func, L)
    results = await client.gather(futures, asynchronous=True)
    return results
```

AsyncIO

If you prefer to use the Asyncio event loop over the Tornado event loop you should use the `AioClient`.

```
from distributed.asyncio import AioClient
client = await AioClient()
```

All other operations remain the same:

```
future = client.submit(lambda x: x + 1, 10)
result = await future
# or
result = await client.gather(future)
```

Python 2 Compatibility

Everything here works with Python 2 if you replace `await` with `yield`. See more extensive comparison in the example below.

Example

This self-contained example starts an asynchronous client, submits a trivial job, waits on the result, and then shuts down the client. You can see implementations for Python 2 and 3 and for Asyncio and Tornado.

Python 3 with Tornado

```
from dask.distributed import Client

async def f():
    client = await Client(asynchronous=True)
    future = client.submit(lambda x: x + 1, 10)
    result = await future
    await client.close()
    return result

from tornado.ioloop import IOLoop
IOLoop().run_sync(f)
```

Python 2/3 with Tornado

```
from dask.distributed import Client
from tornado import gen

@gen.coroutine
def f():
    client = yield Client(asynchronous=True)
    future = client.submit(lambda x: x + 1, 10)
    result = yield future
    yield client.close()
    raise gen.Result(result)

from tornado.ioloop import IOLoop
IOLoop().run_sync(f)
```

Python 3 with Asyncio

```
from distributed.asyncio import AioClient

async def f():
    client = await AioClient()
    future = client.submit(lambda x: x + 1, 10)
    result = await future
    await client.close()
    return result
```

```
from asyncio import get_event_loop
get_event_loop().run_until_complete(f())
```

Use Cases

Historically this has been used in a few kinds of applications:

1. To integrate Dask into other asynchronous services (such as web backends), supplying a computational engine similar to Celery, but while still maintaining a high degree of concurrency and not blocking needlessly.
2. For computations that change or update state very rapidly, such as is common in some advanced machine learning workloads.
3. To develop the internals of Dask's distributed infrastucture, which is written entirely in this style.
4. For complex control and data structures in advanced applications.

Configuration

As with any distributed computation system, taking full advantage of Dask distributed sometimes requires configuration. Some options can be passed as *API* parameters and/or command line options to the various Dask executables. However, some options can also be entered in the Dask configuration file.

User-wide configuration

Dask accepts some configuration options in a configuration file, which by default is a `.dask/config.yaml` file located in your home directory. The file path can be overriden using the `DASK_CONFIG` environment variable.

The file is written in the YAML format, which allows for a human-readable hierarchical key-value configuration. All keys in the configuration file are optional, though Dask will create a default configuration file for you on its first launch.

Here is a synopsis of the configuration file:

```
logging:
    distributed: info
    distributed.client: warning
    bokeh: critical

# Scheduler options
bandwidth: 100000000    # 100 MB/s estimated worker-worker bandwidth
allowed-failures: 3     # number of retries before a task is considered bad
pdb-on-err: False       # enter debug mode on scheduling error
transition-log-length: 100000

# Worker options
multiprocessing-method: forkserver

# Communication options
compression: auto
tcp-timeout: 30         # seconds delay before calling an unresponsive connection dead
default-scheme: tcp
require-encryption: False   # whether to require encryption on non-local comms
tls:
    ca-file: myca.pem
```

```
    scheduler:
        cert: mycert.pem
        key: mykey.pem
    worker:
        cert: mycert.pem
        key: mykey.pem
    client:
        cert: mycert.pem
        key: mykey.pem
    #ciphers:
        #ECDHE-ECDSA-AES128-GCM-SHA256

# Bokeh web dashboard
bokeh-export-tool: False
```

We will review some of those options hereafter.

Communication options

compression

This key configures the desired compression scheme when transferring data over the network. The default value, "auto", applies heuristics to try and select the best compression scheme for each piece of data.

default-scheme

The *communication* scheme used by default. You can override the default ("tcp") here, but it is recommended to use explicit URIs for the various endpoints instead (for example `tls://` if you want to enable *TLS* communications).

require-encryption

Whether to require that all non-local communications be encrypted. If true, then Dask will refuse establishing any clear-text communications (for example over TCP without TLS), forcing you to use a secure transport such as *TLS*.

tcp-timeout

The default "timeout" on TCP sockets. If a remote endpoint is unresponsive (at the TCP layer, not at the distributed layer) for at least the specified number of seconds, the communication is considered closed. This helps detect endpoints that have been killed or have disconnected abruptly.

tls

This key configures *TLS* communications. Several sub-keys are recognized:

- `ca-file` configures the CA certificate file used to authenticate and authorize all endpoints.
- `ciphers` restricts allowed ciphers on TLS communications.

Each kind of endpoint has a dedicated endpoint sub-key: `scheduler`, `worker` and `client`. Each endpoint sub-key also supports several sub-keys:

- `cert` configures the certificate file for the endpoint.

- `key` configures the private key file for the endpoint.

Scheduler options

`allowed-failures`

The number of retries before a "suspicious" task is considered bad. A task is considered "suspicious" if the worker died while executing it.

`bandwidth`

The estimated network bandwidth, in bytes per second, from worker to worker. This value is used to estimate the time it takes to ship data from one node to another, and balance tasks and data accordingly.

Misc options

`logging`

This key configures the logging settings. There are two possible formats. The simple, recommended format configures the desired verbosity level for each logger. It also sets default values for several loggers such as `distributed` unless explicitly configured.

A more extended format is possible following the `logging` module's Configuration dictionary schema. To enable this extended format, there must be a `version` sub-key as mandated by the schema. The extended format does not set any default values.

Note: Python's `logging` module uses a hierarchical logger tree. For example, configuring the logging level for the `distributed` logger will also affect its children such as `distributed.scheduler`, unless explicitly overriden.

`logging-file-config`

As an alternative to the two logging settings formats discussed above, you can specify a logging config file. Its format adheres to the `logging` module's Configuration file format.

Note: The configuration options *logging-file-config* and *logging* are mutually exclusive.

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.

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.
```

Connect

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

```
dask-ec2 ssh 0      # SSH into head node
ipython             # Start IPython console on head node
```

```
>>> from distributed import Client
>>> c = Client('127.0.0.1:8786')
```

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

Destroy

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

```
dask-ec2 destroy
```

Local Cluster

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

```
>>> from distributed import Client, LocalCluster
>>> cluster = LocalCluster()
LocalCluster("127.0.0.1:8786", workers=8, ncores=8)
>>> client = Client(cluster)
<Client: scheduler=127.0.0.1:8786 processes=8 cores=8>
```

You can dynamically scale this cluster up and down:

```
>>> worker = cluster.add_worker()
>>> cluster.remove_worker(worker)
```

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

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

API

class distributed.deploy.local.**LocalCluster**(*n_workers=None*, *threads_per_worker=None*, *processes=True*, *loop=None*, *start=True*, *ip=None*, *scheduler_port=0*, *silence_logs=50*, *diagnostics_port=8787*, *services={}*, *worker_services={}*, *nanny=None*, ***worker_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

processes: bool

Whether to use processes (True) or threads (False). Defaults to True

threads_per_worker: int

Number of threads per each worker

scheduler_port: int

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

silence_logs: logging level

Level of logs to print out to stdout. `logging.CRITICAL` by default. Use a falsey value like False or None for no change.

ip: string

IP address on which the scheduler will listen, defaults to only localhost

kwargs: dict

Extra worker arguments, will be passed to the Worker constructor.

Examples

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

```
>>> c = Client(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

close()
Close the cluster

scale_down(**args*, ***kwargs*)
Remove `workers` from the cluster

Given a list of worker addresses this function should remove those workers from the cluster. This may require tracking which jobs are associated to which worker address.

This can be implemented either as a function or as a Tornado coroutine.

scale_up(**args*, ***kwargs*)
Bring the total count of workers up to `n`

This function/coroutine should bring the total number of workers up to the number `n`.

This can be implemented either as a function or as a Tornado coroutine.

start_worker(*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

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)
```

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

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_dask method:

```
$ ipcluster start
```

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

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

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:

`Client.start_ipython_workers`([workers, ...])	Start IPython kernels on workers
`Client.start_ipython_scheduler`([magic_name, ...])	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)
```

Joblib Integration

Dask.distributed integrates with Joblib by providing an alternative cluster-computing backend, alongside Joblib's builtin threading and multiprocessing backends.

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 `dask.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:

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

with parallel_backend('dask.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.

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

For large arguments that are used by multiple tasks, it may be more efficient to pre-scatter the data to every worker, rather than serializing it once for every task. This can be done using the `scatter` keyword argument, which takes an iterable of objects to send to each worker.

```
# Serialize the training data only once to each worker
with parallel_backend('dask.distributed', scheduler_host='localhost:8786',
                      scatter=[digits.data, digits.target]):
    search.fit(digits.data, digits.target)
```

Publish Datasets

A *published 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.

Publishing datasets is useful in the following cases:

- You want to share computations with colleagues
- You want to persist results on the cluster between interactive sessions

Motivating Example

In this example we load a dask.dataframe from S3, manipulate it, and then publish the result.

Connect and Load

```
from dask.distributed import Client
client = Client('scheduler-address:8786')

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

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

Publish

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

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

Load published dataset from different client

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

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

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

>>> df = client.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.

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.

API

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

Data Streams with Queues

The `Client` 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.

Example

We connect to a local Client.

```
>>> from distributed import Client
>>> client = Client('127.0.0.1:8786')
>>> client
<Client: 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 = client.scatter(input_q)
>>> inc_q = client.map(inc, remote_q)
>>> double_q = client.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 = client.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
...
```

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..

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.

Worker Resources

Access to scarce resources like memory, GPUs, or special hardware may constrain how many of certain tasks can run on particular machines.

For example, we may have a cluster with ten computers, four of which have two GPUs each. We may have a thousand tasks, a hundred of which require a GPU and ten of which require two GPUs at once. In this case we want to balance tasks across the cluster with these resource constraints in mind, allocating GPU-constrained tasks to GPU-enabled workers. Additionally we need to be sure to constrain the number of GPU tasks that run concurrently on any given worker to ensure that we respect the provided limits.

This situation arises not only for GPUs but for many resources like tasks that require a large amount of memory at runtime, special disk access, or access to special hardware. Dask allows you to specify abstract arbitrary resources to constrain how your tasks run on your workers. Dask does not model these resources in any particular way (Dask does not know what a GPU is) and it is up to the user to specify resource availability on workers and resource demands on tasks.

Example

We consider a computation where we load data from many files, process each one with a function that requires a GPU, and then aggregate all of the intermediate results with a task that takes up 70GB of memory.

We operate on a three-node cluster that has two machines with two GPUs each and one machine with 100GB of RAM.

When we set up our cluster we define resources per worker:

```
dask-worker scheduler:8786 --resources "GPU=2"
dask-worker scheduler:8786 --resources "GPU=2"
dask-worker scheduler:8786 --resources "MEMORY=100e9"
```

When we submit tasks to the cluster we specify constraints per task

```
from distributed import Client
client = Client('scheduler:8786')

data = [client.submit(load, fn) for fn in filenames]
processed = [client.submit(process, d, resources={'GPU': 1}) for d in data]
final = client.submit(aggregate, processed, resources={'MEMORY': 70e9})
```

Resources are Abstract

Resources listed in this way are just abstract quantities. We could equally well have used terms "mem", "memory", "bytes" etc. above because, from Dask's perspective, this is just an abstract term. You can choose any term as long as you are consistent across workers and clients.

It's worth noting that Dask separately track number of cores and available memory as actual resources and uses these in normal scheduling operation.

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.

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`:

```
from distributed import Client

def inc(x):
    return x + 1

if __name__=='__main__':
    client = Client('127.0.0.1:8786')
    x = client.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
```

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.

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

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.

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.

Submit tasks from worker

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

We can submit tasks from other tasks. This allows us to make decisions while on worker nodes.

To submit new tasks from a worker that worker must first create a new client object that connects to the scheduler. There is a convenience function to do this for you so that you don't have to pass around connection information. However you must use this function `worker_client` as a context manager to ensure proper cleanup on the worker.

```
from distributed import worker_client

def process_all(data):
    with worker_client() 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 somewhat 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..

```
In [1]: from distributed import Client, worker_client

In [2]: client = Client()

In [3]: def fib(n):
   ...:     if n < 2:
   ...:         return n
```

```
   ...:     else:
   ...:         with worker_client() as c
   ...:             a = c.submit(fib, n - 1)
   ...:             b = c.submit(fib, n - 2)
   ...:             a, b = c.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
```

This example is a bit extreme and spends most of its time establishing client connections from the worker rather than doing actual work, but does demonstrate that even pathological cases function robustly.

Technical details

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

1. The thread on the worker running this function *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 additional 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.

Establishing a connection to the scheduler takes on the order of 10-20 ms and so it is wise for computations that use this feature to be at least a few times longer in duration than this.

TLS/SSL

Currently dask distributed has experimental support for TLS/SSL communication, providing mutual authentication and encryption of communications between cluster endpoints (Clients, Schedulers and Workers).

TLS is enabled by using a `tls` address such as `tls://` (the default being `tcp`, which sends data unauthenticated and unencrypted). In TLS mode, all cluster endpoints must present a valid TLS certificate signed by a given Certificate Authority (CA). It is generally recommended to use a custom CA for your organization, as it will allow signing certificates for internal hostnames or IP addresses.

Parameters

When using TLS, one has to provide additional parameters:

- a *CA certificate(s) file*, which allows TLS to decide whether an endpoint's certificate has been signed by the correct authority;

- a *certificate file* for each endpoint, which is presented to other endpoints so as to achieve mutual authentication;
- a *private key file*, which is the cryptographic means to prove to other endpoints that you are the authorized user of a given certificate.

Note: As per OpenSSL's requirements, all those files should be in PEM format. Also, it is allowed to concatenate the certificate and private key into a single file (you can then just specify the *certificate* parameter and leave the *private key* parameter absent).

It is up to you whether each endpoint uses a different certificate and private key, or whether all endpoints share the same, or whether each endpoint kind (Client, Scheduler, Worker) gets its own certificate / key pair. Unless you have extraordinary requirements, however, the CA certificate should probably be the same for all endpoints.

One can also pass additional parameters:

- a set of allowed *ciphers*, if you have strong requirements as to which algorithms are considered secure; this setting's value should be an OpenSSL cipher string;
- whether to *require encryption*, to avoid using plain TCP communications by mistake.

All those parameters can be passed in several ways:

- through the Dask *configuration file*;
- if using the command line, through options to `dask-scheduler` and `dask-worker`;
- if using the API, through a `Security` object. For example, here is how you might configure a `Security` object for client use:

```
from distributed import Client
from distributed.security import Security

sec = Security(tls_ca_file='cluster_ca.pem',
               tls_client_cert='cli_cert.pem',
               tls_client_key='cli_key.pem',
               require_encryption=True)

client = Client(..., security=sec)
```

Security policy

Dask always verifies the certificate presented by a remote endpoint against the configured CA certificate(s). Certificates are verified for both "client" and "server" endpoints (in the TCP sense), ensuring the endpoints are mutually authenticated. The hostname or IP address for which a certificate has been issued is not checked; this should not be an issue if you are using your own internal Certificate Authority.

It is not possible to disable certificate verification, as it would render the communications vulnerable to Man-in-the-Middle attacks.

Performance implications

Encryption is fast on recent CPUs, most of which have hardware acceleration for AES-based encryption. AES is normally selected by the TLS layer unless you have forced the *ciphers* parameter to something else. However, encryption may still have a non-negligible overhead if you are transferring very large data over very high speed network links.

See also:

A study of AES-NI acceleration shows recent x86 CPUs can AES-encrypt more than 1 GB per second on each CPU core.

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`).

List of Servers

There are a few sets of diagnostic pages served at different ports:

- Main Scheduler pages at `http://scheduler-address:8787`. These pages, particularly the `/status` page are the main page that most people associate with Dask. These pages are served from a separate standalone Bokeh server application running in a separate process.
- Debug Scheduler pages at `http://scheduler-address:8788`. These pages have more detailed diagnostic information about the scheduler. They are more often used by developers than by users, but may still be of interest to the performance-conscious. These pages run from inside the scheduler process, and so compete for resources with the main scheduler.
- Debug Worker pages for each worker at `http://worker-address:8789`. These pages have detailed diagnostic information about the worker. Like the diagnostic scheduler pages they are of more utility to developers or to people looking to understand the performance of their underlying cluster. If port 8789 is unavailable (for example it is in use by another worker) then a random port is chosen. A list of all ports can be obtained from looking at the service ports for each worker in the result of calling `client.scheduler_info()`

The rest of this document will be about the main pages at `http://scheduler-address:8787`.

The available pages are `http://scheduler-address:8787/<page>/` where `<page>` is one of

- `status`: a stream of recently run tasks, progress bars, resource use
- `tasks`: a larger stream of the last 100k tasks
- `workers`: basic information about workers and their current load

Plots

Example Computation

The following plots show a trace of the following computation:

```
from distributed import Client
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


client = Client('127.0.0.1:8786')

incs = client.map(inc, range(100))
decs = client.map(dec, range(100))
adds = client.map(add, incs, decs)
total = client.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.

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.

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`.

Changelog

1.18.0 - July 8th, 2017

- Multi-threading safety (GH#1191), (GH#1228), (GH#1229)

- Improve handling of byte counting (GH#1198) (GH#1224)
- Add get_client, secede functions, refactor worker-client relationship (GH#1201)
- Allow logging configuraiton using logging.dictConfig() (GH#1206) (GH#1211)
- Offload serialization and deserialization to separate thread (GH#1218)
- Support fire-and-forget tasks (GH#1221)
- Support bytestrings as keys (for Julia) (GH#1234)
- Resolve testing corner-cases (GH#1236), (GH#1237), (GH#1240), (GH#1241), (GH#1242), (GH#1244)
- Automatic use of scatter/gather(direct=True) in more cases (GH#1239)

1.17.1 - June 14th, 2017

- Remove Python 3.4 testing from travis-ci (GH#1157)
- Remove ZMQ Support (GH#1160)
- Fix memoryview nbytes issue in Python 2.7 (GH#1165)
- Re-enable counters (GH#1168)
- Improve scheduler.restart (GH#1175)

1.17.0 - June 9th, 2017

- Reevaluate worker occupancy periodically during scheduler downtime (GH#1038) (GH#1101)
- Add `AioClient` asyncio-compatible client API (GH#1029) (GH#1092) (GH#1099)
- Update Keras serializer (GH#1067)
- Support TLS/SSL connections for security (GH#866) (GH#1034)
- Always create new worker directory when passed `--local-directory` (GH#1079)
- Support pre-scattering data when using joblib frontent (GH#1022)
- Make workers more robust to failure of `sizeof` function (GH#1108) and writing to disk (GH#1096)
- Add `is_empty` and `update` methods to `as_completed` (GH#1113)
- Remove `_get` coroutine and replace with `get(..., sync=False)` (GH#1109)
- Improve API compatibility with async/await syntax (GH#1115) (GH#1124)
- Add distributed Queues (GH#1117) and shared Variables (GH#1128) to enable inter-client coordination
- Support direct client-to-worker scattering and gathering (GH#1130) as well as performance enhancements when scattering data
- Style improvements for bokeh web dashboards (GH#1126) (GH#1141) as well as a removal of the external bokeh process
- HTML reprs for Future and Client objects (GH#1136)
- Support nested collections in client.compute (GH#1144)
- Use normal client API in asynchronous mode (GH#1152)

- Remove old distributed.collections submodule (GH#1153)

1.16.3 - May 5th, 2017

- Add bokeh template files to MANIFEST (GH#1063)
- Don't set worker_client.get as default get (GH#1061)
- Clean up logging on Client().shutdown() (GH#1055)

1.16.2 - May 3rd, 2017

- Support `async with Client` syntax (GH#1053)
- Use internal bokeh server for default diagnostics server (GH#1047)
- Improve styling of bokeh plots when empty (GH#1046) (GH#1037)
- Support efficient serialization for sparse arrays (GH#1040)
- Prioritize newly arrived work in worker (GH#1035)
- Prescatter data with joblib backend (GH#1022)
- Make client.restart more robust to worker failure (GH#1018)
- Support preloading a module or script in dask-worker or dask-scheduler processes (GH#1016)
- Specify network interface in command line interface (GH#1007)
- Client.scatter supports a single element (GH#1003)
- Use blosc compression on all memoryviews passing through comms (GH#998)
- Add concurrent.futures-compatible Executor (GH#997)
- Add as_completed.batches method and return results (GH#994) (GH#971)
- Allow worker_clients to optionally stay within the thread pool (GH#993)
- Add bytes-stored and tasks-processing diagnostic histograms (GH#990)
- Run supports non-msgpack-serializable results (GH#965)

1.16.1 - March 22nd, 2017

- Use inproc transport in LocalCluster (GH#919)
- Add structured and queryable cluster event logs (GH#922)
- Use connection pool for inter-worker communication (GH#935)
- Robustly shut down spawned worker processes at shutdown (GH#928)
- Worker death timeout (GH#940)
- More visual reporting of exceptions in progressbar (GH#941)
- Render disk and serialization events to task stream visual (GH#943)
- Support async for / await protocol (GH#952)
- Ensure random generators are re-seeded in worker processes (GH#953)

- Upload sourcecode as zip module (GH#886)
- Replay remote exceptions in local process (GH#894)

1.16.0 - February 24th, 2017

- First come first served priorities on client submissions (GH#840)
- Can specify Bokeh internal ports (GH#850)
- Allow stolen tasks to return from either worker (GH#853), (GH#875)
- Add worker resource constraints during execution (GH#857)
- Send small data through Channels (GH#858)
- Better estimates for SciPy sparse matrix memory costs (GH#863)
- Avoid stealing long running tasks (GH#873)
- Maintain fortran ordering of NumPy arrays (GH#876)
- Add `--scheduler-file` keyword to dask-scheduler (GH#877)
- Add serializer for Keras models (GH#878)
- Support uploading modules from zip files (GH#886)
- Improve titles of Bokeh dashboards (GH#895)

1.15.2 - January 27th, 2017

- Fix a bug where arrays with large dtypes or shapes were being improperly compressed (GH#830 GH#832 GH#833)
- Extend `as_completed` to accept new futures during iteration (GH#829)
- Add `--nohost` keyword to `dask-ssh` startup utility (GH#827)
- Support scheduler shutdown of remote workers, useful for adaptive clusters (:pr: *811* GH#816 GH#821)
- Add `Client.run_on_scheduler` method for running debug functions on the scheduler (GH#808)

1.15.1 - January 11th, 2017

- Make compatibile with Bokeh 0.12.4 (GH#803)
- Avoid compressing arrays if not helpful (GH#777)
- Optimize inter-worker data transfer (GH#770) (GH#790)
- Add –local-directory keyword to worker (GH#788)
- Enable workers to arrive to the cluster with their own data. Useful if a worker leaves and comes back (GH#785)
- Resolve thread safety bug when using local_client (GH#802)
- Resolve scheduling issues in worker (GH#804)

1.15.0 - January 2nd, 2017

- Major Worker refactor (GH#704)
- Major Scheduler refactor (GH#717) (GH#722) (GH#724) (GH#742) (GH#743
- Add `check` (default is `False`) option to `Client.get_versions` to raise if the versions don't match on client, scheduler & workers (GH#664)
- `Future.add_done_callback` executes in separate thread (GH#656)
- Clean up numpy serialization (GH#670)
- Support serialization of Tornado v4.5 coroutines (GH#673)
- Use CPickle instead of Pickle in Python 2 (GH#684)
- Use Forkserver rather than Fork on Unix in Python 3 (GH#687)
- Support abstract resources for per-task constraints (GH#694) (GH#720) (GH#737)
- Add TCP timeouts (GH#697)
- Add embedded Bokeh server to workers (GH#709) (GH#713) (GH#738)
- Add embedded Bokeh server to scheduler (GH#724) (GH#736) (GH#738)
- Add more precise timers for Windows (GH#713)
- Add Versioneer (GH#715)
- Support inter-client channels (GH#729) (GH#749)
- Scheduler Performance improvements (GH#740) (GH#760)
- Improve load balancing and work stealing (GH#747) (GH#754) (GH#757)
- Run Tornado coroutines on workers
- Avoid slow sizeof call on Pandas dataframes (GH#758)

1.14.3 - November 13th, 2016

- Remove custom Bokeh export tool that implicitly relied on nodejs (GH#655)
- Clean up scheduler logging (GH#657)

1.14.2 - November 11th, 2016

- Support more numpy dtypes in custom serialization, (GH#627), (GH#630), (GH#636)
- Update Bokeh plots (GH#628)
- Improve spill to disk heuristics (GH#633)
- Add Export tool to Task Stream plot
- Reverse frame order in loads for very many frames (GH#651)
- Add timeout when waiting on write (GH#653)

1.14.0 - November 3rd, 2016

- Add `Client.get_versions()` function to return software and package information from the scheduler, workers, and client (GH#595)
- Improved windows support (GH#577) (GH#590) (GH#583) (GH#597)
- Clean up rpc objects explicitly (GH#584)
- Normalize collections against known futures (GH#587)
- Add key= keyword to map to specify keynames (GH#589)
- Custom data serialization (GH#606)
- Refactor the web interface (GH#608) (GH#615) (GH#621)
- Allow user-supplied Executor in Worker (GH#609)
- Pass Worker kwargs through LocalCluster

1.13.3 - October 15th, 2016

- Schedulers can retire workers cleanly
- Add `Future.add_done_callback` for `concurrent.futures` compatibility
- Update web interface to be consistent with Bokeh 0.12.3
- Close streams explicitly, avoiding race conditions and supporting more robust restarts on Windows.
- Improved shuffled performance for dask.dataframe
- Add adaptive allocation cluster manager
- Reduce administrative overhead when dealing with many workers
- `dask-ssh --log-directory .` no longer errors
- Microperformance tuning for the scheduler

1.13.2

- Revert dask_worker to use fork rather than subprocess by default
- Scatter retains type information
- Bokeh always uses subprocess rather than spawn

1.13.1

- Fix critical Windows error with dask_worker executable

1.13.0

- Rename Executor to Client (GH#492)
- Add `--memory-limit` option to `dask-worker`, enabling spill-to-disk behavior when running out of memory (GH#485)

- Add `--pid-file` option to dask-worker and `--dask-scheduler` (GH#496)
- Add `upload_environment` function to distribute conda environments. This is experimental, undocumented, and may change without notice. (GH#494)
- Add `workers=` keyword argument to `Client.compute` and `Client.persist`, supporting location-restricted workloads with Dask collections (GH#484)
- Add `upload_environment` function to distribute conda environments. This is experimental, undocumented, and may change without notice. (GH#494)
  - Add optional `dask_worker=` keyword to `client.run` functions that gets provided the worker or nanny object
  - Add `nanny=False` keyword to `Client.run`, allowing for the execution of arbitrary functions on the nannies as well as normal workers

1.12.2

This release adds some new features and removes dead code

- Publish and share datasets on the scheduler between many clients (GH#453). See *Publish Datasets*.
- Launch tasks from other tasks (experimental) (GH#471). See *Launch Tasks from Tasks*.
- Remove unused code, notably the `Center` object and older client functions (GH#478)
- `Executor()` and `LocalCluster()` is now robust to Bokeh's absence (GH#481)
- Removed s3fs and boto3 from requirements. These have moved to Dask.

1.12.1

This release is largely a bugfix release, recovering from the previous large refactor.

- **Fixes from previous refactor**
  - Ensure idempotence across clients
  - Stress test losing scattered data permanently
- **IPython fixes**
  - Add `start_ipython_scheduler` method to Executor
  - Add `%remote` magic for workers
  - Clean up code and tests
- Pool connects to maintain reuse and reduce number of open file handles
- Re-implement work stealing algorithm
- Support cancellation of tuple keys, such as occur in dask.arrays
- Start synchronizing against worker data that may be superfluous
- **Improve bokeh plots styling**
  - Add memory plot tracking number of bytes
  - Make the progress bars more compact and align colors
  - Add workers/ page with workers table, stacks/processing plot, and memory

- Add this release notes document

1.12.0

This release was largely a refactoring release. Internals were changed significantly without many new features.

- Major refactor of the scheduler to use transitions system
- Tweak protocol to traverse down complex messages in search of large bytestrings
- Add dask-submit and dask-remote
- Refactor HDFS writing to align with changes in the dask library
- Executor reconnects to scheduler on broken connection or failed scheduler
- Support sklearn.external.joblib as well as normal joblib

Communications

Workers, the Scheduler, and Clients communicate by sending each other Python objects (such as *Protocol* messages or user data). The communication layer handles appropriate encoding and shipping of those Python objects between the distributed endpoints. The communication layer is able to select between different transport implementations, depending on user choice or (possibly) internal optimizations.

The communication layer lives in the `distributed.comm` package.

Addresses

Communication addresses are canonically represented as URIs, such as `tcp://127.0.0.1:1234`. For compatibility with existing code, if the URI scheme is omitted, a default scheme of `tcp` is assumed (so `127.0.0.1:456` is really the same as `tcp://127.0.0.1:456`). The default scheme may change in the future.

The following schemes are currently implemented in the `distributed` source tree:

- `tcp` is the main transport; it uses TCP sockets and allows for IPv4 and IPv6 addresses.
- `tls` is a secure transport using the well-known TLS protocol over TCP sockets. Using it requires specifying keys and certificates as outlined in *TLS/SSL*.
- `inproc` is an in-process transport using simple object queues; it eliminates serialization and I/O overhead, providing almost zero-cost communication between endpoints as long as they are situated in the same process.

Some URIs may be valid for listening but not for connecting. For example, the URI `tcp://` will listen on all IPv4 and IPv6 addresses and on an arbitrary port, but you cannot connect to that address.

Higher-level APIs in `distributed` may accept other address formats for convenience or compatibility, for example a `(host, port)` pair. However, the abstract communications layer always deals with URIs.

Functions

There are a number of top-level functions in `distributed.comm` to help deal with addresses:

`distributed.comm.`**`parse_address`**(*addr*)
Split address into its scheme and scheme-dependent location string.

```
>>> parse_address('tcp://127.0.0.1')
('tcp', '127.0.0.1')
```

distributed.comm.**unparse_address**(*scheme*, *loc*)
Undo parse_address().

```
>>> unparse_address('tcp', '127.0.0.1')
'tcp://127.0.0.1'
```

distributed.comm.**normalize_address**(*addr*)
Canonicalize address, adding a default scheme if necessary.

```
>>> normalize_address('tls://[::1]')
'tls://[::1]'
>>> normalize_address('[::1]')
'tcp://[::1]'
```

distributed.comm.**resolve_address**(*addr*)
Apply scheme-specific address resolution to *addr*, replacing all symbolic references with concrete location specifiers.

In practice, this can mean hostnames are resolved to IP addresses.

```
>>> resolve_address('tcp://localhost:8786')
'tcp://127.0.0.1:8786'
```

distributed.comm.**get_address_host**(*addr*)
Return a hostname / IP address identifying the machine this address is located on.

In contrast to get_address_host_port(), this function should always succeed for well-formed addresses.

```
>>> get_address_host('tcp://1.2.3.4:80')
'1.2.3.4'
```

Communications API

The basic unit for dealing with established communications is the Comm object:

class distributed.comm.**Comm**
A message-oriented communication object, representing an established communication channel. There should be only one reader and one writer at a time: to manage current communications, even with a single peer, you must create distinct Comm objects.

Messages are arbitrary Python objects. Concrete implementations of this class can implement different serialization mechanisms depending on the underlying transport's characteristics.

abort()
Close the communication immediately and abruptly. Useful in destructors or generators' finally blocks.

close()
Close the communication cleanly. This will attempt to flush outgoing buffers before actually closing the underlying transport.

This method is a coroutine.

closed()
Return whether the stream is closed.

extra_info
Return backend-specific information about the communication, as a dict. Typically, this is information which is initialized when the communication is established and doesn't vary afterwards.

local_address
The local address. For logging and debugging purposes only.

peer_address
The peer's address. For logging and debugging purposes only.

read()
Read and return a message (a Python object).

This method is a coroutine.

write(*msg*)
Write a message (a Python object).

This method is a coroutine.

You don't create `Comm` objects directly: you either `listen` for incoming communications, or `connect` to a peer listening for connections:

distributed.comm.**connect**(**args*, ***kwargs*)
Connect to the given address (a URI such as `tcp://127.0.0.1:1234`) and yield a `Comm` object. If the connection attempt fails, it is retried until the *timeout* is expired.

distributed.comm.**listen**(*addr*, *handle_comm*, *deserialize=True*, *connection_args=None*)
Create a listener object with the given parameters. When its `start()` method is called, the listener will listen on the given address (a URI such as `tcp://0.0.0.0`) and call *handle_comm* with a `Comm` object for each incoming connection.

handle_comm can be a regular function or a coroutine.

Listener objects expose the following interface:

class distributed.comm.core.**Listener**

contact_address
An address this listener can be contacted on. This can be different from *listen_address* if the latter is some wildcard address such as 'tcp://0.0.0.0:123'.

listen_address
The listening address as a URI string.

start()
Start listening for incoming connections.

stop()
Stop listening. This does not shutdown already established communications, but prevents accepting new ones.

Extending the Communication Layer

Each transport is represented by a URI scheme (such as `tcp`) and backed by a dedicated `Backend` implementation, which provides entry points into all transport-specific routines.

class distributed.comm.registry.**Backend**
A communication backend, selected by a given URI scheme (e.g. 'tcp').

get_address_host(*loc*)
Get a host name (normally an IP address) identifying the host the address is located on. *loc* is a scheme-less address.

get_address_host_port(*loc*)
Get the (host, port) tuple of the scheme-less address *loc*. This should only be implemented by IP-based transports.

get_connector()
Get a connector object usable for connecting to addresses.

get_listener(*loc*, *handle_comm*, *deserialize*, ***connection_args*)
Get a listener object for the scheme-less address *loc*.

get_local_address_for(*loc*)
Get the local listening address suitable for reaching *loc*.

resolve_address(*loc*)
Resolve the address into a canonical form. *loc* is a scheme-less address.

Simple implementations may return *loc* unchanged.

Development Guidelines

This repository is part of the Dask projects. General development guidelines including where to ask for help, a layout of repositories, testing practices, and documentation and style standards are available at the Dask developer guidelines in the main documentation.

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
```

Test

Test using `py.test`:

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

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 documentation 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.

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(client=True)
def test_submit(c, s, a, b):
    assert isinstance(c, Client)
    assert isinstance(c, Scheduler)
    assert isinstance(a, Worker)
    assert isinstance(b, Worker)

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

    # result = future.result()  # This synchronous API call would block
    result = yield future
    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, client, 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 Client(('127.0.0.1', s['port']), loop=loop) as c:
            future = c.submit(inc, 1)
            assert future.key in c.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.

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 `dask.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 environment. However, as with all layered abstractions, ours has flaws. Critical feedback is welcome.

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

Endpoint-to-endpoint Communication

The various distributed endpoints (Client, Scheduler, Worker) communicate by sending each other arbitrary Python objects. Encoding, sending and then decoding those objects is the job of the *communication layer*.

Ancillary services such as a Bokeh-based Web interface, however, have their own implementation and semantics.

Protocol Handling

While the abstract communication layer can transfer arbitrary Python objects (as long as they are serializable), participants in a `distributed` cluster concretely obey the distributed *Protocol*, which specifies request-response semantics using a well-defined message format.

Dedicated infrastructure in `distributed` handles the various aspects of the protocol, such as dispatching the various operations supported by an endpoint.

Servers

Worker, Scheduler, and Nanny objects all inherit from a `Server` class.

class `distributed.core.`**`Server`**(*handlers*, *connection_limit=512*, *deserialize=True*, *io_loop=None*)
Distributed TCP Server

Superclass for endpoints in a distributed cluster, such as Worker and Scheduler objects.

Handlers

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

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

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

```
>>> handlers = {'ping': pingpong, 'add': add}
>>> server = Server(handlers)
>>> server.listen('tcp://0.0.0.0:8000')
```

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}`

RPC

To interact with remote servers we typically use `rpc` objects which expose a familiar method call interface to invoke remote operations.

class `distributed.core.`**`rpc`**(*arg=None*, *comm=None*, *deserialize=True*, *timeout=None*, *connection_args=None*)
Conveniently interact with a remote server

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

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

When done, close comms explicitly.

```
>>> remote.close_comms()
```

Examples

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 Server

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

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

s = Server({'add': add, 'stream_data': stream_data})
s.listen('tcp://:8888')   # listen on TCP port 8888

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

Client Side

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

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

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

@gen.coroutine
def g():
    comm = yield connect('tcp://127.0.0.1:8888')
    yield comm.write({'op': 'stream_data', 'interval': 1})
```

```
    while True:
        result = yield comm.read()
        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():
    # comm = yield connect('tcp://127.0.0.1', 8888)
    # yield comm.write({'op': 'add', 'x': 1, 'y': 2})
    # result = yield comm.read()
    r = rpc('tcp://127.0.0.1:8888')
    result = yield r.add(x=1, y=2)
    r.close_comms()

    print(result)

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

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.

User code

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

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

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

print(z.result())
```

Step 1: Client

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

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

The client 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'`.

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.

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.

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.

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`.

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.

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.)

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 client (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.

Overhead

The user experiences this in about 10 milliseconds, depending on network latency.

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.

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 and complex datatypes are handled 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.

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.

CloudPickle for Functions and Some 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.

Note: we actually call some combination of pickle and cloudpickle, depending on the situation. This is for performance reasons.

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

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.

Header

The header is a small dictionary encoded in msgpack that includes some metadata about the message, such as compression.

Serializing Data

For administrative messages like updating status msgpack is sufficient. However for large results or Python specific data, like NumPy arrays or Pandas Dataframes, or for larger results we need to use something else to convert Python objects to bytestrings. Exactly how we do this is described more in the *Serialization documentation*.

The application code marks Python specific results with the `to_serialize` function:

```
>>> import numpy as np
>>> x = np.ones(5)

>>> from distributed.protocol import to_serialize
>>> msg = {'status': 'OK', 'data': to_serialize(x)}
>>> msg
{'data': <Serialize: [ 1.  1.  1.  1.  1.]>, 'status': 'OK'}
```

We separate the message into two messages, one encoding all of the data to be serialized and, and one encoding everything else:

```
{'key': 'x', 'address': 'alice'}
{'data': <Serialize: [ 1.  1.  1.  1.  1.]>}
```

The first message we pass normally with msgpack. The second we pass in multiple parts, one part for each serialized piece of data (see *serialization*) and one header including types, compression, etc. used for each value:

```
{'keys': ['data'],
 'compression': ['lz4']}
b'...'
b'...'
```

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.

Technical Version

A message is broken up into the following components:

1. 8 bytes encoding how many frames there are in the message (N) as a `uint64`
2. 8 * N frames encoding the length of each frame as `uint64` s
3. Header for the administrative message
4. The administrative message, msgpack encoded, possibly compressed
5. Header for all payload messages
6. Payload messages

Header for Administrative Message

The administrative message is arbitrary msgpack-encoded data. Usually a dictionary. It may optionally be compressed. If so the compression type will be in the header.

Payload frames and Header

These frames are optional.

Payload frames are used to send large or language-specific data. These values will be inserted into the administrative message after they are decoded. The header is msgpack encoded and contains encoding and compression information for the all subsequent payload messages.

A Payload may be spread across many frames. Each frame may be separately compressed.

Simple Example

This simple example shows a minimal message. There is only an empty header and a small msgpack message. There are no additional payload frames

Message: `{'status': 'OK'}`

Frames:

- Header: `{}`
- Administrative Message: `{'status': 'OK'}`

Example with Custom Data

This example contains a single payload message composed of a single frame. It uses a special serialization for NumPy arrays.

Message: `{'op': 'get-data', 'data': np.ones(5)}`

Frames:

- Header: `{}`
- Administrative Message: `{'op': 'get-data'}`
- Payload header:

```
{'headers': [{'type': 'numpy.ndarray',
              'compression': 'lz4',
              'count': 1,
              'lengths': [40],
              'dtype': '<f8',
              'strides': (8,),
              'shape': (5,)}],
             'keys': [('data',)]}
```

- Payload Frame: `b'(\x00\x00\x00\x11\x00\x01\x00!\xf0?\x07\x00\x0f\x08\x00\x03P\x00\x00\x00\x
'`

Custom Serialization

When we communicate data between computers we first convert that data into a sequence of bytes that can be communicated across the network.

Dask can convert data to bytes using the standard solutions of Pickle and Cloudpickle. However, sometimes pickle and cloudpickle are suboptimal so Dask also supports custom serialization formats for special types. This helps Dask

to be faster on common formats like NumPy and Pandas and gives power-users more control about how their objects get moved around on the network if they want to extend the system.

We include a small example and then follow with the full API documentation describing the `serialize` and `deserialize` functions, which convert objects into a msgpack header and a list of bytestrings and back.

Example

Here is how we special case handling raw Python bytes objects. In this case there is no need to call `pickle.dumps` on the object. The object is already a sequnce of bytes.

```
def serialize_bytes(obj):
    header = {}  # no special metadata
    frames = [obj]
    return header, frames

def deserialize_bytes(header, frames):
    return frames[0]

register_serialization(bytes, serialize_bytes, deserialize_bytes)
```

API

register_serialization(cls, serialize, ...)	Register a new class for custom serialization
serialize(x)	Convert object to a header and list of bytestrings
deserialize(header, frames)	Convert serialized header and list of bytestrings back to a Python object

distributed.protocol.serialize.**register_serialization**(*cls*, *serialize*, *deserialize*)

Register a new class for custom serialization

Parameters cls: type

serialize: function

deserialize: function

See also:

serialize, *deserialize*

Examples

```
>>> class Human(object):
...     def __init__(self, name):
...         self.name = name
```

```
>>> def serialize(human):
...     header = {}
...     frames = [human.name.encode()]
...     return header, frames
```

```
>>> def deserialize(header, frames):
...     return Human(frames[0].decode())
```

```
>>> register_serialization(Human, serialize, deserialize)
>>> serialize(Human('Alice'))
({}, [b'Alice'])
```

distributed.protocol.serialize.**serialize**(*x*)

Convert object to a header and list of bytestrings

This takes in an arbitrary Python object and returns a msgpack serializable header and a list of bytes or memoryview objects. By default this uses pickle/cloudpickle but can use special functions if they have been pre-registered.

Returns header: dictionary containing any msgpack-serializable metadata

frames: list of bytes or memoryviews, commonly of length one

See also:

deserialize Convert header and frames back to object

to_serialize Mark that data in a message should be serialized

register_serialization Register custom serialization functions

Examples

```
>>> serialize(1)
({}, [b'\x80\x04\x95\x03\x00\x00\x00\x00\x00\x00\x00K\x01.'])
```

```
>>> serialize(b'123')  # some special types get custom treatment
({'type': 'builtins.bytes'}, [b'123'])
```

```
>>> deserialize(*serialize(1))
1
```

distributed.protocol.serialize.**deserialize**(*header*, *frames*)

Convert serialized header and list of bytestrings back to a Python object

Parameters **header: dict**

frames: list of bytes

See also:

serialize

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 analogous 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)
```

add_worker (*scheduler=None*, *worker=None*, ***kwargs*)
Run when a new worker enters the cluster

remove_worker (*scheduler=None*, *worker=None*, ***kwargs*)
Run when a worker leaves the cluster

restart (*scheduler*, ***kwargs*)
Run when the scheduler restarts itself

transition (*key*, *start*, *finish*, **args*, ***kwargs*)
Run whenever a task changes state

Parameters **key: string**

start: string

Start state of the transition. One of released, waiting, processing, memory, error.

finish: string

Final state of the transition.

***args, **kwargs: More options passed when transitioning**

This may include worker ID, compute time, etc.

update_graph (*scheduler*, *dsk=None*, *keys=None*, *restrictions=None*, ***kwargs*)
Run when a new graph / tasks enter the scheduler

Index

H

I

L

M

N

P

Q

R